MASTER

Automatic techniques for protocol conformance testing
study of automatic test generation and test execution applied to the Apple P1394 LinkCore

Sies, O.

Award date:
1996

Link to publication
Master's Thesis:

Automatic techniques for protocol conformance testing
Study of automatic test generation and test execution applied to the Apple P1394 LinkCore

O. Sies

Coaches : Dr. R.L.C. Koymans, Prof.dr.ir. L.M.G. Feijs
Supervisor : Prof.dr.ir. C.J. Koomen
Period : May 1995 - March 1996
Abstract

At present, labor involved in testing might take up to 50% of total efforts in the development of communication protocols. Even a marginal decrease of this testing effort is likely to amount to significant savings.

A lot of research has been done on the development of methods for automated conformance testing for communication protocols. Formal, computer processable, models captured from the applicable standard, serve as basis for test generation.

This report presents a method and tools which have been developed for the purpose of automated test generation and test execution for VHDL models of protocol implementations. The PTT Conformance Kit has been used for the production of abstract test cases in ISO-TTCN. The approach has been worked out for a VHDL implementation of the link layer of the IEEE P1394 protocol (a high speed serial bus protocol). A VHDL run-time environment to execute these abstract test cases has been developed.
### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.2 Timer array</td>
<td>31</td>
</tr>
<tr>
<td>3.2.3 The TTCN engine</td>
<td>31</td>
</tr>
<tr>
<td>3.2.3.1 Two level stack</td>
<td>32</td>
</tr>
<tr>
<td>3.2.3.2 Conformance Log</td>
<td>33</td>
</tr>
<tr>
<td>3.2.3.3 Zero time execution</td>
<td>34</td>
</tr>
<tr>
<td>3.2.3.4 Instruction set</td>
<td>34</td>
</tr>
<tr>
<td>3.2.4 Choices &amp; Implementation</td>
<td>36</td>
</tr>
<tr>
<td>3.2.4.1 Supervisor VHDL source</td>
<td>36</td>
</tr>
<tr>
<td>3.2.4.2 Tool implications</td>
<td>37</td>
</tr>
<tr>
<td>3.3 TTCN Timer</td>
<td>37</td>
</tr>
<tr>
<td>3.3.1 Overview</td>
<td>37</td>
</tr>
<tr>
<td>3.3.2 Timing source</td>
<td>37</td>
</tr>
<tr>
<td>3.3.3 Choices &amp; Implementation</td>
<td>38</td>
</tr>
<tr>
<td>3.4 Observer event recognition</td>
<td>39</td>
</tr>
<tr>
<td>3.4.1 Overview</td>
<td>39</td>
</tr>
<tr>
<td>3.4.1.1 Observable events</td>
<td>39</td>
</tr>
<tr>
<td>3.4.1.2 Finite automaton</td>
<td>40</td>
</tr>
<tr>
<td>3.4.2 Parser</td>
<td>41</td>
</tr>
<tr>
<td>3.4.2.1 Instruction set</td>
<td>41</td>
</tr>
<tr>
<td>3.4.2.2 Extensions to the event concept</td>
<td>42</td>
</tr>
<tr>
<td>3.4.3 Choices &amp; Implementation</td>
<td>42</td>
</tr>
<tr>
<td>3.4.3.1 Observer architecture</td>
<td>42</td>
</tr>
<tr>
<td>3.4.3.2 Observer VHDL source</td>
<td>42</td>
</tr>
<tr>
<td>3.4.3.3 Tool support</td>
<td>44</td>
</tr>
<tr>
<td>3.5 Stimulator event offering</td>
<td>44</td>
</tr>
<tr>
<td>3.5.1 Overview</td>
<td>44</td>
</tr>
<tr>
<td>3.5.2 Vector Processor</td>
<td>45</td>
</tr>
<tr>
<td>3.5.2.1 Extension to the event concept</td>
<td>45</td>
</tr>
<tr>
<td>3.5.2.2 Tables</td>
<td>45</td>
</tr>
<tr>
<td>3.5.3 Choices &amp; Implementation</td>
<td>46</td>
</tr>
<tr>
<td>3.5.3.1 Architecture</td>
<td>46</td>
</tr>
<tr>
<td>3.5.3.2 Stimulator VHDL source</td>
<td>46</td>
</tr>
<tr>
<td>3.5.3.3 Tool implications</td>
<td>47</td>
</tr>
<tr>
<td>3.6 Event queues</td>
<td>47</td>
</tr>
<tr>
<td>3.6.1 Overview</td>
<td>47</td>
</tr>
<tr>
<td>3.6.2 FIFO Choice &amp; Implementation</td>
<td>47</td>
</tr>
<tr>
<td>4 Towards P1394 test execution</td>
<td>49</td>
</tr>
<tr>
<td>4.1 Overview on the test trajectory</td>
<td>49</td>
</tr>
<tr>
<td>4.1.1 Outline</td>
<td>49</td>
</tr>
<tr>
<td>4.1.2 Blank spots</td>
<td>51</td>
</tr>
<tr>
<td>4.2 Modelling efforts</td>
<td>51</td>
</tr>
<tr>
<td>4.2.1 A parameterized EFSM style model</td>
<td>51</td>
</tr>
<tr>
<td>4.2.2 Mapping onto an unparameterized EFSM</td>
<td>54</td>
</tr>
<tr>
<td>4.3 Instancing a VHDL test bench</td>
<td>55</td>
</tr>
<tr>
<td>4.3.1 Overview</td>
<td>55</td>
</tr>
<tr>
<td>4.3.2 Implementation</td>
<td>55</td>
</tr>
<tr>
<td>4.4 Conclusions</td>
<td>57</td>
</tr>
</tbody>
</table>
4.4.1 Limitations of the EFSM model 57
4.4.2 LinkCore deviations 57

5 Conclusions 59
  5.1 Results obtained 59
  5.2 Test paradox 60
  5.3 Recommendations for future work 62

References 65

Appendix A VHDL Sources 67
  A.1 test_pkg.vhdl 67
  A.2 fifo.vhdl 70
  A.3 observer.vhdl 73
  A.4 stimulator.vhdl 79
  A.5 timer.vhdl 84
  A.6 supervisor.vhdl 88
  A.7 test_bench.vhdl 95

Appendix B TTCN compiler example 105
  B.1 Selection of three test cases from a P1394 suite 105
  B.2 After first compiler pass 107
  B.3 After second pass of compilation 108
  B.4 After third pass of compilation 110
  B.5 After fourth pass of compilation 112
  B.6 After fifth pass of compilation 115
  B.7 End result of compilation 117

Appendix C Observer code example 121
  C.1 Input definition file parser.def 121
  C.2 Mask file mask.dat 121
  C.3 Parser code file parser.dat 121

Appendix D Comments regarding the EFSM model 123
  D.1 Packet structure 123
  D.2 Interface description 123
    D.2.1 From transaction layer to link layer 123
    D.2.2 From link layer to transaction layer 123
    D.2.3 From link layer to physical layer 124
    D.2.4 From physical layer link to layer 124
    D.2.5 From node controller to link layer 124
    D.2.6 From link layer to node controller 124
  D.3 Functions definitions 125
  D.4 Notes 126
  D.5 Mapping onto a Conformance Kit EFSM 126
    D.5.1 Mapping procedure 126
    D.5.2 Mapped EFSM 127
    D.5.3 Restriction file 129
Chapter 1

Introduction

1.1 Protocol Standards & Conformance Testing

The first computer communications networks introduced to the industrial market were manufacturer proprietary solutions. Companies often had an array of incompatible systems for specific tasks. Systems from different vendors could not be coupled to a network.

This chaotic situation has led to the arise of standards. The purpose of standards is to make the establishment of open multi-vendor systems, like the one in Figure 1.1, possible. Equipment from a set of different manufacturers (say A, B, C, D and E), conforming to the same standard, must interoperate successfully in a single network. The standard should facilitate manufacturer development and customer procurement of equipment that is to be coupled to such a network. Standards can be drawn up by standardization organizations like ISO (International Organization for Standardization), from inception (de jure standards) or can evolve from proprietary systems that have gained support from other manufactures (de facto standards).

![Multi-vendor systems.](image)

Any manufacturer can make the claim "my equipment conforms to standard ISO-xxxx". The act of conformance testing tries to establish confidence in that claim.

Conformance tests can be conducted by the manufacturer of the equipment itself. This first party conformance testing usually precedes testing by other parties and market introduction. Second party conformance tests are conducted by the potential buyer of the equipment. This is the party that is the ultimate beneficiary of the test outcomes. The
equipment required to conduct conformance testing is very complex and expensive; the entire process of comprehensive conformance testing is a highly specialized task. Tests conducted by independent testing laboratories are referred to as third party conformance tests. The idea is that a certain piece of equipment is offered only once to a single testing laboratory. The laboratory certifies the equipment to conform to the applicable standard only if all test outcomes are positive.

An implementation should not pass the conformance tests of one laboratory and fail the tests of another laboratory. To achieve this, laboratories must use the same standardized test sets and conduct these tests according to the same principles and methods so that they will lead to the same, generally accepted test outcomes. Hereto ISO and CCITT have drawn up the IS-9646 standard (parts 1 through 7): "Information technology - Open Systems Interconnection - Conformance testing methodology and framework". It describes how to define a standardized suite of test cases, the associated implications/requirements on the equipment to be tested, and implications/requirements on the equipment with which the testing is performed. As a consequence IS-9646 has an extremely wide scope governing topics like:

1) How to write base standards.
2) How to specify conformance requirements within standards.
3) General concepts and principles of conformance testing.
4) How to make claims of conformance.
5) The sorts of tests and test methods employed.
6) How to write test specifications, including a language with which to do it.
7) The design of the equipment under test.
8) Responsibilities and roles of all parties involved in conformance testing, including:
   - the test equipment supplier
   - the testing agency
   - the party whose equipment is to be tested

The quality assurance and testing procedures have to be up to standard before a laboratory can be accredited by an accreditation authority. If the specific application field lacks such an authority, laboratories can accredit each other (mutual accreditation).

Only then, can the (pseudo-)certificate resulting from conformance testing be widely accepted and used as an impartial judgement in the procurement process of customers.

1.2 FDTs and formal design

With the best of intentions words can be misinterpreted. Standards formulated in a natural language, like English, risk being interpreted differently by designers and testers alike. Several Formal Description Techniques (FDTs) have been developed to alleviate problems of this kind. Interpretation ambiguities are eliminated by specifying in terms of mathematical objects in a language with formally defined syntax and semantics.

The development of any system usually starts with gathering its operating requirements. Requirements or constraints can be in explicit, technological terms, leaving little doubt as
how to physically implement them. Other constraints are fuzzy, stated in non technological terms, more qualitative than quantitative, or reflect a mere customer wish for functionality (which can be negotiated upon). Constraints of the last type alone are clearly not enough to determine final system realization; design decisions have to be made.

Formal techniques can be used to structure and support the protocol design phase itself. An idealistic view on the formal system development process is depicted in Figure 1.2. To support an iterative design approach, a formal notation is used to capture the intermediate results/model of the desired system. The informal (start) idea, present in the minds of the design team, is transcribed into a formal equivalent $S_0$. The process of ensuring that the transcription to the formal domain matches the original idea is called validation. In each design iteration, some additional constraints are translated into mathematical objects to enhance model $S_i$ into a more detailed version $S_{i+1}$ ($0 \leq i < n$). These newly transcribed constraints should then again be validated. When two models are within the same formal domain, as is presupposed for $S_0$ through $S_n$, the presence of an equivalence relation (like observation equivalence) can be shown through verification. By exploiting inference rules/axioms of the formalism this relation can (sometimes) be mathematically proven. Verification is used to ensure preservation of formalized properties throughout the iterative design trajectory and validation is used to assure the correctness of the properties themselves. Once all constraints have been formalized and the resulting model contains enough information to make a mapping onto a physical implementation $I$ possible; the final realization step can be taken. The reversely directed realization step transcripts all mathematical objects contained in $S_n$ into technology dependent, physical objects. Through testing one tries to assure the success of the realization step by experimenting with the implementation.

Figure 1.2 The system development process based on formal methods.
4 Introduction

Formal models can be passed between design teams without interpretation ambiguities, stored for future reference (exploiting that same benefit) and the possibility for computerized support for symbolic reasoning (verification), simulation (testing), analysis (validation), etc. These (potential) advantages of the formal design approach can however be diminished, or even overshadowed, by the disadvantage of overhead in formalizing, de-formalizing and verification steps. Efforts should be put into tools support, with carefully constructed user interfaces [Koo91], minimizing overhead and exploiting the potential strengths of a formal design methodology.

1.3 Abstractions and abstract testers

Complex problems in general do not have trivial solutions. The design of communication protocols certainly is complex. There are two 'tools' at our disposal to attack complex design problems: decomposition and abstraction.

Decomposition basically splits up a complex problem into a number of subproblems, individually easier to solve. The sum of all subproblem solutions equals the solution of the overall problem. A subproblem can, if still too complex to handle, again be decomposed into yet smaller problems resulting in a hierarchy of problems. The sum of all leaf problem solutions in that hierarchy again builds the overall solution. Detailing through decomposition implies adding structural detail about the manner in which subproblems relate (or interface) with other subproblems to solve the problem one level higher in the hierarchy.

Abstraction is the process of comparing a complex problem, as a whole, with a similar but simpler problem, easier to solve. In other words: we abstract from detail in order to get a clearer picture on an underlying, simpler problem. After we have solved the underlying problem we can add back some off the (problem) detail and build a more detailed solution on the abstract solution. Moving through a number of gradually lowering levels of abstraction the detail level of (or quantity of information contained in) the obtained solution increases until the realization of an implementation is possible. Decomposition could be interpreted as structural de-abstraction in detailing towards an implementation.

The choice for a particular FDT implies a more fundamental, functional abstraction. With a FDT we can denote an abstract, simplified model of a physical implementation. By choosing for a particular FDT alone, we exclude whole categories of implementation details (functionally abstracting from them) that we must address in a later stage of detailing; there is simply no expressive power in the FDT to describe these aspects of our future implementation. When we choose CCS or LOTOS for a FDT we abstract from timing issues and use a strictly synchronous, atomic (one event at a time) communication model (there is a notion of "event A before/after event B" but not of "event A 10 seconds before/after event B"). A FDT can support stepwise functional de-abstraction, up to some level, within its expressive power. In protocol development within CCS we could focus on connection establishment first (and abstract from details of data transport) by constructing models using unparameterized events. In a later stage we then can refine the concept of data transmission by using parameterized events and value passing.
When we look back at the idealistic formal design trajectory depicted in Figure 1.2, we can state that a formal model $S_i$ is more abstract, and thus contains less information/detail, than any model $S_j$ if $0 \leq i < j \leq n$. An intermediate model (here $S_1$) is referred to as the specification when it describes, in the maximum detail possible, the way in which the implementation shall interface with its environment. A specification could be obtained by starting of with a functionally abstract $S_0$ from which we de-abstract by stepwise addition of interface detail. All the informal operating constraints that lend themselves for formalization are formalized at this point. The remaining constraints that could not be transcribed into the formal domain must be taken into account during the realization step.

A good specification refrains from unnecessarily specifying internal implementation details (doing so is in fact over-specifying). Ideally, communication standards should contain such formal specifications so that they can be structurally detailed ($S_1$ through $S_n$) by different manufacturers; guaranteeing conformance, at the abstract level, throughout the design process. We can test the implementation to conclude on the preservation of all (formal) specification properties.

Through experiments one can try to establish confidence in the fact that an implementation coheres to a certain abstract model. There are several approaches to testing thinkable. Through exhaustive testing, a model, at the same abstraction level as our reference model, could be inferred from the test results. The inferred model could be compared with the reference model to show the presence of an equivalence relation. Such an approach, if possible at all, would require many, many tests.

Alternatively, we can look for errors by constructing experiments (test cases) that try to expose examples of non-conforming behavior. The Tree and Tabular Combined Notation (IS-9646 part 3: TTCN) is intended to denote tests of this type at an abstract level, in a standardized way. We can write TTCN test cases that exercise the functionality of the reference model by application of stimuli to an Implementation Under Test (IUT) and evaluation of observed responses. The rules needed to decide on what behavior observed will be considered correct, and what will be considered incorrect, are encoded into each test case. Each test case is an interactive "test program" that tries to satisfy the test purpose associated with it. Observation of correct response behavior leads to either application of more stimuli or (ultimately) to a positive conformance verdict PASS. When intolerable response behavior is observed, test case execution will end with assignment of the FAIL verdict. Execution of a test case that has not revealed any non-conforming behavior, but has not satisfied the corresponding test purpose either, produces an INCONCLUSIVE verdict.

Techniques are being developed to obtain abstract test suites by automatic derivation from an abstract formal model. These techniques are somewhat biased by the FDT used. Essentially they share the same concepts. The original specification is mirrored (input and output identifiers swap places) and selectively traversed according to some kind of strategy. The traversed traces are denoted in TTCN's behavior tree format (dynamic part) and take their stimuli (former specification inputs) and correct responses (former specification outputs) directly from the mirrored specification. Input nodes are expanded with illegal, non-conforming behavior (leading to FAIL verdicts). Test generation specifics will be addressed in more detail in a later stage. For now it suffices to think of an abstract
6 Introduction

test case as a specification of a tester in the same abstract terms as the specification it is derived from. The abstract tester assigns verdicts as a result of interaction with the abstract model underlying the implementation. This need not be the same model as the tester is derived from; the intention of the whole testing effort is to try show that it is not!

The abstract tester needs to be implemented in the same manner as the result of the final detailing step $S_n$ must be implemented. The concept of stepwise de-abstraction, within a formal domain (or the formal plane imposed by the FDT used) and the relation with automatically generated abstract testers is depicted in Figure 1.3.

![Figure 1.3 Abstraction by formalization and generation of abstract testers.](image)

The decision to support the design process by using a particular FDT, confines us to the expressiveness within the upper formal plane. The formalization process is done in a single step here. It need not be so. Stepwise functional de-abstraction could be employed to yield an intermediate model, say $S_1$, that describes the external interface in the maximum detail possible within the formalism used. $S_1$ could then be used to generate the abstract tester $T_1$. Additional detailing steps (structural de-abstraction) add only internal detail to $S_2$ through $S_n$, irrelevant to conformance testing on the external interface. Generation of $T_2$ through $T_n$ must therefore abstract again from the internal detail of the corresponding specifications. If the results of the consecutive design iterations have been properly verified, $T_1$ through $T_n$ should be identical. $T_n$ however contains less detail of the interface itself; the test realization step must account for this lack of detail.

If we would wish to employ white box testing instead of black box testing, the higher detail level of $S_1$ through $S_n$ could be exploited to test individual components/parts of the implementation. The total effort involved in decomposed testing of all individual parts could be significantly less than attacking the overall testing problem as a whole. We will however focus solely on black box conformance testing here.

In Figure 1.3, the idea about the system has been solidified into a full detail, informal specification before making a transcription into the formal domain. This is possibly not the most logical course of action but it reflects the P1394 testing situation. The next section presents an overview of P1394, what we wanted to test and how we planned to do it.
1.4 Conformance testing of the P1394 LinkCore

The intention of my graduation assignment was to reflect on the practical applicability of formal methods in conformance testing in the current state of their development. A practical case study has been coupled to the "in-house" development of an audio-video application component for the IEEE-P1394 serial bus protocol at the IPA group (Information Processing Architectures) of the Philips Research Laboratories, Eindhoven.

1.4.1 General overview on P1394

The P1394 protocol stands on two possible physical environments: the backplane environment (applicable as an alternate for a parallel microcomputer bus for onboard communication) and the cable environment (an external peripherals bus with scaleable, signalling rates up to 400Mbit/sec). The IPA component is intended for use with the cable environment only. Figure 1.4 depicts the three layer P1394 protocol stack.

![Diagram of the P1394 protocol stack]

Figure 1.4 Overview on the P1394 protocol stack.

The cable environment physical layer provides the actual interface between nodes including both the mechanical and electrical interface. It projects a multi access bus onto a topology built up out of point-to-point links. One serial bus supports interconnection of a 63 node maximum in a topology within which any two nodes can be separated by no more than 16 cable hops. At the lowest level it includes the physical connection which consists of the cable media itself, the connectors, and the electrical transceivers. The physical layer also performs a number of higher-level functions including bus state determination, bus arbitration, all the encoding and decoding functions, synchronization of received data to a local clock, and a data repeat function.

The link layer of the serial bus provides connectionless acknowledged data transfer services between a source node and a destination node. In the link layer the basic PDU is
8 Introduction

the packet and its corresponding acknowledge, together defined as a "subaction". There are two types of subactions:

   a) Asynchronous - The source node transmits data in turn, using the appropriate access method.
   b) Isochronous - The source node transmits data every isochronous cycle with a guaranteed maximum latency. The maximum size of each of these packets is set by the bus management entity.

Isochronous subactions have "guaranteed" access to the available bus bandwidth, while asynchronous subactions use the remaining bandwidth. It is responsibility of the isochronous resource management process to guarantee that adequate bus bandwidth remains for asynchronous use.

The transaction layer provides three operations for the transfer of data between nodes:
   a) Write transactions - Data is transferred to an address in a different node.
   b) Read transaction - Data is retrieved from an address in a different node.
   c) Lock transaction - Data is sent to a different node, used to perform an indivisible function, and the results are returned.

Transactions are multi-threaded, in that more than one transaction can be started by a requester before the corresponding response is returned. These are called split-response transactions.

1.4.2 What to test?

Engineers at IPA are constructing an application component that is to fit on top of the three layer P1394 stack. This component is implemented together with the LinkCore (purchased from Apple Computer, inc.) on the same silicon. This application specific component is meant to facilitate the development of a special category of applications that connect to the serial bus.

At the lower side it is intended to be connected with a third party physical layer for the cable environment (for instance Texas Instruments's TSB21LV03 tripe cable transceiver/arbiter). At the upper side it interfaces with a host microprocessor (control part) and related hardware (for data flow).

The development of the component is done using the Cadence VHDL environment. The IPA group functionally refines the design down to a synthesizeable level. The Philips Logic Product Group (Albuquerque, NM, USA) handles synthesis to gates and production of the silicon. The LinkCore is modelled with Verilog. The Cadence Leapfrog simulator can handle mixed VHDL-Verilog models allowing the LinkCore to be embedded into the overall design.

The P1394 standard, available in its most recent draft (8.0v2, July 7, 1995), specifies all layers individually. Questions regarding conformance of the Apple LinkCore with respect to the link layer specification, have been raised.
It is our goal to judge on conformance of the LinkCore, use automatically generated test cases in TTCN, and to gain as much experience as possible in the process of doing so. We hope to illustrate the strengths of the method with the results obtained and hope to chart (potential) bottlenecks and fundamental problems so that they may guide future development of formalisms and abstract testers.

1.4.3 How do we test it?

The test trajectory we have in mind can be divided into two parts: a test preparation part and a test execution part.

The test execution part conducts the interactive experiments with the design under test itself. Based on the test results we can judge on conformance of the LinkCore. We intend to develop a generic tester (or test architecture) that is simulated together with the IUT in a larger design, called a test bench (see Figure 1.5). The tester accepts test specific code produced in the test preparation part. The tester produces a log file that contains relevant, time-stamped test information and the verdicts assigned by the different abstract test cases.

**Figure 1.5 Test execution environment.**

The test preparation part comprises of all those activities, with or without tools support, and their intermediate deliverables, that must be done/obtained off-line prior to test execution. Figure 1.6 sketches the three step preparation trajectory we have in mind.

**Figure 1.6 Test preparation steps.**
10 Introduction

We will be using the PTT Conformance Kit as test generation front-end because of its availability and to benefit from the knowledge and experience that the IT group (Information Technology) already has with this set of tools (see [Luk94], [Kwa94]). The Conformance Kit uses an EFSM formalism as a basis for test generation. The Extended nature of the Mealy style Finite State Machines refers to the use of local machine variables (booleans and finite integer ranges). Transitions are of a coupled stimulus/response style and may be guarded by predicates over the machine variables. Any transition of the input/output automaton can only be triggered by its specified input behavior and may exhibit optional coupled output behavior. The input/output events themselves are unparameterized and the models are therefore more suited to control flow testing than data flow testing. The TTCN test cases produced use only a subset of that language (no test case variables and no event parameters/data; see paragraph 2.2.2).

Testing will take place, as stated earlier, at the simulation level with the Cadence Leapfrog simulator. In doing so we hope to verify the presence of functional properties of the EFSM specification in the Verilog hardware model through testing. We expect these functional properties (verified through testing) then to be correctly maintained by the Synopsis synthesizer. To ensure this, final conformance testing would have to take place on the realized silicon.

The final form of the test architecture is dictated by the operational semantics of TTCN test cases and the way in which the missing detail is to be added to the test cases. Once the test architecture crystallizes and the format (syntax and semantics) of its input file(s) has been determined the work on the supporting tools of the test realization environment can be started.

1.5 Overview on Thesis

This thesis focusses primarily on the test realization part and in particular the conception of a VHDL run-time environment for the execution of ISO-TTCN test cases. Chapter 2 deals with the background of VHDL test benches and the TTCN language. Chapter 3 describes the development of an universally applicable VHDL test environment for the execution of abstract TTCN tests. Chapter 4 summarizes the experiences in applying the method and tools to the VHDL implementation of the P1394 link layer. Chapter 5 presents a rundown of the results and the overall conclusions.
Abstract testing in VHDL

2.1 Test benches

Advocated VHDL design methodology employs testing for maintaining desired/required functionality across design iterations, and for validation of newly added structure. Tests are conducted within a simulation environment in which hardware models can be brought to life. At IPA the Cadence Leapfrog simulator is used for this purpose.

In this section we deal with the additionally needed VHDL architecture that, together with the Entity Under Test (EUT), constitutes a test bench. A test bench is the aggregate of VHDL models needed to make test simulation runs possible.

We first address general controllability and observability issues in VHDL testing. Next we describe a simple vector processor that applies stimulus vectors from file to an EUT. We conclude to describe how this concept has evolved into a test bench that uses programmable vector processors, capable of interactive testing and vector reuse.

2.1.1 Testing a VHDL design

VHDL doesn't force a designer to use particular design methodology, but intends to support a designer in any methodology he or she uses. Motives for using testing at the simulation level can therefore vary. Whatever objective strived for, whichever test method employed; the common factor is that through (series of) predefined experiments one hopes to decide on correct functioning of implemented solutions by analyzing simulation results.

The simulator can be instructed to trace events on a predefined set of signals and variables in the WAVES (Waveform and Vector Exchange Specification) format. After simulation results can be reviewed and analyzed (manually) with the cWaves waveform viewer that displays a "signal/variable data vs. time" log in a logic analyzer like fashion. In general however, the capability of observing alone is not enough to conduct a test; we need a mechanism to supply stimuli to the EUT in order to induce the responses we are looking for. It is customary to use VHDL to describe both the hardware design and the environment used to test the design in a test bench.
Say, design efforts have resulted in a design as depicted in Figure 2.1. This design entity is intended to interface, through its four ports, with an application environment. EUT operations depend on the stimuli, that it reads from signals mapped to the input ports of these four interfaces.

![Figure 2.1](image)

*Figure 2.1  Schematic view on an Entity Under Test.*

To evaluate, and judge on, the correctness of implementation, one should simulate this design together with an adjacent environment that delivers real life stimuli to the inputs of its interfaces. If the EUT is a part of a larger system we could choose to simulate the EUT together with the rest of the system to obtain the desired interactive EUT environment. In Figure 2.2 we have done just that; entities A, B and C are the component instances that make up the EUT's system environment.

![Figure 2.2](image)

*Figure 2.2  The EUT and its surrounding VHDL systems environment.*

Our original problem (how to obtain stimuli resembling real life operations) has now been relocated to the external boundaries of the added entities. Many practical systems ultimately interact with external processes that fall outside the modelling scope of VHDL. Think of a console operator, sensors converting external information into numeric representants, input files for processing, etc. For influences/stimuli that fall outside the scope of VHDL models, we are obliged to build a terminating hardware model that supplies these stimuli somehow. The *std.textio.ALL* functions are the only standardized way to input (or output) data in to (or out off) a VHDL process during simulation. When this data comprises of arrays of type BIT (or related types, like *std_logics*) intended to drive digital busses, these are generally referred to as stimulus *vectors*. Figure 2.3 depicts a test bench that uses vector processors to read stimuli from file during simulation.

We now have a global layout for a test bench that enables us to apply predefined stimuli from file, and we can trace results using *eWaves*. It is a great advantage that no re-compilation is needed to test with modified stimuli; we only have to modify the vector files.
Whether or not to include the adjacent A, B and C components in our test bench depends on a number of issues; some practical, some more fundamental. We do not always have VHDL models of the designs operating environment at our disposal. It might be that they do not exist, or that they are the intellectual property of a competitor. A fundamental question we should ask ourselves is: "what do we really want to test?". If we are only interested in validating functional properties of the IUT itself then the added components are a burden. Testing through a surrounding operating environment (or embedded testing) restricts the controllability of the EUT and requires additional knowledge of this environment. If we are interested in determining whether the total system operates "correctly" in conjunction with the IUT, such interoperability tests can be valuable.

As not to complicate the testing process beyond our functional scope we focus on direct testing of a single entity. The concepts developed are universal and can be used for other test methods as well. Figure 2.4 depicts the test bench we would need for our IUT. The following sections show how a such a vector processor could be implemented.
Abstract testing in VHDL

2.1.2 A simple vector processor

The simplest test bench thinkable, is given in Example 2.1. It consists of an EUT and one stimulus process. The EUT itself is not elaborated upon, but the idea is that we are interested in observing EUT responses to stimuli supplied by the stimulus process, on the result signal.

Example 2.1 A simple test bench using a dedicated stimulus process.

```vhdl
USE std.textio.ALL;
ENTITY testbench IS
  -- portless entity containing an EUT and a vector processor
END test_bench;
ARCHITECTURE behavior OF testbench IS
  SIGNAL a, b, result: bit_vector(3 DOWNTO 0);
  COMPONENT design
    PORT {
      a_in: bit_vector(3 DOWNTO 0);
      b_in: bit_vector(3 DOWNTO 0);
      out: bit_vector(3 DOWNTO 0);
    }
  BEGIN
    -- instantiate design under test
eut: design
      PORT MAP (a, b, result);
    -- a stimulus process to drive EUT inputs
    stimulus: PROCESS
      FILE f: text IS IN "vectors.dat";
      VARIABLE l: line;
      VARIABLE timewhen: time;
      VARIABLE a_value, b_value: bit_vector(3 DOWNTO 0);
    BEGIN
      WHILE NOT endfile(f) LOOP
        readline(f, l);
        read(l, timewhen);
        read(l, a_value);
        read(l, b_value);
        WAIT FOR (timewhen - NOW);
        A <= a_value;
        B <= b_value;
      END LOOP;
      WAIT;
    END PROCESS stimulus;
  END behavior;
```

We will focus on the stimulus process. Formatted input is done in two stages using subprograms from std.textio. The readline() procedure reads the next line from file into the access type line buffer. A line is build up out of three items. The first item indicates the simulation time at which the stimulus vector must be applied. The stimulus vector itself is defined by the next two items. The overloaded read() procedure reads the next item into a variable and removes the item from the start of the line buffer. The function endfile() is used to prevent reading past the end of the vector file. The WHILE LOOP continues to read lines until the file end is reached (in which case the process suspends indefinitely) or the simulation run ends, whichever comes first.
Each second item read from the line buffer is assigned to the \( a \) signal to drive the EUT's \( a_{in} \) input port. Each third item read from the line buffer is assigned to the \( b \) signal to drive the EUT's \( b_{in} \) input port. One could argue that, instead of one, two stimulus vectors are offered simultaneously. In the latter we will use the term stimulus vector for all data, including enumerated types other than BIT, that is offered parallel and sequenced in time at one interface.

The process of writing stimulus vectors for vector processors of this kind it a lengthy task if done by hand. Long lists of time marks and vector data will soon become unintelligible and incomprehensible; clearly some kind of structuring is needed. Reusable, low level 'event' vector patterns could be identified and test sequences could be constructed in terms of such 'events'. In Example 2.2 such an approach is elaborated upon. The event definitions (relative time) in the first column are referenced by the event sequence (absolute time) in the second column. Tool support for such a structured approach is very well possible. The final vector file in the third column could be produced by a compiler that performs additional consistency checking on event definition/application pairs and overlapping events in absolute time.

### Example 2.2 Structured approach to vector file writing.

<table>
<thead>
<tr>
<th>Event definitions (relative time)</th>
<th>Event sequencing (absolute time)</th>
<th>Resulting vector file (absolute time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDLE: ( 0 ) NS ( 0000 ) ( 0000 )</td>
<td>10 NS ( \text{IDLE} )</td>
<td>10 NS ( 0000 ) ( 0000 )</td>
</tr>
<tr>
<td>QUICK_SWEEP: ( 0 ) NS ( 0000 ) ( 1111 )</td>
<td>20 NS ( \text{IDLE} )</td>
<td>20 NS ( 0000 ) ( 0000 )</td>
</tr>
<tr>
<td>( 10 ) NS ( 1111 ) ( 0000 )</td>
<td>30 NS ( \text{IDLE} )</td>
<td>30 NS ( 0000 ) ( 0000 )</td>
</tr>
<tr>
<td>( 20 ) NS ( 0000 ) ( 1111 )</td>
<td>40 NS ( \text{IDLE} )</td>
<td>40 NS ( 0000 ) ( 0000 )</td>
</tr>
<tr>
<td>SLOW_SWEEP: ( 0 ) NS ( 1000 ) ( 0000 )</td>
<td>50 NS ( \text{QUICK_SWEEP} )</td>
<td>50 NS ( 0000 ) ( 1111 )</td>
</tr>
<tr>
<td>( 1 ) NS ( 0100 ) ( 0000 )</td>
<td>60 NS ( \text{QUICK_SWEEP} )</td>
<td>60 NS ( 1111 ) ( 0000 )</td>
</tr>
<tr>
<td>( 2 ) NS ( 0010 ) ( 0000 )</td>
<td>70 NS ( \text{QUICK_SWEEP} )</td>
<td>70 NS ( 0000 ) ( 1111 )</td>
</tr>
<tr>
<td>( 3 ) NS ( 0001 ) ( 0000 )</td>
<td>80 NS ( \text{SLOW_SWEEP} )</td>
<td>80 NS ( 1111 ) ( 1111 )</td>
</tr>
<tr>
<td>( 4 ) NS ( 0000 ) ( 1000 )</td>
<td>( 90 ) NS ( \text{SLOW_SWEEP} )</td>
<td>90 NS ( 0000 ) ( 1111 )</td>
</tr>
<tr>
<td>( 5 ) NS ( 0000 ) ( 0100 )</td>
<td>( 100 ) NS ( \text{SLOW_SWEEP} )</td>
<td>100 NS ( 1111 ) ( 0000 )</td>
</tr>
<tr>
<td>( 6 ) NS ( 0000 ) ( 0010 )</td>
<td>( 110 ) NS</td>
<td>110 NS ( 1111 ) ( 1111 )</td>
</tr>
<tr>
<td>( 7 ) NS ( 0000 ) ( 0001 )</td>
<td>( 111 ) NS ( \text{SLOW_SWEEP} )</td>
<td>111 NS ( 0000 ) ( 1000 )</td>
</tr>
<tr>
<td></td>
<td>( 112 ) NS ( \text{SLOW_SWEEP} )</td>
<td>112 NS ( 0100 ) ( 0000 )</td>
</tr>
<tr>
<td></td>
<td>( 113 ) NS ( \text{SLOW_SWEEP} )</td>
<td>113 NS ( 0010 ) ( 0000 )</td>
</tr>
<tr>
<td></td>
<td>( 114 ) NS ( \text{SLOW_SWEEP} )</td>
<td>114 NS ( 0001 ) ( 0000 )</td>
</tr>
<tr>
<td></td>
<td>( 115 ) NS ( \text{SLOW_SWEEP} )</td>
<td>115 NS ( 0000 ) ( 1000 )</td>
</tr>
<tr>
<td></td>
<td>( 116 ) NS ( \text{SLOW_SWEEP} )</td>
<td>116 NS ( 0000 ) ( 0100 )</td>
</tr>
<tr>
<td></td>
<td>( 117 ) NS ( \text{SLOW_SWEEP} )</td>
<td>117 NS ( 0000 ) ( 0010 )</td>
</tr>
<tr>
<td></td>
<td>( 118 ) NS ( \text{SLOW_SWEEP} )</td>
<td>118 NS ( 0000 ) ( 0001 )</td>
</tr>
</tbody>
</table>

Partitioning the vector writing process into "writing events" and "sequencing events" makes sense for at least two reasons. First, it decomposes the problem into two subproblems, individually easier to address. Second, the efforts put into writing vector files using a two phase approach are reusable to some extend. If we are content with the high level event sequencing, but feel the need to alter event definitions, or visa versa, we can reuse the former and change the latter. By virtue of their confined, structured nature, changes are much more comprehensible and can be done in a shorter time span than in an unstructured approach. If, like in Example 2.2, events are referenced a number of times, the definitions are reused within an event sequence. The compiler saves us the burden of manually duplicating these.
The architecture from Example 2.1 excels in simplicity. This is actually a good quality. It results in a low simulation overhead and reduces the risk of errors in the architecture itself. When we want to pursue a structured approach, another architecture might however be better suited for the job. When the event concept is introduced at the architectural level, repeated occurrences of lengthy vector patterns can be avoided, reducing the need for large VHDL data structures. Another drawback of the architecture resides in the static nature of test execution; all needed stimuli must be foreseen, scheduled and programmed offline prior to test execution. In some cases neither the exact moment, on which to apply the stimuli, nor even the choice of stimuli themselves can be determined statically. For testing of complex designs the non-interactive architecture is sometimes inadequate.

2.1.3 The IPA test bench

The advantages of a structured approach and the need for interactive testing have been recognized by the IPA design team. They have equipped the basic vector processor with an instruction set for flow control, and have extended the simulation environment to allow for evaluation of simple boolean expressions (referring to any design variable or signal) during simulation. In this way control flow can be influenced by simulation results. A general overview on the test bench, and its relation with tools support, is depicted in Figure 2.5. A vector processor recognizes the following flow control codes:

<table>
<thead>
<tr>
<th>line index</th>
<th>instruction mnemonic</th>
<th>first operand</th>
<th>second operand</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;index&gt;</td>
<td>UG</td>
<td>&lt;target_index&gt;</td>
<td></td>
<td>unconditional GOTO</td>
</tr>
<tr>
<td>&lt;index&gt;</td>
<td>CG</td>
<td>&lt;target_index&gt;</td>
<td></td>
<td>conditional GOTO</td>
</tr>
<tr>
<td>&lt;index&gt;</td>
<td>UC</td>
<td>&lt;target_index&gt;</td>
<td>&lt;condition&gt;</td>
<td>unconditional CALL</td>
</tr>
<tr>
<td>&lt;index&gt;</td>
<td>CC</td>
<td>&lt;target_index&gt;</td>
<td>&lt;condition&gt;</td>
<td>conditional CALL</td>
</tr>
<tr>
<td>&lt;index&gt;</td>
<td>RT</td>
<td></td>
<td></td>
<td>return</td>
</tr>
<tr>
<td>&lt;index&gt;</td>
<td>RP</td>
<td>&lt;repeat_count&gt;</td>
<td></td>
<td>repeat sequence</td>
</tr>
<tr>
<td>&lt;index&gt;</td>
<td>ER</td>
<td>&lt;data&gt;</td>
<td></td>
<td>end repeat sequence</td>
</tr>
<tr>
<td>&lt;index&gt;</td>
<td>--</td>
<td></td>
<td></td>
<td>apply vector data to EUT</td>
</tr>
</tbody>
</table>

The IPA solution uses dedicated, simple microcontroller like, processes to stimulate the entity under test. The vector files do not contain explicit time marks. Instead, each stimulator is tied to the clock governing the interface it is connected to. Multiple stimulus processes can work in parallel to drive any number of interfaces, possibly governed by different clocks. A stimulator applies a vector every clock cycle, for the duration of the simulation run.

To understand how the two pass compiler works you need to know what AWK is and what it does. AWK is a pattern directed scanning and processing language for the UNIX environment. It scans a given input file (supplied as a single file or build by appending a sequence of smaller files) for patterns specified in a (sequence of) program file(s). With
each pattern there can be associated actions to be performed when that pattern matches against a line of the input file. These actions produce the AWK output.

In the compiler configuration used, the patterns stored in two separate files. The AWK program "pvg.awk" (Programmable Vector Generator) defines the following control flow keyword patterns for the input command file:

# A line starting with '#' is comment
@ <label> Defines <label> as the index of the next line of the vector file.
A <label> is an arbitrary sequence of characters of the set [A-Za-z0-9_].
GOTO <label> Insert an unconditional 'goto' line in the output file with target_index <label>. The <label> must be defined somewhere in the command file (either forward or backward).
CALL <label> Insert a conditional call vector with target_index <label>.
IF <cond> GOTO <label> Inserts a conditional 'goto' (mnemonic CC) line into the vector file. <cond> is a simple boolean expression.
IF <cond> CALL <label> Inserts a conditional call (mnemonic CC) line into the vector file.
REPEAT <count> Start of sequence to repeat <count> times. Inserts a RP line into the vector file.
ENDREP End of a sequence to repeat.
RET Inserts a return line (mnemonic RT) into the vector file.
END End of the command file. Needed to separate the first from the second instance of the input command file. The input file is supplied twice to perform a second pass of compilation in one AWK run.

Figure 2.5  An overview on IPA's test bench.

In the compiler configuration used, the patterns stored in two separate files. The AWK program "pvg.awk" (Programmable Vector Generator) defines the following control flow keyword patterns for the input command file:
These map directly onto analogues in the VHDL vector processor instruction set. The "pvg.awk" is directly related to the processor architecture and is used (without modification) in code generation for all vector processors. The AWK script defines the stimulator specific event keyword patterns and is unique for each processor. The actions associated with each event keyword, produce a sequence of data lines (mnemonic '--').

Simple boolean expression <cond> strings, from conditional CALL and GOTO input lines, are passed through, untouched, to the <condition> operand of CC or CG output lines respectively. The vector processor calls an externally linked C routine to have these evaluated. The following syntax is used for simple boolean expressions using an infix notation:

\[
\begin{align*}
<exp> & ::= <boolexp> \ [ <dop> <boolexp>] * \\
<dop> & ::= \ ' & ' | \ ' |
<boolexp> & ::= <boolsig> | \\
 & \ '<sig> ' = ' <value> ' | \\
 & \ '! ' <boolexp> | \\
 & \ ' ( ' <exp> ' ) '
<value> & ::= " "<vhdl_value>" " | \\
 & " "<char_value>" " | \\
<integer>
\end{align*}
\]

In which: - Any character enclosed by ` is a literal, - the `&` operator represents the boolean AND function, - the `|` operator represents the boolean OR function, - the `!' operator represents the boolean NOT function and - the `=' represents an equality test.

### 2.2 TTCN’s operational semantics

The production of abstract test cases is only a partial solution to the test problem. The test cases prescribe which test events to offer and how to interpret observed events. Note that events are of an abstract nature and refer, by name, to physical counterparts at the test interface. The physical form of such events is thus governed by the application domain and interface type.

In order to develop a test architecture that enables us to execute automatically generated TTCN test cases we will first establish the subset of TTCN we wish to support. Then we will elaborate on the operational semantics of this subset in a VHDL simulation context.

#### 2.2.1 Overview on TTCN

TTCN [ISO89; part 3] is designed to be an universal language for specifying the abstract behavior of test systems and the protocol entity under test. It is independent of test methods, layers and protocols and reflects the abstract testing methodology defined in ISO-9646. A good introduction to TTCN can be found in [Kni93; pag. 89-150] or [PM92].
Abstract testing in VHDL

A TTCN test suite specifies a number of interactive abstract tests (test cases) in a dynamic behavior tree format. A behavior tree is denoted in a tabular form using indented behavior lines. The indentation level relates to the progression of time in test case execution, and a number of subsequent behavior on the same indentation level constitutes a set of alternatives. Each behavior line is made up out of five fields: line label, TTCN statement, constraints reference, verdict and comment.

The test behavior is denoted with TTCN statements. There are four kinds of statements: abstract test events send by the tester to the EUT, abstract test events received by the tester from the EUT, pseudo-events and constructs. The following statement syntax applies:

\[
\text{<Statement> ::= <Event> | <PseudoEvent> | <Construct>}
\]

\[
\text{<Event> ::= <Send> | <ImplicitSend> | <Receive> | <Otherwise> | <Timeout>}
\]

\[
\text{<PseudoEvent> ::= <TTCNExpression> | <TimerOperation>}
\]

\[
\text{<Construct> ::= <Attach> | <GoTo> | <Repeat>}
\]

An associated constraint dictates exact or allowable values for tester initiated send events and acceptable parameters for IUT initiated receive events. The line label is only used in conjunction with the GoTo construct and statements that complete a test sequence will have an associated verdict.

The suite's test cases are contained in the Test Case Library. At any point in a test behavior tree, one can attach local behavior trees or common test steps from the Test Step Library. Furthermore, each test case can have an associated default behavior tree contained in the Defaults Library.

Execution of TTCN test cases implies evaluation of TTCN statements. The failure or success of statement evaluation influences control flow in test case execution. The success of RECEIVE statements is determined by the behavior displayed by the EUT. Oppositely, it is often the behavior not displayed by the EUT that determines the success of TIMEOUT statements. Test cases are by this virtue of an interactive nature at the abstract level.

2.2.2 RNL TTCN subset

The PTT Conformance Kit does not use the full scope of the TTCN language for its dynamic test cases. The TTCN subset used can be characterized as follows:

* PDUs/ASPs without associated type definitions (fieldless/unparameterized resp.)
* no use of constraint references
* no test suite or test case variables
* no (assignments) nor [qualifiers] used
* no TTCN expressions used
* no GOTO constructs (so no need for labelling of behavior lines)
* no REPEAT constructs
Abstract testing in VHDL

* test case timers are used
* only TimerOperations START and CANCEL are used (no READTIMER)
* only the default timer durations from the timer declarations are used. No overriding of the default duration through START <new_duration> statements.
* DEFAULT test case behavior is used
* tree attachments are used (both local trees and test steps)
* special OTHERWISE event is used
* no temporary verdicts
* final test case verdict either PASS or FAIL (verdict INCONCLUSIVE not used)

Structural properties regarding the KIT's use of this subset are:

* Exactly two timers for each test suite produced. Timer no_output_timer (fixed at a duration of 10 seconds) is used to test outputless transitions of the EFSM model. All behavior observed (trapped by the test case default) between timer START and timer ?TIMEOUT is regarded illegal, and leads to a test case FAIL verdict. Timer testcase_timer (fixed at a duration of 200 seconds) is used to place an absolute upper bound on the time spent in executing a test case. Rather than testing exact, normative operation time the timer is intended to detect deadlocks in tester interaction. It must at least accommodate the maximum number $m$ of consecutive "outputless transition" occurrences in any test case behavior tree. A test case error will occur if $m \times \text{duration(no_output_timer)} > \text{duration(testcase_timer)}$.

* The defaults library always contains precisely one default behavior tree for trapping erroneous IUT behavior. All statement lines assign a FAIL verdict on successful evaluation. The default tree has only one level that contains an ?OTHERWISE trap for each PCO and a ?TIMEOUT testcase_timer line.

2.2.3 Operational semantics of the KIT's TTCN subset

We will elaborate on the operational semantics of this subset. The definition of these operational semantics in [ISO89; part 3: Annex B, pag. 118-134] follows an informal, but precise, two phase approach. In the first phase concrete TTCN texts are mapped onto a simplified TTCN structure, in three subsequent steps:

a) appending of Default behaviors;

b) removal of REPEAT Constructs;

c) expansion of attached trees.

These transformations are described by means of algorithms in a pseudo programming language. In addition there is an explanatory, natural language description provided.

The actual appending of Defaults is done by adding the construct "+DefaultReference" to the end of each set of alternatives in the test case. Appending of defaults can therefore be interpreted as a special case of (implicit) tree attachment. In the following we will first
address the specifics of tree attachment which we then use to clarify the appending of defaults.

2.2.3.1 Tree attachment

Behavior trees may be attached to other behavior trees using the ATTACH construct. Attached trees are expanded by replacing the attach construct "+AttachedTree" with the tree AttachedTree, and subsequently if there was behavior specified following, and indented from, the ATTACH construct to insert this behavior after, and indented from, each leaf in the attached tree.

In effect, tree attachment is a notational macro facility that somewhat resembles the classical subroutine call, but shows two fundamental deviations. When using subroutine calls (in any programming language/environment supporting these) it would be considered improper, or dirty, to call subroutines from which we never expect to return. Nesting such dirty calls would be outright disastrous because of stack pollution/overflow. Clearly a subroutine is not meant to be used like that. An attached tree however, might return to its calling tree, but this is not mandatory. Successful evaluation of a behavior line with a verdict assignment, terminates test case execution irrespective of the location in the calling hierarchy. Even if test execution does return from an attached tree it can do so in one of two ways: at the current level of indentation or at the next level of indentation. In the following we will clarify these peculiarities of attachment with examples.

In Example 2.3 the subtree STEP is attached to the second level of indentation of the calling tree TOP_TREE. The set of alternatives at the second level of the equivalent tree consists therefore of {?A1, ?B1, ?B2, ?A3}. These have to be considered subsequently in a top to bottom order. When the first alternative ?A1 doesn't match against the current PCO snapshot the tree STEP is called. When alternatives ?B1 and ?B2 don't match either, execution of STEP must return to TOP_TREE at the same level of indentation it was called from (i.e. the current level) and continue with matching against ?A3. If this match fails as well a new snapshot of the inbound PCO queues, shall be taken and processing repeats, starting with ?A1.

Example 2.3 Execution of attachment returns to the same level of indentation.

```
TOP_TREE
  !A
  ?A1
  ?A11
+STEP
  ?A3

STEP
  ?B1
  ![B11
  ?B2

TOP_TREE
  !A
  ?A1
  ?A11
  ?B1
  ![B11
  ?B2
  ?A3
```

and is equivalent to
In Example 2.4 the subtree STEP is attached to the second level of indentation of the calling tree TOP_TREE again, but now has subsequent behavior !B. In the equivalent tree this subsequent behavior is appended to all leaves of the attached tree. If one of the alternatives at the root level of the attached tree matches (say ?D1), processing proceeds with subtree behavior subsequent to the matched event (?D1). Processing continues within the called tree until a verdict assignment is encountered or a leaf is reached. In the latter case execution must return to TOP_TREE at the level of indentation subsequent to the one it was called from (i.e. at the next level) and offer !B.

Example 2.4 Execution of attachment returns to the next level of indentation.

2.2.3.2 Appending of default behavior

Test cases and test steps have their own DefaultReference referring to a behavior tree from the DefaultsLibrary. For the root tree of a test case, appending is done, pretty much straightforward. The DefaultReference is attached after the bottommost alternative of each set of alternatives at all levels of indentation.

For local trees of test cases and/or test steps and test steps themselves appending is done (in the same fashion) by attaching the DefaultReference to all but the first level of indentation. This is necessary to prevent multiple defaults from being appended to a given level of indentation of the equivalent expanded behavior tree.

Let us clarify this with an example in which we append to all levels, irrespective of the tree class:
2.2.3.3 Test case execution

Natural language description adapted from [ISO89; par. B.5.3.2]:

Step 1: Test Case evaluation begins at the leftmost level of indentation of the tree.

Step 2: A snapshot of the incoming PCO queue(s) and timeout list is taken. Note: the act of making a snapshot does not remove an event from the PCO. Within the behavior line at the current level of alternatives, consider the first one specified.

Step 3: Evaluate the TTCN statement on the current behavior line, based on what is specified in the leftmost element, except that those TTCN statements that begin with a PCO identifier, are evaluated based on follows the PCO identifier.

The evaluation of each type of TTCN statement is specified in the operational semantics for that TTCN statement type. Timer operations (START & CANCEL) and SEND events are considered always to evaluate to a success (no qualifiers used). The RECEIVE and TIMEOUT events may or may not evaluate to a success depending on the first event on the relevant PCO queue or presence of the relevant TimerId in the timeout list respectively.

Step 4: If a TTCN statement evaluates to a successful match, then go to step 5. Otherwise if there are more alternatives in the current set of alternatives, consider the next behavior line in the set of alternatives and goto step 3.

If there are no more alternatives and yet all PCO queues relevant to this set of alternatives contain at least one event, and all timers from the TIMOUT statements in the current set of alternatives are in the timeout list, then there is a test case error and test case execution shall be stopped indicating test case error. Note: this is a test case error because under these conditions none of the alternatives can ever match.
24 Abstract testing in VHDL

In all other cases take a new snapshot of the PCO queue(s) and timeout list and consider again the TTCN statement on the first behavior line of the current set of alternatives; then go to step 3.

Step 5: If a leaf node has been reached, go to step 6. Otherwise, consider a new set of alternatives for evaluation at the next level of indentation. (Immediately following, and indented a single level from, the TTCN statement that has just matched.) Go to step 2.

Step 6: A leaf node of the tree is reached. Since temporary verdict assignments are not supported, all leaf behavior lines must assign the end verdict (if not a test case error must be raised). Process the verdict by recording it in the conformance log.

2.3 Designing a VHDL test bench for execution of TTCN test suites

2.3.1 A graphical notation for VHDL structure

The structural hierarchy of VHDL models is not that easy to oversee. I have looked into literature but, to my knowledge, no standardized graphical notation has been agreed upon. To clarify my designs I have used the following notation:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Component instance. Naming convention: <code>&lt;component_name&gt;: &lt;entity&gt;(&lt;architecture&gt;)</code></td>
</tr>
<tr>
<td></td>
<td>Process or concurrent signal assignment.</td>
</tr>
<tr>
<td></td>
<td>Component port mode IN.</td>
</tr>
<tr>
<td></td>
<td>Component port mode OUT.</td>
</tr>
<tr>
<td></td>
<td>Component port mode INOUT.</td>
</tr>
<tr>
<td></td>
<td>Signal read by process.</td>
</tr>
<tr>
<td></td>
<td>Signal driven by process.</td>
</tr>
<tr>
<td></td>
<td>Signal driven &amp; read by process.</td>
</tr>
<tr>
<td></td>
<td>Signal in sensitivity list and read &amp; driven by process.</td>
</tr>
</tbody>
</table>

Signal pathways interconnect behavioral constructs (processes) that operate in parallel. Connectors symbols always connect to precisely one signal only. Processes can assign or read signals that are declared at the same, or a higher level in the structural hierarchy. Signal values can propagate to, or be updated from, lower levels in the hierarchy through component ports. Names along side connection lines refer to signal identifiers at their declarative level and to local port references at subsequent lower levels.
2.3.2 Split up of the test bench

We will elaborate on automated execution of TTCN test cases in a VHDL simulation environment. The problem revolves around implementing TTCN's operational semantics within the limitations of a VHDL test bench.

We have chosen to limit our "first try" efforts and aim at developing a test execution environment that accepts the Conformance Kit subset only. In order to facilitate enrichment of this subset in a later stage we decide not to exploit the structural properties of the Kit's suites. We can thus use manually written (or generated by another tool) TTCN test suites without these structural properties. We will split up the problem twice.

The first (functional) split stems from the observation that TTCN statements are dividable into two groups: 1) statements which execution effects extend beyond the tester boundary and addresses the EUT (SEND and RECEIVE events), and 2) statements which execution effects are confined to the tester (the rest). We plan to split up the test bench accordingly. All statements will be processed by a "test coordination" entity, but execution of the complex SEND and RECEIVE events is dispatched to separate "stimulator" and "observer" entities.

We will make a second (hard-/software) split resulting in a test suite independent VHDL execution environment and a tools environment that supplies test suite dependant executable code. This executable code has to be downloaded (at the start of a test run) in the local processor memories of the observers, stimulators and the supervisor. We project a VHDL test bench that resembles the one depicted in Figure 2.1.

![Conceptual view on a VHDL - TTCN test bench.](image)

**Figure 2.1** Conceptual view on a VHDL - TTCN test bench.
Abstract testing in VHDL

In the next chapter will refine and implement this conceptual test bench. We will strive for maximum modularity and genericity to render the bench reusable for other test projects.
This chapter describes the development and implementation of a toolbox of generic components that allows easy and fast instancing of a VHDL test bench for execution of TTCN test cases. The TTCN test cases are coded in an executable form. The coded test cases, together with the information needed to de-abstract test events (down to the detail level of the physical test interface) are read from file at the start of simulation. The test results produced during simulation are written to the conformance log file.

We follow a top down approach, zooming in on detail as we encounter it, to describe the test supervisor, observer and stimulator. The next chapter addresses the instancing of a test bench for the P1394 LinkCore using these components.

3.1 Interconnection of the test bench components

3.1.1 Overview

We have to interconnect the supervisor with the observers and stimulators to enable the supervisor to transport event identifiers (data), and control the test architecture (control). We need a stimulator/observer pair for each EDT interface (TTCN PCO). In the P1394 case we have two test interfaces but in general the number of test interfaces cannot be presumed fixed.

Two approaches are thinkable: either we use congruent, private sets of signal pathways to interconnect each observer and each stimulator to the supervisor, or we can time multiplex the needed connection on a shared bus. The latter could be implemented in VHDL by using resolved guarded signals. Such a signal can be connected to the drivers of a number of contributing processes that can be turned off; either explicitly (a process assigns NULL) or implicitly (concurrent signal assignment with a guard that evaluates to FALSE). The bussed approach has great advantages over the dedicated parallel connection option:

+ Supports modularization of a generic "supervisor" test bench component. Parallel connection to a supervisor entity would require a variable number of connecting ports which is impossible; the number of entity ports must be static.
+ One set of signals, irrespective of the test configuration used. Easy to instance.
Test bench components

+ The task of connecting the observers/stimulators is less error prone and requires far less effort because we can use the same port map for all instanced observers and all stimulators respectively.
+ The concept can be extended to connect the supervisor's internal timers to the bus. Furthermore, the timers can indeed be made internal because the task of connecting to the bus is so transparent that it can be done automatically using a generate statement. Internal timers have the advantage that they need not be instanced separately (they are instanced together with the supervisor) and they help to keep the toplevel design tidy.
+ A PCO is identified by an address on the bus rather than by an identifier of the signal connected to it. A transparent mapping of TTCN identifiers onto PCO addresses could be easily automated by a compiler.

The performance loss usually associated with the use of busses hardly applies here. The bus would reside under total control of the supervisor. The TTCN statements that need access to observers, stimulators and timers specifically address only one such entity at a time. Sequential execution of such statements makes a bus ideal for the job; at worst the bus's capacity is fully utilized.

We have therefore chosen a shared bus for interconnection of the test bench components.

3.1.2 Test bus requirements

We need to provide a number of "mechanisms" on our bus:

1) The supervisor must be able to emit commands on the bus addressed to one of the test bench components from which it expects a response. The addressed component must be sensitive to, and be capable to, discern identical consecutive commands.
2) A command may be accompanied with (transmit) data. For instance a 'write' to a stimulator queue tail must transfer a stimulus event identifier over the bus.
3) The supervisor must suspend after issuing a command so that the signals can be updated. The addressed component must be able to respond to a supervisor command.
4) A component response may be accompanied with data. For instance the response to a 'read' from an observer queue head must transfer the observable event identifier read over the bus.
5) We must provide an addressing scheme that maps easily onto the test suite's PCO identifiers and timer references.
3.1.3 Choices & Implementation

We have implemented the interconnecting bus using the seven following signals:

```plaintext
TYPE control_type IS (cancel, clear, nice, read, release, reset, start, status, write);
TYPE enable_type IS (en_observer, en_stimulator, en_timer);

TYPE int_vector IS ARRAY (integer RANGE <>) OF integer;
FUNCTION resolve_int (input: int_vector) RETURN integer;
SUBTYPE resolved_int IS resolve_int integer;

TYPE bool_vector IS ARRAY (integer RANGE <>) OF boolean;
FUNCTION resolve_bool (input: bool_vector) RETURN boolean;
SUBTYPE resolved_bool IS resolve_bool boolean;

SIGNAL address : integer;
SIGNAL command : boolean;
SIGNAL control : control_type;
SIGNAL data : resolved_int BUS;
SIGNAL enable : enable_type;
SIGNAL new_event : resolved_bool;
SIGNAL response : resolved_bool BUS;
```

The supervisor's control commands use up to five signals (enable, data, control, command and address). The command signal toggles each time a new command is issued. Any event on command reactivates the suspended test bench components which sample the command on control. Confirmed commands are addressed to one enabled component only, and may have associated data. The guarded data signal (type BUS) uses "one driver only" resolution (see page 69, appendix A.1) for bi-directional integer data transport between the supervisor and the test bench components. The supervisor suspends after issuing a command. If the command is expected to result in response data then the driver of data is disconnected prior to suspension to prevent the bus from being overdriven.

Confirmed bus commands result in a toggle of the result signal (which reactivates the supervisor again) indicating that the command has completed and expected response data (if any) is now valid. The guarded response signal (type BUS) is driven by one component at a time. If the command did indeed resulted in response data then the supervisor will issue a confirmed 'release' command to explicitly disconnect the components data driver, thus avoiding an overdriven data bus on the next command with associated data.

Unconfirmed commands do not result in responses to reactivate the supervisor. The supervisor therefore suspends for fixed time to allow the command to take affect. Because it has to accommodate a variable length sequence of δ-cycli only (totalling to zero simulation time) this suspension period can be chosen arbitrarily small. However a lower bound is the simulator resolution (smallest simulation time step possible) and it must be chosen well below the shortest governing clock period.

TTCN's snapshot semantics can be implemented in a number of ways. One way is to let the supervisor sample a given set of alternatives every cycle of the highest frequency clock. This guarantees that any observable event will seen in the same cycle it was parsed. This is somewhat inefficient because then the queue heads are needlessly often sampled. Another approach is to reactivate the supervisor from suspension when a new queue head or timeout is incident.
30 Test bench components

We have chosen to implement the latter approach using the boolean \textit{new\_event} signal. The supervisor is sensitive to the rising edge (\text{FALSE} \rightarrow \text{TRUE}) of the \textit{new\_event} signal that uses "wired or" resolution (see page 69, appendix A.1). Multiple components may contribute to the reactivation of the supervisor at the same time.

The three test bench components together recognize the following bus commands:

<table>
<thead>
<tr>
<th>Component</th>
<th>Command Type</th>
<th>Bus Command</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stimulator</td>
<td>Confirmed</td>
<td>write</td>
<td>Writes event_id on \textit{data} to enabled stimulator (enable &lt;= en_stimulator) at \textit{address}.</td>
</tr>
<tr>
<td></td>
<td>Unconfirmed</td>
<td>reset</td>
<td>Clears stimulator queue and halts processing current event.</td>
</tr>
<tr>
<td>Observer</td>
<td>Confirmed</td>
<td>clear</td>
<td>Clears queue head of indicated observer (enable &lt;= en_observer) at \textit{address}.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>read</td>
<td>Non destructive read of indicated queue head. The \textit{addressed} observer toggles response and drives \textit{data} with the event_id read (0 =&gt; empty queue).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>release</td>
<td>Disconnects the observer's \textit{data} driver (after a 'read').</td>
</tr>
<tr>
<td></td>
<td>Unconfirmed</td>
<td>nice</td>
<td>Clears observer's \textit{new_event} driver. Used before suspension of the supervisor.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>reset</td>
<td>Clears observer's queue and resets parser.</td>
</tr>
<tr>
<td>Timer</td>
<td>Confirmed</td>
<td>start</td>
<td>Starts designated timer (enable &lt;= en_timer) at \textit{address}. A timer must be written before it can be started.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cancel</td>
<td>Stops the designated timer.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>status</td>
<td>Checks whether the timer has timed out. Status read on \textit{data} (1 =&gt; expired, 0 =&gt; stopped or running).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>write</td>
<td>Programs the default duration of the timer through \textit{data} in an integer number of NS units.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>release</td>
<td>Disconnects the timer's \textit{data} driver. (after a 'status')</td>
</tr>
<tr>
<td></td>
<td>Unconfirmed</td>
<td>nice</td>
<td>Clears timer's \textit{new_event} driver. Used before suspension of the supervisor.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>reset</td>
<td>Cancels all running timers.</td>
</tr>
</tbody>
</table>

3.2 Supervisor test coordination

3.2.1 Overview

The supervisor is responsible for execution of a suite of TTCN test cases. Each test case is defined through a number of TTCN statements and each statement concluding a test sequence (multiple test sequences possible per case) must have an associated verdict. The KIT\'s statement subset (see par. 2.2.2) must be supported, including TimerOperations and Timeouts.
Because of their complexity, we have already split off the execution of TTCN's send and receive statements. Now we devise a way to split off the task of executing the TimerOperations and Timeouts. The execution of all useable (pseudo-) events is then isolated into dedicated components.

It remains to devise a 'ttcn_engine' that executes a test program with machine instruction analogues for each of the TTCN statements. Additional instructions are introduced for flow control and conformance logging.

3.2.2 Timer array

We intend to instance a parameterizable timer component for every declared timer in the test suite. These timers are connected to the TTCN engine with the same test bus used for connecting the stimulators/observers. The timers are addressed on the same address signal used to transmit integer PCO identifiers and must be enabled with the 'en_timer' value on the enable bus. See paragraph 3.3 for a detailed timer description.

We have roughly two options for the location of timer instancing: within the 'supervisor' entity itself or in the toplevel 'tesLbench' entity. There is the possibility to do the first by using a simple automatically generateable addressing scheme for a parameterizable number of timer instances (already mentioned in par. 3.1.1). Such an approach would have the advantage that there is no need to separately instance each timer (they are instanced together with the supervisor), it helps to keep the toplevel design tidy and the connection is less error prone.

We have therefore decided to encapsulate the timers together with the 'ttcn_engine' in the supervisor entity. The actual value of the generic 'nr_of_timers' is used (on supervisor instancing) by a generate statement to instance that amount of timers. The timers are parameterized over their own bus address using a simple linear addressing scheme (addresses: 1 through nr_of_timers).

3.2.3 The TTCN engine

The following requirements are now imposed on the TTCN engine:

1) It must interpret (pseudo-) event related instructions and dispatch them to the appropriate test bench component using a sequential communication scheme. The required number of communications varies per instruction. The available bus commands are listed in par. 3.1.3.
2) Successful (pseudo-) events shall be written to the conformance log together with their absolute time of occurrence.
3) A special instruction is needed to log the test case verdict.
4) The timers are unaccessibly embedded inside the supervisor and their default duration has to be programmed during simulation using bus commands. Since the timer default durations are defined within the TTCN suite we need a dedicated instruction to do this.
32 Test bench components

5) The test suite's test cases are to be executed consecutively. In order to keep the log comprehensible, an instruction for logging the test case start (and identifier) is needed. Also, an instruction(s) is required for resetting the components to their initial state (deleting any info from the previous test) before starting a new test.

6) A direct code representant of the TTCN attach construct would be welcome. Expansion of attached trees would result in large code files.

In the following subsections we elaborate towards defining the engines instruction set.

3.2.3.1 Two level stack

According to the operational semantics of TTCN, attached trees have to be expanded prior to test case execution. This has two major downsides though. The first downside resides in the blowup of behavior tree descriptions that this expansion causes. The reuse of common parts of trees is lost in expansion by substituting a copy of the attached subtree at the location of attachment. The second downside is that the act of expansion is an extra operation for the compiler.

One could imagine an executable TTCN form in which the appending and attaching is done on the fly, during execution. Common, reusable pieces of test behavior (test steps, local trees and defaults) are denoted only once. As a consequence the architecture needs to be equipped with facilities for this real time appending, complicating its design. The TTCN compiler, on the other hand, is relieved of this task and its design can therefore be simpler. In effect the task of cohering to required operational semantics is a zero sum game. The gain of simplifying the design of the TTCN compiler has to be paid in the more complex design of the machine that executes it, and vice versa.

We would like to use a stack based subroutine mechanism (with call and return instructions) to implement the attachments directly at the VHDL level. However, the semantics of the attachment (see par. 2.2.3.1) do not allow such a straightforward approach. The success or failure of evaluating an attached behavior tree (at the root level) determines the succeeding test case behavior.

If all root alternatives of an attached (called) tree have failed, the statement directly below the attachment in the same (current) set of alternatives of the originating (calling) tree must be evaluated. If the attachment is the bottommost alternative of that set then a new snapshot shall be taken before the first statement in the current set of alternatives is evaluated again.

If one of the root level alternatives, of the called tree, did evaluate successfully than test execution continues within that tree until one of its leaves is reached. It is at that point that execution must return to the calling tree at the first statement in the subsequent (next) set of alternatives.

If we know the absolute code locations of both the next alternative of the current set and the first alternative of the next set at 'call' time then we can push both these potential return addresses onto the stack. In the following, we will refer to these addresses as
Test bench components

<first_on_next_level> and <next_on_current_level>. If execution reaches a leaf of the called tree than a 'return_next' instruction pops <first_on_next_level> off the stack and jumps to that address. If execution passes the bottom alternative of the root of the called tree than a 'return_current' instruction pops <next_on_current_level> off the stack and jumps to that address. The unused address on the stack is dumped by the return instructions.

A call statement of this type needs two operands. One <target_address> operand to indicate the address of the called code and one operand to indicate the <first_on_next_level> or <next_on_current_level> address. We have to make a choice which coded behavior sequence will be directly below the <index> of the call instruction. This can be either the next level behavior sequence or the current level behavior sequence. To facilitate the construction of a TTCN compiler we have opted to store the next level behavior sequence directly below the call statement (the order of TTCN statements remains unchanged in the compiled code). The call instruction has thus a second <next_on_current_level> operand. Both return instructions are without operands.

We have implemented a subroutine mechanism for direct support of TTCN's attach construct on this basis.

3.2.3.2 Conformance Log

The conformance log has been subjected to limited standardization only. In [ISO89, part 3; appendix B] it is stated (adapted to our TTCN subset) that the success of the following (pseudo-)events must be logged (references apply to paragraphs of the standard):

[5.5.2 - step 8]: SEND events; record the
- PCO address at which the SEND occurred,
- the ASP-PDU identifier that was sent.

[5.6.2 - step 10]: RECEIVE events; record the
- PCO address at which the RECEIVE occurred,
- the ASP-PDU identifier that was received.

[5.7.2 - step 7]: OTHERWISE events; record the
- PCO address at which the OTHERWISE occurred,
- the ASP-PDU identifier that was received.

[5.8.2 - step 10]: TIMEOUT events; record the
- name of the timer that expired.

In addition to the that the following information must be recorded with every entry in the conformance log:

- the timer operations performed (if any),
- the verdict associated with the event line (if any),
- time stamp.

To produce better readable results we additionally log the test case identifier at the test case start and at its conclusion.
3.2.3.3 Zero time execution

The operational semantics of TTCN require that we freeze the simulation (make a snapshot of the queue heads) to evaluate a set of alternatives in a top to bottom order. This evaluation amounts to execution of a piece of TTCN code of which the length cannot be statically determined (i.e. varies from set to set). The execution of the event related instructions results in a variable number, of communications over the test bus. Each of these communications takes only a \( \delta \)-period of simulation time (see par. 3.1.3 also), together totaling to zero simulation time. The execution of the instructions without communication can be done sequentially within one simulation cycle.

In order to let any simulation time pass at all (which is a necessity to conduct tests against the EUT), we need a mechanism that suspends and reactivates the TTCN engine in a controlled way. For this reason we have introduced the 'wait' instruction. This instruction is used to terminate each set of alternatives. After evaluation of all alternatives the TTCN engines is suspended in order to enable the rest of the test bench to make a relevant snapshot (i.e. one that shows changes in comparison with the former). The system indicates that it has taken such a snapshot by assigning TRUE to the signal new_event.

3.2.3.4 Instruction set

To provide easy input to the VHDL environment we need an uniform input line format. The instruction opcodes are abbreviated with two character mnemonics to simplify the process of reading from file. The use of the same operand type for every first, second and third operand makes program storage in an array of instruction records possible.

The total TTCN engine's instruction set consists of 14 instructions that can be divided into the following categories:

- Flow Control: 'call', 'return_current', 'return_next' and 'goto'.
- I/O event related: 'in', 'out' and 'wait'.
- Conformance logging: 'log' and 'verdict'.
- Timer handling: 'declare_timer', 'start_timer', 'timeout' and 'cancel'.
- Others: 'reset'.

Each line of code starts with its integer line <index> that is used for addressing. The line index is followed by the instruction mnemonic and one to three operand fields. The line format of all 14 instructions is given in the following table:

<table>
<thead>
<tr>
<th>Line format supervisor (ttcn_engine) code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
</tr>
<tr>
<td>&lt;index&gt;</td>
</tr>
<tr>
<td>Index</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>&lt;index&gt;</td>
</tr>
<tr>
<td>&lt;index&gt;</td>
</tr>
<tr>
<td>&lt;index&gt;</td>
</tr>
<tr>
<td>&lt;index&gt;</td>
</tr>
<tr>
<td>&lt;index&gt;</td>
</tr>
<tr>
<td>&lt;index&gt;</td>
</tr>
<tr>
<td>&lt;index&gt;</td>
</tr>
<tr>
<td>&lt;index&gt;</td>
</tr>
<tr>
<td>&lt;index&gt;</td>
</tr>
<tr>
<td>&lt;index&gt;</td>
</tr>
<tr>
<td>&lt;index&gt;</td>
</tr>
<tr>
<td>&lt;index&gt;</td>
</tr>
<tr>
<td>&lt;index&gt;</td>
</tr>
</tbody>
</table>
3.2.4 Choices & Implementation

3.2.4.1 Supervisor VHDL source

A structural view on an example supervisor instance with two internal timers is depicted in Figure 3.1. The supervisor VHDL code can be found in appendix A.6 on page 88.

The supervisor is parameterizable over the number of timer instances and the name & location of the input TTCN code file and the output conformance log file. All seven supervisor ports are of type INOUT because any timer-supervisor communication passes through these ports. One party writes (OUT) to a given toplevel signal through the connected port and the receiving party reads (IN) the toplevel signal back through it.
The ttcn_engine uses a dedicated "read_instructions" subprogram to read the TTCN code into the instruction array \textit{ia(i)}. It maps the two character mnemonics onto an enumerated \textit{opcode_type} for higher simulation performance. Instructions with less than three operands leave some of the array space unused. The integer return index stack is implemented using a fixed size \textit{stack} array. The machine state is captured in the program counter \textit{pc} and stack pointer \textit{sp}.

The processor is implemented in a single never ending loop that houses a big case statement for decoding and execution of any instruction. The source is pretty much self explaining. The variable \textit{test\_case\_ID} is used to log the test case identifier at test case conclusion together with the test case verdict.

3.2.4.2 Tool implications

The main task of the TTCN compiler is to transform the suite's behavior tree representation, with indentation levels and top to bottom statement line ordering, into a code representant using calls, gotos, branching statements and absolute addresses. The other transformations are one-on-one mappings of TTCN statements onto their direct code equivalents, making the defaults explicit and mapping the string identifiers onto integer addresses. An elaborated compiler example can be found in appendixes B.1 through B.7.

3.3 TTCN Timer

3.3.1 Overview

We need a timer entity, directly connectable to the seven signal test bus, that accepts the timer related bus commands defined in par. 3.1.3.

The main question revolves around the choice for a timing source. Any timing can only be related to the \textit{simulation time}. We can either do this direct by using a VHDL construct that directly relates to the \textit{simulation time} (like the \textit{std.standard} \textit{NOW} function) or indirect by counting a number of cycles of some clock with a known frequency.

3.3.2 Timing source

A TTCN timer functions like a stopwatch that starts, after the start button has been pressed, to count down from a presetted time down to zero. A signal sounds when the presetted time has elapsed. The stopwatch can be restarted or stopped while counting.

A very simple VHDL process can be constructed to do this by exploiting the characteristics of a delayed signal assignment that operates in inertial mode (default). All pending transactions that are scheduled to occur after a new transaction are deleted from the signal driver. The following process uses a delayed signal assignment of 100 NS to implement a stopwatch.
This solution has a very low simulation overhead and will be extended for programming over the test bus.

3.3.3 Choices & Implementation

The design is split up into a bus_interface, a watch_dog and a timer process as is depicted in the structural overview of Figure 3.2. The VHDL source can be found in appendix A.5 on page 84. TTCN's timeout list is implemented in a distributed fashion; the ttcn_engine must poll all relevant timers to see whether they have timed out or not.

The bus interface extends supervisor control into the timer entity. It is sensitive to any event on the command signal and completes confirmed 'start', 'cancel', 'status', 'write' and 'release' commands and unconfirmed 'nice' and 'reset' commands. It programs the timer, on reception of a 'write' command, by driving the duration signal with the written data in NS units.

The watch dog is sensitive to both the rising edge (FALSE -> TRUE) of timeout and any event on nice_dog as a result of a 'nice' bus command. The timer process resembles the one in the former paragraph but is now programmable through the duration signal.
This approach is easily expandable to accommodate TTCN READTIMER statements. The bus interface can copy the simulation time at timer start (timer_started := NOW), and returns (NOW-timer_started) on the data bus on receipt of a 'read' command.

3.4 Observer event recognition

3.4.1 Overview

3.4.1.1 Observable events

The EUT ports of each test interface can be partitioned into two groups: inputs and outputs. The data conveyed through the output ports is valid on the specified edge of the governing interface clock. The idea is that each abstract, observable event corresponds to a sequence of EUT output vectors.

We would like to construct an observer entity that accumulates a sequence of output vectors (as appropriate for that interface), and identifies the abstract event that the sequence represents. Such a machine would have to fulfill two tasks:

1) **Projection**: Model events abstract from low interface detail and therefore have a (large) number of possible physical appearances. When we want to test against the abstract model only, and it suffices to project possible event appearances onto the same event identifier. We will have to implement this projection function.

2) **Recognition**: The observer must recognize a single event, from the set of events that it can expect, in the physical vector stream. It will have to make time phased decisions that gradually exclude events from the full set, until the remaining set is narrowed down to one event. The last event must then match against the event projection function; otherwise an erroneous event was detected.

The required projection function must be implementable in hardware, and yet be powerful enough to cope with all possible event variants in a concise way. We propose to define observable event projection functions using regular expressions in terms of bit vector masks with don't care values. Let us clarify this with the following example that illustrates three event definitions for an 8 bit output.

<table>
<thead>
<tr>
<th>Event_ID</th>
<th>Regular Expression</th>
<th>Char_ID</th>
<th>Bit Vector Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>fabcd</td>
<td>a</td>
<td>1000xxxx</td>
</tr>
<tr>
<td>2</td>
<td>f'ab'a</td>
<td>b</td>
<td>0100xxxx</td>
</tr>
<tr>
<td>3</td>
<td>f'cd</td>
<td>c</td>
<td>01010101</td>
</tr>
<tr>
<td></td>
<td></td>
<td>d</td>
<td>lx1x0x1x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>f</td>
<td>00000000</td>
</tr>
</tbody>
</table>
3.4.1.2 Finite automaton

The observer must be able to recognize a sequence of events in a given stream of output vectors. Figure 3.3 illustrates this process for an EUT with a 100 MHz interface clock. The first seven vectors observed match against the event_2 projection and the following six vectors match against the event_3 projection.

![Figure 3.3](image)

**Figure 3.3** Example application of EUT projection functions.

The actual event_2 and event_3 instances that the EUT emits, consist of four (abba) and two (cd) vectors respectively. Each event can be preceded by a number of "idle" vectors (f). This stems from the fact that the formal EFSM model, used to generate the TTCN tests, has no notion of time. The model events are atomic (take an infinitely small time span) in nature and are temporally ordered. In VHDL a new vector is emitted every clock cycle and the idle periods have to be absorbed. We could do this by either introducing an "idle_event": f that we then have to introduce in our model, or by absorbing the idle periods in the event definitions themselves. We have opted for the latter so that we can use the EFSM model without modification.

Each regular event expression defines the set of all possible event appearances. The union of all sets defined by all regular expressions can again be defined with a regular expression. This regular expression (defining all recognizable vector sequences) corresponds with a finite state machine and vice versa. This finite state machine is non-deterministic in general, but can always be transformed into a deterministic equivalent by introducing extra states (see [ASU86] and [HU79]). The automaton can be extended to recognize the particular event (regular expression) in a sequence of characters. This can be done deterministically and without backtracking if the regular event expressions denote prefix-free sets of character sequences. The automaton for our example is depicted in Figure 3.4.

The automaton is made complete by adding transitions to an "error" state for character sequences that can not be projected onto an event_id. The transitions that accept observable bit vectors are labelled with their respective input character and the transitions that emit the identified event to the supervisor are labelled with the [event_id].

We need to design a test suite independent VHDL observer that can execute coded automaton of this kind.
3.4.2 Parser

3.4.2.1 Instruction set

The basic task of the parser is to sample ("wait" instruction) the input and compare this vector against the present node's input characters one by one, until a match is found. If the characters define disjunct vector sets, only one match can occur. The "match" instruction combines the compare operation and the conditional jump. If none of the masks compare a transition is made to the error state ("goto" instruction). The recognized event_id is queued for the supervisor's use. The following instructions are to be implemented:

<table>
<thead>
<tr>
<th>Index</th>
<th>Opcode</th>
<th>Operands (integer)</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>'MA'</td>
<td>&lt;jump_index&gt; &lt;mask_ref&gt;</td>
<td><strong>Match</strong> checks whether the observed input vector, sampled at the selected clock edge, matches against the mask stored at location &lt;mask_ref&gt; of the mask array 'ma'. On a successful match execution will continue at &lt;jump_index&gt;; otherwise execution will continue at &lt;next_index&gt; := &lt;index&gt; + 1.</td>
</tr>
<tr>
<td></td>
<td>'GT'</td>
<td>&lt;jump_index&gt;</td>
<td><strong>Goto</strong> jumps unconditionally to index &lt;jump_index&gt;. Used to jump to an &quot;error&quot; state on detection of an unspecified event and to jump to the root after positive identification of an event.</td>
</tr>
<tr>
<td></td>
<td>'OU'</td>
<td>&lt;event_id&gt;</td>
<td><strong>Out</strong> places the integer &lt;event_id&gt; into the observer queue. Used on positive identification of any event.</td>
</tr>
<tr>
<td></td>
<td>'WT'</td>
<td>-</td>
<td><strong>Wait</strong> suspends the observer. The process will be reactivated on the next qualified sample moment.</td>
</tr>
</tbody>
</table>
3.4.2.2 Extensions to the event concept

In the course of defining observable events for the P1394 case we have encountered problems of two kinds:

1) The strict "one vector per clock" pace troubles application to a latched EUT interface. The combination of interface clock edge and latch signal determines the validity of the output vector. At best the latching scheme can be statically foreseen leading to large regular expressions with many "redundant" characters.

2) The fact that every observed vector must be accounted for in an event hampered application to an interface that was intended to be connected to a combined external "packet queue"/"scratch memory". Only validated output is intended to reach the next layer; aborted output should be ignored. This aborted output results in erroneous event recognition.

We have circumvented the first problem by introducing an "enabling qualifier" mechanism based on the external evaluation of boolean simple expressions (see par. 2.1.3). An input can thus be qualified using a boolean expression over any combination of interface signals.

We have used the same technique to introduce an "aborting qualifier" that aborts the parsing process when aborted EUT output is detected.

3.4.3 Choices & Implementation

3.4.3.1 Observer architecture

An overview on the observer architecture is presented in Figure 3.5. It shows the relation with the "parser.dat" and "mask.dat" files that are downloaded at simulation start.

The masks are stored in a separate table and are referenced by the parser code. The three value system (0, 1, x) is replaced by two binary masks. The observed input matches a mask if the following relation holds:

\[(\text{observed\_input OR mask\_x}) = (\text{mask\_1 OR mask\_x})\]

3.4.3.2 Observer VHDL source

The design is split up into a bus_interface, a watch_dog and a parser process and a cyclic_buffer component. Its structure is depicted in Figure 3.6. The VHDL source code can be found in appendix A.3 on page 73.

The observer is parameterizable over its input width, sensitive clock edge and file locations of the parser masks and parser code. Both files are read by the parser at simulation start by dedicated procedures.
The watch_dog is sensitive to any event on new_head (the queue head has changed) or nice_dog (result of a ‘nice’ bus command). The fixed size fifo cyclic_buffer is described in par. 3.6.
44 Test bench components

The observer accepts the applicable bus commands defined in par. 3.1.3. The bus interface extends supervisor control into the observer entity. It is sensitive to any event on the command signal and completes confirmed 'clear', 'read' and 'release' commands and unconfirmed 'nice' and 'reset' commands. It assigns the observer queue head to data on reception of a 'read' command, deletes the head on reception of a 'clear' command and revokes the supervisor interrupt on the new_event signal (if any) when a 'nice' command is received. A 'reset' command toggles the restart signal which resets the parser and empties the event queue.

3.4.3.3 Tool support

A LEX based tool has been developed (see [Dro96]) to generate the finite deterministic automaton and its masks. An example of an input definition file and the generated code can be found in appendix C.

3.5 Stimulator event offering

3.5.1 Overview

The EUT ports of each test interface can be partitioned into two groups: inputs and outputs. The data conveyed through the input ports is sampled by the EUT on the specified edge of the governing interface clock. The idea is that an abstract stimulus event corresponds to a sequence of EUT input vectors.

Again, numerous physical representants are possible for each abstract stimulus event. We will have to make a selection from the possible set by picking a representative physical event we would like to offer to the EUT. This selection has to made by the human operator in the test preparation phase. Once a selection has been made we need an executable description of each event that can be processed by the stimulator when so requested by the supervisor.

![Entity Under Test](image)

**Figure 3.7** Example stimulus events.

Figure 3.7 illustrates the process of event offering for an interface with a 100 MHz clock. When a stimulus id (here no. 6) is incident at the PCO queue head, the stimulator must remove it from the queue and start (at t = 0 NS) to execute the event prescription for event_6. After having offered event_6 (at t = 30 NS), the stimulator must check whether
the queue contains more events to process. If the queue is empty (like in this case) the stimulator shall offer "idle" vectors until a new event_id does arrive.

3.5.2 Vector Processor

3.5.2.1 Extension to the event concept

In the course of defining stimulus events for the P1394 case we have encountered problems with interfaces that use a handshake mechanism to control the vector flow. Rather than offering a vector each specified clock edge we may have to wait for a specific condition that indicates the appropriate moment to do so. We further need the capability to mix qualified and unqualified stimulus vectors in a single test run.

We again use the possibility to externally evaluate boolean simple expressions over the (output) interface signals. We add the possibility to use a different enabling qualifier for each separate vector.

3.5.2.2 Tables

The task of processing a numbered event prescription is so straightforward that we do not need the power of a processor instruction set. We have decided to define/store the events in a tabular format.

![Stimulator architecture](image)
3.5.3 Choices & Implementation

3.5.3.1 Architecture

Figure 3.8 depicts the supervisor architecture. Three arrays are used to store the event definitions. The stimulus vectors are all stored in a vector array. Each vector record contains a bit 'vector' and a (potential) qualifier reference 'qr' to an entry in the qualifier table. Qualifier reference 0 indicates that no enabling qualifier is applicable. The 'start' and 'end' vector index of each event are stored in the event table.

When an event_id is present at the queue head this id is directly used to index the event table. The 'start' and 'end' index from the event table are then used to cycle through the vector table, applying enabling qualifiers as indicated. The processed event_id is cleared from the queue head on event completion. The idle vector on index 0 (fixed) is offered until a new event_id is incident at the queue head.

3.5.3.2 Stimulator VHDL source

The design is split up into a bus_interface and a vector_processor process and a cyclic_buffer component. Its structure is depicted in Figure 3.9. The VHDL source code can be found in appendix A.4 on page 79.

The stimulator is parameterizable over its output width, sensitive clock edge and file locations of the event, vector and qualifier table.
Files are read by the vector processor at simulation start by three dedicated procedures.

The observer accepts the applicable bus commands defined in par. 3.1.3. The bus interface extends supervisor control into the stimulator entity. It is sensitive to any event on the command signal and completes confirmed ‘write’ commands and unconfirmed ‘reset’ commands. The event_id on data is written to the tail of the event queue on reception of a ‘write’ command. A ‘reset’ command toggles the restart signal, which resets the vector processor and empties the event queue.

3.5.3.3 Tool implications

A "Stimulator Code Generator" has not yet been projected. However, it could (assist to):

1) isolate the operator from the low level VHDL array indexing,
2) calculate packet checksums (in this case 32-bit CRCs),
3) generate (semi random) data patterns according to some scheme,
4) and support structuring of the stimulus events (for instance by supporting ASN.1 packet format definitions).

3.6 Event queues

3.6.1 Overview

The operational semantics of TTCN state that both stimulus and observable events must be queued at the different PCGs. The observer queues can be polled non-destructively for the purpose of event matching. Either the head or whole contents of the queue can be cleared.

3.6.2 FIFO Choice & Implementation

FIFO queues come in two types: unbound and of fixed size. We have chosen to implement the FIFO as a cyclic buffer. The VHDL source code of the design can be found in appendix A.2 on page 70. The cyclic buffer is parameterizable in size through generics.

The array fifo is indexed by two pointers: a read pointer ri and a write pointer wi. Initially (empty fifo) both pointers index the same array location. The write pointer is advanced one location on a write to the fifo tail. The read pointer is advanced one location on a read from the fifo head. The basic rule is that the write pointer may not overtake the read pointer and vice versa.

The fifo head is cleared when an event propagates through the clear_head port and the whole fifo is cleared on any event on restart. The integer event_id on fifo_in is written to the fifo tail on any event on fifo_write and fifo_out reflects the actual fifo head.
Chapter 4

Towards P1394 test execution

In the previous chapters we have addressed the development of generic components that facilitate the construction of a VHDL test bench. The bench is intended to be used as an environment for execution of TTCN test suites produced by the Conformance Kit. In this chapter we will address the development of an EFSM style model for the P1394 LinkCore, and construct a toplevel test bench for execution of automatically generated test suites.

4.1 Overview on the test trajectory

4.1.1 Outline

Figure 4.1 depicts an overview on the test trajectory/process as we advocate it. The activities that we must undertake to obtain test results (and some form of conformance statement) can be scheduled in four phases:

1) **Formalization phase**: since we do not have an EFSM model available for input to the Conformance Kit, we must obtain one by analysis. The information needed to do so is captured from the standard, the Apple interface doc. and the design team.

2) **Test generation phase**: this phase is automated by the Conformance Kit. Test generation however, does still require (some) human effort. The generation process can be guided by applying restrictions over parts of the EFSM model (see [Kwa94]) and we can make a selection of cases from the produced test suite.

3) **Test implementation phase**: this phase addresses instancing of a VHDL test bench and writing the stimulus and observable event sets. The first requires little intellectual effort, but the latter requires great insight in all the details of the protocol. The detail that was abstracted from in the formalization phase must now be re-added.

4) **Test execution phase**: the event set definitions are compiled into executable code for the test bench. The execution of the test suite itself is completely automated. The test operator only has to determine the simulation time frame length in which the suite has to complete.
Towards P1394 test execution

Figure 4.1 Overview on the test trajectory for the P1394 LinkCore.
The formalization phase has to precede all other phases. The generation and implementation phase can be started simultaneously. However, completing test implementation requires the integer event and PCO identifiers that are used in by the VHDL test bench. After completion of the simulation test run(s) the

5) **Results analysis phase**: interprets the test results contained in the conformance log. These results will have to be analyzed in order to formulate a trustworthy conformance judgement. Errors may have been introduced in the tests themselves. The manually re-added event detail, and the formalizing phase are the main contributors to these errors. Analysis may therefore result in another test iteration with an adjusted model and/or event sets. The automated parts of the test trajectory make quick test iteration possible.

4.1.2 Blank spots

The present project status is that not all supporting tools have yet been implemented. The TTCN compiler is still under construction and the "Stimulator Code Generator" development must be started soon (see par. 3.5.3.3).

4.2 Modelling efforts

We have followed a two phase approach in defining an input EFSM model for the Conformance Kit. First we have defined an EFSM model that uses parameterized communication events (ie. carry PDU data and service parameters). This model captures the protocol dynamic well but is unsuitable for use with the Conformance kit.

The parameterized EFSM is then mapped onto an unparameterized Conformance Kit EFSM by making a selection over the parameters.

4.2.1 A parameterized EFSM style model

For the first trial runs of our test environment we need a small, yet representative model of the link layer. Modelling is a science in itself, and we only had a limited capacity. We have decided to confine modelling efforts to an isolated section of the P1394 protocol in order to obtain a model with some depth (not too abstract). This depth will be needed for true conformance testing, and in this way we are likely to encounter the problems associated with future test trajectories, making it a relevant test model.

We have modelled the synchronous part of the P1394 link layer. In the link layer state machine (see [IEE95; Figure 6-19, Page 174]) this entails the restriction to states L0-L7 together with state L14 (in other words, leaving out states L8-L13). The inclusion of the isochronous part is a possibility for later extension. The main abstraction with respect to the original state machine concerns the abstraction of symbols from the physical layer into packets. The sending and receiving of packets is considered to be an atomic action. Data delimiters (such as DATA_PREFIX and DATA_END) have been integrated into a single
Towards P1394 test execution

packet (see appendix D.1). Furthermore, timing is not modeled explicitly: we abstract from speed and physical clock indications. The only exceptions are a time-out (modeled as a condition) in state L7 and an output event that should occur fast enough (to keep the bus busy) in state L5.

The model has been represented graphically in Figure 4.3. The notation used is a state-transition diagram with transitions, having labels of the form \(?input / !output\), extended with [conditions] and (variable assignments). The events relate to the services depicted in Figure 4.2. Specific comments can be found in appendix D on page 123.

![Figure 4.2](image)

**Figure 4.2** The link layer services, and service usage in the P1394 protocol stack.

The general form of a transition is as follows:

\[ ?input\_event(in\_pars) \ [condition] / \ \{variable\_assignment\} \ !output\_event(out\_pars) \]

All four parts are optional. Transitions have no duration; time only elapses in states. A transition is enabled whenever its condition evaluates to TRUE. A transition containing an input event is taken the moment the input event occurs while it is enabled. A transition without an input event is taken as soon as it is enabled.

Transition labels can be distributed over two edges to save on paper space. A distributed label lacks the '/' separator. Originating from state L0 the \(?PH\_DATA\_ind(p)\) must be combined with a label of one of the four transitions to L0 or with the label of the
transition to L4. Originating from state L5, the $\text{LINK_DATA}.\text{resp}(\text{ack_code}, \text{bus_occ_ctl})$
must be combined with the edge labels of transitions to L0 and L7. A transition always ends with an arrow to the target state. When traversing the diagram the composition of the labels must form a legal transition; for instance, when starting from L0 in the direction of L4 and taking one of the branches to the left leading back in the direction of L0, one
cannot go down in the direction of L7 because (amongst others) this would pass another `/` separator.

All transitions regarding the INITIALIZE and RESET commands have been left out for clarity of presentation. State L14 then has no function anymore and has also been left out. To complete the diagram for the full functionality of the asynchronous part, state L14 should be re-added together with the following transitions:

`?LK_CONTROL.req(INITIALIZE) / !LK_CONTROL.conf()` from all states to L0, and
`?LK_CONTROL.req(RESET) / !LK_CONTROL.conf()` from all states to L14.

4.2.2 Mapping onto an unparameterized EFSM

We have chosen to implement the minimal map that allows us to take all transitions of the model of Figure 4.3 at least once with a selected set of parameters. This requires some transitions to be mapped for different parameter sets as these enable transitions at other locations in the parameterized EFSM.

For instance, the packet parameters p received on a transition from L0 to L0 or L4 determines which of the five possible transitions is taken. If we want to be able to exercise all five transitions we must select five representative parameterized events that each enable a transition. We can use the Kit's local variables to carry parameters across a number of transitions, ultimately enabling some transition guarded with a predicate over the local variables.

The mapping procedure followed is described in appendix D.5.1. The resulting Unparameterized UEFSM is given in appendix D.5.2. The selected abstract parameters of the Parameterized PEFSM are hard coded into the names of the UEFSM. Representative instances of these events will have to be chosen when writing the event sets.

Three PCOs are introduced. They correspond with the three interfaces between the link layer and the transaction layer, node controller and physical layer. The missing L14 state is introduced again (with the transitions that lead to and from it) and the specification is made complete by adding outputless transitions that transfer back to the state they originate from. PEFSM transitions that do not have input behavior are replaced with the triggering spontaneous event and the timer condition is replaced with an event timer.

The test generating tools of the Conformance Kit require that the input UEFSM is converted to a complete UFSM. These completing transitions are can then excluded from the set of test purposes (they were originally not specified!) by applying restrictions over them in test generation. These restrictions can however not be applied to the test postamble SIOSes (Simple Input Output Sequences). It is not clear how the implementation will react to these unspecified stimuli (robustness testing). A sample restriction file can be found in appendix D.5.3.
Towards P1394 test execution 55

4.3 Instancing a VHDL test bench

4.3.1 Overview

We need to construct a test bench that can execute the generated test suite. For the UEFSM model of the previous section this would require a bench that:

1) has three PCOs (no. 1, 2 and 3) and thus three stimulator-observer pairs,
2) can issue 22 (+2) stimulus events in total (distributed over the PCOs),
3) and recognize 18 observable events (distributed over the PCOs),
4) has 2 timer instances (fixed for all Conformance Kit suites).

The need to handle the event set adequately is at the instancing level a matter of connecting the observers to the right interfaces and selecting only those signals from these interfaces that need be dynamically driven in the course of test execution.

It is at this time that the model and the implementation meet. There must exist an one-on-one relation between the implementation and the model for testing to succeed. If the two deviate substantially, even the connection to the test interface becomes troublesome.

This is exactly the case here. We here have to cope with discrepancies of two kinds: 1) distortion of the UEFSM model by having used unnatural constructs in formalization of important protocol aspects that do not quite into the formal domain, and 2) deviation from the P1394 standard by the Apple implementation.

These discrepancies are elaborated upon in par. 4.4. As the Apple LinkCore is likely to establish itself as a de facto standard, we cannot simply discard it as being non-conforming. We have chosen not to frustrate the testing process and plan to adjust the model to account for the LinkCore deviations. The goal was, and remains, to evaluate our test environment and acquire experience which may guide future developments.

In the next section we illustrate the process of instancing a test bench for the P1394 LinkCore. We will have to use an adjusted model to be able to generate tests for it. The specifics of the Apple implementation are bound by a non-disclosure agreement so we will try not address them.

4.3.2 Implementation

The toplevel test bench design consists of a link component, two stimulator components, two observer components, a supervisor component, a statics process and a clocks process. Its structure is depicted in Figure 4.4 and the VHDL source code can be found appendix A.7 on page 95.

The Apple implementation has nine interfaces in total. We only need three of these to conduct dynamic tests against the asynchronous part of the link layer. The two clock signals that govern those three interfaces (a 50MHz and a 25MHz clock) are driven by the clocks process. The clocks rising edges are matched.
The EUT's input signals that are of no interest to the dynamic tests are driven statically by the *statics* process. The values, that are assigned by the *statics* process to the signals connected to the static EUT ports, are read from the file "statics.dat" at the start of simulation. When all signals have been assigned, the *statics* process suspends for the duration of the simulation run.

The seven signals that make up the test bus are declared at the toplevel. The *supervisor* is instanced with two internal timers and connected to the test bus using a standard port map.
The two observers are instanced to match the total width of the bundle of signals that they observe, and both sample data on the rising edge of the interface clock. Connection of the bundle is done by connecting each signal to a slice of the observer input port "obs_in". Of course the order of connecting must match the one assumed in defining the events sets. The observers are parameterized over their own integer (PCO) bus address using '1' for the first PCO declared in the TTCN test suite, '2' for the next PCO declared, and so on.

The stimulators sample their data on the rising edge of the interface clock, and the width of their output port "stim_output" is matched to the bundle of signals they drive. Connection is done in the same fashion as for the observers and also the same addressing scheme applies.

4.4 Conclusions

4.4.1 Limitations of the EFSM model

The rigid input/output (or input/-) specification style troubles description of event sequences with consecutive output events and spontaneous output behavior. The introduction of a spontaneous@internal event (that models unsolicited transitions) does not solve the problem. It results in production of test cases that explicitly mention that event. Since there is no observable analogue at the test interface, these events must be removed from the produced test cases. This takes a lot of reformatting of the test suite.

The same applies for the introduced timer event. Since it models an internal course of action, there is no such event to be observed at the test interface.

Problems could have been circumvented if the Conformance Kit:

1) Had supported non-deterministic specifications. A set of different behaviors, which occurrence can not be justified due to the semantic model underlying the EFSM, could be considered equally valid. The model thus abstracts from that issue; it should be avoided at all cost that a correct EUT is rejected because of a deficiency of the model used.

2) Allowed specifications with uncoupled input/output behavior descriptions. This matches better to the semantic model underlying the TTCN tests themselves.

3) Allowed richer use of the TTCN timers to judge on the timing properties of the EUT; not just to safeguard test execution.

4.4.2 LinkCore deviations

The Apple LinkCore implementation deviates considerably from the P1394 standard link layer specification. These deviations can be categorized into deviations that concern the functionality (the what) of the LinkCore and the deviations that concern the way in which the functionality is implemented (the how). One could argue that not the separate layer
Towards P1394 test execution

descriptions are normative but the combined operation of all specified layers (ie. the overall operations of a total system must conform to the specification). A counter argument is that implementations of individual layers are appearing on the market (like TI's physical layer chip); interoperability of these sub-implementations would be an asset for industrial customers.

Regarding the discrepancies:

1) The A/B retry protocol has been partly moved from the transaction layer into the link layer (to meet high performance requirements). This has great implications in the way acknowledgements are produced. The standard specifies that acknowledgements are initiated by the transaction layer, the LinkCore transmits acknowledgements autonomously. The upper interface services have been changed accordingly.

2) The confirmed services on the Node Controller interface are replaced by unconfirmed inputs for static parameterization and control. (reset, bus_id,...).

3) Some services have not been implemented at all (like the !lk_event.indO service) or have been poorly documented (!lk_data.conf(ACK_MISSING) was hard to find).

4) Extra options have been implemented (for instance bus snooping).

In general, the implementation possesses a lot of ports that cannot be accounted for solely from the standard's point of view (like the queue interfaces; packet queuing is left unspecified in the standard). The opposite applies as well: a lot of events from the standard could not be mapped onto the implementation interface.

To smooth the testing process and refine our test environment we need to adjust or rewrite the model so that it coheres to the functionality (ie. matches to the interface) of the LinkCore. The functionality across the interfaces can still be viewed against the standard to justify the "conformance" in conformance testing.
Chapter 5

Conclusions

5.1 Results obtained

We have developed a conceptual conformance test trajectory for VHDL implementations of the asynchronous part of the link layer. This trajectory consists of four phases: modelling (formalizing), automatic test generation, test realization (linking interface detail to the abstract tests) and automatic test execution. We have used the PTT's Conformance Kit for test generation. Since the Kit does not support data flow testing we have advocated a two phase modelling approach in which we can gradually add data aspects by service diversification. Although this is not the most straightforward approach thinkable, it will enable us to explore the potential benefits of automatic testing. The biggest asset of the work presented in this thesis lies in development of a VHDL test bench for the automatic execution of TTCN test cases. The conformance log produced reports non-conformance and supplies diagnostic information for fault finding. Human input will ideally be limited to supplying the needed interface details.

Although all that is developed is tailored to the Conformance Kit and the specific mapping onto the P1394 LinkCore interface, we consider the problems encountered to be common for (synthesizeable) VHDL models of protocol layer implementations in general. We have gone to great lengths to make the test bench generally applicable for the TTCN subset used and have not exploited the structural characteristics of the Kit's test suites. The test bench could lend itself for enrichment towards support of a bigger TTCN subset.

A number of questions have been raised with respect to the question whether or not the execution of test cases, obtained in the way we did, alone could justify a conformance judgement on dynamic behavior. This would require:

a) The (EFSM) models used to be normative. The process of transcribing the informal standard into a formal form hasn't been easy nor unambiguous. It has lead to quite some fierce discussions on: what to formalize and what not to formalize, how to interpret the natural language descriptions, how to abstract from important issues that do not quite fit into the formal domain used (timing aspects, concurrency,...).

b) The abstract model alone to contain enough information (detail) to decide on conformance. Even when formal models become part of communication standards,
and the methods for test generation for the respective model class are standardized together with it, they will always abstract of detail. This detail will have to be re-added in testing; leaving a lot of room for choices as how to do this. If the de-abstraction step is not standardized two problems arise: 1) any two test laboratories may choose different test implementations and are thus liable to decide differently on conformance of a single IUT, 2) even low detail issues can interfere with high level interoperability and thus deserve attention in testing.

Since neither of these requirements are even remotely addressed or in our attempts the question might arise if it is such a good idea at all. If we want to establish confidence in the quality of any realization through testing, the complexity of the doing so is directly related to the complexity of the design itself. Strong arguments for following the approach we did are:

a) It automates the most tedious tasks and thus increases total test capacity [cases/sec]. This allows more tests to be executed in the same limited time span. The number of test cases is inherently determined by the chosen abstraction level and are distributed (strategy!) fairly over the entire protocol functionality, avoiding over or under emphasizing by intuitive choices.

b) The structural approach decomposes the test creation problem into two, individually easier to solve, tasks: first create the underlying abstract model and then write representative event sets. Errors traced to back to either part of the tester itself, are confined in nature and thus easier to fix.

c) We have created a VHDL environment allowing peer-interaction with a design at the simulation level. The interactive, high level interaction is difficult, if not impossible, to obtain with sequential stimulus vector files.

d) The test bench is easy to reuse due to its generic qualities. Besides its designated use in conformance testing, it could be used for white box testing in the protocol development phase itself. TTCN test cases could be hand written, using a dedicated editor, to support analysis of high level peer interaction.

5.2 Test paradox

All is well that ends well! Or... Early in the project, I have asked myself the fundamental question whether or not we will ever be able to automate the entire testing process based on formal methods and techniques. I think not.

Picture the following situation. Being the great advocates of formal methods we are, we foresee a (near) future in which the use of FDT's, as the start of any protocol development trajectory, is common practice. We can picture the parallel development of implementation(s) and tester(s), all originating from the same unambiguous formal specification S. Both trajectories are schematically depicted in Figure 5.1.

The followed line of reasoning, specifically revolves around Mealy I/O automata specifications. The general thought however addresses the information content of the models used and therefore applies to FDT's with other underlying semantic models as well.
Our test trajectory basically turns the specification inside-out by mirroring it. The implementation's prescriptive I/O relations become the testers $O_{stimulus}/I_{response}$ test expectations regarding correct behavior (the PASS verdicts). This mirrored spec is then expanded with (all) other possible behaviors regarded as incorrect behavior (the FAIL verdicts). The actual test generation process does not so much generate something new; rather it selects from this mirrored information. This selection process is the test generation strategy (RS, PT or TT) used. This is the main reason that testing usually cannot prove an implementation is correct; the non-exhaustive tester only samples its quality. One hopes to rate generation (=selection) strategies by their fault detecting power at equal test effort. This is at least questionable; for every non-exhaustive tester erroneous models can be thought of that pass such strategies undetected. Whenever we fix the strategy to catch these counter examples, we will still be able to think of other erroneous models that again will pass then new, improved strategy, unless we resort to the canonical tester which runs forever.

I spoke of models that go undetected not without a reason. At this point the tests are still of an abstract nature; more specifically within the same formal domain (at the same functional abstraction level) as the specification we originated from. The task of implementing this test specification is of the same complexity as implementing the protocol all together. This is the point I am aiming at! In order to implement the tester $T_n$, we must create a mechanism (so to speak) that coheres to the abstract test specification $T_s$. In case of the canonical tester the information content of the model of the tester is equal to the original specification (ie. we have not thrown any information away and could make the inverse transformation back to the original specification). If we reason that we could construct the physical tester completely automatic (the information content of the specification should be large enough to facilitate that) than we must be able to do the same for the implementation $I$ itself! We then have eliminated the need for testing all together and our efforts would be better spent on research on construction of an automatic synthesis trajectory.

If the abstract model $S$ used does not allow an one-on-one mapping onto a physical counterpart (ie. its information content is not high enough), like the P1394 situation presented in this thesis, then the missing details have to be filled in by interpretation of informal descriptions or by design choices. We have tried to make the coherence with respect to an abstract model plausible by selection/projection of high detail stimuli and
62 Conclusions

observations. If our goal had been to prove it, canonical selection (and no projections) would have been mandatory. The same line of reasoning applies: if we had some tool that allowed us to link high detail constructs to the abstract model framework (a verified compiler), then the properties of the abstract model would be preserved throughout the development trajectory; again removing the need for testing.

Personally I do not believe in realization of any of the black-white scenarios sketched above. It is my opinion that automatic compilation of a standardized specification into an implementation will never be possible. (There wouldn't be any fun it either; as soon a new standard is released just feed it to your compiler). If it is indeed possible, then obtainment of the specification itself requires the main development effort. I doubt there will be any open systems left at all; obviously it is the best specification that wins and any sensible company will go to great lengths to keep theirs a secret.

It is unlikely that the entire process of testing can ever be completely automated. If it is indeed possible, then testing has been made redundant. Rather, it seems more likely that we must aim at two separate, imperfect trajectories for development of tests and implementations. It's the human factor that makes these trajectories imperfect but possible at all. It therefore seems a good strategy to aim at developing tools that synergize with human experience, knowledge and intuition. To make this synergy work, concentration on algorithms is not enough. Equal attention has to be paid to aiding in inputting human decisions and remarks into both trajectories. By performing both tasks in parallel and by independent teams one hopes that the human imperfections cancel out (partly) to result in a better end quality level than any single trajectory alone could provide. Recognizing this might very well prove to be the winning edge in the worldwide, competitive field of protocol development and testing.

5.3 Recommendations for future work

Short term efforts should address completing the overall test environment. In this first phase we like to ascertain that the architecture is indeed an asset. We should:

1) Build the TTCN compiler

2) Devise an aid for writing stimulus event sets (Stimulus Code Generator)

3) Revise the parameterized EFSM model. The P1394 case study uncovered some nasty modelling problems. The boundaries of the link layer specification do not coincide with the interface of the LinkCore. So the problem of de-abstracting the model (ie. writing the event sets) is hampered by the fact that the abstract model underlying the implementation differs from the IEEE draft standard. The differences are of a magnitude that makes it impossible to be absorbed by the selection and projection margin of the stimulator(s) and observer(s) and is bound to frustrate the testing process before it has even begun. We therefore could decide the Link Core to be non-conforming. However, since, the Apple Link Core is likely to establish itself as a de facto standard, the IEEE standard will probably follow. As not to deprive our conceptual test bench of the material needed to evaluate it, we have
chosen to remodel the input EFSM to be congruent to the Apple solution. Thereby 
the interface boundary is dictated by the LinkCore but for the functionality across 
those interface(s) we must primarily use the IEEE Draft 8.0v2 standard. The accent 
shifts towards evaluation of the test bench and gaining knowledge about the Apple 
solutions. Harsh conformance judgements will be less likely...

4) The revised, parameterized model is best first mapped onto a simple and 
unparameterized form, not yet accounting for data flow aspects. Through execution 
of the tests generated from this first model the VHDL architecture can be debugged 
and the entire concept of asynchronous testing through queues, making the de-
abstracting selections/projections we did can be validated. These first test 
executions will iteratively adjust the model, adjust the event sets and reveal bugs in 
the test bench until flawless testing (against the de facto standard) is achieved.

If the experiences of the first phase are positive, it is worth to pursue improvements to 
both the test environment and the quality of the test results obtained. This second phase of 
future work could:

5) The conformance log produced reports with the integer identifiers used in the 
VHDL architecture. To enhance the log's readability it could be compiled upwards 
(reusing the event mapping files) to use the event identifiers of the EFSM model. 
The addition of the "VHDL log compiler" to the general overview of Figure 4.1 has 
been expressed in Figure 5.2.

6) Elaborate towards a conformance statement by refining the mapping onto an 
unparameterized EFSM model. We can account for a selection of the data flow 
aspects (of the parameterized model) by diversification of input ASPs and relating 
them to correspondingly diversified output ASPs by "name reference" only.
7) Refine the tools environment. The first phase experiences might pinpoint bottlenecks in test preparation/execution that need further attention. For instance: devise dedicated editors that assure complete description of the event mapping files by zooming in empty event description templates (smoothing of the user interface).

The long term success of the project can only be determined by the results obtained with it. However, I would advise to monitor the worldwide acceptance of the TTCN test suite notation especially in the execution domain. The vast majority of TTCN related publications I found dealt with test automatic generation strategies. Ultimately, the added value of these techniques can only be measured in the test execution domain. If all continues to look well we could aim our further research efforts at:

8) Evaluating the possibility to build an own test generator that uses an input formalism that has the same underlying semantic model as TTCN. Thus allowing production of richer test cases without the restriction of fixed input/output and input/-style transitions.

9) Integrating the data flow testing into the test trajectory. Of course, data flow testing is just as important as control flow testing and it is not really supported at the moment. We need to exploit the benefits of the formal approach to ensure the preservation of data flow properties of our initial formal model in the constructed test cases. The intent of TTCN is that stimulus choices and restrictions on response acceptance are fixed in the test suite's constraint references. Test suites of this form should again be compilable into an unparameterized executable form for our VHDL test bench.
References

Compilers: principles, techniques and tools.
Amsterdam: Addison-Wesley, 1986.

[Dro96] Drost, N.
De tool codemaker.

[HU79] Hopcroft, J.E. and J.D. Ullman.
Introduction to automata theory, languages, and computation.
Addison-Wesley, 1979.

[IEE95] IEEE.

[ISO89] ISO.
Information Technology, Open Systems Interconnection, Conformance Testing
Methodology and Framework.

[Kni93] Knightson, K.G.
OSI Protocol conformance testing, IS 9646 explained.

[Koo91] Koomen, C.J.
The design of communicating systems, a system engineering approach.

[Kwa94] Kwast, E et al.
User Manual of the Conformance Kit.
References

_VHDL: hardware description and design._

[Luk94] Lukassen, R.
_FORTE/VESTA, automatic translation of formal specifications for
conformance testing._
Final report of the postgraduate programme Software Technology, Institute for

[PM92] Probert, R.L. and O. Monkewich
_TTCN: the international notation for specifying tests of communicating systems._

[Tur93] Turner, K.J.
_Using formal description techniques, an introduction to Estelle, Lotos and SDL._
Appendix A

VHDL Sources

A.1 test_pkg.vhdl

-- RCSId: $Id$
-- File : test_pkg.vhdl
-- Related Files : -
-- Author(s) : O. Sies
-- Organization : Philips Research Laboratory, Eindhoven
-- Project : Testbench for P1394 Link Core
-- Creation Date : 16.02.1996
-- Simulator : Cadence Leapfrog Simulator
-- Synthesis : none
-- Description : Types used in entity port declarations and other reusable types and subprograms.

Copyright

Copyright (C) 1995 Philips Research Laboratory, Eindhoven
All rights reserved. Reproduction in whole or part is prohibited without the written permission of the copyright owner.

History

$Log$

Package : test_pkg
-- Author(s) : O. Sies
-- Organization : Philips Research Laboratory, Eindhoven
-- Project : Testbench for P1394 Link Core
-- Creation Date : 09.02.1996
-- Description : Visibility of entity port types and commonly used types for all entities and architectures.
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

LIBRARY std;
USE std.textio.ALL;
USE std.standard.ALL;

LIBRARY util;
USE util.std_logic_textio.ALL;
USE util.misc_conv.pkg.ALL;

PACKAGE test_pkg IS
  -- generic types & functions for observers/stimulators
  TYPE edge_type IS (falling, rising);
  FUNCTION sel_edge (sel : edge_type; SIGNAL clock : std_logic) RETURN boolean;

  -- integer to time conversion
  FUNCTION to_time (i : integer; unit : string(1 TO 2)) RETURN time;

  -- reads a (qualifier) string
  PROCEDURE readstring (l : INOUT line; s : OUT string);

  -- types for toplevel busses
  TYPE control_type IS (cancel, clear, nice, read, release, reset, start, status, write);
  TYPE enable_type IS (en_observer, en_stimulator, en_timer);
  TYPE int_vector IS ARRAY (integer RANGE <>) OF integer;
  FUNCTION resolve_int (input: int_vector) RETURN integer;
  SUBTYPE resolved_int IS resolve_int integer;
  TYPE bool_vector IS ARRAY (integer RANGE <>) OF boolean;
  FUNCTION resolve_bool (input: bool_vector) RETURN boolean;
  SUBTYPE resolved_bool IS resolve_bool boolean;
END test_pkg;

PACKAGE BODY test_pkg IS
  -- selectable edge detection
  FUNCTION sel_edge (sel : edge_type; SIGNAL clock : std_logic) RETURN boolean IS
    BEGIN
      IF sel = falling THEN
        RETURN falling_edge(clock);
      ELSE
        RETURN rising_edge(clock);
      END IF;
    END sel_edge;

  -- integer to time conversion
  -- inspired by UTIL.VSTRING.tostring() and UTIL.VSTRING.fromstr()
  FUNCTION to_time (i : integer; unit : string(1 TO 2)) RETURN time IS
    VARIABLE t : time;
    VARIABLE linebuf : line;
VARIABLE ok : boolean;

BEGIN
  write(linebuf, i);
  write(linebuf, unit, right, 3); -- allow a separating blank
  read(linebuf, t, ok);
  deallocate(linebuf);
  IF ok THEN
    RETURN t;
  ELSE
    ASSERT (false)
    REPORT "to_time: error in integer to time conversion."
    SEVERITY error;
    RETURN time'LOW;
  END IF;
END to_time;

-- Reads a (qualifier) string from line l

PROCEDURE readstring ( l : INOUT line;
  s : OUT string) IS
  VARIABLE c : character;

BEGIN
  FOR i IN s'RANGE LOOP
    IF NOT endline(l) THEN
      read(l, c);
      IF (c = cr) OR (c = ht) THEN
        s(i) := null;
        EXIT;
      ELSE
        s(i) := c;
      END IF;
    ELSE
      s(i) := null;
      EXIT;
    END IF;
  END LOOP;
END readstring;

-- Resolution functions for top level busses.
These busses are guarded signals of type 'BUS' and therefore could
be invoked with an array length of zero.

FUNCTION resolve_int (input: int_vector) RETURN integer IS
  BEGIN
    ASSERT (input'LENGTH < 1)
    REPORT "Overdriven integer bus"
    SEVERITY error;
    IF input'LENGTH = 0 THEN
      RETURN 0; undriven value
    ELSE
      RETURN input(input'LOW); put only driver into signal
    END IF;
END resolve_int;

FUNCTION resolve_bool (input: bool_vector) RETURN boolean IS
  BEGIN
    IF input'LENGTH > 0 THEN
      FOR i IN input'RANGE LOOP
        IF input(i) THEN
          RETURN true;
        END IF;
      END LOOP;
      RETURN false;
    END IF;
    RETURN false;
END resolve_bool;

END test_pkg;
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
LIBRARY std;
USE std.textio.ALL;
USE std.standard.ALL;
LIBRARY util;
USE util.std_logic_textio.ALL;
USE util.misc_conv_pkg.ALL;

ENTITY fifo IS
  GENERIC (size : positive);
  PORT (
    clear_head : boolean;
    fifo_in : boolean;
    fifo_out : OUT natural;
    fifo_write : boolean;
    new_head : INOUT boolean;
    restart : boolean
  );
END fifo;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
LIBRARY std;
USE std.textio.ALL;
USE std.standard.ALL;
LIBRARY util;
USE util.std_logic_textio.ALL;
USE util.misc_conv_pkg.ALL;

ENTITY fifo IS
  GENERIC (size : positive);
  PORT (clear_head : boolean;
    fifo_in : boolean;
    fifo_out : OUT natural;
    fifo_write : boolean;
    new_head : INOUT boolean;
    restart : boolean
  );
ARCHITECTURE behavior OF fifo IS
BEGIN

process (restart, fifo_write, clear_head)
BEGIN
  IF restart'EVENT THEN
    asynchronous reset
    wi := fifo_index_type'LEFT;
    ri := fifo_index_type'LEFT;
    fifo_out <= 0;
  ELSE
    IF clear_head'EVENT THEN
      IF ri = wi THEN
        fifo empty?
        ASSERT (false)
        REPORT "cannot remove from an empty fifo"
        SEVERITY WARNING;
      ELSEIF wi = adv(ri) THEN
        fifo contains one id?
      END IF
    END IF
  END IF
END process;

END fifo;
ri := adv(ri);
fifo_out <= 0;
ELSE -- fifo contains two or more id's
    ri := adv(ri);
fifo_out <= fifo(ri);
new_head <= NOT new_head;
END IF;
END IF;
IF fifo_write'EVENT THEN
    IF ri = adv(wi) THEN -- fifo full?
        ASSERT (false)
        REPORT "cannot add to a full fifo"
        SEVERITY WARNING;
    ELSIF ri = wi THEN -- fifo empty?
        fifo(wi) := fifo_in;
        wi := adv(wi);
fifo_out <= fifo(ri);
new_head <= NOT new_head;
ELSE
        fifo(wi) := fifo_in;
        wi := adv(wi);
    END IF;
END IF;
END IF;
END PROCESS cyclic_buffer;
END behavior;

-- Configuration : fifo_conf
-- of Entity : fifo
--
-- Author(s) : O. Sies
-- Organization : Philips Research Laboratory, Eindhoven
--
-- Project : Testbench for P1394 Link Core
--
-- Creation Date : 06.03.1996
--
-- Description : Binds architecture 'behavior' to entity 'fifo'.
--

CONFIGURATION fifo_conf OF fifo IS
    FOR behavior
        -- no components
    END FOR;
END fifo_conf;
A.3 observer.vhdl

LIBRARY ieee; USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

LIBRARY std;
USE std.textio.ALL;
USE std.standard.ALL;

LIBRARY util;
USE util.std_logic_textio.ALL;
USE util.misc_conv_pkg.ALL;

USE work.ALL;
USE work.test_pkg.ALL;

-- the observing entity
ENTITY observer IS
  GENERIC (
    pco_id : positive; -- pco_id of this observer
    edge : edge_type; -- sensitivity to clock edge
    width : positive; -- at least a width of '1'
    i_filename : string; -- parser instructions file
    m_filename : string -- qualifiers & masks file
  );
  PORT ( 
    address : integer;
    clock : std_logic;
    command : boolean;
    control : control_type;
    data : OUT resolved_int BUS;
    enable : enable_type;
    new_event : OUT resolved_bool := false;
    obs_input : std_logic_vector(width-1 DOWNTO 0);
    response : INOUT resolved_bool BUS
  );
END observer;

ARCHITECTURE qualifiers OF observer IS
  -- fifo component declaration
  COMPONENT fifo
    GENERIC ( 
      size : positive 
    );
    PORT ( 
      clear_head : boolean; -- on event clear FIFO head
      fifo_in : positive; -- valid id's are '1' and up
      fifo_out : OUT natural; -- '0' included to indicate empty fifo
      fifo_write : boolean; -- on event write 'fifo_in' into FIFO
      new_head : INOUT boolean; -- event indicates new FIFO head
      restart : boolean; -- reset
    );
  END COMPONENT;

  SIGNAL nice_dog : boolean;
  SIGNAL clear_head : boolean;
  SIGNAL fifo_in : positive; -- only valid id's can be written
  SIGNAL fifo_out : natural; -- initial value = 0 : empty fifo
  SIGNAL fifo_write : boolean;
  SIGNAL new_head : boolean;
  SIGNAL restart : boolean;
BEGIN
parser: PROCESS

-- locate files with the strings passed through the generics
FILE i_file: text IS IN i_filename;
FILE m_file: text IS IN m_filename;

-- constants specific to parser
CONSTANT maxmasks: integer := 127; -- nr. limited by LEX
CONSTANT maxinstrs: integer := 1000; -- max. nr. parse instructions

-- specific types
TYPE masknames IS (msk_1, msk_x);
TYPE masks IS ARRAY (integer RANGE 0 TO maxmasks, masknames) OF std_logic_vector(width-1 DOWNTO 0);
TYPE opcode_type IS (opc_wait, opc_match, opc_goto, opc_out);

TYPE instruction IS
RECORD
  opcode : opcode_type;
  operand_1: integer;
  operand_2: integer;
END RECORD;

TYPE instructions IS ARRAY (integer RANGE 0 TO maxinstrs) OF instruction;

-- fills the instruction array 'ia' of the observer with parser code
PROCEDURE read_instructions (f: IN text;
ai : OUT instructions) IS

VARIABLE l: line;
VARIABLE i: integer;
VARIABLE opc: string(1 TO 3);
BEGIN
  WHILE NOT endfile(f) LOOP
    readline(f, l);
    read(l, i); -- line nr. indexes the array
    read(l, opc); -- opcode string
    CASE opc(2 TO 3) IS
      WHEN "MA" =>
        ai(i).opcode := opc_match;
        read(l, ai(i).operand_1);
        read(l, ai(i).operand_2);
      WHEN "GT" =>
        ai(i).opcode := opc_goto;
        read(l, ai(i).operand_1);
      WHEN "OU" =>
        ai(i).opcode := opc_out;
        read(l, ai(i).operand_1);
    END CASE;
  END LOOP;
END;
WHEN OTHERS => -- "WT"
    ai(i).opcode := opc_wait;
END CASE;
END LOOP;
END read_instructions;

-- the enable string, abort string and masks are contained in one file.
-- this procedure fills 'es', 'as' and 'ma' with their respective contents
-- from file.

PROCEDURE read_masks (
    VARIABLE f: IN text;
    es : OUT string(1 TO 80); -- enable input qualifier
    as : OUT string(1 TO 80); -- abort event qualifier
    ma : OUT masks) IS
    VARIABLE l : line;
    VARIABLE i : integer;

    BEGIN
        readl ine(f, l);
        readstring(l, es); -- 1st line: enable qualifier
        readline(f, l);
        readstring(l, as);-- 2nd line: abort qualifier
        WHILE NOT endfile(f) LOOP
            readline(f, l);
            read(l, i);
            read(l, ma(i, msk_1));-- line nr. indexes array 'ma' 
            read(l, ma(i, msk_x));
            END LOOP;
    END read_masks;

-- function that evaluates a boolean simple expression

FUNCTION eval (s: string) RETURN boolean;
ATTRIBUTE foreign OF eval: FUNCTION IS "RB_C_Lib:Evaluate";

BEGIN
    read_instructions(i_file, ia);
    read_masks(m_file, es, as, ma);
    pc := 0;

    WHEN the enabling & aborting qualifiers aren't used ("none") the external
    C-functions will not be called to enhance simulation speed due to
    the manner in which the simulator evaluates boolean expressions

    WHEN opc_match =>
        mi := ia(pc).operand_2;
        IF (obs_input OR ma(mi, msk_x)) = (ma(mi, msk_1) OR ma(mi, msk_x)) THEN
            pc := ia(pc).operand_1; -- matched so jump
        ELSE
            pc := pc + 1; -- no match, next instruction
        END IF;
    END WHEN;

    WHEN opc_goto =>
        pc := ia(pc).operand_1;
    END WHEN;

    WHEN opc_out =>
        fifo_in <= ia(pc).operand_1; -- return identified event_id
        fifo_write <= NOT fifo_write; -- reactivate 'cyclic_buffer'
    END WHEN;

    WHEN OTHERS =>
        'opc_wait';
    END WHEN;

    WAIT ON clock, restart;
    IF restart'EVENT OR (as(1 TO 4) /= "none" AND eval(as)) THEN
        pc := 0; -- reset parser
    EXIT simulation_time;
    ELSE
        sel_edge(edge, clock) AND (es="none" OR eval(es)) THEN
            EXIT simulation_time;
        END IF;
    END IF;
END LOOP simulation_time;
END CASE;
END LOOP zero_time;
END PROCESS parser;

-- Instantiate a 10 place FIFO that connects to local signals with
-- names identical to the component port names. The local parser writes
-- into this FIFO and the top level supervisor may read or remove id's
-- from the FIFO.

fif: fifo
GENERIC MAP (10)
PORT MAP (clear_head, fifo_in, fifo_out, fifo_write, new_head, restart);

-- Process : bus_interface
--
-- Description :Connects to the multiple access bus that resides
-- under control of the supervisor. This process allows
-- the observer (if addressed) to accept and respond
-- to supervisor commands. A new queue head is signalled
-- to the supervisor to reactivate it from possible
-- suspension. On any, supervisor induced, event on
-- nice dog the 'new_event' signal is reset to false.
-- The queue head (if relevant) is now presumed known
-- to the supervisor will suspend again awaiting the
-- next event on 'new_event'.
--
-- Side Effects :Sensitive to port 'command'.
--Drives ports 'response' & 'data' when addressed and
-- signals 'nice_dog', 'restart' and 'clear_head'.
--Uses ports 'address' & 'control' and signal
--'fifo_out'.

bus_interface: PROCESS(command)
BEGIN
IF address = pco_id AND enable = en_observer THEN -- observer at this PCO?
CASE controL IS
WHEN clear => -- remove the fifo head
  clear_head <= NOT clear_head;
  response <= NOT response;
WHEN read => -- non destructive reading of fifo head
  data <= fifo_out;
  response <= NOT response;
WHEN release => -- disconnect driver from port
  data <= NULL;
  response <= NOT response;
WHEN OTHERS =>
END CASE;
ELSE
  data <= NULL;
  response <= NULL;
END IF;
IF control = nice THEN
  nice_dog <= NOT nice_dog; -- reset 'new_head' flag
ELSIF control = reset THEN
  restart <= NOT restart; -- unconfirmed, asynchronous reset
  nice_dog <= NOT nice_dog; -- all queues are cleared; reset flag
END IF;
END PROCESS bus_interface;

-- Process : watch_dog
--
-- Description :Drives the "Wired_or" resolved 'new_event' when
the fifo has a new head. The 'new_event' signal can be driven simultaneously by multiple drivers. As a reaction the supervisor might "read" one, or some, of the observer queues or poll the status of a number of timers. An event on 'nice_dog', signalled by the 'bus_interface' resets the 'new_event' flag.

-- Side Effects : Drives the 'new_event' port and is sensitive to 'nice_dog' and 'new_head'.

---

watch_dog: PROCESS(nice_dog, new_head)
BEGIN
    IF new_head'EVENT THEN
        new_event <= true;
    ELSE
        new_event <= false; -- signal to (possibly suspended) supervisor
    END IF;
END PROCESS watch_dog;

END qualifiers;

---

Configuration : qualif_obs_conf
of Entity : observer
Author(s) : O. Sies
Organization : Philips Research Laboratory, Eindhoven
Project : Testbench for P1394 Link Core
Creation Date : 09.02.1996
Description : binds architecture 'qualifiers' to entity 'observer' and configures the FIFO component.

---

CONFIGURATION observer_conf OF observer IS
    FOR qualifiers
        FOR ALL: fifo
            USE CONFIGURATION work.fifo_conf;
        END FOR;
    END FOR;
END observer_conf;
A.4 stimulator.vhdl

LIBRARY ieee; USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

LIBRARY std;
USE std.textio.ALL;
USE std.standard.ALL;

LIBRARY util;
USE util.std_logic_textio.ALL;
USE util.misc_conv_pkg.ALL;

USE work.ALL;
USE work.test_pkg.ALL;

ENTITY stimulator IS
  GENERIC (
    pco_id : positive; -- pco_id of this stimulator
    edge : edge_type; -- sensitivity to clock edge
  )
  -- Offers sequences of (possibly qualified) stimulus event vectors to the entity under test under control of the supervisor entity.

END stimulator;

LIBRARY ieee; USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

LIBRARY std;
USE std.textio.ALL;
USE std.standard.ALL;

LIBRARY util;
USE util.std_logic_textio.ALL;
USE util.misc_conv_pkg.ALL;

USE work.ALL;
USE work.test_pkg.ALL;

ENTITY stimulator IS
  GENERIC (
    pco_id : positive; -- pco_id of this stimulator
    edge : edge_type; -- sensitivity to clock edge
  )
  -- Offers sequences of (possibly qualified) stimulus event vectors to the entity under test under control of the supervisor entity.

END stimulator;

LIBRARY ieee; USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

LIBRARY std;
USE std.textio.ALL;
USE std.standard.ALL;

LIBRARY util;
USE util.std_logic_textio.ALL;
USE util.misc_conv_pkg.ALL;

USE work.ALL;
USE work.test_pkg.ALL;

ENTITY stimulator IS
  GENERIC (
    pco_id : positive; -- pco_id of this stimulator
    edge : edge_type; -- sensitivity to clock edge
  )
  -- Offers sequences of (possibly qualified) stimulus event vectors to the entity under test under control of the supervisor entity.

END stimulator;

LIBRARY ieee; USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

LIBRARY std;
USE std.textio.ALL;
USE std.standard.ALL;

LIBRARY util;
USE util.std_logic_textio.ALL;
USE util.misc_conv_pkg.ALL;

USE work.ALL;
USE work.test_pkg.ALL;

ENTITY stimulator IS
  GENERIC (
    pco_id : positive; -- pco_id of this stimulator
    edge : edge_type; -- sensitivity to clock edge
  )
  -- Offers sequences of (possibly qualified) stimulus event vectors to the entity under test under control of the supervisor entity.

END stimulator;

LIBRARY ieee; USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

LIBRARY std;
USE std.textio.ALL;
USE std.standard.ALL;

LIBRARY util;
USE util.std_logic_textio.ALL;
USE util.misc_conv_pkg.ALL;

USE work.ALL;
USE work.test_pkg.ALL;

ENTITY stimulator IS
  GENERIC (
    pco_id : positive; -- pco_id of this stimulator
    edge : edge_type; -- sensitivity to clock edge
  )
  -- Offers sequences of (possibly qualified) stimulus event vectors to the entity under test under control of the supervisor entity.

END stimulator;
width : positive; -- at least a width of '1'
e_filename : string; -- event table
v_filename : string; -- vector table
q_filename : string -- qualifier table
);
PORT (  
  address : integer;
clock : std_logic;
command : boolean;
control : control_type;
data : resolved_int BUS;
enable : enable_type;
stim_output : OUT std_logic_vector(width-1 DOWNTO 0);
response: INOUT resolved_bool BUS  
);
END stimulator;

ARCHITECTURE no_flow OF stimulator IS
-- fifo component declaration

COMPONENT fifo
  GENERIC (  
    size: positive  
  );

PORT (  
  clear_head : boolean; -- on event clear FIFO head
  fifo_in : positive; -- valid id's are '1' and up
  fifo_out : OUT natural; -- '0' included to indicate empty fifo
  fifo_write : boolean; -- on event write 'fifo_in' into FIFO
  new_head : INOUT boolean; -- event indicates new FIFO head
  restart : boolean -- reset
);
END COMPONENT;

SIGNAL clear_head : boolean; -- only valid id's can be written
SIGNAL fifo_in : positive; -- initial value = 0 : empty fifo
SIGNAL fifo_out : natural;
SIGNAL fifo_write : boolean;
SIGNAL new_head : boolean;
SIGNAL restart : boolean;

BEGIN

ARCHITECTURE no_flow OF stimulator IS
-- fifo component declaration

COMPONENT fifo
  GENERIC (  
    size: positive  
  );

PORT (  
  clear_head : boolean; -- on event clear FIFO head
  fifo_in : positive; -- valid id's are '1' and up
  fifo_out : OUT natural; -- '0' included to indicate empty fifo
  fifo_write : boolean; -- on event write 'fifo_in' into FIFO
  new_head : INOUT boolean; -- event indicates new FIFO head
  restart : boolean -- reset
);
END COMPONENT;

SIGNAL clear_head : boolean; -- only valid id's can be written
SIGNAL fifo_in : positive; -- initial value = 0 : empty fifo
SIGNAL fifo_out : natural;
SIGNAL fifo_write : boolean;
SIGNAL new_head : boolean;
SIGNAL restart : boolean;

BEGIN
-- Side Effects: Sensitive to the selected edge of 'clock', to
-- 'new_head' and any event on the local 'restart' signal.
-- Drive 'stim_output' and 'clear_head' and
-- use 'fifo_out'.

-- VHDL Sources

FILE e_file : text IS IN e_filename;
FILE v_file : text IS IN v_filename;
FILE q_file : text IS IN q_filename;

CONSTANT max_qualifiers : integer := 10;
CONSTANT max_events : integer := 100;
CONSTANT max_vectors : integer := 1000;

CONSTANT idle : std_logic_vector(width-1 DOWNTO 0) := (OTHERS => '0');

TYPE event IS (first, last);
TYPE events IS ARRAY (integer RANGE 1 TO max_events, event) OF integer;

TYPE vecrec IS
  RECORD
    vector : std_logic_vector(width-1 DOWNTO 0); -- stimulus vector
    qualifier : integer; -- pointer to qualifier
  END RECORD;

TYPE vecarr IS ARRAY (integer RANGE 0 TO max_vectors) OF vecrec;

TYPE qfarr IS ARRAY (integer RANGE 1 TO max_qualifiers) OF string(1 TO 80);

-- fills the event array 'ea' of the stimulator with indexes in 'va'
PROCEDURE read_events (
  VARIABLE f : IN text;
  ea : OUT events) IS
  VARIABLE l : line;
  VARIABLE i : integer;

  BEGIN
    WHILE NOT endfile(f) LOOP
      readl ine(f, l);
      readCl, i); -- line nr. indexes the array = event_id
      read(l, ea(i, first)); -- start of event in array 'va'
      read(l, ea(i, last)); -- end of event in array 'va'
    END LOOP;
  END read_events;

-- fills the vector array 'va' of the stimulator with vectors & indexes in 'qa'
PROCEDURE read_vectors (
  VARIABLE f : IN text;
  va : OUT vecarr) IS
  VARIABLE l : line;
  VARIABLE i : integer;

  BEGIN
    WHILE NOT endfile(f) LOOP
      readl ine(f, l);
      read(l, i); -- line nr. indexes the array = vector_id
      read(l, va(i).vector); -- stimulus vector
      read(l, va(i).qualifier); -- index in qualifier array 'qa'; 0='no qualifier'
    END LOOP;
  END read_vectors;

-- fills the qualifier array 'qa' of the stimulator with qualifier strings
PROCEDURE read_qualifiers (
  VARIABLE f : IN text;
  qa : OUT qfarr) IS
  VARIABLE l : line;
  VARIABLE i : integer;

  BEGIN
    WHILE NOT endfile(f) LOOP
      readl ine(f, l);
      read(l, i); -- line nr. indexes the array = event_id
      read(l, qa(i)); -- stimulus vector
    END LOOP;
  END read_qualifiers;
WHILE NOT endfile(f) LOOP
  readline(f, i);
  read(i, j); -- line nr. indexes the array = qualifier_id
  readstring(l, qa(i)); -- qualifier string; procedure from test_pkg
END LOOP;
END read_qualifiers;

-- function that evaluates a boolean simple expression

FUNCTION eval (s: string) RETURN boolean;
ATTRIBUTE foreign OF eval: FUNCTION IS "RB_C_Lib:Evaluate";

VARIABLE ea : events; -- references to vector array 'va'
VARIABLE va : vecarr; -- stimulus data and pointers to 'qa'
VARIABLE qa : qfarr; -- qualifier strings

BEGIN
  read_events(e_file, ea);
  read_vectors(v_file, va);
  read_qualifiers(q_file, qa);
  get_event: LOOP
    IF fifo_out = 0 THEN
      stim_output <= va(0).vector; -- idle vector from 'va'
      WAIT ON new_head;
    ELSE
      process_event: FOR i IN ea(fifo_in, first) TO ea(fifo_in, last) LOOP
        WAIT UNTIL restart'EVENT OR (sel_edge(edge, clock) AND
          (va(i).qualifier=0 OR eval(qa(va(i).qualifier))));
        IF restart'EVENT THEN
          WAIT FOR 1 NS; -- allow reset to empty fifo
          NEXT get_event;
        ELSE
          stim_output <= va(i).vector; -- apply vector to EUT
        END IF;
      END LOOP process_event;
      clear_head <= NOT clear_head; -- clear processed event_id from fifo
    END IF;
  END LOOP get_event;
END PROCESS vector_processor;

-- Instantiate a 10 place FIFO that connects to local signals with
-- names identical to the component port names. The toplevel supervisor
-- may write into or reset the FIFO and the local 'vector_processor' may
-- read and clear the FIFO head.

fif: fifo
GENERIC MAP (10)
PORT MAP (clear_head, fifo_in, fifo_out, fifo_write, new_head, restart);

-- Process : bus_interface
-- Description : Connects to the multiple access bus that resides
-- under control of the supervisor. This process allows
-- the supervisor (if addressed) to accept and respond --
-- to supervisor commands. A new queuehead is signalled
-- to the vector_processor to activate it from possible
-- suspension.
-- Side Effects : Sensitive to port 'command'.
-- Drives port 'response' and signals 'fifo_write',
-- 'fifo_in' and 'restart' when addressed. Uses ports
-- 'address', 'control' and 'data'.

bus_interface: PROCESS(command)
BEGIN
  IF address = pco_id AND enable = en_stimulator THEN -- stimulator at this PCO?
    CASE control IS
      WHEN write => -- addressed commands
        fifo_in <= data;
        fifo_write <= NOT fifo_write;
        response <= NOT response;
      WHEN OTHERS =>
        END CASE;
    ELSE -- not our PCO address
      response <= NULL;
      END IF;
    IF control = reset THEN -- one 'response' driver at a time
      restart <= NOT restart;
    END IF;
  END PROCESS bus_interface;
END no_flow;

-- Configuration : stimulator_conf
-- of Entity : stimulator
-- Author(s) : O. Sies
-- Organization : Philips Research Laboratory, Eindhoven
-- Project : Testbench for P1394 Link Core
-- Creation Date : 26.02.1996
-- Description : Binds architecture 'no_flow' to entity 'stimulator'

CONFIGURATION stimulator_conf OF stimulator IS
  FOR no_flow
    FOR ALL fifo
      USE CONFIGURATION work.fifo_conf;
    END FOR;
  END FOR;
END stimulator_conf;
A.5 timer.vhdl

LIBRARY ieee; USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

LIBRARY std;
USE std.textio.ALL;
USE std.standard.ALL;

LIBRARY util;
USE util.std_logic_textio.ALL;
USE util.misc_conv-pkg.ALL;

USE work.ALL;
USE work.test_pkg.ALL;

ENTITY timer IS
    GENERIC (
        timer_id : positive -- determines components address on bus
PORT (
  address : integer;
  command : boolean;
  control : control_type;
  data : INOUT resolved_int BUS;
  enable : enable_type;
  new_event : OUT resolved_bool := false;
  response : INOUT resolved_bool BUS
);

END timer;

ARCHITECTURE behavior OF timer IS
  SIGNAL nice_dog, trigger, stop, timeout boolean;
  SIGNAL duration : time;
BEGIN
  timer: PROCESS (trigger, stop)
  BEGIN
    IF trigger'EVENT THEN
      timeout <= true AFTER duration;
    ELSE
      timeout <= false;
    END IF;
  END PROCESS timer;

  watch_dog: PROCESS (nice_dog, timeout)
  BEGIN
    IF nice_dog'EVENT THEN
      \
new_event <= false;
ELSIF timeout THEN -- at rising edge of 'timeout' only
  new_event <= true;
ELSE
  new_event <= false;
END IF;
END PROCESS watch_dog;

PROCESS watch_dog
  -- at rising edge of 'timeout' only
BEGIN
  IF control = nice THEN
    nice_dog <= NOT nice_dog; -- clear the 'new_event' flag
  ELSIF control = reset THEN
    stop <= NOT stop; -- unconfirmed asynchronous reset
    nice_dog <= NOT nice_dog; -- all timers are cancelled; reset flag
  END IF;
END PROCESS watch_dog;

PROCESS bus_interface
BEGIN
  IF enable = en_timer AND address = timer_id THEN
    CASE control IS
      WHEN start =>
        trigger <= NOT trigger;
        response <= NOT response;
      WHEN cancel =>
        stop <= NOT stop;
        response <= NOT response;
      WHEN status =>
        IF timeout THEN
          data <= 1; -- timer expired
        ELSE
          data <= 0; -- timer not yet expired
        END IF;
        response <= NOT response;
      WHEN write =>
        duration <= to_time(data, "NS"); -- type conversion
        response <= NOT response;
      WHEN release =>
        -- 'release' data bus
        data <= NULL;
        response <= NOT response;
      WHEN OTHERS =>
        END CASE;
    ELSE
      data <= NULL;
      response <= NULL;
    END IF;
  END CASE;
END PROCESS bus_interface;

END behavior;

-- Configuration : timer_conf
-- of Entity : timer
--
-- Author(s) : O. Sies
-- Organization : Philips Research Laboratory, Eindhoven
--
-- Project : Testbench for P1394 Link Core
--
-- Creation Date : 22.02.1996
--
-- Description : Binds architecture 'behavior' to entity 'timer'.

CONFIGURATION timer_conf OF timer IS
   FOR behavior
      -- no components
   END FOR;
END timer_conf;
LIBRARY ieee; USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

LIBRARY std;
USE std.textio.ALL;
USE std.standard.ALL;
LIBRARY util;
USE util.std_logic_textio.ALL;
USE util.miscConv_pkg.ALL;
USE work.ALL;
USE work.test_pkg.ALL;

ENTITY supervisor IS
  GENERIC (
    nr_of_timers : natural; -- nr. of timer entities internally instantiated
    ttcn_filename : string; -- ttcn executable code file
    log_filename : string -- conformance log file produced in execution
  );
  PORT (
    address : INOUT integer; -- mode INout for timer reading
    command : INOUT boolean; -- mode INout for timer reading
    control : INOUT control_type; -- mode INout for timer reading
    data : INOUT resolved_int BUS;
    enable : INOUT enable_type; -- mode INout for timer reading
    new_event : INOUT resolved_bool;
    response: INOUT resolved_bool BUS
  );
END supervisor;

ARCHITECTURE behavior OF supervisor IS

COMPONENT timer
  GENERIC ( 
    timer_id : positive
  );
  PORT ( 
    address : integer; -- no defaults on component declarations
    command : boolean;
    control : control_type;
    data : INOUT resolved_int BUS;
    enable : enable_type;
    new_event : OUT resolved_bool;
    response : INOUT resolved_bool BUS
  );
END COMPONENT;

BEGIN
timer_array: FOR timer_id IN 1 TO nr_of_timers GENERATE

tim: timer
GENERIC MAP ()
timer id)
PORT MAP (address, command, control, data, enable, new_event, response);

END GENERATE;

ttcn_engine: PROCESS
-- TTCN code input file & conformance log output file
FILE ttcn_file: text IS IN ttcn_filename;
FILE log_file: text IS OUT log_filename;

-- constants specific to supervisor core
CONSTANT maxinstrs : integer := 1000; -- max. lines of TTCN executable code
CONSTANT stacksize : integer := 10; -- max. nested tree attachments

-- TTCN executable code
TYPE opcode_type IS (call, cancel, declare, goto, input, log, output, reset,
ret_cur, ret_next, start, timeout, verdict, suspend);

TYPE instruction IS
RECORD
opcode : opcode_type;
operand_1 integer;
operand_2 : integer;
operand_3 : integer;
END RECORD;

TYPE instructions IS ARRAY (integer RANGE a TO maxinstrs) OF instruction;

TYPE stack_type IS ARRAY (integer RANGE 1 TO stacksize) OF integer;

PROCEDURE read_instructions (f-: IN text;
ai : OUT instructions) IS

BEGIN
WHILE NOT endfile(f) LOOP
read(f, l);
read(l, i); -- line nr. indexes the array
read(l, opc); -- opcode string
CASE opc(2 TO 3) IS
WHEN "CA" =>
a(i).opcode := call;
read(l, ai(i).operand_1); -- call address
read(l, ai(i).operand_2); -- next on current level
WHEN "CT" =>
a(i).opcode := cancel;
read(l, ai(i).operand_1); -- timer id
WHEN "DT" =>
a(i).opcode := declare;
read(l, ai(i).operand_1); -- timer id
read(l, ai(i).operand_2); -- duration
WHEN "GT" =>
a(i).opcode := goto;
read(l, ai(i).operand_1); -- jump address
WHEN "IN" =>
a(i).opcode := input;
read(l, ai(i).operand_1); -- pc0 id
read(l, ai(i).operand_2); -- event id (0 = OTHERWISE)
read(l, ai(i).operand_3); -- next on current level
WHEN "LI" =>
a(i).opcode := log;
read(l, ai(i).operand_1); -- test case identifier
WHEN "OU" =>
ai(i).opcode := output;
read(l, ai(i).operand_1); -- pco id
read(l, ai(i).operand_2); -- event id
WHEN "RC" =>
ai(i).opcode := ret_cur;
WHEN "RN" =>
ai(i).opcode := ret_next;
WHEN "RS" =>
ai(i).opcode := reset;
WHEN "ST" =>
ai(i).opcode := start;
read(l, ai(i).operand_1); -- timer id
WHEN "TO" =>
ai(i).opcode := timeout;
read(l, ai(i).operand_1); -- timer id
read(l, ai(i).operand_2); -- next on current level
WHEN "VE" =>
ai(i).opcode := verdict;
read(l, ai(i).operand_1); -- verdict (0=FAIL, 1=PASS)
read(l, ai(i).operand_2); -- 'WT'
WHEN OTHERS =>
ai(i).opcode := suspend;
END CASE;
END LOOP;
END read_instructions;

FUNCTION verdict_string (i : integer) RETURN string IS
BEGIN
CASE i IS
WHEN 0 =>
RETURN "FAIL";
WHEN 1 =>
RETURN "PASS";
WHEN OTHERS =>
RETURN "INCONCLUSIVE";
END CASE;
END verdict_string;

BEGIN
read_instructions(ttcn_file, ia);
pc := 1;
sp := 1;
forever: LOOP
CASE ia(pc).opcode IS
-- flow control instructions
WHEN call =>
  ASSERT (sp < stacksize)
  REPORT "Stack Overflow"
  SEVERITY error;
  stack(sp) := pc + 1; -- first on next level onto stack
  sp := sp + 1;
  stack(sp) := ia(pc).operand_2; -- next on current level onto stack
  sp := sp + 1;
  pc := ia(pc).operand_1; -- jump to target address
WHEN ret_cur =>
  sp := sp - 1;
  pc := stack(sp); -- pop return index on current level
  sp := sp - 1; -- dump return index on next level
WHEN ret_next =>
  sp := sp - 2; -- dump index on current level
  pc := stack(sp); -- pop return index on next level
WHEN goto =>
  pc := ia(pc).operand_1;
-- I/O related instructions
WHEN input => -- check for a match at PCD queue
  command <= NOT command; -- reactivate processes on bus
  enable <= en_observer; -- address the observer
  address <= ia(pc).operand_1; -- at the designated PCD
  control <= read; -- and read the queue head
  data <= NULL; -- disconnect for incoming data
  WAIT ON response; -- wait for command completion
  IF data /= 0 AND -- non empty observer fifo at pco?
    (ia(pc).operand_2=data OR ia(pc).operand_2=0) -- match?
    THEN
      -- successful match
      pc := pc + 1; -- first on next level
      command <= NOT command; -- reactivate
      control <= clear; -- clear matched event at current PCD
      WAIT ON response;
      -- logging
      write(l, NOW, left, 12);
      write(l, string(" Event "));
      write(l, data);
      write(l, string("accepted from PCD queue "));
      write(l, address);
      write(l, '.');
      writeln(log_file, l);
    ELSE
      -- unsuccessful match
      pc := ia(pc).operand_3; -- next on current level
      END IF;
      control <= release; -- disconnect data driver at PCD
      command <= NOT command;
      WAIT ON response;

WHEN output => -- an output to a stimulator is always successful
  command <= NOT command;
  control <= write;
  enable <= en_stimulator;
  address <= ia(pc).operand_1;
  data <= ia(pc).operand_2;
  WAIT ON response;
  -- always successful
  pc := pc + 1;
  -- logging
  write(l, NOW, left, 12); -- formatted for up to 1000 sec. of sim.time.
  write(l, string(" Event "));
  write(l, data);
  write(l, string("offered at PCD queue "));
  write(l, address);
  write(l, '.');
  writeln(log_file, l);

WHEN suspend => -- suspends TTCN processing (simulated time)
  command <= NOT command;
  control <= nice; -- clear 'new head' signal before suspending
  WAIT FOR 1 NS; -- allow nice to clear 'new event'
  WAIT UNTIL new_event;

-- Conformance logging instructions

WHEN log =>
  test_case_ID := ia(pc).operand_1;
  write(l, string("Execution started of test case: "));
  write(l, test_case_ID);
  writeln(log_file, l);

WHEN verdict =>
  write(l, string("Execution concluded of test case: "));
  write(l, test_case_ID);
  writeln(log_file, l);
  write(l, string("The test case verdict is: "));
  write(l, verdict_string(ia(pc).operand_1)); -- 0 = FAIL, 1 = PASS
  writeln(log_file, l);
  writeln(log_file, l); -- empty line

-- Timer handling instructions

WHEN declare =>
  command <= NOT command;
  control <= write;
  enable <= en_timer;
address <= ia(pc).operand_1;
data <= ia(pc).operand_2;
WAIT ON response;
-- always successful
pc := pc + 1;
-- logging
write(l, NOW, left, 12);
write(l, string("Timer "));
write(l, address);
write(l, string("set to "));
write(l, to_time(data, "NS"); -- type conversion
write(l, ', I.T);
write(log_file, l);

WHEN start =>
command <= NOT command;
control <= start;
enable <= en_timer;
address <= ia(pc).operand_1;
data <= NULL;
WAIT ON response;
-- always successful
pc := pc + 1;
-- logging
write(l, NOW, left, 12);
write(l, string("Timer "));
write(l, address);
write(l, string("started."));
write(log_file, l);

WHEN timeout =>
command <= NOT command; -- reactivate processes on bus
enable <= en_timer; -- address the timer
address <= ia(pc).operand_1; -- at the designated PCD
control <= status; -- and check for a timeout
data <= NULL; -- disconnect for incoming data
WAIT ON response; -- wait for command completion
IF data = 1 THEN -- has the timer expired?
-- indeed, we have a timeout
pc := pc + 1; -- first on next level
command <= NOT command; -- reactivate
control <= cancel; -- reset timer
WAIT ON response;
-- logging
write(l, NOW, left, 12);
write(l, string("Timer ") );
write(l, address);
write(l, string("expired.");
write(log_file, l);
ELSE
-- no timeout yet
pc := ia(pc).operand_2; -- next on current level
END IF;
control <= release; -- disconnect data driver at PCD
command <= NOT command;
WAIT ON response;

WHEN cancel => -- cancel timer
command <= NOT command;
control <= cancel;
enable <= en_timer;
address <= ia(pc).operand_1;
data <= NULL;
WAIT ON response;
-- always successful
pc := pc + 1;
-- logging
write(l, NOW, left, 12);
write(l, string("Timer ") );
write(l, address);
write(l, string("cancelled.");
write(log_file, l);

-- Unconfirmed asynchronous instructions

WHEN OTHERS => -- reset
command <= NOT command;
control <= reset; -- affects all entities on the bus
WAIT FOR 1 NS; -- allow reset to take affect
END CASE;
END LOOP forever;
END PROCESS ttcn_engine;

END behavior;

-----------------------------------------------------------------------------------
-- Configuration: supervisor_conf
-- Of Entity: supervisor
--
-- Author(s): O. Sies
--
-- Organization: Philips Research Laboratory, Eindhoven
--
-- Project: Testbench for P1394 Link Core
--
-- Creation Date: 16.02.1996
--
-- Description: Binds architecture 'behavior' to entity 'supervisor' and
-- configures the timers contained in the supervisor.
--
-----------------------------------------------------------------------------------

CONFIGURATION supervisor_conf OF supervisor IS
  FOR behavior
    FOR timer_array
      FOR ALL: timer
        USE CONFIGURATION work.timer_conf;
      END FOR;
    END FOR;
  END FOR;
END supervisor_conf;
A.7 test_bench.vhdl

Note: All information regarding the LinkCore is bound by a non-disclosure agreement between Philips and Apple Computer, Inc. The link component declaration, and its illustrative usage in this source, have been altered to respect this agreement.

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

LIBRARY std;
USE std.textio.ALL;
USE std.standard.ALL;

LIBRARY util;
USE util.std_logic_textio.ALL;
USE util.misc_conv-pkg.ALL;

USE work.ALL;
USE work.test_pkg.ALL;
-- USE work.avtypes.ALL;

ENTITY test_bench IS
END test_bench;

ARCHITECTURE backbone OF test_bench IS
  -- the component declaration of the entity to be observed
  COMPONENT link
    PORT (  
      NotObserved1 : OUT std_logic_vector(3 DOWNTO 0);  
      UpperIn1     : std_logic_vector(32 DOWNTO 0);  
      NotObserved2 : OUT std_logic;  
      UpperIn2     : std_logic;  
      StaticIn1    : std_logic;  
      StaticIn2    : std_logic_vector(33 DOWNTO 0);  
      NotObserved3 : OUT std_logic;  
      StaticIn3    : std_logic;  
      UpperOut1    : OUT std_logic_vector(32 DOWNTO 0);  
      UpperOut2    : OUT std_logic;  
      UpperOut3    : OUT std_logic;  
      UpperOut4    : OUT std_logic;  
      StaticIn4    : std_logic;  
      StaticIn5    : std_logic;  
      StaticIn6    : std_logic;  
      StaticIn7    : std_logic;  
      StaticIn8    : std_logic_vector(9 DOWNTO 0);  
      StaticIn9    : std_logic_vector(5 DOWNTO 0);  
      StaticIn10   : std_logic;  
      NotObserved4 : OUT std_logic;  
      StaticIn11   : std_logic;  
      StaticIn12   : std_logic;  
      NotObserved5 : OUT std_logic;  
      NotObserved6 : OUT std_logic_vector(5 DOWNTO 0);  
      NotObserved7 : OUT std_logic;  
      NotObserved8 : OUT std_logic;  
      NotObserved9 : OUT std_logic;  
      NotObserved10: OUT std_logic;  
      NotObserved11: OUT std_logic;  
      UpperOut5    : OUT std_logic;  
      UpperOut6    : OUT std_logic_vector(3 DOWNTO 0);  
      StaticIn13   : std_logic;  
      StaticIn14   : std_logic;  
      NotObserved12: OUT std_logic;  
      StaticIn15   : std_logic;  
      StaticIn16   : std_logic;  
      StaticIn17   : std_logic;  
      StaticIn18   : std_logic_vector(2 DOWNTO 0);  
      StaticIn19   : std_logic;  
      NotObserved13: OUT std_logic;  
    );
END COMPONENT;
END ARCHITECTURE;
NotObserved14 : OUT std_logic;
Staticln20 : std_logic;
Staticln21 : std_logic;
Staticln22 : std_logic;
Staticln23 : std_logic;
Staticln24 : std_logic_vector(31 DOWNTO 0);
NotObserved15 : OUT std_logic_vector(19 DOWNTO 0);
NotObserved16 : OUT std_logic_vector(11 DOWNTO 0);
Staticln25 : std_logic;
NotObserved17 : OUT std_logic;
NotObserved18 : OUT std_logic;
NotObserved19 : OUT std_logic;
NotObserved20 : OUT std_logic;
NotObserved21 : OUT std_logic;
NotObserved22 : OUT std_logic;
NotObserved23 : OUT std_logic;
Staticln26 : std_logic;
UpperOut8 : OUT std_logic;
NotObserved24 : OUT std_logic;
NotObserved25 : OUT std_logic;
NotObserved26 : OUT std_logic;
NotObserved27 : OUT std_logic;
Staticln27 : std_logic;
Staticln28 : std_logic;
Staticln29 : std_logic_vector(19 DOWNTO 0);
NotObserved28 : OUT std_logic;
NotObserved29 : OUT std_logic;
NotObserved30 : OUT std_logic_vector(3 DOWNTO 0);
NotObserved31 : OUT std_logic_vector(7 DOWNTO 0);
Lowerln1 : std_logic_vector(0 TO 7);
Lowerln1 : std_logic_vector(0 TO 7);
Lowerln4 : std_logic_vector(0 TO 7);
Lowerln5 : std_logic_vector(0 TO 7);
Lowerln6 : std_logic_vector(0 TO 7);
Lowerln7 : std_logic_vector(0 TO 7);
Staticln31 : std_logic;
Staticln32 : std_logic;
Staticln33 : std_logic;
LowerClk : std_logic;
LowerClk : std_logic;
Staticln34 : std_logic;
Staticln35 : std_logic;

END COMPONENT;

-- declaration of all signals to be connected to an instance 'linkcore' of
-- component 'link'
SIGNAL NotObserved1 : std_logic_vector(3 DOWNTO 0);
SIGNAL Upperln1 : std_logic_vector(32 DOWNTO 0);
SIGNAL NotObserved2 ,UpperIn2, Staticln1 : std_logic;
SIGNAL Staticln2 : std_logic_vector(33 DOWNTO 0);
SIGNAL NotObserved3, Staticln3 : std_logic;
SIGNAL UpperOut2, UpperOut3, UpperOut4, Staticln4, Staticln5, Staticln6, Staticln7 : std_logic;
SIGNAL StaticIn8 : std_logic_vector(9 DOWNTO 0);
SIGNAL Staticln9 : std_logic_vector(5 DOWNTO 0);
SIGNAL Staticln10, NotObserved4, Staticln11, Staticln12, Staticln13 : std_logic;
SIGNAL Staticln14 : std_logic_vector(1 DOWNTO 0);
SIGNAL NotObserved5 : std_logic;
SIGNAL NotObserved6 : std_logic_vector(5 DOWNTO 0);
SIGNAL NotObserved7, NotObserved8, NotObserved9 : std_logic;
SIGNAL NotObserved10, NotObserved11, UpperOut5 : std_logic;
SIGNAL StaticIn15, Staticln16, Staticln17 : std_logic;
SIGNAL Staticln18 : std_logic_vector(2 DOWNTO 0);
SIGNAL Staticln19, NotObserved12, UpperOut7, NotObserved12, NotObserved13 : std_logic;
SIGNAL NotObserved14, Staticln20 : std_logic;
SIGNAL Staticln21, Staticln22, Staticln23 : std_logic;
SIGNAL Staticln24 : std_logic_vector(31 DOWNTO 0);
SIGNAL NotObserved15 : std_logic_vector(19 DOWNTO 0);
SIGNAL NotObserved16 : std_logic_vector(11 DOWNTO 0);
SIGNAL Staticln25, NotObserved17, NotObserved18, NotObserved19, NotObserved19 : std_logic;
SIGNAL NotObserved20, NotObserved21, NotObserved22, NotObserved23, Staticln26 : std_logic;
COMPONENT observer
GENERIC (
  pco_id : positive;  -- pco_id of this observer
  edge : edge_type;  -- sensitivity to clock edge
  width : positive;  -- at least a width of '1'
  i_filename : string;  -- parser instructions file
  m_filename : string;  -- qualifiers & masks file
);
PORT (
  address : integer;
  clock : std_logic;
  command : boolean;
  control : control_type;
  data : OUT resolved_int BUS;
  enable : enable_type;
  new_event : OUT resolved_bool;
  obs_input : std_logic_vector(width-1 DOWNTO 0);
  response : INOUT resolved_bool BUS
);
END COMPONENT;

-- the stimulator component declaration

COMPONENT stimulator
GENERIC (
  pco_id : positive;  -- pco_id of this stimulator
  edge : edge_type;  -- sensitivity to clock edge
  width : positive;  -- at least a width of '1'
  e_filename : string;  -- event table
  v_filename : string;  -- vector table
  q_filename : string;  -- qualifier table
);
PORT (
  address : integer;
  clock : std_logic;
  command : boolean;
  control : control_type;
  data : resolved_int BUS;
  enable : enable_type;
  stim_output : OUT std_logic_vector(width-1 DOWNTO 0);
  response : INOUT resolved_bool BUS
);
END COMPONENT;

-- the supervisor component declaration

COMPONENT supervisor
GENERIC (
  nr_of_timers : natural;  -- nr. of timer entities internally instantiated
  ttcn_filename : string;  -- ttcn executable code file
  log_filename : string;  -- conformance log file produced in execution
);
PORT (
  address : INOUT integer;
  command : INOUT boolean;
  control : INOUT control_type;
  data : INOUT resolved_int BUS;
  enable : INOUT enable_type;
  new_event : INOUT resolved_bool;
)
--- Instantiate "obs1" that is to observe the RFD & Status interface. ---
-- Through the generics this observer is configured at 'pco_id'=1 to sample data a 43 bit wide input signal (UpperOut1, UpperOut2, UpperOut3, UpperOut4, UpperOut5, UpperOut6, UpperOut7 & UpperOut8) at the rising edge of LowerClk and to use the datafiles indicated.

obs1: observer
GENERIC MAP (1, rising, 43,
"/user/sies/VHDL/vectors/ut_parser.dat",
"/user/sies/VHDL/vectors/ut_masks.dat"
)
PORT MAP (clock => LowerClk,
obs_input(42 DOWNTO 10) => UpperOut1, obs_input(9) => UpperOut2,
ob_input(8) => UpperOut3, obs_input(7) => UpperOut4,
ob_input(6) => UpperOut5, obs_input(2 DOWNTO 1) => UpperOut6, obs_input(0) => UpperOut8,
address => address, command => command, control => control, data => data,
enable => enable, new_event => new_event, response => response);

--- Instantiate "obs2" that is to observe the Phy/Link interface. ---
-- Through the generics this observer is configured at 'pco_id'=2 to sample data on a 4 bit wide input signal (LowerOut1, LowerOut2, LowerOut3, LowerOut4, LowerOut5 & LowerOut6) at the rising edge of UpperClk and to use the datafiles indicated.

obs2: observer
GENERIC MAP (2, rising, 22,
"/user/sies/VHDL/vectors/lt_parser.dat",
"/user/sies/VHDL/vectors/lt_masks.dat"
)
PORT MAP (clock => UpperClk,
ob_input(21) => LowerOut1, obs_input(20 DOWNTO 13) => LowerOut2,
ob_input(12 DOWNTO 11) => LowerOut3, obs_input(10 DOWNTO 3) => LowerOut4,
ob_input(2 DOWNTO 1) => LowerOut5, obs_input(0) => LowerOut6,
address => address, command => command, control => control, data => data,
enable => enable, new_event => new_event, response => response);

--- Instantiate "stim1" that is to stimulate the ATF interface. ---
-- Through generics this stimulator is configured at 'pco_id'=1 to offer data to a 34 bit wide output signal (UpperIn1 & UpperIn2) at the rising edge of LowerClk and to use the datafiles indicated.

stim1: stimulator
GENERIC MAP (1, rising, 34,
"/user/sies/VHDL/vectors/ut_events.dat",
"/user/sies/VHDL/vectors/ut_vectors.dat",
"/user/sies/VHDL/vectors/ut_qualifiers.dat"
PORT MAP (
clock => LowerClk,
stim_output(33 DOWNTO 1) => UpperIn1, stim_output(0) => UpperIn2,
address => address, command => command, control => control, data => data,
enable => enable, response => response
);

-- instantiate "stim2" that is to stimulate the PHY/Link interface.
-- Through generics this stimulator is configured at 'pco_id'=2 to offer
-- data to a 10 bit wide output signal (LowerIn1 & LowerIn2) at the
-- rising edge of UpperClk and to use the datafiles indicated.
stim2: stimulator
GENERIC MAP (
2, rising, 10,
"/user/sies/VHDL/vectors/lt_events.dat",
"/user/sies/VHDL/vectors/lt_vectors.dat",
"/user/sies/VHDL/vectors/lt_qualifiers.dat"
)
PORT MAP (
clock => UpperClk,
stim_output(9 DOWNTO 8) => LowerIn1, stim_output(7 DOWNTO 0) => LowerIn2,
address => address, command => command, control => control, data => data,
enable => enable, response => response
);

-- instantiate a supervisor with 2 internal timers and specify the
-- names & locations of the input executable code file and the output
-- conformance log file to produce during execution.
sup: supervisor
GENERIC MAP (
2,
"/user/sies/VHDL/vectors/ttcn_code.dat",
"/user/sies/VHDL/vectors/conformance.log"
)
PORT MAP (
address, command, control, data, enable, new_event, response);

-- instantiate linkcore under test, using positional association
linkcore: link
PORT MAP {
NotObserved1, UpperIn1, NotObserved2, UpperIn2, StaticIn1, StaticIn2, NotObserved3,
StaticIn3, UpperOut1, UpperOut2, StaticIn4, UpperOut3, UpperOut4, StaticIn5, StaticIn6,
StaticIn7, StaticIn8, StaticIn9, StaticIn10, NotObserved4, StaticIn11, StaticIn12,
StaticIn13, StaticIn14, NotObserved5, NotObserved6, StaticIn15, StaticIn16,
NotObserved7, NotObserved8, NotObserved9, NotObserved10, NotObserved11, UpperOut5,
UpperOut6, StaticIn17, StaticIn18, StaticIn19, NotObserved12, UpperOut7, NotObserved12,
NotObserved13, NotObserved14, StaticIn20, StaticIn21, StaticIn22,
StaticIn23, StaticIn24, NotObserved15, NotObserved16, StaticIn25,
NotObserved17, NotObserved18, NotObserved19, NotObserved19, NotObserved20,
NotObserved21, NotObserved22, NotObserved23, StaticIn26, UpperOut8, NotObserved24,
NotObserved25, NotObserved26, NotObserved27, StaticIn27, StaticIn28, StaticIn29,
StaticIn30, NotObserved28, NotObserved29, NotObserved30, NotObserved31, LowerIn2,
LowerIn1, LowerOut1, LowerOut2, StaticIn32, StaticIn33, UpperClk, LowerClk, StaticIn34,
StaticIn35
);

clocks: PROCESS
BEGIN
  UpperClk <= '1';
  LowerClk <= '1';
  WAIT FOR 10 ns;
  UpperClk <= '0';
WAIT FOR 10 ns;
UpperClk <= '1';
LowerClk <= '0';
WAIT FOR 10 ns;
UpperClk <= '0';
WAIT FOR 10 ns;
END PROCESS clocks;

-- statics: EUT inputs not dynamically driven by stimulators

FILE f : text IS IN "./user/sies/VHDL/vectors/statics.dat";
VARIABLE l: line;

BEGIN
readline(f, l);
read(l, v1);
StaticIn1 <= v1;
readline(f, l);
read(l, v34);
StaticIn2 <= v34;
readline(f, l);
read(l, v1);
StaticIn3 <= v1;
readline(f, l);

statics: PROCESS

wait;
VHDL Sources

read(l, v1);
Staticln4 <= v1;
readline(f, l);
read(l, v1);
Staticln5 <= v1;
readline(f, l);
read(l, v1);
Staticln6 <= v1;
readline(f, l);
read(l, v1);
Staticln7 <= v1;
readline(f, l);
read(l, v10);
Staticln8 <= v10;
readline(f, l);
read(l, v6);
Staticln9 <= v6;
readline(f, l);
read(l, v1);
Staticln10 <= v1;
readline(f, l);
read(l, v1);
Staticln11 <= v1;
readline(f, l);
read(l, v1);
Staticln12 <= v1;
readline(f, l);
read(l, v1);
Staticln13 <= v1;
readline(f, l);
read(l, v1);
Staticln14 <= v2;
readline(f, l);
read(l, v1);
Staticln15 <= v1;
readline(f, l);
read(l, v1);
Staticln16 <= v1;
readline(f, l);
read(l, v1);
Staticln17 <= v1;
readline(f, l);
read(l, v3);
Staticln18 <= v3;
readline(f, l);
read(l, v1);
Staticln19 <= v1;
readline(f, l);
read(l, v1);
Staticln20 <= v1;
readline(f, l);
read(l, v1);
Staticln21 <= v1;
readline(f, l);
read(l, v1);
Staticln22 <= v1;
readline(f, l);
read(l, v1);
Staticln23 <= v1;
readLine(f, L);
read(l, v32);
Staticln24 <= v32;

readLine(f, l);
read(l, v1);
Staticln25 <= v1;

readLine(f, l);
read(l, v1);
Staticln26 <= v1;

readLine(f, l);
read(l, v1);
Staticln27 <= v1;

readLine(f, l);
read(l, v1);
Staticln28 <= v1;

readLine(f, l);
read(l, v4);
Staticln29 <= v4;

readLine(f, l);
read(l, v8);
Staticln30 <= v8;

readLine(f, l);
read(l, v1);
Staticln31 <= v1;

readLine(f, l);
read(l, v1);
Staticln32 <= v1;

readLine(f, l);
read(l, v1);
Staticln33 <= v1;

readLine(f, l);
read(l, v1);
Staticln34 <= v1;

readLine(f, l);
read(l, v1);
Staticln35 <= v1;

WAIT;  -- suspend indefinitely

END PROCESS statics;

END backbone;

--------------------------------------------------------------------------
-- Configuration : behav_tbench_conf
-- of Entity : observer
--
-- Author(s) : O. Sies
-- Organization : Philips Research Laboratory, Eindhoven
--
-- Project : Testbench for P1394 Link Core
--
-- Creation Date : 22.02.1996
--
-- Description : Binds architecture 'backbone' to entity 'test_bench'
-- and configures the components in this architecture
--
--------------------------------------------------------------------------
CONFIGURATION test_bench_conf OF test_bench IS
FOR backbone
FOR ALL : observer
  USE CONFIGURATION work.observer_conf;
VHDL Sources

END FOR;
FOR ALL: stimulator
USE CONFIGURATION work.stimulator_conf;
END FOR;
FOR ALL: supervisor
USE CONFIGURATION work.supervisor_conf;
END FOR;

-- linkcore is configured by defaults?
END FOR;

END test_bench_conf;
Appendix B

TTCN compiler example

B.1 Selection of three test cases from a P1394 suite

Significant info from TTCN-input file for first pass compilation:

-- use these declarations to build symbol table

$Begin_PCO_Oels
$PCO_Id TRANS
$PCO_Id PHY
$PCO_Id CONTR
$End_PCO_Oels

$Begin_TimerOels
$TimerId no_output_timer $Duration 10 $Unit sec
$TimerId testcase_timer $Duration 200 $Unit sec
$End_TimerOels

$POU_TypeOefs
$POU_Id spontaneous
$POU_Id timer
$POU_Id LK_DATA_REQ_0
$POU_Id LK_DATA_REQ_1
$POU_Id LK_DATA_RESP_0
$POU_Id LK_DATA_RESP_1
$POU_Id PH_ARB_CONF_0
$POU_Id PH_ARB_CONF_1
$POU_Id PH_DATA_IND_SUBGAP
$POU_Id PH_DATA_IND_VAL_ACK
$POU_Id PH_DATA_INDINVAL_ACK
$POU_Id PH_DATA_IND_000xxxx
$POU_Id PH_DATA_IND_001xxxx
$POU_Id PH_DATA_IND_010xxxx
$POU_Id PH_DATA_IND_011xxxx
$POU_Id PH_DATA_IND_10000xx
$POU_Id PH_DATA_IND_1000100
$POU_Id PH_DATA_IND_1001011
$POU_Id PH_DATA_IND_1001100
$POU_Id PH_DATA_IND_1001101
$POU_Id PH_DATA_IND_1001110
$POU_Id PH_DATA_IND_1001111
$POU_Id LK_CONTROL_REQ_0
$POU_Id LK_CONTROL_REQ_1
$POU_Id LK_DATA_CONF_00
$POU_Id LK_DATA_CONF_01
$POU_Id LK_DATA_CONF_10
$POU_Id LK_DATA_CONF_11
$POU_Id LK_DATA_IND_00
$POU_Id LK_DATA_IND_01
$POU_Id LK_DATA_IND_10
$POU_Id LK_DATA_IND_11
$POU_Id PH_ARB_REQ_0
$POU_Id PH_ARB_REQ_1
$POU_Id PH_DATA_REQ_000
$POU_Id PH_DATA_REQ_010
$POU_Id PH_DATA_REQ_011
$POU_Id PH_DATA_REQ_10x
TTCN compiler example

$PDU_Id PH_DATA_REQ_11x
$PDU_Id LK_CONTROL_CONF
$PDU_Id LK_EVENT_IND_01
$PDU_Id LK_EVENT_IND_10
$PDU_Id LK_EVENT_IND_11
$End_PDU_TypeDefs

-- use behavioral section to construct testcases

$TestCaseId test_1
  [1] + ts_l4.3.1
  [2] PHY ? PH_ARB_CONF_1
  [4] TRANS ! LK_DATA_RESP_1
  [5] PHY ? PH_DATA_REQ_10x $VerdictId PASS

$TestCaseId test_2
  [1] + ts_l4.3.1
  [2] CONTR ! LK_CONTROL_REQ_0
  [5] PHY ? PH_ARB_REQ_1 $VerdictId PASS

$TestCaseId test_3
  [1] + ts_l4.3.1
  [2] CONTR ! LK_CONTROL_REQ_1
  [4] PHY ! PH_DATA_IND INV_Ack
  [5] START no_output_timer
  [7] PHY ! PH_ARB_CONF_1
  [8] START no_output_timer
  [9] ? TIMEOUT no_output_timer
  [10] TRANS ! LK_DATA_REQ_0
  [12] ? TIMEOUT no_output_timer
  [13] TRANS ! spontaneous
  [14] START no_output_timer
  [15] ? TIMEOUT no_output_timer $VerdictId PASS

$TestStepId ts_l4.3.1
  [1] + ss
  [2] CONTR ! LK_CONTROL_REQ_0
  [5] PHY ? PH_ARB_REQ_1

$TestStepId ss
  [1] START testcase_timer
  [2] CONTR ! LK_CONTROL_REQ_1
  [3] START no_output_timer
  [4] + gobble_one
  [5] CANCEL no_output_timer

$Header gobble_one
  [1] TRANS ? OTHERWISE
  [1] PHY ? OTHERWISE
  [1] CONTR ? OTHERWISE
  [1] ? TIMEOUT no_output_timer

$DefaultId general_default
  [1] TRANS ? OTHERWISE $VerdictId FAIL
  [1] PHY ? OTHERWISE $VerdictId FAIL
  [1] CONTR ? OTHERWISE $VerdictId FAIL
  [1] ? TIMEOUT testcase_timer $VerdictId FAIL
B.2 After first compiler pass

Significant info after first pass of compilation:

-- fill symbol tables

$Begin_PCO_Dcls
  1. TRANS
  2. PHY
  3. CONTR
$End_PCO_Dcls

$Begin_TimerDcls
  1. no_output_timer 10 sec
  2. testcase_timer 200 sec
$End_TimerDcls

$POU_TypeDefs

0, OTHERWISE -- add an otherwise identifier

1. spontaneous
2. timer
3. LK_DATA_REQ_0
4. LK_DATA_REQ_1
5. LK_DATA_RESP_0
6. LK_DATA_RESP_1
7. PH_ARB_CONF_0
8. PH_ARB_CONF_1
9. PH_DATA_IND_SUBGAP
10. PH_DATA_IND_VAL_ACK
11. PH_DATA_INDINVAL_ACK
12. PH_DATA_IND_000xxxx
13. PH_DATA_IND_001xxxx
14. PH_DATA_IND_010xxxx
15. PH_DATA_IND_011xxxx
16. PH_DATA_IND_10000xx
17. PH_DATA_IND_1001001
18. PH_DATA_IND_1001010
19. PH_DATA_IND_1001011
20. PH_DATA_IND_1000100
21. PH_DATA_IND_1000110
22. PH_DATA_IND_1000111
23. LK_CONTROL_REQ_0
24. LK_CONTROL_REQ_1
25. LK_DATA_CONF_00
26. LK_DATA_CONF_01
27. LK_DATA_CONF_10
28. LK_DATA_IND_00
29. LK_DATA_IND_01
30. LK_DATA_IND_10
31. LK_DATA_IND_11
32. PH_ARB_REQ_0
33. PH_ARB_REQ_1
34. PH_DATA_REQ_000
35. PH_DATA_REQ_001
36. PH_DATA_REQ_011
37. PH_DATA_REQ_10x
38. PH_DATA_REQ_11x
39. LK_CONTROL_CONF
40. LK_EVENT_IND_01
41. LK_EVENT_IND_10
42. LK_EVENT_IND_11
$End_POU_TypeDefs

-- substitute integer id's from symbol table

$TestCaseId test 1
  [1] * ts L4_3_1
  [2] 2 ! 6
  [6] 1 ! 6
$VerdictId PASS
$TestCaseld test 2
[1] + ts_L4_5_1
[2] 3! 23

$TestCaseld test 3
[1] + ts_L4_5_1
[2] 3! 26
[5] START 1
[6] ? TIMEOUT 1
[7] 2 ! 8
[8] START 1
[9] ? TIMEOUT 1
[10] 1 ! 3
[11] START 1
[12] ? TIMEOUT 1
[13] 1 ! 1
[14] START 1

$TestStepld ts_L4_3_1
[1] + ss
[2] 3 ! 23
[4] 2 ! 22
[5] 2 ? 33

$TestStepld ss
[1] START 2
[2] 3 ! 24
[3] START 1
[4] + gobble_one
[5] CANCEL 1

$Header gobble_one
[1] 1 ? 0
[1] 2 ? 0
[1] 3 ? 0
[1] ? TIMEOUT 1

$Defaultld general_default
[1] 1 ? 0 $Verdictld FAIL
[1] 2 ? 0 $Verdictld FAIL
[1] 3 ? 0 $Verdictld FAIL
[1] ? TIMEOUT 2 $Verdictld FAIL

B.3 After second pass of compilation

Significant info after second pass of compilation:

-- program timers in NS units

$Begin TimerDcls
declare 1 10e9
declare 2 200e9

-- add default as explicit attachment for inputs and attachments
-- in cases (all levels) and trees & steps (not the root level)
-- add them at tail (=last of the alternatives)

$TestCaseld test 1
[1] + ts_L4_5_1
[2] 2 ! 8
[4] 1 ! 6
[5] + general_default
[3] + general_default
[1] + general_default
$TestCaseId$ test_2
(1) + ts_l4_5_1
(2) 3 ? 23
(3) 3 ? 39
(4) 2 ? 22
(5) 2 ? 33 + general_default
(6) + general_default
(7) + general_default
(8) + general_default
(9) + general_default
(10) + general_default
(11) + general_default
$VerdictId$ PASS

$TestCaseId$ test_3
(1) + ts_l4_5_1
(2) 3 ? 24
(3) 3 ? 39
(4) 2 ? 22
(5) START 1
(6) ? TIMEOUT 1
(7) 2 ? 8
(8) START 1
(9) ? TIMEOUT 1
(10) 1 ! 3
(11) START 1
(12) ? TIMEOUT 1
(13) 1 ! 1
(14) START 1
(15) ? TIMEOUT 1 $VerdictId$ PASS
(16) + general_default
(17) + general_default
(18) + general_default
(19) + general_default
(20) + general_default
(21) + general_default
$VerdictId$ FAIL

$TestStepId$ ts_l4_3_1
(1) + ss
(2) 3 ? 23
(3) 3 ? 39
(4) 2 ? 22
(5) 2 ? 33
(6) + general_default
(7) + general_default
-- no default at (1)!

$TestStepId$ ss
(1) START 2
(2) 3 ? 24
(3) START 1
(4) + gobble_one
(5) CANCEL 1
(6) + general_default

$Header$ gobble_one
(1) 1 ? 0
(1) 2 ? 0
(1) 3 ? 0
(1) ? TIMEOUT 1
-- no default at (1)!

$DefaultId$ general_default
(1) 1 ? 0 $VerdictId$ FAIL
(1) 2 ? 0 $VerdictId$ FAIL
(1) 3 ? 0 $VerdictId$ FAIL
(1) ? TIMEOUT 2 $VerdictId$ FAIL
-- no default in default!
TTCN compiler example

B.4 After third pass of compilation

Significant info after third pass of compilation:

-- close levels with verdicts, waits, en goto's
-- collect (next on current level) labels for attachments and inputs
-- place labels of style <tree_level_sequence>
-- add reset, goto -> (next_test) instructions at end of each test case

$Begin_TimerDcls
declare 1 10e9
declare 2 200e9
$Begin_TimerDcls
declare 1 10e9
declare 2 200e9
$TestCases1
<1_1_1>
[1] + ts L4_3_1
(1) + (next_at) := <1_1_2>
[2] 2 1 8
[3] 1 2 31
[4] 1 1 6
[5] 2 1 37
$VerdictId PASS
go to
<1_5_2>
[5] + general_default
reset
go to
<1_5_3>
wait
go to
<1_3_2>
[3] + general_default
reset
go to
<1_3_3>
wait
go to
<1_1_2>
[1] + general_default
reset
go to
<1_1_3>
wait
go to
<1_1_1>

-- without TEST_CASE_ERROR verdicts
$TestCases2
<2_1_1>
[1] + ts L4_3_1
(1) + (next_at) := <2_1_2>
[2] 3 1 25
[3] 3 2 39
[4] 2 1 22
[5] 2 1 33
$VerdictId PASS
reset
go to
<2_5_2>
[5] + general_default
reset
go to
<2_5_3>
wait
go to
<2_3_2>
[3] + general_default
reset
go to
<2_3_3>
wait
go to
<2_1_2>
[1] + general_default
reset
go to
<2_1_3>
wait
go to
<2_1_1>

$TestCases3
<3_1_1>
[1] + ts L4_3_1
(1) + (next_at) := <3_1_2>
[2] 3 1 24
[3] 3 2 39
[4] 2 1 11
[5] START
<3_6_1>
[7] 2 1 8
(1) + (next_at) := <3_6_2>
<3_9_1> [8] START 1
<9> ? TIMEOUT 1
<10> 1 ! 1
<11> START 1
<12> ? TIMEOUT 1
<13> 1 ! 1
<14> START 1
<15> ? TIMEOUT 1

$VerdictId PASS reset goto

<3_15_1> -> (next_at_15) := <3_15_2>

<3_15_2> [15] + general_default reset goto
<3_15_3> goto
<3_12_2> [12] + general_default reset goto
<3_11_1> goto
<3_9_2> [9] + general_default reset goto
<3_9_3> goto
<3_6_2> [6] + general_default reset goto
<3_3_2> goto
<3_3_3> goto
<3_1_2> [1] + general_default reset goto
<3_1_3> goto
<3_1_1> goto

<stdio>
goto <stop>

$TestStepId ts_L4_3_1
<4_1_1> [1] + ss
<4_3_1> [3] 3 ! 23
<4_5_1> [3] 3 ? 33
<4_5_2> [4] 2 ! 22
<4_5_3> [5] 2 ? 33

return_next
<4_5_2> [5] + general_default return_next
<4_5_3> goto
<4_3_2> [3] + general_default return_next
<4_3_3> goto
<4_1_2> return_current

<stdio>

$TestStepId ss
<5_1_1> [1] START 2
<5_4_1> [3] START 1
<4_5_1> [4] + gobble_one
<5_4_2> [4] + general_default return_next
<5_4_3> goto
<5_4_1> -> {next_at_4} := <5_4_2>
<5_4_2> -> {last_at_4} := <5_4_3>
<5_4_3> goto
<5_4_1> -> {first_at_4} := <5_4_1>
TTCN compiler example

```cpp
$Header gobble_one
<6_1_1>  (1) 1 ? 0
      return next
<6_1_2>  (1) 2 ? 0
      return next
<6_1_3>  (1) 3 ? 0
      return next
<6_1_4>  (1) ? TIMEOUT 1
      return next
<6_1_5>  return_current

$DefaultId general_default
<7_1_1>  (1) 1 ? 0
      $VerdictId FAIL
      return next
<7_1_2>  (1) 2 ? 0
      $VerdictId FAIL
      return next
<7_1_3>  (1) 3 ? 0
      $VerdictId FAIL
      return next
<7_1_4>  (1) ? TIMEOUT 2
      $VerdictId FAIL
      return next
<7_1_5>  return_current
```

B.5 After fourth pass of compilation

Significant info after fourth pass of compilation:

---

-- discard the indentation level
-- replace tree identifiers with labels
-- fully substitute labels
-- include logging instructions at each test case start

```cpp
$Begin_TimerOcls
declare 1 10e9
declare 2 200e9

$TestCaseId test_1
<test_1>  log 1
<1_1_1>  + <6_1_1> <1_1_2>
<1_3_1>  1 ? 31 <1_3_2>
      output 1 6
<1_5_1>  2 ? 37 <1_5_2>
      $VerdictId PASS
      reset
      goto <test_2>
<1_5_2>  + <7_1_1> <1_5_3>
      reset
      goto <test_2>
<1_5_3>  wait
      goto <1_5_1>
<1_3_2>  + <7_1_1> <1_3_3>
      reset
      goto <test_2>
<1_3_3>  wait
      goto <1_3_1>
<1_1_2>  + <7_1_1> <1_1_3>
      reset
      goto <test_2>
<1_1_3>  wait
      goto <1_1_1>

$TestCaseId test_2
```

```cpp
```
\[ \text{log} 2 \]
\[ + 4 \cdot 1 \cdot 2 \]
\[ 3 \cdot 25 \]
\[ 3 \cdot 39 \cdot 2 \cdot 3 \cdot 2 \]
\[ 2 \cdot 42 \]
\[ 2 \cdot 33 \cdot 2 \cdot 5 \cdot 2 \]
\[ \text{VerdictId} \text{ PASS} \]
\[ \text{reset} \]
\[ \text{goto} <\text{test}_3> \]
\[ + <7 \cdot 1 \cdot 1> <2 \cdot 5 \cdot 3> \]
\[ \text{reset} \]
\[ \text{goto} <\text{test}_3> \]
\[ \text{wait} \]
\[ \text{goto} <2 \cdot 5 \cdot 1> \]
\[ + <7 \cdot 1 \cdot 1> <2 \cdot 3 \cdot 3> \]
\[ \text{reset} \]
\[ \text{goto} <\text{test}_3> \]
\[ \text{wait} \]
\[ \text{goto} <2 \cdot 3 \cdot 1> \]
\[ + <7 \cdot 1 \cdot 1> <2 \cdot 1 \cdot 3> \]
\[ \text{reset} \]
\[ \text{goto} <\text{test}_3> \]
\[ \text{wait} \]
\[ \text{goto} <2 \cdot 1 \cdot 1> \]

\[ \text{TestCaseld test_3} \]
\[ \text{start 1} \]
\[ \text{timeout 1} <3 \cdot 6 \cdot 2> \]
\[ 2 \cdot 18 \]
\[ \text{start 1} \]
\[ \text{timeout 1} <3 \cdot 9 \cdot 2> \]
\[ 1 \cdot 13 \]
\[ \text{start 1} \]
\[ \text{timeout 1} <3 \cdot 12 \cdot 2> \]
\[ 1 \cdot 11 \]
\[ \text{start 1} \]
\[ \text{timeout 1} <3 \cdot 15 \cdot 2> \]
\[ \text{VerdictId} \text{ PASS} \]
\[ \text{reset} \]
\[ \text{goto} <\text{stop}> \]
\[ + <7 \cdot 1 \cdot 1> <3 \cdot 15 \cdot 3> \]
\[ \text{reset} \]
\[ \text{goto} <\text{stop}> \]
\[ \text{wait} \]
\[ \text{goto} <3 \cdot 15 \cdot 1> \]
\[ + <7 \cdot 1 \cdot 1> <3 \cdot 15 \cdot 3> \]
\[ \text{reset} \]
\[ \text{goto} <\text{stop}> \]
\[ \text{wait} \]
\[ \text{goto} <3 \cdot 15 \cdot 2> \]
\[ + <7 \cdot 1 \cdot 1> <3 \cdot 9 \cdot 3> \]
\[ \text{reset} \]
\[ \text{goto} <\text{stop}> \]
\[ \text{wait} \]
\[ \text{goto} <3 \cdot 9 \cdot 3> \]
\[ + <7 \cdot 1 \cdot 1> <3 \cdot 6 \cdot 3> \]
\[ \text{reset} \]
\[ \text{goto} <\text{stop}> \]
\[ \text{wait} \]
\[ \text{goto} <3 \cdot 6 \cdot 3> \]
\[ + <7 \cdot 1 \cdot 1> <3 \cdot 3 \cdot 3> \]
\[ \text{reset} \]
\[ \text{goto} <\text{stop}> \]
\[ \text{wait} \]
\[ \text{goto} <3 \cdot 3 \cdot 3> \]
\[ + <7 \cdot 1 \cdot 1> <3 \cdot 1 \cdot 3> \]
\[ \text{reset} \]
\[ \text{goto} <\text{stop}> \]
\[ \text{wait} \]
\[ \text{goto} <3 \cdot 1 \cdot 1> \]
TTCN compiler example

<stop> goto <stop>

$TestStepId ts_4_3_1
<4_1_1> + <5_1_1> <4_1_2>
3 ! 23
<4_3_1> 3 ? 39 <4_3_2>
<4_5_1> 2 ? 33 <4_5_2>
return_next
<4_5_2> + <7_1_1> <4_5_3>
return_next
<4_5_3> wait
goto <4_5_1>
<4_3_2> + <7_1_1> <4_3_3>
return_next
<4_3_3> wait
goto <4_3_1>
<4_1_2> return_current

$TestStepId as
<5_1_1> START 2
3 ! 24
START 1
<5_4_1> + <6_1_1> <5_4_2>
CANCEL 1
return_next
<5_4_2> + <7_1_1> <5_4_3>
return_next
<5_4_3> wait
goto <5_4_1>

$Header gobble_one
<6_1_1> 1 ? 0 <6_1_2>
return_next
<6_1_2> 2 ? 0 <6_1_3>
return_next
<6_1_3> 3 ? 0 <6_1_4>
return_next
<6_1_4> ? TIMEOUT 1 <6_1_5>
return_next
<6_1_5> return_current

$DefaultId
<7_1_1> 1 ? 0 <7_1_2>
$VerdictId FAIL
return_next
<7_1_2> 2 ? 0 <7_1_3>
$VerdictId FAIL
return_next
<7_1_3> 3 ? 0 <7_1_4>
$VerdictId FAIL
return_next
<7_1_4> ? TIMEOUT 2 <7_1_5>
$VerdictId FAIL
return_next
<7_1_5> return_current
B.6 After fifth pass of compilation

Significant info after fifth pass of compilation:

Direct mappable TTCN constructs:
-- substitute "CA" for "\n
-- substitute "TO" for "? TIMEOUT"
-- substitute "IN" for "?"
-- substitute "OU" for "!"
-- substitute "VE" for "\nVerdictld", "1" for "PASS" and "0" for "FAIL"
-- substitute "CT" for "CANCEL"
-- substitute "ST" for "START"

Other instructions:
-- substitute "GT" for "goto"
-- substitute "RC" for "return_current"
-- substitute "RN" for "return_next"
-- substitute "WT" for "wait"
-- substitute "RS" for "reset"
-- substitute "DT" for "declare"
-- substitute "LI" for "log"

-- remove last (comment) $keywords

<test_1>
DT 1 10^49
DT 2 200^49
LI 1
<1_1_1>
CA <=.1_1> <1_1_2>
OU 2 8
<1_3_1>
IN 1 31 <1_3_2>
OU 1 6
<1_5_1>
IN 2 37 <1_5_2>
VE 1
RS
GT <test_2>
RS
<1_5_3>
CA <=5_1_1> <1_5_3>
RS
GT <test_2>
WT
<1_5_3>
CA <=5_1_1> <1_5_3>
RS
GT <test_2>
WT
<1_5_3>
CA <=5_1_1> <1_5_3>
RS
GT <test_2>
WT
<1_5_3>
CA <=5_1_1> <1_5_3>
RS
<test_2>
LT 2
<2_1_1>
CA <=4_1_1> <2_1_2>
OU 3 23
<2_3_1>
IN 3 39 <2_3_2>
OU 2 22
<2_5_1>
IN 2 33 <2_5_2>
VE 1
RS
GT <test_3>
RS
<2_5_2>
CA <=5_1_1>
RS
GT <test_3>
WT
<2_5_3>
CA <=5_1_1> <2_3_3>
RS
GT <test_3>
WT
<2_3_3>
CA <=5_1_1> <2_3_3>
RS
GT <test_3>
WT
<2_1_3>
CA <=5_1_1> <2_1_3>
RS
GT <test_3>
WT
<2_1_3>
CA <=5_1_1> <2_1_3>
RS
GT <test_3>
WT
<test_3>
LT 3
<3_1_1>
CA <=4_1_1> <3_1_2>
<6_1_4> TO 1 <6_1_5> 
RN
<6_1_5> RC
<7_1_1> IN 1 0 <7_1_2> 
VE 0 
RN
<7_1_2> IN 2 0 <7_1_3> 
VE 0 
RN
<7_1_3> IN 3 0 <7_1_4> 
VE 0 
RN
<7_1_4> TO 2 <7_1_5> 
VE 0 
RN
<7_1_5> RC

B.7 End result of compilation

Significant info after sixth pass of compilation:

---

-- add line numbers and build a "label" symbol table containing line numbers

-- substitute labels in code with absolute addresses from symbol table

```
1 DT 1 10e9
2 DT 2 200e9
3 LI 1
4 CA 101 22
5 OU 2 0
6 IN 1 31 17
7 OU 1 6
8 IN 2 37 12
9 VE 1
10 RS
11 GT 27
12 CA 135 15
13 RS
14 GT 27
15 WT
16 GT 8
17 CA 135 20
18 RS
19 GT 27
20 WT
21 GT 6
22 CA 135 25
23 RS
24 GT 27
25 WT
26 GT 4
27 LI 2
28 CA 101 46
29 OU 3 23
30 IN 3 39 41
31 OU 2 22
32 IN 2 33 36
33 VE 1
34 RS
35 GT 51
36 CA 135 39
37 RS
38 GT 51
39 WT
40 GT 32
41 CA 135 44
42 RS
43 GT 51
44 WT
45 GT 30
46 CA 135 49
47 RS
48 GT 51
49 WT
50 GT 28
```
TTCN compiler example

51 LI 3
52 CA 101 95
53 OU 3 24
54 IN 3 39 90
55 OU 2 11
56 ST 1
57 TO 1 85
58 OU 2 8
59 ST 1
60 TO 1 80
61 OU 1 3
62 ST 1
63 TO 1 75
64 OU 1 1
65 ST 1
66 TO 1 70
67 VE 1
68 RS
69 GT 100
70 CA 135 73
71 RS
72 GT 100
73 WT
74 GT 66
75 CA 135 78
76 RS
77 GT 100
78 WT
79 GT 63
80 CA 135 83
81 RS
82 GT 100
83 WT
84 GT 60
85 CA 135 88
86 RS
87 GT 100
88 WT
89 GT 57
90 CA 135 93
91 RS
92 GT 100
93 WT
94 GT 54
95 CA 135 98
96 RS
97 GT 100
98 WT
99 GT 52
100 GT 100
101 CA 116 115
102 OU 3 25
103 IN 3 39 111
104 OU 2 22
105 IN 2 33 107
106 RN
107 CA 135 109
108 RN
109 WT
110 GT 105
111 CA 135 113
112 RN
113 WT
114 GT 103
115 RC
116 ST 2
117 OU 3 24
118 ST 1
119 CA 126 122
120 CT 1
121 RN
122 CA 135 124
123 RN
124 WT
125 GT 119
126 IN 1 0 128
127 RN
128 IN 2 0 130
129 RN
TTCN compiler example

```
130 IN 3 0 132
131 RN
132 TO 1 134
133 RN
134 RC
135 IN 1 0 138
136 VE 0
137 RN
138 IN 2 0 141
139 VE 0
140 RN
141 IN 3 0 144
142 VE 0
143 RN
144 TO 2 147
145 VE 0
146 RN
147 RC
```
Observer code example

C.1 Input definition file parser.def

```
$BEGIN(QUALIFIERS)
:En1 = '1' AND :En2 = '1'
none
$END(QUALIFIERS)

$BEGIN(CHARDEFS)
a = 100xxOxx
b = 110xx0xx
c = 0x101100
d = 0x101100
e = 0x1101xx
f = 0x0xx0xx
$END(CHARDEFS)

$BEGIN(EVENTDEFS)
1 = f*abcd
2 = f*ab*c
3 = f*cd
$END(EVENTDEFS)
```

C.2 Mask file mask.dat

```
:En1 = '1' AND :En2 = '1'
none
97 10000000 00011011
98 11000000 00011011
99 00101100 01000000
100 00101100 01000000
101 00110100 01000011
102 00000000 01011011
```

C.3 Parser code file parser.dat

```
0 VT
1 MA 10 97
2 MA 14 99
3 MA 5 102
4 GT 33
5 WT
6 MA 10 97
7 MA 14 99
8 MA 5 102
9 GT 33
10 WT
11 MA 17 98
12 MA 21 99
13 GT 33
```
Observer code example

14 WT
15 MA 23 100
16 GT 33
17 WT
18 MA 25 98
19 MA 29 99
20 GT 33
21 OU 2
22 GT 0
23 OU 3
24 GT 0
25 WT
26 MA 25 98
27 MA 21 99
28 GT 33
29 OU 2
30 GT 0
31 OU 1
32 GT 0
33 OU 1000
34 GT 34
Appendix D

Comments regarding the EFSM model

D.1 Packet structure

A packet is a triple (start_delimiter, data, end_delimiter) where start_delimiter is either DATA_START or DATA_PREFIX, end_delimiter is either DATA_END or NONE (i.e., it is physically absent), and data is a sequence of symbols DATA_ZERO and DATA_ONE (representing bits). Normally the 'data' field of a packet is non-empty; however, in exceptional cases (see the comments on the states L5 and L7) a packet with an empty 'data' field is used.

The most complicated packet structure involves the 'concatenated response' (cf. Figure 3-14 on Page 30). It deals with the possibility that the transaction layer wants to transmit a response packet immediately following (concatenated to) the acknowledgement of the last packet received. The transaction layer indicates its willingness to do so by a parameter with the value HOLD; in case it does not want to transmit a concatenated response this value is RELEASE. The case of RELEASE has two subcases: one in which the response packet is received on time (indicated by a ?LK_DATA.req) and another in which a time-out occurs (indicated by TIMER EXPIRES).

D.2 Interface description

D.2.1 From transaction layer to link layer

?LK_DATA.req(pars)

The transaction layer requests the link layer to transmit one asynchronous packet on the bus. The packet to be transmitted must be constructed from the 'pars' parameters. This service shall be confirmed with a !LK_DATA.conf().

?LK_DATA.resp(ack_code, bus_occ_ctl)

The transaction layer uses this service to respond to a received packet; i.e., can instruct the link layer to transmit an acknowledge (construct from the ack_code) and seize (depends on the 'bus_occ_ctl' parameter) the bus as soon as possible for the concatenated transmission of a "response packet".

bus_occ_ctl: {HOLD, RELEASE}

D.2.2 From link layer to transaction layer

!LK_DATA.conf(request_status, ack_code)

The link layer uses this service to confirm transmission of a packet. The 'ack_code' has only meaning in case a valid acknowledgement of a non-broadcast packet was received. request_status:

{ACK RECEIVED, ACK MISSING, BROADCAST SENT}
Comments regarding the EFSM model

\texttt{ILK\_DATA\_ind(pars, packet\_status)}

The link layer uses this service to indicate the transaction layer that a packet has been received and hands over packet data and packet parameters with it. The transaction layer shall respond to such an indication with a \texttt{ILK\_DATA\_resp(p packet\_status: \{GOOD, BROADCAST, DATA\_CRC\_ERROR, FORMAT\_ERROR\)}}

D.2.3 From link layer to physical layer

\texttt{PH\_ARB\_req(arbitration\_class)}

Used by the link layer to gain access to the bus. The physical layer shall confirm on completion of arbitration. Arbitration with class "IMMEDIATE" always results in a \texttt{PH\_ARB\_conf(WON)}.

\texttt{arbitration\_class: \{IMMEDIATE, FAIR\}}

\texttt{PH\_DATA\_req(p)}

Abstract event that maps onto the consecutive transmission of multiple "clock" induced symbols (page 52 of the standard) representing a complete packet; i.e., transmission of a DATA\_PREFIX delimiter, a sequence of DATA\_ZERO or DATA\_ONE symbols and a DATA\_END delimiter.

D.2.4 From physical layer link to layer

\texttt{PH\_ARB\_conf(status)}

The physical layer uses this service to report, to the link layer, the result of a \texttt{PH\_ARB\_req()} request. If the returned status is 'LOST' the link layer can try again later, if the returned status is 'WON' the link layer has gained access to the bus.

\texttt{status: \{WON, LOST\}}

\texttt{PH\_DATA\_ind(p)}

Abstract event that maps onto the reception of multiple data symbols representing a complete packet; i.e., reception of a DATA\_START delimiter, a sequence of DATA\_ZERO or DATA\_ONE symbols and a DATA\_END delimiter.

D.2.5 From node controller to link layer

\texttt{LK\_CONTROL\_req(command)}

The node controller uses this service to direct the link layer into a specific mode. The specification of link layer parameters through this service has been left unspecified.

\texttt{command: \{INITIALIZE, RESET\}}

D.2.6 From link layer to node controller

\texttt{LK\_CONTROL\_conf()}

Used by the link layer to confirm execution of a \texttt{LK\_CONTROL\_req()}. No parameters are communicated.

\texttt{ILK\_EVENT\_ind(link\_event)}

The link layer uses this service to indicate to the node controller error events diagnosed by the link layer. When a received packet \(p\) contains an error, indicated by \texttt{ERROR(p)}(p), a link event, with parameter \texttt{ERROR\_TYPE(p)}(p), is communicated to the node controller.

\texttt{link\_event: \{HEADER\_CRC\_ERROR\_DETECTED, UNKNOWN\_TRANSACTION\_CODE\_DETECTED, BUS\_OCCUPANCY\_VIOLATION\_DETECTED\}}
D.3 Functions definitions

ACK_CODE(p):
Return the acknowledge code (4 bits) extracted from the given packet p.

ASSEMBLE_ACK_1(ack_code):
Calculate the 4-bit parity for the given ack_code and construct the 8-bit acknowledge 'data' field. Return an acknowledge packet starting with DATA_PREFIX, 8-bits acknowledge data and a DATA_END end symbol. This acknowledge packet format is used in case the link layer responds that no response data will be concatenated to this acknowledge (bus_occCtl=RELEASE).

ASSEMBLE_ACK_2(ack_code):
Calculate the 4-bit parity for the given ack_code and construct the 8-bit acknowledge 'data' field. Return an acknowledge packet starting with DATA_PREFIX, 8-bits acknowledge data, but no end symbol (NONE). This acknowledge packet format is used in case the link layer responds that response data will be concatenated to this acknowledge (bus_occCtl=HOLD). The link layer allows the transaction layer a DATA_PREFIX_TIME for the delivery of this response data. On time-out an empty packet will be transmitted. Otherwise, the delivered response data will be concatenated.

ASSEMBLE_ASYNC(pars, node_id):
Return an asynchronous packet based upon pars and node_id. An asynchronous packet always starts with a DATA_PREFIX delimiter and ends with a DATA_END delimiter. Construct the asynchronous packet 'data' field from the given parameter structure pars and node_id. First assemble the consecutive header quadlets and calculate the header_CRC for these quadlets. If the packet carries data, assemble the data quadlets. Use padding if necessary and calculate the data_CRC. Join the header, header_CRC, data and data_CRC.

B'CAST(p):
Check whether packet p is a broadcast packet by extracting its destination_id. FALSE: when p is not a broadcast packet (DEST_ID(p) ≠ $FFFF), TRUE: in case of a broadcast packet (DEST_ID(p) = $FFFF).

DEST_ID(p):
Extract the first 16 bits of the 'data' field of packet p.

DISASSEMBLE(p):
Strip the 'data' field of packet p of its CRC sum(s) and return the extracted parameter list. The parameter list will be used to indicate the reception of a packet to the link layer.

ERROR(p):
Check whether packet p contains an error that has to be reported. Such an invalid asynchronous packet requires an error message to be communicated to the node controller. FALSE: the packet has no errors that require the return of an error event, TRUE: a !LK_EVENT.ind(ERROR_TYPE(p)) must be returned.

ERROR_TYPE(p):
An error type is an element of the set {HEADER_CRC_ERROR_DETECTED, UNKNOWN_TRANSACTION_CODE_DETECTED, BUS_OCCUPANCY_VIOLATION_DETECTED}. If the length of packet p exceeds the MAX_BUS_OCCUPANCY bound, return BUS_OCCUPANCY_VIOLATION_DETECTED. In case the transaction code field of p is of the reserved type (in this model we abstract from isochronous packets), return UNKNOWN_TRANSACTION_CODE_DETECTED. If a packet type could be established, conduct a CRC check on the header, and in case it fails return HEADER_CRC_ERROR_DETECTED. Multiple error conditions could occur simultaneously; a precedence relation is not yet specified.

STATUS(p):
Determine the status parameter of packet p for the !LK_DATA.ind() service primitive. The status parameter is an element of the set {GOOD, BROADCAST, DATA_CRC_ERROR, FORMAT_ERROR}.
Comments regarding the EFSM model

If the data_CRC checks out (if not, return DATA_CRC_ERROR) and the packet does not contain a non-zero reserved field or has another format error (if it does, return FORMAT_ERROR), the packet is considered to be free of errors. If the packet has no errors and is not a broadcast (DEST_ID(p) /= $FFFF), GOOD shall be returned. If the packet has no errors but is a broadcast, BROADCAST shall be returned.

VALID_ACK(p): Check whether packet p is a valid acknowledgement. A valid acknowledge packet shall start with a DATA_START, contain data and may end with DATA_END or NONE (no end delimiter). The length and parity constraints on the received 'data' field are checked to determine whether or not this is a valid 8-bit acknowledge.

VALID(p): Check whether packet p is valid. A valid packet always starts with a DATA_START delimiter, contains data and ends with a DATA_END delimiter. The length, header_CRC and transaction code field are checked to determine whether or not this is a valid asynchronous packet. The length must be a multiple of quadlets and is bounded by MAX_BUS_OCCUPANCY. FALSE: the packet fails one or more of the validity requirements, TRUE: the packet has been found to be valid.

D.4 Notes

- State L6 (see [IEE95; Figure 6-19, Page 174]) plays no distinctive role in our abstraction and has therefore been removed.
- The variables p and p' are of type packet.
- In state L5 the transition / !PH_DATA.req(DATA_PREFIX,-,-) back to L5 is continuously enabled. Putting this dummy type of packet (only consisting of a DATA_PREFIX) on the bus is used to hold the bus while awaiting a ?LK_DATA.resp from the transaction layer. The state-transition model used does not allow to express this timing constraint but the intention is that this transition is taken frequently enough to keep the bus busy. Furthermore, it should be noted that the transition starting with ?LK_DATA.resp, when it comes, has priority over the transition back to L5, and will be taken immediately.
- In state L7 the link layer is awaiting a ?LK_DATA.req from the transaction layer. Whenever this takes too long a time-out is generated, modeled by the condition [TIMER EXPIRES] (again, like in state L5, the state-transition model used only allows for such qualitative timing constraints). In this case the expected concatenated response packet is supposed to be empty and hence such an empty packet consisting of (DATA_PREFIX,-,DATA_END) is put on the bus (see [IEE95; Figure 3-14, Page 30]). This empty packet is also the reason for differentiating between ASSEMBLE_ACK_1 including a DATA_END symbol in the case bus_occ_ctl=RELEASE and ASSEMBLE_ACK_2 without a DATA_END symbol in the case bus_occ_ctl=HOLD when leaving state L5.
- The modeling of a lost arbitration is inadequate in the current diagram. It seems reasonable to assume that the link layer will try for rearbitration without needing another ?LK_DATA.req from the transaction layer. The informal specification is unclear about the way such a retry is done. For instance, should this happen in a separate state, say L0*, in which only rearbitration is requested, or should other events such as a ?PH_DATA.ind still be responded to (in which case the pending ?LK_DATA.req should be "saved" somehow)?

D.5 Mapping onto a Conformance Kit EFSM

D.5.1 Mapping procedure

Three PCO's: - PHY
- TRANS
- CONTR
Comments regarding the EFSM model

Events:

?LK_DATA.req(pars) -> input LK_DATA_REQ_<bcast> at TRANS,
where: pars="non-broadcast packet" -> bcast=0
pars="broadcast packet" -> bcast=1

?LK_DATA.resp(ack_code, bus_occ_ctl) -> input LK_DATA_RESP_<boc> at TRANS,
where: buc_occ_ctl=RELEASE -> boc=0
buc_occ_ctl=HOLD -> boc=1

?PHY_ARB.conf(status) -> input PH_ARB_CONF_<sts> at PHY,
where: status=LOST -> sts=0
status=NON -> sts=1

?PHY_DATA.ind(p) -> input PHY_DATA_IND_<val><error><bcast><addrs><stat> at PHY,
where: VALID(p)=FALSE -> val=0
VALID(p)=TRUE -> val=1
ERROR(p)=TRUE and ERROR_TYPE(p)=HEADER_CRC_ERROR_DETECTED -> error=00
ERROR(p)=TRUE and ERROR_TYPE(p)=UNKNOWN_TRANSACTION_CODE_DETECTED -> error=01
ERROR(p)=TRUE and ERROR_TYPE(p)=BUS_OCCUPANCY_VIOLATION_DETECTED -> error=10
ERROR(p)=TRUE and ERROR_TYPE(p)=UNKOWN_TRANSACTION_CODE_DETECTED -> error=11
B'CAST(p)=FALSE -> bcast=0
B'CAST(p)=TRUE -> bcast=1
DEST_ID(p)!=node_ID -> addrs=0
DEST_ID(p)=node_ID -> addrs=1
STATUS(p)=GOOD -> stat=00
STATUS(p)=BROADCAST -> stat=01
STATUS(p)=DATA_CRC_ERROR -> stat=10
STATUS(p)=FORMAT_ERROR -> stat=11
-> input PHY_DATA_IND_SUBGAP at PHY,
-> input PHY_DATA_IND_VAL_ACK at PHY,
-> input PHY_DATA_IND_INVAL_ACK at PHY.

?LK_CONTROL.req(command) -> input LK_CONTROL_REQ_<cmd> at CONTR,
where: command=INITIALIZE -> cmd=0
command=RESET -> cmd=1

!LK_DATA.conf(request_status, ack_code) -> output LK_DATA_CONF_<rqs> at TRANS,
where: request_status=ACK_RECEIVED -> rqs=00
request_status=ACK_MISSING -> rqs=01
request_status=BROADCAST_SENT -> rqs=10

!LK_DATA.ind(pars, packet_status) -> output LK_DATA_CONF_<psts> at TRANS,
where: packet_status=GOOD -> psts=00
packet_status=DATA_CRC_ERROR -> psts=01
packet_status=FORMAT_ERROR -> psts=10

!PH_ARB.req(arbitration_class) -> output PH_ARB_REQ_<arc> at PHY,
where: arbitration_class=FAIR -> arc=0
arbitration_class=IMMEDIATE -> arc=1

!PH_DATA.req() -> output PH DATA_REQ_<ack><enddl><data> at PHY,
where: p="asynchronous packet" -> ack=0
p="acknowledge" -> ack=1
p="NONE end delimiter" -> enddl=0
p="DATA END end delimiter" -> enddl=1
p="does not contain data" -> data=0
p="contains data" -> data=1
All packets start (implicitly) with a DATA_PREFIX start delimiter.

!LK_CONTROL.conf() -> output LK_CONTROL_CONF at CONTR.

!LK_EVENT.ind(link_event) -> output LK EVENT_IND_<evnt> at CONTR,
where: link_event=HEADER_CRC_ERROR_DETECTED -> evnt=00
link_event=UNKNOWN_TRANSACTION_CODE_DETECTED -> evnt=01
link_event=BUS_OCCUPANCY_VIOLATION_DETECTED -> evnt=10

D.5.2 Mapped EFSM

-- Specification (EFSM) file created with DEFACTO KIT release 2
--
-- Author: sies@ist1 (Olaf Sies)
Comments regarding the EFSM model

VARIABLES
  stat: 0..3; -- stores received packet class
  bcast: BOOL; -- broadcast packet flag

START L14: (stat=0; bcast=False);

CODES complete;

GATES TRANS, PHY, CONTR;

INPUTS spontaneous, -- untriggered transition (model fix)
  timer; -- timeout in state L7 (start upon L5->L7)
  LK_DATA_REQ_0, -- request to send non-broadcast
  LK_DATA_RESP_0, -- RELEASE
  LK_DATA_RESP_1 @TRANS; -- HOLD
  PH_ARB_CONF_0, -- LOST
  PH_ARB_CONF_1, -- WON
  PH_DATA_IND_SUBGAP, -- subaction gap detected on the bus
  PH_DATA_IND_VAL_ACK, -- valid acknowledge received
  PH_DATA_IND_INV_ACK, -- invalid acknowledge received
  PH_DATA_IND_000xxxx, -- [not VALID(p) and not ERROR(p)]
  PH_DATA_IND_001xxxx, -- [not VALID(p) and ERROR(p)]
  PH_DATA_IND_010xxxx,
  PH_DATA_IND_011xxxx,
  PH_DATA_IND_10000xx,
  PH_DATA_IND_1001001,
  PH_DATA_IND_1001010,
  PH_DATA_IND_1001011, -- [VALID(p) and not B'CAST(p) and
  PH_DATA_IND_1000100; -- not node_ID=DEST_ID(p)]
  node_ID=DEST_ID(p)]
  PH_DATA_IND_1000110,
  PH_DATA_IND_1000111 @PHY;
  LK_CONTROL_REQ_0, -- INITIALIZE
  LK_CONTROL_REQ_1 @CONTR; -- RESET

OUTPUTS LK_DATA_CONF_00, -- ACK_RECEIVED
  LK_DATA_CONF_01, -- ACK_MISSING
  LK_DATA_CONF_10, -- BROADCAST_SENT
  LK_DATA_IND_00, -- GOOD
  LK_DATA_IND_01, -- BROADCAST
  LK_DATA_IND_10, -- DATA_CRC_ERROR
  LK_DATA_IND_11 @TRANS; -- FORMAT_ERROR
  PH_ARB_REQ_0, -- FAIR
  PH_ARB_REQ_1, -- IMMEDIATE
  PH_DATA_REQ_00, -- "open empty dummy"
  PH_DATA_REQ_01, -- "closed empty dummy"
  PH_DATA_REQ_011, -- "asynchronous packet"
  PH_DATA_REQ_10x, -- "open acknowledge"
  PH_DATA_REQ_11x @PHY; -- "closed acknowledge"
  LK_CONTROL_CONF;
  LK_EVENT_IND_01, -- HEADER_CRC_ERROR_DETECT
  LK_EVENT_IND_10, -- UNKNOWN_TRANSACTION_CODE_DETECT
  LK_EVENT_IND_11 @CONTR; -- BUS_OCCUPANCY_VIOLATION_DETECT

TRANSITIONS

STATE L0:
- PH_DATA_IND_000xxxx --
- PH_DATA_IND_001xxxx LK_EVENT_IND_01 --
- PH_DATA_IND_010xxxx LK_EVENT_IND_10 --
- PH_DATA_IND_011xxxx LK_EVENT_IND_11 --
- PH_DATA_IND_10000xx --
- PH_DATA_IND_1001001 LK_DATA_IND_01 --
- PH_DATA_IND_1001010 LK_DATA_IND_10 --
- PH_DATA_IND_1001011 LK_DATA_IND_11 --
- PH_DATA_IND_1000100 PH_ARB_REQ_0 (stat=0) L4;
- PH_DATA_IND_1000110 PH_ARB_REQ_1 (stat=2) L4;
- PH_DATA_IND_1001111 PH_ARB_REQ_1 (stat=3) L4;
Comments regarding the EFSM model 129

D.5.3 Restriction file

-- Restrictions and patterns file created with DEFACTO KIT release 2
--
-- Author: sies@ist1 (Olaf Sies)
-- Created: Thu Nov 16 09:53:29 1995
-- Version: %1%
-- Last Rev: %G% on %U%
-- SccsId: %/%I%

EXCLUDE complete IN MAIN PART; -- The transitions that are added to render the
EXCLUDE complete IN TRANSFER PART; -- nor are they required in transfer sequences.

-- The RNL "makesios" tool requires a complete specification to generate sios'es although
-- literature states that UIO's can be generated from incomplete specifications.