A Synthesis Framework Based on Trace and Automata Theory

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

Maurizio Damiani
Dip. Elettronica ed Informatica
Universitá di Padova,
Via Gradenigo 6/A, 35131 Padova, Italy

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

Abstract

In this paper we present a method for modeling *don't cares* at high-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* [5] and represented by an ω-automaton [2]. This type of specification is formally precise and allows the inclusion of *don't cares* 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 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 . 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 [1]. 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 [10] 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 [8] and *HardwareC* [6].

Following Hoare [5], we formalize the notion of process by resorting to *trace theory.* Each process is described by a set of input and output variables (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 lies 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.

Currently, we are targeting control synthesis. Therefore 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* condi-

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

tions from the automata.

The rest of the paper is organized as follows. The next Section introduces the terminology associated with traces and processes. Section (3) presents the set of transformations that generate the automata specification of a circuit from a high-level sequencing graph model. We give experimental results in Section (4). Conclusions are drawn in Section (5).

2 Hardware specifications by interacting processes.

Hardware specifications in terms of interacting processes are 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,P2: repeat {
       send(bus_request);
       while (!bus_rdy) wait;
       fetch_bus;
       execute;
       send(bus_request);
       while(!bus_rdy) wait;
       write_bus;
}
```

A more formal view of a process is 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 one clock period per instruction, 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	...
r_1	0	1	0	0	0	0	...

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.

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 $\mathcal{I}$ and $\mathcal{O}$ denote the sets of input and output variables, repsectively. Let $A = (\mathcal{I} \cup \mathcal{O})$.*

A **synchronous trace** *T, or* **trace** *for short, is an element of the set $\Sigma = (B^{|A|})^\omega$. A* **process** *$P$ is a set of traces, i.e. a subset of $(B^{|A|})^\omega$.* □

Finite representations of processes are necessary for their rapid manipulation. ω-Automata have been proposed for this purpose in a verification context [7].

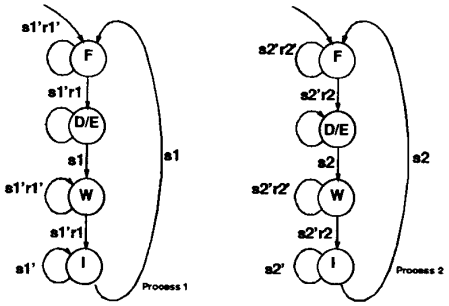

Figure 1: Automata for the processes $P1$ and $P2$ of Example (1). F indicate initial states.

An ω-automaton is described by a finite set $\mathcal{S}$ of states, a subset $\mathcal{S}_0 \subseteq \mathcal{S}$ 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 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 $s_0 \in \mathcal{S}_0$ 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]

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. The two processes $P1$ and $P2$ interface to a bus. To prevent simultaneous reads and writes on the bus, the bus protocol forces 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). □

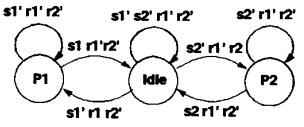

Figure 2: Automaton for the bus of Example (3).

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:

$$S = \otimes_{i=1}^{N} S_i \qquad (1)$$

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 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 processes, with the possibility of specifying timing and data-dependent synchronization constraints. These processes are translated into an ω-automaton representation. This latter representation constitutes a part of 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. A *timing constraint* between two processes $P1$ and $P2$ is any constraint placed on the traces of $done_{P1}$ and $start_{P2}$.

The timing constraints considered in *HardwareC* are 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. 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 can be found in [6]. In the rest of this section we illustrate the transformation of these constraints into an automata format.

3.1 ω-automaton construction.

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 a proces activate upon termination of *at least one* of a set of other processes.

Timing constraints.
Interval-type timing constraints are represented by automata like the one shown in Fig. (3). The automaton recognizes sequences of values on $Done_{Pi}, Start_{Pj}$ formed by a pulse on $Done_{Pi}$ followed by a pulse on $Start_{Pj}$. The duration of each pulse *must* be exactly 1 clock cycle. $Start_{Pj}$ must trail $Done_{Pi}$ by at least Min clock cycles and at most Max clock cycles.

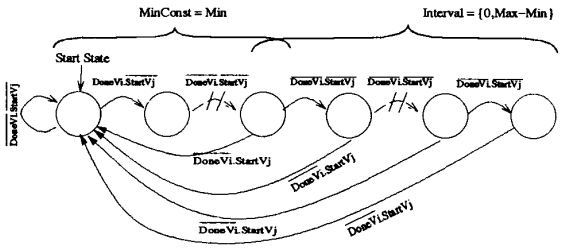

Figure 3: Automaton for timing constraints between two processes v_i and v_j

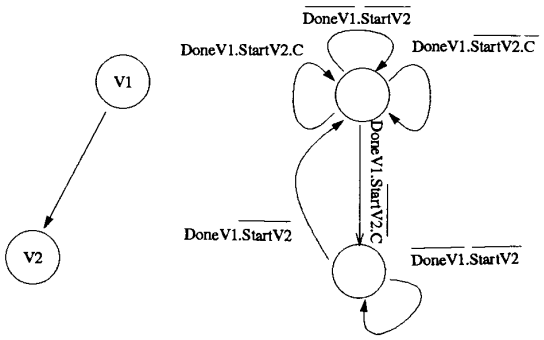

Figure 4: Firing of $V2$ conditional to a Boolean flag C.

Conditional execution of processes.
Data values 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* the result of some , say *if*, statement is **TRUE**. This is represented by a labeled edge on the sequencing graph, as shown in Fig. (4-a). The corresponding automaton is shown in Fig. (4-b). The condition for firing $P2$ is sampled at the time point where $Done_{P1} = 1$.

The "OR" activation constraint.
The following construction is employed. The activation condition for an "OR" activated process P_1 is the union of the *Done* signals of its activators: In this case,

$$Start_{P_1} = Done_{P_2} + Done_{P_3} \qquad (2)$$

We want, however, only one firing, regardless of how many activators issue a "done". So the automaton is more complex than the other ones.

3.2 System representation.

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, whose output takes value 1 corresponding to valid transitions.

Example 4. The circuit of Fig. (5) implements the automaton for a timing constraint on $Start_{P2}$ after 1 clock cycle of $Done_{P1}$. □

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.

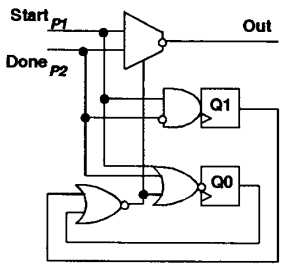

Figure 5: Synchronous circuit implementation of a constraint automaton.

4 Experimental Results.

The traversal of the state space of the product automaton is performed by a state traversal of the corresponding circuit *constrained* to valid transitions only. In this section we present experimental results on state-space traversal. To be able to traverse the state-space is important since it implicitly representis the design space of control.

We adopt implicit state enumeration of FSMs using BDDs [3][9]. The key figure of merit is the size of the BDD for the 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 benchmarks were initially written in *HardwareC*. Their sequencing graphs were derived and the automata modeling the timing and synchronization constraints were constructed. In Table (1), column **Nb. edges** indicates the number of edges in the sequencing graph, corresponding to the number of automata that were constructed. 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. Table (1) 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. The major bottleneck in traversing the automata is the construction of the BDD corresponding to the set of valid transitions.

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_dec.	79808	501	198.5
daio_receiver	104	210	61.8
diff_eq	5866	1151	265.1

Table 1: Automata characteristics

This can be overcome, however, by adding automata one at a time and traversing each partial product. We are also investigating better variable ordering heuristics for the BDD construction, tailored to the type of problem at hand.

5 Conclusions and future work.

In this paper we have presented a framework for modeling degrees of freedom at the behavioral level based on automata. We model hardware as a set of interacting sequential processes, where a process is defined by a set of acceptable execution traces.

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 automaton can be traversed quickly and manipulated using implicit state enumeration methods based on BDDs.

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

We are currently developing a set of algorithms to perform synthesis and optimization by extracting the *don't care* information that exist in the automata [11, 4].

6 Acknowledgments.

The authors thank Jerry Burch and David Filo for the discussions on trace-based models. This research is sponsored in part by NSF/ARPA under grant MIP 91-15-432.

References

[1] R. A. Bergamaschi and D. Lobo A. Kuehlmann. Control optimization in high-level synthesis using behavioral don't cares. In *Proceedings of the Design Automation Conference*, pages 657–661, Anaheim, CA, June 1992.

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

[3] O. Coudert and J.C. Madre. A unified framework for the formal verification of sequential circuits. In *ICCAD, Proceedings of the International Conference on Computer-Aided Design*, pages 126–129, November 1990.

[4] M. Damiani. Nondeterministic finite-state machines and sequential *don't cares*. In *EDAC, Proceedings of the European Design Automation Conference*, February 1994.

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

[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. 10(No. 11), November 1991.

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

[9] H. Touati, H. Savoj, B. Lin, R.K. 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*, pages 130–133, November 1990.

[10] 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*, pages 182–187, Anaheim, CA, June 1992.

[11] J. C.-Y. Yang, G. De MIcheli, and M. Damiani. Scheduling with environmental constraints based on automata representations. In *EDAC, Proceedings of the European Design Automation Conference*, February 1994.