Centrum voor Wiskunde en Informatica
**REPORT***RAPPORT*

# A Two-Level Approach to Automated Conformance Testing of VHDL Designs

J.R. Moonen, J.M.T. Romijn, O. Sies, J.G. Springintveld,
L.G.M. Feijs, R.L.C. Koymans

Software Engineering (SEN)

**SEN-R9707 May 31, 1997**

Report SEN-R9707
ISSN 1386-369X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

# A Two-Level Approach to Automated Conformance Testing of VHDL Designs

Jean Moonen
*Philips Research Laboratories Eindhoven*
jmoonen@natlab.research.philips.com

Judi Romijn*
*CWI, Amsterdam, The Netherlands*
judi@cwi.nl

Olaf Sies
*Electrical Engineering Dept., Eindhoven University of Technology*

Jan Springintveld†
*Computing Science Inst., University of Nijmegen*
jans@cs.kun.nl

Loe Feijs
*Computing Science Dept., Eindhoven University of Technology*
feijs@win.tue.nl
*Philips Research Laboratories Eindhoven*
feijs@natlab.research.philips.com

Ron Koymans
*Philips Research Laboratories Eindhoven*
koymans@natlab.research.philips.com

ABSTRACT

For manufacturers of consumer electronics, conformance testing of embedded software is a vital issue. To improve performance, parts of this software are implemented in hardware, often designed in the Hardware Description Language VHDL. Conformance testing is a time consuming and error-prone process. Thus automating (parts of) this process is essential.

There are many tools for test generation and for VHDL simulation. However, most test generation tools operate on a high level of abstraction and applying the generated tests to a VHDL design is a complicated task.

*Research carried out as part of the project "Specification, Testing and Verification of Software for Technical Applications" at the Stichting Mathematisch Centrum for Philips Research Laboratories under Contract RWC-061-PS-950006-ps.

†Research supported by the Netherlands Organization for Scientific Research (NWO) under contract SION 612-33-006.

For each specific case one can build a layer of dedicated circuitry and/or software that performs this task. It appears that the ad-hoc nature of this layer forms the bottleneck of the testing process. We propose a *generic* solution for bridging this gap: a generic layer of software dedicated to interface with VHDL implementations. It consists of a number of Von Neumann-like components that can be instantiated for each specific VHDL design.

This paper reports on the construction of and some initial experiences with a concrete tool environment based on these principles.

*1991 Mathematics Subject Classification:* 68M15, 68Q68, 92G20, 94C12, 94C15.
*1991 Computing Reviews Classification System:* B.6.2, B.7.3, C.2.2, D.2.5, F.1.1.
*Keywords & Phrases:* Testing, embedded software, VHDL, conformance, test generation, test implementation, simulation.

## 1. Introduction

As is well-known, the software embedded in consumer electronics is becoming increasingly voluminous and complex. Accordingly, testing the software takes up an increasing part of the product development process – and hence of the costs of products. Therefore, Philips considers automating (parts of) the test process a vital issue.

More and more, manufacturers of consumer electronics do not completely develop the software themselves but import parts from other manufacturers. To guarantee well-functioning and interoperability of these parts, it is essential that they are tested for functional conformance w.r.t. internationally agreed standards. Therefore, testing efforts in this area concentrate on functional conformance testing (see [17, 13, 18] for testing terminology and methodology).

To optimise performance (in terms of speed or bandwidth), the lower layers of protocol stacks are often implemented directly in hardware. Testing these layers would imply hardware testing. However, Philips is interested in detecting design errors *before* implementation in silicon, which would mean testing hardware *designs* rather than their implementations.

Nowadays, hardware is designed using internationally standardised Hardware Description Languages. Testing a design then is testing a program in the description language at hand. Among the Hardware Description Languages, VHDL [16] is prominent.

There are many tools for test generation on the one hand and VHDL simulation, analysis and synthesis on the other hand. Moreover a lot of effort is put into extending and refining these tools. Ideally, therefore, the testing process could be automated by generating tests with a test generation tool, and then executing these tests using a simulation tool. However, most test generation tools expect behaviour to be modelled in clean-cut events with a high level of abstraction. Applying such tests to a VHDL design whose interface behaviour consists of complex patterns of signals on ports is by no means a trivial task.

Now, it is always possible to solve this problem by adding a layer of dedicated circuitry and/or software to bridge the gap between low-level events and high-level events, but it appears that the ad-hoc nature of this dedicated circuitry and software forms the bottleneck of the testing process.

We propose a *generic* solution for bridging the gap between generating tests on the abstract level and executing tests on the simulation level. This makes it possible for each of the two different tasks (test generation and test execution) to be performed at the appropriate level

within one test trajectory, with a higher degree of automation. The idea is to build a generic layer of software (written in VHDL), dedicated to interface with VHDL implementations. We call this layer the *test bench*. It consists of a number of components that fulfill various tasks: to offer inputs to interfaces of the implementation, to observe outputs at these interfaces and to supervise the test process. The components are Von Neumann-like in the sense that for each *specific* VHDL design they are loaded with sets of instructions. These sets are compiled from user-supplied mappings between high level and low level events and abstract test cases derived from the specification. In order to be maximally generic, the test bench should accept tests described in a standardised test language. In this way, any tool that complies with this test description language can be used for test generation.

Of course, this test bench will not solve all the problems involved in interpreting abstract tests. But by performing many of the routine (and repetitive) tasks, it enables the tester to concentrate on the specific properties of the interface behaviour of the protocol under test.

This paper reports on the construction of and some initial experiences with a concrete tool environment based on these principles. This prototype tool environment is called *Phact* and has been developed at Philips Research Laboratories Eindhoven, in cooperation with CWI Amsterdam and the universities of Eindhoven and Nijmegen. It consists of a test generation part and a test execution part. The intermediate language between the two parts is the standardised test description language TTCN (*Tree and Tabular Combined Notation* [17], Part 3). In the test execution part we find the test bench written in VHDL, with a front-end that accepts TTCN test suites.

In the current version of our tool environment, test generation is done by the *Conformance Kit* [6, 19] of Dutch PTT. This tool takes as input a specification in the form of an Extended Finite State Machine (EFSM) and generates a TTCN test suite for the specification. The *Leapfrog* tool from *Cadence* [7] is used for VHDL simulation.

This paper is organised as follows. In Section 2, we globally describe the tool environment and the testing process it supports. Section 3 highlights each important step in the test process. In Section 4, we describe our experiences with the use of the environment and discuss its current limits. Finally, in Section 5, we compare our approach with other approaches for analysis of VHDL designs.

*Acknowledgements* We would like to thank Nicolien Drost for developing an initial version of the observer compiler, Rudi Bloks for his help in understanding the finer details of VHDL and the Leapfrog tool, and the anonymous referees for their useful comments.

## 2. Global description of test environment and test process

In this section, we give an overview of the tool environment and the testing process it supports. The next section treats some interesting aspects in more detail. We begin with a short digression on *functional conformance testing*.

*Conformance testing* aims to check that an implementation conforms to a specification. *Functional* conformance testing only considers the external (input/output) behaviour of the implementation. Often the implementation is given as a *black box* with which one can only interact by offering inputs and observing outputs.

In the theory of functional conformance testing many notions of conformance have been

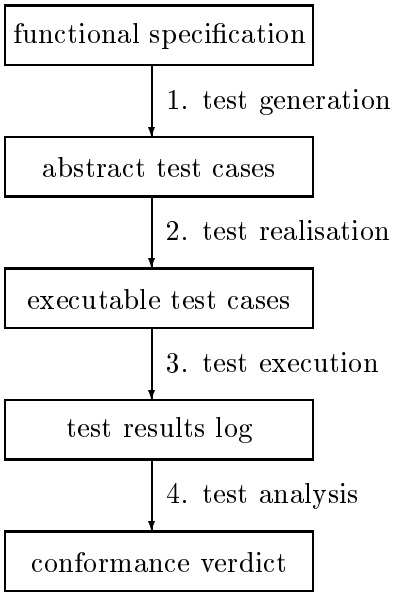


Figure 1: Global conformance testing process

proposed. The differences between these notions arise from (at least) two issues. The first issue is the language in which the specification is described (and the (black box) implementation is assumed to be described). Specifications can be described, e.g., by means of automata, labelled transition systems, or by temporal logic formulas. Secondly, the differences arise from the precise relation between implementation and specification that is required. Typically the different conformance notions differ in the extent to which the external behaviour of the implementation should match the specification.

Thus conformance testing always assumes a specific notion of conformance. However, for most conformance relations, exhaustive testing is infeasible in realistically sized cases: some kind of selection on the total test space is inevitable. So it is generally not possible to fully establish that an implementation conforms to the specification; the selected tests rather aim to show that the implementation approximately conforms to the specification. Conformance then simply means: the resulting test method has detected no errors. An appropriate mixture of theoretical considerations and practical experience should then justify this approach. This holds in particular for the test process supported by our tool environment.

Following ISO methodology [17, 18], the conformance test process can be divided in the sequence of steps given in Figure 1.

Our prototype tool environment automates the test generation and test execution phases and to a lesser extent the test realisation phase. It expects two inputs: the VHDL code for the Implementation Under Test (henceforth called IUT) and the (abstract, formal) functional specification, in the form of a deterministic Extended Finite State Machine (EFSM). From the EFSM specification abstract test cases are derived. These test cases are translated to the VHDL level and executed on the IUT. The history of the test execution is written to a log

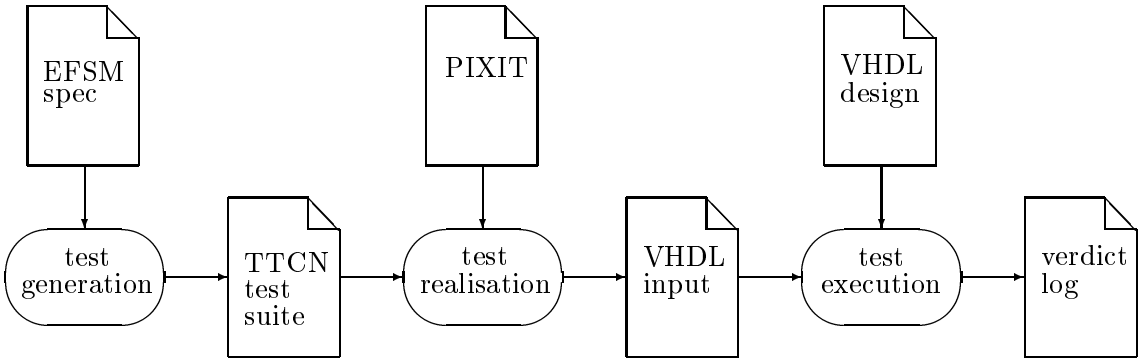


Figure 2: Overview of the test trajectory using *Phact*

file and the analysis phase just consists of inspecting this file and the verdicts it contains.

Note that the EFSM is required to be deterministic. We believe that the restriction to deterministic machines is not a real restriction since we are mostly interested in testing a single deterministic VHDL implementation.

The tool environment consists of two parts, taking care of test generation and test execution, respectively. Each one contains an already existing tool. Test generation is done by the *Conformance Kit*, developed by Dutch PTT Research [6, 19]. When given an EFSM as input, this tool returns a test suite for this EFSM in TTCN notation. The user can to a certain extent determine the parts of the EFSM that are tested and the particular test generation method used. We elaborate on this in Section 3.1.

The test cases in the test suite are applied to the IUT by a *test bench*, which is, like the IUT, written in VHDL. The *Leapfrog* tool from *Cadence* [7] simulates the application of the test suite to the IUT using the test bench. Thus testing an IUT here means: simulating it together with the test bench.

The test bench, which is described in more detail in Section 3.3 and in [21], consists of several components connected by a *bus*: *stimulators*, *observers*, and a *supervisor*. Stimulators apply input vectors to the IUT. Observers observe the output of the IUT and feed this information back to the supervisor. The stimulators and observers are diligent but ignorant slaves to the supervisor, which operates on the basis of the test suite and feedback from the observers. The test bench has been designed generically and only needs to be instantiated for each particular IUT.

*Compilers* connect the test generation part, the output of which is in TTCN notation, to the test execution part, the input of which must be readable for VHDL programs. There are three compilers, one for each type of component of the test bench. The compiler for the supervisor translates the TTCN test suite to an executable format. The compilers for the stimulators and observers map abstract events from the EFSM to patterns of bit vectors at the VHDL level. They require user-supplied translations (comparable to PIXITs in ISO terminology). Section 3.2 discusses this in more detail.

Given an IUT written in VHDL and a specification or standard to test against, the global test set-up from Figure 1 leads in our setting to the following sequence of steps, also depicted

in Figure 2:

0. (Manual) Write an abstract specification EFSM of the IUT.
1. (Automatic) Use the Conformance Kit to derive a test suite for this EFSM, specifying which parts of the EFSM must be tested and what test generation method must be used.
2. (a) (Automatic) Compile the test suite to the executable format for the supervisor.
   (b) (Manual) Define translations between abstract events and patterns of bit vectors (in Figure 2 called PIXITs).
   (c) (Automatic) Compile the translations to input files for the stimulator and observer, respectively.
   (d) (Manual) Instantiate the test bench as appropriate for the IUT. That is: enter the number of stimulator/observer pairs, the precise name and location of the compiled translation files, etc.
3. (Automatic) Run the Leapfrog tool on the instantiated test bench together with the IUT.
4. (Manual) Inspect the resulting conformance log file.

We end this section by remarking that the Leapfrog tool also allows the use of the Hardware Description Language *Verilog* [14]. In particular, the Leapfrog can simulate combinations of VHDL and Verilog programs, which makes it possible to plug a Verilog program as IUT into the VHDL test bench.

## 3. Stepwise through the testing process

The following sections explain the consecutive steps in the testing process more thoroughly.

### *3.1 Generating tests with the Conformance Kit*

The Conformance Kit consists of a collection of tools for test generation.

The Extended Finite State Machine model supported by the Kit is a slight extension of the traditional Mealy-style FSM model. Transitions are labelled with input/output pairs, where input and output are treated as simultaneous events (inputs without outputs are allowed). In addition to states and transitions, an EFSM may contain a finite set of variables that range over the booleans or over finite, convex subsets of the integers. Transitions may modify the values of the variables and may be guarded by simple formulas over the variables. There is also the option to mark transitions. For instance, it often happens that certain transitions are added to the EFSM only to make it complete. These transitions are artificial and should not be tested. This is achieved by marking them with a certain marker and excluding all transitions marked thus from the test generation. Finally, it is possible to specify Points of Control and Observation (PCOs) where inputs and outputs occur. They correspond to interfaces of the IUT.

To allow for test generation, the EFSM should be deterministic. Given a deterministic EFSM, one of the tools in the tool set builds a deterministic, trace-equivalent, and minimal

FSM (i.e., the FSM exhibits the same external behaviour as the EFSM and contains no pair of distinct but trace-equivalent states). Test generation tools proper take this FSM as input and return a TTCN test suite.

We highlight two of the test generation methods (for more information on test generation methods in general we refer to [11, 13]).

- The *Transition Tour* method. This method yields a finite test sequence (i.e., a sequence of input/output pairs) that performs every transition of the FSM at least once. Thus it checks whether there are no input/output errors.

- The *Partition Tour* method. In addition to the previous method this method also checks for each transition whether the target state is correct. It is similar to the UIO-method [20, 1] which in its turn is a variant of the classical W-method [9]. Unlike the Transition Tour method, this method yields a number of finite test sequences, one for each transition of the FSM. Each one is a concatenation of the following kinds of sequences:

  - A *synchronising sequence*, that transfers the FSM to its (unique) start state. Theoretically, such a sequence need not always exist. In practice however, most machines have a reset option and hence a synchronising sequence.
  - A *transferring sequence*, that transfers the FSM from the start state to the initial state of the transition to be tested.
  - The input/output pair of the transition.
  - A *Unique Input/Output sequence* (UIO) which verifies that the target state is correct (that is, all other states will show different output behaviour when given the input sequence corresponding to the UIO). If this sequence does not exist it is omitted.

  Although theoretically the fault-coverage of this method is not total [8] (not even when one correctly estimates the number of states of the implementation), the counter-examples are academic and we expect that the fault coverage in practice is quite satisfactory.

*3.2 From abstract tests to executable tests*

In the EFSM specification the input and output events of the IUT are described at a very abstract level. For instance, a complicated pattern of input vectors, taking several clock cycles, may have been abbreviated to a single event `Input_Datum_1`. The abstraction is needed to get a manageable set of meaningful tests. But when one wants to use the TTCN test suite derived from the EFSM to execute tests on the IUT, one has to go back from the abstract level of the EFSM to the concrete level of the VHDL implementation. This translation must be such that the VHDL test bench knows for each abstract event exactly what input should be fed to the IUT or what output from the IUT should be observed. For stimulators, the abstract input events have to be translated to patterns of input bit vectors. For the observers we have to write parser-code to recognise a pattern of output bit vectors as constituting a single abstract output event.

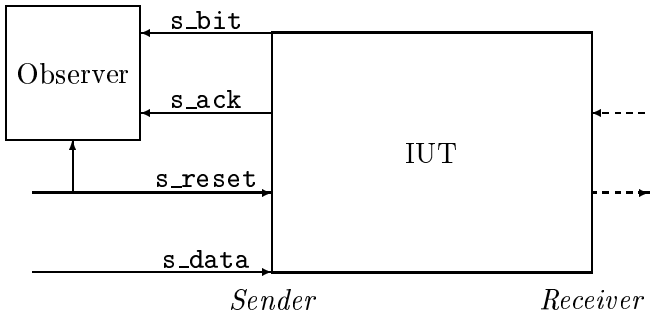


Figure 3: An example IUT

These user-supplied translations may be quite involved and hence sensitive to subtle errors. We expect that in the approach outlined in this paper, this is the part that consumes most of the user's effort.

The translation is constructed in four steps:

1. All abstract events used in the EFSM are grouped per PCO in input and output event groups.
2. All ports of the IUT are grouped into the input or output port group of one interface. Each interface should be associated with exactly one PCO.
3. Each event of an input (output) event group at one PCO is translated to sequences of values of the ports in the input (output) port group at the associated IUT interface. This is done for each interface.
4. All event translations are fed to the compilers that generate code which is understood by the test bench during simulation.

We will give a very simple example of a user-supplied translation that is input for the observer compiler.

The IUT for which the example file is intended is a protocol that transfers data from a Sender to a Receiver and, when successful, sends an acknowledgement back to the Sender. For synchronisation purposes, the acknowledgement is an alternating bit. The IUT has two interfaces (PCOs): *Sender* and *Receiver*. We consider the observer at the *Sender* interface, which should observe acknowledgement events. This situation is depicted in Figure 3.

The *Sender* interface has two output ports (which are connected to the input ports of the observer): `s_bit`, through which the alternating bit is delivered, and `s_ack`, through which arrival and presence of an acknowledgement is indicated. Furthermore, the interface has two input ports: `s_data`, a 4 bit wide port through which the *Sender* communicates data to the IUT, and `s_reset`, which has the value `1` whenever the *Sender* resets the IUT.

An acknowledgement event consists of an announcement that an acknowledgement is coming, followed by the acknowledgement itself. The announcement is indicated by the signal at `s_ack` having the value `1`; the value at the `s_bit` port is not yet relevant. Subsequently, the

acknowledgement is delivered: port `s_ack` still carries `1`, and port `s_bit` has the value `0` or `1` for the alternating bit.

Now we have all information needed to construct the translation that is input for the observer compiler. The translation code is given in Figure 4. Note that the lines preceded with `//` are comments.

First, the translation contains two so-called *qualifiers*, conditions that determine when the parsing of the output of the IUT at this interface should be started or aborted. Parsing should start when an acknowledgement is coming, so the start qualifier uses the value of the `s_ack` port. Parsing should be aborted whenever the IUT is reset, so the abort qualifier uses the value of the `s_reset` port.

Next, the event translation proper is given. Bit masks are defined to recognise individual output bit vectors. In this case the vectors represent two one-bit ports with `s_bit` at the first position and `s_ack` at the second. So mask `ack_coming` has `1` for `s_ack`, and `x` for `s_bit`, indicating that both `11` and `01` match here. Mask `ack_0` only matches when `s_bit` is `0` and `s_ack` is `1`. Output events are defined as regular expressions over the (names for the) bit masks. Here, the arrival of an acknowledgement is recognised by consecutive matching of the two relevant bit masks.

This two-phase definition of events reflects the way the observer parses the output from the IUT during execution.

*3.3 Executing tests at the VHDL level*

In order to test the VHDL implementation with the generated tests, we need to execute the VHDL implementation. Executing VHDL code means hardware simulation, for which we use the Cadence Leapfrog tool.

When simulating a VHDL program which models a reactive system, the program should be surrounded by an environment which behaves – from the program's point of view – exactly like the environment in which the program eventually must operate. This environment should also be able to observe whether the program is operating correctly, and to hand out verdicts reflecting these observations. Finally, since the execution is done by VHDL simulation, the environment itself should be programmed in VHDL too.

Creating the proper environment in VHDL is hard work. However, many tasks remain the same when testing different IUTs. We have therefore created a *generic* VHDL environment, which can easily be instantiated to suit any IUT. The environment we created to perform these tasks is referred to as the *test bench*.

The test bench consists of three kinds of components: a supervisor, some stimulators and some observers. The components communicate with each other by means of a bus. Figure 5 shows the structure of the test bench.

Each component type is dedicated to perform its particular task for any IUT. To achieve this, each component type has its own instruction set. When plugging an IUT into the test bench, each component is loaded with a sequence of instructions which are specific to the IUT in question. Thus the components can be viewed as small Von Neumann machines.

In the following paragraphs we explain the task of each component type in detail. Thereafter, we describe how the generic test bench is instantiated for testing a certain IUT.

The *supervisor* component has control over the whole test bench. It takes the generated TTCN test suite as input, works its way through each test case and outputs a log file with

```
// Observer bit patterns for the PCO at the Sender side

// Observed ports, with number of bits:
// s_bit(1) s_ack(1)

PCO Sender

QUALIFIERS

// Start parsing output when this qualifier is true
  [(:s_ack = '1')]

// Abort parsing when this qualifier is true
  [(:s_reset = '1')]

MASKS

  ack_coming = 'x1'
  ack_0      = '01'
  ack_1      = '11'

EVENTS

  ACK_OUT_0 = ack_coming ack_0;
  ACK_OUT_1 = ack_coming ack_1;
```

Figure 4: Example user-supplied translation for observer

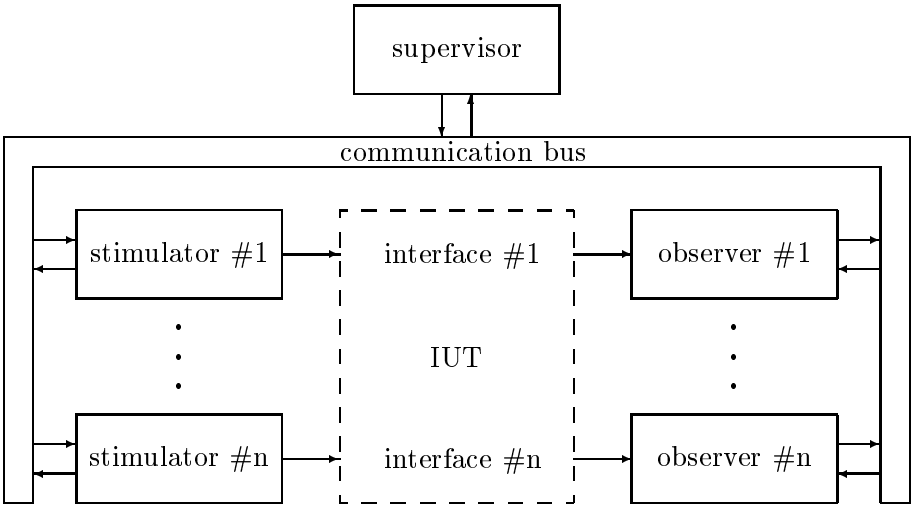


Figure 5: Structure of the VHDL test bench

the verdict and some simulation history. While traversing a test case, it steers the stimulator and observer components and uses a number of timers. Each test case is executed in the following way.

When the current TTCN test case states that input should be provided to the IUT, the supervisor notifies the stimulator at the designated interface. After the stimulator indicates that it has completed this task, the supervisor goes on with the remainder of the test case.

When the TTCN test case states that output should be generated by the IUT, the supervisor checks with the observer at the designated interface to see if this output has been observed. If the output has been observed, the supervisor goes on with the remainder of the test case. If nothing was observed, the supervisor will wait for the observer's notification of new output from the IUT. If output other than the desired output is observed, the TTCN code indicates what action should be taken. The TTCN generated by the Conformance Kit typically hands out the verdict *fail* in such a situation.

When the TTCN test case states that a verdict should be handed out, the supervisor logs this verdict to the output file, and quits the current test case.

The other TTCN commands handled by the supervisor are timer commands. TTCN offers the possibility to use timers for testing timing aspects of the behaviour of a system. These timers may be started, stopped and checked for a time-out. At the start of the TTCN test suite, all timers with their respective duration are declared. The supervisor handles these timer instructions in the obvious way. It can instantiate any number of timers with different durations and use them in the prescribed way.

The TTCN produced by the Conformance Kit, however, employs the timer construction in only two ways. It uses one timer for the maximum time a test case should take. This ensures that the test bench will not get stuck in the simulation. A second timer is used to test transitions from the EFSM that have an input event but no output. Since no output event is specified, the IUT should not generate one. This is tested by letting a timer run for some time, during which the IUT should not generate output. Any output observed before

the timer expires is considered erroneous and leads to the verdict *fail*. The precise value to which the no-output timer should be set is gleaned from the specification.

The *stimulator* component provides input to the IUT. It waits until the supervisor commands it to start providing a certain abstract event, then drives the input ports of the IUT with the appropriate signals. It has access to the user-defined translation of abstract input events to VHDL input signals.

The *observer* component observes output from the IUT and notifies the supervisor of the abstract events it has observed. Like the stimulator component, it has access to the user-defined translation of VHDL output signals to abstract output events.

Observing the ports of a VHDL component and recognising certain predescribed events is no trivial task. The observer must parse the output of the IUT such that the patterns provided by the user are recognised. Parsing is done with the help of a parser automaton, constructed with the UNIX tool Lex (and the user-defined translation). The observer uses this automaton to decide which event matches the current output. When the IUT outputs a sequence of values that does not fit into any of the patterns, the supervisor is notified of an error using a special error event.

The supervisor and stimulators communicate directly in a synchronous way – the supervisor always waits for the stimulators to end their activity before resuming its own task – while the supervisor and observers communicate in an asynchronous way via FIFO queues.

In order to plug an arbitrary VHDL implementation into the test bench as the current IUT, some *instantiating* has to take place. The test bench must have as many instantiations of the observer and the stimulator component as the IUT has interfaces. These instantiations must each be connected to the proper interface of the IUT. The IUT may need some external clock inputs, these have to be provided with the correct speed. The supervisor must have the desired number of timers at its disposal, as specified in the TTCN test suite. Each observer (stimulator) must be given access to the compiled version of the user-defined translation. Likewise, the supervisor must be given access to the compiled version of the TTCN test suite.

When these instantiating actions have been performed, the test bench is ready for simulation.

## 4. Experiences

We experimented with our tool environment by running it on a small protocol example. The protocol was derived from the Alternating Bit Protocol [2], with some modifications to test crucial features of the test bench. The features tested mostly concerned the synchronising mechanisms in the test bench.

During the test runs, the VHDL implementation we constructed for the example protocol proved not to conform to its abstract specification. Among other things, the toggling of the alternating bit was not implemented correctly. Already in this small protocol, multiple errors were detected that were subtle enough to escape a manual inspection of the VHDL code.

After conformance was shown for the corrected implementation, we modified the abstract specification EFSM to have discrepancies the other way around. All of these were detected.

Following this small protocol, we considered a fair-sized, more complex and industrially relevant design. For this we selected a part of the 1394 Serial Bus Protocol, which has been

standardised by IEEE [15]. The 1394 protocol implements a high speed, low cost bus that can handle communication between video and audio equipment, computers, etc. It supports multi-media applications, allows for "plug-and-play", and provides data transfer rates ranging from 100 Mbit/s to 400 Mbit/s.

The experiments have not yet been carried to completion but we can already report some of our findings. We sketch some problems that we encountered. We started off with a natural and abstract specification EFSM suggested by the standard document. However, when constructing the translation from abstract events to low-level events, we found that the interface behaviour of the implementation had a very high degree of interleaving of input and output events at different interfaces. In fact, the low-level representation of one abstract event often turned out to be a complete protocol in itself, involving low-level synchronization schemas and corresponding handshake mechanisms. To enable the test bench to deal with this behaviour, these protocols should be encoded into the stimulator and observer components. Given the simple, generic set-up of the stimulator and observer components, this appeared to be virtually impossible. This problem was worsened by the fact that the documentation of the protocol and the PIXIT information both lacked the degree of precision required to construct the translation.

It remains to be investigated whether the problems encountered with the complicated interface behaviour are specific to the 1394 protocol or occur more frequently and require a refinement or extension of the test bench.

The remainder of this section is devoted to the limits of the test generation method currently supported.

The EFSM specification format imposes certain restrictions. It has difficulties in modelling, e.g., output events without an input, events occurring simultaneously at multiple interfaces, data parameters of events, and timers. Solutions here require more research in the theory of testing.

Regarding the Conformance Kit itself, it would be convenient if the test generation process could be steered more directly by the user. For instance, one may want to transfer the implementation to a certain interesting state, and perform certain experiments in that state, whereas the Kit moves in a completely autonomous way through the state space.

## 5. Related work

Our tool environment has a modular structure and integrates two well-known techniques: one for automatic generation of TTCN test suites based on finite state machines and the other for the simulation of VHDL hardware designs.

A number of papers that employ similar techniques for analysing VHDL designs have appeared. Only [10] seems to follow a similar approach to conformance testing. When keeping the phased trajectory from Figure 1 in mind, the focus in [10] is on the test generation phase, the other phases are not described in detail. The method used for test generation is quite different from the classical graph-algorithmic approach such as applied by the Conformance Kit. Model checking techniques are used to derive the tests automatically from an FSM model of either the implementation or the specification. To test a certain transition, a model checking tool is fed with the FSM and a query asserting the non-existence of this transition. The tool derives a counterexample containing the path to the transition. This path is then used as a test sequence. More general temporal formulas can be used to direct

the counterexample to check certain situations. Selection of interesting transitions is based on a ranking of state variables, as opposed to the transition marking supported by the Kit (see Section 3.1). Although coverage is obtained w.r.t. the 'interesting' state variables, there is no measure for coverage w.r.t. exhaustive testing. It seems that theoretic support for dealing with the state explosion problem is as much an issue for this approach, as it is for ours.

In [12] a tool is described for exhaustive state exploration and simulation of VHDL designs. The VHDL design is transformed into an FSM for which a transition tour is generated (see Section 3.1). This tour induces a finite set of finite sequences of bit vectors which together exercise every transition of the VHDL design. As this tool only concerns simulation, there is no notion of conformance w.r.t. a specification, or a mechanism for automatic error detection.

In [22], a tool environment is described for the automatic execution of test scripts on VHDL components. There is no support for the automation of test script generation itself.

Finally, there exist many tools for the *verification* of VHDL designs (e.g., [3, 4, 5]). Each of them maps VHDL code to some semantical domain, on which the verification algorithms operate. It may be worthwhile to see whether our approach can benefit from techniques used in these tools.

## References


1. A.V. Aho, A.T. Dahbura, D. Lee, and M.U. Üyar. An optimization technique for protocol conformance test generation based on UIO sequences and Rural Chinese Postman Tours. In: IEEE *Transactions on Communications*, Vol. 39, no. 11, pp. 1604–1615, 1991. Also appeared in: Proceedings of *Protocol Specification, Testing and Verification* VIII, pp. 75–86, North-Holland, 1988.
2. K.A. Bartlett, R.A. Scantlebury, and P.T. Wilkinson. A note on reliable full–duplex transmission over half–duplex links. *Communications of the ACM*, 12:260–261, 1969.
3. I. Beer, Sh. Ben–David, C. Eisner, and A. Landver. RuleBase: an industry-oriented formal verification tool. In: Proceedings of the 33rd ACM *Design Automation Conference*, Las Vegas, NV, USA, 1996.
4. M. Bickford and D. Jamsek. Formal specification and verification of VHDL. In M. Srivas and A. Camilleri, editors, *FMCAD'96,* Palo Alto, CA, USA, November 1996, volume 1166 of *Lecture Notes in Computer Science*, pages 310–326. Springer-Verlag, 1996.
5. D. Borrione, H. Bouamama, D. Deharbe, C. Le Faou, and A. Wahba. HDL-based integration of formal methods and CAD tools in the PREVAIL environment. In M. Srivas and A. Camilleri, editors, *FMCAD'96,* Palo Alto, CA, USA, November 1996, volume 1166 of *Lecture Notes in Computer Science*, pages 143–158. Springer-Verlag, 1996.
6. S. van de Burgt, J. Kroon, E. Kwast, and H. Wilts. The RNL Conformance Kit. In: Proceedings 2nd *International Workshop on Protocol Test Systems*, pp. 279–94, North-Holland, 1990.
7. Cadence. Leapfrog VHDL Simulator. Product information at `http://www.cadence.com/software/qx-leap.html`.

8. W.Y.L. Chan, S.T. Vuong, and M.R. Ito. An improved protocol test generation procedure based on UIOs. In: Proceedings of *SIGCOMM '89*, pp. 283–294, ACM, 1989.

9. T.S. Chow. Testing software design modeled by finite-state machines. In: IEEE *Transactions on Software Engineering*, Vol. 4, no. 3, pp. 178–188, 1978.

10. D. Geist, M. Farkas, A. Landver, Y. Lichtenstein, S. Ur, and Y. Wolfsthal. Coverage-directed test generation using symbolic techniques. In M. Srivas and A. Camilleri, editors, *FMCAD'96,* Palo Alto, CA, USA, November 1996, volume 1166 of *Lecture Notes in Computer Science*, pages 143–158. Springer-Verlag, 1996.

11. S. Fujiwara, G. von Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. In: IEEE *Transactions on Software Engineering*, Vol. 17, no. 6, pp. 591–603, June 1991.

12. R. Ho, C.H. Yang, M.A. Horowitz, and D. Dill. Architecture validation for processors. In: Proceedings of the *International Symposium on Computer Architecture*, Santa Margerita Ligure, Italy, June 1995.

13. G.J. Holzmann. *Design and validation of computer protocols.* Prentice Hall, Englewood Cliffs, 1991.

14. IEEE. Standard description language based on the Verilog(TM) hardware description language. International standard 1364, 1995.

15. IEEE. Standard for a high performance serial bus. International standard 1394, 1995. For more information see `http://www.1394ta.org`.

16. IEEE. Standard VHDL language reference manual (ANSI). International standard 1076, 1993.

17. ISO. Information technology, open systems interconnection, conformance testing methodology and framework. International standard IS–9646, 1991.

18. K.G. Knightson. *OSI Protocol Conformance Testing: IS 9646 Explained.* McGraw-Hill, 1993.

19. E. Kwast, H. Wilts, H. Kloosterman, and J. Kroon. User manual of the Conformance Kit. Dutch PTT Research, Leidschendam, October 1991.

20. K.K. Sabnani and A.T. Dahbura. A protocol testing procedure. In: *Computer Networks and ISDN Systems*, Vol. 15, no. 4, pp. 285-297, 1988.

21. O. Sies. Automatic techniques for protocol conformance testing. Master's Thesis, Department of Electrical Engineering, Technological University Eindhoven, March 1996.

22. P. Walsh and D. Hoffman. Automated Behavioral Testing of VHDL Components. In: 1996 *Canadian Conference on Electrical and Computer Engineering*, Calgary, Alberta, Canada, May 1996.