Modeling Monitors in VHDL

Matthias Bauer, Jörg Böttger, Wolfgang Ecker, Peter Jensen
Siemens AG, Corporate Research and Development, ZFE T SE 5
D-81730 Munich, Otto-Hahn-Ring 6
Email: matthias.bauer@zfe.siemens.de, wolfgang.ecker@zfe.siemens.de

Abstract

We present a template based approach for modeling monitors in VHDL with the intention to show a flexible VHDL system level modeling method. First an overview over existing approaches is given and the monitor approach is motivated. Second general software and VHDL aspects are discussed. Subsequent an application modeling of monitors in VHDL is shown. Finally we outlook towards protected Ada types, which have similar behavior as the planned VHDL shared variables.

1 Introduction

1.1 Design and Abstraction

Top down design methodology is widely used to cope with design complexity. Specifications in the traditional sense document interim results of design stages in a natural language. These specifications grow with the complexity of the systems. Currently they are replaced by executable specifications, to make verification, validation, and analysis of design steps and their results possible. Different approaches to abstraction are used to reduce the model expense of executable specifications, e.g.:

- Explicitly described freedom like don't care ('-').
- Using statistic instead of functional models.
- Considering performance aspects only.
- Application of super symbols like records, subroutines or classes as known from software design.
- Incompleteness by omitting partial behaviour like error cases or initialization by reset.
- Abstraction in time, value, and description style.

1.2 Models for Abstraction

A classification of design levels related to abstraction in time, value and description style was presented in [Ram91]. The design cube [EcHo92] defines these abstraction levels independently giving a three dimensional design space assigning each of them to the corresponding axes. Time abstractions namely propagation delay, clock relation and causality are seen as the most important factor in this model. A comparative study of different description or specification languages according to their abstraction mechanisms can be found in [NaGa93]. We will concentrate in the rest of the paper on VHDL based approaches only.

1.3 Related Work

Special abstract modeling approaches, such as the use of Petri Nets [AbCo90, FRBC91, MüKr93, Ram93, SRAJ94, WiMo94], statistic system models [HuTo90], or performance models [CaHP95] are used for early system evaluation. Also the application of software techniques like structured analysis for early real time system modeling is applied [LSK91, SKS91]. Another approach to reduce modeling effort is the application of incomplete specification and incremental design as proposed in [Hoh91].

An application specific approach for architecture evaluation, considering full functionality and timing for analysis, can be found in [PSL91]. Here timing and functionality are modeled in one run but a lot of modeling effort need to be spent. A new approach presented in [ScEc96] reduces the modeling expense for clock related description. However it still requires more model expense than pure causal description.

An extended VHDL-subset for time abstraction is described in [BeSt91] and a pure VHDL based approach can be found in [EcMa93] and [HuDi95]. VHDL modeling of causal synchronization by using semaphores [Eck92] and data exchange by using communication channels and global memory [Wyt95, BaEc93] are the basis for systematic VHDL system level modeling. However, they restrict the application of VHDL to a set of predefined operations.

Monitors are a highly flexible method for causal data exchange and synchronization, well known from concurrent programming. In this paper we show an approach for modeling monitors in VHDL with the intent to present a flexible VHDL system level modeling method.

1.4 Overview

The paper is organized as follows: First monitors and the software method template are introduced. Afterwards VHDL aspects for using signals as global storage object are discussed. The presentation of a VHDL modeling method for monitors follows subsequent. An outlook to Ada's protected type, which closely relate to the currently favorized concept of shared variables, conclude the paper.

2 Software Principles

First a monitor shall be defined. Due to the fact that monitors can not generally be described by using VHDL statements only the software principle template is used in our approach. Thus we discuss this term subsequent. The definitions in this section closely related to the definitions of the Free On-Line Dictionary Of Computing [FOLDOC].

2.1 Monitors

A monitor encapsulates values, concurrent access procedures to this values, and instantiation code within an abstract data type. The monitor's values may only be accessed via its access procedures and only one process may active in the monitor at any time. The access procedures are critical sections i.e. can be executed exclusively only. Additionally an entry condition may be specified which must be satisfied for the execution of access procedures.

A monitor may have a queue of processes which are waiting to access it. If the critical region of the monitor is free the entry condition of all processes accessing the monitor is checked. If more than one of the processes have a true entry condition the process to enter the monitor is selected non-deterministically.

It is very easy to use monitors to model semaphores or shared memory but its major strength lies in the possibility modeling additional behavior within the synchronization mechanism. In this way shared stacks or queues can be modeled.

2.2 Templates

Templates are pseudocode requiring further hand-coding before compilation. This technique is used in cases where capabilities of programming languages for parametrizing or incremental extensions are not capable enough. Templates are often the source of several executable binaries whereas generic units occur only once.

The specification of a generic type is often performed by templates. These templates can be generated mostly by text replacement and expansion. Some languages like Ada do not need templates in this case due to the fact that they support generic types.

Templates are also often used if a piece of code requires incremental extension. This can partially be done by inheritance but inheritance is restricted to extension of the object's state or addition of a new method or redefinition of an existing method. Statements describing the behavior of a method can only completely be replaced but not incrementally be extended.

Inheritance probably would be sufficient for our application but due to the fact that VHDL currently does not support this feature we use a template approach to add user defined functionality to general monitor queuing and synchronization behavior.

3 VHDL Aspects

For the implementation of monitors in VHDL the definition of causality is needed. Causality of sequential statements is implicit specified by their order. To model causality between concurrent statements in VHDL the attribute 'transaction together with a signal assignment statement can be used. In the following this is called signal attribute method [Eck95].An example is given in Figure 1:

```
signal s : integer;

A : process                              B : process
   constant value : integer := 1;           variable value: inte-
ger;
begin                                    begin
   s <= value;◄━━━━━━━━━━━━━━━━━━━━━━━►      wait on s'transaction;
   -- ...                                   value := s;
end process;                                -- ...
```

Figure 1: Synchronization problem using 'transaction

The signal attribute method contains the problem that execution of the concurrent statements dependents on their execution order. If the synchronization statement (s<=value) of process A is executed at least one delta cycle before the synchronization statement (wait on s'transaction) of process B then process B is not able to receive any data. Process B is even blocked until signal s is changed again. Another problem using the signal attribute method is that the attribute 'transaction can not be used in functions and procedures because VHDL implicit generates a new signal for a 'transaction.

To solve this problem a delta handshake is added to synchronize the processes. Also the communication signal are expanded to a record type. Now, the signal not only stores the value but also some management information. This requires the signal to be updated by all synchronized processes. VHDL solves this problem with a resolution function. For each process which updates the signal one driver is initialized. The signal value is derived from the values of the drivers, if at least one process updates the signal. The resolution function can access the old values of the signal even as the new one.

The signal can be accessed at different times. The drivers of the processes can contain different values. To find the right value from the right driver a time stamp is included in the management information. The time stamp is a discrete value and is increased, if any process has access to the signal.

Because of the hierarchical evaluation of the signal tree in VHDL, the management information contains some more entries. If a resolution function returns an invalid value during one level of the hierarchical evaluation of the signal tree, then it is possible to evaluate the correct value in one of the next recursive steps by using the management information of all drivers.

In this approach a signal is used as memory. With this method it is possible to model a monitor in VHDL.

4 Modeling a Monitor Template

Modeling monitors by signals the communication between the processes and the monitor itself is very limited in flexibility. Upon receiving acknowledge for a request, the chosen process must send release back to the monitor after a simulation delta. Therefore the access to and computing of the data resource in the monitor must be carried out instantaneously.

Due to the fact that all values in the memory of the monitor must be transferred to the accessing process and back again, the monitor needs to be told where to fetch the newest and valid data. This is done by the release signal.
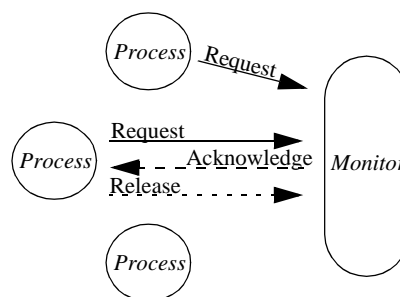

Figure 2: Process monitor interaction

In the template in Figure 3, this handling of handshaking signals is hidden in VHDL procedures. This allows the user

of the template to access the monitor without any knowledge of the handshaking protocol. The VHDL procedure monitor_xxx is a template for a user defined procedure which may gain access to the monitor data. The procedure call of monitor_xxx must be supplied with a PID (Process Identification Number) and a priority. The PID for each process must be unique. The monitor serves the oldest pending processes first, and if more with the same age are pending, the monitor gives access to the process with the highest priority.

The called procedure timestamp_monitor stamps the process for later determination of the age of the request. Hereafter the procedure enters a loop, in which it constantly requests for monitor control if <equation> evaluates TRUE at present time. This equation could be examining the data in the monitor which constantly are updated.

The loop is exited when the process has gained access to the monitor. At this point the data in the monitor may be altered by the process if desired. Hereafter the monitor is given the control back by calling the release procedure.

```
procedure monitor_xxx (xxxxxxxx xxx      : xxx    xxx_type;        -- user def.
                       constant pid      : in     pid_type;
                       constant priority: in     priority_type;
                       signal   monitor : inout monitor_type) is
    variable allow     : boolean;
    variable time_stamp: time_type;
begin
    timestamp_monitor(monitor, time_stamp);
    while not allow loop
       invoke_monitor(<equation>, time_stamp, pid, priority, monitor, allow);
    end loop;
    -- Operations on data in monitor.data
    -- NO WAITS ALLOWED (not even deltas)
    release_monitor(monitor);
end;
```

Figure 3: Template for monitor accessing behavioral

The functionality of the monitor itself is described in the monitor algorithm in Figure 4. This algorithm can be transferred directly into a VHDL resolution function, which computes the monitor signal every time the processes using the monitor change their input to the monitor.

```
Compare all PID's, check uniqueness
Fetch valid monitor data from process according to following priority list:
    1. 'Release' just made
    2. 'Newest' time stamp
Find possibly requesting process according to following priority list:
    1. Oldest pending, requesting process
    2. If more pending processes with same timestamp then let priority weight
Return the valid data with newest timestamp
Return the PID of the possbily chosen requesting process.
```

Figure 4: Monitor algorithm

```
procedure monitor_push (constant stack_data_in : in     stack_data_type;
                        constant pid           : in     pid_type;
                        constant priority      : in     priority_type;
                        signal   monitor       : inout monitor_type) is
    variable allow      : boolean;
    variable time_stamp : time_type;
begin
    timestamp_monitor(monitor, time_stamp);
    while not allow loop
       invoke_monitor((monitor.data.stack_size < STACK_SIZE), time_stamp, pid, priority, monitor, allow);
    end loop;
    monitor.data.stack_data(monitor.data.stack_size) <= stack_data_in;
    monitor.data.stack_size <= monitor.data.stack_size + 1;
    release_monitor(monitor);
end;


procedure monitor_pop (variable stack_data_out : out    stack_data_type;
                       constant pid            : in     pid_type;
                       constant priority       : in     priority_type;
                       signal   monitor        : inout monitor_type) is
    variable allow      : boolean;
    variable time_stamp : time_type;
begin
    timestamp_monitor(monitor, time_stamp);
    while not allow loop
       invoke_monitor((monitor.data.stack_size > natural'low), time_stamp, pid, priority, monitor, allow);
    end loop;
    stack_data_out := monitor.data.stack_data(monitor.data.stack_size - 1);
    monitor.data.stack_size <= monitor.data.stack_size - 1
    release_monitor(monitor);
end;
```

Figure 5: Stack in VHDL

5 Applying the Monitor Template

To demonstrate the application of the monitor template a stack example is chosen. Using this high level modeled global stack initially in the design process avoids the description of the global stack as independent architecture. The stack must be constrained due to the nature of a VHDL signal, but the monitor applies all the needed control structures. Two procedures (Push and Pop) are presented in Figure 5:

6 A Look Towards Ada

Ada95 was extended by protected types, which allow for a synchronized access to global data. A look towards this capabilities is of high interest from the VHDL point of view due to the fact that the planned definition of VHDL shared variables closely relate to this type.

6.1 Short Description of Protected Types in Ada

Ada's protected types are theoretically based on the monitor concept. Thus, a resource (shared data) is encapsulated and any access to this resource is only possible via explicitly specified operations. Both, data and operations, are declared in the protected type. Three kinds of protected operations exist: protected procedures, protected functions and protected entries (Figure 6).

Protected procedures provide mutually exclusive read/write access to the protected data. Protected functions provide concurrent read-only access to the protected data. Simultaneously execution of protected functions is possible, due to the fact that functions do not change the state of a resource (represented by the set of variable values of this resource). Thus, mutual exclusion in protected types in Ada prohibits concurrent write access and read/write conflicts. In this way protected procedures cannot be executed at the same time as another protected operation (procedure or function) has occupied the protected data. A protected function cannot be executed if a protected procedure has occupied the data. Every protected operation that cannot occupy the resource must wait until the protected data is released by the occupying operation.

	protected function	protected procedure	protected entry
operations on protected data	read-only	read / write	read / write
access to protected data	concurrent to other protected functions; mutually exclusive to protected procedures/ protected entries	mutually exclusive	mutually exclusive
additional feature			access to protected data only when barrier evaluates true

Figure 6: protected operations in Ada

The power of protected types in Ada results from protected entries. A protected entry is similar to a protected procedure: it guarantees to execute in mutual exclusion and has a read/write access to the encapsulate data. However, execution of a protected entry and its access to a protected resource is done after evaluation of a boolean expression inside the protected type (called barrier or guard). If this barrier evaluates false, the calling task will be suspended until the barrier is changed to true. Protected entries makes sense when the successful execution of a protected operation depends on the internal state of the protected data (e.g. pop on an empty stack, write to a full buffer). Protected entries realize conditional synchronization.

Additionally, protected types in Ada can be defined inside generic units. Formal parameter of generic units can be variables, types, subprograms and packages. Protected types can be parametrized by this parameter types in a high flexible way.

6.2 Example

The example in Figure 7 describes similar to the VHDL monitor a Stack and its operations Push and Pop implemented by using a protected type in Ada. Realization of shared data with mutual exclusive access is effected only by

the description language.

The entries Push and Pop permit access to the Stack. A Pop call will only be allowed to proceed through the barrier when at least one data item can be retrieve from the stack. If the stack is empty the Pop calling task will be suspended until a Push has stored data on the stack. The entry Push with its barrier when (Top <= Max_Stack_Size) works analogously.

The shortness of this example in comparison to the effort required for VHDL monitor template highlights the expressive power of this language constructs in Ada. However, comparing this example with the VHDL stack example shows the advantage of using the proposed template for VHDL system level modeling. Nevertheless the implementation of shared variables according to this scheme would give VHDL additional power for system level design.

```
constant Max_Stack_Size : natural;
type Stack_Mem_Type is array  (1 .. Max_Stack_Size) of Data_Item;

protected type Stack is                             - specification (interface)
   entry Push(Item : in Data_Item);
   entry Pop(Item : out Data_Item);
private                                              - no visable
   Stack_Mem : Stack_Mem_Type;
   Top : integer range 0 .. Max_Stack_Size := 0;
end Stack;

protected body Stack is                             - body of protected type
   entry Push(Item : in Data_Item) is
      when (Top <= Max_Stack_Size) is               - barrier
   begin
      Top := Top + 1;
      Stack_Mem(Top) := Item;
   end Push;

   entry Pop(Item : out Data_Item) is
      when (Top >= 0) is                            - barrier
   begin
      Item := Stack_Mem(Top);
      Top := Top - 1;
   end Pop;
end Stack;
```

Figure 7: Stack in Ada

7 Conclusion and Outlook

We presented a modeling strategy for monitors and showed thus a new highly flexible approach for using VHDL at system level supporting both concurrent synchronization and data exchange. We apply monitors in actual designs for both behavioral and testbench modeling. Future examination is planned for replacing signals by shared variables application if they are available and support synchronization similar to Ada's protected type.

8 Bibliography

[AbCo90]   Aboulhamid, M.; Cordeau, M.: System Level Modeling in VHDL using Timed Petri Nets. Proceedings of the EURO-VHDL'90

[ARM95]   Ada Reference Manual. ISO/IEC 8652:1995(E)

[BaEc93]   Bauer, M.; Ecker, W.: Communication Mechanisms for Specification and Design of Hardware Starting at System Level. Proceedings of the VHDL-Forum '93 Working Conference.

[BeSt91]   Benders, L.; Stevens, M.: Petri Net Modeling of Task Level Behavioural VHDL for VLSI. Proceedings of the EURO-VHDL'91.

[BuWe95]   Burns, A.; Welling, A.: Concurrency in Ada. Cambridge University Press, 1995

[CaHP95]   Calvez, J-P.; Heller, D.; Pasquier, O.: System Performance Modeling and Analysis with VHDL: Benefits and Limitations. Proceedings of the VHDL-Forum '95 Working Conference.

[Coh96]   Cohen, N.: Ada as a second language. McGraw-Hill, 1996.

[Eck95]   Ecker, W.: Neue Verfahren für den Entwurf digitaler Systeme mit Hardwarebeschreibungssprachen. Shaker, 1996.

[EcSc92]   Scheuer,A.; Ecker, W.: Semaphores in HW-Design. Proceedings of the VHDL-Forum '92 Working Conference

[EcHo92]   Ecker, W.; Hofmeister, M.: The Design Cube - A Model for VHDL Design-Flow Representation. Proceedings of the EURO-VHDL'92.

[EcMä93]   Ecker, W.; März, S.: System level design using VHDL: A case study. Proceedings of the CHDL'93.

[FOLDOC]   Free On-line Dictionary Of Computing: http://wombat.doc.ic.uk/

[FRBC91]   Fermy, A.; Rossignol, B.; Bakowski, P.; Calvez, J.: Tools to Design at a Functional Scheme Level using VHDL. Proceedings of the EURO-VHDL'91.

[GVNG94]   Gajski, D.; Vahid, F.; Narayan, S.; Gong, J.: Specification and Design of Embedded Systems.Prentice Hall, 1994.

[HaBr95]   Hashmi, K.; Bruce, A.: Design and Use of a System-Level Specification and Verification methodology.
Proceedings of the EURO-VHDL'95.

[Hoh91]   Hohl, A.: Incremental Design - Application of a Software-Based Method for High-Level Hardware Design with VHDL. Proceedings of the EURO-VHDL'91.

[HuDi95]   van den Hurk, J.; Dilling, E.: System Level design, a VHDL Based Approach. Proceedings of the EURO-VHDL'95.

[HuTo90]   Hubbard, P.; Torres, J.: Using VHDL for High-Level and stochastic System Modeling. Proceedings of the EURO-VHDL'90.

[LSK91]   Lahti, J.; Sipola, M.; Kivelä, J.: Behavioural System Modeling with structured Analysis and VHDL. Proceedings of the EURO-VHDL'91.

[MüKr93]   Müller, J.; Krämer, H.: Analysis of Multi-Process VHDL Specifications with a Petri net Model. Proceedings of the EURO-VHDL'93.

[NaGa93]   Narayan, S.; Gajski, D.: Features Supporting System-Level Specification in HDLs. Proceedings of the EURO-VHDL'93.

[PSL91]   Pitkänen, P.; Skyttä, J.; Laakso, T.: Comparison of Digital Filter Architectures Using VHDL. Proceedings of the EURO-VHDL'91.

[Ram91]   Rammig, F.: Approaching System Level Design. Proceedings of the EURO-VHDL'91.

[Ram93]   Rammig, F.: Modeling Aspects of System Level Design. Proceedings of the EURO-VHDL'93.

[ScEc96]   Schneider, C.; Ecker, W.: Step-Wise Refinement of Behavioural VHDL Specifications by Separation of Synchronization. To appear in the Proceedings of the EURO-VHDL'96.

[SKS91]   Sipols, M.; Kivalä, J. Soininen, J.: System Real Time Analysis with VHDL from Graphical SA_VHDL. Proceedings of the EURO-VHDL'91.

[SRAJ94]   Swaminathan, G.; Rao, R.; Aylor, J.H.; Johnson, B.W.: A VHDL Based Environment for System Level Design and Analysis. Proceedings of the VIUF Spring Conference 1994.

[WiMo95]   Wisley, P.A.; Mohanty, S.: VHDL Design Libraries for Rapid System Prototyping. Proceedings of the International Conference on Simulation in Engineering Education (ICSEE) 1994.

[Wyt95]   Wytrebowitcz, J.: Modeling Shared Variables in VHDL - a Pragmatic Approach. Proceedings of the VHDL-Forum '95 Working Conference.