YosysHQ MCY

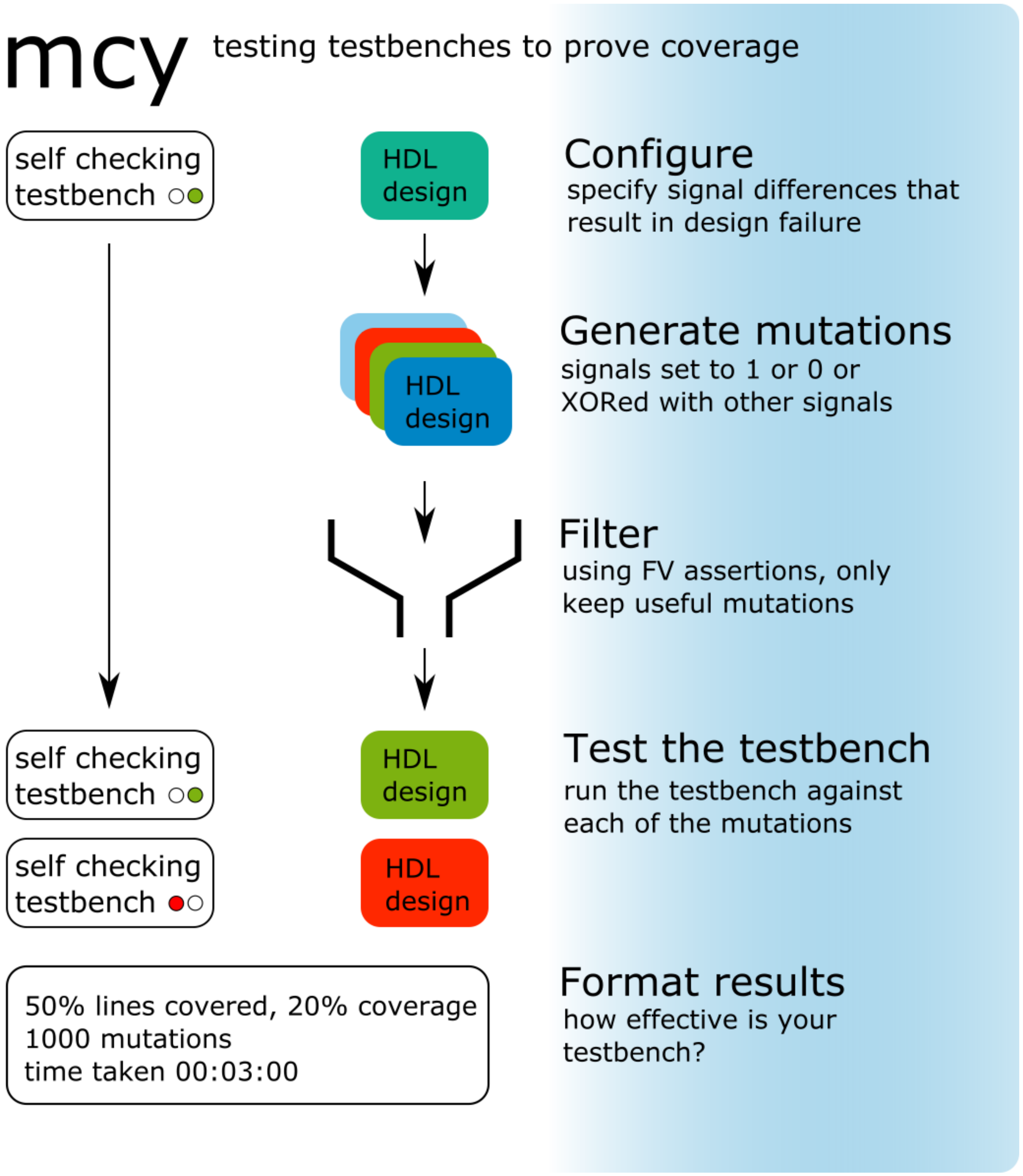
YosysHQ GmbH

Jan 29, 2024

CONTENTS

MCY is a new tool to help digital designers and project managers understand and improve testbench coverage.

If you have a testbench, and it fails, you know you have a problem. But if it passes, you know nothing if you don't know what your testbench is actually testing for.


mcy
testing testbenches to prove coverage
self checking testbench
HDL design
Configure
specify signal differences that result in design failure
HDL design
Generate mutations
signals set to 1 or 0 or XORed with other signals
Filter
using FV assertions, only keep useful mutations
self checking testbench
HDL design
Test the testbench
run the testbench against each of the mutations
self checking testbench
HDL design
50% lines covered, 20% coverage
1000 mutations
time taken 00:03:00
Format results
how effective is your testbench?

Given a self checking testbench, mcy generates 1000s of mutations by modifying individual signals in a post synthesis netlist. These mutations are then filtered using Formal Verification techniques, keeping only those that can cause an important change in the design's output.

All mutated designs are run against the testbench to check that the testbench will detect and fail for a relevant mutation. The testbench can then be improved to get 100% complete coverage.

CHAPTER

ONE

GETTING STARTED

1.1 Dependencies

MCY requires `python` >= 3.5(?).

The example projects in the `examples` directory additionally require Icarus Verilog and SBY.

1.2 Installation

To install mcy, run:

```
make
sudo make install
```

1.3 Example

To check that everything is set up correctly, you can run an example project:

```
cd examples/bitcnt
mcy init
mcy run -j8
```

CHAPTER TWO

METHODOLOGY

This section describes the principles underlying MCY which should be kept in mind during project setup.

The idea behind MCY is to test how good your test suite is at detecting errors in your design, by deliberately introducing modifications (small “mutations” that change the value of a single bit in the design) and checking if they are caught. However, it is not guaranteed that a mutation will break the design: if e.g. the value of the bit only changes on cycles where an associated `valid` signal is low, the design still functions as intended and the test suite would be correct in passing it.

Prerequisite: your test suite is passing on the original design

It is not possible to measure mutation coverage for a failing design. If your original design is already broken, breaking it further by introducing a mutation cannot make a detectable change to the test status.

Many coverage tools would simply make you pick an arbitrary “target coverage rate” lower than 100% based on your intuition or experience, to account for those cases where the mutation does not make a relevant difference. With MCY, you will set up a formal equivalence check that serves as the “truth” about whether a mutation affects the functioning of your design. If the equivalence check can find a valid combination of inputs for which the output of the mutated design differs from the original, but the test suite passes the design anyway, then the test suite is not set up to catch this type of incorrect behavior. The goal then becomes to reach 100% coverage.

Equivalence check

Setting up the formal equivalence check requires a little upfront effort, as you will need to specify which variations in input and output signals are a functional difference according to your design specification, but this is generally not too hard if your interfaces are well specified. In contrast to full formal verification, it is much easier to say that, for example, if the mutated design outputs `5` where the original returns `3`, at least one of the values must be wrong, than it is to write formal properties that calculate that the result *should* be `42` based on inputs that might have been entered over several previous cycles.

However, for the core MCY engine there is no difference in how to run this test compared to the other tests in your test suite, so there is no special handling baked into the configuration file format to guide you in setting your project up correctly. If a single author is writing both the test suite and the equivalence test, the distinction between the two may become blurred. When setting up the equivalence test and especially when writing the `[logic]` section of the configuration, it is crucial that you keep this core idea at the forefront of your mind:

Fundamental Principle

The equivalence check is the reference standard against which your test suite is compared. It will tell you whether there is a relevant behavior change to be found in a mutated design, and your test suite is judged by whether it accurately detects this change or not.

Note: This is **not** saying that a design that passes equivalence check is correct, or a design that fails equivalence is buggy. Since most likely the original design is not perfectly bug-free, being equivalent to it cannot tell you anything about correctness.

What this serves to determine is whether **your testbench is capable of detecting the kind of changes in behavior** that the mutation introduced.

When starting an MCY project, it is recommended to start by using this "naive" logic directly, and keeping the number of mutations small. As you gain confidence that the setup is working, you can progressively add optimizations that let you run larger numbers of mutations.

The `bitcnt` example shows an implementation of the naive logic:

```
sim_okay = result("test_sim") == "PASS"
eq_okay = result("test_eq") == "PASS"

if sim_okay and not eq_okay:
    tag("UNCOVERED")
elif not sim_okay and not eq_okay:
    tag("COVERED")
elif sim_okay and eq_okay:
    tag("NOCHANGE")
elif not tb_okay and eq_okay:
    tag("EQGAP")
else:
    assert 0
```

This unconditionally runs both the test suite (which consists of a single testbench `test_sim`), and the equivalence check `test_eq`. The first three cases are the expected ones: if the equivalence check says there is a relevant difference in behavior between the original and mutated designs, then if the testbench catches it the mutation is covered, and if it fails to catch it it is not covered. If the equivalence check and the testbench both agree that the original and mutated designs behave the same way, the mutation is not useful for evaluating testbench quality and can be ignored.

Keep an eye out for that fourth category! In a correctly set up project, this should never happen. If the equivalence check finds that the mutation does not affect the behavior, but the testbench finds a bug, one of them is wrong. Either the equivalence check is failing to find relevant behavioral changes, or the testbench is failing compliant designs. Investigate these cases thoroughly. If the behavior with the mutation in question is OK, adjust the testbench to pass it. If the behavior is not OK, modify the equivalence check to detect the problematic change. Only proceed to optimize the logic once you are not observing any cases of the fourth type.

For performance reasons, you will often want to end up at a logically equivalent but very different flow in the `[logic]` section, and this can be a source of confusion when looking at more complex, optimized examples. There are two assumptions that can be used to avoid running some of the tests, and still arrive at the same result:

Assumption 1: The equivalence check is correct.

If the equivalence check fails, the mutation changes the functioning of the design, if it passes, there is no functional difference between the original and the mutated design.

In particular, this means that if the equivalence check passes, there is no need to run any of the test suite on this mutation.

Since the mutation does not have any relevant effect, it cannot tell us anything about the test suite's ability to detect bugs.

Assumption 2: The test suite will not return false failures for a working design.

Ensure that your testbenches are tolerant of small changes in behavior that are not relevant to correctness, such as e.g. the values of output bits that are unused in certain modes.

With this assumption, if any test in the suite returns FAIL, there is no need to run either other tests from the test suite or the equivalence check: the mutation is covered. This allows you to run short unit tests first, and determine the result quickly for the more "obviously buggy" mutations.

Any case of `EQGAP` violates at least one of these assumptions. This is why we recommend testing for this case in the beginning, to gain confidence that the assumptions hold. It is also possible to use `rng()` to check that the assumption holds only in a random subset of mutations, to balance execution time concerns with confidence in the test's correctness.

In general, to get the best performance, always run the shortest test first. If you have unit tests, run these first, and integration tests after, in increasing order of runtime. The equivalence test will often be one of the longer-running tests, especially for the worst-case where the designs are equivalent.

Example

If your test suite contains three testbenches of increasing run length: a unit test `test_unit`, an integration test `test_sys` and a hardware-in-the-loop test `test_hw`, and the equivalence check `test_eq` generally takes longer than `test_sys` but less time than `test_hw`, then applying the two assumptions to finish early whenever possible would lead to the following logic:

```
if result("test_unit") == "FAIL":
        tag("COVERED")
        return

if result("test_sys") == "FAIL":
        tag("COVERED")
        return

if result("test_eq") == "PASS":
        tag("NOCHANGE")
        return

if result("test_hw") == "FAIL":
        tag("COVERED")
        return

tag("UNCOVERED")
```

As you can see, the underlying reasoning is no longer obvious from this code. Always keep the fundamental principle and the two assumptions in mind while working with MCY!

CHAPTER

THREE

COMMAND REFERENCE

mcy provides the following commands:

```
> mcy help

Usage:
        mcy [--trace] init [--nosetup]
        mcy [--trace] reset
        mcy [--trace] status
        mcy [--trace] list [--details] [<id_or_tag>..]
        mcy [--trace] run [-jN] [--reset] [<id>..]
        mcy [--trace] task [-v] [-k] <test> <id_or_tag>..
        mcy [--trace] source [-e <encoding>] <filename> [<filename>]
        mcy [--trace] lcov <filename>
        mcy [--trace] dash [<source_dir>]
        mcy [--trace] gui [--src <source_dir>]
        mcy [--trace] purge
```

All commands require the project configuration file `config.mcy` to be present in the current directory.

mcy init [-f] [–nosetup]
: This command initializes the mcy database. It runs the optional setup script from the `[setup]` section in `config.mcy` first, then prepares the design using the script from the `[script]` section, and generates a list of mutations conforming to the settings in the `[options]` section. It queues all mutations to be tested when `mcy run` is called. The command fails if the `database` directory exists. Run `mcy purge` to delete this directory if it is present, or pass `-f` to force overwriting the contents. The `--nosetup` option skips running the setup section. In combination, `mcy init -f --nosetup` allows re-initializing the project without deleting the files present in `database/setup`. This is useful when the configuration file was changed, but the setup script's output does not need to be re-generated.

mcy reset
: This command will reset various state. If the `size` parameter in the section `[options]` of `config.mcy` was increased, it will create additional mutations. It will re-run the tagging logic of the `[logic]` section and re-tag all mutations for which results are cached in the database. It queues the mutations for which results are not available to be tested when `mcy run` is called. It will also delete an existing `tasks` directory.

mcy status
: This command prints the status of the project. It will indicate the number of cached results and queued tests. If some results are available, it will also report the results in the format specified in the `[report]` section of `config.mcy`. The same status is also printed at the end of the commands `init`, `reset`, and `run`.

mcy list [–details] [<id_or_tag>..]
: This command prints the list of selected mutations and the tags applied to them. If the optional selection argument <`id_or_tag`> is not present, all mutations are listed. There can be multiple selection arguments, in which case

mutations matching any of the IDs or tags are listed. If `--details` is passed, it will additionally print the mutation command and the results cached in the database.

mcy run [-jN] [–reset] [<id>..]
This command executes the tests in the queue and any tests subsequently queued based on the results (for conditionally executed tests). The optional argument `-j N` allows up to `N` tasks to be executed in parallel. If `--reset` is passed, `mcy reset` will run first (potentially creating additional mutations or queueing more tasks). The optional selection argument <`id`>, of which there can be multiple, restricts mcy to run only tests on the matching mutation(s). (Tests for which results are available will not be re-run.)

mcy task [-v] [-k] <test> <id_or_tag>..
This command runs the test <`test`> on the mutations matching the ID or tag <`id_or_tag`>, of which there can be multiple. The test is executed even if the result is cached in the database. If the `-v` flag is passed, the output of the task execution is printed to stdout instead of the file `tasks/`<`uuid`>`/logfile.txt`. If `-k` is passed, the temporary task execution directory `tasks/`<`uuid`> is not deleted when the task finishes.

mcy source [-e <encoding>] <filename> [<filename>]
This command reprints the source file(s) <filename>, with annotations on the left side margin for each line of code with the number of mutations tagged COVERED or UNCOVERED (the name of the tags used is hardcoded). The number of COVERED mutations is displayed as a positive number, whereas UNCOVERED mutations are shown as negative numbers, similar to what is shown in `mcy gui`. Source files are printed from database cache, which is written when `mcy init` is called, so the version displayed is always the one the mutations were applied to. The optional `-e` parameter allows specifying the file encoding. (Python's standard encodings are supported, default is utf8.)

mcy lcov <filename>
This command prints coverage information in the format used by `lcov` and similar code coverage visualisation tools.

mcy dash
This command launches the dashboard server. Navigate to the address shown to access the dashboard where you can monitor the MCY status and download the mcy database for viewing with `mcy-gui`. This is intended to be used when running MCY on a remote server or as part of CI.

mcy gui
Launch the graphical explorer. The mcy gui allows navigating the list of mutations by source location, mutation id, or tag, and is the easiest way to access associated information. If you have downloaded a database from the mcy dashboard, invoke it directly as `mcy-gui` <`db_filename`>, as the mcy wrapper script will abort if `config.mcy` is not present.

mcy purge
This command removes all files produced by MCY, i.e. it deletes the `database` and `tasks` directories. Run this command before running `mcy init`.

The optional `--trace` argument to `mcy` is used for debugging and prints all queries performed on the database during execution to stdout.

CHAPTER

FOUR

CONFIGURATION FILE FORMAT

MCY relies on a configuration file named `config.mcy`. There are several sections in an mcy configuration file:

4.1 [options]

This section contains various configuration options for the mutation generation as well as the tags used.

`size <num>`
: The number of mutations to be generated.

`tags <tagname>..`
: The tags used in the `[logic]` and `[report]` sections.

`seed`
: Optional. Random number generator seed, can be set for reproducible mutation lists. (If you wish to know the randomly chosen seed of an existing set of mutations, it can be found in `database/mutations.ys`.)

`select <selection>`
: Optional. Selection of a subset of the design to restrict mutations to. See yosys -h select for a description for the selection pattern format.

> **Note:** The `select` keyword here is not the Yosys `select` command. The argument `<selection>` is used as the optional selection argument to the Yosys `mutate` command. While the selection pattern format is identical, you cannot use select subcommands such as `-module`.

Mutation generation options: MCY attempts to distribute mutations into all parts of the design. The documentation section *Mutation generation* describes the mutation generation algorithm, and how these values affect it.

`weight_cover`
: Optional. Weight for source location coverage. See *Mutation generation* for details. Default: 500

`weight_pq_w weight_pq_b weight_pq_c weight_pq_s`
: Optional. Weights for the per-design wire/bit/cell/src queues. See *Mutation generation* for details. Default: 100

`weight_pq_mw weight_pq_mb weight_pq_mc weight_pq_ms`
: Optional. Weights for the per-module wire/bit/cell/src queues. See *Mutation generation* for details. Default: 100

`pick_cover_prcnt`
: Optional. Chance that source location coverage influences which queue item is picked. See *Mutation generation* for details. Default: 80

4.2 [setup]

Optional. This section can contain a bash script to be executed at the beginning of `mcy init`, before the design is elaborated using the `[script]` section, which is useful for various setup tasks that only need to be done once for use in the design and/or tests. This script is executed in the base project directory (where `config.mcy` is located). If you would like the files created by the setup script to be managed by mcy and deleted when `mcy purge` is run, you can write them to the directory `database/setup`, which is created before the setup script runs.

Execution of this script can be skipped with `mcy init --nosetup`.

4.3 [script]

This section contains the Yosys script to be used to prepare the design to be mutated. Read in the necessary files with calls to `read` (one line per file), then call `prep -top <top_module>` to elaborate the design. (See yosys -h read for the options to the `read` command.)

4.4 [files]

This section lists the files used by the design that should be added to the database. All files read in the above script should be listed here.

4.5 [logic]

This section describes how the mutations should be tagged based on the results of one or more tests. It contains a python script making use of the predefined functions `result(testname)`, `tag(tagname)` and optionally `rng(n)`.

Valid arguments to `result(testname)` are names of tests defined in a `[test testname]` section. Its return value is the return value of the test, as written to `output.txt` in the test script execution. Both values are strings, so e.g. for a test defined as `[test sim]`, the function should be called as `result("sim")` and may return `"PASS"` or `"FAIL"`. Calling this function causes the test in question to be scheduled. This means that it is possible to execute a test conditionally on another result.

Valid arguments to `tag(tagname)` are tags defined in the `[options]` section under `tags`, again as strings. Calling this function causes the mutation to be tagged with this tag.

`rng(n)` takes a positive integer `n` and returns a pseudo-random number between `0` and `n`. It is affected by the `seed` set in `[options]`, or the seed chosen at random if this is not set.

A common combination of using these functions is the following:

```
[logic]
if (result("sim") == "PASS"):
        if (result("eq") == "PASS"):
                tag("NOCHANGE")
        else:
                tag("UNCOVERED")
else:
        tag("COVERED")
```

This causes the test `eq` to only be run if the test `sim` passes.

As this section can contain arbitrary python, the logic can also be defined in a separate file, and used with `import external_logic.py`.

4.6 [report]

This section contains the script to print the results. It can make use of the predefined function `tags(tagname)`, which returns the number of mutations tagged with the given tag.

Example:

```
[report]
if tags("COVERED")+tags("UNCOVERED"):
    print("Coverage: %.2f%%" % (100.0*tags("COVERED")/(tags("COVERED")+tags("UNCOVERED
↪"))))
```

4.7 [test <testname>]

This section defines a test. Details about how to set up tests can be found in *Writing a test script*.

`expect <result>..`
The expected return values of the test in question. (By convention, usually includes `PASS` and `FAIL`, although this is not mandatory). A return value not included in this list will cause the mcy run to be aborted immediately.

`run <command>`
How to run the test. `<command>` is executed in a temporary subdirectory created for the task, `tasks/<uuid>/`. MCY creates a file `input.txt` with a numbered list of mutations to be tested, and expects the results of the test to be written to `output.txt` after execution of `<command>` with the same number identifying the mutation.

`maxbatchsize <X>`
How many mutations to include in a single task. Default is 1. Increasing this number will cause MCY to add up to <X> lines to `input.txt` for each task.

CHAPTER

FIVE

WRITING A TEST SCRIPT

For each testbench in your test suite, you need to write a test script that MCY can call to run that testbench on a mutated design. MCY will create a temporary directory, place a file named `input.txt` with a numbered list of mutations in it, run your test script, and expect to find the return status of the testbench for each mutation in a correspondingly numbered list in a file named `output.txt`. (By default, the list will only contain a single entry, as this is the most straightforward to use. The parameter `maxbatchsize` can be set in the `[test]` section of `config.mcy` to increase the number of mutations included.)

The test script will usually consist of three steps: exporting the mutated source, running the testbench, and reporting the result.

5.1 Exporting the Mutated Source

The first step is to obtain the modified source that includes the mutation(s) listed in `input.txt`. The script `create_mutated.sh` makes this painless as long as your testbench can accept verilog or rtlil sources for the mutated module. When executing the test, MCY sets the variable `$SCRIPTS` to the path of the directory where you can find this script.

If you want to substitute a mutated verilog module with identical interface to the original, simply call it with no arguments:

```
bash $SCRIPTS/create_mutated.sh
```

This will result in a file named `mutated.v` containing a mutated module with the same name as the original module being created in the temporary directory where the test is executed (`task/<uuid>`).

If you want to use the `maxbatchsize` parameter to test multiple mutations in a single call of the test script, call the script with `-c`:

```
bash $SCRIPTS/create_mutated.sh -c
```

This will result in a file `mutated.v` with a module of the same name but with an extra input signal `mutsel`, which you can use to select which mutation to enable.

For more details about mutation generation, see *Mutation export options*.

5.2 Running the Testbench

Substitute the mutated module for the original in your testbench sources (usually by replacing the source file in the sources list with `mutated.v`). If you are testing a single mutation at a time and did not pass the `-c` argument to `create_mutated.sh`, you can now prepare and run your testbench as usual. The mutated module seamlessly replaces the original.

If you did enable multiple mutations to be included in the module, modify your testbench to add the `mutsel` input to the mutated module. If your testbench is compiled, add a way to pass the value of `mutsel` to the test at execution, e.g. via command line argument. The first column in `input.txt` is the number that selects the corresponding mutation. Run the testbench for each value appearing in `input.txt`.

For example, if using `iverilog` (from the `picorv32_primes` example):

```
iverilog -o sim ../../sim_simple.v mutated.v
while read idx mut; do
        vvp -N sim +mut=${idx} > sim_${idx}.out
done < input.txt
```

5.3 Reporting the Result

The results of the testbench run should be written to `output.txt`. Each line should be the number identifying the mutation followed by a status. The status can be an arbitrary string not containing whitespace; however, any value not listed with the `expect` keyword in the test section of `config.mcy` is considered an error and will cause the mcy run to be aborted. Commonly used return values are “PASS”, “FAIL” and sometimes “TIMEOUT”.

If only a single mutation is evaluated at a time, the associated number is always 1. In this case, simply write 1 followed by the outcome of the testbench to `output.txt`:

```
if $test_ok; then
        echo "1 PASS" > output.txt
else
        echo "1 FAIL" > output.txt
fi
```

If more than one mutation are evaluated in a run, each result should be on its own line, preceded by the corresponding value of `mutsel`. The order does not matter.

For the previous example with `iverilog`:

```
while read idx mut; do
        this_md5sum=$(md5sum sim_${idx}.out | awk '{ print $1; }')
        if [ $good_md5sum = $this_md5sum ]; then
                echo "$idx PASS" >> output.txt
        else
                echo "$idx FAIL" >> output.txt
        fi
done < input.txt
```

CHAPTER

SIX

WRITING THE EQUIVALENCE CHECK

For the equivalence check, not only do you need to write a test script similar to those for your test suite, but you will also need to write the miter circuit for the equivalence check itself.

6.1 Creating the miter circuit

The equivalence check is done with a miter circuit, which instantiates the same module (exported with the additional control input) twice, with mutsel values `0` (original) and `1` (mutated).

The contents of the equivalence check are specific to your design. As an example, let us consider a synchronous, resettable module named `example` with two inputs `in1` and `in2`, and an associated flow control signal `in_valid`, as well as an output `out` that has associated handshake control signals `out_valid` and `out_ready` (the latter being an input signal). The specification contains the usual stipulations that the values of `in1` and `in2` are ignored on cycles where `in_valid` is low, and the value of `out` is correct whenever `out_valid` is high and remains stable as long as `out_ready` is low.

First, instantiate the module twice, with `mutsel` bound to `0` for the reference module, and `mutsel` bound to `1` for the mutated module.

For any input of the modules, create two sets of inputs to the miter module. These inputs will be constrained using assumptions whenever necessary. For signals where the values can never diverge without affecting the behavior of the modules, such as e.g. the clock and reset signals, a single input connected to both modules is sufficient.

For outputs, create two sets wires to connect them. These wires will serve to compare the outputs to check if they are equivalent.

```
module miter (
        input clk,
        input rst,
        input ref_in1,
        input ref_in2,
        input ref_in_valid,
        input ref_out_ready,
        input uut_in1,
        input uut_in2,
        input uut_in_valid,
        input uut_out_ready
);

        wire ref_out;
        wire ref_out_valid;
```

(continues on next page)

(continued from previous page)

```
        wire uut_out;
        wire uut_out_valid;

        example ref (
                .clk(clk),
                .rst(rst),
                .in1(ref_in1),
                .in2(ref_in2),
                .in_valid(ref_in_valid),
                .out(ref_out),
                .out_valid(ref_out_valid),
                .out_ready(ref_out_ready),
                .mutsel(1'b0)
        );

        example uut (
                .clk(clk),
                .rst(rst),
                .in1(uut_in1),
                .in2(uut_in2),
                .in_valid(uut_in_valid),
                .out(uut_out),
                .out_valid(uut_out_valid),
                .out_ready(uut_out_ready),
                .mutsel(1'b1)
        );

endmodule
```

Next, add assumptions that inputs are identical whenever the module behavior should be influenced by the input value. For the flow control inputs `in_valid` and `out_ready`, this is at any time. For `in1` and `in2`, only assume identical inputs when `in_valid` is high - the specification says that the value should not matter otherwise.

Add assertions that the outputs are equivalent whenever the specification indicates that a value should be valid. For the flow control output `out_valid` this is at any time, for the output `out` only when `out_valid` is high.

If your module contains internal state that can affect results, you may need to also constrain the initial state, e.g. by assuming that the reset is enabled in the first cycle.

```
module miter (
        ...
);

        ...

        example ref (
                ...
        );

        example uut (
                ...
        );

```

(continues on next page)

(continued from previous page)

```
	initial assume (rst);

	always @(posedge clk) begin
		if (!rst) begin
			assume (ref_in_valid == uut_in_valid);
			assume (ref_out_ready == uut_out_ready);
			if (ref_in_valid) begin
				assume (ref_in1 == uut_in1);
				assume (ref_in2 == uut_in2);
			end

			assert (ref_out_valid == uut_out_valid);
			if (ref_out_valid) begin
				assert (ref_out == uut_out);
			end
		end
	end

endmodule
```

Be as precise as possible in your assume and assert statements! If you overconstrain, you are likely to declare that some mutations do not affect the behavior of the module when there are valid circumstances in which they would change the behavior, but you have excluded them from consideration. If you underconstrain your design, you will probably find that the equivalence check fails to find the original module equivalent to itself, but this is much easier to detect: Simply run the equivalence check on the no-change mutation, which will always be included in the mutation database with ID 1.

6.2 Setting up the test script for the equivalence check

6.2.1 Mutation export

For the equivalence check, it is recommended to test mutations individually (do not set the `maxbatchsize` parameter).

Call the `create_mutated.sh` script with -c to obtain a module where you can turn on and off the mutation using `mutsel`:

```
bash $SCRIPTS/create_mutated.sh -c
```

In general, using this way of exporting the original and mutated simultaneously makes it easier to implement the miter circuit as you do not have to worry about conflicting module names. When using SBY, it additionally offers the advantage that you can use the `fmcombine` optimization pass that analyzes the module and combines any logic that is not in the fanout cone of the mutation, and hence identical between the two module instances.

6.2.2 Running the equivalence check

Set up a script that verifies these assertions using a formal tool. For example, with SBY, you would first create a project file `test_eq.sby`:

```
[options]
mode bmc
depth 10
expect pass,fail

[engines]
smtbmc boolector

[script]
read_verilog -sv miter.sv
read_verilog mutated.v
prep -top miter
fmcombine miter ref uut
flatten
opt -fast

[files]
miter.sv
mutated.v
```

If using BMC, make sure to set the depth sufficiently high to fully explore any pipelines or state machines in the module. Also note the use of `fmcombine` which optimizes the model to remove redundant logic between the two modules.

Then you can use this to run the equivalence check with SBY:

```
ln -s ../../test_eq.sv ../../test_eq.sby .
sby -f test_eq.sby
```

6.2.3 Reporting equivalence

Finally, write the result of the equivalence check to the file `output.txt`, preceded by 1:

```
gawk "{ print 1, \$1; }" test_eq/status >> output.txt
```

CHAPTER

SEVEN

MUTATION EXPORT OPTIONS

The first step in any test script is to obtain the mutated module to run the test on.

There are three configurations that are commonly used:

1. Permanently enabled single mutation. Applying a single mutation without any control options results in a module with the same interface as the original module that can be substituted in a testbench without requiring any modifications. This is generally the most straightforward way to run the testsuite that is being evaluated.
2. Selectable single mutation. Applying a single mutation with a control signal results in a module with one additional input, `mutsel`. The mutation will be enabled if `mutsel==1` and disabled if `mutsel==0`. Generally used in the equivalence test, where the same module can be instantiated twice with different `mutsel` inputs. This avoids problems with module name collision between the original and mutated version, and allows the use of `fmcombine` which can speed up the formal property solver.
3. Multiple selectable mutations. Multiple mutate commands are applied to a module simultaneously. The resulting module will have an additional `mutsel` input and can exhibit a different mutation for each value of `mutsel`. This is useful if the testbench being evaluated has large setup costs, e.g. for compilation. However, it requires modifying the testbench to drive the `mutsel` input from an argument passed at execution time.

By default, each task in MCY will only test a single mutation. To enable multiple mutations to be tested in a single task, set the `maxbatchsize` parameter in the corresponding `[test]` section in `config.mcy` to a value larger than 1 and small enough to be representable with the number of bits of the `mutsel` signal (set in the `mutate -ctrl <width> <value>` option).

7.1 The `create_mutated.sh` script

If your test falls within the common use cases, you can use the script `create_mutated.sh` to export the modified module. This is generally more convenient than writing your own script.

The script is made to work with the files set up by mcy in the temporary task execution directory. MCY also sets up the environment variable `SCRIPTS` pointing to the directory where the scripts are installed. Call it in your test script as follows:

```
bash $SCRIPTS/create_mutated.sh
```

By default it will produce a `mutated.v` with permanently enabled mutation (case 1). Pass the `-c` (or `--ctrl`) option to add the `mutsel` input for cases 2 and 3.

7.2 The task mutation list `input.txt`

For each test, MCY will create a file named `input.txt` that contains lines of the format:

```
<idx> <mutate command (without -ctrl)>
```

The index identifies the mutation within the current batch. If `maxbatchsize` is not set for this task, mutations will be tested one at a time, so the file will contain a single line and `<idx>` will always be `1`. For the bitcount example, it might look like this:

```
1 mutate -mode cnot0 -module bitcnt -cell $and$bitcnt.v:53$263 -port Y -portbit 23 -
↪ctrlbit 57 -src bitcnt.v:53
```

Note that the indices always go from `1` to `maxbatchsize` within a task. They are different from the mutation IDs displayed by `mcy list` or the GUI, do not confuse the two!

7.3 Writing a custom mutation export script

If `create_mutated.sh` is insufficient for your use case, you may need to write a custom script to create the mutated sources. This script should take the mutation commands from `input.txt` and call yosys to apply the mutations and export the mutated source to the right format. The easiest way to do this is to first write the yosys commands to a `.ys` script file and then run it.

The yosys script (`mutate.ys` in this example) that you need to generate should be structured like this:

- Read the intermediary file containing the elaborated design:

  ```
  echo "read_verilog ../../database/design.il" > mutate.ys
  ```

- Apply the mutate command(s) found in `input.txt`.

  If there is only one mutation, and you do not wish to create a `mutsel` input, enter the command as-is, just remove the leading `1`, e.g. like this:

  ```
  cut -f2- -d' ' input.txt >> mutate.ys
  ```

  If you do wish to add a `mutsel` input to the design, you need to add the `-ctrl` parameter to the `mutate` command's arguments:

  ```
  while read -r idx mut; do
    echo "mutate -ctrl mutsel 8 ${idx} ${mut#* }" >> mutate.ys
  done < input.txt
  ```

  This code snippet works for one or many mutations in `input.txt` thanks to the `while` loop. If there are multiple mutations to be applied, it will add all the mutate commands to the script. Always make sure to add the control input when there are multiple mutations!

- Optionally, add any other yosys commands you wish to use to transform the design to work with your testbench. For example, if you want to run the design on hardware, you may synthesize it here:

  ```
  echo "synth_ice40 -top top_module" >> mutate.ys
  ```

  You can also change the name of the module if needed:

```
echo "rename top_module mutated" >> mutate.ys
```

- Finally, export the design to a format that can be used by your testbench. In the example of the hardware test, this might be json:

```
echo "write_json mutated.json" >> mutate.ys
```

After generating the script, execute it with yosys:

```
yosys -ql mutate.log mutate.ys
```

7.4 The Yosys `mutate` command

MCY's mutation capability is backed by the Yosys `mutate` command. This command has two functions: generating a list of mutations for a design, and applying a mutation to a design. For most users, it is not necessary to touch these internals, but understanding the inner workings of MCY may be relevant in advanced use cases.

7.4.1 Mutation generation

If the `-list <N>` argument is given, `mutate` will generate a list of `<N>` mutations. The mutation generation algorithm tries to satisfy a variety of coverage heuristics, in an effort to avoid systematically leaving certain parts of a design untested.

The default weights should provide a decent distribution of mutations for most code bases. If you wish to fine tune the selection, you can influence the algorithm by setting any of the mutation generation options in `config.mcy`. The decision logic works as follows:

A source location coverage scoring algorithm is picked with weight `weight_cover`. This algorithm assigns scores to mutations based on the source locations, with locations that have fewer mutations associated so far scoring higher. (One of) the mutation(s) with top score is picked.

Alternatively, mutations can be associated with wires, wire bits, cells, and source locations. Not all mutations have all associations (e.g., some wires cannot be traced back to specific source lines), but a single mutation can appear in multiple lists. There are 8 candidate lists:

- grouped by wire
- grouped by wire bit
- grouped by cell
- grouped by source location
- grouped by module, then by wire
- grouped by module, then by wire bit
- grouped by module, then by cell
- grouped by module, then by source location

(The second set of lists ensure that even if a module is very small with respect to the number of wires/wire bits/cells/source locations, it will not be overlooked.)

One of these lists is chosen to sample a mutation from with weights `weight_pq_w`, `weight_pq_b`, `weight_pq_c`, `weight_pq_s`, `weight_pq_mw`, `weight_pq_mb`, `weight_pq_mc`, `weight_pq_ms` respectively. Once a list is chosen, there is a pick_cover_prcnt chance that source location coverage score is used to influence which mutation is sampled, otherwise all mutations in the list are considered equally.

The output of mutate -list is a list of mutate commands that can be used to apply the generated mutations to the design. MCY generates these and stores them in the database when mcy init is run, and provides one (or several, if the maxbatchsize parameter is set for this test) in the file input.txt in the temporary folder when executing a task (via mcy run or mcy task).

7.4.2 Applying a mutation

If the -mode <mode> argument is given, mutate will apply a mutation to the current design. Most of the other parameters (-module, -cell, -port, -portbit, -ctrlbit, -wire, -wirebit, -src) serve to identify the element of the design to be modified and simply need to be copied from the file input.txt provided by MCY.

There is one optional parameter of interest, and that is -ctrl <name> <width> <value>. By default (without -ctrl), the mutate command will replace the element to be mutated with the modified version. The resulting module is identical to the original except for this modified element. If -ctrl is specified, the mutate command will instead add a control circuit to enable the mutation at will. It creates an additional input port to the modules, with the name <name> given in the command, and of width <width>. If this input signal is set to the value <value> specified, the mutation is enabled, otherwise it is disabled. The value chosen must be non-zero, as 0 is reserved for the unmodified behaviour of the design. Multiple mutations can be added to the same design by running several mutate commands with the same -ctrl <name> <width> arguments and a different <value>. This results in a design whose behaviour can be switched between different mutations by changing the value of this input signal.

The create_mutated.sh script wraps these two uses of mutate, with the -c/--ctrl flag causing the -ctrl argument to be passed to mutate.