TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Implementation of mutual exclusion in VHDL

Citation for published version (APA):
Boersma, M. V., Benders, L. P. M., & Stevens, M. P. J. (1994). Implementation of mutual exclusion in VHDL. In P. A. Wilsey, & D. Rhodes (Eds.), *Proc. International Conference on Simulation and Hardware Description Languages, 1994 Western Multiconference* (pp. 57-62). Simulation Councils, Inc..

Document status and date:
Published: 01/01/1994

Document Version:
Accepted manuscript including changes made at the peer-review stage

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 02. Oct. 2023

IMPLEMENTATION OF MUTUAL EXCLUSION IN VHDL

M.V. Boersma[a], L.P.M. Benders[b] and M.P.J. Stevens[a]

[a]Eindhoven University of Technology
[b]Open Universiteit Heerlen and Eindhoven University of Technology
Department of Electrical Engineering, Section of Digital Information Systems, EH 10.35, P.O. Box 513,
5600 MB Eindhoven, The Netherlands, e-mail: leon@eb.ele.tue.nl

ABSTRACT

In VHDL it is difficult to implement mutual exclusion at an abstract level since atomic actions are required. A local status model and an arbiter model are presented to achieve mutual exclusion in VHDL. Shared data, protected by a mutual exclusion mechanism, cannot be modelled as a simple, resolved VHDL signal since no resolution function is able to return the correct value. By changing the signal type to a special record type this problem can be solved, using the arbiter model. The specification language Task Level VHDL (TLVHDL) has been developed to specify communication and synchronization mechanisms at an abstract level. In TLVHDL the abovementioned problems are not encountered. A back end compiler converts the abstract TLVHDL description into a VHDL specification, according to a chosen mutual exclusion model. All modifications are handled by the compiler and are of no concern to the designer.

1 INTRODUCTION

In the field of automated system design several phases can be distinguished. The first phase concerns the complete and detailed functional specification of the system to be designed. After some transformations and optimizations the final specification can serve as input to a synthesis system. Before the actual synthesis phase of the system design process the correctness of the specification must be established. For this reason simulation is of great importance. For both specification and simulation VHDL is a useful language.

In Figure 1 an overview of the specification phase of a design process is shown. The system specification (co-design) language Task Level VHDL (TLVHDL) is used to specify embedded systems. TLVHDL consists of a subset of VHDL, extended with several communication and synchronization constructs. TLVHDL describes a system at a more abstract level than VHDL, which simplifies analysis and transformation of the specification. For analysis of the specification coloured Petri nets are used (Benders and Stevens 1991c, 1992) After transformation and optimization the TLVHDL specification is converted into a standard VHDL specification by means of a compiler (Benders and Stevens 1991b). The specification can then be simulated.

Mutual exclusion is a very important synchronization mechanism that can be modelled very easily in TLVHDL. In standard VHDL no constructs for achieving mutual exclusion are available. In this paper we show how mutual exclusion can be modelled in VHDL and how this VHDL description is related to a TLVHDL specification. Besides mutual exclusion also interprocess communication, wait and signal synchronization, shared processing etc. are specific TLVHDL concepts (Benders and Stevens 1991a). In this paper these concepts are not discussed.

When using a mutual exclusion mechanism to protect shared data, only one process at a time is allowed to operate on this protected resource (in VHDL very often represented by a signal). However, many processes contain a driver for the resource, so a resolution function is necessary to determine its actual value. This value should always be equal to the latest value that has been assigned to the resource. However without special measures no resolution function is able to determine the correct value of the data. Often this difficulty is left as a problem to the designer. We have extended our TLVHDL mutual exclusion mechanism in such a way that this problem is taken care of as well.

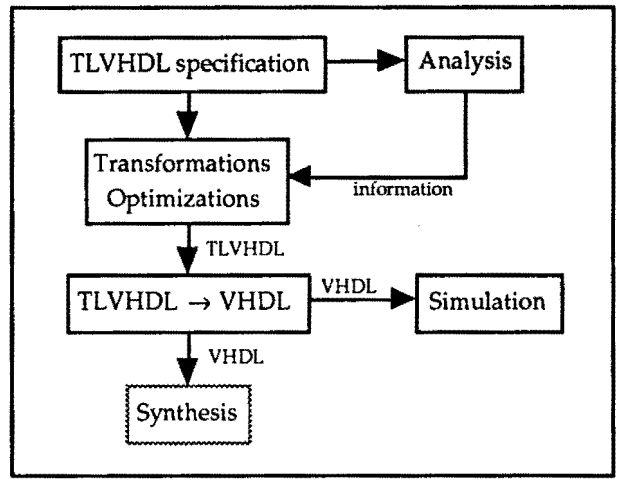

Figure 1: The specification phase of a hardware design process.

2 MUTUAL EXCLUSION IN TLVHDL

In VHDL all communication and synchronization protocols are modelled using the signal concept. It is often very difficult to identify the relationship between these signals and to understand their purpose. This makes analysis and transformation of a VHDL description very hard. To be able to handle these problems a more abstract language has been developed, Task Level VHDL (TLVHDL). The system specification language TLVHDL (Benders and Stevens 1991a) consists of a subset of VHDL, extended with several communication and synchronization constructs which replace the signal concept. A designer only models the desired behaviour and does not have to be concerned about the implementation details.

Mutual exclusion is a very important synchronization mechanism to protect data which are shared by different processes. In TLVHDL a data structure that must be protected by a mutual exclusion mechanism is explicitly declared by means of the keyword 'ME_DATA' (mutual exclusion data). This ME_DATA object does not require any kind of resolution function. Different processes can make use of a ME_DATA

object independently, provided that access is permitted by the mutual exclusion mechanism.

An example of the modelling of mutual exclusion in TLVHDL is shown in Figure 2. The exclusive accessible resource *store* is declared as a ME_DATA object. This object will now be protected by a mutual exclusion mechanism that makes use of the primitives *wait_exclusion* and *release*. Access to the protected resource can only be obtained with the help of these primitives. In Figure 2 two processes claim exclusive right to operate on the protected data structure *store* by means of the statement ***wait_exclusion*** *store*. Only one of these processes actually gets permission to operate on *store* and may proceed. The execution of the other process is temporarily suspended. After execution of the statement ***release*** *store* the resource is released and the suspended process may proceed.

```
architecture behavior of system is
   ...
   type memory is ...
   me_data store: memory;
begin

   proc1: process
   begin
      ...
      wait_exclusion store;
      ... - - operations on store
      release store;
      ...
   end process proc1;

   proc2: process
   begin
      ...
      wait_exclusion store;
      ... - - operations on store
      release store;
      ...
   end process proc2;

end behavior;
```

Figure 2: TLVHDL modelling of mutual exclusion.

In hierarchical systems ME_DATA objects must be declared at the highest hierarchical level on which they are used. Via port and port map constructs (as in standard VHDL) exclusive access to these objects can be claimed in components as well.

3 MUTUAL EXCLUSION IN VHDL

In VHDL mutual exclusion can be implemented in several ways. We will discuss two models, according to which processes may obtain permission to use exclusive accessible resources.
In the first model (in the sequel also called the 'local status model') some well known algorithms are used, which have been modificated for use in VHDL. On the basis of these algorithms every process that requests access to an exclusive accessible resource is able to calculate when access to this resource actually is permitted (i.e. when it is permitted to enter its critical section). However, this model will appear to be unsuitable for use in hierarchical systems.

In the second presented model (the 'arbiter model') a new process is introduced for every exclusive accessible resource: an arbiter. Not the processes itself but the arbiter now determines which process is allowed to operate on the resource exclusively. This model can be applied in hierarchical systems without problems.

3.1 The Local Status Model

In VHDL no standard mechanism to implement mutual exclusion is present, so the designer must apply special algorithms. One of the first algorithms to achieve mutual exclusion was presented by Dijkstra (Dijkstra 1965). This solution to the mutual exclusion problem handles the general case that N processes are involved. On the basis of this algorithm every process that wishes to perform its critical section can determine whether it is allowed to proceed or has to wait. Dijkstra's algorithm was improved several times (Knuth 1966; Bruijn 1967). Eisenberg and McGuire (Eisenberg and McGuire 1972) presented an improved algorithm that met all requirements concerning deadlock and fairness. In their solution the maximum time a process remains waiting before being allowed to enter its critical section is limited to N-1 turns (the word 'turn' refers to a process performing its critical section). In Figure 3 (left) the algorithm of Eisenberg and McGuire is listed. The form in which it is listed is slightly different from the original form and is taken from (Raynal 1986).

```
repeat
   flag[i] ← requesting;

   j ← turn;
   while j ≠ i do
      if flag[j] ≠ passive
            then j ← turn;
            else j ← (j+1) mod N;
      endif;
   enddo;
   flag[i] ← in-cs;

   j ← 0;
   while (j < N) ∧ (j = i ∨
         flag[j] ≠ in-cs) do
      j ← j+1;
   enddo;
until (j = N) ∧ (turn = i ∨
      flag[turn] = passive);

turn ← i;


< Critical Section >

j ← (turn+1) mod N;
while (j ≠ turn) ∧ (flag[j] =
                  passive) do
   j ← (j+1) mod N;
enddo;
turn ← j;


flag[i] ← passive;
```

```
turn <= null;
loop
   flag(i) <= requesting;
   wait for 0 ns;
   j := turn;
   while j /= i loop
      if flag(j) /= passive then
            wait on flag, turn;
            j: = turn;
      else j := (j+1) mod N;
      end if;
   end loop;
   flag(i) <= in-cs;
   wait for 0 ns;
   j := 0;
   while (j < N) and (j = i or
                  flag[j] /= in-cs) loop
      j := j+1;
   end loop;
   exit when (j = N) and (turn = i
         or flag(turn) = passive);
end loop;
turn <= i;
wait for 0 ns;

< Critical Section >

j := (turn+1) mod N;
while (j /= turn) and (flag(j) =
                        passive) loop
   j := (j+1) mod N;
end loop;
turn <= j;
wait for 0 ns;
turn <= null;
flag(i) <= passive;
wait for 0 ns;
```

Figure 3: The mutual exclusion algorithm of Eisenberg and McGuire for N processes.

The algorithm makes use of a global variable *turn* to show which process (0..N-1) enters its critical section. In at most one process at a time a value is assigned to this common variable *turn*. The algorithm can be adjusted for use in VHDL. The global variable *turn* then must be represented by a signal. Though

in at most one process at a time a value can be assigned to *turn*, the signal does have more drivers and so a resolution function must be used to determine its value. To be able to write a resolution function that always returns the latest assigned value (i.e. the value that is assigned to *turn* in the only process that is allowed to enter its critical section), all other drivers of *turn* have to be disconnected at that moment. *Turn* now has at most one driver. In a process a driver is disconnected when in that process *turn* is assigned the value *null*. For this purpose the signal *turn* can be represented by a guarded signal of kind 'bus' or 'register'. When all drivers are turned off, the resolution function should not be called at all, so *turn* must be of the kind 'register'. A simple resolution function will now be sufficient to return the value of the only present driver unchanged (Figure 4).

The algorithm in Figure 3 (left) uses the global variable *flag*, declared as array of N elements, to show the status of the processes. A process i can read all elements of the array, but can assign a value to the i-th element *flag(i)* only. In VHDL *flag* therefore can be represented by a non-resolved signal. The variable *j* is local to each process. In the final algorithm *i* must be replaced by the number of the process ($0 \le i < N$). In Figure 3 (right) the VHDL implementation of the algorithm of Eisenberg and McGuire is listed. The statements *wait for 0 ns* are inserted to ensure that all signal values are updated by the simulator before execution of the algorithm continues. In Figure 4 all necessary type declarations and the resolution function are listed.

Peterson (Peterson 1981) presented a simple solution to the mutual exclusion problem for the case that only two processes i and j are involved (N=2). In Figure 5 (left) this algorithm is listed, again in a slightly different form according to (Raynal 1986). *Flag* is now declared as *array(0 to 1) of boolean*. When using this algorithm it is possible that both processes assign a value to *turn* simultaneously. In that case the resolution function returns one of both assigned values and so exactly one process will enter its critical section. After this process has left its critical section the other process may proceed.

```
package me_types is
    type status is (passive, requesting, in_cs);
    type status_array is array(0 to 1) of status;
    type process_array is array(natural range <>) of integer;
    function resolve_turn(driver: in process_array) return integer;
    subtype resolved_turn is resolve_turn integer;
end me_types;

package body me_types is
    function resolve_turn(driver: in process_array) return integer is
    begin
        return driver(driver'left);
    end resolve_turn;
end me_types;
```

Figure 4: Type declarations and resolution function.

After a value is assigned to the signal *turn* and this signal has been updated, the driver concerned is turned off immediately. If later in exactly one process a value is assigned to *turn*, the resolution function will return this value unchanged. The VHDL implementation of the algorithm is also listed in Figure 5 (right).

The used resolution function (Figure 4) can only return a correct value if every driver of the signal *turn* can be disconnected. However, in a hierarchical description not all drivers can be disconnected, for instance the port parameters of an instantiated component. This makes the model unsuitable for use in hierarchical systems.

```
flag[i] ← true;                          flag(i) <= true;
turn ← i;                                turn <= i;
                                         wait for 0 ns;
                                         turn <= null;
wait until                               while not (flag(j) = false or turn = j) loop
    (flag[j] = false v turn = j);            wait on flag, turn;
                                         end loop;

< Critical Section >                     < Critical Section >

flag[i] ← false;                         flag(i) <= false;
                                         wait for 0 ns;
```

Figure 5: The mutual exclusion algorithm of Peterson for 2 processes.

3.2 The Arbiter Model

To regulate access to exclusive accessible data structures it is also possible to introduce a new process for every protected resource: an arbiter. If a hierarchical description is used, the arbiter process must be inserted at the highest hierarchical level on which the protected resource is used. The arbiter communicates with all processes that request access to the resource by way of bidirectional status channels. Between every involved process and the arbiter a channel, represented by a resolved signal, is created. A channel can take on three possible values: *free*, *request* or *grant*.

If a process has not requested permission to enter its critical section yet or already has left its critical section, the channel will have the value *free*. A process can request permission by changing this value to *request*. The arbiter can then permit the process to perform its critical section by setting the value of the status channel to *grant*.

process	arbiter	channel	action
free	free	free	initial state
request	free	request	process claims access
request	**grant**	grant	arbiter allows access
free	grant	free	process leaves CS
free	**free**	free	return to initial state

Figure 6: Resolution scheme of a status channel.

Both the requesting process and the arbiter process contain a driver for the channel. It follows that the channel must be represented by a resolved signal. In Figure 6 the possible driving values of this signal are shown, as well as the value that is returned by the resolution function. The bold printed values are the values that have actually changed. The assignment of the value *free* to the channel by the arbiter (after the process has released the protected resource) is necessary to return to the initial state. In Figure 7 the necessary type declarations and the resolution function are listed.

In Figure 8 the VHDL implementation of the mutual exclusion mechanism with arbiters is listed. In the example only two requesting processes are involved. To regulate access to the shared resource *store* an arbiter process *arbiter_store* has been introduced. If one of the processes wishes to perform its critical section, it assigns the value *request* to the status channel concerned. Next the process suspends until permission is given by the arbiter. As soon as the arbiter has noticed a request on a status channel, it tests the channels of all possibly requesting processes in a fixed order until the requesting process is found (in this way the arbiter model guarantees fairness). Next the

arbiter permits this process to perform its critical section by assigning the value *grant* to the channel. The arbiter process then suspends until the process leaves its critical section. The process leaves the critical section by assigning the value *free* to the channel. The arbiter detects this release and also assigns the value *free* to the channel. This restores the initial state of the communication channel. For two reasons it is important to return to the initial state before a new request can be made. First it enables the resolution function to distinguish the situation in which a process requests access to a protected resource from the situation in which the arbiter grants this request (Figure 6). Secondly it prevents the assignment *free* from being overwritten by the assignment *request* in case a process requests access to a resource immediately after it has released this same resource (Figure 8).

```
package me_types is
    type status is (free, request, grant);
    type status_array is array(natural range <>) of status;
    function resolve_status(driver: in status_array) return status;
    subtype channel_type is resolve_status status;
    type channel_array is array(natural range <>) of channel_type;
end me_types;

package body me_types is
    function resolve_status(driver: in status_array) return status is
    begin
        for i in driver'range loop
            if driver(i)=request then
                for j in driver'range loop
                    if driver(j)=grant then return grant; end if;
                end loop;
                return request;
            end if;
        end loop;
        return free;
    end resolve_status;
end me_types;
```

Figure 7: Type declarations and resolution function.

After the shared resource has been released the arbiter starts testing every channel again to check whether meanwhile other requests have arrived. If another request is noticed the process concerned will be given permission to access the protected data structure, otherwise the arbiter process suspends until a new request arrives.

In the example in Figure 8 two proceses claim access to a protected resource. When more than two processes are involved, only the size of the array *channel_store* (indicating the number of channels) has to be adjusted.

For every protected resource a different arbiter process must be introduced. The readability of the model can be increased by replacing these different arbiter processes by one single procedure that may be defined in the package *me_types* : ***procedure arbiter(signal channel: inout channel_array)***. The body of this procedure resembles the body of the arbiter process, embedded in a loop statement. The necessary arbiter functions are now performed by concurrent calls of this procedure. In the previous example the process *arbiter_store* would be replaced by the concurrent procedure call *arbiter_store: arbiter(channel_store)*.

4 OPERATIONS ON SHARED DATA

In VHDL it is difficult to perform operations on shared data which are represented by a signal, caused by the fact that these data are used by several processes (not at the same time). An ordinary signal assignment is not sufficient because the resource has more drivers and a resolution function must be

```
architecture behavior of system is
    ...
    type memory is ...
    signal store: .... - - some resolved type, derived from memory
    signal channel_store: channel_array(0 to 1);
begin
    ...
    proc1: process
    begin
        ...
        channel_store(0) <= request; wait until channel_store(0) = grant;
        ... - - operations on store
        channel_store(0) <= free; wait until channel_store(0) = free;
        ...
    end process proc1;

    proc2: process
    begin
        ...
        channel_store(1) <= request; wait until channel_store(1) = grant;
        ... - - operations on store
        channel_store(1) <= free; wait until channel_store(1) = free;
        ...
    end process proc2;

    arbiter_store: process
        constant number_of_processes: integer := channel_store'length;
        variable i: integer := 0;
        variable process_counter: integer;
    begin
        process_counter := 0;
        while (channel_store(i) = free and process_counter <
                                                number_of_processes) loop
            i := (i+1) mod number_of_processes;
            process_counter := process_counter+1;
        end loop;
        if process_counter = number_of_processes
            then wait on channel_store;
        else    - - must have been a request !!
            channel_store(i) <= grant;
            wait until channel_store(i) = free;
            channel_store(i) <= free;
        end if;
        i := (i+1) mod number_of_processes;
    end process arbiter_store;

end behavior;
```

Figure 8: VHDL example of the use of an arbiter. Two processes request exclusive access to the protected resource store.

used to determine its value. However, without further measures no resolution function is able to determine which driver contains the most recently assigned value. We will now describe the way to overcome this problem. For achieving mutual exclusion the mechanism described in subsection 3.2 is used.

It is impossible to define a resolution function that is able to determine the value that is most recently assigned to the resolved signal. A solution to this problem is to disconnect the drivers in all processes that are not allowed to operate on the data at that moment. In that case the resolved signal that represents the data has at most one driver and a resolution function can easily return this value. The signal must be of kind 'register' in order to maintain its value when all drivers are disconnected at the same time. However, this solution can not be applied in hierarchical systems because not every driver of the signal can be turned off in that case.

To be able to deal with hierarchical systems, we developed a more general solution in which disconnection of drivers is avoided. In this solution the protected data structure is represented by a resolved signal that is defined as a record. This

record is defined in the package *mutex_types* (Figure 9) and consists of two fields. The first field (*data*) contains a copy of the original shared data. The second field (*id*) is an identification field that may contain the value *z*, *p* or *a*. On the basis of the value of this identification field the resolution function is able to determine the source of every present driver and to determine the correct value to be returned. The source of a driver with identification field z always is a process that is not permitted to enter its critical section. The source of a driver with identification field *p* always is a process that is actually performing its critical section. The arbiter process always contains a driver with identification field *a*.

Immediately after a process is permitted to operate on exclusive accessible data, the driver of the data must be updated (in fact the driver of the record structure containing the data must be updated). For this purpose the data field is assigned the present value of the data and the identification field is assigned the value *p*. The resolution function now returns the value of this particular driver. From this moment the process can actually perform its critical section and the data can be assigned the desired values. As soon as the process leaves its critical section the value z is assigned to the identification field.

```
package mutex_types is
    type memory is ....        -- non-resolved type of the protected data,
                               -- defined by designer
    type source_status is ('z', 'p', 'a');
    type record_memory is
        record
            data: memory;
            id: source_status;
        end record;
    type record_memory_array is array(natural range <>) of
                                                    record_memory;
    function resolve_memory(driver: in record_memory_array) return
                                                    record_memory;
    subtype mutex_memory is resolve_memory record_memory;
end mutex_types;

package body mutex_types is
    function resolve_memory(driver: in record_memory_array) return
                                                    record_memory is
    begin
        for i in driver'range loop
            if driver(i).id='p' then return driver(i); end if;
        end loop;
        for i in driver'range loop
            if driver(i).id='a' then return driver(i); end if;
        end loop;
        return driver(driver'left);
    end resolve_memory;
end mutex_types;
```

Figure 9: Declaration of the resolved record structure containing the protected data and an identification field.

If no process is performing its critical section (no driver contains the value *p*), the resolution function returns the value of the driver contained in the arbiter process. In order to maintain the actual value of the data, this driver must be updated continuously.

The algorithm in the resolution function is listed in Figure 9. If one of the present drivers contains an identification field *p*, the value of this driver is returned. If no such driver is found, the algorithm looks for a driver with identification field *a*. If a driver is found the function returns the value of this driver. If no such driver is found all drivers apparently contain an identification field with value z. In that case the resolution function returns the value of an arbitrary driver. This latter situation however can only occur as an intermediate result of the resolution of a port in a structural hierarchy. It occurs when in a component no process has access or requests access to a protected resource. All drivers from this component then contain the value z. During the resolution on a higher hierarchical level these values will be neglected, so their value is of no importance. On the highest hierarchical level always an arbiter process is defined, which contains a driver with identification field *a*. *So the finally returned value will be either the value of a driver contained in a process that performs its critical section or, if no such driver is present, the value of the driver contained in the arbiter process.* Hence it follows that the values of the drivers contained in these processes must be updated continuously. The value that will be returned is independent of the sequence of the drivers in the resolution function and therefore is not tool-dependent.

5 THE TLVHDL COMPILER

A designer just specifies the desired operations on the protected data (i.e. the declared ME_DATA object) in TLVHDL. The TLVHDL compiler then automatically produces the related VHDL description, including all necessary data type and resolution function declarations. The generated VHDL implementation (with arbiter processes) derived from the TLVHDL description in Figure 2 is shown in Figure 10. With respect to the description in Figure 8 the lines marked with an asterisk (*) have been added. Besides the signal *store* with type *memory* a signal *mutex_store* with type *mutex_memory* has been declared (Figure 9). The names of the identifier and type are introduced by the compiler and are identical to the existing names respectively, prefixed by *'mutex_'*. The type *mutex_memory* is a record containing a data field (with type *memory*) and an identification field. The processes now operate on the signal *mutex_store* instead of on the signal *store*. In order to hide these adjustments to the outside world, the (originally declared) resource *store* is updated continuously. This updating is performed by means of a concurrent signal assignment statement.

6 CONCLUSIONS AND REMARKS

In TLVHDL mutual exclusion can be modelled more easily than in standard VHDL. A compiler has been developed to convert a TLVHDL description into a VHDL implementation, according to a model that is selected by the designer.

In the local status model some well known algorithms for mutual exclusion are adjusted for use in a VHDL description. The protocols use disconnection of drivers as a method to achieve a resolution function that is able to return the latest assigned value. This concept only works well in a non-hierarchical environment. However, in a hierarchical system not all drivers can be disconnected (f.e. the port parameters of an instantiated component). This makes the model unsuitable for use in hierarchical systems. Our presented arbiter model does not make use of disconnection of drivers and can be applied in hierarchical systems without difficulty.

Because of the driver concept in VHDL it is difficult to implement operations on exclusive accessible data which are represented by signals. We have extended our arbiter model such that values can be assigned to these data structures without problems. All necessary adjustments are made by a TLVHDL compiler.

In both presented models the processes are permitted to enter their critical sections in a fixed, predefined order. The order could be changed by adding a priority indication to the requests. A priority mechanism is not yet implemented, but can easily be implemented in the given arbiter model. The arbiter communicates with every process and can easily compare the

```
architecture behavior of system is
    ...
    type memory is ...
     -- me_data store: memory;
*   signal store: memory;
*   signal mutex_store: mutex_memory;   - - resolved record type
                                        - - containing memory
    signal channel_store: channel_type2;
begin
*   store <= mutex_store.data;   - - continuous updating of the
                                 - - signal store
    ...
    proc1: process
    begin
      ...
      channel_store(0) <= request; wait until
                                        channel_store(0) = grant;
*     mutex_store <= (mutex_store.data, 'p');
      ... - - operations on mutex_store.data
      channel_store(0) <= free; wait until channel_store(0) = free;
*     mutex_store.id <= 'z';
      ...
    end process proc1;

    proc2: process
    begin
      ...
      channel_store(1) <= request; wait until
                                        channel_store(1) = grant;
*     mutex_store <= (mutex_store.data, 'p');
      ... - - operations on mutex_store.data
      channel_store(1) <= free; wait until channel_store(1) = free;
*     mutex_store.id <= 'z';
      ...
    end process proc2;

    arbiter_store: process
      constant number_of_processes: integer:=2;
      variable i: integer := 0;
      variable process_counter: integer;
    begin
      process_counter := 0;
      while (channel_store(i) = free and process_counter <
                                        number_of_processes) loop
        i := (i+1) mod number_of_processes;
        process_counter := process_counter+1;
      end loop;
      if process_counter = number_of_processes
        then wait on channel_store;
      else   - - must have been a request !!
        channel_store(i) <= grant;
        wait until channel_store(i) = free;
        channel_store(i) <= free;
*       mutex_store <= (mutex_store.data, 'a');
      end if;
      i := (i+1) mod number_of_processes;
    end process arbiter_store;

end behavior;
```

Figure 10: Extended description of the mutual exclusion mechanism which takes into account the operation on the resource *store*.

different assigned priorities. It is on the other hand very difficult to introduce a priority mechanism in the local status model. When using priorities all processes must communicate with each other, resulting in an extremely large amount of overhead.

In this paper the specification of mutual exclusion in TLVHDL and the implementation of mutual exclusion in VHDL have been discussed. TLVHDL has been developed to specify the behaviour of systems at system level. Besides mutual exclusion also interprocess communication, wait and signal synchronization, shared processing and conditional control flows are important TLVHDL concepts. A compiler has been developed to translate the TLVHDL description into VHDL, according to models and protocols that are specified by the designer.

At the moment a tool is developed to map the TLVHDL description to synthesizable (behavioural) VHDL. The TLVHDL description has an equivalent executable coloured Petri net model which is used to obtain information about concurrency, queues, communication behaviour, occupation of processors, shared processing etc. This information guides the mapping of the TLVHDL description to synthesizable VHDL.

References

Benders L. and M.P.J. Stevens. 1991a. "Task Level Behavior Hardware Description". *Microprocessing and Microprogramming*, North Holland, No. 32; pp. 323-332.

Benders L. and M.P.J. Stevens. 1991b. "Task level behavioral description translation to IEEE VHDL". *VHDL forum*, Marseille.

Benders L. and M.P.J. Stevens. 1991c. "Petri net modelling of task level behavioral VHDL for VLSI". *Euro VHDL*; 180-184.

Benders L. and M.P.J. Stevens. 1992. "Petri net modelling in embedded system design". *CompEuro 1992*.

Bruijn N.G. de. 1967 "Additional Comments on a Problem in Concurrent Programming Control". *Comm. ACM*, Vol. 10, No. 3 (March); 137-138.

Dijkstra E.W. 1965. "Solution of a Problem in Concurrent Programming Control". *Comm. ACM*, Vol. 8, No. 9 (Sept.); 569.

Eisenberg M.A. and M.R. McGuire. 1972. "Further Comments on Dijkstra's Concurrent Programming Control Problem". *Comm. ACM*, Vol. 15, No. 11 (Nov.); 999.

Knuth D.E. 1966. "Additional Comments on a Problem in Concurrent Programming Control". *Comm. ACM*, Vol. 9, No. 5 (May); 321-322.

Patil S.S. 1971. "Limitations and Capabilities of Dijkstra's Primitives for Coordination among Processes". *Project MAC Computational Structures Group*, Memo 57.

Peterson G.L. 1981. "Myths about the mutual exclusion problem". *Information Processing Letters*, North Holland, Vol. 12, No. 3 (June); 115-116.

Peterson J.L. and A. Silberschatz. 1989. *Operating System Concepts*. Addison-Wesley, Reading, Mass., alternate edition.

Raynal M. 1986. *Algorithms for Mutual Exclusion*. North Oxford Academic.

Scheuer A. and W. Ecker. 1992. "Semaphores in HW-design". *Proc. IFIP*, VHDL-forum, Santander, Spain; 147-153