

OpenVera Technology Background

April 2001

Introduction

Ever increasing design complexity has made it essential to drastically improve verification productivity. Traditional testbench solutions, HDLs and home-grown tools using C/scripts, are woefully inadequate at meeting the verification challenge. HDLs lack both the abstractions and the tools required for productive verification and C/scripts are difficult to use and maintain.

OpenVera™ was developed in response to the need for a powerful, highly productive but easy-to-use verification language. It takes the best features of HDLs, C++ and Java and adds additional constructs that allow testbenches to be written in a compact, reusable and maintainable way. OpenVera enables designers to create self-checking testbenches that include both directed and constrained random tests. Moreover, it lets the user measure the coverage achieved by the test while it is executing and modify the properties of the test generator to get maximum coverage with minimum simulation. OpenVera has been designed to have all the language constructs needed to operate seamlessly with HDL simulators and C-based tools such as instruction set simulators. This white paper explores the requirements for a high-level verification language in detail and then describes how OpenVera meets these requirements.

Verification Language Requirements

A high-level verification language must meet the following key requirements:

- It must support the modeling of testbench functionality at a high level of abstraction. There are several aspects to this. First, the stimulus applied to the device being verified needs to be specified. Second, the expected response of the device to the applied stimulus must be computed. This must be done at a high level of abstraction so that it can be implemented more quickly than creating the RTL itself. Third, the test fixtures that allow tests to be applied and results to be checked must be modeled.
- It must support the specification of stimulus that is either directed at checking specific design functionality or is based on constrained random patterns. The random pattern generator must allow for constraints to be specified in a compact, declarative way. The language must also allow random sequences to be specified.
- It must support the specification of metrics to measure whether or not test goals are being met. In addition, the language should allow these metrics to be queried dynamically so that stimulus generation can be dynamically adjusted to maximize test effectiveness.
- It must allow the user to specify connections to HDLs and C in a convenient way both to connect to the device being verified and to leverage existing models and tools.

The following section describes how OpenVera meets these requirements.

The OpenVera Language

OpenVera has been designed specifically to address the needs of high-level verification. The essential features of the language are as follows:

- It is an object-oriented language with support for complex data-structures, encapsulation and inheritance. The basic language design is very similar to C++ or Java making the learning curve for using OpenVera very low.
- It has built-in data types that allow the user to model arbitrary-width 4-valued signals (0,1, x, z).
- It is a simulation language: it allows the user to model the execution of events in time.
- It allows testbench elements to be written in terms of abstract signals, which can be assigned to actual HDL signals as the testbench executes. This is very useful in allowing reuse of testbench elements.
- It has a mechanism for specifying interface timing for the device under verification; this includes specifying clocks, drive and sense edges and skews. This mechanism allows the testbench to be written in a way that is independent of detailed timing information.

- It has powerful capabilities for checking for expected values over windows of time.
- It has comprehensive mechanisms for dynamically creating and synchronizing concurrent processes. The ability to dynamically create processes is extremely useful in creating testbench elements for a particular test and then discarding them when the test is done.
- It supports constrained random testing in two ways: random object generation and random sequence generation. In both these cases, the values and sequences generated meet user-specified constraints.
- It has a comprehensive functional coverage mechanism. OpenVera routines can query coverage values during simulation and change generated patterns to maximize coverage of untested features.
- It has the ability to specify temporal sequences that are checked during simulation. These sequences can be specified using a compact, declarative syntax.

Creating a Verification Environment with OpenVera

In order to increase productivity during design verification, it is essential that different components of the verification environment be modeled at a high level. OpenVera enables designers to create reusable objects that can be implemented using abstract data-structures, rather than low-level, RTL-like constructs. In the style of Java and C++, OpenVera is an object-oriented language with the capability to separate the interface of an object from its implementation. For example, the user of a bus functional model (BFM) object would interface only with the read and write tasks of the object without worrying about the detailed implementation of the object. Hence, a test suite could be written to use this BFM without concerning itself with detailed signal-level activity. This BFM could be adapted to deal with changes to the bus protocol without affecting the test suite, which sees only the read and write tasks.

OpenVera allows users to create complex objects that can have public, private or protected data and function fields. OpenVera objects can inherit properties from other objects and extend them. For instance, a verification project lead can create a base monitor object that has some data logging capability built into it. Each individual verification engineer can then extend this object to give it properties needed for a particular monitor. This would ensure that all monitors in the testbench have uniform data logging.

In order to facilitate the modeling of hardware, OpenVera has built into it the notion of simulation time. This allows the user to specify delays within the testbench, as well as to synchronize to events in the simulation. Moreover, OpenVera has built-in data-types that allow the user to model arbitrary width 4-value (0,1,x, z) signals. There is a wide range of operators that allow the user to conveniently manipulate these signals.

The ability to describe concurrency is vital to writing a testbench. The next section describes the mechanisms available in OpenVera for this purpose.

Concurrency and Synchronization

OpenVera has a very convenient mechanism for creating concurrently executing threads using the familiar fork-join construct. For example, if the verification engineer wants to test whether two 2-input ports on an NxN switch can operate concurrently, he or she can fork concurrent tasks that write to 2-input ports of the switch and two checkers that check the appropriate output ports. Moreover, the user could specify the join-all condition for checking that all tasks completed. OpenVera gives the user the flexibility to specify join-all, join-any or join-none with the fork-join construct. Join-all causes the parent thread to suspend until all the forked threads have completed. Join-any causes the parent thread to suspend until any one of the forked threads completes, and join-none allows the parent thread to schedule the forked threads to execute without suspending its own execution.

Unlike HDLs, where the structure of the testbench is fixed for the duration of the test, OpenVera allows the verification engineer to vary testbench infrastructure as the test proceeds. For example, one can fork a checking routine for the duration of a particular test. Once the test is over, the checker terminates. OpenVera's ability to dynamically allocate objects and create new threads of execution provides the user with the flexibility to create the exact environment needed for each test.

OpenVera has several mechanisms for synchronizing concurrent threads. These include events, semaphores, mailboxes and regions. Events are the most basic form of synchronizing primitive: a thread can suspend execution and wait for a particular event. Another thread can trigger that event at the appropriate time. For example, if a testbench has reader and writer threads, the reader can wait for the writer to trigger an event indicating that there is data available to be read.

Semaphores can be used for mutual exclusion and for sharing a pool of resources. For example, a set of test routines could co-ordinate the use of a shared bus using a semaphore. Mailboxes are a convenient mechanism for passing information between concurrently executing threads. For example, suppose there are two threads, one generating stimulus and one checking the expected result. The stimulus generator can use a mailbox to pass information needed by the checker to compute expected values. Mailboxes allow arbitrary objects to be passed from one thread to another.

In many tests, it is useful to record values that are in use by a thread so that other threads don't try to use the same values. OpenVera offers the region primitive that allows a thread to reserve values that it needs for its operation. For example, suppose that there is a set of test routines that all operate on the same cache. Each test routine can reserve the cache addresses that it is using with the region primitive.

HDL Interface

In OpenVera, the testbench is connected to the design under test by simply specifying hierarchical paths to the signals that are to be controlled or observed. Detailed timing information, such as clocks, active edges and skews, can be encapsulated using an interface object.

Using OpenVera, designers can write testbench routines using abstract ports. In HDLs, device instances (e.g. test generators or checkers) are permanently connected to signals during the entire simulation. OpenVera's abstract ports allow objects to be connected to appropriate signals as the simulation proceeds. For example, one can create a special-purpose checker and connect it to a bus when the appropriate point in the simulation is reached. When the test is complete, the checker can be removed from the bus. Abstract ports allow testbench routines to be easily reused.

OpenVera facilitates checking of expected values by providing operators that check for values, not just at a given instant, but over windows of time. For example, the verification engineer can specify that a signal should have a particular value at least once during the next 20 cycles. Similarly, OpenVera has a value change alert construct to check that only expected activity happens on a signal.

In order to facilitate the reuse of existing testbench infrastructure, OpenVera can call HDL tasks and HDL tasks can call OpenVera.

Constrained Random Testing

Random testing is a very effective complement to traditional directed testing. OpenVera has a compact syntax for specifying constraints on object values. A solver can then take these constraints and generate random values that meet the constraints. For example, suppose we are testing a bus protocol and want to generate random, yet legal transactions. Using OpenVera, one can create a transaction object that has constraints for all the different types of legal transactions. In OpenVera, constraints can be varied as the test proceeds.

In addition to specifying constraints on objects, OpenVera has the capability to specify legal sequences that are to be generated for a test. Since most devices being verified allow several alternate sequences, in OpenVera one can control the probability that a given sequence will be generated. The probabilities can be varied as the simulation proceeds and different test objectives are achieved.

In order to get the maximum benefit from both directed and random testing, it is essential to have a metric for measuring test effectiveness. As described in the following section, OpenVera's coverage capability provides such a metric.

Coverage

In OpenVera, coverage objects are used to measure test effectiveness. Within each coverage object, designers can define values or sequences of values that are of interest. These coverage objects can then monitor variables in OpenVera or signals in the device being verified. The gathered coverage information can be written out in the form of a report. Perhaps more importantly, the coverage objects can be queried as the test proceeds to check for progress.

The ability to query coverage objects during simulation is particularly effective when combined with random testing. To do this, coverage objects can be created corresponding to test objectives. A set of constrained-random test generators can be used to generate test stimulus. These generators can query coverage objects at appropriate intervals to see whether test objectives are being met. As test objectives are met, the constraints in the test generators can be varied to focus on remaining test objectives. For example, suppose you are interested in verifying that processor interrupts are correctly handled under different conditions. You can set constraints to randomly generate instruction sequences and interrupts. As each type of interrupt is checked as indicated by the appropriate coverage object, you can bias the random patterns towards checking as yet untested conditions.

Temporal Assertions

Many complex protocols can be described efficiently using a temporal-logic-based syntax. In some ways, these temporal assertions are analogous to regular expressions that are used to match character strings. However, rather than matching characters, they match sequences of events. OpenVera includes a comprehensive syntax for specifying temporal assertions. For example, temporal expressions can be specified for checking that a particular type of bus transaction takes place. When a temporal assertion succeeds or fails, other activity in the testbench can be triggered.

Both temporal assertions and the constraints specified on input stimulus can be used by formal or semi-formal tools that complement simulation with other kinds of analysis.

Interface to C

OpenVera includes the capability of calling functions in external C libraries. Moreover, it has the ability to talk to other applications using a socket-based interface. This capability is extremely useful for running OpenVera testbenches with C-based instruction set simulators. This capability also enables distributed simulation where a farm of machines can cooperate on verifying a design.

Conclusions

OpenVera is a comprehensive language for writing testbenches. OpenVera extends the capabilities of general-purpose languages such as C++ with constructs needed for effective design verification. These capabilities include hardware data-types, concurrency, time, constrained-random stimulus generation, coverage and temporal assertions. Because of these capabilities, it is the testbench language of choice for verifying complex designs.

SYNOPSYS®

700 East Middlefield Road, Mountain View, CA 94043 T 650 584 5000 www.synopsys.com
For product related training, call 1-800-793-3448 or visit the Web at www.synopsys.com/services

Synopsys and the Synopsys logo are registered trademarks, and OpenVera is a trademark of Synopsys, Inc. All other products or service names mentioned herein are trademarks of their respective holders and should be treated as such. All rights reserved.
©2001 Synopsys, Inc. 3/01.TM