Fnob: Command Line-Dynamic Random Generator

Haoxiang Hu, Tuo Wang
Infra ASIC, Facebook Inc,
USA email: {haoxhu, tuow}@fb.com
1180 Discovery Way
Sunnyvale, CA 94089

***Abstract*-Constrained random has been a founding methodology for DV to close coverage within an affordable number of tests. However, there're several limitations found in current constraint syntax implementation. For example, duplicated code needs to be created for similar tests with different values, introducing error-prone code which is hard to scale. Second, it takes several layers of abstraction to override random behavior for the same variable across the test, which lengthen the schedule hitting the same coverage goal. At last, re-compile is needed for every new random variation added by constraint syntax, which increases turn-around time for test development. An alternative novel way of constrained random called "Fnob" is introduced in this paper to overcome above shortfalls. By providing user freedom to override both random type and value from both command-line and in-line code without re-compile, "Fnob" speeds up coverage closure through less error-prone constrained random coding and faster test development.**

Keywords—Constrained Random; SystemVerilog; Command-line Override; UVM Configuration Database

I. Introduction

With growing scale of modern ASIC design, both amount and complexity of constrained random stimulus are increasing rapidly to reach wider and deeper part of logic. However, several limitations of current constraint syntax, including duplicated code, lower performance, and inflexibility of constraint overrides, significantly increase DV effort to meet above demands, and result in higher chance of schedule delay as well as potential late bugs.

"Fnob" as Command-line Dynamic Random Generator, is a novel methodology to improve the implementation of constrained-random, with below advantages compared to conventional constraint syntax. First, it produces less error prone testbench coding by taking advantage of predefined templates of each randomization type, which are flexible to take any number of variable sets. Second, faster random regression bring-up with the ability to override both random type and value through either in-line or command-line override, instead of additional constraint coding and re-compile. Lastly, faster DV coverage closure can be achieved by having embedded Fnob coverage on stimulus side, instead of additional functional coverage coding on checking side.

Besides supporting handy override of one random type and values, "Fnob" supports "random selection of random type", which provides capability of randomly select from several different random types. Moreover, how to randomly select different random types can also be user-defined, providing one-liner easy syntax for highly complex random pattern which could save extra coding and triaging effort.

Apart from providing scalability to the DV user, "Fnob" is designed in a developer-friendly coding architecture, which is easy to scale-up by taking new customized random types developed by any user. This opens up the "Fnob" community and improves the knowledge sharing process within and across team, so that new random pattern designed by any team member can be used by everyone in the team seamlessly.

In this paper, each feature mentioned above will be discussed in detail in corresponding chapters, with examples on how this tool could be beneficial compared to conventional constraint syntax in corresponding usage.

II. Fnob Breakdown

A. Fnob Overview

There're three fundamental requirements and primary features to define a Command Line "Dynamic" Random Generator.

1) Clear and comprehensive way to input necessary parameters, including random type and value.
2) Dynamic instantiation of corresponding random from template.
3) Dynamic override of random to any other type or values.

To achieve above requirements, Fnob is architected as a wrapper class to comprise three individual parts when instantiated correspondingly, as described below:

TABLE I. FNOB MAJOR PARTS

Parts	Description
Param Parsing	Translate Fnob input param into random type and value
Random Instantiation	Instantiate custom random class based on input param
Type/Value Override	Override type/value based on config_db

Fnob provides below list of APIs for user reference, further details of each API will be discussed in later chapters:

TABLE II. FNOB API

API	Usage	Description
`new()`	`my_fnob = new("fnob_name", <fnob_type>, <fnob_param>)`	Construct Fnob instance
`gen()`	`my_fnob.gen()`	Generate one random value from Fnob based on <fnob_type> and <fnob_param>
In-line override	`uvm_config_db#(string)::set (null,"*","fnob_name", <type:param>)`	In-line overriding type/value
CLI override	`+uvm_set_config_string=\*,fnb_name,<type:param>`	Command-line overriding type/value

Overview of Fnob usage flow chart is shown as below, details of each stage will be discussed in following chapters.

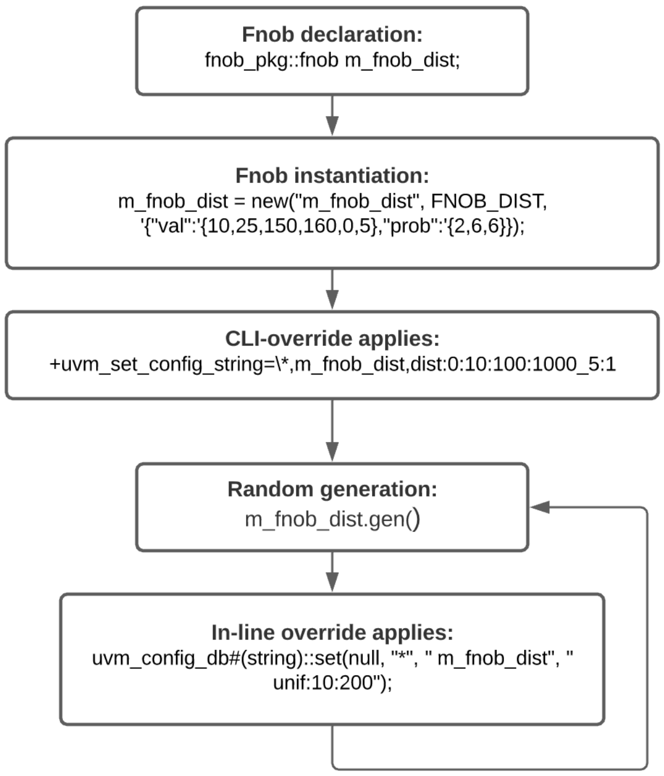
Fnob declaration:
fnob_pkg::fnob m_fnob_dist;
Fnob instantiation:
m_fnob_dist = new("m_fnob_dist", FNOB_DIST, '{"val":'{10,25,150,160,0,5},"prob":'{2,6,6}});
CLI-override applies:
+uvm_set_config_string=*,m_fnob_dist,dist:0:10:100:1000_5:1
Random generation:
m_fnob_dist.gen()
In-line override applies:
uvm_config_db#(string)::set(null, "*", " m_fnob_dist", " unif:10:200");

Fig. 1. Fnob Usage Flow Chart.

B. *Fnob Declaration & Instantiation*

For declaration, "Fnob" follows common format of SystemVerilog class instance, which is pre-defined Fnob class type followed by user defined Fnob name, as shown below:

```
fnob_pkg::fnob m_fnob_name_per_user;
```

For instantiation, "Fnob" takes three input parameters as followed: First parameter is "Fnob name" of type string used for identification and later overriding. Second parameter is random type, in pre-defined enum type. Third parameter is of type associative queue, to provide required value used for corresponding random type. The reason for not using UVM "create" is to have customized parameters in constructor as UVM default does not provides.

Fnob constructor parameter list is shown as below:

```
function new(string name="",
             FNOB_TYPE fnob_type,
             T params[string][$], // Type T is class param default as bit [63:0]
             bit no_name_chk=0);
```

1) When weight is not involved, as "inside" example:

```
m_fnob_inside_list = new("m_fnob_inside_list", fnob_pkg::FNOB_IN_LIST,
'{"val":'{16,7,101,0,56}});
```

2) When weight is involved, as "dist" example:

```
m_fnob_dist = new("m_fnob_dist", fnob_pkg::FNOB_DIST,
'{"val":'{10,25,150,160,0,5},"prob":'{2,6,6}});
```

The 1st parameter is Fnob name. This is the Fnob object name, and will also be the name registered to UVM factory for later override. It's mandatory to have unique name for each Fnob to ensure random behavior is acting on expected usage. Embedded checker is added for the naming uniqueness and will be discussed in Chapter II.G.

The 2nd parameter is Fnob "type" in the type of pre-defined enum list. This defines the expected type of randomization for its usage, which can be frequently used constraint syntax like "inside", "dist", "uniform". Each enum has corresponding string for potential type override, which is discussed in later part of Chapter II.E. Frequently used "Fnob_type" are listed in below table for user reference:

TABLE III. FNOB TYPE

Fnob Type Enum	Description
FNOB_CONST	Return constant value
FNOB_UNIF	Return uniform random value
FNOB_C_UNIF	Return cyclic uniform random value
FNOB_IN_LIST	Return random value from defined list, equivalent to "inside"
FNOB_PATN	Return value following fixed pattern as defined
FNOB_C_PATN	Return cyclic andom value from defined pattern
FNOB_DIST	Return random value based on distribution: equivalent to "dist"
FNOB_LOG	Return random value from logarithmitic distribution
FNOB_NORM	Return random value from normal distribution
FNOB_INV_NORM	Return random value from inverse normal distribution
FNOB_MULTI	Return random value based on random Fnob type

Supported enum list can also be expanded to include custom type of random in developer-friendly way, which significantly broadened its application, as discussed in detail in Chapter VI.

The 3rd parameter is Fnob values list. List type is associative queues with "val" and "prob" as predefined keys, representing value range and probability requirement. This UI design brings in two benefits:

1) Key is of type string, which replaces one long list of elements with clear separation per set of values and allows less error-prone user input.
2) Element is of type queue, which enables dynamic random behavior defined using same template type and avoids duplicated code for the same random type.

Fnob parser takes all three parameters as input and returns one random class instance. Parser first uses the 1st param as instance name, then detects the type of random from 2nd param, at last creates corresponding random instance called "m_fnob_rand" with list of input values passed from 3rd param.

```
m_fnob_rand = fnob_common#(T)::param_2_rand(m_fnob_rand_name, fnob_type, params);
```

Note that "m_fnob_rand" is of type "fnob_rand", which is the parent class for all types of random class implementation. This means "m_fnob_class" is of parent class type but contains child class instance. Polymorphism is taken advantage here to provide unified user APIs, which improves scalability for all various random type implementations.

C. Transformation from Fnob Parameter to Random

Fnob is a wrapper class of "parser", "random" and "override". Let's dive into details on 2nd part: "random", to see how input parameters translate into actual random in Fnob.

Fnob wrapper class declares one random class instance "m_fnob_rand" per Fnob, which is of base type of all customized random. This is to utilize polymorphism of different random type with same random class type for better scalability.

Random class instantiated inside Fnob utilizes scheme called "Template Random Implementation" per pre-defined random type. Each type of template random class is extended from base random class, but can have customized coding struct per developer's favor, except three must-have sub-routines defined as connection to top wrapper class:

1) "new()": constructor of corresponding random class.
2) "init()": calling child class "new()" and return child class instance to parent class handle, used in "param_2_rand()" in Chapter II.A as "new()" itself cannot return.
3) "gen()": return one value based on defined random type per call.

First part is "new()", which takes two input parameters depending on random type:

1) When weight is not involved: "name" of type string and "param" of type queue to include required values.

```
function new(string name="", T params[$]);
```

2) When weight is involved: "name" of type string and "param" of type associative queue to include both values and weights.

```
function new(string name="", T params[string][$]);
```

Second part is "init()". "init()" does the same thing as "new()", but is needed as a separate sub-routine for two reasons:

1) "new()" doesn't return, while "init()" can be customized, hence can return the instantiated random instance back to top wrapper class. This is needed in Fnob "Parser" ("param_2_rand") mentioned in Chapter II.B, as to use same parent handle for child objects, which is the key to unify the implementation, at the same time providing better code scalability, which is the key to code scalability:

```
static function fnob_rand#(T) init(string name, T params[$]);
  fnob_rand_unif#(T) fr = new(name, params);
  return fr;
endfunction
```

2) "init()" needs to be declared as static function, which can be called without class instantiation. During Fnob parser ("param_2_rand") mentioned in Chapter II.B, two things need to be done:

a) Instantiation of random class instance of child class type.

b) Assign child class instance to parent class handle.

Instead of writing multiple lines of code per type as below:

```
fnob_rand_unif m_rand_unif = new(fnob_rand_name, params["val"]);
param_2_rand = m_rand_unif;
```

One line can do the job using static function from different child class:

```
param_2_rand = fnob_rand_unif#(T)::init(fnob_rand_name, params["val"]);
```

This enables cleaner coding structure of "Parsing" logic in II.B and improves scalability to more random types as shown below:

```
static function fnob_rand#(T) param_2_rand(string fnob_rand_name, FNOB_TYPE fnob_type, T
params[string][$]);
    case (fnob_type)
      FNOB_UNIF:    begin param_2_rand = fnob_rand_unif#(T)::init(fnob_rand_name,
params["val"]);end
      FNOB_C_UNIF:  begin param_2_rand = fnob_randc_unif#(T)::init(fnob_rand_name,
params["val"]);end
      FNOB_CONST:   begin param_2_rand = fnob_rand_const#(T)::init(fnob_rand_name,
params["val"]);end
      …
    endcase
endfunction
```

Fig. 2. Fnob Parser

Third part "gen()" is for Fnob Usage, will be discussed in detail in Chapter II.D.

D. Fnob Usage

Fnob provides one common "gen()" subroutine for all random types to generate random values based on user defined definitions. This is the subroutine which returns one value per call based on defined random type and is being called in expected place of TB during simulation. The steps of random value generation are described as below:

1) Each template random class, random value is calculated based on corresponding random type through software-based implementation instead of constraint syntax. At the end of "gen()" of template random class, generated random value is returned. There are two reasons to use "$urandom" based instead of "constraint solver". One is to remain flexible for different value of same random type, as constraint block needs to be re-coded for different value and weight pairs for each override, dwarfs the goal of removing duplicated code. Another is significant performance improvement in simulation time use "$urandom" based compare to "constraint solver", which will be discussed in Chapter IV.

2) Next, in top Fnob wrapper class, a different "gen()" is created to call the "gen()" of "fnob_rand" returned from "param_2_rand" in Chapter II.B, which contains the instance of corresponding template random class. This "gen()" in wrapper class will also be the one called by user in TB.

3) At last, user call "gen()" of corresponding Fnob (top wrapper class) in desired place inside TB to return expected random value for verification usage, as shown below:

```
repeat (number_of_pkt) begin pkt_field = m_fnob_dist.gen(); end
```

To alter the random behavior of Fnob in the middle of test, user can call "In-line override scheme" as below example:

```
// Initial random behavior
repeat (number_of_pkt) begin pkt_field = m_fnob_dist.gen(); end

# <user-defined>ns; // elapsed simulation cycles

<Fnob In-line override on m_fnob_dist >
// Overridden random behavior
repeat (number_of_pkt) begin pkt_field = m_fnob_dist.gen(); end
```

Fig. 3. Fnob In-line Override Flow

E. Fnob Type & Value Override

Now that we know how to define and call Fnob, let's change to a different perspective, and take a look at another powerful feature Fnob provides: "Override". Fnob promises compile-free override of both random type and value by taking advantage of "uvm_config_db". Both command-line and in-line override are supported by exploiting corresponding build-in APIs in "uvm_config_db::set". "Command-line" override takes effect only once per test at the beginning, while "In-line" override can be called multiple times achieved by "uvm_config_db#(string) ::wait_modified" in forever loop of a forked-off parallel thread.

1) "In-line" override follows the same rule defined in "uvm_config_db#(string)::set" in uvm_pkg. Minor difference for corresponding random type exists in cases whether require either value & weight or value-only. Usage examples are shown as below:

a) When weight is not involved, as "inside" example:

```
uvm_config_db#(string)::set(null, "*", "m_fnob_inside_lis", "in_list:9:1:2:100");
```

b) When weight is involved, as "dist" example:

```
uvm_config_db#(string)::set(null, "*", " m_fnob_dist", " dist:0:10:100:1000_5:1");
```

Usage of two separators ":" and "_" are the same as described in "Command-line Override". Note that space within string is allowed, Fnob string parser removes all spaces before recognizing.

Unlike "Command-line Override", "In-line Override" can be executed multiple times within simulation after build_phase(). This provides significant benefit in random test capability in two ways:

a) Various random schemes can be tested within the same test with one-line override.
b) DUT can be easily pushed into expected states with control of random at different stage of simulation.

2) "Command-line Override" is following same rule defined in "+uvm_set_config_string" in uvm_pkg. Minor difference exists for whether corresponding random type requires either value & weight or value-only. Usage examples are shown as below:

a) When weight is not involved, as "inside" example:

```
+uvm_set_config_string=\*,m_fnob_inside_list,in_list:9:1:2:100
```

b) When weight is involved, as "dist" example:

```
+uvm_set_config_string=\*,m_fnob_dist,dist:0:10:100:1000_5:1
```

For both types, pre-defined string for each random type enum is defined in Fnob. Fnob parser uses ":" as separator for each element in the string, pre-defined sequence of element is "type:value0:value1…valueN". Fnob signals fatal error when incorrect sequence of element is passed in and allows quick debug turn-around with early exit.

For random type involving weights, Fnob uses pre-defined "_" as separator between value elements and weight elements. Pre-defined sequence of element is "type:value0..n_weight0..n". Same ":" separator is used within element sequence of value/weight. The reason to pick "_" is to be Linux OS friendly and avoid potential conflict of using special sigils defined for other purpose in OS terminal.

Notice that command-line override takes effect once per test at the end of new() of each Fnob. If multiple CLI overrides are given to the same Fnob based on name, the last one wins.

"In-line Override" requires recompile, while "Command-line Override" will not require a re-compile. For writing different random tests, customized abstraction layer can be added to cleanly attach "Command-line Override" of multiple Fnobs for each test to achieve expected behavior. This enables two major benefits:

1) No new code needed for overridden random behavior for all tests in regression.
2) Compile-free bring-up, enable faster turn-around of new test debug.

F. *Fnob Embedded Coverage*

Functional coverage is used to monitor if randomization constraint covers all the distributed cases as expected. To enable early-stage coverage from stimulus, Fnob provides embedded coverage to enable clearer and faster coverage closure on the stimulus side. Embedded cover-groups are created per each built-in random type and is instantiated per each Fnob instance.

As Chapter II.A illustrated, fnob_rand.sv structures all supported random flavors in separate classes: i.e. "fnob_rand_unif", "fnob_rand_profile", "fnob_rand_const", it is also the place where the covergroups are embedded.

1) Sample all the corners. Take "fnob_rand_unif" for example, min/mid/max bins are implemented as default. The general guideline is: the default covergroups that implemented in Fnob layer are the minimum baseline. If user needs more complex covergroup instead, it is recommended to create customized covergroup in the application layer. This is also due to SystemVerilog language limitation that covergroup could not be overridden from child classes.

```
covergroup fnob_cfg:
  option.per_instance = 1;
  option.goal              = 100;
  option.comment      = "fnob_cg";

  fnob_rand_cg: coverpoint m_gen{
  bins val_min = {m_unif_min};
  bins val_max = {m_unif_max};
  bins val_mid  = {m_unif_mid};
  }
endgroup
```

Fig. 4. Consecutive Covergroup Example in "fnob_rand_unif"

2) Covergroup is sampled every time a new Fnob val is generated through "gen" function, so that every randomization result is recorded.

3) For nonconsecutive Fnob type like "fnob_rand_pattern" or "fnob_rand_inside_list", a dynamic array of covergroups is used to flatten each value.

```
covergroup fnob_pattern_cg(int val) with function sample(int cp):
  option.per_instance = 1;
  option.goal              = 100;
  option.comment      = "fnob_cg";

  fnob_rand_cg: coverpoint cp{
  bins value_all = {val};
  }

class fnob_rand_pattern#(type T=bit[63:0]) extends fnob_rand#(T);
  …
  fnob_pattern_cg cg[];
  …
  function new(string name="", T params[$]);
  …
    cg = new[m_vals.size()];
    for (int ii=-; ii<m_vals.size(); ii++) begin
      cg[ii] = new(m_vals[ii]);
    end
  endfunction: new
endclass
```

Fig. 5. Non-consecutive Covergroup Example of "fnob_rand_pattern"

G. *Fnob Uniqueness Checking*

As Chapter II.E illustrated, Fnob uses the uvm config_db scheme to override the default randomization constraint, which assumes that each Fnob variable is in global scope and should be unique in terms of variable naming. Therefore, a uniqueness checking is implemented in the Fnob package, to ensure each Fnob has unique name which achieves error-free override. Below are the implementation details:

1) In “fnob_common.sv”, a local static m_fnob_pool associative array is maintained. As Chapter II.C mentioned, “new” function is the first step that needed to create a certain Fnob variable. It is also the place where the uniqueness checking is embedded. Every time a “new” constructor is called to create a new Fnob variable, m_fnob_pool is queried for the existence check. If not, register the new Fnob name; otherwise error out.

```
local static bit m_fnob_pool[string]; //register the fnob name once created, used for
uniqueness check
```

2) “no_name_chk” argument is reserved to disable the check(default to enable) in the “new” function.

```
function new(string name=””, FNOB_TYPE fnob_type, T params[string][$], bit no_name_chk=0);
```

III. Fnob Multi: Random on Top of Random

In Chapter II, we have discussed how to describe one type of random behavior without extra implementation code, and how to alter random behavior for different test of within the same test. This suffices most of usage except the case where multiple type of random is needed within one test following pre-defined random order.

“Fnob-Multi” is proposed to resolve the above limitation and created to achieve “random selection of constraint random” with one-liner instead of complex constraint coding. To achieve “random on top of random”, “Fnob-Multi” requires two sets of info as part of input:

1) First Fnob in brackets defines selection scheme within random type list.
2) List of following Fnobs in brackets defines random type candidates to be selected.

Example of “Fnob-Multi” instantiation is shown as below:

```
fnob_pkg::fnob m_fnob_multi = fnob#(bit[63:0])::
new_multi("m_fnob_multi","(pattern:0:0:0:1:2:2:2)(unif:0:10)(constant:100)(dist:0:10:100:1
000_5:1)");
```

Since, “associative queue” used in single Fnob cannot suffice its special input requirement, a different constructor “new_multi()” is created, which takes two inputs: name as type of string, and “random on top of random” also as type of string. First part “name” follows the same rule as single Fnob.

Second part is the string which represents “random on top of random”. Two sets of info are needed as shown below:

1) One Fnob define selection scheme within random type list.
2) One Fnob queue store list of random type candidates to be selected.

In input string, both 1 and 2 are flattened into one list of single Fnob, each enclosed in bracket as pre-defined separator “()”.

All Fnob declarations are the same as specified in override discussed in Chapter II.E.

The first Fnob corresponds to selection knob, value returned from it should be within the size of candidates’ pool, meaning if returned value is 0, then 1st candidate will be picked, if returned value is 2, 3rd candidate will be picked. If out-of-range, “Fnob-Multi” will signal fatal error and early exit the sim for debug.

The following Fnobs are the candidates to be selected, with the sequence of declaration maps to the return value of selecting Fnob. The syntax is the same as single Fnob.

The way to define and use “Fnob-Multi” is roughly the same as single Fnob, user first declares & instantiates it, then calls “gen()” whenever needs to get random value.

```
fnob_pkg::fnob m_fnob_multi = fnob#(bit[63:0])::
new_multi("m_fnob_multi","(pattern:0:0:0:1:2:2:2)(unif:0:10)(constant:100)(dist:0:10:100:1
000_5:1)");

repeat (number_of_pkt) begin pkt_field = m_fnob_multi.gen(); end
```

Fig. 6. Fnob_multi Usage

The underlying implementation is however different from single Fnob as shown in below steps:

1) Store selecting Fnob “type_fnob” and candidate Fnob queue “value_fnob[$]” during instantiation.
2) When “gen()” is called:

   a) Call “type_fnob.gen()” to return “idx” to candidate queue.
   b) Select “value_fnob[idx]” as selected Fnob, which is named as “frand”, for this specific Fnob_multi gen call.
   c) Call “frand.gen()” to return final random value to TestBench.

IV. Fnob Improvement Result

Fnob displays two major improvements in constrained random implementation compared to conventional constraint syntax:

1. Significant coding and turn-around time reduction in constraint override using Fnob compared to constraint syntax due to simplification in overriding flow.

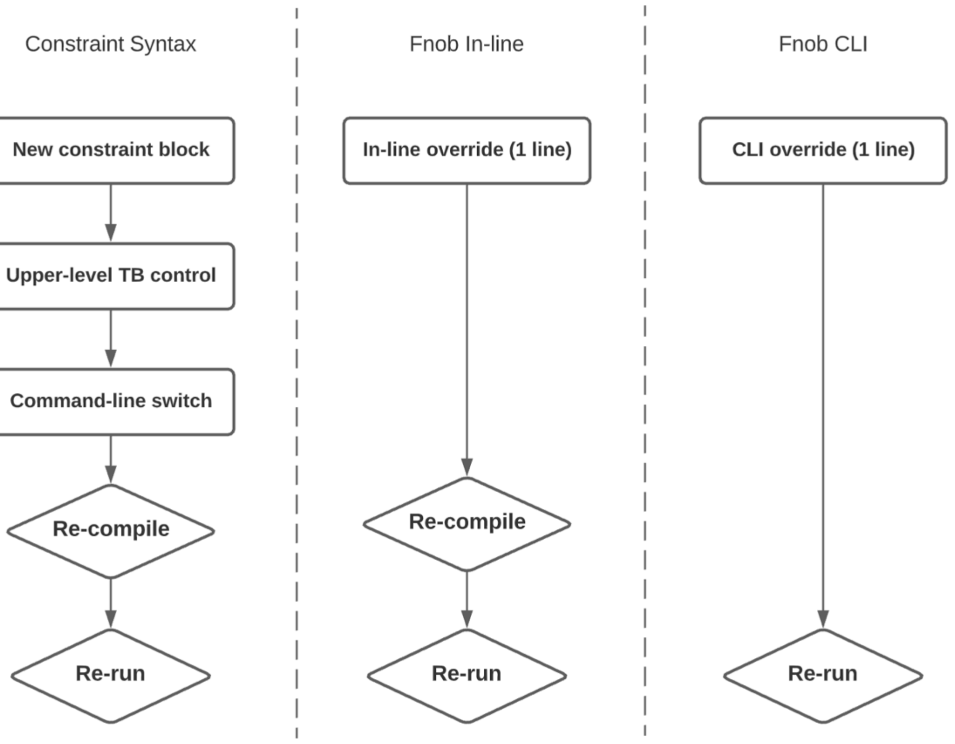

Fig. 7. Constraint Override Flow Comparison.

2. 10-40x performance improvement in simulation system time using Fnob compared to constraint syntax. This is due to difference of underlying random generation scheme between each other.

   “Constraint” is to drive solver to generate set of values which satisfy all of constraints, which means solver needs to traverse entire value space, and apply constraint on each possible candidate to see if it’s a match. Fnob, on the other hand, using system function to define the value scope to a limited range. Fnob hence utilize a smaller range of candidates statistically, translates to less computation needed and leads to higher performance.

   Below table describes the comparison of simulation time between “Constraint” and “Fnob” for exact same randomization. Data is collected through simulator from two Tier-1 EDA vendors.

TABLE IV. PERFORMANCE IMPROVEMENT

Random Keyword	Performance Test				
	Number of Call	***Constraint: vendor 1***	***Fnob: vendor 1***	***Constraint: vendor 2***	***Fnob: vendor 2***
inside	100,000,000	127s	5s	117s	1s
dist	100,000,000	69s	4s	119s	2s
constant	100,000,000	63s	4s	116s	3s
normal	100,000,000	68s	8s	118s	2s

V. FNOB INTEGRATION TO CONSTRAINT

Besides difference from "constraint" syntax, Fnob can also be integrated into existing constraint blocks, to provide user additional flexibility, while retaining the conditional syntax between different random variables. This integration merged the benefit of both methodologies in random implementation, as shown below

1) Fnob: flexible control on high-level random behavior.
2) Constraint: detail control on low-level conditional random behavior.

Example of Fnob integration is shown as below:

```
rand int m_rand0, m_rand1;
fnob_pkg::fnob m_rand0_fnob, m_rand1_fnob0, m_rand1_fnob1;

// constructor
function new();
  m_rand0_fnob   = new("m_rand0_fnob", fnob_pkg::FNOB_CONST, '{"val":'{0}});
  m_rand1_fnob0 = new("m_rand1_fnob0", fnob_pkg::FNOB_CONST, '{"val":'{0}});
  m_rand1_fnob1 = new("m_rand1_fnob1", fnob_pkg::FNOB_CONST, '{"val":'{0}});
endfunction

constraint c0{
  m_rand0 == m_rand0_fnob.gen();
};

constraint c1{
   (m_rand0 <= 100) -> m_rand1 == m_rand1_fnob0.gen();
   (m_rand0 > 1000) -> m_rand1 == m_rand1_fnob1.gen();
};
```

Fig. 8. Example of Fnob Integration to Constraint

In above example, two random variables "m_rand0" and "m_rand1" are declared. "Constraint" syntax in constraint block "c1" is used to describe low-level conditional random behavior in between, while "Fnob" is used to provide high-level control on random behavior for each variable after certain condition held true. Fnob integration to constraint hence expands the capability of "Constraint" and enables faster and easier description on variant random behaviors.

When different random behaviors for "m_rand0" and "m_rand1" are desired in different tests, one can simply describe it through "Fnob CLI-override", as example shown below:

```
// 1st test: m_rand0 90% less equal than 100, 10% between 1000 to 2000; m_rand1 always 10;
+uvm_set_config_string=\*,m_rand0,dist:0:100:1000:2000_9:1
+uvm_set_config_string=\*,m_rand1,constant:10

// 2nd test: m_rand0 70% between 1000 to 2000, 30% less equal than 100; m_rand1 uniform
random between 10 to 50;
+uvm_set_config_string=\*,m_rand0,dist:0:100:1000:2000_3:7
+uvm_set_config_string=\*,m_rand1,unif:10:50
```

Fig. 9. Example of Fnob Override in Constraint

There are three benefits in above test-writing scheme:

1) No new code: writing tests becomes writing “Fnob” on command-line instead of “Constraint” in SV code.
2) No re-compile: “Fnob CLI override” doesn’t need to re-compile TB, speeds up test development.
3) Existing condition remains intact: condition between random variables in “Constraint” block still holds true and can be applied in random generation.

VI. Fnob Developer Interface

Currently, 11 flavors of Fnob type are generalized and implemented. We foresee more types of variations to be added. Hence Fnob separates user layer files from developers to minimize the steps and overhead for type extension.

For users, most of the use cases should be handled using Fnob data types and creating random scenarios embedded in testbench. Focusing on fnob.sv and fnob_random_multi.sv files should be good enough to suffice the needs.

For advanced users and developers in Fnob community, we encapsulated the developer interface into three files:

1) Fnob_common.sv: common Fnob utility functions.
2) Fnob_rand.sv: All the fnob random flavors, could be easily extended for new variations.
3) Fnob_test.sv: Light-weight standalone testing infrastructure.

User layer and developer layer package separation layer are shown in Fig.3 as below:

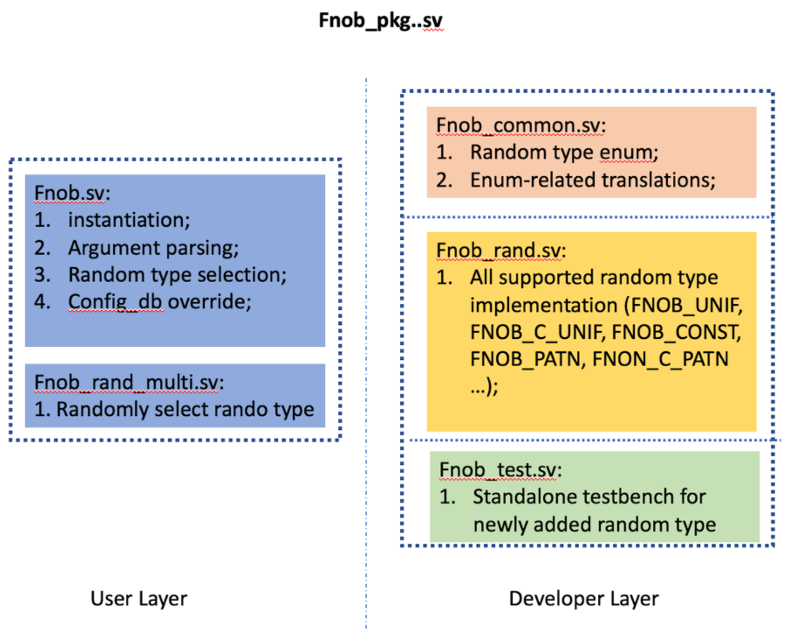
Fnob_pkg..sv
Fnob.sv:
1. instantiation;
2. Argument parsing;
3. Random type selection;
4. Config_db override;
Fnob_rand_multi.sv:
1. Randomly select rando type
Fnob_common.sv:
1. Random type enum;
2. Enum-related translations;
Fnob_rand.sv:
1. All supported random type implementation (FNOB_UNIF, FNOB_C_UNIF, FNOB_CONST, FNOB_PATN, FNON_C_PATN …);
Fnob_test.sv:
1. Standalone testbench for newly added random type
User Layer
Developer Layer

Fig. 10. Fnob package structure.

Below are the detailed steps to add a new type:

1) Fnob_rand.sv: Create a new class extends from fnob_rand#(T); add new random implementation; please note that “new”, “init” and “gen” routines are must-have.
2) Fnob_common.sv: Register new random type enum in FNOB_TYPE; add typedef of new class; add new enum translation in “param_2_rand” and “s_2_type”.
3) Fnob_test.sv: Test new fnob type thoroughly in standalone testbench; add use case example comment.

References

[1] Venessa R. Cooper, Paul Marriott, “Demystifying UVM Configuration Database”, DVCon 2014, 163-HI374-final-feb03.
[2] Ray Salemi, “*The UVM Prime: A Step-by-Step Introduction to Universal Verification Methodology”,* 1st ed, Boston Light Press, 2013.
[3] Paul Marriott, Jonathan Bromley, “Re-imaging Randomization using VCS constraint solver”, SNUG Europe 2014.