Truss: A Standard Verification Framework

CHAPTER 6

Truss, and verify.
Anon.

Have you ever watched a building being constructed? Early in the project, when the frame of the building is just a skeleton, it's not clear what the finished building will look like. However, as construction continues, from the windows down to the cubicles that are our workplaces, the intent of the framework becomes clear. In fact, a large part of the building's presence depends on the fundamental structure.

This same basic process occurs when we build a verification system. Early in the project, the application framework is built. The result of years of best practices from both the verification and software fields, Truss is an application framework for verification. It is an *implementation*, and therefore makes some decisions about how things should be structured. With verification as with construction, the framework sets the tone for the system.

Truss is a layered architecture, so you can choose how to implement the layers. Although it makes very minimal assumptions, Truss does provide some base classes and conventions as a guide.

Overview

This chapter presents three main topics:

- The roles and responsibilities of the various major Truss components
- How these components work together
- How you adapt this framework for your verification system

This chapter builds on the two previous chapters of the handbook. It implements an open-source verification infrastructure based on the discussion in the Layered Approach chapter. It also uses the Teal library described in the last chapter as a connection between C++ and the simulation.

Teal provides the fundamental elements of a verification system and supports a wide array of methodologies. Truss, on the other hand, provides the infrastructure layers above Teal, adding a set of classes, templates, idioms, and conventions to facilitate the construction of an adaptable verification system.

One of the tricks in building a reasonable system is to find the *key algorithm*. The rest of the algorithms can usually fit around that key algorithm. For example, in a video editing program the key algorithm is all about getting the pace of the edits right. When you watch a movie, that happy, sad, or scared feeling you get comes from how well-timed and precise the changes in scene are.[1] The authors, having developed software for video editing systems, know that in this domain the key algorithm is implemented by adjusting the edit points of a few seconds of video while the video is constantly looping around those edits. This is not a trivial thing to do, because multiple streams of video and audio,

1. Okay, emotions also come from the music, but everything works together.

possibly with software algorithms to implement effects, are changing as the user is adjusting the edit points.

In the verification domain, the key algorithm is the sequencing of the various components of the system. The authors refer to this as "the dance," as there are usually a few interacting components involved. As we talked about in the Layered Approach chapter, the top-level dance takes place between the test, the testbench, and the watchdog timer. Truss implements this dance in the `verification_top()` function—but Truss does not stop there. The authors believe that this dance is the key algorithm in several layers of the system, so we created a `verification_component` abstract base class. Also, we created `test_component` and `irritator` base classes to be the "`top`" at the interface and feature layers of the system. Recognizing and reusing the dance is a significant part of Truss.

This chapter explains the major components of Truss, providing code examples where appropriate. Subsequent chapters provide more-detailed examples.

General Considerations

The authors have worked on several different implementations of verification systems before Truss was available. While at a high level verification systems can be described uniformly, the language used to build them has a lot to do with how a specific framework is constructed.

Using a language other than C++

It is possible to build an OOP-based verification framework in languages other than C++, but no other verification language on the market has the OOP capabilities of C++. For example, when a language that does not support operator overloading is used, the generic `operator==()` or copy constructor cannot be used. To provide this basic required functionality, a common generic base must then be used. Unfortunately, this warps the framework and produces a fragile architecture—mostly because of the unsafe type casting. As another example, with a language that has a

compilation library (such as current HDLs), there is usually a failure to make a distinction between interface and implementation. This leads to a more-complicated framework, as test writers must separate the interface from the implementation manually and repeatedly. C++ avoids these problems.

Keeping it simple

A stated goal with both Teal and Truss is to avoid unnecessarily complicated code. C++ has many powerful features, but many times they are not appropriate. It is easy to get distracted with C++ techniques and forget that the real goal is keeping the whole team productive.

For example, implementing a generic interface for a verification component, such as a transactor, as a template can be tricky. Sometimes using the template can be more complicated than simply replicating code.

Sometimes only a convention should be used. An example of this is the *generator* concept. One could define an abstract base class, yet the common methods come down to just `start()`, `stop()`, `report()`, and a few others. It turns out that this concept of `start()`, `stop()`, and so on is common to a large set of verification tasks, and is represented in Truss as the abstract base class `verification_component`. However, the concrete subclasses are inherited from `verification_component` only if they use the bulk of the methods. Any smaller subset uses the same named methods as a convention instead.

> *In this way, the framework is not warped to fit a generic class. Even more important, your design is not warped to fit the generic class.*

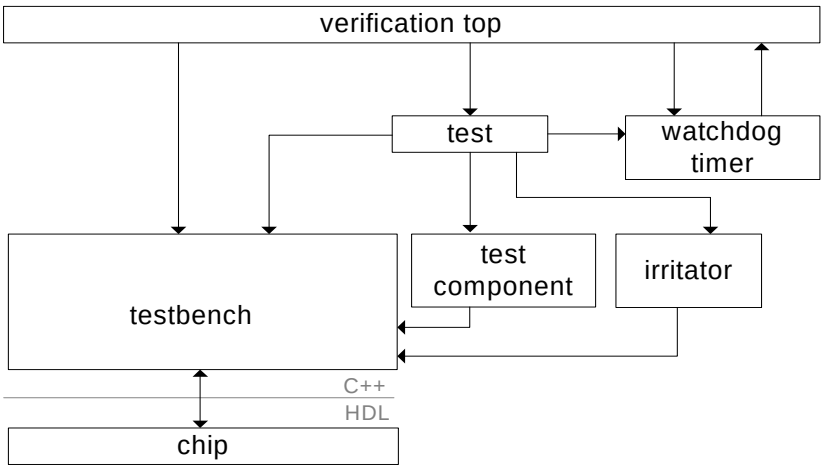
Truss implements a specific methodology for functional verification. As in any endeavor to generalize, the terrain is fraught with peril. Nevertheless, as writing code entails making judgments about what is the "right" decision, Truss attempts to generalize a style of verification. Deciding on the right balance between generic and specific is a judgment call for the team. The idea behind Truss is to foster a small, usable, and adaptable methodology for beginners through experts. As such, Truss

provides an example of the techniques presented in Part III of this handbook.

Major Classes and Their Roles

Truss is an implementation of the layers talked about in the Layered Approach chapter. Consequently, there are only a few top-level components—the verification top, the testbench, the test, and the watchdog timer. Each component has a specific role. These components and their roles have been architected to allow a large amount of flexibility with a relatively simple interface. These top-level components (and those the next level down) are shown below:

Verification Component Hierachy



The top-most C++ component is the `verification_top()` function, whose role is to create and sequence the other components through a standard test algorithm. (The algorithm is explained in detail in the next section.) In addition, `verification_top()` initializes all global services, such as logging, randomization, and the dictionary.

The watchdog timer is a component created by `verification_top()`. This component's role is to shut down a simulation after a certain amount of time has elapsed, to make sure the simulation does not run forever.

The testbench top-level component is the bridge between the C++ verification world and the HDL chip world. As such, the testbench's role is to isolate the tests (and test writers!) from having to know how C++ transactors, traffic generators, monitors, and so on interact with the chip. Whether a bus functional model (BFM) writes to registers or forces wires should not be of concern to the test writer.

In addition, the testbench holds the configuration objects of the chip. This is needed by the BFMs, transactors, and similar agents to be able to configure the chip correctly. There is probably a configuration object for each interface of the chip. For chips that contain internal functions, such as dynamic memory allocation (DMA), there may be a configuration object for each function.

The last, but certainly not the least, top-level component is the test itself, whose role is to execute a specific functionality of the chip. It does this by using the testbench-created BFMs, monitors, and generators. The test is responsible for choosing among the testbench's many configurations and capabilities and exercising some subset of the chip's functionality. In general, the test contains very little code. This is because any code it contains may need to be used in other tests as well. To support code that is more adaptable, a test normally consists of several test components, as will be discussed later. The exception is for directed tests, in which case registers may be overwritten, specific traffic patters sent, or specific corner cases exercised directly in the test component.

Key test algorithm: The "dance"

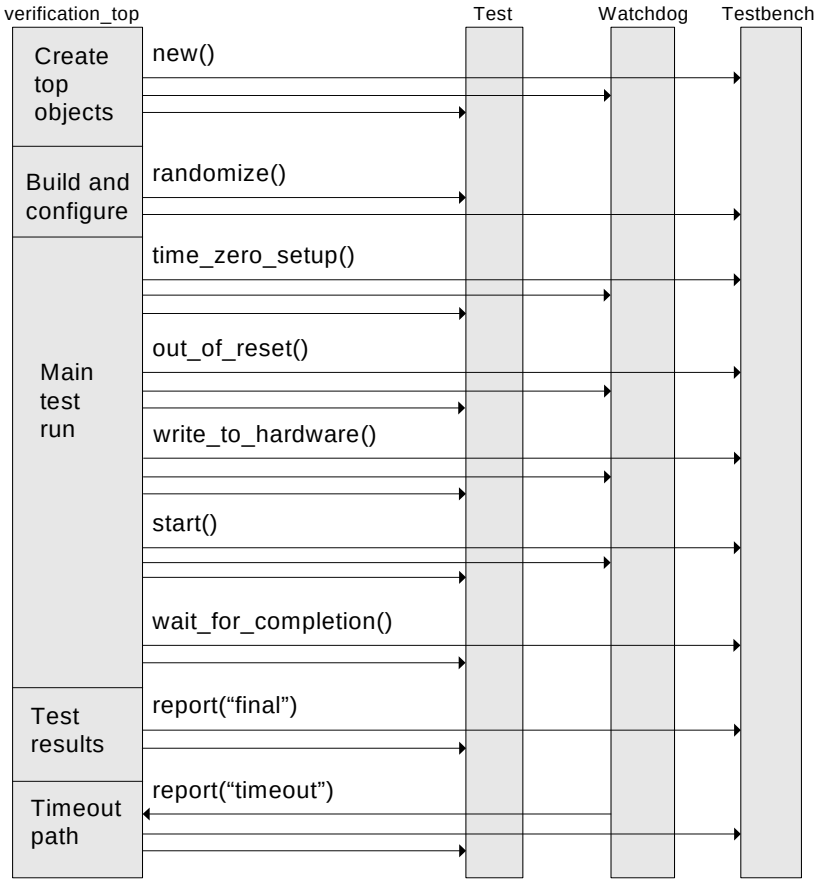
The top-level components of the previous section have a complex, yet necessary, set of interactions. This ensures the maximum flexibility for a test, while providing a known set of interactions. This is one of the tricky parts of a verification system. This section discusses this standard algorithm, which we call the "dance."

In general, the top-level components are created, randomized, and then started. Then `verification_top()` waits for the test and testbench to be completed. This is called the "polite" path. If the watchdog timer

decides that a timeout has occurred, the “impolite” path is taken and the simulation ends.

The order of these calls can be better visualized on an event diagram, as shown below. The four columns show the main components. Execution starts at the top left line, and the arrows represent function calls to the other components.

The Dance



The first thing that `verification_top()` does is build the global logging objects. These provide logging to a file and shut down the simulation after a threshold number of errors have been logged. (See `truss_vout.h`.)

Then, after the global logging objects have been created, `verification_top()` allocates the top-level objects. The test is given a pointer to the testbench, so that it can interact with the testbench. It is also given a pointer to the watchdog timer, in case a part of the test wants to force a shutdown or override the `verification_top()` default timeouts. The watchdog is given a pointer to an object so that it can call the final report method with a "watchdog timeout" string prefix.

At this point, all the top-level objects are constructed. As part of their construction they are expected to have established default constraints.

Then `verification_top()` reads the dictionary file (if it exists). This is to allow the test constraints file to override any default settings put there during the construction of the test, the testbench, and their subordinate components.

After initializing the random-number generator, `verification_top()` calls `test->randomize()`. Once the test is randomized, then `testbench->randomize()` is called.

At this point, it is expected that the test and testbench have built their respective subcomponents and are ready to run the test. The first step is the `time_zero_setup()` method, which is used to force wires and initialize interfaces prior to bringing the chip out of reset.

As expected, the next step is `out_of_reset()`, which is used to bring the chip out of its reset state and set it for initialization through the backdoor or register writes.

The next step, `write_to_hardware()`, is where the BFMs are called to initialize the chip. This can be done by either the test, the testbench, or a combination of the two. What is appropriate depends on your situation, as discussed in subsequent sections.

At this point the system is ready for traffic flow. The `start()` method directs the testbench and test to start running. The testbench is started first, to allow monitors and BFMs to start, followed by the watchdog timer. Finally, the test is told to `start()`, which generates the actual traffic.

Next, `verification_top()` calls `wait_for_completion()` on the *testbench*. If your design makes the testbench aware of what checkers are in use, this call waits for the testbench checkers to complete. If not, this method simply returns.

Then `verification_top()` calls the *test*'s `wait_for_completion()`. If your design makes the test aware of what checkers are in use, this call waits for them to complete. (This is the style used in the examples.)

At this point, the test is almost finished. The testbench and test are called to report their final status.[2]

Then `verification_top()` checks to see if any errors were reported. If none were reported, the test is considered to have passed. It may seem weak to accept that the absence of errors is sufficient to consider a test passing. In practice, however, there is no other choice. At the top level, one must trust that the lower-level objects do their jobs. Note that this usually means that in-flight data must be weeded out as the checker proceeds.

Now if the watchdog timer triggers, a different path is taken. The watchdog immediately calls the report method on `verification_top()`. Note that the watchdog itself uses an HDL-based timeout, so that if the report method hangs, the simulation still ends.

The verification_component Abstract Base Class

While the test and the testbench are completely different classes as far as their roles and responsibilities are concerned, their interface to `verification_top()` is the same. For this reason a common class was created. This common class, used as a base for both the test and testbench, is called the `verification_component`.

The `verification_component` is an abstract base class. As such, it provides pure virtual methods for the dance described in the previous section. In addition, `verification_component` provides a constant

2. The authors have tried using the destructor as the final report mechanism. In practice, however, this becomes a difficult part of the design. This is because some destructors try to access deallocated memory or other objects that have already been destroyed. It then becomes tricky to "shut down" the simulation in the correct order, so as not to cause a crash or hang and still get errors printed out. This is one area where verification is different from software, which generally does use destructors as part of the system design.

name and a logger. The interface for `verification_component` is shown below:

```
namespace truss {
  class verification_component {
    public:
      verification_component(const std::string& n);
      virtual ~verification_component();
      virtual void randomize() = 0;
      virtual void time_zero_setup() = 0;
      typedef enum {cold, warm} reset;
      virtual void out_of_reset(reset) = 0;
      virtual void write_to_hardware() = 0;
      virtual void start() = 0;
      virtual void stop() = 0;
      virtual void wait_for_completion() = 0;
      virtual void report(const std::string prefix) = 0;
      const std::string& name;
    protected:
      mutable teal::vout log_;
  };
};
```

Although `verification_component` is a base for the test and the testbench, it is also useful as a base for other objects.

Detailed Responsibilities of the Major Components

The previous sections discussed the roles of the major components and how they were sequenced to run a test scenario. This section dives down a level, discussing in more detail the specifications of the major components. (Because `verification_top()` was discussed in detail in the previous section, it is not discussed further here.)

The testbench class

The `testbench` class has two main responsibilities. One is to isolate the test writers from the actual wire interfaces. The other is to provide "one-stop shopping" for all the generators, checkers, monitors, configuration objects, and BFMs/drivers in the system. The reason to put all of your components into a single object is to facilitate the adaptation of components into multiple tests. In this way, a test writer can see all of the possible "building blocks" that are available.

The `testbench` class can be a passive collection point for all these components, or it can play an active role in bringing the chip out of reset, generating traffic, and knowing when the test is done. In theory, only the global functionality should be handled by the testbench. For example, the testbench probably should bring the entire chip out of reset, while the test can bring separate functionality out of reset. In practice, the test and the testbench share the work.

In general, it is better to let the test or test components control the simulation. This is because a test or test component can then be adapted for several different types of tests.

A more active testbench may, as a counterpoint, simplify a large number of tests in a way that a test base class cannot, because the testbench has direct access to all the chip's wires.

Understand that the more test knowledge a testbench has, the more all tests must act the same or have control over that testbench's functions. This can be good or bad. The specific responsibilities for control and functionality—test or testbench—are, of course, up to the verification team.

As an implementation detail, Truss provides only a `testbench_base` class. What `verification_top()` builds, however, is a `testbench` object. You must provide a `testbench.h`, which declares a `testbench` class. You will probably also have a `testbench.cpp`, which is inherited from `truss::testbench_base`.

Watchdog timer

The watchdog timer component is responsible for providing an "impolite" shutdown if the test has executed for too long. The timer has two timeout mechanisms: one triggers when the watchdog HDL timer triggers, and the other triggers after the first trigger has occurred.[3]

The watchdog timer uses the dictionary to get its timeout values, which are sent to the HDL on `time_zero_setup()`. The `start()` method starts the timers. The HDL watchdog uses an internal timer. If it were to use a passed-in clock, that clock may inadvertently be shut off.

Once either timer triggers, the watchdog HDL timer is notified and a second timer is started. If this timer expires, `$finish` is called. This might happen, for example, if there is some code in the report that is still reading registers, but the chip is unable to respond.[4]

After the watchdog is notified of an HDL timeout, the `report()` method in `verification_top()` is called. This allows the test to report which checkers have completed and which have not, helping to provide a clue as to why the simulation ran too long.

Test class

The test class is responsible for selecting, configuring, and running all the appropriate generators, BFMs, monitors, and checkers. It is also responsible for selecting the configuration of the chip to be used.

While you could directly implement the above responsibilities in the test class, Truss encourages another style. In Truss the test is intended to consist of a number of independent, smaller components called *test components*. These components are the ones that actually do the work; the test's role is to create, constrain, configure, and sequence the com-

---

3. The watchdog timer is simple in theory, but often hard to execute correctly. To be sure, it must have a clock and a countdown time, but even this basic level can be problematic. Should you use wall clock time, simulation time, or both? Should the HDL timer be internal or external? What resolution should it have? Should the test be able to extend or communicate the expected time of the run?
4. The authors worked on a project where the final report code read the status registers to make sure that functional area of the chip did not have any errors. However, when we added a power-down test irritator, the read hung the system. It took us a while to find the offending code.

ponents, as appropriate for the test at hand. The reasoning behind having multiple independent components is that this is close to the real operation of the chip, where each feature is expected to operate simultaneously. In reality, the chip has common resources that must sequence or arbitrate the use of features. It is in these common resources where the more tricky bugs lurk.

Using this method, the test's direct responsibility is to map the features of the chip (as presented by the testbench's data members) to a set of classes inherited from the `test_component` base class. The test would then add constraints to adapt the test component to the test at hand, as in the following example:

```
class ethernet_basic_packet : public test_base {
  public:
  ethernet_basic_packet(testbench* tb, watchdog* wd) :
    ethernet_data_1(tb->e_generator_1, tb->e_bfm_1,
                    tb->e_checker_1),
    ethernet_data_2(tb->e_generator_2, tb->e_bfm_2,
                    tb->e_checker_2),
    pci_express_1(tb->pci_generator_1, tb->pci_bfm_1,
                    tb->pci_checker_1) {}
  void time_zero_setup(){
    ethernet_data_1.time_zero_setup();
    ethernet_data_2.time_zero_setup();
    pci_express_1.time_zero_setup();
  }
  void out_of_reset(reset r) {
    ethernet_data_1.out_of_reset(r);
    ethernet_data_2.out_of_reset(r);
    pci_express_1.out_of_reset(r);
  }
  void write_to_hardware() {
    ethernet_data_1.write_to_hardware();
    ethernet_data_2.write_to_hardware();
    pci_express_1.write_to_hardware();
  }
   void start(){
    ethernet_data_1.start();
    ethernet_data_2.start();
    pci_express_1.start();
  }
```

```
    void wait_for_completion() {
     ethernet_data_1.wait_for_completion();
     ethernet_data_2.wait_for_completion();
     pci_express_1.wait_for_completion();
    }
    void report(const std::string& prefix) {
     ethernet_data_1.report(prefix);
     ethernet_data_2.report(prefix);
     pci_express_1.report(prefix);
    }
  private:
    ethernet_test_component ethernet_data_1;
    ethernet_test_component ethernet_data_2;
    pci_irritator pci_express_1;
  }
```

In the above example, the `ethernet_basic_packet` test uses three test components, two of which are identical. It connects up the appropriate testbench objects and forwards to every test component the following test calls:

`time_zero_setup()`, `out_of_reset()`, `start()`, `wait_for_completion()`, and `report()`

So why do testing in this more complicated manner? In addition to the previously mentioned idea of simulating close to real-world conditions, an important reason is to maximize the adaptability of the test components. In the example above, we used the same test component for both Ethernet ports. Also, when the test components take in only the parts of the testbench that they need, they (1) make explicit what they are using, and (2) minimize the assumptions on the rest of the chip. This, as will be highlighted in the single UART example in Part IV, allows a test component to be reused for other chips that have only a subset of the original chip's functionality.

Test components are critical to the adaptability of a verification system. In general, the test components themselves do not know whether they are running in parallel with other test components or are part of a series. Thus, the most adaptable components are these test components, as will be discussed further in the following sections.

As an implementation trick, `verification_top()` builds a test by using a `define` called `TEST`. This trickery, set up by the makefile, allows the

`truss` run script to compile in a different test, while leaving the rest of the build image the same for all tests. This allows each test to be its own class (inherited from `test_base`). This cleverness helps one avoid a bad experience in the future. Assume that your team had written on the order of 50 tests, and then a new test was created that required a new subphase to be added to the dance. Although the other tests did not need this new method, you cannot add the default method. This is because all the tests are implemented as a test class. There is only one header `test.h`, and 50 different `test.cpp` files. By defining a base class, and then having the actual test be an inherited class (with a different header file), one can add methods to the base without affecting the existing tests.

There is one more part to a test that needs to be discussed. Often a test is made better by the addition of random background traffic. This traffic, be it register reads and writes, memory accesses, or just the use of other interfaces, can uncover corner cases, such as bus contention, that would not be found otherwise.

These background-traffic test components are called *irritators* and inherit from the `test_component` class. They differ from the standard test component in that they continue their traffic generation until told to stop by the test. Test components, by contrast, decide themselves when they are done, as determined by specified metrics, such as a stop time or the number of packets to send. (Irritators will be describe in more detail later in this chapter.)

With background traffic irritators, the test is written essentially as before. The exception is that the `wait_for_completion()` of the test calls the primary test components' `wait_for_completion()`. When the primary component returns, the test calls `stop_generation()` on all the irritators and waits for them by means of their `wait_for_completion()`. Then the test returns control to `user_main`. (This is explained further in subsequent sections and in the examples in the chapters that follow.)

Test Component and Irritator Classes

As discussed in the previous section, test component-based design is central to a Truss-based test system. The authors have found that separating the test scenarios into test components has maximized the adaptability of the system. By using test components and irritators, test writers have been able to minimize their assumptions and distractions and concentrate on exercising the chip. Furthermore, other test writers can adapt what was done in other functional areas and inherit irritators (if they are not already present) for use as background traffic.

This section describes the responsibilities and interfaces of the `test_component` and `irritator` abstract base classes.

The test component abstract base class

The `test_component` is an abstract base class whose role is to exercise some interface of the chip. As discussed above, this functionality has traditionally been included in the test. The `test_component` describes the interface that all concrete implementations must follow.

In fact, you may have several types of `test_component` for a single interface, for example, a register read/write one, a basic data path one, and an error case one. The fact that these different exercises implement the same interface simplifies reasoning about them.

In practice, most test components use a generator and a wire-level object. Sometimes they may also be given a checker, depending on the designer's intent.

The `test_component` class is not directly a `verification_component`, but it has all the same phases.[5] The `test_component` breaks down some

[5] The primary reason for this is because `verification_component` represents a pattern, while `test_component` is an example of this pattern. The `test_component` has specific implementations of four of the `verification_component` methods. Also, `test_component` introduces some of these same methods as nonvirtual. Finally, the sequencing of the methods is different from the test and testbench, the two top-level components that are verification components. These differences are critical for the integrity of the class.

of the verification_component methods into finer detail, as one would expect of a lower-level object.

Below is the interface for the test_component base class.

```
namespace truss {
  class test_component :
      protected virtual verification_component,
      protected thread {
    public:
      test_component(const std::string& n);
      virtual void time_zero_setup() = 0;
      virtual void out_of_reset(reset) = 0;
      virtual void randomize() = 0;
      virtual void write_to_hardware() = 0;
      void start();
      void stop();
      void wait_for_completion();
      void report(const std::string& prefix);
    protected:
      virtual void start_();
      virtual void run_component_traffic_();
      virtual void start_components_() = 0;
      virtual void generate() = 0;
      virtual void wait_for_completion_() = 0;
      bool completed_;
  };
}
```

The methods time_zero_setup(), out_of_reset(), and write_to_hardware() are provided to allow the test component to interact with a BFM or driver. Note that a different, but equally valid, architecture would keep the wire-layer components private in the testbench and sequence them by means of the top-level dance. This assumes that the testbench knows what subset of the BFMs, drivers, and monitors, to start up.

The start() method is used to start the test_component's generator, BFM, and so on. This method is implemented by a Truss utility class called thread. A thread class runs another virtual method, start_(), in a separate thread or execution. This allows a test class to do the obvious thing and just call start() on all the test components the test uses.

Let's look at the `start_()` method, as it is the main starting point for an interface of the chip. The `start_()` method runs two methods: a `start_components()` pure virtual method, and a virtual `run_component_traffic_()` with a default implementation. The idea behind the `start_components_()` method is that you call `start()` on your generators, BFMs, and so on, as appropriate. (The examples part of this handbook contains examples of `test_component`.)

The default `run_component_traffic_()` method calls `randomize()` (to randomize the test component and its components), and then calls `generate()`. In your `randomize()` method, randomize the data members that will be used by `generate()` to cause some traffic to be generated. In your `generate()`, take these data members and make the appropriate calls to the generators in the testbench.

An AHB example

An example might make the roles a little clearer. (Remember that there are several fully implemented examples in Part IV.) Suppose you are creating a test component to test an AHB[6] arbiter. The test component acts as a master, generating read and write requests to a number of slaves.

The generator in the testbench can generate a burst of reads or writes to a given slave, using a specific burst length. Assume that the generator has a channel interface that can take in an AHB transaction object. The randomize function of your `ahb_test_component` might look like this:

```
void ahb_test_component::randomize() {
  burst_length_ = generate_burst_length(min,max);
  is_read_ = generate_type(min_type, max_type);
  slave_ = generate_slave(min_slave, max_slave);
}
```

The corresponding `generate()` might look like this:

```
void ahm_test_component::generate() {
  //addresses are picked by the generator
  generator_—>queue_burst
    (new AHB_transaction (burst_length, is_read_,
```

6. AMBA (Advanced Microcontroller Bus Architecture) high-performance bus.

```
      slave_));
  done_.signal();  //Signals that test_component is done
}
```

Notice that by nature these calls are executed in a one-shot manner. That is, together they perform a single transaction. This is useful to allow an `irritator` to inherit from this test component later, to sequence this pattern any number of times and possibly change the randomization constraints as well.

So why have two separate methods?

By separating the randomization from the generation phases, one can inherit different classes that either (1) have different randomization characteristics (for example, logarithmic distributions of the burst length, or a pattern); or (2) send the data through a filter first, then to the generator.

So now that the transaction has been generated, what should the `wait_for_completion()` method do? Because the generation is occurring in another thread, there should be a condition variable to communicate when it is done.

So the code might look like this:

```
void AHB_test_component::wait_for_completion_() {
  done_.wait();
}
```

Test-component housekeeping functionality

The `test_component` class also provides a basic housekeeping boolean that tracks when you return from the `wait_for_completion_()` method. This allows the `report()` method to determine whether you have considered the work of the component to have been completed or not. This can be very useful in a timeout situation, to see which components have not completed.

What you decide to do in the `wait_for_completion_()` depends on how you view your `test_component`. One view is that it is a traffic generator only, which can complete when the generation of traffic has been queued. It is then up to the testbench or test to determine when the chip has processed all the data. This will most likely involve a checker or monitor.

Another view is that your `test_component` represents a generate and check path through the chip. In this case, the completion of `test_component` signifies the completion of the entire exercise. (The examples in this handbook use this view.)

> *As always, the team must decide which view is better for their project.*

The irritator abstract base class

As discussed above, the `test_component` is set up as a one-shot traffic generator. This works for tests that are directed, and for tests where the completion event is predetermined—that is, tests that know before the `start()` call what the end conditions are.

However, sometimes it is not good design to have the `test_component` determine when completion is achieved. This is the case when, for example, you want to achieve a certain metric, and the measurement is not appropriate information for the `test_component`.

For example, you may want to send 100 bursts of some AHB traffic. While this could be included in the `ahb_test_component`, you might not want to measure completion by 100 bursts all the time. Instead, you might want to write a test that looks at the number of hits each slave device gets, and stop the test when all slave devices have been targeted. As another alternative, you might want a test to run until some coverage occurs, which could be any of the previous scenarios, or could involve some internal state in the arbiter.

The `irritator`, inherited from `test_component`, is used for situations such as these. The interface is shown below.

```
namespace truss {
  class irritator : public virtual test_component {
   public:
      irritator(const std::string& n);
      virtual ~irritator() {}
      void stop_generation() {generate_ = false;}
   protected:
     virtual void start_();
     virtual void run_traffic_();
```

```
    virtual bool continue_generation();
    virtual void inter_generate_gap() = 0;
    bool generate_;
  };
};
```

The irritator overrides the `run_traffic_()` method of the `test_component` base class. It sets up a loop, calling the one-shot randomization and generation in the `test_component`'s `run_traffic_()` methods. The implementation is shown below.

```
virtual void truss::irritator::run_traffic_() {
  while (continue_generation()) {
    test_component::run_component_traffic_();
    intergenerate_gap();
  }
}
```

The method `continue_generation()` just looks at a boolean, which is toggled to `false` by a call to the `stop_generation()` method. This allows an external class to stop the continual loop of randomization and generation.

Note that there is a new virtual method in the `irritator` class, called `intergenerate_gap()`. Because the irritator is continually generating traffic, you might need a delay mechanism to prevent the generator from flooding the chip.

There are many ways to get this delay. For example, in one solution the generator and attached BFM/driver could execute the generate request as soon as it is called and thus take simulation time. In another solution, the way to get a delay would be to have a fixed-depth generator and BFM/driver channel.[7] This would put back-pressure on this generate loop. In still another solution, the generator could have a delay in clock cycles before returning.

Any of the above solutions is acceptable, but there is yet another choice. That option is to have the irritator itself provide the delay mechanism.

[7] This method is supported in Truss's `channel` class.

The `intergenerate_gap()` is a virtual method allowing you to implement an irritator-based delay. This allows the irritator to decide on the throttle mechanism. Different subclasses could implement different policies. For example, an irritator could wait for a variable number of clock cycles. Another example would be to measure some parameter on the checker (such as packets in flight).

As always, the team must decide what is appropriate.

Using the irritator

The irritator continues this generate/wait loop until a `stop_generation()` is called. But how do you decide when to stop the irritator? The answer, of course, is "When the test reaches its goal." One goal could be that the "main reason" for the test has been achieved. For example, you can have the main goal be a test component, perhaps one that generates a fixed, but randomized, number of packets through a particular chip interface. The global goal in this case would be for the test component to achieve completion. Here is how the test code might look:

```
void noisy_packet_test::wait_for_completion() {
  //assume the data members include
     base_packet_exerciser,
  //the test component of interest and some std
     container
  //class with a list of irritators.
  basic_packet_exerciser_->wait_for_completion();
  std::for_each(irritators_.begin(), irritators_.end(),
               stop_generation());
  std::for_each(irritators_.begin(), irritators_.end(),
               wait_for_competion());
}
```

Ignoring the nontrivial constraining, selecting, and creating of the test component and irritators, what is accomplished in a few lines of code is a shutdown sequence that is powerful, while being a fairly simple idiom.

Note that a verification team could decide to use only irritators in their implementation. In that way, when to stop the test can then be determined by looking either at a checker or possibly at elapsed simulation time.

The complex part of the test would then become the randomization and selection of irritators. The authors have worked on a variant of this methodology, and the resulting verified chip was a first silicon success.

Summary

This chapter introduced Truss, an open-source application framework.

We revisited the benefits of an OOP language such as C++, but stressed the need to keep things simple despite the power of this language, to avoid writing code that is unnecessarily complicated.

We talked about the key algorithm of verification, which the authors called the "dance." We showed how the dance is used by the `verification_top()` program to run a test. We discussed the roles and responsibilities of the test, testbench, and watchdog timer, the main parts of the top-level dance.

We discussed the `verification_component` abstract base class, which provides pure virtual methods for the dance.

We then discussed the `test_component` and `irritator` classes, including their responsibilities and interfaces.