NAME

hsis – Hierachical Sequential Interactive System

SYNOPSIS

hsis [options] [file]

DESCRIPTION

HSIS is a formal verification and sequential circuit optimization program. Currently, HSIS can only be used for formal design verification on finite state-based systems. **HSIS** can accept the system description in a hardware description language like a synthesizable subset of Verilog or in an intermediate format like BLIF-MV. The properties to be verified of the system are expressed either as w-automata (omega-automata) or in a temporal logic, CTL. w-automata can be expressed in an HDL format or in BLIF-MV along with fairness constraints. The fairness constraints can be specified in an intermediate format PIF. CTL formulas are directly expressed in the CTL input language, which is fairly close to a natural logic language. Please refer to the supplied documentation for details on specifying the system and the properties.

HSIS can be run in interactive mode accepting commands from the user, or in batch mode reading commands from a file or from the command line. If no options are given on the command line, **HSIS** will enter interactive mode. Otherwise, **HSIS** will enter batch mode. When running in batch mode, **HSIS** reads its input from the file given on the command line, or from standard input if no filename is given; output is directed to standard output, unless **-o** is used to specify an output filename.

When **HSIS** starts-up, it performs an initial source of the files $HSIS/hsis_lib/.hsisrc. Typically this defines a standard set of aliases for various commands. Following that the files ~/.hsisrc are sourced for user-defined aliases at startup.

OPTIONS

-c cmdline
Run **HSIS** in batch mode, and execute **cmdline**. Multiple commands are separated with semicolons.

-f script Run **HSIS** in the batch mode, and execute commands from the file **script**.

-o output-file
Store the output of **HSIS** in an output file.

COMMAND SUMMARY

All HSIS commands are summarized below. Please use the *help* command to see deatiled usage information.

HSIS Verification Commands	
alias	sets up a short alias for a command
close_system	adds non-deterministic inputs to close an open system
dro_off	turns off BDD dynamic reordering
dro_on	turns on BDD dynamic reordering
lc	does verification using language containment
mc	does verification using model checking
read_blif	reads in a file in standard blif format
read_blif_mv	reads in a file in blif-mv format
read_pif	reads in the fairness constraints on the system and the w-automata
read_vlog	invokes verilog compiler to create blif-mv and reads it in
tlc	does verification using timed language containment
vend	ends verification session - removes verification data structures
vinit	initializes verification - must be done before any verification commands can be done
vset	used to display or set verification parameters
vsim	a simple-minded DFS interactive simulator
write_blif_mv	write out a blif-mv file
write_tr	writes the transition relation of the system

HSIS Usage Commands	
echo	echoes the arguments to standard output
help	prints out all hsis commands
history	gives the history of all hsis commands
quit	terminates hsis
set	sets hsis control parameters and displays the ones already set
source	directs hsis commands to be read from named file
time	prints cpu time since last call to time or initial hsis call
unalias	deletes an alias
unset	deletes a parameter previously set
usage	prints a formatted dump of processor-specific usage statistics

STANDARD ALIASES

When **HSIS** starts, it executes commands from a system startup file (usually $(HSIS)/hsis_lib/.hsisrc. This defines a standard set of aliases, and then sources the file ˜/.hsisrc to allow users to define their own set of aliases. The default alias set includes the following aliases which have proven useful. Note that many of the aliases are intended for compatibility with **SIS** Version #1.0.

Standard Aliases		
alias	command	description
rlmv	read_blif_mv	read in a blif-mv file
read_mv	read_blif_mv	read in a blif-mv file
write_mv	write_blif_mv	write a blif-mv file
wrmv	write_blif_mv	write a blif-mv file
wrtr	write_tr	write the system transition relation
write_transition_relation	write_tr	write the system transition relation
read_verilog	read_vlog	compile verilog file and read in blif-mv
rv	read_vlog	compile verilog file and read in blif-mv
exit	quit	quit hsis

DETAILED COMMAND DESCRIPTIONS

alias [name [string]]

unalias name ...

The **alias** command, if given no arguments, will print the definition of all current aliases. Given a single argument, it will print the definition of that alias (if any). Given two arguments, the keyword **name** becomes an alias for the command string **string**, replacing any other alias with the same name. The **unalias** command removes the definition of an alias.

It is possible to create aliases that take arguments by using the history substitution mechanism. To protect the history substitution character '%' from immediate expansion, it must be preceded by a '\' when entering the alias. For example:

```
hsis> alias read read_\%:1 \%:2.\%:1
hsis> alias write write_\%:1 \%:2.\%:1
hsis> read blif lion
hsis> write eqn tiger
```

will create the two aliases 'read' and 'write', execute "read_blif lion.blif", and then execute "write_eqn tiger.eqn". And...

```
hsis> alias echo2 "echo Hi ; echo \%* !"
hsis> echo2 happy birthday
```

would print:

```
Hi
happy birthday !
```

CAVEAT: Currently there is no check to see if there is a circular dependency in the alias definition. e.g.

```
hsis> alias foo "print_stats -f; print_level -l; foo"
```

creates an alias which refers to itself. Executing the command "foo" will result an infinite loop during which the commands "print_stats -f" and "print_level -l" will be executed.

lc [property-name]

This document describes how to use language containment. The sections are:

1. usage
2. debugger

1. Usage

To use language containment, just type lc, or lc property-name. "lc property-name" makes sure the language of the system is contained in the language of the property. Using "lc" without a property specified, just takes the intersection of the languages of all the processes in the system.

The properties should be specified in a .pif file, which includes the fairness constraints for the system, the definition of the property automata, and the acceptance conditions of each automaton. Use read_pif to read the pif file in. The automaton describing the property should be complete, i.e. for all reachable states, it should have a statement saying what transition the automaton takes under all possible input conditions. The automata describing properties should also be deterministic, having exactly one transition for every set of values of the input variables.

2. Debugger

The debugger is called automatically, during lc when the property is not satisfied, unless the flag vcall_dbg is turned off (to turn it off, use "vset vcall_dbg 0"). The debugger prints a path to a fair cycle. The states printed are shown in differential format, i.e. only the variables which change from one state to the next are printed. The fair cycle is printed next, again in the differential format. Only state values are printed unless "vset vcl_print 1" is executed. To see the parameters and their values that are pertinent to verification, type "vset".

AUTHOR: Ramin Hojati

mc ctl-file-name

Here, we describe how the model checker may be used. The sections are:
1. usage
2. examples
3. syntax
4. fairness constraints
5. debugger.

1. Usage: The model checker can be run by typing the command "mc file-name." The file-name contains a set of CTL formulas which should be checked against the model. The formulas will be checked one at a time. For those formulas which are false, if vcall_dbg is set to 1 ("vset vcall_dbg 1"), the debugger is called.

2. Examples:
AG (!(monitor.state:bad));
AG ((process1.state:critical) + (process2.state:critical));
AG ((request:1) => AF (acknowledge:1));
AGEF (state:reset);

3. Syntax: The CTL formulas are of the following format:

1. All identifiers (to be defined later) are CTL formulas. All identifiers must be enclosed by parenthesis.

2. If f and g are CTL formulas, so are,
f + g (or), f * g (and), !f (not), f => g (implies),
f = g (xnor), f ^ g (xor).
AG f (for all paths globally),
AF f (for all paths finally),
AX f (for all next states on all paths),
EG f (for some path globally),
EF f (for some path finally),
EX f (for some next states).

The semantics of model checking CTL formulas is the standard one, which is defined in the documents accompanying HSIS. The identifiers are defined in three ways:
1. var-name==value or var-name:value
2. var-name=value
3. var-name;value.

var-name is the full-path name of the variable. For convenience, the root's name can be dropped. For instance if the root module is called main, and there is a variable state defined in main, then one can just use state, as opposed to main.state. If var-name is a state variable, then all of the above three definitions reduce to the same thing. However, if var-name is a non-deterministic non-state variable, the above statements may have different meanings. We describe the possible interpretations when var-name is a non-state output variable.

var-name==value and var-name:value have the same meaning, and signify the set of all states, where the output variable var-name "can" produce "value". Note that var-name==value is just used as a compact way to signify some set of states. var-name;value is similar to var-name==value, and denotes the set of states, where var-name "must" take the value "value", i.e. all states where var-name is only assigned value. In other words, those states, where var-name is deterministically assigned "value". The exact semantics of var-name=value is being worked out. However, it is intended to catch errors, caused by non-intuitive interpretation of a formula by the model checker. We will describe this by an example.

Consider the following graph: there are two states 0 and 1. There is an output variable which can take a and b in state 0, and just b in state 1. The transition are defined as follows (0 is the initial state).

ps out ns
0 a 0
0 b 1
1 b 1

Consider the formula out==b => AX (out==b). This formula is false, since out==b denotes the set of states {0,1}. From state 0, a possible next state is 0. However, this interpretation may be counter-intuitive to the users, especially if the formula is read as "if b is produced, then in all next states, the output is b". Note that this English statement is true of the system. To avoid such misunderstandings-understandings var-name=value is being developed. Our recommendation to the user is to USE STATE VARIABLES AS MUCH AS POSSIBLE. If output variables are used, the user has to keep in mind that they are just short-hands for a set of states. The user should exercise even more care if the system can produce non-deterministic outputs. Fortunately, HSIS also offers verification using language containment, which can avoid these kinds of misunderstandings-understanding. For example, the property "if b is produced, then in all next states, the output is b" can be easily described by an automaton.

Consider the formula out;b on the above system. This formula is false, since out;b denotes only state 1, and state 0 (the initial state) is not included in {1}. However, out==b is true, since it denotes the set {0,1}.

Currently, HSIS only supports var-name=value in those instances where in all states where var-name has the value "value", either the assignment is deterministic, or all next states can be reached if var-name is assigned value. In other words, if the above ambiguities do not arise.

4. Fairness Constraints One can place fairness constraints on the system. In this case, only the paths satisfying the fairness constraint are considered. To place fairness constraints, use the command read_pif.

5. Debugger The debugger for "mc" is an interactive one. At every point, the debugger is at some state with some sub-formula being wrong. The debugger will print the formula and the state. The debugger will then unfold the formula one operator at a time, giving the user detailed information about what the debugger is doing. The debugger is further discussed in a paper accompanying HSIS.

Currently, only formulas in PNF (positive normal form) are allowed for debugging, i.e. those formulas where negations are applied only to identifiers.

AUTHOR: Ramin Hojati

read_blif_mv [filename]

Read in a system of interacting finite state machines from the file **filename** which is assumed to be in *blif_mv* format. See the blif_mv.tex document for a complete description of the *blif_mv* format.

A hierarchy of machines is created after reading in the description either from a file or from the standard input and the information is stored in the hierarchy manager.

In order for verification of the system to proceed efficiently, it is essential that the MDD variables corresponding to the wires and latches be ordered properly. Experience indicates that no single approach works best - you can experiment with variations by setting flags. For details do a help on set, or look at the order.ps document.

Usual filename conventions apply: **-** (or no filename) stands for standard input.

read_pif [-p] <filename>

Read in the fairness constraints on the system hierarchy and a set of properties expressed as automata with acceptance conditions from the file **filename** which is assumed to be in the PIF format. See the PIF document for a complete description of the PIF format.

If the environment variable **vinfo_print** is set to 1 using the **vset** construct, read_pif will work in the verbose mode.

-p does not reset the existing system fairness constraints (read using a previous read_pif command). However, the set of existing properties is lost. In the default mode (without -p option), both the existing system fairness constraints as well as the properties are lost.

SEE ALSO :

PIF: a Property Interchange Format for Verification using HSIS

BUGS :

1. If .include construct is used, and there is a syntax error in the PIF file, the reported line number may be offset by 1.

2. The character "-" is not allowed in the filename in the *.include* construct

AUTHOR: Vigyan Singhal

read_vlog [filename.v]

This does two things. It executes vl2mv on "filename.v" to create a blif-mv file, "filename.mv". Then it reads this file in using read_blif_mv.

tlc verify
tlc hint <lb_impl> <ub_impl>
tlc hint <lb_impl> <st_impl> [<st_impl> ...]

The form **tlc verify** invokes timed language containment algorithm. Currently, there is no provision for a separate property automaton, so the property to be verified must be included in the description of the system. The system is specified in BLIF_MVT [1], an intermediate format recognized by **read_blif_mv**.

The **tlc** algorithm is iterative: initially all timing constraints are ignored, and then some of them are enforced in subsequent iterations. The iterations are guaranteed to converge in a finite number of steps. To speed up the convergence, the user can use **tlc hint** prior to **tlc verify** (and after **vinit**) to indicate which timing constraints should be enforced even in the first iteration.

The expression <lb_impl> (<ub_impl>) identifies a set of transition enabled only if some timer satisfies a given lower (upper) bound. For example:

    tlc hint var1:1:2=>t1>5 var2:A:B=>t2<=2

indicates that the transition that var1 can change values from 1 to 2
only if t1>5, and that var2 can change values from A to B only if t2<=2. Note that these constraints must be consistent with the original BLIF_MVT file. If they are not, the hint is ignored and the user is warned.

Given the hint above HSIS construct a two-state automata, with one state corresponding to case when t1-t2>3 and the other corresponding to t1-t2<=3 and allows transitions var1:1:2 and var2:A:B to happen simultaneously only in the former. In specifying <lb_impl> and <ub_impl> you can refer to these newly created automata. For example:

    t1-t2>5

refers to the automaton with states t1-t2>5 and t1-t2<=5. More precisely, it refers to all transitions of that automaton that are enabled only if t1>5. In this case, these are all transitions from t1-t2>5.

Similarly:

    t3-t4<2

refers to all transition of the automata with states t3-t4<2 and t3-t4>=2 that are enabled only if t3<2. In this case, these are transitions to t3-t4<2 while t4 is reset.

So, given a hint

tlc hint t1-t2>5 t3-t4<2

HSIS will first construct an automaton (say A) with states t1-t2>5 and t1-t2<=5, the an automaton (say B) with states t3-t4<2 and t3-t4>=2, and finally an automaton (say C) with states t1-t3>3 and t1-t3<=3. Then HSIS will disallow the transitions where moves from t1-t2>5, B moves to t3-t4<2, C moves from t1-t3<=3 and the rest of the system makes a transition that resets t4.

All timing constraints important to a given property can be described using hints with <lb_impl> and <ub_impl> only. Still, it is occasionally more efficient to use hints with <st_impl>. The expression <st_impl> is used to indicate that a timer must satisfy some upper bound on all transitions from a given state, e.g.:

var:1=>t1<5

indicates that t1<5 holds whenever var has value 1. Again, if this is not consistent with the original BLIF_MVT file, the hint is ignored and the warning is issued.

RESTRICTION:

Expressions <lb_impl>, <ub_impl>, and <st_impl> cannot contain any blanks.

SEE ALSO :

[1] F. Balarin, R. Brayton, S. T. Cheng, D. Kirkpatrick, A. Sangiovanni-Vincentelli, E. Wu,
A methodology for formal verification of real-time systems,
(included in HSIS documentation)

AUTHOR: Felice Balarin

vend

This command frees up the storage associated with verification.

vinit

This command initializes verification. It should be called before simulation, langauge containment, model checking, and minimization. To end verification, use "vend".

AUTHOR: Ramin Hojati

vset [variable_name variable-value]

This is used to set the global variables affecting verification. vset can be used in three ways:

1. If only "vset" is typed, a listing of all variables with their current values is printed.
2. If "vset var-name" is typed, the values for all variables, and their intended meanings are given.
3. If "vset var-name var-value" is typed, the value of var-name is set to var-value.

vsim

This command simulates the current design. The simulation uses a breadth-first search. First, all initial states are simulated, then all states reachable from initial states, and so on. The simulation continues until the user aborts it, or all reachable states have been reached. The user can, at any time, terminate the reach at the current level, and continue with one of the states in that level.

AUTHOR: Ramin Hojati

write_blif_mv [filename]

Write the current system of interacting finite state machines file **filename** in the *blif_mv* format. The information about the interacting machines is obtained from the hierarchy manager.

Usual filename conventions apply: **-** (or no filename) stands for standard output.

write_tr [filename]

Write the transition relation of each model in the current system of interacting finite state machines to the file **filename** in the *blif_mv* format. The information about the interacting machines is obtained from the hierarchy manager.

Usual filename conventions apply: **-** (or no filename) stands for standard output.

echo args ...

Echoes the arguments to standard output.

exit

Stop the program. Does not save the current network before exiting.

help [-a] [command]

Given no arguments, help prints a list of all commands known to the command interpreter. The **-a** option provides a list of all debugging commands, which by convention begin with an underscore. If a command name is given, detailed information for that command will be provided.

history [-h] [num]

Lists previous commands and their event numbers. The **-h** option suppresses printing the event number. If **num** is specified, lists the last **num** events. Lists the last 30 events if **num** is not specified.

History Substitution:

The history substitution mechanism is a simpler version of the csh history substitution mechanism. It enables you to reuse words from previously typed commands.

The default history substitution character is the '%' ('!' is default for shell escapes, and '#' marks the beginning of a comment). This can be changed using the "set" command. In this description '%' is used as the history_char. The '%' can appear anywhere in a line. A line containing a history substitution is echoed to the screen after the substitution takes place. '%' can be preceded by a '\' in order to escape the substitution, for example, to enter a '%' into an alias or to set the prompt.

Each valid line typed at the prompt is saved. If the **history** variable is set (see help page for **set**), each line is also echoed to the history file. You can use the "history" command to list the previously typed commands.

Substitutions: at any point in a line these history substitutions are available

%:0	Initial word of last command.
%:n	n'th argument of last command.
%$	Last argument of last command.
%*	All but initial word of last command.
%%	Last command.
%stuf	Last command beginning with "stuf".
%n	Repeat the n'th command.
%-n	Repeat the n'th previous command.
ˆoldˆnew	Replace "old" w/ "new" in previous command.

Trailing spaces are significant during substitution.
Initial spaces are not significant.

quit

Stop the program. Does not save the current network before exiting.

set [name] [value]
unset name ...

A variable environment is maintained by the command interpreter. The **set** command sets a variable to a particular value, and the **unset** command removes the definition of a variable. If **set** is given no arguments, it prints the definition of all variables.

Different commands use environment information for different purposes. The command interpreter makes use of the following:

autoexec
: Defines a command string to be automatically executed after every command processed by the command interpreter. This is useful for things like timing commands, or tracing the progress of optimization.

hsisout Standard output (normally stdout) can be re-directed to a file by setting the variable hsisout.

hsiserr Standard error (normally stderr) can be re-directed to a file by setting the variable hsiserr.

history Each valid command entered at the prompt can be echoed to a file by setting the variable history.

history_char
: By default the character '%' is used to do the history substitution inside hsis. This can be changed by setting the variable **history_char**.

shell_char
: By default the character '!' is used to do invoke shell commands from inside hsis. This can be changed by setting the variable **shell_char**.
: *In order to switch the interpretation of shell_char and history_char it is necessary to first set history_char and then the shell_char. Alternately, you may escape the current history char by preceding it with a '´ while setting the shell_char. In addition none of them can be set to a '#' which is reserved for comments.*

filec Setting this variable enables the user to use "file-completion" like in the C-shell. An ESC causes the current line to be extended to its unique completion. A CTRL-d generates a list of the possible extensions.

open_path
: **open_path** (in analogy to the shell-variable PATH) is a list of colon-separated strings giving directories to be searched whenever a file is opened for read. Typically the current directory (.) is first in this list. The standard system library (typically $HSIS/hsis_lib) is always implicitly appended to the current path. This provides a convenient short-hand mechanism for reaching standard library files.

prompt defines the prompt string. If the prompt string contains a '%'(or whatever the history_char has been set to using the set command), the '%' will be replaced whenever the prompt is printed by the current event number.

order_mode :
: { file_latches, file_all, local, global, global_interleaved } (Default - global) This option is used to select from a variety of BDD variable ordering routines.

: Setting order_mode to "file_latches" or "file_all" allows the user to specify a file which contains the ordering in advance. In the first case the file should specifies only the ordering of the latches; the ordering of the remaining variables is derived automatically. The file should contain the full names of the wires corresponding to all the present states of the latches in the design - the corresponding ordering on the latches is the ordering in which the names appear in the file. The "file_all" option contains a total ordering on all the variables, and is expected to be used in conjunction with the flag "order_write".

: Setting order_mode to one of "local","global,"global_interleaved" allows the user to pick select between algorithms that order variables locally (i.e. all variables from one module are ordered together) or globally (i.e. the hierarchy is collapsed, and dependencies across machines can be exploited). Depending on the example, it may be the case that any one method may be superior to the other. The routine for "local" ordering is the quickest. In general, for datapath intensive circuits, "global" will work best. Refer to the document order.ps for details.

order_write

{ 0, 1 } (Default - 0) When this flag is set, the total variable ordering is written to a file named <root_name>.order in the current working directory, where <root_name> is the name of the top model in the blif_mv description of the design. Since variable ordering can take up to several minutes on large designs, it may be useful to save the ordering the first time it is read in, and then in the future the design can be read in with the order_mode option set to "file_all" circumventing the ordering algorithms all together.

order_time :

{ 0, 1 } (Default - 0) Print the time taken at various stages of performing ordering.

order_print :

{ 0, 1 } (Default - 0) Print the partial orderings generated at various stages of performing ordering - mainly for debugging purposes.

order_heuristic :

{ la1, la2, touati } (Default - la1) Select between different levels of accuracy in solving a graph partitioning problem that is the basis for the ordering.

order_debug :

{ 0, 1 } (Default - 0) Print statistics for debugging, at various intermediate stages of ordering.

cycle_mode :

{ all, none, one } (Default - one) During the variable ordering phase, cyclic connections through combinational logic in the blif_mv hierarchy are detected. Depending on the value of this flag, all/one/none of these cycles are reported. The semantics of a model with such cycles is well defined - refer to the blif_mv document for details. However, they can lead to counter intuitive behavior. In particular, certain transitions may be disabled, even though oscillations in the cyclic logic may not affect the rest of the system. This can lead to false positives - therefore proceed with caution.

rlmv_pass:

This option is used to get a better ordering of variables. Setting this option value to 2, will invoke the ordering routine twice. The first invocation provides a better dependency graph for the second one. This option should be set before reading in the design (with one of "read_blif_mv" or "read_blif" command). Since it will cause ordering to take twice as much time, it is recommended that you try the one-pass option (default) first, and if you suspect that a better ordering exists, then try setting this to 2.
Usage: set rlmv_pass value The permissible values are 1 (default value) and 2.

hsis_print_mode:

This option is used to print the timing information as the given design goes through various stages in HSIS.
Usage: set hsis_print_mode value The permissible values are 0 (no print, the default value) and 1 (print).

source [-psx] filename

Read commands from a file. The **-p** option prints a prompt before reading each command, and the **-x** option echoes each command before it is executed. The **-s** option silently ignores an attempt to execute commands from a nonexistent file.

Arguments on the command line after the filename are remembered but not evaluated. Commands in the script file can then refer to these arguments using the history substitution mechanism.

EXAMPLE:

Contents of test.scr:

```
read_blif_mv %:2
collapse
write_blif_mv %:2.mv
```

Typing "source test.scr lion.mv" on the command line will execute the sequence

```
read_blif_mv lion.mv
collapse
write_blif_mv lion.mv.mv
```

If you type "alias st source test.scr" and then type "st lion.mv bozo", you will execute

```
read_blif_mv bozo
collapse
write_blif_mv bozo.mv
```

because "bozo" was the second argument on the last command line typed. In other words, command substitution in a script file depends on how the script file was invoked.

time

Prints the processor time used since the last time command, and the total processor time used since **HSIS** was started.

usage

Prints a formatted dump of processor-specific usage statistics. For Berkeley Unix, this includes all of the information in the *getrusage()* structure.

FILES

$HSIS/examples/∗
$HSIS/papers/∗
$HSIS/doc/∗
$HSIS/hsis_lib/system.hsisrc
$HSIS/hsis_lib/help/∗

AUTHORS

Many (Berkeley CAD Group)

BUGS

Few! See help files for individual commands.

COMMENTS