AMD

SystemVerilog: Reusable Class Features, and Safe Initialization of Static Variables

Will Adams

Advanced Micro Devices, Inc.

Austin TX, USA

http://www.amd.com

ABSTRACT

Parameterized inheritance, that is, inheritance from a class given as a parameter, allows features to be coded once and used in multiple classes. It allows some of the flexibility that mixins bring to a language that supports them. The technique is more limited that mixins, but this is not a problem in many cases, in particular UVM components and objects. A couple of examples defining reusable features for UVM component classes are used to illustrate the technique.

Static variables exist for the lifetime of a program, and are initialized at the start of the program. The support for static variables in the various Verilog and SystemVerilog standards is described, and it is shown that none of them ensure that a static variable is initialized before it is referenced. Ways to ensure initialization of static variables before use are described. Guidelines are provided for the safe initialization and use of static variables.

Table of Contents

Table of Figures

1. Reusable Class Features

1.1 Introduction

A *feature* is a set of properties in a class, along with the methods used to access them. For example, a feature for an object name requires a string-type variable for the value of the name, plus methods to read and write the value. Other methods may be included (e.g., methods to format the name for printing), as required by the code.

Within a collection of related classes, there may be several features that are common to two or more classes. Inheritance allows classes to share the implementation of a common feature, so it is only coded once, and has consistent behavior in all classes that include it.

However, single inheritance (as is supported by SystemVerilog) is not sufficient in all cases. Suppose, in a set of classes, some have feature *name*, some have feature *ID*, and some have both. With single inheritance, it difficult to code a class implementing *name*, one implementing *ID*, and one implementing both, without duplicating code. Furthermore, a class that inherits from an external class cannot also inherit from a locally-defined class implementing the features.

1.2 Mixins

A *mixin* is an implementation of a feature that can be used in any class. If a set of mixins is provided, a user can create classes having any subset of the features by using the appropriate mixins. Mixins were first introduced in Flavors (Cannon, 1982), and have since been adopted in other languages, either by providing explicit support for them in the language, or by using existing language features to implement them.

In an object-oriented language that supports multiple inheritance (such as C++), mixins can be implemented as simple base classes. For each feature, a base class is provided that includes the data and methods for the feature, including constructors with the appropriate arguments. Any class that requires the feature adds the mixin class to its list of public base classes, and initializes the base object by calling one of its constructors.

For example, suppose C++ classes `name_feature` and `id_feature` implement features *name* and *ID*. Any class, including classes that inherit from external classes, can add either `name_feature` or `id_feature`, or both, to its list of base classes. Thus the implementation of these features is shared by all classes that use them.

1.3 Adding features to SystemVerilog classes

SystemVerilog does not provide a way to support mixin classes. Multiple inheritance of implementations is not supported.[1]

However, a restricted form of mixins that work with single inheritance can be used. The restrictions require that the classes using the feature have the same constructor arguments. Fortunately, for common cases, such as UVM components and objects, these restrictions are imposed by the environment (all UVM components have the same constructor arguments, in order to support factory creation of components).

---

[1] Interface classes, which were introduced in IEEE Std 1800-2012 (IEEE, 2013), allow multiple inheritance of class interfaces (method declarations), but not of class implementations.

The next sections use examples to explain and illustrate the method.

1.4 Example: Common configuration field

As an example, consider adding support for a common configuration field to a UVM component. The code in Figure 1 shows the definition of a class `enable_component` that adds support for an `enable` configuration field to type `uvm_component`.

Class `enable_component` is derived from `uvm_component`. It defines a local bit variable `enable` and declares this as a configuration field, so it can be set by other components or from the command line, using the configuration mechanism. It also provides a method `is_enabled` that returns the value of `enable`. The initial value of `enable` is specified by parameter `INIT`. A class derived from `enable_component #(.INIT(1'b1))` has an `enable` configuration field that is initially `1'b1`, and all the properties and methods of `uvm_component`.

Automatic configuration, which is run during `uvm_component::build_phase`, sets the value of `enable` according to settings in the configuration database that apply to the field. There is no other code in the class that sets the value of `enable`, and it is declared local, which makes it inaccessible from other code, so `enable` is set to `INIT` when the component is constructed, is set during automatic configuration if any values have been specified for it (provided all subclasses of `enable_component` call `super.build_phase` in `build_phase`, or do not override `build_phase`), and then does not change value for the remainder of program execution.

If `enable` were public or protected, its value could be directly accessed from outside the class, so there would be no need for function `is_enabled`. However, in this case, its value could be changed by any code that has access to it, so the above behavior cannot be guaranteed. Using a local variable with an accessor function means that code can only write enable using configuration options or

```
class enable_component
  #( bit INIT = 1'b1 )
extends uvm_component ;

  `uvm_component_param_utils_begin( enable_component )
    `uvm_field_int( enable , UVM_DEFAULT )
  `uvm_component_utils_end

  function new( string name , uvm_component parent = null ) ;
    super.new( name , parent ) ;
  endfunction

  function bit is_enabled( ) ;
    return enable ;
  endfunction

  local bit enable = INIT ;

endclass
```

Figure 1: Class `enable_component`

```
class enable_component
  #( type T = uvm_component , bit INIT = 1'b1 )
extends T ;
```

Figure 2: Class `enable_component` with a type parameter

function calls, and has read-only access after automatic configuration has been run, which helps avoid bugs caused by external code changing the value in unexpected ways.

An `enable` configuration field can be added to any test bench component class that has `uvm_component` as a direct base class by replacing `extends uvm_component` in the class definition with `extends enable_component`. The same replacement cannot be used for classes extended from subclasses of `uvm_component` (for example, `uvm_scoreboard` and `uvm_driver #(REQ,RSP)`).

The definition of class `enable_component` is changed as shown in Figure 2 so it can be used to add `enable` to any UVM component type. Type parameter `T` is added to the class parameter list, and the class is extended from `T` instead of from `uvm_component`. The rest of the class is identical to the version in Figure 1, and contains no uses of `T`.[2]

Figure 3 shows some examples of the use of `enable_component #(T,INIT)`. Class `ip_scoreboard` is a typedef for a subclass of `uvm_scoreboard` that includes a configuration field `enable` with initial value `1'b1` (the default value of `INIT`). Parameterized class `ip_driver #(REQ,RSP,EN_INIT)` is extended from `uvm_driver #(REQ,RSP)`, and has configuration field `enable` with initial value `EN_INIT`.

```
typedef enable_component #( .T( uvm_scoreboard ) ) ip_scoreboard ;

class ip_driver
  #( type REQ = uvm_sequence_item , type RSP = REQ, bit EN_INIT =
1'b0 )
extends
  enable_component
    ( .T( uvm_driver #( REQ , RSP ) ) , .INIT( EN_INIT ) ) ;

  `uvm_component_param_utils( ip_driver ) ;

  function new( string name , uvm_component parent = null ) ;
    super.new( name , parent ) ;
  endfunction

endclass
```

Figure 3: Components defined using `enable_component #(T)`

[2] The use of a type parameter to code mixin-like classes is discussed in a few places online, for example (Timi, 2014).

1.5 Example: Reset port

Many test bench components must respond to reset events in the RTL. A common way to enable this is to create a reset monitor that observes the reset logic in the RTL and sends an item on an analysis port when reset is entered or exited. Any component that needs to respond to reset events implements an analysis implementation port for the reset items, and the reset monitor's analysis port is connected to the reset port on each component, directly or via some intermediate ports.

Class `reset_component`, shown in Figure 4, adds a reset port to its base class. It uses parameterized inheritance, so it can be used to extend any component type. It provides function

```
class reset_component
  #( type T = uvm_component )
extends T ;

  `uvm_component_param_utils( reset_component )

  // Public analysis export
  uvm_analysis_export #( reset_item ) reset_xp ;

  // Local analysis implementation port
  `uvm_analysis_imp_decl( _reset)
  local uvm_analysis_imp_reset
    #( reset_item , reset_component #( T ) )
    reset_imp ;

  function new( string name , uvm_component parent = null ) ;
    super.new( name , parent ) ;
    reset_xp  = new( "reset_xp"  , this ) ;
    reset_imp = new( "reset_imp" , this ) ;
  endfunction

  function in_reset( ) ;
    return reset ;
  endfunction

  virtual function void connect_phase( uvm_phase phase ) ;
    super.connect_phase( phase ) ;
    reset_xp.connect( reset_imp ) ;
  endfunction

  function void write_reset( reset_item item ) ;
    reset = item.reset ;
  endfunction

  local bit reset ;

endclass
```

Figure 4: Class `reset_component`

in_reset, which subclass objects can use to check the current reset state of the RTL. Code assumes that the reset monitor sends items of type reset_item, which includes field reset that contains the current value of the reset signal in the RTL.

The class declares two ports. Port reset_xp is a public analysis export, and should be connected externally to the analysis port of a reset monitor. Port reset_imp is a local analysis implementation port, and is connected internally (in connect_phase) to reset_xp, so any reset event sent to reset_xp is sent to reset_imp. Function write_reset, which implements reset_imp, sets the local variable reset to the value of field reset in each reset item, and function in_reset returns the value of reset.

Class reset_component could be implemented with a single public analysis implementation port, but using two ports allows subclasses to connect the reset port to child components that require it. A subclass of reset_component that has a child component with a reset analysis export should connect reset_component::reset_xp to the reset export on the child component. If reset_component had a public analysis implementation port, and no analysis export, this would not be possible, since analysis implementation ports can only be connected to receive items, not to send them.

Provided subclasses connect child component reset ports correctly, a component that instantiates an instance of a class derived from reset_component need only connect to the top-level port reset_xp to ensure that the instance and all its sub-components receive reset items.

1.6 Combining features

Figure 5 shows an example with three features for UVM components, feature0, feature1 and feature2, and some UVM components that use these features. The two typedefs define classes feature012_component, which extends uvm_component with all three features, and feature02_monitor, which extends uvm_monitor with feature0 and feature2. The base

```
class feature0 #( type T = uvm_component ) ; … endclass
class feature1 #( type T = uvm_component ) ; … endclass
class feature2 #( type T = uvm_component ) ; … endclass

typedef
  feature0 #( feature1 #( feature2 ) )
  feature012_component ;

typedef
  feature0 #( feature2 #( uvm_monitor ) )
  feature02_monitor ;

class feature12_driver
  #( type REQ = uvm_sequence_item , type RSP = REQ )
extends feature1 #( feature2 #( uvm_driver #( REQ , RSP ) ) ) ;
  // …
endclass
```

Figure 5: Combining features

class for class `feature12_driver` is `uvm_driver #(REQ,RSP)` extended with `feature1` and `feature2`.

1.7 Conclusion: resusable features with parameterized inheritance

For a collection of classes that have the same constructor arguments, parameterized inheritance provides a mechanism for implementing a set of reusable features each of which may be added to any of the classes. It is particularly useful for adding features to UVM components, since these are required to have the same constructor arguments. The technique offers some of the power that mixins bring to languages that support them.

2. Safe Initialization of Static Variables

2.1 Introduction

Static variables exist for the lifetime of the program, and are initialized at the start of the program. If static variable `S` uses the value of static variable `T` in its initialization expression, SystemVerilog does not guarantee that `T` is initialized before `S`, so `S` may be initialized to an unexpected value. The rules for defining and initializing static variables have changed over the various Verilog and SystemVerilog standards. Section 2.2 discusses how different standards define static variables. Section 2.4 shows that the various definitions all allow a static variable to be accessed before it is initialized. Section 2.5 discusses how this problem is addressed in C and C++, and Sections 2.6 – 2.9 show how to adapt the C++ solution to work with SystemVerilog.

2.2 Static and automatic variable lifetime in Verilog and SystemVerilog

IEEE Std 1364-1995 (IEEE, 1996), the first IEEE Verilog standard, specified a static lifetime for all variables. For each variable defined in a module, there is a copy of the variable for each instance of the module. Each copy exists for the lifetime of the program.

Variables defined inside a subroutine (including the arguments and the return value, if any) are treated the same as other module variables. For each instance of the module containing the subroutine, there is a separate copy of each variable associated with the subroutine. A subroutine variable exists for the lifetime of the program, and retains its value between calls to the subroutine.

One result of this definition is that subroutines are not reëntrant. That is, two calls to the subroutine concurrently from two processes may interfere, and overwrite each other's results.

Hierarchical references allow code outside a subroutine to read and write the variables defined in the subroutine. If subroutine `sub` defines a variable `t` outside of any block, code outside the subroutine can access the variable using the hierarchical reference `sub.t`.

Each variable is set to the default value for its type when it is created. An initial block can be used to set the value of a variable to something other than the default.

IEEE Std 1364-2001 (IEEE, 2001), the Verilog-2001 standard, introduced the keyword `automatic`. This can be applied to a subroutine definition, and has the effect of giving all variables in the subroutine automatic lifetime. That is, new copies of the variables are created each time the subroutine is invoked. Each variable is initially set to the default value for its type, and its final value is discarded. This allows reëntrant subroutines, so two processes can concurrently call the same subroutine without interference between the two invocations. All variables not in automatic subroutines are static, as previously.

IEEE Std 1364-2005 (IEEE, 2006), the final IEEE Verilog standard, allows an assignment of an initial value to a variable in its declaration (a *variable declaration assignment*). The initial value must be a constant expression. For a static variable the assignment is executed in an implicit initial block. The implicit initial blocks are executed along with the other initial blocks in the program.

IEEE Std 1800-2005 (IEEE, 2005), the first IEEE SystemVerilog standard, adds finer-grain control over variable lifetime, and changes the way that static variables are initialized. Variable lifetime may be declared as automatic or static for modules, interfaces, programs, subroutines, and variables.

A variable lifetime declaration for a module, interface or program sets a default lifetime for the subroutines declared in the component.

A subroutine has its declared variable lifetime, if any. If none is declared, the lifetime is automatic if the subroutine is a class method, or in a component that is declared automatic, and static otherwise.

Only variables declared in a subroutine or an initial or always block may be declared automatic. A variable that does not declare its lifetime has automatic lifetime if the variable is declared in a subroutine with automatic variable lifetime, or if it is a for-loop variable, and static lifetime otherwise.

In SystemVerilog, any expression is allowed for the value in a variable declaration assignment. The expression is evaluated at run-time, when the assignment is executed. The assignments for static variables are executed before any initial blocks are executed.

2.3 Static variable lifetime and static methods

SystemVerilog has adopted from C and C++ an unfortunate overuse of the keyword `static`. It is used for several similar but distinct concepts. In SystemVerilog, a class method may be declared static. This is different from declaring static variable lifetime for the method, though both require the use of the keyword `static`. Examples of the four possible combinations of static and non-static methods with static and automatic variable lifetimes are shown in Figure 6.

Later versions of the SystemVerilog standard (IEEE Std 1800-2009 (IEEE, 2009) and later) state (in a comment in a code example) that it is illegal to declare a non-static method with static variable lifetime (so method `funcs::ns()` is not legal). The same result can be achieved by explicitly declaring each variable in the subroutine with static lifetime, so this merely forbids a syntactic shorthand, and does not restrict the possibilities.

2.4 The static initialization problem

Verilog-95 and Verilog-2001 both use initial blocks to initialize static variables. This means that the initialization of these variables is executed along with the other initial blocks, in arbitrary order. The code in Figure 7 shows static variable `T` that is initialized by initial block `T_init`, and static variable `S` that is initialized by initial block `S_init`. The variables are in separate source files.

Variable `T` has value `1'b0` from the time it is created until the assignment in `T_init` and is `1'b1`

```
class funcs ;

  // static method, static variable lifetime
  static function static int ss( ) ; /* code */ endfunction

  // static method, automatic variable lifetime
  static function int sa( ) ; /* code */ endfunction

  // non-static method, static variable lifetime
  function static int ns( ) ; /* code */ endfunction

  // non-static method, automatic variable lifetime
  function int na( ) ; /* code */ endfunction

endclass : funcs
```

Figure 6: Static and non-static methods, with static and automatic variable lifetimes

```
static bit T ;          static string S;

initial                 initial
begin : T_init          begin : S_init
  T = 1'b1 ;              S = ( T ? "ENABLED" : "DISABLED" ) ;
end                     end
```

Figure 7: Static variables T and S, initialized by initial blocks

afterwards. The initial blocks are executed in either order. If `T_init` is executed first, the values assigned to `T` and `S` are `1'b1` and `"ENABLED"`. If `S_init` is executed first, the values assigned to `T` and `S` are `1'b1` and `"DISABLED"`. Note that in the latter case the values of `T` and `S` are inconsistent.

It is possible to add synchronization code to the initial blocks to ensure the desired order of evaluation, though this can be complex if there are a lot of dependencies between static variables. Synchronization code cannot be added to variable declaration assignments, which are executed in implicit initial blocks for Verilog.

SystemVerilog initializes static variables before initial blocks are enabled. Thus values have been assigned to static variables when initial blocks are executed, so this removes some of the problems caused by the Verilog initialization rules. However, the order in which the assignments to the static variables are executed is arbitrary, so there is exactly the same issue with the order of execution when a static variable references the value of another static variable in a declaration assignment.

The code in Figure 8 shows `T` and `S` initialized by declaration assignments. As with the initial blocks, `S` is assigned `"ENABLED"` or `"DISABLED"`, depending on the order or evaluation of the assignments. There is no way to add synchronization code to the assignments, so the order cannot be controlled.

This issue is the *static initialization problem* that is well known to C and C++ programmers (ISOCPP, 2016). Ways to solve the problem in these languages are also well known, so it is instructive to look to them for guidance in how to solve the problem in SystemVerilog.

2.5 Solving the static initialization problem in C and C++

The initialization rules for static variables in C and C++ are as follows.

- Each static variable has an initial value. This is the initialization expression in the variable definition, or, if none is specified, the default value for the variable type.
- A static variable defined outside of a function (this includes a static class data member) exists for the lifetime of the program. Its value at the start of the program is arbitrary, and it is set to its initial value at some time before the execution of any function in the same compilation unit (the preprocessor output generated for a single source file). Functions is one compilation unit may be executed before static variables in a different compilation unit are initialized.

```
static bit              static string
  T = 1'b1 ;              S = ( T ? "ENABLED" : "DISABLED" ) ;
```

Figure 8: Static variables T and S, initialized by declaration assignments

```
// C++ function to replace int `static int I( 6 )'
int & I( )
{
  static int ii( 6 ) ;
  return ii ;
}
```

Figure 9: C++ function to replace a static variable

- A static variable defined in a function is created and set to its initial value when its definition is first reached during program execution, and exists until the end of the program. If the definition of a static variable in a function is never executed, the variable is not created.

The simple way to avoid accessing a static variable before it is initialized in C++ is to wrap the static variable in a function that returns a reference to it. The code in Figure 9 shows a C++ function `I()` that is used instead of static variable `I`. The function returns a reference to a local static variable `ii`. The first time the function is called, `ii` is created and initialized, and then a reference is returned. On subsequent calls, there is no initialization, and a reference to the already-existing variable `ii` is returned. To use the function instead of the variable, all references to `I` in the code are replaced with `I()`. The code behaves exactly as before, except that there is no possibility that the static variable is accessed before it is initialized.

C does not support references, but a pointer to the local static variable can be used instead.

For static objects, code such as that in Figure 10 is used. In this case, the static variable is a reference to a class object, which is initialized to reference an object created on the heap during the first call to the function.[3]

2.6 The static initialization problem in SystemVerilog

The C++ solution to the static initialization problem is a useful starting point for developing a solution for SystemVerilog. But there are differences between the languages which mean that the C++ solution cannot be used as is for SystemVerilog. The differences are as follows.

```
// C++ function to replace static object `static C J'
C & J( )
{
  static C & jj( * ( new C ) ) ;
  return jj ;
}
```

Figure 10: C++ function to replace a static object

[3] Creating the object on the heap with a reference to it in the function is preferred to creating a static object in the function because it avoids the related *static destruction problem* (ISOCPP, 2016)[ref]. This problem concerns to the order in which destructors are called on static objects, and is not an issue in SystemVerilog, because all objects are created on the heap, and there are no destructors.

```
// Function to replace static object `static C J = new( )'
function C J( )
  static C jj = new( ) ;   // UNSAFE
  return jj ;
endfunction
```

Figure 11: SystemVerilog function for a static object using unsafe initialization

- Each SystemVerilog static variable is set to the default value for its type when created.
- SystemVerilog initializes all static variables, including variables defined in subroutines, at the beginning of the program.
- SystemVerilog does not allow functions to return references or pointers to variables.

The first difference means that for SystemVerilog there is no problem with initialization of a static variable that does not have a declaration assignment. The variable is set to the default value for its type when it is created, and there is no further initialization.[4]

Because default value initialization is always safe, this should be used when possible. In some cases a variable with a non-default initial value can be replaced with one with a default value. For example, static variable `T` above in the examples above, which is initialized to `1'b1` can be replaced with static variable `not_T` that has no variable declaration assignment, so is initialized to `1'b0`. All references to `T` in the code should be replaced with references to `not_T`, and code updated for the inverted sense of the variable.

2.7 Safe initialization of static objects

The SystemVerilog code in Figure 11 is a direct translation of the C++ code for returning a static object. It returns the handle to an object that is stored in the local static variable `jj`. The initialization of `jj` is executed at the start of the program. If function `J` is used in a declaration assignment for a static variable, it may be called before `jj` is initialized, so the function returns the default value for

```
// Function to replace static object `static C J = new( )'
function C J( ) ;
  static C jj ;
  if ( jj == null )   // SAFE
  begin
    jj = new( ) ;
  end
  return jj ;
endfunction
```

Figure 12: SystemVerilog function for a static object using safe initialization

[4] In C++, initialization is a problem for all static variables defined outside of functions, including those without explicit initialization, because the value of a static variable is arbitrary for some period at the beginning of its lifetime.

jj, the null handle.

Function J is rewritten as shown in Figure 12 to avoid this issue. In this version, there is no declaration assignment for jj, so it is set to null (the default value) when it is created. When the function is first executed, jj is null, so an object is created, and its handle is returned. On subsequent calls, jj is not null, so the handle to the already-created object is returned.

The code in Figure 12 may not behave as expected if the simulator arbitrarily switches between processes during execution. Suppose that two processes call J when jj is null, and the first is preëmpted after it has determined that jj is null, but before an object has been created, then the second process executes the whole function, and finally the first process completes its execution of the function. In this case, jj is null when the second process executes the function, so it allocates an object, and the first process also allocates an object, so two objects are allocated, and two processes get different results from the call.

However, although arbitrary switching between processes is permitted by every Verilog and SystemVerilog standard, current simulators do not implement it, and instead implement switching between processes in a way that ensures that a sequence of statements without any statement that causes the process to wait is executed atomically with respect to other processes. There are no waiting statements in the body of function J, so this is executed atomically on current simulators.

To implement J safely on a simulator that does not guarantee atomic execution of its body requires the use of a semaphore or other synchronization mechanism to ensure atomic execution of the body of the function.

A further refinement of function J is shown in Figure 13. This version uses the implicit return variable for the function to store the handle, and so does not need any other variable. The function is declared with static variable lifetime so the implicit return variable is static. As in the previous version, an object is allocated the first time the function is called.

Because SystemVerilog permits hierarchical references to static variables in functions, the static variable in function J can be altered by external code. J.jj references the static variable in Figure 12, and J.J references the static implicit return variable in Figure 13, so assignments to these references alter the static variables. If this is a concern, the code in Figure 12 can be used with the body of the function, including the declaration of the static variable, enclosed in an unnamed begin-end block. This means that there is no hierarchical reference to the static variable, so it can only be changed by the code in the function.

```
// Function to replace static object `static C J = new( )'
function static C J( ) ;
  if ( J == null )
  begin
    J = new( ) ;
  end
endfunction
```

Figure 13: SystemVerilog function for a static object using a static implicit return variable

```
class Singleton ;

  static function static Singleton Instance( ) ;
    if ( Instance == null )
    begin
      Instance = new( ) ;
    end
  endfunction

  local function new( ) ;
    // Initialize object
  endfunction

  // Other methods and properties

endclass : Singleton
```

Figure 14: Implementing the instance of a singleton class

2.8 Singleton classes

A singleton class should be instantiated at most once. It is common to use a static variable to store the instance. Figure 14 shows a safe way to implement function `Singleton::Instance` to return the handle to the single instance of class `Singleton`. The function is a static method with static variable lifetime.

Function `Singleton::new` is declared local, so the only publicly-accessible way to create a `Singleton` object is to call method `Singleton::Instance`. This helps to ensure that the class truly is a singleton.[5]

2.9 Safe initialization of static variables of built-in types

A C or C++ function used to ensure safe initialization of a static variable of a built-in type returns a pointer or reference, neither of which are available in SystemVerilog. However, a handle to an object is effectively a pointer, so the static variable can be replaced by a handle (returned by a function, as in Section2.7) to a static object that contains a single property of the appropriate type.

The code in Figure 15 is used to replace variable `static int I` with a function `I()` that returns a static handle to an object of class `IntValue` , which has a non-static property `value`. To use the function instead of the static variable, all references to `I` are replaced by `I().value`.

Class `IntValue` can be generalized to a class for any built-in type, as shown in Figure 16. The class has a type parameter. Function `R` returns a static real variable, which is accessed as `R().value`. Class `Value #(T)` can be used to implement static values of any built-in type.

[5] Note: UVM component and object classes that use factory creation must declare `new` as public.

```
// Code to replace `static int I = 6'
class IntValue ;

  int value ;

  function new( int init ) ;
    value = init ;
  endfunction

endclass : IntValue

function static IntValue I( ) ;
  if ( I == null )
  begin
    I = new( 6 ) ;
  end
endfunction
```

Figure 15: Function to replace a static int variable

One reason to avoid static variables, apart from issues with initialization, is that any code that can reference the variable can change it arbitrarily. Thus it is difficult to understand how a static variable behaves without examining all code that has access to it (which is often the whole program). Class mechanisms can be used to limit the ways in which code can access and modify a static variable,

```
// General value class, use with any built-in type
class Value
 #( type T = int ) ;

  T value ;

  function new( T init ) ;
    value = init ;
  endfunction

endclass : Value

// Function to replace `static real R = 8.25'
function static Value #( real ) R( ) ;
  if ( R == null )
  begin
    R = new( 8.25 ) ;
  end
endfunction
```

Figure 16: General value class, and function to replace static real variable

```
class IncrDecrInt ;

  extern function new( int init ) ;
  extern function int value( ) ;
  extern function int incr( ) ;
  extern function int decr( ) ;

  local int value_ ;

endclass : IncrDecrInt

function static IncrDecrInt K( ) ;
  if ( K == null )
  begin
    K = new( 12 ) ;
  end
endfunction
```

Figure 17: Class `IncrDecrInt` and a static instance

which can avoid bugs due to errant code changing the value in an unexpected way.

As an example, the code in Figure 17 defines class `IncrDecrInt` with local variable `value_`, and public functions to increment, decrement and read the value of `value_` (function definitions are trivial, and are not shown). Function `K()` returns a handle to a static instance of the class with the value initialized to 12. In this case the value of the `int` variable is `K().value()`, and it can be changed only by calling `K().incr()` or `K().decr()`.

2.10 Conclusion: safe static variable initialization for SystemVerilog

The following guidelines ensure that SystemVerilog static variables are never accessed before they are initialized.

A. Never use a variable declaration assignment to initialize a static variable.
B. Avoid using an initial block to initialize a static variable.
C. When the default value is acceptable as an initial value, use a static variable.
D. For a static object handle variable with non-default initial value, put the static handle variable in a function that returns the variable each time it is called, and, on the first time it is called, sets the variable to a newly created and initialized object.
E. For a static non-class-type variable with non-default initial value, use a class that contains a non-static version of the variable and use a function to create and return a handle to a static instance of the class, as in case D.

3. Acknowledgments

Jonathan Bromley, and reviewers from SNUG and AMD provided valuable suggestions for improving the presentation.

4. References

Cannon, H. (1982). *Flavors: A non-hierarchical approach to object-oriented programming.* Symbolics, Inc.

IEEE. (1996). *IEEE Std 1364-1995: IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language.* NY, USA: The Institute of Electrical and Electronics Engineers, Inc.

IEEE. (2001). *IEEE Std 1364-2001 IEEE Standard Verilog Hardware Description Language.* NY, USA: The Institute of Electical and Electronics Engineers.

IEEE. (2005). *IEEE Std 1800-2005 IEEE Standard for SystemVerilog --- Unified Hardware Design, Specification and Verification Language.* NY, USA: The Institute of Electrical and Electronics Engineers.

IEEE. (2006). *IEEE Std 1364-2005 IEEE Standard Verilog Hardware Description Language.* NY, USA: The Institutue of Electrical and Electronics Engineers.

IEEE. (2009). *IEEE Std 1800-2009 IEEE Standard for SystemVerilog --- Unified Hardware Design, Specification and Verification Language.* NY, USA: The Institute of Electrical and Electronics Engineers.

IEEE. (2013). *IEEE Std 1800-1012 IEEE Standard for SystemVerilog --- Unified Hardware Design, Specification and Verification Language.* NY, USA: The Institute of Electical and Electronics Engineers.

ISOCPP. (2016). *What's the `static initialization order fiasco'?* Retrieved from Standard C++ FAQ: https://isocpp.org/wiki/faw/ctors#static-init-order-on-first-use

ISOCPP. (2016). *Why doesn't the Construct On First Use Idiom use a static object instead of a static pointer?* Retrieved from Standard C++ FAQ: https://isocpp.org/wiki/faq/ctors#construct-on-first-use-v2

Timi, T. (2014, 09 03). *Fake It 'til You Make It - Emulating Multiple Inheritance in SystemVerilog*. Retrieved from Verification Gentleman: http://blog.verificationgentleman.com/2014/09/emulating-multiple-inheritance-in-system-verilog.html