Avoiding the Pitfalls of Polymorphism
Or
How to build an Extendable Verification Environment

Nancy Pratt
IBM Burlington, VT
npratt@us.ibm.com

Jesse Craig
IBM Burlington, VT
jecraig@us.ibm.com

Divya Jayasree
IBM Bangalore, India
divyavjayasree@in.ibm.com

Santoshkumar Jinagar
IBM Bangalore, India
sjinagar@in.ibm.com

ABSTRACT

It's hard to plan ahead when you don't know the details of what's coming. Without the right start, you'll get into trouble when it comes time to employ tricks with polymorphism. This paper will discuss the lessons learned while verifying multiple BIST engines, in unit, cluster, and system environments, while including a variety of configurable memory types. By understanding potential pitfalls, you'll be more likely to avoid the need for code duplication or the re-work of base classes, and you'll be able to more efficiently extend your environment in many directions.

Table of Contents

Table of Figures

Table of Lessons

Table of Examples

1.0 Introduction

Verification challenges vary depending on design characteristics. Large, complex designs with a long development schedule and a minimal need for reuse of the verification environment from unit to system level can be effectively verified without employing object-oriented techniques. But, when the plan calls for numerous related designs to be developed, each with multiple configurations, and they need to be verified alone and together in a system environment, an object-oriented approach offers many benefits. However, there are many potential pitfalls that may be encountered, especially for those new to this type of programming. There also seems to be an inverse relationship between the schedule duration and the impact of verification environment development problems; a short schedule doesn't allow for much of a learning curve. Therefore, it becomes imperative to learn from the experience of others. This paper will focus on our experience creating an extendable set of Vera/RVM environments which maximize reuse through the application of object-oriented programming concepts.

2.0 Object-Oriented Programming Concepts

Object-Oriented Programming (OOP) is debatably one of the least intuitive, and most powerful, of the programming paradigms. A simple program can be written in an imperative paradigm language, such as C, using only one source file containing a few lines of code. That same program written in the OOP paradigm may require multiple classes to be written, resulting in several files worth of code. In exchange for this extra effort, we are left with source code that is easier to maintain, extend, and reuse in future projects. The only caveat is that this grail of improved maintainability and reuse can only be achieved if the OOP paradigm is used correctly.

At the core of the OOP paradigm is the notion of an object and its class. An object encapsulates data with the functionality that acts on that data. For instance, a String object would contain not only a string of characters, but also methods, like concatenate, that operate on that string of characters. A class is the programmatic description of the data and methods that compose an object. Each time an object is created from its class description, it is said to instantiate that class. The object is also said to have the type of its class. A String object is instantiated from a String class and is of the type String.

An example verification environment designed using OOP might contain a FIFO Queue class that stores Packet objects. The Packet class might contain an address and data fields along with simple methods for setting and accessing those fields. The FIFO Queue class might contain simple methods for adding a packet, detecting if packets are in the FIFO, and for getting the next packet from the FIFO. Internally, the FIFO may store the packets in an array or a linked list; this choice is completely hidden from users of the FIFO Queue class. The encapsulation inherent in OOP partitions the design into logical and maintainable chunks and protects code that is dependant on a class from internal changes to that class. As long as the FIFO Queue class provides the same methods to operate on it, a developer can change the internal implementation of the FIFO without affecting code that depends on that FIFO.

2.1 *Inheritance*

A key feature of the Object-Oriented Programming paradigm is that classes can extend the functionality of other classes. A class that extends another class can inherit all of that classes

functionality. This allows a developer to reuse portions of a class, extending or replacing only those portions that need to be changed to fit some new role. A class is extended by adding more methods and/or properties, and functionality is replaced by overloading or overriding previously defined methods with new, more specialized, implementations of those methods. A class that extends another class is said to be a child class of the original, parent, class. In Figure 1, an example class hierarchy is shown. A base Parent class is extended into two child classes, ChildA and ChildB. ChildA is further extended by the Grandchild class.
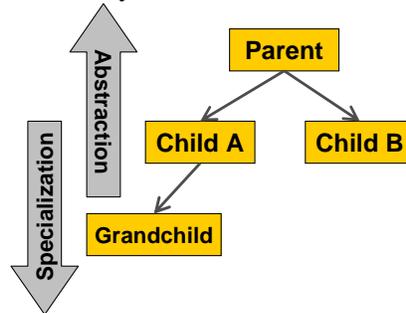
Figure 1: Example Class Hierarchy

Because each new child class extends its parent to solve some new problem, the child classes are always more specialized versions of their parent class. The two classes, ChildA and ChildB, are both more specialized versions of the Parent class, each designed to address some specific problem. The functionality that is universal to both problems is stored in the Parent class and reused by each of the children. In Figure 2, two Venn diagrams are used to show where the functionality is found in a class hierarchy.
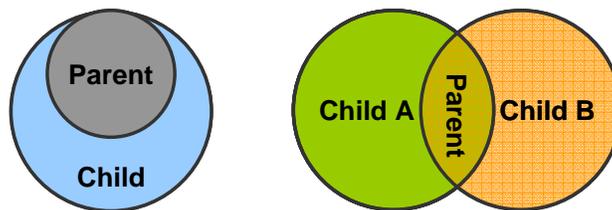
Figure 2: Venn Diagrams of the Function Provided by Related Classes

The first diagram shows the relationship between a parent and its child class. The child class inherits the functionality of the parent class, adding its own functionality where needed. In the diagram, this is shown by the parent class being completely enveloped by the child class. In Figure 2, where two classes extend the Parent class, ChildA and ChildB, both children contain the functionality of the parent, but the additional functionality they offer is not shared. In general, sibling classes should not share any functionality between themselves, since this functionality can be moved into the Parent class. Conversely, any functionality of the Parent class that is used by only one of the child classes should be moved from the parent to that child class, or to a new class that resides between ChildA and Parent in the hierarchy.

An alternative nomenclature for the parent/child relationship is the reference to super/sub classes. A child class refers to its parent class as its *super* class. Conversely, more specialized (i.e. child) classes are considered subclasses. In this paper, the term *base class* is loosely used to

mean a class from which another class is derived, not necessarily the most "super" class which isn't extended from anything. While a "super class" refers to only one level back, we use "base class" to be a node in the tree that is the origin of branches within our project. Many classes are ultimately derived from an RVM base class, but we are concerned with the base level where we begin our specialization.

2.1.1   Multiple Inheritance

Multiple inheritance is a specialized form of inheritance provided by some Object-Oriented Programming languages. It allows a child class to inherit the functionality of two or more parent classes. An example of how this works is given in Figure 3.
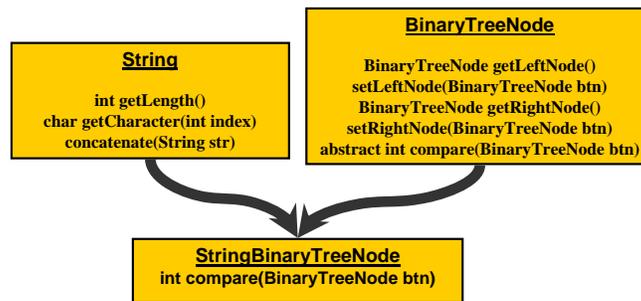
Figure 3: Example of Multiple Inheritance

This example takes a String class and combines it with a BinaryTreeNode class to create a new class that provides the functionality of a string and an object that can be stored in a binary tree. In this way, any data class can be extended to fit inside binary trees. The new child class only needs to override the abstract compare method to create a new binary tree data structure.

This code reuse and extensibility comes at a high cost. Multiple inheritance is widely believed to decrease the maintainability of software architectures. In an environment where multiple inheritance is used, changes to a class must be rectified against dependencies in not only that class' descendant classes, but also against any other parent classes those descendant classes may have. If a new method, int getLength(), was added to the BinaryTreeClass it would have the same method signature as a method in the String class. The StringBinaryTreeNode would then have to be changed to select which of the two methods it would inherit. If the system needed it to inherit both methods, this would become more difficult, if not impossible, to resolve. This is a specialized form of the more general 'diamond problem' that remains a source of debate in computer science.

The Vera programming language does not support multiple inheritance. This was most likely because of the previously discussed drawbacks. In general, the need for multiple inheritance can be avoided by using good software architecture techniques. The most common alternative is to create a member object to provide the functionality that would have come from one of the parents. In the example above, the BinaryTreeNode could be extended to have a String object as a member. Applications that had previously used the StringBinaryTreeNode class would use this new class and access the String functionality by grabbing the member String object and working with that object.

2.2 *Overriding and Overloading*

A child class has two ways of extending its parent's functionality: adding or replacing. The child can add functionality in the form of new methods and properties, or it can replace functionality by creating a new implementation of a method the parent class already provides. Adding functionality is done by creating a new method or property in the child class. Replacing functionality is done using techniques called *overriding* or *overloading*. The terms "overriding" and "overloading" are often interchanged, but there is a subtle difference. When a child *overrides* a parent's virtual method, any calls to that method will execute the child's implementation. This process is transparent to the caller, to the point where the caller can not access the parent's version of the method (i.e. no super.super). To *override* a method, the child class must implement a method that has the same method signature as the parent's method. A method signature is defined as a tuple containing the method's name, return type, and parameter types. The name of the parameters and the implementation details of the method are not part of the signature and need not match between the parent and the child. To *overload* a method, the child class must implement a method that has the same name as the parent's non-virtual method, but the rest of the signature will be different, allowing different number or type arguments and a different return value.

In Vera, the constructor method (i.e. new()) can't be declared as virtual, nor can it be overloaded or overridden. Instead, the constructor methods are automatically "chained" with the most abstract base class constructor being executed first. If the constructor signatures don't match across the extended hierarchy, then the child must call super.new() with the appropriate arguments.

2.3 *Polymorphism*

In its general sense, polymorphism refers to an entity being capable of assuming many forms. In the OOP paradigm, this translates to an object appearing to have many different types. While the constructor method creates a single object in memory, the handle (i.e. the pointer) to that object can be of the object's type or the type of any of its ancestors. Polymorphism permits a handle for the instance of the Grandchild class, shown in Figure 1, to be of the type Grandchild and also of the types ChildA and Parent. By declaring a method as virtual, an extended class can override a base class method and the compiler will determine the correct version of the method to use (i.e. the one from the most "extended" class level). That method can operate on any properties at its own level or on properties in more super levels, since the parent's properties are inherited. That method cannot access any new functionality given classes extended from its level. For example, a virtual method in Parent can be overridden by ChildA. Grandchild can invoke the method, but the method from ChildA will be used and it can only access properties in Parent and ChildA.

The advantage of polymorphism is that an abstract class can be extended and customized as needed, but each derived (child) class can still be pointed to by a common base class. In essence, polymorphism permits classes to be extended without forcing the classes that operate on them to be rewritten.

2.4 *Downcasting*

An unfortunate side-effect of polymorphism and inheritance is the need for downcasting. Downcasting occurs when an instance of a base class is cast into the type of a derived class. This casting asserts that an object is of the derived type and allows a method to access it as that derived type. Assuming the object is actually of that type everything executes successfully, if not a program fault will eventually occur.

```
                        ┌──────────┐        ┌──────────┐
                        │  FIFO    │        │  Packet  │
┌────────────────┐      └────┬─────┘        └────┬─────┘
│ add(Packet p)  │◄·····     │                   │
└────────────────┘          ▼                    ▼
                       ┌──────────────┐    ┌──────────────┐
                       │ ExtendedFIFO │    │ExtendedPacket│
                       └──────┬───────┘    └──────────────┘
┌─────────────────────────────────────────────┐
│ add(Packet p){                              │
│    ExtendedPacket ep = (ExtendedPacket) p;  │
│    ...                                       │
│ }                                            │
└─────────────────────────────────────────────┘
```
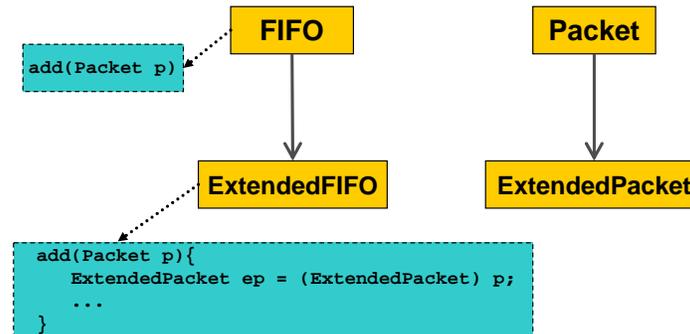
Figure 4: Downcasting Example

Figure 4 shows the original example given in section 2.0 with both the FIFO class and Packet class extended. Unlike the example in section 2.2, the new ExtendedFIFO class is designed to operate on only the new ExtendedPacket class, taking advantage of some new methods found in the ExtendedPacket class. The add() method for the new ExtendedFIFO class overloads the add() method for the parent class, taking a standard Packet as its argument. When an instance of the ExtendedPacket class is added to an instance of the ExtendedFIFO class, the add() method is forced to treat the ExtendedPacket as a base Packet. This causes problems when the ExtendedFIFO needs to use one of the new methods found only in the ExtendedPacket class. A new add() method could be given to the ExtendedFIFO class that takes an ExtendedPacket as its argument, but this would bloat the class and defeat the advantages of OOP. Instead, the add() method of the ExtendedFIFO class can downcast the packet into the ExtendedPacket type, taking advantage of Vera's cast_assign function, and operate on it in that form. While downcasting does require extra verification effort on the part of the developer to ensure that no illegal downcasts are made, it avoids the larger problem of class bloat.

2.5 *Aspect Oriented*

Aspect-Oriented Programming (AOP) is a paradigm that is often combined with Object-Oriented Programming. While OOP deals with the encapsulation of discrete functions into controlled objects, Aspect-Oriented Programming focuses on functionality that is shared between multiple, seemingly unrelated, objects. These functions are often global functions that relate to almost every other function of the system. Examples include the collection and filtering of log messages and the reset of the design-under-test. In each case, the function cross-cuts the hierarchy, being found in many classes that share no common parent classes, and is therefore difficult to handle using OOP. When reset occurs in a verification environment, it triggers a state change in every scoreboard, transactor, and monitor, even though few of those classes are directly related to the generation or verification of reset. While the Vera language does not support the AOP paradigm directly, the RVM does provide functionality similar to an AOP language using callbacks, notifies, and channels. In the RVM, all three of these mechanisms can be used to trigger a cross-

cutting action across many objects. Resetting the design-under-test may trigger a callback to every transactor to be reset, regardless of the specific function of those transactors.

2.6 *Design Patterns*

A great deal of research has been done on ways to simplify the architectural design of OOP-based software applications. One of the most popular advances in this field is the use of Design Patterns. These patterns are ways of partitioning the functions needed by an application into discrete objects and deciding how those objects should interact. Design Patterns were made popular by the book by the same name (see the reference section on page 35), and this book continues to be an excellent reference for the technique. The RVM uses several of these design patterns, a few of which are highlighted in the sections below.

2.6.1 Factory Pattern

The Factory Pattern used by Vera is a specialized form of the Prototype pattern. This pattern defines a way of creating new objects dynamically in an application. It is often the case that a method needs to create new objects. Normally this is done by calling the new() function in Vera. The problem arises when you want to change what type of object that method creates at runtime. The new() function is designed to create a new object of whatever class type is provided. The class type is hard-coded into the application. To allow this type to change dynamically at runtime, the pattern adds a clone() or copy() method to all the objects that the method may wish to create. The method is then passed a prototype object which it clones each time a new object is needed. When the type of the new object needs to change, the system gives the method a new prototype object. As long as all the prototype objects extend some common parent class, polymorphism allows the method to clone them into new objects without any change to the source code. (See Section 10.0 References on page35.)

This pattern is used by the RVM in the generators. Each generator is given a prototype object which it randomizes and copies each time a new data object needs to be generated. If the verification environment wishes to change what type of data object is created, the environment replaces the generator's previous prototype object with a new prototype. This might be done if a verification environment wanted to change from generating all valid data objects to generating invalid data objects. The prototype object would be changed from one whose randomization constraints allow only valid data objects to one whose constraints allow invalid data objects, all at runtime.

2.6.2 State Pattern

The State Pattern is often used in verification environments because state machines are commonly found in the designs being verified. The State Pattern provides a simple method for creating state machines in an OOP manner. This method avoids the common problem of creating an object for the state machine that contains long streams of if/then statements or case statements, all gated by the current state. Methods will often contain several versions of what they should do, depending on the state, which results in hundreds of lines of code for one method. The State Pattern breaks up this code by placing it into several objects, one for each state. A state machine that contained three states, ACTIVE, SLEEPING, and RESETTING would use three helper objects, one for each state, which all extend from a common parent class.

The state machine would keep a pointer to one of those state objects, depending on what state the machine was in, and the calls to any state dependant methods in the state machine would be passed through, to be handled by the state object. This breaks up the code into logical partitions and allows for good encapsulation of function.

2.6.3   Façade Pattern

The Façade Pattern is a structural pattern that defines how sets of objects should be packaged together and accessed. It allows a complex set of objects to be used through a simplified interface, known as a façade. If a graphics application contained classes for loading and saving a variety of image file formats, it might contain a façade object that provides one simplified load and save method. This method would automatically detect which file format was being used and call the respective load or save functions in that format's corresponding object. This is advantageous as it hides the file format complexity from the rest of the application and provides one single point to edit in the source code when the application is extended to support additional file formats.

The RVM uses the Façade Pattern in combination with the callbacks. When a scoreboard, or other similar object, registers with the transactor to be notified of some event, it does not register itself with the callback, but instead registers a façade object. The façade object provides only a minimal interface to the underlying scoreboard. This simplifies the callback process and allows the scoreboard to make certain methods protected, providing better encapsulation by limiting who can alter the scoreboard.

3.0 Project Overview

Our project consists of variations of a set of designs intended to work together to provide built-in-self-test (BIST) and repair for several memory types and configurations.   Designers do a preliminary check of their Verilog RTL designs, and then the verification team does more comprehensive verification of the designs using Vera as our high-level verification language (HVL). (Unfortunately, we got stuck in the "stone age" and are just beginning to migrate from Vera 6.4.4 to the latest-and-greatest version, with all its new features and fixes.) We took advantage of Synopsys' Reference Verification Methodology (RVM) to create a consistent architecture for our unit, cluster, and system level environments.

A unit environment consists of a basic configuration of a single BIST design macro type (perhaps multiple instances of it), along with the controller design macros. A unit environment uses Vera to drive the instruction, Memory I/O, and the repair interfaces.

A cluster environment adds the designs for the repair interface and the Memory I/O interface, as well as a memory model which supports error injection. Vera code is used to drive the instruction interface and to control signals that result in fails being injected into the memory models by Verilog tasks. Each cluster environment includes a single type of BIST with only one (random) memory configuration.

The system environment adds the fuse designs to the cluster environment. In addition, the system allows for the inclusion of a variety of BIST clusters, not just a single type as was done for unit and cluster.

Figure 5 and Figure 6 illustrate the difference between a unit level and a system level verification environment. The cluster environment is left to the reader's imagination.
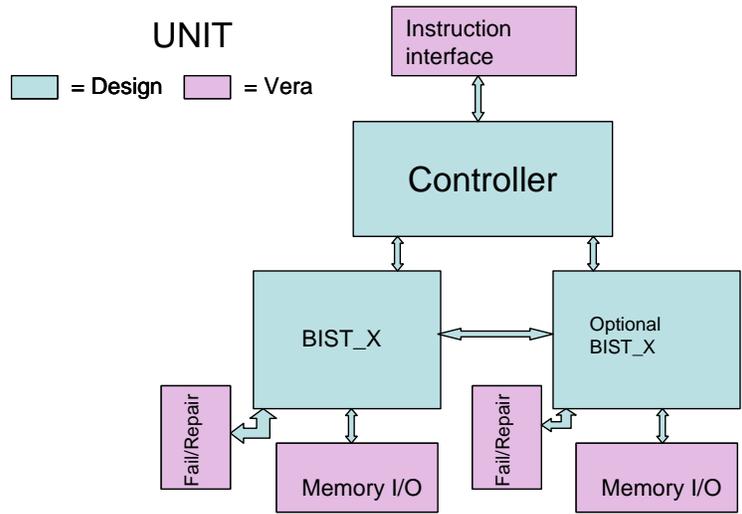

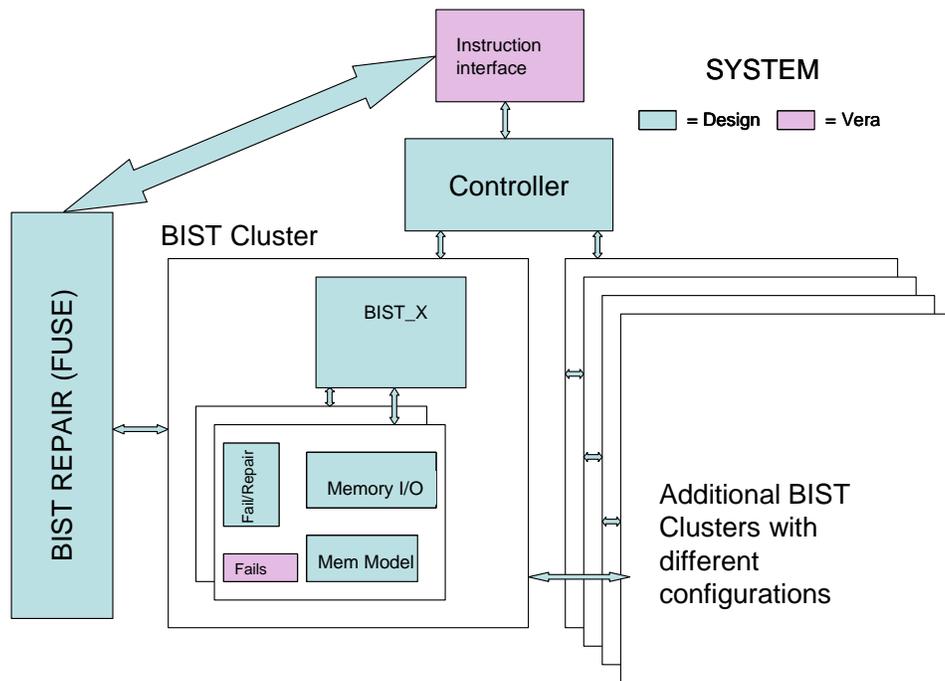**Figure 5: Unit Level Verification**

Figure 6: System Level Verification

The major data objects include: instructions, registers (control, status, repair, counter, etc), BIST and memory configurations, and fail objects (for injecting errors into the memory models). The flow of instructions involves BIST and FUSE instructions which execute in a fairly predictable order, but with a number of possible modes and variations. The main goal of a simulation is to determine if fails can be injected, detected, and repaired (if possible). In order to have a fail detected, patterns are run to access memory addresses in specific sequences. We also need to confirm the accuracy of diagnostic information that can be gathered by reading registers and shifting out data while a running pattern is paused.

4.0 Lessons Learned while Building an Extendable Verification Environment

This section will describe the approach used for each major class type found in all our environments. It will describe the rationale for our original approach, the difficulties that we encountered with that approach, our "better way", and the lessons learned. But nothing is perfect, so in section 5.0, we'll describe some of the pitfalls that caught us as we implemented the "better way".

4.1   *Data Objects (ibm_data extends rvm_data)*

In OOP, most classes revolve around data objects. Data classes are typically the first ones written since their definition is the basis for the rest of the design architecture. When verifying a packet moving design, such as PCI Express, the standard specification defines the format of the packets before the details of the customized architecture are available. For our BIST projects, the heart and soul of the design is in their registers: instruction, control, and status. Their initial description is usually available long before the entire architecture has gelled.

The initial BIST design macro to be verified was a superset of many of its successors. This was a conscious design decision since it's often easier to rip out code to create a simpler design than it is to extend a design that overlooked an architecture issue necessary only for future development. There is also some truth to this concept when designing the verification environment. It helps to understand what will be maintained and changed in future designs, compared to the initial design. Only the common elements belong in the base classes. But, unlike the design development, we don't want to copy and massage code to create future environments for related designs; we want to use an OOP approach and maximize code reuse.

Our initial approach was to jump right in and define all the registers in our first BIST design. A base register class seemed too trivial, since there seemed to be only a few properties that were reused between the registers – string name for messages, size, and address. When would we ever need to reference a trivial register base class? We could simply have placeholders for each register type in classes that accessed the registers and skip using them if they didn't exist. We were even slick enough to think of using a size of zero to indicate that it didn't exist. This worked well for the control and status registers in the unit environment. When we moved to a new unit environment, we included different register types extended from the original, large register type that contained more than we needed. In order to be sure that checkers ignored the values that we no longer needed, we constrained the values to something benign in the

"extended" (actually reduced) version of the register class. We knew this was backwards from the concept of extending classes from abstract to specific, but we were on a mission to get things done fast; no time to pull out that trivial base class now, especially since we had "clever" workarounds.

The instruction data class contained numerous properties based on the instruction type. Everything we needed for the instruction was packed into a single class. We had an enumerated type defined for the instruction modes – RESET, READ, WRITE, RUN, etc. Based on that mode, different constraints applied, different variables were valid/invalid, and the pack and unpack methods worked differently. Initially, the limited number of instruction types and type-related properties lulled us into thinking this was the most streamlined way to deal with the instructions. We used a single instruction data class; what could be more compact?! The generator could simply decide what type it wanted to generate and the single instruction object could be randomized to produce an appropriate instruction. While coverage of this object also required many conditional clauses, at least we only needed to reference a single object.

Things began to fall apart when we added a design that had less in common with the others. In addition, we needed to support several RUN instruction variations and interrupts which made our simplistic instruction object more difficult to generate and process. Unique generators and drivers, customized for alternate scenarios, became necessary.

The final blow was when we tried to mix several BIST types in a single system environment. Instead of having a simple set of registers for one BIST type, we needed registers for potentially all BIST types. This required re-working all the classes that referenced the registers. Now was the time to extract a register base class so that we could simply have an array of registers and swap in the type that we needed once our system configuration was set. We had learned our first lesson.

Lesson 1: Small base classes

Don't be afraid to create trivial base classes; they allow more reuse and are a necessity when mixing and matching extended classes at a system level.

4.1.1   Data Object Constraints

Constraints are applied to data objects to force their random variables to follow rules. The key to controlling a randomized environment is managing constraints. We used four categories of constraints which we named: *valid*, *default*, *test*, and *limitation*. Constraints to generate a legal data object were defined within the data class, like a set of rules; we refer to these as *valid constraints*. *Default constraints*, using Vera's "default" keyword, were used to produce the most typically desired result, and they were also defined internal to the data object. For example, we may want to run in a particular mode (e.g. a more important scenario, or a faster simulation situation) for the majority of the simulations, with only a single test overriding this default. Functional coverage results would ensure that we didn't forget to run that special test. The role of the *test constraints* was to add additional constraints, or to override defaults, in order to steer the simulation in a particular direction. In addition, the data class might contain *limitation constraints*. These were temporary constraints that restricted values for all tests until we were

ready to handle them; therefore, they lived in the class file. Functional coverage results would report a "hole" if we neglected to remove the temporary *limitation constraints*.

Memory fail injection was accomplished by configuring fail data objects using constraints. The fail driver would then apply the randomized fails to the memory model. The fail data object contained valid constraints to ensure that meaningful, yet random, fails could be injected with minimal, if any, additional work in the test. Since all the injected memory fails were dependent on each other (i.e. to force and predict repairable/non-repairable behavior), we had constraints that specifically referenced the fail objects that were previously generated (i.e. randomized). This resulted in tedious constraints with a lot of hard-coded indices based on the number of fails allowed. An improved solution used a SmartQ of fail objects inside a fail package object. There are many useful built-in methods when using a SmartQ such as pick() and constains(). And "foreach" loops in constraints greatly simplified the original complex constraints with hard-coded indexes. (See Example 4 on page 37 and Example 6 on page 38.)

Lesson 2: Hard-coded values in constraints

If you need to write constraints that reference object instances via hard-coded values, something's wrong; try using a SmartQ along with its built-in methods, like pick() and contains(), and take advantage of foreach loops in constraints.

Another issue we encountered was the use of default constraints when we also needed a valid constraint to limit the legal range of values. Any constraint on a variable nullifies the default constraint specified for it.   Therefore, if we wanted to constrain the number of fails to a maximum of nine, but also wanted to make the default be zero, there was no way to do this using the "default" keyword. Tricks like using an enumerated type to restrict us to a limited number of values could be used, but that was contrived in many cases, and only practical if the valid constraint produced a small number of values. Our work-around was to create a constraint that only applied limits when another random flag was set. The flag was then defaulted to be on. This worked, but was awkward to override in a test. If the flag controlled a large set of variables, the entire list needed to be re-constrained even if the test simply wanted to change one. If we had individual constraint blocks for each variable, then we needed separate default flags – yuk! A default constraint is also ignored if it is specified using more than one condition, even if done in a default constraint block! In other words, you can't have a conditional default. Fortunately, a new constraint keyword ("hard_default") should be available in VERA 2006.12 which will not override default constraints. However, it may not address the need to specify a conditional default.

Lesson 3: Default constraints

Decouple *valid constraints* from *default constraints* by using the new "hard_default" constraint keyword (once it's available).

The name of a constraint block is important! When a class is extended, you have the option to override or to add to an existing constraint. If the constraint block has the same name as one in a super class, you will override the super class' constraint block. Be careful to not accidentally do this. If you have a set of *valid constraints*, you don't want to be able to override them. Use a

naming convention to avoid the issue. Perhaps the new "hard_default" constraint will address this too. If you want to override a default constraint block, you need to use the same constraint name.

Lesson 4: Overriding constraint blocks

> Be careful with constraint block names when extending classes; identical names will override, not extend base constraints.

4.2 *Generators (ibm_xactor extends rvm_xactor)*

Generators are needed to create randomized instances of the data objects. We generated random design configurations, BIST instructions, control/status registers, and fail objects to create memory failures. Our system environment also added a fuse instruction generator which needed to be coordinated with the BIST instruction generator.

Initially, we only cared about generating configuration information for a single BIST type and a single associated memory configuration. The configuration generator randomized a basic configuration object with variables for the BIST and its memory information. At system level, we wanted to be able to have each BIST configuration object associated with one or more memory configurations. Since our initial generator simply randomized a BIST configuration object, expecting to get the memory configurations generated for free, we needed to upgrade the configuration object. All the memory-related variables were changed to be a "rand" array whose size corresponded to the number of memory components. But that resulted in ugly hard-coded constraints, based on how many memories were associated with the particular BIST.

Remembering the lessons learned for data objects, we pulled out all the memory related variables and encapsulated them into a separate memory data class. Since the memory object constraints depended on the randomized values in the BIST configuration object, it was natural to keep this array of the memory objects inside the BIST configuration class and make them random. When the generator randomized the BIST configuration object, it automatically produced randomized instances of the corresponding memory objects. But that didn't help the constraint situation much. And with a fixed array size, we had to create the maximum number of elements ahead of time so that it could be randomized simultaneously with the BIST configuration, then, use only the elements that we needed.

Another alternative was to make the generator do more work: First randomize the BIST configuration object, which included the number of memories that should be associated with it. Then have the generator create and randomize a memory configuration object for each one needed, and add it to the array. This was a step in the right direction, and simplified constraints, but using a SmartQ of the memory objects inside the BIST configuration object (similar to the solution for generating fails – see Example 6 on page 38) was… well, smart. (Read more about how and why we built a customized design system using random configuration objects in section 6.0 on page 31.)

Lesson 5: Nested random objects

> To simplify the generator, and make constraints easier, nest a random SmartQ of data objects within another data object if there is a dependency between them.

Our first verification environment had a BIST instruction generator which created the instructions driven to the BIST controller which, in turn, communicated with a single BIST type. We had a specific series of instructions that we wanted to generate, so it seemed easier to create a set of states in the generator and move between those states using a method that could be overridden in the future, if necessary. Each state would correspond to a particular instruction mode. After all, there were only a few modes, and we typically would want to execute them in a particular order. For example, we might want to write all the registers, and then read them all back. Then we could run the patterns and read the registers back to see if the status registers reflected the correct values. One alternative was to put constraints in the instruction data object, or add some logic to its post_randomize method.

Lesson 6: Keep generators simple

> Avoid the temptation to make the generators too smart, just because the options look limited at first; consider moving the intelligence into the data object which can be more easily extended from the test.

Initially, the generator seemed to be a more natural location for the procedural state-changing code, since constraints aren't procedural in nature. However, that belief proved false after we tried to reuse the generator for other BISTs which had different registers to read/write. Also, it became more difficult to change the order of the instructions to create different scenarios for a system environment. Although we were able to make the generator methods more modular, so that they were easier to overload in other environments, it became evident that a dumb generator with smart data objects (i.e. with constraints and post_randomize to control the next instruction generated) would be a better way to go. Alternatively, a scenario generator, and separate instructions for each type (mode) extended from a common base, would have been a good approach. The State Pattern described in section 2.6.2, on page 9, is another option.

Lesson 7: Natural location for code

> Code functionality can neither be created nor destroyed; it just moves from one location to another… but some locations are better than others.

Another job that we burdened the BIST instruction generator with was the generation of the control or status register to be used with read/write instructions. We simply added a list of all the register types to the instruction class, in case they might be needed. The generator would create (i.e. randomize) the appropriate register to be written or read, and assign its handle to the appropriate property in the instruction register. It was easy enough to add to the list of register types in our "base" instruction class, but that broke down at the system level where a variety of BIST types were mixed within the same environment. Now, the generator had to generate read

and write instructions being mindful of what BIST types were in the system. Having a single base register class inside the instruction class became a requirement, not just slick coding. The generator could choose which register to randomize from an array of base register, based on the register address in the randomized instruction object. The registers in the array would have been replaced with the appropriate extended version (i.e. from the build() method in the environment class). The handle for this extended register (or a copy), once randomized, would then be assigned to the single register variable in the instruction object. This works because an extended class is also of the type of its ancestors.

Alternatively, a SmartQ of registers inside the instruction register, similar to the approach used for the generation of configuration and fail objects, could be used for generating instructions that required register members. If we migrated to using individual extended instruction classes for each type of instruction, we would be back to the issue of how the generator would know which type of instruction to generate. The key to generating a variety of extended types (such as registers, or instructions of various types), is to have the generator define the randomized object to be of a base type. (See Example 7 on page 41.)

Lesson 8: Declare the factory template to be an abstract (base) class

A generator should only reference a generic abstract (base) class for objects to be randomized. The build() method in the environment, and/or the test, can then swap in a customized version extended from that abstract (base) class prototype prior to randomization.

The fail generator was responsible for generating a set of coordinated fails which would result in repairable or non-repairable errors, based on a general configuration setting. Remembering to keep the fail generator generic, the work to create unique fails targeting specific goals (repairable or not) was encapsulated in the data object by using constraints. An assumption was made that a pattern that would access the entire address space would be run, guaranteeing that the fail would be detected. Since a system test could take a long time to complete, we took a shortcut and ran the fastest pattern set. Oops! We expected a fail to be detected, but since the design didn't always access the failing address, the actual and expected results would sometimes miscompare (depending on the pattern range and where the fail was injected). Since the fails were generated at the beginning of simulation, we had no way of knowing what patterns would be run. A better time for generating the fails was just prior to starting the pattern-run. At that point, the address range to be covered by the patterns would be known and the fails could be constrained appropriately.

Lesson 9: Check assumptions

Remember what ASSUME stands for, and clearly document, or even better, add a check or constraint for any assumptions to avoid wasted debug time.

Generators generate a data object and pass it along via a channel to some other class, typically a driver. However, we weren't consistent at first about what to do after getting control back from the channel put_t() method. Did the channel receiver defer pulling from the channel until it was finished processing the object (i.e. did it peek, process, get_t())? Or did the receiver pull from the channel, process the object, and then notify that it was DONE? Could we rely on the channel

receiver always moving to the next clock cycle if it didn't need to do anything with the data object? Without a consistent plan, the generator became coupled with the receiver on the other end of the channel.

Lesson 10: Be consistent with blocking

> Be consistent about the use of notify events vs. channel receiver blocking mechanisms, and keep all dependencies on clocks out of the generator.

Once we moved to a system environment and were ready to stimulate the fuse side of the design, we needed to generate fuse instructions. Having learned from our experience with the BIST instruction and its generator, we kept the FUSE instruction generator simple. Instructions were created based on a set of constraints defining legal choices; we now automatically randomized our instructions within legal parameters instead of relying on an overloaded move_state() method in the generator. The order of the generated instructions was no longer locked into decisions made in procedural code, thus allowing for greater randomization.

Lesson 11: Encapsulate to allow more randomization

> Moving instruction type decisions from the generator into the data object allows for randomization of the stream of instructions within legal options.

We didn't anticipate needing to pause the BIST instruction generator, so we didn't initially include the typical wait_if_stopped_t() check in its main_t() loop. Once we realized that the FUSE instruction generator and the BIST instruction generator would be sharing a common interface, and that they required taking turns based on the instruction stream generated, we added the "stopped" check to both generators. A callback hook after each instruction allowed us to extend the callback method to appropriately stop the FUSE generator and start the BIST generator. A similar hook existed in the BIST generator, but up to this point there had been no need to use it. Now we could coordinate taking turns creating instructions, with independent generators, and drive a common interface.

Lesson 12: wait_if_stopped_t()

> main_t() loops should always include a check to see if they are stopped before proceeding, and callback hooks should be strategically placed to allow customized processing.

4.3   *Checkers/Monitors (ibm_xactor extends rvm_xactor)*

We had hopes of reusing the checkers extensively, especially the pattern checker. The BISTs, though different, had many common patterns. Our initial pattern checker contained a few basic methods that called a separate method for each pattern type. The size of the address bus was hard-coded based on our initial BIST type. When we tried to reuse the initial pattern checker for the slimmed down version (1-port vs. the initial 2-port), we discovered that the address bus size had changed, and that a few other signals had also changed slightly. In addition, there were fewer patterns. In order to avoid creating different variables for each BIST type, or subscripting

the largest size, we used #defines to determine the size. Each unit environment had its own global defines file, one for each type of BIST. It worked great! We also escaped at the cluster level because we still used a single BIST type in each cluster environment. By simply conditionally including the correct defines file based on which cluster environment we were in, the pattern checker was reusable. In the system environment, our #define approach proved disastrous, since all the variations had to be included! Since #define values are evaluated at compile time, there was no way to have multiple customized instances of the pattern checker in the same environment using this approach. We had tried to avoid creating separate files to customize each pattern checker, but in the end, that proved to be a small price to pay for the ability to reuse the complex logic in the pattern checker.

#defines were NOT the way to go! They are fixed at compile time and thus the base class is locked into that "shape" for all BIST types. Instead of using #defines, we extended the pattern checker for each BIST type, and in the extended pattern checker's constructor method we re-assigned values that needed to be different from the base pattern checker values.

Lesson 13: Use #defines with care

Avoid using #defines unless they will be constant for all instances; re-define default values in the constructor method instead.

Though many pattern checking methods could be reused for many BISTs, there were situations where we had to calculate and assign just one variable differently; the rest of the big chunk of code for a pattern remaining the same. This resulted in duplicating code that might later need updating in more than one location. With a little experimenting, we were able to find the right granularity and extract smaller methods from the large ones, leaving us with a compact extended pattern checker that only needed to override trivial methods to produce a customized checker for each BIST type. In addition, smaller methods are easier to understand and maintain. If your methods start running on for several pages, and you keep adding more conditional clauses, it's probably time to re-architect.

Lesson 14: Modularize methods

Less is more – Smaller, modularized methods allow for more reuse with less duplicated code in overridden methods.

4.4 *Information Containers (aka scoreboards)*

Verification is all about comparing actual values to expected values. In order to keep track of the registers in our environment, we have a register information class (reg_info). It contains instances of all the status and control registers. When a register is read, the actual value is compared to the expected value stored in the reg_info class. Each BIST instance has its own corresponding reg_info instance.

Initially, prior to creating a register base class, our reg_info class contained a separate register declaration for each type. When a callback method indicated that a register needed updating, a reg_info method used a case statement to select the correct register to update. It then directly

accessed the register's properties (e.g. for status register updates) or copied the register's content from the instruction object's register member (e.g. for control register writes). Our reg_info class quickly developed into more than a passive container class. As we included different BIST types in our environment, each with their own register types, we needed to add new methods and register declarations and to change the extended callbacks to trigger the appropriate update method based on the BIST type. Once again, we were lulled into complacency because this didn't pose any serious limitations in our unit and cluster environments where things were homogeneous (i.e. a single BIST type with a single memory configuration).

At the system level, the absurdity of defining the registers individually with their unique types resulted in enough difficulties that we were ready to take advantage of polymorphism. We switched tactics from individual unique register types to a factory pattern approach where we defined an array of registers of the base register type. In the environment's build() method, we created an instance of each extended BIST register type used, and swapped them into the appropriate reg_info array element, using the register address offset as the index. Once again, we traded code in one location for code in another location. Our decision to use a factory-pattern-swap approach streamlined our reg_info class, eliminating the need for conditional code and case statements previously used to resolve which register to access, but it added to the build code in the environment class. It was a wise trade-off, however, because the build code would be executed once per simulation, while the streamlined code saved us from making decisions on each call for a register update. Furthermore, it is better to optimize code that will be reused in other environments and possibly extended (as is the case with reg_info); the environment class is meant to be customized for each verification environment. However, our approach created new challenges because now reg_info couldn't directly access register properties that no longer existed in the base register class. We needed to use virtual methods to operate on the extended class' properties. These polymorphism snags will be discussed more in section 5.0 beginning on page 26.

Lesson 15: Access with virtual methods

Using a common base class as a factory pattern template allows code to be streamlined and generic, and forces the recommended practice of accessing properties via virtual methods.

4.5   *Drivers (ibm_xactor extends rvm_xactor)*

The drivers started out very simple. The primary driver was the BIST instruction driver. It received an instruction object from the input channel, then called the appropriate method to process the request based on the instruction mode. Processing meant driving a few signals, issuing a callback in case anyone was registered to do some extra work (like check the patterns being generated, or trigger coverage), then returning control to the generator. However, the driver soon evolved into a monitor/checker/generator/driver.

This was particularly true for the "RUN" instruction. Initially, we just cared about running a set of patterns until DONE was indicated by the design. The driver retained control until it saw that the run had completed. But, some of the run instructions included modes that allowed for a variety of interrupts which could pause the BIST while some diagnostics were executed. Little by little, conditional code was added to handle these modes. The communication between the

driver and the generator became more and more complex. The driver started making decisions to generate its own requests because the requests were not specifically instructions, but assertions of independent signals. As usual, when running with a single BIST type, we were able to override parts of the driver to customize as necessary. But when piecing the BIST types together for a system environment, we needed a common instruction driver, but separate monitors and checkers for each BIST. None of our unit drivers were generic enough to be used in the system environment, so it was time to re-architect.

We needed to go back to a single, simple driver which returned control to the generator as soon as it finished driving the request. The generator, after getting control back from the driver for a "RUN" instruction could, via a callback, activate a monitor/checker to watch for interesting scenarios (i.e. completion of the request, time to create an interrupt, time to resume after seeing the design pause, watch-dog timer expiration, etc). The monitor/checker could indicate an error if necessary, or it could communicate back to the generator through the instruction object by setting events or variables in the instruction object. This would then allow the generator to create a new request based on what the monitor saw. Pseudo instructions could be created so that the same instruction object could be used by the driver to send interrupts or to resume. Alternatively, the driver could wait for the monitor to indicate that an interrupt or completion had occurred. Using the callback approach, with the monitoring removed from the driver, allowed us to register as many monitors and checkers as needed.

Lesson 16: Keep drivers simple

Drivers should leave the work of monitoring signals and generating requests to separate transactors, no matter how basic the task. Nothing stays trivial!

At the unit level, we needed to drive the interface that indicated that memory fails were detected and possibly repaired. We wanted to be able to control when the fail signals were driven to the BIST, based on which address was being processed by the patterns in a given cycle. The catch was that for a two port BIST, we had the potential of being in a different clock domain based on which port was active at the time. We thought we'd be clever and define a single fail driver and instantiate it for each port, binding it to the appropriate clock interface. Since only one instance would be triggered at a time, we figured that the independent driver instances would simply drive on the appropriate clock edge. Wrong! By driving the same signal from two different clock interfaces, we got drive signal conflicts. Even using soft drives didn't help since there was no communication between the driver instances. We needed to use a single driver with access to two clock binds. When triggered by a particular port, we moved to the clock edge associated with that port. We then did an 'async' drive of the fail signal to avoid synchronizing with the wrong clock edge.

Lesson 17: Driving the same signal from multiple interfaces blocks

Don't use more than one driver for the same signal bound to different interfaces

Our drivers accessed signals through virtual ports based on the port bind that they were passed in their constructor method. Since our environment included a random design configuration, it was possible to need a large number of binds, based on the number of components included in our

configurable wrapper (see more information on this in section 6.0 on page 31). This resulted in a lot of hard-coded, replicated binds. Too bad the binds couldn't be defined using procedural code, like a loop. Busing related signals in our test_top helped a little. We spent a lot of time expanding the bindings before resorting to a macro. (See Example 3 on page 36.)

Lesson 18: Macros for repetitive code

> Harness the power of macros when systematic duplication of definitions is required, such as in port binds.

4.6   *Coverage*

What's better? Internal or external coverage? We originally decided that external coverage (i.e. a separate coverage class) was the way to go. It provided the flexibility to look at values and events from multiple classes, and allowed one developer to work on coverage while another developer worked on the classes that it was referencing. In addition, the "cumulative" keyword enabled us to track instance-based coverage.

We started with a register coverage class for all the registers in our initial BIST type. Since we had access to all the registers, cross coverage between registers was simple. As we worked on unit environments for other BIST types, we needed to extend or replace the original register coverage class so that we could use the new register types in our definitions. With new BIST types, some registers no longer existed, or new ones were added, or the register fields were different. If we wanted to extend our base register coverage class, we needed a way to override some of the coverage groups. However, Vera doesn't support overriding of coverage groups, so we were forced to play some games.

Since we couldn't guarantee that a register existed, or that a particular member existed, we couldn't directly reference the properties in the coverage sample statement. We created local variables in the coverage object and set them when we were called to get information from a particular register… then triggered the coverage group that would use those variables. The plethora of duplicated variables raised a red flag; we were doing something wrong! Since we didn't want pessimistic coverage results, we had to conditionally get rid of coverage groups or bins when we didn't need them, due to the design configuration being used. We tried using an array of flags in the constructor method to determine which coverage groups of bins to keep and which to nullify or de-activate. All of this worked, but the code was sloppy, making it too complicated to follow and extend. Good thing programmers are a lazy lot; we weren't satisfied until we discovered that defining coverage as part of the data class ("internal") would be a lot easier than passing the object handle to a separate coverage class ("external").

Yikes! How can we suggest including the coverage definitions inside transactions (i.e. data objects)? Aren't tons of data objects generated during a simulation? Wouldn't this create too many coverage group instances, making instance-based coverage explode out of control? Maybe! But in our situation, we only cared about instance-based coverage for data objects that were limited in number – configurations or fails, for example.   While we execute many instructions, we only had one active at a time, so it was only necessary to create a single instance. The generator, through a factory method, would check to see if the instruction object (s)

were instantiated (i.e. by the environment class or by the test if it wanted to customize it). That instance could then be randomized repeatedly. Register objects were a bit different. We needed register instances in the READ/WRITE instructions, and also expect/actual versions of the registers in reg_info (the scoreboard). For now, we ignore instance-based coverage for registers, but strongly suspect that a simple solution would be to only create the coverage groups for registers included in the scoreboard (i.e. assign them to null for other instantiations). At any rate, a little extra CPU/memory overhead is a small price to pay for making coverage groups extend naturally along with the data object, without the overhead of managing external coverage groups.

Lesson 19: Internal coverage groups advantages

> Using internal coverage groups can prevent the need for conditional coverage methods and flags, excessive local variables, and extended or duplicated coverage classes.

Coverage of our design configurations also had similar issues when coded in a separate class. The number of BISTs and the number of their associated memories was random. When we initially had an array of memory variables inside the BIST configuration object, we needed to have hard-coded coverage groups based on the number of memories that were valid for that BIST. Although the coverage for the BIST configuration was instance-based, the coverage for the memory instances couldn't be instance-based. After moving the coverage into the BIST and memory configuration objects, they only existed if the object was created.

Lesson 20: More advantages of internal coverage groups

> Locating coverage groups within the data objects allows for more natural instance-based coverage.

But what about cross coverage between multiple instances? Since the trigger for filling a coverage group typically comes from a callback, and since the callback method typically includes a handle to the calling program, you will have access to a lot of information. In addition, the constructor for the callback can require an argument that is a handle to other objects, such as configuration objects or information containers. Depending on what values are being crossed, you should choose the more natural location for the cross. For example, if you want to be sure that each legal memory configuration is generated for each BIST type, the memory instance can sample the BIST type that it belongs to and cross that with its memory type. Trying to do this cross from the BIST configuration object would be more difficult, since the number of memory instances it contains is variable.

Sometimes, cross coverage is not the best way to collect data between instances. If you want to be sure one memory was injected with repairable fails, and another memory on the same BIST was injected with non-repairable fails, you'll need to access multiple memory instances. Since the BIST configuration object contains instances of memory objects, the BIST configuration object is the natural place to check this. However, since the number of memory instances is configurable, it is probably easier to do a check for this condition and set a coverage variable that can be sampled.

Lesson 21: Cross coverage

Find the natural location for collecting coverage between instances, and consider that cross coverage may not be the best way to check that a particular scenario occurred.

In order to generate a coverage report that is easy to evaluate, advanced planning is needed. Numerous, small coverage-groups allow for more granularity in the coverage comment, and less sub-levels to have to navigate through in the report. On the other hand, limiting each data object to a single coverage-group will provide a more concise top-level report, and offers the ability to cross more samples (since crosses can only operate on samples within a group). Since coverage groups are listed in reverse alphabetical order in the report, there won't be much rhyme or reason to the order unless you keep this in mind as you name the groups. Following a guideline as simple as "name coverage groups with numbers padded to equal lengths" can simplify looking through the list. (i.e. ctl_reg_00….. ctl_reg_15). If you hope to keep related classes together in the report, choose the common part of the group name for the beginning and add the newer tag to the end of the name. (Note: There are many updates to the latest Vera release in the area of coverage, and we are just beginning to explore them.)

Lesson 22: Planning for coverage reports

Choose coverage group names and hierarchy carefully in order to generate a manageable coverage report.

4.7   *Callbacks*

Callbacks – Can't live with them. Can't live without them! At least we got this right the first time… almost. Callbacks may seem cumbersome at first. Why do we need all these extra little classes? They create file management issues: Do we group them together in one file? Do we create many individual files? Do we embed callbacks in the file with the class that they will be referencing?

Once again, what worked great at a unit level fell apart at a system level. File explosion is an issue when there are many base classes and class extensions all in separate files. Adding a plethora of small extended callback classes adds more file clutter, so tucking them into the classes that they are calling helps a bit. Also, dependencies are more obvious when the calls and the methods are in the same file. However, since the extended callbacks inherit from a callback façade, when those base class callbacks are no longer used, there will be compile problems. This missing base class situation can occur when a driver used at a unit level is replaced by a design. Now, the base class is not included in the compile list, and the unused extended callback needs to be moved to its own file so that it can be included only when needed. It's possible to store an extended callback with its base class, but the extended class may access other classes that aren't yet available (i.e. compiled), and excessive typedefs would be required.

Lesson 23: A home for extended callbacks

Keep extended callback classes in their own file.

Once you find a home for the extended callbacks, you still have to deal with the issue of keeping track of who is calling whom and whether or not you got the method names right. It's best to use some common naming convention for methods in the base callback façade, such as pre_trans_t() and post_trans_t(), to make it easier to get the name right in the child class. If the extended callback doesn't define the overridden method using the correct signature, it will be creating an overloaded (not overridden) method that will never be called. Quite an interesting debug exercise! One trick that ensures that any extended method is overriding a base method would be to invoke super.<method_name> with the same arguments. Since the façade only contains empty virtual methods, it won't do anything, but the compiler will detect that the parent method doesn't exist.

Lesson 24: Match callback façades

Be careful to match a callback's façade when overriding it. Invoke the empty super.<method_name> to assure that the extended callback has the potential to be called.

The argument list used by a callback should be kept as simple as possible, with the calling class' handle likely to be the only argument needed. But what if the calling class can be defined as a base class or an extended version? What type should the argument be? Fortunately, if the lessons about keeping generators and drivers simple have been learned, there shouldn't be much need to have an extended version of them, thereby saving you from needing to choose between the parent and child classes for your argument type. In general, if you must choose, it's best to define the callback façade from the base transactor class, and therefore use that type as the argument used by the methods.

Lesson 25: Callback arguments

Callback method argument lists should be as minimal as possible and reference the base class of the calling transactor.

Despite the challenges with file management, and with keeping method names and arguments straight, using callbacks is definitely the way to go. They give you the ability to communicate between classes, and allow customization at different verification levels. Instead of extending or altering a generator or driver, a callback hook allows for insertion of additional monitoring/checking or triggering of coverage events. Think of a callback as a way to compress a stream of conditional calls to methods in other classes into a single line of code (a callback macro). Although the "removed" code will still exist in the extended callback classes, you now have the ability to conditionally include it and the ability to customize arguments through the extended callback's constructor method. The flow of the code that triggers the callback stays uncluttered, making it less complex and therefore less error-prone.

Lesson 26: Advantages of callbacks

Callbacks are worth the extra effort; they allow the transactors to remain as generic as possible while allowing for customization and inter-class communication.

5.0 Polymorphism Pitfalls

Our attempts to maximize reuse of the verification code across similar design versions (horizontal reuse) and with a hierarchy of design instances (vertical reuse), taught us many lessons. Even when we "saw the light", and knew we were on the right path, we encountered snags that made us question whether we had overestimated the benefits of polymorphism. Until we gained some experience, we got caught in a few polymorphism pitfalls and had to debug our way out.

5.1   *Creating Class Hierarchy Wisely*

As we discussed in section 4.1, "less is more" – small base classes are more reusable. Think about what is common to all variations of the class you want to create. Only include in the base class what you will need for **all** extensions. The base rvm_data class is a good starting point, but if you are making a base register class, you'll also need other information for all registers, such as size, name, address, etc. The individual register fields will be customized in the extended version. If you find out later that you are repeating something in all extensions – put it in the base class. If you are repeating many, but not all properties and/or methods, perhaps an intermediate class would help; more levels of hierarchy allow for less duplicate code.

What about an instruction object that represents what is written to an instruction register? Since it's not variable in size and the fields vary based on the instruction, should you have a bunch of random fields that are constrained and used based on the instruction type, or should you have separate instruction classes extended from a common base, where each specialized type only contains properties that pertain to that instruction type? What if the instruction will read or write a register? How do you couple the instruction with a register and still allow the test to have control of constraining the register number or fields to be written? Do you have a generator create the instruction and also create the register when the instruction requires it, or do you have extended read and write instructions contain a random register? Enough questions! How about some answers? Answer: "It depends. You can do whatever you want to."

Asking questions is a good way to evaluate what's important to the team. As with directory hierarchy, some people like a wide, shallow tree, while others may prefer deeper hierarchy with a few directories at the top. With objects, the choice to nest related data objects, or create and manage them independently depends somewhat on preference, but also on their relationship to each other. If one object needs to be randomized before the other, then the nested approach would need to add constraints to control the randomization order (i.e. "solve before"). The non-nested approach would rely on getting information from the first object to the second. In general, if object "A" needs a variable number of object "B" instances, then you should probably nest a random SmartQ of type "B" in "A" (e.g. BIST configuration). If "A" only needs a single instance from a list of type "B", use a SmartQ of type "B" and the pick() method in pre_randomize() to assign a random element to a "rand" member of type "B" (e.g. write instruction). The new version of Vera allows "pick() with" for even more flexibility! For selections from SmartQs whose members are of type string, integer, or bit/reg, a constraint using "contains()… with" is useful. When nesting random dependent objects, enforcing the dependency through constraints and SmartQ methods may not be intuitive. Experimenting with sample code (check the appendix!) using vera_cs is a fast way to gain experience. Once you "get

it", you'll have an almost trivial generator and it will be easier for tests to manage customization of related objects.

If you decide to go the nested route, be sure to nest the base class type inside the other base class; don't use the extended classes directly. Once some morphing starts, all objects can interact through a set of base class virtual methods…. if you thought of all the methods you'll need. The rvm_data base class already has a comprehensive set of virtual methods that should accommodate most operations that need to be done on a base class. With "kind" flags, and overloaded methods, you should be able to prevent an explosion of new methods. Most overloaded methods will want to call the parent method first (super.<method_name>) so they doesn't have to duplicate the work.

In our first design, we had a register class that wasn't used in the next design. Since other classes that referenced registers objects had a hard-coded list of all registers (before we wised up), we were never sure if we were allowed to operate on a particular register; we were afraid of getting a null object access error. We could test for null each time, but that would be a nuisance. Or we could extend the base register and override the size to indicate that it didn't really exist. Or, we could make the default in our new register base class be an undefined register! That way, if we didn't extend it and override it, it wouldn't be valid when tested using the rvm is_valid() method, which was already being referenced… and that register could be ignored. Once a fixed array of registers is replaced with a SmartQ, the base register class should default to an error if a non-extended version is ever referenced. It makes more sense to create a register from the correct type when it's needed than it does to use a dummy base register as a place holder in an array.

One reason to extend a class is to apply additional constraints. This is typically done in a test. Another reason could be to apply across-the-board limitations or defaults for a particular environment. This brings us back to the short-and-wide vs. tall-and-thin debate. If we have a general object that extends from a base class, and we want to add one teensy-tiny extra bit of information to it for a few special conditions… and all instances need to have the same new constraint applied, what do you do?
1. You could go back to putting extra information in the base class and use or ignore those properties conditionally, and even include the new constraints in that same class – fewest files/classes.
2. You could create a single string of extensions, adding more information in each extension, and then add the constraints to the final extension – tall-and-thin.
3. You could create separate extensions for each special case, then extend each of those and apply the identical constraints to each one – short-and-wide.

Plan (1) is "out" since it violates the lessons learned earlier – keep base classes simple. Plan (2) has the advantage of modularizing the changes to allow for branching at several points, and it ends up with a single class at the final extension, but it inherits unnecessary baggage from each ancestor class. Plan (3) is logical, but the last extension has a lot of duplicated code, making it a candidate for maintenance problems. How about a fourth option that adds the constraint extension just above the common point? That would eliminate the duplicate code, and keep the special conditions isolated from each other. The down side to this approach is that the special constraints are usually thought of after the special extensions have already been created, so it would mean editing all the extended classes to change their base class. However, if you plan

ahead for this kind of thing, you could create a dummy buffer extension that shields you from touching the general base class, but allows for later updates. Temporary or permanent additions, like more constraints, could live in this level. This concept is similar to the RVM recommendation to insert a "business unit" layer between the RVM base classes and the project classes.

5.2   *Using Virtual Methods to Operate on Extended Properties*

You've heard it before: "Use virtual methods to access class properties; don't access them directly." In practice, it seems like a lot of overkill. Why create a method when you can simply reference the properties through the object handle? But when you want to declare an object to be of a base class type, most of the properties don't exist. If a property doesn't exist in a base class, even if an extended class is swapped in, the compiler won't let the base class directly reference the extended class' properties. This happened to us in several places after we learned that we needed polymorphism to successfully reuse code at a system level. For example, our original version of the register information class (reg_info) declared each register to be of their extended type. We had access to all the properties, so when an update was needed, we simply called a reg_info update method and accessed the property and zapped it with the new value. No problem! Once we switched over to having all registers be of the base type, we couldn't use the update methods in reg_info anymore. We were forced to access the properties via overridden virtual methods. But how do you create a generic virtual method to access a variety of properties of different shapes and sizes? You'd need to create functions that returned all types of values, or accepted a variety of arguments. We were able to contrive a general enough method that capitalized on the fact that the register properties that needed access were one or more bits, up to a maximum size of 32, at least up to this point. After overriding the newly created get() and put() virtual methods in all the registers, we coded up some pretty cryptic calls to access the properties. How could this possibly be better than directly accessing the properties as we did when we had the extended class declarations in place? We had been forced to move to this base class approach for some good reasons, so something was still wrong.

If the reg_info class was smart enough to know how to update a particular register by manipulating the arguments and return values of the overridden function, then it was not playing its passive role of a simple information container. The job of updating the register fields really belonged encapsulated in the register class itself. Instead of reg_info performing all the conditional updates, it should simply call an update method in the appropriate register class. If reg_info passed its own handle to the register class that needed updating, that register would have access to the same information that reg_info had previously used to make update decisions. The register class would have access to all its properties, and we could now use polymorphism the way it was intended – keep the generic classes basic, and encapsulate the work in its natural location without worrying about what form the data object might have changed into.

By now, you probably see a pattern. When something is simplified, something else gets more complex. Seems like an extension of $E=MC^2$. But, when the change results in better encapsulation of the work, it is in a more natural location and therefore easier to extend and maintain. And sometimes you get lucky and can significantly reduce the amount of code.

5.3   *Problems with Extending*

You can extend a class and override its virtual methods, properties, and constraint blocks, but not enumerated types or coverage groups. The power of overloading or overriding can cause surprises when not used correctly. The inability to override or extend enumerated types and coverage groups can be frustrating, but there are ways to cope with the limitation.

When declaring a property in an extended class using the same name as in the base class, you are creating two copies. If you intend for these to be the same variable or instance, you may be in for a surprise. A method will operate only on variables that are in its scope. (See Example 5 on page 38.) If you really want to overload the property, you'll want to overload all base methods that access it and only reference it from the extended class. However, this defeats the purpose of using extended classes for inheritance, although it does allow you to "cheat" and directly reference extended properties. Instead, "morph" the base property by swapping an extended instance for the base instance, and then operate on its members through virtual methods.

Overriding constraint blocks is handy when you want to change a set of defaults. But the constraint block name must match. Again, be sure you are doing this intentionally. And remember that all the members of that block need to be redefined unless you want to get rid of the default for some of the variables. If you want to generate an invalid object (i.e. for error injection), you can generate the good object, and then corrupt it by applying specific types of errors through methods. Or you can turn off the valid constraint; however, this doesn't insure that a bad object will be created. Overriding the valid constraint with an invalid constraint would work, if you get the constraints correct. The base class should have the broadest constraints to generate a legal value. Additional constraints to further restrict the value would be added in an extended class. If your constraints don't seem to be generating the correct results, look for accidental constraint block overrides, or constraints that might nullify a default constraint. Keep in mind that "default" constraints are very easy to nullify. A default block that conditionally offers alternate constraints for a variable, will compile and run, but not provide a default constraint.

You may want to extend an enumerated list (i.e. override it and add more values) in an extended class, perhaps because there are new instructions, states, etc. However, Vera doesn't support extending the definition for an enumerated type. So what do you do? Edit the enum declaration in the base class? Make the enum definition global and re-define it, as long as you are only adding to it? Define a new variable in the extended class to handle the other possibilities? If you try to override (re-declare) an enum, you'll get a compile error, so at least you won't waste time debugging, as you may with other subtle overload issues. The only solution we came up with was to add an "extension" value to a base class enum, and constrain it off by default. If an extended class needed more enum values, it would define a new enum variable that would be used when the base enum resolved to the "extension" value (i.e. if the default constraint was turned off). If the extra extended values make sense for all extensions, we might cheat and simply update the base definition. It's not a perfect solution, especially when the enum is a random variable. In that case, the distribution of the used enum values is off balance.

It may seem like a lot of unnecessary work to "override" or "extend" an enumerated type. Why would you need to use an enumerated type anyway? Enumerated types are handy for giving a nice name to an encoded value. They print nicely using %s. By assigning a value to each enum you are able to avoid the need for procedures to convert between string values and encoded values. However, there is a caveat. An enum variable can be directly assigned to an integer or bit vector (maximum 32 bits). However, when assigning an integer or bit vector to an enum variable, you will have to ensure that only legal values are used, and the compiler will require that you guarantee this via a cast_assign. This is not unlike an object declared as a base class being assigned to an object of an extended type. You need a cast_assign to be your promise that you will either swap in a matching type object prior to the assignment, or that you won't need to access any of the extended properties. (See Example 2 on page 36.)

If an extended coverage class wants to nullify a coverage group defined in the base class, so that it doesn't show up in the report, it's out-of-luck. If the group in the base class is assigned a null value from the extended class' constructor method, it has no effect. If the new/null of a coverage group is moved out of the constructor method, in an attempt to override it (you can't overload the constructor)… you'll get a compile error. You could use the flag trick described in Lesson 19, but you are better off adding coverage groups as you extend, not starting large and then putting the base class on a diet.

5.4   *Where Did My Filled-in Object Go?*

There are lots of ways to mess things up when making a new instance of an object or making a copy of an object. Sometimes you inadvertently create a new instance when you intend to copy into an existing instance. Or you send an instance into a channel, then clobber it by re-randomizing it before you are finished using it.

Don't pass a class argument of one type and then, instead of assigning it to your local copy, do a new() of the local version to try to allow the extended definition in the local copy. You might be tempted to do this if your argument type could not be assigned the class member, because of a type mismatch. You'll see the value get passed in correctly, but then it will mysteriously disappear. You are essentially ignoring the object being passed in.

Also, watch out for making copies of objects using an overloaded copy method. If you add a new variable and forget to add it to the copy list, you'll get most of the copy, but not all, and you can get misleading errors. Beware of shallow copies of object members too!

When generating a randomized object, you may want to make a copy of it before putting it into the channel. This avoids the possibility of having a downstream class replace the customized factory prototype with another type. It also prevents the generator from accidentally altering (i.e. re-randomizing) an object that might still be used by the channel receiver. This could happen if the receiver (e.g. a driver) is not blocking the channel until it finishes processing the object (e.g. perhaps a forked process is still using it). To have the driver block, peek into the channel and only do the get_t() when the driver has finished processing the object. Note that the channel should be of the data object base type so that all related extended types can use the generator and driver channel connection.

Also remember, if an extended class declares a variable with the same name as one in the base class, you've created relatives with the same name. Depending on how the variable is referenced, you may not get what you expect.

5.5   *super but no super.super*

When extending a class, if you want to add new arguments to the constructor method, then be sure to use super.new() as the first statement in the extended constructor. This allows you to access all the base class goodies. Or, if you simply want to do something new and different in the extended constructor – like re-define "constants" (like we did with the pattern checker) -
call super.new(), and then do your re-defining.   If your extended class is simply adding more constraints, the default constructor method is adequate.

But what if you want to reach back a few levels and specifically reference something from a base class that has been overridden. There is no super.super in Vera, so you'd have to use a virtual method, somehow. If you find a need to reach back like this, something probably isn't extended right. Re-evaluate how your class hierarchy is architected.

5.6   *Interface Files, Virtual Ports, and Binds*

Too bad the interface, virtual ports, and bind files are #include files. Their values are determined at compile time, not run time. Maybe System Verilog will improve things. We have numerous design configurations and therefore we need to generate many binds. Loading an array of these binds is a very hard-coded process since we can't build the name of the bind from runtime variables. We can use a macro to make this more compact, but it's still mostly hard-coded and requires a large number of ifdefs, in case our design configuration pre-empted some declarations. We also need to define all possible signals in the test_top so that the interface file can be static; conditional code isn't allowed in include files since we're not using a multi-pass compiler.

One-dimension configurability can be handled by making a bus of the signals in the test_top and then having Vera index into that array to get the right instance of the signal bit. But if we have multiple BIST clusters and each cluster has multiple BIOs, we need a two-dimensional Verilog vector (not available in Verilog-95). The solution is lots of typing, or a script to do the typing for you.

Then there's the issue of not being able to create instances of Verliog tasks; they need to be created separately. This was an issue for fail injection into the memory. Maybe there is a better way to access the array inside the memory model than using a Verilog task, but we didn't find one.

6.0 Dealing with an Extendable Environment on the RTL Side

Today, reuse of a verification environment includes the ability to plug-n-play a variety of related design components. It requires a methodology for creating a generic wrapper that is capable of adapting to design modifications or new designs with minimal (re-)work.

If this is the case with designs, then we also need equally "configurable", "modifiable", and "reusable" testbenches to cope with time-to-market challenges. Some of the earlier testbenches

needed to be tweaked beyond recognition or had to be re-written to test/verify all the variations of the re-usable IP, and this is very time consuming.

Effective progress in this direction requires new ways of customizing a design configuration and coupling that configuration with the HVL environment. This section is an effort to explain some of the tricks used in our re-usable verification environment.

6.1   *The Configurable Wrapper*

When building an extendable environment, reuse of the design wrapper, test_top, interface file, and ports/binds needs to be considered. These aspects of the environment do not consist of object elements, so we are limited in how we can make them reusable across multiple designs and configurations. The approach we took was to build a configurable wrapper that could conditionally instantiate different design modules. The verification environment needed to be aware of the configuration so that it could adjust accordingly. The communication link between the configurable wrapper and the Vera side was a file that contained customized `defines.

The first step when architecting the configurable wrapper is to determine what aspects are variable between configurations. The most obvious variations were the types of modules that would be included, and the number of each. For our unit level testing, we limited ourselves to a single BIST type for each environment. We allowed for multiple copies, but constrained the number to be small so that we didn't have to do as much work creating the binds. The binds were still a nuisance, even though use of a macro helped. For cluster and system level testing, we expanded to allow a variety of BIST types with each BIST having a variable number of differently configured memories attached. Again, we constrained the numbers so that we didn't have to write as many binds. These variations required that we provide some `defines.

Depending on what type of BIST was being instantiated, there were different signals to connect to the test_top (and to other design macros in the cluster/system environments). That required another step to conditionally connect or tie off signals that needed customization. This variation required even more `defines. And for cluster/system level, the memory configuration required a huge number of options. Fortunately, the naming convention of the memories, combined with the BIST type, could be used to determine the presence or absence of many of the pins. Our 2-pass methodology, described in section 6.2, ensured that compatible/legal `defines were created in a file that was included by the wrapper.

The test_top included all possible signals, sometimes aliased to a common name if the functionality was related between BIST types or memory configurations. The `defines also provided the necessary information for the Vera side of things. The interface file connected to all test_top signals that needed to be accessed. Virtual ports were bound to the appropriate signals based on which BIST or memory instance they applied to.

Some test_top, wrapper, and design model code depended on information obtained via a *parameter* keyword. Whether it was for clock speed information, or the name of a file that needed to be read, the Vera code needed to be aware of the value of the *parameter*. Once again, the `define came to the rescue.

The last mechanism we used to allow the wrapper to be configurable was to use helper verification signals driven by the Vera environment to communicate with Verilog tasks. This was primarily used for fail injection into the Verilog memory models. Other verification-only signals were used to help provide debug information when viewing the waves (e.g. cycle counters when shifting instructions).

6.2 *The 2-Pass Methodology*

Our simulation script allows us to run a preliminary, stand-alone (i.e. no design) Vera simulation, using vera_cs, to generate the customized defines file. This allows us to harness the power of random variables and test constraints to define our design configuration. Since our env class (extended ibm_env which extends from rvm_env) includes this magical defines file, it needs to be available at compile time in order for the custom defines to be generated. `Tis a chicken-and-egg scenario! To get around this, a default defines file is provided to allow the initial compile to work. On the second pass, the include search order will ensure that the correct defines file is picked up (i.e. the generated one instead of the default one).

On the first pass, a configuration object is randomized, then a correct-by-design defines files is written, based on the results of the generated (i.e. randomized) configuration object. The env.gen_cfg() method then exits. It knows that it is working on the first pass because of the existence of a predetermined Vera plusarg. The script also accommodates two random seeds. The first seed is the "cfgseed" used for the first pass. The second seed is the seed used for the second pass.

On the second pass, the test starts again, and this time the env.gen_cfg() by-passes the generation of the configuration through the randomize() mechanism. Instead, on this pass, it accesses the `defines and uses the information to fill in the appropriate variables that will be used by many of the other classes. These variables provide information about the number of BISTs included and their type, the number of memories and their types for each BIST cluster, clocking information, and other information decoded from the memory name.

7.0 Wish List

Creating an extendable verification environment poses many challenges. Using Vera with the Reference Verification Methodology enables you to surmount most of them, especially as OOP experience improves. But the more we learn, the more we appreciate the need for some "new toys". Here's our wish list in no particular order:

- *Auto Documentation* – We have primitive scripts to do this to some degree, and 3rd party tools can help here, but it would be nice to have something packaged with Vera so that the parser can be reused... OK, and so we don't have to pay extra or do more work.
- *Vera Linter* – Synopsys has it for HDLs (i.e. LEDA), but how about for Vera? Perhaps LEDA will be, or already is, expanded to have rules for the HVL portion of System Verilog? Again, we can write our own or turn to a 3rd party source.
- *Class/Method Hierarchy GUI* – The creation of extensive class hierarchy makes it more difficult to keep track of all the files. A visual representation of the verification "tree",

created automatically from the source, would be a great aid for training new team members and it would provide accurate documentation of the environment structure.

- *Coverage GUI and post-processing* – Evaluating coverage is consuming more time as our environment becomes increasingly sophisticated. The ability to have more flexibility when viewing the coverage reports would help immensely: Show me the "holes"; recalculate coverage statistics after-the-fact; ignore certain groups/bins; let me mark critical holes as a high priority…. Rumor has it that the latest version of Vera has addressed the "show me the holes" issue! Wahoo!

- *System Verilog* Translator – There seems to be a migration toward System Verilog. Even if the designs don't move to System Verilog, testbenches can take the leap on their own. However, now that we've built an extendable environment, how do we convert all the legacy code from Vera to System Verilog without some help? An automatic translator that offers the ability to be prompted for approval prior to finalizing changes (while offering an explanation for the change), would be a boon.

8.0 Conclusion and Recommendations

Polymorphism and other OOP practices provide the opportunity to improve the reusability of a verification environment. By being aware of the potential pitfalls, it will be easier to architect the classes in a hierarchical manner. This will allow the basic infrastructure to be extended to accommodate design variations and the addition of multiple designs in a more broadly scoped environment. By encapsulating features in objects, design architecture changes are less disruptive to the overall verification structure.

No class template can foresee all options. There will be some re-work, especially when transitioning from one design or verification level to another. Avoid the temptation to pack too much into a class or into an individual method; modularization allows configurability and makes code easier to understand and maintain. Keep generators and drivers as generic as possible; encapsulate rules into the data objects via constraints. Take advantage of callbacks for customizing code. And remember this maxim: "When you lose, do not lose the lesson." Don't wait until you dig yourself into a deeper hole before you make repairs.

Object-oriented programming may seem like overkill for initially trivial situations. But as complexity increases, and it always does, OOP will grow on you. It takes experience to appreciate and harness its power!

9.0 Acknowledgements

10.0 References

Reference Verification Manual, Synopsys.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reuseable Object-Oriented Software.*. Addison-Wesley Professional, Boston, MA, 1995.

11.0 Appendix (Examples)

Example 1: Copy method

```
function ibm_data bist1sab_ctl_reg4::copy (ibm_data to = null) {
  bist1sab_ctl_reg4 cpy;

 if (to == null) {
  cpy = new();
 }
 else if (!cast_assign(cpy, to, CHECK)) {
   rvm_error(log, "Attempted to copy to an invalid type");
   copy = null;
   return;
 }
 cast_assign(cpy,super.copy(cpy));
 cpy.new_field = this.new_field;  // New field defined in this extended register class
 copy = cpy;
}
```

Example 2: Cast_assign with Enumerated types

```
enum mode_t  RESET=0, READ=1, WRITE=2, RUN=3, OTHER=4;
integer instruction;
mode_t  mode;
……

instruction = mode;              // No problem
mode = instruction;              //  Compile error
cast_assign(mode, instruction); // Safe as long as instruction only = 0:4
```

Example 3: Macro for binds

```
// Macro Define
#define SYS_FAIL_BIND_NAME_GEN(bist_type_short, one, bio_idx, bind_idx ) \
sys_fail_slowsysclk_/**/bist_type_short/**/_/**/one/**/_/**/bio_idx/**/_/**/bind/**/bind_idx \

// Macro Usage
bind_0["BIST1SAB_1_2"] = SYS_FAIL_BIND_NAME_GEN(1SAB,1,1,0);
```

Example 4: SmartQ.contains() in a constraint

```
// Enums declared globally in another file
enum inst_type_t = READ=3'b001, WRITE=3'b010, RUN=3'b011, FINISHED=3'b110,
                   OTHER=3'b111;
num inst_type_run_other_t = START_RUN, PAUSE_RUN, RESUME_RUN,
                            RELOAD_RUN;
// Sample of code for a READ Instruction Class using Smart Q in a constraint
class inst_read extends inst_reg {
  rand reg [3:0] in_array;
  reg[3:0] reg_addr_sq[$];  //  Smart Q of register addresses
  reg [3:0] sq_pick;
  // object_id is set by generator and represents the randomize count
  constraint in_sq_con {
      reg_addr _sq.contains(in_array) with object_id: (reg_addr_sq[object_id]);
  }
  task new();
   virtual task post_randomize();
   virtual function inst_type_t determine_next_type();
}
  // Avoid using constructor arguments for data objects to simplify extensions
  task inst_read::new() {
    current_type = READ; // defined in base class to be a "global" enumerated type
    load_list();  // Adds address of registers that need to be read (based on the BIST types
                  //  included) to the Smart Q
  }
  task inst_read::post_randomize() {
   remove_from_list(reg_addr);
   if (reg_addr_list == null) next_type =  determine_next_type();  // next_type is in base class
   else next_type = current_type;
  }
  function inst_type_t inst_read::determine_next_type() {
    inst_type_t new_type = OTHER; // default for expansion
    // last_type is set by the generator base on the last instruction type used prior to this one
    case (last_type) {
      WRITE: new_type = RUN;
       RUN: {
         if (cfg.num_runs_complete == cfg.num_runs_needed) new_type = FINISHED
         else new_type = OTHER;  // results in inst_type_run_other being used based on setting
                                 // by monitor, with default be START_RUN

       }
       default: {
          rvm_error(log, psprintf("Don't know what to do with last_type=%s",last_type));
        }
     }
   }
```

Example 5: Overriding properties

```
// Base Class A
class baseA {
   bist_reg  my_reg;  // Contains a bit flag called "active"
   task new() {
      my_reg = new();
   }
   virtual task update(bit value) {
       my_reg.active = value;
   }
}

// Extended Class B
class extendedB extends baseA {
   bist_reg_extended my_reg;

   task new() {
     super.new();
     my_reg = new();
   }
}

// Note that when an object declared as type baseA has its update method called, it will be
updating the properties in its copy of my_reg.  When baseA.update() is called, the extended copy
of my_reg will be updated.
```

Example 6: Complete (simplified) fail package program

```
#include <vera_defines.vrh>
// >>>>>>>These constraints apply when there is row and col redundancy
// Should have base class constraints for row repair
// and extend if col repair exists
class mem_cfg {
  rand bit[2:0] num_fails = 0;
  rand bit repairable;

  constraint repairable_con {
   //num_fails in {0:1} => repairable == 1; // If only 1 row repair allowed
   num_fails in {0:2} => repairable == 1; // If 1 row and 1 col can be repaired
   solve num_fails before repairable;
  }
  task display() {
   printf("num_fails %0h\n",num_fails);
```

```
    printf("repairable %0h\n",repairable);
  }
}

class fail_data {
 rand bit[1:0] bank;
 bit[3:0] decode;
 rand bit[11:0] row;
 bit[7:0]  col_bit;
 rand bit[11:0]  col_bit_decode; // Concatenation of col_bit and decode (to simplify constraints)
 rand bit  data_value;       // value to stick the fail to

 task display() {
   printf("bank is %0h\n",bank);
   printf("decode is %0h\n",decode);
   printf("row is %0h\n",row);
   printf("col_bit is %0h\n",col_bit);
   printf("data_value is %0d\n",data_value);
   printf("col_bit_decode is %0d\n",col_bit_decode);
 }
 task post_randomize() {
   decode = col_bit_decode[3:0];
   col_bit = col_bit_decode[7:0];
 }
}

// This results in unique fails
class fail_package {
 mem_cfg cfg;
 fail_data fail;
 rand fail_data fail_addr_sq[$];

 // Fail scenarios...
 // - Same Bank, all in a repaired row (repairable)
 // - Same Bank, all in a repaired col (repairable)
 // - Same Bank, all in either the repaired row or col (repairable) - Cross point interesting case
 // - Different banks (repairable)
 // - Same bank, more than 1 row repair and more than one col repair (non-repairable)
 constraint unique_con {
   foreach(fail_addr_sq,i) {
     foreach(fail_addr_sq, j) {
       (i != j) => {
         ((fail_addr_sq[i].bank    != fail_addr_sq[j].bank  ) ||
          (fail_addr_sq[i].row     != fail_addr_sq[j].row   ) ||
          (fail_addr_sq[i].col_bit_decode != fail_addr_sq[j].col_bit_decode));
        }
```

```
      }
    }
  }

  constraint repairable_con {
   if (cfg.repairable) {
     foreach(fail_addr_sq,i) {
       foreach(fail_addr_sq, j) {
         (i != j) => {
         // Different banks, or same bank and match rows or col
           fail_addr_sq[i].bank   != fail_addr_sq[j].bank ||
           (fail_addr_sq[i].bank == fail_addr_sq[j].bank &&
            ((fail_addr_sq[i].col_bit_decode  == fail_addr_sq[j].col_bit_decode) ||
             (fail_addr_sq[i].row  == fail_addr_sq[j].row)));
         }
       }
     }
   }
  }
  constraint non_repairable_con {
   if (!cfg.repairable) {
     foreach(fail_addr_sq,i) {
       foreach(fail_addr_sq, j) {
         (i != j && i<3 && j<3) => {
         // The first 3 fails don't intersect. After that, it doesn't matter.
           ((fail_addr_sq[i].bank == fail_addr_sq[j].bank) &&
            (fail_addr_sq[i].col_bit_decode  != fail_addr_sq[j].col_bit_decode) &&
            (fail_addr_sq[i].row    != fail_addr_sq[j].row));
         }
       }
     }
   }
  }

  task pre_randomize() {
   integer i;

   fail_addr_sq.delete(); // clear the queue
   for (i = 0; i<cfg.num_fails; i++) {
     fail = new();
     fail_addr_sq.push_front(fail);
   }
  }

  task display() {
   printf(">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>\n");
```

```
    cfg.display();
    foreach(fail_addr_sq,idx) {
      printf("fail_addr_sq[%0d]...\n",idx);
      fail_addr_sq[idx].display();
      printf("-------------\n");
    }
  }
}

program test
{
  integer i, status;
  mem_cfg cfg = new;
  fail_package dut = new;

  dut.cfg = cfg;
  for (i = 0; i<10; i++) {
    status = cfg.randomize();
    status = dut.randomize();
    dut.display();
  }

  printf("That's all folks\n");
}
```

Example 7: Factory Method for instruction with its random register

```
// Dumbed-down instruction class (A) is extended for a particular BIST and
// its registers are added to the smartQ.
class bist_reg {
  integer reg_addr;
  string name = "dummy";
  rand integer rand_var = -1;
  virtual task display(string prefix="") {
    printf("%s: reg name %s\n",prefix,name);
    printf("%s: reg addr %0d\n",prefix,reg_addr);
    printf("%s: rand_var %0d\n",prefix,rand_var);
    printf("----------\n");
  }
}
class myReg0 extends bist_reg {
  constraint addr_con {
   rand_var == 0;
  }
  task new() {
```

```
      super.new();
      name = "myReg0";
      reg_addr = 0;
  }
}
class myReg1 extends bist_reg {
  constraint addr_con {
   rand_var == 1;
  }
  task new() {
    super.new();
    name = "myReg1";
    reg_addr = 1;
  }
}

class A {
  rand bist_reg inst_reg_all; // Randomly picked from sq in pre_randomize
  rand bist_reg inst_reg_unused; // Randomly picked from sq in pre_randomize
  bist_reg all_regs_sq[$];
  bist_reg unused_regs_sq[$];

  task new() {
    if (unused_regs_sq.empty()) {
      FactoryMethod("unused");
    }
    if (all_regs_sq.empty()) {
      FactoryMethod("all");
    }
  }
  virtual task FactoryMethod(string kind="") {
    printf("ERROR: Must override FactoryMethod and fill in all_regs_sq\n");
    exit(1);
  }
  task pre_randomize() {
   inst_reg_unused = unused_regs_sq.pick();
   inst_reg_all = all_regs_sq.pick();
  }
  task post_randomize() {
    // Remove inst_reg from unused_regs_sq
  }

  task display() {
   inst_reg_unused.display("unused");
   inst_reg_all.display("all");
  }
```

```
}
class myA extends A {
  myReg0 reg0 = new();
  myReg1 reg1 = new();
  task new() {
    super.new();
  }
  virtual task FactoryMethod(string kind="") {
    printf("NOTE: Filling Queues with myA regs. kind=%s.\n",kind);
    case (kind) {
      "all": {
        all_regs_sq.push_front(reg0);
        all_regs_sq.push_front(reg1);
      }
      "unused": {
        unused_regs_sq.push_front(reg0);
        //unused_regs_sq.push_front(reg1);
      }
      default: {
        printf("ERROR: Unknown SmartQ kind: %s.\n,kind");
        exit(1);
      }
    }
  }
}

program test
{
  integer i, status;
  myA dut = new;


  for (i = 0; i<4; i++) {
    printf("NOTE: Randomizing instruction #%0d.\n",i);
    status = dut.randomize();
   // status = dut.inst_reg.randomize();
    dut.display();
  }

  printf("That's all folks\n");
}
```