Hardware Verification with the Unified Modeling Language and Vera

Kevin Thompson
Ladd Williamson

Cypress Semiconductor

kbt@cypress.com
ldw@cypress.com

ABSTRACT

A method is proposed whereby the Unified Modeling Language and accompanying process is used with the Synopsys Hardware Verification Language Vera to create a verification test suite. The Rational visual modeling tool Rose C++ edition output is modified and used to generate Vera code stubs. These stubs may then be filled with the actual verification code. The use of UML with Vera helps the designer to write a more robust and flexible verification test suite.

Acronyms Used	
AMBA	Advanced Microcontroller Bus Architecture
AHB	Advanced High Performance Bus
HDL	Hardware Description Language
HVL	Hardware Verification Language
OO	Object-Oriented
ROPES	Rapid Object-Oriented Process for Embedded Systems
RUP	Rational Unified Process
UML	Unified Modeling Language
USB	Universal Serial Bus
UTMI	Universal Transceiver Model Interface

1.0 High Level System Description and System Constraints

In the summer of last year we were tasked with creating a test bench for a new IP block. At the heart of the new IP block was a USB 2.0 core. Added to this IP block were FIFOs, memory, supporting logic, and a DMA engine. This block also had an industry standard UTMI front end and an industry standard AMBA AHB interface on the back end. The block diagram is shown below:

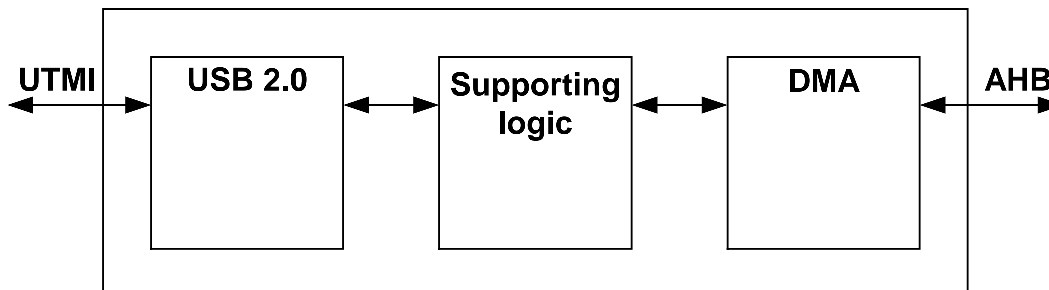

Figure 1, block diagram

1.1 Test Bench Constraints

Due to schedule constraints, a requirement for the new test bench was that we reuse a USB 2.0 VHDL test harness. This test harness implemented the entire USB protocol from basic signaling at the bit level to the higher-level protocols. It was needed to produce the USB traffic for the new IP block. The problem was that this test harness was large, complex, and not extensible. It worked well for testing only the USB 2.0 core, but would be difficult to use in our environment.

Another requirement for this test bench was that it was to be written using Synopsys' Hardware Verification (HVL) language, Vera. We wanted to use Vera because we did not have an HDL model of the AMBA AHB bus and without it we would not have a back end interface. We did

however have one in Vera. Another decision that influenced our use of Vera was that as a company we were beginning to move to Vera as our HVL of choice.

Before writing any code for the test bench we decided to first model what we needed and how everything was to fit together. This decision was influenced by past successes with modeling in our company. Modeling techniques had first been used in our company to help in the construction of inkjet controller firmware. The firmware that was produced in that project was robust, extensible, and well documented. The modeling language used was the Unified Modeling Language (UML). Since UML had been successfully used in the past, it gave us more confidence to proceed with it in this project.

1.2 Roadmap for the Paper

Before going into a description of how we used UML to help model the test bench and how it worked, we need to cover some foundational concepts first. We will start off with an explanation of why we model at all. Two different modeling paradigms, structural and object-oriented will be discussed. Next we will give a brief explanation of UML and how it integrates with object-oriented modeling concepts. Then a tool that automates the use of UML will be introduced. We will describe some of the features of Vera and how it integrates with UML. We will next give a description of how we modeled the test bench and solved some integration problems. Finally we will discuss the results and future possibilities.

2.0 Modeling

2.1 Why we model

The concept of modeling is nothing new per se and is an accepted practice in many different industries. In the aerospace industry before a new plane is built, many different models of various types are generated. First there will be a multitude of computer simulations written to describe how the plane will react in situations that it is likely to encounter, i.e. wind, rain, snow. The next step is to create to-scale physical models of the plane that will be produced and test them to acquire another view of how the plane will react. This all occurs before the plane is even built. In the construction industry, no one would attempt (or even be allowed) to start a high-rise building project without first having detailed blueprints of the building, describing everything from the electrical system to the internal skeleton of the building.

The main reason for modeling in both of these examples is that these endeavors are so complex that they cannot be understood in detail all at once. Modeling allows one to figuratively break the plane or building into pieces that can be viewed and understood, one aspect at a time. We make many models because no one model is sufficient to describe the entire system. Each of the different models gives a different view that may be analyzed and understood.

Software, while it is very complex, is at times constructed without any formalized modeling, or any modeling period. (A point for clarification may be needed here before continuing. When speaking of software this does not necessarily mean some end user application. Software in this context is meant to describe the verification tests that are written to test hardware, regardless of

the test language.) Creating software without modeling can result in software that might not satisfy testing requirements, is difficult to understand and extend, or both. This might mean that the ASIC you are charged with testing will not receive enough testing or the proper testing. We will now contrast two different types of modeling that are commonly used.

2.2 Software Modeling

Two of the most common types of software modeling are structural and object-oriented. Each modeling technique produces software that has certain characteristics and predetermined attributes. We will discuss each software modeling technique in turn and show why we think object-oriented software modeling is the better of the two. We are not here to argue that there is anything inherently wrong with structural modeling and the code that it helps to produce. Instead we believe that it is time for the evolution from structural modeling to object-oriented modeling to proceed.

2.2.1 Structural

Structural modeling is the older and more prevalent of the two types of modeling. It is a methodology whereby a problem is broken into smaller, more easily understandable pieces. It is also known as top-down or action-oriented modeling.

One of the shortfalls of code produced using structural modeling is that it is not easily extensible. Extensibility in this case means that changes can be made to the code in a relatively straightforward manner. If the code is extensible, this increases the chance that it will be reused in another project.

One reason that structural software is not easily extensible is that data is not incorporated with the functions but instead is a separate entity. Thus, though you may have carefully partitioned the functions and have a very logical flow of control, it can still be the case that the data structures are strung haphazardly throughout your code. If there is a change in a data structure that is used by many routines, that change has to be made throughout the whole program. The chance of the change being made without error decreases as the size of the software program increases. This leaves the developer with the opportunity to spend “quality” time debugging the problem.

A second shortfall of structural modeling is that it lacks the concept of concurrency and will not easily produce software that can handle concurrency. This is a serious shortcoming in hardware because there are concurrent processes and the verification software should have the ability to test and respond to the concurrent processes.

2.2.2 Object Oriented

Object-oriented (OO) modeling overcomes the extensibility and concurrency problems present in structural modeling. It is more extensible because it groups data and functions together into one unit and the concept of concurrency can be more easily built in.

The grouping of data and functions into a unit is called a class. In a class, there is a set of well-defined public interface functions that are used by the outside world to interface with the class. The data and any other functions are not accessed by the outside world. So unlike structural software, there are no central data structures used by many functions. Each class contains the data it needs along with the functions to act upon that data. If a change needs to be made to the class and the public interfaces are not changed, it should be transparent to users of the class.

Software produced by object-oriented modeling is composed of instantiations of classes called objects. Since an object is a grouping of data and functions as one unit, one object could potentially be run separately from any other object. Therefore object-oriented modeling can represent concurrency in a hardware system. A side benefit to this is that since objects can run as individual entities, they also have the potential to be tested as individual entities.

Now that we have given a brief explanation of why we think object-oriented modeling is preferable, we will now discuss a graphical language for constructing the object-oriented models. The language is the Unified Modeling Language and it is designed to assist with modeling object-oriented systems.

2.3 Overview of UML

UML is a language for creating models of object-oriented systems. With it you can specify, document, visualize, and construct object-oriented software. The different model views that UML provides are used to communicate to customers and other engineers what the software is to do and how it is to do it, all in a standardized way.

Since UML is a graphical language, one of the main means of communicating what you are modeling is with diagrams. We will now briefly discuss a few of the different diagrams that are available in UML and what they are used for. The UML diagrams that we will be discussing are the use case diagram, the sequence diagram, and the class diagram. There are other UML diagrams that may be used that we will not be discussing in this paper.

A point of clarification may be needed here before continuing. UML is a language and not a process. What we mean is that UML does not tell you how to go about creating the models of the software system, it is a language for communicating what the software system is going to do and later how it is going to do it. There are several processes such as the Rational Unified Process (RUP) and Rapid Object-Oriented Process for Embedded Systems (ROPES) that can be used and which tie in well with UML. Therefore when a diagram such as a use case diagram is presented, it is shown as a method of communicating information, how it came about is not discussed.

For a more extensive explanation of UML and processes there are many good books written upon the subject. See the appendix for a few that we have used.

2.3.1 Use case diagrams

Use case diagrams are UML diagrams used mainly in the early stages of the design analysis. They describe what the system under design is to do and how it interacts with the outside world.

The use case diagram stipulates what a system is going to do without specifying how the system is going to do it. Thus the use case diagrams can become a nexus where it can be determined whether the requirements for the system are sufficiently specified or not.

Shown below in figure 2 is an example use case diagram. In the UML use case diagram, two of the primary icons are the use cases themselves and the actors. Use cases are depicted as ovals and delineate what the system is going to do. An actor is represented by a stick-man and is outside of the system under consideration but interacts with the system. An actor can be a person interacting with the system or another piece of hardware. Arrows drawn between actors and use cases are called associations. The associations describe the interactions between the actors and use cases.

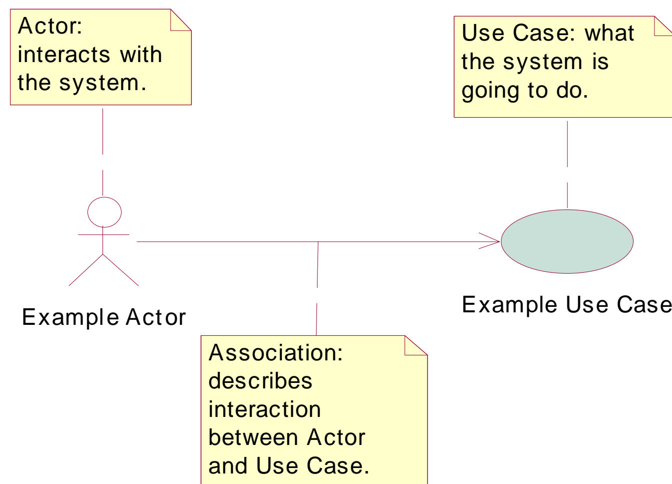

Figure 2, Use Case Diagram

In figure 2, we have shown one example actor, association, and use case. There of course can be as many actors and use cases as you need to model your system, with necessary associations. Figure 2 is a contrived example to show some of the salient points of a use case diagram.

2.3.2 Sequence diagrams

Sequence diagrams are UML diagrams that can be used in the early analysis phase with the use case diagrams to describe scenarios of the system. Scenarios communicate a particular interaction between an actor and the system, involving order dependency and message flow. Using scenarios is a way to expand upon the behavior of the system. For each use case there is a primary scenario, where things work correctly, and permutations of the primary scenario, where they do not. By using the sequence diagrams to show the use case/actor interaction, you can potentially uncover new requirements and be reasonably sure that the expectations for the system under consideration are met. Figure 3 shows an example sequence diagram.

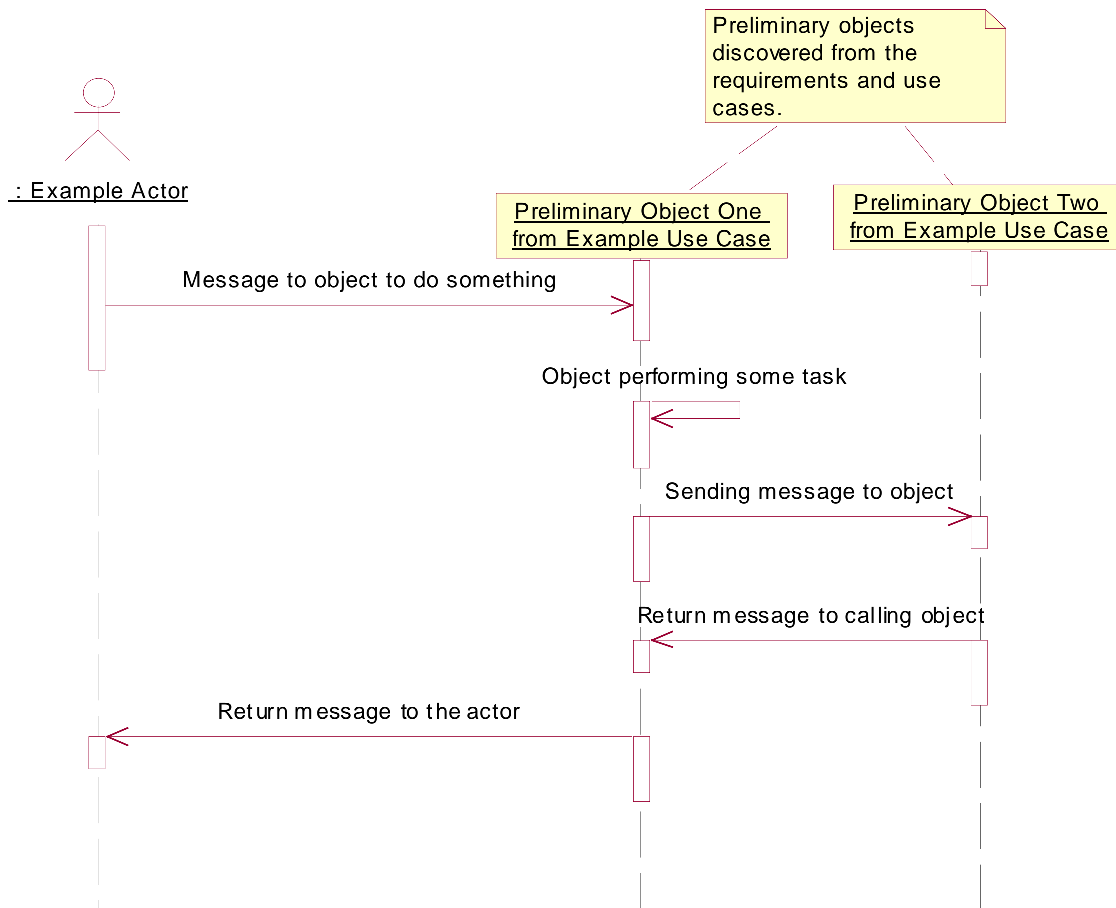

Figure 3, Sequence Diagram

Sequence diagrams can later be used to refine the interactions between discovered objects. You use sequence diagrams with the discovered objects to make sure that the objects have all of the functions necessary in them to do what is needed. You can also discover missing objects this way.

2.3.3 Class diagrams

The last UML diagram we will look at is the class diagram. To understand what the class diagram is for, remember back to the use case diagram. In the use case diagrams we specified what the system was going to do and how it was to interact with the outside world. In the UML class diagram we have broken up the "to-do list" of the use cases into tangible pieces of responsibility. Each class has certain operations that it knows how to do and also has the information necessary to perform the operations. The interactions between the classes are delineated here, which class will interact with which class and what that relationship will be.

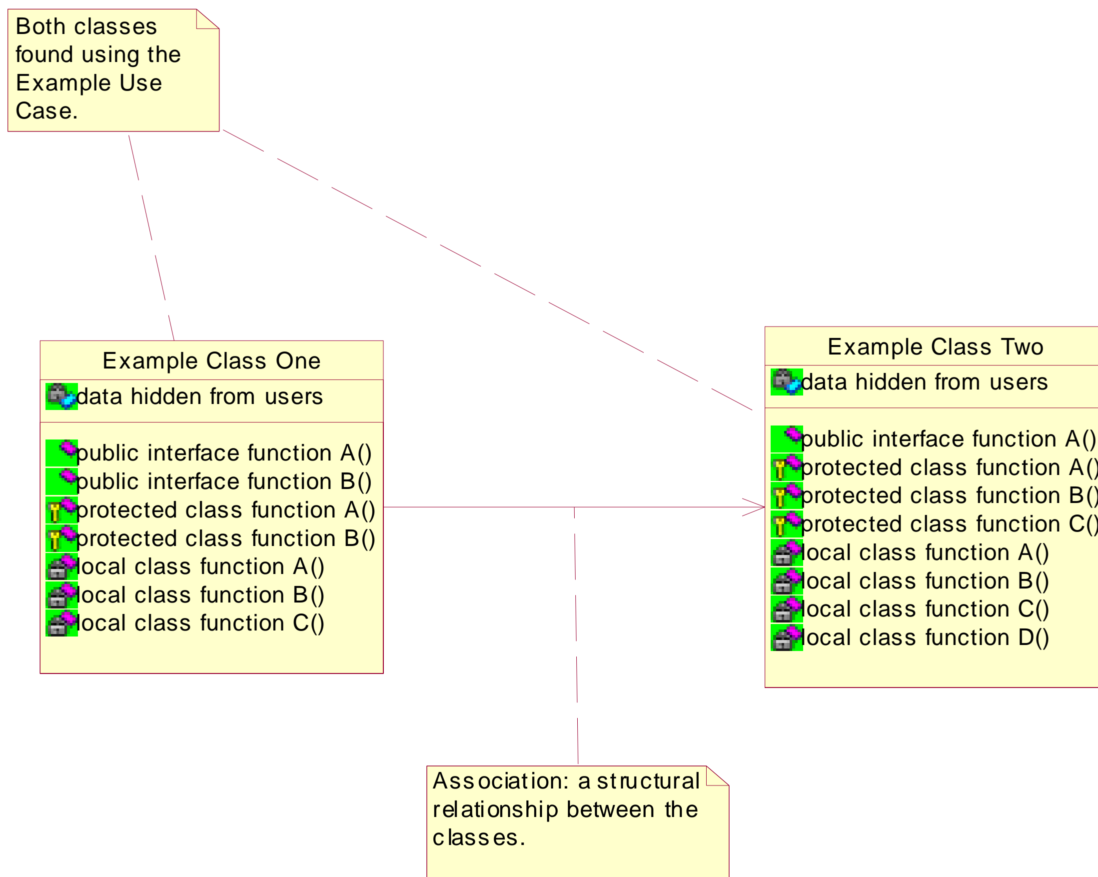

Figure 4, Class Diagram

The fully specified class diagrams provide the point where you can now start actually writing the software for your project.

As mentioned before, UML is a graphical method to represent a system. All of the different UML diagrams that we have mentioned -- use case, sequence, and class -- are different graphical views into the system. While it is true that you could draw the different UML diagrams by hand, this is not necessary as there are several UML based visual modeling tools that will generate the UML diagrams for you. The tool used on this project was Rational Rose 2000 from Rational Software.

3.0 Rational Rose

Rational Rose is a UML based visual modeling tool. The drawings that were presented for the use case, sequence diagram, and class diagram were all created using Rational Rose C++ edition. Unfortunately at this time they do not have a Vera edition, but with some small modifications, the C++ edition of Rose proved to be quite capable of creating all of the diagrams for our verification test suite.

When you start with Rose, the programming language does not matter. You are modeling with UML. The choice of programming language comes in when you have your class diagrams completed. You can then set your programming language, C++ in this case, and have Rose generate modules with your classes in them. The classes are shells, code stubs, which you fill in with your verification code. You also have the ability to add functions or attributes to your class diagrams, and regenerate the shells. Any work you have done is saved and the new portions are placed in automatically.

As stated before it is a C++ version and not a Vera version, so there are some things that must be dealt with when it finally comes time to compile the code. When it comes time to compile, you will need to remove C++ specific artifacts. These can either be done by hand or automated, with your choice of scripting language. At this point if you use Rose to add functions or attributes, you will have to strip out the artifacts again. It is a minor inconvenience for the advantages that you receive.

4.0 Vera as a Hardware Verification Language

While HDLs have been successfully used to write test benches, certain things are hard to test. For example, concurrency is hard to control in VHDL, since different processes cannot drive the same signals. Using an HVL such as Vera has many tangible benefits. High-level data structures are available. Items such as concurrency and randomness are built into the language, and tests can be created quickly.

4.1 Object oriented properties

Vera is class-based, which means that we can concentrate on creating test benches that accurately describe our hardware. As mentioned before, we desired to use UML to build the models, and Vera is easily extensible in that way. All of the important concepts for abstracting in UML, such as classes, objects, and inheritance, are supported.

Another benefit to using OO methods and UML is the increased chance of reuse. For example, a test bench may be designed so that core tests are kept separate from specific block interfaces. In this case, a USB test bench may have one class that defines parts of the USB protocol and how to run USB tests. Another class would understand the specific hardware being tested and make any modifications needed for that block. At this point, the heart of the USB test bench need not change for each new block, only a new interface class is required.

4.2 Concurrency built in

The ability to control concurrency greatly strengthens a test bench. For an application such as USB, a test bench can be created which has many objects working in a client/server mode. Several different objects act as the clients and request packets to be sent. Once the framework is in place, adding more clients is trivial. The same code block can be the basis for launching several independent parallel threads. This way, the hardware can be stressed just as it will be in the real world.

Vera offers rich support to control concurrent processes. Processes can communicate through shared variables and mailboxes. Mutual exclusion is accomplished via semaphores. Threads are controlled with fork/join statements. Each thread invocation causes local variables to be created, ensuring thread isolation and reentrance.

4.3 Integrates with both VHDL and Verilog

For projects that use mixed languages, Vera offers reasonable methods of communication. Vera can be built within a VHDL or Vera test bench, call VHDL, Verilog, or C functions, and be called by VHDL, Verilog, or C functions.

5.0 Modeling the Test Bench

In the beginning of the paper it was stated that we needed to design a test bench for a new IP block. This IP block had as its heart a USB 2.0 core, supporting logic that included FIFOs, memories, and registers, and a DMA engine. We also had the constraint that we needed to reuse the legacy USB VHDL test harness and use the Vera AMBA AHB models.

This will be a brief explanation of the process that we used with UML to model this test bench. This section is not meant to be a tutorial on the subject. Good references are given in the appendix.

Rational Rose was the tool used to create the use case diagrams, sequence diagrams (scenarios), and class diagrams.

5.1 Use Case Diagrams

In section 2.3.1 we stated that in a use case diagram you specify what your system is going to do, but not how it is going to do it. You only specify how a use case is to do something if there is a predefined requirement that you must live with. An example of this would be our legacy USB test bench. It was a requirement that we use it to provide USB traffic to our IP block. So this predefined requirement dictated how USB traffic was going to be fed into our IP block.

Keeping this in mind we started defining what our use cases were by writing down the requirements that we were given for the verification test. From there we could group the requirements into actions that the verification test would have to perform. We could also find the actors by observing who or what would be interacting with our system.

5.2 Scenarios

Once we had the use cases we could proceed with defining the scenarios of the system and classes. Remember that the scenarios are the interactions between the system and the actors, what can go right and what can go wrong. Here you discover the messages and interactions that must occur between actors and the system and also the messages and interactions between different objects in the system.

One constraint that we discovered during this process was that it was going to be difficult to communicate with the legacy VHDL test bench. We originally wanted to leave the VHDL test bench alone completely and simply call higher-level tests from under the control of the Vera test bench. But the presence of global variables and impure functions made this impossible. At this point we considered many alternatives. The two main areas we explored were how to make the Vera and VHDL pieces communicate and what other ways we could model AHB. Since modeling AHB in Verilog, VHDL, or C was both time consuming and took us farther from our eventual goal of having reusable Vera pieces, we concentrated on fixing our communication problem.

We thought of adding several signals to the VHDL, which Vera could drive as a sideband bus for control. We also investigated creating a Vera interface to drive signals directly. But what we finally decided on was to use a simple file interface. In this scheme, the Vera test bench will write text files with test configurations and data, and then signal the legacy test bench to begin. The legacy test bench will read and parse the text file and run the requested test. This is illustrated in the figure below.

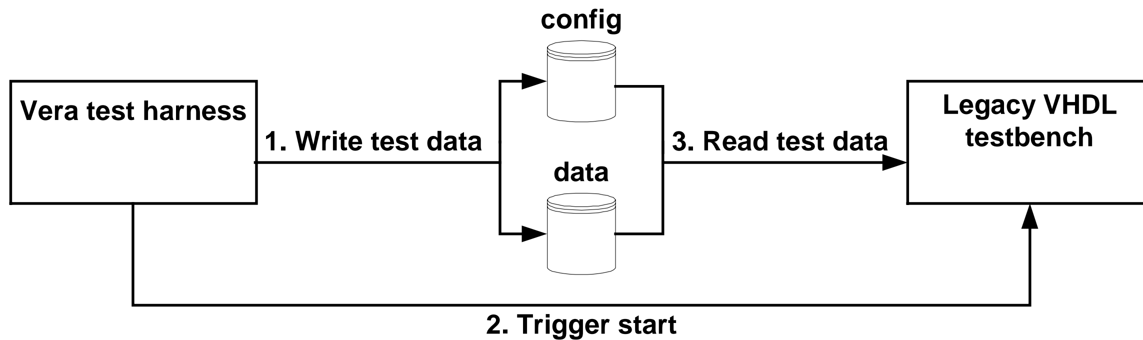

Figure 5, Start Test

This scheme will not afford complete reuse of the legacy test bench. For example the legacy test bench may need to change the device configuration before the requested test may be run. So special routines will be added to do any needed preparation and then run the existing routines. At completion, the legacy test bench will write the test results and signal that it is finished, as shown in the figure below. At this point, the Vera controller will read the results, compare data, and signal any errors. While we were not able to exploit all of the features of Vera that we desired, this allowed us high degree of reuse while providing a simple communication path.

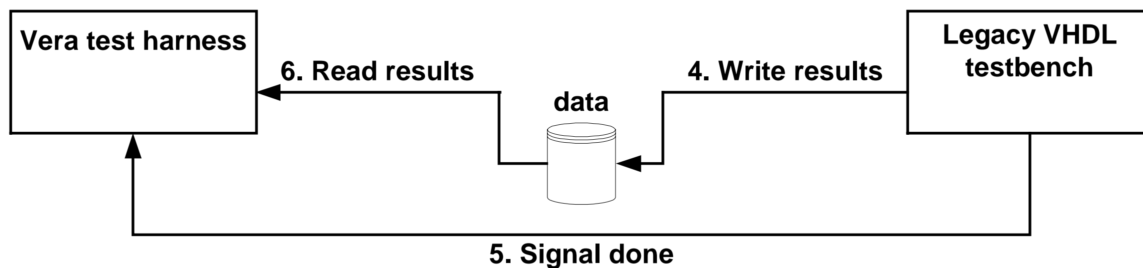

Figure 6, Test Finished

The results from the scenarios were fed back into the use case diagrams and they were modified accordingly. We had not as yet written any code but had already discovered and fixed a potential problem.

5.3 Object Identification and Class Diagrams

Identifying the objects can be done using several different methods. You can use the requirement documents that you created while producing the use cases and underline the nouns. These nouns are then potential objects. You can also use the scenarios to discover missing objects. As an example, you might start with known objects and try to complete a scenario. If you cannot, there is a good chance that you have missed an object. These and other methods overlap in discovering objects. Also during this process you discover the methods and attributes of the objects. The methods will represent in the domain model the messages between objects, and the attributes can be the state that the object needs to preserve.

Once the objects are discovered, classes are abstracted from them. Abstracted in this sense means that common objects are found and grouped into units called classes. These classes are used to construct the class diagrams that we saw in section 2.3.3. The relationships between the classes are discovered by determining the relationship between the objects that the classes came from. Once we had the class diagrams in Rational Rose we could generate the code shells and fill them with the Vera code.

5.4 Writing the Verification Code

A benefit of modeling is that when it comes time to fill in the code shells with the actual code, it is relatively simple. This is because you have already thought about what you are going to do. You have come at the problem from many different angles and defined what each object is going to do and how it will interact with other objects. You can now write each function one at a time, concentrating on it alone. You do not need to worry what the interface will be, or how it will interact in the system. This has already been done. You also have the UML diagrams to look at (use case, sequence, etc.) if you do need to step back and “get the big picture”.

6.0 Results and Conclusions

We found that the time we invested in using UML to first model our Vera verification tests, and then using the UML diagrams to help us write the verification tests was beneficial. We will now discuss three benefits we found from using UML.

As stated in section 5.4, the first benefit came about when it was time to actually write the verification code. We found that the process of writing the verification code was relatively easy.

The second benefit can be found in section 5.2. There, by first modeling the system we found a potential “show-stopping” difficulty in communicating with the legacy USB test harness. We were able to fix the problem without major rework.

The final benefit is the built in documentation. For example say one year from now, classes from this project are going to be used in another project. If we did not have the UML class diagrams, our only recourse for discovering what the classes do, what the interfaces are, and how they interact is to read the code. But with UML class diagrams, we can glance at visual class diagrams and discover interfaces, functions, and data. We have the sequence diagrams to describe how the classes were designed to be used with other classes. When we do actually need to look at the code and use it, this will be a much less painful process.

7.0 Future Plans

7.1 Vera

Vera was brought into our company about a year ago, and the different reactions have been interesting. Some designers took the attitude that our test benches have always been done in VHDL, and should always be done in VHDL. Others were guardedly optimistic. Now that we have some experience with Vera, it seems that most people have been very pleased with the ease of use and how efficient it is compared to writing test benches in traditional HDLs. While we have not as yet had an opportunity to test this, we can envision a much greater chance of reuse. Creating a test bench may go much more quickly as the number of Synopsys VIP models available increases. As we gain more experience with Vera, we receive greater management support on expanding Vera's role in future projects.

7.2 UML and Rational Rose

We found that Vera, UML, and Rational Rose integrated well together and we plan to continue to use this combination to develop our hardware verification tests. As was stated in the results section, during the development of this verification test, the requirements changed and the object-oriented paradigm was able to handle these changes and integrate them. Another bonus is that documentation is part of the process. The one item on our wish list is that Rational Software will produce a Vera version of Rose.

7.3 UML with Hardware Description Languages

In the future we will explore using UML and Rational Rose with a hardware description language like Verilog or VHDL. This is natural extension if you consider a present day ASIC, a million gate ASIC is not uncommon. That is a great deal of complexity to deal with and the gate count will only continue to rise. It would seem only natural that you would want to model the ASIC for the same reasons they blueprint a building, because you cannot understand the whole ASIC in detail.

The object-oriented concepts of UML also map well with the concept of partitioning in the ASIC. You could break up the ASIC into modules of functionality with well-defined interfaces. These interfaces allow use of the modules functionality without specifying how the module performs the function. The module can therefore change how it does things without affecting the users of the module.

The flow for using UML with an HDL would probably proceed somewhat similarly to that used to model the verification tests. You would still create the use cases based upon the initial requirements, go through the scenarios, discover the potential objects, and abstract them to classes. All of this is language independent. There is of course the difference that Verilog and VHDL do not have classes as Vera does. What this will mean is that you will be creating modules of code and not classes. Therefore you will have to be certain that you do not model any object-oriented constructs that are not supported by the HDLs.

7.3.1 UML Used to Design Systems

Often a system is built by first designing a hardware architecture, then creating a test bench, and finally figuring out what firmware and software will be required. Many times a difficulty found in the firmware or software would not have existed if the whole system architecture was designed from the start. Some decisions that are almost arbitrary to the hardware can have huge impacts on code execution speed. Even if a better way of partitioning the system becomes apparent early in a project, usually it is too late to redefine the scope of the hardware.

Once UML tools allow generation of HDLs and HVLs as well as the traditional high-level languages, then a whole new avenue of system level design opens up. UML is meant to model systems, and “system” is not confined to just software or test bench design. So instead of drawing up the hardware block and software block and trying to work on each piece separately, why not design the whole system in UML first? Then, using criteria such as desired chip size, speed, and price, partition the system into the optimum combination of hardware and software.

8.0 Appendix

M. Fowler, “UML Distilled”, Addison-Wesley, 1997.

G. Booch, J. Rumbaugh, and I. Jacobson, “The Unified Modeling Language User Guide”, Addison-Wesley, 1999.

J. Rumbaugh, I. Jacobson, and G. Booch, “The Unified Modeling Language Reference Manual”, Addison-Wesley, 1999.

B. P. Douglass, “Real-Time UML”, Addison-Wesley, 2000.

A. J. Riel, “Object-Oriented Design Heuristics”, Addison-Wesley, 1996.

F. I. Haque, K A. Kahn, J. Michelson, “The Art of Verification with Vera”, Verification Central, 2001.

P. Kruchten, “The Rational Unified Process: An Introduction, Second Edition”, Addison-Wesley, 2000

OpenVera 1.0 Language Reference Manual, March 2001.