esa
estec

european space agency
european space research
and technology centre

WSM/SH/010
Issue 1
February 1996

VHDL Models for Board-level Simulation

Prepared by S. Habinc

Spacecraft Control and Data Systems Division (WS)
Keplerlaan 1 - Noordwijk - The Netherlands
Mail address: Postbus 299 - 2200 AG Noordwijk - The Netherlands
Tel: +31-71-565 4722 - Telex: 39098 - E-mail: sandi@ws.estec.esa.nl - Fax: +31-71-565 4295

Page intentionally left blank

Table of contents

Page intentionally left blank

1 INTRODUCTION

1.1 Purpose and scope

This document provides recommendations for development and usage of VHDL models intended for board-level simulation. This document is intended to be read together with the VHDL Modelling Guidelines, RD1. It could be used in ESA developments of models for board-level simulation and for simulation of board designs comprising such models.

The information herein is not to be considered as requirements, although sometimes expressed as such, but merely as useful hints and recommendations.

The purpose of these recommendations is to define modelling criteria that will produce models for board-level simulation that are highly accurate in both functionality and timing, and that will provide sufficient simulation performance to facilitate long simulation runs. The document also provides sufficient information to allow someone with little VHDL knowledge to perform a simulation of a board design using models for board-level simulation.

Parts of the document, such as the discussion on model verification, can also be used for ASIC developments. Requirements on models for board-level simulation and VHDL models used for synthesis are dissimilar and therefore are no synthesis aspects discussed in this document. The document does not address distribution of models for board-level simulation nor the protection of design information, issues which are discussed in “The Usage of VHDL in the European Space Agency”, RD6.

This document is not intended to be a guide to the VHDL language itself. On the contrary is the reader expected to have previous VHDL knowledge before developing a model intended for board-level simulation.

1.2 Document organisation

This document is divided in five major parts, the first one covering the definition and benefits of board-level simulation, followed by guidelines for developing models for board-level simulation. The third part covers the verification of these model, and is followed by a description on how to model and verify a board design. The final part specifies requirements on model documentation.

For each requirement or suggestion stated in the document there is normally an accompanying explanation and often a code example or a figure. All code examples in the document have been taken from a complete model for board-level simulation, although the VHDL code may sometimes have been reduced to highlight the essential parts. The code examples are therefore not always possible to be analysed by a VHDL simulator as presented. A complete model is not included in the document due to the prohibitive size of the source code, but is made available via *ftp* as described in section 1.3.

1.3 References

This document is available from ESA in the PostScript format via *ftp* at URL ftp://ftp.estec.esa.nl/pub/vhdl/doc/BoardLevel.ps.

Additional information on board-level simulation, including a complete example, is available at URL http://www.estec.esa.nl/wsmwww/vhdl/boardlevel.html.

The following documents are being referenced:

RD1 **VHDL Modelling Guidelines**,
P. Sinander, ESA ASIC/001, European Space Agency, The Netherlands, 1994,
URL ftp://ftp.estec.esa.nl/pub/vhdl/doc/ModelGuide.ps

RD2 **IEEE Standard VHDL Language Reference Manual**,
IEEE Std 1076-1993, IEEE, New York, USA, 1994

RD3 **IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164)**, IEEE Std 1164-1993, IEEE, New York, USA, 1993

RD4 **IEEE Standard VITAL ASIC Modelling Specification**, Version 3.0,
IEEE, New York, USA, URL http://vhdl.org/vi/vital

RD5 **Built-In Test for VLSI: Pseudorandom Techniques**,
P. H. Bardell et al., John Wiley & Sons, New York, USA, 1987

RD6 **The Usage of VHDL in the European Space Agency**,
P. Sinander, European Space Agency, The Netherlands, 1995,
URL ftp://ftp.estec.esa.nl/pub/vhdl/doc/UseOfVHDL.ps

RD7 **VHDL Coding Styles and Methodologies: an In-Depth Tutorial**,
B. Cohen, Kluwer Academic Publishers, USA, 1995

1.4 Conventions

A *component model* in this document is a gate level netlist or a synthesisable Register Transfer level, RTL, description, in VHDL or any other notation, representing the logic design from which the component has been manufactured and being suitable for simulation.

The types and subprograms declared in the packages *ESA.Simulation* defined in RD1, *Std.Standard* and *Std.TextIO* defined in RD2, *IEEE.Std_Logic_1164* defined in RD3, and *IEEE.Vital_Timing* and *IEEE.Vital_Primitives* defined in RD4, will be addressed in this document without necessarily further stating their origin or in which package they belong.

2 BOARD-LEVEL SIMULATION

This section defines board-level simulation and describes its benefits and limitations. By knowing the purpose and characteristics of models intended for board-level simulation it is more likely that such a simulation will be successful. Knowing what can and cannot be achieved using board-level simulation can reduce unprofitable efforts in advance.

2.1 Definition of board-level simulation

Board-level simulation can be defined as simulating the functionality of one or several printed circuit boards built with standard components, possibly incorporating Application Specific Integrated Circuits, ASIC, and Application Specific Standard Products, ASSP. Board-level simulation is also known under the names rapid or virtual prototyping and sometimes system simulation. The purpose of board-level simulation is to verify the behaviour of the board design, e.g. that the components operate correctly in the selected operating modes.

When board designs contain processors it is also possible to perform verification of the hardware-software interaction, such as verifying that ASIC registers can be programmed and software drivers work properly etc. In addition, the performance of the processor board could be evaluated. Board-level simulation will also give some information about timing correctness, though it can probably not replace worst-case timing analysis.

Board-level simulation does not comprise verification of individual ASICs during their development. It does not comprise system performance simulation including aspects such as throughput, latency, buffer allocation and utilisation, where neither accurate data nor clock behaviour is considered being essential.

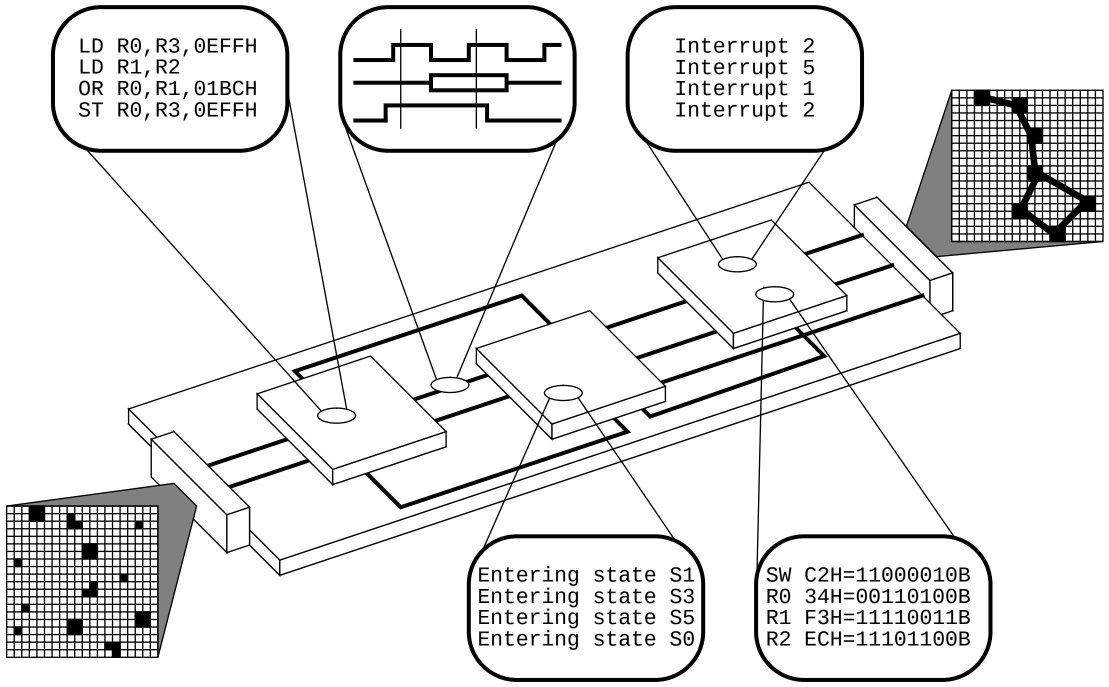
```
LD R0,R3,0EFFH
LD R1,R2
OR R0,R1,01BCH
ST R0,R3,0EFFH
```
```
Interrupt 2
Interrupt 5
Interrupt 1
Interrupt 2
```
```
Entering state S1
Entering state S3
Entering state S5
Entering state S0
```
```
SW C2H=11000010B
R0 34H=00110100B
R1 F3H=11110011B
R2 ECH=11101100B
```

Figure 1: *The designers view of a board design when using board-level simulation.*

2.2 Benefits of board-level simulation

The feasibility and benefits of board-level simulation have already been proven in several projects. It supports a top-down methodology, allowing simulation of boards not fully implemented, enabling the designer to work with incomplete specifications of the own system or component and facilitates early verification of the design requirements. It also allows the designer to explore different design solutions and to prototype manageable parts of larger systems.

Product specifications can be verified before any manufacturing or breadboarding is started. This is useful when defining a system or component in a proposal or to ensure that when a breadboard is built it does not contain functional errors.

When designing an ASIC, its operation in a board design can be verified before manufacture. If models intended for board-level simulation are provided before the first silicon, significant savings in schedule can be obtained.

Integration can be performed earlier and the first design and verification loop can be done without any hardware manufacturing. The manufacturing can be postponed until all specifications have settled and all interfaces have been verified. It permits full laboratory integration without any available hardware, allowing the first integration to be done earlier in the design. The designer can deliver a board design comprising models for board-level simulation to the user for early system verification.

Special-built equipment for check-out and unit or system testing, which is nowadays as complex as the actual design, can be modelled as well. Allowing the prototype and the test equipment to be simulated together before any of them are built, which can reduce interfacing problems, or even reduce the need for the test equipment being built.

Models for board-level simulation allow the test engineer to simulate situations that are difficult to capture in real hardware due to timing synchronisation etc., resulting in a more thorough verification of the board design. Board-level simulation provides the designer with unlimited probing and acquisition points, not always possible to realise for the hardware.

Models for board-level simulation provide limited simulation support during parallel development of hardware and software, since this type of simulations usually take long time to perform, but delivers high functional and timing accuracy. However, it has been shown that by carefully selecting which software parts to simulate the time spent simulating can be reduced to manageable lengths. It is perhaps not always feasible to boot a complete operating system and launch applications, but all the firmware and hardware drivers could be verified using board-level simulation.

Board-level simulation enables hardware and software designers to work together in an early stage and to solve interfacing problems before committing the hardware for manufacturing. This area of simulation will become more interesting with the continuously increasing speed of simulators.

Board-level simulation should be carefully planned. Time spent modelling and simulating has to be weighted against what can be gained or lost compared to the replaced or reduced non-simulation activities. Efficient use of board-level simulation can lead to the reduction of other design activities.

It is important to establish by whom the board design model development and simulation should be performed while planning the activity to prevent unnecessary educational costs induced by assigning engineers to the task with no VHDL experience, even though only little experience is actually needed.

2.3 Board-level simulation using VHDL

A major issue for board-level simulation is the availability of simulation models of the components used in the board design. Despite commercial models being available for many standard components, increasingly often ASSPs, ASICs and other unusual components are used. Hardware modellers can solve this problem, though they are expensive, complicated to use and have limitations in the number of components that can be used simultaneously.

Using VHDL models is therefore an interesting alternative when no other models are available, which is the typical case for almost all components used on board spacecraft. By using VHDL the effort to support several platforms and simulators is greatly reduced, since VHDL models require no or only minor modifications for each new simulator.

It has been demonstrated that VHDL models of components can be integrated together to design and debug embedded systems in their entirety using hardware-less design methodology. Due to VHDL simulator performance reasons board-level simulation is normally limited to the digital domain.

Using VHDL for board-level simulation enables the user to also perform true mixed-level simulation, since detailed models are mostly written in VHDL and the number of ASIC libraries written in VHDL is rapidly increasing. Still, all different models have to follow some guidelines to ensure interoperability. This document and RD1 form such guidelines. The board design simulation can also contain non-VHDL representations such as netlists or schematics, being useful when verifying a board design containing an ASIC for which no VHDL library exists.

It is important to establish by whom the simulation models should be provided, which can become a critical issue if there is no model for board-level simulation available. A survey of existing models has therefore to be performed well before the simulation begins, allowing for the development of missing models.

3 MODELS FOR BOARD-LEVEL SIMULATION

A model for board-level simulation is characterised by its accurate modelling of the component behaviour, simulation performance, and ease of use for board designers. All such models delivered to ESA should be developed in accordance with RD1, and their implementation could benefit from following the suggestions made in this document.

The behaviour of the model as seen from the outside should be the same as for the modelled component and should include the full functionality, though specific test modes only used for manufacturing test need not be implemented. The interface signals of the model should have the exact waveform behaviour as observed for the component. Since the internal structure and state of the model do not need to reflect the modelled component, internal signals should not be used during the analysis of acquired simulation results since they could provide information not being fully correct.

A model for board-level simulation should be verified against a *component model* when possible, which could be in VHDL or any other representation suitable for simulation. The purpose of the verification should be to ensure the correctness of the model w.r.t. the component behaviour. When no other representation of the component is available for simulation, the model verification should be based on the information found in a Data Sheet or similar. Each model intended for board-level simulation should be provided with a test bench verifying its behaviour, which is described further in section 4.

Bus functional models, sometimes called bus interface models, are considered being reduced models for board-level simulation, modelling only the timing and behaviour of the interfaces. The timing and format of output drivers for data/control/addresses etc. are modelled as accurately as possible, while the internal functionality of the component is not necessarily modelled at all. Using bus functional models does not provide the full potential of board-level simulation since they simulate only a portion of the component. Nevertheless, the development of such models should follow the suggestions made herein since they should be possible to use together with models for board-level simulation.

The simulation performance should be assessed when transforming a model written on the Register Transfer level, RTL, to a model intended for board-level simulation. The assessment can be supported by comparing the source code with the requirements in this document. RTL models do normally not have sufficient simulation performance and will need to be modified since they are normally written for synthesis which imposes conflicting requirements on the VHDL code w.r.t. models for board-level simulation. The main development effort would then be to optimise the code accordingly, and to implement the model interfaces and develop the verification test bench.

Experience has shown that a proper review of the requirements in RD1 should be performed before a development begins and each model should be reviewed thoroughly before being released.

The source code header of the model entity should contain all information necessary for the user to simulate the model in a board design, and is also allowing distribution of analysed models containing no source code. A User’s Manual should be delivered with every model intended for board-level simulation, as specified in section 6.1.

3.1 Hierarchy

Hierarchy for models is introduced to obtain good source code readability and to separate different modelling aspects. The outlined hierarchy scheme below is based on two of these aspects, namely timing and functionality. Since these usually stem from two different lines of documentation and representation, the Data Sheet and the *component model*, the model intended for board-level simulation should be partitioned taking this into account. A partitioning also enables separate verification of the two domains of the model.

The model should be divided in two hierarchical levels; the top-level architecture and its functional core, to clearly separate the timing and checking for unknown input values from the functionality aspect of the model, as shown in figure 2. It is recommended that there are no other than these two hierarchical levels in the model, since multiple levels could reduce the code readability if not carefully used. The top-level architecture should be independent of the functional core where possible to reduce the need for changing it if only the functionality needs to be modified.

The two-level hierarchy could be flattened for improving simulation performance by reducing the number of signals interconnecting the hierarchy, although it is not recommended. This approach should only be used in extreme cases or for small models, and is shown in figure 3.

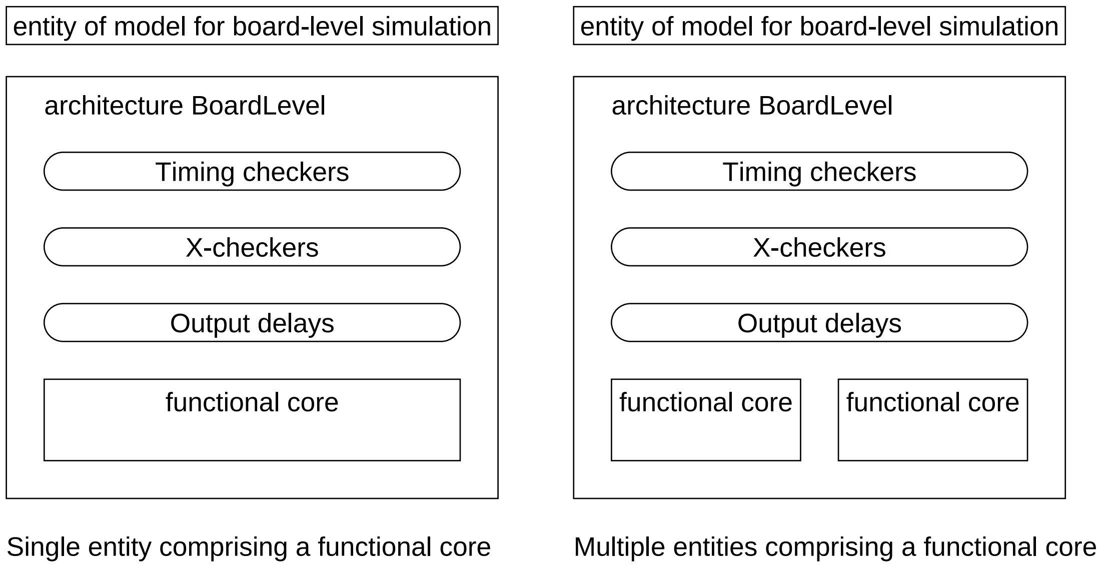
entity of model for board-level simulation
architecture BoardLevel
Timing checkers
X-checkers
Output delays
functional core
Single entity comprising a functional core
entity of model for board-level simulation
architecture BoardLevel
Timing checkers
X-checkers
Output delays
functional core
functional core
Multiple entities comprising a functional core

Figure 2: *Preferred two-level hierarchies (squares with rounded corners are processes or concurrent procedures, regular squares are subcomponents).*

The functionality of the component should be modelled in the functional core, excluding any timing aspects and without internal delays. A functional core could comprise more than one entity for larger designs, each functional block would then be a component instantiated in the top-level architecture. There should not be more components for the functional core than there are blocks in the architectural block diagram. In the functional core comprises many modules, which could be the case when a model is based on an RTL model, an additional hierarchy level could be considered, as shown in figure 3. These modelling aspects will be further referenced as *functional modelling*.

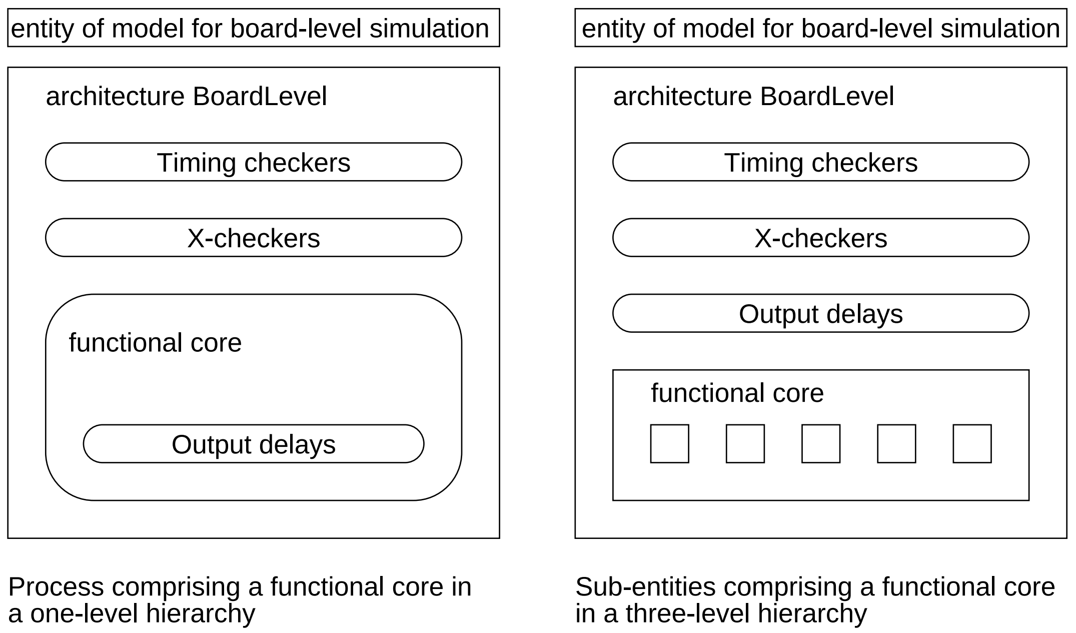
entity of model for board-level simulation
architecture BoardLevel
Timing checkers
X-checkers
functional core
Output delays
Process comprising a functional core in a one-level hierarchy
entity of model for board-level simulation
architecture BoardLevel
Timing checkers
X-checkers
Output delays
functional core
Sub-entities comprising a functional core in a three-level hierarchy

Figure 3: *Optional one- and three-level hierarchies.*

The external timing of the model should be contained in the top-level architecture, including setup and hold time checking, clock-to-output and propagation delay scheduling. It could be modelled in the functional core when simulation performance is critical and when source code readability is not reduced. Management of unknown input values can be divided between the two hierarchical levels as described section 3.3.2. These modelling aspects will be further referenced as *interface modelling*.

When a model has more than one hierarchical level the subcomponents should be explicitly bound using a configuration declaration, never relying on default binding. The generics of the subcomponents should be associated to the corresponding generics of the preceding entity as shown for *InstancePath* in example 1. Each component declaration should have the same name, generic and port declarations as the corresponding entity. Configuration specifications in the architecture should be avoided, permitting the usage of the more flexible configuration declarations outside the model.

```
library BitMod_Lib;

configuration BitMod_Configuration of BitMod is
   for BoardLevel
      for FunctionalCore: BitMod_Core
         use entity BitMod_Lib.BitMod_Core(Behavioural)
            generic map(InstancePath => InstancePath);
      end for;
   end for;
end BitMod_Configuration;
```

Example 1: *Configuration declaration for a model for board-level simulation.*

3.2 Functional modelling

Functional modelling comprises in this context the part of the model representing the logical functions of the modelled component. The following sections will describe how to develop a model with high functional accuracy and good simulation performance, being two important characteristics of models for board-level simulation.

The functionality of the component should be contained in the functional core of the model for board-level simulation. It should be independent of the top-level architecture, although some functions normally implemented with tristate buffers in a component could be modelled outside the functional core, as shown in figure 4. The functional core should be modelled with zero delay on outputs and without internal signal delays if possible.

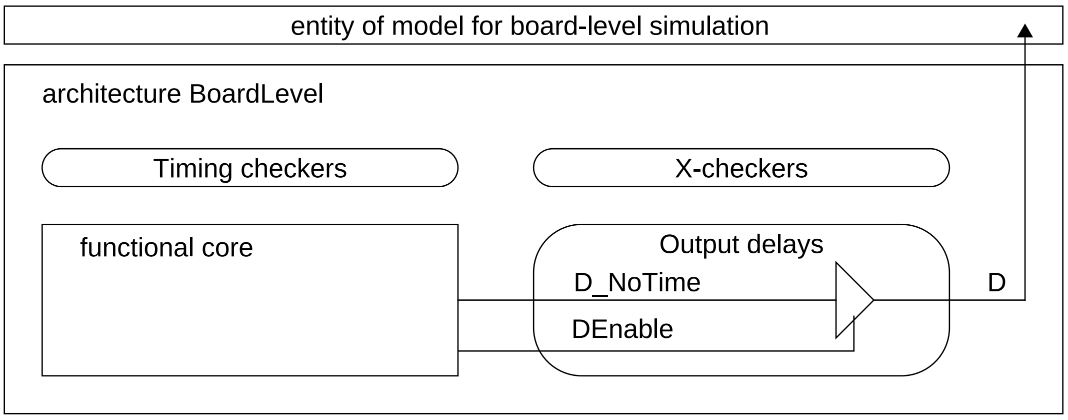

Figure 4: *Implementation of tristate buffer in the top-level architecture.*

3.2.1 Modelling for functional accuracy

Models for board-level simulation have to reflect the functional behaviour of components accurately enough to allow board designs to be verified for functionality and timing. Simulating boards using models with high functional accuracy will reduce the number of errors found on the manufactured board. Errors not found in the simulation, located in the models or in the board design itself, will eventually be discovered in the real hardware.

There are two major approaches to modelling for board-level simulation; independently develop the model from a Functional Specification or Data Sheet, or enhance the RTL model. The first approach is necessary when no RTL model is available to the developers. It can also be the case when the model for board-level simulation is developed in parallel with the component. The component development could then benefit from the independent interpretation of the specification. The two models should be compared to each other, first visually and later automatically when both mature. In the second approach, when a RTL model is to be revised to fulfil the requirements posed on models for board-level simulation, the protection of the design should be addressed since the resulting model could possibly be synthesised. Many of the suggestions in section 3.2.2 describing how to model for simulation performance will often reduce the probability that a component could be reverse engineered.

Care should be taken when a model is developed using only a Data Sheet as input, since the component is not always described in a Data Sheet as actually being implemented. The information in the Data Sheet could have been simplified, e.g. the description of an interface protocol may be more constrained than the actual design requirements. The source describing the functionality from which the modelling is performed should be identified. Any unresolved issues should be submitted to the foundry or company supporting the component and be documented.

It is not always obvious whether to model the behaviour that is described by the *component model* or the Data Sheet when there are inconsistencies between them. It may be that some functionality of the *component model* has been simplified or omitted in the Data Sheet, e.g. proprietary design features. In such a case it is recommended to model the full functionality and issue a warning when used, instead of excluding the function and consequently have an incorrect simulation.

The inclusion of unsupported or undocumented functionality of the component in the model for board-level simulation could simplify its comparison versus the *component model*, using the same set of stimuli. The model should therefore always reflect the component behaviour when there are inconsistencies or differences between the Data Sheet and the *component model*, otherwise the deviations will possibly turn up as failures when breadboarding.

Independently of how the model is developed, the full functionality should be modelled and verified versus the *component model* when available, as per section 4.

3.2.2 Modelling for simulation performance

The performance of present workstations and VHDL simulators provides a means for simulating board designs comprising several complex components such as microprocessors and ASICs. However, to be able to tap this simulation performance the simulation models have to be efficiently coded for simulation. An absolute requirement on simulation performance for models intended for board-level simulation cannot easily be defined, although unnecessarily slow or cumbersome implementations should be avoided.

The guidelines presented below are based on experiences with modelling and using models for board-level simulation. This is not an exhaustive list of issues to be addressed when a model is tuned for simulation performance. It should also be remembered that each suggestion might not be true for all situations and simulators. The best advice on simulation performance modelling is to use common sense in case of uncertainty. A good way to choose between two approaches is to simulate both and to select the one being most efficient. The stimuli used for such comparative simulations should be based on possible and probable input to the model. Different simulators have different performance characteristics, for obtaining the complete picture simulations should be performed on all foreseen simulators to be used.

Many rules and techniques that apply to writing optimised software, such as loop unrolling, code in-lining etc., also apply to models with good simulation performance since VHDL has many characteristics of a programming language. Some VHDL simulators have less built-in optimisation capabilities than state of the art optimising compilers for software, it is therefore often beneficial to manually perform optimisation at the source code level.

Standard packages, such as the *IEEE.Std_Logic_1164* and *IEEE.Vital_Timing IEEE.Vital_Primitives*, are sometimes accelerated for simulation performance. But since this is not always the case, it could be necessary to assess whether to use other types or subprograms when simulation performance is an issue.

3.2.2.1 Processes

Each process invocation has a cost in terms of simulation performance and in principle the number of processes should therefore be kept small. Each concurrent assignment is treated as a process, and should be avoided where possible. Note that block and generate-statements can incur the same cost as processes.

Processes should use sensitivity lists that can be statically allocated and have therefore potentially better simulation performance than when using wait-statements that are allocated dynamically. Process invocation should be minimised, only essential signals should be included in the sensitivity list. Functions sensitive to the same signals should be grouped in the same process, reducing the number of processes to invoke for each signal event. Following the approach above, all functions related to the same clock should be grouped together. One process per clock region could be used when multiple clocks exist for the component. Functions related to different clock regions should be placed in different processes, not to invoke the process for each event on the irrelevant clocks. This approach has been found efficient when clocks have dissimilar switching frequencies.

It should be decided whether to use single or multiple processes and which signals to include in the sensitivity lists when modelling combinational and asynchronous logic, based on comparative simulations. Combinational logic only related to a single clock signal should be included in the process modelling that clock region.

Code blocks, such as checkers and autonomous functions that can be disabled by means of generics or mode pins, could benefit from being placed in separate processes using generate-statements to prevent them from executing in modes when not needed. It is not sufficient to place such functions in a process and protect them with a conditional statement, since the process will still be invoked each time there is an event on signals in the sensitivity list. The generate-statement around the process will exclude the process from the simulation when disabled, eliminating all invocation costs when not used.

The outline of the functional core architecture shown in example 8 represents such structuring. The architecture in the example is divided in two processes, representing the synchronous and the asynchronous regions of the component. The sensitivity lists of the processes have been kept short to avoid unnecessary invocations.

The synchronous region in the example is clocked by the *Clk* input and is reset by the *Reset_N* input. The *ClkRegion* process implementing this region is made sensitive only to these two signals. The first part of the process handles the asynchronous reset of the region. The succeeding two parts model the functionality related to the rising and falling *Clk* edges. An edge on the *Clk* input is detected using the functions *Rising_Edge* and *Falling_Edge*, which also handle unknown input values.

The last part of the *ClkRegion* process is only invoked when no reset has been issued and neither of the clock edges have been detected. It is used for detecting and reporting unknown values on the *Clk* input as described in section 3.3.2. This method for checking for unknown input values will only negligibly contribute to the performance penalty, compared to checking the clock input at each signal event which occurs frequently.

The process *AsynchronousRegion* implementing the asynchronous region is made sensitive only to those inputs directly affecting its behaviour. The inputs *CS_N* and *RW_N* control the asynchronous write accesses to internal registers and are therefore included in the sensitivity list. The data and address buses are latched on the rising *CS_N* edge as shown in example 3, and do not affect the process when changing values. They need therefore not be sensed by the process and are not included in the sensitivity list, allowing for better simulation performance than if they were included. This modelling is inexact since the accessing scheme has been somewhat simplified.

3.2.2.2 Signals

Variables should be used instead of signals wherever possible, since each signal requires one or several drivers, specific handling (event scheduling,) and memory storage, which takes more instructions to execute (and likely decreases the cache hit ratio). Signals should preferably be used only for communication between processes. VHDL ’93 shared variables could potentially be used instead of signals, but should be used with precaution since potentially introducing indeterministic behaviour in the simulation. Another reason for merging processes is that the number of signals used for the communication between them is consequently reduced.

Resolved types should be avoided internally in the model where possible, since the calculation of the resulting value will need to call a resolution function for each event on the driving signals. Using unresolved types instead could potentially increase the simulation performance. Resolved types should therefore only be used when the resolution function is needed. This does not apply to variables, since no resolution function is needed and there should be no difference in simulation performance.

It has however been observed that there is no significant difference between using *Std_ULogic* instead of *Std_Logic*, since the latter is accelerated in some simulators. Some simulators also ensure that the resolution function is not invoked for signals with only one driver. This can be seen as analogous to replacing such signals with their unresolved base type, e.g. *Std_Logic* signals with one driver becomes *Std_ULogic* signals. An additional benefit of using unresolved types is that unwanted short-circuit connection between signals is automatically detected at analysis time since a signal of an unresolved type can only have one driver.

When moving a concurrent signal assignment into a process, it should be ensured that it is not updated more often than it would had been as a concurrent assignment. Reassigning a signal its current value should be avoided, since each such assignment requires that a transaction is scheduled for that signal. One should also be careful not to recalculate a signal value expression too often when moving the signal assignment into a process, e.g. for each clock cycle instead of each time one of the relevant input signals changes. In such case the simulation performance will decrease due to the increase of unnecessary calculation, even if the same calculated value is not reassigned to the signal. Similarly, removing static signals that seldom change will not improve the simulation performance significantly or will even decrease it.

Signal generating attributes such as *'Stable* should be avoided since they result in the creation of implicit signals which have to be handled in the scheduler. Instead should the attribute *'Event* be used where possible.

3.2.2.3 Types

Numerical data types such as *Integer* normally result in better simulation performance than arrays such as *Std_Logic_Vector* and *Bit_Vector*, and could be used for extensive calculations directly using the arithmetics of the processor on the host machine. However, one should be careful when the bit field information is required in the simulation, e.g. during instruction decoding in microprocessor models, since retrieving such information from an *Integer* could potentially be more costly than using an array in the first place. A trade-off should be performed between the cost for: performing type conversions between *Std_Logic_Vector* and *Integer*, and subsequent calculations using *Integer*; or directly performing calculations on *Std_Logic_Vector.* Findings indicate that the time required for converting a *Std_Logic_Vector* signal to an *Integer* variable, adding two *Integer* variables, and to convert the result back to a *Std_Logic_Vector* variable is faster than to make an addition between two *Std_Logic_Vector* signals.

The computations on data and address in example 3 are made using the type *Integer* instead of *Std_Logic_Vector* (not shown). The conversion is made directly from *Std_Logic_Vector* to *Integer* with the custom made function *To_Integer*, which is only called when the value is needed. This function also checks for unknown values and issues assertion reports when detected, its declaration is shown in example 2.

```
---------------------------------------------------------------------
-- Converts unsigned Std_Logic_Vector to Integer, leftmost bit MSB
-- Error message for unknowns (U, X, W, Z, -) being converted to 0
---------------------------------------------------------------------
function To_Integer(
   constant Vector:      Std_Logic_Vector;
   constant VectorName:  String           := "";
   constant HeaderMsg:   String           := "To_Integer:";
   constant MsgSeverity: Severity_Level := Warning)
   return                Integer;
```

Example 2: *Declaration of a Std_Logic_Vector to Integer converter function taking into account unknown values on the input.*

Type conversions on input signals should be performed in the functional core where the actual value is needed and only when necessary, and is described further in section 3.3.2. To illustrate this, the conversion between *Std_Logic_Vector* and *Integer* and the checking for unknown values on the data and address buses in example 3 are only necessary when the values are latched. No type conversion is therefore necessary in the top-level architecture for the two buses. This scheme has better simulation performance than if each signal would be type converted each time there is an event. Unnecessary assertion reports are also avoided since checking for unknown values is only done when the value is used.

It has been seen that enumerated types have better simulation performance than array type, especially for coding of finite state machines using case-statements, where *Bit_Vectors* and *Std_Logic_Vectors* are slower than enumerated types.

```
--------------------------------------------------------------------
-- Implementation of all asynchronous functionality.
-- Latching of data to be written into the internal registers.
-- Generation of external data bus enable. Checks for unknown
-- values are done for the input signals.
-- The modelling is not fully correct w.r.t. a typical
-- RAM I/F, since some relaxations have been introduced.
--------------------------------------------------------------------
AsynchronousRegion: process(CS_N, RW_N, Reset_N)
begin
   if Reset_N = '0' then
      -- To_X01 on Reset_N is done in the top-level architecture
      -- Asynchronous reset of model
      DEnable <= False;
   elsif Rising_Edge(CS_N) then
      -- End of access
      if To_X01(RW_N, "RW_N", InstancePath, Error)='0' then
         -- Write access to internal registers
         -- X on CS_N is treated as no event (no access)
         -- X on RW_N is treated as 1 (no write access)
         -- X on A and D_In are treated as 0
         -- A and D_In are converted to Integer
         AWrite  <= To_Integer(A,    "A", InstancePath, Error);
         DWrite  <= To_Integer(D_In, "D", InstancePath, Error);
      end if;
      DEnable <= False;
   elsif Now /= 0 ns then
      -- Asynchronous behaviour
      -- Enabled for read cycles after Reset
      -- X on RW_N is treated as 0
      -- X on CS_N is treated as 1
      DEnable <= (To_X01(RW_N, "RW_N", InstancePath, Error)='1') and
                 (To_X01(CS_N, "CS_N", InstancePath, Error)='0') and
                 (Reset_N='1');
   end if;
end process AsynchronousRegion;
```

Example 3: *Implementation of asynchronous write access to internal registers.*

When modelling large memory elements the memory actually used by the simulator on the host machine should be taken into account, since the cache hit ratio will decrease with larger memory usage, and as a consequence the simulation performance will be decreased as well. Note that it is not the size of the memory being allocated by the simulator that is critical, but the size and the distribution of the memory which is being frequently accessed. Since the allocation of the modelled memory into the actual memory differs between simulators, operating systems and hardware, it is difficult determine what impact the method chosen will have on the simulation performance. But, as a general recommendation the memory usage should be minimised as much as possible. Also from a memory usage point of view signal declarations are more costly than variable declarations.

For example, an eight bit wide register would at least require 48 bits of memory if modelled as a *Std_Logic_Vector* since each bit would be represented as 4 bits in memory, covering all nine *Std_Logic* strengths. The same register contents could be represented as an *Integer* and would then require 4 bytes in most simulators, which is less than the *Std_Logic_Vector* representation, but which cannot represent all the nine *Std_Logic* strengths. The memory usage should be measured for the two approaches and be used, together with the requirements on the level of detail for the data representation, for deciding on how to model such registers.

3.2.2.4 Subprograms

Passing large data structures as parameters to subprograms decreases the simulation performance when the data structure size increases, which should be considered when deciding whether to represent data as *Std_Logic_Vector* or *Integer*. In addition, calls to subprograms declared in packages are difficult to optimised by the analyser since the package body can be reanalysed without necessitating that the code where the call is made from is reanalysed as well. It is therefore necessary to manually replace subprogram calls by in-line coding in the source code when optimising a model for simulation performance.

3.2.2.5 Expressions

Globally static constants, such as deferred constants, cannot be evaluated at analysis time by the analyser. A way to work around this is to declare a local constant that is computed once during the elaboration from the deferred constant or constants, e.g. "**constant** *LocalTpd*: *Time* := *GlobalTpd /2*;" where *GlobalTpd* is a deferred constant.

It is believed that some simulators do not optimise expressions for common terms and it is therefore necessary to manually make the optimisation in the code, as in example 4.

```
-- Original code:                -- Optimised code:
Result0 := A+B*C;                Temp0   := B*C;
Result1 := D-B*C;                Result0 := A+Temp0;
                                 Result1 := D-Temp0;
```

Example 4: *Common term in expressions being expressed as a temporary variable.*

On the other hand, unnecessary usage of temporary expressions could potentially reduce the simulation performance since each temporary variable assignment has a certain cost, as shown in example 5. Needless to say is that when performing calculations with temporary signals instead of variables, the penalty is worse.

```
-- Original code:               -- Optimised code:
Temp1   := A+B;                 Result2 := (A*B)/(C*D)*(E mod F);
Temp2   := C-D;
Temp3   := E mod F;
Result2 := Temp1/Temp2*Temp3;
```

Example 5: *Unnecessary temporary expressions merged into one.*

Since VHDL specifies short-circuit boolean evaluation, terms that would short-circuit an expression evaluation should be placed as early as possible in the expression. Short-circuit evaluation is specified for the types *Boolean* and *Bit*. The logical operators (**and**, **or**, etc.) do not evaluate short-circuit for *Std_Logic*, but can be exploited as in example 6. The two signals *A* and *B* are of type *Std_Logic*, but each expression within the parenthesis will result in a *Boolean* value, and the or-operator could thus benefit from short-circuit evaluation in case the first parenthesis result is *True*.

```
signal A, B: Std_Logic;
...
if (A='1') or (B='0') then
   ...
end if;
```

Example 6: *Expression with potential short-circuit evaluation.*

3.2.2.6 Conditional statements

A fundamental rule when modelling for simulation performance using VHDL is to only execute code when necessary. Therefore, conditional statements should be used to reduce unnecessary execution of code. The outer conditional statement should reduce the necessity to evaluate enclosed conditional statements, based on an assessment on how often subsequent code needs to be executed. This is done by using nested if- and case-statements, ordered so that the branches with the highest probability are executed first. Conditional expressions in the statements should be ordered for maximum boolean short-circuit evaluation and the complexity should be minimised. Efficiency of conditional statement structures can be analysed using code coverage results. It has been shown that an assignment of a signal is between one and two orders of magnitude more costly in terms of simulation time compared to reading a signal or variable in an if-statement. This suggests that one can use large structures of if-statements to prevent a signal to be unnecessarily assigned, and still gain in simulation performance.

It can be seen in example 3 that the if statements have been nested. The conditional expression in the outer if-statement is the *Rising_Edge* function. The conditional expression in the inner if statement is a custom made *To_X01* function that will detect and report unknown values on the input.

The *Rising_Edge* function is believed to execute faster than the custom *To_X01* function, since it is accelerated in most VHDL simulators, and is consequently placed in the outer if statement that will be executed more frequently. The two conditions are not combined in one expression, not to evaluate any part of the expressions unnecessarily.

The *DEnable* signal in example 3 is used for enabling the tristate buffer isolating the outgoing *D_Out* bus from the external port *D* in the top-level architecture. The type *Boolea*n was being simple to use in if statements and could potentially simulate faster than *Std_ULogic*. Its usage in the top-level architecture is described further in example 22 and figure 4.

3.2.3 Evaluation of simulation performance

The simulation performance of a model should be evaluated continuously during the development to identify simulation bottlenecks and accordingly modify the code for improvements. An analysis of the simulation speed should compare the model intended for board-level simulation and the actual hardware performance, stating the relative simulation performance measured in terms of instructions per second, etc.

The test suite used for this purpose should have a low influence on the performance measurements, but should also reflect realistic simulation scenarios and execute large portions of the model. The measurements should be compared for different simulators and platforms when possible, avoiding simulation pitfalls.

A code coverage utility is a useful means for identifying simulation bottlenecks during model development. The output from such tools usually states the number of times each statement in the source code has been executed during a simulation, allowing an identification of statements frequently executed.

Simulation performance can often be improved by reordering the code or modifying the structure of the conditional statements. It is also possible to identified redundant and unnecessary code that could complicate maintenance.

The Coverage utility in the Synopsys® VSS simulator records the number of times a statement is executed when running a particular simulation. The Leapfrog® VHDL simulator from Cadence has been announced to includes a code profiler, comparable with the Coverage utility, but which also identifies where in the code most of the simulation time is spent.

The VHDLCover™ tool is a simulator independent coverage utility, that adds VHDL code to the model which can then be simulate on any VHDL simulator, and the results can be further analysed the tool. This tool claims to analyse branches taken and other things besides the executed statement count.

3.2.4 Outline of entity and architecture declarations for functional cores

The entity of the functional core should have the same name as the model for board-level simulation but with *_Core* suffixed when only one entity is needed. The architecture name should reflect the nature of its contents: *Behavioural* or *Structural*. When multiple entities are used for the functional core they should be named after their function. If a three-level hierarchy is needed, the name of the second level should have *_Core* suffixed. The *InstancePath* generic should be passed down the hierarchy where the checkers are implemented. The default value should then be the same as the name of the corresponding entity, which will not appear in an assertion report when the *InstancePath* is passed down correctly. It should only be used for sub-module verification during the development.

The entity shown in example 7 has ports of type *Std_ULogic* since no tristate drivers are implemented in the functional core. No type conversion is needed in the top-level architecture between the *Std_Logic* ports of the top-level entity and the *Std_ULogic* ports of the functional core, since they are equivalent and compatible. Ports of *Std_Logic_Vector* type have not been converted to the unresolved *Std_ULogic_Vector* type in the top-level architecture. It is done in the functional core where and when the value is truly used and the ports are therefore of type *Std_Logic_Vector*. The output port *D_Out* of type *Integer* is converted to *Std_ULogic_Vector* in the top-level architecture only when its value is used, and is therefore not converted in the functional core.

```
library IEEE;
use IEEE.Std_Logic_1164.all;

entity BitMod_Core is
   generic(
      InstancePath: String := "BitMod_Core:"); -- For assertions
   port(
      -- System signals
      Test0:   in   Std_ULogic;                -- Test mode
      Clk:     in   Std_ULogic;                -- Master Clock
      Reset_N: in   Std_ULogic;                -- Master Reset
      -- Interface to internal registers
      A:       in   Std_Logic_Vector(0 to 1);  -- Address bus
      CS_N:    in   Std_ULogic;                -- Chip select
      RW_N:    in   Std_ULogic;                -- Read/write
      D_In:    in   Std_Logic_Vector(0 to 7);  -- Data bus input
      D_Out:   out  Integer range 0 to 255;    -- Data bus output
      DEnable: out  Boolean;                   -- Data bus enable
      -- Serial Interface
      SClk:    in   Std_ULogic;                -- Serial clock
      SData:   in   Std_ULogic;                -- Serial input
      MData:   out  Std_ULogic);               -- Serial output
end BitMod_Core;
```

Example 7: *Outline of a functional core entity for a model for board-level simulation.*

All scheduling of output delays and timing checking are performed in the top-level architecture. All type conversions and checking for unknown value on the input ports are performed in the functional core, except the static signals *Reset_N* and *Test* that are converted in the top-level architecture.

The outline of the functional core architecture in example 8 covers some aspects of modelling for functional accuracy and simulation performance described earlier. The *ClkRegion* process covers all functionality related to the *Clk* input and is only made sensitive to the *Clk* and *Reset_N* inputs, being asynchronously reset by *Reset_N*. An if-statement divides the process in four regions: reset of the process, functionality related to the rising *Clk* edge, functionality related to the falling *Clk* edge, and checking for unknown values on *Clk* when neither in reset nor an edge is detected.

The *AsynchronousRegion* process includes all asynchronous functionality in the example and is made sensitive to *Reset_N*, *CS_N* and *RW_N*, which are the only inputs that can induce any changes on the outputs due to an event. An if-statement divides the process in three regions: reset of the process, functionality related to the rising *CS_N* edge and functionality related to *RW_N* input when there is no rising *CS_N* edge.

```
architecture Behavioural of BitMod_Core is
   -- Local signal declarations.
begin
   -- Implementation of all functionality driven by Clk
   ClkRegion: process(Reset_N, Clk)
   begin
      if Reset_N = '0' then
         -- Asynchronous reset of model
      elsif Rising_Edge(Clk) then
         -- Rising Clk edge region
      elsif Falling_Edge(Clk) then
         -- Falling Clk edge region
      else
         -- Check for unknown Clk value, since the model is not
         -- being reset and neither rising nor falling Clk edge
         -- is detected.
         -- No assertions at start up of simulation
         assert not (Is_X(Clk) and (Now /= 0 ns))
            report InstancePath & " 'X' on Clk input"
            severity Error;
      end if;
   end process ClkRegion;

   -- Implementation of asynchronous functionality
   AsynchronousRegion: process(Reset_N, CS_N, RW_N)
   begin
      if Reset_N = '0' then
         -- Asynchronous reset of model
      else Rising_Edge(CS_N) then
         -- Asynchronous behaviour related to CS_N
      else
         -- Asynchronous behaviour related to RW_N
      end if;
   end process AsynchronousRegion;
end Behavioural;
```

Example 8: *Outline of a functional core architecture for a model for board-level simulation.*

3.3 Interface modelling

Models having similar user interfaces for simulation condition selection, the same type and format of error messages etc., provide the user with a single interface to learn and understand. The model interfaces in this document have been made as simple as possible, without necessarily sacrificing the potential of the language, promoting their usage by others than experienced VHDL users.

The following interface modelling aspects are covered in this section:
- Definition of timing parameters;
- Checking for timing constraint violations;
- Scheduling of output delays;
- Management of unknown input values;
- Reporting of model messages.

3.3.1 Timing modelling

In the VHDL Modelling Guidelines, RD1, the timing modelling concept is based on the IEEE VHDL Initiative Toward ASIC Libraries activity, VITAL, as described in RD4. This allows the VITAL subprograms to be reused, saving coding effort as well as potentially offering high simulation performance since several simulators already provide accelerated versions of VITAL subprograms.

Full VITAL compliance has not been achieved since a different technique for the selection of the simulation conditions (e.g. minimum or maximum delay) has been specified. VITAL is based on using an external delay calculator, where the actual timing values for a specific simulation condition are back-annotated using the Standard Delay File format, SDF, which is well adapted for simulation of ASICs.

The ESA timing modelling concept defines the notion of an operating point and provides a single generic with which a user can change the simulation condition for all components.

For models intended for board-level simulation developed for ESA the selection of the simulation condition should be controlled by the *SimCondition* generic of type *SimConditionType* declared in the package *ESA.Simulation*, and should have the default value *WorstCase*. The generic *TimingChecksOn* of type *Boolean* should be used for disabling the timing checkers and should have the default value *False*.

It is recommended to only report timing violations and not to generate unknown values on the outputs. In case it is implemented, the *XGenerationOn* generic of type *Boolean* with the default value *False* should be declared in the top-level entity declaration, disabling generation of unknowns when set to *False*, which should also be the default value. Generation of unknown values on outputs is usually only implemented for components with low complexity where the propagation of the unknown values could be useful to follow and analyse. Note that the VITAL specification uses the name *XOn* instead of *XGenerationOn*.

Normally, the following generics should not be needed. The generic *XChecksOn* of type *Boolean* should be used for disabling the checking for unknown input values, and should have the default value *True*. The generic *MsgOn* of type *Boolean* should be used for disabling the generation of messages from timing checkers, which could be used in conjunction with *XGenerationOn* when only unknown values on the outputs are wanted, and should have the default value *True*.

Since models developed following the proposed scheme are not VITAL compliant, the VITAL compliance checkers possibly found in VHDL analysers should be switched off not to generate unnecessary warning and error messages during the analysis. No back-annotation should be performed using SDF and the negative constraint calculation phase, as specified in RD4, should be disabled not to change the generics used for negative setup and hold constraints as described further in section 3.3.1.2.1.

The two attributes *Vital_Level0* and *Vital_Level1* defined in RD4 should not be used since models for board-level simulation using *ESA.Simulation* are not compliant with either.

3.3.1.1 Timing parameters

The timing parameters should preferably be of the *Time Array* types declared in the packages *ESA.Simulation* and *ESA.Timing* defined in appendix C, supporting most of the types declared in *Vital_Timing* used for timing information. The *Time Array* types are all indexed by the type *SimConditionType*, as shown in example 9, making it possible to select the value corresponding to the simulation condition, which is performed in the top-level architecture. There is no need for having a *Time Array* with *VitalDelayType* elements since they are equivalent to the type *Time* for which a *Time Array* type is already declared in the package *ESA.Simulation*.

The two packages *ESA.Simulation* and *ESA.Timing* should never be redefined or moved to a different library since the intention is to provide only one format for the selection of simulation condition for all models, normally originating from different developers.

```
-- Definition of the SimConditionType type
type SimConditionType is (WorstCase, TypCase, BestCase);

-- Definition of Time Array type, used with Time.
type TimeArray     is array(SimConditionType) of Time;

-- Definition of Time Array types, used with Vital Delay Types.
type TimeArray01   is array(SimConditionType) of VitalDelayType01;
type TimeArray01Z  is array(SimConditionType) of VitalDelayType01Z;
type TimeArray01ZX is array(SimConditionType) of VitalDelayType01ZX;
```

Example 9: *Contents of the Simulation and Timing packages.*

The intended purpose of the *Vital Delay Array Type*s declared in *Vital_Timing* is for specifying the timing for each individual element of an array, such as for data or address signals, and should never replace the usage of the *Time Array* types declared above, since they are indexed with the subtype *Natural* and not the required *SimConditionType*.

No *Time Array* types have been declared in package *ESA.Timing* for any of the *Vital Delay Array Types*, since it is not possible to define a constrained array of unconstrained arrays.

The level of detailed timing information that is represented by the *Vital Delay Array Types* is not necessary for most models. Should such detailed timing information be necessary, the required declarations should be done for those array widths needed and placed in the timing package of the model. Example 10 shows a declaration of a *Time Array* type with *VitalDelayType01ZX* elements that is used for holding timing information related to a *Std_Logic_Vector(0 to 7)* port.

```
type TimeArray01ZX_0_7 is array(SimConditionType) of
                                       VitalDelayArrayType01ZX(0 to 7);
constant tpd_Clk_Data: TimeArray01ZX_0_7;
```

Example 10: *Declaration of Time Array type for a port with individual timing on each of the eight elements, supporting the full Vital Transition Type range.*

The timing parameters needed by the model should be declared in a separate timing package as a deferred constant as it has been shown in example 11. The package body can then be modified if necessary and analysed without the need to re-analyse the complete design, which can be necessary when new data from foundries become available. This also eases distribution of analysed models.

The timing parameters in the timing package can be used directly in the top-level architecture or be passed via generics to the model. The latter option permits the user to modify the timing parameters for each individual component instantiation by using generic maps, useful e.g. when modelling large capacitive loads on boards.

```
library ESA;
use ESA.Simulation.all;
use ESA.Timing.all;

package BitMod_Timing is
   -- Deferred constants for the timing parameters.
...
   constant tpd_Clk_MData: TimeArray01;                        -- T9
...
end BitMod_Timing;
package body BitMod_Timing is
...
   constant tpd_Clk_MData: TimeArray01 :=                      -- T9
                                          ((25 ns, 24 ns),     -- WC
                                           (11 ns, 13 ns),     -- TC
                                           ( 7 ns,  8 ns));    -- BC
...
end BitMod_Timing;
```

Example 11: *Package containing timing parameters declared as deferred constants.*

The constants in the timing package and the generics in the entity declaration can have the same names due to visibility rules in the language. It is therefore no need to have suffixes such as *_Default* attached to the constant names.

Timing parameters should have names compliant with RD4, or alternatively use names from the Data Sheet. It is recommended that timing parameters containing signal names with underscores should be written without them. For example, a timing parameter for the signal *CS_N* could be written as *tperiod_CSN*, which is compliant with RD4. The timing parameter suffixes defined in RD4 should be used where applicable: *posedge* for rising signal edges, *negedge* for falling signal edges etc.

The timing parameters should be given in an integer number of nanoseconds with values rounded in a pessimistic way, to avoid simulation time limitations. Simulators supporting 64 bit implementation of the time counter support approximately 300 years of simulation time with a resolution of 1 ns, and 32 bit implementations support only 2 seconds but should be sufficient for limited hardware and software co-simulation.

3.3.1.2 Timing constraint checking

All timing constraint checkers should be contained in the process *TimingCheck* in the top-level architecture, as outlined in example 12, but the process could be divided for performance reasons when found beneficial. The processes should be sensitive to all signals checked or referenced. All code in the process should be possible to disable with the *TimingChecksOn* generic and a generate-statement, also shown in example 12, to reduce the performance penalty when not used. This outline supports only positive constraints, a version supporting negative constraints is shown in example 16. Timing parameters to be checked should be assigned to the subprogram formals using named association, indexed by the *SimCondition* generic to allow selection of timing parameter values corresponding to the simulation condition as shown in example 15.

Timing constraint checkers should be enabled individually when relevant for the simulation. This should be done by using the subprogram parameter *CheckEnabled*, as shown in example 15 and example 18. Note that this parameter does not prevent the checker from being executed, it only masks the assertion reports and the assertion of the *Violation* parameter in *Vital_Timing* subprograms, and should not be used instead of the *TimingChecksOn* generic used in the generate-statement.

Care should be taken when establishing the conditions for which each checker is enabled, e.g. some checkers are not enabled during reset of the model, other checkers may only be enabled after write operations. Enabling conditions of checkers related to clock period timing constraints should be carefully modelled not to enable when not relevant to the simulation, e.g. during reset. The frequent value changes on clock inputs could cause many unnecessary subprogram invocations, decreasing the simulation performance, and should be disabled for a clock not used in some mode or similar.

Enabling of the checkers for some timing constraints could be a rather complex task, as for the setup and hold checker in example 15. The expression for the enabling variable *DataCheckEnabled* is shown in example 13. The checker is enabled when there is a falling edge on either *CS_N* or *RW_N* while the other input is asserted, which is when the registers are written. It is kept enabled until there is a falling *CS_N* edge while *RW_N* is de-asserted, which is when the registers are read. The checker will therefore be enabled at the beginning of a write access and be kept enabled until the next read access begins.

```
TimingGenerate: if TimingChecksOn generate
   TimingCheck: process(Clk, SClk, D, CS_N, RW_N, Reset_N_X01)
      -- Variables containing information for period checkers
      -- Variables containing information for setup & hold checkers
      -- Variables for enabling timing checkers
   begin
      -- Enabling of various checkers
      -- Reset_N low time w.r.t. Clk checking
      -- Register interfaces checked for illegal events etc.

      -- Checkers using custom made subprograms.
      PeriodCheck(...);           -- SClk period
      CheckWidth(...);            -- CS_N de-assertion width

      -- Timing checkers using Vital_Timing subprograms.
      VitalPeriodPulseCheck(...); -- Clk period, high and low times
      VitalPeriodPulseCheck(...); -- CS_N width for write access=
      VitalSetupHoldCheck(...);   -- D setup & hold w.r.t. CS_N
   end process TimingCheck;
end generate TimingGenerate;
```

Example 12: *The process TimingCheck resides in the top-level architecture.*

If the checker in example 13 was disabled on the rising *CS_N* or *RW_N* edge at the end of the write access, the hold constraint would not be checked since it is normally longer than 0 ns relative to that event. The same enabling scheme could be implemented by delaying the control signals but would have lesser performance. Note the usage of the accelerated functions *Falling_Edge* and *To_X01* in the example.

The type conversion is made locally in the *TimingCheck* process, since the signals are not converted in the top-level architecture but only in the functional core, and will not contribute any simulation performance penalty when timing checking is disabled. It is preferable to formulate a smart enabling condition for an accelerated *Vital_Timing* subprogram than to develop a completely new timing checker.

```
-- Enables the setup and hold checker for D during write operations.
-- The checker is enabled when both CS_N and RW_N are asserted,
-- until the next read access begins, since the data hold constraint
-- is longer than either CS_N or RW_N is de-asserted  after write.
if ((Falling_Edge(CS_N) and To_X01(RW_N)='0') or
    (Falling_Edge(RW_N) and To_X01(CS_N)='0')) then
   DataCheckEnabled := True;                  -- Enable checker
elsif (Falling_Edge(CS_N) and To_X01(RW_N)='1') then
   DataCheckEnabled := False;                 -- Disable checker
end if;
```

Example 13: *A complex enable expression for a setup and hold checker.*

3.3.1.2.1 Timing constraint checking using Vital_Timing subprograms

It is recommended that the timing checkers declared in the *Vital_Timing* package are used. The *Vital_Timing* procedures *VitalSetupHoldCheck*, *VitalRecoveryRemovalCheck* and *VitalPeriodPulseCheck* are declared for the types *Std_ULogic* and *Std_Logic_Vector*. Note that *Std_ULogic_Vector* and *Std_Logic_Vector* are not compatible, but *Std_ULogic* and *Std_Logic* are, which excludes the possibility to check *Std_ULogic_Vector* inputs.

The *VitalSetupHoldCheck* procedure detects a setup or a hold violation on the test signal with respect to the corresponding reference signal. The timing constraints are specified through parameters representing the setup and hold times for low and high test values. This procedure assumes non-negative values for the timing constraints. The setup and hold checker shown in example 15 supports only positive timing constraint values, thus neither of the timing parameters can have a negative value.

The *VitalRecoveryRemovalCheck* detects the presence of a recovery or removal violation on the test signal with respect to the corresponding reference signal. The timing constraints are specified through parameters representing the recovery and removal times associated with a reference edge of the reference signal. This procedure also assumes non-negative values for the timing constraints.

The *VitalPeriodPulseCheck* checks for minimum periodicity and pulse width for low and high values of the test signal. The timing constraint is specified through parameters representing the minimal period between successive rising and falling edges of the test signal and the minimum pulse widths associated with high and low values. Note that the procedure cannot be used for checking maximum period widths, when such checkers are needed they have to be developed separately. Note also that the timing parameter names defined in RD1 containing the suffixes *_min* and *_max* are no longer VITAL compliant, but could still be used for models intended for board-level simulation developed for ESA.

```
-- Variables containing information for checkers
variable Period_Clk: VitalPeriodDataType := VitalPeriodDataInit;
variable Period_CSN: VitalPeriodDataType := VitalPeriodDataInit;
variable Timing_D:   VitalTimingDataType := VitalTimingDataInit;
```

Example 14: *Initialisation of variables used by Vital_Timing subprograms.*

Variables used by the *Vital_Timing* subprograms for storing intermediate results should be initialised as shown in example 14 for correct operation. Most *Vital_Timing* subprograms apply implicitly *To_X01* conversion on their inputs. It is therefore not necessary to perform any type conversions on the signals external to the subprograms, as shown in example 15. These implicit internal type conversions should be implemented in any custom developed timing checkers when possible, to have similar interfaces as *Vital_Timing* subprograms. The severity level of most *Vital_Timing* subprogram assertion reports can be controlled via the *MsgSeverity* parameter as shown in example 15, which makes it possible to implement timing checkers that are compliant to the scheme suggested in RD1. The *RefTransition* parameter used by most *Vital_Timing* subprograms is of type *VitalEdgeSymbolType* which allows the user to specify complex signal transitions for the reference signals. The value ‘*R*’ used in example 15 denotes any possible rising edge.

When it is chosen no to generate any unknown values on outputs at timing violations, it is sufficient to declare only one variable for all the *Violation* parameters of the timing checkers, since the value of the variable will not be used. The parameter *XOn* is used in conjunction with *XGenerationOn*, preventing an ‘*X*’ from being assigned to the *Violation* parameter when a timing violation is detected. It should have no impact when unknown value generation is not implemented.

```
VitalSetupHoldCheck(                  -- D setup & hold w.r.t. CS_N
   Violation      => Violation,
   TimingData     => Timing_D,
   TestSignal     => D,
   TestSignalName => "D",
   RefSignal      => CS_N,
   RefSignalName  => "CS_N",
   SetupHigh      => tsetup_D_CSN(SimCondition),
   SetupLow       => tsetup_D_CSN(SimCondition),
   HoldHigh       => thold_D_CSN(SimCondition),
   HoldLow        => thold_D_CSN(SimCondition),
   CheckEnabled   => DataCheckEnabled,
   RefTransition  => 'R',
   HeaderMsg      => InstancePath,
   XOn            => False,
   MsgOn          => True,
   MsgSeverity    => Warning);
```

Example 15: *Implementation of positive timing constraints using Vital_Timing subprograms.*

Since the *VitalSetupHoldCheck* and *VitalRecoveryRemovalCheck* procedures accept only positive setup and hold values, the relation between the test and reference signals has to be adjusted. Negative setup and recovery times correspond to an internal delay on the reference signal. Negative hold or removal times correspond to an internal delay on the test signal.

Negative timing constraints are handled internally in the model by delaying the test or reference signals using the function *VitalSignalDelay* as shown in example 16. The *VitalSignalDelay* procedure is called in the top-level architecture to delay the appropriate test or reference signal in order to accommodate negative constraint checks. When the delays are associated with other signals they may need to be appropriately adjusted so that all constraint intervals overlap the delayed reference signals.

When negative timing constraints are to be used in a model, two extra timing parameters need to be declared for each setup and hold or recovery and removal pair. Timing parameters on the format *ticd_<ClkPort>* should be used for declaring the time with which a reference signal should be delay, and *tisd_<InPort>_<ClkPort>* should be used for declaring the time with which a test signal should be delayed.

The additional local signals needed and the concurrent procedure calls to *VitalSignalDelay* should all be placed in a block statement within the generate-statement containing the procedure *TimingCheck*, as shown in example 16. It is suggested that the label of the block is named *TimingBlock*. By including the signal declarations in the block instead of in the declarative part of the architecture, the signals will not be allocated if timing checking is disabled, potentially reducing the memory usage. The block statement, the signal declarations and the delaying of the signals should only be used when negative timing values are checked, since it would else unnecessarily reduce the simulation performance.

```
-- In the timing package (and in generic declaration):
constant tsetup_IO_Clk: TimeArray := (10 ns, 10 ns, 10 ns);
constant thold_IO_Clk:  TimeArray := (10 ns, 10 ns, 10 ns);
constant ticd_Clk:      TimeArray := (15 ns,  0 ns,  0 ns);
constant tisd_IO_Clk:   TimeArray := ( 0 ns,  0 ns, 15 ns);

-- In the architecture BoardLevel:
TimingGenerate: if TimingChecksOn generate
   TimingBlock: block
      signal IO_Delay:  Std_ULogic;             -- Delayed test
      signal Clk_Delay: Std_ULogic;             -- Delayed reference
   begin
      VitalSignalDelay(Clk_Delay, Clk, ticd_Clk(SimCondition));
      VitalSignalDelay(IO_Delay,  IO,  tisd_IO_Clk(SimCondition));
      TimingCheck: process(Clk_Delay, IO_Delay)
         variable IO_TD: VitalTimingDataType := VitalTimingDataInit;
         variable Violation: X01             := '0';
      begin
         VitalSetupHoldCheck(
            Violation      => Violation,
            TimingData     => IO_TD,
            TestSignal     => IO_Delay,
            TestSignalName => "IO",
            TestDelay      => tisd_IO_Clk(SimCondition),
            RefSignal      => Clk_Delay,
            RefSignalName  => "Clk",
            RefDelay       => ticd_Clk(SimCondition),
            SetupHigh      => tsetup_IO_Clk(SimCondition),
            SetupLow       => tsetup_IO_Clk(SimCondition),
            HoldHigh       => thold_IO_Clk(SimCondition),
            HoldLow        => thold_IO_Clk(SimCondition),
            CheckEnabled   => True,
            RefTransition  => 'R',
            HeaderMsg      => InstancePath,
            XOn            => False,
            MsgOn          => True,
            MsgSeverity    => Warning);
      end process TimingCheck;
   end block TimingBlock;
end generate TimingGenerate;
```

Example 16: *Implementation of negative timing constraints using Vital_Timing subprograms.*

The calculation of the effective timing constraint values for the setup and hold checker in example 16 is explained in example 17.

```
--------------------------------------------------------------------
-- The effective setup and hold times can be calculated from the
-- tsetup, thold, ticd and tisd timing parameters as follows:
--
-- formulas:  tsetup  = tsetup_IO_Clk - ticd_Clk + tisd_IO_Clk
--            thold   = thold_IO_Clk  + ticd_Clk - tisd_IO_Clk
--            twindow = tsetup + thold
--
-- The formulas will give the following effective parameter values:
--
-- WorstCase: tsetup  = 10 ns - 15 ns +  0 ns = -5 ns
--            thold   = 10 ns + 15 ns -  0 ns = 25 ns
--            twindow = -5 ns + 25 ns         = 20 ns
-- TypCase:   tsetup  = 10 ns -  0 ns +  0 ns = 10 ns
--            thold   = 10 ns +  0 ns -  0 ns = 10 ns
--            twindow = 10 ns + 10 ns         = 20 ns
-- BestCase:  tsetup  = 10 ns -  0 ns + 15 ns = 25 ns
--            thold   = 10 ns +  0 ns - 15 ns = -5 ns
--            twindow = 25 ns + -5 ns         = 20 ns
--
-- The timing parameters define a negative setup time of -5 ns for
-- WorstCase, 10 ns setup and 10 ns hold for TypCase, and negative
-- hold time of -5 ns for BestCase. For all cases is the window
-- 20 ns wide in which the IO input must be stable.
--
-- WorstCase, setup -5 ns, hold 15 ns, window 20 ns:
--                   -->|      stable region      |<--
-- Test      XXXXXXXXXXX____________________________XXXXXXXXXXX
--               ______________________________________________
-- Reference ___/
--              |    -->|                          |<-- hold
--           -->|       |<-- negative setup
--
-- TypCase, setup 10 ns, hold 10 ns, window 20 ns:
--                   -->|      stable region      |<--
-- Test      XXXXXXXXXXX____________________________XXXXXXXXXXX
--                                    _________________________
-- Reference ________________________/
--                      |         -->|             |<-- hold
--                   -->|   setup    |<--
--
-- BestCase, setup 15 ns, hold -5 ns, window 20 ns:
--                   -->|      stable region      |<--
-- Test      XXXXXXXXXXX____________________________XXXXXXXXXXX
--                                                          ___
-- Reference ______________________________________________/
--                      |         negative hold -->|       |<--
--                   -->|         setup            |<--
--------------------------------------------------------------------
```

Example 17: *Calculation of negative timing constraints*

3.3.1.2.2 Timing constraint checking using non-Vital_Timing subprograms

There may be some timing constraint types that are unique for a design and cannot easily be checked using subprograms from the *Vital_Timing* package. In such cases it is preferable to develop new timing checkers for the model instead of inefficiently using *Vital_Timing* subprograms. Format and parameter names could resemble those of the *Vital_Timing* subprograms, to make it easier for the user to recognise each parameters purpose and usage. Each parameter should have a default value when possible, allowing the user to assign only those parameters needed for the application.

For example, a timing checker could have a parameter specified in number of clock periods. Since the clock period can vary, no fixed time value can be provided to a *Vital_Timing* subprogram. It is usually possible to examine the actual design and transform the timing constraint stating number of periods to state number of relevant clock edges, which are easier to detect and count than clock periods.

A subprogram with the declaration shown in example 18 could cover several such timing constraint types. An inappropriate solution would be to sample the clock period and to use the measured value as a parameter to a *Vital_Timing* subprogram. That approach would not be able to handle clocks with irregular although correct behaviour, e.g. a clock with changing period length.

```
---------------------------------------------------------------------
-- Checks the relation between two clocks
-- If FasterThanRef = True,
-- then the TestSignal may not have more than Period rising edges
-- between two RefSignal rising edges.
-- If FasterThanRef = False, then vice versa.
---------------------------------------------------------------------
procedure PeriodCheck(
   variable Violation:      out   X01;
   variable PeriodData:     inout Integer;
   signal   TestSignal:     in    Std_ULogic;
   constant TestSignalName: in    String         := "";
   signal   RefSignal:      in    Std_ULogic;
   constant RefSignalName:  in    String         := "";
   constant Period:         in    Integer        := 0;
   constant FasterThanRef:  in    Boolean        := True;
   constant CheckEnabled:   in    Boolean        := True;
   constant HeaderMsg:      in    String         := "PeriodCheck:";
   constant XOn:            in    Boolean        := True;
   constant MsgOn:          in    Boolean        := True;
   constant MsgSeverity:    in    Severity_Level := Warning);
```

Example 18: *Custom made period checker that checks the relation between two clocks.*

There are types of checkers that could be directly incorporated in the *TimingCheck* process since their small code sizes do not motivate development of subprograms. An example could be a checker verifying that an input signal does not change while the reference signal is asserted. Some memories require that the read/write selector may not change while the component is selected. This is checked for in example 19, and could be further optimised not to report unnecessary events, such as transitions between equivalent

Std_Logic strengths, by first converting the input *RW_N* to the subtype *X01*. Note that *CS_N* requires *To_X01* conversion in this example, since not previously done in the top-level architecture and both the strengths '*0*' and '*L*' should be interpreted as asserted.

```
-- RW_N may not change when CS_N is asserted.
assert not (RW_N'Event and To_X01(CS_N)='0' and Reset_N_X01='1')
   report InstancePath & " RW_N event while CS_N asserted"
   severity Warning;
```

Example 19: *Simple checker that could be included in the timing checker process.*

Two types of timing constraints can be identified; timing parameters that change with the simulation condition and timing parameters that are fixed by the architectural design. An example of the former is an absolute setup time related to a clock edge. An example of the latter is a setup time that is related to the number of periods of a reference clock.

The first type could vary for different simulation conditions and should therefore be included as a deferred constant in the timing package, allowing to be changed by the user. These timing constraints can normally be checked using *Vital_Timing* subprograms as described in section 3.3.1.2.1.

```
-- Asserts that there are at least tpw_ResetN_negedge number of
-- falling Clk edges during the assertion of Reset_N.
-- This timing checker is approximated w.r.t. the data sheet.
-- Only the number of Clk edges are counted for tpw_ResetN_negedge.
if (Falling_Edge(Clk) and Reset_N_X01='0' and (Period_Reset>0)) then
   Period_Reset := Period_Reset - 1;
end if;

if Falling_Edge(Reset_N_X01) then            -- Reset begins
   Period_Reset := tpw_ResetN_negedge;
elsif Rising_Edge(Reset_N_X01) then          -- Reset ends
   assert (Period_Reset = 0)
      report InstancePath & " Signal width too short on Reset_N"
      severity Error;
end if;
```

Example 20: *Check of timing constraint on the Reset_N input.*

The second type of parameter will normally not change, since established by the design of the component, and could be declared in the declarative part of the top-level architecture or in the *TimingCheck* process. The checker in example 20 verifies that *Reset_N* is asserted during a minimum number of *Clk* periods. Its implementation is somewhat approximated, only checking number of falling *Clk* edges. It could be included in the *TimingCheck* process due to its small size.

3.3.1.3 Scheduling of output delays

Scheduling of output values with appropriate timing delays should be performed in the top-level architecture, not to introduce timing related information in the functional core. This could simplify the design and verification of the functional core since the behaviour of synchronous designs will mostly be related to the clock cycle.

When adequate timing information is not available in the Data Sheet, the design house or foundry responsible for the design should be consulted. Each output delay should be related to the relevant driving clock or signal edge, and should be modelled to reflect the actual component. The output values on signals being driven should be restricted to the *Std_Logic* strengths '*U*', '*0*', '*1*', '*X*' and '*Z*'. Signals with internal pull up or pull down could be assigned the strengths '*L*', '*H*' and '*W*' as well. The *don't care* strength '-' should not appear at any output, not representing a logic level that should be expected in a board design. Local signals in the top-level architecture not being delayed, i.e. outputs from the functional core, should have *_NoTime* suffixed to their names.

Scheduling of output delays should be done by using concurrent signal assignments. *VitalPathDelay* is a function used to select the propagation delay path and schedule a new output value, but should be avoided since most output delays could be implemented by the simpler and faster **after** construct. It is normally sufficient to have a single timing parameter for both the rising and falling signal transitions. The *VitalCalcDelay* function could be used when more elaborated timing modelling is required, as shown in example 21. The function accepts the *Vital Delay Types* and selects the correct delay time based on the previous and new signal values.

```
MData <= MData_NoTime after VitalCalcDelay(
                             NewVal => MData_NoTime,
                             OldVal => MData_NoTime'Last_Value,
                             Delay  => tpd_Clk_MData(SimCondition));
```

Example 21: *VitalCalcDelay usage.*

Complex timing relations can be modelled with small code size overhead. Example 22 shows a memory interface where the timing of output data is related to both the chip select signal and to the arrival of the address. It illustrates that complex timing relations need not be modelled in the functional core, but can readily be done in the top-level architecture.

```
-- Generation of tristate or drive for the external data bus.
-- D_NoTime is delayed w.r.t. the address. DEn_NoTime is delayed,
-- with different timing for tristate. The D assignment includes an
-- Integer to Std_Logic_Vector conversion.
DEn_Delayed <= transport DEn_NoTime after
                                   tpd_CSN_D_negedge(SimCondition)
                  when DEn_NoTime else
               DEn_NoTime after tpd_CSN_D_posedge(SimCondition);
D_Delayed   <= transport D_NoTime   after tpd_A_D(SimCondition);
D           <= To_StdLogicVector(D_Delayed, 8)
                  when DEn_Delayed else (others => 'Z');
```

Example 22: *Timing and tristate modelling of the data output of a memory interface.*

3.3.2 Management of unknown input values

Inputs should be checked for unknown values which should be reported to the model user. This information is very useful since the behaviour of the real component is normally not specified for unknown input values and could result in failure. The nine *Std_Logic* strengths are reduced to the three strengths '*0*', '*1*' and '*X*', where '*0*' corresponds to the equivalent strengths '*0*' and '*L*', where '*1*' corresponds to the equivalent strengths '*1*' and '*H*', and '*X*' corresponds to the rest of the strengths consider being incorrect input values. The detection, propagation and handling of unknown input values should be implemented in accordance with RD1, and be documented in the source code header.

Type conversion and checking for unknown values on inputs should be performed only where and when the data are used. This can be performed per statement or per process if the value is evaluated and used in more than one place at the same time. The reason for this is to avoid unnecessary type conversion when events occur on inputs but the value is not used by the model. It will also reduce the number of unnecessary assertion reports. This can be done in the top-level architecture as well as in the functional core, depending on the usage of the input value.

For example, if an input is only used in one process in the functional core the type conversion should be then done only there, on the other hand if an input is used in several processes in which it is evaluated simultaneously then the type conversion could be performed in the top-level architecture to reduce the number of conversions and reports. A trade off based on measured simulation performance should be performed before deciding which approach is the most efficient. Detection of unknown input values should be inhibited when not relevant to the simulation, not to induce unnecessary assertion reporting, e.g. during reset of the model.

The severity level of assertion reports when unknown values are detected should be related to their impact on the simulation as specified in RD1. An unknown value changing the state of the model should have the severity level *Error*, but when only changing the data being consumed or produced and not affecting the state of the model it should have the severity level *Warning*.

Inputs should be converted to the *X01* subtype of *Std_ULogic*, reducing the number of signal strengths to be handled in the functional core. Signals being converted should have *_X01* suffixed to their names. A *To_X01* function could be implemented using the functions *To_X01* and *Is_X* together with an assertion statement to both make the type conversion and check for unknown input values.

Conversion of inputs of the type *Std_Logic_Vector* to the subtype *X01* should only be done when necessary. In example 3 the data and address buses are converted directly to the type *Integer* when being used in the functional core. A custom subprogram could be developed which converts an *Std_Logic_Vector* array to the type *Integer* and also performs checking, reporting and handling of unknown values. No type conversion would therefore be needed for the two buses in the top-level architecture.

Propagation of unknown values to outputs should be implemented in the functional core and only be done when not changing the state of the model, i.e. unknown values should only be propagated when used as data. The propagation of unknown values is useful for tracing the migration of possible faults in a system. The propagation is normally closely related to the functional behaviour of the model and could therefore benefit from being implemented in the functional core, reducing any simulation performance penalties.

Handling of unknown input values on signals used for clocking and latching should be done by using functions such as *Rising_Edge*, *Falling_Edge* and *Is_X* when possible, reducing the need of *To_X01* conversions since the functions perform the conversion implicitly an are accelerated for simulation performance. If-statements should be structured to explicitly check whether an input has the desired value as shown in example 23, never relaying on an else statement.

The input *CS_N* in example 3 is checked for unknown values in the else statement when the model is not being reset and no rising *CS_N* edge has been detected, and is done for simulation performance reasons. Unknown values on *CS_N* are treated as '*1*' during read and write accesses, i.e. the access is being ignored.

Clock inputs could be checked in the functional core with an efficient checking scheme using if statements as shown in example 8.

```
if CS_N_X01='0' and RW_N_X01='0' then
   -- Write access, unknown value on CS_N_X01 or RW_N_X01 is
   -- handled as '1' which will not activate the access.
elsif CS_N_X01='0' and RW_N_X01='1' then
   -- Read access, unknown value on CS_N_X01 or RW_N_X01 is
   -- handled as '1' which will not activate the access.
else
   -- Neither write nor read access, check for unknown values.
end if;
```

Example 23: *Handling of unknown input values.*

Inputs that are used in more than one process or that change state infrequently can be checked for unknown values in the top-level architecture. To avoid unnecessary invocation of processes, these checkers should be divided in two or more processes as shown in example 8, grouping inputs together that change with comparable frequencies.

The *To_X01* conversion could be performed in the top-level architecture when an input is used in a static manner, e.g. mode pins that are normally held to defined levels when used, or used in many concurrent statements in the functional core. In example 24, static inputs or inputs with infrequent state changes are checked for unknown values in the *CheckStaticInputs* process. The checking is only performed when reset is inactive or when reset is deactivated, requiring the *Reset_N* signal to be in the sensitivity list of the process. Static inputs that are not allowed to change after reset are checked for such illegal events as well. All checkers in the example are disabled during the initial simulation cycle to prevent unnecessary assertion reporting.

The production test is activated by the *Test* input but is not modelled in the example and any attempt to use it will result in an assertion report with the severity level *Note*. The *Reset_N* input is also checked for unknown values in this process, using the *Is_X* function only for the sake of conformance since already being converted to the subtype *X01*. Clock inputs and other inputs with frequent state changes are checked for unknown values in the *CheckDynamicInputs* procedure in example 25, which is kept simple to reduce the performance penalty for each invocation. The *Is_X* function will make the *To_X01* conversion implicitly.

```
CheckStaticInputs: process(Reset_N_X01, Test_X01)
begin
   if (Reset_N_X01='1') and (Now /= 0 ns) then
      -- No assertions at start-up or when Reset_N is asserted
      -- The Test input is a vector, element 0 is used for
      -- production test only.
      assert (Test_X01(1)='0')
         report InstancePath & " Prod. test not modelled"
         severity Note;
      assert not Is_X(Test_X01)            -- Note: done on a vector
         report InstancePath & " 'X' on Test input"
         severity Error;
      -- Check if the static pin changed after reset
      assert not Test_X01'Event
         report InstancePath & " Test changed after reset"
         severity Error;
   elsif Reset_N_X01'Event and (Now /= 0 ns) then
      -- Check for X on Reset_N
      assert not Is_X(Reset_N_X01)
         report InstancePath & " 'X' on Reset_N input"
         severity Error;
   end if;
end process CheckStaticInputs;
```

Example 24: *Check for unknown values on static inputs.*

Separation of dynamic and static input checking is important when the number of static inputs is large and would slow down the simulation if each was checked for every event that occurred on the dynamic inputs. It can be worthwhile to assess the performance impact on whether to use more than one process when checking dynamic inputs.

```
--------------------------------------------------------------------
-- Check for unknown values on the SClk input.
-- The Clk input is checked in the functional core.
--------------------------------------------------------------------
CheckDynamicInputs: process(SClk)
begin
   -- No assertions at start
   assert not (Is_X(SClk) and (Now /= 0 ns))
      report InstancePath & " 'X' on SClk input"
      severity Error;
end process CheckDynamicInputs;
```

Example 25: *Check of dynamic inputs which change states frequently.*

3.3.3 Reporting model messages

Each message should contain the *InstancePath* generic in the beginning of the report string. In case of more than one hierarchical level, the sub-level path should be the same as for the top-level entity, since the user would normally not need to know from which of the subcomponents the report originates. The purpose of the *InstancePath* generic is to provide the user with a mean for naming each instance with a unique name corresponding to the component identifier on the simulated board. This scheme makes it easier for the user to identify the source of a report than if all the instances of the same component type would produce exactly the same name.

The *InstancePath* generic should have a default value corresponding to the model name as shown in example 26. A colon is used as a delimiter in the *String* returned by the attributes *Path_Name* and *Instance_Path* defined for VHDL '93, and should therefore also be used for the *InstancePath* generic in models for board-level simulation. The usage of the *InstancePath* generics will be discussed further in section 5 where an example of a board design is presented.

```
  InstancePath: String := "BitMod:";
```

Example 26: *Declaration of InstancePath generic with default value.*

No features for masking assertions and their reports need to be included in the model, since most VHDL simulators can stop the simulation at a preset severity level.

3.3.4 Outline of entity and architecture declarations for models

The generics and ports of the top-level entity provide the interface of the model. Generics are used for passing timing parameters and to select simulation conditions etc. Ports represent the pins of the component. The generic and port names should be selected in accordance with RD1 and section 3.3.1. Any control parameters and files names used by the model should be declared as generics, allowing them to be flexibly selected using configuration declarations.

The only standard packages which should be made visible when using a model for board-level simulation are *Standard*, *TextIO*, *Std_Logic_1164*, *Vital_Timing*, *Vital_Primitives*, *ESA.Simulation* and *ESA.Timing*. The only model specific packages made visible are those containing the timing parameters of the models. The packages made visible to the entity declaration should only be *Std_Logic_1164*, *ESA.Simulation* and *ESA.Timing*.

The entity declaration in example 7 contains the simulation condition selector *SimCondition*, *InstancePath* for messages, and *TimingChecksOn* for enabling the timing checkers, all of them having default values. The timing parameters are passed to the model via the timing generics declared as types declared in *ESA.Simulation* and *ESA.Timing*, each containing three values for the simulation conditions. Note the usage of the suffixes *posedge* and *negedge*. The default values are fetched from the *BitMod_Timing* package already made visible to the entity. There is no need to have different names for the timing constants in the timing package and the timing generics.

The VITAL packages have not been made visible since the types declared there are not used explicitly. If a timing generic would need to contain individual timing data for each element of a *Std_Logic_Vector*, the needed *Time Array Type* would have been declared in the *BitMod_Timing* package and would be visible to the entity. The ports are grouped after functionality and the arrays have the most significant bit to the left. Each parameter has been commented on the line where being declared.

```
library IEEE;
use IEEE.Std_Logic_1164.all;

library ESA;
use ESA.Simulation.all;
use ESA.Timing.all;

library BitMod_Lib;
use BitMod_Lib.BitMod_Timing.all;            -- Default timing

entity BitMod is
   generic(
      SimCondition:      SimConditionType := WorstCase;
      InstancePath:      String           := "BitMod:";
      TimingChecksOn:    Boolean          := False;

      tperiod_Clk:       TimeArray        := tperiod_Clk;    -- TClk
      tpw_Clk_posedge:   TimeArray        := tpw_Clk_posedge;-- TCHi
      tpw_Clk_negedge:   TimeArray        := tpw_Clk_negedge;-- TCLo
      tpw_CSN_negedge:   TimeArray        := tpw_CSN_negedge;  -- T1
      tsetup_D_CSN:      TimeArray        := tsetup_D_CSN;     -- T2
      thold_D_CSN:       TimeArray        := thold_D_CSN;      -- T3
      tpd_CSN_D_negedge: TimeArray        := tpd_CSN_D_negedge;-- T4
      tpd_CSN_D_posedge: TimeArray        := tpd_CSN_D_posedge;-- T5
      tpd_A_D:           TimeArray        := tpd_A_D;          -- T6
      tpd_Clk_MData:     TimeArray01      := tpd_Clk_MData);   -- T9

   port(
      -- System signals (4)
      Test:    in     Std_Logic_Vector(0 to 1);     -- Test mode
      Clk:     in     Std_Logic;                    -- Master Clock
      Reset_N: in     Std_Logic;                    -- Master Reset

      -- Interface to internal registers (12)
      A:       in     Std_Logic_Vector(0 to 1);     -- Address bus
      CS_N:    in     Std_Logic;                    -- Chip select
      RW_N:    in     Std_Logic;                    -- Read/write
      D:       inout  Std_Logic_Vector(0 to 7);     -- Bidirectional

      -- Serial Interface (3)
      SClk:    in     Std_Logic;                    -- Serial clock
      SData:   in     Std_Logic;                    -- Serial input
      MData:   out    Std_Logic);                   -- Serial output
end BitMod;
```

Example 27: *Outline of entity declaration for a model for board-level simulation.*

The architecture declaration in example 28 contains: component declarations for the functional core; local signal declarations; timing parameters which are fixed by the architecture and do not change with simulation conditions; *To_X01* conversion for static inputs and inputs used in multiple processes; process for checking dynamic inputs for unknown values; process for checking static inputs for unknown values; process for timing checking contained in a generate-statement; assignment of output delays and instantiation of subcomponents. Each components should be declared including at least the *InstancePath* generic when any checking is performed in the functional core. The component instantiation should associate the *InstancePath* generic of the entity to that of the component. The packages *Vital_Timing* and *BitMod_Definition* which contains custom made timing checkers are made visible only to the architecture, not to the entity.

```
library IEEE;
use IEEE.Vital_Timing.all;

library BitMod_Lib;
use BitMod_Lib.BitMod_Definition.all;        -- For custom functions

architecture BoardLevel of BitMod is
   -- Component declarations.
   -- Local signal declarations.
   -- Declaration of timing parameters not to be changed by user.
begin
   -- Strength stripping to X01 for some signals using
   -- Std_Logic_1164 subprogram To_X01.

   -- Check for unknown values on dynamic input.
   CheckDynamicInputs: process(...)
   begin
   end process CheckDynamicInputs;

   -- Check for unknown values on the static inputs.
   CheckStaticInputs: process(...)
   begin
   end process CheckStaticInputs;

   -- Timing checks on inputs (setup, hold, period, pulse width).
   TimingGenerate: if TimingChecksOn generate
      TimingCheck: process(...)
      -- Variables containing information for checkers
      begin
         -- Enabling of various checkers.
         -- Checkers implemented in the process.
         -- Checkers using custom made subprograms.
         -- Timing checkers using Vital_Timing subprograms.
      end process TimingCheck;
   end generate TimingGenerate;

   -- Assignment of output delays.
   -- Instantiation of subcomponents.

end BoardLevel;
```

Example 28: *Outline of a top-level architecture.*

4 VERIFICATION OF MODELS FOR BOARD-LEVEL SIMULATION

Model verification is performed to ensure that the model for board-level simulation fulfils its requirements, both on functionality and timing. The method outlined in this section is based on the existence of a *component model* for comparison. When such model is not available, emphasis should be put on performing the verification independently from the model development. Two categories of verification can be identified; to ensure that the model has the same functionality as the *component model* (performed during model development), and to verify that the model works for a certain combination of simulator and platform (performed by the user with test suites provided by the model developer).

The first category of verification should be performed at the end of the model development to ensure that the model reflects the functionality of the component. Such verification should include all functional test stimuli used during the component development. It is the responsibility of the model developer to verify the functionality of the subcomponents of the model. Normally this type of verification is best performed by the designer who knows all the details of the model. Subcomponent verification should be an exhaustive test which includes every input combination with both legal and illegal values, and be applied for every state. Exhaustive testing at higher level is not always feasible because of complexity reasons. It could be advantageous to begin the development of the test programme already during the design of the component when a development of a model for board-level simulation is foreseen. A verification of the model should be performed by placing it in a typical board design to detect any system problems.

The second category of verification should be performed by the user when installing the model in his particular simulation environment to ensure that it will operate correctly when simulated. This should be done by using a test bench provided by the model developer, developed in accordance with RD1, including one or more test benches, reporting whether a test has passed or failed etc. The reason for performing a verification when a model is installed is that there are still some differences between VHDL simulators. Each model should preferably be verified by the model developer for more than one combination of computer platform, operating system and simulator before being released. The test should allow automatic verification and be suitable as a maintenance vehicle for verifying the model after modifications. As for all verification activities, the test suites should be developed by somebody not involved in the development of the model itself to avoid masking of errors.

The structure of the test environment differs between the two categories. For the first category several different structures can be used, depending on the simulator. For the simulation of the model in its environment, the test structure will normally be developed as a board design, which is described in section 5.1. Test structure for the second category should provide full controllability and observability of the test object.

The following sections will present an approach to development of test benches for the second category of verification. The first category of verification should also follow the suggestions presented below as far as possible. Since being more related to the stimuli and data from the component development, it could be performed in several ways and is therefore not completely covered in this document.

4.1 Test bench

The test bench contains the model to be verified, referred to as the test object, a test generator, and occasionally objects external to the test object that are necessary for its operation, such as memories or buffers, as outlined in figure 5. For complete verification, all external objects should be modelled in the test generator, e.g. protocol machines, bus interfaces etc. allowing for generation of non-nominal stimuli such as inducing incorrect or corrupted accesses, error injection etc., which is normally only possible when having full controllability and observability. The test generator can have several architectures implementing different test suites.

The test bench should have no ports or generics in the entity declaration, since this is potential not portable. Therefore should there be no needed to make any declarations in the test bench entity and it should be independent of any packages or design units, i.e. the entity declaration should be completely empty.

Each architecture of the test bench should have a purely structural composition reflecting only possible interconnections of physical components and contain no functionality, allowing the test bench to be replaced with a schematic containing the VHDL test generator and the *component model* when using mixed-level simulation. Signals not present as physical pins on any external objects, e.g. debugging umbilical to microprocessor model, should not be connected to the test object, since the test object is used for Board-Level simulation where only physically possible connections are allowed.

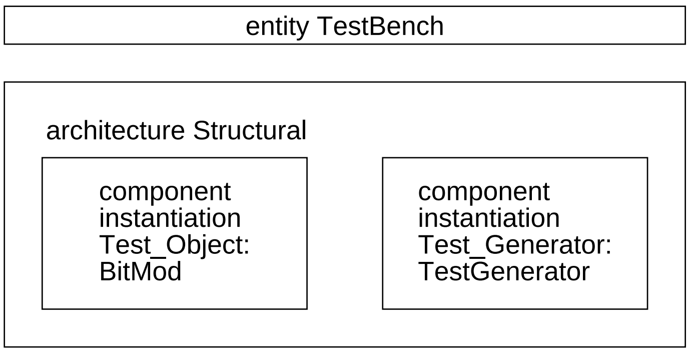
entity TestBench
architecture Structural
component instantiation Test_Object: BitMod
component instantiation Test_Generator: TestGenerator

Figure 5: *Entity and architecture of the TestBench.*

Each component declaration in the test bench architecture should have the same name, generic and port declarations as for the corresponding entity. Since the test bench root entity should not have any generics, the selection of the simulation condition and test suite could either be made in the test bench architecture or by using configuration declarations. The selection of the simulation condition is necessary for the verification of the timing interfaces of the model.

The first approach is to have one test bench architecture for each simulation condition and test suite pair. The component instantiation and interconnection of the test object and test generator will be the same for all architectures, only differing in the values associated in the generic maps and the selection of the test generator architecture in the configuration specification.

The second approach is more advanced and utilises the full power of the language and yields less code. The component declarations for the test object and the test generator should only contain the ports declared for each entity, not any generic declarations, as shown in example 29.

```
entity TestBench is
end TestBench;

library IEEE;
use IEEE.Std_Logic_1164.all;

architecture Structural of TestBench is
   component BitMod
      port(...);
   end component;
   component TestGenerator
      port(...);
   end component;
   -- Local signal declarations.
   begin
   Test_Object: BitMod
      port map(...);
   Test_Generator: TestGenerator
      port map(...);
end Structural;
```

Example 29: *Outline of entity and architecture of test bench.*

Configuration declarations are used to bind the component instances in the test bench architecture using configuration specifications; selecting corresponding entities and architectures, binding port and generic formals with actuals. Each test suite is selected by its architecture name in the configuration specification for the test generator instance, as shown in figure 6.

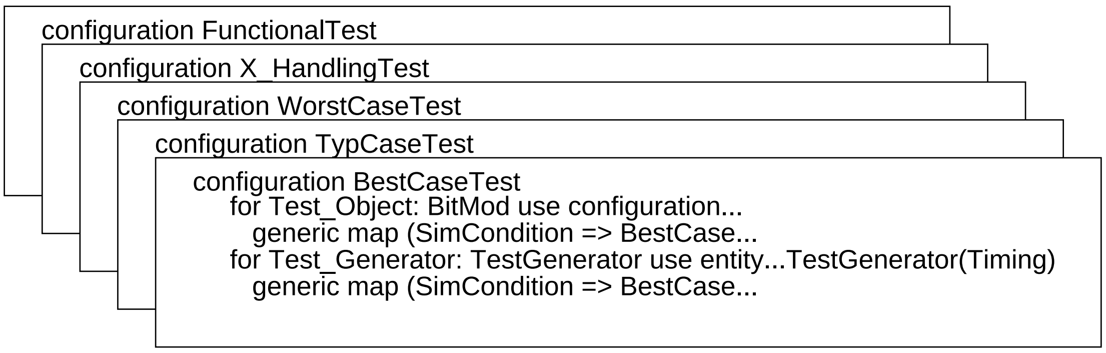
configuration FunctionalTest
configuration X_HandlingTest
configuration WorstCaseTest
configuration TypCaseTest
configuration BestCaseTest
for Test_Object: BitMod use configuration...
generic map (SimCondition => BestCase...
for Test_Generator: TestGenerator use entity...TestGenerator(Timing)
generic map (SimCondition => BestCase...

Figure 6: *Configuration declarations for the TestBench.*

The configuration declaration of the test object should be selected in the configuration specification, as shown for *BitMod_Configuration* in example 30, since the test object normally has a hierarchy. The binding of the component and entity ports is done by default named association when their declarations are identical. Values can be assigned to the entity generics using generic maps in the configuration specifications, e.g. timing parameters, file names, simulation condition, since the generic declarations have been omitted in the component declaration in the architecture. The declarations made in the *ESA.Simulation* package should be made visible to the configuration declarations using library and use statements as shown in example 30.

```
library BitMod_Lib;

library BitMod_TB_Lib;

library ESA;
use ESA.Simulation.all;
use ESA.Timing.all;

configuration FunctionalTest of TestBench is
   for Structural
      for Test_Object: BitMod
         use configuration BitMod_Lib.BitMod_Configuration
            generic map(
               SimCondition   => WorstCase,
               InstancePath   => ":TestBench:Test_Object:",
               TimingChecksOn => False);
      end for;
      for Test_Generator: TestGenerator
         use entity BitMod_TB_Lib.TestGenerator(Timing)
            generic map(
               SimCondition   => WorstCase,
               InstancePath   => ":TestBench:Test_Generator:");
      end for;
   end for;
end FunctionalTest;
```

Example 30: *Outline of configuration declaration of test bench.*

By providing one configuration declaration for each simulation condition and test suite pair, the amount of code duplication can be reduced compared to the first approach since only one architecture is necessary for the test bench. It is generally a good practice to select only configuration declarations, not entities, when binding components of a test bench with configuration specifications. The test bench, test generator and its packages should be analysed to a library separate from the test object. The name of the library should be the same as for that of the test object, but with *_TB_Lib* suffixed instead of *_Lib*.

4.2 Test object

The test object is instantiated in the test bench architecture as explained above. The component declaration should be identical to the entity declaration, but should not include the generics when the second approach described above is used for the selection of simulation conditions and test suites. The development of the test object has been described in section 3.

4.3 Test generator and result checker

The test generator should generate stimuli and acquire response data for comparison with the expected results in order to verify the behaviour of the test object. The test generator could have more than one architecture, implementing different test suites with different functions as outlined in figure 7. An architecture of a test generator should include processes that generate the test suite, evaluate test results, generate list files, perform output data compression etc. Since several of these functions are normally used in more than one architecture, subprograms could be declared in a separate package or in the declarative part of the entity.

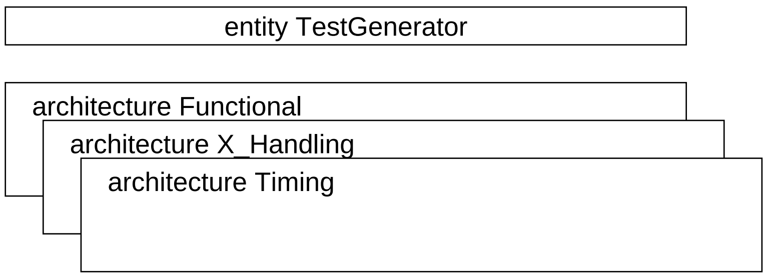

Figure 7: *Entity and architectures of the TestGenerator.*

The test generator entity should have the same port declaration as the test object, but with the opposite directions for the signal flow, as shown in example 31. The entity declaration should include the *InstancePath* generic and *SimCondition* generic for the selection of simulation condition as described in section 3.3.1, to allow for efficient verification of timing checkers. Enabling of optional functions, e.g. generation of log files or test statistics, should be done using generics to allow the usage of configuration declarations. File names should be passed to the test generator using generics of type *String*, enabling the users to provide a file path supported by their operating system.

When test suites consist of several sequential sub-tests, each such sub-test should run independently from the preceding tests, i.e. the sub-test results should not affect the adjacent sub-tests. This is useful during the development of the test suite to reduce the simulation time by being able to exclude preceding sub-test in long simulation runs. Each sub-test should always begin and end with the test object in a known state and each signal being observed by the test bench should have the same value, which is normally the case when the model is reset. When compressing the output data, the signature should be checked and reset between sub-tests, allowing modifications of sub-tests without changing the signatures for all other sub-tests, as described in section 4.3.3

Each test or sub-test should report whether it has passed or failed, using the severity levels *Note* for passed and *Error* for failed and the generic *InstancePath*, as shown in example 32. When a sub-test has failed the assertion report should identify the error to allow tracing of the fault, including the name of the test suite and state the simulation condition. The criterion used for determining whether a test has passed or failed should be documented, i.e. state what is automatically verified. One method to indicate that a test suite is completed is to generate an assertion report with the severity level *Failure*.

Unnecessary assertion reports should be avoided, reducing the amount of output to be reviewed by the user.

When input and output files are used they should be of the type *Std.TextIO.Text* to ensure portability. Binary files should not be used as input or output to the test generator. If the natural representation of data is in binary format, e.g. in image processing applications, a C program converting data between the binary and the hexadecimal representation, and vice versa, should be provided with the binary files.

```
library IEEE;
use IEEE.Std_Logic_1164.all;

library ESA;
use ESA.Simulation.all;

entity TestGenerator is
   generic(
      SimCondition:   SimConditionType := WorstCase;
      InstancePath:   String           := "TestGenerator:");
   port(
      -- System signals (4)
      Test:    inout  Std_Logic_Vector(0 to 1); -- Test mode
      Clk:     inout  Std_Logic := '0';         -- Master Clock
      Reset_N: inout  Std_Logic;                -- Master Reset
      -- Interface to internal registers (12)
      A:       inout  Std_Logic_Vector(0 to 1); -- Address bus
      CS_N:    inout  Std_Logic;                -- Chip select
      RW_N:    inout  Std_Logic;                -- Read/write
      D:       inout  Std_Logic_Vector(0 to 7); -- Bidirectional
      -- Serial Interface (3)
      SClk:    inout  Std_ULogic := '0';        -- Serial clock
      SData:   inout  Std_ULogic := '0';        -- Serial input
      MData:   in     Std_Logic);               -- Serial output
end TestGenerator;
```

Example 31: *Outline of an entity declaration for a test generator.*

The test generator need not be optimised for simulation performance. However, simulation lengths should not discourage the user from performing the verification. The usage of wait-statements has been shown efficient due to their flexibility when modelling test suites, even though not having optimum simulation performance. The duration of the simulated time of each test suite should be documented.

Test suites could be assembled using calls to procedures that encapsulate low level interfacing to the test object. These procedures should be implemented with all timing values passed as parameters, allowing easy modification of the interfaces.

There should be a separation between the verification of the functionality of the test object and the verification of its interface modelling. The following sections will more in detail describe the purpose and implementation of these two different verification procedures.

4.3.1 Verification of functionality

The functional verification should cover the full functionality of the test object. The test suites implementing this are subject to several diverse requirements. The test suite should detect any differences between the board-level and detailed models, employing several different testing methods as outlined below. It should be possible to run the test suite for both the test object and the *component model* during model development. It should be possible to evaluate the efficiency of the test suite using fault simulation as described in section 4.4. The results from a simulation performed by the user should either be compared to a reference file or with reference signatures when data compression is used to determine whether a test has passed or failed.

The development time of the functional test suite should not be underestimated. It has been often shown that it takes at least the same amount of time as the development of the test object itself.

To ensure that any differences between the two models are detected, all inputs should be asserted a couple of clock cycles before and after their expected sampling points, implementing a sliding window. It is sufficient to verify the clock cycle behaviour for fully synchronous designs. The reset of the test object should be treated as any other functionality and be adequately exercised. The behaviour of the test object before reset, during reset and after reset should be verified. The timing checkers of the test object could be disabled during this type of verification since the correct behaviour of the model after a timing violation is often not required.

The test object should not be verified only for its nominal behaviour but also for robustness. All input combinations should be generated including erroneous usage of the test object interfaces, incorrect accessing schemes etc. The test object should be run in all modes, entering all internal states.

Independent high-level checkers should be implemented for the test suite. These checkers should evaluate the data sent and received by the test object, verifying that protocols are correct, ensure that interface requirements are meet, etc. The incorporation of high-level checkers in the verification is a complement to the comparison with the *component model*, and is essential when such a reference model does not exist.

It has been shown that random generation of input values, the order and type of accesses etc., detects many differences between two model representations and should therefore be included in the test suite when found beneficial. The implementation of the random function should never use the type *Real*, since its realisation is often simulator dependent and could give inconsistent simulation results.

All inputs to the test object should be synchronously asserted at relevant clock edges and with appropriate clock-to-output delays, not to violate any setup and hold constraints. The logical values on the inputs of the test object should be compatible with the simulator used for the *component model*, which is normally ‘*1*’ and ‘*0*’. The use of unknown values on the inputs will most likely not produce the same result for the two models and should be avoided. When multiple input clocks are used, they should be phased locked to avoid unnecessary timing violations during the simulation of the *component model*.

The test suite could generate formatted output files showing status, output data etc., which could be useful to the model developer. For example, a serial output data stream could be converted to a parallel format before being evaluated in the test bench or written to a file for further reviewing of the simulation results. These files should not be generated when the test suite is run by a user, neither should it be necessary for a user to review those files.

The outputs of the test object should be sampled with respect to a driving clock edge and be written to a list file or be compressed using a Multiple Input Signature Register, MISR, as described in section 4.3.3. Care should be taken when deciding the sampling point within the clock cycle, allowing all outputs of the model to settle for all simulation conditions. The test suite should allow the outputs of the *component model* to obtain other values than unknown after reset or start up of the simulation, before beginning the comparison of the results. The format of the list file should allow straightforward comparison with the output generated by the *component model* simulator.

All inputs to the test object, including those from external components, could be sampled at their assertion point and written to a force file. The format of the stimuli in force files should be readable by the simulator used for the *component model*. If that simulator can use the VHDL test bench directly, the file would then not be necessary.

Force, list and output files can be used when verifying the functionality of the model intended for board-level simulation against the *component model*. When the verification has been successful, the list file from the *component model* simulation should be provided to the user as a reference file for comparison with obtained simulation results. When the test suite is run by the user, the outputs should be sampled and compared to a reference file to establish whether each sub-test has passed. The comparison could be done within the test generator to avoid generating an output file for comparison outside the simulator. Since reference files tend to become large and cumbersome, compression of the output data is recommended and is described in section 4.3.3.

When the test suite is to be evaluated using fault simulation as described in section 4.4, the test stimuli could be transferred between the simulators using tool specific methods, otherwise the previously mentioned force file could be used. In case bidirectional ports are used, it could sometimes be necessary to generate force files to obtain the correct waveforms even when tool specific methods are available. When Built In Self Test, BIST, is not fully implemented in the model for board-level simulation, e.g. it is modelled only as a delay before the nominal operation after reset, it should be verified in a separate test suite.

The force, list and output file generation should be possible to disable by a *Boolean* generic with the default value *False*, and it should be disabled when not used for debugging purposes. It is suggested that the implementation of such file generators should not induce any simulation penalties when not used and be disabled using generate statements as shown in example 12.

An architecture containing a functional test suite is shown in example 32. The test suite in the example will test the full functionality of the model, except the BIST operation.

The activation of the BIST would preclude the test suite to be evaluated using fault simulation. The BIST is tested separately in the architecture *X_Handling* as outlined in section 4.3.2. Only the *Std_Logic* strengths *'0'*, *'1'*, *'Z'* are applied to the inputs, to be able to apply the same stimuli to the *component model* as well. The outputs are sampled and the data is compressed in a concurrent procedure. The resulting signature is compared to an expected signature at the end of each sub-test, which is done in the test suite. The test object is reset and the MISR is cleared between each sub-test. The length of the *MISR* adapts itself to the length of the *Input* parameter. The MISR will be sampled on each rising *Clk* edge. The source code of the *MISR* procedure can be found in appendix B.

```
library BitMod_TB_Lib;
use BitMod_TB_Lib.MISR_Definition.all;

architecture Functional of TestGenerator is
   signal   MISRegister: Std_Logic_Vector(0 to 15);
   signal   MISRInput:   Std_Logic_Vector(0 to 15);
   signal   MISRReset:   Boolean;
begin
   TestSuite: process
      variable TestFailed: Boolean := False;
   begin
      ... Initialisation
      ResetMISR(MISRReset);
      ... Test suite
      CheckMISR(MISRegister, "1234", TestFailed,
                InstancePath&"Functional:TestSuite:Sub-test 41");
      ResetMISR(MISRReset);
      Reset(Test, Clk, Reset_N, A, CS_N, RW_N, Sclk, SData);
      ... Test suite
      CheckMISR(MISRegister, "5678", TestFailed,
                InstancePath&"Functional:TestSuite:Sub-test 42");
      assert not TestFailed
         report InstancePath&"Functional:TestSuite: Test failed."
         severity Error;
      assert TestFailed
         report InstancePath&"Functional:TestSuite: Test passed."
         severity Note;
      assert False
         report InstancePath&"Functional:TestSuite: End of test."
         severity Failure;
      wait;
   end process TestSuite;
   MISRInput<= MData & SData & SClk & D & A & CS_N & RW_N & Reset_N;

   MISR(Clk          => Clk,
        Reset        => MISRReset,
        Input        => MISRInput,
        MISR         => MISRegister,
        Rising       => True,
        Falling      => False,
        HeaderMsg    => InstancePath&"Functional:MISR:",
        Sense        => 45 ns);
end Functional;
```

Example 32: *Outline of architecture containing a functional test suite.*

4.3.2 Verification of interfaces

The interfaces of the test object should be verified to function correctly, and should be performed separately from the functional verification. The test results must normally be visually inspected by the user since automatic verification is not always possible.

Test suites resulting in assertion reports should be delivered together with a reference file listing the expected assertion reports, including time of assertion and severity level, which should be used for manual comparison with the results obtained by the user. All message reports that can be generated by the test object should be verified.

Each input of the test object should be applied all nine *Std_Logic* strengths, which should result in the detection of unknown values and assertion reports. Since inputs are checked for unknown values only when needed in models for board-level simulation, the test suite needs to execute the test object to the point when the input value is actually used and affects the simulation. Models with internal memory elements which are observable at the interface should be verified to return the *Std_Logic* strength '*U*' at simulation start up if not reset.

The handling and propagation of unknown values should be verified for each input. The outputs should be sampled and the test suite could use a reference file or output compression, with expected responses first being examined by the model developer. Comparison with a *component model* will normally not be possible since its unknown value propagation is implemented differently.

The architecture *X_Handling* shown in figure 7 contains a test suite that will test the following: all inputs should be applied all nine *Std_Logic* strengths; all checkers for unknown values on inputs; the handling of each unknown input value should be checked; the propagation of each unknown input value should be checked. This test suite should also test the BIST functionality of the model, and should not be evaluated using fault simulation since it would activate portions of the component not modelled for board-level simulation.

The verification of the timing of the test object should include all input-to-output delays, clock-to-output delays, setup and hold times etc. The timing verification should be done for all simulation conditions and be selected by the *SimCondition* generic. Outputs from the test object do not need to be compared with the *component model* during the verification of the timing, since they would not necessarily give identical results. The purpose is to verify the operation of the timing checkers, not the functionality of the model after a timing violation. No reference file nor output compression is therefore needed for this type of verification.

The test suite should not use the timing package provided with the test object to avoid possible error masking. It is therefore not necessary that the test suite should be able to verify the timing of a test object for which the timing package has been modified by the user. This approach will also provide a means for detecting whether a timing package has been modified since delivered.

Each timing constraint checker in the model should be verified by asserting the tested signal from two units before to two units after the critical point, implementing a sliding window over the timing constraint value. The unit should be *ns* for absolute parameters and periods for clock relative parameters. The implementation of the sliding window should take into account that the timing parameters could change during the life time of the test object. Each checker in the model should also be checked under conditions which should not result in an assertion report.

The output delay timing for each signal, including bidirectional signals, should be verified, observing all possible signal transitions and timing parameters.

The architecture *Timing* shown in figure 7 should contain a test suite that will test all timing constraint checkers (both with and without timing violations). The architecture with the timing test suite should be bound for the three simulation conditions by using the three configuration declarations *WorstCaseTest*, *TypCaseTest* and *BestCaseTest*. A listing of simulator outputs (assertion reports) should be provided in the reference files named *worstcase.ref*, *typcase.ref* and *bestcase.ref*.

4.3.3 Verification result compression

A method useful for verifying that the exact behaviour of a model has not changed is to generate a signature, similar to implementing BIST for ASICs. One usage of this technique is to prove the exact behaviour of a delivered VHDL model to the user.

Data from the sampled outputs of the test object could be compressed using a Multiple Input-Signature Register, MISR, which is compared to a predetermined signature to determine whether a test has passed or failed. The benefit of compressing output data is the elimination of large reference files. This method is also useful when combined with high level checkers, since it can provide accurate clock cycle observation of signals when the stimuli and responses have been approved. A MISR can be used for regression tests during model and ASIC developments, as well as for verification test benches. A MISR implemented as a VHDL procedure is shown in appendix B.

Data compression is most feasible for data that have been synchronously sampled, and should therefore mainly be done for the test suites verifying the functional behaviour on a per clock cycle basis. A MISR can also be clocked by the changes on its input vector, it should then be independent of any delta cycles since these can differ between VHDL simulators. A way to implement this is to allow all events on the input vector to take place in each simulation cycle before shifting the MISR. The input vector should therefore be stored for each event, and the MISR should be shifted only when there has been an increase of the simulation time since the last event.

All outputs from the test object should be input to the MISR, including those interfacing external components such as memories. Inputs to the test object could also be input to the MISR, especially when there is a need to assure that input stimuli have not changed. The MISR should have sufficient number of register stages to allow for long test suites without the risk of error detection masking due to register contents repeating themselves. It should therefore be structured as a primitive binary polynomial implementing a maximum length Linear Feedback Shift Register, LFSR, as shown in figure 8. A list of binary polynomials for different register lengths can be found in RD5.

The MISR should be able to detect any differences on its inputs, taking into account all nine *Std_Logic* strengths. For an LFSR to function correctly, the *xor* operator is assumed to operate on operands with only the logic strengths '*0*' and '*1*'. LFSRs are therefore not capable to work with the type *Std_Logic* directly. The nine *Std_Logic* strengths should therefore be transformed to a binary vector representation with the length 4, assigning a unique binary combination for each *Std_Logic* strength. The resulting vector could then be input to the MISR which would be four times longer than the input vector. A way to reduce the vector length is to divide the input vector in four and shift the MISR four times for each sampling point. The MISR would then have the same length as the input vector.

All signals observed by the MISR should be input each clock cycle, be that by applying the full input vector once or to split it and clock the MISR multiple times as explained above. It is recommended that MISRs containing only *xor* operators are set to all ones when reset, to reduce the risk of error detection masking for signatures with all bits zero.

As for the generation of list files, the sampling point within the clock cycle should allow the outputs to settle for the test object under all simulation conditions. The MISR should therefore be clocked with the same, or derived, clock as the test object, but after an appropriate delay. The usage of multiple MISRs should be considered when the test object has more than one input clock.

The MISR signature should be checked and the MISR reset between each sub-test. Both the board-level and detailed models should be in a known state and each sampled input or output should always have the same value between two tests. This will enable the sub-tests to be modified, reordered, or removed without affecting adjacent sub-tests, since the MISR signature of each sub-test would be independent of the preceding simulation results.

Each test suite should be developed with data compression in mind, even when using reference files since these often tend to grow during the development of the test suite and will probably grow too large to be manageable and be eventually replaced by output compression.

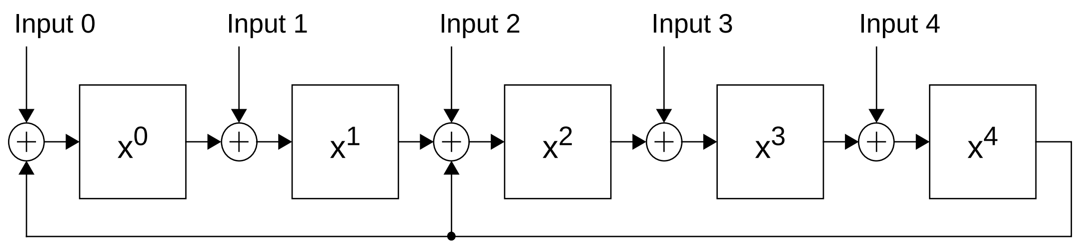

Figure 8: *MISR implementing the polynomial $x^5 + x^2 + 1$*

4.4 Evaluation of verification coverage

The efficiency of the test suites can be assessed and evaluated using quantitative measurement methods such as code and fault coverage. The purpose of the verification should always be to verify the complete functionality of the model for board-level simulation, not to solely satisfy the code and fault coverage goals since these are merely measurements of the verification efficiently. The calculation of the code coverage should be done in accordance with RD1. The code and fault coverage of each test suite should be documented.

The functional coverage of the test suites verifying the functionality of the model can be approximated as the fault coverage achieved when applying the same stimuli to a fault simulation of the *component model*. This type of coverage measurement is of course only feasible when a *component model* exists. Only the fault coverage for the functional core of the *component model* need to be considered, excluding test structures and logic used for production or self-test when not fully implemented in the test object.

Functions used for internal testability, e.g. BIST, should not be activated by the test suites during the fault simulation when not fully implemented in the test object. The testability logic could else cover faults in parts of the *component model* which are not verified by the stimuli when applied to the model for board-level simulation. This would result in a fault coverage not being proportional to the functional coverage.

Normally only stimuli stemming from clock cycle oriented test suites can readily be used as input to fault simulators.

5 MODELLING AND SIMULATING BOARD DESIGNS

Board-level simulations are performed to verify board designs. The verification strategy should be analysed before the simulation commences, assessing its fulfilment by the intended simulation. To be able to make the assessment, an understanding of the properties and capabilities of the simulation models is necessary. The simulation models define the types of design characteristics that can be simulated. The level of correctness in terms of functionality and timing should be assessed for each model, otherwise potential errors will be found on the real breadboard after manufacturing. The quality of simulation results is rarely higher than the quality of the models used in the simulation. Models not meeting the requirements imposed by the purpose of the simulation have to be modified or replaced else the desired result will not be achieved.

Board designs can be simulated in full or only partially. When a full board design is simulated it is recommended that the interfaces of the board are modelled sufficiently accurately to allow the board design to be used in further simulations without the necessity for the user to have excessive knowledge about the board’s structure. Board designs can be described in VHDL or in a description suiting mixed-level simulation when required. Board designs described exclusively in VHDL have all the benefits as earlier described for models for board-level simulation, allowing portability and usage with different simulators. The rest of this section will outline two approaches to modelling and verification of full board designs described in VHDL.

5.1 Board designs in VHDL

The interfaces of a board design should follow the same requirements as previously suggested for models for board-level simulation. All timing parameters, such as clock-to-output and propagation delays, which are affected by the interconnection between the board design and other equipment, should have a corresponding timing generic with a default value declared as a deferred constant in a separate timing package. All signals interfacing the board design should be declared as ports.

The board design should have the same control generics as models for board-level simulation, such as *SimCondition* and *InstancePath*, with appropriate default values, as shown in example 33. Each component instantiated on the board should have a label matching the component number on the board or other unique name. The *InstancePath* generics of the component instantiations should be associated with the *InstancePath* generic of the board design entity and the instantiation label name. The generics *SimCondition*, *TimingChecksOn*, etc. should be passed from the entity to the components.

There are two approaches to how components in a board design can be configured: in separate configuration declarations or using configuration specifications in the architecture. The first approach keeps the timing and interconnection separated, placing the interconnection in the architecture and the timing in the configuration declaration, allowing for greater flexibility if the user of the board wants to modify timing parameters internal to the board. The second approach allows the user to contain all the information about timing and interconnection in one place. Both approaches allow the user to modify timing parameters that are declared as generics in the board design entity.

```
library ESA;
use ESA.Simulation.all;
use ESA.Timing.all;

library IEEE;
use IEEE.Std_Logic_1164.all;

library BoardDesign_Lib;
use BoardDesign_Lib.BoardDesign_Timing.all;

entity BoardDesign is
   generic(
      SimCondition:   SimConditionType := WorstCase;
      InstancePath:   String           := "BoardDesign:";
      TimingChecksOn: Boolean          := False;
      tpd_Clk_MData:  TimeArray01      := tpd_Clk_MData);
   port(
      Test:    in    Std_Logic_Vector(0 to 1);-- Board Test mode
      Clk:     in    Std_Logic;               -- Board Master Clock
      Reset_N: in    Std_Logic;               -- Board Master Reset
      A:       in    Std_Logic_Vector(0 to 1);-- Board Address bus
      D:       inout Std_Logic_Vector(0 to 7);-- Board Bidirectional
      RW_N:    in    Std_Logic;               -- Board Read/write
      CS0_N:   in    Std_Logic;               -- Chip select, IC0
      CS1_N:   in    Std_Logic;               -- Chip select, IC1
      CS2_N:   in    Std_Logic;               -- Chip select, IC2
      SClk:    in    Std_ULogic;              -- Serial Clock
      DataIn:  in    Std_ULogic;              -- Serial input data
      DataOut: out   Std_Logic);              -- Serial output data
end BoardDesign;
```

Example 33: *Outline of an entity declaration for a board design.*

The architecture of the first approach contains component declarations without any generics, since the generics of the board-level entities will be associated with values in the configuration declaration. The architecture statement part contains only component instantiations representing the connectivity of the board, as shown in example 34 and figure 9. The port maps should be declared using named association. This approach does not need to make any libraries or packages visible to the architecture. The components on the board are defined by the component declarations and are bound to entities in the separate configuration declaration, allowing the user of the board to select entities independently of the board design model.

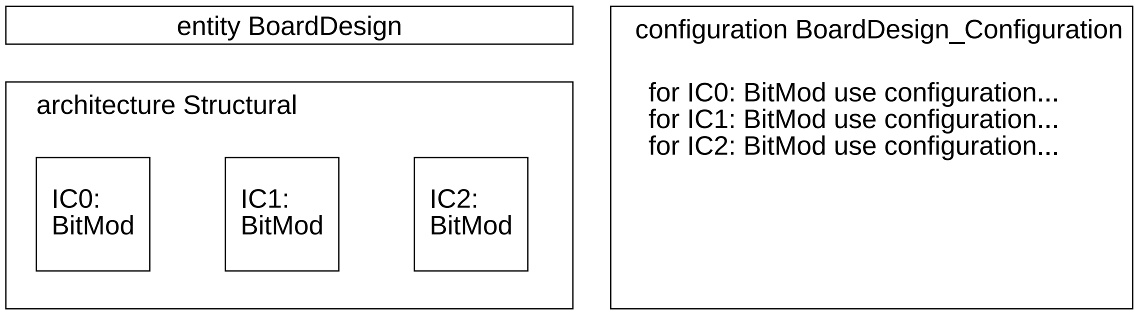
entity BoardDesign
architecture Structural
IC0: BitMod
IC1: BitMod
IC2: BitMod
configuration BoardDesign_Configuration
for IC0: BitMod use configuration...
for IC1: BitMod use configuration...
for IC2: BitMod use configuration...

Figure 9: *Board design architecture and configuration declaration.*

```
architecture Structural of BoardDesign is
   component BitMod
      port(...);
   end component;
begin
   IC0: BitMod
      port map(...);
   IC1: BitMod
      port map(...);
   IC2: BitMod
      port map(...);
end Structural;
```

Example 34: *Outline of board design architecture without configuration specifications.*

The configuration declaration of the board design above is shown in example 35. The *SimCondition* and *TimingChecksOn* generics in the board design entity are propagated down the hierarchy. Only the timing packages themselves are made visible to the models, not their contents. In the general case it is not possible to make all the contents of the timing package visible to the configuration declaration, since when there are more than one component in the design, each having its own timing package, it could result in naming conflicts for some of the timing parameters. Therefore is the default timing parameter *tpd_Clk_MData* referenced using complete named selection when used in the generic map of instance *IC0*. If timing parameter values are not needed, as for instance *IC2* which is using the default generic value of the entity, the timing package need not be made visible. A new configuration declaration could be derived when other timing values are needed, e.g. the output delays of the components can be annotated with signal path delays or delays due to capacitive loads on the circuit board.

The architecture of the second approach is shown in example 36 and figure 10. The generic declarations of the component need only to include those generics which will be associated in the architecture. The rest will take default values declared for the corresponding entity. Each timing generic declaration need to have the same default values as have been declared for the entity, due to language rules. In this example the instantiation *IC2* will use the default value for the timing parameter *tpd_Clk_MData*. The default value is fetched from the timing package with named selection not to conflict with the timing parameter names of other models used in the architecture (not shown here).

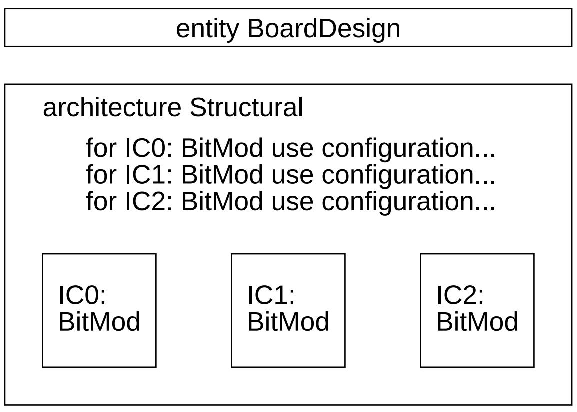
entity BoardDesign
architecture Structural
for IC0: BitMod use configuration...
for IC1: BitMod use configuration...
for IC2: BitMod use configuration...
IC0: BitMod
IC1: BitMod
IC2: BitMod

Figure 10: *Board design architecture with configuration specifications.*

All instances of component *BitMod* are bound in the configuration specification to the configuration declaration *BitMod_Configuration* in library *BitMod_Lib*.

```
library IEEE;
use IEEE.Vital_Timing.all;

library BitMod_Lib;

configuration BoardDesign_Configuration of BoardDesign is
   for Structural
      for IC0: BitMod                          -- Annotated timing
         use configuration BitMod_Lib.BitMod_Configuration
            generic map(
               SimCondition   => SimCondition,
               InstancePath   => InstancePath&"IC0:",
               TimingChecksOn => TimingChecksOn,
               tpd_Clk_MData  =>
               ((BitMod_Lib.BitMod_Timing.tpd_Clk_MData(WorstCase)
                    (tr01)+ 5 ns,
                BitMod_Lib.BitMod_Timing.tpd_Clk_MData(WorstCase)
                    (tr10)+ 5 ns),
               (BitMod_Lib.BitMod_Timing.tpd_Clk_MData(TypCase)
                    (tr01)+ 5 ns,
                BitMod_Lib.BitMod_Timing.tpd_Clk_MData(TypCase)
                    (tr10)+ 5 ns),
               (BitMod_Lib.BitMod_Timing.tpd_Clk_MData(BestCase)
                    (tr01)+ 5 ns,
                BitMod_Lib.BitMod_Timing.tpd_Clk_MData(BestCase)
                    (tr10)+ 5 ns)));
      end for;

      for IC1: BitMod                           -- Absolute timing
         use configuration BitMod_Lib.BitMod_Configuration
            generic map(
               SimCondition   => SimCondition,
               InstancePath   => InstancePath&"IC1:",
               TimingChecksOn => TimingChecksOn,
               tpd_Clk_MData  => ((5 ns,5 ns),
                                  (5 ns,5 ns),(5 ns,5 ns)));
      end for;

      for IC2: BitMod                           -- Unmodified timing
         use configuration BitMod_Lib.BitMod_Configuration
            generic map(
               SimCondition   => SimCondition,
               InstancePath   => InstancePath&"IC2:",
               TimingChecksOn => TimingChecksOn);
      end for;
   end for;
end BoardDesign_Configuration;
```

Example 35: *Outline of configuration declaration for a board design.*

```
library BitMod_Lib;

library IEEE;
use IEEE.Vital_Timing.all;

architecture Structural of BoardDesign is
   component BitMod
      generic(
         SimCondition:   SimConditionType := WorstCase;
         InstancePath:   String           := "BitMod:";
         TimingChecksOn: Boolean          := False;
         tpd_Clk_MData:  TimeArray01      :=
                            BitMod_Lib.BitMod_Timing.tpd_Clk_MData);
      port(...);
   end component;
   ...
   for all: BitMod
      use configuration BitMod_Lib.BitMod_Configuration;
begin
   IC0: BitMod                                 -- Annotated timing
      generic map(
         SimCondition   => SimCondition,
         InstancePath   => InstancePath&"IC0:",
         TimingChecksOn => TimingChecksOn,
         tpd_Clk_MData  =>
            ((BitMod_Lib.BitMod_Timing.tpd_Clk_MData(WorstCase)
                (tr01) + 5 ns,
              BitMod_Lib.BitMod_Timing.tpd_Clk_MData(WorstCase)
                (tr10)+ 5 ns),
             (BitMod_Lib.BitMod_Timing.tpd_Clk_MData(TypCase)
                (tr01)+ 5 ns,
              BitMod_Lib.BitMod_Timing.tpd_Clk_MData(TypCase)
                (tr10)+ 5 ns),
             (BitMod_Lib.BitMod_Timing.tpd_Clk_MData(BestCase)
                (tr01)+ 5 ns,
              BitMod_Lib.BitMod_Timing.tpd_Clk_MData(BestCase)
                (tr10)+ 5 ns)))
      port map(...);
   IC1: BitMod                                 -- Absolute timing
      generic map(
         SimCondition   => SimCondition,
         InstancePath   => InstancePath&"IC1:",
         TimingChecksOn => TimingChecksOn,
         tpd_Clk_MData  => ((5 ns,5 ns), (5 ns,5 ns), (5 ns,5 ns)))
      port map(...);
   IC2: BitMod                                 -- Unmodified timing
      generic map(
         SimCondition   => SimCondition,
         InstancePath   => InstancePath&"IC2:",
         TimingChecksOn => TimingChecksOn)
      port map(...);
end Structural;
```

Example 36: *Outline of board design architecture with configuration specifications.*

5.2 Verification of board designs

A test bench should be developed for the board design analogous to the test bench for a model for board-level simulation, as shown in figure 11. A test generator with one or multiple architectures with test suites should be instantiated together with the board design in the test bench architecture. Selection of simulation conditions and test suites should be done by using configuration declarations with the same names as have been defined for the verification of models for board-level simulation*: WorstCaseTest, TypCaseTest* and *BestCaseTest*.

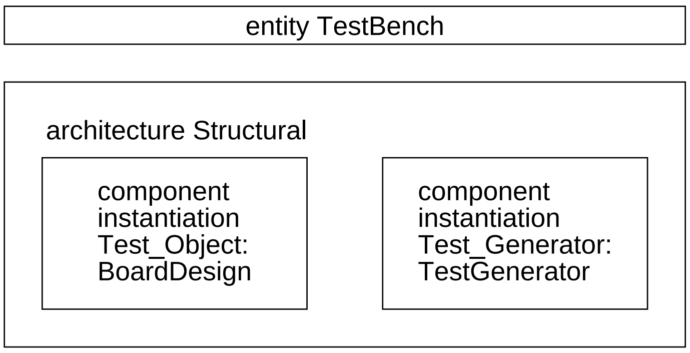
entity TestBench
architecture Structural
component instantiation Test_Object: BoardDesign
component instantiation Test_Generator: TestGenerator

Figure 11: *Test bench containing the BoardDesign and the TestGenerator.*

An outline of the test bench is shown in example 37. Only the ports are declared in the component declarations, since the generics will be associated with their values in each of the configuration declarations used. The only package needed is *Std_Logic_1164* since the architecture is purely structural only containing the connections between the object being tested and the test generator. The port maps should be declared using named association as shown for the port *Test* and the local signal *Test*.

```
entity TestBench is
end TestBench;

library IEEE;
use IEEE.Std_Logic_1164.all;

architecture Structural of TestBench is
   component BoardDesign
      port(...);
   end component;
   component TestGenerator
      port(...);
   end component;
   -- Local signal declarations.
begin
   Test_Object: BoardDesign
      port map(Test => Test, ...);
   Test_Generator: TestGenerator
      port map(Test => Test, ...);
end Structural;
```

Example 37: *Outline of entity and architecture of a test bench for a board design.*

The package *ESA.Simulation* is made visible to the configuration declaration in example 38 to allow the selection of the simulation condition. The libraries *BoardDesign_Lib* and *BoardDesign_TB_Lib* are made visible to allow the selection of the test object and the test generator.

The timing parameter values for the board design are selected by using the *BoardDesign_Configuration* in the configuration specification for the *Test_Object*.

The *InstancePath* generics of the *Test_Object* and the *Test_Generator* have been chosen to contain the same *String* value as would have been returned by the VHDL ’93 attribute *Path_Name*. Note that the root of the path begins with a colon.

```
library ESA;
use ESA.Simulation.all;

library BoardDesign_Lib;

library BoardDesign_TB_Lib;

configuration WorstCaseTest of TestBench is
   for Structural
      for Test_Object: BoardDesign
         use configuration BoardDesign_Lib.BoardDesign_Configuration
            generic map(
               SimCondition   => WorstCase,
               InstancePath   => ":TestBench:Test_Object:",
               TimingChecksOn => True);
      end for;
      for Test_Generator: TestGenerator
         use entity BoardDesign_TB_Lib.TestGenerator(Timing)
            generic map(
               SimCondition   => WorstCase,
               InstancePath   => ":TestBench:Test_Generator:");
      end for;
   end for;
end WorstCaseTest;
```

Example 38: *Outline of a configuration declaration of a test bench for a board design.*

A more detailed approach to verification of board designs can be found in RD7.

6 DESIGN DOCUMENTATION

All documentation should be in English. The documentation should be well structured and easily readable. The documentation should be consistent, e.g. the same item should have the same name in all documentation and code. Diagrams should be introduced where beneficial for the understanding of the text.

Every time a document is updated it should include a detailed change list, and all significant changes marked using a change bar in the margin.

6.1 User's Manual

The purpose of the User's Manual should be to allow any board-level designer, with little or no VHDL experience, to efficiently use the developed models to perform board-level simulation without needing the full source code. The manual shall be independent from any VHDL simulator specific features.

The text should be oriented towards the user and be spell checked. Naming and numbering conventions used for the model and documentation should be documented.

The component which is modelled should be unambiguously identified, including the component name and number, and foundry. The sources describing the functionality and timing from which the modelling is performed should be identified. To avoid potential documentation errors, information from already established and publicly available documents, such as Data Sheets, can be referenced using a complete reference (document title, reference number, issue number and date, section).

Any limitation introduced during the model development should be documented, including assumptions or restrictions regarding the usage of the model, unresolved coding errors (and workarounds), and non-compliances w.r.t. the component functionality.

The model should be identified including library and configuration declaration names.

All interfaces of the model should be described, including but not restricted to:
- Ports (purpose and signal polarity);
- Generics (purpose and default values, limitations on negative values);
- Input and output files (data formats and default names);
- Timing package (maximum loading for each timing parameter, description of how the values for all three simulation conditions have been obtained).

Management of unknown input values should be documented, including X-checking, X-handling and X-propagation. The conditions under which the X-checkers are enabled or disabled should be identified.

The timing constraints which are checked by the model should be identified, including any limitations or assumptions. Any timing constraints not checked should be identified. The output delays on the ports should be documented for each operating mode of the model, including X-generation when applicable.

The provided test bench should be identified, including library and configuration declaration names. The level of verification achieved by the test bench should be documented, including code coverage figures and the capabilities of the self-checking mechanism.

The level of verification should be documented, including whether comparison has been made versus the *component model*, whether functional or production test vectors have been reused, etc. The version number of the simulators, platforms and operating systems on which the model has been verified should be documented.

The document should include all necessary steps for installing the model and its test bench, performing a verification and analysing the obtained results to determine whether the model simulates correctly in the installed environment. If any file conversion tools are needed, their usage should be documented.

The organisation of the delivered files should be fully described, including source files, script files, reference files, and input and output files. The version of each module to which the User's Manual corresponds to should be identified.

When a distribution channel for the model has been established, a point of reference for model support and maintenance should be identified, including name, address, email address, phone and fax numbers.

A suggested outline of the User's Manual can be found in appendix A. However, it should be noted that it is not necessarily complete. The User's Manual should be delivered both as an unbound paper copy and an electronic copy in ESA developments.

6.2 Supplement

A supplement, including all information necessary for the maintenance of the model and its verification, should be attached as an appendix to the User's Manual when requested.

The complete hierarchy and structure of the model, the test bench and test suites, should be described, taking into account all dependencies, such as the usage of packages. An accurate block diagram showing the relationship between different modules, their input and output signals etc. should be created. Information readily found in the source code should not be repeated.

The verification of the model should be documented in detail. When the functional and production test vectors from the *Detailed Design* of the component have been used during the verification, it should be documented. The comparison between the board-level and the *component model* simulation results should be documented. A compliance matrix should be included showing the correspondence between the model intended for board-level simulation and the component, any discrepancies should be documented. Any source code lines not possible to cover should be included together with an explanation. When simulation performance measurements have been performed, they should be documented.

APPENDIX A: OUTLINE OF USER'S MANUAL

Table of Contents

Introduction
- Reference data and documentation
- Conventions and abbreviations

Limitations

Description of the model
- Component and library names
- Interfaces
- Management of unknown input values
- Timing constraints
- Output delays

Description of the test bench
- Test bench and library names
- Self-checking capability
- Calculated code coverage
- Simulators, platforms and operating systems

Usage of the model and test bench
- Installation procedure
- Verification procedure

File organisation
- Source files and analysis order
- Reference/input/output files
- Script/make files

Supplement {Upon request, as an appendix}
- Detailed description of the model
  - Hierarchy/Structure {Including diagrams}
- Detailed description of the test bench
  - Hierarchy/Structure {Including diagrams}
- Verification
  - Comparison versus gate-level model
  - Compliance matrix
  - Not covered code lines
- Simulation performance

APPENDIX B: MULTIPLE-INPUT SIGNATURE REGISTER

```
--============================================================================--
-- Multiple-Input Signature Register (MISR)
-- This procedure implements a variable length MISR. The length is determined by
-- the length of Input, ranging from 4 to 100. The Input can be sampled on
-- either Clk edge or both, delayed by Sense, selected with the Rising and
-- Falling parameters. If neither option is selected, events on Input will
-- determine the sampling point. Events happening in the same simulation cycle,
-- differing only in delta cycles, will be sampled when the last event has
-- occurred, and the MISR will then be shifted.
--
-- The Reset input will reset the MISR to all-ones. When sampling is made with
-- Clk, it will re-start on the next relevant edge after the asserting edge of
-- Reset. If Reset is detected between an Clk edge and the sampling point, the
-- MISR will be reset and the sample will be ignored. When asynchronous sampling
-- is used, the next event on Input will be the first sample after reset. The
-- MISR signal can be read at any point and should be compared with a
-- predetermined signature.
--
-- The MISR is implemented as a primitive polynomial with up to five terms. The
-- terms are taken from the text book Built-In Test for VLSI: Pseudorandom
-- Techniques, by Bardell et al. The elements in the Input vector are expanded
-- to four bits, each Std_Logic value having a unique bit pattern,
-- the intermediate vector is then divided in four and each part is shifted into
-- the MISR separately. The procedure can be used as a concurrent subprogram,
-- not needing any surrounding process or block.
--
-- Inputs:  Clk,   sample clock used with Rising and Falling
--          Reset, reset of MISR when an event is detected and Reset is True
--          Input, input vector to the MISR, same length as MISR
--          Rising/Falling:
--          False  False  sample at each Input event
--          False  True   sample Input after Sense on falling Clk edge
--          True   False  sample Input after Sense on rising Clk edge
--          True   True   sample Input after Sense on rising or falling Clk edge
--          Sense, positive time after the Clk edge when Input is sampled
--          HeaderMsg, message header
-- In/Outs: MISR, Multiple-Input Signature Register
--
-- Author:  Sandi Habinc, ESTEC Microelectronics and Technology Section (WSM)
--          P.O. Box 299, 2200 AG Noordwijk, The Netherlands
--------------------------------------------------------------------------------
library IEEE;
use IEEE.Std_Logic_1164.all;

package MISR_Definition is
   procedure MISR(
      signal   Clk:       in    Std_ULogic;           -- Sample clock
      signal   Reset:     in    Boolean;              -- Reset of MISR
      signal   Input:     in    Std_Logic_Vector;     -- Input vector
      signal   MISR:      inout Std_Logic_Vector;     -- MISR
      constant Rising:    in    Boolean := True;      -- See above
      constant Falling:   in    Boolean := False;     -- See above
      constant HeaderMsg: in    String  := "MISR:";   -- Message header
      constant Sense:     in    Time    := 0 ns);     -- Sense time after clock
end MISR_Definition; --=============== End of package header =================--
```

```
package body MISR_Definition is

   --------------------------------------------------------------------------------
   -- Local declarations of minimum and maximum MISR lengths.
   --------------------------------------------------------------------------------
   constant MaxLen:   Integer := 100;
   constant MinLen:   Integer := 4;

   --------------------------------------------------------------------------------
   -- Local declarations defining subtypes and types needed for the definition
   -- of the Std_Logic to 4bit vector transfer function. Each Std_Logic value
   -- has a unique 4bit vector associated.
   --------------------------------------------------------------------------------
   subtype  Index  is Integer range 0 to 3;           -- Definition of table
   subtype  Vector is Std_Logic_Vector(Index);        -- with a 4bit vector for
   type     VTable is array (Std_Logic) of Vector;    -- each Std_Logic value
   constant Std4:     VTable := ('U' => "0001", 'X' => "0010", '0' => "0100",
                                 '1' => "1000", 'Z' => "0011", 'W' => "0110",
                                 'L' => "1100", 'H' => "0111", '-' => "1110");

   --------------------------------------------------------------------------------
   -- Expands every Std_Logic_Vector element to four bits returning a
   -- Std_Logic_Vector with four times the length of the input.
   -- Input:  V Std_Logic_Vector defined (0 to n)
   -- Output:   Std_Logic_Vector defined (0 to n) with same length as V
   --------------------------------------------------------------------------------
   function To01(V:         Std_Logic_Vector;
                 HeaderMsg: String := "MISR:") return Std_Logic_Vector is
      variable R: Std_Logic_Vector(0 to (V'Length*4-1));
   begin
      assert (V'Length<=MaxLen) and (V'Length>=MinLen) -- Check length
         report HeaderMsg&" Only vectors of Length 4 to 100 are supported."
         severity Failure;

      for i in V'Range loop
         R(i*4+0 to i*4+3):= Std4(V(i));               -- Expand input
      end loop;
      return R;
   end To01;

   --------------------------------------------------------------------------------
   -- Logical and operation between Std_ULogic scalar and Std_Logic_Vector.
   -- Input:  B Std_ULogic scalar, V Std_Logic_Vector vector
   -- Output: Std_Logic_Vector with same array constraints as V
   --------------------------------------------------------------------------------
   function "and" (B: Std_ULogic;
                   V: Std_Logic_Vector) return Std_Logic_Vector is
      variable R: Std_Logic_Vector(V'Range);
   begin
      for i in V'Range loop
         R(i):= B and V(i);                            -- logical and
      end loop;
      return R;                                        -- return vector
   end "and";
```

```
--------------------------------------------------------------------------
-- Expands the Input four times using To01, the resulting vector is then
-- shifted into the MISR in four parts. The resulting MISR value is returned.
--------------------------------------------------------------------------
function Shift(MISR:  Std_Logic_Vector;
               Input: Std_Logic_Vector;
               Poly:  Std_Logic_Vector) return Std_Logic_Vector is
   variable M: Std_Logic_Vector(MISR'Range)          := MISR;
   variable T: Std_Logic_Vector(0 to 4*MISR'Length-1) := To01(Input);
begin
   for i in 0 to 3 loop                              -- Four MISR shifts
      M := ('0'&M(0 to MISR'Length-2))               xor
           T(MISR'Length*i to MISR'Length*(i+1)-1) xor
           (M(MISR'Length-1) and Poly);              -- Scalar and vector
   end loop;
   return M;                                         -- Return resulting MISR
end Shift;

--------------------------------------------------------------------------
-- Returns a Std_Logic_Vector of length L (1 to 100) containing a primitive
-- polynomial with up to five terms (always including x+1).
-- Input:  L, the length of the resulting polynomial Std_Logic_Vector
-- Output: polynomial Std_Logic_Vector defined (0 to n)
--------------------------------------------------------------------------
function Polynomial(L:         Natural;
                    HeaderMsg: String:="MISR:") return Std_Logic_Vector is
   variable P:        Std_Logic_Vector(0 to L-1) := (others => '0');
   subtype  Degree is Integer range 0 to MaxLen;    -- Exponent range
   type     Terms  is array (1 to 3) of Degree;     -- Triplet
   type     TTable is array (1 to MaxLen) of Terms; -- Triplets
   constant Table:    TTable := (                   -- Look-up table
      (0,0,0),   (1,0,0),   (1,0,0),   (1,0,0),   (2,0,0),   (1,0,0),   --  1
      (1,0,0),   (6,5,1),   (4,0,0),   (3,0,0),   (2,0,0),   (7,4,3),   --  7
      (4,3,1),   (12,11,1), (1,0,0),   (5,3,2),   (3,0,0),   (7,0,0),   -- 13
      (6,5,1),   (3,0,0),   (2,0,0),   (1,0,0),   (5,0,0),   (4,3,1),   -- 19
      (3,0,0),   (8,7,1),   (8,7,1),   (3,0,0),   (2,0,0),   (16,15,1), -- 25
      (3,0,0),   (28,27,1), (13,0,0),  (15,14,1), (2,0,0),   (11,0,0),  -- 31
      (12,10,2), (6,5,1),   (4,0,0),   (21,19,2), (3,0,0),   (23,22,1), -- 37
      (6,5,1),   (27,26,1), (4,3,1),   (21,20,1), (5,0,0),   (28,27,1), -- 43
      (9,0,0),   (27,26,1), (16,15,1), (3,0,0),   (16,15,1), (37,36,1), -- 49
      (24,0,0),  (22,21,1), (7,0,0),   (19,0,0),  (22,21,1), (1,0,0),   -- 55
      (16,15,0), (57,56,1), (1,0,0),   (4,3,1),   (18,0,0),  (10,9,1),  -- 61
      (10,9,1),  (9,0,0),   (29,27,2), (16,15,1), (6,0,0),   (53,47,6), -- 67
      (25,0,0),  (16,15,1), (11,10,1), (36,35,1), (31,30,1), (20,19,1), -- 73
      (9,0,0),   (38,37,1), (4,0,0),   (38,35,3), (46,45,1), (13,0,0),  -- 79
      (28,27,1), (13,12,1), (13,0,0),  (72,71,1), (38,0,0),  (19,18,1), -- 85
      (84,83,1), (13,12,1), (2,0,0),   (12,0,0),  (11,0,0),  (49,47,2), -- 91
      (6,0,0),   (11,0,0),  (47,45,2), (37,0,0));                       -- 97
begin
   assert (L <= MaxLen) and (L > 0)                 -- Check length
      report HeaderMsg&" Only polynomial degree of 1 to 100 is supported."
      severity Failure;
   for i in 1 to 3 loop
      P(Table(L)(i)) := '1';                        -- Insert terms
   end loop;
   return '1'&P(1 to L-1);                          -- Return polynomial
end Polynomial;
```

```
   -----------------------------------------------------------------------------
   -- See definition in the package header.
   -----------------------------------------------------------------------------
   procedure MISR(
      signal   Clk:       in    Std_ULogic;           -- Sample clock
      signal   Reset:     in    Boolean;              -- Reset of MISR
      signal   Input:     in    Std_Logic_Vector;     -- Input vector
      signal   MISR:      inout Std_Logic_Vector;     -- MISR
      constant Rising:    in    Boolean := True;      -- See above
      constant Falling:   in    Boolean := False;     -- See above
      constant HeaderMsg: in    String  := "MISR:";   -- Message header
      constant Sense:     in    Time    := 0 ns) is   -- Sense time after clock

      constant L:    Integer                    := Input'Length;
      constant Poly: Std_Logic_Vector           := Polynomial(L);
      constant Ones: Std_Logic_Vector(0 to L-1) := (others => '1');
      variable Temp: Std_Logic_Vector(0 to L-1) := Input; -- Last Input
      variable Last: Time                       := 0 ns;  -- Last sim cycle
   begin
      assert (L <= MaxLen) and (L >= MinLen)          -- Check length
         report HeaderMsg&" Only MISRs of length 4 to 100 are supported."
         severity Failure;
      assert L = MISR'Length                          -- MISR=Input length
         report HeaderMsg&" The length of the Input and the MISR differs."
         severity Failure;

      while True loop                                 -- Loop never exited
         if not Rising and not Falling then
            -- Clk input has no influence on the MISR in asynchronous mode.
            -- Sampling only the final input vector in each simulation cycle.
            wait on Input, Reset;                     -- Asynchronous sample
            if not (Reset'Event and Reset) and        -- Check not reset
               Input'Event then                       -- Check for Input event
               if Now = Last then                     -- New delta cycle
                  Temp := Input;                      -- Store until next event
               else                                   -- New sim cycle
                  MISR <= Shift(MISR, Temp, Poly);    -- Shift MISR 4 times
                  Temp := Input;                      -- Store until next event
                  Last := Now;                        -- Store sim time
               end if;
            end if;
         else
            wait on Clk, Reset;                       -- Synchronous sample
            if not (Reset'Event and Reset) and        -- Check not reset
               ((Rising and Rising_Edge(Clk)) or      -- Check rising edge
                (Falling and Falling_Edge(Clk))) then -- Check falling edge
               wait on Reset until Reset for Sense;   -- Delay until sample
               MISR <= Shift(MISR, Input, Poly);      -- Shift MISR 4 times
            end if;
         end if;
         if Reset'Event and Reset then
            MISR <= Ones;                             -- Reset MISR
            Temp := Ones;                             -- Reset Temp
         end if;
      end loop;
   end MISR;
end MISR_Definition;  --================ End of package body =================--
```

APPENDIX C: TIMING PARAMETER TYPES

```
--============================================================================--
-- Design unit  : Timing (Package declaration)
--
-- File name    : timing.vhd
--
-- Purpose      : This package defines three array types, indexed by the ESA
--                SimConditionType, needed for timing generics when using
--                Vital Delay Types. The types are intended to be used
--                in VHDL models for board-level simulation. This package
--                should not be modified or moved to a different library.
--
-- Note:          The type TimeArray has been defined in ESA.Simulation.
--
--                This package does not define any types related to the
--                Vital Delay Array Types, since it is not possible to
--                define a constrained array of unconstrained arrays. Such
--                declarations should be done in the timing package of the
--                component.
--
-- Errors:      : None known
--
-- Library      : ESA
--
-- Dependencies : ESA.Simulation, IEEE.Vital_Timing.
--
-- Author       : Sandi Habinc, Peter Sinander
--                ESTEC Microelectronics and Technology Section (WSM)
--                P.O. Box 299
--                2200 AG Noordwijk
--                The Netherlands
--
-- Simulator    : Synopsys v. 3.2c, on Sun Sparcstation 10, SunOS 4.1.3
--------------------------------------------------------------------------------
-- Revision list
-- Version Author Date       Changes
--
-- 1.0     SH      1 July 95 New version
--------------------------------------------------------------------------------
library ESA;
use ESA.Simulation.all;

library IEEE;
use IEEE.Vital_Timing.all;

package Timing is

   -- Definition of Time array types, which can be used for the timing
   -- parameters with Vital Delay Types.
   type TimeArray01   is array (SimConditionType) of VitalDelayType01;
   type TimeArray01Z  is array (SimConditionType) of VitalDelayType01Z;
   type TimeArray01ZX is array (SimConditionType) of VitalDelayType01ZX;

end Timing; --==================== End of package body ======================--
```

APPENDIX D: ABBREVIATIONS

ASIC	Application Specific Integrated Circuit
ASSP	Application Specific Standard Product
BIST	Built In Self Test
ESA	European Space Agency
ESTEC	European Space Research and Technology Centre
FTP	File Transfer Program
HTML	HyperText Mark-up Language
I/F	Interface
IEEE	Institute of Electrical and Electronics Engineers
LFSR	Linear Feedback Shift Register
MISR	Multiple Input Signature Register
MSB	Most Significant Bit
RTL	Register Transfer Level
SDF	Standard Delay File
URL	Uniform Resource Locator
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VITAL	VHDL Initiative Towards ASIC Libraries