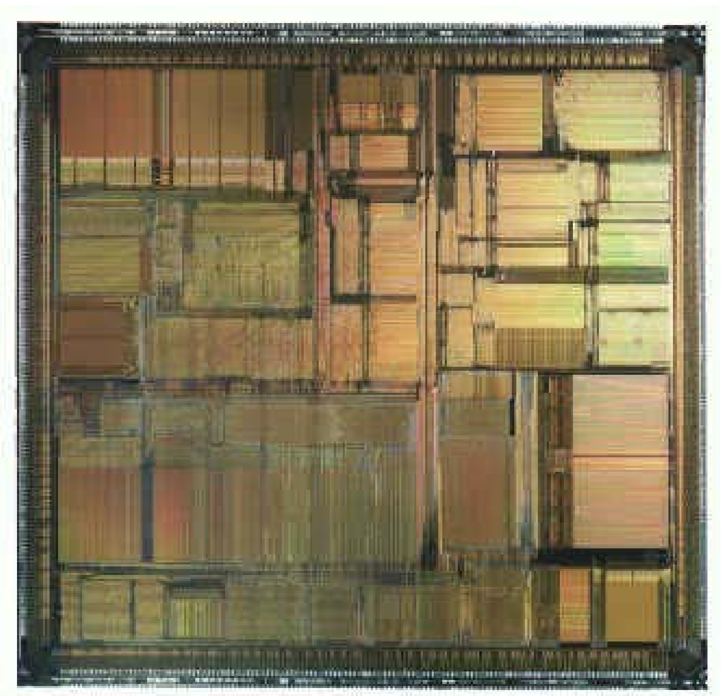
Conpro^2 High-Level Synthesis Framework

Stefan Bosse

Programming of Concurrent Hardware and Software Systems

System-On-Chip Design



Introduction

Design of Embedded Systems and programming of concurrent Hardware and Sofware Systems using a Multi-process model and Interprocess communication

1. Summary

Embedded Systems used for control and data processing, for example in Cyber-Physical-Systems (CPS), perform the monitoring and control of complex physical processes using applications running on dedicated execution platforms in a resource-constrained manner. Application-specific System-On-Chip (SoC) designs providing the execution platform have advantages compared with traditionally used program-controlled multi-processor architectures.
SoC designs can be modelled on structural and behavioural level. The behavioural level is generally a more sophisticated modelling level. In the context of CPS, these are mainly reactive systems with dominant and complex control paths. The major contribution to concurrency appears on control path level.
A new SoC design methodology is presented using the behavioural hardware compiler ConPro providing an imperative programming model based on concurrently communicating sequential processes (CSP) with an extensive set of interprocess-communication primitives and guarded atomic actions. The behavioural programming level describes the behaviour of the full design interacting with the environment using processes and (shared) objects. The programming language and the compiler-based synthesis process enables the design of constrained power- and resource-aware embedded systems with pure Register-Transfer-Logic efficiently mapped to FPGA and ASIC technologies. Concurrency is modelled explicitly on control- and datapath level. Additionally, concurrency on datapath level can be explored and optimized automatically by different schedulers.
The CSP programming model can be synthesized to different levels, not only used for hardware circuit synthesis: software models (C, ML), intermediate µCode, RTL state level, and finally VHDL. A common source on programming level for both hardware and software implementations with identical functional behaviour is used. The necessary abstraction of hardware blocks and IPC is per-

formed by using an object-orientated model layer, part of the programming model. The presented design methodology has the benefit of high modularity, freedom of choice of target technologies, and system architecture. Algorithms can be well matched to different suitable execution platforms and implementation technologies, using a unique programming model, providing a balance of concurrency and resource complexity.

2. Introduction and Overview

Embedded Systems used for control, for example in Cyber-Physical-Systems (CPS), perform the monitoring and control of complex physical processes using applications running on dedicated execution platforms in a resource-constrained manner. System-On-Chip designs are preferred for high miniaturization and low-power applications. Traditionally, program-controlled multi-processor architectures are used to provide the execution platform, but application-specific digital logic gains more importance.
There are two different ways to model and implement System-on-Chip-Designs (SoC) used in those embedded systems: using 1. a structural and/or 2. a behavioural level. The structural level decomposes a SoC into independent sub-modules - processor cores (or data processing units in general), memories, and peripherials - interacting with each other using centralized or distributed networks and communication protocols. The behavioural level usually describes the behaviour of the full design interacting with the environment without detailed assumptions about system architecture, generally a more sophisticated modelling level. In the context of CPS, these are mainly reactive systems with dominant and complex control paths. The major contribution to concurrency appears on control path level, which can be explicitly modelled on algorithmic level.
A new SoC-design methodology is presented using the behavioural hardware compiler ConPro providing an imperative programming model based on concurrently communicating sequential processes (CSP) **[5]** and guarded atomic actions **[4]** with an extensive set of interprocess-communication primitives. The programming language and the compiler-based synthesis flow enables the design of application-specific constrained power- and resource-aware embedded systems on Register-Transfer-Level efficiently mapped to FPGA and ASIC technologies. Concurrency is modelled explicitly on control- and datapath level. Additionally, concurrency on data path level can be explored and optimized automatically by different schedulers.
Hardware blocks (including IPC and externally modelled) can be accessed transparently from programming level with a generic object-orientated approach.
The CSP programming model can be synthesized to different other levels, not only used for hardware circuit synthesis: software models (C, ML), intermediate μCode, RT state level, and finally to hardware behaviour level, e.g. VHDL. A common source for both hardware and software implementation with identical

functional behaviour matches different embedded architecture levels and enables code reuse. The Metalanguage ML (OCaML) is well suited for simulation and test-pattern based functional model checking.

Why a new language? Traditional programming languages like C are designed for sequential programming only, and concurrency is present to some extent through the use of libraries **[1]**. Concurrency should be controlled by first-class language constructs **[3]** to enable optimized design of massive parallel systems and hardware synthesis. There are several examples of new designed languages for concurrent programming, like SystemJ **[1]** or X10 **[3]**. C-like languages used for hardware-synthesis are wide spread, but are not fully suitable for RTL synthesis due to strong dependency on memory model (pointers) and the missing concurrency model.

What is novel compared with other high-level-synthesis approaches? One language targets both concurrent software and hardware programming, the hardware synthesis process can be fine-grained controlled on programming level using parameterized blocks. A traditional compiler approach with μCode intermeadiate representation (without loss of concurrency) enables fast and optimized synthesis. Object-orientated access of hardware blocks using the External Module Interface (EMI) - part of the programming model - provides a modern and transparent interface for both software and hardware designers, closing the gap between software and hardware models. The extended set of IPC primitives enables concurrent programming of complex control and data processing systems.

3. System-On-Chip Design Using a Behavioural Model Approach and High-Level Synthesis

Concurrency has great impact on system and data processing behaviour concerning latency, data throughput, and power consumption. Streaming and functional data processing requires fine-grained concurrency (on data path level), however, reactive control systems (for example communication) require coarse-grained concurrency (on control path level).

The **structural level** decomposes a SoC into independent submodules interacting with each other using centralized or distributed networks and communication protocols, mainly program-controlled multi-processor architectures.

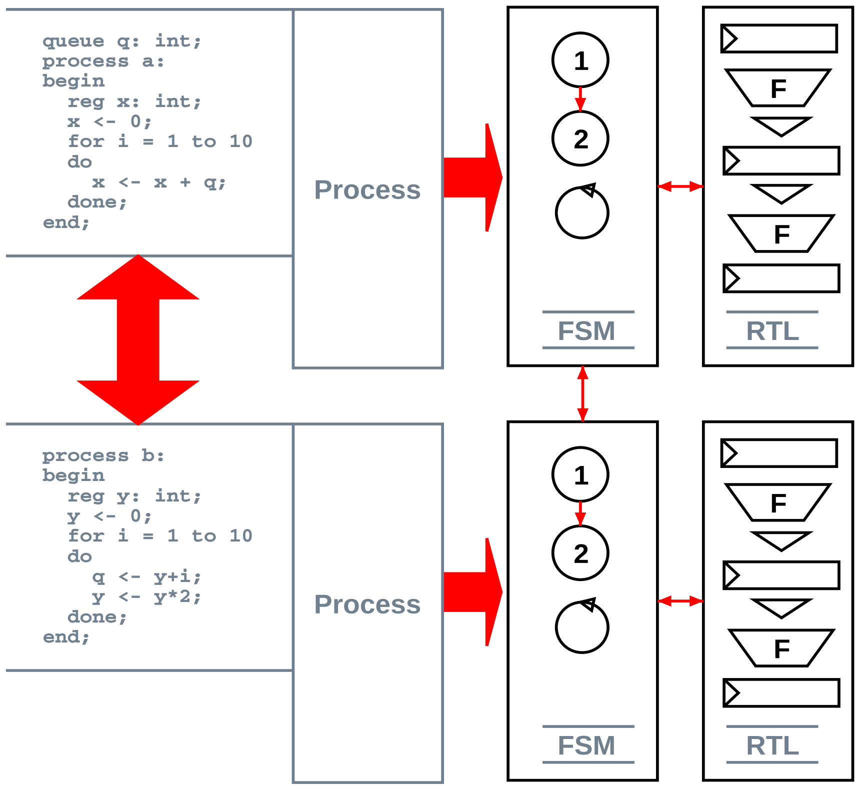
The **behavioural level** usually describes the functional behaviour of the full design interacting with the environment. Most applications and data processing are modelled on algorithmic behavioural level using some kind of imperative programming language.

The ConPro high-level synthesis of SoC designs uses a behavioural imperative programming language with a compiler-based synthesis approach from algorithmic programming level to register-transfer level mappable directly to digital logic **[2]**.

Concurrency is modelled explicitly on control path level with processes executing

a set of instructions sequentially, initially independent of any other process. Inter-process-communication (IPC) provides synchronization with different objects (mutex, semaphore, event, timer) and data exchange between processes using queues or channels, based on the **Communicating Sequential Processes** (CSP, Hoare 1985) model.
There are local and global resources (storage, IPC) , accessed by one process and several processes, respective. Concurrent access of global resources is automatically guarded by a mutex scheduler, serializing access, and providing atomic access without conflicts.
There are process and top-level instructions. Top-level instructions are evaluated during synthesis (configuration). Process instructions are transformed and mapped to states of a clock-synchronous finite-state-machine (FSM) controlling the process RTL data path temporally and spatially, shown in figure **1**.
More fine-grained concurrency is provided on data path level using bounded blocks executing several instructions (only data path, e.g. data assignments) in one time unit. Block level parallelism can be enabled explicitly or implicitly explored by a basicblock scheduler **[2]**.
The complete synthesis process can be fine-grained parameterized on programming block level, for example selection of different expression models (allocation) or activation of specific schedulers and optimizers.

Figure 1. Mapping of the proposed multi-process model to FSM-RTL architecture using high-level synthesis.



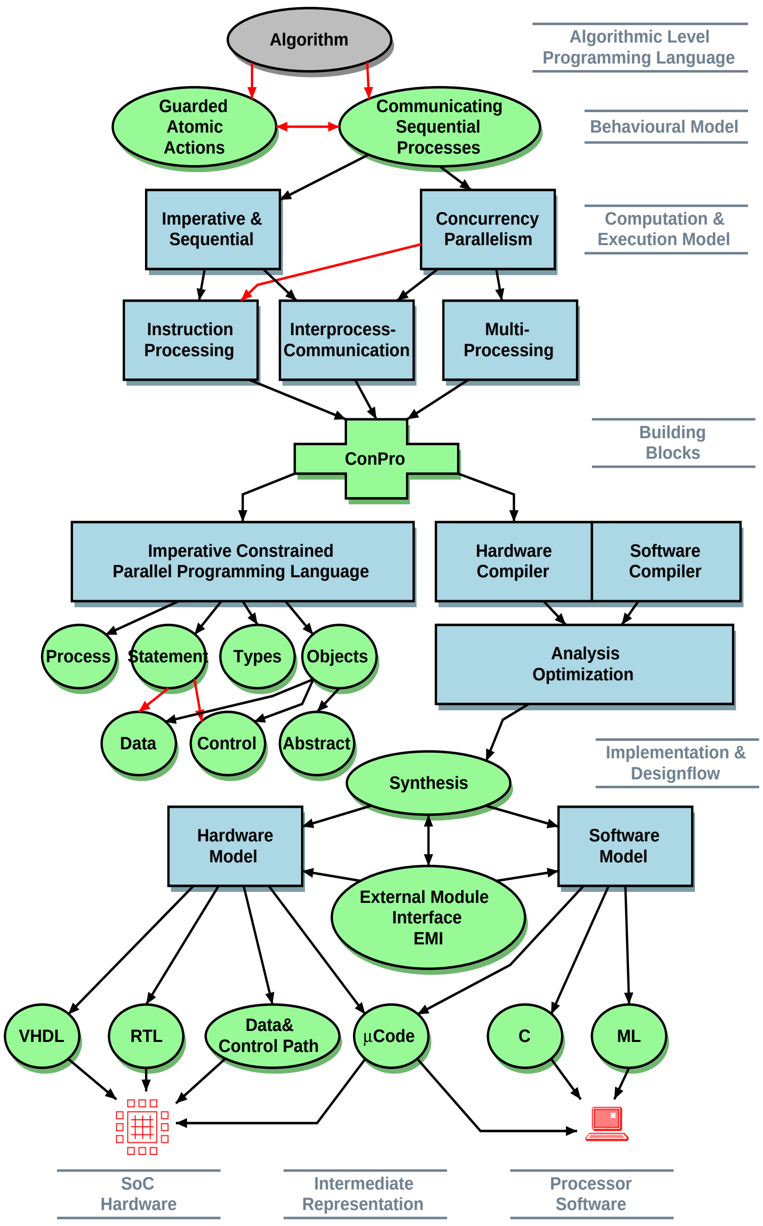
```
queue q: int;
process a:
begin
  reg x: int;
  x <- 0;
  for i = 1 to 10
  do
    x <- x + q;
  done;
end;
```

```
process b:
begin
  reg y: int;
  y <- 0;
  for i = 1 to 10
  do
    q <- y+i;
    y <- y*2;
  done;
end;
```

Hardware blocks, modelled on hardware level (VHDL), can be accessed from the programming level using an object-orientated programming approach with meth-

ods. All hardware blocks, including IPC, are treated like abstract data type objects (ADTO) with a defined set of methods accessible on process level and top level (only applicable with configuration methods, for example setting the time interval of a timer). The bridge between the hardware and software model is provided by the External Module Interface (EMI).
The relationship of the proposed programming and execution model and the required building blocks of Conpro (programming language and synthesis) are illustrated in figure **2**.

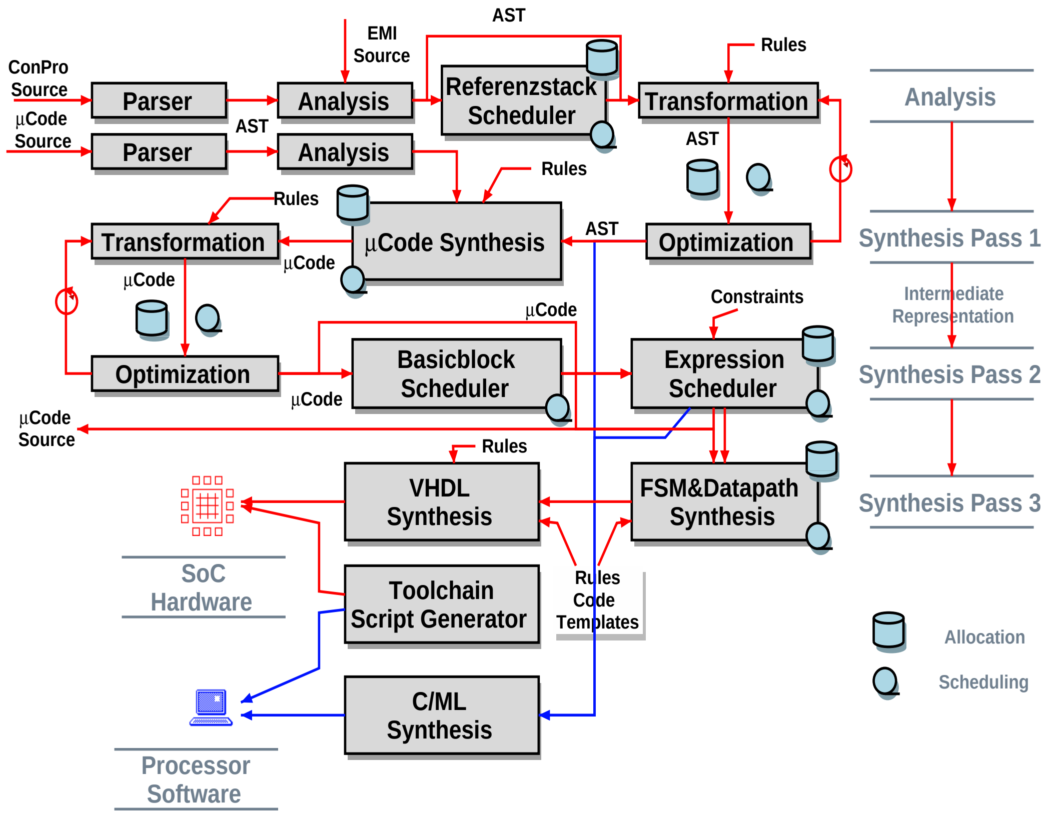
Figure 2. Building blocks: from the programming model to hardware using high-level synthesis.



The programming language supports different types of storage objects (single registers and variables in shared RAM blocks, true bit-scaled), different aggrega-

tion types (array, structure) and abstract objects. Programming statements can modify data (expressions, assignments) or have impact on the control flow (conditional and counting loops, conditional branches, concurrent multi-value selection).
Figure **3** gives an overview of the design flow guiding through different levels provided by the ConPro framework. After the source code is parsed and transformed into an abstract syntax tree (AST), there are different allocation, scheduling, and optimization stages. The reference stack scheduler performs symbolic analysis on AST level and resolves constant and storage propagation, conditional assignments and multiple assignments. This ALAP scheduler has impact on scheduling and allocation done by optimization. The intermediate µCode representation was choosen for simplified RTL synthesis and optimization (synthesis pass I).
The basicblock scheduler partitions the program code into blocks without control side entries containing only data assignments (basicblocks). For each basicblock a data-dependency analysis is performed. Independent data assignments can be bound to the same time unit. These optimizing schedulers can be activated or deactivated on block level. Finally in synthesis pass III the RTL is synthesized and mapped to VHDL. Alternatively, after pass I (AST) or II (µCode) software output with same functional and simulated/scheduled concurrency behaviour can be compiled.

Figure 3. SoC design flow using the high-level synthesis framework ConPro providing mapping of a parallel programming model to RTL hardware and alternatively to software.



Equation 1.

$$\chi\ :\ \mathrm{CP} \rightarrow \mathrm{AST} \rightarrow \mu\mathrm{CODE} \rightarrow \mathrm{RTL} \rightarrow \mathrm{VHDL}$$

is defined by a set of rules χ. Each set consists of subsets which can be selected by parameter settings (for example scheduling like loop unrolling, or different allocation rules) on block level.
Example **1** shows a concurrent computation system performing data modification by an array of four processes `sum[0..3]`. They access the global register `x`. Though the access of `x` is atomic and guarded, the expression in line 9 is it not, thus a mutual exclusion lock `m` is required. A master process `someother` controls the system and waits for completion of all `sum` processes using a semaphore. A timer `t` performs group synchronization (here just for fun). The synthesis is controlled on block level with different settings (loop unrolling in line 10, scheduling in line 11, object constraints in lines 2 & 3). Line 14 creates a bounded block for data assignments to registers `a` and `b` (using a colon instead of a semicolon).

Example 1. Parts of a ConPro source code example.

```
1  open Mutex; open Timer; open Process; ...
2  object m: mutex with scheduler=”fifo”;
3  object t: timer; t.time(1 millisec);
4  object s: semaphore;
5  reg x: int[12];
6  array sum: process[4] of begin
7    for i = 1 to 10 do begin
8      t.await ();
9      m.lock(); x <- x + 1; m.unlock ()
10   end with unroll=true; s.up ();
11 end with schedule=”basicblock”;
12 process someother: begin
13   reg a,b: int[10];
14   a <- x+1, b <- x-1; x <- a;
15   t.init (); t.start (); s.init(0);
16   for i = 0 to 3 do sum.[i].start();
17   for i = 1 to 4 do s.down();
18 end;
```

Objects (like IPC) belong to a module, which have to be opened first (line 1). Each module is defined by a set of EMI implementation files providing all necessary informations about objects of this module (like method declarations, object access and implementation on hardware level).

4. Bibliography

[1] Malik, Avinash and Salcic, Zoran and Roop, Partha S., *SystemJ compilation using the tandem virtual machine approach*, ACM Trans. Des. Autom. Electron. Syst., Vol 14, (2009)

[2] S. Bosse, *ConPro: Rule-Based Mapping of an Imperative Programming Language to RTL for Higher-Level-Synthesis Using Communicating Sequential Processes*, Technical Paper, BSSLAB, Bremen, 2009

[3] Charles, Philippe et al., *X10: an object-oriented approach to non-uniform cluster computing*, OOPSLA ’05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (2005)

[4] Daniel L. Rosenband and Arvind, *Modular Scheduling of Guarded Atomic Actions*, Proceedings of the 41st annual conference on Design automation (2004)

[5] C. Hoare, *Communicating Sequential Processes*, Prentice Hall, 1985

[6] S. Bosse, D. Lehmhus, *Smart Communication in a Wired Sensor- and Actuator-Network of a Modular Robot Actuator System Using a Hop-Protocol with Δ-Routing*, Smart Systems Integration, Como, Italy, 23-24.3.2010

Processes and Process Management

Models, Architectures, and API

1. Description

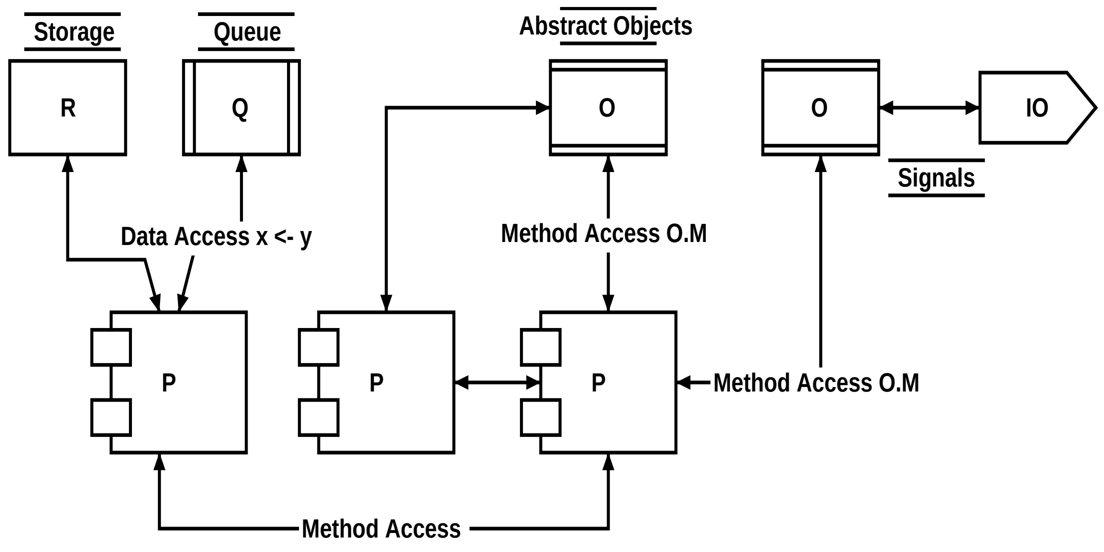
Processes are the main execution environment of a ConPro module. A process executes a set of instructions in sequential order. Processes communicate with each other and the outside world using shared objects (Interprocess communication IPC).

2. Multi-Process Architecture

The figure **1** shows a typical multi-process architecture. Processes execute a set of instructions in a strict sequential order (sequential process). Instructions can modify the data- and control path. A process is a local execution context. Processes can communicate using global Interprocess-Communication objects, like globale storage or explicit synchronization objects like mutex, semaphore, timer

(Communicating Sequential Processes model by C. Hoare).

Figure 1. Multi-Process architecture



Abstract objects are accessed by a defined set of methods. The objects are guarded providing atomic guarded action semantic during concurrent access by multiple processes using a mutex guarded access scheduler.
A process is mapped to register-transfer-logic and finite-state machines.
The states of the finite-state machine of a process relates to instructions of a process on programming level. ConPro instructions are mapped to states in a unique way. Normally, each simple instruction like data assignement and expressions are mapped to one state. One state requires at least on clock cycle. A group of data instructions can be bound to one state, providing concurrency on data path level. This bounding can be explored automatically by the Basicblock-Scheduler. Complex instructions like loops or branches are split into several different states.

3. Programming Interface API

Multiple processes are defined on programming level using the process environment. The process environment consists of a unique process name and the process body, consisting of local object definitions and instructions. The formal syntax specification for a process definition can be found in definition **1**, and for a process array in definition **2**. To distinguish a single process of an array, an array identifier # can be used in expressions, giving the index of the process starting with index value zero.
Processes are abstract objects, too. Thus there is a set of methods which can be applied to processes, described in table **1**. They are controlling the state of a pro-

cess.

Definition 1. Formal syntax specification for a process definition

```
process-def ::= 'process' identifier ':'
                'begin' process-body 'end' parameters? ';' .
paramters ::= parameter // 'and' .
parameter ::= identifier '=' value .
process-body ::= obj-definition* instruction* .
obj-definition ::= storage-definition | object-definition | type-definition .
```

Definition 2. Formal syntax specification for a process array definition

```
process-array-def ::= 'array' identifier ':' 'process' '[' number ']' .
                      'begin' process-body 'end' parameters? ';' .
process-identifier ::= '#' .
```

Table 1. Process object methods

Method	Description
start	Start the specified process concurrently to calling process.
stop	Stop the specified process.
call	Start the specified process sequentially. This suspends the calling process untill the called process terminates (reaches his end state).

An example of process definitions and the control of processes using object methods is shown in example **1**. Two global queues `q1` and `q2` are used for interprocess-communication.

Example 1. Example of process definition and method access

```
queue q1,q2: int[6] with depth=4;
process x:
begin
  reg t: int[6];
  always do
  begin
    t <- q1;
    t <- t * 100;
    q2 <- t;
  end;
end;
process y:
begin
  reg u: int[6];
  reg sum: int[16];
  sum <- 0;
  for i = 1 to 10 do
  begin
    u <- i asl 2;
    q1 <- u;
    u <- q2;
    sum <- sum + u;
  end;
end;
process z:
begin
  x.start ();
  y.call ();
end;
```

Table 2. Summary of process definition and access.

Code	Description

`process P:`
`begin ... end;`
	Definition and implementation of a process with name `P`.

`process P:`
`begin ... end`
`with param=value;`
	Definition of a process with additional parameters.

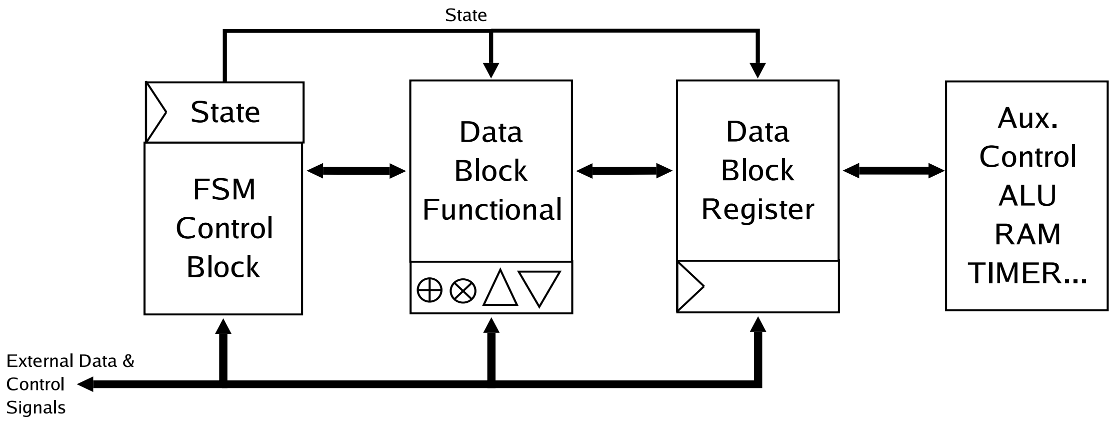
`array PA: process [N]`
`begin ... end`
`with param=value;`
	Definition of a process `PA` array of size `N` with additional parameters.

`p.start ():`
`p.call ();`
`p.stop ()`
	Start, call, and stop the specified process.

4. Process Hardware Architecture

The hardware process architecture is shown in figure **2**. The main entity and architecture of the hardware implementation of a ConPro process is shown in algorithm **1**, and each hardware process from figure **2** is shown in algorithms **2** to **5**.

Figure 2. Hardware architecture of a process



Instructions of a ConPro process are mapped to states of the finite state machine, named `S_xx`. There is a start and end state of each process.
The data path is divided into a pure combinational and a transitional part. The first one accesses local (read only) and global objects (read, write, and control access).
Access of global ressources is always guarded by the object access scheduler (mutual exclusion lock). A request signal (`RE`/`WE` or `CALL`) is activated in the data path. A guard signal `GD` is read by the finite state machine. The actual state, accessing the object, is hold untill the guard signal changes to low level (only for one clock period).

Algorithm 1. Hardware model of a ConPro process: VHDL hardware entity and architecture

```
entity <process>
port(
  --
  -- Access of globale objects: 1. control signals, 2. data path signals
  signal <object>_RE: out std_logic;
  signal <object>_WE: out std_logic;
  signal <object>_GD: in std_logic;
  signal <object>_RD: in std_logic_vector[];
  -- Clock and Reset signals for processes
  signal conpro_system_clk: in std_logic;
  signal conpro_system_reset: in std_logic
);
end <process>;
```

```
architecture main of <process> is
  -- local signals
  -- local objects
  -- local types
  -- state machine
  type pro_states is (
    S_start,
    S_i1,
    S_i2,
    ...
    S_main_end -- PROCESS0[:0]
    );
  signal pro_state: pro_states := S_main_start;
  signal pro_state_next: pro_states := S_main_start;
begin
  state_transition: proess
  control_path: process
  data_path: process
  data_trans: process
end;
```

Algorithm 2. Hardware model of a ConPro process: VHDL state transition hardware process (transitional)

```
state_transition: process(
          PRO_main_ENABLE,
          pro_state_next,
          conpro_system_clk,
          conpro_system_reset
  )
begin
  if conpro_system_clk'event and conpro_system_clk='1' then
    if conpro_system_reset='1' or PRO_main_ENABLE='0' then
      pro_state <= S_main_start;
    else
      pro_state <= pro_state_next;
    end if;
  end if;
end process state_transition;
```

Algorithm 3. Hardware model of a ConPro process: VHDL control path hardware process (pure combinational)

```
control_path: process(
        PRO_init_GD,
        <object>_GD,
        pro_state
        )
begin
  PRO_main_END <= '0';
  case pro_state is
    when S_start =>
      pro_state_next <= S_i1;
    when S_i1 =>
      pro_state_next <= S_i2;
    when S_i2 =>
      if <object>_GD = '1' then
        pro_state_next <= S_i2;
      else
        pro_state_next <= S_i3;
      end if;
    when S_main_end =>
      pro_state_next <= S_main_end;
      PRO_main_END <= '1';
  end case;
end process control_path;
```

Algorithm 4. Hardware model of a ConPro process: VHDL data path hardware process (pure combinational)

```
data_path: process(
        <object>_RD,
        pro_state
        )
begin
  -- Default values
  <object>_WR <= <expression>;
  <object>_WE <= '0';
  case pro_state is
    when S_start =>
      null;
    when S_i1 =>
      <object>_WR <= <expression>;
      <object>_WE <= '1';
    when S_i2 =>
      <object>_<method> <= '1';
    ...
end;
```

Algorithm 5. Hardware model of a ConPro process: VHDL data path hardware process (transitional)

```
data_trans: process(
        <object>,
        conpro_system_clk,
        conpro_system_reset,
        pro_state
        )
begin
  if conpro_system_clk'event and conpro_system_clk='1' then
    if conpro_system_reset = '1' then
      <object> <= <expression>;
    else
      case pro_state is
        when S_start =>
          null;
        when S_i2 =>
          null;
        when S_i3 =>
          <object> <= <expression>;
        ...
end;
```

Object and Data Types

ConPro type system

1. Introduction

Objects are specified by their object type α and a data type β. There are data storage and abstract type objects. User defined types providing product types using arrays and structures and restricted sum types with enumerated symbolic name lists are available.
There are top-level objects shared by a set of processes, and local process objects accessed only by one process. Shared objects can be accessed concurrently. To guarantee atomic and consistent access in this case, a mutual exclusion lock scheduler is used for each object.

2. Object Types

The set of objects consists of data storage objects $\Re$ with different access behaviour and abstract objects Θ including interprocess-communication objects $\Im$. Data storage objects can be used directly in expressions. Abstract objects are accessed and manipulated by a set of methods. Objects can be defined locally in a process context within the process body, or globally in a toplevel module context.
Queues and channels are both data storage and abstract objects (part of inter-

process-communication).

Table 1. Object Types

type α=OT	Description
`reg`	Single storage register with CREW access behaviour.
`var`	Variable storage element stored in a shared RAM block with EREW access behaviour.
`sig`	Hardware signal used for hardware component interconnect and access.
`channel`	Synchronized data based IPC communication (buffered or unbuffered)
`queue`	Synchronized data based IPC communication with FIFO access behaviour
`object`	Abstract data type object accessed and modified with methods

3. Data Types

Table **2** lists all available data types which can be used with expressions, functions and assignments. These data types can be applied to a subset of available object types (data storage and some interprocess communication objects). There are type conversion functions for each basic type.

Table 2. Data Types

type β=DT	Description
`logic`	Single logic bit
`logic[`ω`]`	Unsigned integer or logic vector of width ω bit
`int[`ω`]`	Signed integer type of width ω bit (includung sign bit)
`char`	Character byte ($\equiv$ logic[8] type, allocates 8 bits)

type β=DT	Description
`bool`	Boolean type (≡ logic type, allocates one bit)

4. Data Storage Objects

True bit-scaled data types (TYPE β) and storage objects (subset $\Re$ of TYPE α) are supported. The data width can be choosen in the range ω={1,2,…,64} Bit. The formal syntax of scalar object definition is shown in definition **1**, and is summarized in table **3**.
A data storage object $\Re$ is specified and defined by a cross product of types (α×β). Storage objects can be read in expressions and can be written in assignments. Definition **2** gives the formal syntax specification.
Registers are single storage elements either used as a shared global object or as a local object inside a process. In the case of a global object, the register provides concurrent read access (not requiring a mutex guarderd scheduler) and exclusive mutex guarded write access. If there is more than one process trying to write to the register, a mutex guarded scheduler serializes the write accesses. There are two different schedulers available: static priority and dynamic FIFO scheduled (see examples **1** and **2**).
Variables are storage elements inside a memory block either used as a shared global object (the memory block itself) or as a local object inside a process (see examples **1** and **2**). A variable provides always exclusive mutex guarded read and write access.
Different variables concerning both data type and data width can be stored in one or several different memory blocks, which are mapped to generic RAM blocks. Address management is done automatically during synthesis and is transparent to the programmer. Direct address references or manipulation (aka pointers) are not supported.
The memory data width, always having a physical type logic/bit-vector, is scaled to the largest variable stored in memory. To reduce memory data width, variables can be fragmented, that means a variable is scattered about several memory cells.
Different memory blocks can be created explicitly, and variables can be assigned to different blocks.
Queues are storage elements with FIFO access order and interprocess communcation objects, too. They are always used as a shared global object. Queues and channels can be used directly in expressions like any other storage object, see example **3**.
Channels are primarily interprocess communcation objects, too. They are always used as a shared global object. They can be buffered (behaviour like a queue with one cell, depth is 1) or unbuffered (providing only a handshaked data transfer).
Register, variables, queues, and channels can be defined for product types (ar-

rays and structure), and sum types (enumeration type), too, see section **7** to **9**.

Definition 1. Formal syntax specification of a data object definition.

```
dataobj-definition ::= obj-type ( identifier // ',' ) ':' data-type
  [ with parameter-list ] ';' .
obj-type ::= reg | var | queue | channel .
data-type ::=  logic |
   logic '[' number ']'  |
   int '[' number ']'  |
   bool  |
   char  .
```

Definition 2. Formal syntax specification of a data object access in expressions.

```
dataobj-access ::= identifier | bit-selector | type-conversion .
identifier ::= name .
bit-selector ::=  identifier '[' range ']' .
range ::= number | number to number | number downto number .
type-conversion ::=    to_logic '(' dataobj-access ')'  |
   to_int '(' dataobj-access ')'  |
   to_bool '(' dataobj-access ')'  |
   to_char '(' dataobj-access ')'  .
```

Table 3. Summary of scalar data storage object definitions and access in expressions

Definition	Description
`reg` *name*: *DT*[*N*];	Defines a new storage object of type register with a specified data type *dt* and optional data width *N* bits.
`queue` *name*: *DT*[*N*]; `queue` *name*: *DT*[*N*] `with depth=8`;	Defines a new interprocess communication data object of type queue with a specified data type *dt* and optional data width *N* bits. Optional parameters are passed using the `with` statement.
`block` *RAM*; `var` *name*: *DT*[*N*] `in` *RAM*; `var` *name*: *DT*[*N*];	Defines a new storage object of type variable with a specified data type *dt* and optional data width *N* bits. The object is stored in a shared RAM block (optional).
x ← y;	Appearence of stoarge objects on left and right hand side of an assignment.
x[*J*] ← y[*I*];	Bit index selector used in expression with vector objects.

Example 1. Definition of global storage objects and access in expressions

```
reg x,y,z: int[5];
var v: int[8];
process xyz:
begin
  for i = 1 to 10 do
  begin
    x ← x + 1;
    y ← x * z - 1;
    if z < 10 and x > 100 then v ← v + z;
  end;
end;
```

Example 2. Definition of local storage objects and access in expressions

```
process xyz:
begin
  reg x,y,z: int[5];
  var v: int[10];
  for i = 1 to 10 do
  begin
    x ← x + 1;
    y ← x * z - 1;
    if z < 10 and x > 100 then v ← v + z;
  end;
end;
```

Example 3. Definition of interprocess-communication objects (queues) and access in expressions

```
queue q1,q2: char;
process flip:
begin
  reg c: char;
  c ← 'a';
  for i = 1 to 10 do
  begin
    q2 ← c + 1;
    c ← q1;
  end;
end;
process flop:
begin
  reg c: char;
  for i = 1 to 10 do
  begin
    c ← q2;
    q1 ← c + 1;
  end;
```

```
20 end;
```

5. Signals

Signals are interconnection elements without a storage model. They provide an interface to external hardware blocks. Signals are used in component structures, too, shown in example **4**. Lines 1 to 7 define a signal port interface of a component, and line 8 instantiates a component of this type.
Signals can be used directly in expressions like any other storage object. Signals can be read in expressions, and a value can be assigned in assignments, shown in example **4** (for example line 11 using a static map statement, and line 33). Reading a signal returns the actual value of a signal, for example line 23, but writing to a signal assigns a new value only for the time the assignment is active, otherwise a default value is assigned to the signal, for example in line 34. Therefore, there may be only one assignment for a signal.
Signals are non-shared objects, and have no access scheduler. Only one process may assign values to a signal (usually using the `wait for` statement), but many processes may read a signal concurrently. Additionally, signals can be mapped to register outputs using the `map` statement.

Definition 3. Formal syntax specification of a signal object definition.

```
signal-definition ::= obj-type ( identifier // ',' ) ':' data-type
  [ with parameter-list ] ';' .
obj-type ::= sig .
data-type ::=  logic |
   logic '[' number ']'  |
   int '[' number ']'  |
   bool  |
   char .
```

Example 4. Example of signal definitions and signal access. Component structure elements are signals, too.

```
1  type dev_type : {
2    port leds: output logic[4];
3    port rd: input logic[8];
4    port wr: output logic[8];
5    port we: output logic;
6    port act: input logic;
7  };
8  component DEV: dev_type;
9  export DEV;
10 reg stat_leds: logic[4];
11 DEV.leds << stat_leds;
12 signal s1: int[8];
```

```
signal s2: logic;
reg xs: int[8];
export s1,s2;
process p1:
begin
  reg x: int[8];
  x ←0;
  stat_leds[0] ← 1;
  for i = 1 to 5 do
  begin
    x ← x + s1;
  end;
  xs ← x;
  stat_leds[0] ← 0;
end;
process p2:
begin
  stat_leds[1] ← 0;
  for i = 1 to 5 do
  begin
    wait for DEV.act = 1 with s2 ← 1;
    DEV.we ← 1,DEV.wr ← to_logic(xs);
  end;
  stat_leds[1] ← 0;
end;
```

6. Abstract Object Types

The set of abstract data type objects Θ define objects implementing reactive blocks interacting with the processes and the environment, for example interprocess-communication or data links. They are not directly accessible in expressions like registers (with some exceptions). Abstract objects belong to modules, defined by the External Module Interface (EMI). A module assigns a type to an abstract object.
Before abstract objects of a particular type can be used, the appropiate module must be opened first, shown in definition **4**.

Definition 4. Opening of a module.

open-module ::= open *mod-name* ';' .

ADT objects can be accessed by their appropiate method set θ={θ_1,θ_2,...}. A method is applied using the selector '.' operator followed by a list of arguments passed to method parameters, with arguments separated by a comma list encapsulated between paranthesis, shown in definition **5**.

Definition 5. Object method calls. The object must be first created with the object defintion statement.

```
object-defintion ::= object obj-name ':' obj-type ';' .
object-call ::= obj-name '.' method-name '(' ( argument \\ ',' ) ')' ';' .
```

Methods which do not expect arguments are applied with an empty argument list `()`. Table **4** summarizes the statements required for using abstract object types.

Table 4. Summary of abstract object module inclusion, object definition and object access.

Statement	Description
`open Module;`	Open specified ADTO module
`object obj: objtype;`	Defines and instantiates a new object of specified ADT.
`object obj: objtype with param=valu;`	Defines and instantiates a new object of specified ADT type with additional parameter settings.
`obj.meth`	Object method access using the selector operator

7. Array Types

Arrays are product types and can be applied to data storage objects including queues and channels. Additionally, arrays can be applied to abstract objects, too, providing indexed object selection. Array elements can be accessed with static selectors (constant values or expressions foleded to a constant value), and with dynamic selectors (expressions referencing storage objects).
Arrays of variables are implented in memory blocks. The element access is performed by address calculation and access of the memory block. Arrays of any other object type are always implemented with single objects. If elements of such an array only accessed with static selectors, the array access is replaced by the appropiate object. If at least there is one element access with a dynamic selector expression, all objects of this array are accessed my multi- and demultiplexer blocks (simulated memory block implementation).
Arrays of processes can be defined, too. A process identifier symbol #=[0,N-1] can be used in expressions of each process created by the array definition.
Multi-dimensional arrays are supported, too. But in this case each dimension must be of power 2 (or will be aligned during the synthesis). Multi-dimensional arrays are mapped to one-dimensional arrays to simplify hardware synthesis. the

one-dimensional idnex is calculated by the following equation:

Equation 1.

$$I(I_0, I_1, \ldots) = I_0 + \sum_{i=1}^{D-1} I_i S_{i-1}$$

Formal syntax definitions **6** and **7** show definition and access of arrays, summarized in table **5**. An extended example **5** demonstrates object and storage arrays.

Definition 6. Formal syntax specification for array definition

```
array-storage-definition ::= array ( identifier // ',' ) ':'
    object-type '[' dim ']' of data-type
    [ with parameter-list ] ';' .
array-abstract-object-definition ::= array ( identifier // ',' ) ':'
    object object-type '[' dim ']'
    [ with parameter-list ] ';' .
array-process-definition ::= array ( identifier // ',' ) ':'
    process '[' dim ']'
    begin
      instructions
    end [ with parameter-list ] ';' .
dim ::= ( number // ',' ) .
```

Definition 7. Formal syntax specification for array element selection and access

```
array-selector ::= identifier  '.'  ( static-selector | dyanmic-selector ) .
static-selector ::= '[' ( number // ',' ) ']' .
dynamic-selector ::= '[' ( expression // ',') ']' .
expression ::= identifier | simple-expression .
dim ::= ( number // ',' ) .
```

Table 5. Summary of array type definition and array element access.

Statement	Description

`array A: OT[N] of DT;`
`array A: OT[N,M,O] of DT;`
	Defines an storage array of size N with object type OT and data type DT. Second line defines a multi-dimensional array (matrix).

`array A: OT[N] of DT`
`        with param=value;`
	Defines an storage array of size N with object type OT and data type DT, with additional parameter settings.

`array A: object obj[N]`
`        with param=value;`
	Defines an abstract object array of size N with object type obj, with additional parameter settings. Note: object parameters must be preceeded by the module name and the dot selector!

`array A: process[N] of`
`begin`
`  B`
`end`
	Defines an array of N processes.

`a.[2] <- a.[i+1] + a.[0];`
`timer.[1].await ();`
`timer.[i+1].await ();`
	Access of storage array elements using the dot selector on right-hand and left-hand sides of an expression. Second lines selects an abstract object and applies the method to this object.

Example 5. Arrays of storage and abstract objects (including processes)

```
open Core;
open Process;
open Semaphore;

array fork: object semaphore[5] with
            Semaphore.depth=8 and Semaphore.scheduler="fifo";
array eating,thinking: reg[5] of logic;

process init:
begin
  for i = 0 to 4 do
  begin
    fork.[i].init (1);
  end; -- with unroll;
  ev.init ();
end;

function eat(n):
begin
  begin
```

```
    eating.[n] <- 1;
    thinking.[n] <- 0;
  end with bind;
  wait for 5;
  begin
    eating.[n] <- 0;
    thinking.[n] <- 1;
  end with bind;
end with inline;

array philosopher: process[5] of
begin
  if # < 4 then
  begin
   always do
   begin
     -- get left fork then right
     fork.[#].down ();
     fork.[#+1].down ();
     eat (#);
     fork.[#].up ();
     fork.[#+1].up ();
   end;
  end
  else
  begin
   always do
   begin
     -- get right fork then left
     fork.[4].down ();
     fork.[0].down ();
     eat (#);
     fork.[4].up ();
     fork.[0].up ();
   end;
  end;
end;

process main:
begin
  init.call ();
  for i = 0 to 4 do
  begin
    philosopher.[i].start ();
  end;
end;
```

8. Structure Types

Structure types are used to define a product of types from the set of core data types, providing different data widths, too. In contrast to arrays, a structure type must be defined first without creation of any data object. After type definition storage data objects of this type can be created (instantiated). Supported storage object types are: register, variable, queue, channel. Additionally structure types can be used to construct signal types and component (port) interfaces.
There are three different subclasses of structures for different purposes:

Type Structure
The generic structure type binds different named structure elements with different data types to a new user defined data type, the native product type.

Bit-Type Structure
This structure subclass provides a bit-index-name mapping for storage objects. All structure elements have the same data type. The bit-index is either one bit number or a range of bits. This structure type provides symbolic/named selection of parts of vector data type (for example logic vector and integer types) and clarifies bit access of objects.

Component Structure
This structure defines hardware component ports, either of a ConPro module toplevel port, or of an embedded hardware component (modelled on hardware level). This structure type can only be used with component object defintions. The component type has equal behaviour like the signal type.

The structure type defintion therefore contains only data types, with binding of one object type using the generic object definition statement. A structure type binds a set of different structure elements, distinguished by their names.

Tip

The member names of structures should begin with a lower case letter, the elements of a enumerated symbolic list should begin with a uppercase letter.

Elements of a structure can be accessed using the dot selector: the object and element name is concatenated with a dot. Further selections (array, bit range) can be applied, too.
In the case the object type of a structure is a register, a set of independent registers are created. In the case of a variable type, structure elements are stored into a memory block.
Arrays from structure types can be created. For each structure element a different array is created.
Hardware component port types are defined with structures, too, with the difference that for each structure element the direction of the signal must be specified. Some care must be taken for the direction: if the component is in lower hierarchi-

cal order (an embedded external hardware component), the direction is seen from the external view of the hardware component. If the component is part of the top-level port interface of a ConPro module, it must be seen from the internal view. Formal syntax definitions for structure definitions and structure element access can be found in definitions **8** to **9**. An extended demonstration of the capabilities of structure types can be found in example **6**. Table **6** summarizes the definition and usage of structure types.

Definition 8. Formal syntax specification for structure type definition

```
struct-type-definition ::= type ( identifier // ',' ) ':' '{'
                           ( identifier ':' data-type ';' // )
                           '}' ';' .
bit-type-definition ::= type ( identifier // ',' ) ':' '{'
                           ( identifier ':' range ';' // )
                           '}' ';' .
component-type-definition ::= type ( identifier // ',' ) ':' '{'
                           ( 'port' identifier ':' data-dir data-type ';' // )
                           '}' ';' .
object-definition ::= object-type ( identifier // ',' ) ':' struct-type .
component-definition ::=  component ( identifier // ',' ) ':' component-type .
range ::= number | number 'to' number | number 'downto' number .
```

Definition 9. Formal syntax specification for structure element selection and access

```
struct-selector ::= identifier  '.'  identifier .
```

Example 6. Structures with register, variable and component object types.

```
-- Multi-type structure type definition
type registers : {
  ax : logic[32];
  bx : logic[32];
  sp : logic[16];
 };
type image : {
  row: logic[32%4];
  col: logic[32%4];
 };
-- Component structure type defintion
type uart : {
  port rx : input logic[2];
  port tx : output logic[2];
  port re : output logic;
  port we : input logic;
 };
-- Bit-type structure type defintion
type command : {
  ack:  0;
  cmd: 1 to 2;
  data: 3 to 7;
 };

block ram1;
reg regs : registers;
var vegs : registers in ram1;
var vim : image in ram1;
var after : logic[16] in ram1;
component dev1: uart;
reg cmd: command;
process p1:
begin
  reg x: logic[2];
  type cpu_regs : {
   ax : logic[8];
   bx : logic[8];
   sp : logic[8];
  };
  var cpu : cpu_regs in ram1;
  reg row: logic[32];
  ...
  regs.ax ⟵ row;
  regs.ax ⟵ regs.ax + 1;
  vegs.ax ⟵ vegs.ax + intern;
  ...
  wait for dev1.re = 1;
  x ⟵ dev1.rx;
  dev1.tx ⟵ x, dev1.we ⟵ 1;
  cmd ⟵ 0;
  cmd.ack ⟵ 1;
  cmd.cmd ⟵ x;
```

```
end;
```

Table 6. Summary of structure type definition and structure element access.

Statement	Description

`tpye ST: {`
`  e1: DT1;`
`  e2: DT2; ...`
`};`
	Defines a new structure type with elements e1,e2,... of data types DT1,DT2,...

`tpye CT: {`
`  port e1 : DIR1 DT1;`
`  port e2 : DIR2 DT2; ...`
`};`
	Defines a new component port type with port elements e1,e2,... of data types DT1,DT2,... with specific signal direction DIR1,DIR2,...

`tpye BT: {`
`  e1: BN1;`
`  e2: BN2A to BN2B; ...`
`};`
	Defines a new bit type structure with elements e1,e2,... and bit-widths BN1, BN2...

`reg R: ST;`
`var R: ST;`
`reg R; BT;`
	Defines scalar storage objects of structure type ST and bit type BT.

`array AS: OT[N] of ST`
`          with PARAMS;`
	Defines an array of storage objects with object type OT and structure type ST.

`component C: CT;`
`export C;`
	Defines a new component structure and exports the component structure.

`ST.e1 <- ST.e2;`	Access of structure elements in assignments and expressions.

9. Enumeration Types

Enumeration types define a mapping of symbolic names to constant integer numbers, with formal syntax definition **10**. Elements of a enumerated type can be used in expressions like any other storage objects. Object of a enumerated type can be created like any other storage object definition, shown in example **7**.

Definition 10. Formal syntax specification for enumeration type definition

```
enum-type-definition ::= type ( identifier // ',' ) ':' '{'
                         ( identifier  ';' // )
                         '}' ';' .
object-definition ::= object-type ( identifier // ',' ) ':' enum-type .
```

Example 7. Enumeration types

```
type states : {
  S_START;
  S_1;
  S_2;
  S_END;
};
reg state,next_state: states;
process fsm:
begin
  while state <> S_END do
  begin
    match state with
    begin
      when S_START: state ← S_1;
      when S_1: state ← S_2;
      when S_2: state ← S_END;
    end;
  end;
end;
```

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. 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,

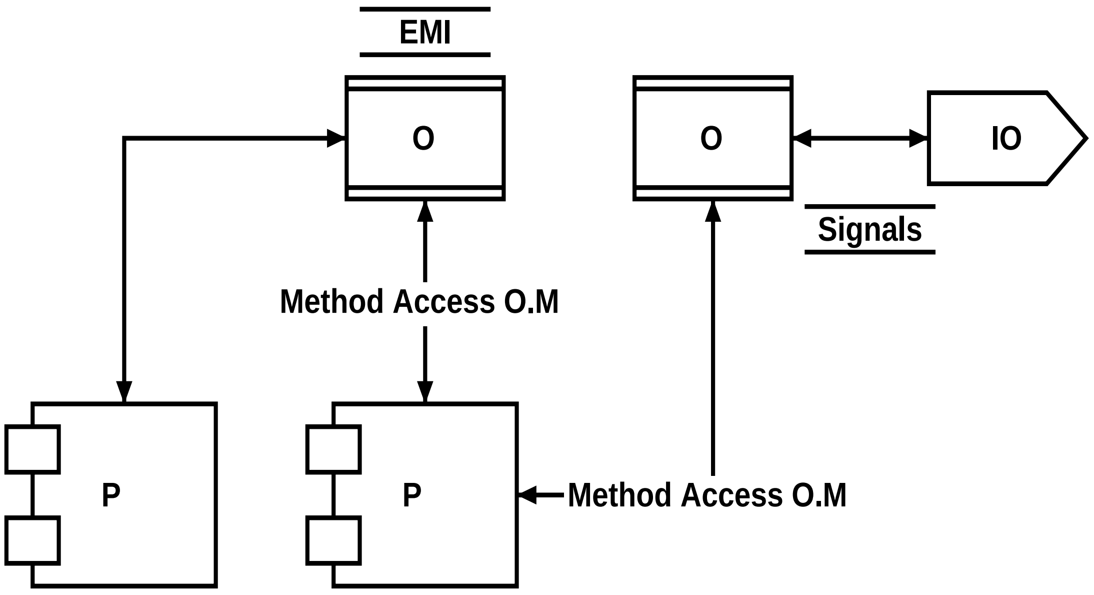
   the method access, on ConPro process level defining data and control path parts and the implementation (scheduling) of the object access, too,

   all required signals for process interconnect and implementation,

   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).
Figure **1** shows a typical architecture of embedded abstract objects accessed by different processes. There are two different classes of abstract objects: (1) objects without external system interaction and (2) objects connected to the system hardware interconnect, for example bus, memory, or link objects.

Figure 1. Typical system architecture with embedded abstract objects modelled and implemented with EMI.



3. 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 .

4. 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;
```

5. 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,

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;
```

6. 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;
```

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

8. 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;
```

9. 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;
```

10. 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 (VHDL) =>

```
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;
```

11. 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,
                "  %%]" );
```