2024 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
MUNICH, GERMANY
OCTOBER 15-16, 2024

Improved Performance of Constraints

Milos Pericic, Veriest, New Belgrade, Serbia (*milosp@veriests.com*)

***Abstract*—Constraints in *SystemVerilog* language are useful for randomization. When there are connections between randomized variables, then constraints could be used for randomizing everything together. In a larger project, the number of connections can grow and expand, resulting in complex randomized test scenarios. In those cases, constraints should be written as good as possible, and their performance should be tested. In this paper, a few useful techniques will be covered for improving performance of constraints.**

Keywords—Constraints, SystemVerilog, Randomizations, Performance, Functional verification

I. Introduction

Constraints in *SystemVerilog* are resolved using internal constraint solver. The order of randomizing dependent values is under constraint solver domain and its own algorithm, but user who write them can also affect order. When dependencies between randomized values start growing, the way of writing constraints becomes crucial. Variables can be randomized in constraints by calling functions or tasks, and in that case the order of variables' randomization is important. If there are lots of randomizations and dependencies between them, then adding new randomization might be hard, because it could introduce more randomization dependencies. In that case, new randomization must be added exactly in correct order between other dependent randomization cases to work properly. To not think about randomization order of all randomization cases or when a lot of dependencies exist between the variables, then constraint blocks could be used to randomize everything together. Constraint blocks exist in objects where variables are randomized.

II. Related Work

The paper [2] shows how constraint blocks can be organized in multiple layers and written in objects (classes). The author from [3] reveals engineering formulas for constraint blocks and their manipulation during simulation by using containers (wrappers). In the work [4], various kinds of soft constraints are explained with practical examples for verification engineers. While the previous papers show techniques for improving and organizing constraint blocks, none of them are fully focused on constraint blocks' performances and run-time improvements. In this paper, multiple techniques will be described to accelerate the randomization time of constraint blocks.

III. Constraints Improvement

A. Engineering formulas definition

Changing parts of specification definitions can be made to simplify engineering formulas and randomizations. Because everything starts with definitions, some decisions can be assumed without considering real-life usage. If that is the case, then specific definitions or their parts could be simplified to resolve too complicated constraints. If that is not the case, then other techniques written below can be tried.

B. Order helpers

SystemVerilog solve-before construct can be useful for explicitly setting order of randomization. It helps constraint solver to decide which values to randomize first and to significantly reduce decision time for finding the correct randomization order of given constraints with solve-before construct. Single or multiple solve-before constructs can be used inside constraint. For example, *solve* A *before* B means first randomize value A and then randomize value B, in that order. If there are other solve-before constructs inside constraint, they are also taken into consideration and the final ordering decision is made.

Let us define a few variables: *sub*, *mac*, *req*, *resp* and *mid*. Type of tests are defined as enumerated type *test_type_t* with values *BLOCK* (block level tests), *TOP* (top level tests), *DIRECT* (direct tests) and *RANDOM*

(random tests). Solve-before construct is defined only for random variables with keyword *rand*. Non-random variables are not used in solve-before construct because they are not randomized. In the given example, bits *use_long_values* and *use_short_values* are fixed to default values and can be changed from any function or task for specific tests if needed and they are not used in solve-before construct. The same applies to constant integer value *LONG_FACTOR* which is fixed to 256. In constraint *packets_data_c*, variables *sub* and *mac* are randomized according to *use_long_values* and *test_type* variables. Since *use_long_values* and *use_short_values* variables are not random, they are not used inside solve-before construct because their values are evaluated before randomization starts. Random variable *test_type* is considered while randomizing *sub* and *mac* variables and it must be randomized first. Solve-before construct *solve test_type before sub, mac* means randomize *test_type* variable before *sub* and *mac* variables. Also, *req*, *resp*, *sub* and *mid* variables are considered while randomizing *mac* variable and they must be randomized first. Solve-before construct *solve req, resp, sub, mid before mac* means randomize *req*, *resp*, *sub* and *mid* variables before *mac* variable. If it is omitted, then *mac* variable could get wrong non-randomized values for *req*, *resp*, *sub* and *mid* variables.

```
rand bit [3:0]  sub;
rand bit [63:0] mac;
rand bit [9:0]  req;
rand bit [6:0]  resp;
rand bit [12:0] mid;

typedef enum bit[1:0] { BLOCK, TOP, DIRECT, RANDOM } test_type_t;
rand test_type_t test_type;

bit use_long_values = 1, use_short_values = 0;
const int LONG_FACTOR = 256;

constraint packets_data_c {
  solve test_type before sub, mac;
  solve req, resp, sub, mid before mac;

  if (use_long_values && test_type inside { TOP, RANDOM }) {
    sub inside { [2 : 14] }; sub % 2 == 0;
    mac == (req - resp) * LONG_FACTOR - sub + mid;
  }
  ...
}
```

C. *Function in constraint*

If constraint is implementing complex logic, then it can cause constraint solver slowness. Because syntax for constraints is specific, sometimes it is not straightforward to write such constraints. The simple but effective solution is to create a function which will implement that complex logic and return final value as result. Function would accept functional arguments as inputs and use them inside function for calculations. When the final value is returned from function, then it can be written to the value inside constraint. It also simplifies constraint definition and relaxes constraint solver.

```
rand int stl, pic, fir, mult, volt, len, rsize;
bit is_low = 0, is_fill = 1, not_empty = 1;

typedef enum bit[1:0] { ZERO, ONE, TWO, THREE } speed_t;
rand speed_t speed;

typedef enum bit { FAST, SLOW } prec_t;
rand prec_t prec;

typedef enum bit[1:0] { LEFT, RIGHT, UP, DOWN } dir_t;
rand dir_t dir;

rand real round, fir_con, fir_mul;
rand int vrange, eq, pulse, fir_trun;
```

```
constraint rsize_c {
  solve dir, stl, pic, fir, prec, speed, mult, volt, len before rsize;

  if ((is_fill || is_low) && (not_empty || (dir inside { LEFT, RIGHT }))) {
    if (speed != ZERO) {
      if (is_low) { pulse == volt; }
      else        { pulse == stl;  }

      if (mult >= 32) {
        eq == mult * pulse + 10 * pic + 2 * (len + 1) + 32;
      }
      else {
        eq == mult + 3 * pulse + 5 * pic + 12;
      }

      if ((prec == FAST) || (speed == ONE) || (speed == TWO)) {
        rsize == eq;
      }
      else {
        fir_con  == 0.8;
        fir_mul  == fir_con * fir;
        fir_trun == int'(fir_mul);

        (vrange >= eq      ) && (vrange <= fir_trun) ||
        (vrange >= fir_trun) && (vrange <= eq      );

        round == real'(vrange) + eq * 2;
        rsize == int'(round / 4.0) * 16 + (len + 1);
      }
    }
    ...
  }
  ...
}
```

Let us define a few variables: *stl*, *pic*, *fir*, *mult*, *volt*, *len* and *rsize*. Variable *rsize* is randomized in complex constraint *rsize_c*, while other variables are randomized similar, each in its own complex constraint. Configurations bits *is_low*, *is_fill* and *not_empty* are set to initial values, but they can be set from test directly before randomization

starts. There are three enumerated types defined, first for speed with type *speed_t* and values *ZERO*, *ONE*, *TWO* and *THREE*, second for precision with type *prec_t* and values *FAST* and *SLOW*, third for direction with type *dir_t* and values *LEFT*, *RIGHT*, *UP* and *DOWN*. Please note that the last few random variables *round*, *fir_con*, *fir_mul*, *vrange*, *eq*, *pulse* and *fir_trun* are only used as intermediate values for calculations inside constraint *rsize_c* and they are not required in solve-before construct like listed variables. Solve-before construct in constraint *rsize_c* has variables which are randomized in their own constraints.

It is not that only randomization for constraint *rsize_c* is complex, but also variables used inside constraint *rsize_c* listed in solve-before construct could be randomized similar, which means that big chains of complex randomizations are made. Also, constraints for other variables outside of the given constraint could add more complexity, especially if the total number of active constraints in the entire system is large. After everything is randomized together, it can slowdown constraint solver randomization a lot.

```
function int calc_rsize(int m_stl, int m_pic, int m_fir, speed_t m_speed, prec_t m_prec,
bit m_is_low, int m_mult, int m_volt, int m_len);
  int m_rsize, m_eq, m_pulse;

  m_pulse = m_is_low ? m_volt : m_stl;

  if (mult >= 32) begin
    m_eq = m_mult * m_pulse + 10 * m_pic + 2 * (m_len + 1) + 32;
  end
  else begin
    m_eq = m_mult + 3 * m_pulse + 5 * m_pic + 12;
  end

  if ((m_prec == FAST) || (m_speed == ONE) || (m_speed == TWO)) begin
    m_rsize = m_eq;
  end
  else begin
    real m_round, m_fir_con, m_fir_mul;
    int m_vrange, m_fir_trun;

    m_fir_con  = 0.8;
    m_fir_mul  = m_fir_con * m_fir;
    m_fir_trun = int'(m_fir_mul);

    m_vrange   = $urandom_range(m_eq, m_fir_trun);
    m_round    = real'(m_vrange) + m_eq * 2;
    m_rsize    = int'(m_round / 4.0) * 16 + (m_len + 1);
  end

  return m_rsize;
endfunction : calc_rsize
```

```
constraint rsize_c {
  solve dir, stl, pic, fir, prec, speed, mult, volt, len before rsize;

  if ((is_fill || is_low) && (not_empty || (dir inside { LEFT, RIGHT }))) {
    if (speed != ZERO) {
      rsize == calc_rsize(stl, pic, fir, speed, prec, is_low, mult, volt, len);
    }
    ...
  }
  ...
}
```

Constraint logic is moved to the function *calc_rsize*. Equal sign = is used normally like in any other function. Ternary operator *?* is used for quick *if-else* one-line condition and it is also allowed in function. Please note that *vrange* variable is randomized in previous example with relational operators >= and <= because system calls are not allowed in constraints, but in function they are possible. Also condition *(vrange >= eq) && (vrange <= fir_trun) || (vrange >= fir_trun) && (vrange <= eq)* is randomizing *vrange* variable between *eq* and *fur_trun* values, and during randomization one will be minimum, one maximum value, so relations >= and <= are written in both directions. But in function, *$urandom_range* system call can be used with any order of minimum and maximum system call arguments. Finally, the function returns the result for final randomized value which is written in *rsize* variable in constraint *rsize_c*. When writing complex randomizations inside functions instead of directly in constraints, performance of constraint solver is improved a lot, especially when there are many variables randomized with complex formulas.

Functions are useful for debugging, because any kind of print is allowed such as *$display* or *`uvm_info*. Values for variables can be easily printed to debug complex randomizations when they do not work as expected. Function called inside constraint can call another function, and complex randomization could be broken down into multiple parts, each part grouped into single function, so functional nesting is allowed.

D. Unique values

Pulling unique values outside of constraints and pre-randomizing them. They can be randomized before randomization starts in *pre_randomize* function of *uvm_object* if their calculations do not require other values, but they are still used by other constraints. For example, constant values, enumerated values, fixed or discrete ranges of numeric values can be pre-randomized. When randomization starts, they are already randomized and can be used by other constraints. The constraint solver is relaxed in this case because values are pre-randomized before constraint solver starts. Randomization in *pre_randomize* function does not consume time like constraint solver does.

```
rand bit [6:0] phy;
bit free_addr = 1, skew_part = 1, max_part = 0;

constraint phy_c {
  if (free_addr) {
    phy inside { 2, 8, 14, 20, 26, 32 };

    if (skew_part) {
      phy inside { 2, 20, 32 };
    }
    else if (max_part) {
      phy == 32;
    }
  }
  ...
}
```

Constraint *phy_c* is randomizing *phy* variable. Few flags are used for constraint conditions *free_addr*, *skew_part* and *max_part*, which are set with default values, but can be changed from test before randomization starts. Since *phy* variable is randomized with only constant values which does not depend on any other randomization, then it can be pre randomized before randomization starts. The whole randomization part can be moved to *pre_randomize* function of the configuration object (*uvm_object*) and randomized there. When pulling out values from constraints, any value which is not dependent on other values could be pulled. Sometimes whole constraint can be pulled, sometimes only parts.

```
rand bit [6:0] phy;
bit [6:0] phy_pre_rand;
bit free_addr = 1, skew_part = 1, max_part = 0;

function void eth_config::pre_randomize();
  ...
  std::randomize(phy_pre_rand) with {
    phy_pre_rand inside { 2, 8, 14, 20, 26, 32 };

    if (skew_part) {
      phy_pre_rand inside { 2, 20, 32 };
    }
    else if (max_part) {
      phy_pre_rand == 32;
    }
  }
  ...
endfunction : pre_randomize

constraint phy_c {
  if (free_addr) { phy == phy_pre_rand; }
  ...
}
```

Variable *phy* is pre randomized in *pre_randomize* function of *eth_config* configuration object. Since *eth_config* is *uvm_object*, function *pre_randomize* is already built in. Pre randomized value is written to *phy_pre_rand* variable which is not random because it is not randomized inside constraint. For pre randomizing *phy_pre_rand* variable, standard package *std* is used and *randomize* function is called using *with* block and constraint like syntax. Because

pre_randomize function is called before actual randomization, when randomization starts, then pre randomized value already exists in variable *phy_pre_rand*. In modified constraint *phy_c*, pre randomized value from *phy_pre_rand* variable is simply assigned to original *phy* variable without any constraint solver effort. When pre randomizing is done for many variables or values, then it improves constraint solver performance.

E. Delayed randomization

Post randomization of independent values. These values can be randomized after randomization is finished in *post_randomize* function of *uvm_object* if they are not used by other constraints. The constraint solver is relaxed in this case with delayed randomizations because they are randomized outside of constraints. Also, because given randomized values are independent, they can be randomized in any function, but for convenience, *post_randomize* function is used. Randomization in *post_randomize* function is instant and does not consume time.

Delayed randomization is like pre randomization from *Unique values*, but the difference between them is that values which are post randomized in *post_randomize* function are totally independent and they are not used anywhere in any other constraint. Pre randomized values are used in other constraints and that is the reason they must be pre randomized before randomization starts. For post randomized values, they can be completely moved from randomization to post randomization to relax constraint solver. Each randomization not used anywhere in any other constraint can be moved to post-randomization. If there are many independent variables or values in the randomization chain, then post randomization improves constraint solver performance.

F. Soft constraints

Since they are not directly causing constraints slowness, they often result in unexpected randomizations. Coverage gaps in randomizations are such an example. Soft constraints can be misinterpreted as default values for randomizations, which is true for some cases. When the range of values are randomized for variable and that variable has soft constraint on value which is inside randomization range, soft value will be applied instead of range. Looks simply, but in more complex randomizations, they can cause confusion and unexpected results. If soft constraint must be used in environment, then it can be disabled with *disable soft* construct in another constraint which randomize the same value for specific case, just to make sure that it will not take soft constraint value instead of randomized value. Soft constraints are best to use when they are really needed.

Code examples in this paper are verified in *Synopsys VCS Verdi* [5] and *Cadence Xcelium Logic* [6] simulators.

IV. CIRCLE RANDOMIZATION

It is randomization where each variable is related to another variable, and they end up in the circle of dependencies. For example, circle definition would be A <– B <– C <– D <– A. To randomize B, A must be randomized before B, to randomize C, B must be randomized before C, etc., and finally to randomize A, D must be randomized before A. Current randomization cannot be solved because to randomize B, C and D, case A must be randomized first, but A also depends on D which implicitly depends on A over C and B.

In real-life scenarios, each constraint can use more than one variable for randomization and the number of dependencies can be large. However, when circular randomization is detected (usually by tool), then randomization definitions could be changed using *Engineering formulas definition*. To solve circular randomization, simple definition change will help.

V. RESULTS

Before and after applying constraints techniques, simulation performance is examined. The tools used are simulators *Synopsys VCS Verdi* [5] and *Cadence Xcelium Logic* [6]. The following three cases are analyzed in Table I. Constraints performances. In the first case, constraints are causing timeouts and constraint solver could not solve them properly. The first case does not include any technique explained in the *Constraints Improvement* chapter. In the second case, constraints are solved, but with large slowness. The second case includes the first three points from *Constraints Improvement* chapter (*Engineering formulas definition*, *Order helpers* and *Function in constraint*). In the last case, constraints are solved with improved run-time performance. The last case includes all points from the

Constraints Improvement chapter. Please note there is a significant difference between using a few or all techniques explained in *Constraints Improvement* chapter and if it is possible, the best case is to use them all or most of them in conjunction because every improvement matters. All performance cases are measured for test case with most constraints involved and with longest constraints solver running. Total number of constraints involved are 3 for each agent, 22 for environment, 109 for main configuration object, 8 for sub configuration of main configuration, 2 for additional configuration object, and 322 for tests.

Table I. Constraints performances

Tools	**Constraints improvements cases[a]**		
	Initial implementation	***First optimization***	***Final optimization***
Synopsys VCS Verdi [5]	4464	2640	5
Cadence Xcelium Logic [6]	4356	2400	4

[a.] *Time measured in seconds [s]*

VI. CONCLUSIONS

Advanced randomization challenges in *SystemVerilog* constraints are solved using specific techniques from chapter *Constraints Improvement*. Given practical solutions to complex randomization problems can be used in any project where randomization is written using *SystemVerilog* constraints. Randomization timeouts are solved, and slow randomization cases are improved with given detailed techniques. Also, randomization process in complex verification environments is significantly accelerated, and logical errors are prevented. Even if issues do not exist now, systematic approaches to write *SystemVerilog* constraints can help in the future, and they can save a lot of debugging time.

REFERENCES

[1] IEEE 1800-2023: IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language, https://standards.ieee.org/ieee/1800/7743/

[2] John Dickol, SystemVerilog Constraint Layering via Reusable Randomization Policy Classes, Austin TX, DVCon US 2015

[3] Jeremy Ridgeway, Engineered SystemVerilog Constraints, Fort Collins, CO 80525, DVCon US 2015

[4] Mark Strickland, Joseph Hanli Zhang, Jason Chen, Dhiraj Goswami, Alex Wakefield, Soft Constraints in SystemVerilog: Semantics and Challenges, DVCon US 2012

[5] Synopsys Inc. VCS Simulator, Available: https://www.synopsys.com/verification/simulation/vcs.html

[6] Cadence Design Systems, Xcelium Logic Simulator, Available: https://www.cadence.com/en_US/home/tools/system-design-and-verification/simulation-and-testbench-verification/xcelium-simulator.html

[7] Cadence Design Systems, Simulation Performance Coding Guidelines for SystemVerilog, Xcelium 19.03, March 19, 2019, Available: https://support.cadence.com/apex/ArticleAttachmentPortal?id=a1Od000000050LUEAY&pageName=ArticleContent