HACKT PRSIM

A simulator manual

David Fang

This manual describes the usage and operation of HACKT's `prsim` simulator.

This document can also be found online at http://www.csl.cornell.edu/~fang/hackt/hacprsim.

The main project home page is http://www.csl.cornell.edu/~fang/hackt/.

Short Contents

Table of Contents

1 Introduction

`hacprsim` is a production rule simulator, which is part of the HACKT tool set. `hacprsim` simulates circuits at a digital level of abstraction. It is based on older versions of `prsim`. In the remainder of this document `prsim` will refer to the new version `hacprsim`.

TODO: examples and tutorials taken from test suite?

2 Usage

FYI: the documentation here is extracted from source file '`main/prsim.cc`'.

`-a` *file* [User Option]
Automatically save checkpoint *file* upon exit, regardless of the exit status. Useful for debugging and resuming simulations.

`-b` [User Option]
Batch mode. Run non-interactively, suppressing the prompt and disabling tab-completion (from readline or editline). Useful for scripted jobs. Opposite of '`-i`'.

`-d` *ckpt* [User Option]
Print textual dump of prsim checkpoint file *ckpt*.

`-D` *time* [User Option]
Override the default delay value applied to unspecified rules.

`-f` *flag* [User Option]
See Section 2.1 [General Flags], page 4.

`-h` [User Option]
Print command-line options help and exit.

`-H` [User Option]
Print list of all interpreter commands and exit.

`-i` [User Option]
Interactive mode. Show prompt before each command. Enable tab-completion if built with readline/editline. Opposite of '`-b`'.

`-I` *path* (repeatable) [User Option]
Append *path* to the list of paths to search for sourcing other command scripts in the interpreter.

`-O` *lvl* [User Option]
Optimize internal expanded representation of production rules. Optimizations do not affect the event outcome of simulations. Current valid values of *lvl* are 0 (none) and 1. For more details, See Section 2.2 [Optimization Flags], page 4.

`-r` *file* [User Option]
Startup the simulation already recording a trace file of every event. Trace file is automatically close when simulation exits. This is equivalent issuing `trace` command at the beginning of a simulation session.

`-t` *type* [User Option]
Instead of using the top-level instances in the source file, instantiate one instance of the named *type*, propagating its ports as top-level globals. In other words, use the referenced type as the top-level scope, ignoring the source's top-level instances. Convenient takes place of copy-propagating a single instance's ports.

`-c` [User Option]
Pass to indicate that input file is a source (to be compiled) as opposed to an object file.

`-C` *options* [User Option]
When input is a source file, forward *options* to the compiler driver. **NOTE:** This feature does not work yet, due to non-reentrant `getopt()`.

`-v` [User Option]
Print version and exit.

2.1 General Flags

General flags are all prefixed with '`-f`'. Unless otherwise noted, all options are negatable with a '`-f no-`' counterpart.

`-f default` [User Option]
Reset to the default set of configuration options. Not negatable.

`-f run` [User Option]
Actually run the simulator's interpreter. Enabled by default. '`-f no-run`' is explicitly needed when all that is desired are diagnostic dumps.

`-f dump-expr-alloc` [User Option]
Diagnostic. Print result of expression allocation prior to execution of the simulator.

`-f check-structure` [User Option]
Run some internal structural consistency checks on nodes and expressions prior to simulation. Enabled by default.

`-f dump-dot-struct` [User Option]
Diagnostic. Print a `dot` format representation of the whole-program production rule graph. Recommend using with '`-f no-run`'.

`-f fast-weak-keepers` [User Option]
`-f no-weak-keepers` [User Option]
By default, '`iskeeper=1`' rules are omitted entire from simulation because undriven nodes are assumed to be state-holding, and do not change value. With this option turned on, rules marked '`iskeeper`' are enabled, but interpreted as having attributes '`weak=1`' and '`after=0`', i.e. weak and delay-less.

2.2 Optimization Flags

`-f fold-literals` [User Option]
Collapse leaf nodes of literals directly into their parent expressions. Dramatically reduces the number of expression nodes allocated, and shortens propagation paths to output nodes. Enabled at level '`-O 1`' and above.

`-f denormalize-negations` [User Option]

Apply DeMorgan's rules to transform expressions by pushing negations as close each rule's root node as possible. Production rules are restructured into equivalent expressions. Reduces the number of negation expressions, enabling better folding of negated literals. Enabled at level '`-O 1`' and above.

3 Commands

This chapter documents the various commands available in the interpreter. Commands are organized into categories.

FYI: the command documentation has been extracted from source file '`sim/prsim/Command-prsim.cc`'.

3.1 Built-in Commands

The following commands are listed in the `builtin` category.

`help` *cmd* [Command]
Help on command or category *cmd*. '`help all`' gives a list of all commands available in all categories. '`help help`' tells you how to use `help`.

`echo` *args ...* [Command]
Prints the arguments back to stdout.

`#` *...* [Command]
`comment` *...* [Command]
Whole line comment, ignored by interpreter.

`exit` [Command]
`quit` [Command]
Exit the simulator.

`abort` [Command]
Exit the simulator with a fatal (non-zero) exit status.

`repeat` *n cmd...* [Command]
Repeat a command *cmd* a fixed number of times, *n*. If there are any errors in during command processing, the loop will terminate early with a diagnostic message.

`interpret` [Command]
Open an interactive subshell of the interpreter, by re-opening the standard input stream. This is useful when you want to break in the middle of a non-interactive script and let the user take control temporarily before returning control back to the script. The `exit` command or *`Ctrl-D`* sends the EOF signal to exit the current interactive level of input and return control to the parent. The level of shell is indicated by additional `>` characters in the prompt. This works if `hacprsim` was originally launched interactively and without redirecting a script through stdin.

```
$ hacprsim foo.haco
prsim> !cat foo.prsimrc
# foo.prsimrc
echo hello world
interpret
echo goodbye world
prsim> source foo.prsimrc
hello world
```

```
prsim>> echo where am I?
where am I?
prsim>> exit
goodbye world
prsim> exit
$
```

The following command is useful for showing each executed command.

`echo-commands` *arg* [Command]
Enables or disables echoing of each interpreted command and tracing through sourced script files. *arg* is either "on" or "off". Default off.

The following commands pertain to command aliases.

`alias` *cmd args* [Command]
Defines an alias, whereby the interpreter expands *cmd* into *args* before interpreting the command. *args* may consist of multiple tokens. This is useful for shortening common commands.

`unalias` *cmd* [Command]
Undefines an existing alias *cmd*.

`unaliasall` [Command]
Undefines *all* aliases.

`aliases` [Command]
Print a list of all known aliases registered with the interpreter.

The following commands emulate a directory like interface for navigating the instance hierarchy, reminiscent of shells. By default, *all instance references are relative to the current working directory*, just like in a shell. Prefix with '`::`' to use absolute (from-the-top) reference. Go up levels of hierarchy with '`../`' prefix. The hierarchy separator is '`.`' (dot).

`cd` *dir* [Command]
Changes current working level of hierarchy.

`pushd` *dir* [Command]
Pushes new directory onto directory stack.

`popd` [Command]
Removes last entry on directory stack.

`pwd` [Command]
Prints current working directory.

`dirs` [Command]
Prints entire directory stack.

Shell commands may be executed by prefixing a line with '!'. For example, '`!whoami`'.

New: Block comments are pseudo C-style, using `/*` and `*/` to enclose comments. It is recommended to start use block-comment delimiters on their own lines to avoid confusion. The line parser is very crude. Nested comments are supported. Files with unterminated comments will be reported as errors. `#`-comments are allowed within block comments.

3.2 General Commands

The following commands are listed in the `general` category.

`source` *script* [Command]
: Loads commands to the interpreter from the *script* file. File is searched through include paths given by the [‘`-I`’], page 3 command-line option or the [`addpath`], page 9 command.

`addpath` *path* [Command]
: Appends *path* to the search path for sourcing scripts.

`paths` [Command]
: Print the list of paths searched for source scripts.

3.3 `simulation` Commands

`initialize` [Command]
: Resets the variable state of the simulation (to unknown), while preserving other settings such as mode and breakpoints. The random number generator seed is untouched by this command.

`reset` [Command]
: Similar to `initialize`, but also resets all modes to their default values. This command can be used to quickly bring the simulator to the initial startup state, without having to exit and relaunch. This also resets the random number generator seed used with `seed48`.

Running the simulation.

`step` [*n*] [Command]
: Advances the simulation by *n* time steps. Without *n*, takes only a single step. Time steps may cover multiple events if they are at the exact same time. To step by events count, use `step-event`.

`step-event` [*n*] [Command]
: Advances the simulation by *n* events. Without *n*, takes only a single event. A single event is not necessarily guaranteed to advance the time, if multiple events are enqueued at the same time.

`advance` *delay* [Command]
: Advances the simulation *delay* units of time.

`cycle` [Command]
: Execute steps until the event queue is exhausted (if ever). Can be interrupted by *`Ctrl-C`* or a `SIGINT` signal.

Coercively setting values.

`set` *node val* [*delay*] [Command]
: Set *node* to *val*. If *delay* is omitted, the set event is inserted ‘now’ at the current time, at the head of the event queue. If *delay* is given with a `+` prefix, time is added relative to ‘now’, otherwise it is scheduled at an absolute time *delay*.

`setr` *node val* [Command]
Same as the `set` command, but using a random delay into the future.

`setf` *node val* [*delay*] [Command]
Set forcefully. Same as the `set` command, but this overrides any pending events on *node*.

`setrf` *node val* [Command]
Same as `setf` and `setr` combined; forcefully set *node* to *val* at random time in future, overriding any pending events.

`unset` *node* [Command]
Cancel any pending `set` commands on *node*. This effectively causes a node to be re-evaluated based on the state of its fanin. If the evaluation results in a change of value, a firing is scheduled in the event queue. This command may be useful in releasing nodes from a stuck state caused by a coercive `set`.

`unsetall` [Command]
Clears all coercive `set` commands, and re-evaluates *all* nodes in terms of their fanins.

Breakpoints.

`breakpt` *node* [Command]
Set a breakpoint on *node*. When *node* changes value, interrupt the simulation (during `cycle`, `advance`, or `step`), and return control back to the interpreter.

`breaks` [Command]
Show all breakpoints (nodes).

`nobreakpt` *node* [Command]
`unbreak` *node* [Command]
Removes breakpoint on *node*.

`nobreakptall` [Command]
`unbreakall` [Command]
Removes all breakpoints.

Rescheduling events.

`reschedule` *node time* [Command]
`reschedule-from-now` *node time* [Command]
`reschedule-relative` *node time* [Command]
`reschedule-now` *node* [Command]
If there is a pending event on *node* in the event queue, reschedule it as follows: `reschedule` interprets *time* as an absolute time. `reschedule-from-now` interprets *time* relative to the current time. `reschedule-relative` interprets *time* relative to the pending event's presently scheduled time. The resulting rescheduled time cannot be in the past; it must be greater than or equal to the current time. Tie-breakers: given a group of events with the same time, a newly rescheduled event at that time will be *last* among them. `reschedule-now`, however, will guarantee that the rescheduled event occurs next at the current time. Return with error status if there is no pending event on *node*.

`execute` *node* [Command]

Reschedules a pending event to the current time and executes it immediately. Equivalent to `reschedule-now node`, followed by `step-event`.

3.4 `channel` Commands

The simulator currently provides some limited features for interacting with channels and environments at run-time. The *channel* features allow a user to connect arbitrary sources and sinks to channels, as well as perform assertion checks and value logging. For consistency, all `channels` commands are prefixed with `channel-`.

`channel` *name type bundle rails* [Command]

Registers a named channel (with constituents) in a separate namespace in the simulator, typically used to drive or log the environment. The *name* of the channel should match that of an instance (process or channel) in the source file.

- *name* is the name of the new channel in the simulator's namespace
- *type* is a regular expression of the form `[ae]?[nv]?:[01]?`, where
  - `a` means active-high acknowledge
  - `e` means active-low acknowledge (a.k.a. enable)
  - `n` means active-low validity (a.k.a. neutrality)
  - `v` means active-high validity. These are also the names of the channel signals.
  - `[01]` is the initial value of the acknowledge during reset, which is only relevant to channel sinks.
- *bundle* is the name of the data bundle (rail group) of the channel in the form `[name]:size`, where *size* is the number of rail bundles (M in Mx1ofN). If there are no bundles, then leave the name blank, i.e. just write `:0` If there is only one bundle (1x1ofN), use *size* 0 to indicate that named bundle is not an array.
- *rails* (`[~]rname:radix`) is the name and size of each bundle's data rails, *rname* is the name of the data rail of the channel. *radix* is the number of data rails per bundle (N in Mx1ofN). Use *radix* 0 to indicate that rail is not an array (1of1). If *rails* is prefixed with a `~`, then the data rails will be interpreted as active low.

For example, `channel e:0 :0 d:4` is a conventional e1of4 channel with active-high data rails `d[0..3]`, and an active-low acknowledge (enable) reset to 0, no bundles.

The channel names used in the simulator must correspond to an actual channel (or process) in the input description. (The name used for registration actually resides in the simulator's own namespace, separate from the compiled circuits.) Upon registering a channel name, the simulator locates all relevant subnodes of the channel by appending `.e` or `.a` (or `.v` or `.n`) and `.d[i]` (or however the rails are named) to the end of the channel's name. The data rails' name may be prefixed with `~` to indicate that the rails are active-low. The following are examples of `channel` commands.

```
channel A e:0 :0 d:2
channel B e:1 :0 d:2
channel C e:1 :0 r:2
channel D e:1 :0 d:0
```

```
channel E a:1 :0 r:4
channel F e:1 d:4 r:4
channel G ev:0 :0 d:2
channel H : :0 d:2
channel J e:0 :0 ~d:2
channel H : :0 ~d:2
```

Respectively, the channel declarations are: (A) an `e1of2` channel with `.e` initially low (if coming from the environment), (B) an `e1of2` with `.e` initially high, (C) an `e1of2` with array data rails named `r`, (D) an `e1of2` with one non-array data rail `r`, (E) an `a1of4` with `.a` initially high, (F) an `e4x1of4` channel, (G) an `ev1of2` (enable-valid protocol), (H) an acknowledgeless `1of2` channel (just data-rails), (J) an `e1of2` with active-low data rails, and (K) an acknowledgeless, active-low dual-rail.

A channel can be declared without an acknowledge by omitting the `a` or `e` desginator and the initial value after the colon, as in examples H and K, above. Acknowledgeless channels cannot be used as sources or sinks, however, they can still be watched, logged, and checked against expected values. (Watching, logging, and checking values on channels does not use the acknowledge signal of channels.)

The shared-valid protocols use a additional validity (or neutrality) signal in the channel to perform the handshake. For example, the validity signal can be generated by the completion tree from the sender, and sent to the receiver so the receiver can reuse the completion signal without recomputing it. Shared-validity channels operate slightly differently than other channels. Data is considered valid when the validity is true, not necessarily when the data rails are in a valid state. (Of course, in the cases of properly constructed and connected completion trees, the data will be valid.) Thus, data is logged, printed, checked only when the validity signal becomes active, which is usually after the data rails are valid. (More on sourcing and sinking of shared-validity channels below.)

Another class of channels use level-encoded dual rail (LEDR). Such channels are declared using `channel-ledr`.

`channel-ledr` *name ack:init bundles:num data:init repeat:init* [Command]

Registers a level-encoded dual-rail (LEDR) channel. LEDR channels do not follow a return-to-null protocol; there is exactly one transition per iteration on the forward path. Currently, LEDR channels only encode 1-bit of information per channel. The data rail represents the logic level, and the repeat rail is toggled to communicate another token with the same value. The channel acknowledge (if present) also fires onces per handshake. Together, they are used for 2-phase protocols. The *name* of the channel should match that of an instance (process or channel) in the source file.

- *name* is the name of the new channel in the simulator's namespace
- *ack* is a regular expression of the form `id:[01]`, where
  - if an identifier `id` is given before the `:`, it is interpreted as the name of an acknowledge signal.
  - The value after the `:` (required) is interpreted as the initial state of the acknowledge wire. There is no need to express whether the acknowledge is active-high or active-low.
  - `:` with no name represents a channel with no acknowledge.

- *bundles* is the name of the data-repeat bundle followed by `:` and the number of bundles. Pass just `:0` to indicate that there is only one data-repeat pair.
- *data* is the name of the data rail, interpreted with active-high logic levels. The *init* value specifies the initial value of the data rail on an empty channel.
- *repeat* is the name of the repeat rail. *init* specifies the initial value of the repeat rail on an empty channel.

The initial values of the three rails determines the "empty-parity" of the channel, the parity of the rails when the channel is in its empty state. The initial values are used when the channels are connected up to driving environments such as sources and sinks. For bundled channels, the initial values of data and repeat apply to all bundles.

```
channel NAME e:0 :0 d:0 r:0
channel NAME e:1 :0 d:0 r:0
```

One can get information about channel configurations with the following commands:

`channel-show` *chan* [Command]
Print the current configuration and state of channel *chan*. This also shows the sequence of values associated with sources and expectations with sequence position, if applicable. Looping values are indicated with `*`. This also shows the origin of the value sequence and the name of the current log file to which values are dumped, if enabled.

`channel-show-all` [Command]
Print the current configuration for all registered channels.

`channel-get` *chan* [Command]
This prints the current handshake state of a channel, including the current value, if valid, and the expected activity (e.g., waiting for data from sender, or ack from receiver).

`channel-assert` *chan args...* [Command]
This asserts the current state of a channel. Legal values for arguments (in any order and combination):

- <int> the integer value of the data rails; passes only if data is valid and matches the expected value.
- `valid` (four-phase or two-phase) passes if the channel data rails are in the valid state, or the validity signal (if any) is active, or a two-phase channel is in the set-phase (full).
- `neutral` (four-phase or two-phase) passes if the channel data rails are all neutral/null.
- `full` is synonymous with `valid`
- `empty` is synonymous with `neutral`
- `ack` (four-phase only) passes if the acknowledge is in the active state, whether the signal is active-high or active-low.
- `neg-ack` (four-phase only) passes if the acknowledge is in the negative state.

- `waiting-sender` (four-phase or two-phase) passes if the channel is in a state of the handshake that expects the next action from the sender of the channel.
- `waiting-receiver` (four-phase or two-phase) passes if the channel is in a state of the handshake that expects the next action from the receiver of the channel.

The error-handling policy in the case of a failed assertion is controlled by `channel-expect-fail`.

To control which channels should report values to the console, the simulator provides basic watch commands.

`channel-watch` *chan* [Command]
: Report value of data rails when channel *chan* has valid data. Data validity is only determined by the state of the data rails, and not the acknowledge signal. An unstable channel (that can transiently take valid states) will report *every* transient value. Channels in the stopped state will still be reported, make sure that they are resumed by `channel-release`.

`channel-unwatch` *chan* [Command]
: Remove channel *chan* from watch list.

`channel-watchall` [Command]
: Report values on all channels when data rails become valid.

`channel-unwatchall` [Command]
: Silence value-reporting on all channels.

`channel-report-time` [*on*|*off*] [Command]
: Set this switch on to show simulation timestamps when watched channels are printed or logged channels are written to file. Default: off

Channel values can also be logged to a file or compared against expected values.

`channel-log` *chan file* [Command]
: Record all valid data values on channel *chan* to output *file*. File stream automatically closes upon end of simulation, or with an explicit `channel-close`. Channels in the stopped state will NOT be reported, make sure that they are resumed by `channel-release`.

`channel-close` *chan* [Command]
: Close any output file streams associated with channel *chan*. This flush the current log file, closes the file, and stops logging. This does not affect source nor expect value sequences since those files are read in their entirety upon configuration.

`channel-close-all` [Command]
: Apply `channel-close` to all channels.

`channel-expect-file` *chan file* [Command]
`channel-expect` *chan file* [Command]
: Compare data values seen on channel *chan* against a sequence of values from *file*. Error out as soon as there is a value mismatch. In this variant, once value sequence is exhausted, no more comparisons are done, and channel values go unchecked. See also `channel-expect-file-loop`.

`channel-expect-file-loop` *chan file* [Command]
`channel-expect-loop` *chan file* [Command]
Like `channel-expect-file` but repeats value sequence infintely.

`channel-expect-args` *chan values...* [Command]
Tells a channel *chan* to expect *values* on data rails. Stops checking values after last value is used.

It is legal to log and expect values on the same channel.

The following commands can further control when channels log or check values. Ignoring can be useful for masking out atypical phases of behavior or turning off checking. Ignoring channels is independent of the stopped/released state of a channel.

`channel-ignore` *chan* [Command]
Stop logging and checking expected values on channel *chan*. This can be useful for momentarily ignoring a sequence of values. An ignored channel will continue to respond to changes until it is stopped, by a `stop` command.

`channel-heed` *chan* [Command]
Take channel *chan* out of the ignored state.

`channel-ignore-all` [Command]
Stop logging and checking all channel values.

`channel-heed-all` [Command]
Continue logging and checking all channel values.

Value files: The files referenced by `channel-expect` and `channel-source` may contain `#` comments and blank lines, which are skipped. Only the first value on each line is used, so value sequences should be newline-separated. For now, the remained of each line is simply ignored, so you may use them for comments, but this may change in the future. The other legal value in the file is `X`, which is interpreted as *don't care* for expected values, and *random* for source values.

Channels can be configured to operate as environments when they are not already connected to inputs or outputs. The only conflicting (illegal) configuration combination is that a channel cannot act as source while expecting values. (Why would you want to do that anyways?) Channels configured as sources or sinks can be controlled through the following commands.

`channel-source-file` *chan file* [Command]
`channel-source` *chan file* [Command]
Configure channel *chan* to source values from the environment. Values are take from *file* and read into an internal array. Once values are exhausted, the channel stops sourcing. To repeat values, use `channel-source-file-loop`. A channel configured as a source should have the production rules drive the acknowledge signal and no other rules driving the data rails (otherwise the simulator will issue a warning).

`channel-source-file-loop` *chan file* [Command]
`channel-source-loop` *chan file* [Command]
Like `channel-source-file` except that value sequence is repeated infintely.

`channel-source-args` *chan* [*values...*] [Command]
Source values on channel *chan* using the *values* passed on the command. Sourcing stops after last value is used. Legal values are integers and 'X' for random. If no values are given, then the channel will not source any values, but it will still reset the data rails to neutral state.

`channel-rsource` *chan* [Command]
Configures a channel to source random data values. This is useful for tests that do not depend on data values.

`channel-sink` *chan* [Command]
Configure a channel to consume all data values (infinitely). A sink-configured channel should have data rails driven by the production rules and nothing else driving the acknowledge signal (simulator will issue warning otherwise). A sink-configured channel can also log and expect values. Mmmmm... tokens! Nom-nom-nom...

It is legal to source and sink on the same channel.

It is often useful to query the status of a channel that is sourcing or expecting values.

`channel-assert-value-queue` *chan val* [Command]
For channels that are sourcing or expecting values, assert the state of the channel value array being used. *val* is 1 to assert that values are still remaining, 0 to assert that values are empty (channel is finished). Looped sources and expects will never be empty. This is useful for checking that finite sequence tests have actually completed. Exits fatally if assertion fails.

Channel sources and sinks can be configured to respond with a different timing from the global policy.

`channel-timing` *chan* [*mode* [*args*]] [Command]
With no additional arguments, report the timing mode of channel *chan*. Timing only applies to channels that are configured as a source or a sink. Modes:

- `global` : use the global simulation-wide timing policy.
- `after [delay]` : use a fixed delay.
- `random [[min]:[max]]` : if *max* is specified, use a uniform distribution delay bounded by *max*, otherwise return an exponential variate delay with a minimum of *min*. Unspecified *min* bounds defaults to 0. Unspecified *max* defaults to +INF.

Shared-validity environments: Shared-validity sources operate slightly differently from the other standard channels. Sources of such channels will drive both the data-rails and the validity signal. The validity-signal will automatically react when the data-rails enter a valid state, thus it is treated as both an input and output to the source. However, the validity signal should not be driven by any other circuit, i.e. it should have no fanin. Shared-validity sinks do not respond to data-rails at all, they only respond to the validity signal with the acknowledge. Thus it is the responsibility of the circuit under test to provide the validity signal.

After configuring channels as sourcing or sinking environments, there is one more additional step to enabling them. Channels startup in the *stopped* state, in which they do no

respond to any changes in the circuit, data-rails or acknowledges. Resetting a channel forces a channel into its initial state. For sources, the data rails are always neutral. For sinks, the acknowledge is in the initial state that was specified when the channel was declared. A channel will begin to respond to the circuit only after it has been *released*. Channels may be individually stopped or released, and reset-all and release-all are also provided for convenience.

`channel-reset` *chan* [Command]
: Force a environment-configured channel into its reset state, i.e. a source will reset its data rails to neutral (ignoring state of acknowledge), and a sink will set the acknowledge to the initial value (from configuration) regardless of the data rails (and validity). **IMPORTANT**: This command also freezes a channel in the stopped state, like `channel-stop` and will not respond to signal changes until resumed by `channel-release`.

`channel-reset-all` [Command]
: Force all source- or sink- configured channels into their reset state, as done by `channel-reset`. This is typically done at the same time as global reset initalization.

`channel-stop` *chan* [Command]
: Freeze a source- or sink-configured channel so that it stops responding to signal transitions from the circuit. Stopped channels will not log data nor assert expected values because they may be in a transient state. A channel can be unfrozen by `channel-release`.

`channel-stop-all` [Command]
: Applies `channel-stop` to all channels.

`channel-stop-on-empty` *chan* [Command]
`channel-continue-on-empty` *chan* [Command]
: For channels that are sinking and expecting values (non-loop), stop sinking as soon as expected values are exhausted. The default behavior for a sink is to continue sinking regardless of checking against expected values.

`channel-release` *chan* [Command]
: Releases a source- or sink-configured channel from the stopped state, so that it begins to respond to circuit signal transitions (and continue logging and expecting). Upon resuming, the channel evaluates its inputs and adds events to the event queue as deemed appropriate.

`channel-release-all` [Command]
: Applies `channel-release` to all channels. This is typically used at the end of a reset initialization sequence as the circuit is brought out of the reset state.

Timing: delays are given some default value, except in random timing mode, where delays are randomized. TODO: configure after delays on sources and sinks.

Re-initialization: The `initialize` and `reset` also affect the state of channels. `initialize` retains the configuration (source, sink, watch, expect) of all channels,

however, the data rail tracking is reset to account for all nodes being set to X. All output log streams are closed. Value sequences for sourcing and expecting are retained, but the position index is reset to 0, the beginning. (Rationale: it is uncommon to start at different offsets in the value sequences.) `reset` will completely wipe all registered channels, as if the simulator had just started up.

3.5 `info` Commands

The first subset of commands give information about the properties of the simulated production rules and contain no stateful information.

`attributes` *node* [Command]
: Prints the list of attributes attached to the named *node*.

`fanin` *node* [Command]
: Print all production rules that can fire *NODE*.

`fanin-get` *node* [Command]
: Print all production rules that can fire *NODE*. Also prints current values of all expression literals as 'node:val' and subexpressions as '(expr)<val>'.

`fanout` *node* [Command]
: Print all production rules that *NODE* participates in.

`fanout-get` *node* [Command]
: Print all production rules that *NODE* participates in. Also prints current values of all expression literals as 'node:val' and subexpressions as '(expr)<val>'.

`rules` *proc* [Command]
`rules-verbose` *proc* [Command]
: Print all rules belonging to the named process *proc*. '`.`' can be used to refer to the top-level process. The `-verbose` variant prints the state of each node and expression appearing in each rule.

`allrules` [Command]
`allrules-verbose` [Command]
: Print *all* rules in the simulator, similar to `hflat`. The `-verbose` variant prints the state of each node and expression appearing in each rule.

`rings-mk` *node* [Command]
: Print forced exclusive high/low rings of which *node* is a member.

`allrings-mk` [Command]
: Print all forced exclusive high/low rings.

`rings-chk` *node* [Command]
: Print all checked exclusive rings of which *node* is a member.

`allrings-chk` [Command]
: Print all checked exclusive rings of nodes.

`what` *name* [Command]
: Print the type of the instance named *name*.

`who` *name* [Command]
`who-newline` *name* [Command]
: Print all equivalent aliases of instance *name*. The '`-newline`' variant separates names by line instead of spaces for improved readability.

`ls` *name* [Command]
: List immediate subinstances of the instance named *name*.

The following commands give information about the state of the simulator and the simulated production rules.

`get` *node* [Command]
: Print the current value of *node*.

`getports` *struct* [Command]
`getinports` *struct* [Command]
`getoutports` *struct* [Command]
: Print the state of all port nodes of *struct*. Useful for observing boundaries of channels and processes. `getinports` and `getoutports` partition the set of ports into inputs and outputs. Directionality of inputs/outputs is inferred by the presence of fanin local to the *struct* process instance (its type).

`getlocal` *struct* [Command]
: Print the state of all publicly reachable subnodes of *struct*. Recursive search does not visit private subnodes. Useful for observing channels and processes.

`getall` *struct* [Command]
: Print the state of all subnodes of *struct*. Useful for observing channels and processes.

`assert` *node value* [Command]
: Error out if *node* is not at value *value*. The error-handling policy can actually be determined by the `assert-fail` command. By default, such errors are fatal and cause the simulator to terminate upon first error.

`assertn` *node value* [Command]
: Error out if *node is* at value *value*. Error handling policy can be set by the `assert-fail` command. By default such errors are fatal.

`assert-pending` *node* [Command]
: Error out if *node* does not have a pending event in queue. The error handling policy is determined by the `assert-fail` command. By default, such assertion failures are fatal.

`assert-pending` *node* [Command]
: Error out if *node* does have a pending event in queue. The error handling policy is determined by the `assert-fail` command. By default, such assertion failures are fatal.

`assert` *node value* [Command]
Error out if *node* is driven with strength *value*. The error-handling policy can actually be determined by the `assert-fail` command. By default, such errors are fatal and cause the simulator to terminate upon first error.

`queue` [Command]
Print the event queue.

`assert-queue` [Command]
Error out if event queue is empty. Useful for checking for deadlock. The error handling policy is determined by the `assert-fail` command. By default, such assertion failures are fatal.

`assertn-queue` [Command]
Error out if event queue is not empty. Useful for checking for checking result of cycle. The error handling policy is determined by the `assert-fail` command. By default, such assertion failures are fatal.

`tcount` *node* [Command]
`tcount` shows the number of non-X transitions that have ever occurred on *node*.

`check-invariants` [Command]
Checks every invariant expression in the design. Returns true if there were any certain violations, excluding possible violations of invariants.

`backtrace` *node* [*val*] [Command]
Trace backwards through a history of last-arriving transitions on node *node*, until a cycle is found. If *val* is omitted, the current value of the node is assumed. Useful for tracking down causes of instabilities, and identifying critical paths and cycle times.

`why-x` *node* [Command]
`why-x-verbose` *node* [Command]
`why-x-1` *node* [Command]
`why-x-1-verbose` *node* [Command]
`why-x-N` *node maxdepth* [Command]
`why-x-N-verbose` *node maxdepth* [Command]
Print causality chain for why a particular node (at value X) remains X. In expressions, X nodes that are masked out (e.g. 1 | X or 0 & X) are not followed. The verbose variant prints more information about the subexpressions visited (whether conjunctive or disjunctive), and pretty prints in tree-indent form. Recursion terminates on cycles and already-visited nodes. The '`-1`' variant only queries through depth 1, and the '`-N`' variant queries to a maximum depth of *maxdepth*.

`why` *node* [*val*] [Command]
`why-verbose` *node* [*val*] [Command]
`why-1` *node* [*val*] [Command]
`why-1-verbose` *node* [*val*] [Command]
`why-N` *node maxdepth* [*val*] [Command]

`why-N-verbose` *node maxdepth* [*val*] [Command]

Print reason for node being driven to a given value, 0 or 1. X is not a valid value for this procedure. If *val* is not given, it is assumed to be the current value of the node. The algorithm examines each node's fanins and follows nodes on paths where the subexpression is true (path through transistors is on). The analysis will terminate at state-holding nodes that are not being driven to their current value. This is an excellent aid in debugging unexpected values. The verbose variant prints expression types as it auto-indents, which is more informative but may appear more cluttered. The '`-1`' variant only queries through depth 1, and the '`-N`' variant queries to a maximum depth of *maxdepth*.

`why-not` *node* [*val*] [Command]
`why-not-verbose` *node* [*val*] [Command]
`why-not-1` *node* [*val*] [Command]
`why-not-1-verbose` *node* [*val*] [Command]
`why-not-N` *node maxdepth* [*val*] [Command]
`why-not-N-verbose` *node maxdepth* [*val*] [Command]

Print reason for node not being a given value, 0 or 1. X is not a valid value for this procedure. If *val* is not given, it is assumed to be opposite of the current value of the node. "Why isn't this node changing?" The algorithm examines each node's fanins and follows nodes that prevent the relevant expression from evaluating true. This is an excellent tool for debugging deadlocks. The verbose variant prints expression types as it auto-indents, which is more informative but may appear more cluttered. The '`-1`' variant only queries through depth 1, and the '`-N`' variant queries to a maximum depth of *maxdepth*.

`status` *value* [Command]
`status-newline` *value* [Command]

Print all nodes whose current value is *value*. Frequently used to find nodes that are in an unknown ('X') state. Valid *value* arguments are [0fF] for logic-low, [1tT] for logic-high, [xXuU] for unknown value. The `-newline` variant prints each node on a separate line for readability.

`get-driven` *node* [Command]

Reports the drive state of pull-up/dn on a *node*. See also `fanin-get` for details.

`status-interference` [Command]
`status-weak-interference` [Command]

Print all nodes that have strongly interfering fanins, i.e. the pull-up and pull-downs are on and causing shorts. `status-weak-interfere` reports possible interferences where at least one direction is being pulled `X` (unknown). This command is useful for checking the safety of a particular state or snapshot of your circuit.

`status-driven` *val* [Command]
`status-driven-fanin` *val* [Command]

Reports all nodes that are in a particular drive-state. The drive-state of a node is the strongest pull in any direction. The current value of the node is not considered. *val* is 0 for undriven nodes, X for X-driven nodes, and 1 for driven nodes (which may

include interfering nodes). The `status-driven-fanin` variant filters out nodes with no fanin (inputs), which are always undriven.

`time` [Command]
What time is it (in the simulator)?

`unused-nodes` [Command]
Print all nodes with no fanins and no fanouts, regardless of state.

`unknown-inputs` [Command]
Print all nodes with value X that have no fanins, i.e. input-only nodes. Connections to channel sinks or sources can count as inputs (fake fanin). Great for debugging forgotten environment inputs and connections! This variant includes X-nodes with no fanouts (unused nodes).

`unknown-inputs-fanout` [Command]
Print all nodes with value X that have no fanins, i.e. input-only nodes. This variant excludes X-nodes with no fanouts (unused nodes).

`unknown-outputs` [Command]
Print all nodes with value X that have no fanouts, i.e. output-only nodes. Connections to channel sinks and sources can counts as outputs (fake fanout). This will not catch output nodes that are fed back into circuits. This variant excludes X-nodes with no fanouts (unused nodes).

`unknown-fanout` [Command]
Print all nodes with value X that have fanouts. Connections to channel sinks and sources can counts as outputs (fake fanout).

`unknown-undriven-fanin` [Command]
Print all nodes with value X that have fanins, but are not being pulled. These nodes are typically candidates for adding resets to fix.

3.6 `view` Commands

View commands affect what is displayed while the simulation is running.

`confirm` [Command]
`noconfirm` [Command]
Controls whether or not correct assertions are reported.

`watch` *nodes...* [Command]
Adds *nodes* to a watch-list of nodes to display when their values change, much like `breakpt`, but doesn't interrupt simulation.

`unwatch` *nodes...* [Command]
Removes *nodes* from list of watched nodes.

`watches` [Command]
Print list of all explicitly watched nodes.

`watchall` [Command]
`nowatchall` [Command]
Display every node value change (regardless of present in watch-list). `nowatchall` restores the state where only explicitly watched nodes are displayed on value changes.

`cause` [Command]
`nocause` [Command]
`cause` displays causal information of nodes as they change value. `nocause` hides cause information, but silently keeps track.

`tcounts` [Command]
`notcounts` [Command]
`tcounts` displays transition counts on nodes as they change value. `notcounts` hides transition count information. Only transitions to 0 or 1 are counted; transitions to X are not counted. Transitions are always counted, just not always displayed.

`zerotcounts` [Command]
Reset all transition counts to 0.

`watch-queue` [Command]
`nowatch-queue` [Command]
Show changes to the event-queue as events on only watched nodes are scheduled. Typically only used during debugging or detailed diagnostics.

`watchall-queue` [Command]
`nowatchall-queue` [Command]
Show changes to the event-queue as *every* event is scheduled. Typically only used during debugging or detailed diagnostics.

3.7 `modes` Commands

This section lists commands that affect the execution of the simulation.

`checkexcl` [Command]
`nocheckexcl` [Command]
Enables mutual exclusion checking for checked exclusive node rings. Checking is enabled by default. Users of old `prsim` should replace uses of `CHECK_CHANNELS` with these commands.

`eval-order` [*mode*] [Command]
With no argument, reports the current evaluation ordering mode. With *mode* (either `inorder` or `random`), fanouts are evaluated either in a pre-determined order, or in random order. Random ordering is useful for emulating random arbitration among fanouts of the same node. Default mode is `inorder`.

Timing mode.

`timing` [*mode*] [*args*] [Command]
Modes: '`uniform`' *delay* applies the same delay to *all* rules. `uniform` is useful for getting quick transition counts. '`random`' gives every event a different randomly assigned

delay. `random` is most useful for detecting non-QDI logic violations. '`after`' applies a different delay for each rule, as determined by the `after` PRS rule attribute.

The `after_min` and `after_max` rule attributes only have any effect in random mode or on nodes marked `always_random`. In random-mode, `after_min` specifies a lower bound on delay, and `after_max` specifies an upper bound on delay. When no upper bound is specified, the delay distribution is an exponential variate; when an upper bound is specified, a delay is generated with uniform distribution between the bounds. If only a lower bound is specified, its value is added to the exponentially distribtued random delay.

Timing '`random`' also takes additional optional arguments for default min and max delays for *unspecified rules*; user-written values from the source will always take precedence. A max delay value of 0 is interpreted as being unbounded.

- '`timing random`' preserves the default min/max delays
- '`timing random :`' will clear the default min/max delays
- '`timing random X:`' sets the default min delay
- '`timing random :Y`' sets the default max delay
- '`timing random X:Y`' sets the default min and max delays

Timing '`binary`' randomly chooses between a min and max delay value with a specified probability, like a skewer coin-flip. This mode completely disregards any user-specified delay attributes in the source, including delay '`after=0`'.

- '`timing binary 10:90 0.5`'
- '`timing binary 10:50 0.95`'

`random` [Command]
Deprecated, but retained for legacy compatibility. Synonymous with '`timing random`'.

`norandom` [Command]
Deprecated, but retained for legacy compatibility. Synonymous with '`timing uniform`'.

`seed48` [*int int int*] [Command]
Corresponds to libc's seed48 function. With no argument, print the current values of the internal random number seed. With three (unsigned short) integers, sets the random number seed. Note: the seed is automatically saved and restored in checkpoints. The seed value is reset to 0 0 0 with the `reset` command, but not with the `initialize` command.

The simulator now supports *weak rules* which can drive un-pulled nodes but always yield to normal rules. Weak rules are marked with the `[weak=1]` production rule attribute. The use of weak-rules can be globally enabled or disabled.

`weak-rules` [*on*|*off*|*show*|*hide*] [Command]
Simulation mode switch which globally enables or disables (ignores) weak-rules. Weak-rules can only take effect when normal rules pulling a node are off. The `hide` and `show` options control whether or not weak rules are displayed in rule queries, such as `fanin`, `fanout`, and `rules`.

Diagnostic controls. The following commands control the simulation policy for run-time logic violations. Allowed arguments are: `ignore`, `warn`, `notify`, and `break`. `ignore` silently ignores violations. `notify` is the same as `warn`, which prints a diagnostic message without interrupting the simulation. `break` reports an error and stops the simulation. When no argument is given, just reports the current policy.

`unstable` [*mode*] [Command]
: Set the simulator policy in the event of an instability. A rule is unstable when it is enqueued to fire, but a change in the input literal/expression stops the rule from firing. Stability is a requirement of quasi-delay insensitive circuits. Default mode is `break`.

`weak-unstable` [*mode*] [Command]
: Set the simulator policy in the event of a weak-instability. A rule is weakly-unstable when it is enqueued to fire, but a change in the input literal (to unknown) *may* stop the rule from firing. Default mode is `warn`.

`interference` [*mode*] [Command]
: Set the simulator policy in the event of interference. A rule is interfering when it is fighting an opposing (up/down) firing rule. Interference will always put the conflicting node into an unknown state. Non-interference is a requirement of quasi-delay insensitive circuits. Default mode is `break`.

`weak-interference` [*mode*] [Command]
: Set the simulator policy in the event of weak-interference. A rule is weakly interfering when *may* fight an opposing (up/down) firing rule. Weak-interference will put the conflicting node into an unknown state. Default mode is `warn`.

`invariant-fail` [*mode*] [Command]
: Set the error-handling policy for certain invariant violations, when an invariant expression evaluates to `false`.

`invariant-unknown` [*mode*] [Command]
: Set the error-handling policy for possible invariant violations, i.e. when an invariant expression evaluates to `X`.

`assert-fail` [*mode*] [Command]
: Set the error-handling policy for when the `assert` command fails.

`channel-expect-fail` [*mode*] [Command]
: Set the error-handling policy for when the a channel encounters a value different from was expected.

`checkexcl-fail` [*mode*] [Command]
: Set the error-handling policy for when an exclusion check fails.

`mode` [*md*] [Command]
: Without arguments, reports the current simulation policies on logic violations. With argument *md*, `run` is the default set of policies, `reset` is only different in that `weak-unstable` is `ignore`d. `reset` is useful during the initalization phase, when

some rules may transiently and weakly interfere, as they come out of unknown state. `paranoid` causes the simulation to break on weak-instabilities and weak-interferences, which is useful for debugging. `fatal` causes the simulation to exit immediate with non-zero exit status, which is useful for non-interactive batch testing. **Caution:** `fatal` also causes the following diagnostic conditions to exit fatally: *invariant-fail*, *invariant-unknown*, *assert-fail*, *channel-expect-fail*, *excl-check-fail*

policy	*default*	`reset`	`run`	`paranoid`	`fatal`
interference	break	break	break	break	fatal
weak-interference	warn	ignore	warn	break	fatal
unstable	break	break	break	break	fatal
weak-unstable	warn	warn	warn	break	fatal
assert-fail	fatal	-	-	-	fatal
excl-check-fail	fatal	-	-	-	fatal
channel-expect-fail	fatal	-	-	-	fatal
invariant-fail	break	-	-	-	fatal
invariant-unknown	warn	-	-	-	fatal

Two additional commands control the behavior of unstable rules. These are particularly useful for simulating circuits that expect to glitch, such as synchronous (clocked) circuits.

`unstable-unknown` [Command]
When set, this causes unstable rules to be result in an unknown value on the output node. The opposite effect is the `unstable-dequeue` command.

`unstable-dequeue` [Command]
When set, this causes unstable rules to be dequeued from the event queue. The opposite effect is the `unstable-unknown` command. This option also allows events that drive a node to 'X' in the queue to be *overtaken* and replaced with known values if the fanin pull of the node is resolved to a non-interfering direction *before* the 'X' event on the node is dequeued.

Some additional commands are available for examining and controlling some internal lookup table caching. Most users won't need to worry about these.

`frame-cache-half-life` [*int*] [Command]
Sets the period (in event count) at which the internal cache of footprint frames (lookup-tables) should be aged.

`frame-cache-halve` [Command]
Manually age the cache, as if a half-life period elapsed. One typically never needs to do this unless the memory usage has gone out of hand. The output reports the total amount of weight lost in the cache, which is meaningless unless you know how the cache works.

`frame-cache-dump` [Command]
Print the contents of the global footprint-frame cache. Really only intended for memory diagnostics.

3.8 `tracing` Commands

Checkpointing is useful for saving the state of the simulator, which allows one to interrupt and resume long simulations, and also examine points of failure in detail.

`save` *ckpt* [Command]
Saves the current state of the production rules and nodes into a checkpoint file *ckpt*. The checkpoint file can be loaded to resume or replay a simulation later.

`load` *ckpt* [Command]
Loads a `hacprsim` checkpoint file into the simulator state. Loading a checkpoint will not overwrite the current status of the auto-save file, the previous autosave command will keep effect. Loading a checkpoint, however, will close any open tracing streams.

`autosave` [*on*|*off* [*file*]] [Command]
Automatically save checkpoint upon end of simulation, regardless of exit status. The `reset` command will turn off auto-save; to re-enable it with the same file name, just *`autosave on`*. The '`-a`' command line option is another way of enabling and specifying the autosave checkpoint name.

The simulator supports trace file recording. Unlike checkpoints, trace files contain information for the entire history of execution, not just the state at one point in time.

`trace` *file* [Command]
Record events to tracefile *file*. Overwrites *file* if it already exists. A trace stream is automatically closed when the `initialize` or `reset` commands are invoked. See the '`-r`' option for starting up the simulator with a newly opened trace stream.

`trace-file` [Command]
Print the name of the currently opened trace file.

`trace-close` [Command]
Finish writing the currently opened trace file by flushing out the last epoch and concatenating the header with the stream body. Trace is automatically closed when the simulator exits.

`trace-flush-notify` [*0*|*1*] [Command]
Enable (1) or disable (0) notifications when trace epochs are flushed.

`trace-flush-interval` *steps* [Command]
If *steps* is given, set the size of each epoch according to the number of events executed, otherwise report the current epoch size. This regulates the granularity of saving traces in a space-time tradeoff.

`trace-dump` *file* [Command]
Produce textual dump of trace file contents in *file*.

3.9 `debug` Commands

This section is reserved for commands that are only useful for debugging the simulator. Some commands that end in `-debug` have already been mentioned in the other sections.

`check-structure` [Command]
: Check internal graph/tree/map data structures for consistency and coherence.

`check-queue` [Command]
: Check internal event queue (as seen by `queue`) for inconsistencies, such as missing back-links, multiply referenced nodes... (Because the simulator was not perfect.)

`memstats` [Command]
: Show memory usage breakdown of the simulator.

4 Execution

This chapter describes the internal event-driven execution algorithm.

5 Diagnostics

Over the course of simulation, one is likely to encounter various diagnostic messages. This chapter describes some of the message one might encounter in simulations.

5.1 Interactive Diagnostics

```
WARNING: pending event for node ‘$1’; ignoring request.
```

The above message is displayed whenever the user tries to `set` a node to a value, when it is already scheduled to take on a different value in the event-queue. This is a safe-guard to prevent users from accidentally modifying events that are caused by the circuits. This message also appears if a user tries to set a node to different values without `cycle`-ing in between. To forcefully override the event queue, use the `setf` command, which will issue the following warning.

```
WARNING: pending event for node ‘$1’ was overidden.
```

Assertion failures are pretty self-documenting.

5.2 Delay-insensitivity Violations

A rule is said to be stable if its guards remain stably true before the output fires. An unstable rule can have its guards momentarily evaluate true before the output fires. In the simulation this translates into an event being enqueued, and dequeued (or altered) before it fires. The simulator uses the policy set by `unstable` to choose an action when there is an instability. For any policy except `ignore`, the simulator prints:

```
WARNING: unstable ‘$1’+/-
>> cause: ‘$2’ (val:$3)
```

This means that the new transition on node `$2` to value `$3`, caused the guard for rule `$1 +` or - to become false while the event was still scheduled.

```
WARNING: weak-unstable ‘$1’+/-
```

just means that the instability was caused by the guard expression becoming `X`, meaning that there *may* be an instability.

In the `unstable-unknown` (default) mode, the previous pending event on `$1` will be overwritten to become `X`, which is a conservative model of what could potentially happen in a real circuit. The alternative is to use the `unstable-dequeue` policy, which simply dequeues the unstable rule from firing.

Rules are interfering if the pull-up and pull-down rules can both fire at the same time, i.e. they form a short path between Vdd and GND.

```
WARNING: interference ‘$1’
>> cause: ‘$2’ (val:$3)
```

reports the latest rule that turned on to cause interference. Usually this means that there is an error in the production rules. Take a look at `fanin-get $1` to see which subexpressions in the rules are on.

A weak-interference occurs when an `X` is fighting a `1` or `X` pull. This can frequently occur transiently during reset, when nodes are still coming into known values. `mode reset` can be used to ignore these, for example, during the reset phase.

5.3 Exclusion Violations

The production rules can be annotated with one class of invariant directives for mutual exclusion: `exclhi` and `excllo`. These check at run-time that mutual exclusion among rings of nodes is maintained throughout execution. They assume that `X` values of nodes are safe and do not cause violations. (Implementation note: each ring's exclusion state is actually a single boolean value, one bit of a long bit-field.) A violation of exclusion will result in the following message:

```
ERROR: exclhi/lo violation detected!
ring-state:
node1 : val1
node2 : val2
...
but node ‘$1’ tried to become $2.
The simulator’s excl-check-lock state is no longer coherent;
do not bother trying to continue the simulation,
but you may further inspect the state.
You probably want to disable excl-checking with ‘nocheckexcl’
if you wish to continue the simulation.
```

This just identifies participants of the ring that is violated, and the last member that tried to fire. Since the data structure for the ring locks does not support counting, you will have to disable exclusion checking with `nocheckexcl` to continue. Debugging the state of the simulation at the time of violation is highly recommended. (Use `backtrace` and `why`.)

5.4 Channel Diagnostics

Typically when a channel is configured as a source, the data-rails have no fanin from the production rules, and the acknowledge is driven by the production rules. When this is not the case, the simulator issues a warning:

```
Warning: channel acknowledge ‘$1.a/e’ has no fanin!

Warning: channel data rail ‘$1’ ($2,$3) has fanin.
```

While this is not fatal, it usually indicates a user-error in configuration.

Likewise, sinks expect to have data rails driven by production rules, and the acknowledge fanout to production rules.

Channels also expect data rails to behave sanely, maintaing mutual exclusion within each data bundle. Violation will produce:

```
Channel data rails are in an invalid state!
In channel ‘$1’, got $2 high rails, whereas only $3 are permitted.
```

This is similar in meaning to exclusion violations.

TODO: this section doesn't exhaust the list of diagnostic message. Add as needed.

5.5 Fatal Diagnostics

In the worst-case scenario, the simulator may enter an invalid state where some invariant (assertion) no longer holds. (This can indicate some corruption of data structures, for

instance.) If you see such an internal simulation error (ISE), calmly back away from the computer and file a bug-report.

```
Internal simulator error: in some_function at some_file:line:
*** Please submit a bug report including version
*** "HACKT-...", VERSION,
*** (preferably reduced) test case, steps to reproduce, and configuration,
*** if appropriate, to <email@address>.
*** This program will now self-destruct.  Thank you, and have a nice day.
```

6 Co-simulation

One may desire to run `hacprsim` with another simulator. This chapter describes ways in which this can be accomplished.

6.1 Verilog PLI Setup

Popular commercial tools sometimes provide an interface for mixing simulators. In this section, we describe how `hacprsim` can be integrated with Synopsys' VCS Verilog Simulator. The interface they provide is called VPI for Verilog PLI (Programming Language Interface).

All of the functionality of the `hacprsim` is provided in the '`libhacktsim.la`' library. If you've built the HAC tools with shared libraries, then all the necessary libraries should already be available in your installation path.

The first step is to enable support for building a plug-in to be loaded into the VPI. During the configure step, pass the option: `--with-vpi=`*`/path/to/vpi/development-files`*

```
./configure --with-vpi=/usr/local/cad/synopsys/vcs
```

The path should point to one directory up from where the C headers reside. (Basically exclude the `/include` from the path argument, as it is automatically appended.) `configure` checks for '`vpi_user.h`' in the given path. If it is found, then compilation will create a module for VCS's VPI, '`vpihacprsim.la`' (which is installed with the actual `.so/dll/dylib` shared library). This configuration check also looks for `vcs` in the path; `vcs` is used to compile Verilog into a simulation executable.

Disclaimer: the following instructions are taken from the author's trials, and do not necessary reflect the actual documented use. Consult the vendor's VPI documentation for more information.

To enable VPI in the VCS compiler, pass '`+vpi`' to the `vcs` command.

To specify the prsim module, pass: '`-use_vpiobj /prefix/lib/hackt/vpihacprsim.so`' and '`-L/prefix/lib/hackt`'. The '`-L`' flag is forwarded to the linker used by VCS in the final stage of compilation, and is *also required* at run-time for finding dependent shared libraries. The method described here uses a dynamic library (plug-in module), however a static library could also suffice, as long as all symbols were resolved at link time. Dependent libraries should be linked with the '`-l`' option which is forwarded to the linker.

VCS produces an executable, say '`simv`', which is dynamically linked to '`vpihacprsim.so`'. If the HAC libraries do not already reside in one of the default run-time library search paths (searched by `ld.so`), then you must help `simv` find dependent libraries by passing the same linker flags through `LD_LIBRARY_PATH`.

```
sh$ LD_LIBRARY_PATH=/prefix/lib/hackt:$LD_LIBRARY_PATH ./simv

csh% env sh$ LD_LIBRARY_PATH=/prefix/lib/hackt:$LD_LIBRARY_PATH ./simv
```

6.2 VPI Basics

Now that you've setup your VCS compilation environment to link in the `hacprsim` library. It's time to connect your Verilog instances to `hacprsim`. This section walks you through the basic steps.

The '`vpihacprsim.so`' object you linked in to your simulation executable defines some new functions that one can call from Verilog. The first thing you should do is tell VCS what HAC circuit you are co-simulating.

`$prsim` *obj* [Function]

Loads HAC object file *obj* for `hacprsim` co-simulation. The object file need not be compiled through the allocation phase, the library will automatically compile it through the allocation phase for you if needed.

```
initial
begin
  $prsim("my_hac_circuit.haco-a");
  ...
end
```

The named object file should be compiled from HAC source. Though the object file doesn't not needed to be compiled through the allocation phase, it is recommended to catch errors as early as possible. In any case, the `hacprsim` library will automatically compile the object file further if needed. It is important to load this in the `initial` block before any further actions are done with `hacprsim`. Before running the simulation executable, you need to make sure that the object file loaded by `$prsim()` exists. (Otherwise the simulation will fail with a thrown exception, showing a stack dump.)

Continuing in the `initial` block, you can make connections between Verilog and `hacprsim`. Suppose your HAC file instantiates the following:

```
defproc inv(bool a, b) { prs { a => b- } }
inv foo, bar;
```

You could write connections in Verilog with the following functions:

`$to_prsim` *vname pname* [Function]

Establish a connection from Verilog signal *vname* to `hacprsim` signal *pname*. When *vname* changes value, `hacprsim` will be updated accordingly. This command should be invoked prior to any events in simulation.

`$from_prsim` *pname vname* [Function]

Establish a connection from `hacprsim` signal *pname* to Verilog signal *vname*. When *pname* changes value in `hacprsim`, the Verilog simulator will be notified accordingly.

A signal can go both ways between the simulation environments, as in the following example:

```
module TOP;
  reg a, b, c;
initial
begin
  $prsim("my_hac_circuit.haco-a");
  $to_prsim("TOP.a", "foo.a");
  $from_prsim("foo.b", "TOP.b");
  $to_prsim("TOP.b", "bar.a");
  $from_prsim("bar.b", "TOP.c");
end
endmodule
```

`"TOP.b"` is both driven by `hacprsim` and also fans out to circuits in `hacprsim`.

There are other useful functions in the `hacprsim` VPI library.

`$prsim_cmd` *cmd* [Function]
: Runs an arbitrary command *cmd* (string) that would normally be interpreted by `hacprsim`. This is the one command to rule them all, the last command you will ever need.

`$prsim_default_after` *time* [Function]
: Set the default delay for unspecified rules to *time*, in `hacprsim`'s time units (unitless). This is analogous to the command-line '`-D time`' option. This command should be invoked *before* loading the object file (`$prsim()`) that initializes the state of the simulation.

`$prsim_sync` [Function]
: Synchronize callbacks with current `hacprsim` event queue. This is needed because some `$prsim_cmd` commands may introduce new events into the event queue, which requires re-registration of the callback function with updated times.

  Update: this command is now deprecated because now all `$prsim_cmd` calls automatically re-synchronize the event queues, as a conservative measure.

`$prsim_set` *node val* [Function]
: Sets a *node* in `hacprsim` to *val*. Synonymous with `$prsim_cmd("set node val");`.

`$prsim_get` *node* [Function]
: Prints value of *node* in `hacprsim`. Synonymous with `$prsim_cmd("get node");`.

`$prsim_watch` *node* [Function]
: Register a prsim-driven node watch-point. Synonymous with `$prsim_cmd("watch node");`.

`$prsim_mkrandom` *v* [Function]
: For *v* 1, synonymous with `$prsim_cmd("timing random");`. For *v* 0, synonymous with `$prsim_cmd("timing after");`.

`$prsim_resetmode` *v* [Function]
: For *v* 1, synonymous with `$prsim_cmd("mode reset");`. For *v* 0, synonymous with `$prsim_cmd("mode run");`.

The following sections walk you through examples of co-simulating hacprsim with VCS. The author highly recommends copying these examples to run them and study them.

6.3 VPI Example

Several examples are installed in the '`/prefix/share/hackt/doc/example/ARCH/vpiprsim-inverters`' directory, where *ARCH* is the host-triplet of the machine you are running on, as determined by configure (See `hackt version`). This example demonstrates several inverters communicating across the `hacprsim` and VPI boundary, in a shoelace connection.

Copy the contents of the directory to a temporary directory, and rename '`Makefile.copy`' (created from template '`example.mk`') to '`Makefile`'. The Makefile is

already setup to include '`hackt.mk`' and provides some basic rules for compiling and running the Verilog simulation. The '`shoelace.v`' file is compiled to an executable '`shoelace.vx`' which runs the simulation. '`shoelace.v`' references '`inverters.haco-a`', which is a prerequisite for running (but not compiling) the simulation executable.

`make check` will run all the necessary steps in order and print the result of running the executable to '`stdout`'. Note that the invocation of the executable is prefixed with an environment extension for `LD_LIBRARY_PATH`. This is not necessary if you have already included `/prefix/lib/hackt` (a.k.a. `pkglibdir`) in your environment.

6.4 VPI with Channels

The same example directory contains an example that uses `hacprsim`'s channel features and commands, called '`channel-source-sink.v`', See Section 3.4 [Channel Commands], page 11. The HAC source file contains only declarations for a pair of e1of2 channels – the rest is set up in Verilog.

In the Verilog source, we connect the L and R channel rails with delay-elements. Using `$prsim_cmd`, we configure L as a source and R as a sink, as one normally would in an `hacprsim` session.

Here's the important part! The `channel-reset` and `channel-release` commands inject events into `hacprsim`'s event queue, however the host Verilog simulator is not aware of direct updates to that event queue! (This is also the case when you invoke `$prsim_cmd("set ...");`.) To notify the Verilog simulator and synchronize its main event queue, we need to invoke `$prsim_sync()` immediately after any command that affects the `hacprsim` event queue, including `channel-reset[-all]` and `channel-release[-all]`. Prior to `channel-release[-all]`, we also call `$prsim_sync()` to flush out any remaining events in `hacprsim` that have not caught up to the present time.

Until there is any other user-driven change to the `hacprsim` event queue, there should be no need to call `$prsim_sync()` again.

6.5 Hierarchical co-simulation

The final example demonstrates how to co-simulate with Verilog placeholder definitions in HAC. This allows one to co-simulate any Verilog definition with circuits *anywhere* in the instance hierarchy in HAC.

Suppose you have some Verilog library that contains definitions that you wish to connect in HAC but simulate in Verilog.

```
// "lib.v"
module FLIPFLOP(d, q, clk);
  in d, clk;
  out q;
...
end module
```

In HAC one would define a placeholder definition:

```
defproc FLIPFLOP(bool d, q, clk) { }
```

and you could instantiate FLIPFLOPs anywhere and everywhere in the circuit hierarchy. It would be extremely tedious to connect each HAC instance to a corresponding Verilog instance by hand, using just $to_prsim and $from_prsim calls.

This problem is solved by taking advantage of modularity of Verilog module definitions.

```
// "lib.v-wrap"
module HAC_FLIPFLOP;
  wire d, q, clk;
  parameter prsim_name="";
  reg [64*8:1] verilog_name;
  FLIPFLOP dummy(d, q, clk);
  initial begin
  #0
  if (prsim_name != "") begin
    $sformat(verilog_name, "%m");
    $from_prsim({prsim_name, ".d"}, {verilog_name, ".d"});
    $from_prsim({prsim_name, ".clk"}, {verilog_name, ".clk"});
    $to_prsim({verilog_name, ".q"}, {prsim_name, ".q"});
  end
  end
endmodule
```

Every Verilog instance of HAC_FLIPFLOP instantiates a local FLIPFLOP and automatically connect its ports to hacprsim, provided you set the *prsim_name* parameter using defparam. The explicit #0 timestamp guarantees that the functions are not called until *after* initial statements, when the HAC object file is supposed to be loaded. For every instance of FLIPFLOP in the HAC hierarchy:

```
module TOP;
...
HAC_FLIPFLOP __0();
defparam __0.prsim_name="x.y[0]";
HAC_FLIPFLOP __1();
defparam __1.prsim_name="x.y[1]";
HAC_FLIPFLOP __2();
defparam __2.prsim_name="x.q.w";
...
endmodule
```

How do you find every instance of FLIPFLOP in the HAC hierarchy? The current method is to query the *allocate*-compiled object file:

```
$ hacobjdump inst_foo.haco-a > inst_foo.objdump 2>&1
$ sed -n  '/Globally allocated state/,$p' inst_foo.objdump | \
    sed -n '/\[global process entries\]/,/\[global.*entries\]/p' | \
    grep "^[0-9]" | cut -f5-6 > inst_foo.processes
```

The resulting 'inst_foo.proceses' lists every unique process with its type[1]. The user can construct a database of known Verilog types to extract the canonical prsim names of all

[1] This would much more elegant using the Scheme interface of hacguile.

Verilog wrapper processes in the HAC namespace. From there, it is trivial to generate top-level Verilog instantiations of all wrappers.

What if there are a hundred definitions across Verilog libraries? Fortunately, we have already written such script to convert entire libraries into wrapper definitions.

```
$ bindir/wrap_verilog_modules_to_hacprsim.awk lib.v > lib.v-wrap
```

Limitations: arrays and buses (in progress), template parameters (not yet).

To use the output file '`lib.v-wrap`' in your top-level Verilog file:

```
`include "lib.b"
`include "lib.v-wrap"

module TOP;
// instantiate Verilog wrappers, set their prsim_names
...
endmodule
```

If you've copied the example directory already, the target that demonstrates this process is '`and_tree.vx-log`'. The Verilog library is '`standard.v`' which begets '`standard.v-wrap`'. '`and_tree.hac`' instantiates and connects a tree of AND gates which are defined in the Verilog library. '`and_tree.v`' includes the library and its wrapper, and instantiates top-level instances to connect to `hacprsim`. The simulation stimulus just toggles some of the input signals to the AND-tree.

6.6 VPI Hacker's Guide

This section is for documenting development of the VPI `hacprsim` module. The module is compiled and installed as a dymamic shared library, a run-time loadable plug-in. The advantage is that the simulation executable need not link a static copy of the library; nor does it need to be recompiled for non-ABI changing updates to the library.

How it works...

event queues

breakpoints

callbacks

synchronization

7 Tips

This section contains the collected wisdom of users of the simulator. Think of this as an FAQ.

7.1 Interactive mode

Readline completion and history Hopefully, you or your package has built the HACKT tools with the GNU Readline library. Readline provides numerous line-editing, history, and completion features that will make the interactive mode much more efficient. The most basic feature is cycling through history with the up and down arrow keys. Consult the Readline documentation for details, See section "Readline" in *The GNU Readline Library*.

7.2 Scripting

Long simulations. Q: If I have a simulation that is supposed to run infinitely, deadlock-free, with the `cycle` command how do I check it non-interactively? (By now you've figured out that in interactive mode, you can interrupt the simulation with `Ctrl-C` to give you back the command prompt, or equivalently, send the process a `SIGINT` signal.)

A: Instead of `cycle`, use `advance` or `step` or `step-event` to run for a finite time. Upon completion of the command, however, one typically wants to verify that the system did not deadlock. `assert-queue` will error out if the event queue is empty.

Checkpointing. You can manually save a checkpoint at any time using the `save` command, and restore to using the `load` command. You can also have the simulator automatically take a checkpoint upon exit, regardless of the exit status by using:

```
autosave on FILENAME
```

This is particularly useful for analyzing failed simulations that stop in some unexpected manner. After a failed run, you have a checkpoint that takes you right to the moment of failure for further examination.

Command Index

W

Z

Concept Index