CHAPTER 2 VERIFICATION TOOLS

As mentioned in the previous chapter, one of the mechanisms that can be used to improve the efficiency and reliability of a process is automation. This chapter covers tools used in a state-of-the-art functional verification environment. Some of these tools, such as simulators, are essential for the functional verification activity to take place. Others, such as linting or code coverage tools, automate some of the most tedious tasks of verification and help increase the confidence in the outcome of the functional verification.

Not all tools are mentioned in this chapter. It is not necessary to use all the tools mentioned.

It is not necessary to use all of the tools described here. Nor is this list exhaustive, as new application-specific and general purpose verification automation tools are regularly brought to market. As a verification engineer, your job is to use the necessary tools to ensure that the outcome of the verification process is not a Type II mistake, which is a false positive. As a project manager responsible for the delivery of a working product on schedule and within the allocated budget, your responsibility is to arm your engineers with the proper tools to do their job efficiently and with the necessary degree of confidence. Your job is also to decide when the cost of finding the next functional bug has increased above the value the additional functional correctness brings. This last responsibility is the heaviest of them all. Some of these tools provide information to help you decide when you've reached that point.

No endorsements of commercial tools.

I mention some commercial tools by name. They are used for illustrative purposes only and this does not constitute a personal

endorsement. I apologize in advance to suppliers of competitive products I fail to mention. It is not an indication of inferiority, but rather an indication of my limited knowledge. All trademarks and service marks, registered or not, are the property of their respective owners.

LINTING TOOLS

Linting tools find common programmer mistakes.

The term “lint” comes from the name of a UNIX utility that parses a C program and reports questionable uses and potential problems. When the C programming language was created by Dennis Ritchie, it did not include many of the safeguards that have evolved in later versions of the language, like ANSI-C or C++, or other strongly-typed languages such as Pascal or ADA. lint evolved as a tool to identify common mistakes programmers made, letting them find the mistakes quickly and efficiently, instead of waiting to find them through a dreaded segmentation fault during execution of the program.

lint identifies real problems, such as mismatched types between arguments and function calls or mismatched number of arguments, as shown in Sample 2-1. The source code is syntactically correct and compiles without a single error or warning using gcc version 2.8.1.

Sample 2-1. Syntactically correct K&R C source code

```
int my_func(addr_ptr, ratio)
   int   *addr_ptr;
   float ratio;
{
   return (*addr_ptr)++;
}

main()
{
   int my_addr;
   my_func(my_addr);
}
```

However, Sample 2-1 suffers from several pathologically severe problems:

1. The *my_func* function is called with only one argument instead of two.

2. The *my_func* function is called with an integer value as a first argument instead of a pointer to an integer.

Problems are found faster than at runtime.

As shown in Sample 2-2, the lint program identifies these problems, letting the programmer fix them before executing the program and observing a catastrophic failure. Diagnosing the problems at runtime would require a runtime debugger and would take several minutes. Compared to the few seconds it took using lint, it is easy to see that the latter method is more efficient.

Sample 2-2. Lint output for Sample 2-1

```
src.c(3): warning: argument ratio unused in
function my_func
src.c(11): warning: addr may be used before set
src.c(12): warning: main() returns random value
to invocation environment
my_func: variable # of args.    src.c(4)  ::
src.c(11)
my_func, arg. 1 used inconsistently
src.c(4)  ::  src.c(11)
my_func returns value which is always ignored
```

Linting tools are static tools.

Linting tools have a tremendous advantage over other verification tools: They do not require stimulus, nor do they require a description of the expected output. They perform checks that are entirely static, with the expectations built into the linting tool itself.

The Limitations of Linting Tools

Linting tools can only identify a certain class of problems.

Other potential problems were also identified by lint. All were fixed in Sample 2-3, but lint continues to report a problem with the invocation of the *my_func* function: The return value is always ignored. Linting tools cannot identify all problems in source code. They can only find problems that can be statically deduced by looking at the code structure, not problems in the algorithm or data flow.

For example, in Sample 2-3, lint does not recognize that the uninitialized *my_addr* variable will be incremented in the *my_func* function, producing random results. Linting tools are similar to spell checkers; they identify misspelled words, but do not determine if the wrong word is used. For example, this book could have several instances of the word “with” being used instead of “width”. It is a type of error the spell checker (or a linting tool) could not find.

Sample 2-3. Functionally correct K&R C source code

```
int my_func(addr_ptr)
   int *addr_ptr;
{
   return (*addr_ptr)++;
}

main()
{
   int my_addr;
   my_func(&my_addr);
   return 0;
}
```

Many false negatives are reported.

Another limitation of linting tools is that they are often too paranoid in reporting problems they identify. To avoid making a Type II mistake—reporting a false positive, they err on the side of caution and report potential problems where none exist. This results in many Type I mistakes—or false negatives. Designers can become frustrated while looking for non-existent problems and may abandon using linting tools altogether.

Carefully filter error messages!

You should filter the output of linting tools to eliminate warnings or errors known to be false. Filtering error messages helps reduce the frustration of looking for non-existent problems. More importantly, it reduces the output clutter, reducing the probability that the report of a real problem goes unnoticed among dozens of false reports. Similarly, errors known to be true positive should be highlighted. *Extreme* caution must be exercised when writing such a filter: You must make sure that a true problem is not filtered out and never reported.

Naming conventions can help output filtering.

A properly defined naming convention is a useful tool to help determine if a warning is significant. For example, the report in Sample 2-4 about a latch being inferred on a signal whose name ends with "_lat" would be considered as expected and a false warning. All other instances would be flagged as true errors.

Sample 2-4. Output from a hypothetical Verilog linting tool

```
Warning: file decoder.v, line 23: Latch
inferred on reg "address_lat".
Warning: file decoder.v, line 36: Latch
inferred on reg "next_state".
```

Do not turn off checks.

Filtering the output of a linting tool is preferable to turning off checks from within the source code itself or via the command line. A check may remain turned off for an unexpected duration, potentially hiding real problems. Checks that were thought to be irrelevant may become critical as new source files are added.

Lint code as it is being written.

Because it is better to fix problems when they are created, you should run lint on the source code while it is being written. If you wait until a large amount of code is written before linting it, the large number of reports—many of them false—will be daunting and create the impression of a setback. The best time to identify a report as true or false is when you are still intimately familiar with the code.

Enforce coding guidelines.

The linting process, through the use of user-defined rules, can also be used to enforce coding guidelines and naming conventions[1]. Therefore, it should be an integral part of the authoring process to make sure your code meets the standards of readability and maintainability demanded by your audience.

Linting Verilog Source Code

Linting Verilog source code catches common errors.

Linting Verilog source code ensures that all data is properly handled without accidentally dropping or adding to it. The code in Sample 2-5 shows a Verilog model that looks perfect, compiles without errors, but produces unintended results under some circumstances in Verilog-95.

Sample 2-5. Potentially problematic Verilog code

```
module tristate_buffer(in, out, enable);
parameter WIDTH = 8;
input  [WIDTH-1:0] in;
output [WIDTH-1:0] out;
input              enable;

assign out = (enable) ? in : 'bz;

endmodule
```

Problems may not be apparent under most conditions.

The problem is in the width mismatch in the continuous assignment between the output "*out*" and the constant "'*bz*". The unsized constant is 32-bits wide (or a value of 32'hzzzzzzzz), while the output

1. See Appendix A for a set of coding guidelines.

has a user-specified width. As long as the width of the output is less than or equal to 32, everything is fine: The value of the constant will be appropriately truncated to fit the width of the output.

However, in Verilog-95, the problem occurs when the width of the output is greater than 32 bits: Verilog-95 *zero-extends* the constant value to match the width of the output, producing the wrong result. The least significant 32-bits are set to high-impedance, while all the other more significant bits are set to zero. This "feature" has been fixed in Verilog-2001.

It is an error that could not be found in simulation, unless a configuration greater than 32 bits was used, *and* it produced wrong results at a time and place you were looking at. A linting tool finds the problem every time, in just a few seconds.

Linting VHDL Source Code

Because of its strong typing, VHDL does not need linting as much as Verilog. Much of the checks performed by a linting tool are required to be performed by the VHDL compiler. However, potential problems are still best identified using a linting tool.

Linting can find unintended multiple drivers.

For example, a common problem in VHDL is created by using the STD_LOGIC type. Since it is a resolved type, STD_LOGIC signals can have more than one driver. When modeling hardware, multiple driver signals are required in a single case: to model buses. In all other cases (which is over 99% of the time), a signal should have only one driver. The VHDL source shown in Sample 2-6 demonstrates how a simple typographical error can go undetected easily and satisfy the usually paranoid VHDL compiler.

Typographical errors can cause serious problems.

In Sample 2-6, both concurrent signal assignments labeled "*statement1*" and "*statement2*" assign to the signal "*s1*" (ess-one), while the signal "*sl*" (ess-ell) remains unassigned. Had I used the STD_ULOGIC type instead of the STD_LOGIC type, the VHDL toolset would have reported an error after finding multiple drivers on an unresolved signal. However, it is not possible to guarantee the STD_ULOGIC type is used for all signals with a single driver. A

Sample 2-6. Erroneous multiple drivers

```
library ieee;
use ieee.std_logic_1164.all;
entity my_entity is
   port (my_input: in std_logic);
end my_entity;

architecture sample of my_entity is
   signal s1: std_logic;
   signal sl: std_logic;
begin
   statement1: s1 <= my_input;
   statement2: s1 <= not my_input;
end sample;
```

linting tool is still required to report multiple driver signals regardless of the type, as shown in Sample 2-7.

Sample 2-7. Output from a hypothetical VHDL linting tool

```
Warning: file my_entity.vhd: Signal "s1" is
multiply driven.
Warning: file my_entity.vhd: Signal "sl" has no
drivers.
```

Use naming convention to filter output.

It would be up to the author to identify the signals that were intended to model buses and ignore the warnings about them. Using a naming convention for such signals facilitates recognizing warnings that can be safely ignored, and enhances the reliability of your code. An example of a naming convention, illustrated in Sample 2-8, would be to name any signals modeling buses with the "_bus" suffix[2].

Sample 2-8. Naming convention for signals with multiple drivers

```
--
-- data_bus, addr_bus and sys_err_bus
-- are intended to be multiply driven
--
signal data_bus : std_logic_vector(15 downto 0);
signal addr_bus : std_logic_vector( 7 downto 0);
signal ltch_addr: std_logic_vector( 7 downto 0);
signal sys_err_bus: std_logic;
signal bus_grant  : std_logic;
```

2. See Appendix A for an example of naming guidelines.

Linting can identify inferred latches.

The accidental multiple driver problem is not the only one that can be caught using a linting tool. Others, such as unintended latch inference in synthesizeable code, or the enforcement of coding guidelines, can also be identified.

Linting OpenVera and *e* Source Code

Because of their strong typing, *e* and OpenVera do not need linting as much as Verilog. But like Verilog, potential problems are still best identified using a linting tool. For example, Sample 2-9 shows a race condition between two concurrent execution branches that will yield an unpredictable result (this race condition is explained in details in the section titled “Write/Write Race Conditions” on page 212). This type of error would be easily detectable by a linting tool. Linting tools for OpenVera and *e* are starting to emerge.

Sample 2-9. Race condition on OpenVera code

```
{
   integer i;
   fork
      i = 1;
      i = 0;
   join
}
```

Code Reviews

Reviews are performed by peers.

Although not technically linting tools, the objective of code reviews is essentially the same: Identify functional and coding style errors before functional verification and simulation. Linting tools can only identify questionable language uses. They cannot check if the intended behavior has been coded. In code reviews, the source code produced by a designer is reviewed by one or more peers. The goal is not to publicly ridicule the author, but to identify problems with the original code that could not be found by an automated tool. Reviews can identify discrepancies between the design intent and the implementation. They also provide an opportunity for suggesting coding improvements, such as better comments, better structure or better organization.

Identify qualitative problems and functional errors.

A code review is an excellent venue for evaluating the maintainability of a source file, and the relevance of its comments. Other qualitative coding style issues can also be identified. If the code is

well understood, it is often possible to identify functional errors or omissions.

Code reviews are not new ideas either. They have been used for many years in the software design industry. Detailed information on how to conduct effective code reviews can be found in the *resources* section at:

```
http://janick.bergeron.com/wtb
```

SIMULATORS

Simulate your design before implementing it.

Simulators are the most common and familiar verification tools. They are named simulators because their role is limited to approximating reality. A simulation is never the final goal of a project. The goal of all hardware design projects is to create real physical designs that can be sold and generate profits. Simulators attempt to create an artificial universe that mimics the future real design. This type of verification lets the designers interact with the design before it is manufactured, and correct flaws and problems earlier.

Simulators are only approximations of reality.

You must never forget that a simulator is an approximation of reality. Many physical characteristics are simplified—or even ignored—to ease the simulation task. For example, a four-state digital simulator assumes that the only possible values for a signal are 0, 1, unknown, and high-impedance. However, in the physical—and analog—world, the value of a signal is a continuous function of the voltage and current across a thin aluminium or copper wire track: an infinite number of possible values. In a discrete simulator, events that happen deterministically 5 ns apart may be asynchronous in the real world and may occur randomly.

Simulators are at the mercy of the descriptions being simulated.

Within that simplified universe, the only thing a simulator does is execute a description of the design. The description is limited to a well-defined language with precise semantics. If that description does not accurately reflect the reality it is trying to model, there is no way for you to know that you are simulating something that is different from the design that will be ultimately manufactured. Functional correctness and accuracy of models is a big problem as errors cannot be proven *not* to exist.

Stimulus and Response

Simulation requires stimulus.

Simulators are not static tools. A static verification tool performs its task on a design without any additional information or action required by the user. For example, linting tools are static tools. Simulators, on the other hand, require that you provide a facsimile of the environment in which the design will find itself. This facsimile is called a testbench. Writing this testbench is the main objective of this textbook. The testbench needs to provide a representation of the inputs observed by the design, so the simulator can emulate the design's responses based on its description.

The simulation outputs are validated externally, against design intents.

The other thing that you must not forget is that simulators have no knowledge of your intentions. They cannot determine if a design being simulated is correct. Correctness is a value judgment on the outcome of a simulation that must be made by you, the verification engineer. Once the design is subjected to an approximation of the inputs from its environment, your primary responsibility is to examine the outputs produced by the simulation of the design's description and determine if that response is appropriate.

Event-Driven Simulation

Simulators are never fast enough.

Simulators are continuously faced with one intractable problem: They are never fast enough. They are attempting to emulate a physical world where electricity travels at the speed of light and millions of transistors switch over one billion times in a second. Simulators are implemented using general purpose computers that can execute, under ideal conditions, up to one billion sequential instructions per second. The speed advantage is unfairly and forever tipped in favor of the physical world.

Outputs change only when an input changes.

One way to optimize the performance of a simulator is to avoid simulating something that does not need to be simulated. Figure 2-1 shows a 2-input XOR gate. In the physical world, if the inputs do not change (Figure 2-1(a)), even though voltage is constantly applied to the output, current is continuously flowing through the transistors (in some technologies), and the atomic particles in the semiconductor are constantly moving, the *interpretation* of the output electrical state as a binary value (either a logic 1 or a logic 0)

does not change. Only if one of the inputs change (as in Figure 2-1(b)), does the output change.

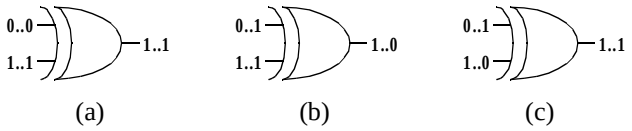


Figure 2-1. Behavior of an XOR gate

Change in values, called *events*, drive the simulation process.

Sample 2-10 shows a VHDL description (or model) of an XOR gate. The simulator could choose to execute this model continuously, producing the same output value if the input values did not change. An opportunity to improve upon that simulator's performance becomes obvious: do not execute the model while the inputs are constants. Phrased another way: Only execute a model when an input changes. The simulation is therefore driven by changes in inputs. If you define an input change as an *event*, you now have an *event-driven* simulator.

Sample 2-10. VHDL model for an XOR gate

```
XOR_GATE: process (A, B)
begin
   if A = B then
      Z <= '0';
   else
      Z <= '1'
   end if;
end process XOR_GATE;
```

Sometimes, input changes do not cause the output to change.

But what if both inputs change, as in Figure 2-1(c)? In the logical world, the output does not change. What should an event-driven simulator do? For two reasons, the simulator should execute the description of the XOR gate. First, in the real world, the output of the XOR gate *does* change. The output might oscillate between 0 and 1 or remain in the "neither-0-nor-1" region for a few hundredths of picoseconds (see Figure 2-2). It just depends on how accurate you want your model to be. You could decide to model the XOR gate to include the small amount of time spent in the

unknown (or x) state to more accurately reflect what happens when both inputs change at the same time.

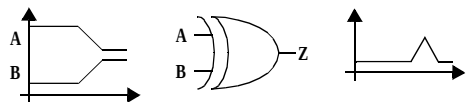


Figure 2-2. Behavior of an XOR gate when both inputs change

Descriptions between inputs and outputs are arbitrary.

The second reason is that the event-driven simulator does not know apriori that it is about to execute a model of an XOR gate. All the simulator knows is that it is about to execute a description of a 2-input, 1-output function. Figure 2-3 shows the view of the XOR gate from the simulator's perspective: a simple 2-input, 1-output black box. The black box could just as easily contain a 2-input AND gate (in which case the output might very well change if both inputs change), or a 1024-bit linear feedback shift register (LFSR).

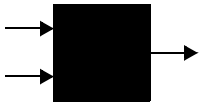


Figure 2-3. Event-driven simulator view of an XOR gate

The mechanism of event-driven simulation introduces some limitations and interesting side effects that are discussed further in Chapter 4.

Acceleration options are often available in event-driven simulators

Simulation vendors are forever locked in a constant battle of beating the competition with an easier-to-use, faster simulator. It is possible to increase the performance of an event-driven simulator by simplifying some underlying assumptions in the design or in the simulation algorithm. For example, reducing delay values to identical unit delays or using two states (0 and 1) instead of four states (0, 1, x and z) are techniques used to speed-up simulation. You should refer to the documentation of your simulator to see what acceleration options are provided. It is also important to understand what are the consequences, in terms of reduced accuracy, of using these acceleration options.

Cycle-Based Simulation

Figure 2-4 shows the event-driven view of a synchronous circuit composed of a chain of three 2-input gates between two edge-triggered flip-flops. Assuming that Q1 holds a 0, Q2 holds a 1 and all other inputs remain constant, a rising edge on the clock input would cause an event-driven simulator to simulate the circuit as follows:

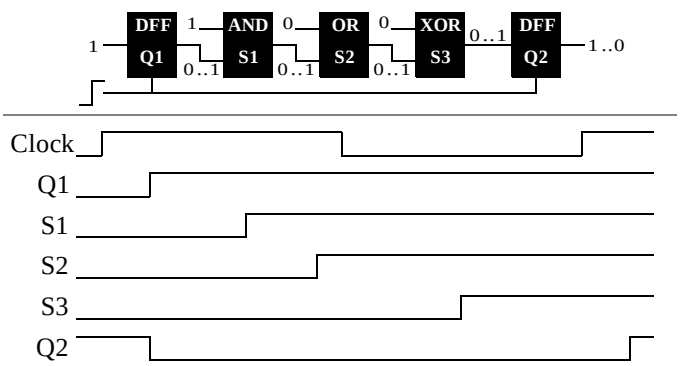


Figure 2-4. Event-driven simulator view of a synchronous circuit

1. The event (rising edge) on the clock input causes the execution of the description of the flip-flop models, changing the output value of Q1 to 1 and of Q2 to 0, after a delay of 1 ns.
2. The event on Q1 causes the description of the AND gate to execute, changing the output S1 to 1, after a delay of 2 ns.
3. The event on S1 causes the description of the OR gate to execute, changing the output S2 to 1, after a delay of 1.5 ns.
4. The event on S2 causes the description of the XOR gate to execute, changing the output S3 to 1 after a delay of 3 ns.
5. The next rising edge on the clock causes the description of the flip-flops to execute, Q1 remains unchanged, and Q2 changes back to 1, after a delay of 1 ns.

Many intermediate events in synchronous circuits are not functionally relevant.

To simulate the effect of a single clock cycle on this simple circuit required the generation of six events and the execution of seven models (some models were executed twice). If all we are interested in are the final states of Q1 and Q2, not of the intermediate combinatorial signals, then the simulation of this circuit could be optimized by acting only on the significant events for Q1 and Q2: the

active edge of the clock. Phrased another way: Simulation is based on clock cycles. This is how cycle-based simulators operate.

The synchronous circuit in Figure 2-4 can be simulated in a cycle-based simulator using the following sequence:

Cycle-based simulators collapse combinatorial logic into equations.

1. When the circuit description is compiled, all combinatorial functions are collapsed into a single expression that can be used to determine all flip-flop input values based on the current state of the fan-in flip-flops.

   For example, the combinatorial function between Q1 and Q2 would be compiled from the following initial description:

   ```
   S1 = Q1 & '1'
   S2 = S1 | '0'
   S3 = S2 ^ '0'
   ```

   into this final single expression:

   ```
   S3 = Q1
   ```

   The cycle-based simulation view of the compiled circuit is shown in Figure 2-5.

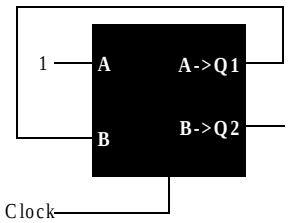


Figure 2-5. Cycle-based simulator view of a synchronous circuit

2. During simulation, whenever the clock input rises, the value of all flip-flops are updated using the input value returned by the pre-compiled combinatorial input functions.

The simulation of the same circuit, using a cycle-based simulator, required the generation of two events and the execution of a single model. The number of logic computations performed is the same in both cases. They would have been performed whether the "A" input changed or not. As long as the time required to perform logic computation is smaller than the time required to schedule intermediate events,[3] and there are many registers changing state at every clock cycle, cycle-based simulation will offer greater performance.

Cycle-based simulations have no timing information.

This great improvement in simulation performance comes at a cost: All timing and delay information is lost. Cycle-based simulators assume that the entire design meets the setup and hold requirements of all the flip-flops. When using a cycle-based simulator, timing is usually verified using a static timing analyzer.

Cycle-based simulators can only handle synchronous circuits.

Cycle-based simulators further assume that the active clock edge is the only significant event in changing the state of the design. All other inputs are assumed to be perfectly synchronous with the active clock edge. Therefore, cycle-based simulators can only simulate perfectly synchronous designs. Anything containing asynchronous inputs, latches or multiple-clock domains *cannot* be simulated accurately. Fortunately, the same restrictions apply to static timing analysis. Thus, circuits that are suitable for cycle-based simulation to verify the functionality are suitable for static timing verification to verify the timing.

Co-Simulators

No real-world design and testbench is perfectly suited for a single simulator, simulation algorithm or modeling language. Different components in a design may be specified using different languages. A design could contain small sections that cannot be simulated using a cycle-based algorithm. Testbenches may (and should) be written using an HVL while the design is written in VHDL or Verilog.

Multiple simulators can handle separate portions of a simulation.

To handle the portions of a design that do not meet the requirements for cycle-based simulation, most cycle-based simulators are integrated with an event-driven simulator. As shown in Figure 2-6, the synchronous portion of the design is simulated using the cycle-based algorithm, while the remainder of the design is simulated using a conventional event-driven simulator. Both simulators

3. And they are. By a long shot.

(event-driven and cycle-based) are running together, cooperating to simulate the entire design.

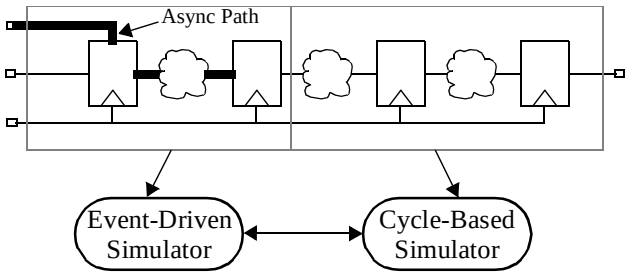


Figure 2-6. Event-driven and cycle-based co-simulation

Other popular co-simulation environments provide VHDL and Verilog, HDL and HVL or digital and analog co-simulation. For example, Figure 2-7 shows the testbench (written in *e*) and a design co-simulated using Specman Elite and a HDL simulator.

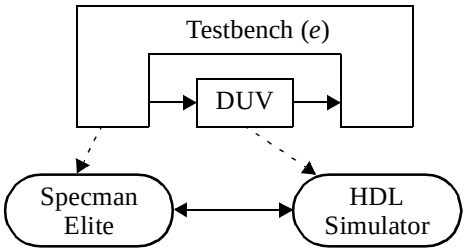


Figure 2-7. HVL and HDL co-simulation

All simulators operate in locked-step.

During co-simulation, all simulators involved progress along the time axis in lock-step. All are at simulation time T_1 at the same time and reach the next time T_2 at the same time. This implies that the speed of a co-simulation environment is limited by the slowest simulator. Some experimental co-simulation environments implement *time warp* synchronization where some simulators are allowed to move ahead of the others.

Performance is decreased by the communication and synchronization overhead.

The biggest hurdle of co-simulation comes from the communication overhead between the simulators. Whenever a signal generated within a simulator is required as an input by another, the current value of that signal, as well as the timing information of any change in that value, must be communicated. This communication usually

involves a translation of the event from one simulator into an (almost) equivalent event in another simulator. Ambiguities can arise during that translation when each simulation has different semantics. The difference in semantics is usually present: the semantic difference often being the requirement for co-simulation in the first place.

Translating values and events from one simulator to another can create ambiguities.

Examples of translation ambiguities abound. How do you map Verilog's 128 possible states (composed of orthogonal logic values and strengths) into VHDL's nine logic values (where logic values and strengths are combined)? How do you translate a voltage and current value in an analog simulator into a logic value and strength in a digital simulator? How do you translate an x or z value into a 2-state *e* value? How do you translate the timing of zero-delay events from Verilog (which has no strict concept of delta cycles)[4] to VHDL?

Co-simulation should not be confused with single-kernel simulation.

Co-simulation is when two (or more) simulators are cooperating to simulate a design, each simulating a portion of the design, as shown in Figure 2-8. It should not be confused with simulators able to read and compile models described in different languages. For example, Cadence's *NCSIM* simulator and Model Technology's *ModelSim* simulator can both simulate a design described using a mix of VHDL and Verilog. Synopsys's VCS simulator can simulate Verilog and a subset of OpenVera. As shown in Figure 2-9, all languages are compiled into a single internal representation or machine code and the simulation is performed using a single simulation engine.

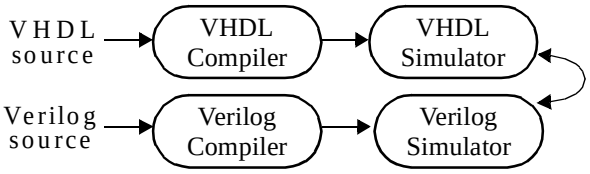


Figure 2-8. Co-simulator

4. See "The Simulation Cycle" on page 194 for more details on delta cycles.

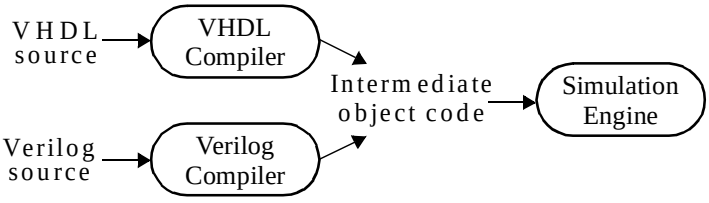


Figure 2-9. Mixed-language simulator

VERIFICATION INTELLECTUAL PROPERTY

You can buy IP for standard functions.

If you want to verify your design, it is necessary to have models for all the parts included in a simulation. The model of the RTL design is a natural by-product of the design exercise and the actual objective of the simulation. Models for embedded or external RAMs are also required, as well as models for standard interfaces and off-the-shelf parts. If you were able to procure the RAM, design IP, specification or standard part from a third party, you should be able to obtain a model for it as well. You may have to obtain the model from a different vendor than the one who supplies the physical part.

It is cheaper to buy models than write them yourself.

At first glance, buying a simulation model from a third-party provider may seem expensive. Many have decided to write their own models to save on licensing costs. However, you have to decide if this endeavor is truly economically fruitful: Are you in the modeling business or in the chip design business? If you have a shortage of qualified engineers, why spend critical resources on writing a model that does not embody any competitive advantage for your company? If it was not worth designing on your own in the first place, why is writing your own model suddenly justified?

Your model is not as reliable as the one you buy.

Secondly, the model you write has never been used before. Its quality is much lower than a model that has been used by several other companies before you. The value of a functionally correct and reliable model is far greater than an uncertain one. Writing and verifying a model to the *same degree of confidence* as the third-party model is always more expensive than licensing it. And be assured: No matter how simple the model is (such as a quad 2-input NAND gate, 74LS00), you'll get it wrong the first time. If not functionally, then at least with respect to timing or connectivity.

There are several providers of verification IP. Many are written using an HVL or C code; others are provided as non-synthesizeable

VHDL or Verilog source code. For intellectual property protection and licensing technicalities, most are provided as compiled binary models. Verification IP includes, but is not limited to functional models of external and embedded memories, bus-functional models for standard interfaces, protocol generators and analyzers, assertion sets for standard protocols and black-box models for off-the-shelf components and processors.

Hardware Modelers

What if you cannot find a model to buy?

You may be faced with procuring a model for a device that is so new or so complex, that no provider has had time to develop a reliable model for it. For example, at the time the first edition of this book was written, you could license full-functional models for the Pentium processor from at least two vendors. However, you could not find a model for the Pentium III. If you want to verify that your new PC board, which uses the latest Intel microprocessor, is functionally correct before you build it, you have to find some other way to include a simulation model of the processor.

You can "plug" a chip into a simulator.

Hardware modelers provide a solution for that situation. A hardware modeler is a small box that connects to your network. A *real, physical* chip that needs to be simulated is plugged into it. During simulation, the hardware modeler communicates with your simulator (through a special interface package) to supply inputs from the simulator to the device, then sends the sampled output values from the device back to the simulation. Figure 2-10 illustrates this communication process.

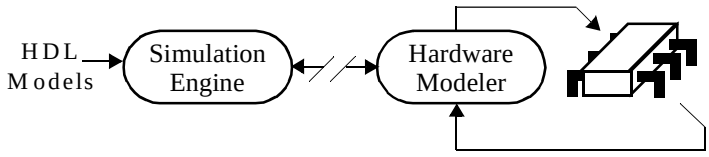


Figure 2-10. Interfacing a hardware modeler and a simulator

Timing of I/O signals still needs to be modeled.

Using a hardware modeler is not a trivial task. Often, an adaptor board must be built to fit the device onto the socket on the modeler itself. Also, the modeler cannot perform timing checks on the device's inputs nor accurately reflect the output delays. A timing shell performing those checks and delays must be written to more accurately model a device using a hardware modeler.

Hardware modelers offer better simulation performance.

Hardware modelers are also very useful when simulating a model of the part at the required level of abstraction. A full-functional model of a modern processor that can fetch, decode and execute instructions could not realistically execute more than 10 to 50 instructions within an acceptable time period. The real physical device can perform the same task in a few milliseconds. Using a hardware modeler can greatly speed up board- and system-level simulation.

WAVEFORM VIEWERS

Waveform viewers display the changes in signal values over time.

Waveform viewers are the most common verification tools used in conjunction with simulators. They let you visualize the transitions of multiple signals over time, and their relationship with other transitions. With such a tool, you can zoom in and out over particular time sequences, measure time differences between two transitions, or display a collection of bits as bit strings, hexadecimal or as symbolic values. Figure 2-11 shows a typical display of a waveform viewer showing the inputs and outputs of a 4-bit synchronous counter.

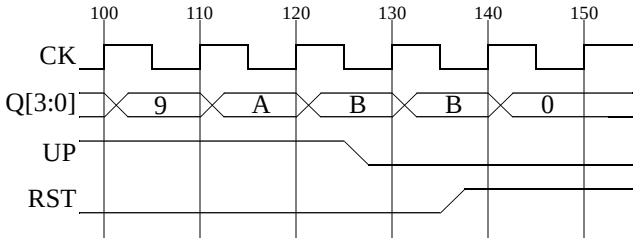


Figure 2-11. Hypothetical waveform view of a 4-bit synchronous counter

Waveform viewers are used to debug simulations.

Waveform viewers are indispensable during the authoring phase of a design or a testbench. With a viewer you can casually inspect that the behavior of the code is as expected. They are needed to diagnose, in an efficient fashion, why and when problems occur in the design or testbench. They can be used interactively during the simulation, but more importantly offline, after the simulation has completed. As shown in Figure 2-12, a waveform viewer can play back

the events that occurred during the simulation that were recorded in some trace file.

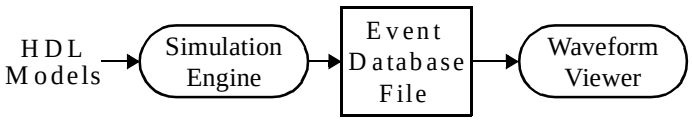


Figure 2-12. Waveform viewing as post-processing

Recording waveform trace data decreases simulation performance.

Viewing waveforms as a post-processing step lets you quickly browse through a simulation that can take hours to run. However, keep in mind that recording trace information significantly reduces the performance of the simulator. The quantity and scope of the signals whose transitions are traced, as well as the duration of the trace, should be limited as much as possible. Of course, you have to trade-off the cost of tracing a greater quantity or scope of signals versus the cost of running the simulation over again to get a trace of additional signals that turn out to be required to completely diagnose the problem. If it is likely or known that bugs will be reported, such as the beginning of the project or during a debugging iteration, trace all the signals required to diagnose the problem. If no errors are expected, such as during regression runs, no signal should be traced.

Do not use a waveform viewer to determine if a design passes or fails.

In a functional verification environment, using a waveform viewer to determine the correctness of a design involves interpreting the dozens (if not hundreds) of wavy lines on a computer screen against some expectation. It can be an acceptable verification method used two or three times, for less than a dozen signals. As the number of signals and transitions increases, so does the number of relationships that must be checked for correctness. Multiply that by the duration of the simulation. Multiply again by the number of simulation runs. Very soon, the probability that a functional error is missed reaches one.

Some viewers can compare sets of waveforms.

Some waveform viewers can compare two sets of waveforms. One set is presumed to be a golden reference, while the other is verified for any discrepancy. The comparator visually flags or highlights any differences found. This approach has two significant problems.

How do you define a set of waveforms as “golden”?

First, how is the golden reference waveform set declared “golden”? If visual inspection is required, the probability of missing a signifi-

cant functional error remains equal to one in most cases. The only time golden waveforms are truly available is in a redesign exercise, where cycle-accurate backward compatibility must be maintained. However, there are very few of these designs. Most redesign exercises take advantage of the process to introduce needed modifications or enhancements, thus tarnishing the status of the golden waveforms.

And are the differences really significant?

Second, waveforms are at the wrong level of abstraction to compare simulation results against design intent. Differences from the golden waveforms may not be significant. The value of all output signals is not significant all the time. Sometimes, what is significant is the relative relationships between the transitions, not their absolute position. The new waveforms may be simply shifted by a few clock cycles compared to the reference waveforms, but remain functionally correct. Yet, the comparator identifies this situation as a mismatch.

CODE COVERAGE

Did you forget to verify some function in your code?

Code coverage is a tool that can identify what code has been (and more importantly *not* been) executed in the design under verification. It is a methodology that has been in use in software engineering for quite some time. The problem with false positive answers (i.e., a bad design is thought to be good), is that they look identical to a true positive answer. It is impossible to know, with 100 percent certainty, that the design being verified is indeed functionally correct. All of your testbenches simulate successfully, but are there sections of the RTL code that you did not exercise and therefore not triggered a functional error? That is the question that code coverage can help answer.

Code must first be instrumented.

Figure 2-13 shows how a code coverage tool works. The source code is first *instrumented*. The instrumentation process simply adds checkpoints at strategic locations of the source code to record whether a particular construct has been exercised. The instrumentation method varies from tool to tool. Some may use file I/O features available in the language (i.e., use *$write* statements in Verilog or

textio.write procedure calls in VHDL). Others may use special features built into the simulator.

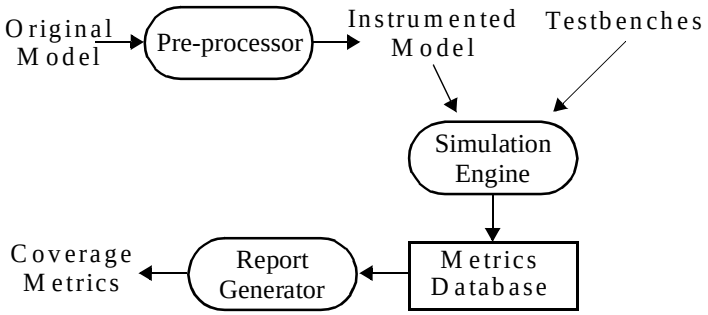


Figure 2-13. Code coverage process

No need to instrument the testbenches.

Only the code for the design under verification is instrumented. The objective is to determine if you have forgotten to exercise some code in the design. The code for the testbenches need not be traced to confirm that it has executed. If a significant section of a testbench was not executed, it should be reflected in some portion of the design not being exercised. Furthermore, a significant portion of the testbench code is executed only if an error is detected. Code coverage metrics on testbench code are therefore of little interest.

Trace information is collected at runtime.

The instrumented code is then simulated normally using all available, uninstrumented, testbenches. The cumulative traces from all simulations are collected into a database. From that database, reports can be generated to measure various coverage metrics of the verification suite on the design.

Statement and block coverage are the same thing.

The most popular metrics are statement, path and expression coverage. Statement coverage can also be called block coverage, where a block is a sequence of statements that are executed if a single statement is executed. The code in Sample 2-11 shows an example of a statement block. The block named *acked* is executed entirely whenever the expression in the *if* statement evaluates to TRUE. So

counting the execution of that block is equivalent to counting the execution of the four individual statements within that block.

Sample 2-11. Block vs. statement execution

```
if (dtack == 1'b1) begin: acked
   as     <= 1'b0;
   data   <= 16'hZZZZ;
   bus_rq <= 1'b0;
   state  <= IDLE;
end
```

But block boundaries may not be that obvious.

Statement blocks may not be necessarily clearly delimited. In Sample 2-12, two statements blocks are found: one before (and including) the *wait* statement, and one after. The *wait* statement may have never completed and the process was waiting forever. The subsequent sequential statements may not have executed. Thus, they form a separate statement block.

Sample 2-12. Blocks separated by a *wait* statement

```
address <= 16#FFED#;
ale     <= '1';
rw      <= '1';
wait until dtack = '1';
read_data := data;
ale     <= '0';
```

Statement Coverage

Did you execute all the statements?

Statement, line or block coverage measures how much of the total lines of code were executed by the verification suite. A graphical user interface usually lets the user browse the source code and quickly identify the statements that were not executed. Figure 2-14 shows, in a graphical fashion, a statement coverage report for a small portion of code from a model of a modem. The actual form of

the report from any code coverage tool or source code browser will likely be different.

Figure 2-14. Example of statement coverage

```
if (parity == ODD || parity == EVEN) begin
   tx <= compute_parity(data, parity);
   #(tx_time);
end
tx <= 1'b0;
#(tx_time);
if (stop_bits == 2) begin
   tx <= 1'b0;
   #(tx_time);
end
```

Why did you not execute all statements?

The example in Figure 2-14 shows that two out of the eight executable statements—or 25%—were not executed. To bring the statement coverage metric up to 100%, a desirable goal[5], it is necessary to understand what conditions are required to cause the execution of the uncovered statements. In this case, the parity must be set to either ODD or EVEN. Once the conditions have been determined, you must understand why they never occurred in the first place. Is it a condition that can never occur? Is it a condition that should have been verified by the existing verification suite? Or is it a condition that was forgotten?

It is normal for some statements not to be executed.

If it is a condition that can never occur, the code in question is effectively dead: It will never be executed. Removing that code is a definite option; it reduces clutter and increases the maintainability of the source code. However, a good defensive (and paranoid) coder often includes code that is not meant to be executed. This additional code simply monitors for conditions that should never occur and reports that an unexpected condition happened should the hypothesis prove false. This practice is very effective (see "Assertions" on page 64). Functional problems are positively identified near the source of the malfunction, without having to rely on the

5. But not necessarily achievable. For example, the *default* clause in a fully specified VHDL *case* statement should never be executed.

possibility that it produces an unexpected response at the right moment when you were looking for something else.

Sample 2-13. Defensive programming technique

```
case (mode[1:0]) // synopsys full_case
2'b00: ...
2'b10: ...
2'b01: ...
// synopsys translate_off
// coverage off
default: $write("Case was not really full!\n");
// coverage on
// synopsys translate_on
endcase
```

Your model can tell you if things are not as assumed.

Sample 2-13 shows an example of defensive modeling in synthesizeable *case* statements. Even though there is a directive instructing the synthesis tool that the *case* statement describes all possible conditions, it is possible for an unexpected condition to occur during simulation. If that were the case, the simulation results would differ from the results produced by the hardware implementation, and that difference would go undetected until a gate-level simulation is performed, or the device failed in the system.

Do not measure coverage for code not meant to be executed.

It should be possible to identify code that was not meant to be executed and have it eliminated from the code coverage statistics. In Sample 2-13, significant comments are used to remove the defensive coding statements from being measured by our hypothetical code coverage tool. Some code coverage tools may be configured to ignore any statement found between synthesis translation on/off directives. It may be more interesting to configure a code coverage tool to ensure that code included between synthesis translate on/off directives is indeed *not* executed!

Add testcases to execute all statements.

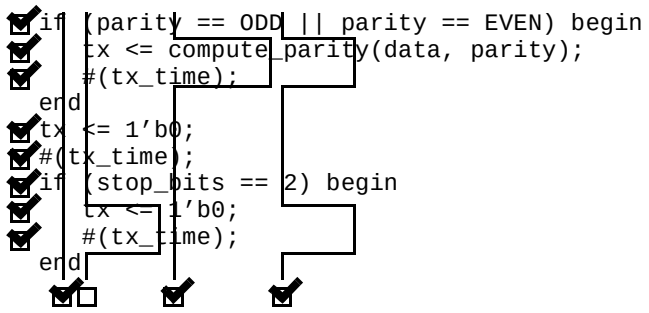
If the conditions that would cause the uncovered statements to be executed should have been verified, it is an indication that one or more testbenches are either not functionally correct or incomplete. If the condition was entirely forgotten, it is necessary to add to an existing testbench, create an entirely new one or make additional runs with different seeds.

Path Coverage

There is more than one way to execute a sequence of statements.

Path coverage measures all possible ways you can execute a sequence of statements. The code in Sample 2-14 has four possible paths: the first *if* statement can be either true or false. So can the second. To verify all paths through this simple code section, it is necessary to execute it with all possible state combinations for both *if* statements: false-false, false-true, true-false, and true-true.

Sample 2-14. Example of statement and path coverage



Why were some sequences not executed?

The current verification suite, although it offers 100% statement coverage, only offers 75% path coverage through this small code section. Again, it is necessary to determine the conditions that cause the uncovered path to be executed. In this case, a testcase must set the parity to neither ODD nor EVEN and the number of stop bits to two. Again, the important question one must ask is whether this is a condition that will ever happen, or if it is a condition that was overlooked.

Limit the length of statement sequences.

The number of paths in a sequence of statements grows exponentially with the number of control-flow statements. Code coverage tools give up measuring path coverage if their number is too large in a given code sequence. To avoid this situation, keep all sequential code constructs (in Verilog: *always* and *initial* blocks, *tasks* and *functions*; in VHDL: *processes*, *procedures* and *functions*) to under 100 lines.

Reaching 100% path coverage is very difficult.

Expression Coverage

There may be more than one cause for a control-flow change.

If you look closely at the code in Sample 2-15, you notice that there are two mutually independent conditions that can cause the first *if* statement to branch the execution into its *then* clause: parity being set to either ODD or EVEN. Expression coverage, as shown in Sample 2-15, measures the various ways paths through the code are executed. Even if the statement coverage is at 100%, the expression coverage is only at 50%.

Sample 2-15. Example of statement and expression coverage

```
if (parity == ODD || parity == EVEN) begin
   tx <= compute_parity(data, parity);
   #(tx_time);
end
tx <= 1'b0;
#(tx_time);
if (stop_bits == 2) begin
   tx <= 1'b0;
   #(tx_time);
end
```

Once more, it is necessary to understand why a controlling term of an expression has not been exercised. In this case, no testbench sets the parity to EVEN. Is it a condition that will never occur? Or was it another oversight?

Reaching 100% expression coverage is extremely difficult.

FSM Coverage

Statement coverage detects unvisited states.

Because each state in an FSM is usually explicitly coded using a choice in a *case* statement, any unvisited state will be clearly identifiable through uncovered statements. The state corresponding to an uncovered *case* statement choice was not visited during verification.

FSM coverage identifies state transitions.

Figure 2-15 shows a bubble diagram for an FSM. Although is has only five states, it has significantly more possible transitions: 14 possible transitions exist between adjoining states. State coverage of 100% can be easily reached through the sequence Reset, A, B, D, then C. However, this would yield only 36% transition coverage. To

completely verify the implementation of this FSM, it is necessary to ensure the design operates according to expectation for all transitions.

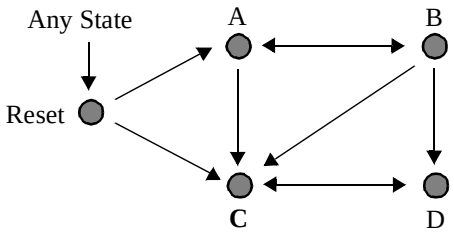


Figure 2-15. Example FSM bubble diagram

What about unspecified states?

The FSM illustrated in Figure 2-15 only shows five specified states. Once synthesized into hardware, a 3-bit state register will be necessary (maybe more if a different state encoding scheme, such as one-hot, is used). This leaves three possible state values that were not specified. What if some cosmic rays zap the design into one of these unspecified states? Will the correctness of the design be affected? Logic optimization may yield decode logic that creates an island of transitions within those three unspecified states, never letting the design recover into specified behavior unless reset is applied. The issues of design safety and reliability and techniques for ensuring them are beyond the scope of this book. But it is the role of a verification engineer to ask those questions.

What Does 100% Code Coverage Mean?

Completeness does not imply correctness.

The short answer is: Everything you wrote was executed. Code coverage indicates how *thoroughly* your entire verification suite exercises the source code. But it does not provide an indication, in any way, about the *correctness* or *completeness* of the verification suite. Figure 2-16 shows the reconvergence model for automatically extracted code coverage metrics. It clearly shows that it does

not help verify design intent, only that the RTL code, correct or not, was fully exercised.

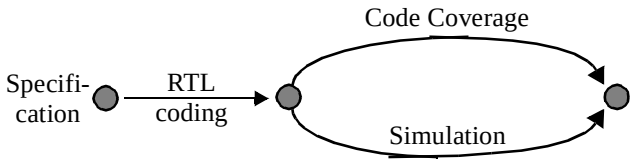


Figure 2-16. Reconvergent paths in automated code coverage

Results from code coverage tools should be interpreted with a grain of salt. They should be used to help identify corner cases that were not exercised by the verification suite or implementation-dependent features that were introduced during the implementation. You should also determine if the uncovered cases are relevant and deserve additional attention, or a consequence of the mindlessness of the coverage tool.

Code coverage lets you know if you are not done.

Code coverage indicates if the verification task is *not* complete through low coverage numbers. A high coverage number is by no means an indication that the job is over. Code coverage is an additional indicator for the completeness of the verification job. It can help increase your confidence that the verification job is complete, but it should not be your only indicator.

Code coverage tools can be used as profilers.

When developing models for simulation only, where performance is an important criteria, code coverage tools can be used for *profiling*. The aim of profiling is the opposite of code coverage. The aim of profiling is to identify the lines of codes that are executed most often. These lines of code become the primary candidates for performance optimization efforts.

FUNCTIONAL COVERAGE

Did you forget to verify some condition?

Functional coverage is another tool to help ensure that a bad design is not hiding behind passing testbenches. Although this methodology has been in use at some companies for quite some time, it is a recent addition to the arsenal of general-purpose verification tools. Functional coverage records relevant metrics (e.g., packet length, instruction opcode, buffer occupancy level) to ensure that the verification process has exercised the design through all of the interesting values. Whereas code coverage measures how much of the implementation has been exercised, functional coverage measures how much of the original design specification has been exercised.

It complements code coverage.

High functional coverage does not necessarily correlate with high code coverage. Whereas code coverage is concerned with recording the mechanics of code execution, functional coverage is concerned with the intent or purpose of the implemented function. For example, the decoding of a CPU instruction may involve separate *case* statements for each field in the opcode. Each *case* statement may be 100% code-covered due to combinations of field values from previously decoded opcodes. However, the particular combination involved in decoding a specific CPU instruction may not have been exercised.

It will detect errors of omission.

Sample 2-16 shows a *case* statement decoding a CPU instruction. Notice how the decoding of the RTS instruction is missing. If I relied solely on code coverage, I would be lulled in a false sense of completeness by having 100% coverage of this code. For code coverage to report a gap, the unexercised code must a priori exist. Functional coverage does not rely on actual code. It will report gaps in the recorded values whether the code to process them is there or not.

Sample 2-16. Example of coding error undetectable by code coverage

```
type OPCODE_TYP is [ADD, SUB, JMP, RTS, NOP];
...
case (OPCODE) is
when ADD => ...
when SUB => ...
when JMP => ...
when others => ...
end case;
```

It must be manually defined.

Code coverage tools were quickly adopted into verification processes because of their low adoption cost. They require very little additional action from the user: At most, execute one more command before compiling your code. Functional coverage, because it is a measure of values deemed to be interesting and relevant, must be manually specified. Since relevance and interest are qualities that are extracted from the intent of the design, functional coverage is not something that can be automatically extracted from the RTL source code. Your functional coverage metrics will be only as good as what you implement.

Metrics are collected at runtime and graded.

Like code coverage, functional coverage metrics are collected at runtime, during a simulation run. The values from individual runs are collected into a database or separate files. The functional coverage metrics from these separate runs are then merged for offline analysis. The marginal coverage of individual runs can then be graded to identify which runs contributed the most toward the overall functional coverage goal. These runs are then given preference in the regression suite, while pruning runs that did not significantly contribute to the objective.

Coverage data can be used at runtime.

Functional coverage tools usually provide a set of runtime procedures that let a testbench dynamically query a particular functional coverage metric. The testbench can then use the information to modify its current behavior. For example, it could increase the probability of generating values that have not been covered yet. It could decide to abort the simulation should the functional coverage not have significantly increased since the last query.

Although touted as a powerful mechanism by vendors, it is no silver bullet. Implementing the dynamic feedback mechanism is not easy: You have to correlate your stimulus generation process with the functional coverage metric, and ensure that one will cause the other to converge toward the goal. Dynamic feedback works best when there is a direct correlation between the input and the measured coverage, such as instruction types. It may be more efficient to achieve your goal with three or four runs of a simpler testbench without dynamic feedback than with a single run of a much more complex testbench.

Item Coverage

Did I generate all interesting and relevant values?

Item coverage is the recording of individual scalar values. It is a basic function of all functional coverage tools. The objective of the coverage metric is to ensure that all interesting and relevant values have been observed going to, coming out of, or in the design. Examples of item coverage include, but are not limited to, packet length, instruction opcode, interrupt level, bus transaction termination status, buffer occupancy level, bus request patterns and so on.

Define *what* to sample.

It is extremely easy to record functional coverage and be inundated with vast amounts of coverage data. But data is not the same thing as information. You must restrict coverage to only (but all!) values that will indicate how thoroughly your design has been verified. For example, measuring the value of the read and write pointers in a FIFO is fine if you are concerned about the full utilization of the buffer space and wrapping around of the pointer values. But if you are interested in the FIFO occupancy level (Was it ever empty? Was it ever full? Did it overflow?), you should measure and record the *difference* between the pointer values.

Define *where* to sample it.

Next, you must decide where in your testbench or design is the measured value accurate and relevant. For example, you can sample the opcode of an instruction at several places: at the output of the code generator, at the interface of the program memory, in the decoder register or in the execution pipeline. You have to ensure that a value, once recorded, is indeed processed or committed as implied by the coverage metric.

For example, if you are measuring opcodes that were executed, they should be sampled in the execution unit. Sampling them in the decode unit could result in false samples when the decode pipeline is flushed on branches or exceptions. Similarly, sampling the length of packets at the output of the generator may yield false samples: If a packet is corrupted by injecting an error during its transmission to the design in lower-level functions of the testbench, it may be dropped.

Define *when* to sample it.

Values are sampled at some point in time during the simulation. It could be at every clock cycle, whenever the address strobe signal is asserted, every time a request is made or after randomly generating a new value. You must carefully chose your sampling time. Over-

sampling will decrease simulation performance and consume database resources without contributing additional information.

The sampled data must also be stable so race conditions must be avoided between the sampled data and the sampling event (see “Read/Write Race Conditions” on page 209). To reduce the probability that a transient value is being sampled, functional coverage tools may delay the sampling of values to the end of their simulation cycle, before time is about to advance (see “The Simulation Cycle” on page 194 and “The Co-Simulation Cycle” on page 196).

High-level testbench functions usually operate on different values in zero-time within the same simulation cycle. If the functional coverage tool delays its sampling to the end of the simulation cycle, it will not be possible to sample all of the intermediate values. Only the last value will be recorded, resulting in under-sampling.

Define *why* we cover it.

If functional coverage is supposed to measure interesting and relevant values, it is necessary to define what makes those values so interesting and relevant. For example, measuring the functional coverage of a 32-bit address will yield over 4 billion “interesting and relevant” values. Not all values are created equal—but most are. Values may be numerically different but functionally equivalent. By identifying those functionally equivalent values into a single set, you can reduce the number of interesting and relevant values to a more manageable size. For example, based on the decoder architecture, addresses 0x00000001 through 0x7FFFFFFF and addresses 0x80000000 through 0x8FFFFFFE are functionally equivalent, reducing the number of relevant and interesting values to 4 sets (min, 1 to mid, mid to max-1, max).

It can detect invalid values.

If you can define sets of equivalent values, it is possible to define sets of invalid or unexpected values. Functional coverage can be used as an error detecting tool, just like an *if* statement in your testbench code. However, you should not rely on functional coverage to detect invalid values. Functional coverage is an optional runtime tool that may not be turned on at all times. If functional coverage is not enabled to improve simulation performance and if a value is defined as invalid in only the functional coverage, then an invalid value may go undetected.

It can report holes.

The ultimate purpose of functional coverage is to identify what remains to be done. During analysis, the functional coverage tool

can compare the number of value sets that contain at least one sample against the total number of value sets. Any value set that does not contain at least one sample is a hole in your functional coverage. By enumerating the empty value sets, you can focus on the holes in your test cases and complete your verification sooner rather than continue to exercise functionality that has already been verified.

For this enumeration to be possible, the total number of value sets must be relatively small. For example, it is practically impossible to fill the coverage for a 32-bit value without broad value sets. The number of holes will be likely in the millions, making enumeration impossible. You should strive to limit the number of possible value sets as much as possible. For example, Specman Elite has a default limit of 16 value sets for hole enumeration.

Cross Coverage

Did I generate all interesting *combination* of values?

Whereas item coverage is concerned with individual scalar values, cross coverage measures the presence or occurrence of combinations of values. It helps answer questions like, “Did I inject a corrupted packet on all ports?” “Did we execute all combinations of opcodes and operand modes?” and “Did this state machine visit each state while that buffer was full, not empty and empty?” Cross coverage can involve more than two scalar items. However, the number of possible value sets grows factorially with the number of crossed items.

Similar to item coverage.

Mechanically, cross coverage is identical to item coverage. Specific values are sampled at specific locations at specific points in time with specific value sets. The only difference is that two or more values are sampled instead of one.

Can be done in post-processing step.

It may be possible to perform offline cross coverage by post-processing the item coverage metrics. In some tools, it is the only cross coverage mechanism available. To enable offline cross-coverage analysis, it is necessary to sample the simulation time along with each individual value sample. The recording of additional information such as simulation time increases the size of the coverage data and reduces runtime performance. Most tools make gathering additional cross coverage information optional, and this feature is usually turned off by default. As shown in Sample 2-17, the cross

coverage report is generated by identifying the values that were sampled at the same simulation time.

Sample 2-17. Example of offline cross-coverage analysis

```
 Packet    Packet           Length x Valid
 Length     Valid
-------------------------------------------
short @10  good@10               good   bad
long  @20  bad @20       short     X
medium@30  good@30       medium    X
                         long            X
```

Offline cross-coverage reports may yield false samples.

As long as each item value is sampled at different points in time, offline cross-coverage analysis works fine. When recording values located inside an RTL design or a bus-functional model sampled at clock edges, there can be only one value per simulation time. However, covering values in higher-level testbench functions, which operate in zero-time, may result in multiple values sampled at the same simulation time. Sample 2-18 shows the same item coverage as before, but sampled when all of the packets were generated at the same time. This approach yields an incorrect offline cross-coverage report. The same cross-coverage measure, if collected at runtime, would yield a correct report.

Sample 2-18. Example of invalid offline cross-coverage analysis

```
 Packet    Packet           Length x Valid
 Length     Valid
-------------------------------------------
short @10  good@10               good   bad
long  @10  bad @10       short     X      X
medium@10  good@10       medium    X      X
                         long      X      X
```

Transition Coverage

Did I generate all interesting *sequences* of values?

Whereas cross coverage is concerned with combination scalar values at the same point in time, transition coverage measures the presence or occurrence of sequences of values. Transition coverage helps answer questions like, “Did I perform all combinations of back-to-back read and write cycles?” “Did we execute all combinations of arithmetic opcodes followed by test opcodes?” and “Did this state machine traverse all significant paths?” Transition coverage can involve more than two consecutive values of the same sca-

lar item. However, the number of possible value sets grows factorially with the number of transition states.

Similar to item coverage.

Mechanically, transition coverage is identical to item coverage. Specific values are sampled at specific locations at specific points in time with specific value sets. The only difference is that a sample is said to have occurred in a value set after two or more consecutive item samples instead of one. The other difference is that transition can overlap, hence two transition samples may be composed of the same item sample.

Similar to FSM path coverage.

Conceptually, transition coverage is identical to FSM path coverage (see "FSM Coverage" on page 52). Both record the consecutive values at a particular location of the design (for example, a state register), and both compare against the possible set of paths. But unlike FSM coverage tools, which are limited to state registers in RTL code, transition coverage can be applied to any sampled value in testbenches and behavioral models.

Transition coverage reflects intent.

Because transition coverage is (today) manually specified from the intent of the design or the implementation, it provides a true independent path to verifying the correctness of the design and the completeness of the verification. It can detect invalid transitions as well as specify transitions that may be missing from the implementation of the design.

What Does 100% Functional Coverage Mean?

It indicates completeness, not correctness.

Functional coverage indicates which interesting and relevant conditions were verified. It provides an indication of the *thoroughness* of the implementation of the verification plan. Unless some value sets are defined as invalid, it cannot provide an indication, in any way, about the *correctness* of those conditions or of the design's response to those conditions. Functional coverage metrics are only as good as the functional coverage model you have defined. Coverage of 100% means that you've covered all of the coverage points you included in the simulation. It makes no statement about the *completeness* of your functional coverage model.

Results from functional coverage tools should also be interpreted with a grain of salt. Since they are generated by additional testbench code, they have to be debugged and verified for correctness

before being trusted. They will help identify additional interesting conditions that were not included in the verification plan.

If a metric is not interesting, don't measure it.

It is extremely easy to define functional coverage metrics and generate many reports. If coverage is not measured according to a specific purpose, you will soon drown under megabytes of functional coverage reports. And few of them will ever be close to 100%. It will also become impossible to determine which report is significant or what is the significance of the holes in others. The verification plan (see the next chapter) should serve as the functional specification for the coverage models, as well as for the rest of the verification environment. If a report is not interesting or meaningful to look at, if you are not eager to look at a report after a simulation run, then you should question its existence.

Functional coverage lets you know if you are done.

When used properly, functional coverage becomes a formal specification of the verification plan. Once you reach 100% functional coverage, it indicates that you have created and exercised all of the relevant and interesting conditions you originally identified. It confirms that you have implemented everything in the verification plan. However, it does not provide any indication of the completeness of the verification plan itself or the correctness of the design under such conditions.

VERIFICATION LANGUAGES

VHDL and Verilog are simulation languages, not verification languages.

Verilog was designed with a focus on describing low-level hardware structures. Verilog-2001 only recently introduced support for basic high-level data structures. VHDL was designed for very large design teams. It strongly encapsulates all information and communicates strictly through well-defined interfaces. Very often, these limitations get in the way of an efficient implementation of a verification strategy. VHDL and Verilog also lack features important in efficiently implementing a modern verification process.

Verification languages can raise the level of abstraction.

As mentioned in Chapter 1, one way to increase productivity is to raise the level of abstraction used to perform a task. High-level languages, such as C or Pascal, raised the level of abstraction from assembly-level, enabling software engineers to become more productive. Similarly, computer languages specifically designed for verification are able to raise the level of abstraction compared to general-purpose simulation languages. Hardware verification lan-

guages maintain important concepts necessary to interact with hardware: time, concurrency and instantiation. They also offer features that help raise the level of abstraction: complex data types, object-orientedness with inheritance and temporal assertions.

Verification language can automate verification.

If the main benefit of a hardware verification language was the higher level of abstraction and object-orientedness, then *C++* would have long been identified as the best solution[6]: It is free and widely known. The main benefit of HVLs, as shown in Figure 2-17, is their capability of automating a portion of the verification by randomly generating stimulus, collecting functional coverage to identify holes then the ability to add constraints easily to create more stimulus targeted to fill those holes. To support this productivity cycle, some HVLs offer constrainable random generation, functional coverage measurement and a code extension mechanism.

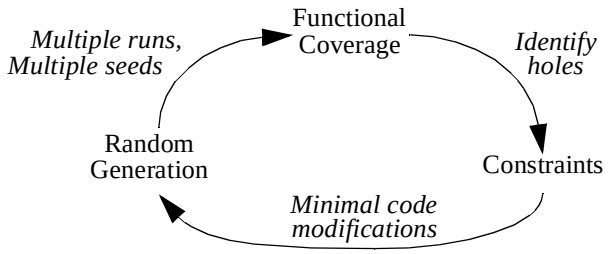


Figure 2-17. HVL productivity cycle

Several verification languages exist.

At the time of this writing, commercial verification language solutions include *e* from Verisity, *OpenVera* from Synopsys and *RAVE* from Forte. Open-source or public-domain solutions are available: the *SystemC Verification Library*[7] from Cadence and *Jeda* from Juniper Networks. Accelera is working on introducing HVL functionality in *SystemVerilog*. There is also a plethora of home-grown proprietary solutions based on *C++*, Perl or TCL.

The definition of what makes a language an HVL is still nebulous. Of the languages mentioned, most do not include all of the features identified in the productivity cycle. The large number of verification language solutions confirms that the industry recognizes the

6. C++ still lacks a native concept of time, concurrency and instantiation.
7. Formerly known as *TestBuilder.*

limitations of Verilog and VHDL for verification. It is also a classic characteristic of a young market, before coalescing around one or two market leaders and de-facto standards.

You must learn the basics of verification before learning verification languages.

In this book, I use VHDL and Verilog as the first implementation medium for the basic components of the verification infrastructure and to introduce the concept of self-checking transaction-level directed testbenches. Even though HVLs make implementing such testbenches easier (especially the self-checking part), you still need to plan your verification, define your verification objectives, design its strategy and architecture, design the stimulus, determine the expected response and compare the actual output. These are concepts that can be learned and applied using VHDL or Verilog.

Coverage-driven constrained random approach requires HVLs.

All HVLs can be used as if they were souped-up Verilog or VHDL languages. In fact, most HVL solutions are just that. But if your HVL contains all of the features required to support the HVL productivity cycle, the verification process must be approached—and implemented—in a different fashion.

This change is just like taking advantage of the productivity offered by logic synthesis tools: It requires an approach different from schematic capture. To successfully implement a coverage-driven constrained random verification approach, you need to modify the way you plan your verification, design its strategy and implement the testcases. Because Verilog and VHDL lack all of the required features, *e* and OpenVera will be used to illustrate these concepts.

ASSERTIONS

Assertions detect conditions that should always be true.

An assertion boils down to an *if* statement and an error message should the expression in the *if* statement become false. Assertions have been used in software design for many years: the *assert()* function has been part of the ANSI C standard from the beginning. In software for example, assertions are used to detect conditions such as NULL pointers or empty lists. VHDL has had an *assert* statement from day one too, but it was never a popular construct—except to terminate a simulation (see Sample 5-44 on page 264 for an example).

Hardware assertions require a form of temporal language.

A software assertion simply checks that, at the time the *assert* statement is executed, the condition evaluates to TRUE. This simple zero-time test is not sufficient for supporting assertions in hardware designs. In hardware, functional correctness usually involves behavior over a period of time. Some hardware assertions such as, "This state register is one-hot encoded." or "This FIFO never overflows." can be expressed as immediate, zero-time expressions. But checking simple hardware assertions such as, "This signal must be asserted for a single clock period." or "A request must always be followed by a grant or abort within 10 clock cycles." require that the assertion condition be evaluated over time. Thus, assertions require the use of a temporal language to be able to describe relationships over time.

There are two classes of assertions.

Assertions fall in two broad classes: those specified by the designer and those specified by the verification engineer.

- Implementation assertions are specified by the designers.
- Specification assertions are specified by the verification engineers.

Implementation assertions verify assumptions.

Implementation assertions are used to formally encode the designer's assumptions about the interface of the design or conditions that are indications of misuse or design faults. For example, the designer of a FIFO would add assertions to detect if it ever overflows or underflows or that, because of a design limitation, the *write* and *read* pulses are ever asserted at the same time. Because implementation assertions are specified by the designer, they will not detect discrepancies between the functional intent and the design. But implementation assertions will detect discrepancies between the design assumptions and the implementation.

Specification assertions verify intent.

Specification assertions formally encode expectations of the design based on the functional intent. These assertions are used as a functional error detection mechanism and supplement the error detections performed in the self-checking section of testbenches. Specification assertions are typically *white-box* strategies because the relationships between the primary inputs and outputs of a modern design are too complex to be described in today's temporal languages. For example, rather than relying on the scoreboard to detect that an arbiter is not fair, it is much simpler to perform this check using a block-level assertion.

Assertion specification is a complex topic.

This simple introduction to assertions does not do justice to the richness and power—and ensuing complexity—of assertion. Entire books ought to be (and probably are being) written about the subject.

Simulation Assertions

The OVL started the storm.

Assertions took the hardware design community by storm when Foster and Bening's book[8] introduced the concept using a library of predefined Verilog modules that implemented all of the common design assertions. The library, available in source form as the *Open Verification Library*,[9] was a clever way of using Verilog to specify temporal expressions. Foster, then at Hewlett-Packard, had a hidden agenda: Get designers to specify design assertions he could then try to prove using formal methods. Using Verilog modules was a convenient solution to ease the adoption of these assertions by the designers. The reality of what happened next proved to be even more fruitful.

They detect errors close in space and time to the fault.

If a design assumption is violated during simulation, the design will not operate correctly. The cause of the violation is not important: It could be a misunderstanding by the designer of the block or the designer of the upstream block or an incorrect testbench. The relevant fact is that the design is failing to operate according to the original intent. The symptoms of that low-level failure are usually not visible (if at all) until the affected data item makes its way to the outputs of the design and is flagged by the self-checking structure.

An assertion formally encoding the design assumption immediately fires and reports a problem at the time it occurs, in the area of the design where it occurs. Debugging and fixing the assertion failure (whatever the cause) will be a lot more efficient than tracing back the cause of a corrupted packet. In one of Foster's projects, 85% of the design errors where caught and quickly fixed using simulated assertions.

8. Harry Foster and Lionel Bening, "*Principles of Verifiable RTL Design*," second edition, Kluwer Academic Publisher, ISBN 0-7923-7368-5.
9. See `http://verificationlib.org`.

Formal Assertion Proving

Is it possible for an assertion to fire?

Simulation can show only the presence of bugs, never prove their absence. The fact that an assertion has never reported a violation throughout a series of simulation does not mean that it can never be violated. Tools like code and functional coverage can satisfy us that a portion of a design was thoroughly verified—but there will (and should) always be a nagging doubt.

Model checking can mathematically prove or disprove an assertion.

Formal tools called *model checker* or *assertion provers* can mathematically prove that, given an RTL design and some assumptions about the relationships of the input signals, an assertion will always hold true. If a counter example is found, the formal tool will provide details on the sequence of events that leads to the assertion violation. It is then up to you to decide if this sequence of events is possible, given additional knowledge about the environment of the design.

Some assertions are used as assumptions.

Given total freedom over the inputs of a design, you can probably violate any and all assertions about its implementation. Fortunately, the usage of the inputs of a design are subject to limitation and rules to ensure proper operation of the design. Furthermore, these input signals usually come from other designs that do not behave (one hopes!) erratically. When proving some assertions on a design, it is thus necessary to supply assertions on the inputs or state of the design. The latter assertions are not proven. Rather, they are assumed to be true and used to constrain the solution space for the proof.

Assumptions need to be proven too.

The correctness of a proof depends on the correctness of the assumptions[10] made on the design inputs. Should any assumption be wrong, the proof no longer stands. An assumption on a design's inputs thus becomes an assertion to be proven on the upstream design supplying those inputs.

Semi-formal tools combine model checking with simulation.

Semi-formal tools are hybrid tools that combine formal methods with simulation. Semi-formal tools are an attempt to bridge the gap between a familiar technology (simulation) and the fundamentally

10. The formal verification community calls these input assertions "constraints." I used the term "assumptions" to differentiate them from random-generation constraints, which are randomization concepts.

different formal tools. They use intermediate simulation information—such as the current state of a design—as a starting point for proving or disproving assertions.

Use formal methods to prove cases uncovered in simulation.

Formal verification does not replace simulation or make it obsolete. Simulation (including simulated assertions) is the lawnmower of the verification garden: It is still the best tool for covering broad swaths of functionality and for weeding out the easy-to-find and some not-so-easy-to-find bugs. Formal verification puts the finishing touch on those hard-to-reach corners in critical and important design sections and ensures that the job is well done. Using functional coverage metrics collected from simulation (for example, request patterns on an arbiter), identifies conditions that remain to be verified. If those conditions would be difficult to create within the simulation environment, using these conditions as assumptions, proves the correctness of the design for the remaining uncovered cases.

REVISION CONTROL

Are we all looking at the same thing?

One of the major difficulties in verification is to ensure that what is being verified is actually what will be implemented. When you compile a Verilog source file, what is the guarantee that the design engineer will use that *exact same file* when synthesizing the design?

When the same person verifies and then synthesizes the design, this problem is reduced to that person using proper file management discipline. However, as I hope to have demonstrated in Chapter 1, having the same person perform both tasks is not a reliable functional verification process. It is more likely that separate individuals perform the verification and synthesis tasks.

Files must be centrally managed.

In very small and closely knit groups, it may be possible to have everyone work from a single directory, or to have the design files distributed across a small number of individual directories. Everyone agrees where each other's files are, then each is left to his or her own device. This situation is very common and very dangerous: How can you tell if the designer has changed a source file and maybe introduced a functional bug since you last verified it?

It must be easy to get at all the files, from a single location.

This methodology is not scalable either. It quickly breaks down once the team grows to more than two or three individuals. And it does not work at all when the team is distributed across different physical or geographical areas. The verification engineer is often the first person to face the non-scalability challenge of this environment. Each designer is content working independently in his or her own directories. Individual designs, when properly partitioned, rarely need to refer to some other design in another designer's working directory. As the verification engineer, your first task is to integrate all the pieces into a functional entity. That's where the difficulties of pulling bits and pieces from heterogeneous working environments scattered across multiple file servers become apparent.

The Software Engineering Experience

HDL models are software projects!

For about 25 years, software engineering has been dealing with the issues of managing a large number of source files, authored by many different individuals, verified by others and compiled into a final product. Make no mistake: Managing an HDL-based hardware design project is no different than managing a software project.

Free and commercial tools are available.

To help manage files, software engineers use source control management systems. Some are available, free of charge, either bundled with the UNIX operating systems (RCS, CVS, SCCS), or distributed by the GNU project (RCS, CVS) and available in source form at:

```
ftp://prep.ai.mit.edu/pub/gnu
```

Commercial systems, some very sophisticated, are also available.

All source files are centrally managed.

Figure 2-18 shows how source files are managed using a source control management system. All accesses and changes to source files are mediated by the management system. Individual authors

and users interact solely through the management system, not by directly accessing files in working directories.

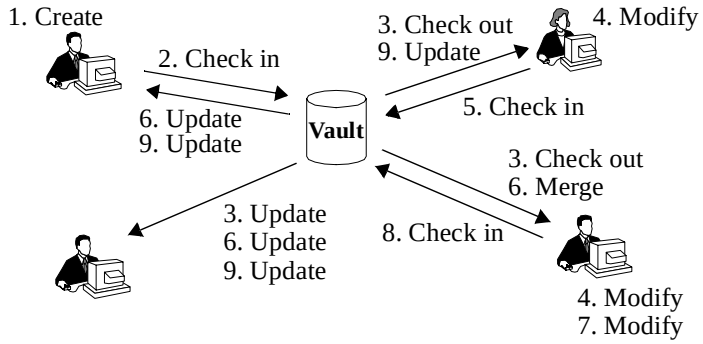


Figure 2-18. Data flow in a source control system

The history of a file is maintained.

Source code management systems maintain not only the latest version of a file, but also keep a complete history of each file as separate *versions*. Thus, it is possible to recover older versions of files, or to determine what changed from one version to another. It is a good idea to frequently *check in* file versions. You do not have to rely on a backup system if you ever accidentally delete a file. Sometimes, a series of modifications you have been working on for the last couple of hours is making things worse, not better. You can easily roll back the state of a file to a previous version known to work.

The team owns all the files.

When using a source management system, files are no longer owned by individuals. Designers may be nominally responsible for various sections of a design, but anyone—with the proper permissions—can make any change to any file. This lets a verification engineer fix bugs found in RTL code without having to rely on the designer, busy trying to get timing closure on another portion of the design. The source management system mediates changes to files either through exclusive locks, or by merging concurrent modifications.

Configuration Management

Each user works from a *view* of the file system.

Each engineer working on a project managed with a source control system has a private *view* of all the source files (or a subset thereof) used in the project. Figure 2-19 shows how two users may have two different views of the source files in the management system. Views need not be always composed of the latest versions of all the files. In fact, for a verification engineer, that would be a hindrance. Files checked in on a regular basis by their authors may include syntax errors, be simple placeholders for future work, or be totally broken. It would be very frustrating if the model you were trying to verify kept changing faster than you could identify problems with it.

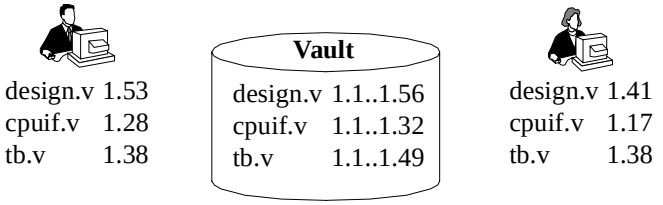


Figure 2-19. User views of managed source files

Configurations are created by tagging a set of versions.

All source management systems use the concept of symbolic tags that can be attached to specific versions of files. You may then refer to particular versions of files, or set of files, using the symbolic name, without knowing the exact version number they refer to. In Figure 2-19, the user on the left could be working with the versions that were tagged as “ready to simulate” by the author. The user on the right, the system verification engineer, could be working with the versions that were tagged as “golden” by the ASIC verification engineer.

Configuration management translates to tag management.

Managing releases becomes a problem of managing tags, which can be a complex task. Table 2-1 shows a list of tags that could be used in a project to identify the various versions of a file as it progresses through the design process. Some tags, such as the “Version_M.N” tag, never move once applied to a specific version. Others, such as the “Submit” tag, move to newer versions as the development of the design progresses. Before moving a tag, it may be a good idea to leave a trace of the previous position of a tag. One possible mechanism for doing so is to append the date to the tag name. For example, the old “Submit” version gets tagged with the new tag

"Submit_000302" on March 2nd, 2000 and the "Submit" tag is moved to the latest version.

Table 2-1. Example tags for release management

Tag Name	Description
Submit	Ready to submit to functional verification. Author has verified syntax correctness and basic level of functionality.
Bronze	Passes a basic set of functional testcases. Release is sufficiently functional for integration.
Silver	Passes all functional testcases.
Gold	Passes all functional testcases and meets coding coverage guidelines (requires additional corner-case testcases).
To_Synthesis	Ready to submit to synthesis. Usually matches "Silver" or "Gold".
To_Layout	Ready to submit to layout. Usually matches "Gold".
Version_M.N	Version that was manufactured. Matches corresponding "To_Layout" release. Future versions of the same chip will move tags beyond this point.
ON_YYMMDD	Some meaningful release on the specified date.

Working with Releases

Views can become out-of-date as new versions of files are checked into the source management system database and tags are moved forward.

Releases are specific configurations.

The author of the RTL for a portion of the design would likely always work with the latest version of the files he or she is actively working on, checking in and updating them frequently (typically at relevant points of code development throughout the day and at the end of each day). Once the source code is syntactically correct and its functionality satisfies the designer (by using a few ad hoc test-

benches), the corresponding version of the files are tagged as ready for verification.

Users must update their view to the appropriate release.

You, as the verification engineer, must be constantly on the lookout for updates to your view. When working on a particularly difficult testbench, you may spend several days without updating your view to the latest version ready to be verified. That way, you maintain a consistent view of the design under test and limit changes to the testbenches, which you make. Once the actual verification and debugging of the design starts, you probably want to refresh your view to the latest "ready-to-verify" release of the design before running a testbench.

Update often.

When using a concurrent development model where multiple engineers are working in parallel on the same files, it is important to check in modifications often, and update your view to merge concurrent modifications even more often. If you wait too long, there is a greater probability of collisions that will require manual resolution. The concept of concurrently modifying files then merging the differences sounds impossibly risky at first. However, experience has shown that different functions or bug fixes rarely involve modification to the same lines of source code. As long as the modifications are separated by two or three lines of unmodified code, merging will proceed without any problems. Trust me, concurrent development is the way to go!

You can be notified of new releases.

An interesting feature of some source management systems is the ability to issue email notification whenever a significant event occurs. For example, such a system could send e-mail to all verification engineers whenever the tag identifying the release that is ready for verification is moved. Optionally, the e-mail could contain a copy of the descriptions of the changes that were made to the source files. Upon receiving such an e-mail, you could make an informed decision about whether to update your view immediately.

ISSUE TRACKING

All your bug are belong to us!

The job of any verification engineer is to find bugs. Under normal conditions, you should expect to find functional irregularities. You should be *really* worried if no problems are being found. Their occurrence is normal and do not reflect the abilities of the hardware designers. Even the most experienced software designers write code that includes bugs, even in the simplest and shortest routines. Now that we've established that bugs *will* be found, how will you deal with them?

Bugs must be fixed.

Once a problem has been identified, it *must* be resolved. All design teams have informal systems to track issues and ensure their resolutions. However, the quality and scalability of these informal systems leaves a lot to be desired.

What Is an Issue?

Is it worth worrying about?

Before we discuss the various ways issues can be tracked, we must first consider what is an issue worth tracking. The answer depends highly on the tracking system used. The cost of tracking the issue should not be greater than the cost of the issue itself. However, do you want the tracking system to dictate what kind of issues are tracked? Or, do you want to decide on what constitutes a trackable issue, then implement a suitable tracking system? The latter position is the one that serves the ultimate goal better: Making sure that the design is functionally correct.

An issue is *anything* that can affect the functionality of the design:

1. Bugs found during the execution of a testbench are clearly issues worth tracking.
2. Ambiguities or incompleteness in the specification document should also be tracked issues. However, typographical errors definitely do not fit in this category.
3. Architectural decisions and trade-offs are also issues.
4. Errors found at all stages of the design, in the design itself or in the verification environment should be tracked as well.
5. If someone thinks about a new relevant testcase, it should be filed as an issue.

When in doubt, track it.

It is not possible to come up with an exhaustive list of issues worth tracking. Whenever an issue comes up, the only criterion that determines whether it should be tracked, should be its effect on the correctness of the final design. If a bad design can be manufactured when that issue goes unresolved, it *must* be tracked. Of course, all issues are not created equal. Some have a direct impact on the functionality of the design, others have minor secondary effects. Issues should be assigned a priority and be addressed in order of that priority.

You may choose not to fix an issue.

Some issues, often of lower importance, may be consciously left unresolved. The design or project team may decide that a particular problem or shortcoming is an acceptable limitation for this particular project and can be left to be resolved in the next incarnation of the product. The principal difficulty is to make sure that the decision was a conscious and rational one!

The Grapevine System

Issues can be verbally reported.

The simplest, and most pervasive issue tracking system is the *grapevine*. After identifying a problem, you walk over to the hardware designer's cubicle (assuming you are not the hardware designer as well!) and discuss the issue. Others may be pulled into the conversation or accidentally drop in as they overhear something interesting being debated. Simple issues are usually resolved on the spot. For bigger issues, everyone may agree that further discussions are warranted, pending the input of other individuals. The priority of issues is implicitly communicated by the insistence and frequency of your reminders to the hardware designer.

It works only under specific conditions.

The grapevine system works well with small, closely knit design groups, working in close proximity. If temporary contractors or part-time engineers are on the team, or members are distributed geographically, this system breaks down as instant verbal communications are not readily available. Once issues are verbally resolved, no one has a clear responsibility for making sure that the solution will be implemented.

You are condemned to repeat past mistakes.

Also, this system does not maintain any history. Once an issue is resolved, there is no way to review the process that led to the decision. The same issue may be revisited many times if the implementation of the solution is significantly delayed. If the proposed resolution turns out to be inappropriate, the team may end up going

in circles, repeatedly trying previous solutions. Without history, you are condemned to repeat it. There is no opportunity for the team to learn from its mistakes. Learning is limited to individuals, and to the extent that they keep encountering similar problems.

The Post-It System

Issues can be tracked on little pieces of paper.

When teams become larger, or when communications are no longer regular and casual, the next issue tracking system that is used is the 3M Post-It™ note system. It is easy to recognize at a glance: Every team member has a number of telltale yellow pieces of paper stuck around the periphery of their computer monitor.

If the paper disappears, so does the issue.

This evolutionary system only addresses the lack of ownership of the grapevine system: Whoever has the yellow piece of paper is responsible for its resolution. This ownership is tenuous at best. Many issues are "resolved" when the sticky note accidentally falls on the floor and is swept away by the janitorial staff.

Issues cannot be prioritized.

With the Post-It system, issues are not prioritized. One bug may be critical to another team member, but the owner of the bug may choose to resolve other issues first simply because they are simpler and because resolving them instead reduces the clutter around his computer screen faster. All notes look alike and none indicate a sense of urgency more than the others.

History will repeat itself.

And again, the Post-It system suffers from the same learning disabilities as the grapevine system. Because of the lack of history, issues are revisited many times, and problems are recreated repeatedly.

The Procedural System

Issues can be tracked at group meetings.

The next step in the normal evolution of issue tracking is the procedural system. In this system, issues are formally reported, usually through free-form documents such as e-mail messages. The outstanding issues are reviewed and resolved during team meetings.

Only the biggest issues are tracked.

Because the entire team is involved and the minutes of meetings are usually kept, this system provides an opportunity for team-wide learning. But the procedural system consumes an inordinate amount of precious meeting time. Because of the time and effort involved in tracking and resolving these issues, it is usually reserved for the

most important or controversial ones. The smaller, less important—but much more numerous—issues default back to the grapevine or Post-It note systems.

Computerized System

Issues can be tracked using databases.

A revolution in issue tracking comes from using a computer-based system. In such a system, issues must be seen through to resolution: Outstanding issues are repeatedly reported loud and clear. Issues can be formally assigned to individuals or list of individuals. Their resolution need only involve the required team members. The computer-based system can automatically send daily or weekly status reports to interested parties.

A history of the decision making process is maintained and archived. By recording various attempted solutions and their effectiveness, solutions are only tried once without going in circles. The resolution process of similar issues can be quickly looked-up by anyone, preventing similar mistakes from being committed repeatedly.

But it should not be easier to track them verbally or on paper.

Even with its clear advantages, computer-based systems are often unsuccessful. The main obstacle is their lack of comparative ease-of-use. Remember: The grapevine and Post-It systems are readily available at all times. Given the schedule pressure engineers work under and the amount of work that needs to be done, if you had the choice to report a relatively simple problem, which process would you use:

1. Walk over to the person who has to solve the problem and verbally report it.
2. Describe the problem on a Post-It note, then give it to that same person (and if that person is not there, stick it in the middle of his or her computer screen).
3. Enter a description of the problem in the issue tracking database and never leave your workstation?

It should not take longer to submit an issue than to fix it.

You would probably use the one that requires the least amount of time and effort. If you want your team to use a computer-based issue tracking system successfully, then select one that causes the smallest disruption in their normal work flow. Choose one that is a

simple or transparent extension of their normal behavior and tools they already use.

I was involved in a project where the issue tracking system used a proprietary X-based graphical interface. It took about 15 seconds to bring up the entire interface on your screen. You were then faced with a series of required menu selections to identify the precise division, project, system, sub-system, device and functional aspect of the problem, followed by several other dialog boxes to describe the actual issue. Entering the simplest issue took *at least* three to four minutes. And the system could not be accessed when working from home on dial-up lines. You can guess how successful that system was...

Email-based systems have the greatest acceptance.

The systems that have the most success invariably use an e-mail-based interface, usually coupled with a Web-based interface for administrative tasks and reporting. Everyone on your team uses e-mail. It is probably already the preferred mechanism for discussing issues when members are distributed geographically or work in different time zones. Having a system that simply captures these e-mail messages, categorizes them and keeps track of the status and resolution of individual issues (usually through a minimum set of required fields in the e-mail body or header), is an effective way of implementing a computer-based issue tracking system.

METRICS

Metrics are essential management tools.

Managers love metrics and measurements. They have little time to personally assess the progress and status of a project. They must rely on numbers that (more or less) reflect the current situation.

Metrics are best observed over time to see trends.

Metrics are most often used in a static fashion: “What are the values today?” “How close are they to the values that indicate that the project is complete?” The odometer reports a static value: How far have you travelled. However, metrics provide the most valuable information when observed over time. Not only do you know where you are, but also you can know how fast you are going, and what direction you are heading. (Is it getting better or worse?)

Historical data should be used to create a baseline.

When compared with historical data, metrics can paint a picture of your learning abilities. Unless you know how well (or how poorly) you did last time, how can you tell if you are becoming better at

your job? It is important to create a baseline from historical data to determine your productivity level. In an industry where the manufacturing capability doubles every 18 months, you cannot afford to maintain a constant level of productivity.

Metrics can help assess the verification effort.

There are several metrics that can help assess the status, progress and productivity of functional verification. One has already been introduced: code coverage.

Code-Related Metrics

Code coverage may not be relevant.

Code coverage measures how thoroughly the verification suite exercises the source code being verified. That metric should climb steadily toward 100% over time. From project to project, it should climb faster, and get closer to 100%.

However, code coverage is not a suitable metric for all verification projects. It is an effective metric for the smallest design unit that is individually specified (such as an FPGA, a reusable component or an ASIC). But it is ineffective when verifying designs composed of sub-designs that have been independently verified. The objective of that verification is to confirm that the sub-designs are interfaced and cooperate properly, not to verify their individual features. It is unlikely (and unnecessary) to execute all the statements.

The number of lines of code can measure implementation efficiency.

The total *number of lines of code* that is necessary to implement a verification suite can be an effective measure of the effort required in implementing it. This metric can be used to compare the productivity offered by new verification languages or methods. If they can reduce the number of lines of code that need to be written, then they should reduce the effort required to implement the verification.

Lines-of-code ratio can measure complexity.

The *ratio of lines of code* between the design being verified and the verification suite may measure the complexity of the design. Historical data on that ratio could help predict the verification effort for a new design by predicting its estimated complexity.

Code change rate should trend toward zero.

If you are using a source control system, you can measure the *source code changes* over time. At the beginning of a project, code changes at a very fast rate as new functionality is added and initial versions are augmented. At the beginning of the verification phase, many changes in the code are required by bug fixes. As the verification progresses, the rate of changes should decrease as there are

fewer and fewer bugs to be found and fixed. Figure 2-20 shows a plot of the expected code change rate over the life of a project. From this metric, you are able to determine if the code is becoming stable, or identify the most unstable sections of a design.

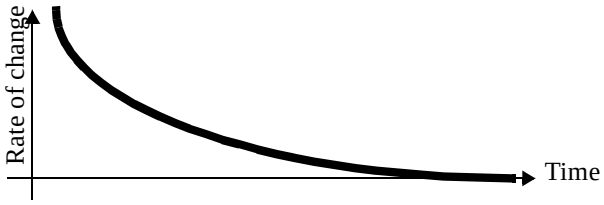


Figure 2-20. Ideal code change rate metric over time

Quality-Related Metrics

Quality is subjective, but it can be measured indirectly.

Quality-related metrics are probably more directly related with the functional verification than other productivity metrics. Quality is a subjective value, yet, it is possible to find metrics that correlate with the level of quality in a design. This is much like the number of customer complaints or the number of repeat customers can be used to judge the quality of retail services.

Functional coverage can measure testcase completeness.

Functional coverage measures the range and combination of input and output values that were submitted to and observed from the design, and of selected internal values. By assigning a weight to each functional coverage metric, it can be reduced to a single functional coverage grade measuring how thoroughly the functionality of the design was exercised. By weighing the more important functional coverage measures more than the less important ones, it gives a good indicator of the progress of the functional verification. This metric should evolve rapidly toward 100% at the beginning of the project then significantly slow down as only hard-to-reach functional coverage points remain.

A simple metric is the number of known issues.

The easiest metric to collect is the *number of known outstanding issues*. The number could be weighed to count issues differently according to their severity. When using a computer-based issue tracking system, this metric, as well as trends and rates, can be easily generated. Are issues accumulating (indicating a growing quality problem)? Or, are they decreasing and nearing zero?

Code will be worn out eventually.

If you are dealing with a reusable or long-lived design, it is useful to measure the *number of bugs found during its service life*. These

are bugs that were not originally found by the verification suite. If the number of bugs starts to increase dramatically compared to historical findings, it is an indication that the design has outlived its useful life. It has been modified and adapted too many times and needs to be re-designed from scratch. Throughout the normal life cycle of a reusable design, the number of outstanding issues exhibits a behavior as shown in Figure 2-21.

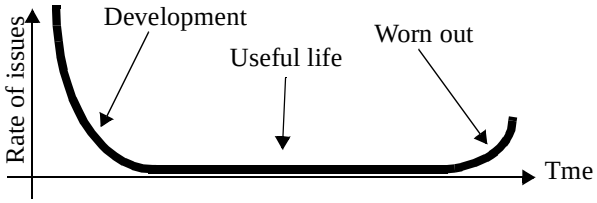


Figure 2-21. Number of outstanding issues throughout the life cycle of a design

Interpreting Metrics

Whatever gets measured gets done.

Because managers rely heavily on metrics to measure performance (and ultimately assign reward and blame), there is a tendency for any organization to align its behavior with the metrics. That is why you must be extremely careful to select metrics that faithfully represent the situation and are correlated with the effect you are trying to measure or improve. If you measure the number of bugs found and fixed, you quickly see an increase in the number of bugs found and fixed. But do you see an increase in the quality of the code being verified? Were bugs simply not previously reported? Are designers more sloppy when writing their code since they'll be rewarded only when and if a bug is found and fixed?

Make sure metrics are correlated with the effect you want to measure.

Figure 2-22 shows a list of file names and current version numbers maintained by two different designers. Which designer is more productive? Do the large version numbers from the designer on the left indicate someone who writes code with many bugs that had to be

fixed? Or, are they from a cautious designer who checkpoints changes often?

alu_e.vhd	1.15	cpuif_e.vhd	1.2
alu_rtl.vhd	1.234	cpuif_rtl.vhd	1.4
decoder_e.vhd	1.12	regfile_e.vhd	1.1
decoder_rtl.vhf	1.155	regfile_rtl.vhf	1.7
dpath_e.vhd	1.7	addr_dec_e.vhd	1.3
dpath_rtl.vhd	1.176	addr_dec_rtl.vhd	1.6

Figure 2-22. Using version numbers as a metric

On the other hand, Figure 2-23 shows a plot of the code change rate for each designer. What is your assessment of the code quality from designer on the left? It seems to me that the designer on the right is not making proper use the revision control system.

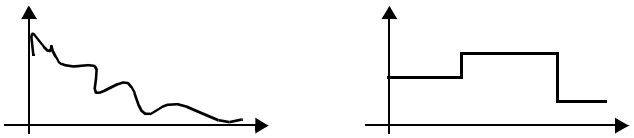


Figure 2-23. Using code change rate as a metric

SUMMARY

Despite their reporting many false errors, linting and other static code checking tools are still the most efficient mechanism for finding certain classes to problems.

Simulators are only as good as the model they are simulating. Simulators offer many performance enhancing options and the possibility to co-simulate with other languages or simulators.

Assertion-based verification is a powerful addition to any verification methodology. This approach allows the quick identification of problems, where and when they occur.

Hardware verification languages offer an increase in productivity because of their specialization to the verification task and their support for coverage-driven random-based verification.

Use code and functional coverage metrics to provide a quantitative assessment of your progress. Do not focus on reaching 100% at all cost, nor should you consider the job done when you've reached your coverage goals.

Use a source control system and an issue tracking system to manage your code and bug reports.