White Paper

The Benefits of Anonymity

A MODEST PROPOSAL TO ENHANCE THE SYSTEMVERILOG VERIFICATION LANGUAGE

15 NOV 2019

Revision 1.0, 15-NOV-2019

Adam Rose adam.rose@verilab.com

www.verilab.com

Table of Contents

1 Introduction

1.1 Motivation

This document has a number of streams of thought running into it. The first is a long standing prejudice against macros, particularly complicated macros. Recourse to macros is usually an example of language failure. If we can only succinctly express our thoughts using a “hard working” macro, then there is probably something missing from the language.

The second was a detailed engagement with python just before and during my recent masters degree. Python has a lot of really cool language features such as functions as first class objects, lambda syntax and executable classes. It was fun to learn and use, but for me the experience cut the gordian knot you get into when debating the relative merits of e and SystemVerilog as verification languages. The conclusion I came to is that we should have built verification languages on top of python.

The third stream of thought was my recent re-engagement with C++. C++ had for many years been a fairly stable language, at least at its core. For many years, the main development was the movement of features such as the standard template library into the core language. But recently the language has undergone some surprisingly rapid change, incorporating some of the nice features that I had previously found in python, such as lambda syntax, pleasant loop syntax and functions as objects.

The final stream of thought is some of the ideas that I have been exposed to recently at Verilab such as policy constraint classes and N-Way randomization.

Finally – I am not a language lawyer. This is not intended to be a precise or complete or even consistent formal definition. The intention is to explain the basic ideas and to start a discussion.

1.2 Overview

Section 2 introduces some usability improvements to SystemVerilog. These are not fundamental to what follows but would make it easier to produce concise and maintainable code. Readers not interested in these suggestions can skip straight to Section 3, which introduces the key ideas of the lambda syntax and treating constraints and methods as first class objects.

Sections 4 to 6 show how to apply these techniques, introducing refinements to the basic syntax as we proceed. Section 4 introduces introspection and a replacement for the uvm_field macros. Section 5 shows how to implement and apply method decoration, a pattern used in python, to problems like transaction recording of tasks. Section 6 shows how to replace the uvm_factory.

Section 7 briefly discusses compatibility requirements and Section 8 summarises the key arguments.

2 Some Useful Language Enhancements

We start with some language enhancements that are not directly related to the main subject of this document, but which do improve the conciseness and maintainability of SystemVerilog code [1].

2.1 Loop Syntax

SystemVerilog nearly has a pleasant loop syntax, but not quite, since you effectively have to index the loop twice eg

```
foreach( list[i] ) list[i].print();
```

This could be improved to something like:

```
for( p in list ) p.print();
```

2.2 Import / Export

A significant weakness of the SystemVerilog package mechanism is its lack of support for composition of packages. As a result, packages consist of large lists of include files rather than short lists of import statements. This leads to name bloat, and also makes it hard to make relatively minor changes to a larger package. For example, with the existing syntax, we are forced to do the following:

```
package a1_pkg;
  class A1;
  endclass
endpackage

package a2_pkg;
  class A2;
  endclass
endpackage

package b_pkg;
  import a1_pkg::*;
  import a2_pkg::*;
  class B;
    A1 a1;
    A2 a2;
  endclass
endpackage

package c_pkg;
  import a1_pkg::*;
  import a2_pkg::*;
  import b_pkg:*;

  class C;
    A1 a1;
```

```
    A2 a2;
    B b;
  endclass
endpackage
```

In the c_pkg above, we have to import a1_pkg and a2_pkg, even though we have already imported b_pkg which has already imported those two packages. The fix for this is very simple : simply allow import pkg::* and export pkg::* in the same package. For example:

```
package a1_pkg;
  class A1;
  endclass
endpackage

package a2_pkg;
  class A2;
  endclass
endpackage

package b_pkg;
  import a1_pkg::*;
  import a2_pkg::*;
  class B;
    A1 a1;
    A2 a2;
  endclass
  export a1_pkg::*;
  export a2_pkg::*;
endpackage

package c_pkg;
  import b_pkg:*;
  class C;
    A1 a1;
    A2 a2;
    B b;
  endclass
endpackage
```

Enabling composition in this way will allow the creation of larger packages from sets of smaller ones while still allowing users of the package to import a single package. Composing large packages from smaller sub-packages also allows users to only import a sub package if they only need a subset of the larger package's functionality.

2.3 Parameterized Functions and Automatic Inference

Not having parameterized functions is a pain. It means we often have to define a class with a single static method, just so that we can parameterize a function.

So instead of:

```
class named_object_printer #( type T );
  static function void print( string name , T t );
```

```
    $display("%s %s" , name , T.to_string() );
  endfunction
endclass

T t = new();
named_object_printer #( T )::print("a" , this );
```

we can have the simpler:

```
function #( type T ) void name_print( string name , T t );
  $display("%s %s" , name , t.to_string() );
endfunction

T t = new();
name_print( "a" , t );
```

The call to name_print above is parameterized, but we have not specified the parameter. Instead, we use automatic inference to work out what the parameter should be.

2.4 Parameter Specialisation

The code above works for classes, but not literals. What if we want to write the same user code that works differently for different types ?

```
int i = 0;
T t = new();

name_print("i" , i );
name_print("t" , t );
```

Parameter specialisation enables precisely this. The example below is for functions, but the same principle can apply to classes.

```
function #( type T ) void name_print( string name , T t );
  $display("%s %s" , name , t );
endfunction

function #( object_base ) void name_print( string name , T t );
  $display("%s %s" , name , t.to_string() );
endfunction
```

Now, if T inherits from object_base, it will call the to_string() method, otherwise it will just print t itself. This is a good argument for a root class that all classes inherit from. Even if there's not much in such a base class, having one allows us to specialize on the basis of built-ins vs classes.[1]

3 Constraints and Methods as First class Objects

[1] Actually, since every class has randomize, pre_randomize and post_randomize methods, you could argue that there is already such a base class, although it's not yet made explicit.

3.1 Constraints as first class objects

Having constraints tied into the class hierarchy of an object is not ideal. It means that constraints clutter up the definition of the class, and it also makes constraint policies a little involved to implement. Sure, sometimes we do want to build the constraint into the class hierarchy but there are also good reasons for not doing so for other constraints.

What follows is a suggested syntax, but the main idea is that constraints should be first class objects that can exist outside of a class. Since a constraint relates to a class, we can take advantage of scope syntax to express that relationship, but there are probably other ways of doing it.

```
class trans;
  rand int x;
endclass

constraint trans::max { x < 100; };
constraint trans::min [ x > 0; };

constraint trans::in_range { max && min; }

trans t = new();

t.randomize( { min , max } );
t.randomize( { in_range } );
constraint trans::modified_max extends max { x != 90; }

t.randomize( { min , modified_max } );
```

3.2 Lambda Methods and Methods as first class objects

3.2.1 Introduction to Lambda Methods

Lambda methods are sometimes also known as anonymous methods or closures.[2] It's a way of quickly creating a method and passing this method elsewhere. One unique feature of lambda methods is that they can capture data from the place where they are defined, and then use this in the place that they are called. One way of thinking about it is that it provides syntactic support for callbacks, although it has a wider application than that. The syntactical sugar for creating the function in the first place is not much use without some syntax for passing around functions as if they were variables.

The classic case for using lambda syntax is inside loops. First of all we create a function which itself takes a function as one of its arguments:

```
function #( type T )
void print_selected( T list[$] , function bit selector( trans ) );
  for( p in list ) if selector( p ) $display("%s" , p.to_string() );
```

[2] Actually, they are referred to as lambda functions, since other languages only have functions, not tasks and functions.

```
endfunction
```

Then we create local anonymous functions and pass them into the function we have just created:

```
int threshold = 20;
void print_selected( t_list ,
                     anon( threshold ) function( trans t );
                       return t.x >= threshold; );

auto less_than = anon( threshold ) function ( trans t )
                   return t.x < threshold;

void print_selected( t_list , less_than );
```

The two anonymous functions above use the anon operator to capture their thresholds, and are then used to print elements of the list that are above or below that threshold. The code above also uses auto declarations, where the compiler deduces the type of the object from the thing assigned to it - which is in this case a function.

Having established the basic idea, it is clear that there are obvious applications to callbacks, ports, etc.

3.2.2 uvm_do

We now have enough new syntax to rewrite uvm_do and the other similar macros. [2]

```
class uvm_sequence #( type ITEM_TYPE ) extends uvm_object;
  …
  task uvm_do( constraint c = null ,
               function void post_randomize( trans t ) = null );

    ITEM_TYPE t = ITEM_TYPE::type_id::create("t");

    start_item( t );
    if( !t.randomize( { c } );
      `uvm_fatal( … );
     If( post_randomize != null )
       post_randomize( t );
    finish_item( t );
  endtask
endclass
```

uvm_do is now a task that takes an optional constraint and an optional function as arguments. Randomize has been modified to allow constraints to be passed to it.

Now we use lambda constraints and lambda functions to write code like the code below, all with no macros.

```
  task body();
    int priority = 4;
    bit parity = 0;
```

```
    uvm_do();
    uvm_do( anon(parity) constraint trans::{ x[0] == parity; } );
    uvm_do( anon(parity) constraint trans::{ x[0] == parity; } ,
            anon(priority) function(trans t)
               t.set_priority(priority); );

    auto c = anon(parity) trans::constraint { x[0] == parity };
    auto p = anon(priority) function(trans)
               t.set_priority(priority);

    uvm_do( c , p );

    auto c1 = anon constraint trans::{ x[31] == 1; }[3]
    uvm_do( c && c1 , p );
  endtask
```

The last line is a lambda-syntax implementation of constraint policies. [3]

4 Introspection

4.1 uvm_field_* macros

The real target of this line of thought is the ‘uvm_field_* macros. The suggestions above lay the ground work for a syntactically pleasant re-implementation of these macros, but we are not quite there.

In order to replace the ‘uvm_field_ macros, we need an introspection mechanism. In order to do this, we provide all variables with a set of useful methods such as compare, print, record, pack and unpack. By default, these have the “obvious” default implementations. We would probably also add name, type, and size to the introspection mechanism, and possibly also accessor methods such as set and get.

Then, to provide custom implementations or to turn the behaviour off altogether, we would write code like this:

```
class C;
  int x;
  bit y;
  int list[$];
  D m_d;
  E m_e;

  function new( E e );
    super.new();

    m_d = new();
    m_e = e;

    list.compare = anon function(int l1[$], int l2[$]);
```

[3] And hardly worth documenting … we are declaring new variables in the middle of a code block.

```
                                    return l1[0] == l2[0];
    m_e.compare = null;
  endfunction
endclass
```

In the code above, we have used the introspection mechanism to modify two of the classes compare methods. The compare method for the field “list” only takes into account the zeroth element (for simplicity, ignoring the possibility of zero size lists), while there is no compare method for the field “m_e”.

4.2 Other Applications for Introspection

Now that we have an introspection mechanism, we should be able to do all sorts of things with it. For example, there should be a way to create the underlying infrastructure for N-Way randomization. [4]

5 Method Assignment and Decoration

A strong interpretation of “methods as first class objects” is to allow assignment to methods just like we would for variables. Python takes “methods as first class objects” to an interesting extreme, enabling a pattern called decoration. This chapter explores some syntax that enables this useful pattern in SystemVerilog.

5.1 Argumentless methods

SystemVerilog currently follows Verilog by not distinguishing between m and m() when m is an argumentless method. Unfortunately, this makes assignment of named methods a little awkward. For example, if we write:

```
function int g();
  return 0;
endfunction

auto f = g;
```

is f an int function or an int ?

In what follows below, we assume that when assigning to auto or to a method variable, the answer is “int function” not int. In other words, we return the method itself and not the value returned by the method. If this doesn’t work, we may have to add some non-optimal syntax to make it clear which we mean, for example:

```
auto f = method( g );
```

5.2 A trivial example of method assignment

Below is a somewhat trivial example, using the assumption about argumentless methods outlined above:

```
function int one();
  return 1;
endfunction

function int two();
  return 2;
endfunction

auto temp = one;
one = two;
two = temp;

assert( one() == 2 );
assert( two() == 1 );
```

5.3 Transaction Recording using Decoration

This example above is trivial, but the technique does allow us to do some interesting things. For example, we can add transaction recording to a task:

```
task execute( trans t );
  …
endtask

function #( type T ) auto add_recording( task e( T t ) );
  return anon( e ) task( T t1 );
    t1.begin_recording();
    e( t1 );
    t1.end_recording();
  endtask;
endfunction

execute = add_recording( execute );
```

Now execute will begin and end transaction recording whenever it is called. This is an example of decoration, since we have decorated a task with transaction recording. [5]

5.4 Class methods and Decoration

We have shown above how to decorate global methods. But we will want to do the same for class methods.

One way to do this is pass in “this”, allowing us to call public methods in the transaction:

```
class trans;
  task execute();
    …
  endtask
endclass

function #( type T ) auto add_recording( T this , task e() );
  return anon #( this , e ) task( T t1 );
```

```
    this.begin_recording();
    this.e();
    this.end_recording();
  endtask;
endfunction

trans.execute = add_recording( trans , trans.execute );
```

Whatever syntax we use, we need some way to decorate class methods.

5.5 Function Declarations with No Function Definition

In most of the code above, we have avoided the issue of how to explicitly declare a function or task variable by extensive use of "auto". If we want to be precise about the type, we need to be able to explicitly declare functions and tasks that have no implementation.

However, in SystemVerilog as it currently is, the following code is illegal:

```
function int f();

class C;
endclass
```

since we have not completed the definition of function f before we start the class definition.

But this is exactly what we want to do if we're not going to use auto. To return to the trivial example above, we might want to write:

```
function int temp();

temp = one;
one = two;
two = temp;
```

While we can sometimes avoid the issue by initialising the function to be a previously defined function:

```
function int temp() = one;

one = two;
two = temp;
```

There will be occasions we are not able to do this, so we need some syntax to help compilers understand we have defined a method with no implementation. We can do this by assigning it to null:

```
function int temp() = null;
```[4]

[4] The meaning of "= null;" is quite similar to the "pure" keyword, and reminiscent of the C++ syntax used for pure virtual methods. Maybe "= null" becomes a synonym for "pure" when used to define a pure virtual method ?

```
temp = one;
one = two;
two = temp;
```

6 Anonymous Classes and the Factory

One way to implement the factory pattern using the new syntax is to use function assignment to replace the new method:

```
class C;
  int x;
endclass

class D extends C;
  int y;
endclass

C::new = D::new;

C c = new(); // c is actually a “D”

c.x = 2;
D d;

$cast( d , c );
d.y = 3;
```

The instance c is actually a D because we changed C::new to be the constructor for D. But a syntax based on anonymous classes is more succinct:

```
class C;
  int x;
endclass

C = anon class extends C;
  int y;
endclass

C c = new(); // c is actually an anonymous extension of C

c.x = 2;
c.y = 3;     // no need for $cast
```

The code above uses an anonymous extension of C in order to add “int y” to the definition of C.

7 Backward and Forward Compatibility

7.1 Backward Compatibility (Language)

We need to maintain backward compatibility of the language, so that code that worked in older SystemVerilog versions continues to work in later SystemVerilog versions

The only exceptions to this are where we have defined new keywords which mean that they cannot be used as names, eg “anon” and “auto”.

7.2 Forward Compatibility (IP)

IP in existing testbenches includes VIP, sequences, tests, coverage and scoreboards. There is a huge investment in such IP in the industry.

We would need to maintain forward compatibility of IP, so that that old IP can work alongside new IP in the enhanced versions of the language and the methodology that used the new language features.

8 Summary

We have suggested a number of concise enhancements to the SystemVerilog language, inspired by developments in mainstream programming languages. The most important of these are to treat methods and constraints as first class objects, and to introduce a lambda syntax.

These enhancements increase the expressive power of the language sufficiently to avoid the need for hard working macros such as the various uvm_do_* macros. By introducing an introspection API, we can avoid the need to in effect declare each variable twice, once to declare it in SV and then again using the uvm_field_* macros to give it the required set of compare, print, copy, record and pack/unpack methods. The new features also enable the method decoration pattern, which can be used to implement features like transaction recording for task calls. We have also shown how to use these features to implement a native factory mechanism, avoiding the need for the uvm_factory.

In many ways, these enhancements simply bring SystemVerilog up to date with recent developments in programming languages. By using them, Verification engineers will be able to enhance their productivity by writing more concise, modular and maintainable code.

References

[1] IEEE Std 1800-2012, *IEEE Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language,*.

[2] Accellera, “UVM Website,” [Online]. Available: http://www.accellera.org/downloads/standards/uvm.

[3] J. Dickol, “SystemVerilog Constraint Layering via Reusable Randomization Policy Classes,” in *DVCon* , 2015.

[4] K. Johnson and J. Bromley, “Is Your Testing N-wise or Unwise?,” in *DVCon Europe* , 2015.

[5] “Python Tutorial : Easy Introduction into Decorators and Decoration,” [Online]. Available: https://www.python-course.eu/python3_decorators.php. [Accessed 11 11 2109].