Checking Typos in Simulation Command Line Plusargs

Thinh Ngo

Broadcom Corporation
Austin, Texas USA

www.broadcom.com

ABSTRACT

Modern tests are configured internally via constrained random configuration objects and externally via simulation command line plusargs. Various test intents and verification effects are realized via this combined approach. Typos in plusargs can completely void the intended test configurations and verification effects resulting in false passes and unverified features. Therefore, it is important to check the plusargs associated with each test and to fail the test if they are mistakenly specified. The following procedure can be used to implement such a checker:

1. *Store the expected plusargs used inside every test by creating/calling a function to register each expected plusarg*
2. *Retrieve the plusargs used by every test during simulation by calling uvm_cmdline_processor.get_args*
3. *Check that the plusargs used by every test match the expected plusargs stored in step 1 otherwise fail the test.*

Table of Contents

1 Introduction

There are two types of typos. The first type of typo results in a meaningless word and the second type results in a different meaningful word. In verification, both these types of typos have various effects, depending on their locations; these range from compile errors, simulation errors, false fails, false passes or unverified features. A typo in a comment would result in a trivial consequence; whereas a typo in a plusarg could result in unverified features. In this paper we address typos in plusargs, their adverse effects and how to check them.

2 Adverse Effects of Plusargs with Typos

Modern tests are configured via randomized configuration objects within each test which in turn pass down to sequences and transactions. Additionally, plusargs in simulation command lines are used to provide further constraints to the randomization of test configuration objects. Since plusargs are user-defined entities, they cause no compile or simulation errors when they contain typos. However, they can cause false passes and unverified features. Plusargs with typos completely void the verification effects of the intended plusargs. The resulting verification effect is like being without the plusarg or with another plusarg. This can lead to false passes and unverified features (e.g. intended verification of the intended plusargs).
Furthermore, plusargs should be test-specific. Plusargs created for one test would not have any effect on other tests. The result of misused plusargs (e.g. 2nd type of typo) would be none of desired verification effects, possibly leading to unverified features. Table 1 summarizes the above discussion.

Typo Type	Plusarg Consequence	Likelihood	Test Consequence	Verification Consequences w/o Checker	w/ Checker
1	unlike any plusarg of any test	most	w/o the intended plusarg	false pass - posible unverified features	Failed
2	another plusarg of the same test	often	w/o the intended plusarg	false pass - posible unverified features	Passed
2	another plusarg of a different test	often	w/o the intended plusarg	false pass - posible unverified features	Failed

Table 1.

Coverage would help detect unverified features but would likely be determined at a later time during the coverage analysis stage. It would be better to prevent them by immediately failing tests with typos in plusargs. In the next section, we present a flow to fail tests with typos in their plusargs.

3 Checking Typos in Plusargs

A test with typos in its plusargs should be considered as an errornous test - since it does not accomplish what it is intended to do. Consequently, such a test should fail. Below is the flow to implement a checker to fail tests with typos in their plusargs.

3.1 Checker Flow:

1. Register expected plusargs used inside every test.
2. Retrieve plusargs used by every test during simulation by calling uvm_cmdline_processor.get_args.
3. Check that plusargs used by every test match the expected plusargs stored in step 1 otherwise fail the test.

3.2 Detailed Implementation:

1. Create/Call a function init_plusargs_db to store the default arguments (i.e. log file name, incdir) and common plusargs (i.e. in the testbench environment) in plusargs_db in the build_ph of the base test.
2. Create/Call a function register_plusarg to store each expected plusarg in a test into plusargs_db created in Step 1 when adding a new plusarg in the build_ph of the base test or derived tests.
3. Retrieve actual plusargs used by every test on the simulation command lines by calling uvm_cmdline_processor.get_args after calling init_plusargs_db in the build_ph of the base test.
4. Create/Call a function adjust_plusargs to remove the plusarg values (i.e. "+UVM_TESTNAME=" instead of "+UVM_TESTNAME=default_test") after retrieving actual plusargs in the build_ph of the base test.
5. Create/Call a function check_plusargs to compare the expected plusargs in plusargs_db with actual plusargs in Step 4 in the end_of_elaboration_ph of the base test.

4 Example Codes

4.1 Codes in the base_test

There are four functions to be implemented in the base_test: init_plusargs_db, register_plusarg, adjust_plusargs and check_plusargs. The first three are called in the build_phase and the last one is called in the end_of_elaboration_phase.

4.1.1 Function init_plusargs_db:

This function is used to initialize the plusargs database, named plusargs_db, with default arguments. These arguments are common to all tests, generated by a simulation script and displayed at the top of the simulation log file. They are simulation directives for the simulator (i.e. –assert noposttproc) and plusargs for the testbench environment (i.e. clocks).

```
class base_test extends uvm_test;

string plusargs_db[string];
string plusargs_da[];
string Str, args_que[$];
```

```
uvm_cmdline_processor cmdline =
  uvm_cmdline_processor::get_inst();

function void init_plusargs_db();
  `uvm_info("init_plusargs_db", "Entering ...", UVM_NONE);
  plusargs_db["./simv"] = 1;
  plusargs_db["+vcs+lic+wait"] = 1;
  plusargs_db["+vmm_log_nofatal_at_1000"] = 1;
  plusargs_db["+vmm_log_nowarn_at_200"] = 1;
  plusargs_db["-assert"] = 1;
  plusargs_db["nopostproc"] = 1;
  plusargs_db["+ntb_random_seed_automatic"] = 1;
  plusargs_db["+UVM_TESTNAME="] = 1;
  plusargs_db["+incdir+../sim_files"] = 1;
  plusargs_db["-l"] = 1;
  plusargs_db["logs/runsim.log"] = 1;
endfunction
```

4.1.2 Function register_plusarg:

This function is used to store every test's plusarg in plusargs_db and is called by every test with additional plusargs.

```
function bit register_runarg(string runarg);
  `uvm_info("register_plusargs", $sformatf("runarg: %s",
    runarg), UVM_NONE)
  plusargs_db[runarg] = 1;
endfunction // register_plusargs
```

4.1.3 Function adjust_plusargs:

This function is used to seperate the plusarg strings identity from the pluarg values in plusargs_db. The application of this function is plusarg-specific – whether misused values should be considered or not.

```
function adjust_plusargs();
  string cpy;
  `uvm_info("adjust_plusargs", "Entering ...", UVM_NONE);

  args_da = args_que;
  foreach ( args_da[i] ) begin
    cpy = args_da[i];
    foreach ( cpy[j] )
      if (cpy[j] == "=") begin
        cpy = args_da[i].substr(0, j);  //copy upto "="
        break;
      end
    `uvm_info("adjust_plusargs", $sformatf("runarg: %s cpy %s",
```

```
      args_da[i], cpy), UVM_NONE)
    args_da[i] = cpy;
  end
endfunction
```

4.1.4 Function check_plusargs:

This function is used to check plusargs used in every test match the expected plusargs. Test fails if there is any mismatches.

```
function bit check_plusargs();
  `uvm_info("check_plusargs", "Entering ...", UVM_NONE);
  foreach( args_da[j] ) begin
    `uvm_info("check_plusargs", $sformatf("runarg: %s",
      args_da[j]), UVM_NONE)
    if (!plusargs_db.exists(args_da[j]))
      `uvm_error("check_plusargs", $sformatf("runarg: %s does
        not exist", args_da[j]))
  end
endfunction
```

4.1.5 Function calls:

Function init_plusargs_db, adjust_plusargs and register_plusarg are called in the build_phase and function check_plusargs is called in end_of_elaboration_phase:

```
function void build_phase(uvm_phase phase);
  super.build_phase(phase);
  `uvm_info("build_phase", "Entering...", UVM_LOW)

  init_plusargs_db();

  register_plusarg("+has_functional_coverage=");
  if (cmdline.get_arg_value("+has_functional_coverage=", Str))
  begin
    has_functional_coverage = Str.atoi();
    ...
  end
  …
  cmdline.get_args(args_que);

  for (int i=0; i<args_que.size(); i++)
     `uvm_info("build_phase", $sformatf("arg %0d = %0s", i,
       args_que[i]), UVM_NONE)

  adjust_plusargs();

  `uvm_info("build_phase", "Exiting...", UVM_LOW)
  endfunction // build_phase
```

```
function void end_of_elaboration_phase(uvm_phase phase);
  `uvm_info(get_type_name(), "In end_of_elaboration_phase",
UVM_NONE)
  check_plusargs();
  uvm_top.print_topology();
  factory.print();
endfunction // end_of_elaboration_phase
...
endclass
```

4.2 Codes in a test with additional plusargs

This test is extended from the base_test with an additional plusarg "`sequence_type`". It calls register_plusarg in the build_phase.

```
class sequence_test extends base_test;
...
function void build_phase(uvm_phase phase);
  super.build_phase(phase);

  register_plusarg("+sequence_type=");
  if (cmdline.get_arg_value("+sequence_type=", sequence_type))
  begin
    `uvm_info("build_phase", $sformatf("sequence_type = %0s",
       sequence_type), UVM_LOW)
    case (sequence_type)
    "streaming" :
set_type_override_by_type(arbiter_base_sequence::get_type(),
arbiter_streaming_sequence::get_type());
...
    endcase
  end
endfunction // build_phase
...
endclass
```

5 Conclusion

This paper addresses a common issue in verification, namely typos in plusargs of simulation command lines. It discusses the possible adverse effect of unverified features.
A solution is presented to check the tests with typos in their plusargs and cause them to fail during simulation as a test error.

6 References

[1] Janick, B., Eduard, C., Alan H.and Andrew N. 2005. Verification Methodology Manual for SystemVerilog.