Scheduling with Environmental Constraints based on Automata Representations

Jerry Chih-Yuan Yang
Center for Integrated Systems
Stanford University, CA 94305
U.S.A.

Giovanni De Micheli
Center for Integrated Systems
Stanford University, CA 94305
U.S.A.

Maurizio Damiani
Dipartimento di Elettronica ed Informatica
Universitá di Padova, Via Gradenigo 6/A
35141 Padova, Italy

Abstract

We introduce a framework in which design information can be represented using an automaton *model. We present a novel scheduling algorithm under environmental constraints where both the design and constraints are represented using automata. This model offers the advantage of supporting different constraints (e.g. timing, resource, synchronization, etc) with a uniform formalism. All feasible schedules are captured with a single product automaton. The automaton is constructed and traversed using efficient BDD-based implicit state-traversal techniques. We present an algorithm that generates a minimum-latency schedule. This apporach is able to exploit degrees of freedom among interacting components of a multi-module system during scheduling.*

1 Introduction

Synthesis systems must provide ways to deal with design specification and the *representation* of constraints (traditionally timing and resource). Existing high-level synthesis tools often rely on ad-hoc methods to cope with the design information and constraints. Moreover, the increased interaction among components in system design, as well as changing of design styles demand design tools to deal with non-traditional constraints.

For example, when considering scheduling independently two (or more) modules that exchange data, there are constraints due to data transfers. These constraints give rise to flexibility that can be used to optimize individual components. Similarly, when considering scheduling processors that run application specific programs, environmental constraints arise from the interaction between hardware and software. The processor may only issue a limited set of instructions to hardware, producing *don't care* conditions which can be used to optimize hardware. To fully take advantage of all such degrees of freedom, system design tools need to handle different forms of constraints in a uniform and efficient manner.

This paper addresses the scheduling problem under constraints, *e.g.* timing, resource, and synchronization. Scheduling problems (*e.g* under resource constraints and/or release-times/deadlines) are intractable[1]. Therefore, exact algorithms are often inefficient for large problems. Scheduling have been formulated and solved exactly as integer-linear programming problems [2], as well as using heuristic algorithms. 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 processes prevent the algorithm for treating general synchronization constraints.

Recently, an exact method using BDDs to solve ILP formulation of scheduling under constraints is presented in [5]. By using BDDs to uniformly capture sequencing dependecies and constraints, the method demonstrates that it is efficient for scheduling under resource and timing constraints.

However, in a system-level design environment, all previous approaches lack the ability to capture constraints among interacting components. Failure to do so leads to non-optimal synthesis results because degrees of freedom between design and its environment are not amply explored. For control-dominated designs, the problem is exacerbated since the design space is more sensitive to the constraints than datapath designs.

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 problem.

We introduce a framework based on automata that targest the representation of design space in high-level synthesis. Automata modeling is able to capture design information at many levels during synthesis[6, 7]. In this paper, we present a method to perform scheduling for control-dominated designs using automata. The design information, as well as the constraints, can be uniformly treated using finite automata as the underlying model. The automata model is suited for describing sets of schedules since a schedule for a process can be described by the sequences of values, or *traces*, which appear on its inputs and outputs. The set of allowable traces represents all feasible schedules for the design, and therefore is a *complete* representation of the design space. The types of constraints that can be modelled by automata can be very general. In this work, we show how timing, resource, and synchronization constraints can be represented using automata.

An important aspect of our approach is that the automata model is a *specification* tool, meaning that nothing structural is implied about the final implementation. Representing the set of possible execution traces, or schedules, through automata allows us to reason about the behavior of the design. Thus, our work is different from the large body of research related to control-unit optimization [8, 9].

We propose an exact solution method to the scheduling problem by leveraging results from implicit state enumer-

ation of FSMs using BDDs. The resulting BDD model implicitly enumerates all feasible schedules that satisfy the specification and its constraints. From this model, optimization with respect to the latency is performed.

Our framework fits into a typical design process as follows. First, a design is modeled using hardware description language (HDL), and compiling the design into a set of interacting automata (Section 2). Constraints are also represented in automata form. An automaton representing all degrees of freedom can then be generated by taking the *product* of the constraint automata and the specification automaton. If solutions satisfying the constraints exist, then the product automaton will be non-empty (Section 3). A minimum-latency schedule is then computed from the product automaton using a shortest path algorithm (Section 4). We present some experimental results in Section 5.

2 Modeling behavior with automata

In this section, we consider behavior for synchronous, sequential systems. A system can be viewed as a set of interacting components, specified by an HDL such as VHDL, Verilog or *HardwareC*. Without loss of generality, we use *HardwareC* [10] as our specification language. We assume hardware models can be interpreted as a set of operations and dependencies [3]. Operations are assumed to be synchronized to a fixed-rate clock and take a single cycle to complete. Multiple-cycle operations are modelled by a chain of single-cycle of operations.

Each component in the system can be modelled by the notion of a 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. A *trace* is a (possibly infinite) sequence of binary values taken over the input/output ports over each clock cycle. A *process* is a set of traces that describe the input-output behavior of the design.

An efficient finite representation is necessary in order to rapidly manipulate traces. To this purpose an automaton model is used. A *non-deterministic finite automaton* is described by a finite set of $\mathcal{S}$ of states, a subset $\mathcal{I} \subseteq \mathcal{S}$ of initial states, and a transition relation $\delta : \mathcal{S} \times \Sigma \rightarrow 2^{\mathcal{S}}$, computing the set of possible next states corresponding to each state and input symbols Σ. In the context of describing processes, the initial state of an automaton is also the state at which the process begins its execution. The final state is also the state in which all possible execution flows for the process terminate.

It is important to note that the behaviors being modeled are *non-deterministic*: *i.e.* for a given input, a set of output behaviors is allowed. The non-determinism represents the flexibility in schedules.

Throughout the rest of the paper, we will illustrate our concepts through the following two designs.

Example 1 (Addition Example)
The following segment of generic HDL specifies a simple execution of loading, adding and storing some values.

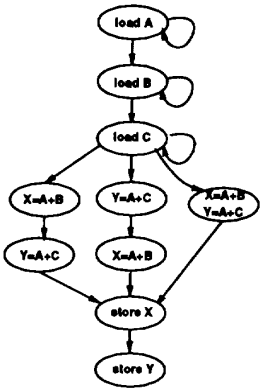

Figure 1: Specification automaton for computation segment example

```
procedure add (A, B, C, X, Y)
  in port A, B, C;
  out port X, Y;
{
   load A;
   load B;
   load C;
   X = A + B;
   Y = A + C;
   store x;
   store y;
}
```

We make some simplifying assumptions. The load and store statements are executed in sequence, *i.e.* there is only one physical port for I/O operations. Initially, we assume that load operations can take infinitely to complete, and store operations take 1 cycle. The add operations require one cycle to complete. The automaton for this segment is shown in Figure 1. Each possible path in the automaton describes an acceptable trace for the execution of the HDL segment. The segment first sequentially loads values for A, B, and C, and then computes $X = A + B$ and $Y = A + C$. The original automaton exhibits all possible scheduling of operations. The left path in the automaton first computes X, followed by Y, using only 1 adder. The middle path first computes Y then X using 1 adder. The right path computes X and Y in parallel, with 2 adders required. Finally, the values for X and Y are stored sequentially. □

Example 2 (Communication Example)

This example shows how automata-based specification can be used to model the degrees of freedom among interacting components. We

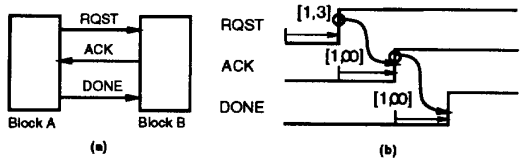

Figure 2: (a) Block diagram and (b) timing diagram for communication example

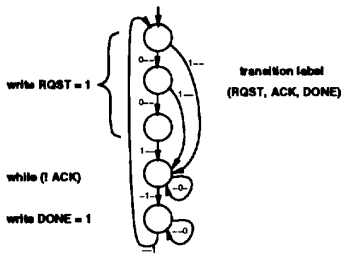

Figure 3: Automaton description for Block A.

use this example to illustrate how synchronization constraints can be used to optimize the block. Block A represents a component in a system to be optimized. It interacts with another component in the system, Block B, which acts as a *synchronization* constraint (Figure 2.a). From Block A's interaction with the environment, the implementation for Block A can be optimized. The timing specification for Block A (Fig. 2.b) is:

Block A

1. Sends $RQST$ signal in 1 to 3 cycles.
2. Waits (up to ∞ time) for ACK signal.
3. Finally, after a possibly ∞ time, block A emits a $DONE$ signal.

This type of communication can take place, for example, in a DMA controller requesting access for a bus. The HDL description for the block can be written as :

```
procedure block_A(RQST, ACK, DONE)
  out port RQST, DONE;
  in port ACK;
{
 write RQST = 1;     /* write rqst in 1-3 cycles */
 while (! ACK) {}    /* wait for ack */
 write DONE = 1;     /* write done in >=1 cycles */
 reset signals;
}
```

The automaton description for Block A can be derived by looking at the set of traces it can accept. For example,

	1	2	3	4	5	6
$RQST$	0	1	–	–	–	–
ACK	–	–	0	1	–	–
$DONE$	–	–	–	–	0	1

is a valid trace for Block A ("-" denote *don't care* condition). However,

	1	2	3	4	5	6	7	8
$RQST$	0	0	0	1	–	–	–	–
ACK	–	–	–	–	0	1	–	–
$DONE$	–	–	–	–	–	–	0	1

is *not* a valid trace since $RQST$ signal took over 3 cycles to assert, violating the timing constraint.

By constructing the set of acceptable traces, we arrive at the automaton representation for Block A, shown in Figure 3. □

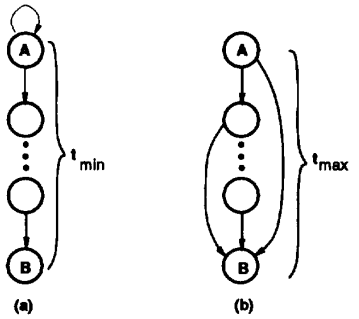

Figure 4: (a) Automaton for minimum timing constraints, (b) automaton for maximum timing constraints.

3 Modeling constraints with automata

In this section we show how constraints can be modeled. Constraints are themselves represented in automata form, and incorporated into the design automata by forming the *product* automaton[11].

The *product* of two automata A_1 and A_2, indicated by $A = A_1 \otimes A_2$. A has a state space defined by $\mathcal{S} = \mathcal{S}_1 \times \mathcal{S}_2$ with initial states $\mathcal{I} = \mathcal{I}_1 \times \mathcal{I}_2$. The transition relation of A is $\delta = \delta_1 \bigcup \delta_2$.

The underlying technique used in constraint incorporation and minimal schedule generation is the traversal of the representation automaton. We leverage results from BDD-based implicit techniques[12] to traverse the automata and form their product.

One modification to the standard traversal procedure is the introduction of a *restriction function*, which is used to further restrict the product to contain only the desired transitions. The restriction function specifies the set of allowable transitions to next states. During each iteration of product traversal, the restriction function is conjoined with the set of next states to insure only the allowable set of states are reached.

In the cases where there are no solutions that satisfy the constraints, the product formation process will result in a *null automaton*. Intuitively, this means that there is no intersection between the design space and the constraint space. Conversely, as long as the product is not null, there exists a feasible solution to the problem.

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

3.1 Timing constraints

Figure (4) shows automata model for timing constraints. We assume that each transition in the automaton represents one clock cycle. To model a minimum timing constraint of t_{min} cycles between two operations A and B, the constraint automaton contains t_{min} serial transitions to "NOOP" states, and a self-loop transition on state A (Figure 4.a). The automaton forces at least t_{min} cycles to expire before B is reached. For a maximum timing constraint of t_{max} between A and B (Figure 4.b), each intermediate "NOOP" state contains a transition to state B. State B therefore must be reached by *at most* t_{max} cycles.

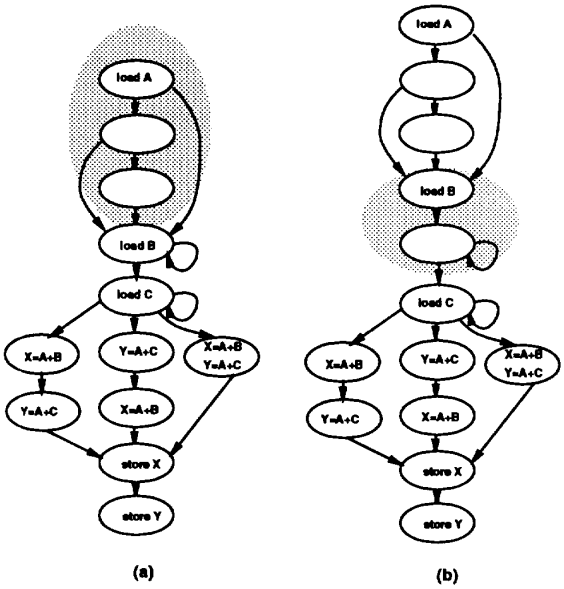

Figure 5: Timing constraint example: (a) 3-cycle maximum timing constraint on **load A**; (b) 2-cycle minimum timing constraint on **load B**.

Example 3 (Timing constraint) Consider the addition computation of Example (1), suppose we wish to impose a *maximum* timing constraint of 3 cycles to the **load** operation on A. The resulting product automaton after applying the constraint is shown in Figure 5.a. The automaton models all possible execution traces representing the behavior that **load A** must complete in 3 cycles.

Suppose we wish to impose a *minimum* timing constraint of 2 cycles on the **load** operation on B. The resulting automaton is shown in Figure 5.b. The automaton models all possible execution traces for **load B** to complete after the minimum of 2 cycles. □

3.2 Resource constraints

Given a constraint on the number of instances for a particular resource, we construct a *comparator* automaton for that resource. The automaton compares the number of resources and the number of instances being utilized. The comparator outputs a 1 if $num\_instances < num\_resources$, otherwise it outputs 0. The comparator output is then intersected with the global output function. Therefore, during implicit traversal of the product automaton, those schedules that violate the resource constraint (therefore causing the global output to be 0) are not constructed, leaving only those schedules that satisfy the constraint.

Example 4 (Resource constraint) Consider the addition segment of Example (1). Suppose we place a resource constraint restricting the number of adders to one. A comparator (Fig. 6.a) automaton testing for schedules utilizing only one adder is constructed as part of the global output function. The resulting product automaton is shown in Figure 6.b. The path which computes the additions in parallel (which requires 2 adders) is eliminated. However, all schedules which require one adder are still present. □

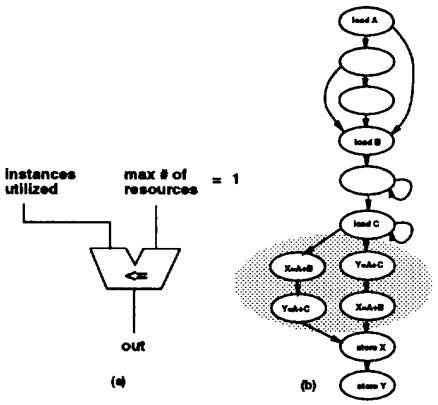

Figure 6: Resource constraint example: (a) Comparator used during implicit traversal, (b) Adder segment constrained by one adder.

3.3 Synchronization constraints

The specification and incorporation of more complicated constraints due to interaction of a hardware module with the environment (or between two modules) is done with a similar procedure. The environmental constraint is specified by an automaton (derived from an HDL), and product between the design and constraint is then formed.

Example 5 (Synchronization constraint)

Referring to the communication example, suppose Block A interacts with a Block B, where Block B is specified as:

Block B (*constraint*)

1. Waits (up to ∞ time) for $RQST$ signal.
2. Outputs ACK signal in 1 to 2 cycles.
3. Waits for the $DONE$ signal.

The generic HDL description is as follows:

```
procedure block_B(RQST, ACK, DONE)
  in port RQST;
  out port ACK, DONE;
{
 while (!RQST) {}   /* wait for rqst */
 ACK = 1;           /* send ack in 1-2 cycles */
 while (! DONE) {}  /* wait for done */
 reset signals;
}
```

The automaton description for Block B is shown in Figure 7.a. After constraining Block A with B, the product automaton is formed using implicit traversal. The result, in Figure 7.b, shows all acceptable execution traces for Block A. The original specifications for A and B specify a communication protocol which is fully blocking, *i.e.* the processes wait until the required signals become available. This type of communication can take place, for example, in a CPU communicating with a DMA controller. □

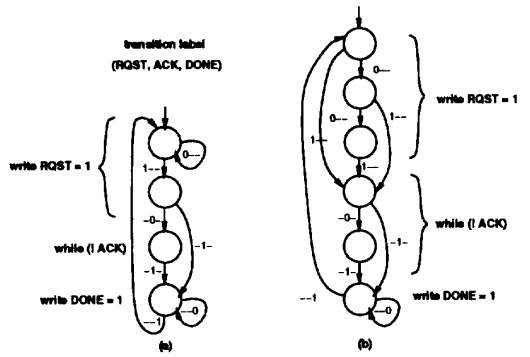

Figure 7: Synchronization constraint example: (a) Automaton for environmental constraint (Block B), (b) Product automaton for Block $A \otimes$ Block B.

4 Scheduling using automata

So far, we have described the formation of automaton under a variety of constraints.

In general, there are many possible optimization cost criteria. Past approaches in using automata for sequential synthesis have focused mainly on minimization of *states* among interacting finite state machines, which loosely corresponds to area minimization (*e.g.* [13]). Other sequential logic level algorithms are surveyed in [9].

Our optimization goal is to target *performance*. Since we are dealing with designs at a high-level, we take the clock cycle as a fixed parameter and minimize the latency, where the *latency* of a process is the number of clock cycles required to execute the process.

In this section, we present an algorithm that computes a schedule which minimizes the latency of the process using the product automaton. Despite the intractable nature of the problem, we note 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.

The product automaton contains all the degrees of freedom from the original specification and the constraints. In devising the algorithm, we distinguish between two types of non-determinism that may exist in the behavior:

1. First, there is the non-determinism that arises from different output schedules given constant inputs. This corresponds to the design space satisfying the constraints. The algorithm must choose *one* minimum-latency schedule from the set of possible output schedules. In this case, there is only one execution path for the process, and is said to be *data-independent* since the latency of the process is independent of data inputs.

2. The second type is the existence of *alternate* execution paths due to runtime changes in the inputs. We define those processes whose execution latency depends on inputs at execution time to be *data-dependent.*

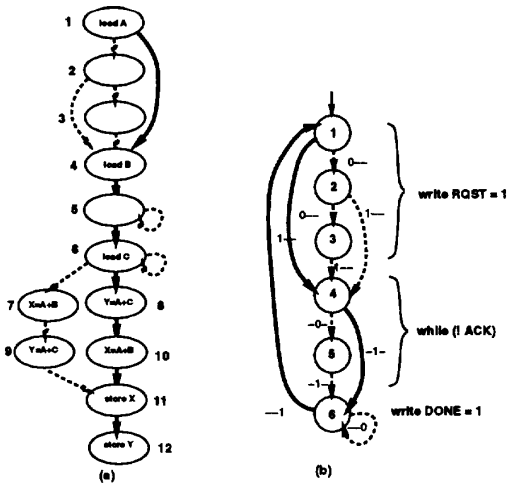

Figure 8: Shortest path for (a) Addition example, (b) Communication example. The shortest paths are marked by dark transitions.

We first address processes exhibiting non-determinism of the first type. We then generalize to the second.

4.1 Solution technique for data-independent models

For data-independent designs, all paths in the product automaton correspond to feasible schedules that satisfy constraints. We state without proof, the following observation regarding the *minimum-latency* schedule of the process.

Proposition 1 *Given an automaton representing the execution flows of a data-independent process, the set of shortest paths from initial to final state is the set of* minimum-latency *schedules.*

Example 6 (Shortest path) In this example, we show the shortest paths for both the addition and communication examples. Starting with the product automata, which represent the design space after incorporating the various constraints, we extract the minimum-latency schedules for each.

Figure 8.a shows the shortest path for the product automaton for addition example. The shortest path corresponds to the minimum-latency schedule since all operations are executed at the earliest time possible. The alternate path, going through states 7 and 9, would require the same latency.

Similarly, Figure 8.b shows the shortest path for the product automaton for communication example. The shortest path corresponds to the minimum-latency schedule to implement Block A.

This is the minimum-latency implementation since the execution time of Block A is 3 cycles. This corresponds to an implementation of the protocol in which communication is *non-blocking*, where the signals can be asserted without any waiting. This means that the communication is taking place as fast as possible. This optimization is only made possible because the environmental

information provided by Block B allows this degree of freedom. As a side note, the similar optimization can be done for Block B as well. □

Shortest path algorithm The product automaton can be seen as a DAG with each edge of weight 1. Given a graph in explicit representation, any single-source shortest path algorithm (such as Dijkstra's algorithm [14]) 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 probably be ineffective for large designs. In this section we describe a shortest-path algorithm that can be applied in the *implicit traversal* framework.

Before proceeding with the algorithm, we define some terminology. Given the product automaton's transition relation $\delta : X \rightarrow Y$, and a subset of its domain $S \subseteq X$, the *image* of S under δ is $\delta(S) = \{\delta(x) : x \in S\}$. Conversely, given a function $\delta : X \rightarrow Y$ and a subset of its co-domain $T \subseteq Y$, the *inverse image* of T under δ is $\delta^{-1}(T) = \{x \in X : \delta(x) \in T\}$. Given a state set S_k, the states reachable from S_k in one transition are the *image* of S_k under the transition function δ. Similarly, the states that can reach a state set S_k in one transition are the *inverse image* of S_k under δ. Given a function f and a variable x, the *existential quantification*, or *smoothing* of f with respect to x is $\exists_x f = f_x + f_{x'}$.

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. During reverse traversal, the set of *previous states* that can be reached in one transition are computed using the inverse-image operation in each iteration. The shortest path is extracted by taking the state(s) with the highest label while performing previous state computations. The algorithm is described in Figure 9. Note that the forward traversal can be seen as an extension of the implicit state traversal procedure[12]. The only additional task required is labeling to state sets during traversal.

In the cases where more than one shortest path exists, the algorithm can do one of two things: if only one path is required, then algorithm picks one state to follow in during reverse traversal. If all shortest paths are desired, then the reverse traversal must be performed for each of the possible states in each iteration.

Example 7 Using Figure 8, we perform the shortest path computation using the above algorithm. For the addition example, following the state labeling in Figure 8.a, the run of the algorithm looks like:

```
forward              reverse
s_0: 1               s_7: 12
s_1: 2   4           s_6: 11
s_2: 3   5           s_5: 10   (picking 1 path)
s_3: 6               s_4: 8
s_4: 7   8           s_3: 6
s_5: 9   1           s_2: 5
s_6: 11              s_1: 4
s_7: 12              s_0: 1
```

The resulting path can be traced by following the reverse traversal starting at s_0 to s_7, corresponds to the darkend path in the figure. For communication example, it can be easily verified that the shortest path is through states $1, 4, 6, 1$. □

```
ForwardTraversal (A) {
    S_0 = InitialState (A);
    i = 1;
    while (FinalState (A) ⊈ S_i) {
        S_i = image(S_{i-1}) − ⋃_{j=0}^{i-1} S_j;
        i = i + 1;
    }
    control_steps = i;
}
ReverseTraversal(A, control_steps) {
    n = control_steps;
    s_n = FinalState(A);
    i = n;
    while (i > 0) {
        s_i = inverse_image(S_i) ⋂ S_{i-1};
        i = i − 1;
    }
}
```

Figure 9: The implicit shortest path algorithm.

4.2 Solution technique for data-dependent models

Given automata models whose execution latencies depend on input changes, we can easily extend the shortest path concept from the previous section. Since the input values are not known at synthesis time, paths for all possible input data values must be explored. In other words, a feasible schedule for data-dependent operations is one that contains an implementation for *all possible* combinations of input data values. The minimum-latency schedule is the feasible schedule that has the lowest number of control steps.

For processes with data-dependent operations, there are two types of variables in the product automaton BDD: *state* variables, which describe sequencing within an execution flow; and *input data* variables, which describe alternate execution flows due to data dependencies. A product automaton for data-dependent designs can be transformed into a data-independent one by removing the nondeterminism of the input data variables. This is achieved by applying the *smoothing* operation on the input data variables. The result is a product automaton BDD that consists of state variables only. We then use the same shortest path algorithm of the previous section to compute the minimum-latency schedule.

5 Results

The communication example shows the utilization of environmental constraints arising from component interaction. The degrees of freedom explored can not be achieved by previous scheduling algorithms. Individually scheduled, Block A and Block B would require blocking mechanisms in its synchronization. Using the automaton model, since

Benchmark	Control steps	BDD size	cpu(sec)
gcd	4	884	5.6
diffeq (1 alu)	5	3290	13.3
tseng (1 alu)	5	7299	49.0
parker86 (1 alu)	10	7704	103.8
ecc.encode	18	15438	155.2
ecc.decode	19	2788	90.4

Table 1: Resource constrained scheduling results

multipliers	ALUs	Control steps	cpu(sec)
3	3	15	357.3
2	3	15	273.2
1	3	16	551.3
3	2	17	605.9
2	2	17	673.1
1	2	17	1010.7
3	1	28	536.2

Table 2: *Elliptic* schedule variances due to changes in resources

Block A can be scheduled with Block B as a *constraint*, the communication is made non-blocking, which results in a faster schedule and probably less hardware in the implementation.

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 control-flow graph intermediate form known as SIF. From SIF, the set of interacting automata is built and the product automaton is formed. The last step is to extract the shortest path which consists of the minimum-latency schedule.

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 feasibililty of the method. It is possible to construct an effective ordering because the automaton structure is known *a priori*. In Table (1), we show the time it takes to 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 5000/240.

As an example, we vary resource constraints on *elliptic*. Table (2) 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.

6 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. The experimental results show that it is feasible to apply our method to practical examples.

7 Acknowledgement

Jerome Fron implemented the SIF to automata compiler. The authors would like to thank David Ku for his comments. This research is sponsered by NSF-ARPA, under grant No. MIP 9115432.

References

[1] M. Garey and D. Johnson, *Computers and Intractability.* W. Freeman and Company, 1979.

[2] C. Gebotys and M. Elmasry, *Optimal VLSI Architectural Synthesis: Area, Performance and Testability.* Kluwer Academic Publishers, 1992.

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

[4] D. Filo, D. C. Ku, C. N. Coelho, and G. D. Micheli, "Interface optimization for concurrent systems under timing constraints," *IEEE Transactions on VLSI Systems*, vol. 1, no. 3, pp. 268–281, Sept. 1993.

[5] I. Radivojević and F. Brewer, "A new symbolic technique for control-dependent scheduling," tech. rep., U.C. Santa Barbara, Deptartment of Electrical and Computer Engineering, Oct. 1993.

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

[7] M. Damiani, "Non-deterministic finite state machines and sequential don't cares," 1994. To appear in proceedings of *European Conference on Design Automation.*

[8] G. Saucier, M. C. Depaulet, and P. Sicard, "Asyl: A rule-based system for controller synthesis," *IEEE Transactions on CAD/ICAS*, pp. 1088–1097, Nov. 1987.

[9] P. Ashar, S. Devadas, and A. R. Newton, *Sequential Logic Synthesis.* Kluwer Academic Publishers, 1992.

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

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

[12] O. Coudert and J. Madre, "A unified framework for the formal verification of sequential circuits," in *ICCAD, Proceedings of the International Conference on Computer-Aided Design*, pp. 126–129, Nov. 1990.

[13] J. Kim and M. M. Newborn, "The simplification of sequential machines with input restrictions," *IEEE Transactions on Computers*, pp. 1440–1443, Dec. 1972.

[14] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms.* San Francisco CA: McGraw Hill, 1990.