So There's My Bug!
Debugging Universal Verification Methodology (UVM) Environments

Mike Floyd
Cadence Design Systems, Inc.
270 Billerica Rd.
Chelmsford, MA 01824
1- 978-262-6241
mikef@cadence.com

ABSTRACT

Modern verification environments like those built with the Universal Verification Methodology (UVM) more closely resemble software applications than hardware applications. The challenge is that the teams building and debugging such environments are more often trained in hardware verification than software verification. Debug considerations start in the verification component development phase where bugs can lurk in sloppy data structures and object inheritance management. Most of the bugs should be removed during block-level verification where the new issues for designers are debugging dynamic data types and through class inheritance hierarchies. In scaling to the system level, we are squarely in the transaction verification space searching among thousands of concurrent transactions – often in multiple verification languages – to find the last few bugs in our system. The new techniques presented in this paper will provide verification engineers with the information to both limit the introduction and speed the removal of bugs in SystemVerilog- and ***e***-based verification environments leading to faster convergence and higher quality silicon realization.

1. INTRODUCTION

Hardware engineers have built expertise debugging complex designs but the constant in each project has been the code and data structures. UVM, implemented with SystemVerilog or ***e***, changes that by introducing both dynamic data types and dynamic code that can both created and destroyed during the execution of a given simulation run or test.

It is this transient nature that is the heart of the challenge. For example, UVM testbenches often create threads of execution during a given test run and the engineer must be cognizant of the specific thread being displayed in the source browser. Compounding the challenge is that the flow of execution in the dynamic code flows from method to method within and across inheritance trees rather than through static, hierarchical interfaces. A similar situation occurs with data where an object grows and shrinks through simulation execution, which may be untimed, so debug methods that expect data persistence have to be adjusted.

The net result is a new set of challenges for hardware engineers. The most common issue is memory management where data structures grow unexpectedly so new approaches are needed to recognize and resolve these testbench bugs. Often related to memory management is understanding class inheritance to avoid obfuscating data and methods and debug tools can help the engineer understand the environments they both build and receive. Since UVM leverages both class environments and efficient memory usage, it has these challenges and some unique ones related to its dependence on transaction verification and constraint randomization. Developing new skills is critical to efficiently debugging these modern verification environments.

2. RECOGNIZING AND MANAGING MEMORY LEAKS

2.1 Memory Management

In static environments the process size does not grow except to handle recording of waveform data or for users PLI code which may allocate memory. With class based environments and dynamic data types this is not the case. Memory is allocated as dynamic objects grow and as class objects are created. Care has to be used to properly manage the dynamic data or simulation process size can continue to grow and results in the process terminating due to memory allocation failures. Even if memory allocation does not fail it is possible to grow the process size to the point at which it does not fit within the physical memory on the system. This then leads to paging and swapping which causes simulation performance to degrade.

2.2 Memory Leaks

In order to understand how a memory leak is created one need to understand the difference between class variables and class objects. A class variable is a pointer that references a class object that has been allocated by calling new. Multiple class variables can have references to the same class object. A class object exists and the memory is not reclaimed until the reference count on the object drops to zero. If care is not used in storing references to class objects it is possible to create a memory leak.

A real case example of this occurred with several customers. In one example a customer allocated a data member by calling new and then stored this in a dynamic array. Later in the code they then assigned NULL to the class data member. By doing this, the customer thought they had freed the memory; they had not because the dynamic array held a reference to the allocated object. This resulted in the dynamic array growing for every data member allocated.

There are several ways a debug environment must help in tracking down memory explosions. The first is using a class browser to show all the instances of a class in existence. A class browser shows the inheritance hierarchy of the classes in the design and is useful for many debugging tasks. Looking at a list of instances can show that objects are being created but not freed. This is the most basic approach. Another more powerful approach is to use a memory analyzer. A memory analyzer must be able to show the current heap

memory allocation and to graph the heap memory usage over time. This can then be used to show what objects are growing in size over time and can allow the user to better focus on the parts of the environment that are using the largest amount of memory. By graphing individual object size over time, it become clear where memory leaks may be occurring. It was by using the memory analyzer that the customer was able to locate the source of the memory leak in the dynamic array. The graphing capability and instance specific data presented to the user allowed the user to quickly find the area where memory was exploding and to determine the object that was growing in size. By analyzing the source they were able to easily find the cause and quickly implement a solution.

2.3 Duplicating Data – Inefficient Memory Usage

Another source of excessive memory use is duplicating data members in derived classes. Cases have been seen where a user has derived a class from a base class defined in a class library such as UVM. The customer then declared a data member in the derived class to hold some information. It turns out that there already existed a data member in a parent class that held the same data. The user has now increased the size of the object to contain duplicate data that the object was already holding. This is easily avoidable.

A debug environment can help in several ways. First, a class browser should be able to show all of the data members of a class definition including those that are inherited from a parent class. This data must include the visibility of the data members in derived classes so the user knows what is accessible in a derived class. Additionally, if the user is viewing an instance of the object, they should be able to see all of the data members in both a flattened view and by inheritance from parent classes. This is also true when viewing waveforms for a class object, the user must be able to view all of the data members of the object for the lifetime of the object.

3. OBFUSCATION OF CODE AND DATA

One of the challenges of debugging a class based verification environment is the creation of dynamic scopes. This can result in multiple instances of a class. To debug an instance of a class the user must be able to locate it quickly and once located, the environment must support the viewing and controlling simulation specific to this instance. This is best accomplished by debugging at the source level. The user must be able to use a class browser to quickly find all the instances of a class and to view that instance in a source browser. In Figure 1, the class browser sidebar is displayed with an instance of the bus monitor selected. All source value annotation must be for data member in the specific instance of the class. The user also has to be able to set line breakpoints in methods of the class for any instance and for a specific instance of the class. When a breakpoint is hit, the user must be able to traverse the call stack to see when the method was called from. Without knowing the context of the call it is almost impossible to understand the flow of control in simulation to track down a bug in the test bench.

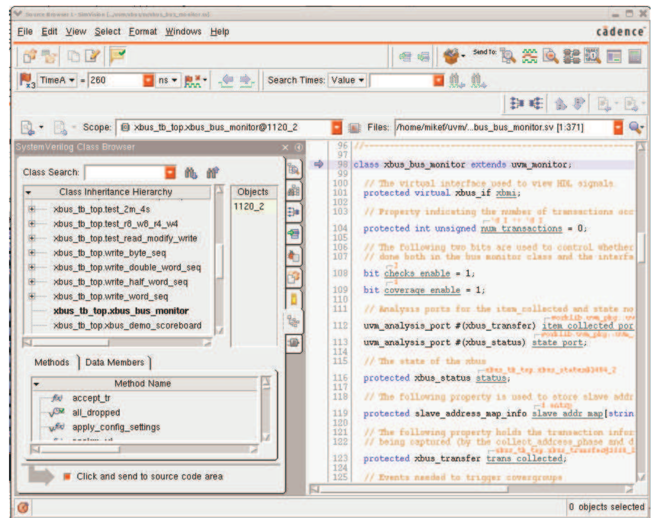

Figure 1: Object from Class Browser Displayed in Source

Class variables also hold references to class objects. If the user is starting with a class variable or a data member of another class instance, they must be able to have the debug environment automatically dereference the variable to show the class instance that is pointed to by the variable in the source browser for accessing instance specific data. Take for example a class variable that has a reference to a sequence generator. Being able to take the class variable and have the source be shown for the instance of the generator simplifies the users debugging of the generator by being able to quickly traverse into the generator instance and set line breakpoints for that instance of the generator.

Finally the user should be able to create waveforms and view class objects and there data members for the lifetime of the object. Class variables should hold references to class objects so that the user can see how class variables are holding references to class objects over time with the ability to focus on an individual class object and its data members..

4. UVM-SPECIFIC CONSIDERATIONS

UVM is quasi-static in nature. This means that once constructed, the UVM verification component hierarchy does not change. As a result a verification environment must be able to show the quasi-static UVM verification component hierarchy as a separate hierarchy from that of the design hierarchy. This provides a clean separation of the verification hierarchy from the design hierarchy making it easier for the verification engineer to focus on the verification components. This allows quick traversal of the verification hierarchy allowing the user to quickly display source, probe the hierarchy to waveforms and to verify the correct construction of the verification hierarchy. Figure 2 shows UVM component hierarchy after construction.

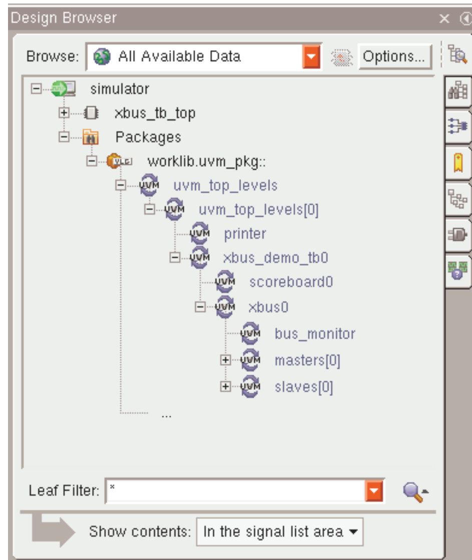

Figure 2: UVM Verification Hierarchy Display

UVM is also a transaction based verification methodology by using sequencers to generate sequence item to stimulate the design under verification. A debug environment must support the recording of the transactions to a database. These transactions can then be viewed in waveforms. However, when debugging sequences it is imperative that the order of sequences in the system is known. This can only be handled by displaying transaction items in a stripe chart display. This type of display allows the user to see the order of transactions for all components in the hierarchy or for selected components. The stripe chart must be linked to the waveform window so that the duration of the transaction and the other transactions from the component can be viewed. Transactions also have predecessor/successor relationships. The strip chart must be able to show these as well as parent/child relationships.

Finally, in a verification environment it is important to take advantage of constraint randomization. When randomizing an object it is possible that values for a variable or variables cannot be solved due to an over constraint violation. When this occurs simulation stops. A debug environment must be able to debug constraint violations. To do this, the constraint debugger must automatically focus on the constraints that are in conflict. The debugger must allow the user to change the rand state of variables, enable/disable constraints and to define a new constraint on the fly. The user must then be able to run the constraint solver to verify the changes before continuing simulation with solved values.

5. ACKNOWLEDGMENTS

The recommendations outlined in this document have come from gathering input from many sources. This includes customers, Cadence application engineers, product engineers, and R&D. I wanted to extend my thanks to all that have contributed, directly or indirectly to making a more robust verification environment.

6. REFERENCES

[1] IEEE *IEEE Standard for SystemVerilog — Unified Hardware Design, Specification, and Verification Language*. New York: IEEE 2005

[2] IEEE *IEEE Standard for the Functional Verification Language* ***e***. New York: IEEE 2008

[3] Accellera *Universal Verification Methodology (UVM) 1.0 Early Adopter* California: Accellera 2010