HSIS

Berkeley CAD Group in Verification

June 30, 1994

1 Introduction

HSIS is a prototype system for formal design verification. The HSIS system allows a user to specify a process or system (including its environment) in Verilog and a set of properties described as automata in Verilog or by CTL formulae. HSIS can then be used to formally prove that the process has the specified properties. If a property is not satisfied by the process, then HSIS prints out an error trace giving an example of a behavior that does not satisfy the property. The error trace is designed to be succinct so that it is easier for the user to find his error.

The HSIS system is designed to grow and evolve. It should serve two functions, first as a prototyping system where new ideas can be implemented and tried on realistic problems, and as a system for industrial use with state-of-the-art algorithms available. In this sense it is similar to SIS which has evolved over many years and has been supported and improved by the Berkeley CAD group. Some of the advantages of HSIS are,

- contains state-of-the-art algorithms
- allows properties to be stated with either fair CTL formulas or ω-automata
- continuing support provided over many years (e-mail errors and comments to: *hsis@ic.eecs.berkeley.edu*)
- free to institutions and industries (anonymous ftp from berkeley - send mail to *hsis@ic.eecs.berkeley.edu* requesting information)
- large user community (hopefully)
- uses common HDL language Verilog, so that simulation of source is straight-forward. A large synthesis subset of Verilog is supported through our Verilog compiler (vl2mv).
- enhanced debugging capabilities via short error traces and early failure detection.
- intermediate format BLIF-MV allows input from other languages and example exchanges with other institutions.

The present version can be regarded as a beta test version. While our development is ongoing, we are releasing this version, which is reasonably operational, to obtain as much feedback as possible from users who are somewhat experienced in formal verification techniques or from users who have access to interesting and complex examples and are willing to share these with us, or from people who want to begin to play around with the concepts of formal design verification.

This document contains the following sections:

1. introduction,
2. a few caveats for the potential user,
3. a brief discussion of "fairness constraints",
4. an example of formal verification of a set of properties on a simple dynamical system to provide some insight on how to use HSIS effectively, and
5. a list of documentation items that are available.

The entire system with examples, documentation and source code is available by anonymous ftp as discussed in the associated document. For more information, send a message to "hsis@ic.eecs.berkeley.edu".

2 Caveats

1. As pointed out in the introduction, HSIS is at an early stage of development. As such, we cannot guarantee that it will work properly on all the problems that a user may try. We will, in the "Berkeley tradition", do our best to support the use of the system by fixing "fatal" bugs as soon as possible. To expedite the bug-fixing process, we request that you please send comments and well documented bug reports by e-mail to the following address:

   hsis@ic.eecs.berkeley.edu

   Also, since HSIS is based on BDD's which can become very large in some cases, we suggest that you start out by trying HSIS on some smaller examples first. Variable ordering can affect the size of the BDD's, so if you experience memory problems, we suggest you look into some of the various variable ordering methods included in this release. (Do "help set" for a description).

2. Research on formal design verification techniques is very much in its adolescence. Many theoretical results are available, but only a few software systems have been recently developed that can be used for "real-life" designs. Even for those systems, there are examples that cannot be verified formally due to the potential explosion in memory usage and/or CPU time for some examples. HSIS is no exception. HSIS is based on implicit computations using BDD's. [1] It is known that BDD's are very sensitive to the ordering given to the variables. We have developed automatic ordering methods, but still we may have examples in which the size of

[1] We provide two different BDD packages that have somewhat different capabilities. The most efficient one was developed by David Long at CMU now at AT&T Bell Laboratories) which is the default version. We thank him for his continued support.

the BDD becomes too large. We have recently included improved algorithms to eliminate some of these problems. In particular, for variables which are parts of data-paths where arithmetic operations are applied, we know that the associated bits should be interleaved in the ordering. An ordering option "global_interleaved" should be tried for these types of examples. Additionally, we have *included* a manual ordering option for the user who may have some insight into a good ordering. We include the ability to write out an ordering from HSIS, so that the next time the same system is tried, the old ordering can be used and the ordering process can be by-passed. However, some problems will still be too big, so that it is recommended to try HSIS on problems with mostly control type variables with data-paths abstracted to only a few bits.

Additionally, dynamic reordering is an option controlled by the command dro_on (dro_off) which may be used as a last option. However, so far we have not had much luck with dynamic ordering turned on.

3. At the moment, not all the capabilities of fairness constraints are supported. In particular, negative fair edges for automata, and positive fair edges for processes are not supported. For an explanation, see the PIF document.

3 Fairness Constraints and Acceptance Conditions

HSIS allows the user to specify properties in two different ways, by ω-automata and computation tree logic (CTL) formulae. The properties specified by ω-automata consist of a transition structure and some *acceptance conditions* which define the accepted set of behaviors allowed by the property. In addition, we allow fairly general *fairness constraints* to make it easy to restrict the set of behaviors allowed in a model.

Since fairness constraints and acceptance conditions have been a source of confusion to many people learning about formal design verification for the first time, in this section we try to make it a little more intelligible.

The important point to distinguish between fairness constraints for the system and acceptance conditions for property automata is that for the former we *exclude* behaviors described by fairness constraints, and for the latter we *accept* only those behaviors described by the acceptance conditions. Sometimes the term fairness constraint is also used to denote acceptance conditions for the property automata but we will use fairness and acceptance terms separately. Depending on whether these constraints are on the system model or on a property automaton, they should be interpreted accordingly.

3.1 Fairness Constraints for the System

These are called fairness constraints for historical reasons. Generally, for one reason or another, we may want to describe a process by describing it in two parts, a state transition structure and a set of constraints. The first generally will have too much behavior in it. For example, in the ping_pong example described below, we may describe a ping pong game where the time of flight is variable. So we introduce an arbitrary delay from the time a ball is hit until the opponent hits it back. However, we insist on realism, hence we want to disallow any behavior where the ball stays in the air forever. We do this by imposing a set of external constraints. We can exclude behavior in two ways, positive and negative. Negative constraints directly describe behaviors we want to exclude— we exclude behaviors which satisfy *any* negative constraint. Positive constraints can be complemented and seen as negative ones, i.e. we exclude all behaviors that do *not* satisfy any positive constraints. For the system specification, it is expected that only negative constraints will be used most of the time.

Thus fairness constraints are simply an easy way of refining a given set of behaviors to obtain precisely the set we want. It is worthwhile to comment that if your process describes a piece of hardware you intend to build, then no fairness constraints are necessary for that, since you would not build something with extra behavior you did not want. But in describing, for example, an environment (which only describes the environment in which you intend to operate your hardware), it may be easier to first describe an abstract transition structure that has too much behavior and then exclude some of it with constraints.

For example, suppose a compiler is intended for a computer with the behavior where instruction BRANCH is never followed by STORE. One way to describe this is with an automaton with three states A,B,C. In addition, each state produces an output which is a random instruction (nondeterministic Moore machine). The automaton starts in state A and transits to B whenever a BRANCH is generated. In B, if a STORE is generated it transits to C and stays there forever; if a BRANCH is generated in B again it stays in B; else it goes back to A.

Without any constraint, the behavior of this, in terms of a stream of instructions, is a random stream. However, if we specify a "negative fairness" constraint of the form { C }, this will exclude any behavior that goes to state C of our automaton. The behavior remaining is precisely what we wanted, i.e. no BRANCH is ever followed by STORE.

3.2 Acceptance Conditions for Property Automata

Automata are used to describe properties, i.e. the set of all behaviors that satisfy some given property, e.g. the set of all behaviors such that no two components write to a bus at the same time. In thinking about a property we can first think of a "monitor" or observer which we create just for convenience to focus on the states and signals of a process that are relevant to the property. It is like an assertion statement often used in programming languages. This monitor is a state transition structure, usually a very simple one, with no outputs. The next thing we need is the "acceptance" conditions which are a set of conditions on the behavior of the monitor. Those behaviors which satisfy *any* of these conditions are acceptable. Since these are usually thought of as acceptance conditions, these conditions are typically positive fairness constraints, but not necessarily so.

Note that the transition structure of the automaton was created merely for the convenience of easily describing the behavior we want. It can have as few or as many states as we like. Its transitions and states simply record sequences of events that are significant for the behavior or property we want to describe.

Now suppose, on the other hand, we have built a compiler and want to see if it has the property that it never generates a BRANCH followed immediately by a STORE. Then the same transition structure as above can be used, except "instruction" is output by our compiler and not by random outputs from the states A,B,C as before. Usually for automata, we think of them in terms of behaviors that we want to **include**. The behaviors we want are described by "including" {A,B}, stating that all acceptable behaviors, ultimately get into the set of states {A,B} and stays in this set forever. This is a positive acceptance condition. Positive acceptance conditions are typical for automata. However, there are cases where it is easier to describe behaviors to be excluded - in this example we could exclude {C} with a negative acceptance condition. In general, there can be simultaneously both positive and negative acceptance conditions.

For a complete description of what type of fairness constraints and acceptance conditions are allowed and how to describe them, see the PIF documentation.

We will see more about how to use and specify these constraints and conditions in the example of the next section.

4 Ping_Pong Example

As mentioned already, HSIS is a verification system which is set up to model a process described in Verilog (actually, other languages are acceptable as long as there is an appropriate compiler to BLIF-MV), and to state and prove properties about this process. The process to be described should include a description of its environment, also modeled in Verilog as just another component. We require that the overall system or process being described be *closed*, i.e. no inputs are provided nor outputs created. Since processes of interest do have inputs and outputs we remark that a closed system can be obtained by describing an *environment* that encloses the given process[2]. The environment reacts to the outputs of the process and generates the inputs needed by the process. The environment can be as simple as a single state process with no inputs and random outputs, outputs that serve as the inputs to the process (the same as the environment we would obtain with the command "close_system"). The properties we wish to verify can be expressed in two ways: automata described by Verilog, or by CTL formulas which are input from a .ctl file. The entire Verilog language is not available. The language is limited to a large subset that includes almost all synthesizable constructs. It has also been extended to allow nondeterminism, usually necessary to model the environment compactly, as well as extended to allow symbolic variables for convenience and ease of debugging.

The best way to understand HSIS and its capabilities is by example. Below, we illustrate the more important points with the example, PING-PONG.

4.1 The Process

This system models a ping pong game with two balls. The header to ping_pong.v gives an explanation of how the system functions.

`file ping_pong.v:`

[2]IN HSIS, we include the ability to read in a BLIF file which has a similar but not exactly the same syntax as BLIF-MV. Most BLIF files have inputs and outputs, so we include an HSIS command "close_system" which will close the system by creating an environment which provides non-deterministic inputs. This can also be used on a BLIF-MV file if the system is not closed.

```
/*
 * This code implements a game of ping pong (table tennis) between 2 players, with
 * a slight twist: each player puts a ball into play at the same time.  There can
 * be an indefinite delay from the time one player hits a ball until the time that
 * the other player returns it.  If both balls are in play, then both players
 * always hit each ball at the same instant.
 *
 * While both balls are in play, exactly one player may choose not to return his
 * ball.  This leaves one ball in play.  If just one ball is in play, the ball
 * should remain in play forever.
 *
 * There are two properties that should hold:
 * 1) It's never the case that no balls are in play.
 * 2) Each player will hit a ball infinitely often (i.e. at any point in time,
 *    each player will hit a ball sometime in the future).
 */

typedef enum {BALL, NO_BALL} player_status;
typedef enum {HIT, NO_HIT} action_type;

module ping_pong(clk);
input clk;

/* signal declarations */
action_type wire action_A, action_B;
player_status wire state_A, state_B;

/* the ping pong players */
player player_A(clk, action_B, action_A, state_A);
player player_B(clk, action_A, action_B, state_B);

endmodule


/*
 * Two state process.  If the player doesn't have the ball (state NO_BALL) and the
 * opponent hits the ball, then go to state BALL.  If the player has the ball, then
 * if he doesn't hit the ball, then he remains in state BALL.  If he hits the ball,
 * and the opponent doesn't also hit the ball, then go to state NO_BALL.
 */
module player(clk, opponent, me, state);
input clk;
```

```
input opponent;
output me;
output state;

action_type wire opponent, me;
player_status reg state;

initial state = BALL;

/* set the value of the output of the player */
assign me = (state == BALL) ? $NDset(@(posedge clk), HIT, NO_HIT) : NO_HIT;

always @(posedge clk) begin
    case(state)
        BALL:
            begin
            if ((me == HIT) && (opponent == NO_HIT))
                state <= NO_BALL;
            end

        NO_BALL:
            begin
            if (opponent == HIT)
                state <= BALL;
            end

        default:;
        endcase;
end
endmodule
-------------------------------------------
end of file ping_pong.v
```

4.2 Properties and Fairness Constraints

What has been described in the file *ping_pong.v* is a process composed of two components, the two players, which are instances of the module "player". Now we want to prove some properties about this process. We will do this in two different ways, first, by describing an automaton representing each property, and second, by CTL formulae. In each case, we will need to also describe what are called *fairness constraints*.

It is a good idea to keep the following rules in mind. First, **negative** fairness constraints give those behaviors which we want to **exclude** from the set of behaviors we are looking for, i.e. if a behavior satisfies any one of these constraints then that behavior should be excluded. **Positive** fairness constraints describe those behaviors that we want to **include**, i.e. we include only behaviors which satisfy all of the positive fairness constraints. In general, for both kinds of fairness, we treat the constraints in exactly the same way but just interchange the words "include" and "exclude".

Generally there are two reasons to impose "fairness" constraints. First, they describe the acceptable behavior of a property automaton and so these behaviors are **included** with positive fairness constraints imposed on the automaton. We refer to these fairness constraints as acceptance conditions. Second, fairness constraints describe behavior that, even though allowed by the staansition structure of our process model, we don't want to consider; an example is a behavior that is trapped in one state forever. We therefore exclude it with a negative fairness constraint. Although, generally positive constraints are used for automata, and negative ones for processes, we do not make this restriction. Whichever way it is most convenient, for either the processes or automata, is allowed. Just remember that positive ones include behaviors, and negative ones exclude behaviors. We will see examples of both of the use of both types of constraints below.

4.3 Automata Properties for Ping_Pong

The two properties we want to verify are:

1. *There is always at least one ball in play.* We first describe an automaton in Verilog for this:

```
file prop1.v
------------------

/*
 * If it's ever the case that neither player has a ball, then go to state BAD
 * and stay there.
 */
typedef enum {BALL, NO_BALL} player_status;
typedef enum {GOOD, BAD} prop1_status;

module never_no_balls(clk, state_A, state_B);
input clk;
input state_A;
input state_B;

player_status wire state_A, state_B;
prop1_status reg state;

initial state = GOOD;

always @(posedge clk) begin
    case(state)
        GOOD:
            begin
```

```
            if ((state_A == NO_BALL) && (state_B == NO_BALL))
                state = BAD;
            end

        BAD:
            begin
            state = BAD;
            end

        default:;
        endcase;
end
endmodule
--------------------
end of file prop1.v
```

We use *vl2mv* to compile this into BLIF-MV:

```
vl2mv prop1.v
```

which creates a file prop1.mv.

The Verilog file above just describes a transition system, but no acceptance conditions (everything is accepted). To make this describe the property *there is always at least one ball in play* we need to say that a good behavior must eventually get into state GOOD and stay there forever. To describe such fairness constraints, the file *prop1.pif* is used:

```
file prop1.pif
----------------------------------
.include process-fairness.pif

.properties

# property 1
# Begin instantiation of automaton with name nnb. We can give it the same
# name as the Verilog module name (never_no_balls) or abbreviate it as nnb.
# The name nnb will be used later to refer to this property

.automaton nnb

# include blif-mv file obtained from Verilog description

.blifmv prop1.mv
```

```
# make connections to the process ping_pong(formal = actual). They are the
# same name in this case.

.connections (state_A = state_A, state_B = state_B)

# begin fairness constraints on this automaton.
# created above. Since this is an automaton, the fairness constraints
# are positive, since we are describing good behavior with them.

.state st1 = state:GOOD
.posfair
.subsets {st1}
.endfair
.endautomaton
.endproperties
------------------------------------
```

Ignore the first line above for the moment. This file says the good behavior accepted by the automaton structure described in file prop1.mv is that which eventually goes to state GOOD and stays there forever.

2. *If player_A is not hitting the ball now, then eventually he will hit the ball.* The automaton is described in Verilog by:

```
file prop2.v
--------------------------------------
/*
 * If player does not hit the ball now, then he will sometime in the future.
 */
typedef enum {HIT, NO_HIT} action_type;
typedef enum {OKAY, PENDING} prop2_status;

module infinitely_often_hit(clk, player_action);
input clk;
input player_action;

action_type wire player_action;
prop2_status reg state;

initial state = OKAY;

always @(posedge clk) begin
    case(state)
```

```
        OKAY:
            begin
            if (player_action == NO_HIT)
                state = PENDING;
            end

        PENDING:
            begin
            if (player_action == HIT)
                state = OKAY;
            end

        default:;
        endcase;
end
endmodule
-----------------------------------
end file prop2.v
```

Again this is compiled into BLIF-MV using *vl2mv*:

```
vl2mv prop2.v
```

which creates a file prop2.mv. Now we describe the acceptance conditions for this automaton:

```
file prop2.pif
----------------------------------
.include process-fairness.pif

.properties
# property 2
# Begin instantiation of automaton with name ioh.

.automaton ioh
# include blif-mv file obtained from Verilog description
.blifmv prop2.mv
# make connections to the process ping_pong
.connections (player_action = action_A)

# begin fairness constraints on this automaton.
# created above. Since this is an automaton, the fairness constraints
# describe the set of all acceptable (good) behaviors. Hence they
# are positive constraints.
```

```
.state st1 = state:OKAY
.state st2 = state:PENDING
.posfair
.subsets {st1}
.edges (st2, st1)
.endfair
.endautomaton
# This ends the description of the automaton

.endproperties
-----------------------
```

Again, these behaviors represent acceptable behaviors. These constraints say that we can either eventually stay in state OKAY forever, or if not, then must traverse edge (PENDING,OKAY) infinitely often.

To describe a process where both properties hold, in each .pif file we have included file "process-fairness.pif" which puts negative "cycle sets" on state BALL for both players:

```
file process-fairness.pif
---------------
# we begin with any fairness constraints that we want to
# put on the process. Since this refers to a process, the constraints
# are negative since we want to describe behavior that should be excluded
# from the process.
.model player
.state st1 = state:BALL
.negfair
.subsets {st1}
.endfair
.endmodel
---------------
```

This is an example of having behavior in our process which we don't want to consider. It was easier to model a player to allow returning the ball with an arbitrary delay, but keeping the ball forever and not hitting it back is not an interesting behavior and so we exclude it for both players. We do this by excluding it in our model of a player and hence it is excluded for all instances of "player", in this case players A and B. Optionally we could have excluded this behavior on each player separately. For example for player_A:

```
.instance player_A
.state st1 = state:BALL
.negfair
.subsets {st1}
```

```
.endfair
.endinstance
```

Without these constraints on the two players, property 2 does not pass. Language containment *lc* will produce an error trace. Try language containment using a .pif file with the line

```
.include process-fairness.pif
```

commented out (# at the beginning of the line) to see what a bad error trace looks like. For property 2 you will see an error trace where both players have a ball but neither one ever hits it back.

4.4 Running the Example

The commands to run the ping_pong example are:

```
vl2mv ping_pong.v
```

This translates the Verilog file into a set of interacting FSM's described in BLIF-MV using multi-level logic. Since no output file was specified, the result is stored in the file "ping_pong.mv", created automatically from the input file name.

The various fairness constraints are described in the files **prop1.pif, prop2.pif**. To run the example with HSIS:

```
hsis
```

This just starts up HSIS and gives the HSIS response

```
hsis>
```

It also establishes aliases defined in .hsisrc files, a system one and a user defined one. Now HSIS is ready to receive commands (you can type "help" here to see which commands are available and "help Command_name" to get more information about a particular command). Now enter,

```
read_bliv_mv ping_pong.mv
```

(alternately "read_vlog ping_pong.v" would get you to the same point). This reads in the BLIF-MV file generated by vl2mv.

```
vinit
```

is an initialization process that must be run before verification. (Ultimately HSIS will be used not only for verification but also for synthesis.) At this point you can optionally enter,

```
vsim
```

This is a breadth-first/depth-first search routine that can interactively print out all the states that can be reached. It is useful initially to run this to see if things look reasonable. Easy early bugs can be found simply using "vsim".

```
read_pif prop1.pif
```

Read in the .pif file containing the automaton description and fairness constraints for both the process and property 1.

```
lc nnb
```

Recall that "nnb" is the name given to the property described in "prop1.pif". This runs the language containment check to see if the process satisfies the property "nnb" i.e. to see if there is any behavior left after all the good behavior has been excluded. To check the second property, do

```
read_pif prop2.pif
lc ioh
```

The name of the property in this case was "ioh". Note when a second .pif file is read in, the first one is thrown out (unless the "-p" option of read_pif is used), so only the automata and the fairness constraints in the second file are active.

It is sometimes convenient to have a number of properties in a single .pif file and then to read them in all at once. The user can then selectively choose to verify them one at a time, by just selecting them as the argument of lc. For this we would set up a file as follows.

```
file properties.pif
---------------------
.include process-fairness.pif

.properties

#property 1
.automaton nnb
.blifmv prop1.mv
.connections (state_A = state_A, state_B = state_B)
#constraints
.state st1 = state:GOOD
.posfair
.subsets {st1}
.endposfair
.endautomaton

#property 2
.automaton ioh
.blifmv prop2.mv
```

```
.connections (player_action = action_A)
#constraints
.state st1 = state:OKAY
.state st2 = state:PENDING
.posfair
.subsets {st1}
.edges (st2, st1)
.endposfair
.endautomaton

.endproperties
-------------------
end of file properties.pif
```

A scenario for running HSIS on this is as follows:

```
read_blif_mv ping_pong.mv
vinit
read_pif properties.pif
lc nnb
lc ioh
quit
```

4.5 CTL Properties for Ping_Pong

Instead of describing a property by an automaton and checking the property by checking that the language of the process is contained in the language of the automaton, we can also write properties in CTL. This is often the easiest way to check a simple property.

The following file contains three CTL formulae.

```
file ping_pong.ctl
-------------
# there is always at least one ball in play
AG((player_A.state:BALL) + (player_B.state:BALL));

# if player_A is not hitting the ball right now, then he will hit it sometime
# in the future
AG((player_A.me:NO_HIT) => AF(player_A.me:HIT));

#Eventually only one ball is in play. Note ^ denotes XOR.
AF((player_A.state:BALL) ^ (player_B.state:BALL));
-----------------
end of file ping_pong.ctl
```

CAUTION: When a process is non-deterministic, there are some subtleties and correct but non-intuitive results that can happen with CTL formulae that refer not to states (regs in Verilog), but to combinational variables (wires in Verilog), such as the second formula above. You will be safe if all your CTL formulae refer only to states by state names, as do the first and third formulae. For more details about referring to a combinational value in a CTL formula, read the CTL help page for the command "mc".

The HSIS command for model checking these CTL formulae is

```
mc ping_pong.ctl
```

This reads in the CTL formulae contained in the file and runs model checking on each (all three) to see whether the process satisfies the CTL formula. Note that any process fairness constraints put into effect by reading in the last .pif file are still in effect. (In contrast, the acceptance conditions of any property is only in effect when that property is being checked by "lc".) New ones can be used to replace these by simply reading in the appropriate .pif file, e.g.

```
read_pif process-fairness.pif
```

which includes only the fairness constraints on the process. Note that CTL combined with the ability to add fairness constraints provides the expressiveness of *fair CTL*.

Since error tracing is somewhat different for a CTL formula, this is done interactively in HSIS. The second and third CTL formulas above are not satisfied if there are no negative fairness constraints on the players, However, if before you run it, the "process-fairness.pif" file is read in, the formula will be satisfied. These constraints disallow self loops in each player state BALL where each player has a ball forever.

Thus the script

```
read_blif_mv ping_pong.mv
vinit
read_pif process-fairness.pif
mc ping_pong.ctl
```

returns saying that all three formulae pass, but leaving out the read_pif line will lead to the interactive CTL error tracing for both the second and third CTL formula. You can try this just after the last command above by doing the following.

```
vend
vinit
mc ping_pong.ctl
```

The "vend vinit" combination clears out any constraints that exist.

Another useful command is *vset*. It controls a set of run-time options and can be used to control how HSIS runs its internal algorithms and how much information it prints out during debugging. Type

```
help vset
```

to see the various variables that control the run-time options, and their current settings. For example, to reset a "vset" variable to another value type

```
vset vcall_dbg 0
```

This turns off debugging. For further detailed information about "vset", consult the man pages or type "help vset".

5 List of available documentation

1. **Verilog summary.** A good general description of the Verilog hardware description language can be found in "The Verilog Hardware Description Language", by Donald E. Thomas and Philip Moorby, Kluwer Academic Publishers, 1991. We support a fairly complete synthesis subset plus some extensions added to Verilog to support nondeterminism and symbolic variables. A description of these can be found in the VL2MV document.

2. **VL2MV document.** VL2MV is a Verilog compiler which translates the Verilog description of a set of interacting processes into an intermediate format, BLIF-MV which is an extension of the intermediate format BLIF used in our sequential synthesis tool SIS and other logic synthesis programs.

3. **BLIF-MV document.** A description of BLIF-MV. This includes some new extensions which make it more readable and succinct. Generally, BLIF-MV describes hardware as an interconnection of tabular relations each with symbolic inputs and outputs. Generic latches are used to describe states.

4. **PIF Document** A .pif file holds a description of all the ω-automata properties to be verified for a process. The automata transition relations can be described using Verilog writing to a .mv file that can be included using the .blifmv command in PIF. In addition, PIF is used to describe fairness constraints. Properties can also be described using CTL. These are contained in a .ctl file.

5. **Directory of examples.** Verilog description of a number of examples from the literature written in other languages, such as SMV, Murphi, S/R, etc. For each file we have a script that can be run with a summary of the expected results. The descriptions can be found in the "examples" subdirectory. Each example has its own subdirectory. In each there is an associated "script" in the file "hsis_script". The script gives the set of commands that can be exercised on the example. Regression tests for HSIS are performed by running the scripts for all the examples and comparing results. Each directory also contains the associated .v, .mv, .pif, .ctl files required. A summary of the expected results when hsis_script is run is contained in the file "hsis_scr_master_OUT".

6. **Man pages.** Description of HSIS commands and their expected effect. The set of commands can be obtained by typing *help* in HSIS, and the corresponding man page to the command *"command"* can be obtained by typing *help "command"* in HSIS. The subdirectory "help"

contains a complete listing of the manpages in .ps format to be printed out directly at your site.

7. **HSIS paper** A paper describing HSIS, intended for submission to the DAC'94 conference. This paper gives an overall summary of the different HSIS capabilities, and should be read first to gain a better understanding of how and why to use HSIS. This is in the "papers/technical" directory.

8. **Relevant Technical Papers** A number of technical papers which detail some of the algorithms and theory used in HSIS. They can be found in the "papers/technical" directory as .ps files that can be printed out directly on a postscript printer. In addition, various tutorial papers can be b found in teh "papers/tutorial" directory. This includes the present document, called "tutorial.ps".

9. **Anonymous ftp instructions** A complete set of instructions about how to obtain HSIS and install it on a computer system can be found in the README file in the ftp directory. It is a regular text file, not a postscript file.

10. **CTL document** The only document for this right now is in the man pages. You can get some information by starting up hsis and typing "help mc".