A Synthesis Framework Based on Trace and Automata Theory

Jérôme Fron Jerry Chih-Yuan Yang Maurizio Damiani Giovanni De Micheli

Center for Integrated Systems
Stanford University

Abstract

There has been intense research in using degrees of freedom (or *don't care* conditions) to optimize sequential and combinational circuits. The scope of logic synthesis, however, is limited by the lack of information being passed from high-level synthesis tools.

In this paper, we present a general framework to *model* the degrees of freedom at the behavioral level, in a form that can be used by sequential logic synthesis. Behavior is specified by a set of *concurrent, interacting processes*. Each process is described formally by its *set of execution traces* [1], and represented by an ω-automaton [2]. This type of description allows the inclusion of *don't care* conditions in the specification by allowing multiple execution traces for a given input. Moreover, it allows us to cast the synthesis problem into a language containment problem, and to provide a formal description of these *don't cares* .

We have developed a prototype version of a synthesis system based on this framework, targeting the synthesis of the control portion of a circuit. Starting from a hardware description language *HardwareC*, a specification is expressed in terms of a set of interconnected ω-automata . These ω-automata specify the set of acceptable control schedules of a system. The synthesis process entails navigating this set, and implicit state traversal algorithms can be used to this purpose. In this paper, we demonstrate the feasibility of the approach by showing the possibility of traversing the state space of the *specification* automata.

1 Introduction

Logic synthesis systems have proven themselves effective for the optimization of complex functional blocks at the logic level. Theoretical understanding and engineering approaches have been thoroughly explored and provide effective tools for optimization at the combinational as well as at the sequential level. A relevant component of the success of combinational and sequential logic synthesis is due to the availability of precise formal models.

The overall effectiveness of a synthesis system would be improved by the possibility of extracting *don't care* conditions from a high-level description. For example, the knowledge that a particular operation can be scheduled at different time points represents a degree of freedom for the control unit, and can be used for its optimization.

The problem of extracting *don't care* information from high-level specifications is receiving increasing interest. Several observations to this regard have been made by Bergamaschi in [3]. He presents a description of *don't cares* associated with the various structural elements of an RTL description of a circuit. Such *don't cares* were derived, however, mostly by a structural analysis of the circuit rather than a formal approach.

Wolf introduces in [4] the concept of *behavioral finite state machine*. Unlike ordinary FSMs, BFSMs allow the compact modeling of slacks in the scheduling of operations. Other degrees of freedom, however, such as the reordering of those operations, cannot be represented.

In this paper, we consider specifications in terms of *concurrent, interacting, synchronous processes*. This specification style is used in several Hardware Description Languages, such as VHDL [5] and *HardwareC* [6].

Following Hoare [1] and Dill [8], we formalize the notion of process by resorting to *trace theory*. Each process is described by a set of input and output variables (for short, the process *terminals*), and by

a set of *execution traces*. Informally, a trace of a process is a sequence of symbols, recording the values taken by its terminals over time [1].

The appeal of trace theory relies in the fact that the enumeration of all the acceptable execution traces for a system implies capturing *all* the degrees of freedom on its functionality. A circuit *satisfies* the specifications if its execution traces are acceptable to the specifications, *i.e.* if they are contained in the trace set of the specifications. The *synthesis* problem for a circuit can then be cast into that of finding a minimum-cost realization that satisfies this containment property.

In order to make a synthesis system practical, it is necessary to provide compact descriptions of trace sets. In [2] and [7], the use of finite ω-automata was proposed for describing trace sets. These automata describe the desired functionality as well as the degrees of freedom. They can be used to perform synthesis by exploring directly the design space, or can be used as external *don't cares* for the local optimization of an already existing design. In this respect, we provide in Section (4) a general representation result. We show that the degrees of freedom, or *don't care*, associated to an implementation FSM embedded synchronous circuit can be expressed in terms of an ω-automaton . This ω-automaton is the product [7] of the ω-automata of the specifications and of those representing the rest of the circuit.

The use of *don't care* conditions to simplify interacting FSMs was first explored by Kim and Newborn[10]. This is later expanded in works by Devadas [11] and Rho *et al*[12]. In this work, we take the same general idea as Kim and Newborn, but apply a much more general environment to perform optimization. In particular, we provide the degrees of freedom from a higher level of specification than the logic level. Therefore, we show that the types of results achieved are not obtainable using conventional interacting FSM techniques.

Currently, we are targeting control synthesis. Therefore the ω-automata specify the control schedules and communication protocol constraints among the various processes. We implemented a preliminary version of a synthesis tool based on the use of ω-automata . The language *HardwareC* is used as a front-end for entering a specification in terms of a set of interacting processes, each described by an ω-automaton .

We show empirically that automata of reasonably complex circuits can be efficiently represented and manipulated, and that it is possible to traverse efficiently their state space. The specification can then be used as a method to *synthesize* a design from high-level specifications or to *optimize* an existing design by extracting relevant *don't care* conditions from the automata. For example, in the present case, it can be applied to the synthesis/optimization of schedules and control units.

The rest of the paper is organized as follows. The next section introduces the terminology associated with traces and processes. Section (3) presents relationship between processes and *don't cares* . In particular, we introduce a theorem describing the degrees of freedom of a sub-circuit with respect to its environment. Section (4) talks about a set of transformations that generate the automata specification of a circuit from a high-level sequencing graph model. We give experimental results in Section (5). We conclude the paper and describe future work in Section (6).

2 Hardware specifications by interacting processes.

Specifications of hardware in terms of interacting processes are fairly common. Processes are typically described in a programming language style:

Example 1.

The following code specifies the behavior of two units sharing a bus. Each unit uses the bus to fetch an instruction, and then to write a result after an execution step.

```
P1: repeat {                          P2: repeat {
        send(bus_request);                    send(bus_request);
        while (!bus_rdy) wait;                while (!bus_rdy) wait;
        fetch_bus;                            fetch_bus;
```

[1] Hoare was actually interested in asynchronous systems, and therefore traces represented sequences of *events*.

```
        execute;                                    execute;
        send(bus_request);                          send(bus_request);
        while(!bus_rdy) wait;                       while(!bus_rdy) wait;
        write_bus;                                  write_bus;
}                                           }
```

□

A more formal view of a process can be obtained by examining directly the signals through which it communicates with the environment. In the case of the process $P1$ of Example (1), these signals are for example the bus request signal s_1 and the bus ready signal r_1. Over time, the pattern these signals follow can be used to describe the process itself. For example, assuming that each line of code takes exactly one clock period, a pattern

s_1	1	0	0	0	1	0
r_1	0	1	0	0	0	0

is a possible trajectory for s_1 and r_1, while

s_1	1	0	1	0	1	0	$\cdots$
r_1	0	1	0	0	0	0	$\cdots$

is not, because two bus requests must be separated by at least three clock cycles, and by one bus ready signal.

2.1 Terminology.

The notion of process is formalized here in the context of *trace theory*. A process is described by a set of input and output variables (for short, the process *terminals*), and by a set of *execution traces*. For example, the variables of process $P1$ of Example (1) are s_1 and r_1.

Informally, a *trace* of a synchronous process is an infinite sequence of values taken on its input and output ports over each clock cycle. Traces of a synchronous process allow multiple transitions to take place between two consecutive clock edges. This would not be allowed in interleaving semantics, which is typically used for modelling asynchronous processes [8].

Traces can be of finite or infinite length. Finite-length traces, originally considered by Hoare, represent only partial executions of a process. A trace of length n can represent an execution only up to the time-point n. Certain important properties of a process, such as liveness or fairness properties, require a description in terms of infinite-length traces [8].

Definition 1.

Let $B = \{0, 1\}$. The set of all possible 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 $A = (\mathcal{I} \cup \mathcal{O})$.

A **synchronous trace** *T, or* **trace** *for short, is an element of the set*

$$(B^\omega)^{|A|} \tag{1}$$

□

A process is represented as a set of possible execution traces:

Definition 2.

A **process** *P is a set of traces.* □

Finite representations of processes are however necessary for their rapid manipulation. ω-Automata have been proposed for this purpose in a verification context [9].

An ω-automaton is described by a finite set S of states, a subset $S_0 \subseteq S$ of initial states, and a transition relation $\delta : S \times \Sigma \rightarrow 2^S$, computing the set of possible next states corresponding to each state and input symbol. A *run* of an automaton over a sequence $\sigma_0, \ldots, \sigma_n, \ldots$ of input symbols is a sequence of states $s_0, \ldots, s_n, \ldots$ such that $s_0 \in S_0$ and for every $n \geq 0$ $s_{n+1} \in \delta(s_n, \sigma_n)$. The description of an automaton is completed by an *acceptance rule*. The acceptance rule decides if a sequence of symbols belongs to a trace set, based on which states are visited during a run of the sequence. Several flavors of acceptance rules exist in the literature [2]. Their distinction is immaterial for our purposes, as long as the intersection of the two processes can be computed essentially by the traditional product rule for automata.

The *product* of two ω-automata A_1 and A_2 is indicated by $A = A_1 \otimes A_2$. Informally, a product machine A has the state space that is the Cartesian product of A_1 and A_2, and the transition function is the logical conjunction of A_1 and A_2. A formal definition for a product can be found in [7]

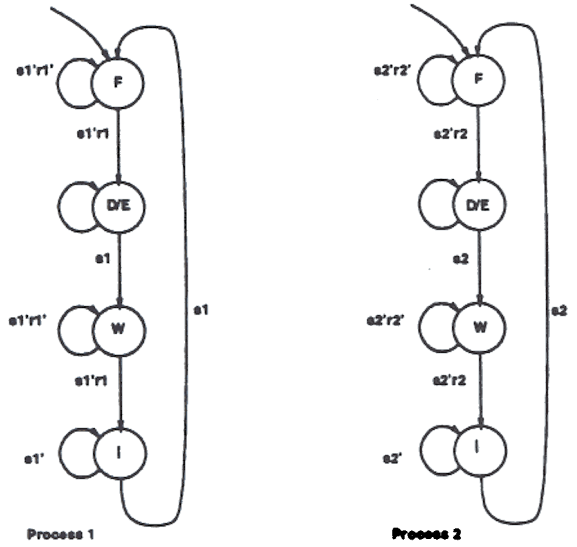

Figure 1: Automata for the processes $P1$ and $P2$ of Example (1). F indicate initial states.

Example 2.

The automata of Fig. (1) describe the behavior of the processes $P1$ and $P2$ of Example. (1). At each clock tick, each process can decide whether to execute the line or to idle. The idling transitions are represented by the unlabeled self-loops in the state diagram. Labels indicate the execution of the corresponding instruction. □

2.2 Environment constraints.

When synthesizing a system, it is necessary to take into account its interactions with the environment. A system needs to communicate with other modules through handshaking protocols, or may need to use memory and hardware resources that are shared with the environment.

Such interactions impose constraints on the systems (*i.e.* protocols must be satisfied), as well as information that can be used during synthesis. For example, knowing that the design interfaces by means of a specific protocol implies that only certain input sequences will be received through the input pins.

Example 3.

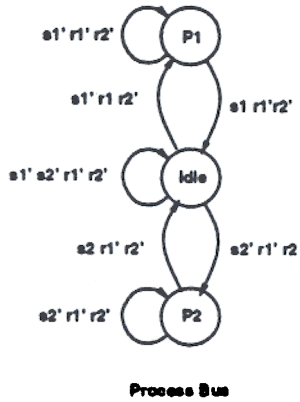

Figure 2: Automaton for the bus of Example (3).

The two processes $P1$ and $P2$ interface to a bus. To prevent simultaneous reads and writes on the bus, the bus protocol forces signals r_1 and r_2 to be mutually exclusive. Again, only some sequences of values of r_1 and r_2 can occur. The possible execution traces for the bus are represented by the ω-automaton of Fig. (2). □

Example (3) outlined that environment constraints can be modeled by processes (trace sets) as well. The specifications S can therefore be cast in general in the form of a product of several ω-automata S_i, each of which represents a process or a constraint:

$$S = \otimes_{i=1}^{N} S_i \tag{2}$$

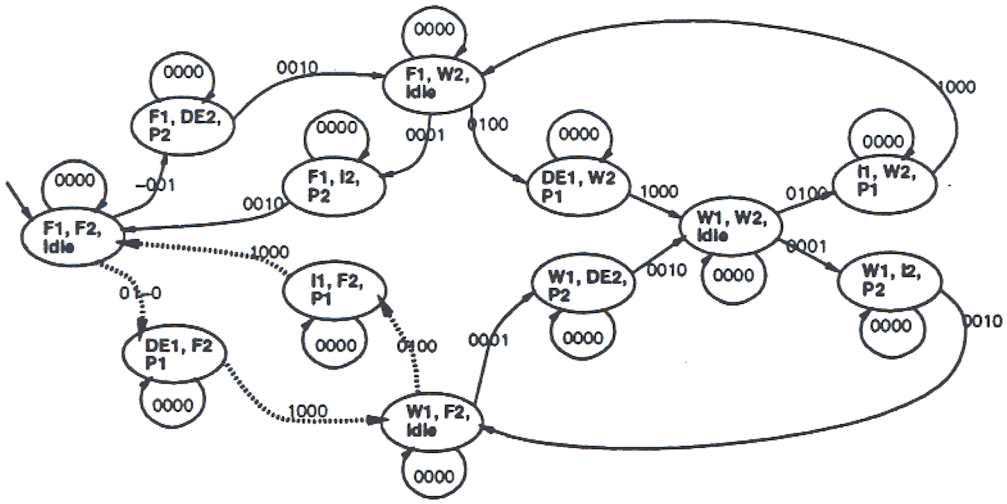

Figure 3: Product automaton for Example (4). Each transition shows the logic values of $s_1 r_1 s_2 r_2$. Each state shows which corresponding state in (P_1, P_2, bus) are being traversed. Dashed transition illustrate a starvation trace for P_2.

Example 4.

For P_1, P_2, and the bus, the composite automaton S that represents the set of acceptable traces is shown in Figure (3). Note that the bus protocol successfully enforces the mutual

exclusion requirement. However, we can detect a bug in the protocol by noticing that there are traces where both processes would starve (Dashed transitions show a trace where P_2 is starved). □

3 Processes and *don't cares*.

The end result of synthesis is a circuit implementation whose terminal behavior satisfies the specifications. In a synchronous environment, such circuits are described by finite-state machines. In general, more than one state machine satisfies the specifications.

3.1 Sequential optimization using *don't cares*.

In practice, the optimization of large synchronous circuits is carried out by isolating small sub-circuits and optimizing them separately. In other words, a synchronous circuit implementation is often regarded as an interconnection of FSM circuits, denoted by $\{M_j, j = 1,\ldots,N\}$, each of which is optimized independently. The existence of degrees of freedom in the specifications and the embedding of the sub-circuit in a larger circuit gives rise to *don't care* conditions that can be used in its optimization, much the same way as in the combinational case.

In order to derive a representation of the *don't cares* associated to an embedded sub-circuit M_i, it is convenient to represent also the functionality of all sub-circuits implementation by processes. Each process can be described by an ω-automaton A_j, where $j = 1,2,\ldots,N$. The state diagram of each A_j can be readily obtained from that of M_j by removing the distinction between inputs and outputs.

The behavior of the circuit implementation is thus described by the process

$$C = \otimes_{j=1}^{N} A_j \tag{3}$$

The following result links the *don't cares* associated with a sub-circuit implementation M_i to external specifications and to the processes A_j of the rest of the network:

Theorem 3.1 *Let S be the process describing all possible trace sets of the behavior. Let A be the set of automata describing the circuit M implementation of the design. The* **degrees of freedom** *of sub-circuit M_i is its set of possible execution traces, denoted by the ω-automaton D_i:*

$$D_i = S \otimes (\otimes_{j=1, j\neq i}^{N} A_j) \tag{4}$$

The proof, although not difficult, is lengthy and therefore omitted here.

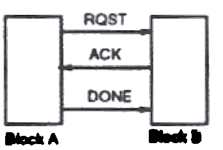

Figure 4: Block diagram for extended example

The theorem says that the *don't care* of a sub-circuit can be found by knowing the specifications for the entire circuit and the interaction of the sub-circuit with its environment. Note that while Theorem (3.1) is similar to the Kim and Newborn procedure in appearance, there is additional power offered by our framework. In particular, the automata specification gives the set of all acceptable traces for the behavior, so it allows the exploration of design solutions that are not usually reachable by normal FSM analysis. The framework can be used in conjunction with other techniques for interacting FSM optimization, since it presents the relevant optimization information at a higher level.

We illustrate a possible optimization using automata framework with the following extended example.

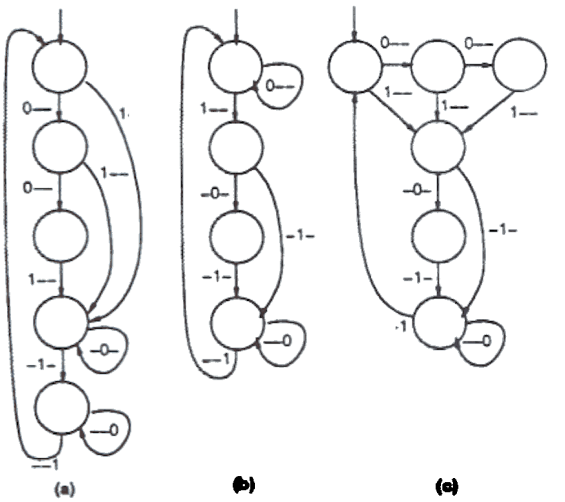

Figure 5: (a) and (b) show the *specification* automata S_A and S_B for blocks A and B. Each transition shows the logic values of $RQST$, ACK, $DONE$. (c) shows the composite specification automaton S.

3.2 Extended *don't care* example

For the purposes of illustrating Theorem (3.1), we use a fairly simple communication example. The block diagram of the example is in Figure(4). The *specification* of the two process involved is briefly stated:

- Block A - Sends a $RQST$ signal between zero to three cycles. It then waits (up to infinite time) for an ACK signal. Finally, after a possibly infinite time, block A outputs a $DONE$ signal. Shown by Figure(5.a)
- Block B - Awaits (up to infinite time) for the $RQST$ signal. It then outputs ACK signal within one or two cycles. It then waits infinitely for the $DONE$ signal. Shown by Figure(5.b)

The complete set of allowed interaction between the two processes can be found by taking the product automaton. This is shown in Figure (5.c).

Now, assume via some independent synthesis process, an FSM *implementation* is generated for each of the blocks. The FSMs are shown in Figure (6). The FSMs, when connected, can indeed communicate. It can be easily verified that I_A, I_B, and $I_A \otimes I_B$ are all contained by S, and therefore they meet the specification.

The goal is to optimize both I_A and I_B by using the *number of states* as a cost function. First, we optimize I_A. This is done by first extracting its *don't care* conditions. Following Theorem 3.1, we first form the product $\otimes_{j=1,j\neq i}^{N} A_j$. Since there are only two FSMs under consideration, this product is just I_B.

We then find the degrees of freedom for block A by $D_A = S \otimes I_B$. D_A is shown in Figure (7.a). Since D_A contains all degrees of freedom for I_A, it naturally contains the trace formed by $I_A \otimes I_B$. Since the goal is to find a more optimal solution than the current implementation, we do not consider the trace $I_A \otimes I_B$. This trace is represented by dash lines. The solid transitions left are the possible degrees of freedom that have not been explored. It is clear that the bolded transition will yield the fewest states in the transition graph.

We then perform the same procedure on I_B. For uniformity sake, we use the un-optimized I_A. D_B, found by $I_A \otimes S$, is shown in Figure (7.b). We again use dashed transitions to indicate the current implementation's trace. The only unexplored transition turns out to be a more optimal choice. The optimized versions of I_A and I_B are shown in Figure (8). By using conventional FSM reduction techniques, it is not clear that such degrees of freedom can be explored. In particular, note that for I_A, the second and third states are eliminated, giving a final implementation of three states. For I_B, the second

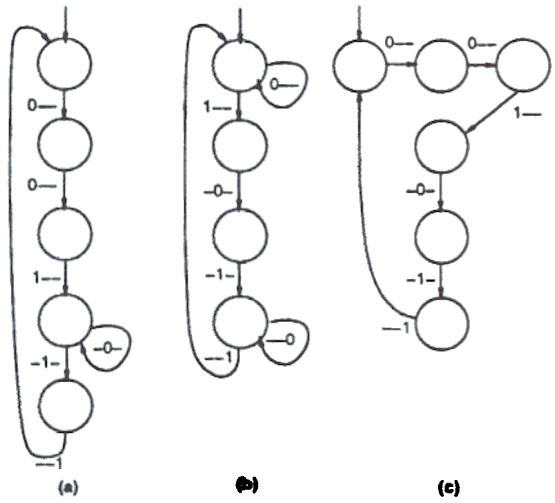

Figure 6: (a) and (b) show FSM *implementations* I_A , I_B for blocks A and B. Distinction between input and output is removed. (c) shows the product $I_A \otimes I_B$.

state is eliminated, giving a final of three states. Also, note all the idle transitions are eliminated as well. The resulting optimum FSMs for A and B are identical, and can communicate without any blocking.

In general, several different implementations can satisfy the specification. Corresponding to the same input sequence, the original I_A and the optimized I_A would respond differently, i.e. they would exhibit a different terminal behavior on the outputs.

Without the specification information, the optimization would not have been possible. The states could be eliminated from the implementation FSMs only because the states are deemed unnecessary by the *specification*. For this reason, no current interacting FSM algorithms can reach the same result.

Algorithms for finding the minimum cost once given the degrees of freedom are still under development. Theorem (3.1) indicates that local optimization based on *don't cares* from automata can be performed provided that the state space of the product automaton of the specifications and the local machines $M_j, j = \{1, \ldots, N\}$ can be efficiently traversed.

In this paper, we first present the empirical results that show the feasibility of a state space traversal of the specification automaton. This is the subject of Section (5).

4 Implementation.

An implementation of the framework presented above is under development using *HardwareC* as the input HDL [6]. *HardwareC* describes hardware in terms of concurrent sequential processes, with the possibility of specifying timing and data-dependent synchronization constraints among the processes.

These processes are translated into an ω-automaton representation. This latter representation constitutes the eventual *formal specification* of the hardware under synthesis. Each process P is in particular associated a pair of input and output signals, named $start_P$ and $done_P$, respectively. Each process is idle until receiving a pulse on its input $start_P$. The termination of execution is signaled by a pulse on $done_P$, after which the process returns to its idle state.

HardwareC allows the specification of *timing constraints* between operations. An operation is called here an *elementary process*. A *timing constraint* between two elementary processes $P1$ and $P2$ is any constraint placed on the traces of $done_{P1}$ and $start_{P2}$.

The timing constraints considered in *HardwareC* are essentially of interval type: a process $P2$ must be started *no sooner* than $l_{1,2}$ clock periods and *no later* than $u_{1,2}$ clock periods after $P1$ is done. These timing constraints are summarized in a *sequencing graph*. The *HardwareC* sequencing graph also supports control structures (i.e. conditional and loop structures). The detailed description of the sequencing graph

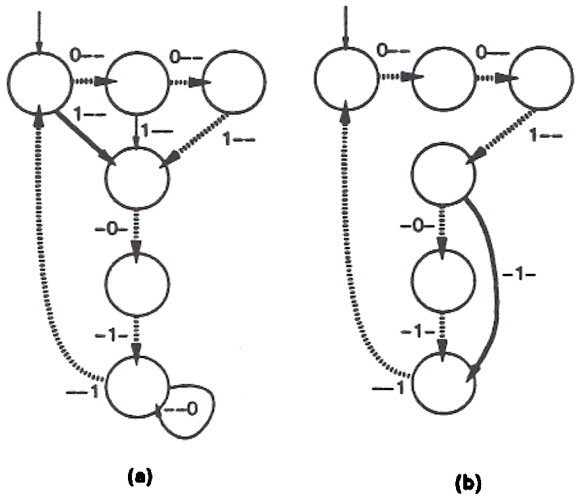

Figure 7: (a) and (b) show D_A and D_B. Dashed transition represent current implementation's trace. All solid transitions are unexplored degrees of freedom. Bold transition represents the optimal transition.

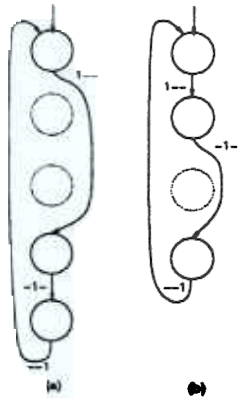

Figure 8: (a) and (b) show optimized versions of I_A and I_B, with the eliminated states in dashed circles.

can be found in [6]. In the rest of this section we illustrate the transformation of these constraints into an automata format.

4.1 Construction of the ω-automata from the sequencing graph.

The rest of this section illustrates the construction of ω-automata corresponding to the following building blocks of the sequencing graph:

- An interval-type timing constraint;
- The conditional execution of a process;
- The modeling of an OR-scheduled process. An OR-scheduled process is the termination condition for conditionals where only *one* of the conditional branches needs to complete.

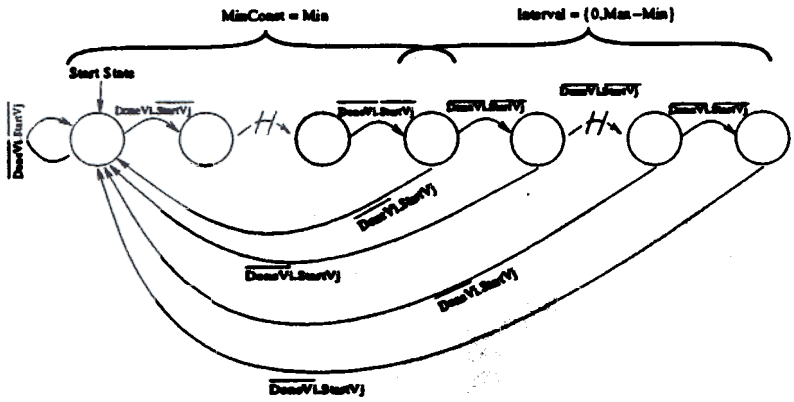

Figure 9: General automaton body for modeling minimum and maximum time constraints between two elementary processes v_i and v_j

Representation of timing constraints.

Interval-type timing constraints are represented by automata like the one shown in Fig. (9).

The automaton recognizes sequences of values on the signals $Done_{P_i}, Start_{P_j}$ formed by a pulse on $Done_{P_i}$ followed by a pulse on $Start_{P_j}$. The duration of each pulse *must* be exactly 1 clock cycle. Moreover, $Start_{P_j}$ must trail $Done_{P_i}$ by at least Min clock cycles and at most Max clock cycles.

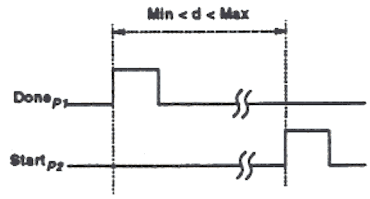

Figure 10: Acceptable waveforms for the signals $Done_{P1}, Start_{P2}$.

The possible waveforms are illustrated in Fig. (10).

Conditional execution of processes.

Values of data signals can impose conditions on the execution of processes: a process P_i (associated with V_i) can fire only if it satisfies its timing constraints *and* a condition C holds true. This is represented by a labeled edge on the sequencing graph, as shown in Fig. (11-a). The corresponding automaton is shown in Fig. (11-b). Notice in particular that the condition for firing $P2$ is sampled at the time point where $Done_{P1} = 1$.

The "OR" activation constraint.

Consider the situation of Fig. (12). Process $P1$ (denoted by can fire either because of $P2$ *OR* because of $P3$.

Automata representing this type of constraint do not have a regular and simple structure, and therefore they are currently not modeled by a separate automaton. Rather, the following construction is employed. We refer again to the situation of Fig. (12). The activation condition for P_1 is the union of the $Done$ signals of its predecessors. In this case,

$$Start_{P_1} = Done_{P_2} + Done_{P_3}$$

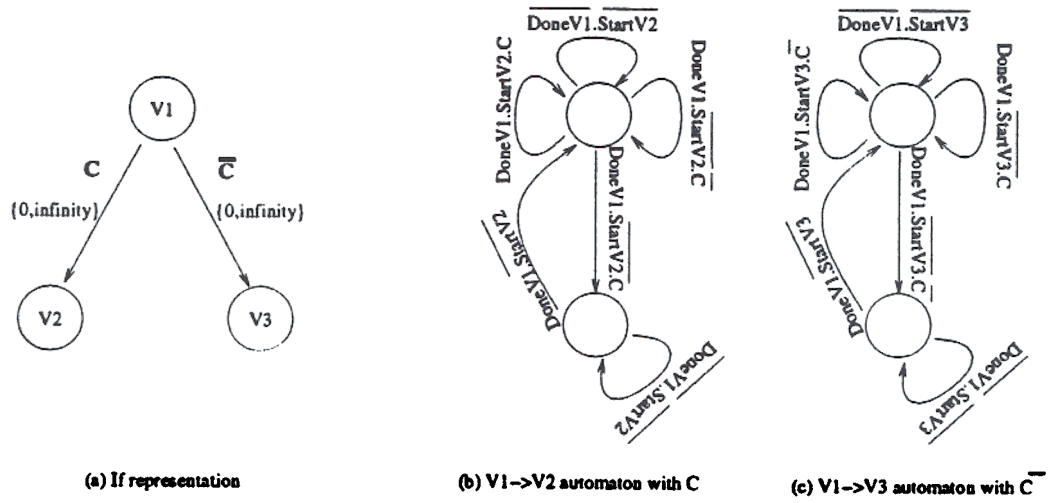

Figure 11: Control structure example : *if* statement

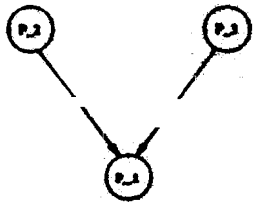

Figure 12: The "OR" Synchronization constraint.

4.2 Global representation of the system.

Having translated each process and each execution constraint into an automaton, the specifications reduce to a collection of ω-automata .

Each automaton is mapped into a synchronous circuit. The circuit has an output taking value 1 corresponding to valid transitions of the automaton, and zero otherwise.

Example 5.

> The automaton of Fig. (13-a) represent a timing constraint (a minimum delay of 1 clock cycle) on $Start_{P2}$ with respect to $Done_{P1}$. Its valid transitions are described by the function $Out = Done'_{P1}(Q0+Q1)' + Start'_{P2}(Q0+Q1)$. The automaton is represented by the circuit of Fig. (13-b). □

Given a collection of automata, the only valid transitions are those for which all output functions take value 1. The function describing the valid transitions of the system are therefore the logic AND of the functions Out of the individual automata.

5 Experimental Results.

The traversal of the state space of the product automaton can be performed by a state traversal of the corresponding circuit *constrained* to valid transitions only. In this section we present preliminary experimental results on state-space traversal. To be able to traverse the state-space is important since it is the implicit representation of the degrees of freedom of the behavior.

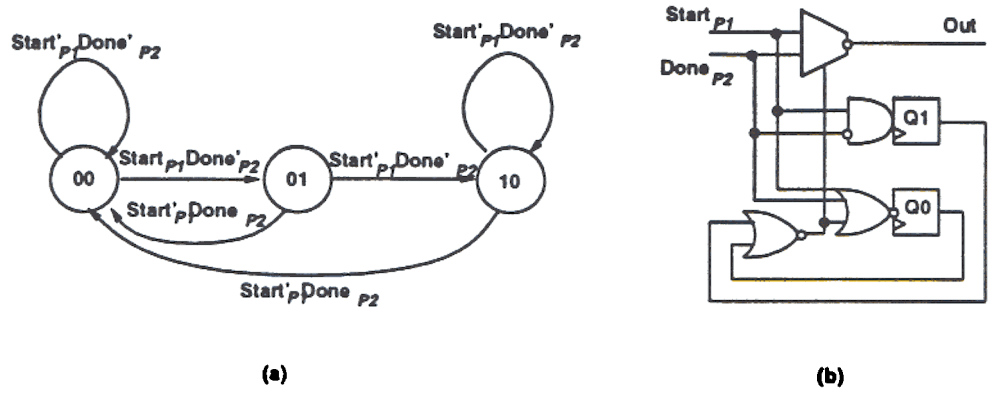

Figure 13: Example on timing constraint and corresponding automata

We conducted two different sets of experiments. The first set consisted of building the automata for large basic scheduling structures and verifying the traversability of their state space. The structures considered were:

- Serial scheduling of a number of processes;
- Parallel scheduling of a number of process;
- Conditional execution of a process;

The reason behind these experiments is to understand the bounds on the CPU time and memory space requirements for these basic scheduling constructs. This information can then be used in estimating *a priori* the requirements for more complex structures.

The second set of experiments consisted of verifying the traversability of the state space of automata extracted from a set of existing high-level synthesis benchmark examples.

We adopt the technique for implicit state enumeration of FSMs using BDDs [13][14][15] as our representation model for automata. Because the reachable state space is represented using a BDD, the key figure of merit is the size of the BDD representing the set of reachable states. The number of states reachable is also important, since traversal through those states to find a minimum cost solution will be an issue in optimization.

The section is concluded by an analysis of implementation issues and on possible improvements.

5.1 First set of experiments.

Nb processes	States visited	BDD size	CPU[2]
4	14	30	0.9
6	18	38	1.5
8	22	46	1.9
10	26	54	2.2
50	104	214	47.8
100	206	414	343.4

Table 1: Automata characteristics for processes in *series*

We considered first a sequencing graph consisting of a linear chain of vertices and edges (as in Fig. (1)). This situation corresponds to, for example, a chain of arithmetic operations. Each edge in the graph is modeled by the edge-automaton described earlier.

The experimental results are illustrated in Table (1). **Nb processes** represents the number of processes in the chain. **Nb states** and **BDD size** refer to the number of states traversed and the size of the resulting BDD, respectively. Table (1) indicates that the number of states grows linearly with the length of the chain.

The second structure we considered consisted of a set of parallel processes (for example, a set of parallel read/write operations), eventually synchronized by a " join" construct.

Nb processes	States visited	BDD size	CPU
1	10	18	0.5
2	20	36	0.5
4	260	72	1.4
5	1028	90	1.9
10	1048580	180	4.6
20	1.1×10^{12}	360	16.5
40	1.2×10^{26}	720	97

Table 2: Experimental results for processes in *parallel*

Nb processes	States visited	BDD size	CPU
2	12	35	0.9
4	18	58	6.6
8	30	106	13.1
16	54	202	72.3
32	102	394	194

Table 3: Automata characteristics for conditional processes.

Unlike the previous case, the number of states visited grows at exponential rate. It is worth noting, however, that the BDD representing the set of reachable states remains remarkably small, and the CPU time is likewise well contained.

The final structure we considered was the control structure of nested "IF" statements. Table (3) shows that again CPU time and BDD size are very contained even for large-sized problems.

5.2 Selected benchmark results

We now present preliminary experimental results collected on some high-level synthesis benchmarks.

The benchmarks were initially written in *HardwareC*. Their sequencing graphs were derived and the automata modeling the timing and synchronization constraints were constructed. In Table (4), column **Nb. edges** indicates the number of edges in the sequencing graph, corresponding to the number of automata that were constructed.

All benchmark processes perform reasonably complex tasks. Most benchmarks implement control or communication protocols. For example, *encode* and *decode* perform the handshaking and computation for an error correcting system[6]. *DMA_rcvd*, *xmit_bit*, *rcvd_bit* and others are all communication processes for an ethernet controller [16]. Table (4) indicates that for all such benchmarks, the state space can be traversed quickly and represented compactly, as the sizes of the BDDs are all less than 1000 nodes.

example	states	BDD size	cpu
gcd	97	146	12.4
decode	2351	540	492.5
encode	16273	1071	242.7
CPUQUEUE	124	205	10.1
DMA_rcvd	356985	265	13.8
xmit_bit	509	238	26.3
rcvd_bit	22722	232	295.3
parker86	6762	413	384.6
daio_phase_decoder	79808	501	198.5
daio_receiver	104	210	61.8
diff_eq	5866	1151	265.1

Table 4: Automata characteristics for selected high-level synthesis benchmarks

5.3 Improvements.

The major bottleneck in traversing the automata is the construction of the BDD corresponding to the set of valid transitions. Currently, we are investigating three ways to overcome this difficulty: The first way is to avoid the explicit construction of the global *Out* function (which defines the set of all valid transitions within all the automata) by looking at implicit methods. The second way is to reduce the number of edges in the sequencing graph. This would result in a reduction in the number of automata, and can be accomplished, for example, by removing redundant edges in the sequencing graph. Algorithms for removing redundant edges in a sequencing graph were presented by Ku *et al.* [6]. The third way consists of investigating better variable ordering heuristics for the BDD construction, possibly tailored to the type of problem at hand.

6 Future Work and Acknowledgements

In this paper, we have presented a framework for modeling degrees of freedom at the behavioral level based on automata. In particular, we presented a method to model hardware as a set of interacting sequential processes, where a process is defined by a set of acceptable execution traces. Hardware specifications (in particular including degrees of freedom) are thus ultimately modeled by a set of automata.

We have presented a preliminary implementation of this approach, in which we translate a hardware description language (*HardwareC*) into an automata-based specification. We show that the state space of the resulting automata can be traversed quickly and manipulated using implicit state enumeration methods based on BDDs.

We have showed that this formulation can model reasonably complex systems, consisting of several processes and synchronization constraints.

In the next phase, we plan to develop a set of algorithms to perform synthesis and optimization by extracting the *don't care* information that exist in the automata.

The authors would like to thank Jerry Burch for his clarification on the definition for traces, and David Filo for his suggestions on the extended example. This research is sponsored in part by NSF/ARPA under grant MIP 91-15-432.

References

[1] C. A. R. Hoare, *Communicating Sequential Proceses*. Prentice Hall International, 1985.

[2] Y. Choueka, "Theories of automata on omega-tapes, a simplified approach," *Journal of Computer Systems Science*, vol. 8, pp. 117–141, 1974.

[3] R. A. Bergamaschi and D. L. A. Kuehlmann, "Control optimization in high-level synthesis using behavioral don't cares," in *Proceedings of the Design Automation Conference*, (Anaheim, CA), pp. 657–661, June 1992.

[4] W. Wolf, A. Takach, C.-Y. Huang, R. Manno, and E. Wu, "The princeton university behavioral synthesis system," in *Proceedings of the Design Automation Conference*, (Anaheim, CA), pp. 182–187, June 1992.

[5] R. Lipsett, C. Schaefer, and C. Ussery, *VHDL: Hardware Description and Design*. Kluwer Academic Publishers, 1989.

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

[7] R. P. Kurshan and K. L. McMillan, "Analysis of digital circuits through symbolic reduction," *IEEE Transactions on CAD/ICAS*, vol. Vol. 10, no. No. 11, Nov. 1991.

[8] D. L. Dill, *Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits*. The MIT Press, 1988.

[9] E. Macii, B. Plessier, and F. Somenzi, "Verification of systems containing counters," in *ICCAD, Proceedings of the International Conference on Computer-Aided Design*, pp. 179–182, Nov. 1992.

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

[11] S. Devadas, "Optimizing interacting finite state machines using sequential don't cares," *IEEE Transactions on CAD/ICAS*, vol. 10, no. 12, pp. 1473–1484, Dec. 1991.

[12] J.-K. Rho, G. Hachtel, and F. Somenzi, "Don't care sequences and the optimization of interacting finite state machines," in *ICCAD, Proceedings of the International Conference on Computer-Aided Design*, pp. 418–421, 1991.

[13] 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.

[14] H. Touati, H. Savoj, B. Lin, R. Brayton, and A. Sangiovanni-Vincentelli, "Implicit state enumeration of finite state machines using BDD's," in *ICCAD, Proceedings of the International Conference on Computer-Aided Design*, pp. 130–133, Nov. 1990.

[15] H. Cho, G. Hachtel, S.-W. Jeong, B. Plessier, E. Shwarz, and F. Somenzi, "Atpg aspects of fsm verification," in *ICCAD, Proceedings of the International Conference on Computer-Aided Design*, pp. 134–137, Nov. 1990.

[16] R. Gupta and C. Coelho, "Ethernet controller design," in *private communications*, 1991.