The Minimization and Decomposition of Interface State Machines
Ajay J. Daga, William P. Birmingham
EECS Department
The University of Michigan
Ann Arbor, MI, 48109

Abstract—There is a well-recognized need for accurate timing-verification tools that account for the functional behavior of component interfaces, and thereby do not traverse false sequential and combinational paths. Such tools, however, are susceptible to an exponential increase in task complexity as circuit-size and functional-complexity of component interfaces increase. The viability of accurate timing verifiers, therefore, hinges on their ability to efficiently analyze the smallest subset of circuit behaviors, while verifying timing characteristics of the overall space of behaviors. This paper presents theoretical results that address this issue for the timing-verification of circuits composed of interacting state machines.

1. Introduction
Complex VLSI and board-level circuits may be modeled as being composed of interacting synchronous and asynchronous state machines. Designers of such circuits need to ensure that circuit behavior conforms to specifications. There are two aspects to the verification task. Functional verification establishes if components correctly implement the desired functionality. Given that a circuit is functionally correct, timing verification determines if components take too much or too little time to compute functions and transfer information among each other. The timing-verification task takes as input a circuit specification and the interface behavior of components in the circuit, and verifies the satisfaction of timing constraints at these component interfaces for all possible circuit behaviors.

Until recently, timing-verification methods [1, 2] ignored the functional behavior of component interfaces, and as a result reported conservative results (false violations). The need for more accurate timing-analysis tools is now apparent. Functional specifications, such as state machines for sequential components and truth tables for combinational components, facilitate accurate timing analysis by providing the information necessary to reject false sequential and combinational paths. A false combinational path [3] cannot be sensitized given the logic behavior of components in that path and signal activity applied to the combinational logic. A false sequencing path [4] is a sequence of states that is not consistent with the state-transition behavior of a component and signal activity applied to it.

Timing verification in the presence of functional information is susceptible to an exponential increase in task complexity as circuit size (number of interconnect signals) and functional-complexity (complexity of state-transition behavior and number of timing constraints at component interfaces) increase. This stems from the huge space of functional and timing behaviors that most realistic circuits exhibit, and the apparent need to represent and analyze each of these behaviors to perform timing verification [4]. The complexity of the task approaches \( O(c^s) \), where \( s \) is the average number of component-interface states and \( c \) the number of components in a circuit.

We recognize, however, that the timing-verification task is only required to verify the time-separation between events on a circuit. Consequently, circuit behaviors that are functionally dissimilar may, in fact, be equivalent from a timing-verification standpoint. It should, therefore, be possible to verify the timing characteristics of the overall space of circuit behaviors by efficiently analyzing a small subset of this space. In fact, the viability of a timing-verification tool hinges critically on its ability to do so [5]. This, in turn, requires addressing the following fundamental issue in a formal manner:

- Under what conditions are functionally-dissimilar circuit behaviors timing equivalent?

We address this issue by introducing a notion of signal activity to capture both functional and temporal aspects of observed circuit behavior and specified component-interface behavior. We present a criterion for signal-activity equivalence, and prove its correctness. This criterion is used to prevent the analysis of redundant circuit behavior during timing verification and also to identify timing-equivalent states in a state machine prior to timing verification. We extend the classical state-minimization technique [6] to identify timing-equivalent states in a completely specified interface state machine.

The signal-activity-equivalence criterion reduces timing-verification task complexity by ensuring that only the relevant subset of circuit behaviors are analyzed. In effect, this reduces the task-complexity to \( O(s_j^2) \), where \( s_j \ll s \). The task still remains computationally expensive because of the apparent need to investigate and represent all reachable circuit states (that are unique from a timing-verification standpoint) from the reset state of a circuit, and then analyze this composite representation of circuit behavior to perform timing verification [4].

We recognize, however, that interactions among interface state machines are inherently modular. These interactions primarily implement the exchange of information among state machines. An interface state machine may, therefore, be viewed as having distinct, independent, modes of operation, wherein the activity initiated by one mode of operation is entirely independent of that initiated by another. As a result, interface state machines are decomposable into multiple independent state machines (ISMs).

ISMs that are enabled based on primary-input signal values (denoted PISMs) initiate activity on a circuit. Activity initiated by a PISM is independent of the state a circuit is in due to the prior execution of another PISM. As a result, the timing-verification task may be performed by analyzing the interaction among component interfaces in the transitive fanout of each PISM. This approach does not require building and analyzing an exponentially large representation of reachable circuit states from the reset state of a circuit. The complexity of this approach is \( O(usa \times c) \) where \( usa \) is the average number of unique signal activities initiated at a component interface, and \( c \) the number of components in a circuit [5].

State-machine decomposition, therefore, allows a dramatic reduction in timing-verification task complexity. This paper presents a criterion and algorithm for interface-state-machine decomposition, and results from the application of this decomposition algorithm.

* This research was supported by Digital Equipment Corporation and the National Science Foundation grant MIPPS-905781. All views expressed here are those of the authors, and not necessarily those of the funding agencies.
The remainder of this paper is organized as follows. Section 2 presents a model of interface behavior that encapsulates both functional and timing specifications, and defines the notion of signal activity. Section 3 presents a criterion for signal-activity equivalence and proves its correctness. In Section 4 use of the signal-activity-equivalence criterion to minimize interface state machines is shown. Section 5 presents an algorithm for decomposition of minimal interface state machines and experimental results on its application. Results are summarized in Section 6. The Intel 82c288 bus-controller [7] is used to illustrate the various concepts in this paper.

2. Interface State Machines

A component may be viewed as consisting of interface circuitry and internal circuitry. Internal circuitry implements component functionality, while interface circuitry allows a component to communicate with its environment. The timing-verification task requires an abstract model for component-interface behavior, while viewing internal circuitry as a black box. We refer to this model of component-interface behavior as an interface state machine.

An interface state machine is composed of external and internal signals that either receive (input signals) or transmit (output signals) information. External signals are used to connect one or more component interfaces with each other. Internal signals are primary-input or primary-output signals on a component interface, and specify the stimuli that may be applied (received) to (from) a component interface by its internal circuitry. Figure 2.1 shows the signals on the 82c288 interface state machine.²

An interface state machine is composed of functional and timing specifications. Functional specifications establish how output-signal values are determined from input-signal values. For sequential components, a state-transition table provides a functional specification, while truth tables do so for combinational components.

![Figure 2.1: Structural model of 82c288 component interface.](image)

Timing specifications define the events and timing functions between events for each state in a component interface. An event is a change in signal value. There are two kinds of events: input and output. An input (output) event occurs on an input (output) signal. Events on the CLK signal in Figure 2.2 are examples of input events, while that on DEN is an example of an output event. There are two kinds of timing functions: delay and constraint. Delay functions specify the minimum and maximum occurrence time of an output event $e_{to}$ relative to an event $e_{from}$.³ Timing link $\tau_d$ in Figure 2.2 is a delay function: it establishes the delay in driving DEN high relative to an event on CLK. Constraint functions establish the minimum or maximum time at which an input event $e_{to}$ has to occur relative to an event $e_{from}$ for an interface to function correctly. Depending on whether a constraint function establishes a minimum or maximum event-separation requirement, it is classified as a min- or max-constraint function. Timing links $\tau_d$ and $\tau_h$ in Figure 2.2 are examples of min-constraints, and specify the setup and hold times that need to be satisfied when sampling READY. The constraint $\tau_C$ in Figure 2.2 is a max-constraint, and establishes the maximum permissible period of the CLK signal.

![Figure 2.2: Signal activity for 82c288 state IR.](image)

Figure 2.3 shows the state diagram for the 82c288 interface state machine, and Figure 2.4 shows the output-signal values for each state in Figure 2.3. All state transitions in Figure 2.3 occur at the falling edge of CLK. The state diagram in Figure 2.3 is minimal: transforming it to a Mealy machine, or application of the state-minimization procedure for fully-specified state machines [6] will not reduce the number of states. In state 1, the 82c288 samples the input signal STATUS. Based on this sampled value, the 82c288 issues either an I/O-read (states IR, RD_TR and R_IR), memory-read (states MR, RD_MR and R_MR), interrupt-acknowledge (states IA, RD_IA and R_IA), I/O-write (states W_IW, WD_IW, WD2_IW and IW) or memory-write (states W_MW, WD_MW, WD2_MW and MW) access. The specific access is indicated by setting to zero one of the bits in the five-bit wide signal CMD#. In each access, prior to entering the state in which information is exchanged, the 82c288 asserts ALE (ALE and DEN) in states with prefix "R_" ("W_"). If the signals are asserted in conformance with the Multibus timing protocol [7], or their assertion is delayed by the input signal CMDY, then the 82c288 enters states with prefix "RD_" ("WD_" and "WD2_") if a read (write) access is requested. Note the similarity in state-transition behavior among read accesses (shown to the left and top of state 1) and among write accesses (shown to the right of state 1).

![Figure 2.3: 82c288 state diagram.](image)

---

¹ A bi-directional signal is modeled as a pair of input and output signals.
² All signals are external signals. For simplicity we have eliminated some of the 82c288 signals.
³ The event $e_{from}$ occurs, or is required to occur, before $e_{to}$.
Figure 2.5 tabulates the timing specifications for each state in Figure 2.2. Output-signal values are driven with some delay relative to the CLK falling edge at which a state is entered. Input signals are sampled at the subsequent rising edge, as shown in Figure 2.4, and must satisfy setup and hold min-constraints. Figure 2.5 presents the timing specifications for each 82c288 state. Note that not all output signals are driven to a new value, nor are all input signals sampled in each state. Some signals retain their values in an earlier state. This manifests a "gated-clock" hardware implementation, where the clocking of input and output registers is a function of signal values.

![Timing Specifications Table]

Figure 2.5: Timing specifications for each 82c288 state.

2.1 Signal Activity

The combination of functional and timing specifications yields signal activity. Signal activity specifies the following for some time duration:

- events that occur (or are expected to occur) on output (input) signals,
- timing functions between events, and
- the event type for each event. Event type specifies the value taken by a signal after the occurrence of an event. Signal values are the following: any binary value in the range 0 to 2^N-1 for an n-bit wide signal, S (any stable signal value), X (indicating a don't-care value on an input signal and an unknown value on an output signal) and Z (high impedance).

Signal activity describes overall behavior (both functional and timing aspects) on a circuit or component interface for some time duration. For a synchronous interface, such as the 82c288, this time duration is the state transition time. The signal activity for 82c288 state I*R, for example, is shown in Figure 2.2.

Signal activity is represented using an event graph [8], which is a directed, acyclic graph (DAG) whose nodes represent events, and arcs represent timing functions between events. The set of events on an event graph EG is denoted E. Associated with each node e ∈ E are the following attributes: s(e), which indicates the signal on which the event occurs, and t(e), which indicates event type. There are two types of arcs (delay and constraint) on an event graph corresponding to the two kinds of timing functions. A delay (constraint) arc is directed from t' from e to t to e'from. The set of delay (constraint) arcs on an event graph EG is denoted DA (CA). Associated with each delay (constraint) arc a ∈ DA (a ∈ CA) is an attribute v(a) that specifies the minimum and (or) maximum delay (constraint) values. The event graph for the signal activity in Figure 2.2 is shown in Figure 2.6.

Each node in an event graph has at least one delay or constraint arc incident on, or emanating from, it. If the directionality of arcs is ignored, then an event graph is connected, i.e. it is possible to determine the expected or actual time-separation between an arbitrary pair of events.

![Event Graph Example]

Figure 2.6: Event graph for 82c288-state I*R.

3. Signal-Activity Equivalence

We present a criterion for signal-activity equivalence that is used to restrict the space of functional behaviors that need to be analyzed during timing verification.

**Theorem 1:**

Two signal activities are equivalent iff they have identical timing functions. From a graph-theoretic perspective the signal-activity-equivalence criterion is the following. If \( EG_1 = (E_1, DA_1, CA_1) \) and \( EG_2 = (E_2, DA_2, CA_2) \) are isomorphic event-graph representations of two signal activities \( SA_1 \) and \( SA_2 \) such that:

1. \( s(e_1) = s(e_2) \) for a node \( e_1 \) \( e_1 \) that maps to a node \( e_2 \) \( E_2 \), and
2. \( v(a_1) = v(a_2) \) for an arc \( a_1 \in DA_1 (a_2 \in CA_2) \) that maps to an arc \( a_2 \in DA_2 (a_2 \in CA_2) \), then \( SA_1 \) and \( SA_2 \) are equivalent. Note that it is not necessary for \( t(e_1) = t(e_2) \) for a node \( e_1 \) \( E_1 \) that maps to a node \( e_2 \) \( E_2 \).

**Proof:**

As \( EG_1 \) and \( EG_2 \) are isomorphic they may be merged to yield an event graph \( EG \). A node \( e_1 \in E_1 \) that maps to a node \( e_2 \in E_2 \) such that \( t(e_1) \neq t(e_2) \) is represented by a node \( e \in EG \) with a pair of event-type attributes \( t_1(e) = t(e_1) \) and \( t_2(e) = t(e_2) \). If \( EG \) contains such nodes, then it is not a valid representation of signal activity; each node in an event graph must have only a single attribute \( t(e) \) that specifies event type. If, however, \( EG \) can be shown equivalent, through transformation, to an event graph \( EG' \) where all nodes have a single event-type attribute \( t(e) \) then, as \( EG \) is a valid representation for a signal activity \( SA' \), we have shown that signal activities \( SA_1 \) and \( SA_2 \) are equivalent to the signal activity \( SA' \), and therefore, by associativity, \( SA_1 \) and \( SA_2 \) are equivalent. The event graph \( EG \) for signal activities associated with states I*R and MR is shown in Figure 3.1a.

The set of nodes in \( EG \) that have a single attribute \( t(e) \) is denoted \( N \). \( N \) in Figure 3.1a contains the nodes labeled 1, 3, 4, 5, 6, 7, and 8. Now, consider the nodes that have a pair of attributes \( t_1(e) \) and \( t_2(e) \). These nodes are placed in one of two sets K or \( K' \) as follows. Set \( K \) is defined to contain those nodes with a pair of attributes \( t_1(e) \) and \( t_2(e) \) such that: \( \forall k \in K \) either there is no delay arc \((k, j) \in DA, \) or for all delay arcs \((k, j) \in DA, j \in N \). The set \( K \) for the graph \( EG \) in Figure 3.1a contains the node labeled 2. Set \( K' \) is defined to contain those nodes with a pair of attributes \( t_1(e) \) and \( t_2(e) \) such that: \( \forall k' \in K' \) there are one or more delay arcs \((k', j) \in DA, j \notin N \). The set \( K \) for the graph \( EG \) in Figure 3.1a
\( \emptyset \). Note that the sets \( N, K \) and \( \overline{K} \) are disjoint, and also that \( E = N \cup K \cup \overline{K} \).

\[
\begin{align*}
N & = \{1, 3, 4, 5, 6, 7, 8\}, K = \{2\}, \overline{K} = \emptyset \\
N & = \{1, 2, 3, 4, 5, 6, 7, 8\}, K = \emptyset, \overline{K} = \emptyset
\end{align*}
\]

Figure 3.1: Illustration of criteria correctness proof.

A node \( k \in K \) has either no delay arcs emerging from it, or all such arcs are incident on nodes that have a single attribute \( t(e) \). Consequently, regardless of \( k \)'s event type the same signal activity is generated by it. For example, in Figure 3.1a regardless of node 2's event type, there is no subsequent signal activity generated by it. As a result, the attributes \( t_f(k) \) and \( t_s(k) \) of any event \( k \in K \) may be coalesced into a single attribute \( t(k) \), and \( k \) removed from the set \( K \) and placed in the set \( N \). This, for example, results in coalescing event types '11011' and '10111' for node 2 in Figure 3.1a into the event-type S and the graph shown in Figure 3.1b results. It is possible that a node \( k \in K \) that is moved to \( N \) has delay arc incident on it from a node \( k' \in \overline{K} \). If node \( k' \) now satisfies conditions required for it to be in \( K \) then it is moved from the set \( \overline{K} \) to \( K \).

In this manner, all nodes from \( K \) are moved to \( N \) and, as a result, nodes from \( \overline{K} \) moved to \( K \), requiring the process to repeat itself until finally \( K \) and \( \overline{K} \) are empty and all nodes in \( EG' \) are in \( N \). The process has to halt as at each step either the set \( K \) or \( \overline{K} \) become smaller. The reason the process halts only when \( K = \overline{K} = \emptyset \) and \( N = E \) is the following. If \( K \) is not empty, then nodes in \( K \) remain to be moved to \( N \). Therefore, \( K \) has to be empty for the process to stop. If \( \overline{K} \) is not empty, then there is a node \( k' \in \overline{K} \) that has a chain of delay arcs emerging from it that ultimately lead to a leaf-node \( k \in K \) that has no arcs emerging from it (note the graph is acyclic). Consequently, if \( K \) is not empty nor is \( K \) and for \( K = \emptyset \) to be satisfied, \( \overline{K} = \emptyset \) also has to be satisfied. Therefore, \( K = \overline{K} = \emptyset \) and \( N = E \) when the process halts. As all nodes in \( N \) have a single attribute \( t(e) \) for event type the resulting event graph \( EG' \) is a valid representation of some signal activity \( SA' \). Therefore, we have shown that two signal activities \( SA_1 \) and \( SA_2 \) that satisfy conditions (1) and (2) may always have their event-graph representations reduced to that for a single unique signal activity \( SA' \).

4. Minimal Interface State Machines

The signal-activity-equivalence criterion is used to identify timing-equivalent states in a state machine prior to performing timing verification. This improves the efficiency of timing-verification algorithms, as discussed in Section 1.

Two or more states are timing equivalent, iff their signal activities and state-transition behavior are equivalent. The state-transition behavior of two or more states is equivalent if they have transitions to equivalent states under the same conditions. A minimal interface state-machine is obtained by coalescing states that are timing equivalent. The procedure used to minimize fully-specified state machines [6] is used to yield a minimal interface state machine, given that the signal-activity-equivalence criterion, rather than output signal values, is used to distinguish states.

Figure 4.1 illustrates how the 82c288 state machine with 19 states is reduced to a minimal interface state machine with nine states. The initial equivalence partition contains all states. Partition 1 places states with distinct signal activities into distinct sets (equivalence classes). As each state in an equivalence class in Partition 1 transitions to states that are contained in the same equivalence class (under the same state-transition conditions) Partitions 1 and 2 are the same, causing the minimization algorithm to halt. States IA, MR and IR are reduced to state RA (read access), states IW and MW are reduced to state WA (write access), and so on.

![Diagram of state machine with truth tables](image)

Figure 4.1: Minimal 82c288 interface state machine.

This state reduction indicates that, from a timing-verification standpoint, there are three distinct classes of status signal values (0, 1, 5), (2, 6), (3, 4, 7) that impact 82c288 signal activity. Separately analyzing 82c288 behavior for status signal values that belong to the same class is inefficient, and analyzing any less than the three distinct classes would result in an incomplete analysis of 82c288 interface behavior. Minimizing state machines and using the signal-activity-equivalence criterion during timing verification ensures minimal, yet exhaustive, coverage of circuit behavior.

In states RA and WA the value 'S' is placed on CMD#. The value on CMD#, as a function of STATUS, is maintained through truth tables shown in Figure 4.1 that are necessary for accurate timing analysis. If, for example, CMD# value in state WA influences the number of events or their occurrence times in a circuit, then the truth table is used to maintain consistency between CMD# and STATUS values assumed by a timing verifier.

5. Decomposition of Interface State Machines

A minimal interface state machine may be decomposed into multiple independent state machines (ISMs). Decomposition is performed so that a state machine is broken into n ISMs that implement distinct functionality over the same set of input and output signals.
output signals on a component interface. Decomposing a state machine into n ISMs may be viewed as partitioning the functionality of a complex component into n simpler components, exactly one of which is enabled at any given time.

Decomposing state machines into ISMs is possible because of the inherently modular and decoupled activities performed by component interfaces. These activities, such as reading or writing data between component interfaces, or resource arbitration, are commonly referred to as "bus cycles" [7] or "interface transactions" [8]. For example, the 82c288 state machine in Figure 4.1 may be decomposed into the three ISMs shown in Figure 5.1, corresponding to read, write and idle bus cycles.

ISM have one or more start and final states. The write-bus cycle ISM in Figure 5.1 has W_D1 and W as its start states and WW as its final state. Start (final) states are distinguished from other states as they have a single entry (exit) arc entering (leaving) the state that is not connected to any other state in an ISM. Entry (exit) arcs specify the start (final) states of an ISM, and the entry conditions (exit conditions) under which these states are reached (left). For example, the entry condition for W_D1 is MB, and the value for W is MB; the exit condition from WW is always satisfied. Once enabled, an ISM enters a start state and remains enabled until it is in a final state and an exit condition is enabled. At this time, signal values on a subset of the input signals on a component interface, denoted I_c, are used to determine the next enabled ISM, which, in turn, enters its start state. The set I_c for the 82c288 contains STATUS. The signal values on I_c, for which an ISM is enabled is called the ISM Initiation Condition. This condition for the write bus cycle is "STATUS = 2 or 6".

5.1 Criterion and Algorithm for Decomposition
The criterion for state-machine decomposition is the following:
A state machine may be partitioned into n ISMs if it may be reconstructed from these ISMs by joining the exit arc from the final state s_f of each ISM with the entry arc to each start state s_i for each ISM, given that the state-transition condition for this arc is the conjunction of the following: exit condition for state s_i; entry condition for state s_f; and initiation condition of the ISM for which s_i is a start state. For example, joining the exit arc from state I of ISM-2 (read bus cycle) with the entry arc to state W_D1 of ISM-3 (write bus cycle), along with the conjunction of state-transition conditions for these arcs with the initiation condition for ISM-3 yields the arc from I to W_D1, in Figure 4.1, under the condition "MB & STATUS = 2 or 6". The fact that the 82c288 state machine in Figure 4.1 is reconstructable from the ISMs in Figure 5.1 in a similar manner should be easily discernable.

We now describe the algorithm for state-machine decomposition, which is shown in Figure 5.2 using pseudo-C syntax. The algorithm operates on a state-transition graph (STG) representation of a state machine. An STG = (S, A) is a directed graph with a node s ∈ S representing a state and an arc a ∈ A between a pair of nodes s_left and s_right representing a state-transition arc. STGs for interface state machines are strongly connected graphs. Associated with each arc a ∈ A is an attribute stc(a) that specify the state-transition condition for a. Signals that are used in a state-transition condition are called control signals. The function cs(a) is used to define the set of control signals in stc(a) for an arc a ∈ A. For example, cs(I, I) is [STATUS]. We also define a function fo(s) that returns the set of nodes in the fanout of s ∈ S; fo(s) = {Vs ∈ S: (s, s′) ∈ A}. For example, fo(I) is {I, R, W_D1, W}.

max fanatic = maximum |fo(s)| over all states;
for (num_pos = max fanatic, num_achieve = 1;
num_pos > num_achieve, num_pos--
for (i = 0; (num_pos + 1) <= max fanatic; i++)
{ state_list = all states whose |fo(s)| = num_pos + i;
for (each element s in state_list)
{ for (an untimed combination of choosing num_pos from |fo(s)|) {
start_sates = choose num_pos states from |fo(s)|;
non_start_sates = fo(s) - start_states;
if ((|cs(; s_i, s_f|) ∩ cs(s_f, s_f)) = ∅)
exit loop; /* s_i ∈ start_states and s_f ∈ non_start_sates*/
mark_final_states();
for (each element s_i in start_states)
{ism_s_i = traverse(s, stc(s_i));
if (an ism contains all states) //
(a state is not contained in any ism))
exit loop;
else
update_num_achieve();
}
}
Figure 5.2: Decomposition algorithm
The algorithm commences by initializing variables num_pos and num_achieve that specify the number of ISMs in the best possible and achieved decompositions, respectively. A decomposition is better than another if it has more ISMs. While num_achieve is always initialized to one, num_pos is initialized to the maximum fanout among all states in the STG. This is because the final state of each ISM must have an arc to the start state of all ISMs, and therefore there can only be as many ISMs as fanout arcs from a final state. For the STG in Figure 4.1 num_pos is four.

Figure 5.1: ISMs for 82c288.
If the set I_c on a component interface contains an internal signal (primary-input signal), then its ISMs are called primary ISMs (PISMS), and if not the ISMs are called slave ISMs (SISMS). PISMS initiate signal activity on a circuit while SISMS respond to activity applied at their inputs by other interface state machines in a circuit.

The qualifier "minimal interface" is dropped in future references to state machines.
The algorithm identifies decompositions by determining if there is any combination of start and final state markings that would allow the best possible decomposition to be achieved. If not, num_poss is decremented and the search continues until num Achieve is greater than or equal to num_poss.

For the STG in Figure 4.1 the algorithm would first place in the set state_list all states that have a fanout of four. These states are S1 and w. Next, one of these states, s1, is chosen from state_list and marked a final state. Assume that state S1 is chosen first. Now, the fanout of S1 is analyzed to identify a marking of start states that has been seen before and is valid. As we are attempting to achieve the best possible decomposition first, all states in fo(x) = (I, R, W, D, W), are placed in the set start states. It is assumed that there are as many ISMs as start states, and we denote the ISM associated with a start state s1, ISM1. The initiation condition of ISM1 is assumed to be the state-transition condition from s1 to s2.

Next, the algorithm identifies states in the STG that have transitions to each state in start states, and under the same state-transition condition as s1. These states are marked final states. For the STG in Figure 4.1 the state w satisfies these requirements and is consequently placed in the set final states. At this point, all the start and final states of the decomposition have been marked. Note that it is possible for a state to receive both start and final state markings. This is so for state I and manifests a situation wherein an ISM has a single state, which is both a start and final state.

The algorithm now identifies the set of states for each ISM. This is performed by traversing the STG rooted at each start state since until a final state s1 is encountered. No arc is traversed from s1, as it represents an inter-ISM state-transition arc. All states reachable from s1, including s2, are identified as belonging to the set of states associated with ISM1. A state once visited is not revisited during STG traversal. Also, a check is made during STG traversal to determine if a non-final state has been visited by another ISM. If so, the two ISMs are merged, with the initiation condition for the new ISM being the disjunction of their initiation conditions. This, for example, takes place when traversing the STG rooted at w given that w_d2 has already been classified as belonging to the ISM1.D1. As a result, ISM1 and ISM1_d2 are merged to yield a new ISM with initiation condition "status = 2 or 6". The decomposition achieved at the completion of STG traversal is shown in Figure 5.1.

Once a decomposition has been achieved it is necessary to test its validity and optimality. A decomposition is invalid if an ISM contains all STG-states or if a state in the STG does not belong to any ISM. A decomposition is sub-optimal if its number of ISMs is less than that already achieved by a previous decomposition. The decomposition in Figure 5.1 is valid and is saved.

As a result of ISM merging, it may be the case that the number of ISMs in a decomposition is less than the expected number indicated by the fanout of state s1. For example, state I had a fanout of four, but the decomposition achieved in Figure 5.1 has only three ISMs. In this situation, and also when a decomposition is invalid, it is necessary to investigate another marking of start and final states. For the STG in Figure 4.1 the state w is now selected from state_list and a decomposition attempted with w being marked s1. This, however, yields the same start-state marking as that of the existing solution. As state_list is now empty, num_poss is decremented by one. Now, as num_Achieve = num_poss the algorithm comes to a halt and reports its results.

Figure 5.3 tabulates results on the application of the decomposition algorithm to different state machines. The columns under decomposition results specify the following: "time" - CPU execution time on a SPARC-10 workstation with 32MB of RAM, "ISMs" - number of ISMs in the optimal decomposition, "attempts" - number of attempted decompositions. Notice that the optimal decompositions in all cases were identified after no more than three attempted decompositions. The reason is that interface state machines are inherently decomposable: if a decomposition exists it is easily identified. Also, the criterion for state-machine decomposition is restrictive enough to facilitate the early rejection of invalid decompositions.

<table>
<thead>
<tr>
<th>Components</th>
<th>STG Information</th>
<th>Decomposition Results</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>States</td>
<td>Transitions</td>
</tr>
<tr>
<td>2B8 unminimized</td>
<td>19</td>
<td>53</td>
</tr>
<tr>
<td>2B8 minimized</td>
<td>9</td>
<td>25</td>
</tr>
<tr>
<td>MC88020</td>
<td>10</td>
<td>46</td>
</tr>
<tr>
<td>B056</td>
<td>10</td>
<td>30</td>
</tr>
<tr>
<td>CPU</td>
<td>7</td>
<td>16</td>
</tr>
</tbody>
</table>

6. Summary

This paper has presented theoretical results that reduce task-complexity for the timing verification of circuits composed of interacting state machines from O(n^2) to O(nw x c). To achieve this reduction in task-complexity we have addressed the manner in which functionality affects timing when performing timing verification. We have defined a notion of signal activity that captures both functional and temporal aspects of interface behavior. We have developed a criterion for signal-activity equivalence that is used to, both, minimize an interface state machine prior to verification, as well as to limit the signal activities that need to be analyzed during verification. We have proved the correctness of this criterion from a timing-verification standpoint. Based on the inherently modular nature of interactions among state machines, we have developed a criterion and algorithm for state-machine decomposition. We have outlined the important insights into how state-machine decomposition may be used to dramatically reduce timing-verification task complexity by eliminating the need to build and analyze composite representations of circuit behavior.

References


a The "CPU" performs both a two-wire handshake protocol and an 8086-like read/write protocol.