Where OOP Falls Short of Hardware Verification Needs

Matan Vax
Cadence Design Systems, Inc.
8 Hamelacha St., Rosh Ha'ain, 48091, Israel
matan@cadence.com

ABSTRACT

Functional verification requires, among other things, dedicated programming constructs and mechanisms. Such are accessible to a wide community of verification engineers today more than in the past thanks to the SystemVerilog language. Along with many verification specific constructs it features object-oriented programming (OOP) framework. OOP has been extremely successful in facilitating reuse in many software application domains. This observation suggests that it should serve just as well for reuse in the verification domain. The present paper critically examines this supposition. It demonstrates issues with the naive application of object-oriented mechanisms, and how they are partially overcome by more sophisticated design techniques. Still these techniques don't scale well and increase code complexity. The same needs are shown to be met more naturally with modularization mechanisms that go beyond strict OOP. This analysis leads to an interesting observation on the nature of reuse in verification. It turns out to differ substantially from that of classical software application domains. If hardware verification languages are to address the challenges of creating reusable verification code, they must accommodate for its unique character.

Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Design Aids—*Verification*; D.3.3 [**Programming Languages**]: Language Constructs and Features – *Classes and Objects, Patterns*.

General Terms

Languages, Verification.

Keywords

Object-Oriented Design, Hardware Verification Languages.

1. INTRODUCTION

Simulation-based functional verification of hardware designs[1] has established itself in recent years as an engineering discipline in its own right. It has its own dedicated languages, tool set, methodologies, library eco-systems, and body of knowledge. The turning point in the evolution of the discipline was the adoption of coverage-driven methods in conjunction with constrained-random stimuli generation [11, 16]. Replacing traditional approaches that were either strictly directed or fully randomized, it deals more systematically with the explosion of simulation state and stimuli spaces. As the discipline matures, reuse of libraries and verification IPs becomes ubiquitous and crucial for the success of projects.

With the adoption of coverage-driven methods existing programming languages were no longer sufficient for testbench development, as they lacked the means to express constraints on stimuli generation and functional coverage definitions. So languages dedicated for the task emerged – hardware verification languages (HVLs). The most widely used public domain HVLs today are ***e*** [6], SystemVerilog [7], and SystemC/SCV [13].[2] All three languages support a rich set of verification specific constructs to express constraints, coverage points, temporal properties, hardware interfaces, and more. However, they differ in the basic modularization mechanisms they provide.

Developing testbenches for hardware devices is in many senses a software engineering task. In order to exercise the design under test (DUT) and check its behavior the testbench needs to emulate the behavior of the environment in which the device is embedded, as well as the device's own operation, at various levels of abstraction. This essentially involves models, algorithms, and data structures naturally expressed in high-level programming languages. Advanced testbenches are, in this sense, large scale software systems.

Much of the code of a testbench, if built well, may live for years and be reused across the hardware project's proliferations, various different projects, and even different companies. A different and more elementary dimension of reuse in verification is that between the numerous tests in a single project's test suite [5]. This kind of reuse has no obvious counterpart in general software systems, but is somewhat reminiscent of the variability found in software product lines. In any case, verification engineers nowadays must draw upon state-of-the-art software engineering methods to meet the challenge of effective code reuse.

The dominant programming paradigm in the industrial software world is object-oriented programming. OOP makes use of the notion of class as the primary unit of

[1] Henceforth I shall use the unqualified term 'verification' in this sense.

[2] These languages are not exclusively used for verification. SystemVerilog is also a hardware design language, and SystemC is primarily used for hardware modeling. SystemC is referred to as a language, although it is actually a C++ library compiled with standard C++ compilers. Vera and OpenVera are still used in the industry, but are proclaimed to be superseded by SystemVerilog. There are many proprietary verification languages and language extensions mostly developed in-house by chip makers.

modularization, and relies on class inheritance as its main abstraction mechanism. In so far as verification is ultimately about writing software, it is only natural to expect that OOP would serve just as well for verification needs. Indeed this expectation is echoed in all three leading languages.

However, in practice even with object-oriented mechanisms in place the design of testbenches does not resemble all that much mainstream software design. This is manifested in the extensive use of certain design patterns and auxiliary design constructions. Similar techniques are applied in other domains, but never in such sweeping manner. This phenomenon was observed before but the present paper proposes a generalized analysis for it under the title – the layering problem. The main contribution of the paper is in suggesting what it is about the task of verification that does not lend itself naturally to classic object-oriented design. The answer has to do with the unique form that code reuse must take in this domain.

The remainder of the paper is organized as follows: Section 2 presents a design situation encountered regularly in verification, known as "layering", and explains why it does not map directly to object-oriented constructs. Section 3 surveys and evaluates techniques to address layering within OOP. Section 4 shows how the required relationship can be captured directly with a different set of language constructs. Section 5 builds on previous observations and tries to address the main question framed above. Work related to this kind analysis is surveyed in section 6, and section 7 concludes.

2. THE LAYERING PROBLEM

The constrained-random verification flow is essentially about running a multitude of tests, each focusing on some specific feature, condition, or aspect of the DUT. There may be hundreds or even thousands of such tests in a regression suite, and each test is executed multiple times with different random seeds. The verification environment is the common infrastructure for test execution for a given DUT, or simply put, the code that is shared between all tests.

2.1. Layering Constraints

It would be instructive to open the exposition of the problem through an example. Let us consider a testbench for a switching device that interfaces through some packet protocol. Here is a simplistic definition of packet data type in SystemVerilog:[3]

```
class packet;
   rand bit[15:0] addr;
   rand bit[7:0]  data[];
   constraint addr_range {addr >= 'h10;}
endclass
```

This class has two randomized member variables (fields): a 16 bit address and an array of data bytes. It also has one member constraint restricting the range of values for variable *addr*. The verification environment should provide the signal level logic for actually driving a packet into the DUT, as in:

```
class packet_env;
   ...
   task drive_packet(packet p);
      // write packet onto DUT signals
   endtask
endclass
```

Consider a very simple test on top of this environment that drives 20 packets having lower-end addresses into the DUT. Here is its code:

```
class low_addr_packet extends packet;
   constraint low_addr { addr <= 'hFF; }
endclass

program test1;
   packet_env env;

   initial begin
      env = new();
      random_scenario();
   end

   task random_scenario();
      int i;
      for (i = 0; i < 20; i++) begin
         low_addr_packet p = new();
         assert(p.randomize());
         env.drive_packet(p);
      end
   endtask
endprogram
```

Now imagine another test that drives 20 upper-end address packets into the DUT, and yet another that drives random packets with addresses distributed in some other way. These tests share the base packet definition and the driving logic. However, they have in common also the scenario code that generates 20 packets, with the sole difference being the additional constraint. Unfortunately this code cannot be shared between them because the `new` operator considers only the static type of its operand. If the code of task *random_scenrio* above had used class *packet* instead of *low_addr_packet* randomization would have ignored *lower_addr* constraint. Note that using inline constraints (the `randomize with` operator) may obviate the subclasses of *packet*, but the recurrence of generation code in all three tests would still remain.

In this trivial example the loop that generates and drives packets is obviously not worth the trouble of factoring out and reusing. But it so happens that the structure of legitimate transaction streams may involve some of the most intricate logic of the protocol. Ideally most tests would reuse this logic and entirely abstract from it. When constructing a testbench for a device that interacts through a USB port, for example, the environment may wish to hide the complex packet handshaking flows while letting the test merely affect the data packets delivered. But this is rendered impossible using the language's constructs in this naive manner.

[3] For simplicity, all code examples in this section and the next are in SystemVerilog. The same issues arise in SystemC/C++ and ***e*** so far as standard object-oriented inheritance is used.

2.2. Layering Behavior

The above situation is often referred to as the *layering* problem – the test writer needs to layer some property on top of the environment's data type. This problem is by no means restricted to constraints. To illustrate, imagine that in the above testbench some legality check is implemented for packets and is invoked after a packet has been read off the DUT interface. Here is this aspect of the environment:

```
class packet;
   ...
   virtual function void check_length();
      // perform check
   endfunction
endclass

class packet_env;
   ...
   task monitor();
      packet p;
      forever begin
         collect_packet(p);
         p.check_length();
         ...
      end
   endtask

   task collect_packet(output packet p);
      p = new();
      // read packet off DUT signals
  endtask
endclass
```

If a test writer needs to refine the check in some way, she cannot expect to achieve this just by overriding function *check_length* in a derived class. This is again because task *collect_packet* allocates packets by *new*-ing the base type. Editing the code of class *packet_env* may not be an option, as the source code for this part of the environment is not necessarily available (as in the case of commercial IPs, for example). But even if it were, the decoupling of test and environment would be sacrificed. Inheriting *packet_env* and overriding *collect_packet* is impractical for reasons that shall be discussed next. Note also that non trivial behavioral extensions often require auxiliary member variables and functions and may affect more than one class.

2.3. Layering on top of Components

So far we focused on layering properties or behavior on top of data item classes provided by the verification environment. However, most checks and coverage points are defined outside the data model, and so are some aspects of stimuli such as timing and error injection. Here too individual tests may need to intervene with the environment's operation in certain respects to hit corner cases or observe specific DUT behavior correctly. The naive object-oriented approach would be again to subclass the environment's classes that handle these tasks and make the required adaptations. But in itself this move is futile, because just like in the case of the data model, instantiation of these classes is something the test cannot take over. Here is why.

The main tasks of the environment are decomposed into separate objects classified by their function. Different methodologies differ in details and terminology, but are alike in spirit with respect to this classification. They name entities such as monitors, drivers, sequencers, collectors, loggers, generators, etc. These are composed together to form agents or transactors, and multiple agents are further grouped in module and system environments. All these objects function as software components in the sense that they are created and configured during environment setup and live throughout the simulation. Other kinds of components, such as scoreboards, abstract scenario libraries, register file models, and behavioral "golden" models, may also be instantiated in the same environment and interconnected.

The architecture of a large scale testbench may become rather complex. Fortunately, it remains constant (perhaps with minor variations) for the entire test suite since it corresponds to the structure of the DUT. It is unrealistic to expect that a test would construct and configure the entire component hierarchy for itself just to be able to control the actual type of some specific monitor or driver. After all, this structure is exactly the kind of non-trivial definition that should be abstracted from and reused. So for purposes of test writing, inheritance of component classes in its simple form is out of the question.

Finally it should be stressed that the layering problem is not restricted to tests. Standalone module-level environments can be reused as verification IPs by integrating them into different system environments. The large scale environment will typically need to customize the verification IP and adapt it to its needs. This is done in much the same way as tests would, since the IP logic that needs to be reused involves instantiation of its own object model. Packaged IP code is usually not open for integrators to edit even if they wanted, and they rarely do.

3. DESIGN PATTERNS FOR LAYERING

3.1. Layering with Factories

In object-oriented languages virtual method calls are bound dynamically, that is, dispatched based on the actual type of the receiver object. The `new` operator, on the other hand, determines the type of the object in the first place and so must be bound statically with some class. The way to work around this is seemingly straightforward – use another object for the instantiation and leverage its polymorphic behavior. This auxiliary class is known as a *factory*. In this way objects can be created in multiple places using an abstract factory interface, while the actual class being thus instantiated is determined by the type of a concrete factory object. This is actually an old technique that was named a design pattern in the famous Gang-of-Four book [3].

If we apply the pattern to the example in section 2.1 above, we would need to replace all explicit uses of the `new` operator with calls to a virtual method *create()* of an abstract factory class. The test would then only have to instantiate a concrete factory once, and it in turn will affect the behavior of the entire environment. Leaving out much of the detail, the environment code may now look thus:

```
class packet_factory;
   virtual function packet create();
      // by default create a packet
      create = new();
   endfunction
endclass

class packet_env;
   packet_factory pfact;
   ...
      packet p = pfact.create();
      ...
      packet p = pfact.create();
   ...
endclass
```

This way layering additional properties or refined behavior on top of the basic packet definition does not require any intrusive editing of the environment code. A test can simply define a factory subclass and register its instance with the environment class, as follows:

```
class test1_packet_factory
                 extends packet_factory;
   virtual function packet create();
      low_addr_packet p = new();
      return p;
   endfunction
endclass

program test1;
   packet_env env;

   initial begin;
      // factory registration
      test1_packet_factory pf;
      pf = new();
      env.pfact = pf;
      ...
   end
endprogram
```

A utility library can reduce the coding overhead of the abstract factory pattern for both environment developer and test writer. The OVM-SV library, for example, includes a very sophisticated version of the Abstract Factory pattern [10]. It involves global type registry and bookkeeping code encapsulated in base classes and in preprocessor macros. So the environment developer needs to derive from a common base class, and use a factory method for creation supplied by the library rather then `new`. The test writer simply needs to call a library function to "override" all allocations of some class with that of a subclass. Neither test nor environment need to explicitly define or instantiate a factory class.

3.2. Limitations of Factories

The use of factories solves the layering problem in its simple form. However, often when tests are defined as refinement layers on top of an existing object model, useful tests may be obtained by composing two or more such refinements. This cannot be done with the Abstract Factory pattern, at least not in a single inheritance language such as SystemVerilog.

Imagine a test with another variant of *packet*, call it *round_addr_packet*, which gives more weight to 4-byte aligned addresses. It may make sense to apply this property also in conjunction with the *low_addr* property defined above to create a third test. But since a different class is used to define each of the variants and both derive from *packet* they cannot be applied together to the same instances. Conversely, if *round_addr_packet* explicitly derives from *low_addr_packet* it cannot be applied separately.

With multiple inheritance, as in SystemC (C++), this kind of reuse can be achieved by deriving a third class from both subclasses of *packet*. This way each test could choose between the base class, one of the two subclasses, or the third class that combines the two. However, this still requires explicit definition of the combination as a separate class. So the two definitions are not truly pluggable independently in a test. The situation gets worse with more independent properties, of which arbitrary subsets may be used in different tests. The number of explicit classes that need to be created and maintained grows exponentially with the number of such independent properties.

A related requirement on the part of test writers has to do with leveraging existing abstractions of the data model. Tests may need to layer a property on top of an abstract class so as to affect all its subclasses equally. Consider a protocol that defines several variants of its basic data item which have some properties in common and others not. For example, in an environment for USB interface it would be natural to define *data* packet, *token* packet, and *handshake* packet, as separate classes, all deriving from an abstract packet class. Now some tests may need to layer an additional constraint on top of all three subclasses. Ideally the new property would be associated with the common base class, rather than duplicated for each concrete class. The test writer in this case need not even be aware of the subclasses. But even if factories have been used to instantiate these classes throughout the environment, deriving a new class from the base class would not help. The new class does not "blend in" with any of the existing.

Here too multiple inheritance can be used to overcome the reuse issue. But this is, again, a very lean sense of reuse. It is achieved at the price of explicitly crossing the new property with every existing subclass. The number of synthetic classes generated by this cross may be large, causing significant coding and maintenance overhead. Moreover, the test writer cannot remain agnostic of the concrete data model.

3.3. Behavioral Extensions with Callbacks

We turn now to discuss another well known object-oriented technique to tackle the layering problem. It involves using "listeners" or callback objects, and is akin to Command and Observer design patterns [3]. We shall use the term 'callback', as it is more commonly used in this context.

The principle is simple. The environment declares an abstract class with one or more virtual functions that are called at designated points in the execution. The classic observer is intended to monitor state changes within some model, but any kind of occurrence can be monitored just as well. The key here is that a single occurrence may trigger the notification of multiple callbacks. In its simplest form, the idea may be implemented in the environment thus:

```
class env_callback;
   virtual task execute();
   endtask // does nothing by default
endclass

class packet_env;
   // callback queue
   env_callback pre_drive_cbs[$];

   task drive_packet(packet p);
      // invoke all registered callbacks
      for (int i = 0;
           i < pre_drive_cbs.size(); i++)
         pre_drive_cbs[i].execute();

      ... // do the actual driving
   endtask
endclass
```

Here the environment triggers all callbacks registered in *pre_driver_cbs* queue right before actually driving a packet into the DUT. A test that needs to modify the driving behavior in this sense can simply define its own callback object and register it. The following test utilizes the above preparation to force random delay before each packet:

```
class my_delay_cb extends env_callback;
   virtual task execute();
      int unsigned delay;
      delay = $urandom_range(10);
      #(delay); // wait 0..10 time units
   endtask
endclass

program test1;
   packet_env env;
   my_delay_cb my_cb;

   initial begin
      // ...

      // regiester callback with the env
      my_cb = new();
      env.pre_drive_cbs.push_back(my_cb);
   end
endprogram
```

Different observer classes (and queues) may be used to encapsulate different function signatures. Conversely, several callback functions may be grouped together in a single abstract callback class to encapsulate the handling of related events.

Just like in the case the Abstract Factory pattern, it is worthwhile to encapsulate pattern related services in a utility library. The Verification Methodology Manual (VMM), for example, recommends a very wide application of callbacks, and provides related services encapsulated in library base classes and macros [1, 15]. These handle registration, unregistration, and the actual invocation of callback objects.

3.4. Advantages and Limitation of Callbacks

First it should be noted that callback objects can only implement behavioral extensions. If structural properties need to be added to an existing class, such as constraints, fields, or new methods, other techniques must be applied, such as factories.[4]

Callbacks require upfront design of extension points and are only as good as the preparation they have in the environment code. Unlike factories, they cannot be used to apply unanticipated modifications to a class. In particular, callbacks cannot completely override existing functionality, unless explicit preparation for this has been made. It could be argued that this restriction is actually an advantage, as it enforces more disciplined usage. Perhaps this position is valid in other domains, but often turns out to be too rigid for real life situations in verification (see a case for this claim in [12]). A related issue with callbacks is the fact that they encapsulate the extension code in a separate class, while it is often tightly coupled with the extended class.

On the other hand, callbacks are much more flexible with respect to layer composition. Any number of independent callback objects can be registered to monitor the same event, as long as it has the required preparation. In fact, callbacks are independent in a way that does not allow one to affect another, even when this is needed. Further techniques can be applied to implement more sophisticated notification semantics. In particular, Chain of Responsibility pattern [3] can be used to enable control flow variability between the calls. With more flexibility comes more complication of flow, which impedes readability and understandability of the code.

4. FROM DESIGN PATTERNS TO NATIVE CONSTRUCTS

Limitations of specific design patterns can usually be overcome with yet more sophisticated techniques. But the one inherent down side of design patterns is their very existence in the code. They put both cognitive and coding overhead on the programmer, even when reduced to the minimum using utility libraries, and have negative impact on traceability and understandability of the code [2]. The compiler and other development tools remain ignorant of the design intent expressed in these terms. The pain grows with density and variety of patterns applied. It seems that with advanced object-oriented verification methodologies this becomes a serious issue.

Gamma et al. readily admitted that design patterns are a matter of point of view – "One person's pattern can be another person's primitive building block" [3]. The difference between two such persons may simply be the programming

[4] It should be s,//tressed that neither this section, nor the rest of the paper, attempts a comparison of OVM and VMM. Both include more concepts and services that bear on the considerations raised here. The intention is merely to evaluate the two design techniques *per se* with respect to the needs outlined previously.

language they use. Bosch writes "... design patterns are part of the software engineer’s paradigm and it is the task of the programming language to represent the concepts in the paradigm as accurate as possible" [2]. Gil and Lorentz call design patters "puppy language features" and argue that in many cases they compensate in one language for capabilities that already exist in other languages [4]. This is evident, for example, with aspect-oriented languages such as AspectJ that have the Observer collaboration (as in the callback pattern) as their prototypical application [9].

The ***e*** language has unconventional features with direct bearing on the design requirements described above. This should come as no surprise, as they were devised from the outset to tackle exactly these challenges. They are inheritance-like mechanisms on top of the standard single inheritance – *class extensions* and *"when" subtyping*. Both have been presented previously in [6]. Their affinity to key aspect-oriented concepts, namely advice and introductions, has been analyzed in [12, 14]. In this chapter we shall explain very briefly these two mechanisms, leaving details out, and demonstrate their application to the layering problem. The main purpose of this is to substantiate the claim that design techniques discussed above indeed compensate for limited expressive power of strict object-oriented languages.[5]

4.1. Class Extensions

In ***e*** a class is initially declared with the `struct` or `unit` keyword,[6] optionally deriving from another class in the standard object-oriented sense. Within the scope of the class' initial definition new members can be defined, and existing members can be overridden.

Having been initially defined in this way, a class may be extended elsewhere using the keyword `extend`. In the scope of this construct the class may be further defined in exactly the same manner – new members can be added and existing members can be overridden "in-place" so to speak. New members and new definitions to existing members affect all instances of that class, including classes inherited from it, just as if they were part of its initial definition. This is regardless of where the instantiation of (or inheritance from) the class occurs in the code.

Overriding existing definition in the strict sense is often not required. Subclasses inherited in the standard way, and more so in-place class extensions, often need to augment existing definitions rather than replace them altogether. For this purpose ***e*** makes available two additive modes of member overriding, or more aptly – *refinement*. They place the new definitions after or before the exiting definitions, and are denoted by the key phrases `is also` and `is first` respectively. The standard object-oriented replacement semantics is denoted by `is only`.

[5] This is not to claim that ***e***'s are the only conceivable mechanisms for that purpose, nor even that they are the best ones.

[6] The difference between the two corresponds to a distinction made previously between data types and components. For present purposes both keywords prefix class definitions.

4.2. Applying Class Extensions in Layering

The constraint layering example from section 2.1 above could be rewritten in ***e*** such that a generic random scenario is defined in the environment using the `new` operator directly. The ***e*** equivalent of the environment's definition for class *packet* would be:

```
struct packet {
   addr: uint(bits:16);
   data: list of byte;
   keep addr_range is { addr >= 0x10; }
};
```

The code of *test1* would consist simply of an extension to class *packet* with the additional constraint thus:

```
extend packet {
   keep addr < 0xFF;
};
```

The reader can appreciate the way the `extend` construct expresses the intention of the test in a concise and intuitive way without the need for auxiliary definitions. If defining the same class in multiple locations should seem strange at first glance, it may be just because we disregard the fact that this is in fact what we try to achieve here, and what the patterns discussed above try to emulate.

Moreover, the limitations listed in section 3.2 do not plague this mechanism. Consider another property defined in some other test, for example:

```
extend packet {
   keep data.size() in [2,4,8];
};
```

The two properties thus defined could be loaded together in the same session and would then apply equally to all generated packets, be it within the environment code or outside. Note that this would have been true even if packet was an abstract class in the environment and only subclasses of it were actually generated.

Now consider the example from section 3.3. The same approach would apply here just as well. The test only needs to extend class *packet_env* and refine time-consuming method *drive_packet* (***e***'s equivalent of SystemVerilog's member task):

```
extend packet_env {
   drive_packet() @clock is first {
      var delay: int[0..10]
      gen delay;
      wait [delay];
   };
};
```

If a number of refinements for the same method are compiled or loaded in the same session they all get executed upon every call. The order of execution is determined according to dependency between the modules (source files). The effect thus achieved is similar to that of the use of callbacks. But no up-front preparation in the base environment is required and the coding overhead is gone.

4.3. Taking it Further with *When* Subtypes

A *when* subtype is simply a class under some condition. Assume that class *packet* has a field (member variable) named *kind* of an enumerated type, whose possible values are *token*, *data*, and *handshake*. In ***e*** these values imply the existence of 3 subtypes of packet, namely *token packet*, *data packet*, and *handshake packet*. An instance of class *packet* with field *kind* equals to *token* is actually an instance of the type *token packet*. The phrase `token packet` represents a type for all purposes – declaring variables, casting, etc. *When* subtypes may be extended just like any other class type, making the definitions in the extension scope conditional. They apply to an object in as far as it is an instance of that subtype, that is, it is an instance of the base class and its corresponding field is of the right value at that time.

When subtypes are ideal for data modeling where different values of a field correspond to structural differences of the data item in which it is embedded. This is often the case in hardware protocols and processor instruction sets. More importantly, with *when* subtypes it is possible to constrain and randomize the type of data items.

In the context of the present discussion *when* subtypes are a way to introduce independent sub-classifications of objects and thus gain finer control over extensions. A constraint may be put on *data packet* without affecting other kinds of packets. Imagine another field of *packet* indicating whether it is legal or corrupt. This field would expand a sub-classification on an orthogonal dimension. So under *corrupt packet* there could be a refined implementation of some method, and it would apply to all corrupt packets, regardless of whether they are data packets or of other kinds.

This use for *when* subtypes is very common also in customizing components. Bus agents, for example, are typically classified into masters and slaves, a classification that may affect both their structure and behavior. From the testbench point of view some may be active in the sense that they actually inject synthetic traffic, while others passive, that is, merely monitoring traffic. These classifications are naturally captured as *when* subtype extensions to class *agent*. Any given agent is either a master or a slave, and independently either active or passive. In all four cases all and only the relevant definitions apply.

5. DISCUSSION: THE REUSE MODEL IN VERIFICATION

The object-oriented programming paradigm emerged in the 1970's with the development of Smalltalk. One of its first and most distinctive applications was in the design of a graphical user interface (GUI) framework [8]. The language mechanisms forged for this purpose, most notably class inheritance and polymorphism, were extremely successful in capturing the commonalities between library entities, and in supplying useful abstractions to the end application.

Let us consider the mode of reuse in the context of this classical application for OOP. Reuse here means first and foremost factoring the code-intense definitions of graphical elements into a library so that they can easily be shared between many different applications. Applications define their custom GUI by specializing, instantiating, and configuring components, such as buttons, panes, menus etc. These are placed within a window, and behavior is associated with events reported by them. The library provides a pure object model for these components while instantiation is left exclusively to the application.

What in verification is analogous to this classic relation between library and end-application? Where does the main "reuse divide" pass? In a single verification project much of logic is common to all tests in a regression suite. Its complexity is typically greater than that of any specific test, and so is the effort that goes into its development. But a verification environment is very different in nature from a class library, just as tests are very different from GUI applications.

Section 2 above already outlined what a verification environment should consist of. It is *not* merely a set of abstract building blocks for writing test. Rather, it is a *concrete* and *fully operational* model simulating in great details relevant aspects of the environment in which the device is embedded. If built properly a verification environment could, at least in principle, execute as-is (or almost as-is) achieving real coverage and possibly uncovering bugs.[7] It obviously makes no sense to expect something of the sort from the average class library.

Similarly, individual tests, when distilled to their essentials, don't resemble standalone applications in any way. Tests may need to configure some properties of the environment, perhaps make slight adaptations to its behavior. But most commonly tests use constraints and procedural code to guide random stimuli into the scenario's areas of interest. In this sense tests are merely variations on the environment's definitions. If fact, test may involve a number of independent variations combined together, where variations themselves may be shared across some (but not all) tests. Interestingly, this mode of reuse is not restricted to tests. It is often manifested also when integrating verification IPs in system-level environments.

At the heart of what we identified as the layering problem lies the fact that the model defined by the environment is concrete. Unlike the case with class libraries, the verification environment does much, if not all, of the actual instantiation of classes it defines. Generation of data item streams for stimulating the DUT obviously requires instantiation, and so does collection of DUT responses for checking. The construction of environment's component architecture, which may become fairly complex, also involves instantiation of objects.

There are two basic tasks that a software system needs to handle – the definition of a model and its instantiation. The object-oriented paradigm focuses on modularization of the model's definition *prior* to its instantiation, and indeed this is where potential for reuse lies in many application domains. But in verification the balance between the two tasks is different from a reuse point of view. Much of the logic of a

[7] In fact, some commercial verification IPs do just that and even claim to cover significant portions of the verification goals relative to a given plan.

verification environment involves instantiation, while the tests typically just need refine some aspect of the model. This mode of reuse constitutes a different programming paradigm. As quoted previously, it is the task of the programming language to represent the concepts in the paradigm as accurate as possible.

6. RELATED WORK

Hollander et al. [5] survey the unique mechanisms for separation of concerns of the ***e*** language, and their relation to the specific challenges encountered in verification. The present paper is very much following the lead of hints in their work, in particular in its focus on the special relation between a verification environments and individual tests in a test suite. However, in their paper Hollander et al. do not discuss possible solutions within the bounds of object-oriented programming, and their limitations. The present paper attempts exactly this analysis. Here ***e***'s mechanisms are mentioned mostly to substantiate the criticism of object-oriented languages in this respect.

Robinson's book [12] is strongly related to issues discussed in the present paper. Apart from being a valuable book for practitioners, it contains some insights on the very idea of aspects and AOP. Unlike the present paper, it stresses the general problem of crosscutting concerns and how AOP promotes better modularization of them. Robinson is surely right in that, being large software systems, testbenches have crosscutting concerns. He does not stress enough, though, the special traits of the verification domain that make these AOP-like mechanisms *indispensable*. Unlike Robinson, the present paper focuses on problems of reuse in verification that are not germane to most other domains.

7. CONCLUSION

This paper presented a software design situation that is ubiquitous when building reusable verification environments. It concerns layering properties and behavior on top of a model without owning its instantiation. This design situation does not map naturally to native object-oriented constructs. Design patterns and techniques are applied across the board by verification methodologies to overcome this shortcoming, but these techniques are inherently cumbersome and not wholly satisfactory. Language mechanisms that go beyond the strict object-oriented paradigm do a much better job in capturing the same design intent. In fact, the object-oriented techniques seem to emulate these very mechanisms.

What makes the programming paradigm that works so well for most complex software systems not right for verification? Some insight can be gained by comparing the mode of reuse in classical object-oriented application domains and that of verification environments. In a nutshell, object models make very good candidates for reuse, but construction and generation logic does not. The latter is extremely important in verification, and less so in other domains.

The upshot of this discussion is twofold. It reinforces the claim that simulation-based verification is a unique kind of challenge, not reducible in its methods to other engineering disciplines. From the point of view of language design, there is a lesson to be learned too. A domain specific language is not all about domain specific types, operators, or even constructs. It may well need a suitable programming paradigm to supply the right toolbox for modularization, abstraction, and reuse.

8. ACKNOWLEDGEMENTS

I wish to thank Brett Lammers, Shlomi Uziel, and Zeev Kirshenbaum of Cadence, and JL Gray of Verilab, who provided valuable feedback on earlier drafts of this paper.

9. REFERENCES

1. Bergeron, J., Cerny, E., Hunter, A., and Nightingale, A. 2005 *Verification Methodology Manual for Systemverilog*. Springer-Verlag New York, Inc.

2. Bosch, J. 1998. Design Patterns as Language Constructs. In *Journal of Object-Oriented Programming*, vol 11, 18-32.

3. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995 *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc.

4. Gil, J. and Lorenz, D. H., 1998, Design patterns vs. language design. In *Proceedings of the 11 th European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, vol. 1241, 9-13.

5. Hollander, Y., Morley, M., and Noy, A. 2001. The e language: A fresh separation of concerns. In Proceedings of the International Conference on Technology of Object-Oriented Languages and Systems (TOOLS 2001 Europe Conference, Zurich, Switzerland).

6. IEEE Standard for the Functional Verification Language 'e', IEEE Computer Society, IEEE, New York, NY, IEEE Std 1647—2006

7. IEEE Standard For System Verilog - Unified Hardware Design, Specification and Verification Language, IEEE Computer Society, IEEE, New York, NY, IEEE Std 1800—2005

8. Kay, A. C. 1993. The early history of Smalltalk. In *the Second ACM SIGPLAN Conference on History of Programming Languages* (Cambridge, Massachusetts, United States, April 20 - 23, 1993). HOPL-II. ACM, New York, NY, 69-95.

9. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. 2001. An Overview of AspectJ. In Proceedings of the 15th European Conference on Object-Oriented Programming (June 18 - 22, 2001). J. L. Knudsen, Ed. Lecture Notes In Computer Science, vol. 2072. Springer-Verlag, London, 327-353.

10. OVM web site. http://ovmworld.org

11. Piziali, A. 2007 *Functional Verification Coverage Measurement and Analysis*. 1st. Springer Publishing Company, Incorporated.

12. Robinson, D. 2007, *Aspect-Oriented Programming with the e Verification Language: A Pragmatic Guide for Testbench Developers*. Elsevier Inc.

13. SystemC web site. http://www.systemc.org

14. Vax, M. 2007. Conservative aspect-orientated programming with the *e* language. In *Proceedings of the 6th international Conference on Aspect-Oriented Software Development* (Vancouver, British Columbia, Canada, March 12 - 16, 2007). AOSD '07, vol. 208.

15. VMM web site. http://vmm-sv.org

16. Wile, B., Goss, J., and Roesner, W. 2005 *Comprehensive Functional Verification: the Complete Industry Cycle (Systems on Silicon)*. Morgan Kaufmann Publishers Inc.