External Module Interface EMI

Interconnect of hardware and algorithmic programming level using abstract object types and methods

1. DESCRIPTION

The external module interface provides an object orientated programming model and interface of hardware blocks modelled on hardware behavioural level using a modified subset of VHDL.

2. TABLE OF CONTENT

3. INTRODUCTION

The External Module Interface (EMI) is used to connect and interface hardware blocks modelled on behavioural hardware level with the ConPro process and abstract object framework on algorithmic programming level.
The purpose of the EMI module interface is to embed and connect external VHDL coded components directly into a ConPro implementation with direct access from the ConPro programming level (process- and top-level) using the Abstract Data Type Object (ADTO) interface and method calls applied to those objects. In contrast to external VHDL components (ConPro component interface) requiring signal objects for interconnection with ConPro modules, in this case VHDL blocks are accessed and linked invisible and transparently to the programming level with the ConPro ADTO interface.
An EMI module can be opened and compiled using the `open` statement.
The EMI module is splitted in the access and implementation part of an abstract object. The EMI module file (file suffix `.mod`) defines

1. all methods available for object access,

2. the method access, on ConPro process level defining data and control path parts and the implementation (scheduling) of the object access, too,
3. all required signals for process interconnect and implementation,
4. the implementation of the abstract object using VHDL processes, at least the object access scheduler.

Each EMI module defines a new abstract object type. Objects can be created from this new type using the `object` statement.
The EMI language is a modified subset of VHDL on behavioural (and structural) level with embedded interpreter statements evaluated during synthesis, targeting the ConPro programming language level (ADTO interface).

4. EMI STRUCTURE

An EMI module file is divided into several sections, where each section consists of a section header and a section body. Each section has a class identifier starting with the # character. Additionally there are labeled sections with a specified name label. Sections can be conditional and are evaluated depending on boolean expressions using EMI parameters.
Overview:

#parameter
Declaration and definition of EMI module parameters.

#methods
Declaration of EMI object methods (type signature).

#access
Definition of EMI object access on hardware level (request, reply, and acknowldge interaction by ConPro processes during method call).

#interface
Defines signals required in VHDL port for object access.

#mapping
Defines signal mapping required on toplevel for object access (toplevel ConPro process interconnect).

#signals
Declaration of hardware signals required for implementation of EMI objects.

#process
Definition of VHDL hardware processes required for implementation of EMI objects .

5. PARAMETER SECTION

Name

```
#parameter
```

Syntax

```
#parameter
begin
  $name;          -- (1)
  $name <= m;     -- (2)
  $name[n1,n2,n3,...] <= m;  -- (3)
  $name[a to b] <= m;  -- (4)
  ...
end;
```

Description

This is the paramter section. Parameters can be used inside the EMI module file. New values can be assigned either on object creation or using method calls (set method class). Parameters are either scalar or vector (array/ list) types.
This section defines the parameter variables used in an external module interface definition. Different forms of parameter definitions are provided:

(1) Giving only the parameter name preceeded by the `$` character defines a variable without any default and initialized value. On object instantiation the parameter must be assigned a value otherwise an error occurs during synthesis, or using the set class method alternatively.

(2) In this case the parameter variable gets a default value. Parameter value assignment on object instantiation is optional.

(3) In this case the parameter variable gets a default value. Parameter value assignment on object instantiation is optional. Additional a set of allowed values is included in paranthesises after the parameter name.

(4) In this case the parameter variable gets a default value. Parameter value assignment on object instantiation is optional. Additional a range of allowed values is included in paranthesises after the parameter name.

Summary

Table 1. Parameter section

Syntax	Description
`#parameter begin ... end;`	Definition of a parameter section
`$P;`	Definition of a parameter with polymorph type.
`$P[A,B,..];`	Definition of a parameter with a set of possible values (sum type, elements *A*,*B*,..).
`$P <= N;`	Definition of parameter with initialization (type derived from value *N*).

Example

```
#parameter
begin
  $datawidth[8,10,12,14,16] <= 8;
  $seed <= 0xffff;
  $arch001["fifo","static"] <= "fifo";
end;
```

6. METHODS SECTION

Name

`#methods`

Syntax

```
#methods
begin
  name(exprside:datatype [,exprside:datatype]);
  ...
end;

exprside ::= #lhs | #rhs | #lrhs
datatype ::= logic | logic[width] |
             int[width] | bool | natural
```

Description

This is the method programming interface declaration section of the EMI module file. This section defines all exported and accessible methods, specifying the method name and the method call parameter type declaration:

1. the way the argument objects are used during method call, either on left-hand-side (LHS) or right-hand-side (RHS) (or both) of an expression, meaning read or write access respectively of the object used as an argument,
2. the expected data type of the argument object, though width scaling of the actual argument, used in method call, to the expected method paramter, used in the EMI-ADTO implementation, is performed by the EMI compiler.

If a method doesn't expect an argument, an empty paranthesis pair `()` is used in the definition. Up to 9 method call parameters can be specified.

Example

```
#parameter
begin
  $datawidth[8 to 16] <= 8;
  $addrwidth <= 16;
end;

#methods
begin
  init();
  read(#lhs:logic[12]);
  time(#rhs:natural);
  read2(#lhs:logic[$datawidth],#rhs:logic[$addrwidth]);
end;
```

7. INTERFACE SECTION

Name

```
#interface
```

Syntax

```
#interface
begin
  signal name_$O : dir datatype;     -- (1)

  foreach $p in $P do                 -- (2)
  begin
    signal name_$O : dir datatype;
    ...
  end;

  foreach $p in $P.meth do            -- (3)
  begin
```

```
    signal name_$O : dir datatype;
    ...
  end;

  ...
end;
dir ::= in | out | inout
datatype ::= logic | logic[width] |
             int[width] | bool |
             std_logic | std_logic_vector[width] |
             signed[width]
```

Description

ConPro-Process-Level

This interface section defines the part of the VHDL component port interface required for the hardware implementation of ConPro processes accessing methods of this ADT object. ConPro processes are synthesized to VHDL components and RTL using a finite-state-machine (FSM). The ConPro module toplevel is synthesized and mapped to a VHDL component, too, implementing ConPro toplevel objects. Additionally, the synthesized ConPro-process VHDL components are structurally connected on this module level. An abstract object access requires data and control signals, routed up from the ConPro process level to the ConPro module level where the abstract object is implemented, except of abstract objects defined locally on ConPro process level.

There are two ways for adding process port signals:

(1) Generic signals independent on a particular method access. These signals are added to the VHDL process port for each process using the abstract object. The signal name can contain the object name variable `$O`.

(2) Signals depending on ConPro processes accessing this object and a specific method. The signal name can contain the object name variable `$O`. The signal definifion adds the signal only for ConPro processes using this object method.

The signal port direction and the signal type must be specified.
Supported signal directions are:

in
The related process reads from this signal.

out
The related process writes to this signal.

inout
The related process both reads from and writes to the signal. This is the bidirectional bus behaviour.

Supported signal types are aligned to the core ConPro data type system, and they are:

logic
ConPro `logic` data type, width 1 bit, mapped in general to the VHDL `std_logic type`.

logic[n]
ConPro `logic` data type, width n bit, index range is in general `[n-1 downto 0]`, mapped in general to the VHDL `std_logic_vector(n-1 downto 0)` type.

int[n]
ConPro signed integer (`int`) data type, width n bit, index range is in general `[n-1 downto 0]`, mapped in general to the VHDL `signed(n-1 downto 0)` type.

bool
ConPro boolean (bool) data type, mapped in general to the VHDL `std_logic` type.

std_logic
VHDL `std_logic type`

std_logic_vector[n]
VHDL `std_logic_vector(n-1 downto 0)` type. Index direction and range depends also on ConPro synthesis settings.

signed[n]
VHDL `signed(n-1 downto 0)` type. Index direction and range depends also on ConPro synthesis settings.

Example

```
#interface
begin
  foreach $p in $P.read do
  begin
    signal F_$O_RE: out std_logic;
    signal F_$O_RD: in std_logic_vector[$datawidth];
  end;
  foreach $p in $P.init do
  begin
    signal F_$O_INIT: out std_logic;
  end;
  foreach $p in $P do
  begin
    signal F_$O_GD: in std_logic;
  end;
end;
```

8. MAPPING SECTION

Name

    #mapping

Syntax

```
#mapping
begin
  foreach $p in $P do        -- (1)
  begin
    signame_$O => signame_$O_$p;
  end;
  foreach $p in $P.meth do   -- (2)
  begin
    signame_$O => signame_$O_$p;
  end;
  ...
end;
```

Description

ConPro-Module-Level

This mapping section defines the part of the VHDL component port mapping required for toplevel interconnect of ConPro processes accessing methods of this ADT object. ConPro processes are synthesized to VHDL components and RTL and a finite-state-machine (FSM). The ConPro module toplevel is synthesized and mapped to a VHDL component, too, implementing ConPro toplevel objects. Additionally, the synthesized ConPro-process VHDL-components are structurally connected and mapped on this module level. An abstract object access requires data and control signals, routed up from the ConPro-process level to the ConPro-module level where the abstract object is implemented, except of abstract objects defined on ConPro process level. On the left hand side there is the (local) ConPro-process context level (VHDL entity port interface) signal, on the right hand side there is the (global) ConPro-module context level signal (VHDL component port mapping on instantiation). The object name $O and process name $P are replaced respectively. All signal mappings appearing in this section must be defined in the `#interface` section.

(1) The signal mapping is applied to each process accessing this object.

(2) The signal mapping is applied to each process accessing this object with a specified method.

Example

```
#mapping
begin
  foreach $P.read do
  begin
    F_$O_RE => F_$O_$P_RE;
    F_$O_RD => F_$O_$P_RD;
  end;
  foreach $P.init do
```

Last modified on October 15, 2010

```
  begin
    F_$O_INIT => F_$O_$P_INIT;
  end;
  foreach $P do
  begin
    F_$O_GD => F_$O_$P_GD;
  end;
end;
```

9. ACCESS SECTION

Name

`#access`

Syntax

```
method:#access
begin
  #data
  begin
   signame_$O <= expr1 when $ACC else expr0;  -- (1)
   $ARG# <= signame_$O when $ACC else expr0; -- (1b)
   ...
  end;

  #control
  begin
   null;  -- (2)
   wait for cond-expr;     -- (3)
  end;

  #set
  begin
   $param <= $ARG#;        -- (4)
   ...
  end;
end;
```

Description

ConPro-Process-Level

ConPro processes are synthesized to VHDL components and RTL and a finite-state-machine (FSM). The ConPro module toplevel is synthesized and mapped to a VHDL component, too, implementing ConPro toplevel objects. Additionally, the synthesized ConPro-process VHDL-components are structurally connected and mapped on this module level. An abstract object access requires data and control signals, routed up from the ConPro-process level to the ConPro-module level where the abstract object is implemented,

except of abstract objects defined on ConPro process level.For each method defined in the `#methods` section there is an access definition.
An access definition consists of the data and control path of a ConPro-process defined in subsections #data and #control respectively. There are methods only required for setting object parameters on toplevel. In this case the #set subsection is used instead, but can be used additionally to data and control path sections.
The data path defines expression assignments 1. of local signals, 2. of values to object access signals defined in the `#interface` section, or 3. alternatively assigning these object access signals to a method call argument. The method call arguments are related with the variables `$ARG1, $ARG2, $ARG3...` for the first, the second, the third ... method call argument.
The control path is used to suspend the ConPro process control state machine (calling this method) untill a condition is satisfied, mainly the object guard.

(1) Expression `expr1` is assigned to the LHS signal during access, expression `expr0` otherwise. Independent of the data type of the LHS, expr0 can be the natural number 0. The data type of the LHS object is determined automatically in this case, and hence the VHDL value to be assigned, too.
The LHS is an object access signal. The RHS can be an object access signal, a method call argument or a constant value.

(1b) An object access signal or a constant value is assigned to the method call argument `$ARG#`. The variable `$ARG#` is substituted with the actual argument signal name. Control signals required for the method argument acces (like the write enable signal) are generated automatically by the synthesis compiler!

(2) There is no control path statement. Object access never blocks control path and consumes exactly on time unit. Else case (3) must be applied:

(3) The method access blocks the control path of the calling ConPro process untill condition cond-expr is satisfied. Else case (2) must be applied. Usually the object guard signal is used for blocking:

```
signame_$o_GD = '0';
```

(4) The actual argument value is assigned to the parameter variable on the LHS. This is a method call statement used only for configuration of the object, either on toplevel outside processes or inisde a process! Either an integer value can be assigned or a data object can be imported and accessed inside the ADTO implementation (actually only signals and registers with read access only).

Environment variables are always array types. Each time the set subsection is used to assign argument values to an environment variable, the actual value is added to this array. Therefore some array functions exists which can be used in expressions. For example it is desired to work with different baud rates of a serial communication link, and the baud rate should be changeable during runtime. In this case it is not usefull to pass the original baud rate value eacht time a aspecified set method occurs in ConPro-processes, it is more likely to pass an index value requiring much less bits to each method call. Here is an example to implement such an method access (time) for the case of a timer (using the environment variable $time):

```
time: #access
begin
  #set
  begin
    $time <= $arg1;
  end;
  #data
  begin
    TIMER_$O_TIME_SET <= '1' when $ACC else '0';
    TIMER_$O_TIME <= #index($time,$ARG1) when $ACC else 0;
  end;
  #control
  begin
    wait for TIMER_$O_GD = '0';
  end;
end;
```

Example

```
init: #access
begin
  #data
  begin
    F_$O_INIT <= '1' when $ACC else '0';
  end;
  #control
  begin
    null;
  end;
end;

read: #access
begin
  #data
  begin
    F_$O_RE <= '1' when $ACC else '0';
    $ARG1 <= F_$O_RD when $ACC else 0;
  end;
  #control
  begin
    wait for F_$O_GD = '0';
  end;
end;

time: #access
```

```
begin
  #set
  begin
    $time <= $ARG1;
  end;
end;
```

10. SIGNALS SECTION

Name

`#signals`

Syntax

```
#signals   [(cond)]
begin
  signal $signame_$O : datatype;          -- (1)
  ...
  foreach $p in $P do
  begin
    signal $signame_$O_$p : datatype;     -- (2)
    ...
  end;
  foreach $p in $P.meth do
  begin
    signal $signame_$O_$p : datatype;     -- (3)
    ...
  end;
  type typname is {                       -- (4)
    el1;
    el2;
    ...
  };
  type typname array[range]               -- (5)
      of datatype;
  ...
end;

datatype ::= 'logic' | 'logic[' width ']' |
             'int[' width ']' | bool |
          'std_logic' | 'std_logic_vector[' width ']' |
          'signed[' width ']'
cond ::= '$' param '=' value [ 'and' '$' param '=' value...]
range ::= a 'to' b | a 'downto' b | size
```

Description

ConPro-Module-Level

This section defines VHDL signals required for object implementation on

global module level, and data and control signals required for method access from ConPro processes. Remember the ConPro system hierarchy: there is a process level and a module level containing processes. Each ConPro process is synthesized into a VHDL component entity. A ConPro module is also synthesized into a VHDL component entity, providing the interconnections for all contained ConPro processes. Each process accessing this ADT object requires it own set of data and control signals.
There can be exist more than one signals section. There are unconditional (the usual case) and conditional signals sections, only applied if parameter conditions are satisfied.
There are three different signal classes:

(1) Generic signals independent on ConPro processes and method access, mainly implementation dependent.

(2) Signals depending on ConPro processes accessing this object. The signal name can contain the ConPro process name variable `$P` (array, foreach statement required) and the object name variable `$O`. The signal definifion adds for each ConPro process accessing this object the specified signal, the process variable is replaced by the related process name.

(3) Signals depending on ConPro processes and method access. The signal name can contain the ConPro process name variable `$P` (array, foreach statement required) and the object name variable `$O`. The signal definifion adds for each ConPro process accessing this object and applying the specified method the specified signal, the process variable is replaced by the related process name.

(4) Definition of an enumerated symbolic type.

(5) Definition of an array type.

Supported signal types are aligned to the core ConPro data type system, and they are:

logic
ConPro `logic` data type, width 1 bit, mapped in general to the VHDL std_logic type.

logic[n]
ConPro `logic` data type, width n bit, index range is in general [n-1 downto 0], mapped in general to the VHDL std_logic_vector(n-1 downto 0) type.

int[n]
ConPro signed integer (`int`) data type, width n bit, index range is in general `[n-1 downto 0]`, mapped in general to the VHDL `signed(n-1 downto 0)` type.

bool
ConPro boolean (bool) data type, mapped in general to the VHDL

`std_logic` type.

std_logic
VHDL `std_logic type`

std_logic_vector[n]
VHDL `std_logic_vector(n-1 downto 0)` type. Index direction and range depends also on ConPro synthesis settings.

signed[n]
VHDL `signed(n-1 downto 0)` type. Index direction and range depends also on ConPro synthesis settings.

Example

```
#signals
begin
  --
  -- Implementation signals
  --
  signal F_$O_d_in: std_logic;
  signal F_$O_data_shift: std_logic_vector[$datawidth];
  signal F_$O_data: std_logic_vector[$datawidth*2-1];
  signal F_$O_shift: std_logic;
  signal F_$O_init: std_logic;
  signal F_$O_avail: std_logic;

  foreach $p in $P.read do
  begin
    signal F_$O_$p_RE: std_logic;
    signal F_$O_$p_RD: std_logic_vector[$datawidth];
  end;

  foreach $p in $P.init do
  begin
    signal F_$O_$p_INIT: std_logic;
  end;

  foreach $p in $P do
  begin
    signal F_$O_$p_GD: std_logic;
  end;
end;

#signals ($datawidth=8)
begin
  signal F_$O_count: std_logic_vector[3];
end;

#signals ($datawidth=10)
begin
```

```
  signal F_$O_count: std_logic_vector[4];
end;
```

11. PROCESS SECTION

Name

`#process`

Syntax

```
procname:#process   [(cond)]
begin
  statement;
  statement;
  ...
end;

cond ::= '$' param '=' value ['and' '$' param '=' value...]
```

Description

ConPro-Module-Level

This section defines a named VHDL hardware process (procname) required for the implementation of an object on hardwae behaviour level. There can be several process sections, each defining one process appearing on ConPro-module level, or in some limited cases on ConPro-process level iff the object has only a ConPro process local context and was defined inside a ConPro process. A VHDL hardware process implementation can be conditional (cond).

At least one process must exist for the object implementation: the access scheduler guarding the (usually) shared object. Several ConPro processes can access a shared object, therefore some kind of mutual exclusion lock must be implemented. The main object implementation, modelling the behaviour of this ADTO, is usually modelled within a separate VHDL process definition.

Parameter variables are extensively used inside the VHDL hardware process definition:

$CLK

This parameter is used inside conditional expressions and is only true if there is a system clock event. The clock edge is determined by the ConPro program and compiler settings, and expands to VHDL:

```
$CLK => conpro_system_clk'event and conpro_system_clk = '1' --
    rising edge
$CLK => conpro_system_clk'event and conpro_system_clk = '0' --
    falling edge
```

The clock signals are already defined and may not be defined in the `#signals` section.

$RES

This parameter is used inside conditional expressions and is only true if the system reset is active. The active reset signal logic level is determined by the ConPro program and compiler settings, and expands to VHDL:

```
$RES => conpro_system_reset = '1'
$RES => conpro_system_reset = '0'
```

The reset signals are already defined and may not be defined in the `#signals` section.

$myreg

ConPro data objects (actually only signals) can be imported into an object module. For example a module implements a bus interface, than external bus signals must be imported. They are attached to a module parameter $myreg (or any other name excpet reserved parameters) using the access set method, defined in the `#access` section.

```
EMI:
  #methods
  begin
    set(#rhs:logic[8]);
  end;
  set:#access
  begin
    #set:
    begin
      $myreg <= $ARG1;
    end;
  end;
  #process
  begin
    s <= $myreg;
    $myreg <= s;
  end;
...
ConPro:
  myobj.set(sigx1y);

VHDL:
  s <= $myreg; => s <= sigx1y_RD;
  $myreg <= s; => sigx1y_WR <= s;
```

The imported signals are already defined and may not be defined in the `#signals` section.

The sensivity list of the VHDL process is computed automatically.

VHDL Subset

Only a subset of VHDL is supported, and there are some adjustments on syntax level to the ConPro programming language.

if-then-else

Syntax is slightly modified. A group of statements requires block environment begin-end.

```
if expr then statement ;
if expr then statement else statement;
statement ::= single-statement | 'begin' statement-list 'end'
```

if-then-else-cascade

Either modelled using the elsif VHDL or else if ConPro construct or modelled with the sequence construct.

```
if expr then statement else if expr then statement ...;
if expr then statement elsif expr then statement ...;
statement ::= single-statement | 'begin' statement-list 'end'

sequence
begin
  if expr then statement;
  if expr then statement;
  ...
  foreach $p in $P do
  begin
    if expr then statement;
    ...
  end;
  foreach $p in $P.meth do
  begin
    if expr then statement;
    ...
  end;
  if others then statement;
end;
```

The sequence is expanded to a if-then-elsif cascade. The last case (optional) is the default case if no other conditional expression can be applied.

Example:

```
sequence
begin
  if a = '1' then s <= 0;
  if b = '1' then s <= 2;
  foreach $p in $P.init do
  begin
    if c_$p = '1' then s <= 3;
  end;
  if others then s <= 4;
end;
```

=> expands to =>

```
if a = '1' then s <= 0
elsif b = '1' then s <= 2
elsif c_p1 = '1' then s <= 3
elsif c_p2 = '1' then s <= 3
else s <= 4;
```

During synthesis, conditional expressions, containing only constant values and environment variables, are tried to be evaluated to constant values. Depending on the result either the true or the false case statements are replaced by the conditional statement.

case

Syntax is slightly modified and aligned to the ConPro programming language. A group of statements requires block environment begin-end.

```
case expr is
begin
  when val1 : statement;
  when val2 : statement;
  ...
  when others : statement;
end;
statement ::= single-statement | 'begin' statement-list 'end' |
              'null'
```

for

Syntax is slightly modified and aligned to the ConPro programming language. A group of statements requires block environment begin-end. The loop variable i can be used in expressions within the loop body. Environment array variables (preceeded with a $) can be iterated in a foreach-loop, too.

```
for i = expr dir expr do
  statement;
foreach $i in $array do
  statement;
dir ::= 'to' | 'downto'
statement = single-statement | 'begin' statement-list 'end'
```

constant values

Syntax is slightly modified:

Table 2. Constant Values

Value	Format

Integer	*DDDD* with *D*={0,1,..,9}
	0x*XXX* with *X*={0,1,...9,A,..,F}

Bit	'*B*' with *B*={0,1,Z,H,L,x} 0b*B*

Bit vector	0b*BBB* with *B*={0,1,Z,H,L}
	0x*XXX* with *X*={0,..,F}

process variables

VHDL process variables are defined at the beginning of the process section body:

```
#process:
begin
  variable vname: datatype;
  ...
end;
```

Expressions

VHDL expressions can contain any VHDL operator (arithmetic, logic, relational, boolean), vector subranges [] and the object selector '''.

```
+ - * / ...
< > = /= <= >=
and or xor ...
obj[range]
obj'sel
range ::=  a 'to' b | a 'downto' b
```

Process Access and Scheduler

Access of ADT objects requires control and data signals. In the case of data based objects (for example a queue or RAM), ConPro method calls activate control signals, and either write to or read from data signals, commonly:

Control Signals

```
T_$O_RE: Read Request Enable
T_$O_WE: Write Request Enable
T_$O_GD: Object Guard
```

Data Signals

```
T_$O_RD: Read Data Signal Vector
T_$O_WR: Write Data Signal Vector
```

In the case of pure control objects (for example a semaphore), only control signals are activated. Of course for special purpose objects different signals

are required.
Because several ConPro processes can access a shared object, access serialization and blocking of method caller processes are required. If there is actually already an object access, method call from other processes must be blocked untill the reosurce is available. For this purpose the gurad signal is used. As long as the signal is in state '1', the calling process FSM will be blocked.

Warning and Limitations

Only a subset of VHDL is supported, and there are some adjustments on syntax level to the ConPro programming language. Mainly, the EMI-language is context free. That means that function call arguments are enclosed in round paranthesis, thereby range expressions (both in signal declarations and within expressions) are enclosed in bracket paranthesis!

Example

```
TIMER_$O_SCHED: #process
begin
  if $CLK then
  begin
    if $RES then
    begin
      TIMER_$O_ENABLED <= '0';
      TIMER_$O_MODE <= '0';
      TIMER_$O_COUNTER <=
       to_logic(0,width((max($time)*$clock)/1000000000));
      TIMER_$O_COUNT <=
       to_logic((nth($time,1)*$clock)/1000000000,
                width((max($time)*$clock)/1000000000));

      foreach $p in $P do
      begin
        TIMER_$O_$p_GD <= '1';
      end;
      foreach $p in $P.await do
      begin
        TIMER_$O_$p_LOCKed <= '0';
      end;
    end
    else
    begin
      foreach $p in $P do
      begin
        TIMER_$O_$p_GD <= '1';
      end;
      if $arch002 = 2 then
      begin
        if TIMER_$O_ENABLED = '1' then
        begin
      if TIMER_$O_COUNTER =
         to_logic(0,width((max($time)*$clock)/1000000000))
      then
      begin
        foreach $p in $P.await do
        begin
          if TIMER_$O_$p_LOCKed = '1' then
          begin
            TIMER_$O_$p_LOCKed <= '0';
            TIMER_$O_$p_GD <= '0';
          end;
        end;
        if TIMER_$O_MODE = '0' then
        begin
          TIMER_$O_COUNTER <= TIMER_$O_COUNT;
        end
        else
          TIMER_$O_ENABLED <= '0';
      end
      else
      begin
        TIMER_$O_COUNTER <= TIMER_$O_COUNTER - 1;
      end;
    end;
  end;

  sequence
  begin
    foreach $p in $P.init do
    begin
      if TIMER_$O_$p_INIT = '1' then
      begin
        TIMER_$O_COUNTER <=
          to_logic(0,width((max($time)*$clock)/1000000000));
        TIMER_$O_COUNT <= to_logic((nth($tim
    if $arch001 = 1 then
    begin
      foreach $p in $P.await do
      begin
        if TIMER_$O_$p_AWAIT = '1' and TIMER_$O_$p_LOCKed  = '0' then
        begin
          TIMER_$O_$p_LOCKed <= '1';
        end;
      end;
    end;
    if $arch001 = 2 then
    begin
      if expand($P.await,$p,or,TIME_$O_$p_AWAIT = '1' and
                TIMER_$O_$p_LOCKed = '0') then
      begin
```

```
            foreach $p in $P.await do
            begin
              if TIMER_$O_$p_AWAIT = '1' then
              begin
                TIMER_$O_$p_LOCKed <= '1';
              end;
            end;
          end;
        end;
        foreach $p in $P.start do
        begin
          if TIMER_$O_$p_START = '1'  then
          begin
            TIMER_$O_COUNTER <= TIMER_$O_COUNT;
            TIMER_$O_ENABLED <= '1';
            TIMER_$O_$p_GD <= '0';
          end;
        end;
        foreach $p in $P.stop do
        begin
          if TIMER_$O_$p_STOP = '1'  then
          begin
            TIMER_$O_COUNTER <=
              to_logic(0,width((max($time)*$clock)/1000000000));
            TIMER_$O_ENABLED <= '0';
            TIMER_$O_$p_GD <= '0';
          end;
        end;
        foreach $p in $P.time do
        begin
          if TIMER_$O_$p_TIME_SET = '1'  then
          begin
            TIMER_$O_$p_GD <= '0';
            sequence
            begin
              foreach $this_time in $time do
              begin
                if TIMER_$O_$p_TIME = index($time,$this_time) then
                  TIMER_$O_COUNT <=
                   to_logic(($this_time*$clock)/1000000000,
                            width((max($time)*$clock)/1000000000));
              end;
            end;
          end;
        end;
        foreach $p in $P.mode do
        begin
          if TIMER_$O_$p_MODE_SET = '1'  then
          begin
            TIMER_$O_$p_GD <= '0';
          TIMER_$O_MODE <= TIMER_$O_$p_MODE;
        end;
      end;
      if $arch002 = 1 then
      begin
        if others then
        begin
          if TIMER_$O_ENABLED = '1' then
          begin
            if TIMER_$O_COUNTER =
               to_logic(0,width((max($time)*$clock)/1000000000)) then
            begin
              foreach $p in $P.await do
              begin
                if TIMER_$O_$p_LOCKed = '1' then
                begin
                  TIMER_$O_$p_LOCKed <= '0';
                  TIMER_$O_$p_GD <= '0';
                end;
              end;
              if TIMER_$O_MODE = '0' then
               begin
                 TIMER_$O_COUNTER <= TIMER_$O_COUNT;
               end
               else
                 TIMER_$O_ENABLED <= '0';
            end
           else
            begin
              TIMER_$O_COUNTER <= TIMER_$O_COUNTER - 1;
            end;
          end;
        end;
      end;
    end;
  end;
end;
end;
```

12. ENVIRONMENT

The environment of a EMI module consists of a set of environment variables.

Environment Variables

VHDL expressions can contain environment variables, those names are preceeded by a $, both on left-hand- and right-hand-side of expressions. Usually values are assigned in `#parameter` sections, during object creation and within `#set` subsections of `#access` sections. Environment variables are always of array type. That means each time a new value is assigned to an environment variable, a new array element is created an appended. A scalar read access of a environment variable returns the top of the array (the last element stored). There are several builtin functions to access array elements.

$name
This is the scalar read operation of a environment variable and returns the first element of the array.

size($array)
Returns number of array elements actually stored in the array.

width($array)
Returns the number of bits required for the encoding of maximum value in the specified array, assuming weighted binary encoding.

index_width($array)
Returns the number of bits required for the encoding of the index of the specified array, assuming weighted binary encoding.

index($array,value)
Returns the binary encoded index selector for the specified (unique) value element contained in the specified array.

nth($array,index)
Returns the n-th element given by index of the specified array.

min($array)
Returns the minimum element from the specified array.

max($array)
Returns the maximum element from the specified array.

Iteration

There is a generic loop construct for the iteration of environment variable arrays:

```
foreach $a in $A do
begin
 ... $a ...
end;
```

The loop variable `$a`, set to an element of the variable array `$a`, can be used in expressions and concatenated names inside the loop body.
Abstract objects are accessed by different ConPro processes. There is a special environmen tvariabl `$P` which holds informations about all processes accessing a particular EMI object. This array (which is indeed a set of arrays with each array related to a particular method) can be used in loop iterations, too:

```
foreach $p in $P do
begin
 ... $a ...
 ... X_$a_yyy ...
```

```
end;
foreach $p in $P.meth [or $P.meth2 ...] do
begin
 ... $p ...
 ... O_$p_yyy ...
end;
```

The iteration set can be constructed from different subsets using boolean `or` and `and` operators.
In expressions a set or subset of array elements can be expanded using the expand operator:

```
if expand ($P.meth,$p,or,OO_$p_YYY = ‘1’) then
...
```

This application of the expand operator results in an expression or’ing the last expression argument by iterating all array elements of the subset `$P.meth`.

Printing

During object synthesis informational text lines can be printed to the standard output channel using the `print` function. The print function prints a list of arguments to the standard output channel. The arguments can contain expressions and environment variables.

```
#print("Achieved baud rate accuracy [bit/s]: ",
       "[actual = ",$clock / (($clock / (16 * $ARG1))*16),"] ",
       "[requested = ",$ARG1,"] ",
       "[error = ",((($clock / (($clock /
                  (16 * $ARG1))*16))*1000)/$ARG1)-1000,
                " %%]" );
```