

Exploiting the Power of Vera: Creating Useful Class Libraries

Janick Bergeron, VP Technical Wisdom

Dave Simmons, ASIC Implementation Specialist

Qualis Design Corporation

janick@qualis.com

ABSTRACT.

This paper illustrates the strategic process of creating and using Vera class libraries. Through well thought out class library development, System-on-a-Chip (SoC) and Intellectual Property (IP) block verification time can be greatly reduced. This paper specifically addresses:

- The importance of a sound verification methodology, and how class libraries can greatly reduce verification time/effort
- Strategic planning of classes and support routines based on domain analysis
- How to structure the libraries for ease of use
- Coding guidelines

Examples of well-designed libraries and how they are used in a testbench are provided.

1.0 Introduction

Vera can be easy to use and can provide considerable capabilities for accelerating the verification process. To reap these rewards, it is necessary to use sound software engineering principles. To those conversant in C++ programming, some of what follows will seem obvious. For those coming from an RTL background, there will be some new disciplines that need to be understood.

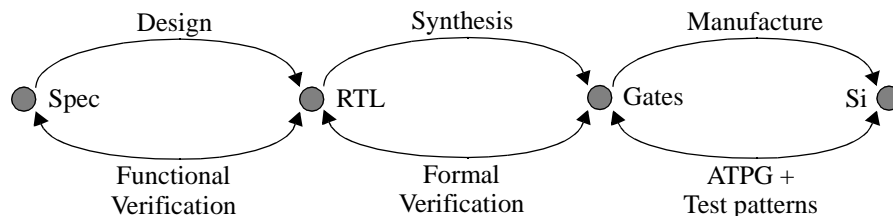
To a large extent, a verification project must be managed as a software project, imposing many of the same management and engineering requirements. Also, the methodology for object-oriented software design is somewhat different from what is required for the more conventional behavioral code typically found in VHDL or Verilog verification projects.

Before discussing the particulars of structuring classes and how using Vera can reduce the verification effort, this paper explains some general concepts regarding verification. First, this paper reviews the definition and purpose of functional verification. Next, the importance of planning the verification effort and the key elements of a verification plan are discussed. This foundation enables readers to appreciate the guidelines presented in the reusability, class structure, and coding sections that follow.

2.0 Functional Verification Process

In general, verification consists of verifying the result of a transformation against an original specification. Functional verification is a process that ensures a design - as coded in behavioral or RTL code, conforms to the design's specification. As shown in Figure 1, it is different from testability or formal verification. Testability and test patterns ensures that a design has been properly manufactured, while formal verification demonstrates that the synthesis tool is bug-free.

Figure 1.
Verification Processes



The design specification is the source of the verification effort. If there is no specification document, there is nothing to verify against. Furthermore, if the designer also performs the verification of the block he or she has designed, then the verification is accomplished against his or her own interpretation of the specification. The verification must be performed by individuals different from the designers.

The other difficulty resides in the subjective nature of “correctness” and what will be necessary to demonstrate it. From the design specification, it is not possible to make an

accurate estimate of the time to complete the verification of a design without first knowing:

- What needs to be tested
- What is necessary to provide stimulus
- How to check the response

From this, a determination can be made of how many testbenches are needed, how complex they are, and what associated infrastructure is required. Then, and only then, can a reasonable estimate be made of time to completion for the project, based on available resources in equipment and personnel.

The process of planning a verification project is the conscious definition of “*first-time success*” in the form of a verification plan. Every project aims for first-time success. Without a verification plan, the project progresses with no objective definition of when it is completed. Typically, verification continues until there is some consensus, usually based on intangible perceptions, that the design will work, or that schedule pressures, usually driven by marketing or economic considerations, force the design into fabrication in whatever state of verification (or the lack thereof) it happens to be at the time.

Planning the verification process and writing a test plan before proceeding with coding is very important. Except for pieces of code that are of immediately obvious value, no coding should begin until a test plan is available. Otherwise, it is a virtual certainty that much more time will be lost in developing code that will turn out not to be needed, restructuring code to fit actual needs, and in dealing with inefficiencies and poorly designed code.

This is true for any verification project, but it is even more critical for an object-oriented verification environment. If not planned correctly from the outset, it will not be possible to take advantage of the features offered by such an environment. The interdependencies of poorly-planned object-oriented structures can produce considerable code inefficiency and difficulty in debugging. A properly-planned and documented object-oriented structure can be easy to use, reusable, and robust.

To begin the planning process:

1. Outline what tests are needed for verification.
2. Outline what stimulus must be provided to apply the tests.
3. Outline how the response of the design will be checked.

The resulting outline determines what features are needed in the verification infrastructure. Then, these requirements can be mapped into a set of testbenches with a known level of complexity, which can then be assigned based on available personnel. At this point, enough information is available to define a verification schedule.

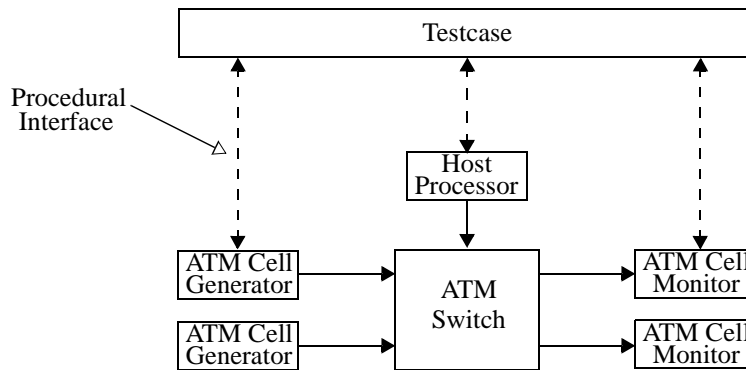
The design of the functional verification infrastructure will impact ease of writing and maintaining the testbenches. It has a direct influence on the time to completion for the project as it will be the most-leveraged code. To provide the shortest time to completion for the project, the design of the test facilities must be optimized through careful plan-

ning. The verification plan lets the verification infrastructure be optimized by maximizing concurrent development, reusing verification components across several testbenches, and minimizing functionality to bare essentials.

3.0 Verification Components

In high-level, transaction-based testbenches, transactors or bus functional models (BFMs) are written to stimulate and monitor the interfaces of the design. Every testcase interacts with the design-under-verification through these bus-functional models via a procedural interface. Figure 2 shows the structure of a testbench for an ATM switch. Read and write cycles from the host processors are initiated by the processor bus-functional model. ATM traffic is generated by the ATM cell generators while the ATM cell monitors sink the traffic coming out of the switch.

Figure 2.
Testbench structure for
an ATM switch.



The bus-functional models that surround the design under verification are used by all testcases that compose the verification suite, as defined by the verification plan. An intermediate level of test infrastructure, the *test harness*, containing or instantiating the design under verification, the bus-functional models, and any related utility functions should be constructed. The test harness contains all the connectivity information between the design and the bus-functional model and is reusable across all testbenches. Using a test harness also eliminates the need for each testbench to capture that information repeatedly.

In Verilog or VHDL, the bus-functional models and test harness would be implemented as additional levels of hierarchy in a module or architecture. In the object-oriented environment provided by VERA, they are implemented as objects. The following sections describe guidelines and techniques for properly and efficiently implementing these objects.

4.0 Reusability Guidelines

Engineers experienced in procedural programming often have the wrong impression when they are first introduced to object-oriented programming. At the surface, object-

oriented programming appears to be very verbose. It requires a lot of effort to properly declare and implement object types. At first glance, it is difficult to understand how this new approach to programming can be more efficient. If you measure efficiency as the total number of lines of code used to implement a complete verification suite, then an object-oriented implementation may very well prove to be no more efficient than a procedural one. A more suitable measure of efficiency is the time required to complete the testbenches, with a suitable degree of confidence in the correctness of the verification code. Object-oriented design helps on two fronts: implementations are more robust through abstraction and encapsulation, and promoting reuse of well-defined and verified base classes.

The *productivity gap* that motivates reuse on the design side is present in the verification effort as well. There is a greater amount of verification code than RTL code in use during a design project. It only makes sense that it be an even better candidate for reuse than design. Also, reusability may be more easily realized for verification code, since there are no implementation characteristics or constraints such as clock frequency.

4.1 Requirements for Reuse

The most significant factor for reusability is to select an architecture that promotes reuse. It should be possible to decouple the various elements of code so they can be easily reused in a different context. For instance, decouple bus-functional models from the testbench, decouple utility packages from the bus-functional models and from the testbench, etc. so that objects can function autonomously.

It is important to define the requirements of the verification objects. These requirements should be mapped in existing and new classes from the outset. The alternative is to develop haphazardly and implement functionality on a speculative basis, resulting in unneeded code and in application-specific features, resulting in rewriting of non-reusable code. Before designing and implementing the object classes, it is necessary to know exactly what functionality is required of each object and how they are going to interact.

4.2 Programming Practices

To obtain reusability, there are a number of programming practices to which code should conform.

4.2.1 Do not use global controls

In Verilog, global variables are the norm, so Verilog programmers may have to learn to resist the temptation to continue using global variables in Vera. Since using global constructs are to some extent endemic to hardware realizations, there is an innate tendency among hardware designers to use these constructs gratuitously.

For example, consider a bus-functional model providing a read operation. The address to be read could be passed as a globally-visible parameter, then a flag set to initiate the read operation. A *done* flag would be monitored to detect the completion of the opera-

tion and the data be available in another globally-visible parameter. Such structures and access protocols are common to RTL and hardware implementations. Once globally-visible variables and flags are externally accessed, changes to their implementation and usage will propagate unpredictably throughout all code that calls or uses them, thereby reducing reusability. Sample 1 shows an example of a poorly-design CRC object. It requires the user to properly initialize the CRC variable and the divisor, and pass to the computation function the variable that contains the computed CRC value. It is a clear indication of procedural thinking.

Sample 1.
Class design with poor
reusability.

```
class compute_crc {
    bit [8:0] divisor;
    function bit [7:0] update(bit [7:0] crc,
                             bit [7:0] dat);
}

program test {
    bit [7:0] payload[48];
    bit [7:0] my_crc;
    compute_crc crc = new;
    integer i;

    ...
    my_crc = 8'h00;
    crc.divisor = 9'b100000111;
    for (i = 0; i < 48; i++) {
        my_crc = crc.update(my_crc, payload[i]);
    }
    ...
}
```

4.2.2 Use procedural interfaces

The interface to an object should be entirely procedural. All data members should be local to the class implementation and not visible to the user. It should not require globally-visible variables to be assigned to configure or perform its operation. All state information should be kept internally. All accesses should be done through functions and tasks. Sample 2 shows the CRC computation class shown in Sample 1. This time, it is properly encapsulated, with all state information maintained within the object. To compute a different CRC value in parallel, simply instantiate to CRC computation objects.

Of course, the *compute_crc* class in Sample 2 is definitely not complete. Based on the requirements of the verification infrastructure, it requires a procedure to obtain the current value of the CRC, to compare it against a known value, and to randomly corrupt it.

In some circumstances, you may end up with an implementation composed of a series of *get* and *set* procedures for local data members. It may be an indication that the object was not designed with a procedural interface in mind. Try to look at the object in terms of the operations that it can perform, not its internal state. These operations will compose the procedural interface, modifying the state of the object internally.

Sample 2.
Class design with proper
reusability.

```
class compute_crc {
    local bit [7:0] crc = 8'h00;
    local bit [8:0] divisor = 9'b1000000111;

    task update(bit [7:0] dat);
}

program test {
    bit [7:0] payload[48];
    compute_crc crc = new;
    integer i;

    ...
    for (i = 0; i < 48; i++) {
        crc.update(payload[i]);
    }
    ...
}
```

4.2.3 Encapsulate data and subprograms within classes

Everything should be an object. Coming from a procedural programming background, it is tempting to continue to structure shared procedures as a collection of loosely correlated tasks and functions lumped into a few header files.

There are two problems with this approach:

- First, these procedures are all declared at the same level and “pollute” the name space.
- Second, there is no indication of any inter-relationships between related procedures.

The pollution of the name space prevents anyone from using the same name for a task, function, or variable. It may be acceptable for the author who has an idea of the naming convention and has a limited view of the application of these shared procedures. But if you are trying to integrate shared procedures implemented by two different authors with your own code, it is likely that name collisions will need to be resolved. Name collisions are almost guaranteed if the author has chosen meaningful but widely applicable identifiers¹. Sample 3 shows a poor packaging of a set of tasks used to provide consistent error reporting.

Procedures rarely exist on their own. They usually have companion procedures to perform complementary or similar operations. If all procedures are lumped into a few header files for the sake of convenience, it is not possible to determine which procedures are related to others. By putting related procedures into a single class, their relationship is made obvious. The encapsulation of related procedures into classes may also highlight some additional data hiding and object-oriented design opportunities. The class containers can still be lumped into the same number of header files, providing the same level of convenience for the users. Sample 4 shows the same error reporting pro-

1. I am personally annoyed by the VERA language designers who have used “ANY” and “ALL” and other meaningful - yet generic - identifiers for pre-defined global symbols, rendering them effectively reserved. *enums* should be local to classes!!!

Sample 3.
Poor packaging of
shared procedures.

```
In file syslog.vrh:
task error(string msg)
{
    ...
}

task warning(string msg)
{
    ...
}

In file test.vr:
#include "syslog.vrh"
program test {
    ...
    if (...) error("...");
    ...
}
```

cedures properly encapsulated into a class. This encapsulation creates the opportunity to introduce an error counter.

Sample 4.
Proper packaging of
shared procedures.

```
In file syslog.vr:
class syslog {
    local static integer err_cnt = 0;
    task error(string msg);
    task warning(string msg);
}

In file test.vr:
#include "syslog.vrh"
program test {
    syslog log = new;

    ...
    if (...) log.error("...");
    ...
}
```

Look for tasks or functions that take a class object as an argument. It is a good indication that this procedure would probably be better implemented as a method of that class.

4.2.4 Balance usability and reusability

The necessary compromises between usability and reusability should be made by design, by thinking the problem through to a solution, rather than by accident. Data members within classes should be hidden by making them local, as much as possible. If a data member is made public, its implementation and use cannot be changed once it is in use. For example, its type, its name, the format and timing of its information cannot be changed without requiring changes in any code that uses it.

The author of a class should resist the temptation of making a data member public simply to provide a quick solution to an access or configuration problem. This approach might be described as “design by accident”. This interface to each class should be carefully considered and designed. Similarly, making all data members and methods public at the outset might be called “accident by design”. Users will access anything they feel is useful and will misuse it. It is better to keep their prying fingers out of the detailed

implementation of an object, letting them access it through controlled interfaces, rather than endlessly debugging preventable misuses.

When upgrading or modifying class methods, consider the arguments to the methods, and use default arguments to provide backward compatibility. Add new arguments to a task or function at the end and provide default values that maintain the previous behavior. This default value will be used when the new version of the procedure is called by existing code, unaware that new arguments are available. This approach minimizes disruption to the users of a class. Sample 5 shows how a task can be modified without affecting an original user.

Sample 5.
Backward compatibility
of procedural interfaces.

Original implementation:

```
class syslog {
    task error(string msg);
}
```

```
syslog log = new;
```

New implementation:

```
class syslog {
    task new(string name = "");
    task error(string msg);
}
```

```
syslog log = new;
```

4.2.5 Plan class inheritance carefully

To maximize reusability, make use of and plan class inheritance carefully. Avoid the tendency to lump everything into one large class containing many unrelated data elements and methods for unrelated applications, or to duplicate identical information in separate classes. Start with a generic base class containing the basic functionality shared by all applications. Progressively derive subclasses as needed for specific applications, adding or overriding data members and methods. Structure the genealogy of the classes into thin layers, with any data members or methods shared by an application implemented as a single common derivative.

If class structure is properly planned in this manner, the effects of subsequent code modifications is reduced. Modifications required for a specific application or testbenches are made on an independent derivative and do not disturb existing code known to work. Changes can be made to lower-level classes then verified outside of the more complex context of the various applications. Bugs in lower-level classes found in one applications are automatically resolved for the other applications.

5.0 Guidelines for Structuring Classes

Proper structuring of classes is an art that must be based on experience. However, the optimum structure of class hierarchies often maps to the structure of the testbench and the data. One of the most obvious applications is in packet-based systems, where various packet flavors can be mapped to a genealogy of classes, and communication sys-

tems where the OSI protocol layer can also be mapped to a genealogy of classes. Whether an appropriate class hierarchy immediately suggests itself, some general guidelines apply.

5.1 Use Level of Controllability to Determine Layers

Based on the verification plan, determine what level of controllability of the bus-functional models is needed, then implement the layers accordingly. Start by creating a class implementing the basic operations that can be written and debugged autonomously. Then add more complex operations as derivative objects, creating layers of procedural interfaces. Locate related functions in the appropriate layer, corresponding to the proper level of controllability. For instance, access to physical bits needs to be located in a class at lowest layer of the hierarchy, with system control functions defined in a higher layer. Users instantiate the object, using the derivative that offers the required level of functionality, with finer levels of controllability available through the lower classes.

For example, a base ATM cell class would provide all required data fields, data packing, and HEC computation facilities. Generating random content in the ATM cell would be provided by a derived class. Similarly, providing random traffic patterns with some specific VPI and VCI distributions would be implemented in a secondary derivative, as illustrated in Sample 6.

Sample 6.
Class genealogy for
ATM cells.

```
class atm_cell {
    rand bit [3:0] gfc;
    rand bit [7:0] vpi;
    ...
    task do_pack();
    function bit [7:0] compute_hec(bit bad = 0);
}

class random_atm_cell extends atm_cell {
    task new();
    task post_randomize();
}

class shaped_traffic_atm_cell extends random_atm_cell {
    task new();
    task pre_randomize();
    task post_randomize();
    task distribution(...);
}
```

5.2 Do Not Develop Unneeded Code

Another tendency to avoid is developing code without being sure that it is necessary. Once class development begins, it is tempting to add functionality just because it is interesting or attractive. Develop only what is immediately needed, plan for and add additional functionality later. Use default arguments to implement new functionality while maintaining backward compatibility. Also, beware of developing large amounts of code without testing it, or without making sure that what is being developed is applicable to the testing that will be performed. Implementing large amounts of code in its entirety without intermediate testing produces a lot of rewriting.

5.3 Develop Basic Application Testbench Early

The process of developing the class hierarchy involves a certain amount of successive approximation or tuning toward a solution. Develop basic application testbenches as early as possible. They will be used to help prove the class implementation approach, maintain conformity in the class structures and identify restructuring needs.

5.4 Build Classes in a Strictly Bottom-Up Fashion

Classes should be built on top of each other, not intertwined. If two classes need to interact closely (see the list example below), they should be declared and implemented in a single file. This communicates their close relationship clearly and indicates that they must be verified as a whole.

If two classes, declared and implemented in separate files, each include objects of the other class, it complicates maintenance and compilation. First, one of the header file must be generated manually. Second, it is not possible to verify the implementation of a class separately from the other. Sample 7 shows an example of how classes can be poorly interlocked. The *syslog* class contains a private static data member containing the list of all *syslog* object instance. Every time a new *syslog* object is allocated, a handle to its instance is added to the list. Notice how the *list* class also makes use of the *syslog* class.

Sample 7.
Improper interlocking of
unrelated classes.

```
In syslog.vr:
#include "list.vrh"
class syslog extends list_el {
  static local list all_logs;
  task new() {
    all_logs.append(this);
  }
}

In list.vr:
#include "syslog.vrh"
class list_el {
  ...
}

class list {
  local syslog log;
}
```

A better implementation will stick to the strict bottom-up implementation guideline. Even though the *list* object has the proper functionality for the application, the *syslog* class will have to implement the list of all *syslog* object instances using VERA primitive. Sample 8 shows an example using an associative array.

5.5 Overload Methods at Each Layer

Do not create identical or similar methods with different names when deriving classes. Instead, overload common methods such as “print” or “display” at each layer while pro-

Sample 8.
Untangling unrelated
classes.

In `syslog.vr`:

```
class syslog {
    static local integer log_cnt = 0;
    static local syslog all_logs[];
    task new() {
        all_logs[log_cnt++] = this;
    }
}
```

In `list.vr`:

```
#include "syslog.vrh"
class list {
    local syslog log;
}
```

gressively deriving classes, using the names “print” or “display” at each layer. This lets a user display the content of an object without knowing exactly what type it is so that the user can call the proper method.

Overloading of methods can produce severe debugging problems, most of which can be avoided by adhering to some useful conventions. Once a method at one layer of a class structure is overloaded, it should be subsequently overloaded in all derived classes. If no functional changes are made, simply invoke the method in the parent class using the *super.* prefix. Don’t skip overloading anywhere in the sequence of derived classes. If there are gaps in the sequence, it may be difficult to determine the origin of the base method being overloaded. By overloading at each layer, it will be necessary to examine only the next layer up to find the method that is being inherited at the current layer.

Because they provide complementary capabilities, virtual methods and overloading of methods should be planned together.

5.6 Use Defensive Programming Techniques

Standard defensive programming techniques should be applied to programming in an object-oriented environment as well. Use error detection code generously. Check for invalid conditions, states that should not be reached, misuse, invalid arguments or combinations thereof.

5.7 Fix The Automatically Generated Header

Vera is capable of automatically generating the header file that contains the class definition of all classes implemented in a source file. Users of these classes only need to include that header file to use the class definitions within it.

However, there is a hidden pitfall with this last capability. Vera will not put in the generated header file any “#include” statements present in the original source file and required by the class definition. Sample 10 shows an example of an automatically generated header file from the source code shown in Sample 9. Anyone attempting to

include this header file would encounter a syntax error because the definition for the type *syslog* is lacking.

Sample 9.
Class definition using
another externally
defined class.

```
#include "syslog.vrh"
class list {
    syslog log;
    ...
}
```

Sample 10.
Generated header file
for *list* class.

```
////////////////////////////////////
// Vera Header file created from list.vr
////////////////////////////////////

#ifndef INC__TMP_LIST_VRH
#define INC__TMP_LIST_VRH

extern class list {
    syslog log;
    ...
}
#endif
```

One solution would be to manually include the file *syslog.vrh* whenever the automatically-generated header file is included. But this violates the most basic data hiding principle: as a user, I should not need to know the details of a class implementation nor the required ancillary files to compile it. Furthermore, if the class definition is modified, requiring the inclusion of another file, all other source files using this class will also need to be modified to include this additional file.

A simple solution is to fix the generated header file by including any “#include” directive found in the original source file. This can be easily accomplished through a simple PERL script¹. An example of the properly modified generated header file can be found in Sample 11. This fix can be easily automated by using the following makefile target:

```
%.vro %.vrh: *.vr
    vera -cmp -h $*.vr && vrhfix $*.vrh
```

Sample 11.
Fixed header file for *list*
class.

```
////////////////////////////////////
// Vera Header file created from list.vr
////////////////////////////////////

#ifndef INC__TMP_LIST_VRH
#define INC__TMP_LIST_VRH

// Added by vrhfix:
#include "syslog.vrh"

extern class list {
    syslog log;
    ...
}
#endif
```

1. The *vrhfix* script can be found in the *resources* section of:
<http://janick.bergeron.com/wtb>

5.8 Include Utility Methods

When designing classes, there will be some “utility” methods that should be routinely included in every class. These will vary somewhat from one implementation to another, but any class that contains or operates on a data object normally contains “print” or “display” methods. Packet-oriented classes may routinely contain packing and integrity-check methods. Recognizing the necessity for such utility methods and methodically deriving and overlaying them makes coding testbenches easier and improves their uniformity.

6.0 Documentation

Because so much of its benefits are derived from reuse, object-oriented code generally requires more documentation than more traditional code. Class inheritance, correct use of class methods, and the manner in which class methods access class data elements, can be difficult to determine by analyzing code, sometimes even for the person who wrote the code. Those writing testbenches may be only vaguely familiar with the object-oriented structures that they will be using. The verification project will proceed much faster if it is not necessary for the engineers to analyze the class definitions before understanding how to use them.

Promoting the implementation of class structures with private data members and limited public methods carries with it the necessity for fully documenting the public methods that the classes provide. This is particularly the case in object-oriented applications where source code for class structures is not necessarily available to applications programmers. Private methods and data elements should be documented in a separate implementation document.

7.0 A Complete Example

In this section, I will describe the implementation of a set of reusable classes, from a low-level message logging utility class, through a general-purpose list class, derived ATM cell classes, Utopia bus-functional models classes, and finally, a test harness object class to be reused by all testbenches on the ATM switch design illustrated in Figure 2.

Because the focus of this paper is on class usefulness and reusability, the examples are only described using their public interfaces. All implementation details, once properly verified, are functionally irrelevant to a class usefulness and reusability. Some implementation details may be presented to illustrate a specific point or technique.

7.1 *syslog* Class

The *syslog* class, shown in Sample 12, is used to report messages in a consistent format, to ease parsing of the simulation result and determine if the simulation was successful or not. To be able to manage the logger object, from any location in the Vera code, it can be

controlled by name. A task is provided to identify when and if error or warning messages are expected. Their absence would be reported as an error and a failure of the testcase.

Sample 12.
syslog class.

```
class syslog {
    //
    // Message management
    //
    // Some management methods can be applied to a named logger.
    // The specified name must match a known logger.
    // The name "this" indicates this logger.
    // The name "all" indicates all known loggers.
    //
    // For example, to have all loggers report their status:
    //     log.report("all");
    //
    task new(string name);

    task expect(integer n_warns = 0,           // Expect # of warns
               integer n_errs = 0,         // and # of errs
               string  which  = "this");   // on this named logger
    task set_debug(bit on_off = 0,          // Turn dbg msg on/off
                  string  which  = "this"); // on this named logger
    task report(string which = "this");     // Report status

    //
    // Issue messages
    //
    task debug  (string msg);
    task note   (string msg);
    task warning(string msg);
    task error  (string msg);
    task fatal  (string msg);
}
```

7.2 list Class

Vera does not have a built-in list type. A generic list class can be created to provide this most useful construct. Because Vera does not have a generic object handle like *void** in C++, it is not possible to write a list object that can handle any type of content. To work around that problem, a base class called *list_el* is defined. Any object that you want to be able to put on a list must be derived from this base class. Since Vera does not support multiple inheritance, you have to make sure that all base classes are derived from the *list_el* class. Through polymorphism, subsequently derived classes will be automatically put into a list object. Since objects coming out of a list will come out as *list_el*'s, it will be necessary to use the *cast_assign()* function to promote the *list_el* object back to its original class. Fortunately, run-time checking is performed to ensure that objects are promoted to a valid higher-level class.

This approach has one limitation: an object can only be put on a single list at any given time. Proper error checking in the list class implementation can detect a violation of this constraint, as shown in Sample 14. In the absence of *friend* classes, as in C++, it is sometimes necessary to make some methods or data members public to give the necessary functionality and control to a cooperating class. It is the case with the *list_el::insert*

method. It is intended to be used by the *list* class only, but there are no mechanisms, other than documentation, to prevent it.

The concept of the iterator is a nice illustration of the object-oriented design principle. Instead of relying on the user to properly maintain the current state of an index variable to iterate over all elements of a list, it is encapsulated in the list itself. Iterating using an external index would also have required random access capabilities into the list, something that may not be easy to implement. Sample 15 shows an example of using a list iterator.

Notice how the *list_el* and *list* classes contain synchronization events. They can be used to synchronize with significant events occurring on the list. One mistake, stemming from an RTL coding mindset, is to use polling to synchronize with a particular event as shown in Sample 16. It is very inefficient because it consumes simulation resources while waiting and requires the concept of a time interval, which may not be available if you do not directly interact with the HDL under verification. Using the built-in asynchronous synchronization primitives are more efficient. Sample 17 shows an example of a single server listening to multiple queues.

Sample 13.
list class.

```
#include "syslog.vrh"

//
// Base type for all objects that must be put on a list.
// Use cast_assign() to promote this base class back to the
// appropriate class.
//
typedef class list;

class list_el {
    // Event to sync with the element being taken off a list.
    // This event is triggered ONE_SHOT: all sync() will be
    // resumed when triggered.
    //
    event unlisted;

    task new();

    // Walk along the list
    function list_el next_in_list();    // Go toward the end
    function list_el prev_in_list();  // Go toward the beginning

    // Modify the list around this element
    task remove_from_list();          // Remove this from list
    task append_in_list(list_el el);  // Insert after this el
    task prepend_in_list(list_el el); // Insert before this el

    // Get the list this element is in
    function list is_in_list();

    // Insert into the specified list
    // This method should not be used directly
    task insert(list_el next_el,
               list_el prev_el,
               list  in_list);
}
```

A Complete Example

Sample 13.
list class (cont'd).

```
//  
// The list object itself  
//  
class list {  
    // Events to synchronize with the list becoming (non) empty.  
    // These events are triggered ON/OFF: sync() will not wait if  
    // the list is already (non) empty.  
    // e.g. sync(ANY, l.not_empty);  
    //  
    event not_empty;  
    event is_empty;  
  
    task new(string name = "named list");  
  
    function integer length(); // What is its length?  
  
    //  
    // The list functions follow the PERL convention  
    //  
    task    push(list_el el); // Add at the end  
    function list_el pop(); // Remove from the end  
    task    unshift(list_el el); // Add the the front  
    function list_el shift(); // Removed from the front  
  
    //  
    // List iterator  
    //  
    function list_el first(); // Reset the iterator to beginning  
    function list_el last(); // Reset the iterator to end  
    function list_el curr(); // Current pos of the iterator  
    function list_el next(); // Advance the iterator forward  
    function list_el prev(); // Advance the iterator backward  
}
```

Sample 14.
Constraint violation
detection in *list* class.

```
//  
// Add an element at the end of the list  
//  
task list::push(list_el el)  
{  
    // Verify that the new element  
    // is not already on a list  
    if (el.is_in_list() != null) {  
        this.log.fatal("Attempt to push el already on a list");  
        return;  
    }  
    ...  
}
```

Sample 15.
Using the iterator in the
list class.

```
#include "list.vrh"
#include "atm_cell.vrh"

program test {
    uni_atm_cell cell;
    list          cell_list = new("list of cells");

    ...
    for (void = cell_list.first();
         cell_list.curr() != null;
         void = cell_list.next()) {
        cast_assign(cell_list.curr(), cell);
    }
    ...
}
```

Sample 16.
Polling to synchronize
to an event.

```
#include "list.vrh"

program test {
    uni_atm_cell cell;
    list          cell_list = new("list of cells");

    ...
    while (cell_list.length() == 0) {
        @ (posedge ...);
    }
    ...
}
```

Sample 17.
Asynchronous
synchronization to an
event in a multiple list
server.

```
#include <vera_defines.vrh>
#include "list.vrh"
#include "atm_cell.vrh"

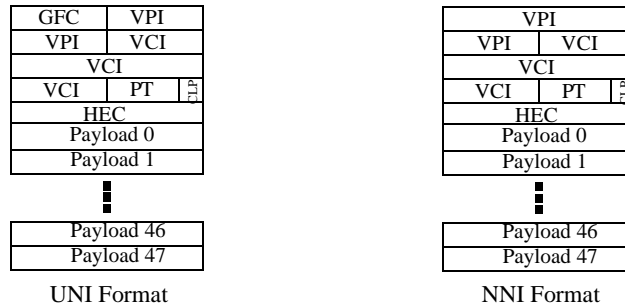
program test {
    uni_atm_cell cell;
    list          cell_list[4]; // 4 separate cell queues

    ...
    sync(ANY, cell_list[0].not_empty,
         cell_list[1].not_empty,
         cell_list[2].not_empty,
         cell_list[3].not_empty);
    // Randomly serve a non-empty queue,
    // giving more weight to the longest queue
    randcase {
    cell_list[0].length(): ...
    cell_list[1].length(): ...
    cell_list[2].length(): ...
    cell_list[3].length(): ...
    }
}
```

7.3 atm_cell Classes

The *atm_cell* classes are a perfect example of derived classes. ATM cells come in two formats: UNI and NNI. Figure 3 shows the structure of the UNI and NNI cells. Most fields are identical except for the VPI and GFC fields. The fields common to both formats are implemented in an *atm_cell* base class. In Sample 18, this class is declared as *virtual* because in and of itself it is not a complete object: a virtual class **cannot** be used

Figure 3.
ATM cell formats



by itself. Only a derived class can be used. The *atm_cell* class is also derived from the *list_el* class to be able to create lists of *atm_cell* objects. To automatically create cells with random content, all data fields are tagged with the *rand* attribute. A new cell can be initialized with non-random content by providing explicit values for the required fields in the *atm_cell::new* task. The default value of -1 indicates that a random content is desired. The HEC field is not randomized because it is a computed field based on the header content. The *atm_cell::compute_hec* method is used to set its value to a known good or bad value. It is invoked by the *atm_cell::post_randomize*¹ task to make sure a new cell instance is valid.

The *uni_atm_cell* and *nni_atm_cell* classes are derived from the *atm_cell* class. They provide the additional data members unique to each cell format. Because they inherit the *list_el* class derivative, they can also be put on lists. Because the *uni_atm_cell* and *nni_atm_cell* classes are closely related through a single base class, it is possible to write a model in terms of the *atm_cell* class. This model would not know (or care) whether it is currently processing a UNI or NNI formatted cell. Either object type would simply look like an *atm_cell* object. The objects could later be promoted back to the proper *uni_atm_cell* or *nni_atm_cell* class for format-specific processing.

To allow a model to process *uni_atm_cell* and *nni_atm_cell* objects as *atm_cell* objects without knowing which specific class it is, *virtual* methods are provided. A *virtual* method, when called from the base class, actually invokes the implementation in the derived class. For example, the *atm_cell::display* task, which is virtual, will actually invoke the *uni_atm_cell::display* task or the *nni_atm_cell::display* task, depending on when the particular instance of *atm_cell* gets displayed. The virtual tasks in the derived class can then invoke the virtual task in the base class using the *super* qualifier, as shown in Sample 19.

The built-in *pack* function cannot be used to pack the various header fields into a compact array-of-bytes representation. First of all, it is not virtual and would not pack the fields unique to each cell format when invoked from the base class' perspective. Second, it packs the class content in a strict bottom-up process. What is required is a top-down packing process because the unique fields are located first in the cell format.

1. Even though this method is not intended to be public, it is a predefined method and cannot be made private.

Sample 18.
Derived classes and
virtual methods:
atm_cell class.

```
#include "list.vrh"

//
// Base ATM cell object
//
virtual class atm_cell extends list_el {
    rand bit [15:0] vci;
    rand bit [ 2:0] pt;
    rand bit      clp;
    bit [ 7:0] hec;
    rand bit [ 7:0] payload [48];

    bit [7:0] packed_header [5];

    task new(integer vci = -1,
            integer pt  = -1,
            integer clp = -1);

    // Compute or corrupt the HEC
    function bit [7:0] compute_hec(bit bad = 0);

    virtual task display();

    virtual task pack_header();

    task post_randomize();
}

//
// UNI-format ATM cell
//
class uni_atm_cell extends atm_cell {
    rand bit [3:0] gfc;
    rand bit [7:0] vpi;

    task new(integer gfc = -1,
            integer vpi = -1,
            integer vci = -1,
            integer pt  = -1,
            integer clp = -1);

    task pack_header();

    task display();
}

//
// NNI-format ATM cell
//
class nni_atm_cell extends atm_cell {
    rand bit [11:0] vpi;

    task new(integer vpi = -1,
            integer vci = -1,
            integer pt  = -1,
            integer clp = -1);

    task pack_header();

    task display();
}
```

Sample 19.
Implementation of a
derived *virtual* method.

```
//  
// Display the UNI-format portion of the cell  
//  
task uni_atm_cell::display()  
{  
    printf("UNI ATM cell:\n");  
    printf("   gfc = 3'b%b\n", this.gfc);  
    printf("   vpi = 8'h%h\n", this.vpi);  
    // Display the common portion  
    super.display();  
}  
  
//  
// Display the NNI-format portion of the cell  
//  
task nni_atm_cell::display()  
{  
    printf("NNI ATM cell:\n");  
    printf("   vpi = 12'h%h\n", this.vpi);  
    // Display the common portion  
    super.display();  
}
```

Packing is accomplished in a fashion similar to displaying, using virtual methods, in a top-down fashion, as shown in Sample 20.

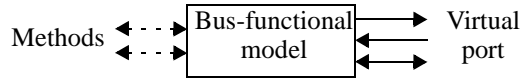
Sample 20.
Implementation of a
virtual packing method.

```
//  
// Pack the header of the cell (virtual)  
//  
task atm_cell::pack_header()  
{  
    bit [7:0] tmp;  
  
    // The GFC/VPI will be packed by the derived uni/nni class  
    tmp = packed_header[1];  
    '{tmp[3:0], packed_header[2], packed_header[3],  
    packed_header[4]} = {vci, pt, clp, hec};  
    packed_header[1] = tmp;  
}  
  
//  
// Pack the UNI-format portion of the header  
//  
task uni_atm_cell::pack_header()  
{  
    '{packed_header[0], packed_header[1]} = {gfc, vpi, 4'b0000};  
    super.pack_header();  
}  
  
//  
// Pack the NNI-format portion of the header  
//  
task nni_atm_cell::pack_header()  
{  
    '{packed_header[0], packed_header[1]} = {vpi, 4'b0000};  
    super.pack_header();  
}
```

7.4 Utopia Bus-Functional Model Classes

Bus-functional models can be pictured, as in Figure 4, as having a procedural interface on one side, and pins on the other. This model maps directly with Vera's class methods and virtual ports. The methods provide the procedural interface, while the virtual port provides the connectivity to the design under verification. Sample 21 shows a class implementing a bus-functional model for a physical-layer Utopia transmitter.

Figure 4.
Bus-functional model
interfaces



Sample 21.
Implementation of a
bus-functional model
using a class.

```

//
// Virtual port for a PHY->ATM Utopia interface
//
port utopia_rx_port
{
    data;          // [7:0] data                (PHY->ATM)
    soc;          // Start-of-cell marker        (PHY->ATM)
    enb;         // Active-low byte enable                    (ATM->PHY)
    empty_clav;  // Cell/octet flow control                  (PHY->ATM)
    clk;        // All signals sampled at rising edge      (ATM->PHY)
}

#include "atm_cell.vrh"

//
// Symbolic constants
//
enum block_typ {NON_BLOCK=0, BLOCKING=1};

//
// Utopia PHY->ATM transactor object
//
class utopia_phy_tx {
    task new(utopia_rx_port tx_port,          // Tx on this port
            string name = "unamed Utopia PHY Tx");

    // Send a cell
    task send(atm_cell cell,                  // The cell to send
             block_typ blocking = BLOCKING); // Blocking or not
}
  
```

The virtual port used to interface with the design under verification is first declared. Because the syntax of virtual ports does not specify the direction, it is important that they be documented clearly. The bus-functional model class is “connected” to a specifying binding of the virtual port when it is instantiated through the *new* task. The binding is stored internally. Thus, each instance remains permanently connected to the specified port.

The testbench generates operations on the virtual port by using the *send* method to send a cell. It can be invoked in either blocking or non-blocking mode. In blocking mode, the *send* task returns only when the specified cell has been transmitted to the ATM layer device. In non-blocking mode, the cell is queued for transmission in an internal list and

the *send* task returns immediately. Sample 22 shows an example using this bus-functional model. The cells are physically transmitted from the internal list by an autonomous process, analogous to an *always* block, detached from the main execution thread in the *new* task using the *join none* statement, as shown in Sample 23.

Sample 22.
Using a bus-functional
model object.

```
#include "utopia_phy_tx.vrh"
...

program test {
  uni_atm_cell      cell;
  utopia_phy_tx atm_gen;
  integer          i;

  atm_gen = new(rx_port_bind);
  // Transmit 10 valid random cells
  for (i = 0; i < 10; i++) {
    cell = new;
    atm_gen.send(cell, BLOCKING);
  }
  // Send a bad cell
  cell = new;
  cell.hec = cell.compute_hec(1);
  atm_gen.send(cell);
  // Follow with 10 more cells with VPI = 10
  for (i = 0; i < 10; i++) {
    cell = new(*, 10);
    atm_gen.send(cell, BLOCKING);
  }
  ...
}
```

Sample 23.
Detaching an
autonomous process.

```
class utopia_phy_tx {
  ...
  local task tx_daemon();
}

task utopia_phy_tx::new(...)
{
  ...
  // Start the daemons
  fork
    this.tx_daemon();
  join none;
}
```

7.5 Test Harness Class

Once this basic infrastructure has been implemented, it is possible to create a test harness object that will be used by all testcases. The test harness encapsulates in an object all the information that does not change from testcase to testcase, from testbench to testbench.

Each testbench instantiates the test harness and has direct access to the public procedural interface methods provided by the bus-functional model objects in the harness. The harness hides the tedious details of how each bus-functional model is connected to the design under verification. Should the physical interface of the design change, the

test harness can likely be modified without affecting any of the testbenches. Sample 24 shows the implementation of a test harness as illustrated in Figure 2.

Sample 24.
Test harness object.

```
#include "utopia_phy_tx.vrh"
#include "utopia_phy_rx.vrh"
#include "host_cpu.vrh"
#include "syslog.vrh"
...

class harness {
    syslog log;           // Message logger
    utopia_phy_tx tx[2]; // ATM cells -> design
    utopia_phy_rx rx[2]; // design -> ATM cells
    host_cpu cpu;        // Host processor

    task new();
}

task harness::new()
{
    log = new("ATM switch harness");
    tx[0] = new(rx_port_0_bind, "ATM gen #0");
    tx[1] = new(rx_port_1_bind, "ATM gen #1");
    rx[0] = new(tx_port_0_bind, "ATM mon #0");
    rx[1] = new(tx_port_1_bind, "ATM mon #1");
    cpu = new(host_cpu_bind, "Host");
}
```

8.0 Summary

When you use a different tool, you must adapt your methodology, techniques and practices. If you use the same back-and-forth movement when using a chainsaw as when using a handsaw, you will not observe the increase in productivity that the salesperson promised. The issue is the same with EDA tools. The tool is not a solution in and of itself. You must adapt your methodology, your thinking patterns, and your behavior to make the best use of the tool.

Verification languages offer a great promise of productivity and efficiency for functional verification. But to achieve these new levels of productivity, it will be necessary to adopt some of the recommendations outlined in this paper. Object-oriented coding is *different* from procedural or RTL coding. It offers new opportunities, but only through careful planning and consideration for reuse.

Functional verification languages and methodologies are still in their infancy. They are at the same point as RTL coding was in 1990. Early adopters are defining coding styles, approaches, and processes that facilitate the verification task in the long run and produce predictable results. But unlike RTL coding, object-oriented verification methodologies can inspire themselves from the many years of experience in the software industry. I hope that this paper was successful in introducing some of that object-oriented software engineering experience in the field of hardware verification.