Using Parameterized Classes and Factories: The Yin and Yang of Object-Oriented Verification

David Rich
Adam Erickson
Mentor Graphics

SystemVerilog is the culmination of many design and verification engineers' experiences with various solutions to the problem of verification. Most features of the language have histories with one particular verification environment or another, each bringing some measurable benefit to SystemVerilog. Sometimes, the combination of certain features produces a benefit that is greater than the sum of the individual benefits. One example of such a combination is the joining of the class-based, object-oriented programming paradigm with the parameterized type system; the combination of which creates parameterized classes. Layered on top of this symbiosis is the *factory* pattern, a creational design pattern used in software programming. The type parameterization employs *static polymorphism*, while the factory employs *dynamic polymorphism*. This is the yin and yang of object-oriented programming.

This paper reviews the origins of the class-based and parameterized type features in SystemVerilog and discusses the synergies created by the combination of these features. We will compare these to generic programming practices in other object-oriented programming languages such as C++. We will consider the advantages and disadvantages of these practices. We will present issues specific to SystemVerilog, which arise when using parameterized classes in the context of the complete language; such as name search resolution, static variable/method declarations, and inheritance.

Finally, this paper will introduce the factory pattern, which has been used with parameterized classes as a proven technique for writing reusable verification class components, and it will provide examples along the way. We will provide guidelines for choosing when it is better to specify items statically with a type parameter versus dynamically with a factory configuration at runtime, and we will provide tips for structuring your class inheritance hierarchy using parameterized classes.

INTRODUCTION

The notion that seemingly opposing forces can beneficially transform each other has been around for millennia and is found in all facets of nature and human existence. Hardware Description Languages are no exception. HDLs are a delicate balance of declarative/procedural, combinational/sequential, and strongly/weakly typed behaviors.

Two concepts we shall examine are *static* and *dynamic* typed polymorphism. These terms alone conjure up opposing forces. SystemVerilog uses them in a variety of different contexts, but generally, static implies fixed, either in existence or in size, for an entire simulation, as opposed to dynamic, which implies something that can change during the simulation. *Polymorphism* is the ability to manipulate different data types with a uniform interface — reusability being the key benefit.

Generic programming is one application of static polymorphism, and the *factory* design pattern is one application of dynamic polymorphism. [1]

Origins of Generic Programming with Parameters

The concept of generic programming is based on the ability to write code for an algorithm in a particular programming language that is independent of certain parameters that will be specified at a later point. Many languages call these parameters *generics*, hence the term generic programming. Ada is widely credited with pioneering this style of programming, which is also a precursor of VHDL. [2]

The intent of generic programming is to facilitate reuse by writing a portion of code once and invoking or instantiating it many times with different sets of generic parameters. Verilog has had this since its inception, with the instantiation of modules using overridden module parameters. Verilog only allowed the parameterization of values that might control bit widths or array sizes, but SystemVerilog adds the full parameterization of all data types, including class declarations. [3]

Origins of Programming with Factory Design Patterns

Design patterns, such as the factory, are proven techniques for structuring code that is highly reusable and adaptable to change. The seminal book, *Design Patterns* [4], codified a core set of patterns that were universally present in applications that have withstood the test of time. These patterns have become the hallmark of good object-oriented design. *All* design patterns, including the factory, take advantage of dynamic polymorphism, a fundamental feature of object-oriented programming (OOP) languages such as SystemVerilog.

The intent of the factory pattern is to decouple classes from the concrete objects they instantiate by delegating their creation to a separate factory object. A concrete object requests from the factory a new instance of a given type. The factory may return an object of the requested type, or it may be configured to return an object that is a subtype of the requested type. The requesting object does not care as long as the returned object is “at least” the type it requested. The requesting object is thus reusable for the objects it requests as well as any their subtypes. Furthermore, dynamic polymorphism allows reconfiguration of the factory so that it can return different subtypes throughout simulation.

PARAMETERS — THE YIN

Parameters, Constants, and const

What exactly are parameters, and how do they relate to constants? Why is a `const` variable not the same as a constant? These questions seem to be an immense source of confusion, even for experienced SystemVerilog users. To explain this it is helpful to understand the phases involved with accomplishing a simulation.

Most simulators go through three distinct phases: *compilation, elaboration, and execution*. These phases could be accomplished in separate command steps or automatically combined in a single step.

Compilation converts a set of human readable files into a form suitable for processing in another phase. This phase also processes included files, conditional text, and text macros into a single stream of statements called a compilation unit. The compiler syntactically recognizes each statement according to the BNF rules, but generates no useful executable code in this phase.

Elaboration globally resolves all instantiations, propagating parameter values and types, and connecting ports. It allocates the space for all static design units and objects. Elaboration resolves all hierarchical references, so that it can identify the data types involved in expressions, and it then generates code to evaluate each expression.

Execution, or runtime, initializes all static objects and spawns all processes where procedural code exists. The simulation kernel controls the execution of all processes.

Now that we have defined these phases, let us look at the example below and see how the simulator interprets the various constant concepts.

```
`define THREE '{1,2,3}
module top;
parameter FOUR = 2*2;
typedef struct {int A,B,C;} ABC_t;
int V;
middle #(ABC_t,FOUR) m1(`THREE, V);
initial #5 V = 10;
endmodule
```

A constant is any expression of literals, parameters, and functions involving other constants. The literal numbers 1, 2, 3, 5 and 10 are all constants. The macro `` `THREE `` is replaced with text before any syntax recognition. Its text also represents a constant for an aggregate data type. The `2*2` is a constant expression of two literal constants. The parameter `FOUR` is constant. Constant expressions do not contain any variables. Elaboration determines the final value of a constant expression.

The compilation phase determines that `middle` will be an instantiation of a design unit with two parameter overrides and `'{1,2,3},V` connected to its ports. The design unit could be a module, program block, or interface that has not been defined yet. This compilation phase does not perform the semantic checks necessary to see if it is legal to override those parameters or if there are two ports to connect.

Later, the compilation phase reads the definition of middle:

```
module middle #(type T, int N=8)
       (input T Port1, Port2=5);
  T Fix[ N] ;
  typedef bit [ 0:N*8] vector_t;
  vector_t Dyn[ ] ;

initial begin
#25 // move away from time 0
    run();
    end
task automatic run();
  const int C = Port2;
  Fix = `{ default:Port1} ;
  Dyn = new[ C] ;
endtask
endmodule
```

The compilation phase recognizes that T will be a data type, but each instance could result in T having a different type. The size for the fixed array Fix and the width of vector_t must be declared using constant expressions. The size of the dynamic array Dyn can be any expression because it will be constructed at runtime.

During the elaboration phase, an instance of middle, named m1, is created with type T overridden by data type ABC_t and N overridden by a constant expression computed with value 4. Port1 is connected to the constant expression '{1,2,3}, and Port2 is connected to the variable V. The elaboration phase allocates the static, fixed-size array variable top.m1.Fix with four elements. It computes the data type vector_t to be a 32-bit integral type and creates the static, dynamically sized array variable top.m1.Dyn with no elements.

Finally, the runtime phase starts by propagating the constant '{1,2,3} to top.m1.Port1 at time 0. At time 5, top.V is assigned to 10 and propagates to top.m1.Port2. Task run is called at time 25. The call creates the local const variable C. The only difference between a const variable and a normal variable is that the only allowed assignment to the const variable is at the point of its declaration.

In this case, task run has an automatic lifetime, which means that each call to the task allocates its local variables. The initialization of C occurs on each call. Therefore, a const variable is not really a constant as far as elaboration or parameterized types are concerned.

Classes and Paramters with the SystemVerilog Type System

Objects and Handles

A SystemVerilog class declaration is a data type that defines an instance of a class object. Just as in the module previously defined, a class may contain other data types that are defined during elaboration. However, construction of class objects occurs at runtime using only the special function new(). A class variable can hold a handle to a constructed object. For example:

```
module example #(int W = 8)();
  class C;
       bit [ 0:W-1] p;
  endclass

  C h1;
  initial config();
  task automatic config();
     C h2;
     h1 = new();
     h2 = new();
  endtask
endmodule
```

This example contains a class definition `C` in the module `example`. If there are no other modules, example will become the single top-level instance with one class definition. The module contains a static class variable `h1`. In the runtime phase, the call to task `config` constructs an automatic class variable h2, as well as constructing two new class C objects. If there are multiple instances of a module example, each instance creates a new class definition.

```
module test;
  example #(16) e1();
  example #(32) e2();
endmodule
```

Now there are two definitions of class `C`, along with two sets of static variables: `test.e1.h1` and `test.e2.h1`. SystemVerilog defines the two classes to be assignment incompatible simply because they are declared in different scopes, regardless of whether or not they contain identical declarations. In this case, the class property `p` in `test.e1.C` is 16 bits wide and is 32 bits wide in `test.e2.C`.

Parameterized Classes

Taking the concepts of Verilog parameterization a step further, SystemVerilog allows classes to have their own set of parameters that may be overridden when declaring a class variable. A class declared with parameters is called a *generic* class and is not a true data type until a class variable with specific parameter overrides is declared, which is called a *specialized* class.

```
class PC #(type T, int N);
  int i;
 T sub;
  bit [ 0:N-1] m_int;
endclass
PC #(C1, 8) h1;
PC #(C2, 16) h2;
PC #(C1, 16/2) h3;
```

At this point, a generic class with specific parameters becomes a concrete specialized class. The class variables `h1` and `h2` are handles to two different specializations of the generic class `PC`. The class variable `h3` matches the class specialization of `h1` and is, therefore, a matching type.

Using Containment Hierarchy

Containment is the HAS-A relationship of OOP, where one class definition instantiates other class variables. A class definition can use its parameters to parameterize the contained class member variables.

A verification environment will typically group class components into a containment hierarchy as follows.

```
class driver #(type REQ,RSP) extends component;
  …
endclass
class monitor #(type P) extends component;
…
endclass
class agent #(type REQ,RSP) extends component;

  driver #(REQ,RSP) my_driver = new();
  monitor #(REQ)    req_monitor = new();
  monitor #(RSP)    rsp_monitor =new();
…
endclass

class test extends component;
  agent #(C1,C2) agent12 = new();
  agent #(C1,C3) agent13 = new();
endclass
```

Assuming C1, C2, and C3 are unique class definitions, the declaration of test creates two specializations of the agent and driver classes and three specializations of the monitor class.

Using Inheritance Hierarchy

Inheritance is the IS-A relationship of OOP, where an extended class inherits all of the properties and methods from its base so that they all can be referenced directly from the extended class. Parameters applied to the extended class may propagate up to the base class. Using the same driver class from the example above:

```
class my_driver #(type RQST)
               extends driver #(RQST,RQST);
…
endclass
class test;
  my_driver #(C1) d1;
  my_driver #(C2) d2;
endclass
```

Each specialization of my_driver creates a corresponding specialization of driver.

Dealing with Static Class Properties

In a non-parameterized class, declaring a static class property creates one variable for each declaration of a class that is shared with all constructed objects of that class. A static class property in a generic class is not created until the class becomes a concrete specialization. A simple `typedef` can create a specialized class.

```
class component #(type T=int);
  string m_name;
  static component #(T) registry[$];
  function new(string name);
    m_name = name;
    registry.push_back(this);
  endfunction
endclass

typedef component #(C1) C1_component_t;
component #(C2) C2_component;
component #(C3) C3_component;
```

The above example creates three static variables, all queues:

```
component_t#(C1)::registry
component_t#(C2)::registry
component_t#C3)::registry.
```

There is no `component#(int)::registry` because no one has defined a type or instantiated a class variable using the default parameter values.

In order to create a single, static variable common to all specializations of a single generic class, you must move the static variable declaration into a common, unparameterized base class:

```
class component_base;
static component_base registry[$];
  function new();
    registry.push_back(this);
  endfunction
endclass
class component #(type T) extends
component_base;
  string m_name;
  function new(string name);
    super.new();
    m_name = name;
  endfunction
endclass
```

Type Compatibility Issues

A common mistake in casting parameterized classes is that class inheritance of a parameter does not imply inheritance of the class it is specializing.

```
class Base;
endclass
class SubType extends Base;
endclass
class PC #(type T);
endclass

PC #(Base) P;
PC #(SubType) C = new;
P = C; // illegal
Base p;
SubType c = new;
p = c; // legal
```

Each unique set of parameters applied to a parameterized class creates a unique class type. Because `Base` and `SubType` are different classes, the two specializations of PC are not assignment compatible.

FACTORIES — THE YANG

The Non-Virtual Constructor Problem

A typical task in transaction-level verification is to define a transaction class and a set of components that operate on that type, such as a stimulus generator and driver.

Here is a naïve attempt at creating a driver for a particular transaction type.

```
class ApbTrans; ... endclass
class ApbDriver;
  task run();
    forever begin
      ApbTrans trans = new; ⇓ no reuse
      trans.randomize();
      ... execute transaction
    end
  endtask
endclass
```

The problem lies with our direct use of the `ApbTrans` class's `new` method, commonly called its *constructor*. The constructor is specific to the class in which it is defined, so any class that directly calls another's constructor is hard-coded to that specific class. We cannot employ dynamic polymorphism, because we cannot substitute the type created with a subtype without modifying the creating class. We are stuck with `ApbTrans`.

To inject a different type of stimulus, typically a subtype of `ApbTrans`, we would have to write a new driver class. As bad as this sounds, the problem runs deeper: the driver can sit at the bottom of a deep hierarchical composition of objects in the testbench. Changing the class type requires changing the class type of its containing parent, and so on all the way to the top-level container. When we want to make another similar change, we must fork yet another branch of class definitions. In such environments, seemingly small changes require large-scale modifications and class explosion.

Ideally, we want to design our testbench hierarchy once, yet during construction we want the ability to modify the types and number of objects that make up that hierarchy. That is what the factory pattern does for us.

The Basic Factory Pattern in SystemVerilog

The factory pattern allows polymorphic substitution by delegating object creation to a separate factory object. Rather than call `new` directly, the instantiating class calls a method, for example `create`, of the factory object to get a new instance. By decoupling instantiating classes from the actual concrete types they instantiate, we are able to create deep compositional hierarchies of objects, each of which are polymorphically substitutable with any of their subtypes. We can substitute the transaction types a driver operates on, and we can substitute the driver itself, as long as these objects delegate instantiation to a factory.

Employing the factory pattern involves the following.

- Define an abstract virtual class that defines a method for creating objects, for example:

```
virtual class Object;
  pure virtual function Object create();
endclass
```

- Define a factory class that can produce any subtype of `Object`:

```
class Factory;
  Object obj;
  virtual function Object create();
    // delegate creation to subtype itself
    create = obj.create();
  endfunction
endclass

Factory factory = new;
```

A typical factory implementation employs the Singleton pattern [4], which ensures only one instance is present in the testbench. We will employ this pattern in a later example, but for now, we simply declare a global instance of our factory.

- Define concrete subtypes to `Object` that implement `create`:

```
class ApbTrans extends Object;
  typedef enum {WRITE,READ} cmd_e;
  rand int unsigned addr;
  rand int unsigned data;
  rand cmd_e cmd;
  virtual function Object create();
    ApbTrans trans = new();
    return trans;
  endfunction
endclass
```

```
class ApbWriteTrans extends ApbTrans;
  constraint write_only { cmd == WRITE; }
  virtual function Object create();
    ApbTrans trans = new();
    return trans;
  endfunction
endclass

class ApbReadTrans extends ApbTrans;
  constraint read_only { cmd == READ; }
  virtual function Object create();
    ApbTrans trans = new();
    return trans;
  endfunction
endclass
```

You can define any number of subtypes you want. The important thing is that they all extend from `Object`.

With the above in place, we can now begin to design our classes to use the factory instead of calling `new`.

```
class ApbDriver;
  task run();
    forever begin
      ApbTrans trans;
      $cast(trans,factory.create());
      trans.randomize();
      ... execute transaction
    end
  endtask
endclass
```

The factory returns the subtype instance in a handle to the super class, `Object`, so `ApbDriver` must cast it back into the particular subtype it expects. Before we can use the factory, it must be initialized *externally* with a particular instance of the subtype we want it to produce. This is because the factory class is designed to produce objects of type `Object`, which is an abstract virtual type that cannot be allocated directly. We can only allocate concrete *subtypes* of `Object`. The following code configures the factory to produce different subtypes of `Object` over time. The driver, running concurrently, will receive objects of whatever type the factory is configured with at the time it calls `factory.create()`.

```
initial begin
  // create driver
  ApbDriver driver = new;

  // initialize factory to produce ApbTrans
  ApbTrans trans = new;        // reads/writes
  ApbWriteTrans w_trans = new; // writes only
  ApbReadTrans  r_trans = new; // reads only
```

```
  // start the driver
  fork driver.run(); join_none

  // plug in different objects to produce
  // over time. Driver unaware, and unmodified
  factory.obj = trans;
  #1000;
  factory.obj = w_trans;
  #1000;
  factory.obj = r_trans;
  #1000;
  $finish;
end
```

The factory's `obj` property is assigned a particular concrete subtype. The calling class (`ApbDriver`) calls the factory's `create` method, which delegates creation to this subtype. The new subtype object is then returned back to the caller as a handle of type `Object`. The driver then casts this handle back to the concrete subtype that it expects.

The factory pattern solves our original problem by delegating the task of creation to a factory and, ultimately to the subtypes themselves. This does not come without costs:

1) The factory needs an instance of the type it produces as it calls the non-static, virtual `create` method. This can be an expensive overhead, especially if a general factory mechanism is employed to produce any type in the system.

2) The created objects require a common class that implements a virtual method for construction. This constructor has a fixed prototype, which fixes the number and type of arguments available to construct an object.

3) The caller to the factory must cast its return value to get the concrete type. That is a red flag, as it opens the possibility that the type returned is not compatible with the type expected. The factory could have been initialized with `PciTrans`, which may extend `Object` to meet the factory requirements yet not meet the `ApbDriver's` requirement for an `ApbTrans` or subtype. In such cases, you get a highly undesirable, fatal run-time error. If there is both a `PciDriver` and `ApbDriver` in the system, how can they possibly use the same factory instance when a factory can only be configured with one particular subtype?

The first problem is solved using the *proxy* pattern. The second and third problems are solved using a parameterized factory — the Yin of object-oriented verification.

Lightweight Object Proxies

A proxy is a lightweight substitute for the object it represents. With the proxy, the factory's requirement for an instance of an object can be satisfied while deferring the costs of creating the actual object until it is requested.

```
class ObjProxy #(type OBJ=int) extends Object;
  virtual function Object create();
```

```
    OBJ subtype = new;
    return subtype;
  endfunction
endclass
```

The proxy's `create` method does not create itself — it creates the object it is a proxy to directly, using `new`. This removes the requirement that the created object itself implements the `create` method. Now when we configure the factory, we plug in an instance of the proxy to a subtype:

```
ObjProxy #(ApbWriteTrans) w_proxy = new;
factory.obj = w_proxy;  // writes only
```

COMBINING THE YIN WITH THE YANG

Dynamic casting with `$cast` opens the possibility for fatal run-time errors. That is bad programming practice; thankfully, it is avoidable.

So how do we design a single factory class that can produce any object deriving from `Object`, yet avoid `$cast` and possible run-time errors for components that want only `ApbTrans` and its subtypes?

We solve the problem by making the factory a parameterized class, where the parameter specifies the subtype family.

```
class Factory #(type BASE=int);
  Object obj;
  virtual function Object create_obj();
    create_obj = obj.create();
    create_obj.randomize();
  endfunction

  // Make sure only 1 instance per sim
  local static Factory #(BASE) factory;
  protected function new(); endfunction
  static function Factory #(BASE) get();
    if (factory == null)
      factory = new;
    return factory;
  endfunction

  // Configure which subtype to produce
  static function void set_object(Object obj);
    Factory #(BASE) f = get();
    f.obj = obj;
  endfunction

  static function BASE create();
    Factory #(BASE) f = get();
    $cast(create,f.create_obj());
  endfunction
endclass
```

In the factory definition above, we have applied the Singleton pattern, which has any factory specialization spring into existence upon first use. Subsequent uses simply return the same instance. This makes sure we have only a single instance for any given factory specialization.

We have also provided a static `create` method that gets the Singleton instance of itself and calls the internal `create` method. It performs the `$cast` for us, but because the factory and its `obj` property are both parametrically typed using the same `BASE` type, the `$cast` is guaranteed not to fail.

When we write our drivers now, we have them make requests to the factory specialization that is matched to the base types we want created. We avoid the `$cast`, and all type errors are caught during compilation.

```
class ApbDrv #(type BASE=ApbTrans);
  task run();
    forever begin
      BASE trans;
      trans = Factory #(BASE)::create();
      trans.randomize();
      ... execute transaction
    end
  endtask
endclass
```

Our original factory implementation used a global, non-parameterized factory that would presumably be used by all components making requests for different types. What if we had a `PciDriver` and an `ApbDriver` making requests at random times? How would you coordinate the factory's configuration to produce the correct subtype when you cannot know the caller at any given time?

The parameterized factory allows for a separate specialized factory for each type being requested, and we avoid having to create instances of each specialized factory ourselves. A call to the specialized factory's static `set_object` or `create` methods cues its allocation on demand.

Here's our new top level, which is highly reconfigurable and reusable. We have added a `PciDrv` component, defined in the same fashion as `ApbDrv`.

```
// Create drivers that operate on base types
ApbDrv #(ApbTrans) apb_drv = new;
PciDrv #(PciTrans) pci_drv = new;

// Create proxies to particular subtypes
ObjProxy #(ApbWriteTrans) apb_wr_trans = new;
ObjProxy #(PciShortBurst) pci_sh_burst = new;

// Configure factories with subtype proxies
Factory #(ApbTrans)::set_object(apb_wr_trans);
Factory #(PciTrans)::set_object(pci_sh_burst);

initial begin
  // start the drivers; each to produce their
```

```
  // respective subtypes
  fork
    apb_drv.run();
    pci_drv.run();
  join_none
end
```

Because we have not accommodated more than one subtype override for any given base type, even the parameterized factory as written here is limited to system-wide type overrides. A truly generic factory would accommodate storage of any number of subtype overrides that are accessed by some unique key, such as hierarchical position of the component making the request, so that a particular instance of `ApbDriver` could be configured to produce different subtypes than other `ApbDriver` instances elsewhere in the environment. The Open Verification Methodology's (OVM) factory and proxy registration classes [5] employ this approach.

CONCLUSION

We have explored the static and dynamic aspects of type polymorphism in SystemVerilog as well as explained the synergies in combining the two within a single environment. All of this has led to improved code reusability by reducing the code required for each specialization.

REFERENCES

[1] Steve McConnell, Code Complete: A Practical Handbook of Software Construction 2nd Edition. Microsoft Press, Redmond, Wa. June 2004.

[2] "Generic programming." Wikipedia, The Free Encyclopedia. 9 Jan 2009, <http://en.wikipedia.org/w/index.php?title=Generic_programming&oldid=262940482>.

[3] "IEEE Standard for SystemVerilog- Unified Hardware Design, Specification, and Verification Language," IEEE Std 1800-2005, 2005.

[4] E. Gamma, R.Helm, R. Johnson, J. Vlissides, Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley Publishing Company, Reading Massachusetts, 1995.

[5] "Open Verification Methodology User Guide - Version 2.0.1, October 2008", Mentor Graphics, Corp and Cadence Design Systems, Inc.

This paper appeared in *Proceedings of DVCon 2009*, San Jose, CA, pp. 158–165.

Corporate Headquarters
Mentor Graphics Corporation
8005 S.W. Boeckman Road
Wilsonville, Oregon 97070USA
Phone: 503-685-7000
North American Support Center
Phone: 800-547-4303

Silicon Valley
Mentor Graphics Corporation
1001 Ridder Park Drive
San Jose, California 95131 USA
Phone: 408-436-1500
Sales and Product Information
Phone: 800-547-3000

Europe
Mentor Graphics
Deutschland GmbH
Arnulfstrasse 201
80634 Munich
Germany
Phone: +49.89.57096.0

Pacific Rim
Mentor Graphics Taiwan
Room 1001, 10F
International Trade Building
No. 333, Section 1, Keelung Road
Taipei, Taiwan, ROC
Phone: 886-2-27576020

Japan
Mentor Graphics Japan Co., Ltd.
Gotenyama Garden
7-35, Kita-Shinagawa 4-chome
Shinagawa-Ku, Tokyo 140-0001
Japan
Phone: 81-3-5488-3033

MGC 4-09 TECH8190-w