VIS User's Manual

Tiziano Villa Gitanjali Swamy Thomas Shiple

The VIS Group

Adnan Aziz[1]
Robert Brayton[1]
Stephen Edwards[1]
Gary Hachtel[2]
Sunil Khatri[1]
Yuji Kukimoto[1]
Abelardo Pardo[2]
Shaz Qadeer[1]
Rajeev Ranjan[1]
Alberto Sangiovanni-Vincentelli[1]
Shaker Sarwary[3]
Thomas Shiple[1]
Fabio Somenzi[2]
Gitanjali Swamy[1]
Tiziano Villa[1]

[1]University of California, Berkeley
[2]University of Colorado, Boulder
[3]Now at Lattice Semiconductor

Contents

Chapter 1

Introduction to VIS

This document introduces VIS (Verification Interacting with Synthesis). We describe what VIS is, what it can do, how to write limited Verilog code for its input, its commands, and an extended example for the new user. For more details, see the VIS home page `http://www-cad.eecs.berkeley.edu/Respep/Research/vis/doc/packages/index.html`.

1.1 What is VIS ?

VIS is a verification and synthesis system for finite-state hardware systems, which is being developed at Berkeley and Boulder. It improves upon first generation tools like HSIS and SMV by:

1. providing a better programming environment,
2. providing some new capabilities, and
3. improving performance in some cases.

VIS is divided into three parts: a common front end for reading in a description of a design, verification (VIS-v), and synthesis (VIS-s).

1.2 History

Many first generation tools for automatic formal verification were based on two theoretical approaches. The first is temporal logic model checking, where the properties to be checked are expressed as formulas in a temporal logic, and the system is expressed as a finite state system. In particular, Computational Tree Logic (CTL) model checking is a technique pioneered by Clarke and Emerson to verify whether a finite state system satisfies properties expressed as formulas in a branching-time temporal logic called CTL. SMV, a system developed at CMU, belongs to this class of tools.

Certain properties are not expressible in CTL, but they can be expressed as ω-automata. The second approach, language containment, requires the description of the system and properties as ω-automata, and verifies correctness by checking that the language of the system is contained in the language of the property. Note that certain types of CTL properties involving existential quantification are not expressible by ω-automata. COSPAN, a system developed at Bell Labs, offers language containment.

A combination of both approaches is offered by the HSIS [6] system, which was developed at the University of California, Berkeley. Our experience with verification tools (in particular HSIS) led to the conclusion that sometimes, the simpler and more limited the approach, the more efficient it can be. A number of design decisions that we made for HSIS made it unacceptably slow for some large examples.

With these problems in mind, we set about writing a tool that was more efficient, easily extendible, and offered a good programming environment, in order that it can be more easily upgraded in the future as more efficient algorithms are developed.

VIS also has the capability to interface with SIS to optimize logic modules; hence, VIS is an integrated system for hierarchical synthesis, as well as verification. We plan to pursue research on the interaction between verification and synthesis in the future; hence the name VIS, verification interacting with synthesis.

1.3 Overview of VIS

Fig. 1.1 presents of an overview of VIS. VIS has three main parts: a front-end to read and traverse a

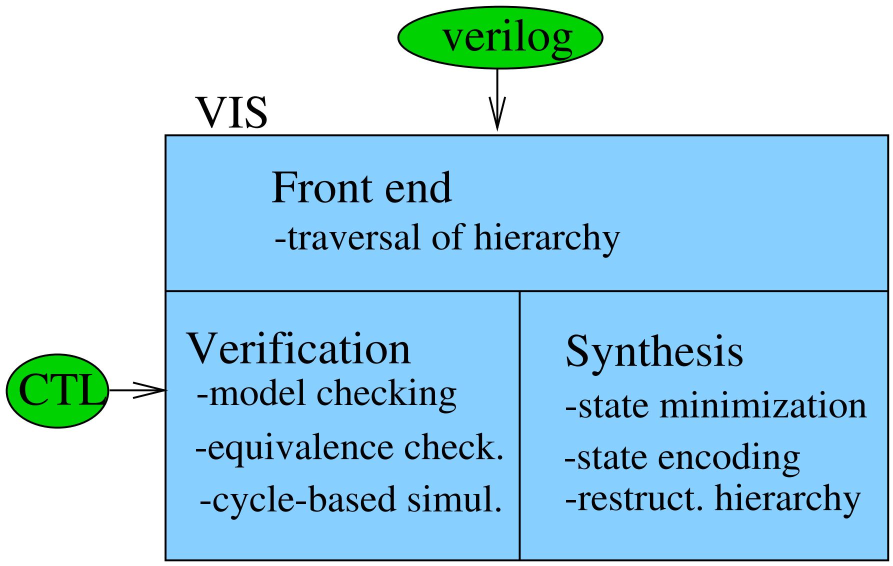

Figure 1.1: Block diagram of VIS.

hierarchical system described in BLIF-MV, which may have been compiled from a high-level language like Verilog; a verification core, VIS-v, to perform model checking of Fair CTL and test language emptiness; and a path to SIS, VIS-s, to optimize parts of the logic.

1.3.1 VIS-v Philosophy

We decided to offer limited but efficient capabilities. We felt that in the future, it would be easy to add more features, as they are required, using a well defined programming interface. In line with this **keep it simple** philosophy, VIS provides the following verification capabilities.

- Only CTL formulas can be checked. Language containment may be handled in a later release. However, we do handle language emptiness checks.
- Fairness constraints must be of Büchi type, i.e., sets of states that must be visited infinitely often. However, the internal VIS data structures do have the capability to support more complicated fairness constraints.

1.3.2 VIS-s Philosophy

VIS can interact with SIS to assist the task of verification by simplifying parts of the system. Another objective is to support a full-fledged hierarchical synthesis flow, that translates a Verilog description into an optimized multi-level circuit at the gate level. Unlike existing logic optimization systems like SIS, VIS can support hierarchical synthesis.

Chapter 2

Describing Designs for VIS

Given the special needs of hardware simulation, verification, and synthesis, specialized languages to describe hardware have been defined. These are called hardware description languages (HDLs) and they resemble general-purpose programming languages. Modern HDLs enable the designer to mix different levels of design abstraction.

2.1 Verilog HDL

The two most widely used languages for digital design are Verilog, based on C, and VHDL, based on ADA. Currently VIS only supports Verilog, but our intermediate format, BLIF-MV, was designed to support translation from many languages.

Verilog allows mixed-level descriptions of hardware in terms of static structures and dynamic behaviors. Dynamic behavior is described by means of high-level constructs as found in general-purpose programming languages, like conditional, control of loops, and process fork-join.

A specification in Verilog consists of one or more *modules*. The *top level module* specifies a closed system containing both test data and hardware models. Component modules normally have input and output *ports*. Events on the input ports cause changes on the outputs. Events can be either changes in the values of *wire* variables (i.e., combinational variables) or in the values of *reg* variables (i.e., register variables), or can be explicitly generated abstract events. Modules can represent pieces of hardware ranging from simple gates to complete systems (e.g., microprocessors), and they can be specified either *behaviorally* or *structurally*, or by a combination of the two. A behavioral specification defines the behavior of a module using programming language constructs. A structural specification expresses a module as a hierarchical interconnection of submodules. The components at the bottom of the hierarchy are either primitives or are specified behaviorally. Verilog has a library of predefined primitives. A good reference for Verilog can be found in [1].

2.2 VL2MV: from Verilog to BLIF-MV

VIS operates on an intermediate format called BLIF-MV, which is an extension of BLIF, the intermediate format for logic synthesis accepted by SIS and other tools. VIS includes a stand-alone compiler from Verilog to BLIF-MV, called VL2MV.

See [2] for a description of the synthesizable subset of Verilog that can be handled by VL2MV and of the extensions of Verilog that are also supported by VL2MV. In this section we survey the key features of Verilog for VL2MV. Conceptually, it would be easy to provide a translator from any other HDL language, like VHDL or Esterel, to BLIF-MV.

The relationship between a behavioral description language like Verilog and a machine description language like BLIF-MV is similar to that between a high-level programming language and an assembly language. Basic constructs of BLIF-MV are module declarations/instantiations, input-output relational tables which allow descriptions of nondeterminism, symbolic wires, and latches. In BLIF-MV, symbolic latches are implicitly controlled by a global clock. This *clock* does not need to be a real wire in the hardware sense. All symbolic latches transit instantaneously to the next state indicated by the relevant transition tables. At each clock cycle, each table continuously updates its outputs according to the inputs it sees until convergence is reached. [1] In the very beginning of the next cycle, all latches simultaneously update their present state outputs according to their next state inputs. Then again tables update their outputs accordingly.

VL2MV extracts a set of interacting finite state machines (FSMs) that preserve the behavior of the source Verilog program defined in terms of simulated results. Allocation of hardware gates to operators in Verilog (resource binding) is based on the assumption of unlimited resources, where resources are all possible gates expressible in one table in BLIF-MV. No scheduling and optimization are performed, so the extracted FSMs are not guaranteed to be optimal (for area, speed, and so on). In order to optimize the logic, a synthesis program like SIS can be invoked on modules of the system. [2]

A design in a synthesizable subset of Verilog consists of a set of modules (either hardware or software). The first module encountered is regarded as the root module. All modules run in parallel and communicate with each other through a set of channels (set of wire variables declared in the modules to which these channels belong). It is assumed that communication through channels is instantaneous. Within each module, values on channels can be accessed through a set of ports, that can be either wires or registers. Through wire ports, a module can input and output from and to channels instantaneously, while through register ports it takes one time unit. A wire port has no storage element associated with it, while a register port has one storage element associated with it.

A Verilog module contains declarations, module instantiations, continuous assignments and procedural blocks. Continuous assignments begin with the keyword `assign` and are always active; they can be thought of as combinational blocks. Procedural blocks are referred to as `always` statements; statements within a procedural block are executed sequentially.

Module instances, continuous assignments, and procedural blocks within a module run concurrently. Execution of each continuous assignment, basic block in a procedural block and module instance is assumed to be atomic within each instant. If there is more than one procedural block in the same module, and outputs of one are inputs to another, the simulated result may depend on how expressions from different blocks are interleaved by the simulator.

VL2MV can be invoked as a stand-alone tool on a Verilog file to produce a BLIF-MV file. This can be read in VIS with the command *read_blif_mv*. As an alternative, the command *read_verilog* can be directly used to read in a Verilog file. This invokes VL2MV internally.

2.3 Features of Verilog Supported by VL2MV

VL2MV supports a synthesizable subset of Verilog, and also extends it minimally to make it usable for formal verification. We survey the features that characterize Verilog as supported by VL2MV.

[1] Circuits with combinational cycles are legal in BLIF-MV, but currently they are not processed by VIS.

[2] VL2MV can also extract quantitative timing information from a timed Verilog program, producing BLIF-MVT, based on timed automata, that is an extension of BLIF-MV with timing constructs [3]. Since verification with quantitative timing is not handled in the current version of VIS, this feature is of no further interest here.

2.3.1 Assignments

Continuous assignments are always active, i.e., whenever any input changes, the output is updated instantaneously. Only `wire` variables can be used at the left hand side of continuous assignments. Continuous assignments describe the combinational behavior of a circuit.

Procedural assignments (= within a procedural block), also referred to as *blocking assignments*, execute sequentially within a procedural block, changing the content of state variables, until the execution is blocked by a pause. VL2MV compiles procedural blocks based on the assumption that each basic block will be executed atomically if the delay/event control of the block is satisfied. VL2MV assumes also that execution of procedural assignments takes zero hardware time. All procedural blocks with active event controls get executed concurrently. Notice that a Verilog simulator does not treat simple blocks as atomic. If there is more than one procedural block sharing the same `reg` variables, caution should be taken to make sure that the desired behavior does not depend on a specific interleaving among processes.

Procedural assignments update variables instantaneously, meaning that they change the left-hand side variable so that the statement following the assignment (in the same process, or `always` statement) can observe the value change. On the other hand, other processes (for instance, other `always` statements or continuous assignments) cannot see the change until the next clock cycle. Because of this, race conditions might arise among multiple procedural assignments. *Non-blocking procedural assignments* (<=) provide a mechanism that defers the assignment without blocking the execution of statements in a block. On encountering a non-blocking assignment, the right hand-side of the assignment is evaluated according to the most recent values of the referred variables. However, without changing the variable on the left hand-side, program execution continues. Then variables are updated simultaneously at the very beginning of the next time slot. For VL2MV, non-blocking procedural assignments should never be used, since they might introduce unwanted nondeterminism.

2.3.2 Nondeterminism

Non-blocking assignments also provide a way to introduce nondeterminism on `reg` variables. If there is more than one non-blocking assignment in the current time slot assigning to the same register variable, then the value of that register variable in the next clock cycle will be nondeterministically chosen from those assigned values. *Even though* VL2MV *accepts this way of specifying nondeterminism, in VIS, unlike in HSIS, multiple assignments are not considered legal nondeterminism.*

Instead, a nondeterministic construct, `$ND`, has been added to Verilog to specify nondeterminism on wire variables and is the only legal way to introduce nondeterminism in VIS. For example, to require that the output at a particular state is nondeterministically GO or NOGO, one can introduce a new variable, r, and write the following Verilog fragment.

```
assign r=$ND{GO,NOGO};
.
.
always@(posedge clk) begin
.
.
state = r;
.
.
end
```

2.3.3 Symbolic Variables

Sometimes it is desirable to specify and examine the value of some variables symbolically, rather than having to explicitly encode them. VL2MV extends Verilog to allow users to declare symbolic variables using an enumerated type mechanism similar to the one available in the C programming language. As an example, we introduce a symbolic type named `door`:

```
typedef enum {OPEN,OPENING,CLOSED,CLOSING} door;
```

2.4 Implicit vs. Explicit Clocking

The clocking discipline is determined by the definition of the Verilog simulator, and it can be either implicit or explicit. Implicit is the default. Explicit may be required in some cases.

A Verilog simulator is an event-driven *passive scheduler*. A simulator schedules events generated from Verilog modules and then sends them to modules which are sensitive to these events. Statements with sensitized events (active statements) are executed and in turn more events are generated, which are then scheduled by the simulator. The simulator itself does not generate any event, but it coordinates between the producers and consumers of events. Hence, to write a synchronous system, a designer needs to write a small clock generator, i.e., an event generator which creates events in time. The produced events provoke a chain of reactions among modules. The system reaches a stable state when there are no more events other than the clocking event. The next clocking event is then chosen by the simulator, and simulation time is advanced according to the time stamp of the newly scheduled clocking event. We call the system *implicitly clocked* when all transitions are synchronized by an implicit *time*. For an implicitly clocked system hardware resources will not be allocated for a synchronizing variable. Also, for implicitly clocked designs, one symbolic latch (or state variable) is allocated for each `reg` variable, and synchronization variables are dropped. By default, implicit clocking semantics is assumed.

On the other hand, for some designs, the operation of a system depends explicitly on several phases (rising edge, falling edge, 1-level, 0-level) of one or more synchronizing signals (generally referred to as *clocks*). In such a case the clock signals should be interpreted literally and hardware resources should be allocated. A design is called *explicitly clocked* if synchronizing signals are to be compiled literally into hardware. For explicitly clocked systems, each `reg` variable is modeled by a symbolic latch along with some extra logic to emulate the clocking mechanism. An example of explicit clocking declared by the user is the following. Suppose that a system is composed of parallel components that progress differently according to synchronization signals exchanged among them by means of `wait` statements. Then it is necessary to declare an explicit clocking signal:

```
module env;
reg clk;

initial clk=0;

always #1 clk = !clk;

endmodule
```

This code generates a clocking signal `clk` with a cycle of two time units used to drive the whole system and make it simulatable. In this case VL2MV must be invoked with the options *-c -F*, meaning explicit clocking (*-c*), and ignore all timing (*-F*).

2.5 Verilog for VL2MV: Hints and Traps

In this section a list of hints to follow, and traps to avoid, is provided for writing Verilog for VIS.

1. Inside an `always` block, only blocking assignments to `reg` variables are allowed. Therefore do not write to an intermediate variable (that is a `wire` by definition) inside an `always` block and do not use non-blocking assignments (<=) ever.

2. If variables that must be assigned depend on each other, assign them in separate `always` blocks, otherwise the behavior may depend on the order of execution.

3. Inside an `always` block, blocking assignments = are sensitive to the order of the statements. Thus the following two fragments evaluate differently:

```
state = 1;
out = state;
```

```
out = state;
state = 1;
```

   Since we do not allow non-blocking assignments (<=) inside an `always` block, we have to analyze the order of evaluation to be certain that we have the desired behavior.

4. It is not legal to have a block of assignments, as in:

```
assign begin
     x = 1;
     y = 2;
end
```

   However, it is legal to have a block of assignments for an `initial` statement:

```
initial begin
     x = 1;
     y = 2;
end
```

5. In `always` blocks, at the next clock, `reg` variables keep their previous values if they are not explicitly assigned to.

6. Introduce nondeterminism using only `$ND` assignments to wires. Unlike in HSIS, multiple assignments such as:

```
always@(posedge clk) begin
state <= GO;
state <= NOGO ;
end
```

   are not considered legal nondeterminism in VIS.

7. VL2MV will reject a Verilog description containing an unspecified initial state. If the user wants a nondeterministic initial state, it should be specified explicitly using a `$ND` construct, for example: `initial x = $ND(a,b,c);` in this case, a nondeterministic constant will be created with a name as `x$initial_n23`.

8. `for` statements are supported by VL2MV. Here is an example:

```
always@(posedge clk) begin
// randomly push floor buttons
for (i=0;i<=`floor-1;i=i+1) begin
        if (random_up[i]) up_floor_buttons[i]=ON;
        if (random_down[i]) down_floor_buttons[i]=ON;
end
```

Note that (unfortunately) a for loop can only be used inside an always block. VL2MV by default macro-expands (unrolls) the Verilog code before processing it. Use option *-u* to suppress unrolling.

9. A wire can be a vector but not an array. However, a reg can be an array: wire[1:10] a; is correct but wire a[1:10]; is not. As an example of how things differ for wire and reg variables consider:

```
typedef enum {UP,DOWN} dir;
wire[1:`elev] stop_next;
dir reg direction[1:`elev];

typedef enum {on, off, interm} onoff;
```

onoff reg a[1:10] is correct, but onoff wire a[1:10] and onoff wire[1:10] a are not correct. Also reg [1:`width] locations[1:`elev] is correct, but onoff reg [1:`width] locations[1:`elev] is not correct, since the latter are a two dimensional array of symbolic type.

10. VL2MV puts an extra buffer for $ND constructs when the *-Z* option is used , while by default it does not. In other words, by default VL2MV connects the left-hand side variable directly to the nondeterministic table for $ND. Notice that the only legal usage of $ND when *-Z* is not used is: assign <var> = $ND(...); where the assign statement is a continuous assignment. The generated nondeterministic table will use <var> as the output variable. Instead if the *-Z* option is turned on, one can use $ND definitions in expressions , as in: assign a = $ND(0,1) + b, or assign a = (sel) ? $ND(0,1) : b. In this case intermediate variables are generated for the $ND construct. We recommend only using the default value and explicitly naming the nondeterministic value, since this will become a pseudo-input to VIS and will in this case have a name given by the user.

11. In VIS we insist on having nondeterminism only for single output constants. A BLIF-MV table like

```
.table -> x
-
```

is allowed and leads to a pseudo-input. However a table like

```
.table -> x<0> x<1>
0 0
0 1
1 0
```

is not allowed. The reason is that this table represents a relation and cannot be split into two independent, nondeterministic, single output tables, since replacing it with

```
.table -> x<0>
-
.table -> x<1>
-
```

would lead to the possibility of x = 1 1.

Such a situation comes up naturally when we want a variable to have any of the integers 0,1,2. But we have to assign 2 bits to hold the variable, and we want to be able to increment or decrement the variable later on (so it must be an integer, rather than a symbolic variable):

```
wire[0:1] x;
assign x = $ND(0,1,2);
```

VL2MV generates BLIF-MV for this code that is not accepted by VIS. An awkward way around this is:

```
assign temp=$ND(0,1,2,3);
assign location = (temp==3)?2:temp;
```

2.6 BLIF-MV

BLIF-MV is a low-level language designed for describing hierarchical symbolic sequential systems with nondeterminism. A system can be composed of interacting sequential subsystems, each of which can be again described as a collection of communicating sequential subsystems. This makes it possible to describe systems in a hierarchical fashion. The internal data structure of SIS does not support hierarchical representations. Hence, even though BLIF can describe hierarchy, BLIF descriptions are flattened into a single-level representation within SIS. In VIS, however, the original hierarchy specified in BLIF-MV is preserved in internal data structures so that true hierarchical synthesis and verification is possible.

BLIF-MV also allows nondeterministic gates [3] and hence makes it possible to model nondeterministic systems. For instance, a design in its early stages may contain nondeterminism, as many aspects may not be yet decided. Lastly, BLIF-MV supports multi-valued variables, which can be used to simplify system descriptions.

The semantics of BLIF-MV is defined over flattened networks, using a combinational/sequential concurrency model. There are four basic primitives: *variables*, *tables* (intuitively nondeterministic gates), *wires* and *latches*. A *variable* takes values from some finite domain. A relation defined over a set of variables is represented using a *table*. The variables of a table are divided into inputs and outputs. A particular variable can be designated as an output in at most one table. Tables are inter-connected using *wires*. If a table is deterministic and Boolean, it may also be thought of as a logic gate. Wires may only take values in the domain of the corresponding variable. A *latch* is a specialized element that can be placed on a wire. The latch divides the wire into two parts; the input to the latch, and the output of the latch. A set of initial values is associated to every latch; they must be a subset of the set of values of its wire. A state is an assignment of values to the latches of a model, where a value assigned to a latch must be in its domain. An initial state is a state where every latch takes a value from its set of initial values. Note that the system can have more than one initial state in general.

At every time point, the system is in some state, where each latch has a value. At every clock tick, all the latches update their values. These values then propagate through tables until all the wires have a consistent set of values. If a latch is encountered during the propagation, i.e., an output of a table is an input of an latch, the propagation process through that latch is stopped. Note that because of nondeterminism, given a single state, there may be several consistent sets of values. This semantics can be seen as a simple extension of the standard semantics of synchronous single-clocked digital circuits. In fact, if every table is deterministic and every latch has a single initial state, the two semantics are exactly equal. The only differences are in the interpretation of nondeterministic tables and latches with multiple initial states.

In VIS the command *read_blif_mv* reads a BLIF-MV description created by VL2MV, or some other means, and then sets up a corresponding internal data structure. The *write_blif_mv* command writes a BLIF-MV description to a file. The BLIF-MV format is not meant to be read or written directly by the user, even though simple examples in BLIF-MV may exhibit some degree of clarity. In the VIS documentation, the syntax of BLIF-MV is described in the document entitled “BLIF-MV”.

---

[3] These gates generate some output from the set of pre-specified outputs.

2.7 BLIF

BLIF (Berkeley Logic Interchange Format) is an intermediate format to describe sequential circuits. It has been defined as an entry point to logic optimizers such as SIS, the synthesis system developed at UC Berkeley. A BLIF file represents a sequential circuit either as an interconnection of logic gates and latches or as the state transition table of a finite state machine (FSM) or in both ways (an FSM and a corresponding gate-level implementation). It is possible to have VIS and SIS interact, by sending to SIS a binary encoded and deterministic sequential circuit and receiving back an optimized version of the same. Notice that even though SIS can also handle KISS files (i.e., partially encoded and partially deterministic FSMs), currently VIS outputs hardware FSM descriptions (i.e., a netlist describing completely encoded and completely deterministic FSMs), for SIS input. For a description of BLIF and SIS we refer to the tutorial paper [4]. A BLIF description can be read directly into VIS by the command *read_blif*, while *write_blif* converts the internal VIS data structure into a BLIF file readable by SIS. The synthesis path from VIS to SIS and back and related commands are described in Chapter 5.

2.8 Nondeterminism and Incomplete Specification

The only form of nondeterminism supported in VIS is the construct `$ND` in Verilog. A system so described is considered internally as deterministic, because pseudo-input variables are introduced to "control" this form of nondeterminism. Pseudo-input variables are, by definition, those variables introduced by a `$ND` construct. A Verilog nondeterministic assignment, like `assign rand_choice = $ND(0,1);` is translated by VL2MV into the table:

```
# assign rand_choice  = $NDset ( 0,1 )
.names -> rand_choice
0
1
```

There are other ways of introducing nondeterminism in Verilog that are supported by VL2MV and HSIS, but are not supported by VIS.

VL2MV always produces completely specified BLIF-MV tables. However, a BLIF-MV file not produced by VL2MV (but by another tool or manually) may contain incomplete specification. When the internal data structure is built, each table is checked for determinism and complete specification (with the exception of the pseudo-inputs). This is a conservative test, in the sense that one or more tables may be nondeterministic while the entire network is deterministic. Similarly, one or more tables may be incompletely specified while the network is completely specified.

2.9 Example: a Traffic Light Controller

In this tutorial, we will use the example of a traffic light controller (TLC), first introduced by Mead and Conway [5], to illustrate several concepts.

A little used farm road intersects a multi-lane highway; a traffic light controls the traffic at the intersection. The light controller is implemented to maximize the time the highway light remains green. The *main* module ties together a timer, a sensor, a farm light control and a highway control submodules.

The timer submodule implements a timer, that outputs "short" and "long" timeouts. The highway light stays green for at least "long" time. Any time after "long" time, if there is a car waiting on the farm road, then the farm light turns green. The farm light remains green until there are no more cars on the farm road, or until the long timer expires. The yellow light for both directions stays yellow for "short" time. Note that only a single timer is used for both the farm road and highway controllers. In theory, this could lead to conflicts; as implemented, such conflicts are avoided. From the START state, the timer produces the signal “short” after a nondeterministic amount of time. The signal “short” remains asserted until the timer is reset (via the signal “start”). From the SHORT state, the timer produces the signal “long” after a nondeterministic amount of time. The signal “long” remains asserted until the timer is reset. Notice that the use of nondeterminism in the description of the timer models an infinite number of actual implementations, each with a different set-up for the “short” and “long” periods.

The farm light stays RED until it is enabled by the highway control. At this point, it resets the timer, and moves to GREEN. It stays in GREEN until there are no cars, or the long timer expires. At this point, it moves to YELLOW and resets the timer. It stays in YELLOW until the short timer expires. At this point, it moves to RED and enables the highway controller.

The highway light stays RED until it is enabled by the farm control. At this point, it resets the timer, and moves to GREEN. It stays in GREEN until there are cars on the farm road and the long timer expires. At this point, it moves to YELLOW and resets the timer. It stays in YELLOW until the short timer expires. At this point, it moves to RED and enables the farm controller.

There is a single sensor that detects the presence of a car in either direction of the farm road. At each clock tick, it nondeterministically reports that a car is present or not.

The fact that the timer is a Moore machine (while the highway and farm controllers are Mealy machines) ensures that the component FSMs can be combined to form a well-defined product FSM (without combinational cycles).

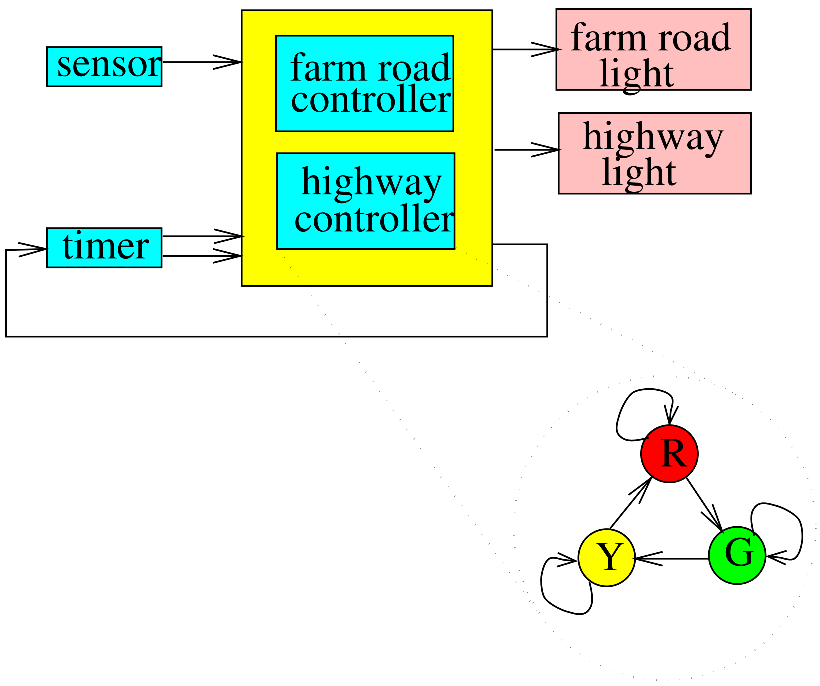

Figure 2.1: Block diagram of traffic light controller.

Fig. 2.1 is a block diagram for the entire controller, and Fig. 2.2 describes the four FSMs that make up the system.

This entire example is written in Verilog as:

```
/* Written by Tom Shiple, 25 October 1995 */

/* Symbolic variables */
typedef enum {YES, NO} boolean;
typedef enum {START, SHORT, LONG} timer_state;
```

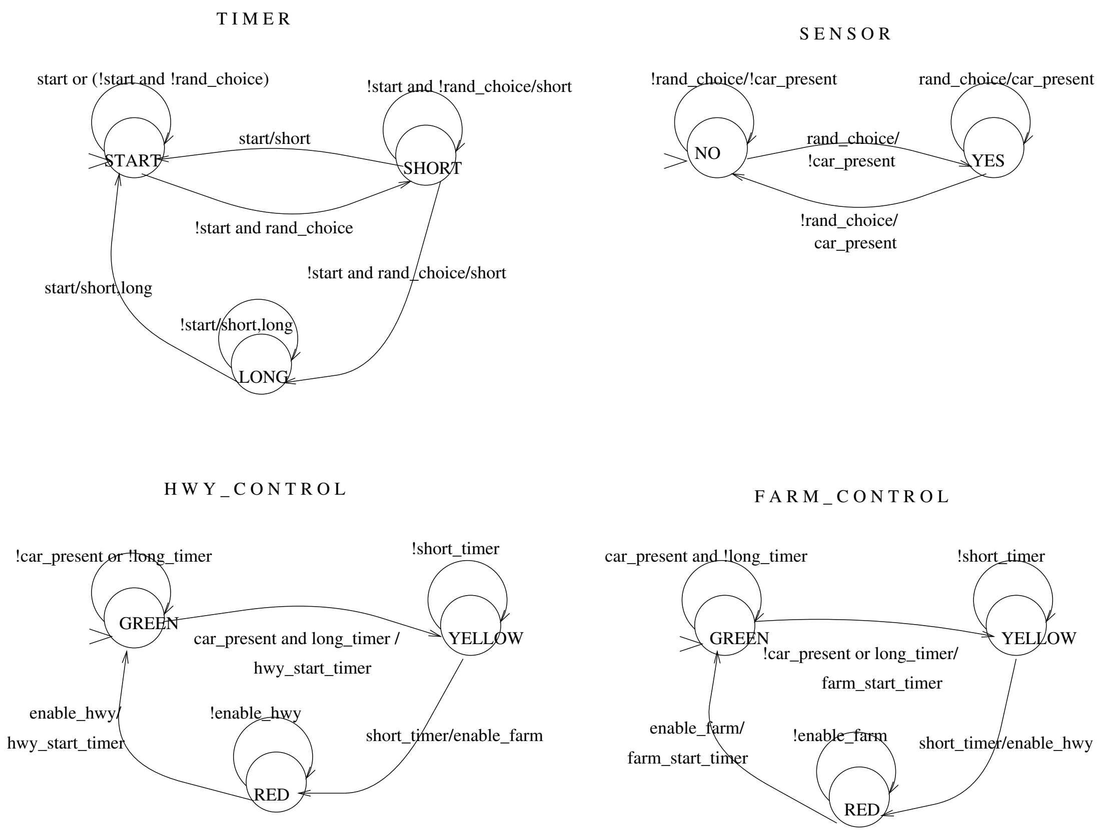

Figure 2.2: State transition graphs of FSMs of TLC.

```
typedef enum {GREEN, YELLOW, RED} color;

module main(clk);
input clk;

color wire farm_light, hwy_light;
wire start_timer, short_timer, long_timer;
boolean wire car_present;
wire enable_farm, farm_start_timer, enable_hwy, hwy_start_timer;

assign start_timer = farm_start_timer || hwy_start_timer;

timer timer(clk, start_timer, short_timer, long_timer);
sensor sensor(clk, car_present);
farm_control farm_control(clk, car_present, enable_farm, short_timer, long_timer,
      farm_light, farm_start_timer, enable_hwy);
hwy_control hwy_control (clk, car_present, enable_hwy,  short_timer, long_timer,
      hwy_light, hwy_start_timer, enable_farm);

endmodule

module sensor(clk, car_present);
input clk;
output car_present;

wire rand_choice;
boolean reg car_present;

initial car_present = NO;
assign rand_choice = $ND(0,1);

always @(posedge clk) begin
```

```
    if (rand_choice == 0)
        car_present = NO;
    else
        car_present = YES;
end
endmodule

module timer(clk, start, short, long);
input clk;
input start;
output short;
output long;

wire rand_choice;
wire start, short, long;
timer_state reg state;

initial state = START;
assign rand_choice = $ND(0,1);

/* short could as well be assigned to be just (state == SHORT) */
assign short = ((state == SHORT) || (state == LONG));
assign long  = (state == LONG);

always @(posedge clk) begin
    if (start) state = START;
        else
            begin
            case (state)
            START:
                    if (rand_choice == 1) state = SHORT;
            SHORT:
                    if (rand_choice == 1) state = LONG;
                    /* if LONG, remains LONG until start signal received */
            endcase
            end
end
endmodule

module farm_control(clk, car_present, enable_farm, short_timer, long_timer,
      farm_light, farm_start_timer, enable_hwy);
input clk;
input car_present;
input enable_farm;
input short_timer;
input long_timer;
output farm_light;
output farm_start_timer;
output enable_hwy;

boolean wire car_present;
wire short_timer, long_timer;
wire farm_start_timer;
wire enable_hwy;
wire enable_farm;
color reg farm_light;

initial farm_light = RED;
assign farm_start_timer = (((farm_light == GREEN) && ((car_present == NO) || long_timer))
                          || (farm_light == RED) && enable_farm);
assign enable_hwy = ((farm_light == YELLOW) && short_timer);

always @(posedge clk) begin
     case (farm_light)
     GREEN:
             if ((car_present == NO) || long_timer) farm_light = YELLOW;
     YELLOW:
             if (short_timer) farm_light = RED;
     RED:
```

```
            if (enable_farm) farm_light = GREEN;
     endcase
end
endmodule

module hwy_control(clk, car_present, enable_hwy, short_timer, long_timer,
      hwy_light, hwy_start_timer, enable_farm);
input clk;
input car_present;
input enable_hwy;
input short_timer;
input long_timer;
output hwy_light;
output hwy_start_timer;
output enable_farm;

boolean wire car_present;
wire short_timer, long_timer;
wire hwy_start_timer;
wire enable_farm;
wire enable_hwy;
color reg hwy_light;

initial hwy_light = GREEN;
assign hwy_start_timer = (((hwy_light == GREEN) && ((car_present  == YES) && long_timer))
                         || (hwy_light == RED) && enable_hwy);
assign enable_farm = ((hwy_light == YELLOW) && short_timer);

always @(posedge clk) begin
     case (hwy_light)
     GREEN:
            if ((car_present == YES) && long_timer) hwy_light = YELLOW;
     YELLOW:
            if (short_timer) hwy_light = RED;
     RED:
            if (enable_hwy) hwy_light = GREEN;
     endcase
end
endmodule
```

Chapter 3

Introduction to Formal Verification

Formal verification is the process of checking whether a design satisfies some requirements (properties). We are concerned with the formal verification of designs that may be specified hierarchically (as illustrated in the previous section); this is also consistent with how a human designer operates. In order to formally verify a design, it must first be converted into a simpler "verifiable" format. The design is specified as a set of interacting systems; each has a finite number of configurations, called states. States and transition between states constitute FSMs. The entire system is an FSM, which can be obtained by composing the FSMs associated with each component. Hence the first step in verification consists of obtaining a complete FSM description of the system. Given a present state (or current configuration), the next state (or successive configuration) of an FSM can be written as a function of its present state and inputs (transition function or transition relation).

We note that this entire framework is one of discrete functions. Discrete functions can be represented conveniently by BDDs (binary decision diagram; a data structure that represents boolean (2-valued) functions) and its extension MDDs (multi-valued decision diagram; a data structure that represents finite valued discrete functions). We use BDDs and MDDs to represent all quantities required in this discrete space (more specifically the transition functions, the inputs, the outputs and the states of the FSMs). For BDDs and MDDs to be efficient representations of discrete functions, a good ordering of input variables (actual inputs, outputs, state) of the functions must be computed. In general, BDDs operate on sets of points rather than individual points; this is called *symbolic manipulation*.

The two most popular methods for automatic formal verification are language containment and model checking. The current version of VIS emphasizes model checking, but it also offers to the user a limited form of language containment (language emptiness).

3.1 Model Checking of Temporal Logic

A finite state system can be represented by a labeled state transition graph, where labels of a state are the values of atomic propositions in that state (for example the values of the latches). Properties about the system are expressed as formulas in temporal logic of which the state transition system is to be a "a model". Model checking consists of traversing the graph of the transition system and of verifying that it satisfies the formula representing the property, i.e., the system is a model of the property.

3.1.1 Computation Tree Logic

Temporal logic expresses the ordering of events in time by means of operators that specify properties such as "p will eventually hold". There are various versions of temporal logic. One is computational tree logic (CTL). Computation trees are derived from state transition graphs. The graph structure is unwound into an infinite tree rooted at the initial state. Fig. 3.1 shows an example of unwinding a graph into a tree. Paths in this tree represent all possible computations of the system being modeled. Formulae in CTL refer to the computation tree derived from the model. CTL is classified as a branching time logic because it has operators that describe the branching structure of this tree.

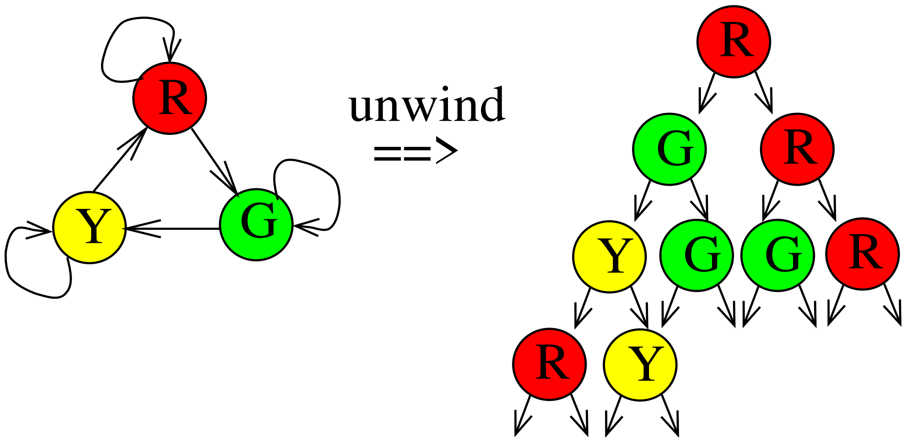

Figure 3.1: Unwinding of state transition graph.

Formulae in CTL are built from atomic propositions (where each proposition corresponds to a variable in the model), standard boolean connectives of propositional logic (e.g., AND, OR, XOR, NOT), and temporal operators. Each temporal operator consists of two parts [1]: a path quantifier (A or E) followed by a temporal modality (F, G, X, U). All temporal operators are interpreted relative to an implicit "current state". There are in general many execution paths (sequences of state transitions) of the system starting at the current state. The path quantifier indicates whether the modality defines a property that should be true of all those possible paths (denoted by universal path quantifier A) or whether the property needs only hold on some path (denoted by existential path quantifier E). The temporal modalities describe the ordering of events in time along an execution path and have the following intuitive meaning:

1. $F\ \phi$ (reads "ϕ holds sometime in the future") is true of a path if there exists a state in the path where formula ϕ is true.

2. $G\ \phi$ (reads "ϕ holds globally") is true of a path if ϕ is true at every state in the path.

3. $X\ \phi$ (reads "ϕ holds in the next state") is true of a path if ϕ is true in the state reached immediately after the current state in the path.

4. $\phi\ U\ \psi$ (reads "ϕ holds until ψ holds", called "strong until" [2]) is true of a path if ψ is true in some state in the path, and ϕ holds in all preceding states.

In the VIS documentation there is a description of the syntax of CTL in the document entitled "CTL Syntax". In this chapter CTL formulas will be written in a simplified syntax.

The state of a system consists of the values stored in all latches. Each formula of the logic is either true or false in a given state; its truth is evaluated from the truth of its subformulas in a recursive fashion, until one reaches atomic propositions that are either true or false in a given state. A formula is satisfied by a system if it is true for all the initial states of the system. If the property does not hold, the model checker will produce a counterexample, that is an execution path that witnesses the failure. An efficient algorithm

[1]A formula that contains any temporal modality (F, G, X, U) without an associated path quantifier (A, E) is <u>not</u> a legal CTL formula.

[2]"Weak until" is when ϕ holds forever, i.e., ψ is not required to hold at some state in the future.

for automatic model checking used also in VIS has been described by Clarke et al. [7]. The following table shows examples of evaluations of formulas on the computation tree of Fig. 3.1:

formula	T/F
EG (RED)	true
E (RED U GREEN)	true
AF (GREEN)	false

3.1.2 Specification of Properties in CTL

Temporal logic formulas can be difficult to interpret, so that a designer may fail to understand what property has been actually verified. Therefore it is important to be familiar with the most common constructs of CTL used in hardware verification.

1. $AG(req \rightarrow AF\ ack)$
   For all reachable states (AG), if req is asserted in the state, then always at some later point (AF) we must reach a state where ack is asserted. AG is interpreted relative to the initial states of the system. AF is interpreted relative to the state where req is asserted. In other words, it is always the case that if the signal req is high, then eventually ack will also be high. A common mistake would be to write $req \rightarrow AF\ ack$, instead of $AG(req \rightarrow AF\ ack)$. The meaning of the former is that if req is asserted in the initial state, then it is always the case that eventually we reach a state where ack is asserted, while the latter requires that the condition is true for any reachable state where req holds. If req is identically true, $AG(req \rightarrow AF\ ack)$ reduces to $AG\ AF\ ack$.

2. $AG\ AF\ enabled$
   From every reachable state, for all paths starting at that state we must reach another state where $enabled$ is asserted. In other words, $enabled$ must be asserted infinitely often.

3. $AG\ EF\ restart$
   From any reachable state, there must exist a path starting at that state that reaches a state where $restart$ is asserted. In other words, it must always be possible to reach the restart state.

4. $EF(started \wedge \neg ready)$
   It is possible to get to a state where $started$ holds, but $ready$ does not hold.

5. $AG(send \rightarrow A(send\ U\ receive))$
   It is always the case that if $send$ occurs, then eventually $receive$ is true, and until that time, $send$ must continue to be true.

6. $AG(inp \rightarrow AX\ AX\ out)$
   Whenever inp goes high, out will go high within two clock cycles.

7. $EF(a \wedge EX(a \wedge EX\ a)) \rightarrow EF(b \wedge EX\ EX\ c)$
   If it is possible for a to be asserted in three consecutive states, then it is also possible to reach a state where b is asserted and from there to reach in two more steps a state where c is asserted.

We summarize the most common CTL templates with the corresponding English language meaning:

1. AGp is "nothing bad ever happens" ($\neg p$ is bad). Used to specify an invariant, i.e., a condition that must be true in all states. Helpful for partial correctness (no wrong answers are produced), mutual exclusion (no two processors are in a critical section simultaneously), deadlock freedom (no deadlock state is reached).

2. $AF\ AG\ p$ is "eventually the system is confined to states where p is always true" or "the system stays out of states where p is true only a finite number of times". It can be used to specify the property of finite number of failures in the system.

3. $AG(p \rightarrow\ \ AF\ q)$ is "from all reachable states where p is true, something good, q, eventually happens". Helpful to express total correctness (termination eventually occurs with correct answers), accessibility (eventually a requesting process will enter its critical section), starvation freedom (eventually service will be granted to a waiting processor). If p is always true, it reduces to $AG\ AF\ q$.

4. $AG\ \ AF\ \ q$ is "infinitely often q", i.e., from any reachable state one must reach a state where q is asserted. It can be used, for instance, to enforce a reset condition from any state.

5. $AF\ q$ is "something good, q, eventually (or finally) happens" (less restrictive than $AG\ AF\ q$).

6. $AG\ EF\ p$ is "always p possible". It can detect, for instance, the absence of deadlocks, by requiring that is it always possible to reach deadlock-free states. This is an example of a CTL property that cannot be represented by an ω-automaton [3].

7. $AG\ true$ forces a complete traversal of the states of the system.

8. $EF\ p$ is "p is possible". This is another example of a CTL property that cannot be represented by an ω-automaton.

Caveats

1. The variables appearing in a CTL formula must be a function of register variables (e.g., states or outputs attached to states). Variables that depend on inputs or pseudo-inputs are not allowed, since this could lead to a state where both p and $\neg p$ are true, depending on the input.

2. The propositional logic operator $\rightarrow$, as in $a \rightarrow b$ is equivalent to $\neg a + b$, and is satisfied by $\neg a$. Do not use it in place of $a \star b$, which is true if and only if a and b are both true.

3. The syntax of CTL and of Verilog are different. For instance, we have:

```
Verilog CTL      meaning

&&       *       AND
||       +       OR
==       =       equal
a!=NO    !(a=NO) not equal
         ->      implies
         ^       xor
```

3.1.3 Fairness Constraints

It is often necessary to introduce some notion of fairness. For example, if the system allocates a shared resource among several users, only those paths along which no user keeps the resource forever should be considered. CTL by itself cannot express assertions about correctness along fair paths.

Fair CTL is a modification of CTL to handle fairness. Fair CTL is characterized by the introduction of *fairness constraints*, which are sets of states expressed by means of CTL formulas, each giving a fairness condition; a *fair path* is a path along which each fairness condition is satisfied infinitely often. These types

[3] It is possible to show two transition systems that recognize the same language, of which one satisfies $AG\ EF\ p$, and the other does not.

of fairness constraints are called Büchi type. More general fairness constraints, such as Street type, are not allowed currently. Fair CTL has the same syntax as CTL, but the semantics is modified so that all path quantifiers only range over fair paths. VIS supports Fair CTL; in the documentation we may sometimes refer to CTL, where we really mean Fair CTL.

An example of a fairness condition is p, that restricts the system to only those paths where p is asserted infinitely often.

3.2 Properties and Fairness Conditions of Traffic Light Controller in CTL

Not all behavior exhibited by the description of the Traffic Light Controller is valid. In order to restrict the behavior we impose the following two fairness constraints. The first is:

```
!(timer.state=START);
```

The timer must eventually leave the START state. This constraint prevents it from staying in START forever. The second fairness constraint:

```
!(timer.state=SHORT);
```

ensures that the timer must eventually leave the SHORT state. Liveness properties (e.g, cars on farm road and highway will eventually cross) would not pass if these fairness constraints are not placed on the timer.

One obvious property to check is that the light is not green in both directions at the same time, ensuring that collisions do not occur between traffic on the farm road and highway. This property is written as the CTL formula:

```
AG ( !((farm_light = GREEN) * (hwy_light = GREEN)) );
```

To ensure that a car on the farm road eventually crosses the intersection, we require that if a car is present on the farm road, and the timer is long, then eventually the farm light will turn green. In CTL this is written as:

```
AG(((car_present = YES) * (timer.state = LONG)) -> AF(farm_light = GREEN));
```

In addition, regardless of what happens on the farm road, the highway should always be green in the future:

```
AG(AF(hwy_light = GREEN));
```

The presence of a car on the farm road does not guarantee that eventually the farm light will turn green. A car may approach, and then back away, all before the timer goes long. This property is not necessary for safety, it just maximizes the time that the highway light is green. Thus, it is desirable that the system satisfies the following property:

```
!(AG((car_present = YES) -> AF(farm_light = GREEN)));
```

3.3 Language Containment

There are properties of practical interest that cannot be described in CTL. An example is the "almost always" property: a condition, q, always holds after a finite number of transitions (note that formulas $FG\ q$ and $AF\ G\ q$ would express this, but these are not legal CTL formulas). This property looks a lot like $AF\ AG\ q$, but it is not the same. One can exhibit a transition system where $AF\ G\ q$ is true, while $AF\ AG\ q$ is false.

A solution would be to use a more expressive type of temporal logic (for instance, the previous property could be expressed in PLTL or CTL*). But there would be drawbacks, such as the higher complexity of algorithms for model checking. An alternative is to use another verification paradigm, called language containment, based on the theory of ω-automata. For example, it is easy to express the previous "almost always" property using an automaton.

Currently VIS supports a restricted form of language containment. We review briefly the idea of language containment: for a system to satisfy a property it must be that $L(S) \subseteq L(T)$, where S is an ω-automaton representing the system, T is an ω-automaton representing the property and L is the language accepted by the automaton. It is a fact that $L(S) \subseteq L(T)$ is equivalent to $L(S) \cap \overline{L(T)} = \emptyset$.

To achieve language containment checking we represent the composition of the given system with a model representing the negation of the property and check it for language emptiness. The language of the composed system is empty if and only if the system satisfies the property T.

Language emptiness is used not only to verify properties that cannot be expressed in Fair CTL, but also to check whether the abstraction of a system still contains the original system. In both cases one must complement an ω–automaton (T), and this is hard to do if the automaton is nondeterministic (as is usually the case for an abstraction). The fact that complementation of a deterministic property is easy, while complementation of a nondeterministic property may be hard, is a key problem with language containment. This has prompted a lot of research on different classes of ω-automata with different expressiveness and difficulty of complementation. Currently VIS supports language emptiness of nondeterministic Büchi automata; only it is the responsibility of the user to derive the complement of a given nondeterministic property. Büchi automata acceptance conditions are states that must be reached infinitely often and they are specified by means of fairness constraints. Thus to use language containment, the user must insert in the Verilog hierarchy a monitor, which represents the complement automaton structure, and impose a set of fairness conditions to specify the complement automaton acceptance conditions, i.e., the acceptance conditions are specified in terms of fair paths.

As a final note, inside VIS, language emptiness (language containment) is reduced to CTL, by checking the CTL formula $E\ G\ true$ on the system (system composed with complemented property), i.e., whether there is an infinite path (notice that $true$ is always satisfied), satisfying appropriate fairness constraints.

Chapter 4

Formal Verification in VIS

In this chapter we describe the usage and the relation between the VIS commands that perform formal verification. The main sections are:

1. building an internal representation of the finite-state system,
2. FSM traversal,
3. specification of fairness constraints,
4. language emptiness,
5. model checking,
6. equivalence checking, and
7. simulation.

4.1 Representing the System for Verification

In this section, we describe the steps involved in converting a BLIF-MV description into an internal FSM representation.

4.1.1 Building the Flattened Network

The compound *init_verify* command executes the entire set of required initialization commands. When a BLIF-MV description is read into VIS, it is stored as a "hierarchy" tree, which is a hierarchical description of the design; it consists of modules (also called *hnodes*) that in turn consist of sub-modules (also hnodes) that are related in some fashion. This relation is represented as a table, which implements the output function in terms of the sub-module inputs. The *print_hierarchy_stats* command in VIS prints hierarchy information, and the *print_models* command lists statistics on all the models in the hierarchy. Other useful print commands are *print_io* and *print_latches*.

The hierarchy can be described by a tree. The root of the tree is the main module, and the leaves are lower level instantiations of modules. The hierarchy in VIS can be traversed in a manner similar to traversing directories in UNIX. It is possible to reach a desired node in the tree by walking up and down with the *cd* command. At any node simulation, verification and synthesis operations can be performed. The command *pwd* prints the name of the current node. The command *ls* lists all the nodes (submodules) in the current node; *ls -R* lists all the nodes in the current subtree.

The first step towards verification consists of "flattening" this hierarchical description into a single network (netlist of multi-valued logic gates). The output is computed from the inputs of the design by the network circuit, which consists of logic gates, interconnections between them, and latches to represent the sequential elements. The *flatten_hierarchy* command creates this network, and the *print_network* command can be used to print it. Other related commands are *print_network_stats* command that prints statistics about the network, and *test_network_acyclic* command that checks the network for combinational cycles. On the Traffic Light Controller example these commands work as follows :

```
UC Berkeley, VIS Release 1.0 (compiled 11-Dec-95 at 10:36 AM)
VIS> read_blif_mv tlc.mv
Warning: Some variables are unused in model main.
vis> print_hierarchy_stats
Model name = main, Instance name = main
inputs = 0, outputs = 0, variables = 12, tables = 3, latches = 0, children = 4
vis> print_models
Model name = hwy_control
inputs = 4, outputs = 3, variables = 49, tables = 44, latches = 1
subckts = 0
Model name = sensor
inputs = 0, outputs = 1, variables = 12, tables = 11, latches = 1
subckts = 0
Model name = main
inputs = 0, outputs = 0, variables = 12, tables = 3, latches = 0
subckts = 4
Model name = timer
inputs = 1, outputs = 2, variables = 40, tables = 38, latches = 1
subckts = 0
Model name = farm_control
inputs = 4, outputs = 3, variables = 49, tables = 44, latches = 1
subckts = 0
vis> flatten_hierarchy
vis> print_network_stats
main  combinational=142  pi=0  po=0  latches=4  pseudo=2  const=40  edges=206
vis> test_network_acyclic
Network has no combinational cycles
vis> ls
farm_control
hwy_control
sensor
timer
vis> cd hwy_control
vis> print_io
inputs: car_present enable_hwy long_timer short_timer
outputs: enable_farm hwy_light hwy_start_timer
vis> print_latches
hwy_light
vis> flatten_hierarchy
vis> print_network_stats
hwy_control  combinational=45  pi=4  po=3  latches=1  pseudo=0  const=12  edges=68
```

Note that when a node is arrived at for the first time, there is no network for that node until *flatten_hierarchy* is called for that node.

Also *flatten_hierarchy* automatically checks each table in the network for being deterministic (except for pseudo-inputs) and completely specified. Since this checking takes some time, it can be turned off safely using the option *flatten_hierarchy -b*, <u>after</u> a BLIF-MV file has been checked once.

4.1.2 Ordering

The next step towards verification consists of converting this network representation into a functional description that represents the output and next state variables as a function of the inputs and current state variables. We use the BDD (binary decision diagram) and its extension the MDD (multivalued decision diagram) to represent boolean and discrete functions. Before creating the MDDs, it is necessary to order the variables in the support of the MDD. This is accomplished by the *static_order* command, which gives an initial ordering. Networks with combinational cycles cannot be ordered. If the MDD variables have already been ordered, then *static_order* does nothing. To undo the current ordering, reinvoke the command *flatten_hierarchy*. At any stage the current variable ordering can be written out to a file using the *write_order* command.

4.1.3 Computing FSM Information

The *build_partition_mdds* command computes the transition function MDDs. Depending on the partitioning method selected, the MDDs for the combinational outputs (COs) are built in terms of either the combinational inputs (CIs) or some subset of intermediate nodes of the network. The MDDs built are stored in a DAG called a "partition". The vertices of a partition correspond to the CIs, COs, and any intermediate nodes used. Each vertex has a multi-valued function (represented by an MDD) expressing the function of the corresponding network node in terms of the partition vertices in its transitive fanin. Hence, the MDDs of the partition represent a partial collapsing of the network. The *inout* method represents one extreme where no intermediate nodes are used, and *total* represents the other extreme where every node in the network has a corresponding vertex in the partition. If no *method* is specified on the command line, then the value of the flag *partition_method* is used as default (this flag is set by the command *set partition_method*), unless it does not have a value, in which case the *inout* method is used. The partition graph can be printed to a file with the *print_partition* command. Another related command is the *print_partition_stats* command that prints statistics on the partition graph.

The complete set of commands included by *init_verify* are:

1. *flatten_hierarchy*,
2. *static_order*, and
3. *build_partition_mdds*.

```
UC Berkeley, VIS Release 1.0 (compiled 11-Dec-95 at 10:36 AM)
vis> read_blif_mv tlc.mv
Warning: Some variables are unused in model main.
vis> flatten_hierarchy
vis> static_order
vis> build_partition_mdds
vis> print_partition_stats
Method Inputs-Outputs, 8 sinks, 10 sources, 14 total vertices, 78 mdd nodes
```

4.1.4 Advanced Ordering

Dynamic ordering of variables may be enabled and disabled using the *dynamic_var_ordering* command. Dynamic ordering is a technique to reorder the MDD variables to reduce the size of the existing MDDs. The commands *flatten_hierarchy* and *static_order* must be invoked before this command. Available methods for dynamic reordering are *window* and *sift*. Dynamic ordering may be time consuming, but can often reduce the size of the MDDs dramatically.

Dynamic ordering is best invoked explicitly (using the *dynamic_var_ordering -f <method>* option) after the *build_partition_mdds* and *print_img_info* commands. If dynamic ordering finds a good ordering, then you may wish to save this ordering (using *write_order <file>*) and reuse it (using *static_order -s <method> <file>*). With option *dynamic_var_ordering -e <method>* dynamic ordering is automatically enabled whenever a certain threshold on the overall MDD size is reached. Enabling dynamic ordering may slow down the verification, but it can make the difference between completing and not completing a verification task.

```
UC Berkeley, VIS Release 1.0 (compiled 13-Dec-95 at 8:36 AM)
vis> read_blif_mv tlc.mv
Warning: Some variables are unused in model main.
vis> init_verify
vis> print_partition_stats
Method Inputs-Outputs, 8 sinks, 10 sources, 14 total vertices, 78 mdd nodes
vis> write_order
# UC Berkeley, VIS Release 1.0 (compiled 13-Dec-95 at 8:36 AM)
# network name: main
# generated: Wed Dec 13 14:13:57 1995
#
# name               type            mddId vals levs
sensor.rand_choice  pseudo-input        0    2 (0)
timer.state         latch               1    3 (1, 2)
hwy_light           latch               2    3 (3, 4)
car_present         latch               3    2 (5)
car_present$NS      shadow              4    2 (6)
farm_light          latch               5    3 (7, 8)
timer.rand_choice   pseudo-input        6    2 (9)
timer.state$NS      shadow              7    3 (10, 11)
farm_light$NS       shadow              8    3 (12, 13)
hwy_light$NS        shadow              9    3 (14, 15)
vis> dynamic_var_ordering -f sift
Dynamic variable ordering forced with method sift....
vis> print_partition_stats
Method Inputs-Outputs, 8 sinks, 10 sources, 14 total vertices, 70 mdd nodes
vis> write_order
# UC Berkeley, VIS Release 1.0 (compiled 13-Dec-95 at 8:36 AM)
# network name: main
# generated: Wed Dec 13 14:14:20 1995
#
# name               type            mddId vals levs
sensor.rand_choice  pseudo-input        0    2 (0)
timer.state         latch               1    3 (1, 2)
hwy_light           latch               2    3 (3, 6)
farm_light          latch               5    3 (4, 5)
car_present$NS      shadow              4    2 (7)
car_present         latch               3    2 (8)
timer.rand_choice   pseudo-input        6    2 (9)
timer.state$NS      shadow              7    3 (10, 11)
farm_light$NS       shadow              8    3 (12, 13)
hwy_light$NS        shadow              9    3 (14, 15)
vis> write_order tlc.sift
vis> quit
```

```
/projects/vis/vis/mips/bin/vis
UC Berkeley, VIS Release 1.0 (compiled 13-Dec-95 at 8:36 AM)
vis> read_blif_mv tlc.mv
Warning: Some variables are unused in model main.
vis> flatten_hierarchy -b
vis> static_order -s input_and_latch tlc.sift
vis> write_order
# UC Berkeley, VIS Release 1.0 (compiled 13-Dec-95 at 8:36 AM)
# network name: main
# generated: Wed Dec 13 14:34:08 1995
#
# name               type            mddId vals levs
sensor.rand_choice  pseudo-input        0    2 (0)
timer.state         latch               1    3 (1, 2)
hwy_light           latch               2    3 (3, 4)
farm_light          latch               3    3 (5, 6)
car_present$NS      shadow              4    2 (7)
car_present         latch               5    2 (8)
timer.rand_choice   pseudo-input        6    2 (9)
timer.state$NS      shadow              7    3 (10, 11)
farm_light$NS       shadow              8    3 (12, 13)
hwy_light$NS        shadow              9    3 (14, 15)
```

Dynamic ordering moves around binary valued variables, possibly separating a group of variables which encode a single multi-valued variable. Note, however, that the resolution of reading and writing variable ordering files is at the multi-valued variable level, not the bit-level. Therefore when the ordering found by dynamic ordering is read back, the BDD variables which encode the MDD variables do not necessarily occupy the same levels as reported in the file *tlc.sift*. See for example variable `hwy_light` in the example above. The only information that is used from the file *tlc.sift* is the order of the MDD variables in the first column. By editing the file *tlc.sift* any order can be imposed. Given an ordering of MDD variables, BDD variables which encode them are assigned to the first adjacent available levels.

4.2 FSM Traversal and Image Computation

FSM traversal is the core computation in design verification. Efficient traversal requires grouping the MDDs, in a manner optimal for traversal. To traverse the FSM, the present state, input, and next state variables are organized for easy manipulation. All this information is included in an FSM data structure created in the *compute_reach* command. This also invokes traversal of the entire reachable state set of the FSM representing the design, and may be invoked with different verbosity options to get varying amounts of traversal information. On subsequent calls to *compute_reach*, the reachability computation is not reperformed, but statistics can be printed using *-v*.

The reachability computation makes extensive use of image computation. There are several user-settable options that affect the performance of image computation. The documentation for the *set* command lists these options. Use the command *set image_method* to change the image computation method, and then re-initialize verification (starting at the *flatten_hierarchy* command [1]). The *print_img_info* prints current image information. Notice that while *print_partition_stats* prints information on the next state **functions**, *print_img_info* prints information on the next state transition **relations**. The command *print_img_info* creates transition relations from transition functions by clustering several functions together. The result is a partitioned transition relation. It is often a good idea to force dynamic variable reordering (for instance, *dynamic_var_ordering -f sift*) at this point to reorder these relation MDDs. The reachability computation is an optional step of the model checking algorithm; unreachable states may be used as don't cares to minimize the BDD representation.

The following illustrates the command *compute_reach* on the Traffic Light Controller:

```
UC Berkeley, VIS Release 1.0 (compiled 11-Dec-95 at 10:36 AM)
vis> read_blif_mv tlc.mv
Warning: Some variables are unused in model main.
vis> init_verify
vis> compute_reach -v 1
Computing reachable states using the iwls95 image computation method.
Printing Information about Image method: IWLS95
        Threshold Value of Bdd Size For Creating Clusters = 1000
                (Use "set image_cluster_size value " to set this to desired value)
        Verbosity = 0
                (Use "set image_verbosity value " to set this to desired value)
        W1 =  6 W2 = 1 W3 = 1 W4 = 2
                (Use "set image_W? value " to set these to desired values)
        Shared Bdd Size of    1 components is        97
*******************************
Reachability analysis results:
FSM depth =                 8
reachable states =         20
MDD size =                  8
analysis time =             0
```

[1]Whenever a hierarchy is reinitialized, the option *flatten_hierarchy -b* can be used safely for efficiency.

4.3 Specifying Fairness Constraints

Fairness constraints are used to restrict the behavior of the design. Each fairness condition specifies a set of states in the machine, and requires that in any acceptable behavior these states must be traversed infinitely often (i.e., these states must be on a cycle). Such constraints are called "Büchi fairness" constraints. Fairness constraints are stored in fairness files (with extension `.fair` by convention); the syntax for fairness files can be found in `http://www-cad.eecs.berkeley.edu/Respep /Research/vis/doc/packages/read_fairnessCmd.html`. A fairness file is read in by the *read_fairness* command. Active fairness conditions can be displayed by means of *print_fairness*. The *reset_fairness* command is used to reset the fairness constraint to "true"; by default, there is one fairness condition that contains all states.

Fairness constraints remove unwanted behavior from a system. They are a powerful, but dangerous tool, because it is easy to make a faulty system pass wanted properties by a careless use of fairness constraints.

4.4 Language Emptiness

The language of a design is given by sequences over the set of reachable states that do not violate the fairness constraint. If the language is empty, we know that the system does not exhibit any behavior. VIS supports the command *lang_empty* as an alias for model checking the formula $EG\ true$. This is relevant in the context of language containment, where the properties to be verified are also specified as automata and a modified system, consisting of the behavior of the system that does not satisfy the property, is tested for emptiness. Before invoking model checking, *lang_empty* can also be used to ensure that the system is non-trivial. This is pertinent because the fairness constraint specified may make the entire system "unfair", and an empty system passes all universal properties.

VIS produces a debug trace to help the designer understand the cause of the failure. Common corrective actions are the correction of an error in the original system description or addition of fairness constraints.

The language emptiness trace for the Traffic Light Controller example with a fairness constraint is:

```
UC Berkeley, VIS Release 1.0 (compiled 14-Dec-95 at 1:04 AM)
vis> read_blif_mv tlc.mv
Warning: Some variables are unused in model main.
vis> init_verify
vis> read_fairness tlc.fair
vis> print_fairness
Fairness constraints:
!(timer.state=START);
!(timer.state=SHORT);
vis> lang_empty -i
# LE: language is not empty
# LE: generating path to fair cycle
# LE: path to fair cycle:


--State 0:
car_present:NO
farm_light:RED
hwy_light:GREEN
timer.state:START
```

This indicates that there is valid behavior in the system, and an example of this is given; a closed path that begins at the initial state, where no car is present $car\_present : NO$, the farm light is red $farm\_light : RED$, the highway light is green $hwy\_light : GREEN$, and the timer is in its start state $timer.state : START$. From the initial state the machine loops through a fair cycle, which has 8 states, and is described below. Note that this trace is differential for both states and inputs; only variables that have changed in the last step are printed.

```
# LE: fair cycle:


--State 0:
car_present:NO
farm_light:RED
hwy_light:GREEN
timer.state:START

--Goes to state 1:
car_present:YES
timer.state:SHORT

--On input:
sensor.rand_choice:1
timer.rand_choice:1

--Goes to state 2:
timer.state:LONG

--On input:
<Unchanged>

--Goes to state 3:
hwy_light:YELLOW
timer.state:START

--On input:
timer.rand_choice:0

--Goes to state 4:
timer.state:SHORT

--On input:
timer.rand_choice:1

--Goes to state 5:
car_present:NO
farm_light:GREEN
hwy_light:RED
timer.state:START

--On input:
sensor.rand_choice:0
timer.rand_choice:0

--Goes to state 6:
car_present:YES
farm_light:YELLOW

--On input:
sensor.rand_choice:1

--Goes to state 7:
timer.state:SHORT

--On input:
timer.rand_choice:1

--Goes to state 8:
car_present:NO
farm_light:RED
hwy_light:GREEN
timer.state:START

--On input:
```

```
sensor.rand_choice:0
timer.rand_choice:0
```

4.5 Model Checking Operations

4.5.1 Performing Model Checking

The *model_check* command calls model checking in VIS. A description of the syntax of CTL for VIS is presented in `http://www-cad.eecs.berkeley.edu/Respep/Research/vis/doc/ctl/ctl ctl.html`. By convention CTL properties are in a file with extension *.ctl*. The following illustrates the functioning of *model_check* on the Traffic Light Controller example. Note that in this session the fairness constraints are not read in. Debugging error traces is explained in the next section.

```
UC Berkeley, VIS Release 1.0 (compiled 14-Dec-95 at 1:04 AM)
vis> read_blif_mv tlc.mv
Warning: Some variables are unused in model main.
vis> init_verify
vis> model_check -i tlc.ctl


MC: formula passed --- AG(!((farm_light=GREEN * hwy_light=GREEN)))
```

This indicates that the property passed (i.e. the system satisfies the property).

```
MC: formula failed --- AG(((car_present=YES * timer.state=LONG) -> AF(farm_light=GREEN)))
MC: Calling debugger
```

This indicates that the property failed, and gives the following error trace that shows behavior seen in the system that does not satisfy the property.

```
--State
car_present:NO
farm_light:RED
hwy_light:GREEN
timer.state:START

fails AG(((car_present=YES * timer.state=LONG) -> AF(farm_light=GREEN)))
--Counter example is a path to a state where
((car_present=YES * timer.state=LONG) -> AF(farm_light=GREEN)) is false

--State 0:
car_present:NO
farm_light:RED
hwy_light:GREEN
timer.state:START

--Goes to state 1:
car_present:YES
timer.state:SHORT

--On input:
sensor.rand_choice:1
timer.rand_choice:1

--Goes to state 2:
timer.state:LONG

--On input:
<Unchanged>

--State
car_present:YES
```

```
farm_light:RED
hwy_light:GREEN
timer.state:LONG

fails ((car_present=YES * timer.state=LONG) -> AF(farm_light=GREEN))


--State
car_present:YES
farm_light:RED
hwy_light:GREEN
timer.state:LONG

passes (car_present=YES * timer.state=LONG)


--State
car_present:YES
farm_light:RED
hwy_light:GREEN
timer.state:LONG

passes car_present=YES


--State
car_present:YES
farm_light:RED
hwy_light:GREEN
timer.state:LONG

passes timer.state=LONG

--State
car_present:YES
farm_light:RED
hwy_light:GREEN
timer.state:LONG

fails AF(farm_light=GREEN)

--A fair path on which farm_light=GREEN is always false:

--Fair path stem:

--State 0:
car_present:YES
farm_light:RED
hwy_light:GREEN
timer.state:LONG

--Goes to state 1:
hwy_light:YELLOW
timer.state:START

--On input:
sensor.rand_choice:1
timer.rand_choice:0

--Fair path cycle:

--State 0:
car_present:YES
farm_light:RED
hwy_light:YELLOW
timer.state:START

--Goes to state 1:
<Unchanged>
```

```
--On input:
sensor.rand_choice:1
timer.rand_choice:0
```

This is the end of the debug trace for this CTL formula. The command *model_check* continues with the next formula.

```
MC: formula failed --- AG(AF(hwy_light=GREEN))
MC: Calling debugger
```

This indicates that the property failed, and it is followed by an error trace that shows behavior seen in the system that does not satisfy the property. To save space, we omit the error trace.

```
MC: formula passed --- !(AG((car_present=YES -> AF(farm_light=GREEN))))
```

This indicates that the property passed (i.e. the system satisfies the property).

4.5.2 Debugging for Model Checking

If model checking or language emptiness checks fail, VIS reports the failure with a counterexample, i.e., an error trace of sample "bad" behavior (i.e., behavior seen in the system that does not satisfy the property - for model checking, or valid behavior seen in the system - for language emptiness). This is called the "debug" trace. Debug traces list a set of states that are on a path to a fair cycle and fail the CTL formula.

In the previous section, the second and third properties fail during model checking. This may be rectified by reading in the fairness constraints previously described for the Traffic Light Controller example. If the fairness constraints are read in, the valid behavior is restricted and these properties pass. In particular, the fairness constraint `!(timer.state=START)` disallows behavior, where the system stays forever in the state:

```
car_present:YES
farm_light:RED
hwy_light:YELLOW
timer.state:START

--On input:
sensor.rand_choice:1
timer.rand_choice:0
```

More precisely, the fairness constraint disallows behavior, where there is a car in the farm road, but the timer is stuck in its initial state, by forcing the timer to progress in finite time to the next state.

```
UC Berkeley, VIS Release 1.0 (compiled 11-Dec-95 at 10:36 AM)
vis> read_fairness tlc.fair
vis> model_check tlc.ctl


MC: formula passed --- AG(!((farm_light=GREEN * hwy_light=GREEN)))

MC: formula passed --- AG(((car_present=YES * timer.state=LONG) -> AF(farm_light=GREEN)))

MC: formula passed --- AG(AF(hwy_light=GREEN))

MC: formula passed --- !(AG((car_present=YES -> AF(farm_light=GREEN))))
```

4.5.3 Checking Invariants

An important class of CTL formulas is *invariants*. These are formulas of the form $AG\ f$, where f is a quantifier-free formula. The semantics of $AG\ f$ is that f is true in all reachable states. The command *check_invariant* implements an algorithm that is specialized for these formulas. In the following example, f is the formula

```
!((farm_light = GREEN) * (hwy_light = GREEN));
```

contained in the file *tlc.invar*.

```
UC Berkeley, VIS Release 1.0 (compiled 13-Dec-95 at 8:36 AM)
vis> read_blif_mv tlc.mv
Warning: Some variables are unused in model main.
vis> init_verify
vis> check_invariant tlc.invar

INV: formula passed --- !((farm_light=GREEN * hwy_light=GREEN))
```

4.5.4 Advanced Model Checking: Abstraction and Reduction

When performing model checking and checking invariant properties, one can use the reduce option *-r*, to perform model checking on a "pruned" FSM, i.e., one where parts that do not affect the formula (directly or indirectly) have been removed.

This mechanism can be combined with the abstraction mechanism available through the command *flatten_hierarchy <file>*. *<file>* contains the names of variables to abstract. For each variable `x` appearing in *<file>*, a new primary input node named `x$ABS` is created to drive all the nodes that were previously driven by `x`. Hence, the node `x` will not have any fanouts; however, `x` and its transitive fanins will remain in the network. Abstracting a net effectively allows it to take any value in its range, at every clock cycle. This mechanism can be used to perform manual abstractions.

We show an example, where the file `tlc.abstract` contains the variable `timer.start`. By abstracting `timer.start`, the timer module is disconnected from the rest of the Traffic Light Controller.

Then we perform model checking of the CTL property read from the file `tlc.reduce.ctl`:

```
AG((timer.state = START) -> AF (timer.state = LONG));
```

This property refers only to the timer module. Since the timer has been disconnected, the rest of the system can be pruned away when testing this property. As expected this property fails, since no fairness constraint has been read in.

```
UC Berkeley, VIS Release 1.0 (compiled 15-Dec-95 at 2:18 PM)
Sourcing .visrc of Tiziano
vis> read_blif_mv tlc.mv
Warning: Some variables are unused in model main.
vis> flatten_hierarchy tlc.abstract
vis> static_order
vis> build_partition_mdds
vis> model_check -i -r tlc.reduce.ctl

MC: formula failed --- AG((timer.state=START -> AF(timer.state=LONG)))

MC: Calling debugger

--State
car_present:NO
farm_light:RED
hwy_light:GREEN
timer.state:START
```

```
fails AG((timer.state=START -> AF(timer.state=LONG)))
since (timer.state=START -> AF(timer.state=LONG)) is false at this state

--State
car_present:NO
farm_light:RED
hwy_light:GREEN
timer.state:START

fails (timer.state=START -> AF(timer.state=LONG))


--State
car_present:NO
farm_light:RED
hwy_light:GREEN
timer.state:START

passes timer.state=START

--State
car_present:NO
farm_light:RED
hwy_light:GREEN
timer.state:START

fails AF(timer.state=LONG)

--A fair path on which timer.state=LONG is always false:

--Fair path stem:

--State 0:
car_present:NO
farm_light:RED
hwy_light:GREEN
timer.state:START

--Fair path cycle:

--State 0:
car_present:NO
farm_light:RED
hwy_light:GREEN
timer.state:START

--Goes to state 1:
<Unchanged>

--On input:
sensor.rand_choice:0
timer.rand_choice:0
```

In this particular example, the same effect of "restricted" model checking can be obtained by changing (using the *cd* command) to the timer node and performing model checking. When at the timer node, the inputs to timer from the rest of the system are considered free inputs. Notice that the names of variables in the CTL property in the file `tlc.reduce.ctl` must be revised as follows:

```
AG((state = START) -> AF (state = LONG));
```

since the convention for names is to drop the current node and all nodes above from the namepath.

```
UC Berkeley, VIS Release 1.0 (compiled 14-Dec-95 at 1:04 AM)
Sourcing .visrc of Tiziano
vis> read_blif_mv tlc.mv
Warning: Some variables are unused in model main.
vis> cd timer
```

```
vis> init_verify
vis> model_check tlc.reduce.ctl

MC: formula failed --- AG((state=START -> AF(state=LONG)))
```

However, there are more complex situations that cannot be emulated so simply.

4.6 Combinational and Sequential Equivalence

In VIS it is also possible to check the equivalence of two networks. The command *comb_verify* verifies the combinational equivalence of two flattened networks. In particular, any set of functions (the roots), defined over any set of intermediate variables (the leaves), can be checked for equivalence between two networks. Roots and leaves are subsets of the nodes of a network, with the restriction that the leaves should form a complete support for the roots. The correspondence between the roots and the leaves in the two networks is specified in a file. The default option assumes that the roots are the combinational outputs and the leaves are the combinational inputs. Two networks are declared combinationally equivalent iff they have the same outputs for all combinations of inputs and pseudo-inputs. An important usage of *comb_verify* is to provide a sanity check when using SIS to re-synthesize portions of a network, as explained in Chapter 5.

The command *seq_verify* tests the sequential equivalence of two networks. In this case the set of leaves has to be the set of all primary inputs. This produces the constraint that both networks should have the same number of primary inputs. The set of roots can be an arbitrary subset of nodes. Moreover, no pseudo-inputs should be present in the two networks being compared. Sequential verification is done by building the product finite state machine. The command verifies whether any state, where the values of two corresponding roots differ, can be reached from the set of initial states of the product machine. If this happens, a debug trace is provided.

4.7 Simulation

Simulation, although not "formal verification", is an alternate method for design verification. After the command *build_partition_mdds* is invoked, the network can also be simulated. In VIS we provide internal simulation of the BLIF-MV description generated by VL2MV, via the *simulate* command. Thus, VIS encompasses both formal verification and simulation capabilities. *simulate* can generate random input patterns or accept user-specified input patterns.

```
UC Berkeley, VIS Release 1.0 (compiled 15-Dec-95 at 10:24 PM)
vis> read_blif_mv tlc.mv
Warning: Some variables are unused in model main.
vis> init_verify
vis> simulate -n 10
# UC Berkeley, VIS Release 1.0 (compiled 15-Dec-95 at 10:24 PM)
# Network: main
# Simulation vectors have been randomly generated


.inputs  sensor.rand_choice timer.rand_choice
.latches car_present farm_light hwy_light timer.state
.outputs
.initial NO RED GREEN START

.start_vectors

# sensor.rand_choice timer.rand_choice ; car_present farm_light hwy_light timer.state ;

0 0 ; NO  RED    GREEN  START ;
1 1 ; NO  RED    GREEN  START ;
0 0 ; YES RED    GREEN  SHORT ;
```

```
1 0 ; NO  RED    GREEN  SHORT ;
1 1 ; YES RED    GREEN  SHORT ;
0 1 ; YES RED    GREEN  LONG  ;
0 1 ; NO  RED    YELLOW START ;
0 0 ; NO  RED    YELLOW SHORT ;
0 0 ; NO  GREEN  RED    START ;
1 0 ; NO  YELLOW RED    START ;
# Final State : NO  YELLOW RED    START
vis> cd farm_control
vis> simulate -n 10
There is no network. Use flatten_hierarchy.
vis> init_verify
vis> simulate -n 10
# UC Berkeley, VIS Release 1.0 (compiled 15-Dec-95 at 10:24 PM)
# Network: farm_control
# Simulation vectors have been randomly generated


.inputs  car_present enable_farm long_timer short_timer
.latches farm_light
.outputs enable_hwy farm_light farm_start_timer
.initial RED

.start_vectors

# car_present enable_farm long_timer short_timer ; farm_light ; enable_hwy farm_light farm_start_timer

NO  1 0 0 ; RED    ; 0 RED    1
YES 1 1 1 ; GREEN  ; 0 GREEN  1
NO  1 0 1 ; YELLOW ; 1 YELLOW 0
YES 0 0 0 ; RED    ; 0 RED    0
NO  1 1 0 ; RED    ; 0 RED    1
NO  1 1 1 ; GREEN  ; 0 GREEN  1
YES 1 1 1 ; YELLOW ; 1 YELLOW 0
NO  0 1 0 ; RED    ; 0 RED    0
NO  0 0 0 ; RED    ; 0 RED    0
YES 0 1 0 ; RED    ; 0 RED    0
# Final State : RED
```

Any level of the specified hierarchy may be simulated. The user may traverse the hierarchy to reach the relevant level via the *cd* command. The *init_verify* command must be called to set up the appropriate internal data structures before simulation.

Chapter 5

Synthesis in VIS

VIS can interact with SIS in order to optimize the existing logic. There are two possible goals/scenarios:

1. Synthesis for verification.
   Synthesis can be used to optimize the logic that represents the system, for simpler verification.

2. Front-end to synthesis.
   Files described in Verilog and compiled into *blif_mv* (using VL2MV or another tool) can be synthesized by using VIS and SIS together.

A key fact is that only the current level of the hierarchy is sent to SIS, and not the subtree rooted at the current node. [1] Modules at a lower level are treated as external and the boundary variables are carefully preserved, by reintegrating their multi-valued status after the optimization step in SIS (SIS requires that boundary variables are completely encoded, i.e., are binary variables).
Caveat To prevent that a signal (possibly referred to in a CTL property) is optimized away during synthesis, declare it as an output of a module.

In the current version, only combinational logic is sent to SIS: latches are cut away from the module sent to SIS and they are reincorporated when the design is read back into VIS. Therefore we cannot take advantage of sequential optimizations in SIS, either at the level of a completely encoded sequential network or of a symbolic state table. The boundaries between modules are established when the initial hierarchy is described, and they are rigid in the sense that optimizations can never bridge them, but only operate within them. Notice that there is a way to replace a subtree of the hierarchy with another one by using *read_blif_mv -r*; this feature could be used to change boundaries in the original specification.

5.1 Writing and Reading from SIS

VIS communicates with SIS via the *write_blif* and *read_blif* commands.

Operations performed by *write_blif* are:

1. All variables are encoded, i.e., values of multi-valued variables are replaced by binary vectors. For variables at the boundary with modules at different levels of the hierarchy the encoding assignments are stored into a file with extension `.enc`, so that it is possible to reintegrate the multi-valued boundaries between modules when coming back to VIS.

2. All unspecified input combinations in the tables are specified by assigning zero code vectors as outputs. Default constructs in the specification of tables are handled appropriately.

[1] One would need a flattening routine different from the one which starts the verification flow already in VIS, and such a routine to flatten for synthesis is not yet available.

3. Nondeterministic tables are determinized by adding pseudo-inputs. As a result a file with extension `.blif` is created that can be read and optimized by SIS. SIS must be invoked outside of VIS by means of a different shell. All SIS operations to optimize combinational logic can be applied.

In summary, *write_blif* scans all the tables of a given node in the hierarchy and encodes all symbolic variables, determinizes the tables by adding pseudo-inputs, and resolves incomplete specification by associating unspecified input combinations to outputs encoded by zero binary vectors.

Operations performed by *read_blif* are:

1. Restore the symbolic values of multi-valued I/O variables of the node being read in. This is done using the information in the file with extension `.enc` (e.g., *read_blif -e model.enc s-sim.blif*), which was written out during the *write_blif* process.

2. Replace in the hierarchy the old node with the new node.

5.2 Flow of Operations for Synthesis

The typical flow of operations of synthesis for verification is:

- *read_blif_mv*
- *write_blif*
- optimization by SIS
- *read_blif*
- *init_verify*
- suite of verification operations

The typical flow of operations for direct synthesis is:

- *read_blif_mv*
- *write_blif*
- optimization by SIS
- *read_blif*

It is possible to verify that after optimization with SIS the new global network (where the node returned from SIS is plugged back in the original network) is equivalent to the old global network, by using the command *comb_verify* that checks combinational equivalence of networks. Combinational equivalence can be checked at each level of the network hierarchy, from root to leaves. Before applying *comb_verify*, the command *init_verify* must be invoked.

5.3 Example of Synthesis of Traffic Light Controller

The following script demonstrates the path from VIS to SIS and back. We have chosen to optimize the network of the leaf `farm_control`. We verify that the initial global network and the new network, after replacement of the network in the leaf `farm_control` by the one optimized by SIS, are combinationally equivalent. The script used to run SIS (in a different shell) is shown too. Experiments report big savings in literals for the optimized modules, since the BLIF-MV files generated by VL2MV have a lot of redundancy.

```
UC Berkeley, VIS Release 1.0 (compiled 11-Dec-95 at 10:36 AM)
vis> read_blif_mv tlc.mv
Warning: Some variables are unused in model main.
vis> init_verify
vis> ls
hwy_control
sensor
timer
farm_control
vis> print_network_stats
main  combinational=142  pi=0  po=0  latches=4  pseudo=2  const=40  edges=206
vis> cd farm_control
vis> write_blif farm_control.blif
Writing encoding information to farm_control.enc
vis> read_blif -e farm_control.enc farm_control.opt.blif
Warning: Some variables are unused in model farm_control[0].
vis> cd ..
vis> init_verify
vis> comb_verify tlc.mv
Networks are combinationally equivalent.
vis> print_network_stats
main  combinational=132  pi=0  po=0  latches=4  pseudo=2  const=34  edges=186
```

```
sis> read_blif farm_control.blif
Warning: network `farm_control', node "[1]0" does not fanout
Warning: network `farm_control', node "[5]0" does not fanout
Warning: network `farm_control', node "[11]0" does not fanout
sis> print_stats
farm_control    pi=18   po= 6   nodes= 62       latches= 0
lits(sop)= 709  lits(fac)= 419
sis> source script.rugged
sis> print_stats
farm_control    pi=18   po= 6   nodes= 24       latches= 0
lits(sop)=  34  lits(fac)=  34
sis> write_blif farm_control.opt.blif
```

In the previous example, the command *init_verify* has been given only in order to do *print_network_stats* before logic synthesis, to compare the networks before and after optimization by SIS.

Appendix A

Commands in VIS

A.1 List of Commands in VIS

The following list contains a one line summary of all the commands available within VIS. The list can also be found in `http://www-cad.eecs.berkeley.edu/Respep/Research/vis/doc/packages/cmdIndex.html`. Fig. A.1 graphically illustrates the suite of commands available within VIS, and their dependencies. A command cannot be executed before its predecessors (unless the predecessor is also a successor). Default aliases are defined, type *alias* to list them.

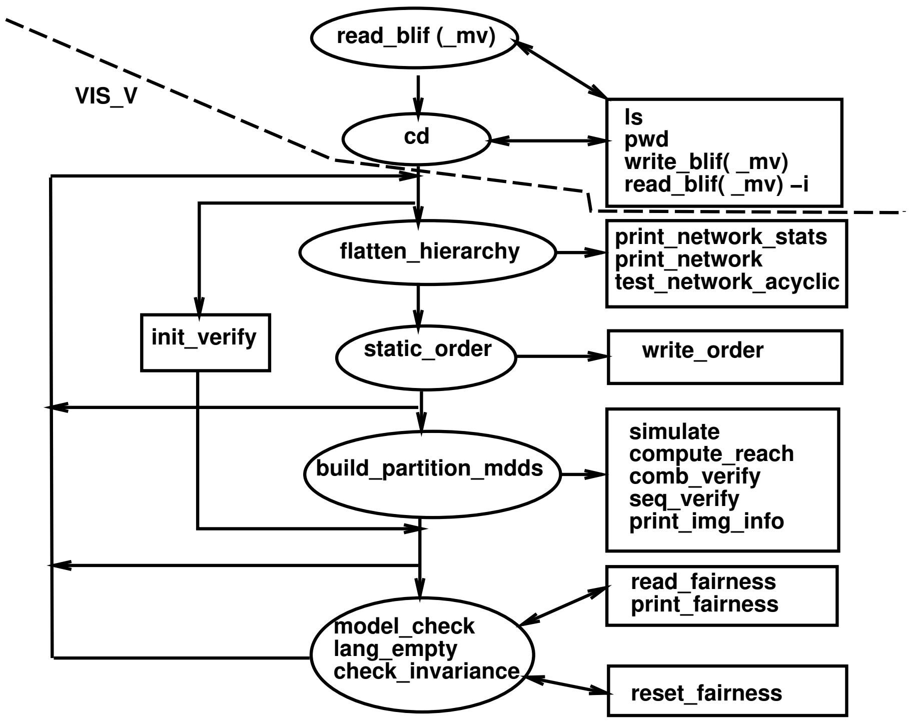

Figure A.1: A Flow Chart of Commands in VIS.

1. alias: provide an alias for a command
2. build_partition_mdds: build a partition of MDDs for the current network
3. cd: change the current node

4. check_invariant: checks all states reachable in flattened network satisfy specified invariants
5. comb_verify: verifies the combinational equivalence of two networks
6. compute_reach: compute the set of reachable states of the FSM
7. dynamic_var_ordering: control the application of dynamic variable ordering
8. echo: merely echoes the arguments
9. flatten_hierarchy: create a flattened network
10. help: provide on-line information on commands
11. history: a UNIX-like history mechanism inside the VIS shell
12. init_verify: create and initialize a flattened network for verification
13. lang_empty: performs BDD based check of language emptiness under Buchi fairness
14. ls: list all the child nodes at the current node
15. model_check: performs BDD based fair CTL model checking on a network
16. print_bdd_stats: print the BDD statistics for the flattened network
17. print_fairness: print the fairness constraints of the flattened network
18. print_hierarchy_stats: print the statistics of the current node
19. print_img_info: print information about the image method currently in use
20. print_io: print the names of inputs/outputs in the current node
21. print_latches: print the names of latches in the current node
22. print_models: list all the models and their statistics
23. print_network: print the flattened network
24. print_network_stats: print statistics about the flattened network
25. print_partition: write a file in the "dot" format describing the partition graph
26. print_partition_stats: print statistics about the partition graph
27. pwd: print out the full path of the current node from the root node
28. quit: exit VIS
29. read_blif: read a blif file
30. read_blif_mv: read a blif-mv file
31. read_fairness: read a set of fairness constraints
32. read_verilog: read a verilog file
33. reset_fairness: reset the fairness constraints

34. seq_verify: verifies the sequential equivalence of nodes in two networks
35. set: set an environment variable
36. simulate: simulate the flattened network
37. source: execute commands from a file
38. static_order: order the MDD variables of the flattened network
39. test_det_and_comp_spec: test if the outputs are completely specified and deterministic
40. test_network_acyclic: determine whether the network is acyclic
41. time: provide a simple elapsed time value
42. unalias: removes the definition of an alias
43. unset: unset an environment variable
44. usage: provide a dump of process statistics
45. which: look for a file called name
46. write_blif: determinize, encode and write an hnode to a blif file
47. write_blif_mv: write a blif-mv file
48. write_order: write the current order of the MDD variables of the flattened network

Bibliography

[1] D.E. Thomas, P.R. Moorby. The Verilog Hardware Description Language. Kluwer Academic Publishers, Nowell, Massachusetts, 1991.

[2] S.-T. Cheng. Compiling Verilog into automata. Tech. Rep. UCB/ERL M94/37, May 1994.

[3] F. Balarin, and R. Brayton, and S-T. Cheng, and D. Kirkpatrick, and A. Sangiovanni-Vincentelli. A Methodology for Formal Verification of Real-Time Systems. Tech. Rep. UCB/ERL M95/11, February 1995.

[4] E.M. Sentovich et al. SIS: a system for sequential circuit synthesis. Tech. Rep. M92/41, May 1992.

[5] C. Mead, L. Conway. Introduction to VLSI systems. Addison-Wesley, 1980.

[6] R. K. Brayton et al. HSIS: A BDD based system for formal verification. Proc. of Design Automation Conference, 1994.

[7] E. Clarke, and O. Grumberg, and K. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. Proc. of Design Automation Conference, 1995.