Compiling Verilog into Timed Finite State Machines

Szu-Tsung Cheng [*†] Robert K. Brayton [‡] Gary York [§] Katherine Yelick [*] Alexander Saldanha [§]

Abstract

The lack of formal semantics for HDLs has made it hard to make a formal bridge between simulation tools based on HDLs and synthesis/verification tools based on finite state machines. In this paper we address the problem of finding a larger subset of Verilog HDL (which includes timing constructs) and a systematic way of extracting FSMs from programs built using the subset. Using timed FSMs as the target language for HDL compilation gives us two potential advantages. First, FSMs can be used to model systems that do not have hardware implementation. Second, FSMs can be used to model systems that are implementable but not automatically synthesizable.

1 Introduction

Simulation at various levels (physical, RTL, behavioral, etc) has made it possible for many designers to try out various possibilities and prototypes without fabricating designs. It has also aided designers to find various bugs before the circuit is manufactured. The fast turnaround time for simulations has sped up the design process. The introduction of HDLs for simulation has made it simple to design systems in terms of their behavior as well as their implementation. This makes it easy to write and test a partial design since one need not write the implementation of the whole design before it can be simulated. In addition, using behavioral descriptions makes it easy to write abstractions of designs/environments. Abstraction, which can be used to reduce system complexity, is an important aspect of hierarchical synthesis/verification of large systems. Using HDLs, designers can design pretty much the way they write software programs. Recently, many existing designs have been written in HDLs like Verilog[TM91], VHDL[vhd88], etc.

However, most of these HDLs are designed for simulation and their semantics are either defined in terms of simulation results or left undefined. One problem with this kind of approach is that since the simulator results determine the semantics of a Verilog program, different implementations of a simulator can give different results. Even worse, the same simulator can give different results for the same program; just by swapping two "concurrent" statements.

The lack of a formal semantics makes it hard to apply advanced synthesis/verification systems to existing designs written in these HDLs since it is hard to guarantee that the synthesized/verified circuit has, in a certain sense, the same behavior exhibited by a simulator. Many advanced automatic synthesis/verification systems, for example [ABB+ 94], make use of formal transition models such as automata as their underlying model of the real world. To make it possible to utilize state-of-the-art synthesis/verification algorithms and systems we need to bridge the semantics gap (or define it if it does not exist) between HDLs and the formal models (such as Finite State Machines) used by various synthesis/verification tools.

In this paper, we distinguish between the two terms, *synthesizable* and *implementable*. *Synthesizable* means deciding if a program written in a certain HDL can be compiled into another lower level language, which is simple and close to hardware implementation. *Implementable* means determining, generally automatically, if a design has a corresponding hardware implementation. Traditionally, synthesizing a HDL program meant compiling it into a circuit, i.e. synthesizable implied implementable. Using timed FSMs as the target language for HDL compilation has the following advantages. First, timed FSMs can be used to model systems that do not have implementations (for example, an abstracted module that generates 'even' or 'odd' nondeterministically). Second, timed FSMs can be used to model systems that do have hardware implementation but current synthesizers cannot identify appropriate implementations for them yet (for example, a signal waveform generator). Thus, by separating the problem of *synthesizable* and the problem of *implementable*, we have a technology independent, formal representation for Verilog. Potentially, a larger subset of Verilog can be translated into the intermediate format since the translation process is free from the problem of implementability. Thus, algorithms (synthesizers, verifiers, or even simulators) based on the intermediate format (FSMs) can take advantage of the larger subset.

In this paper we address the following problems:

- defining a subset of Verilog that is "synthesizable",
- defining the formal semantics of the subset of Verilog in terms of timed finite state machines,
- defining a modeling style (*timing machines* and *untimed machines*) used to "emulate" Verilog programs,
- providing a systematic way to translate programs in the above subset of Verilog into finite state machines.

The translated finite state machines are represented either using BLIF-MVT [BBC+] (a timed extension of BLIF-MV [BCH+ 91], which is a multi-valued extension of BLIF) or SMV+ [McM94]. However, since the latter does not support notation for timing, at present SMV+ can only represent Verilog without timing constructs. Algorithms presented in this paper has been incorporated into a compiler called vl2mv. vl2mv takes a program in the subset of Verilog described in this paper and compiles it into FSMs.

Note that since we use FSMs as the target language, which is essentially synchronously "clocked", we need to ensure that the generated FSMs (modeling logical hardware) can directly represent implementation hardware. For example, in a multiple-clock circuit, a physical edge triggered latch is modeled in FSMs using a symbolic latch (state variable), an edge detector (consisting of a hidden state variable and a table for finding the change of a signal), and a mux (used to latch data input at the appropriate time). However, in general, the generated FSMs have good hardware interpretations. So we use "FSMs" and "hardware" interchangeablely when it does not cause confusion.

The main contribution of the paper is: defining a subset of Verilog containing timing constructs that can be compiled systematically into Finite State Machines with timing constraints. By using timed finite state machines to model behaviors of Verilog programs, we separate the problem of determining whether a program is "implementable" from the problem of deciding if a program is "synthesizable". Thus, with timed FSMs, we can model programs that are not "synthesizable" for most of the advanced hardware compilers for HDLs (e.g., a signal waveform generator). Almost arbitrary programs, with a few exceptions (refer to section 5), built from the constructs defined in our synthesizable subset of Verilog can be

[*] Computer Science Division, EECS Department, University of California at Berkeley, California

[†] This work is supported by Cadence Design System, Siemens, CA Micro, and Fujitsu

[‡] EECS Department,University of California at Berkeley, California

[§] Cadence Berkeley Laboratory, California

modeled by the extracted timed FSMs. It is possible that, with the advant of more advanced synthesis algorithms, such timed FSMs can be synthesized and an optimized implementation can be found. In addition, formal verifiers for real-time systems [BBC$^+$] can be applied to extracted timed FSMs so that systems with timing constraints (not necessarily implementable) can be formally verified.

Note that the compilation process presented in this paper differs from high-level synthesis [GDWL92]. First, allocation of hardware resources for variables and operators in Verilog is based on the assumption of unlimited resources. The resource pool consists of all possible gates expressible in one table/equation in the target language. Second, no scheduling, partitioning is performed and no optimization is applied on the Verilog source. Thus, extracted FSMs are not guaranteed to be "optimal" in any sense. The goal of our compilation scheme is to extract timed FSMs to "emulate" Verilog programs.

The organization of this paper is as follows. In Section 2, basic terminology is introduced. A modeling style of Verilog programs is given in Section 3. Section 4 presents the algorithms used to compile Verilog programs into timed finite state machines. Some limitations of our modeling style are presented in Section 5. Section 6 concludes the paper. Appendix A deals with miscellaneous Verilog constructs. Several techniques used to minimize compiled FSMs are given in Apppendix B.

2 Background and Terminology

2.1 Timed Finite State Machines

The timed finite state machines used as the target language of the compilation process are basically Timed Automata with Linear Equations (TALE, [BBC$^+$]). These finite state machines are traditional FSMs plus some timer variables. Each transition is labelled with input/output alphabets as well as linear inequalities among timer variables and/or actions (generally resetting) on timer variables.

In the target language we use (BLIF-MVT or SMV+), each transition is further decomposed into smaller tables/equations. When multiplied together, these tables/equations define the input, output, present-state, next-state (of state variables) relations. BLIF-MVT allows the labeling of current-state/next-state transitions with linear timing constraints which further restrict the set of possible behaviors that can be exhibited by a system.

2.2 Terminology

Verilog is intended to be used to model circuits which are in general composed of several components running in parallel. We define a Verilog *process* to be the finest unit of program that can run concurrently with the other processes (or the largest program unit such that no two sub-units of it can be executed in parallel). Within a process, statements get executed sequentially. Each individual continuous assignment, `always` statement, `initial` statement, gate instantiation, or primitive instantiation, is basically a process. In our subset, the most basic form of a statement (*simple statement*) is an assignment statement. Simple statements are assumed to be executed instantaneously without being delayed. For example, `o <= a + b;` is a simple statement but `o <= #3 a + b;` is not. *Compound statements* are built by combining other statements, simple or compound, using conditionals (`if/else`), n-way branches (`case`), loops (`for`, `while`, `repeat`), or blocks (`begin/end`).

In addition, a compound statement may express timing information by *pauses* (*event guards* or *delays*). An *event* is defined by the rising or falling edge of a named signal/variable, say `x`. In Verilog, *event guards* that wait on such events are specified using one of the following: `@(posedge x)`, `@(negedge x)`, or `@(x)`. These denote a wait on a rising edge, falling edge, or any change for variable `x`. A *delay* is specified by the delay operator `#`, which indicates how long the program execution should be halted when a control flow encounters the operator. A conditional statement containing at least one pause is called a *significant* conditional statement. Otherwise it is called *insignificant*.

To translate Verilog, we need to identify blocks of code that are free of branches and events and build a graphical representation of a program. A *basic block* is a linear sequence of simple statements free of pauses and branches except at the very end. A *basic path* is a sequence of basic blocks that has no pauses except at the very end. A *maximal* basic path is a basic path that is not contained in other basic paths. Note that a basic path can wrap around a loop and two maximal basic paths can overlap.

2.3 Graphical Representation of Verilog Programs

To represent the control flow of the execution of a Verilog program, we use *control flow graphs* (CFG), which are multi-graphs $G = (V_p + V_c, E)$ where:

- V_p is the set of all distinct pauses.
- V_c is the set of all conditional statements in the program. Each node in V_c is called a *conditional node*. More specifically, for each conditional statement, a node is allocated to represent the beginning of the conditional statement.
- E is the set of edges, $e = (v_1, v_2) \in E$ iff $v_1, v_2 \in V_p + V_c$, v_1 occurs before v_2 for at least one execution of the program, and the statements between v_1 and v_2 form a basic path without passing through any conditional node. An arc originating from a conditional node is called a *conditional arc/edge*. Conditional edges are labelled with a logical formula which evaluates to true iff the corresponding branch is taken. The formula is denoted by $L(e)$ where e is a conditional arc.

3 Program = Timing Machines + Untimed Machines

Conceptually, a Verilog process is logically modeled by the product of two sets of machines; *timing machines* and *untimed machines*.

Timing machines determine how long a program (or the resulting product machine) can stay in a certain state. They control not only the timing of updating register and wire variables but also the sequencing of statements. Timing machines use *program context* information (values of logical expressions in conditional statements) from untimed machines along with values of *timers* in timed FSMs to determine their next states.

Untimed machines use program context and transitions of timing machines to determine whether "hardware" registers/latches self-loop or go to the next states. Untimed machines not only compute the next states of program variables, but also resolve contention among variable updates. In addition, untimed machines select appropriate definitions of variables for variable reference. Within a process, untimed machines control intra-pause sequencing while timing machines control inter-pause sequencing.

3.1 System Modeling

A run of a Verilog process consists of a sequence of two phases, *computation phase* and *idling phase*, as shown in Figure 1. During the computation phase, a process transits from the currently active pause to the next one that is going to be encountered during the execution of a program, all the statements between the two pauses are executed, and the next state variables are updated accordingly. In the idling phase, a process self loops in the current pause without changing the content of any program variables. Idling phases are used to provide time for the other processes to compute.

3.2 Halting "Hardware Time" When There are Still Unprocessed Events in a Certain Time Slot

In the compilation process, "hardware" (tables/equations in the target language in which FSMs are described) are allocated according to operators appearing in the program and interconnected according to their grammatical relationship. If the same piece of code gets executed twice in the same simulation time, then more than one computation phase is needed in order to reuse the same piece of

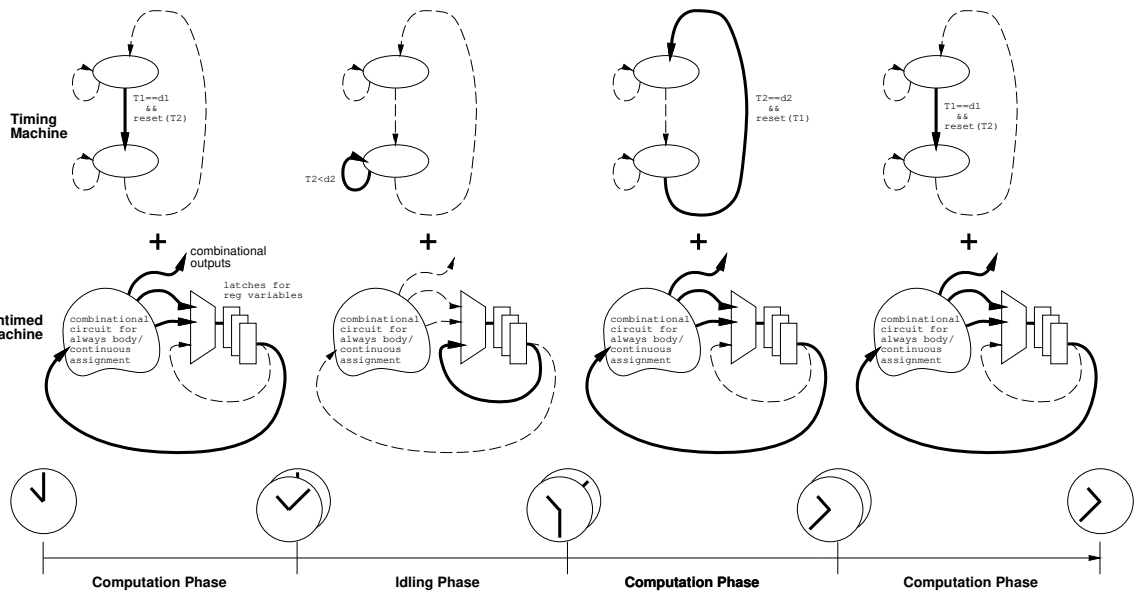

Figure 1: System Modeling

hardware. In the mean time, the other processes need to self-loop in the same state and timing machines should stop the progression of time. Another occasion where timing machines need to be halted is that there may be a series of events result from "domino" effects among them. That is, one process generates an event that is sensitive to the second process which then generates another event that activates the third process, and so on. All of these events happen in the same simulation time so while one process is processing an event, the other processes need to self-loop and halt their timers.

4 Algorithms For Extracting Control Flow Graphs, and Building Timing and Untimed Machines

4.1 Control Flow Graph Extraction

A control flow graph (CFG) can be extracted in a single pass traversal over a Verilog program. We have the following correspondence between Verilog constructs and CFG components.

- `always`, `for`, `while` loop statements introduce backward arcs (so that cycles may be formed in the generated CFG).
- `if/else`, `case` `for`, `while` introduce conditional nodes and conditional arcs.
- `@(posedge x)`, `@(negedge x)`, `@(x)`, `#(min,max)`, `#(delay)` introduce pauses.

4.2 Untimed Machines

4.2.1 Simple Statements - Assignments

There are basically four kinds of assignments in Verilog, namely *blocking procedural assignments*, *non-blocking procedural assignments*, *continuous assignments*, and *quasi-continuous assignments*. Among them, each continuous assignment constitutes one process. The other three kinds of assignments are used in *procedural statements* (statements beginning with `always` or `initial`). Each `initial` statement is executed only once when a simulator starts. Each `always` statement is executed repeatedly forever.

A continuous assignment updates its left-side variable whenever any of its operands changes. A blocking procedural assignment, when executed, reads its operands and updates the left-side variable(s) immediately. On the other hand, a non-blocking procedural assignment, when executed, evaluates the right-side and "remember" the result without touching the left-side variable(s). Then whenever a simulator advances time, left-side variables get updated using the remembered values. Quasi-continuous assignments are a procedural version of continuous assignments. They override all the other procedural assignments, blocking or non-blocking. Quasi-continuous assignments can be disabled by `deassign`.

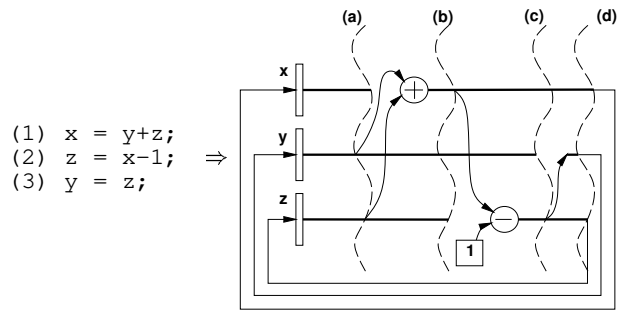

Figure 2: Circuit generated from sequence of statements. (a) Before any statement is encountered. (b) After `x=y+z;` is encountered. (c) After `z=x-1;` is encountered. (d) After `y=z;` is encountered.

For each assignment, a new hardware (FSM) signal is allocated to represent the new value of the left-side variable(s). A symbol table is used to store the association between program variables and allocated signals. There are three separate symbol tables to store associations resulted from blocking procedural assignments + continuous assignments, non-blocking procedural assignments, and quasi-continuous assignments, respectively.

4.2.2 Program Sequencing

Variable references (read references) always refer to the symbol table for blocking assignments + continuous assignments. The next state variable associated with a register variable comes from the last assignment to that variable. Refer to Figure 2 for an example.

However, in the presence of pauses, sequences of statements in-

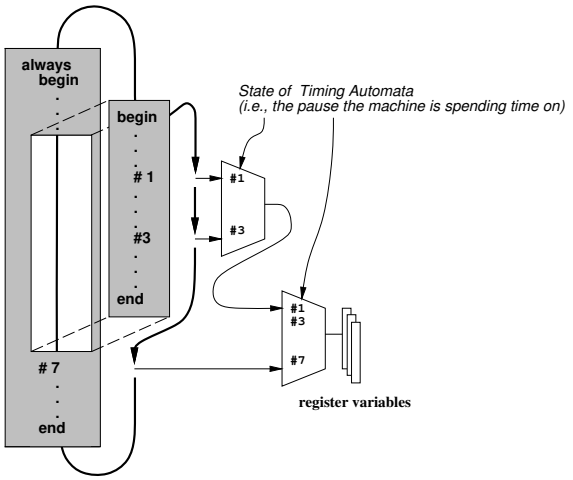

Figure 3: Example hierarchy of segment selector.

side the same block might not be executed in the same hardware time. Thus, another level of logic (*segment selector*) is used for next state variable selection. We refer to the segment of code executed at a particular point of time as *active* at that time. Next-state values of timing machines are used to determine which segment of code is active and should affect the next states of `reg` variables. Due to the hierarchical structures of statements (a block of statements may consist of sub-blocks), segment selectors also have similar hierarchical structures (Figure 3). The following algorithm recursively builds segment selectors for simple/composite statements.

segment-selector (composite-statement)
for each sub-statement s in composite-statement **do**
  Let p_{ns} be next state of the timing machine.
  if (s is simple[1])
    continue;
  fi
  if (s is delayed simple[1])
    let d be the pause controlling s
    let p be the set of value signals available
      for all `reg` variables immediately before d
    add one branch in segment-selector which says
    If there is an inter-pause transition
      due to time-out
      and if $p_{ns} == d$
    then value of p *is taken.*
    continue;
  fi
  if (s is composite[1])
    p_l = segment-selector(s)
    d_l = set of pauses in s
    add one branch in segment-selector which says
    If there is an inter-pause transition
      due to time-out
      and if $p_{ns} \in d_l$
    then value of p_l *is taken.*
    continue;
  fi
od

4.2.3 Variable Selection for Conditionals

For conditional statements (`if/else`, `for`, `while`), logic which generates the same truth value as that of conditional expressions is

[1]A delayed simple statement is a simple statement controlled by a pause. A composite statement is a `begin/end`, `for`, `while`, `if/else`, or `case` statement.

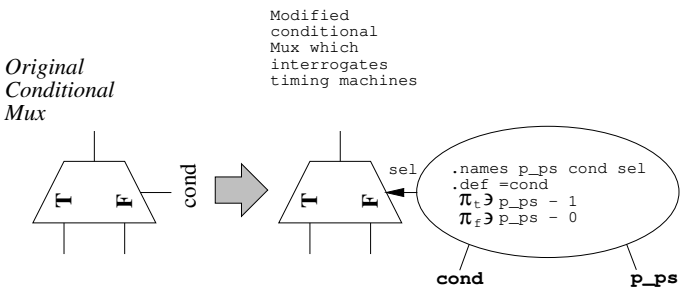

Figure 4: p_{ps} is the present state of the timing machine. $\pi_t(\pi_f)$ denotes the set of pauses in the *true* (*false*) branch of a conditional statement which the mux stands for.

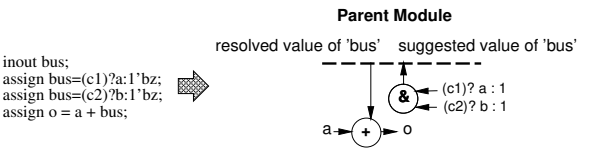

Figure 5: Translation of a tri-state/`inout` variable.

produced. The outputs of this logic are used to control muxes which select the appropriate definitions of variables. Statements following the conditional statement look at outputs of these muxes for read references. In the presence of pauses inside branches of conditional statements, outputs of the conditional logic are pre-empted by the present states of timing machines (Figure 4). The reason for this pre-emption is that the outputs of the conditional logic might be different when execution enters and when it leaves a conditional statement.

4.2.4 Tri-State Variable Resolution and `inout` Ports

In the compilation process, binary functions are used to model operators in Verilog programs (this makes it easier to synthesize the generated FSMs). In order to model tri-state buses and bidirectional ports, extra logic is introduced.

A resolution function is allocated for each "tri-state variable" (a variable whose value can be high impedance `Z` for some time) in each module. The resolution function is basically a wired-and (refer to Figure 5). It collects all definitions to the tri-state variable, say `x`, replaces high impedance `Z` with 1, and takes the logical conjunction of all the definitions. It should be noted that this scheme does not check if there is an assignment conflict (one statement tries to assign 1 to `x` while another one tries to assign 0 to it). One should be sure that there is no such conflict before applying the compiler.

Each bidirectional variable (which is declared by an `inout` declaration), say `x`, is split into two finite state machine variables. Local definitions (write reference) of `x` are resolved and sent to its parent so that its parent can use the submitted value as one of the definitions to the variable. The determined value of the variable is passed down from a parent to children and all read references to the variable are redirected to the value given by its parent (Figure 5).

4.2.5 Intra-Process Arbitration

The next state of a variable, say `x`, comes from the last procedural blocking assignment that touches `x`. In the presence of non-blocking assignments, the next state of `x` will be chosen nondeterministically from active non-blocking assignments, as suggested by [BY93]. This is used to model nondeterminism on state variables. If there is an active quasi-continuous assignment (the quasi-continuous assignment is executed some time in the past and has not been disabled

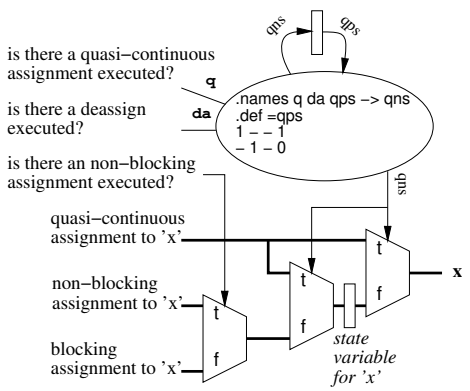

Figure 6: Arbitration among assignments.

by `deassign`), then the next state as well as any read references to `x` are redirected to the quasi-continuous assignment. The arbitration among blocking procedural assignments, non-blocking procedural assignments, and quasi-continuous assignments (in increasing priority order) is translated into a circuit as shown in Figure 6.

4.2.6 Inter-Process Arbitration

When there is more than one process trying to change the content of variable `a` and a conflict exists between the values being assigned (e.g. 1 and 0), a Verilog simulator outputs a unknown `X`. Since we only use binary functions, a nondeterministic 0 or 1 is used to model `X`. A resolution function which collects all assignments from different processes is used to arbitrate among different processes contending to update the same variable.

4.3 Event Guards, Delays

Each pause (event guard or delay) is basically a place in a Verilog program where execution flow must halt for a specified period or wait for the occurrence of a designated event. We allocate a unique state in the timing machine to indicate the state of waiting. The enabling condition to make a timing machine leave a waiting state is that either a specified time has elapsed since the last time a timer was reset or a specified event has occurred. To detect the occurrence of an event, we use *event detectors*, as shown in Figure 7.

4.4 Timing Machines

4.4.1 `always` Statements

Each `always` statement is executed repeatedly forever. The CFG for it is a cycle indicating that the statement is going to be executed again and again. The timing machine for an `always` statement has a similar structure to its CFG. The following algorithm extracts timing machines from CFGs. It enumerates all the pause-free paths (except the source and destination) of a CFG and generates inter-pause transitions for its corresponding timing machines.

for each $p_s, p_d \in V_p$ **do**
  for each simple path
    $p : p_s \rightsquigarrow p_d, p \cap (V_p - \{p_s, p_d\}) = \phi$ **do**
    /* i.e. no pauses in between */
    Let $C = \{l \mid l = L((c,v)), c \in V_c, c \in p\}$
    if p_s's corresponding delay is of the form $\#\delta$,
      then put a transition $s_s \rightarrow s_d$ labelled with
        C,$T_s == \delta(p_s)$[1],and $T_d = 0$
        and a self-loop $s_s \rightarrow s_d$ labelled with
        $T_s < \delta(p_s)$[2]
    if p_s's corresponding delay
        is of the form $\#(\delta_{min} : \delta_{max})$,
      then put a transition $s_s \rightarrow s_d$ labelled with
        $C, \delta_{min}(p_s) \leq T_s \leq \delta_{max}(p_s)$[1], and $T_d = 0$
        and a self-loop $s_s \rightarrow s_d$ labelled with
        $T_s < \delta_{max}(p_s)$[2]
    if p_s's corresponds to an edge event control
        (`@(posedge x)`/`@(negedge x)`/`@(x)`),
      then put a transition $s_s \rightarrow s_d$ activated
        by the corresponding edge detector
        and a self-loop $s_s \rightarrow s_d$
        when the edge detector gives false
  od
od

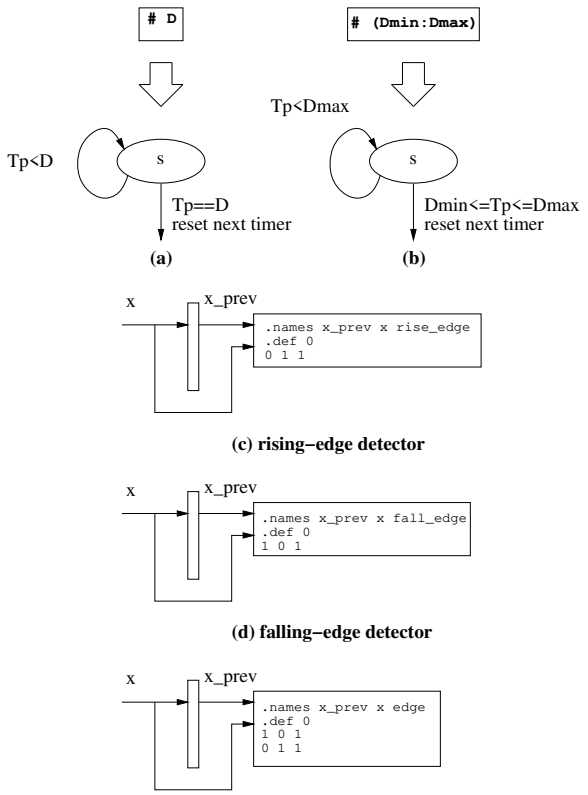

Figure 7: (a), (b) Pauses for timing constraints. (c), (d) Rising/Falling edge detector for `@(posedge x)`, `@(negedge x)`, respectively. (e) Bidirectional edge detector for `@(x)`.

The preceding algorithm, which enumerates all the pause-free paths, can spend time exponential in the number of non-overlapping insignificant conditional blocks between the source and destination pause. This could be relieved by a preprocess over the CFG to eliminate all insignificant conditional blocks, which can be done in $O(|V|)$ [CB94].

Consider the following `always` statement, `always begin stmt1; #3 stmt2; end`. A Verilog simulator first executes `stmt1`, then waits for 3 time units and executes `stmt2` and `stmt1`, and so on. In the translated FSMs, for the first time the process is executed, the logic that is involved (used to evaluate `stmt1;`) is different from the logic used afterward (to evaluate `stmt2; stmt1;`). To emulate the first-time execution of a process, an extra node is introduced in the CFG. Additional arcs originating from the newly introduced node are inserted whenever it is possible for a first time execution to end in the destination node (pause) without passing

[1]This kind of timing constraint is called a *Time-out-constraints.*

[2]This kind of timing constraint is called an *Idling-constraints.*

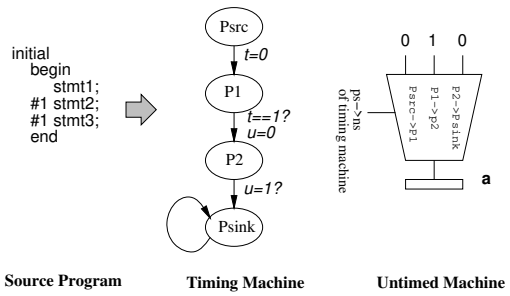

Figure 8: Example `initial`-statement an its corresponding timed FSMs.

through the other pause. A set of muxes is also introduced to select the appropriate definitions of variables. For example, in the preceding example, for the first time execution, variable references from within `stmt1` refer to values given by the `initial` statement, if there are any. For non-first time execution, `stmt1` asks `stmt2` for variable definitions. The following algorithm is used to extract initial transitions from a CFG for initial execution of `always` statements. p_0 is the node allocated to denote the first-time execution of a process.

for each $p_d \in V_p$ **do**
  for each simple path
    $p : p_0 \rightsquigarrow p_d, p \cap (V_p - \{p_d\}) = \phi$ **do**
    Let $C = \{l \mid l = L((c, v)), c \in V_c, c \in p\}$
      put a transition $s_0 \rightarrow s_d$ labelled with
        $C, T_d = 0$.
  od
od

4.4.2 `initial` Statements

If an `initial` statement can be executed instantaneously, it is translated into initialization of state variables in the target FSMs. For an `initial` statement that can not be executed within zero hardware time, its CFG is built and an additional *sink* state is introduced to denote that the `initial` statement has been executed completely and will stay inactive afterward. As an example, refer to Figure 8.

4.4.3 `for`/`while`/`repeat` Loops

Given that the iteration count argument of `repeat` is compile-time evaluable, a `repeat` loop body is simply unrolled. So far our compiler can not handle a `repeat` loop with a variable iteration count.

Each `for`/`while` loop statement is basically a conditional statement except that execution flow, on finishing executing the loop body, always jumps back to the beginning of the loop statement. For `for` and `while` loops, outputs from conditional logic (which is translated from conditional expressions in the loop statements) are used by timing machines to determine appropriate next states. There are two possible ways to deal with `for` and `while` loops, namely *loop unrolling* and *legal loops*.

- `loop unrolling` - A loop can be unrolled if the arguments of the loop statement are compile-time evaluable so that the number of copies of the loop body can be determined at compile-time.
- `legal loops` - In case a loop is *legal*, i.e., there is no pause-free cycle in its corresponding CFG, a loop can be compiled into a set of muxes, which are used to select appropriate definitions of variables, as well as conditional logic, which is used to control the program flow. Figure 9 shows an example `for`-loop and its FSMs.

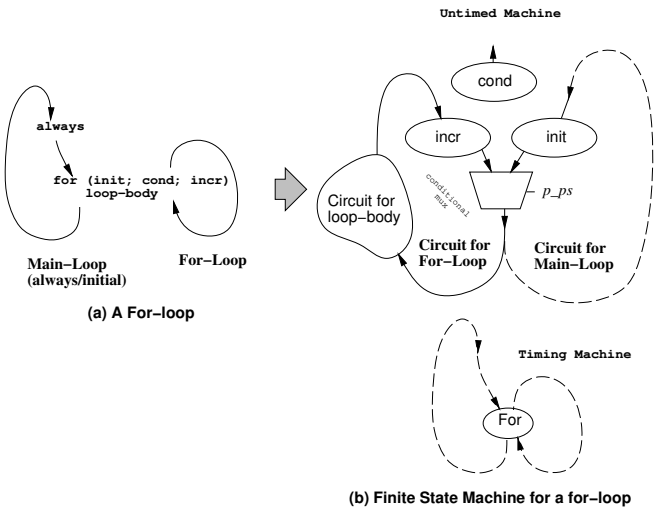

Figure 9: Circuit for `for`-loop. p_{ps} is the present state of the process' timing machine.

5 Limitations

The limitations of our compilation scheme result mainly from four factors.

First, FSMs are used as the target language. Since resources (intermediate variable, transition function/relation, state variable, timers, etc.) of a FSM are statically allocated and alphabets of generated FSMs are always finite, infinite data structures such as stacks, queues cannot be handled (and it is called non-"synthesizable"). [1]

The following are known cases where FSM resources need to be allocated dynamically.

- Statements like `o <= #5 a + b;` need to allocate timer variables dynamically when the relationship between the frequency that the statement is executed and the delay inside the assignment is unknown. It is an open question as to how to handle this kind of statement.
- `fork`/`join` - The most general way to handle this kind of construct in programming languages is to use *heap*. FSM resources like intermediate variables, transition relations, state variables need to be allocated dynamically if the maximum amount of heap space needed to handle `fork`/ `join` cannot be determined at compile-time.

Second, binary functions are used to model operators instead of three or more valued functions. Thus tri-state variables can only be modeled in a very restricted way, as explained in Section 4. In addition, strength reduction (if supply-1 encounters charged-0 on a wire, then the result is a strong-1) is not implemented.

Third, primitives that need the dynamic support of a simulator and/or an operating system can not be handled. For example, system tasks like `display`, `fopen`, `system`, `finish`, etc.

Fourth, there are a few extreme cases for which the compiled FSMs can not model behavior given by a simulator.

- In Verilog, a `task` is implemented by spawning a new process and control is transferred from the calling process. Control is returned when the created process is finished and execution continues from the statement immediate after the `task` statement. If there exists a pending update due to a non-blocking assignment inside a `task` statement when the control is given back to its parent, the pending update is "forgotten" when the `task` gets terminated. Since `task`

[1] A possible extension to this is that if the maximum amount of resources can be statically determined, then it is still synthesizable.

is modeled by a parameterized macro expansion, this kind of "amnesia" of non-blocking assignments can not be reproduced by the compiled FSMs.

- Some versions of a Verilog simulator use a non-stack mechanism to evaluate functions. This can lead to different results for nested functions. For example, suppose the definition of function `add` is:

```
function [3:0] add;
   input [3:0] a, b;
   add = a + b;
endfunction
```

  For a statement like `add(3, add(2, 1))`, compiled FSMs returns 6 but some simulators can give 5. The reason is that, due to the lack of a stack, on calling the outer `add`, 3 is copied into `a`. It is then found there is an unresolved function evaluation. Therefore, inner `add` is called and 2 and 1 are copied into `a` and `b`, respectively. When the inner `add` returns 3 to the argument `b` of the outer `add`, `a` has been changed to 2. Thus 5 is the final result given by such stack-less simulators. However, we will not worry about these incorrect versions of Verilog.

- If the behavior of a design depends on the order various processes are scheduled, then the behavior might not be reproducible by the compiled FSMs. For example, consider the following two "equivalent" program segments.

```
assign a1 = a-1;                    | assign a1 = a-1;
always @(posedge clk) a=b;          | always @(posedge clk) b=a1+2;
always @(posedge clk) b=a1+2;       | always @(posedge clk) a=b;
```

  Both segments are the same except for the order of the second and third statements. However, simulation results of the two programs are different. Assume that `a` and `b` are initially 0 (and `a1` is -1). When the rising edge of `clk` occurs in the left program, `a` and `b` are updated to 0 and 1, respectively. On the other hand, when the positive edge of `clk` occurs in the right program, both `a` and `b` change to 1. This is due to the different order the two procedural statements are executed. Similarly, racing can be observed in several examples. For the preceding two programs, the generated FSMs give the same trace. The reason is that all variable references ("read" references) are made to the current states of registers and all assignments ("write" references) are made to the next states. In this way, race conditions as in the previous example are avoided.

  So far, `disable`, which disable the execution of a named block, is not implemented in vl2mv. It can be done by making the untimed machine of the controlled process (the process which is to be disabled) consult the transition of the timing machine of the controlling process(es) (the process which has `disable` instruction to control the execution of the controlled process).

A complete description of the subset of synthesizable subset of Verilog for vl2mv can be found in [CB94].

6 Conclusions

Algorithms presented in this paper have been incorporated into a compiler called vl2mv. Over 90 benchmarks (80 are behavioral, 10 are structural) have been compiled by vl2mv and the extracted timed FSMs have been verified. Sizes of these benchmarks range from tens of lines to thousands of lines. vl2mv has also been successfully applied to system designs consisting of more then two hundred thousand lines of behavioral/structural code. vl2mv addresses the problem of compiling a large subset of Verilog HDL into FSMs. With it, engineers can design in HDL and still have state-of-the-art verification/synthesis/simulation algorithms to help verify and optimize their designs.

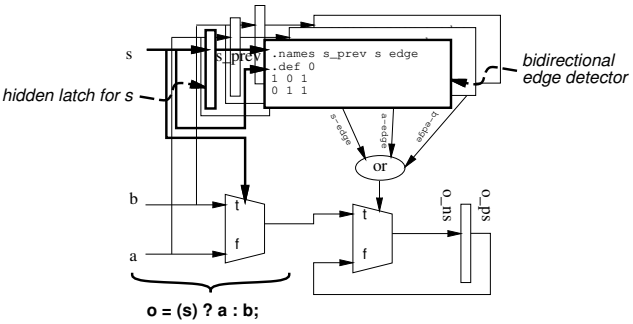

Figure 10: Finite state machine for `always @(s or a or b) o = (s)?a:b;`

Acknowledgements

We would like to thank Cadence, Siemens, CA Micro, and Fujitsu for support during the research. We want to acknowledge Felice Balarin for helpful advice and inspiring discussions. We also want to thank the many users who provided precious opinions, feedback and field tested vl2mv for the development of the compiler. We thank Huey-Yih Wang for his help during the preparation of this paper.

Appendix

A Miscellansous Verilog Constructs

A.1 Primitives

A Verilog `primitive` basically lists the relationship between input and output values. To translate a combinational `primitive`, one just transcribes the relation listed in the `primitive`. For a sequential `primitive`, edge detectors are used to detect specific transitions on a signal. Outputs of edge detectors are used to control the update of state variables in the `primitive`.

A.2 Tasks, Functions

A `function`, which is essentially a combinational block, is compiled into a separate module. Each function call is translated into a module instantiation. A `task` is compiled as a parameterized macro and expanded in-line wherever it is invoked. The arguments and parameters are substituted accordingly.

A.3 Parameters

A new master FSM is allocated for each `module/function` that is invoked with parameter values different than the default values. The call to the `module/function` is then translated into a module instantiation of the newly created master.

A.4 Nondeterminism

Nondeterminism plays an important role in abstraction. Verilog HDL does not have a well-defined way of describing nondeterminism. We use non-blocking assignments to model next state nondeterminism [BY93]and `NDset` to model combinational nondeterminism.

B Optimization for Special Cases

B.1 Combinational Reduction

`assign o = (s)?a:b;` and `always @(s or a or b) o = (s)?a:b;` are effectively the same (given the same scheduling). However, from the schemes given in previous sections, a circuit shown in Figure 10 is generated for the second statement. A lot of redundant latches and logic are generated. This redundant logic and state variables can introduce an unnecessary burden on synthesizers/verifiers which use the generated FSMs as input. We use the following sufficient conditions to optimize FSM generation.

1. A variable is used only in one procedural statement where the procedural statement has only one event guard.
2. An event guard is only sensitive to bidirectional change of the variables listed in the event guard.
3. All variables used as operands are listed in the sensitivity list.
4. The control flow graph for the process is legal.
5. All conditional statements are *complete*. i.e., all `if` statements are accompanied by an `else` branch. Branches in a `case` statement cover all possible values of the switching variable.

Note that the last condition is quite restrictive. A possible relief for this is to use a data-flow analysis [ASU85], [FRJL88] and make sure that each variable used on the left-hand-side is assigned for each possible pause-to-pause execution over the process.

Once the above sufficient conditions are met, behavior of assigned variables, say `x`, is "combinational". i.e., whenever an operand that can potentially affect `x` changes, `x` is re-evaluated. No latch is allocated for `x` and no edge detectors are allocated for the variables that `x` is sensitive to.

B.2 Explicit Clocking v.s. Implicit Clocking

Given that a system is synchronous, can be judged to have only one global clock, and every process updates its local variable at the same phase of the clock, it is sufficient to allocate one state variable in the generated FSM for each register variable in the Verilog program. We call this kind of design *implicitly clocked* [2] since the main purpose of the global clock is to make sure every process has a consistent idea about how time progresses. The global clock may or may not have a corresponding hardware wire. In translating an implicitly clocked design, no edge detectors are necessary; thus a lot of logic can be saved in the target FSM to model the source program. On the other hand, if a design is not implicitly clocked (it is called *explicitly clocked*), edge detectors and auxiliary logic are allocated in order to model clocking mechanism on different phases (rising, falling, or bi-directional) of various signals.

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

[ASU85] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1985.

[BBC+] 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. To be submitted to DAC'96.

[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.

[BY93] Felice Balarin and Gary York. Verilog HDL modeling styles for formal verification. In *CHDL*. North-Holland, 1993.

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

[FRJL88] Charles N. Fischer and Jr. Richard J. LeBlanc. *Crafting a Compiler*. The Benjamin/Cummings Publishing Company, Menlo Park, California, 1988.

[GDWL92] Daniel Gajski, Nikil Dutt, Allen Wu, and Steve Lin. *High-Level Synthesis*. Kluwer Academic Publishers, Nowell, Massachusetts, 1992.

[McM94] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 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*. Institute of Electrical and Electronics Engineers, 1988.

[2] Implicit/explicit clocking should be distinguished *implicit/explicit FSMs* used by Cadence for their hardware compiler for Verilog.