# A Scalable Method of Propagating Event Messages Throughout A Vera Testbench

David Robinson and Jason Sprott

Verilab Ltd.

**Abstract**: Managing the flow of event messages between disparate classes of a testbench (RTL monitors, BFMs, simulation control classes) is a complex process, and one that can prove impossible if left until late in the design cycle. While ad hoc communication interfaces are simple enough to develop between classes, problems arise when classes that were not designed to communicate with each other suddenly have to. This paper presents a scalable method for propagating both blocking and nonblocking event messages throughout a testbench. The technique is based on a software Design Pattern solution, called the Observer, to which modifications have been made. Although Vera implementation details are provided, including several techniques to work around current Vera limitations, the technique is applicable to other object-oriented verification languages.

## 1 Introduction

One problem in testbench design is the propagation of event messages throughout the testbench. Event messages, referred to as events from herein (not to be confused with the built-in Vera events), can be generated by any number of disparate sources, existing at any of the testbench abstraction levels. Classes that monitor the RTL may send events describing the current operation of the RTL; bus functional models (BFMs) may send messages informing other objects of what their current operation is; and watchdogs may send events warning that the simulation is terminating because of a timeout. Consumers of events may decide to stall the operation of the sender (blocking events) until certain actions have been taken, or they may yield control immediately (nonblocking events), and schedule their event response for a later time [1]. With so many potential sources and consumers of events, it can be difficult to ensure that they can be routed to the correct destination, and that program flow proceeds correctly.

During testbench conception, it is common to design ad hoc communication interfaces between classes to distribute the required event messages. For instance, if, when a read occurs, class A decodes bus transaction information from the RTL and sends an event R to class B, then it is easy to create a communication interface between these two classes for this purpose. However, problems may arise later in the project, when it is discovered that the required events and their distribution predicted at the start of the project are inadequate. This may be due to requirement changes, or a lack of visibility, common during the design phase. To continue the above example: later in the project (as deadlines loom), class C now requires access to event R, and class B requires access to event E, which was not originally intended to be a propagated event, but would now be useful. In addition, when C receives event R, it needs to stall the operation of A. However, B must receive event R as it occurs, not at some later time when C is finished with it.

Making these modifications may be impossible given the time scales and code structure, but even if they can be made, they will probably be quick fixes that introduce problems as they break the class hierarchy. These quick fixes often include code duplication, which leads to both efficiency and maintainability problems.

This paper presents a solution to the event propagation problem. It makes use of the Observer design pattern [2], to which enhancements have been made to allow any number of event generators to send any number of events to any number of event consumers. The approach also allows the event consumers to selectively and dynamically register for specific events, and it handles both blocking and nonblocking events. A Vera implementation of the pattern is provided, and the paper discusses practical implementation problems and offers possible solutions.

The rest of this paper is structured as follows: Section 2 introduces design patterns and the Observer Pattern. Section 3 discusses how the Observer Pattern can be of use in testbenches, and gives two examples of its usage. It also compares the Observer Pattern events with the event type built into Vera. Section 4 covers the implementation details of the Observer Pattern in Vera, and section 5 shows how to use the Observer Pattern in a testbench. Our conclusions are presented in section 6.

## 2 Design Patterns and The Observer Pattern

Design Patterns are described as "*... simple and elegant solutions to specific problems in object-oriented software design*" [2]. Simply put, they are descriptions of tried and tested solutions to common programming problems. One commonly used design pattern is the Observer Pattern which has the following intent: "Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically". That is, when something important happens in an object (the Subject) that other objects (the Observers) depend on, the Subject will inform the Observers without knowing at compile time what these Observers are, or how many there are. The design pattern was originally developed just to let others know if the Subjects state changed, but extensions allow the object under observation to specify exactly what changed.

Figure 2-1 shows the UML [3] diagram for the Observer Pattern. The Subject and Observer classes are abstract base classes and must be overridden using inheritance. The Subject base class contains references to all of the Observers that have registered for notification when the Subject's state changes. These references are set by the ConcreteObservers calling the Subject's attach() task. State changes in the ConcreteSubject are propagated by the ConcreteSubject calling the notify() task of its base class (Subject). This then iterates round its list of observers and calls their update() tasks. Due to inheritance, it is the update tasks of the ConcreteObservers that are actually called. These can then interrogate the ConcreteSubject to get its new state. This approach uses the data pull mechanism, where no data is sent to the Observer. Instead, data is pulled from the ConcreteSubject to determine what happened. If a ConcreteObserver no longer wishes to be notified of state changes, it calls the Subject's detach() task.

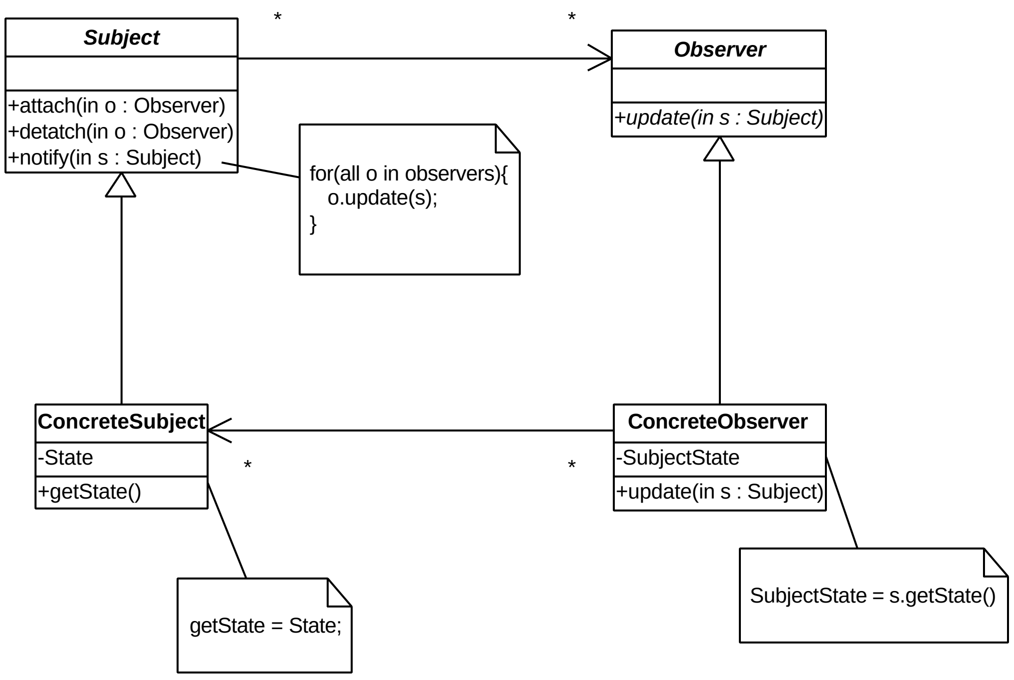

Figure 2-1: UML diagram for Observer Pattern

To see how the Observer Pattern can be of use in verification, it is beneficial to map some of the pattern's terminology to terminology more suited to the verification domain. This can be seen in Table 2-1.

| Pattern Terminology | Testbench Terminology | Description |
|---|---|---|
| State | Event | An occurrence of a condition that is of interest to part of the testbench |
| Subject | Event Generator | Any object that generates events and needs to propagate these into the test-bench. The Event Generators may be monitoring the RTL and sending events when certain conditions occur, or may be other testbench components such as BFMs or scoreboards |
| Observer | Event Consumer | Any object that needs to know when an event occurred |

Table 2-1: Mapping the Observer Pattern terminology to the verification realm

The Observer Pattern is a useful way of automatically propagating changes in objects to dependent objects. However, as it stands, it is not entirely suitable for use as an event generator because it only supports one event - "the subject has changed". An event generator has the following requirements:

| | |
|---|---|
| ➢ No limit on the number of event types: | The Observer Pattern only supports one event - the subject has changed |
| ➢ No limit on the number of sources: | The consumer can only differentiate between different subjects if the language allows down casting from a Subject base class to the original concrete class. Vera does support this |
| ➢ No limit on the number of consumers: | The Observer Pattern supports this |
| ➢ Selective and dynamic event registration: | The Observer Pattern supports this for the one event it does have. Consumers can subscribe or unsubscribe from notifications at any time |
| ➢ Support for blocking and nonblocking events | The Observer Pattern makes no distinction between these types of events, and only one type can be supported. Which type depends on the designer's implementation |

To be suitable for use in testbenches, the Observer Pattern needs to be enhanced to support multiple events, and to support both blocking and nonblocking events.

## 3 Motivation for Using the Observer Pattern in Testbenches

Before describing in detail an implementation of the Observer Pattern suitable for testbenches, this section demonstrates some scenarios where the Observer Pattern can be used in a testbench. These examples are based on the AMBA AHB bus, but they are not meant to be complete. They only refer to some of the bus's features.

### 3.1 Separate Event Generators and Event Consumers

This example, shown graphically in Figure 3-1, consists of two Event Consumer classes receiving events from two Event Generator classes.

| | |
|---|---|
| AHB Bus Decoder: | This class is an example of a concrete Event Generator that monitors the AHB bus, and can send three events - ADDRESS_COMPLETE, DATA_COMPLETE and BUS_ERROR |
| Stimulus Injector: | This class is an example of a concrete Event Generator. It injects bursts of test stimuli onto the AHB bus. Before starting each burst, it sends the TEST_STARTING event to its Event Consumers |
| Simulation Controller: | This class is the top-level simulation object, and is a concrete Event Consumer. It receives the TEST_STARTING event from the Stimulus Injector, which it then uses for internal housekeeping. It also subscribes to the BUS_ERROR event |
| Watchdog: | This class is an example of a concrete Event Consumer and it has the remit to terminate the simulation if 200 clock cycles pass with no activity of the AHB bus. To do this, it has subscribed to the ADDRESS_COMPLETE and the DATA_COMPLETE events from the AHB-Bus-Decoder Event Generator |

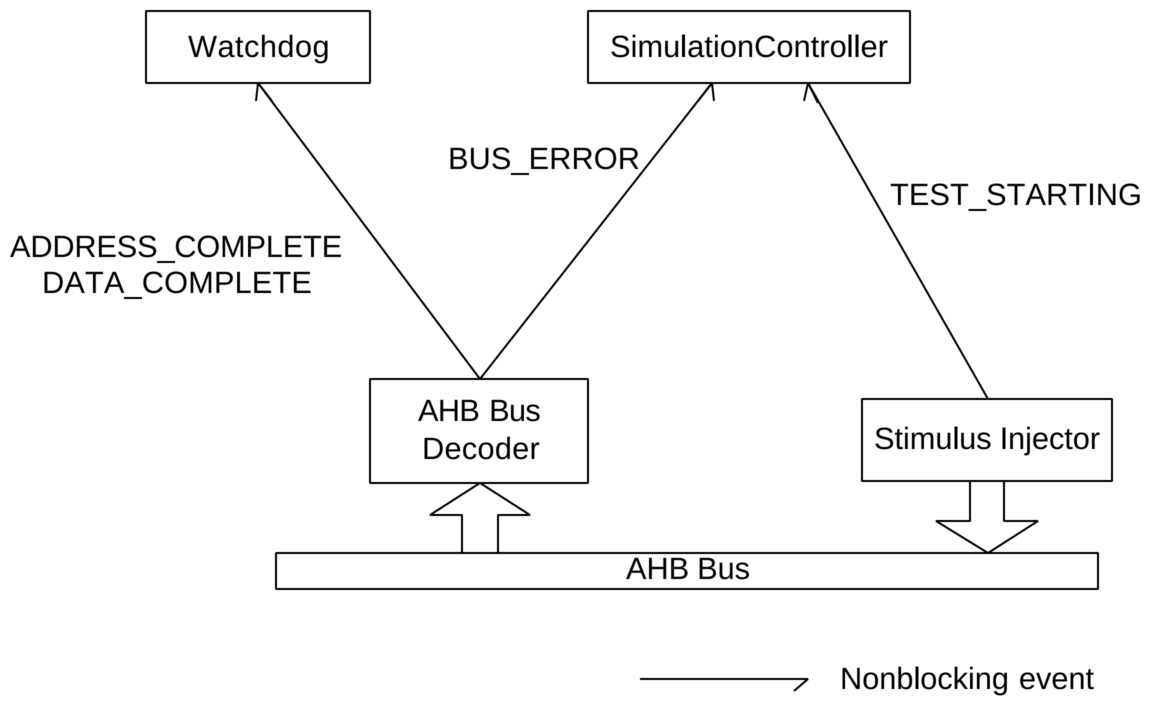

Figure 3-1: Simple event propagation example

## 3.2 Combined Event Generators and Event Consumers with Blocking and Nonblocking Events

This example is based on the previous example, but extends it by having objects that can be both Event Generators and Event Consumers. It also uses both the blocking and nonblocking versions of an event. In particular, the TEST_STARTING event occurs in both modes. The following classes have been modified:

| | |
|---|---|
| Simulation Controller: | Event Generator functionality has been added to this class. When it receives a TIMEOUT event from the Watchdog, it has to terminate a number of other objects. It does this by sending the blocking version of the TERMINATE event to its Event Consumers. Once all of the Event Consumers have finished processing the event, control is returned to the Simulation Controller, and it terminates the simulation. The TERMINATE event has to be blocking or else the simulation would be terminated before the Event Consumers had an opportunity to deal with the event |

| | |
|---|---|
| Watchdog: | Event Generator functionality has been added to this class. When it reaches its timeout value, rather than terminate the simulation, it sends the TIMEOUT event to its Event Consumers. One of these will have to do the actual termination |
| Stimulus Injector: | Event Consumer functionality has been added to this class. It registers to receive the blocking TERMINATE event from the Simulation Controller, and uses this to print the number of tests being cancelled |

The following class has been added:

| | |
|---|---|
| Scoreboard: | This is a concrete Event Consumer and it implements some generic form of scoreboard function. The Scoreboard observes the Simulation object for the TERMINATE event, and when it receives the event, it checks itself to see that it is empty. If not, it issues an error. It also registers to receive the blocking version of the TEST_STARTING event from the Stimulus Injector. It receives the blocking version, because it will prevent the Stimulus Injector continuing until the scoreboard is empty (signifying the previous test completed) |

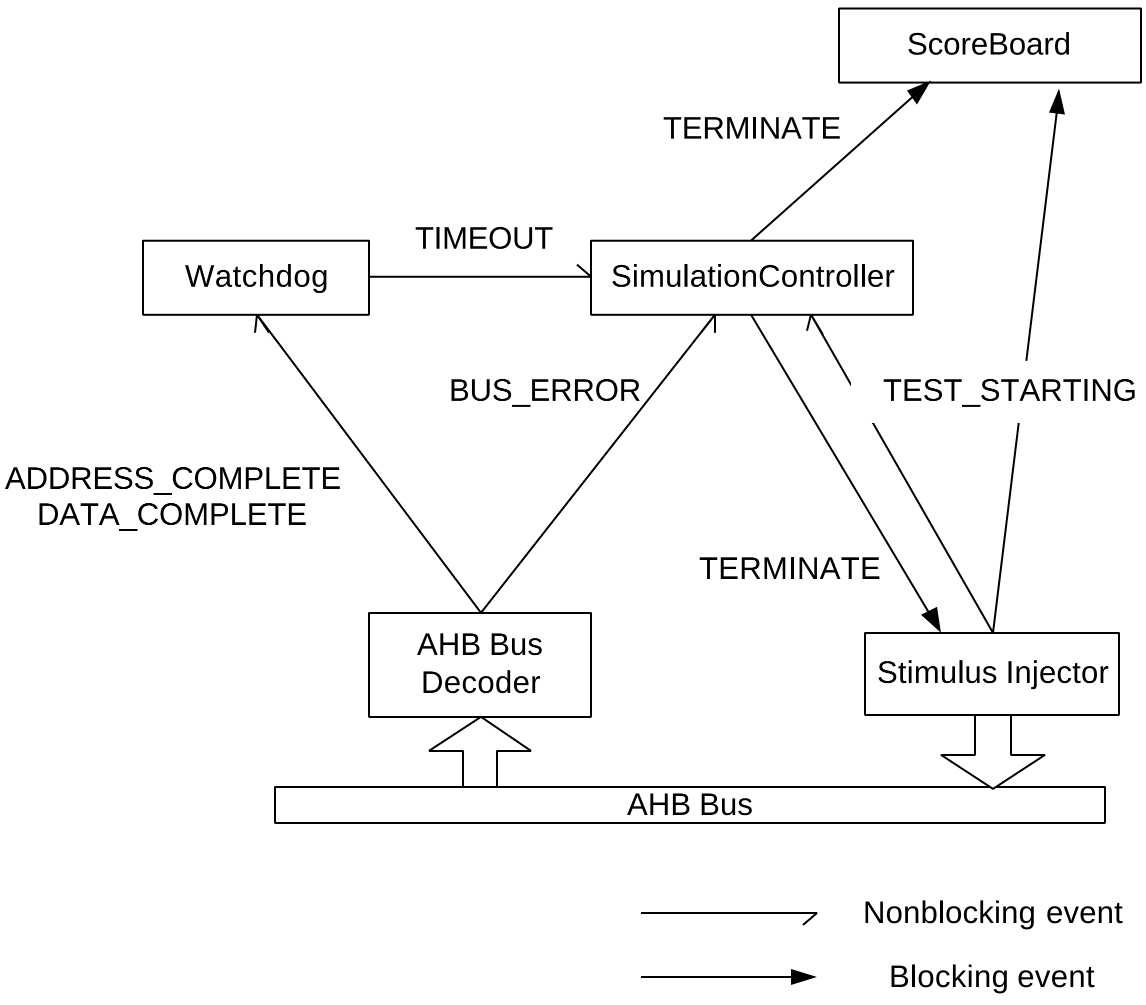

Figure 3-2: More complex event propagation example

## 3.3 Comparison With Vera *event* Variables

Vera has some built-in features for synchronising concurrent processes, which can also be used to implement subjects and observers. Vera *event* variables can be used in conjunction with the *trigger* function (to implement an Event Generator's notification of state change), and the *sync* function (to implement an Event Consumer's detection of notification). An example might be where variables are assigned values based on signals sampled on a clock edge. Another process wishing to use these variables must ensure they have been assigned before use. A dedicated event might be used, where the *trigger* is enabled after the variables have been assigned and any interested processes would have a corresponding *sync* watching the *event*.

There are a number of problems with the Vera events that are not present in the Observer Pattern solution.

### 3.3.1 Ad hoc Usage

The *event* variables are really only the medium by which *sync* and *trigger* communicate. Using this methodology on a more distributed problem, where classes become subjects and observers, we run into some limitations. Although *sync* can detect a triggered *event*, it does not have any defined way of getting back state from the sender of the event. In fact, there is no consistent way of even finding out which object in the testbench sent the event. These problems can be solved by adding extra code, but this approach leads to ad hoc code; one of the reasons for using the Observer Pattern in the first place. It is possible that the enhanced Observer Pattern discussed in this paper could be implemented using Vera events, but this approach has not been investigated.

### 3.3.2 Complexity and Debugging

Vera events supply four types of synchronisation options, and five types of trigger options [4]. While these may bring many advantages to many complex thread synchronisation problems, when used for their message passing abilities, they are far too complex. Debugging Vera events is difficult because of lack of visibility. For performance reasons, neither the Vera debugger nor waveform dumping are normally used. This means that most Vera debugging occurs using log file messages. As Vera events are part of the language, the designer cannot easily add code to consistently log when an event is sent or received (although ad hoc solutions can be implemented). Event complexity coupled with limited debugging capabilities make Vera events unattractive to use in most situations.

This is not the case with the solution proposed in this paper. There are few options for sending and receiving events, as they can only be blocking or nonblocking. Debugging is also much easier, as all events are sent from the Event Generator's notify task, so global event generation logging can be easily controlled. Event reception is slightly harder to implement, as the events are received by the concrete Event Consumers' update() tasks. However, as long as the designer puts a call to the base class's version of update (or an equivalent debugging task), then event reception debugging can be easily controlled.

### 3.3.3 Blocking Events

The biggest limitation with Vera events is that they are always nonblocking. The *trigger* command always returns before the events are dealt with, and there is no obvious way to prevent this. The best that can be achieved is to limit the *trigger* option to HAND_SHAKE, and have some form of communication from the Observer to the Subject that specifies when an event has been handled. The Subject would have to stall sending the event to the next Observer until this is received. This makes them very difficult to use as a generic message passing mechanism.

## 4 Vera Implementation

This section describes how to implement the modified Observer Pattern in Vera. Rather than present all of the source code here, only solutions to the following four problems are included. The source code is available in its entirety from www.verilab.com. The following issues have been selected for further elaboration because they are the ones that we have found to cause most problems when the pattern is implemented for the first time.

Before the problems are introduced, certain aspects of our coding style must be explained. For historical reasons, we generate and maintain the header files ourselves. These header files are included in the class' vr file, and in any file that uses the class. Normally the only difference between these two usages is the use of the extern keyword before the class declaration. If the header is being used to compile the class' vr file, extern should not be present. We use a text macro to specify how the header is being used. The class' vr file must define this before including the header file (Figure 4-2 line 1). If this is not defined, then another text macro is defined to be the word extern. This text macro is placed before the class definition (Figure 4-1 line 7). Therefore, if the header is being used locally to the class body, the class definition will be "virtual class EventGeneratorBaseClass". If it is being used in another class, it will be "extern virtual class EventGeneratorBaseClass".

```
1.  #ifndef EVENTGENERATORBASECLASS_CLASS
2.  #define EVENTGENERATORBASECLASS_EXTERN extern
3.  #else
4.  #define EVENTGENERATORBASECLASS_EXTERN
5.  #endif
6.
7.  EVENTGENERATORBASECLASS_EXTERN virtual class EventGeneratorBaseClass{
8.  task new();
9.  }
10. #endif
```

Figure 4-1: Code fragment showing our normal header file format

```
#define EVENTGENERATORBASECLASS_CLASS
#include "EventGeneratorBaseClass.vrh"
task EventGeneratorBaseClass::new(){
}
```

Figure 4-2: Code fragment showing our normal vr file format

### 4.1 How does the Event Generator Know Which Event Consumers are Subscribed ?

One requirement of the Event Generator base class is that it stores and manages the details of the Event Consumers that have registered for each event. One method of doing this is to use two dimensional arrays, indexed by the event identifier and the Event Consumers that have subscribed to these events. That is, the event identifier is effectively used to access an array of Event Consumer object handles. As the number of events and the number of Event Consumers are unknown at compile time, two dimensional associative arrays are required. However, Vera does not support multi-dimensional arrays. This problem is solved by creating an array class that is local to the Event Generator. Although Vera does not support multi-dimensional associative arrays, it does support an associative array filled with handles to objects that contain associative arrays. Figure 4-3 shows a fragment from the EventGeneratorBaseClass' header file.

As described in the previous section, the identifier EVENTGENERATORBASECLASS_CLASS (line 1) is defined when this header is included from the class' vr file. The section between lines 1 and 5 is therefore only compiled when this header is used in conjunction with the body of the class; external references to this do not see these lines of code. This section creates a local class that is an associative array of EventConsumerBaseClass handles. In the EventConsumerBaseClass, a static integer is used to assign a unique identifier to every Event Consumer in the design. It is this identifier, rather than the object's handle, that is used to provide a reference into the array. Line 7 is the declaration of the EventGeneratorBaseClass. Lines 9 to 11 define the two dimensional array needed to store the Event Consumers subscribed to each event.

```
#ifdef EVENTGENERATORBASECLASS_CLASS
class EventConsumerArrayClass{
 EventConsumerBaseClass m_Array[];
}
#endif

EVENTGENERATORBASECLASS_EXTERN virtual class EventGeneratorBaseClass{

#ifdef EVENTGENERATORBASECLASS_CLASS
 local EventConsumerArrayClass m_EventConsumers[];
#endif
}
```

Figure 4-3: Code fragment showing how Event Consumers are stored

## 4.2 How are Blocking and Nonblocking Events Differentiated ?

A blocking event is one where the Event Generator that sent the event is blocked from continuing until the Event Consumer is finished with it [1]. Therefore, the same event may be sent to different Event Consumers on different clock cycles because some of the Event Consumers were time-consuming. Nonblocking messages on the other hand do not yield control to the Event Consumers. All Event Consumers subscribed to a particular event from a particular Event Generator should receive the event at the same simulation time.

To support two types of the same event, the EventGeneratorBaseClass maintains two lists of Event Consumers. One of these is for those Event Consumers who registered for the blocking version of an event, and the other is for those that registered for the nonblocking version. The Event Generator's attach, detach and notify tasks also require an extra boolean parameter to specify which type of event is being referenced. This parameter is used to select which Event Consumer array to use.

```
EVENTGENERATORBASECLASS_EXTERN virtual class EventGeneratorBaseClass{
#ifdef EVENTGENERATORBASECLASS_CLASS
  local EventConsumerArrayClass m_BlockingEventConsumers[];
  local EventConsumerArrayClass m_NonblockingEventConsumers[];
#endif
 task new();

 task attach(EventConsumerBaseClass EventConsumer, integer Event,
             bit Blocking);
 task detach(EventConsumerBaseClass EventConsumer, integer Event,
             bit Blocking);
 task notify(EventGeneratorBaseClass EventGenerator, integer Event,
             bit Blocking);
}
```

Figure 4-4: Code fragment showing how both blocking and nonblocking events are handled in the EventGeneratorBaseClass header

It is up to a concrete Event Generator to decide when to send each type of an event. The nonblocking should always be sent first to ensure that all relevant Event Consumers receive it at the time the event occurred. For blocking events, the Event Generator's notify task calls the Event Consumers' update tasks directly, but for nonblocking events, it calls each one from a fork-join block.

## 4.3 How do the Event Consumers Identify the Source of the Event?

When an Event Consumer's update task is called, it is passed a handle to an EventGenerator-BaseClass and an integer event identifier. To promote loose object coupling, it is undesirable to maintain a central list of all event identifiers. Therefore, events from different concrete Event Generators may have the same identifier. However, event identifiers from within a specific concrete Event Generator must have unique identifiers.

The first task of an Event Consumer's update task is to identify the type of Event Generator that sent the event. Vera provides a task called cast_assign() which can be used to convert a base class into the derived class it originally was. This task can take an optional parameter CHECK which prevents the task issuing an error if the cast is not possible. The update task uses cast_assign in a series of if statements to try and assign the EventGeneratorBaseClass object into each of the possible Event Generator classes that this Event Consumer expects events from. Once the correct Event Generator class is determined, a simple case statement on the event identifier can be used to identify the action to take for that event.

```
task ConcreteEventConsumer::update(EventGeneratorBaseClass EventGenerator,
                                   integer Event, bit Blocking){
  EventGeneratorFoo     Foo;
  EventGeneratorBar     Bar;
  if (cast_assign(Foo, EventGenerator, CHECK)) {
      case(Event){
      EVENT_0: printf("ConcreteEventConsumer received event 0\n");
      default: printf("ConcreteEventConsumer received an unregistered
                       event (%0d) from a EventGeneratorFoo\n", Event);
  }
  else if (cast_assign(Bar, EventGenerator, CHECK)) {
      case(Event){
      EVENT_0: printf("ConcreteEventConsumer received event 0\n");
      EVENT_1: printf("ConcreteEventConsumer received event 1\n");
      default: printf("ConcreteEventConsumer received an unregistered
                       event (%0d) from a EventGeneratorBar\n", Event);
  }
  else {
    printf("ConcreteEventConsumer received an event from an unknown type of
            EventGenerator\n");
  }
}
```

An alternative approach is to ensure that each event in the testbench has a unique identifier. Although this increases class coupling, it does offer the advantage of replacing a nested if-else tree with a single case statement, and delaying the call to cast_assign until it is actually required, which may be never if additional event information is not required from the Event Generator. In Event Consumers that can receive a large number of disparate events, this approach may be more efficient.

## 4.4 How do the Event Consumers Obtain Information from the Event Generators ?

It is not normally sufficient to just inform an Event Generator's Event Consumers that an event has occurred. As a minimum, the Event Consumer needs to know which of its Event Generators sent the event. Then, depending on the event, more information may be required. There are two common approaches to transferring this information [2]. One approach is the "push model", where the Event Generator sends the data as parameters in the Event Consumers' notify tasks. This approach can be problematic, as all Event Consumers share a common prototype for their notify tasks. This means that every unrelated Event Consumer in the design that monitors an Event Generator will have to share a common prototype. This becomes complex when Event Consumers can monitor more than one type of Event Generator, or when new Event Consumer types are added to a design. The push model also forces the Event Generators to know what kind of information their Event Consumers need. Again, this becomes complex when an Event Generator has more than one type of Event Consumer, or when new Event Consumer types are added later in the design.

The second approach is the "pull model". Here, the Event Generators make no assumption about the Event Consumers' requirements, and it is up to the Event Consumers to interrogate the Event Generator to retrieve the required information. The "pull model" simplifies the communication between the Event Generators and Event Consumers, because no Event-Consumer specific information has to be sent with the event notification. This allows multiple types of Event Consumers to easily monitor multiple types of Event Generators. Adding new Event Consumer types requires no changes to this approach.

This paper advocates the "pull model", shown in

Figure 4-5. When an Event Generator calls its base class's notify task, it passes its own handle as a parameter. This is up-cast in the call to an EventGeneratorBaseClass handle. In the Event Consumer's update task, it can down-cast this to the appropriate concrete Event Generator class, and then query the concrete Event Generator for all of the relevant information, using the Event Generator's public interface.

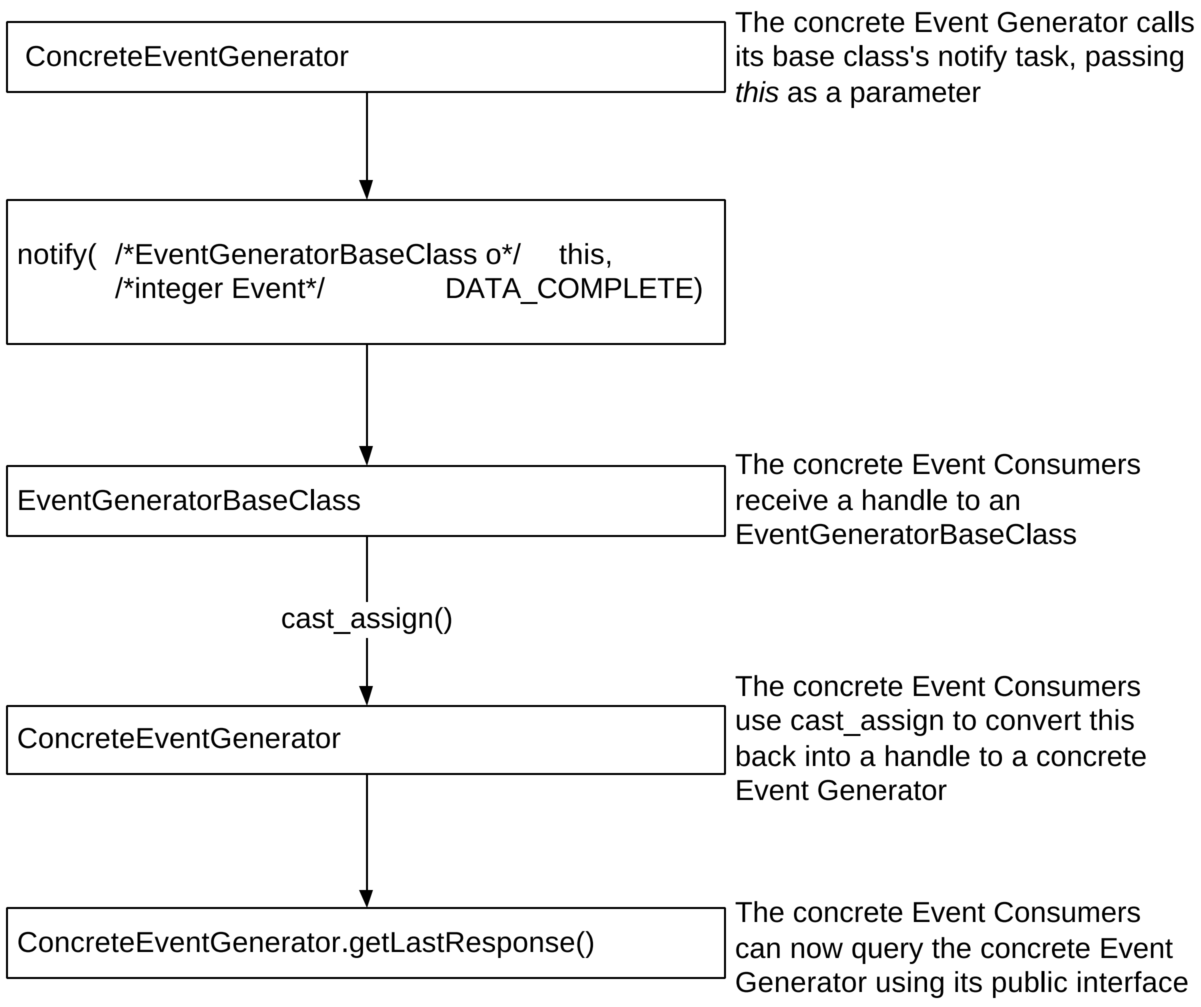
ConcreteEventGenerator
The concrete Event Generator calls its base class's notify task, passing *this* as a parameter
notify( /*EventGeneratorBaseClass o*/ this,
/*integer Event*/ DATA_COMPLETE)
EventGeneratorBaseClass
The concrete Event Consumers receive a handle to an EventGeneratorBaseClass
cast_assign()
ConcreteEventGenerator
The concrete Event Consumers use cast_assign to convert this back into a handle to a concrete Event Generator
ConcreteEventGenerator.getLastResponse()
The concrete Event Consumers can now query the concrete Event Generator using its public interface

Figure 4-5: Passing data from a concrete Event Generator to a concrete Event Consumer

### 4.5 How can a Class be an Event Generator and an Event Consumer ?

We have shown previously in section 3.2, that it is useful to have classes that can function as both Event Generators and Event Consumers. These classes can function as intermediaries between classes at the lowest and highest levels of abstraction in the testbench. However, to become both an Event Generator and an Event Consumer a class has to be derived from both base classes. Vera does not support multiple inheritance. Our solution to this problem is to merge the Event Generator and Event Consumer functionality into a single base class, called the EventConsumerAndGeneratorBaseClass (Figure 4-6). Any class derived from this base class will have the ability to perform both Event Generator and Event Consumer roles.

```
EVENTCONSUMERANDGENERATORBASECLASS_EXTERN virtual class
EventConsumerAndGeneratorBaseClass{
/** An ID that can be used as an array reference in the Subject class*/
  local static bit [63:0] m_IDseed = 0;
  local bit [63:0] m_ConsumerID;

#ifdef EVENTCONSUMERANDGENERATORBASECLASS_CLASS
  local EventConsumerArrayClass m_BlockingEventConsumers[];
  local EventConsumerArrayClass m_NonblockingEventConsumers[];
#endif

  task new();

/** Event Consumer tasks and functions*/
  virtual task update(EventConsumerAndGeneratorBaseClass Subject,
                      integer Events, bit Blocking);
/** Returns the ID of this consumer for array indexing purposes*/
  function integer getConsumerID();

/** Event Generator tasks */
  task attach(EventConsumerAndGeneratorBaseClass EventConsumer,
                      integer Event, bit Blocking);
  task detach(EventConsumerAndGeneratorBaseClass EventConsumer,
                      integer Event, bit Blocking);
  task notify(EventConsumerAndGeneratorBaseClass EventGenerator,
                      integer Event, bit Blocking);
}
```

Figure 4-6: Code fragment showing how a class can be both an Event Generator and an Event Consumer

Of course, having an Event Consumer that is also an Event Generator introduces the possibility of deadlock [5], which may occur, for example, when object A observes object B, which in turn is observing object A. Careful programming is required to avoid this problem.

## 5 Using the Observer Pattern in a Testbench

The previous sections in this paper have introduced the Observer Pattern, showed how it can be of use in a testbench, and discussed its Vera implementation. This section finishes the technical discussion of how to add the Observer Pattern in a testbench. Once the base classes have been created, adding the Observer Pattern to a testbench is a very simple process.

To create a concrete class, a verification engineer just needs to make a class extend the appropriate base class. If the concrete class is to act as an Event Consumer, then the concrete class also has to provide an implementation of the update() task. This is the task that decides what action to take when events are received, so cannot be generalised in the base class.

If the concrete class is to act as an Event Generator, then a public interface to certain data may have to be provided, to allow the Event Consumers to interrogate it for any information that may be required to process a particular event properly. Again, as this information is event specific, this interface cannot be generalised in the base class. When the Event Generator detects a condition that should trigger an event, it should make sure the appropriate sections of the public interface contain up-to-date data, and then trigger the event. This is done by calling its base class' notify() task, passing a reference to itself (this), the event identifier, and a flag to say whether to send the blocking or nonblocking version of the event, as parameters.

At some point in the testbench, Event Generators and Event Consumers must be brought together to allow them to exchange events. To register for an event, the handle of the Event Consumer must be available. The Event Generator's attach() task is called, and the Event Consumer's handle is passed as a parameter, along with the event identifier, and the synchronise flag.

## 6 Conclusions

Event propagation forms an important part of high-level testbenches, but without proper planning, it is easy to produce non-maintainable, non-expandable, ad hoc solutions that appear to work until late in the project when seemingly small additions are required.

This paper has taken a standard software solution called the Observer Pattern, and made enhancements to make it suitable for use as a generic and scaleable event propagation mechanism in a threaded testbench environment. Using our approach, Event Consumers can selectively subscribe or unsubscribe, at any time, to either the blocking or nonblocking version of any event. New events, event generators or event consumers can be added easily at any stage of the development process, without interfering with the existing operation of the testbench.

Although applicable to any object-oriented verification language, this paper has concentrated on a Vera implementation of the enhanced Observer Pattern. We have shown how some Vera limitations can be overcome to implement this pattern, and have also evaluated the inbuilt Vera *events* to assess their suitability for event propagation.

## 7 References

[1] D. Harel, "Real-time UML", Addison-Wesley, 1998

[2] E. Gamma, R. Helm, R. Johnson and J. Vlissides, "Design Patterns - Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995

[3] J. Rumbaugh, I. Jacobson and G. Booch, "The Unified Modelling Language Reference Manual", Addison-Wesley, 1999

[4] F. Haque, K. A. Khan and J. Michelson, "The Art of Verification with Vera", Verification Central, 2001

[5] C. Szallies, "On Using the Observer Design Pattern", Web Publication, Aug. 1997, http://www.wohnklo.de/patterns/observer.html