A New Flexible VHDL Simulator

Arlet Ottens Wilco van Hoogstraeten Henk Corporaal

Department of Electrical Engineering
Delft University of Technology
Mekelweg 4, P.O. Box 5031
2600 GA Delft, The Netherlands

Abstract

This paper describes a new VHDL simulator. Its object-oriented design makes it very flexible, both towards VHDL and the simulation algorithm and hardware. It is designed according to a strict set of design goals that were prompted partly by its future use in performance evaluation of distributed VHDL simulation.

The simulator is divided in three parts with well defined interfaces. This makes the above-mentioned flexibility possible, and allows the simulator to be worked on concurrently for most of the implementation time.

At the end of the implementation phase it has become clear that is possible to have a flexible, object-oriented simulator for full VHDL that can be used with the most advanced distributed simulation algorithms.

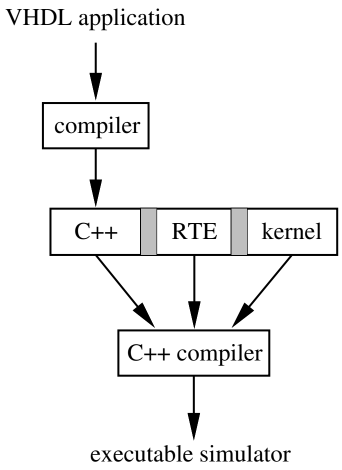

Figure 1: The VHDL simulator.

1 Introduction

This paper discusses a new implementation of a VHDL simulator developed at the University of Delft. The simulator is developed in the framework of the ESPRIT CHESS project which researches the feasibility of parallel VHDL simulation.

The simulator consists of three parts, as depicted in Figure 1. The first is a compiler that translates the VHDL application into our target language, C++. This code uses the second part, the *runtime environment* (RTE), extensively. The third part is the simulation kernel that contains the simulator core algorithm and simulation functionality.

Flexibility of the simulator is one of the most important parts of the requirements. It should be very simple to modify and extend the simulator (e.g. for the migration from VHDL'87 to VHDL'92), and to rewrite it for parallel simulation. These modifications should not be visible to the compiled VHDL application so that the compiler can be used both for sequential and parallel simulation.

The simulator is intended to be used for the performance evaluation of parallel VHDL simulation with distributed simulation algorithms. This is the main reason for the definition of a strict set of design goals for the simulator before the design was started.

Section 2 explains our design goals for the simulator. Sections 3 and 4 describe the internals of the simulator. Section 5 describes the VHDL to C++ compiler. Section 6 shows a small performance measurement with the simulator. Section 7 has some observations on parallelising the simulator. Section 8 concludes this paper.

2 Design goals of the simulator

From the beginning of the development of the simulator it was clear that we needed a *compiled-code* simulator, i.e. VHDL is compiled into another language which is compiled to give an executable simulator. This choice was made with respect to the further use of the simulator for performance evaluation of distributed VHDL simulation.

The needs of the project and our own insight prompted the following design goals:

- Full VHDL,
- Simple compilation,
- Flexibility for modifications and extensions,
- Concurrent development.

These goals have shaped the current simulator to a very great deal. The remainder of this section is dedicated to the discussion of these goals.

2.1 Full VHDL

This may not seem such an adventurous design goal, but it was a great step to take. The simulator was intended to be used for distributed simulation with *optimistic* algorithms [3], and supporting full VHDL poses enormous problems in this area. This type of algorithms keeps previous states of the simulation for purposes of synchronisation. The high-level constructs of VHDL, such as file I/O, recursion, and dynamically allocated memory, complicate the saving and restoring of this state to a large extent. Most of the former research effort in this area has focussed on a "manageable" subset of VHDL, for instance by Chawla et al. which has used a subset that made comparisons with logic simulation possible [1].

The support of full VHDL prompted the use of multi-threading for the switching between VHDL processes. This gives an elegant solution for waveform assignments in (recursive) procedures. The overhead that the context switch of the multi-threading involves, does seem relatively minor compared to a resumption point scheme, where only the next entry point of the process is saved; upon resumption a jump is made into the process to the correct resumption point. However, the overhead for small stateless processes will be a penalty.

2.2 Simple compilation

One of the main objectives in the project was to have a simple compiler. Resources are low on the project and there is not much time, and the compiler was identified to require a lot of effort and time. This is why an elaborate run time environment was created which raises the semantic level of the target language of the compiler, C++, almost to VHDL.

Another advantage of a simple compiler is shown by Hueber et al. [2]. Their observations are that the main problem of compiled-code VHDL simulators is located in the compilation part. The compiler often takes tremendous amounts of time and memory, and sometimes is not able to compile at all. We did not want to make this mistake, and with the simple compiler that we have now, it is very easy to observe what the influences are of optimisations on the complexity of the compiler.

As an example, let us look at a small VHDL example and the output of the compiler. The following architecture body:

```
architecture simple of Nor_Gate is
begin
   process
   begin
      y <= a nor b after 10 fs;
      wait on a, b;
   end process;
end simple;
```

can be translated to:

```
int Nor_Gate::main( )
{
   while ( TRUE )
   {
      y.waveform_assign( a.nor(b), 10 );
      wait_on( a, b );
   }
}
```

The definition of signals and initialisation of signals and processes is done elsewhere, but is not in any way more difficult than the above.

The compiler generates function calls and class instantiations that are defined in the runtime environment (RTE). The RTE acts as a uniform interface to the compiler as well as to the simulation kernel.

2.3 Flexibility

Flexibility of the simulator is very important when it is used for research purposes. It must be easy to implement optimisations without having to modify large parts dependent on it. This is why the simulation environment is object-oriented from top to bottom, giving a very flexible structure that can be modified very easily and without much effort.

As an example, let us look at an example of this flexibility. The base types of VHDL are now in an elaborate class structure that involves smart pointers and simple garbage collection (more details in Section 3); this method has certain advantages for code generation. However, it is very easy to compile certain instances in VHDL to much more efficient code involving the native types of the target language. The compiler then selects the appropriate functions and casts automatically. In this manner the simulator can be made more efficient on a step by step basis.

2.4 Concurrent development

With a very short period to finish the simulator and the following use of it in the performance evaluation of distributed simulation algorithms, it was vital that the simulator would be finished as soon as possible, including the VHDL to C++ compiler. This is why the RTE was developed first, resulting in a well-defined interface, both towards the compiled-code as towards the simulation kernel. The implementation of the RTE and the simulation kernel and the compiler could therefore be done concurrently without time-consuming interaction between implementors.

3 Values, Signals and Drivers

This section describes the runtime environment. It describes the interface between the generated C++ code and the simulation kernel. The interface is described in this chapter by showing the various C++ classes constituting it. The runtime environment is designed to minimise the semantic gap between the generated C++ and the original VHDL code. The classes that are shown here are all closely modelled on VHDL constructs, such as values, signals, implicit signals, and drivers. The operations that VHDL allows on these constructs are implemented as member functions of the respective classes.

3.1 The Value class

The Value class is used to implement every possible VHDL type. It is implemented as a pointer to the actual value. This approach has the advantage of being easy to extend, and of simple argument passing. It uses reference counting as a mechanism to implement garbage collection. Every object of class `Val` contains an integer `refs`, that represents the number of `Value` objects that contain a pointer to it. When objects of class `Value` are assigned to a variable the reference count is increased by one. The `Val` object that the variable contained earlier has its reference count decreased by one. As soon as the number of references becomes zero, the object is destroyed.

This garbage collecting scheme has two disadvantages: it is slow, and cannot be used with cyclic data structures. Fortunately, VHDL does not use cyclic data structures, so that is not a problem. The lack of efficiency is something that has to be studied. If is turns out to be a problem, handling of values should be optimised. It is possible to do this by making a special case for the integer data type, which is by far the most common data type used in VHDL. These integer routines could be optimised, and used whenever appropriate. For all other types the general `Value` class can still be used. This should sufficiently reduce the overhead of the `Value` class.

The implementation of the Value class is like this:

```
class Value
{
public:
   Value( );
   Value( const Value &V );
   Value( Val* V );
   virtual ˜Value( );
   Value& Operator=( const Value &r );
   ValType type( ) const;
   boolean is_composite( ) const;
   Value& operator[]( int index ) const;

protected:
   Val* rep;
};
```

Note that this is the structure visible to the user. The `rep` pointer is a reference to the container class that holds the actual value.

3.2 The Signal class

The `Signal` class is implemented like the `Value` class, in a two layer structure using a pointer to the real data. All operations on signals are mapped to operations on the data that the pointer is referring to. The pointer to the data is stored in the member `SignalBase* base`. A practical difference between the implementation of values and signals, is that signals are never destroyed, and therefore do not need any reference counting code. The structure via the pointer is only used to use different types of signals indiscriminately; these types include scalar signals, composite signals and various types of implicit signals.

The scalar signals contain an element of the `Value` class to hold the current value of the signal. Composite signals contain an array of signals.

3.3 The Driver class

Drivers are objects that contain the projected output waveform for a signal, it contains a list of times and associated values. Drivers are connected to a signal, and as time proceeds, the values from the head of the list are used to set the current value of the signal. Drivers are implicit structures that are not declared directly by the user who writes the VHDL description, but are implicitly created whenever a signal is declared. Some signals in VHDL (the resolved signals) can have multiple drivers, all other signals can only have one driver. In the runtime environment this has been implemented differently (see figure 2): all signals have a most one driver. When resolved signals are used in VHDL, they are translated in the following way:

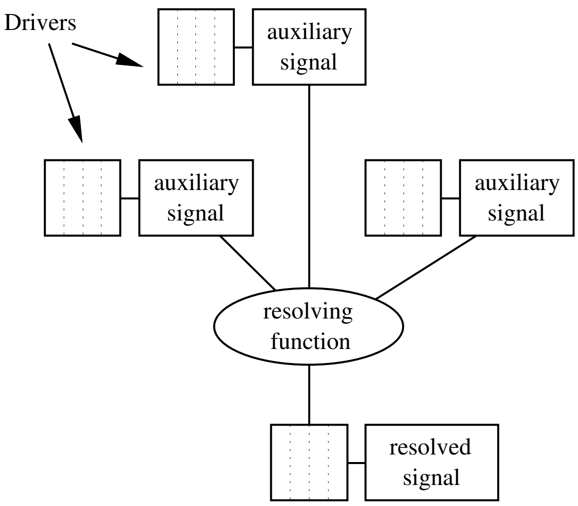

Figure 2: Resolved signals

- The resolved signal itself is implemented by a normal (unresolved signal), this signal is given a signal driver, which is hidden for the user.
- The driver for the resolved signal gets its input from a special process, that implements the resolving function.
- The process that controls the resolved signal is sensitive for events on a number of auxiliary signals, each driven by a driver that was originally created for the resolved signal. As soon as one of the inputs of the process changes value, the resolving function is recalculated and the result is used to update the driver of the resolved signal.

4 Processes

The process class is used to implement VHDL processes. These processes are running independently with separate threads of control. The execution of a process can be suspended by letting the process *wait* for a certain event. The only event that is used in the runtime environment is the change of a signal. It is possible for a process to wait for more than one signal at the same time; in that case the process becomes active again as soon as an event occurs on any of the signals.

The following definition is used:

```
class Process: public LWP
{
 public:
   Process( const char* n = "<process>" );
   void  wait_on( Signal s );
   void  wait_on( Signal a, Signal b );
   void  wait_on( int n, ... );
   void  wait_on( );
   void  kill( int signal );
   const char* name( );
};
```

As can be seen from this definition, the `Process` class inherits from the `LWP` class, a class for light weight processes. The `LWP` class implements all of the multi-threading features. The `Process` class just rephrases all features in term of VHDL processes.

4.1 The LWP library

The LWP library is a special package for creation of light weight processes. This package has been created to have a uniform light weight process package across different platforms. Existing packages were lacking in that respect. Light weight processes are processes that share the same address space, so they can communicate using shared memory. The user is responsible for synchronisation between these processes, although very few conflicts arise because the processes are not preemptive, meaning that a context switch from one process to another only occurs at the command of the running process. One way to force a context switch is to call the function `dispatch()`, this function selects a different process to run. If no other process of the same (or higher) priority is available, the same process remains running. The other way to accomplish a context switch is to wait for a resource. If the resource is not available, then the process will be become sleeping, until the resource is available again.

The LWP class is defined as follows:

```
class LWP: public Link, public Resource
{
 public:
   LWP( const char *n );
   void exit( int n );
   void exec( int stacksize = 4 );
   void wait( Resource& rsc );
   virtual int main( );
   void dispatch( );
};
```

It is not possible to run objects of class `LWP` straight away, first an extra level of inheritance must be created. A class must be made, that inherits from `LWP` and that defines the virtual routine `main`. This function contains the code that will be run inside the light weight process.

When a process is created it does not start executing right away. To do this, a call to `exec()` must be made. The optional argument to `exec()` is the size of the stack frame that should be made available to the process. The size is measured in units somewhat larger than one kilobyte. As soon as `exec()` is called, execution continues at the `main()` routine of the task. The very first time that `exec()` is called, the main program is converted to a light weight process as well.

The stacks of each of the processes are taken from the main stack. At every call of `exec()` another piece of stack frame is allocated. Using the `setjmp()` and `longjmp()` calls a context switch is implemented. This technique is not guaranteed to work on all platforms, but so far no problems have been encountered. Instead every thread has its own private stack. There is a real risk that these stack suffer from overflow, for example when recursion or large local data structures are being used. An alternative method is to have each thread use the normal program stack, at every context switch copy the current contents of the stack to an freshly allocated area on the heap, and replace it by the stack of the new thread. The advantage of this method is that it allows arbitrary growth of the stack, but it was felt that copying the stacks would be too expensive.

4.2 Resources

Resources are a part of the LWP library. A resource is in general something that a process can wait for. Resources are very flexible. It is possible to make resources that only one process can use at a time, or multiple processes can use simultaneously. The characteristics of resources can be designed by inheriting from the basic `Resource` class, and filling in the virtual functions. The idea behind the design of the `Resource` class is that one resource waits for one or more other resources. Consequently every resource can be waited for by an arbitrary number of other resources. It seems strange that we do not have a scheme in which processes wait for resources, but the current method is much more elegant. The run time environment uses resources for signals. Every signal is a resource that a process can wait for. The process will continue as soon as the signal changes value. If a signal changes value, then the corresponding resource is made available.

5 The compiler

The VHDL to C++ is based upon the tools provided by CLSI[1] implementing the front end of the compiler. The toolset includes an analyser, which parses the VHDL code and translates it into an abstract syntax tree. Using a special library, this abstract syntax tree can be loaded by the compiler back end. Subsequent passes in the back end walk through the tree and generate the C++ code.

[1] Cad Language Systems Inc.

The job of the compiler back end is relatively straightforward, because the semantic gap between VHDL and C++ is small. The elaboration phase is done during the execution of the final C++ code. Generate statements are translated to a while loop in C++. Generic parameters are implemented as normal parameters indistinguishable from port parameters.

6 Performance

Because the compiler was not ready at time of writing this paper, it is not very easy to give performance measurements with the complete system. However, because the runtime environment and simulation kernel are all but completed, it was possible to translate one medium-sized VHDL benchmark to C++ by hand. The benchmark is an implementation of the butterfly operation and some characteristics are shown in Table 1.

number of	quantity
processes	162
concurrent waveform assignments	146
signals	2844
simulation cycles	432
events	17421
transactions	56723

Table 1: Some characteristics of the benchmark.

Table 2 shows the number of cpu seconds of the benchmark running on our simulator and with the Vantage SpreadSheet system. The time includes both elaboration and simulation. The Vantage simulator has been used with as few options as possible. The simulation runs have been done on a HP 735 workstation at 99 Mhz. Due to the unavailability of our compiler we have not been able to compare compilation times. Note that the times only give a rough estimate of the performance of the two simulators, and are useful only to compare the order of magnitude of the running times.

	cpu seconds
Our simulator	10.1
Vantage	8.9

Table 2: Simulation times of the benchmark.

It seems that our flexible and object-oriented approach does not introduce a very large penalty, at least the performance of the two simulators is in the same order of magnitude. Taking into account that our runtime environment can be optimised to a high extent, it should be capable of delivering performance in the range of the fastest commercial VHDL simulators.

7 Turning sequential into parallel simulation

The simulator was designed to make a migration of sequential simulation to parallel simulation simple. There are however a multitude of problems that emerge when simulation must be done parallel. Most of these problems are new with the parallel case, because they are non-existent in the sequential case. The most eye-catching of these is the partitioning of the VHDL source onto multiple processors.

A large piece of work is required to make the simulation kernel suitable for parallel simulation. This involves message-passing, and some synchronisation problems. These changes can however be made without compiler support.

The remainder of this section gives a more detailed discussion of partitioning and the parallel kernel.

7.1 Partitioning

One of the most important problems for efficient parallel simulation of VHDL is the partitioning of the VHDL source onto multiple processors. The simulation will run fastest when the processors have more or less the same load and message traffic is not high (this may depend on the actual machine though). However, the load that a VHDL simulation will produce in the general case, is very hard to analyse.

A good VHDL partitioner takes at least into account the number of connections between nodes with their expected message traffic and the execution time of one VHDL simulation cycle on each processor. Because these characteristics are very hard to come by, and can vary enormously during simulation, estimations are very important. For simplicity's sake, the first partitioner will only take into account the number of connections between processors. In later stages more characteristics may be taken into account.

7.2 Parallel Kernel

For parallel execution, every processor node has its own simulator with all the ingredients that have been discussed in earlier sections. This simulator kernel has specific extensions to handle signals shared with other processors by means of message passing.

The kernel receives messages that contain events on signals that are used on the current node, but driven on an other node. The kernel has to send messages to other nodes that use signals driven on the current node.

8 Conclusions

We have succeeded in creating a simulator that:

- can handle full VHDL,
- has a simple compiler, and
- is flexible for easy modification and extension.

Although the compiler is not yet finished, we have been able to use the sequential simulator with a hand coded benchmark. Further, we have been able to use the simulator in a parallel performance evaluation framework with only small modifications necessary in the runtime environment and simulation kernel and without any changes in the generated C++ code.

The performance of the current unoptimised compiler is not shocking, but it is not shockingly bad either, so that some strategic optimisations could well lead to a very workable simulator. More advanced optimisations such as proposed by Willes and Siewiorek [4] are very applicable because of the flexibility and would speedup the simulator even more.

References

[1] P. Chawla, H.W. Carter, and P.A. Wilsey. An investigation of the performance of a distributed functional digital simulator. In *Proceedings of the 32nd Midwest Symposium on Circuits and Systems*, pages 470–476, August 1989.

[2] M. Hueber, S. Laserre, and A. de Monteville. VHDL experiments on performance. In *Proceedings of EURO-VHDL*, pages 282–292, 1991.

[3] D.R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[4] J.C. Willis and D.P. Siewiorek. Optimizing VHDL compilation for parallel simulation. *IEEE Design & Test*, pages 42–53, September 1992.