See the Forest for the Trees – How to Effectively Model and Randomize a Directed Rooted Tree Structure

Harry Duque and Lars Viklund
Axis Communications AB
Gränden 1, Lund, Sweden

***Abstract*—The data flow between sub-units in a hardware unit with configurable data paths can be represented as a set of directed rooted trees (DRT). Using constraint-random techniques to verify such units requires modeling the space of legal DRTs in constraints. The representation should be simple, scalable, feasible to implement, and allow to steer the randomization to ease coverage closure. Furthermore, it should enable a constraint solver to produce a solution in a reasonable amount of time. This paper describes a solution in which a DRT is modeled with a composite design pattern. Each vertex in the DRT, starting from the root, is recursively randomized to produce a complete branch of the DRT.**

INTRODUCTION

The design verification of algorithmic units within an image processing pipeline is generally well suited for coverage-driven, constrain-random verification. These functional units perform distinct algorithmic operations such as noise filtering, white balancing, and tone mapping [1]. Optimum algorithmic results require statistics such as histograms and mean or median pixel calculation for a complete frame or a particular region of interest. These computations require operations on every pixel at full throughput. Dedicated functional units for statistic computations are then needed in the pipeline, with a high degree of configurability, i.e., they should be able to perform different statistics operations from pixel streams sourced at various points in the pipeline. An illustration of a standard image processing pipeline, including a statistic unit, is depicted in Fig. 1.

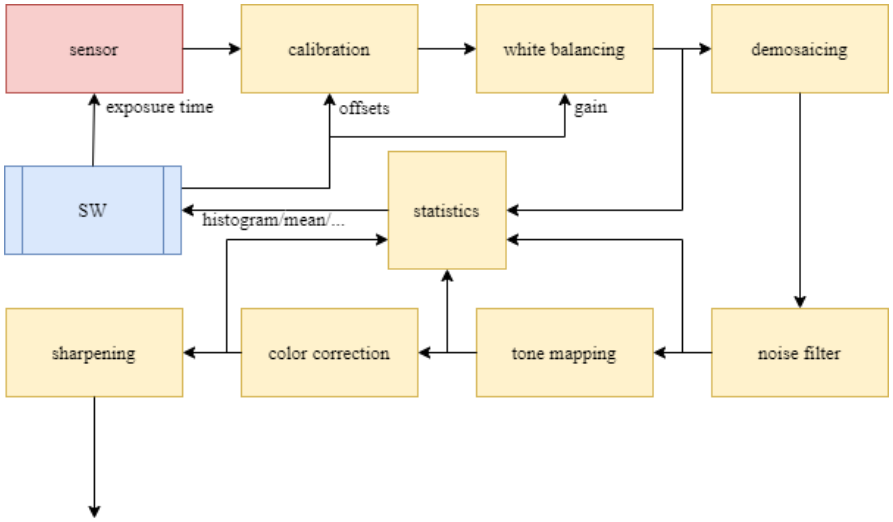

Figure 1. An image-processing pipeline

The statistic unit is then composed of several sub-units, either collecting statistics and producing results, a *sampler*, or data conversions units, a *converter*. An example of a converter can be a function to transform from one color space to another, e.g., from RGB to YCbCr.

A sampler can be connected to a converter, and a converter can connect to another converter or a primary input of the statistic unit. The connectivity of the complete statistic block is configured through a software register interface.

The configured connectivity should follow the following rules:

1. Each enabled sub-unit shall be connected.
2. Each enabled sub-unit shall have a single connection to its input.
3. All paths in the connectivity shall be acyclic.
4. There is an orientation in the connectivity; data flows in one direction.
5. There is a single start point for each set of connected sub-units.

Any configuration not satisfying these rules will likely result in a hang in the pipeline.

The described structure can be represented as a directed acyclic graph (DAG), more specifically as a directed rooted tree (DRT). The root vertex of the tree will be a primary input, and each following vertex will represent a sub-unit, either a sampler or a converter. Each edge represents a connection between two sub-units. See Fig. 2 for an illustration of such a representation.

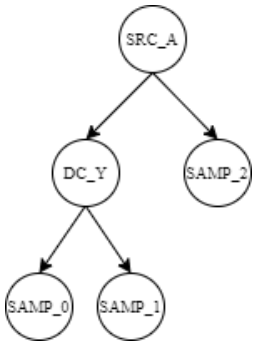

Figure 2. A simple labeled tree with five vertices and four edges

Modeling and randomization of such structures present the following challenges:

- **Validity**: To accurately model the DRT structure through constraints, satisfying the specified rules.
- **Controllability**: Possibility to easily steer or direct the solver towards different paths in the tree.
- **Performance**: The constraint solver should be able to produce a solution in a reasonable amount of time.
- **Scalability**: The model should scale favorably with the number of vertices in terms of constraint entry and randomization performance.

Perhaps the simplest solution would be to list each possible tree. The number of possible trees on n labeled vertices is n^(n-2) [2]. The sub-set of legal configurations is enumerated from this maximum possible set, and a constraint solver could pick from the enumeration. The space of possible trees grows exponentially with n, which makes it a feasible approach for only a very small number of vertices.

A more complex model of the DRT structure is then required. In theory, producing one that randomizes a DRT in a single `randomize()` call could be possible. Each legal vertex in the DRT can only be connected to a set of other vertices. This could be expressed directly in constraints on the configuration registers. Helper variables could be used to ease the constraint entry. High-level knobs and other constraint techniques could be used to produce the acyclic structure. In practice, during our ASIC project, where such statistics units were first verified, the attempts to create such a model were unsuccessful. Expressing the resulting structure was too verbose and cumbersome, it did not provide a mechanism to steer the randomization easily, and ultimately it simply failed to produce valid results in a reasonable time. After stepping back and examining the problem from above, we were able to see the forest from the trees. The key point was to note that this type of search problem is suitable for a back-tracking, recursive algorithm. Such an approach should prove more elegant and likely to meet the listed challenges. The problem was broken down into two sub-problems: how to model the DRT structure and how to randomize it recursively.

MODELING THE DIRECTED ROOTED TREE

The framework we have developed is based on the SystemVerilog [3] UVM [4] class library. A base vertex class extending from `uvm_object` represents each vertex in the tree, see Fig 3. This class is parameterized with the type `id_t`, which would typically be an enum defining the list of legal vertex ids. A class member `id` of the `id_t` type will then hold the unique identifier of the instance. The class is also parameterized with the type of the register

configuration class `config_t`, which is used to define the class member used to configure the low-level registers procedurally after the high-level tree model has been randomized.

The tree structure is suitable to be represented with the composite design pattern [5]. The vertex class contains a handle to the parent vertex, passed to the object when initialized. Each vertex has a queue of vertices used to capture all the children of an instance.

```
class vertex_base#(type id_t, type config_t) extends uvm_object;
  id_t id, parent;
  vertex_base#(id_t, config_t) children[$];

  function void init(id_t id, parent);
    this.id     = id;
    this.parent = parent;
    ...
  endfunction : init

  function void add_child(vertex_base#(id_t, config_t) child);
    this.children.push_front(child);
  endfunction : add_child

...
```

Figure 3. Vertex base class

The connectivity of a particular vertex is then defined in a derived class. For each vertex defined in the `id_t` enum, constraints are written to describe their space of legal children. A `START` vertex is defined to indicate the tree's root and a `STOP` vertex to indicate an endpoint of a branch. A made-up example of such a class for a simple statistics unit is shown in Fig. 4. The following naming conventions are used.

- `SRC_<*>`: A primary input
- `DC_<*>`: A converter sub-unit
- `SAMP_<*>`: A sampler sub-unit

```
class vertex extends vertex_base#(id_t, config_t);

  constraint c_connectivity {
    this.id inside {START}        -> this.child_id inside {SRC_A, SRC_B, SRC_C, SRC_D};

    this.id inside {SRC_A}        -> this.child_id inside {DC_X, DC_Y, SAMP_0, SAMP_1, SAMP_2};
    this.id inside {SRC_B}        -> this.child_id inside {DC_X, DC_Y};
    this.id inside {SRC_C, SRC_D} -> this.child_id inside {DC_X, DC_Y, DC_Z};

    this.id inside {DC_X}         -> this.child_id inside {DC_Y, SAMP_0};
    this.id inside {DC_Y}         -> this.child_id inside {DC_X, SAMP_0, SAMP_1};
    this.id inside {DC_Z}         -> this.child_id inside {DC_X, DC_Y, SAMP_0};

    this.id inside {SAMP_0,
                    SAMP_1,
                    SAMP_2}       -> this.child_id inside {STOP};
}

...
```

Figure 4. Vertex connectivity

Note that the problem of controlling the randomized DRT becomes a matter of extending the vertex class to constrain the connectivity further and produce any desirable hard-to-hit tree structure. For example, considering our base legality vertex class defined in Fig. 4, creating a test that always includes a `SAMP_0` vertex in the tree could be desirable. Fig. 5 presents one possible way to constrain randomization to produce such results. Each source in the tree is constrained to either have `SAMP_0` directly as a child or, alternatively, one of the three available converters. In turn, each converter is constrained to only have the sampler `SAMP_0` as a child. The end result is that any generated tree will contain the `SAMP_0` vertex.

```
class sampler0 extends vertex#(id_t, config_t);

  constraint c_samp_0 {
    this.id inside {SRC_A}        -> this.child_id inside {DC_X, DC_Y, SAMP_0};

    this.id inside {DC_X}         -> this.child_id inside {SAMP_0};
    this.id inside {DC_Y}         -> this.child_id inside {SAMP_0};
    this.id inside {DC_Z}         -> this.child_id inside {SAMP_0};
  }

...
```

Figure 5. Vertex constrained connectivity

A visualization of the complete connectivity described by constraints in Fig. 4 is shown as a directed graph in Fig. 6.

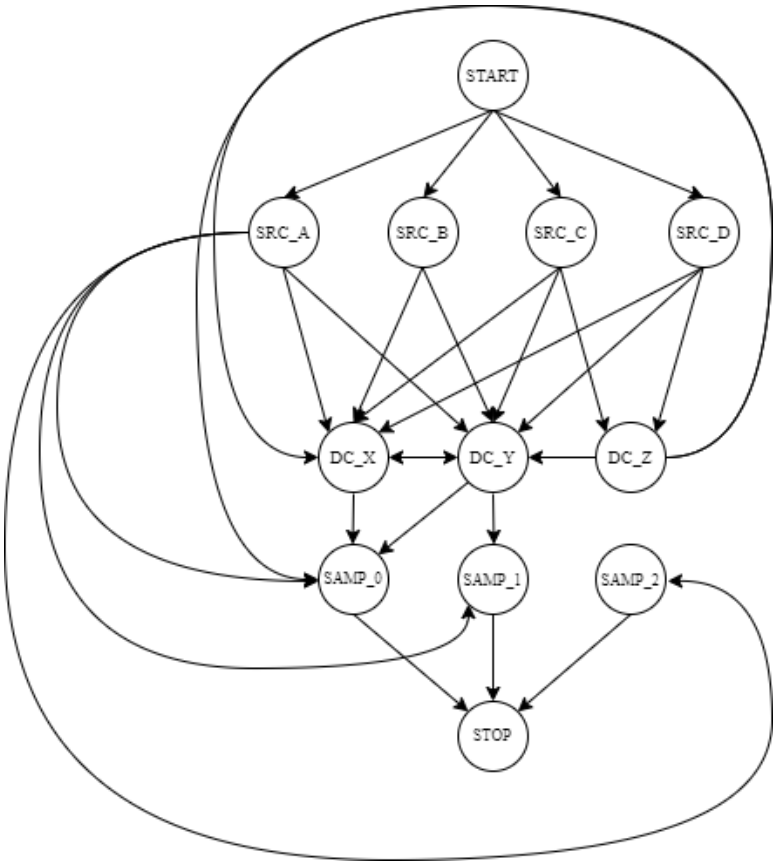

Figure 6. Vertex connectivity visualization

RECURSIVE RANDOMIZATION

The remaining problem is randomizing a particular DRT from the connectivity specified in constraints. The recursive method does not attempt to randomize the complete tree in a single `randomize()` call. Instead, it uses a depth-first approach where each vertex in a branch will be individually randomized, starting from the `START` of the tree until reaching the `STOP` vertex. If successful, it allocates the selected child vertex to randomize in the next iteration. If unsuccessful, or if the selected child is the `STOP` vertex, we backtrack to the parent of the randomized vertex and attempt to create a new branch from that vertex.

A data structure is maintained to track the set of available vertices for randomization. If a vertex is successfully randomized and used in the tree, or if a vertex fails to randomize a single child, we add it to a list of vertices to withdraw from the following randomization iteration.

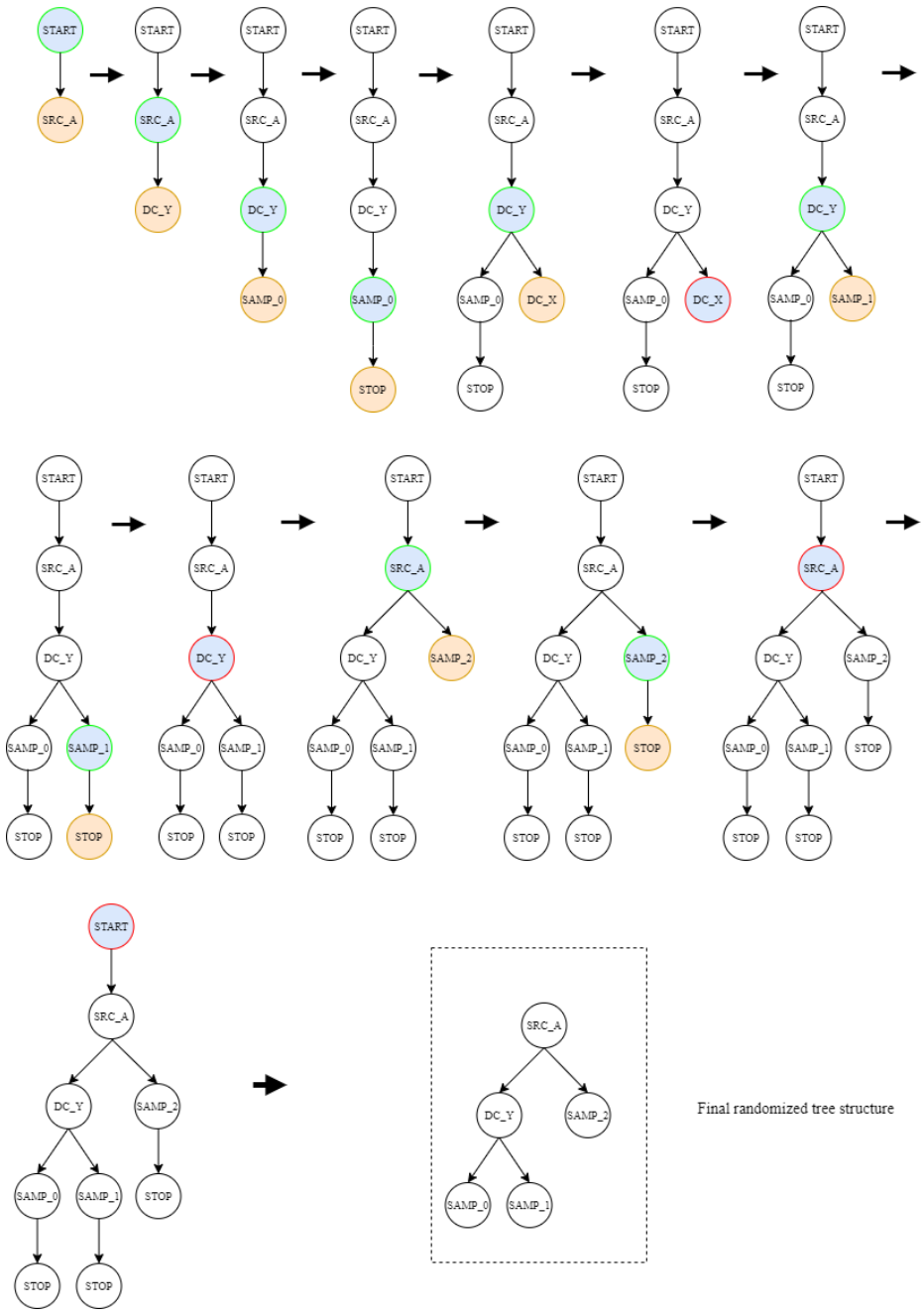

Figure 7. Randomization of a simple tree

What is described above is basically a back-tracking algorithm where the search space is limited by eliminating possible candidates. This approach guarantees that the tree structure is acyclic and with non-converging paths, and additionally, it is guaranteed to produce a result in a bounded amount of time.

Fig. 7. illustrates the randomization process for the connectivity example described above. The blue color denotes the vertex under randomization in a particular iteration. The green color outline denotes successful randomization, whereas red outlines a randomization failure. The orange color denotes the randomized child vertex when randomization is successful.

A separate `allocator` class, see Fig. 8, is used to implement the randomization scheme described above. A `vertex_base` class member holds the handle to the root of the tree to be randomized, i.e., the `START` vertex. The queue class member `withdrawn` tracks any vertex that should not be considered in upcoming randomization iterations. A high-level knob `branch_cnt` is used to control and randomize the maximum number of branches the tree will possess. This knob creates a healthy spread of solutions in terms of tree dimensions.

```
class allocator#(type id_t, type config_t) extends uvm_object;
  vertex_base#(id_t, config_t) root; // handle to root of tree
  id_t withdrawn[$]; // list of vertices withdrawn from tree randomization

  int unsigned max_branch_cnt;
  rand int unsigned branch_cnt;

  constraint c_branch_cnt {
    this.branch_cnt inside {[1:this.max_branch_cnt]};
  }
...
```

Figure 8. Allocator class members and related constraints

The iterative nature of the randomization scheme can be seen in the `make_tree()` method, see Fig. 9. A number of `branch_cnt` or fewer branches are produced by iteratively executing the `find_path()` method, which starting from the `START` vertex, attempts to randomize a single branch from the iterated "current" vertex to the `STOP` vertex. Regardless of the randomization outcome, a new vertex is selected to be the current in the next `find_path()` iteration by executing the `next_vertex()` method, which in our outlined implementation would return the parent of the current vertex. These iterations will continue until `branch_cnt` branches are produced or the root is reached, and no further paths to `STOP` can be randomized.

```
  function void make_tree(id_t stop);
    vertex_base#(id_t, config_t) current = this.root;
    int unsigned branch_cnt = this.rand_branch_cnt();
    do begin
      if (find_path(current, stop)) begin
        branch_cnt--;
      end
      current = this.next_vertex();
    end while ((current != null) && (branch_cnt > 0));
  endfunction : make_tree
```

Figure 9. Allocators class `make_tree()` method

The `find_path()` method, shown in Fig. 10, randomizes the current vertex with an inline constraint to avoid any withdrawn vertices. Any randomization failure would terminate the `find_path()` recursion and add the failed vertex to the `withdrawn` queue, as it can no longer be used to find a path to `STOP`. The other base case of the recursion is when the randomization is successful, and the solver selects the STOP vertex as a child. In this case, the randomized vertex is also annexed to the `withdrawn` queue to avoid any cyclic or convergent paths. Finally, if the randomization is successful but with a randomized child different than `STOP`, another recursive step of `find_path()` is started from the new child to `STOP`.

```
function bit find_path(vertex_base#(id_t, config_t) current, id_t stop);
  vertex_base#(id_t, config_t) child;

  forever begin
    if (!current.randomize() with {!(this.child_id inside {local::withdrawn});}) begin
      this.withdrawn.push_front(current.id);
      return 0;
    end
    if (current.child_id == stop) begin
      this.withdrawn.push_front(current.id);
      return 1;
    end
    child = vertex_base#(id_t, config_t)::type_id::create();
    child.init(current.child_id, current.id, this);
    this.withdrawn.push_back(current.id);
    ok = find_path(child, stop);
    this.withdrawn.pop_back(current.id);
    if (ok) begin
      this.withdrawn.push_back(current.id);
      current.add_child(child);
      return 1;
    end
  end
endfunction : find_path
```

Figure 10. Allocators class `find_path()` method

The process of creating a tree is started by the stimuli modeling object of the unit's test bench. During its `pre_randomize()` method, the root of the tree is allocated and initialized. An instance of the `allocator` class is also created and initialized, followed by a call to its `make_tree()` method, which will recursively build the tree. At this stage, a tree structure is produced, i.e., the unit's connectivity is known, and the configuration registers which govern the unit's connectivity are procedurally configured to produce the established tree. The `randomize()` stage of the unit stimulus object is then entered, where the rest of the configuration registers are randomized, particularly those related to the algorithmic functions of the sub-units present in the randomized tree.

DEALING WITH FAILURES (OR YOU HAVE TO FAIL TO SUCCEED)

A feature of the scheme described in this paper is how randomization failures are inherent to randomizing a DRT structure. Let's repeat this: to randomize a valid DRT, a number of `randomize()` calls executed in the `find_path()` recursion are fully expected to fail, i.e., return null. As the different branches are built, vertices are added to the `withdrawn` queue, which in return will limit the number of children that a particular vertex could randomize up to a point where there are none available remaining in the legal configuration space for that vertex. Instead of backpedaling on the approach due to this behavior, it was fully embraced alongside the following implications:

1. Lower debuggability while setting up the DRT legality constraints in the derived `vertex` class. For example, a particular vertex could be constrained so that it could never be connected directly, or through its children, to `STOP` and hence, never be part of a tree. Each time this vertex is selected for randomization, it will either fail or the respective randomization of one of its descendants will fail. This will not be directly noted, as these failures will simply trigger a new randomization iteration. Ultimately, such issues would be caught in the coverage closure step.
2. The approach requires that the simulation does not terminate due to a randomization failure. Hence, if needed, the simulator shall be configured to not exit or produce a fatal by default on a randomization failure. This means that other test bench code should always explicitly check for randomization failures and handle such failures appropriately, e.g., by invoking `uvm_fatal`. Such coding practices are standard in the industry and part of our coding guidelines; hence it was deemed a non-issue.

RESULTS

The development of the DRT randomization scheme described in this paper was part of an ASIC project effort to verify a statistics unit in an image processing pipeline. As such, once a viable solution was in place, there was no additional need to produce a different method. Hence it is not possible to do a comparative analysis of a spread of solutions. Nevertheless, it is possible to analyze the outcome in terms of how it performed against the previously listed criteria. We can additionally guess how it would compare against a flat, single `randomize()` call approach, but again, this would be only an educated guess as no such solution was developed.

Validity

An approach involving a single `randomize()` call was explored, but such efforts did not produce a valid solution. This does not exclude that there exists such a method in the solution space. On the other hand, given a legal set of constraints describing the set of possible trees, the recursive approach is guaranteed to produce a DRT structure satisfying all the given rules. This was successfully proven during the ASIC project, where all the coverage metrics related to the tree structure were covered.

Controllability

Closing coverage of the statistics unit involved developing several tests to target hard-to-hit functional range related to algorithmic functions inside the different sampler and converter sub-units. For obvious reasons, when targeting such functional coverage, the respective sub-units needed to be part of the randomized tree. Given the number of possible trees for n labeled vertices, the probability of any particular vertex being in a generated tree could be quite low for certain vertices. Hence these tests constrained the DRT configuration space to force the required sub-units to be part of a branch in the generated tree in very much the same way as shown in Fig. 5.

Performance

The general constraint satisfaction problem (CSP) is an NP-complete problem [6]. The described solution splits the randomization of the tree into N different `randomize()` calls, where the number N is proportional to the number of available vertices. This reduces the input size, i.e., the number of rand variables, in each `randomize()` call. Hence it is a reasonable claim that this recursive approach would be lighter to execute for a generic constraint solver than randomizing the complete structure in a single call. Although we can only guess how the recursive solution would fare against other potential solutions to the same problem, we *can* compare against the randomization of stimulus objects of other typical image processing unit test benches. When doing so, it was measured that the recursive randomization did not perform any worse than these, with the percentage of the total CPU time spent in the constraint solver step within the normal ranges for this type of test bench.

Scalability

As noted earlier, the performance of a recursive solution is proportional to the number of vertices added to the tree. In the project where this approach was first used, the number of available vertices for a particular unit was twenty-five. In a later project, the number was increased to thirty-two. This did not incur any noticeably increased CPU time. Furthermore, adding the respective constraints to incorporate the new vertices comprised a reasonable effort.

To get a more accurate measurement of how the solution scales, a simple UVM environment was created. Fixed relationships were used between the total number of available vertices, the number of converter and sampler vertices, and the maximum number of branches. Performance measurements were captured by using the profiler feature from the simulation tool. Hundreds of randomizations were performed for each measured point, and the values were averaged.

Fig. 11 (a) shows how the total solver runtime scales with the available number of vertices. The data points from the graph exhibit a polynomial time complexity (P). This is the best expected outcome, given the widely accepted conjecture in the area, that a CSP for a given fixed constrain language is either P or NP-complete [7].

Fig 11 (b) shows the average time for each `randomize()` call. The scaling here looks almost linear, and the larger input space can explain the increased effort by the solver. It is worth noting that this is a made-up example with relatively simple constraints, i.e., connectivity between the vertices. A real-world problem would likely have a more complex relationship affecting the performance.

All-in-all we assess that the solution is computationally tractable and does scale favorably with the number of vertices.

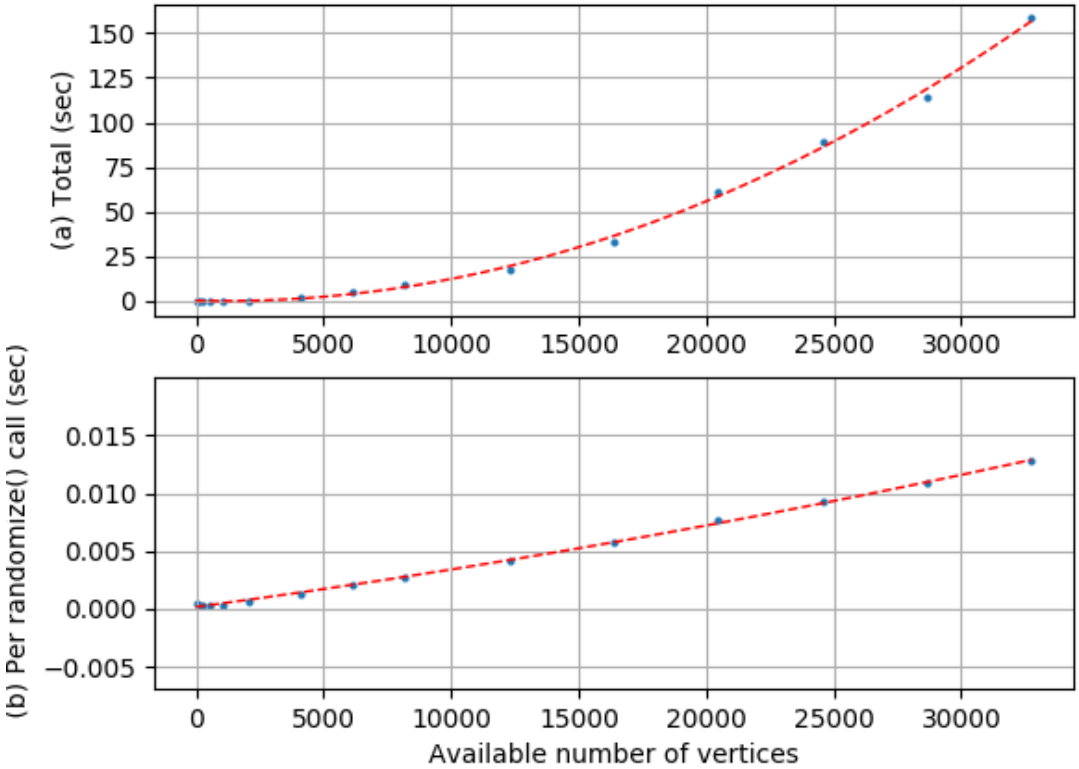

Figure 11. Performance scaling of the recursive algorithm

It is also worth noting that the developed framework is solely based on UVM and provides an API to plug in any `uvm_object` class containing the register configuration of a DUT (see the `config_t` parametrization of the `vertex_base` class). In other words, it is a generic framework that can be generally applied to generate DRTs for any application that requires it, provided that the TB is based on UVM.

Conclusion

The randomization of stimuli in constrain-random test benches usually favors a single randomization step to allow a solver to produce a solution for a set of related random variables. This makes it easy to express the relationship between such variables solely using constraints.

This paper shows an alternative approach to the problem of modeling and randomizing a valid DRT structure, where the tree was incrementally built using a back-tracking algorithm. This allowed us to successfully complete the verification of complex statistics units with configurable data paths. The developed framework provides a convenient mechanism to control the randomization and enables a constraint solver to produce a solution within bounded runtime.

References

[1] J. Nakamura, *Image Sensors and Signal Processing for Digital Still Cameras*. CRC Press, 2005.
[2] A. Cayley, “A Theorem on Trees,” Quarterly Journal of Mathematics, vol 23, pp. 376–378, 1889.
[3] IEEE 1800-2017 - IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language, 2017.
[4] IEEE 1800.2-2017 - IEEE Standard for Universal Verification Methodology Language Reference Manual, 2017.
[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
[6] U. Montanari, “Networks of Constraints: Fundamental Properties and Applications to Picture Processing,” Information Sciences, vol. 7, pp. 95–132, 1974.
[7] Tomas Feder and Moshe Y. Vardi, “The Computational Structure of Monotone Monadic SNP and Constraint Satisfaction: A Study through Datalog and Group Theory,” *SIAM Journal on Computing*, vol. 28, no. 1, pp. 57–104, 1998.