Development of a Microprocessor Verification Environment in Java

Functional verification of microprocessors by means of logic simulation is a subarea of electronic design automation (EDA). A brief history of EDA is provided here. The Verilog hardware description language (HDL) was invented in 1984.[1]

Java is well known for many things, but not usually as the language for writing a microprocessor verification environment. Nevertheless, I have found Java useful for such a task, not only for the test bench, but also as the HDL of the design under verification (DUV) itself.

In this article, FSS is used as a case study, but it is also a commercial product under development by Cosmic Horizon.[2] I will highlight the development of FSS, a software application, for this audience with the intent to open discussion.

There are three related ongoing projects at Cosmic Horizon:

> Project 1: Development of FSS
>
> Project 2: Development of the Sputnik microprocessor
>
> Project 3: Architectural verification of Sputnik using FSS

FSS is a SPARC simulator. If a user has a microprocessor design that needs to be verified, one that is intended to comply with the SPARC Version 9 (SPARC-V9) architecture, FSS will verify that design architecturally.

Project 2 is necessary because FSS needs to be tested, and I know of no other SPARC-V9 design written in JHDL (see More about JHDL). FSS is unique, but it also has a drawback. FSS is part of a system that is written entirely in Java, written so that it can run anywhere.[3] But the DUV must be modeled in JHDL. The Sputnik microprocessor is a relatively simple design, implemented in JHDL at the register transfer level (RTL) of abstraction. It aspires to comply with the SPARC-V9 architecture. Sputnik can be simple because it exists solely for the purpose of exercising FSS. The job of FSS is to find SPARC-V9 rules violations in Sputnik and other DUVs.

Functional verification of hardware designs is a software engineering discipline. There are best practices from software engineering that should be embraced. Software application development begins with a Software Requirements Specification (SRS), such as the FSS SRS. All of the commercial EDA tools (e.g., simulation engines) that verification engineers use began with an SRS. But what about the software that verification engineers develop? Typically, the simulation engine already exists, and a team of verification engineers is tasked with writing a test bench, which is itself a software project. There should be a verification plan, which is different from an SRS, to drive a chip verification project, like Project 3.

In this article, I explain where FSS fits in with the activities of a microprocessor verification team. Next, I describe the development of the system from a software engineer's point of view. That development, as proof that Java is useful in this field, is the focus of this article.[4] I will include additional information about JHDL, and I will then address the ramifications of language choice. There is a trend toward the use of specialized hardware verification languages (HVLs) in writing test benches.

---

1 http://www.edac.org/htmfiles/KaufmanAward/PhilMoorby.htm

2 Cosmic Horizon is a sole proprietorship and I am the owner. Collaborating with me is Dr. Bimal Mishra, an expert in applied mathematics with interest in science and technology.

3 anywhere there is Java Runtime Environment (JRE) Version 6 or higher

4 The customer would be developing a SPARC-V9 microprocessor model in JHDL, which is also Java.

If a programming language is used instead, typical features of HVLs need to be addressed. Finally, I cover a couple of miscellaneous topics from eating one's own dog food to the problem of modeling a huge address space.

TEAM ROLES & RESPONSIBILITIES

A microprocessor verification team is responsible for verifying many functions. I will not list the function categories; the verification plan provides a detailed function list. For example, consider block replacement on a miss in an associative cache. For the sake of illustration, suppose the functional specification states that a least recently used (LRU) strategy is to be employed. This function will be included in the verification plan, and the team would then write software to check that the cache correctly implements the LRU strategy. Architectural verification is another category altogether. SPARC-V9 does not require cache (e.g., Sputnik does not have cache). Although the SRS has a few other features (e.g., measurement of instructions per clock), the main purpose of FSS is to verify that the DUV complies with the SPARC-V9 architecture, so the question of roles and responsibilities in the remainder of this section pertains only to architectural verification.

FSS is different from from a typical C/C++ library for functional verification. A major difference lies in team roles and responsibilities. Rather than provide users with an application program interface (API) so they can write a test bench in a high-level language and access FSS's simulation engine, FSS provides much of the test bench[5]. The users provide the SPARC-V9 programs that stimulate the DUV. FSS knows how to check that the DUV properly executed those programs. That is, FSS provides the checkers too. So the dividing line between the commercial EDA tool and the intellectual property developed by the verification team has shifted. Nevertheless, the verification team is certainly engaged in software engineering. And so is Cosmic Horizon, as it is developing a significant part of a microprocessor verification environment.

The testing environment and the test sequence must be designed. Consider the testing environment to be FSS, which is provided by Cosmic Horizon. The test sequence has the following three major components:

1.Clock method of the FullChipTestBench Java class

> FullChipTestBench, provided by Cosmic Horizon, is the HDL test bench. Like a microprocessor socket on a motherboard, it surrounds and connects to the ports of the DUV. The clock method of this class is called by the simulation engine after each clock phase. The method defines a sequence of actions including assertion of the reset signal. It also drives data to and from memory.

2.Bootstrap loader

> Cosmic Horizon provides the bootstrap loader in the read-only part of the AddressSpace class[6]. The bootstrap loader is a SPARC-V9 program that takes the DUV from RED_state to the beginning of a SPARC-V9 test program.

3.SPARC-V9 test programs

> The verification team creates these architectural verification programs. See Constrained Random Generation for a discussion of Project 3 and how future work may affect roles

5 The test bench is like a motherboard, except that it exists for the purpose of functionally verifying the microprocessor design. FSS provides the reusable test bench, but an “adapter” will be needed so that an arbitrary SPARC-V9 microprocessor can be accommodated by the virtual socket. The adapter may be accomplished using a dynamic mapping mechanism whereby an input file supplies the test bench with a signal mapping to the port names of the DUV.

6 in the RED_state trap vector; from power-off a SPARC-V9 microprocessor starts in Reset, Error, and Debug state

and responsibilities.

DEVELOPMENT OF FSS

Software Development Process

The structured approach of the waterfall model is appealing. “A requirements document ... specifies the overall purpose of [FSS] and *what* it must do. Throughout the [development of FSS], we refer to the requirements document to determine precisely what functionality the system must include.”[7] As suggested by Michael Leisch in his article, Application Development: Methodology Tools, a project with stable requirements is a possible match for the waterfall model. FSS is a relatively stable project, because SPARC-V9 is not changing. Furthermore, Cosmic Horizon has failed to engage its user community in a two-way dialogue, which is admittedly a bad thing. But the glass is half full: Instability caused by changing user requirements is almost nonexistent.

Although the waterfall model may seem appropriate for this project, I have found the model to be impractical. Project 3 makes Cosmic Horizon's verification engineers members of the FSS user community. Feedback forced a change in the SRS after the software requirements phase; implementing a change violates the waterfall model.[8]

The change to the SRS was required because I was unable to understand a failure in Sputnik's integer multiplication unit using FSS's debugging features alone. The SRS required that FSS be a cycle-based simulator. Therefore, FSS could not display any intra-cycle information about signal value history on a wire. For the particular failure that I was attempting to debug, I found it necessary to view this information. Sputnik requires a four-phase clock schedule,[9] and FSS is intended to verify *any* SPARC-V9 implementation. Therefore, FSS must be (and had been) able to drive such a clock, but viewing each phase was another matter. The SRS cycle-based requirement was in conflict with the users' need for phase-based information.

To prevent the complete destruction of the structured approach, I sought a model that minimizes the deviation from the waterfall model. The sashimi system[10] is similar to the waterfall model, but it allows a partial overlap among development phases. Moving to sashimi allowed me to justify a change in the SRS to support phase-based simulation.

Software Design Phase

To promote loose coupling among subsystems, FSS adheres to the Model-View-Controller (MVC) architectural pattern shown in Figure 1. SparcSimulation is the top-level class of the model subsystem. This subsystem is the simulator logic, which represents the operation of the SPARC-V9 system. All calls to the JHDL library are confined to this subsystem. SparcView is the top-level class of the view subsystem. This subsystem is the display of the simulator on screen (so users can view it graphically). SparcController is the top-level class of the controller subsystem. This subsystem is the graphical user interface, which allows users to control the simulation.

---

7 Java: How to Program, Sixth Edition

8 Dr. Winston W. Royce described the waterfall model in “Managing the Development of Large Software Systems” (Proceedings, IEEE WESCON, August 1970). His Figure 2 clearly illustrates that there is no turning back once a subsequent phase has begun.

9 See the explanation of clock schedule in JHDL. A “cycle” corresponds to the entire schedule String passed to the byucc.jhdl.Logic.Logic.clockDriver method.

10 Takeuchi, T., and I. Nonaka, 1986. “The New New Product Development Game,” *Harvard Business Review*, January-February 1986.

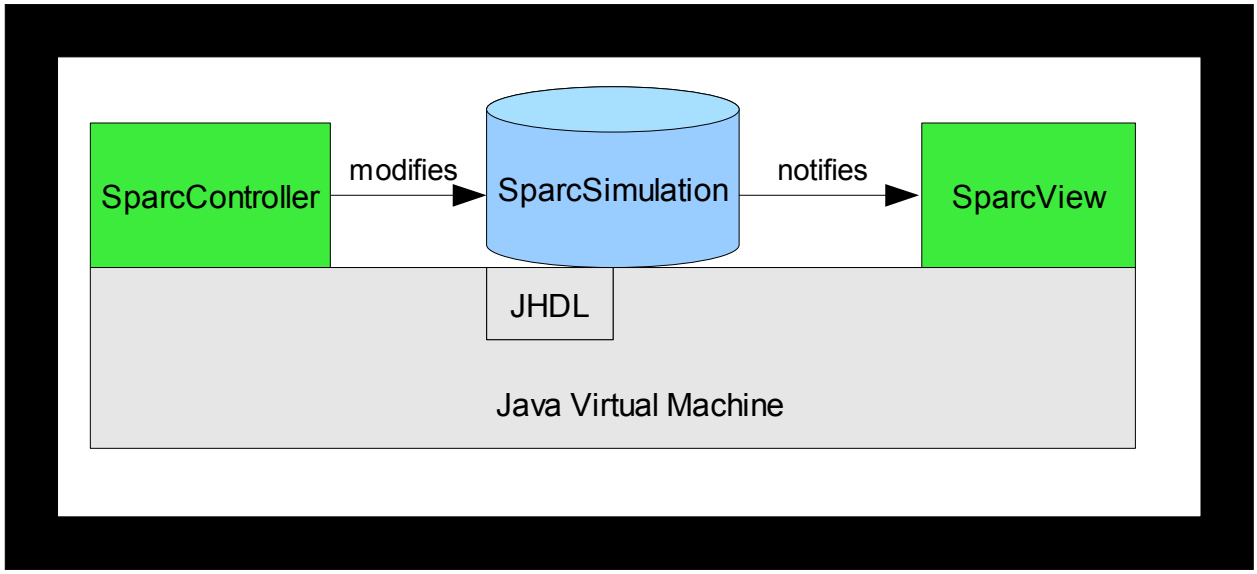

Figure 1: Model-View-Controller (MVC) Architecture

In the design phase, Cosmic Horizon uses UML 2.0 to model FSS structure and behavior. The specification defines 13 diagram types. Commonly used ones are highlighted.

- Use Case Diagram
- Class Diagram
- State Machine Diagram
- Activity Diagram
- Communication Diagram
- Sequence Diagram
- Object Diagram
- Component Diagram
- Deployment Diagram
- Package Diagram
- Composite Structure Diagram
- Interaction Overview Diagram
- Timing Diagram

Our chosen UML tool is MagicDraw, which runs on "any Java 5 or 6 compatible virtual machine."

Revision Control Systems

I recommend *Comprehensive Functional Verification: The Complete Industry Cycle*, hereafter called CFV, to anyone interested in functional verification. However, revision control systems are barely mentioned. The authors write that

> "the added complexity of [a particular DUV] requires a programming or high-level verification language (HVL) infrastructure. If multiple people share the verification effort, this infrastructure must include a code revision tool (RCS, CVS, ClearCase, etc.). This allows the verification team to control the data management of the environment code via *check-in* and *check-out* from a centrally shared repository."

Not only does the book lack emphasis on revision control systems, but I would not limit their applicability to cases of multiple verification engineers. A revision control system is required, whether

or not multiple people share the verification effort, as its benefits go beyond management of multiple developers in the same source code. For example, every time developers commit, they should supply a descriptive log message. If problems arise, that history makes it easier to return to something that works. A manager should not be satisfied with the mere fact that revision control is in place; the system must also be used correctly. All deliverables should be delivered by means of committing to the repository and by no other means. The same principle applies to the sharing of code among team members. In my experience, I have observed that branches are used far too infrequently. For example, branching a project is appropriate for experimental work. Verification engineers are never forced to choose between (a) doing extensive development in their sandbox without the protection of revision control and (b) committing to the trunk. Whether working on a branch or not, if no code is checked in until the end of a project, the revision control system is being misused. Although the sandbox might be backed up, committing to the repository at milestones would give the verification engineer a number of *documented* fall-back positions. Revision control also facilitates reproducibility, and software reuse with a little forethought. It has the potential to vastly increase the efficiency of the verification team.

Cosmic Horizon uses a revision control system, but I will not compare strengths and weaknesses of competing systems in this article. The revision control system benefits a single developer, and for multiple developers, access to a central repository is essential when developers are working in different cities or traveling.

MORE ABOUT JHDL

JHDL is an HDL, an alternative to VHDL or Verilog. “Dr. Brad Hutchings [of Brigham Young University (BYU)] ... initiated what came to be the JHDL project in 1997.”[11] Hardware that is described in JHDL is actually Java code that uses the JHDL API. All JHDL is Java. The reverse, of course, is not true. One of the benefits of implementing the DUV in JHDL is that there will be no language barrier between the DUV and the verification environment. It is all Java. Another benefit is that the *entire* system is portable.

From a software architecture point of view, JHDL is a Java class library used by FSS and its DUV. The full JHDL software has some functional verification features that could be applied to a JHDL microprocessor model, but FSS is designed specifically for microprocessor verification. In fact, FSS is even more specialized than that, intended solely for SPARC-V9 architectural verification.[12] Cosmic Horizon restricts its dependency to the core of the JHDL software (i.e., the simulation engine and the modeling primitives).

Cosmic Horizon has created a derivative work. We only make changes to JHDL that benefit everyone, nothing FSS specific. Cosmic Horizon JHDL is simply a better JHDL. A good place to start learning more about JHDL is the Cosmic Horizon JHDL main page. Links to BYU JHDL are provided there.

WHY JAVA?

The answer depends on the meaning of the question. If it is to ask why I chose to develop FSS/Sputnik in Java/JHDL, I made those decisions before Cosmic Horizon was formed. The language choice story is not an example of great engineering, and I will not be addressing it in this article. Interested readers are referred to my Origin blog item. On the other hand, if the question is why a user might consider running simulations in Java and implementing a microprocessor in JHDL, I will address that question

11 Nelson, B.E., "The Mythical CCM: In Search of Usable (and Resuable [*sic*]) FPGA-Based General Computing Machines," Application-specific Systems, Architectures and Processors, 2006. ASAP '06. International Conference on , vol., no., pp.5-14, Sept. 2006

12 Dr. Mishra is encouraging extension to other architectures.

in this article.

Jove is a Java library for general-purpose hardware design verification. Jove requires the DUV to be modeled in Verilog. Jove cannot verify Sputnik or any other design implemented in JHDL.

The most compelling argument for running microprocessor simulations in Java is probably its portability. If the customer wanted to change computers on which simulations run, a well-designed microprocessor verification environment implemented in Java will run unmodified on any platform. Another reason for Java is to keep the verification team productive. On occasion, access to a network can be a challenge. Engineers work from home, coffee shops, airplanes, cars and cabins in the woods. Allowing simulations to be run locally without regard to the computer type empowers verification engineers to work from anywhere. Portability also opens microprocessor simulation to independent researchers, again regardless of computer platform.

Objections to Java

A customer may object to implementing a microprocessor in JHDL rather than Verilog or VHDL. First, that may not be necessary. Cosmic Horizon is currently working on a project that automates the translation of Verilog RTL to JHDL. The automated translation will allow an existing SPARC-V9 Verilog RTL implementation (e.g. OpenSPARC) to be architecturally verified by FSS, using an equivalent JHDL implementation.

Second, for new microprocessor projects (i.e., lead designs), I encourage consideration of JHDL, as the most important properties of VHDL and Verilog are also present in JHDL. If JHDL is discovered to be lacking, Cosmic Horizon will be very interested in addressing those issues.

Another objection to adoption of Java for microprocessor simulation is the belief that Java is slow. In fact, Java is much faster than it used to be. Sun Microsystems developed Java Virtual Machines (JVMs). They are clearly committed to improving performance with each new release and many performance flags are available to control runtime characteristics of the JVM.[13]

Cosmic Horizon, also very performance minded, is developing FSS and has contributed to improving JHDL. On 2007-06-04, I measured FSS's speed at 117.24 cycles/second on a Sun Fire V210, an entry-level server that reached “end-of-life” on 2007-07-16. Several factors can affect simulation performance: Model size, length of test (a short test will be dominated by database access time[14]), number of phases per clock cycle, and speed of the host computer. Well-funded verification efforts will use computers much more powerful than the Sun Fire V210. For more on Java performance, see Interpretation Versus Compilation.

COMPARISON WITH HARDWARE VERIFICATION LANGUAGES

HVLs are domain-specific languages that are particularly suitable for creating simulation environments for functional verification of hardware models. CFV outlines typical features of modern HVLs. I am proposing the use of the Java general-purpose programming language in the hardware verification domain rather than an HVL. Therefore, I will compare typical HVL features with Java's capabilities, as demonstrated by FSS.

Simulation Independence

Simulation independence refers to test bench code that can run on a variety of simulation engines. The

13 http://java.sun.com/performance/reference/whitepapers/6_performance.html

14 We will look into overlapping database access with the simulation run by expanding our use of multithreading.

FSS SRS does not require simulation independence. The only known JHDL simulation engine is included with the JHDL software. Despite not having achieved simulation independence, JHDL specifics are confined to FSS's model subsystem. It may be desirable and possible to shrink and lower this simulation engine abstraction layer. Much of the current FSS test bench is indistinct from this layer, and part of that is HDL code.

Full Visibility

JHDL enables the reading or writing of any facility in the DUV. FSS, however, has more specific goals than JHDL. The purpose of FSS is SPARC-V9 architectural verification of a microprocessor DUV, and the product should be applicable to *any* SPARC-V9 implementation. Therefore, Cosmic Horizon is using mostly a black box approach and avoiding examination of the implementation details of the microprocessor DUV:

> "... structural changes inside the DUV have little impact on the verification code, as the function is independent of implementation. If an internal pipeline had to change because of a timing constraint, there is little to no impact on the verification environment. Furthermore, the ability to predict functional results based on inputs alone ensures that the reference model remains independent from the DUV algorithms."[15]

FSS (a self-checking test bench) will use the SPARC-V9 Standard Reference Model internally as the primary means of verifying the DUV.

One notable exception to the black box approach is that FSS allows any signal to be viewed as a waveform for debugging purposes. Another is the architectural register view, which is documented in the SRS but not yet implemented.

I will provide more detail on the waveform viewer. Users prepare a file, listing in canonical form the signals to be traced. If users want to collect signal activity, they indicate that in the graphical user interface (GUI). This causes a JFileChooser dialog to appear, so users can select an existing signal list file. FSS's SparcSimulation class, which extends byucc.jhdl.base.HWSystem, uses the canonical name to look up the corresponding byucc.jhdl.base.Nameable, a "class for providing name capability for Wires and Nodes in the JHDL circuit graph." If the Nameable is found, the SparcSimulation object ensures that the Nameable is an `instanceof` the byucc.jhdl.base.Wire class. If so, the name String is good to use as the key into a collection of wire histories. SparcSimulation implements byucc.jhdl.base.SimulatorCallback, so its simulatorUpdate method is called after "each step in the clock schedule of the simulator." The simulatorUpdate method uses the key to find the Wire.[16] SparcSimulation also implements byucc.jhdl.base.Browser, which allows its simulatorUpdate method to pass `this` to Wire's getBV method and thereby get "the value currently driven onto the wire."

High-Level Programming Language Features

Java is a high-level programming language, so it compares favorably with any HVL regarding high-level programming language features. For example, programmers can use Java's reference types to specify arbitrarily complex data types. An object-oriented language like Java is a natural fit for modeling real objects in the microprocessor as well as test bench constructs. These test bench constructs include abstractions of physical objects with which a microprocessor would interface, along with intangible objects such as a loader, process images, a results database, and waveforms to name a few. Java provides methods, classes, and packages to help organize work and handle complexity.

---

15 CFV

16 I smell a speedup opportunity here. Do you see it?

Temporal Expressions

Java has the `assert` statement. Assertion-based verification is enabled by the use of such statements in Cosmic Horizon's Sputnik microprocessor. Users of FSS are encouraged to employ the `assert` statement in their JHDL microprocessor designs as well.

The SRS requires FSS's simulator to be two-value. That is, "only two logic states (0 and 1) are computed." For verification of reset line initialization, Sputnik's state-holding circuit elements such as the state memories in finite state machines (FSMs) use JHDL's `reset` callback[17] to randomize the state. Elsewhere in the FSM, Java `switch` statements based on the state have `default` cases containing assertions that should never be reached. Actually, they should *almost* never be reached. Before reset wire activation, it is possible. A temporal expression is therefore needed. Rather than

```
assert ( false ) : "impossible state reached in foo";
```

I use the following form:

```
assert ( ! TestBenchView.hasBeenAssertedResetWire() ) :
"impossible state reached in foo";
```

At compile time, the test bench depends on a DUV. Therefore, the DUV cannot reference the test bench without creating a circular dependency. Cosmic Horizon's TestBenchView class solves that problem.[18] And its public static method minimizes impact on the source code in the DUV hierarchy because no object reference needs to be passed down.

Future work may include a JHDL implementation of the Open Verification Library (OVL), a library of predefined assertions. Such an implementation appears to be straightforward.

Constrained Random Generation

In automated verification mode, FSS pulls pregenerated random test cases from a database. I have developed (up to a certain level of capability) for our in-house use a separate random test program generator (RTPG) that populates this database. RTPG, a Java application, generates SPARC-V9 assembly language then calls the exec method of the java.lang.Runtime class to assemble and link the test program. Solaris SPARC platforms, which Cosmic Horizon has, include a SPARC assembler and link-editor, but this dependency prevents integrating RTPG with the FSS application suite. Doing so would break the portability of FSS.

A template tells RTPG the desired type of test case. RTPG is evolving into an expert system. As RTPG's level of sophistication increases, it will be tempting to resolve its portability issue and productize it.

RTPG includes a constraint solver. A scenario arose recently involving the SPARC-V9 architecture's MULX instruction, which multiplies two 64-bit integer operands to produce a *64-bit result*. The challenge was that 64-bit integer multiplication tends to produce 128-bit results. The vast majority of unconstrained random operand pairs will overflow a SPARC-V9 multiplication unit. For the new multiplication implementation in Sputnik, it was reasonable to initially constrain the operands so that each was positive[19] and their product would not overflow. One solution for generating such a

17 Not to be confused with the reset wire, this Java method is called by the simulation engine once before simulation.

18 TestBenchView was created to support assertions in the model while avoiding circular dependencies. FullChipTestBench knows about TestBenchView and writes information there. Units in the DUV (e.g. Sputnik's ControlNextStateLogic) know about TestBenchView and read information from there.

19 The reason for the positive operands constraint is that new multiplication hardware is being verified. Therefore, it is reasonable to prove correct multiplication of positive operands before doing anything else.

constrained pair would have been to generate one positive operand that was otherwise unconstrained and use that operand to further constrain the second operand. This would produce an imbalance because the operands would not be independent. Independence could be achieved by randomly generating individual bits of operands and discarding operand pairs that did not satisfy the constraint. It was easy to prove that if both operands had bits 63 to 32 zeroed, then there could be no overflow.

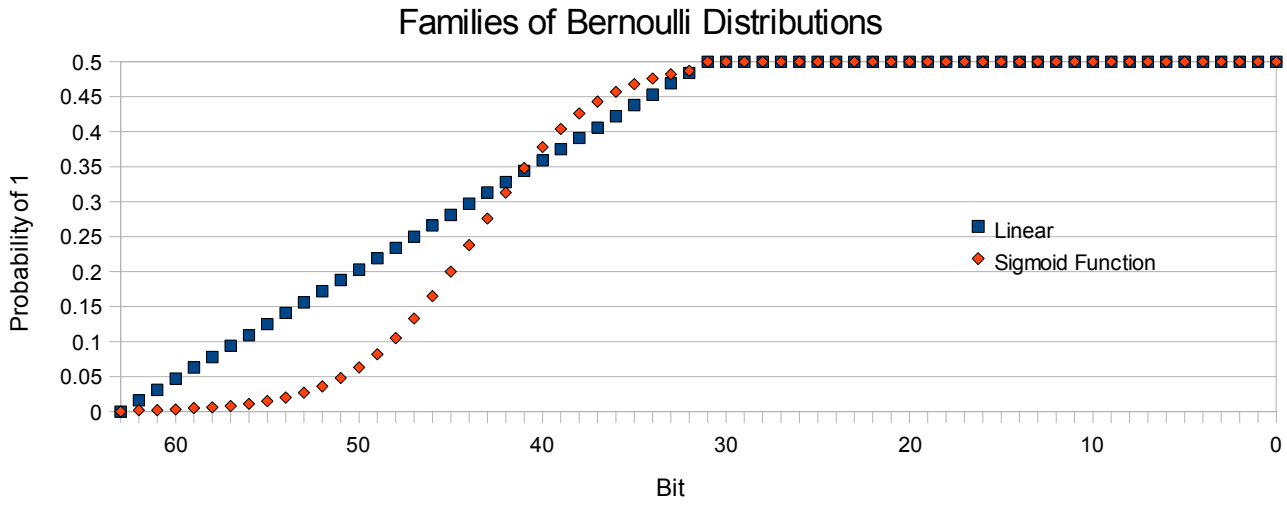

Figure 2: Families of Bernoulli Distributions

Therefore, the probability that a particular bit will be 1 could be 0.5 for bits 31 to 0. The probability that bit 63 will be 1 had to be 0 to guarantee a positive operand. The simplest curve was linear, as shown in .

However, measurements taken on a Sun Fire V210 server showed that this linear curve, although orders of magnitude faster than if all bits had a probability of 0.5, was still far too slow at solving the constraints.

I envisioned a sigmoid function to reduce the probability of large operands and speed up the constraint solver, also shown in .

The sigmoid function data points came from a parameterized closed-form expression, with a particular parameter value chosen. I am using that parameter to adjust the shape of the sigmoid function so that speed is reasonable without unnecessarily suppressing large operands.

The above constraint solvers that produce independent operands use a trial-and-error algorithm. There are 128 random bit generations before the operand pair is checked against the product constraint. To improve the efficiency of this process, Dr. Lei Chen, a CPU performance engineer at IBM, suggested a fail-fast scheme whereby the constraint solver generates pairs of bits, one from each operand. Proceeding from left to right, it is possible to know much sooner if this pair of operands violates the constraint. For example, if bits 62 from both operands are 1 (although not very probable), this operand pair can be immediately discarded.

Coverage Collection

Coverage collection is not required by the FSS SRS for version 1.0. Nevertheless, there is interest in the feature among the users of the software.

A results database is a prerequisite for coverage collection. The database is already in place for storing the results of each test. However, I need to discern how best to express which coverage information to collect over time. There are a number of different coverage models to be considered. Once these data have been collected, I will want to create a feedback path back to RTPG (or whatever RTPG has become by that time).

Automatic Garbage Collection

Java performs automatic garbage collection. The garbage collector runs in its own thread.

There is a common misconception that memory leaks are impossible with garbage collection. Java's garbage collector reclaims memory that is no longer needed, but if an object is still referenced, the garbage collector considers the object to be needed. I found such a memory leak in JHDL when FSS first became capable of executing an overnight automated verification workload. JHDL was keeping references to objects that it was not using and should have released. BYU JHDL still has the issue, but I fixed this in Cosmic Horizon JHDL. See my Automated Verification blog item for details.

Interpretation Versus Compilation

Java is an interpreted language in the sense that the JVM can directly interpret the bytecodes produced by the Java compiler. These bytecodes are not the machine language of the host platform.

> "Today's JVMs typically execute bytecodes using a combination of interpretation and so-called just-in-time (JIT) compilation. In this process, the JVM analyzes the bytecodes as they are interpreted, searching for hot spots—parts of the bytecodes that execute frequently. For these parts, a … JIT compiler—known as the Java HotSpot compiler—translates the bytecodes into the underlying computer's machine language. When the JVM encounters these compiled parts again, the faster machine-language code executes."[20]

The authors of CFV thought that the ability of some HVLs to avoid compilation altogether during test bench development is very appealing. During any software development, developers will find it useful to quickly see the effect of code that they just wrote without having to wait for the compiler. Java cannot do that.

EATING ONE'S OWN DOG FOOD

Sometimes, while debugging Sputnik, I could use a source-level debugger (I would use jdb or NetBeans IDE[21]) to find the root cause of failure faster. However, once it is determined that Sputnik is to blame, I resist that temptation. How would the user approach the problem? The debugging capabilities of FSS need to be improved, and with FSS exclusively being used to debug Sputnik, Cosmic Horizon is able to discover what improvements would be most valuable. FSS's waveform viewer, in particular, has benefited considerably from this exercise.

IF A TREE FALLS IN THE FOREST ...

Finally, I want to consider the modeling of memory, which is outside the microprocessor and therefore part of the test bench. The memory model, written in Java, does not depend on the JHDL library

---

20 Java: How to Program, Sixth Edition

21 jdb can handle basic source-level debugging tasks. NetBeans IDE has more powerful features such as conditional breakpoints and a profiler, but has a large memory footprint. As of this writing, neither Eclipse IDE for Java Developers nor IntelliJ IDEA is available for Solaris.

because the model is not RTL and will never be presented to a synthesis tool.[22] Instead, software speed is a primary concern.

Suppose I want to model a 64-bit address space. I cannot simply create an ordinary array of 2^{64} bytes. Dynamic memory allocation on the host platform would certainly fail. Even if I could somehow do that, it would be extremely wasteful because most of the memory locations will never be accessed.

In practice, the 64-bit address space seen by a SPARC-V9 microprocessor is a sparse array. Specifically, I chose to model it as

```
public class AddressSpace extends Hashtable<
org.apache.axis2.databinding.types.UnsignedLong, Byte > {
…
```

If there is an address that is never written and never read, it does not require storage on the simulation host computer. The situation gets interesting when an address is read before it is written such as when the microprocessor disobeys the software running on it (i.e., a bug causes the DUV to present the wrong address to memory). Another way this can happen is when the software tells the microprocessor to read an uninitialized address. For example in C, ¨if an object that has automatic storage duration is not initialized explicitly, its value is indeterminate.¨[23] A SPARC-V9 implementation is required to correctly execute even badly written software. I chose to have AddressSpace return random data to the microprocessor. That is not so interesting. But what if the microprocessor reads the same address more than once before writing? In that case, it is necessary to return the *same data* that was returned the first time, even though it was random. In other words, that first randomization must be accompanied by a behind-the-scenes write to the address being read by the microprocessor. The hash table grows with each such read.

CONCLUSION

In this article, I addressed specific challenges and issues in defining and implementing FSS. Any software project amounts to solving a series of problems. I want to emphasize that even when a microprocessor and its verification environment are implemented in Java, problems are not insurmountable. If Cosmic Horizon is to provide the entire test bench, then FSS users will not be developing that Java code. But the microprocessor design implementation becomes a Java development. I hope that the portability of the overall system along with powerful features of FSS (such as its planned ability to architecturally verify any SPARC-V9 implementation without redesign of the test bench) will be attractive enough to encourage serious consideration of JHDL for the DUV.

22 As stated in Microprocessor Design, “synthesis tools automatically convert from RTL to layout.”
23 ISO/IEC 9899:TC2