DECOMPOSITION OF MULTI-PHASE TIMED FINITE STATE MACHINES

by

Szu-Tsung Cheng and Robert K. Brayton

Memorandum No. UCB/ERL M95/67

14 September 1995

COVER PAGE

DECOMPOSITION OF MULTI-PHASE TIMED FINITE STATE MACHINES

by

Szu-Tsung Cheng and Robert K. Brayton

Memorandum No. UCB/ERL M95/67

14 September 1995

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Decomposition of Multi-Phase Timed Finite State Machines

Szu-Tsung Cheng * Robert K. Brayton*

Abstract

We present a novel approach to synthesizing hardware implementation from Hardware Description Language (HDL) programs that cannot be automatically synthesized before. We deal with multi-phase/ multi-stage designs, and demonstrate that this problem can be mapped into a class of timed automata which is called "multi-phase" finite state machines (FSM). We propose three procedures to decompose a multi-phase FSM into a network of interacting single-phase FSMs. The first two procedures are based on the region graph expansion of a timed automata [AD90]. The first procedure extracts single-phase FSMs by traversing a region graph. The second procedure formulates the region graph decomposition as an integer linear programming. These two region-graph-based procedures may have an explosion in the number of states. The third procedure, without building intermediate transition structures, constructs single-phase FSMs directly from the transition structure of a multi-phase FSM. It is more efficient but redundancy might exist in the constructed FSMs. Not only can these procedures be used for the synthesis from a multi-phase design, they can also be used to speed up FSM-based simulation.

*EECS Department, University of California, Berkeley

1 Introduction

Hardware description languages (HDL) for event-driven simulation (e.g., VHDL[vhd88], Verilog [TM91]) have been widely used in circuit and system design. These HDLs are designed for efficient simulation, but their formal semantics are not predetermined. Therefore, it is possible to write an HDL program which has no corresponding hardware implementation. Some algorithms [syn94, vhd92] have been proposed to compile some HDL programs into implementations. However, these state-of-the-art HDL compilers are limited to a subset of the language as well as to single-phase processes. In some designs, like multi-phase pipeline circuits, multi-phase protocols, etc., this is too restrictive.

The synchronous finite state machine (FSM) model used in hardware synthesis interprets transitions as controlled by a central clock [Koh78, SSL+92, VSV90, LN89, DHLN91]. On the other hand, multi-phase FSMs are useful to model HDL programs in which a process is controlled by several clocking schemes. In a multi-phase FSM, whether a transition is made depends on the "phase" of the current state. Unfortunately, without analyzing the phase relationship among different clocking schemes, it is impossible to obtain a correct implementation for a multi-phase FSM. As a result, timing information must be preserved in the modeling of HDL programs so that synthesis can be performed.

Timed automata [AD90] are ordinary automata augmented with timing elements for modeling real-time systems. They have been successfully used in verification of timing constrained systems [AD90, ACH+92, CDHWT92, LB93, LB94, BSV92, BBC+95]. In [CBY+95] timed automata were proposed to model a general class of Verilog programs. The extracted information can also be used for formal verification [ABB+94, BBC+95].

In this paper, we address the problem of synthesizing circuits from multi-phase FSMs and multi-phase HDL programs. First, we give some motivation and definitions in Section 2. We demonstrate that a multi-phase FSM can be decomposed into a network of interacting single-phase FSMs. Two decomposition procedures based on *region graphs* are proposed in Sections 3.1 and 3.2. One is based on region graph traversal, the other on an integer programming formulation. In Section 3.3, we present a more efficient procedure. These techniques are HDL independent since they are based on a formal model, "multi-phase FSMs". More discussion on these approaches is given in Section 4. Two applications (for synthesis and for FSM simulation) are presented in Section 5. In Section 6, we present some preliminary experimental results on the size of the single phase FSMs versus the more compact multi-phase FSMs.

Our main contribution is that we provide a technique for synthesizing multi-phase HDL programs which also lead to more efficient simulation techniques as well (see section 5).

2 Motivation and Definitions

When HDLs like Verilog and VHDL are used for synthesis, it is not easy to make circuits out of general HDL programs even though these languages are called "hardware description languages". Two difficulties arise. First, these languages were designed for efficient simulation. No formal semantics exist; their semantics semantics depend on the specific schedule a simulator chooses at run time. Therefore, it is hard to "guess"

```
always                              process
   begin                               begin
   @(posedge phi1) i_fetch;            wait until (phi1'event and phi1 = '1'); i_fetch;
   @(posedge phi2) ...                 wait until (phi2'event and phi2 = '1'); ...
   @(posedge phi3) ...                 wait until (phi3'event and phi3 = '1'); ...
   end                                 end
                                    process
```

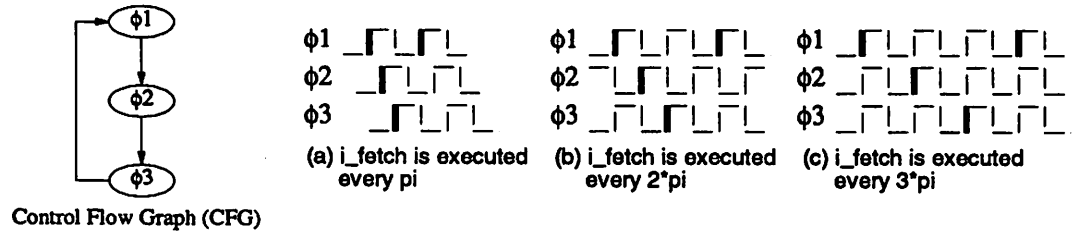


Table 1: A multi-phase system in Verilog/VHDL and the impact of clock configurations on system behavior.

the behavior of a program without a simulator. Second, HDLs are so powerful that they can describe behaviors that are difficult to emulate by a circuit. For example, due to delta-time in Verilog and VHDL, it is possible to perform an infinite number of calculations in zero hardware time.

Nonetheless, it is attractive to be able to synthesize HDL programs. Typically a subset of the HDL (synthesizable subset) [syn94, vhd92] is used for synthesizing circuits. Among other things, these methods can synthesize single-phase processes into sequential circuits. A single-phase process is one that may have multiple synchronization points, but each is controlled by a single clocking signal. This requirement may be limiting for applications like multi-phase system design. For example, designs like the one in Table 1 are excluded from the "synthesizable subset" due to the above restriction.

In this paper we are concerned with multi-phase synchronous HDL programs not only for synthesis but also for efficient simulation (see section 5). We assume that a design is delay independent: all delay information can be discarded. An HDL program is modeled by a sequencer and a datapath. Data-paths are allocated to emulate the data operations of a program. A sequencer is used to control register access. Control flow graphs (CFG) are used to represent the abstract execution of an HDL program. But a control flow graph alone does not provide enough information to devise a sequencer. Consider, for instance, the simplified pipeline processor in Table 1 (a Verilog program on the left, a VHDL process for the same processor on the right). Assume that `phi1`, `phi2`, and `phi3` are running at the same frequency (with period π). `i_fetch` may get executed every one, two, or three π time units, depending on the phase relationship among `phi1`, `phi2`, and `phi3`. Hence, register variables touched by `i_fetch` get updated every one, two, or three π. Note that these cases have the same control flow graph. Without the timing information from a multi-phase HDL program, the implementation of the program cannot be determined.

Definition 2.1 (Phase) *A* phase ϕ *is a pair* (π, ρ) *where* $\rho \in \mathcal{R}^+ \cup \{0\}, \pi \in \mathcal{N}$. ϕ *is a periodical event. Event* ϕ *is generated every* π *time units. It is first emitted at* $\rho + \pi$ *time units.*

Definition 2.2 (Multi-Phase FSM) *A Finite State Machine (FSM)* M *is a tuple* $< Q, q_0, \Sigma, \delta, \lambda >$. Q *is the set of states.* $q_0 \in Q$, *is the initial state.* $\Sigma = I \cup O$ *is the input/output alphabet of* M. δ *is the transition function,* λ *is the output function.* $\delta : Q \times I \rightarrow Q$. $\lambda : Q \times I \rightarrow O$. δ *and* λ *are completely specified.*
A Multi-Phase FSM *(MPFSM)* M *is a tuple* $< Q, q_0, \Sigma, \delta, \lambda, \phi >$. $\phi : Q \rightarrow \mathcal{N} \times [\mathcal{R}^+ \cup \{0\}]$. *It is a FSM in which each state is labelled with a phase* $\phi(s)$. *A* Single-Phase FSM *(SPFSM) is a special case of multi-phase FSMs where all states are labeled by the same phase. A* ϕ-machine *is a single-phase FSM that is controlled by phase* ϕ.
The set of all phases used in M *is denoted by* $\Phi(M) = \{\phi_s \in \mathcal{N} \times [\mathcal{R}^+ \cup \{0\}] \mid s \in Q, \phi(s) = \phi_s\}$.

If a state s in a multi-phase FSM is labelled with phase $\phi(s) = (\pi, \rho)$, then a transition from s can be made only when the corresponding phase event happens (and the labels on the transition are met). We use multi-phase FSMs to model HDL programs in the following way. First, a CFG is extracted from an HDL program. For each event control node (node that corresponds to `@(...)` in Verilog, or `wait` in VHDL) in the CFG, a multi-phase FSM state is allocated. Transitions in the multi-phase FSM are derived by an event control to event control path traversal [CB94]. In this paper, we assume that all synchronization signals are periodic and the phase relationships among them are known.

3 Decomposing Multi-Phase Finite State Machines Into Single-Phase Finite State Machines

We present three algorithms that extract implementations from multi-phase FSMs. An extracted implementation consists of a network of interacting single-phase FSMs. Each single-phase machine can be regarded as a FSM in traditional sequential synthesis. The first two algorithms first build a timed automata that emulates a multi-phase FSM derived from an HDL program. A region graph is then constructed from the timed automata [AD90, ACH+92]. A region graph is a untimed transition structure where timing information is incorporated in each *region.* The first algorithm decouples the region graph into a set of interacting single-phase FSMs by graph traversal. The second algorithm uses integer programming to solve the single-phase FSM extraction from a region graph. The third algorithm decomposes a multi-phase FSM into a *control-token network* and a set of *difference counters.* The control-token network has a structure similar to the transition structure of the multi-phase FSM. Control token passing inside the network depends on the state of difference counters. Difference counters are used to provide each single-phase FSM with information regarding relative phases of the other clocks. Although the region-graph-based algorithms have more opportunity for optimization, they can encounter a size explosion during region graph construction. On the other hand, the direct structure-based algorithm is more efficient, but it may generate a circuit that is not optimal.

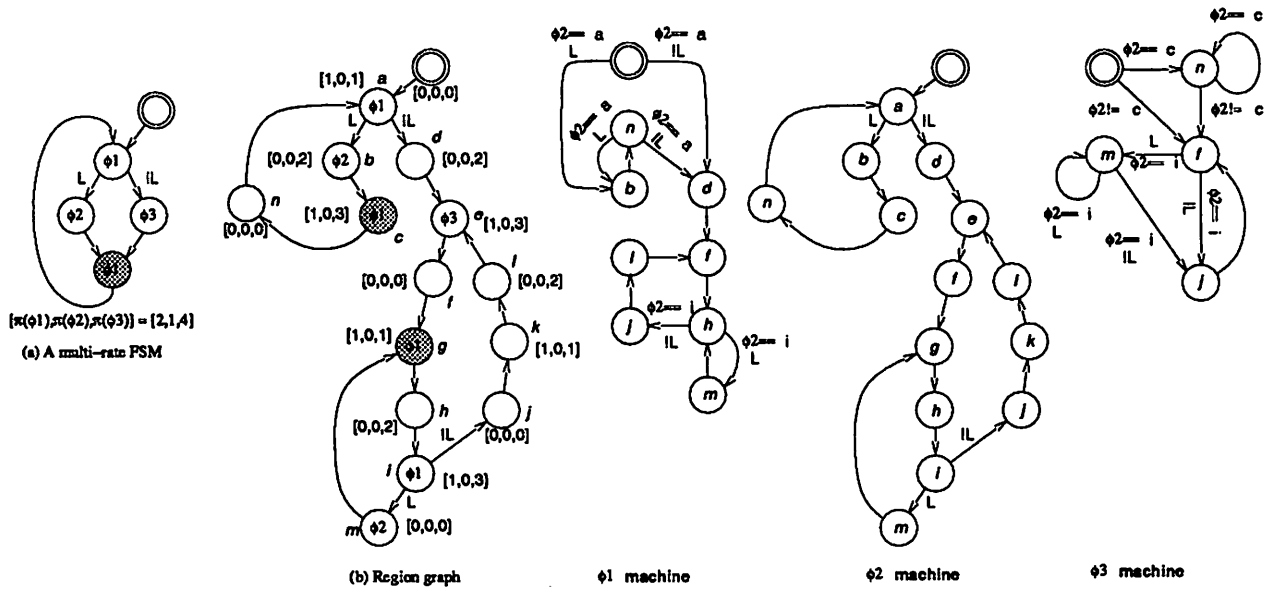


Figure 1: A multi-phase FSM, its region graph, and extracted single-phase FSMs.

3.1 Algorithm I - Decomposing Region Graph of a Multi-Phase FSM Using Graph Traversal

Given a multi-phase FSM, the way a state responds to a sequence of inputs may depend on the configuration of the clock that control the progress of the MPFSM. For instance, consider Figure 1.a. The periods of the clocking signals ϕ_1, ϕ_2, and ϕ_3 are 2, 1, and 4, respectively. Assume that L is false and the machine has just reached unshaded ϕ_1 from the starting state (where all clocks are reset to zero), then after 4 time units, shaded ϕ_1 will be visited. Afterward, if unshaded ϕ_1 is visited again and L is false, shaded ϕ_1 can not be visited in 6 time units.

Snapshots [AD90] are used to differentiate a state with various clock configurations. A snapshot is a pair consisting of a state in the automaton and assignments to timer variables. However, there is an infinite number of snapshots. Therefore, transition systems built from snapshots may be infinite. Fortunately, for our application, many snapshots are equivalent and the number of equivalence classes is finite. Each equivalence class forms a *region* [AD90, ACH+92].

In this section, a set of timed automata is used to emulate a multi-phase FSM. The emulating timed automata is then expanded to derive a region graph [ACH+92, CDHWT92]. We then present a class of procedures that produce single-phase FSMs by walking the extracted region graph. The quality of the generated single-phase FSMs depends heavily on the way each transition is generated and labeled.

A multi-phase FSM $M =< Q, q_0, \Sigma, \delta, \lambda >$ can be modeled by the product of a set of timed automata. Basically, we create a phase signal generator for each synchronization signal. We then copy the transition graph of M. Each transition is controlled by associated labels as well as the phase of the present state. In general, a timed automata T is built from M according to the following rules.

- Label each transition $(s, i * o, t)$ of M with $\phi(s)$.

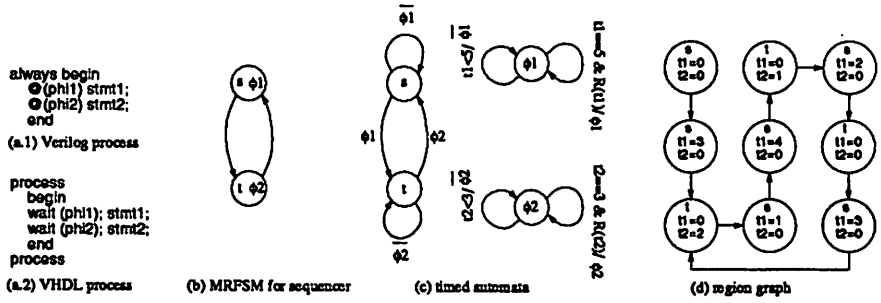


Figure 2: Region expansion and multi-phase FSM of a hardware description language (HDL) process.

- Add self-loop to each state s of M. The loop is labelled with $\overline{\phi(s)}$.
- For each $\phi = (\pi, \rho) \in \Phi(M)$,
  1. Allocate a state $\tilde{\phi}$.
  2. Allocate a timer t for $\tilde{\phi}$.
  3. Add a self-loop on state $\tilde{\phi}$ labelled with $\overline{\phi}, t < \pi$.
  4. Add a self-loop on state $\tilde{\phi}$ labelled with $\phi, t == \pi, Reset(t)$.
  5. If $\rho \neq 0$ allocate a state $\hat{\phi}$.
  6. If $\rho \neq 0$ add a transition from $\hat{\phi}$ to $\tilde{\phi}$ labelled with $t == \rho$.

After building an emulating timed automata for a MPFSM, the methods of [ACH+92, CDHWT92] can be used to derive its region graph R. Figure 2 shows an example HDL process (Verilog and VHDL), the multi-phase FSM extracted from the control flow graph of the process, and the timed automata constructed according to the above rules. In this example, $\phi(s) = \phi_1 = (\pi_1, \rho_1) = (5, 0)$ and $\phi(t) = \phi_2 = (\pi_2, \rho_2) = (3, 0)$. We choose the lowest boundary class [AD90] within each region as the representative for that region. For example, state $< s, (2 \leq x < 3, 0 \leq y < 1) >$ is represented as $< s, [2, 0] >$. For a region state $r \in R$, we also define the phase of the region, $\phi(r)$, to be the set of clocks that count from zero in that region. That is, on entering the region, phase signals in $\phi(r)$ are emitted and timers for those phase signals are reset.

Definition 3.1 (Phase of a region) *Given a region $\alpha =< s, (\tau_1, \ldots, \tau_n) >$ in a region graph R, $\phi_i \in \phi(\alpha)$ iff $0 \in \tau_i$.*

For example, Figure 1.b is the region graph derived from Figure 1.a. $\phi(a) = \{\phi_1\}$ and $\phi(b) = \{\phi_1, \phi_2\}$.

The following procedure, for each phase ϕ_i, builds transitions for a ϕ_i-machine by enumerating paths on region graph R of a multi-phase FSM M. An enumerated path $p : s \rightsquigarrow t$ has to satisfy the following criteria:

1. $\phi(s) = \phi(t) = \phi_i$.
2. Except s and t, p contains no region r such that $\phi_i \in \phi(r)$.

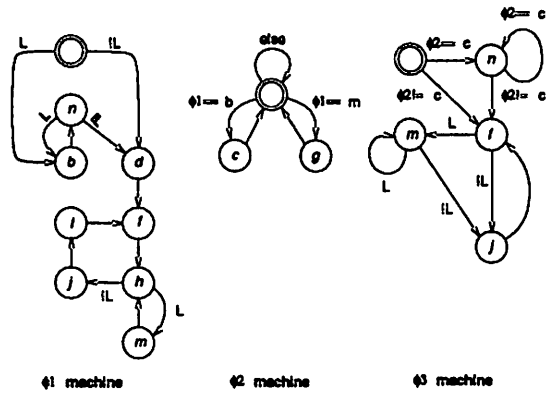


Figure 3: Relabelling ϕ_1 and ϕ_3 in Figure 1 to optimize the single-phase FSM for ϕ_2.

```
for φ_i ∈ φ(M)
   foreach simple path p : s ⇝ t, φ(s) = φ(t) = φ_i
         (except s and t, p passes no node whose phase is φ_i)
      generate a φ_i-machine transition, labelled with L(s ⇝ t).
```

There is a wide choice for $\mathcal{L}$. One possibility is:

$$\mathcal{L}(p : s \rightsquigarrow t) = \begin{cases} \text{unconditional} & p \text{ has no branch} \\ r & \text{predecessor, } r, \text{ of } t \text{ in } R \text{ has no branch} \\ r, L & t \text{ is on branch of } r \text{ labelled with } L. \end{cases} \quad (1)$$

Single-phase FSMs generated using this method leave a lot of space for optimization. For example, consider Figure 1. It shows ϕ_1, ϕ_2, and ϕ_3 machines that are extracted from Figure 1.b using region graph traversal and labelling function 1. Suppose that transitions of ϕ_1 and ϕ_3 machines are carefully relabelled as shown in Figure 3. ϕ_2 machine can be reduced to a 3-state machine from the original 12-state machine. The optimality of an implementation can be influenced by the way the states of a single-phase FSM are referred to by other machines.

3.2 Algorithm II - Extracting Single-Phase FSMs from Region Graph Using Integer Programming

In this section, integer programming [Pap82] is used to solve single-phase FSM extraction. Given a multi-phase FSM M and its region graph R, an integer programming instance is formulated in the following way.

For each state $s \in R$, allocate a variable n_s. n_s indicates if there is a single-phase FSM state that is allocated to represent region s. For each transition $s \rightarrow t \in R$, allocate a variable e_{st}. e_{st} indicates if region transition $s \rightarrow t$ should be identifiable with the produced single-phase FSMs. For each transition $< m, (\ldots) > \rightarrow t \in R$, if $\phi(m) \in \phi(t)$, then the transition is called "essential". The present region of an essential transition is called an essential region. If a transition is in a solution, then the present

and the next regions of the transition must also be in the solution. Finally, a solution must guarantee that all "essential transitions are reachable". Put these together, we have the following integer programming formulation.

$\min f$
s.t.

$e_{st} = 1$, if $s =< m, (\ldots) >$, and $\phi(m) \in \phi(t)$. /* essential transition */

$e_{st} \leq n_s$

$e_{st} \leq n_t$

$\sum_{s \to t} e_{st} \geq n_t$, for each $t \in R$./* if a state should be included in the solution */ set, at least one of its input arc must be included in the solution.

$\sum_{\substack{i \in S \\ j \in \overline{S}}} e_{ij} \geq 1$, for all nontrivial partition $(S, \overline{S})$ of regions of R such that there is an essential region $r \in \overline{S}$ and $< q_0, [0, \ldots, 0] > \in S$. /* essential transitions are reachable */

$e_{st}, n_s \in \{0, 1\}$

Cost function f can be any function involving e_{st} and n_s. For example, if the goal is to reduce the sum of numbers of states in extracted single-phase FSMs, then the cost function can be $\sum_{s \in R} n_s$.

Given a solution to the above integer programming, a set of single-phase FSMs are built using the following rules:

- For each $n_s = 1$ allocate a state in a single-phase FSM that is controlled by ϕ_i, $\phi_i \in \phi(s)$.
- For each $e_{st} = 1$, if
  - State α in phase ϕ_i SPFSM is allocated for region s.
  - State β in phase ϕ_j SPFSM is allocated for region t.
  - The original s to t region transition is $s \xrightarrow{\ell} t$.

  allocate a ϕ_j transition $(-, \alpha * \ell, \beta)$.
- For each ϕ_i-machine, if its transitions are not complete, then allocate a dummy state s_{ϕ_i} for ϕ_i-machine. Make ϕ_i machine complete by directing all unspecified transitions to s_{ϕ_i}.

By formulating the single-phase FSM extraction problem using integer programming, an exact solution can be found. However, the synthesis procedure suffers from both space and time explosion due to region graph construction and the complexity for solving an integer programming.

3.3 Algorithm III - Direct Construction

The problem with region-graph-based single-phase FSM extraction is that the intermediate representation, region graph, of a multi-phase FSM explodes pretty fast. In this section a procedure is proposed which builds single-phase FSMs directly from

the transition structure of a multi-phase FSM M. The procedure does not guarantee the optimality of the solutions it finds. However, since the procedure constructs single-phase FSMs by a direct mapping from a multi-phase FSM, the complexity is approximately the size of the input multi-phase FSM. To be more precise, the complexity of the procedure is $O(M) + O(\sum_{\substack{\phi_i, \phi_j \in \Phi(M) \\ \phi_i \neq \phi_j}} \frac{l.c.m.(\pi_i, \pi_j)}{\pi_j})$. Besides, the procedure can find good solutions for programs whose registers are controlled by clocking signals of the same frequency.

Instead of embedding the knowledge of the relative phase of other clocks into each state. The algorithm in this section uses a set of counters, *difference counters*, to count the status of other machines. A ϕ_j to ϕ_i counter, denoted by $\phi_j \# \phi_i$, is a $\frac{l.c.m.(\pi_i, \pi_j)}{\pi_i}$-state ring counter controlled by ϕ_i. States of $\phi_j \# \phi_i$ are labelled $\rho_i \bmod \pi_j, \rho_i + \pi_i \bmod \pi_j, \ldots, \rho_i + k\pi_i \bmod \pi_j, \ldots$. It constitutes ϕ_i-machine's perception of ϕ_j-machine's status.

For a multi-phase FSM, the extracted single-phase FSMs consist of two parts, namely the *control-token network* and the *difference counters*. A control-token network has a structure similar to the transition graph of the multi-phase FSM. It is used to designate the current focus of control. For each state $s \in M$, a register is allocated controlled by $\phi(s)$ (*control-token register*, denoted by $\mathcal{N}(s)$). [1] It is used to hold a control-token. If there is a transition $s \xrightarrow{\ell} t$ in M, then $\mathcal{N}(s)$ can pass token to $\mathcal{N}(t)$. Control-token passing among control-token registers is guarded by the states of difference counters as well as the conditional values that are sampled. Difference counters provide information regarding the validity of control-tokens. That is, from the status of difference counters, it can be determined whether a token is "stale", and whether a token needs to be held longer in order to be passed down to a slow reacting successor.

For example, assume that there is a transition $s \xrightarrow{\ell} t$ in a multi-phase FSM, $\phi(s) = \phi_j = (5,0)$, $\phi(t) = \phi_i = (3,0)$, and s is now holding a control-token. Assume that the state of $\phi_j \# \phi_i$ is 4. The control-token is stale and should be rejected by the $\phi(t)$-machine because from the content of $\phi_j \# \phi_i$ it is known that s has had the control token for 4 time units. Therefore, t should have received a control-token from s already (at 3 time units before current time slot). t should not get another control-token from s until s has a chance to refresh its state. As another example, assume there is a multi-phase FSM transition $s \xrightarrow{\ell} t$, $\phi(s) = \phi_j = (3,0), \phi(t) = \phi_i = (5,0)$, and s is holding a token to be passed to t. Assume $\phi_i \# \phi_j = 1$. Before s refreshes its content, which is 3 time units later, t does not have a chance to have a look at s's token. Therefore, if s relinquishes its control-token at this time, the token is going to get lost forever.

In general, assume there is a multi-phase FSM transition $s \xrightarrow{\ell} t$. A token from s is not stale if

$$\begin{cases} 1 \leq \phi(s) \# \phi(t) \leq \pi(\phi(t)), & \text{if } \pi(t) < \pi(s). \\ \text{the token is always fresh}, & \text{if } \pi(t) \geq \pi(s). \end{cases}$$

A token to t needs to be held, for at least one $\phi(s)$ cycle, until $0 \leq \phi(t) \# \phi(s) \leq \pi(\phi(s)) - 1$. After this, the control-token should be released, unless s receives another token.

Difference counters are allocated only for those multi-phase transitions which relate states of different phases. To be more precise, for each $s \xrightarrow{\ell} t$, if $\pi(\phi(s)) \neq \pi(\phi(t))$,

[1] s and $\mathcal{N}(s)$ are used interchangeably if it does not cause any confusion.

then $\phi(s)\#\phi(t)$ and $\phi(t)\#\phi(s)$ are allocated. In addition, if $\phi_j\#\phi_i$ has only one state (i.e., $\pi_j|\pi_i$), $\phi_j\#\phi_i$ is eliminated. In this case, a control-token is always fresh and it is not necessary to prolong the holding of a token.

For each state s in M, let $E_O(s) = \{s \to t_1, \ldots, s \to t_n\}$ denote the set of transitions from s. If $n > 1$ allocate a latch that is capable of recording $\{0, 1, \ldots, n-1\}$. The latch is called *branch indicator* and is used to record the labels that occur when s receives a control-token. A successor t of s determines if it should receive a control-token from s by looking at (1) s's branch indicator, (2) $\phi(s)\#\phi(t)$, and (3) $\mathcal{N}(s)$. The branch indicator for s is updated every time $\mathcal{N}(s)$ gets a control-token.

In summary, for a multi-phase FSM M, a set of single-phase FSMs are constructed according to the following procedure. enc is a function that maps a label ℓ on a transition $s \xrightarrow{\ell} t$ in M into {0, 1, ..., out-degree(s)-1}.

for each state $s \in M$ allocate a latch, named $\mathcal{N}(s)$, controlled by $\phi(s)$
for each transition $s \xrightarrow{\ell} t$,

1. Allocate $\phi(s)\#\phi(t)$. If it has only one state then remove it.
2. Allocate $\phi(t)\#\phi(s)$. If it has only one state then remove it.

for each state s with out-degree greater than 1
allocate a branch indicator $\mathcal{B}(s)$ controlled by $\phi(s)$
$\mathcal{B}(s)$ is synchronized with $\mathcal{N}(s)$ in the sense that
when $\mathcal{N}(s)$ receives control token, $\mathcal{B}(s)$ samples the occurred conditional values
for each edge $e : s \xrightarrow{\ell} t \in M$, the following rules determine
when each control-token register receives/releases control.

1. $\mathcal{N}(t)$ gets a control-token (set to 1) if:
   - $\mathcal{N}(s) == 1$.
   - $\mathcal{B}(s) ==$ enc(ℓ).
   - $1 \leq \phi(s)\#\phi(t) \leq \pi(\phi(t))$.
2. $\mathcal{N}(s)$ relinquishes its control-token (reset to 0) if:
   - $\phi(t)\#\phi(s)$ indicates that $\mathcal{N}(t)$ has gotten token in time.

Initially, for all $s \in M, s \neq q_0$, $\mathcal{N}(s)$ are set to zero, and $\mathcal{N}(q_0)$ is set to 1.

4 Comparison Between the Procedures

The procedures in Section 3.1 and Section 3.2 embed timing information into each region. In contrast, the procedure in Section 3.3 explicitly builds difference counters to keep track of the relative phase of various clocks. Obviously, the first two procedures have more room for optimizing the generated FSMs. However, the drawback of the region-graph-based procedures is that region graphs explode pretty fast even for moderate sized timed automata. In general, it might be worthwhile to try the

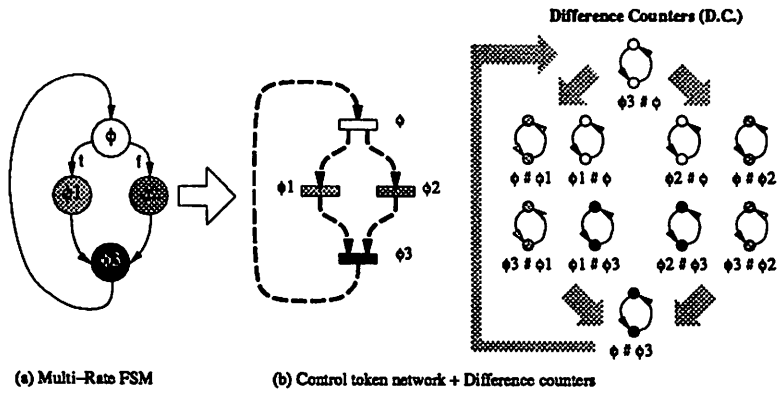


Figure 4: Decomposing a multi-phase FSM into a token network and difference counters.

third procedure. Furthermore, if transitions are generated carelessly, the quality of the implementations generated by the first procedure might be even worse than those generated by the direct construction procedure.

Consider the pipeline process in Table 2 and assume that `phi1`, `phi2`, and `phi3` are running at the same frequency (with period 3). We consider two initial phase configurations, $(\rho_1, \rho_2, \rho_3) = (0, 0, 0)$ and $(\rho_1, \rho_2, \rho_3) = (1, 2, 3)$. The direct construction algorithm always gets a three latch single-phase machine. There are no additional latches for difference counters (since each difference counter has only one state). If the first procedure is applied to the same problem and labelling function 1 is used, the results can be optimal (when $(\rho_1, \rho_2, \rho_3) = (1, 2, 3)$) or poor (when $(\rho_1, \rho_2, \rho_3) = (0, 0, 0)$). When $(\rho_1, \rho_2, \rho_3) = (1, 2, 3)$, the first procedure generates a network of one-state SPFSMs. Therefore, no additional latch is required for the sequencer of the program. This is obviously the optimum solution. If $(\rho_1, \rho_2, \rho_3) = (0, 0, 0)$, each single-phase machine has 3 states. Consequently, a total of 6 latches are required. This result is worse than that is produced by the direct construction procedure. But, if transitions are carefully relabelled then each single-phase machine needs only two states. This leads to a 3-latch implementation of the sequencer. A even smarter way to implement the multi-phase FSM is to use a single counter to count for the three single-phase FSMs. This leads to an optimum implementation which requires only 2 latches. This solution can be found by the second procedure.

5 Applications

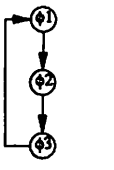
5.1 Hardware Synthesis:

The procedures proposed in this paper can be applied to the synthesis of HDL programs. For example, consider the following program segment:

```
always begin
   @(negedge phi1) r = a | b;
   @(negedge phi2) s = (x)?r:0;
   @(negedge phi3) t = arr[s];
```

Pipeline process:

```
always
   begin
   @(phi1) i_fetch;
   @(phi2) exec;
   @(phi3) w_back;
   end
```



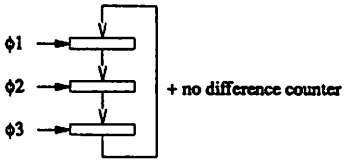
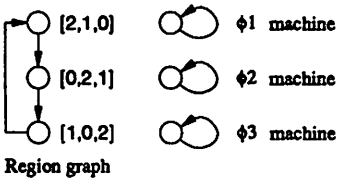
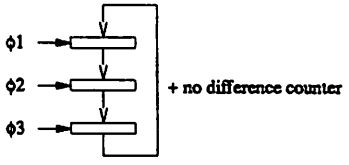
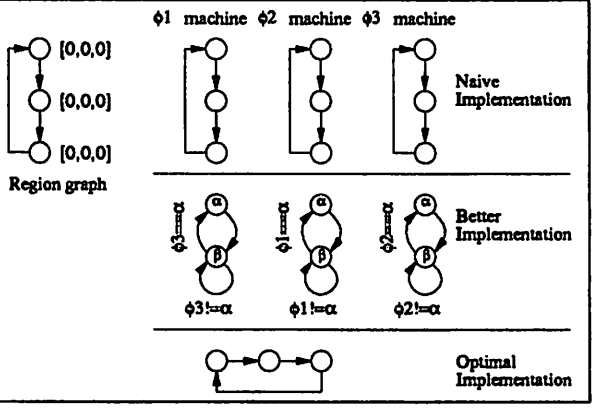
Phase configuration	Direct Construction	Region-based Construction
$\pi = [3,3,3]$, $\rho = [1,2,3]$		
$\pi = [3,3,3]$, $\rho = [0,0,0]$		

Table 2: A simplified multi-phase pipeline processor and the implementation extracted using various procedures for two phase configurations.

```
end
```

If `phi1`, `phi2`, `phi3` are three different synchronization signals, a traditional hardware compiler cannot yet handle such cases [syn94, vhd92]. However, processes like this can be modeled by a control flow graph (CFG) and a data-path [CBY+95]. The data-path is used to compute `a | b`, `(x)?r:0`, and `arr[s]`. The CFG is used to model the sequencing of the program and to schedule the updates on register variables `r`, `s`, and `t`. It is easy to synthesize hardware which implements the functionality of a data-path. Nonetheless, it is in general not easy to derive a sequencer from a CFG. If a program is multi-phase and there is no delay in it, then its CFG is a multi-phase FSM. We can apply the procedures presented in this paper to derive a network of interacting SPFSMs which can be used as an implementation of the CFG. The extracted single-phase machines can also be fed into a sequential synthesizer for a more optimized result. The controlling phase of a variable can be determined by a backward CFG traversal from the point where an assignment to that variable is made. For example, in the above example, `r` is controlled by `phi1` and `s` is controlled by `phi2`. A realization of an

HDL program is simply the implementations for datapath, CFG, and each variable,

5.2 FSM-Based Simulation:

Another application is FSM-based simulation. Consider the previous example. If the whole pipeline machine is compiled into a single multi-phase FSM [CBY+95], then on the occurrence of `phi1`, all data-path for `r`, `s`, and `t` need to be reevaluated. This is because there is no way to tell the phase of each variable. On the other hand, if we can determine the phase and controlling logic for each variable, then we do not need to evaluate logic cones for `s` and `t` when `phi1` occurs. i.e., approximately two thirds of the computation can be eliminated. This saves a lot of unnecessary computation which only ends up by updating latches with their current states. Furthermore, if [CBY+95] is used to generate timed FSMs for modeling a multi-phase process, then the produced logic contains many "edge detectors". These are used to detect the occurrence of a specific signal transition. Given the information regarding the period and initial phase of each synchronization signal, we can bypass these edge detectors but still determine the timing/sequencing of expression-evaluation/variable-update. Thus, more savings can be achieved by avoiding evaluating edge detectors.

Note that even though the implementation of a process may not exist, the proposed technique still applies for simulation. For example,

```
always begin
   @(negedge phi1) r = a | b;
   @(negedge phi2) r = (x)?r:0;
   @(negedge phi3) t = r - t;
   end
```

In this example, `r` is controlled by both `phi1` and `phi2`. In reality, we do not yet know what is the synchronous implementation for such a variable. However, by separating the process into a network of single-phase machines, the phases and control logic for each variable can be decided. Therefore, the logic cone for `t` (`r`) need not to be evaluated when `phi1` or `phi2` (`phi3`) occurs.

6 Experimental Results

We have implemented the direct construction procedure in a Verilog to FSM compiler. This compiler takes programs in a subset of Verilog as defined in [CBY+95]. The resultant implementations can be represented in either SMV [McM94] or BLIF-MV [BCH+91]. Table 3 shows experimental results for some Verilog programs. Programs `tmst1`, `tmst2`, `tmst3`, `tmmx1`, `tmmx2`, and `tmmx3` contain a mixtures of conditional expressions and event-controls. Programs `pipe1`, `pipe2`, and `pipe3` are pipeline processes with a varying number of stages and synchronization signals.

MPFSMs are a compact representation for HDL programs. However, when it comes to implementation, timing analysis must be performed. In contrast, the size of the SPFSM networks in our experiments is substantially larger. This is due to the "difference counters" which differentiate states in various time periods. Our preliminary

Program	# regions	SPFSM		MPFSM	
		cpu	space	cpu	space
tmst1	15	0.07	6	0.046	6
tmst2	7.2×10^{12}	1.6	336	1.152	78
tmst3	3.26×10^{19}	3.5	704	2.386	160
tmmx1	-	0.554	112	0.332	26
tmmx2	-	0.652	120	0.445	32
tmmx3	-	1.014	192	0.664	44
pipe1	-	0.058	4	0.039	3
pipe2	-	0.187	27	0.140	12
pipe3	-	0.261	40	0.195	15

Table 3: Experimental results of SPFSM extraction.

# regions:	number of regions in the region graph of the controller of a program.
SPFSM:	use direct construction procedure to build SPFSMs.
MPFSM:	use [CBY+95] to build timed automata to model a program.
cpu:	CPU time in seconds on a DEC 5000/125 with 64MB memory
space:	size of the generated circuits (datapath + controller, represented in SMV+) in kilo-bytes.

results indicate that our direct construction procedure looks promising for synthesizing programs with a large region space.

7 Conclusions and Future Work

We proposed three procedures to extract implementations from multi-phase HDL programs. Two are based on region graph expansion of a timed automata. Although region-graph-based algorithms are capable of generating better results, state explosion during region graph extraction can make them impractical. The third procedure maps the transition structure of a MPFSM directly into an implementation. Since the number of states in a MPFSM is proportional to the number of event controls in an HDL program, this procedure is the most efficient. However, it may produce results that are not optimal in size. These techniques can also be used to speed up FSM-based simulation.

Future work will extend this work to incorporate various HDL timing constructs. In addition, we would like to improve the integer linear programming based procedure in Section 3.2 so that it can find an optimal solution under practical space/time constraints.

References

[ABB+94] Adnan Aziz, Felice Balarin, Robert K. Brayton, Szu-Tsung Cheng, Ramin Hojati, Sriram C. Krishnan, Rajeev K. Ranjan, Alberto L. Sangiovanni-Vincentelli, Thomas R. Shiple, Timothy Kam Vigyan Sing-

hal, Serdar Tasiran, and Huey-Yih Wang. HSIS: A BDD-based environment for formal verification. In *DAC94*, San Diego, CA, June 1994.

[ACH+92] Rajeev Alur, C. Courcoubetis, N. Halbwachs, David Dill, and H. Wong-Toi. Minimization of timed transition systems (extended abstract). In *Proceedings of CONCUR'92*, Stony Brook, N. Y., 1992.

[AD90] Rajeev Alur and David Dill. Automata for modeling real-time systems. In *Proceedings of 17th International Colloquium on Automata, Languages and Programming.* Springer Verlag, 1990. LNCS 443.

[BBC+95] Felice Balarin, Robert K. Brayton, Szu-Tsung Cheng, Desmond A. Kirkpatrick, Alberto L. Sangiovanni-Vincentelli, and Ephrem Wu. A methodology for formal verification of real-time systems. Memorandum UCB/ERL M95/11, University of California at Berkeley, 1995.

[BCH+91] R. K. Brayton, M. Chiodo, R. Hojati, T. Kam, K. Kodandapani, R. P. Kurshan, S. Malik, A. Sangiovanni-Vincentelli, E. M. Sentovich, T. Shiple, K. J. Singh, and H.-Y. Wang. BLIF-MV: An interchange format for design verification and synthesis. Memorandum UCB/ERL M91/97, University of California at Berkeley, 1991.

[BSV92] Felice Balarin and Alberto Sangiovanni-Vincentelli. A verification strategy for timing constrainted systems. In *International Conference on Computer-Aided-Verification*, 1992.

[CB94] Szu-Tsung Cheng and Robert K. Brayton. Compiling verilog into automata. Memorandum UCB/ERL M94/37, University of California at Berkeley, 1994.

[CBY+95] Szu-Tsung Cheng, Robert K. Brayton, Gary York, Katherine Yelick, and Alexander Saldanha. Compiling verilog into timed finite state machines. In *Proceedings of Internation Verilog Conference.* IEEE, March 1995.

[CDHWT92] C. Courcoubetis, David Dill, N. Halbwachs, and H. Wong-Toi. An implementation of three algorithms for timing verification based on automata emptiness. In *Real-Time Systems Symposium.* IEEE, 1992.

[DHLN91] X. Du, G. D. Hachtel, B. Lin, and A. R. Newton. MUSE : A Multilevel Symbolic Encoding Algorithm for State Assignment. In *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, pages 28–38, January 1991.

[Koh78] Zvi Kohavi. *Switching and Finite Automata Theory.* McGraw-Hill Book Company, New York, 1978.

[LB93] William Lam and Robert K. Brayton. Alternating RQ timed automata. In *International Conference on Computer Aided Verification*, 1993.

[LB94] William Lam and Robert K. Brayton. Criteria for the simple path property in timed automata. In *Computer Aided Verification*, 1994.

[LN89] B. Lin and A. R. Newton. A Generalized Approach to the Constrained Cubical Imbedding Problem. In *International Conference on Computer Design*, October 1989.

[McM94] Kenneth L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1994.

[Pap82] Christos H. Papadimitriou. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1982.

[SSL+92] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. SIS: A System for Sequential Circuit Synthesis. Technical Report Memorandum UCB/ERL M92/41, University of California at Berkeley, University of California, Berkeley, May 1992.

[syn94] *Synergy HDL Synthesizer and Optimizer Modeling Style Guild Version 2.0.* Cadence Design Systems, Inc., 1994.

[TM91] Donald E. Thomas and Philip R. Moorby. *The Verilog Hardware Description Language.* Kluwer Academic Publishers, Nowell, Massachusetts, 1991.

[vhd88] *IEEE Standard VHDL Language Reference Manual.* The Institute of Electrical and Electronics Engineers, Inc., New York, 1988.

[vhd92] *Synopsys VHDL Compiler Reference Manual Version 3.0.* Synopsys, Inc., 1992.

[VSV90] T. Villa and A. Sangiovanni-Vincentelli. NOVA: State Assignment for Optimal Two-level Logic Implementations. In *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, pages 905–924, September 1990.