THE ConPro^2 BOOK

Rule-Based Mapping of an Imperative Programming Language to RTL for Higher-Level-Synthesis Using Communicating Sequential Processes

Dr. Stefan Bosse

BSSLAB • Independent Research Laboratory • Bremen

ConPro^2 • **V2.1 • 22.1.2010**

THE ConPro^2 BOOK

Table of Content

1 Introduction and Overview

The ConPro programming language, a new enhanced imperative programming language is mapped to Register-Transfer-Logic using a higher-level-synthesis approach performed by the synthesis tool ConPro. In contrast to other approaches using modified existing software languages like C, this language is designed from scratch providing a consistent model for both hardware design and software programming. The programming model and the language provide parallelism on control path level using a multi-process model with communicating sequential processes (CSP), and on data path level using bounded program blocks. Each process is mapped to a Finite-State-Machine and is executed concurrently. Additionally, program blocks can be parameterized and can control the synthesis process (scheduling and allocation). Synthesis is based on a non-iterative, multi-level and constraint selective rule-set based approach, rather than on a traditional constrained iterative scheduling and allocation approach. Required interprocess communication is provided by a set of primitives, entirely mapped to hardware, already established in concurrent softwareprogramming (multi-threading), implemented with an abstract data type object model and method-based access. It is demonstrated that this synthesis approach is efficient and stable enough to create complex circuits reaching the million gates boundary.

...

Multi-Process-Architecture

...

Scheduling & Allocation

...

Rule Based Scheduling

...

3 Language Specification

Introduction

The ConPro programming language consist of two classes of statements: 1. process instructions mapped to FSM/RTL, and 2. type, and object definitions. It is an imperative programming language with strong type checking. Imperative programs which describe algorithms that execute sequentially from one statement to the next, are familiar to most programmers. But beneath algorithmic statements the programming language must provide some kind of relation to the hardware circuit synthesized from the programming level.

The syntax and semantics of the programming language is consistently designed and mostly self-explanatory, without cryptic extensions, required in most hardware C-derivates, like Handel-C or System-C, providing easy access to digital circuit development, also for software programmer.

Additionally there is a requirement to get full programmability of the design activities themselves, that means of the synthesis process, too **[RU87]**, implemented here with constrained rules on block level, providing fine-grained control of the synthesis process. The synthesis process can be parameterized by the programmer globally or locally on instruction block level, for example scheduling and allocation.

The set of objects is splitted into two classes: 1. data storage type set $\mathfrak{R}$, and 2. abstract data object type set (ADTO) Θ, with a subset of the IPC objects $\mathfrak{I}$. Though it is a traditional imperative programming language, it features true parallel programming both in control and data path, explicitly modelled by the programmer.

Processes provide parallelism on control path level, whereby arbitrary nested bounded blocks inside processes provide parallelism on data path level.

There is an extended interface to connect to external hardware objects.

Modules and Processes

A ConPro design hierarchy consists of a behavioural module level (Module-B) containing global (shared) objects and processes. A module is mapped to a circuit component with a toplevel hardware port interface. Structural modules (Module-S) can be composed of behavioural modules with optional internal interconnect components. Each process (process level) consists of local (non-shared) objects and a process instruction sequence, specifying the control and data flow. Abstract object types are implemented with abstract object modules (Module-O).

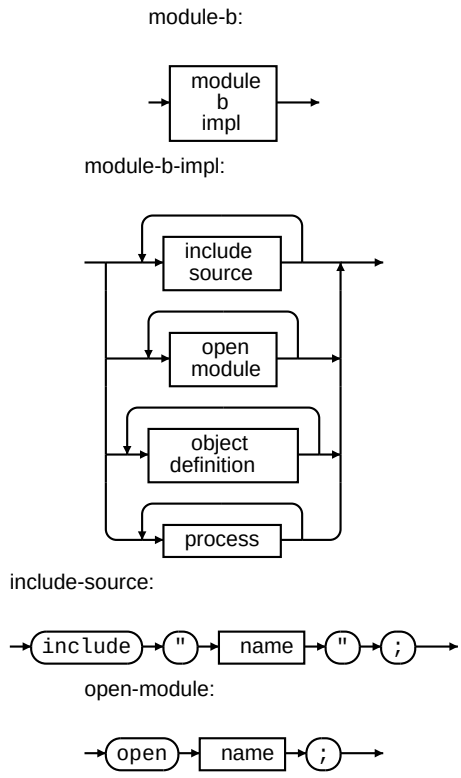
Behavioural Modules

Behavioural modules implement objects and processes. A behavioural module is defined by the source code file itself. Actually there is only one module

hierarchy level, the main module. More source code file can be included using the `include` statement.

There are two kinds of modules: a module embedding objectes and processes, and modules providing access and implementation of abstract data type objects (ADTO). These are mainly interprocess communication and synchronization objects, for example mutex, sempahore, timer and some communication links. Each ADTO module to be used must be opened using the `open` statement.

DEFINITION 1: **Formal syntax specification of a behavioural module.**

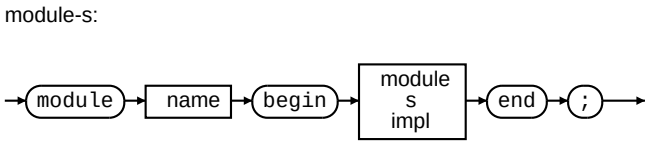


Abstract Object Modules

This module provides access and implementation of abstract data type objects (ADTO). These are mainly interprocess communication and synchronization objects, for example mutex, sempahore, timer and some communication links. Each ADTO module to be used must be opened using the `open` statement. They are defined by the External Module Interface (see section *6.*).

Structural Modules

Structural modules are used to build System-On-Chip (SoC) circuits from behavioural modules.



DEFINITION 2: **Formal syntax specification of a structural module.**

Syntax	Description

`module MS`
`begin`
`  import`
`  component`
`  structtype`
`  mapping`
`end;`
	Defines a new structural module with specified name.

`import MB;` `component C1,C2,..:MB;`	Import of behavioural modules and instantiation of components (ciruits).

`type ICT:{ port...};`
`component IC:ICT :=`
`{`
`  C1.TOP.S1,`
`  C1.TOP.S2,...`
`  C2.TOP.S1,...`
`};`
	Defines and instantiate a port interface of interconnect component with initial signal mapping.

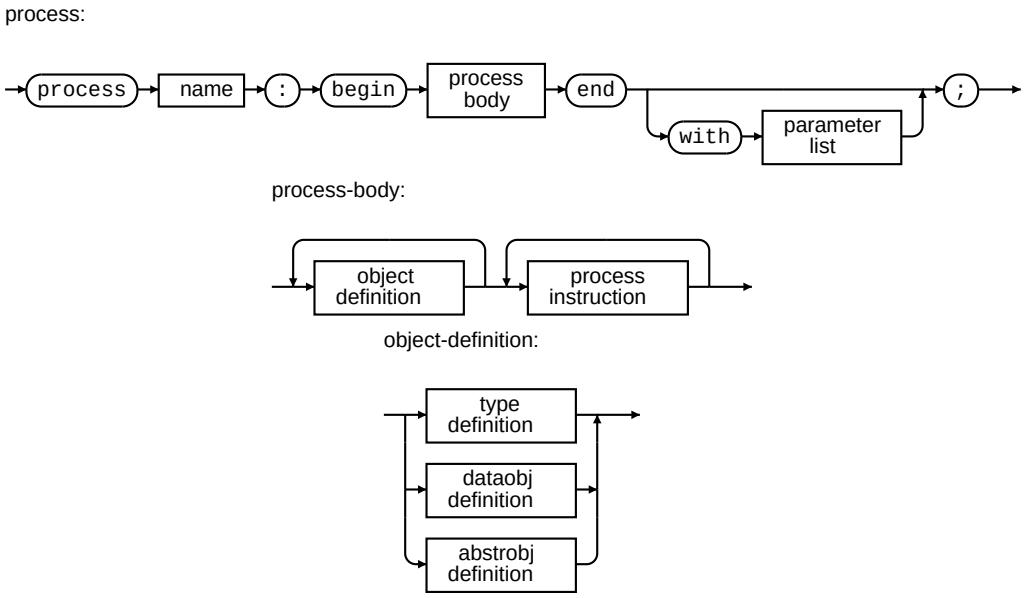
`IC.S1 >> IC.S2;`	Internal interconnect using mapping statements

TABLE 1: **Summary of module-s definition and interconnect.**

EXAMPLE 1: **Example of combined behavioural and structural module definitions. A SoC is composed of two components Lc1 and Lc2, instantiated from behavioural module Link_simu, the actual toplevel module (source file link_simu.cp).**

```
1:  open Core;
2:  open Process;
3:  open Link;
4:  open System;
5:
6:  type dev_type:{
7:    port ln_din: input logic[8];
8:    port ln_din_ack: output logic;
9:    port ln_dout: output logic[8];
10:   port ln_dout_ack: input logic;
11: };
12: component DEV: dev_type;
13: export DEV;
14:
15: object sys: system;
16:   sys.simu_cycles(100);
17:
18: -
19: - Async. Link
20: -
21: object ln: link with datawidth=4;
22:   ln.interface(DEV.ln_din,DEV.ln_din_ack,DEV.ln_dout,DEV.ln_dout_ack);
23: reg xa: int[8];
24: export x,xa;
25:
26: exception Exit;
27:
28:
29: -
30: - Producer-Consumer process
31: -
32: process p1:
33: begin
34:   reg err:bool;
35:   reg d:logic[4];
36:
37:   d ← 1;
38:   try
39:   begin
40:     for i = 1 to 10 do
41:     begin
42:       xa ← 'w',x ← to_int(d);
43:       ln.write(d,err);
44:       if err = true then raise Exit;
45:       xa ← 'r';
46:       ln.read (d,err);
47:       if err = true then raise Exit;
48:       x ← to_int(d),d ← d + 1;
```

```
49:       end;
50:     end
51:     with
52:     begin
53:       when Exit: ln.stop ();
54:     end;
55:     xa ← '.';
56:   end;
57:
58:
59:   process main:
60:   begin
61:     ln.init ();
62:     ln.start ();
63:     p1.start ();
64:   end;
65:
66:   - SoC: two identical blocks Lc1 and Lc2 containing
67:   - one link ln, process p1 and main from this module Link_simu are
68:   - cross linked (Lc1: dout => Lc2: din, Lc2:dout => Lc1:din)
69:   -
70:
71:   module Link2_simu:
72:   begin
73:     -
74:     - Instantiate components, import
75:     - THIS toplevel module.
76:     -
77:     import Link_simu;
78:     component Lc1,Lc2: Link_simu;
79:
80:     -
81:     - Structural Interconnection
82:     -
83:     type l_connect: {
84:       port Lc1_ln_din: output logic[8];
85:       port Lc1_ln_din_ack: input logic;
86:       port Lc1_ln_dout: input logic[8];
87:       port Lc1_ln_dout_ack: output logic;
88:       port Lc2_ln_din: output logic[8];
89:       port Lc2_ln_din_ack: input logic;
90:       port Lc2_ln_dout: input logic[8];
91:       port Lc2_ln_dout_ack: output logic;
92:     };
93:     component L_c: l_connect :=
94:                      {Lc1.DEV.ln_din,
95:                       Lc1.DEV.ln_din_ack,
96:                       Lc1.DEV.ln_dout,
97:                       Lc1.DEV.ln_dout_ack,
98:                       Lc2.DEV.ln_din,
99:                       Lc2.DEV.ln_din_ack,
100:                      Lc2.DEV.ln_dout,
101:                      Lc2.DEV.ln_dout_ack};
102:    -
```



DEFINITION 3: **Formal syntax specification of a process definition.**

```
103:    - Interconnect
104:    -
105:    L_c.Lc1_ln_din << L_c.Lc2_ln_dout;
106:    L_c.Lc1_ln_din_ack >> L_c.Lc2_ln_dout_ack;
107:    L_c.Lc1_ln_dout >> L_c.Lc2_ln_din;
108:    L_c.Lc1_ln_dout_ack << L_c.Lc2_ln_din_ack;
109:  end;
110:
```

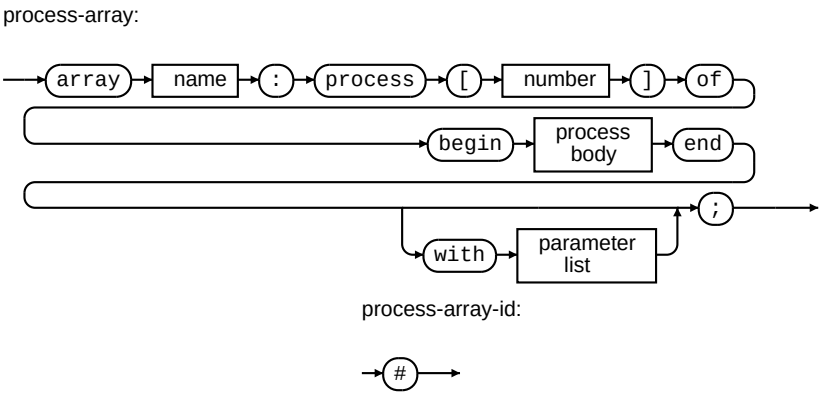
Processes

A process definition consists of a unique process name identifier and the process body. The process body consists of local object definitions (types, data and some abstract objects) and an instruction sequence. Definition **3** shows the formal specification, and table **2** summarizes and explains different process defintions and management.

Array of processes can be defined using the array definition statement. To distinguish processes of an array, a process can access its unique process identifier number (array index). Definition **4** shows the formal syntax definition for process array definitions.

After definition, a process (itself an ADTO) must be started using the `start` or `call` method by another process (see table **3**), usually the main process (named `main`), which is the only process started some clock cycles after system reset.

EXAMPLE 2: **An example showing process definitions and process arrays.**



DEFINITION 4: **Formal syntax specification of a process array definition.**

Syntax	Description

`process pname`
`begin`
`  definitions`
`  instructions`
`end;`
	Defines a new process with specified name.

`process pname`
`begin`
`  definitions`
`  instructions`
`end with param=value;`
	Defines a new process with specified name. Additional parameter settings are applied.

`array paname:`
`  process[size] of`
`begin`
`  definitions`
`  instructions`
`end;`
	Defines a new array of *size* different processes. Optional parameter settings can be applied.

`pname.start ();`
`pname.stop ();`
`pname.[i].start ();`
`myid←#;`
	Process control using ADTO methods and array selectors.

TABLE 2: **Summary of process definitions and access.**

Method	Description
`pname.start ();`	Start process `pname`. The caller process will not be blocked.
`pname.stop ();`	Stop process `pname`. The caller process will not be blocked.
`pname.call ();`	Call process `pname`. The caller process will be blocked untill the called process reaches it end state.

TABLE 3: **Summary of process management.**

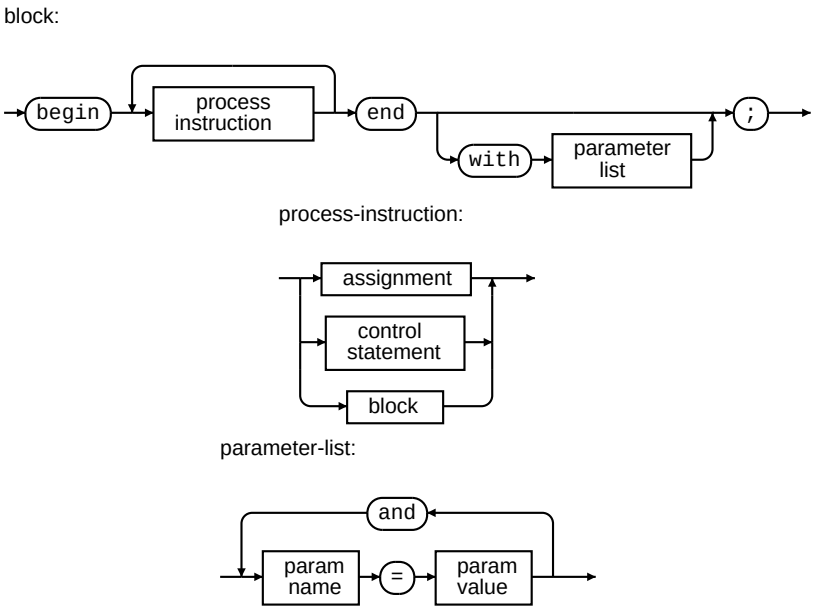
```
1:  -
2:  - Dining philosophers problem using semaphores.
3:  -
4:  - Five philosophers sit around a circular table. Each philosopher
spends
5:  - his life alternately thinking and eating. In the center of the
table is a
6:  - large platter of spaghetti. Each philosopher needs two forks two
eat. But
7:  - there are only five forks for all. One fork is placed between each
pair
8:  - of philosophers, and they agree that each will use only the forks
to the
9:  - immeadiate left and right.
10: -
11: - [Andrews 2000, Multihtreaded, Parallel, and Distributed
Programming]
12: -
13:
14: open Core;
15: open Process;
16: open Semaphore;
17: open System;
18: open Event;
19: object sys: system;
20:   sys.simu_cycles (500);
21: object ev: event;
22: array eating,thinking: reg[5] of logic;
23: export eating,thinking;
24:
25: array fork: object semaphore[5] with depth=8 and init=1 and
26:                                        scheduler="fifo";
27:
28: process init:
29: begin
30:   for i = 0 to 4 do
31:     fork.[i].init ();
32:   ev.init ();
33: end;
34:
35: function eat(n):
36: begin
37:   begin
38:     eating.[n] <- 1;
39:     thinking.[n] <- 0;
40:   end with bind;
41:   wait for 5;
42:   begin
43:     eating.[n] <- 0;
44:     thinking.[n] <- 1;
45:   end with bind;
46: end with inline;
47:
48: array philosopher: process[5] of
```

```
49:  begin
50:    if # < 4 then
51:    begin
52:     ev.await ();
53:     always do
54:     begin
55:       - get left fork then right
56:       fork.[#].down ();
57:       fork.[#+1].down ();
58:       eat (#);
59:       fork.[#].up ();
60:       fork.[#+1].up ();
61:     end;
62:    end
63:    else
64:    begin
65:     always do
66:     begin
67:       - get right fork then left
68:       fork.[4].down ();
69:       fork.[0].down ();
70:       eat (#);
71:       fork.[4].up ();
72:       fork.[0].up ();
73:     end;
74:    end;
75:  end;
76:
77:  process main:
78:  begin
79:    init.call ();
80:    for i = 0 to 4 do
81:    begin
82:      philosopher.[i].start ();
83:    end;
84:    ev.wakeup ();
85:  end;
```

Block Structures

The ConPro architecture is structured into modules and processes on top level. Each process consists of at least one block: the process body, which binds instructions (and definitions) to the named process environment. Blocks, either on same level or nested, can be used to bind instructions to a specific environment with different scheduling and allocation strategies. Blocks can be parameterized, shown in definition **5** and table **4**. Additionally, blocks are used in control statements.

In contrast to common imperative programming languages like C, there are no object defintions and hidden locality inside a block, except in function and process body blocks.



DEFINITION 5: **Formal syntax specification of a block definition.**

Predefined block parameters and their possible values are listes in table **5**. A parameter of type boolean can be used without a value. This assigns automatically the true value to the parameter.

Data Objects and Data Types

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 object definition is shown in definition **6**, and both data and object types are described in tables **6** and **7**.

Syntax	Description

`begin`
`  instructions`
`end;`
	Defines a new block containing instructions.

`begin`
`  instructions`
`end with param=value;`
	Defines a new block containing instructions with additional parameters.

`if expr then ...`
`while expr do ...`
`for i = a to b do ...`
`begin`
`  instructions`
`end;`
	Blocks used with control statements (conditional branch block, conditional loop block, counting loop block)

TABLE 4: **Summary of block definitions.**

Parameter	Values	Description
bind	true,false	Instructions (only assignments) are bound to one time unit.
unroll	true,false	Unrolls a counting for-loop.
schedule	"SL[,SL]" SL=basicblock | refstack | expr	Specifies scheduling strategy and optimization (basic block, reference stack and expression scheduler).
expr	"EXPR" EXPR=flat | binary | shared	Specifies expression model (flat is non-shared, ALU is shared resource model).
inline	true,false	Defines a function either be placed inline (macro substitution) or implemented as a shared function block.

TABLE 5: **Summary of block parameters. Highlighted values are default settings of each parameter.**

A data object $\mathfrak{R}$ is specified by a type cross product of ($\alpha \times \beta$).

There are local and global data objects. Global data objects are shared by several processes concurrently and require an access scheduler providing a mutual exclusion lock.

Shared Objects and Scheduler

Access of shared objects muts be guarded inherently by a mutex using a mutual exclusion scheduler. This scheduler is responsible to serialize concurrent access.

There are two different scheduler available:

Static Priority Scheduler [default]

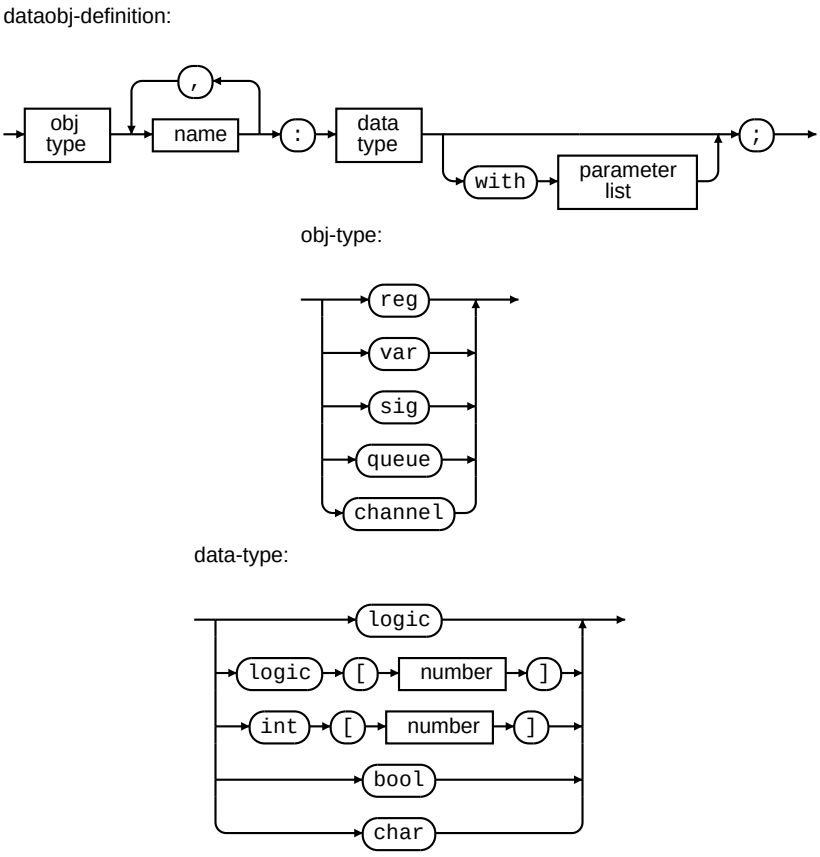
This is the simplest scheduler and requires the lowest amount of hardware resources. Each process ever accessing the resource gets a unique ordered priority. If there are different processes accessing the resource concurrently, the scheduler always grants access to the process with the highest priority. There is a risk of race conditions using this scheduling strategy.

Commonly, the order of processes appearing in the source code determines their priority: the first process gets the highest priority, the last the lowest.

A scheduled access requires at least two clock cycles.

Dynamic FIFO Scheduler

The dynamic scheduler provides fair scheduling using a process queue.



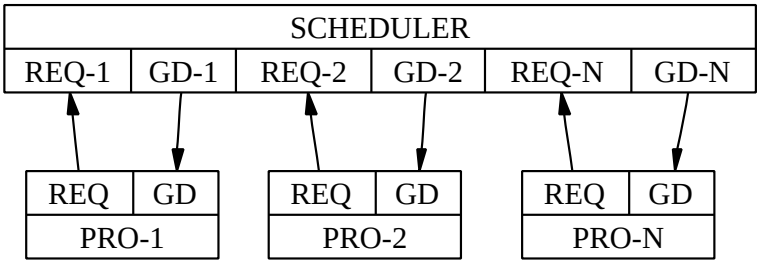
DEFINITION 6: **Formal syntax specification of a data object definition.**

Keyword	Name	Description
`reg`	Register	Single register for CREW-access-behaviour
`var`	Variable	Variable embedded in RAm block with EREW-access-behaviour
`sig`	Hardware Signal	Interconnection signal, synthesizing always to a wire.
`queue`	Data Queue	Buffered Queue (FIFO). Depth and data type can be specified.
`channel`	Data Channel	Buffered (Queue with depth=1) or unbuffered (withour register) synchronized data exchange

TABLE 6: **Summary of provided set of data (core) objects $\mathfrak{R}$, TYPE α.**

Keyword	Type	Description
`logic`	Bit Value	Single bit type. This data type is commonly synthesized to VHDL `std_logic` type. Value set: L={0,1,H,L,Z}
`logic[size]`	Unsigned Bit Vector	Bit vector type of data width size. This data type is commonly synthesized to VHDL `std_logic_vector(size-1 downto 0)` type. Value set: L={0,1,H,L,Z}
`int[size]`	Signed Integer Vector	Signed integer type of data width size (including sign bit). This data type is commonly synthesized to VHDL `signed(size-1 downto 0)` type. Value set: I={-n,...,-1,0,1,...n-1}
`bool`	Boolean	Boolean type. This data type is commonly synthesized to VHDL `std_logic` type Value set: B={0,1,true,false}
`char`	Character (one byte)	8-bit vector (unsigned). This data type is commonly synthesized to VHDL `std_logic_vector(7 downto 0)` type. Value set: C= {'0'...'9','A'...'Z',...} ∪ I

TABLE 7: **Summary of provided set of data (core) types, TYPE β.**



GRAPH 1: **Scheduler Block Architecture**

```
if REQ-1 ∧ ¬LOCKED then
  LOCKED←TRUE;
  Raise ACT; Start Service for Process 1;
else if REQ-2 ∧ ¬LOCKED then
  LOCKED←TRUE;
  Raise ACT; Start Service for Process 2;
...
else if ACK ∧ REQ-1 ∧ LOCKED then
  Release GD-1;
  LOCKED ← FALSE;
else if ACK ∧ REQ-2 ∧ LOCKED then
  Release GD-2;
  LOCKED ← FALSE;
...
```

ALGORITHM 1: **Static Priority Scheduler: From/to process i:{REQ-i,GD-i}, from/to shared resource block:{ACT,ACK}. A process-i request activates REQ-i, and if the resource is not locked, the request is granted to the next process in the if-then-else cascade. If the request is finished, then ACK is activated and releases the locked object and releases GD-i for this respective process indicating that the request is finished.**

Each process wanting to access the resource is stored in this FIFO ordered queue. The oldest one in the queue is choosen by the scheduler if the resource is released by the previous owner. The dynamic scheduler avoids race conditions, but requires much more hardware resources.

Registers

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 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 section *3.4.1.* for details.

Registers can be read in expressions, and can be wrote in assignments,

```
if REQ-1 ∧ LOCKED=[] ∧ ¬PRO-1-LOCKED then
  LOCKED ← [PRO-1];
  PRO-1-LOCKED ← TRUE;
  OWNER←PRO-1;
  Raise ACT; Start Service for Process 1;
else if REQ-2 ∧ LOCKED=[] ∧ ¬PRO-2-LOCKED then
  LOCKED ← [PRO-2];
  PRO-2-LOCKED ← TRUE;
  OWNER←PRO-2;
  Raise ACT; Start Service for Process 2;
...
else if REQ-1 ∧ LOCKED ≠ [] ∧ ¬PRO-1-LOCKED then
  LOCKED ← LOCKED @ [PRO-1]; Append Process 1 to Queue
  PRO-1-LOCKED ← TRUE;
else if REQ-2 ∧ LOCKED ≠ [] ∧ ¬PRO-2-LOCKED then
  LOCKED ← LOCKED @ [PRO-2]; Append Process 2 to Queue
  PRO-2-LOCKED ← TRUE;
...
else if REQ-1 ∧ Head(LOCKED)=PRO-1  ∧ OWNER≠PRO-1 then
  Raise ACT; Start Service for Process 1;
  OWNER←PRO-1;
else if REQ-2 ∧ Head(LOCKED)=PRO-2 ∧ OWNER≠PRO-2 then
  Raise ACT; Start Service for Process 2;
  OWNER←PRO-2;
...
else if ACK-1 ∧ Head(LOCKED)=PRO-1 then
  Release GD-1;
  PRO-1-LOCKED ← FALSE;
  OWNER←NONE;
  LOCKED ← Tail(LOCKED);
else if ACK-2 ∧ Head(LOCKED)=PRO-2  then
  Release GD-2;
  PRO-2-LOCKED ← FALSE;
  OWNER←NONE;
  LOCKED ← Tail(LOCKED);
...
```

ALGORITHM 2: **Dynamic Queue Scheduler: From/to process i:{REQ-i,GD-i}, from/to shared resource block:{ACT,ACK}. A process-i request activates REQ-i, and if the resource queue is empty or this process is at head of the queue, the request is granted to the process in the if-then-else cascade. If the request is finished, then ACK is activated and removes the process from the resource queue and releases GD-i for this respective process indicating that the request is finished.**

Syntax	Description
`reg rname: DT[N];`	Defines a new register of specified type and width.
`reg rname: stype;`	Defines a register of a structure type.
`reg rname: DT[N];` `with param=value;`	Defines a register of specified type with additional parameters.
`if rname = expr then ...` `rname ← expr;`	Register used in expression and assignment.

TABLE 8: **Summary of register definitions and access in expressions.**

summarized in table **8** and shown in example **3**.

EXAMPLE 3: **Example of global and local register definitions and register access.**

```
1:  reg xs: int[10];
2:  reg ys: logic[11];
3:  process p1:
4:  begin
5:    reg x: int[8];
6:    x ←0;
7:    for i = 1 to 5 do
8:    begin
9:      x ← xs + x;
10:   end;
11:   ys ← to_logic(x);
12: end;
13: process p2:
14: begin
15:   xs ← xs - 1;
16:   while ys < 10 do
17:   begin
18:     xs ← xs + to_int(ys);
19:   end;
20: end;
```

Variables

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. A variable provides always exclusive mutex guarded read and write access.

Different variables concerning both data type and data width are stored in one memory block, which is mapped to a RAM. Address management is

Syntax	Description
`var rname: DT[N];`	Defines a new register of specified type and width.
`var rname: stype;`	Defines a register of a structure type.
`var vname: DT[N];` `with param=value;`	Defines a register of specified type with additional parameters.
`if vname = expr then ...` `vname ← expr;`	Variable used in expression and assignment.

TABLE 9: **Summary of variable definitions and access in expressions.**

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 of 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.

There are two different schedulers available: static priority and dynamic FIFO scheduled. See section *3.4.1.* for details.

Variables can be read in expressions, and can be wrote in assignments, shown in example **4**.

EXAMPLE 4: **Example of variable definitions and variable access.**

```
1:  block ram1,ram2;
2:  var xs: int[10] in ram1;
3:  var ys: logic[11] in ram1;
4:  var zs: logic[11] in ram2;
5:  process p1:
6:  begin
7:    reg x: int[8];
8:    x ←0;
9:    for i = 1 to 5 do
10:   begin
11:     x ← xs + x;
12:   end;
13:   ys ← to_logic(x);
14:   zs ← ys * 2;
15: end;
16: process p2:
17: begin
```

```
18:     zs ← ys;
19:     ys ← zs - 1;
20:     while ys < 10 do
21:     begin
22:       xs ← xs + to_int(ys);
23:       ys ← zs * 2;
24:     end;
25:   end;
```

Signals

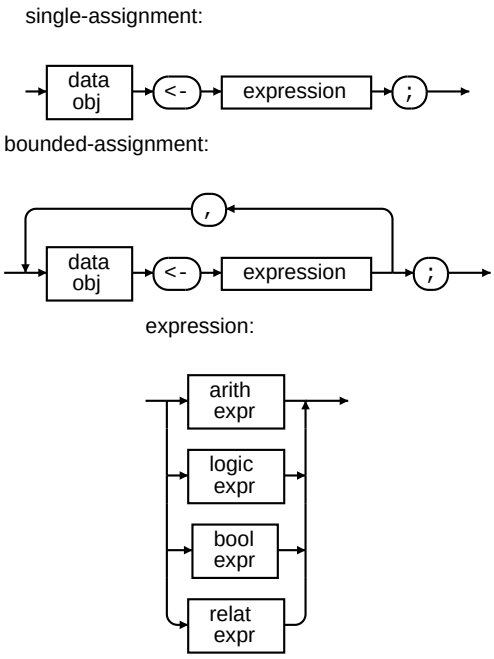
Signals are interconnection elements without a storage model. They provide an interface to external hardware blocks. Signals are used in component structures, too.

Signals can be read in expressions, and a value can be assigned in assignments, shown in example **5**. Reading a signal returns the actual value of a signal, but writing to a signal assigns a new value only for the time the assignment is active, otherwise a default values is assigned to the signal. Therefore, there may be only one assignment for a signal.

Signals can be used in conjunction with `wait` and `wait-for` control statements.

EXAMPLE 5: **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];
13:  signal s2: logic;
14:  reg xs: int[8];
15:  export s1,s2;
16:  process p1:
17:  begin
18:    reg x: int[8];
19:    x ←0;
20:    stat_leds[0] ← 1;
21:    for i = 1 to 5 do
22:    begin
23:      x ← x + s1;
```



DEFINITION 7: **Formal syntax specification of single and bounded assignments, executed concurrently.**

```
24:     end;
25:     xs ← x;
26:     stat_leds[0] ← 0;
27:   end;
28:   process p2:
29:   begin
30:     stat_leds[1] ← 0;
31:     for i = 1 to 5 do
32:     begin
33:       wait for DEV.act = 1 with s2 ← 1;
34:       DEV.we ← 1,DEV.wr ← to_logic(xs);
35:     end;
36:     stat_leds[1] ← 0;
37:   end;
```

Expressions and Assignments

Assignments transfer data from expression results to storage elements and they can be used with register, variables, queue and channels objects (with special treatment for signals). An assignment finishing with a semicolon at the end is usually executed within one time unit.

A group of data independent assignments with up to one guarded object access can be executed within one time unit concurrently. All assignments of this group are separated by a colon, shown in definition **7**. Tables **10**, **11**, **12** and **13** summarize the available operations.

Data Type	Decsription
`a + b`	Addition
`a - b`	Subtraction
`-a`	Negation
`a * b`	Multiplication
`a / b`	Division
`a asl n`	Shifts operand `a` `n` digits left.
`a asr n`	Shifts operand `a` `n` digits right.
`2 ^ n`	2^n, only constant values `n`
`n ~ 2`	$\log_2(n)$, only constant values

TABLE 10: **Summary of available arithmetic operators. Supported data type set: {logic[N],int[N],char}**

Data Type	Decsription
`a lor b`	Or-operation for each bit of `a` and `b`.
`a land b`	And-operation for each bit of `a` and `b`.
`a lxor b`	Xor-operation for each bit of `a` and `b`.
`lnot a`	Negation
`a lsl n`	Shifts operand `a` `n` digits left.
`a lsr n`	Shifts operand `a` `n` digits right.

TABLE 11: **Summary of available logic operators. Supported data type set: {logic[N],int[N],char}**

Data Type	Decsription
`a bor b`	Boolean or-operation
`a band b`	Boolean and-operation
`a bxor b`	Boolean xor-operation
`bnot a`	Negation

TABLE 12: **Summary of available boolean operators. Supported data type: bool**

Data Type	Decsription
`a < b`	Lower-than compare
`a <= b`	Lower-than-equal compare
`a > b`	Greater-than compare
`a >= b`	Greater-than-equal compare
`a = b`	Equal compare
`a <> b`	Not-Equal compare

TABLE 13: **Summary of available relational operators. Supported data type: {logic[N],int[N],char}**

Types

Objects are specified by their object type α and a data type β. There are data type and abstract type objects. User defined types providing oroduct types using arrays and structures and restricted sum types with enumerated symbolic name lists are available.

Data Object Types

Data objects (including IPC objects queue and channel) can be used directly in expressions and assignments.

Data Object Types `TYPE a'={reg,var,sig,queue,channel}`

Data Object Type	Decsription
`reg`	Register
`var`	Variable ≡ memory block
`sig`	Signal
`queue`	Queue (IPC)
`channel`	Channel (IPC)

TABLE 14: **Summary of available Data Object Types.**

Data Types

Table **15** lists all available data types which can be used with expressions, function arguments and assignments. These data types can be applied to a subset of available object types (data and some interprocess communication objects):

See also tables **6** and **7** and definition **6** for more informations.

Data Types `TYPE β'={logic,int,char,bool}`

Data Type	Decsription
`logic`	Single logic bit
`logic[N]`	Logic vector of width N bit
`int[N]`	Signed integer of width N bit
`char`	Character (≡ logic[8])
`bool`	Boolean (≡ logic)

TABLE 15: **Summary of available Data Types.**

Abstract Data Object Types

Table **16** summarizes all abstract object types $\mathfrak{I}$ used for interprocess communication and synchronization (IPC) and additional ones used for communication and storage. Implementation and Interface of each ADTO are defined using the External Module Interface (EMI). More ADT objects can be added and embedded in a ConPro design using this interface. ADTOs can only be accessed with their respective methods.

Abstract Data Type	Decsription
`mutex`	IPC: mutual exclusion
`semaphore`	IPC: sempahore
`barrier`	IPC: barrier
`event`	IPC: event
`timer`	IPC: periodical event timer
`ram`	Storage: RAM blocks (external and internal)
`random`	Pseudo-Random-Generators
`latch`	Asynchronous Latch Memory
`uart`	COMM: Serial receiver and transmitter
`process`	Process object

TABLE 16: **Available Abstract Data Objects Types.**

Product Types: Array

Product Types: Structure

New user defined types can be used to aggregate objects with heterogene types and data widths. In contrast to arrays, a new type structure must be defined first without creation of any data object. After type definition data objects of this type can be created (instantiated). Supported data object types are: signal, register, variable, queue, channel, component.

Statement	Decsription
`array A:` `  OT[N] of DT;`	Define storage array of size N with object type OT and data type DT.

`array A:`
`  OT[N] of DT`
`  with PARAMS;`
	Define storage array of size N with object type OT and data type DT and parameter settings.

`array A:`
`  object obj[N]`
`  with PARAMS;`
	Define abstract object array of size N. Optional object parameter settings require prefixed ADTO module selector.

`array A:`
`  process[N] of`
`begin`
`  B`
`end;`
	Defines a new array of N different processes. Optional process block parameters can be applied.

TABLE 17: **Summary of array definitions.**

Structure Subclasses

There are three different subclasses of structures for different purposes:

Multi-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 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, and no object types. A structure type binds N 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 an uppercase

Statement	Decsription

type ST: {
  e1: DT;
  e2: DT;
  ...
};
	Defines a new structure type with data type DT specification for each element..

type CT: {
 port e1: DIR DT;
 port e2: DIR DT;
  ...
};
	Defines a new component structure type with data type DT and signal direction DIR specification for each element.

type BT: {
  e1: BN;
  e2: BN;
  ...
};
	Defines a new bit structure type with bit-index BW specification for each element.

reg S: ST	Defines a new storage object of type register.
OT S: ST	Defines a new object of type OT (sig,reg,var,queue,channel,component).
ST.e1 ← ST.e1	Access of structure element objects (write and read).

array AS:
  OT[N] of ST
  with PARAMS;
	Defines a new array of N different structure objects. Optional process block parameters can be applied.

TABLE 18: **Summary of structure type definition, data object definitions and structure access.**

letter. Elements of a structure can be accessed with the object and element name concatenated with a dot.

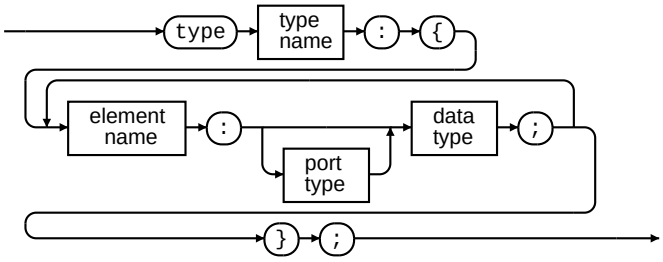
In the case the object type of a structure is a register, just N independent registers are created. In the case of a variable type, N objects are stored into a RAM block.

Arrays from structure types can be created. For each structure element a different array is created.

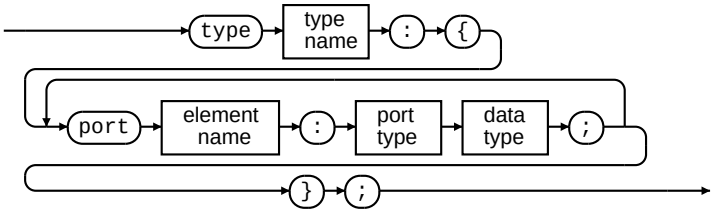
Hardware component port types are defined with structures, too, with the difference that for each structure element the direction of the signal must be specified. Some care must be taken for the direction: if the component is in lower hierarchical order (an embedded external hardware component), the direction is seen from the external view of the hardware component. If the component is part of the toplevel port interface of a ConPro module, it must be seen from the internal view.

DEFINITION 8: **Formal syntax definition for structure type definition, object definition and structure object access.**

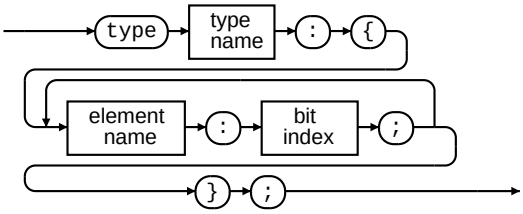
type-definition:



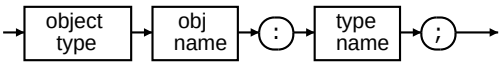
component-type-definition:



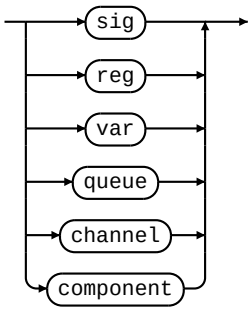
bit-type-definition:



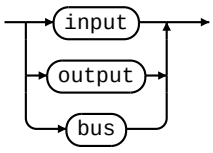
object-definition:

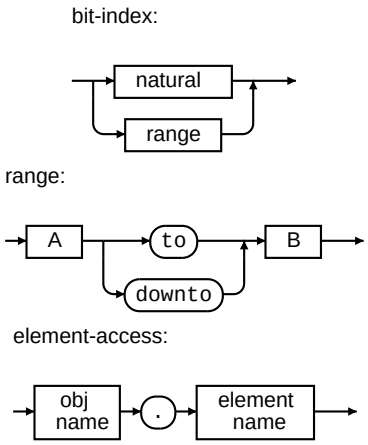


object-type:



port-type:





EXAMPLE 6: **Structures with register, variable and component object type.**

```
- 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];
```

Statement	Decsription
`exception E;`	Define a new exception type E.
`raise E;`	Raise exception E. The control flow is directed to exception handler environment, if any.

`try B with`
`begin`
`  when E1: B1;`
`  when E2: B2;`
`  ...`
`end;`
	Defines an excpetion handler environment. An exception raised in B is caught by a particular case in the when list which executes the particluar instruction (block) B..

TABLE 19: **Summary of array definitions.**

```
  };
  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;
```

Sum Types: Enumeration

...

Exceptions

Exceptions provide the only way to leave a control structure, for example loops, conditional branches or functions itself. Exception are abstract signals, which can be raised anywhere and caught within a try-with exception handler environment, either within the process/function where the execption was raised, or outside. Thus exceptions are automatically propagated along a call path of processes and functions using exception state registers if they are not caught within the raising process/function.

Control Statements

Counting for-Loop

...

Conditional while-Loop

...

Endless always-Loop

...

Blocking wait-for-Loop

...

Conditional if-else-Branch

...

Multicase match-Branch

...

Exception Handler

...

Functions

A function definition consists of a unique function name identifier, the function application interface, and the function body. The function body consists of local object definitions (types, data and some abstract objects) and an instruction sequence. Definition **9** shows the formal specification, and table **20** summarizes and explains different function defintions and function application (calling).

The function application interface specifies function parameters with theit name and type. Function can return values. If there is no value returned, the function behaves like a procedure.

Functions can be used within simple assignments, but not within expressions. If a function returns more than one value, a tuple must be used on the left hand side of the assignment.

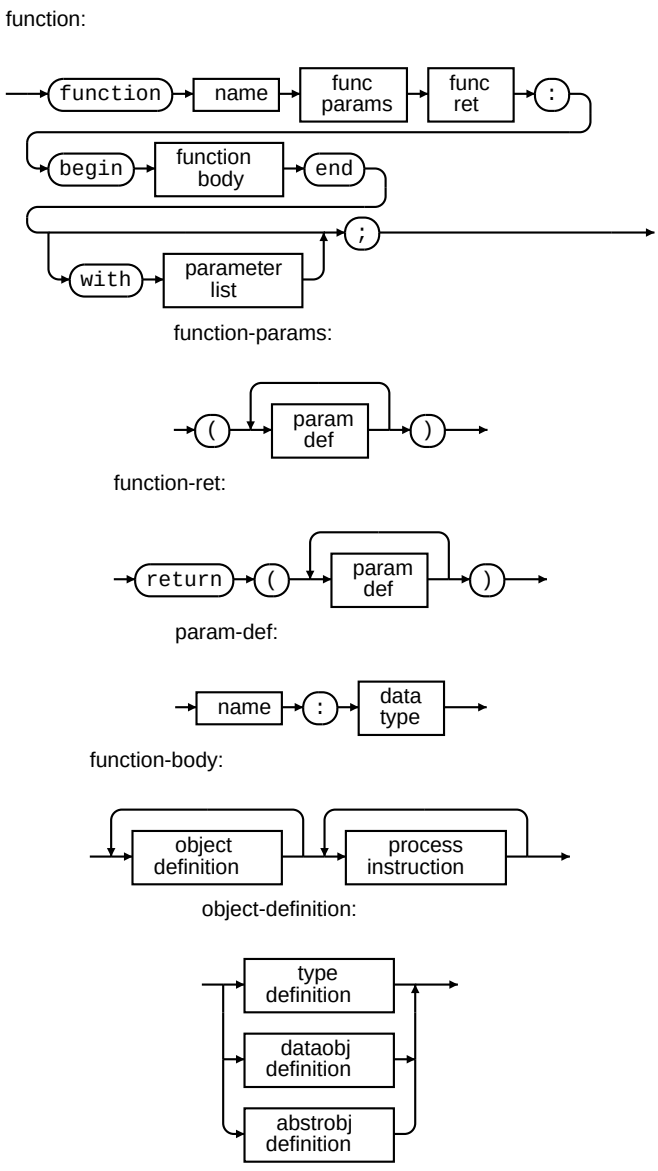
Each function parameter and the set of return value paramters are handled like registers. Only call-by-value semantic is supported. Values of function arguments are copied to the respective parameters on function call, and return values are copied after function call has finished. Within the function body, all parameters and the (named) return parameter can be used in assignments and expression like any other register. There is no return statement. The last value assigned to the return parameter is automatically returned.

There are two different types of functions: inlined and shared. The inlined function type is handled likle a macro definition. Each time a function is

applied (called), the function call is replaced by the function body, and all function parameters are replaced by the function arguments (including return value parameters).

Shared functions are implemented using the process model using the call method, and with an additional function call wrapper. Each time a shared function is called the argument values are passed to the function parameters (global registers), and the return value(s) are passed back, if any.

DEFINITION 9: **Formal syntax specification of a function definition.**



Syntax	Description

`function pname`
`(x:int[8],n:bool):`
`begin`
`  definitions`
`  instructions`
`end;`
	Defines a new procedure with specified name (no return value).

`function fname`
`(x:int[8],n:bool)`
`return (a:int[4]):`
`begin`
`  definitions`
`  instructions`
`end with param=value;`
	Defines a new function with specified name. Additional parameter settings are applied (`inline`: inlined function macro).

`pname();`
`pname(i,1);`
`y ← fname(x);`
`{y1,y2} ← fname(x);`
	Function and procedure application (call with arguments).

TABLE 20: **Summary of function definitions and application (call).**

EXAMPLE 7: **An example showing function definitions and function application. The first function returns one result value, the second a tuple of two values, assigned a tuple of storage objects (of same type) in line 42.**

```
1:  const div_n: value := 16;
2:  const div_n1: value := 15;
3:  const div2_n: value := 32;
4:  const div2_n1: value := 31;
5:
6:  -
7:  - optimized fast sequentiel division
8:  -
9:  function div (a:logic[div_n],b:logic[div_n]) return(z:logic[div_n]):
10: begin
11:   reg q,b2: logic[div2_n];
12:   reg i: logic[5];
13:   const l0: logic[1] := 0;
14:
15:   q <- a;
16:   b2 <- b lsl div_n;
17:   i <- 0;
18:
19:   while i < div_n do
20:   begin
21:     begin
22:       q <- ((q lsl 1)-b2) lor 1;
23:       i <- i + 1;
24:     end with bind;
25:     if q[div2_n1] = 1 then
```

```
26:         q <- (q + b2) land 0xFFFFFFFE;
27:     end;
28:     z <- q[0 to div_n1];
29:   end;
30:
31:   function swap(a:logic[div_n],b:logic[div_n])
32:            return(c:logic[div_n],d: logic[div_n]):
33:   begin
34:     c ← b, d ← a;
35:   end;
36:
37:   process calc:
38:   begin
39:     reg x,y,z: logic[div_n];
40:     x ← 456, y ← 32;
41:     z ← div(x,y);
42:     {x,y} ← swap(x,y);
43:     z ← div(x,y);
44:   end;
45:
```

I/O: Hardware Port Interface

Components: Interfacing HDL

...

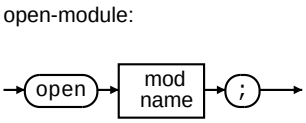
External Module Interface: Embedding HDL

...

Abstract data type objects Θ define objects not directly accessible in expressions like registers (with some exceptions).

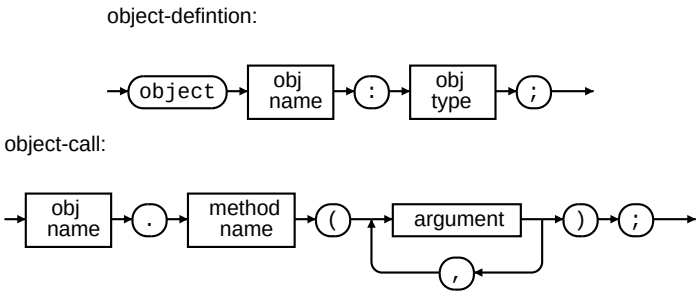
Before abstract objects of a particular type can be used, the appropiate module must be opened first:

DEFINITION 10: **Opening of a module.**



ADT objects can be accessed by their appropiate method set υ={υ_1,υ_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:

DEFINITION 11: **Object method calls. The object must be first created with the object defintion statement.**



Methods which do not expect arguments are applied with an empty argument list (). Table **21** summarizes the statements required for using abstract object types.

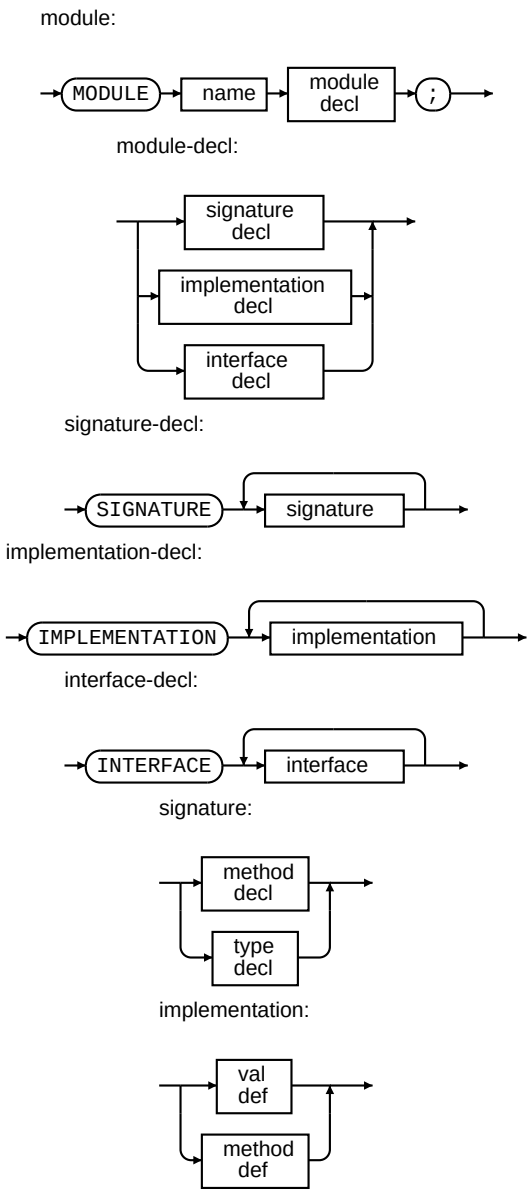
Statement	Decsription
`open Module;`	Open specified ADTO module
`object obj: objtype;`	Defines and instantiates a new object of specified ADT.
`obj.meth`	Object method access using the selector operator

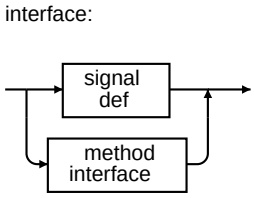
TABLE 21: **Summary of abstract object module inclusion, definition and access.**

Pseudo-Notation

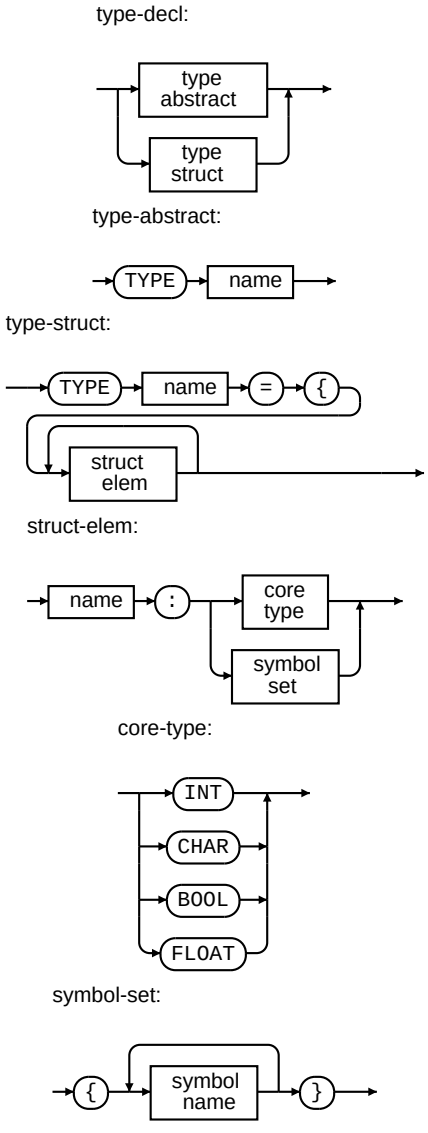
In the following sections, an abstract syntax notation is used to define modules of an particular ADTO type with their supported methods. Defintions **12** to **15** show the formal syntax of this notation. The signature declaration specifies types and the type signature for each method, the interface declaration defines signals and the RTL access for each method. The implementation definition specifie the behaviout for each method.

DEFINITION 12: **Pseudo notation for abstract modules: signature and interface declarations, and implementation defintion.**

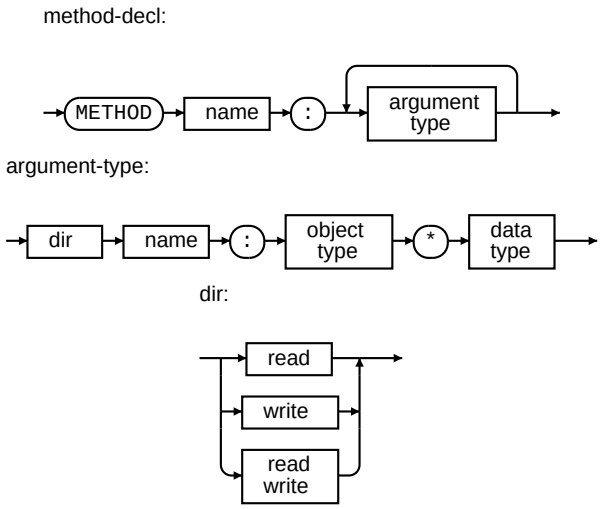




DEFINITION 13: **(Cont.) Pseudo notation for abstract modules: type definition.**



DEFINITION 14: **(Cont.) Pseudo notation for abstract modules: method type signature.**



DEFINITION 15: **(Cont.) Pseudo notation for abstract modules: object- and data types.**

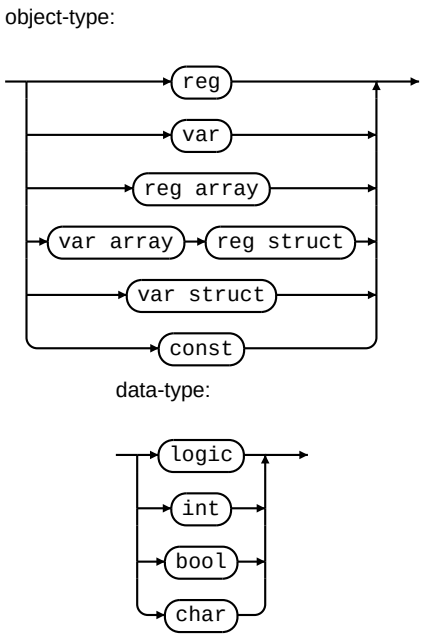


Table **22** explains symbols used in the notation.

Interprocess-Communication

The following ADTOs are available for interprocess communication:

1. Mutex
2. Semaphore
3. Event

Statement	Decsription
Γ	Control Path
Δ	Data Path
Θ	Abstract Object (Type)
υ	Abstract Object Methods
$\Re$	Data Storage Objects
$\Im$	IPC Objects
$\mathtt{D}$	Abstract Computational Objects
$\mathtt{E}$	Abstract External Communication Objects
α	Set of objects (Type)
β	Data Type Set
χ	Synthesis Rule Set
$	Parameter variable
σ	State
σ+	Next State
Θ Ξ	List of (blocked) Processes
$\uparrow$	Output/Write
$\downarrow$	Input/Read
$\perp$	Suspend a process
$\therefore$	Resume a process
$\emptyset$	Empty set or argument list
$a\|_b$	Expression a depends on expression b
`[]`	List/empty List
`a :: l`	Appends element a to head of list l
`l1 @ l2`	Concatenates two lists l1 and l2

TABLE 22: **Summary of symbols used in pseudo notation.**

4. Barrier
5. Timer
6. Queue [core]
7. Channel [core]

Mutex

The Mutex module implements a mutual exclusion lock required for protection of shared resources accessed concurrently. All shared atomic objects (both storage and ADTO) are already implicitly guarded by a mutex lock, serializing the access of the object, including this mutex object, too!

The following object methods are available:

METHODS

lock

A process requests the lock with this method. If the mutex is unlocked (not owned by any other process), the calling process gets the lock and continues operation. If the lock is already owned by another process (mutex is locked), the calling process is blocked untill the mutex owener release the lock using the unlock method. In this case the calling process P is added to a wait-list Ξ.

unlock

The unlock method releases a previously locked mutex. If there are blocked processes awaiting the release. the next waiting process is scheduled and the lock is transferred to this process.

init

Initialize the mutex object.

PARAMETERS

scheduler

Selects static or FIFO scheduler policy.

model

The model parameter provides two different access models: owner: only the owner process of a mutex can unlock the mutex, group: each member of a process group can unlock the mutex.

Definitions **16** and **17** specifiy the signature and the implementation of the mutex module, and **18** the object interface. Two schedulers are available: static priority and dynmaic priority FIFO scheduler.

DEFINITION 16: **Signature of ADTO Module Mutex.**

```
MODULE Mutex
SIGNATURE
  TYPE mutex
  TYPE parameters = {
    scheduler: {static,fifo}
    model: {owner,group}
  }

  METHOD NEW ≡
    object name: mutex [with parameters]: mutex

  METHOD init: ∅
  METHOD lock: ∅
  METHOD unlock: ∅
```

DEFINITION 17: **Implementation of ADTO Module Mutex.**

```
MODULE Mutex
IMPLEMENTATION
  VAR lock: bool
  VAR Θ,Φ: process list
  VAR P,P': process
  OBJ S: scheduler

  METHOD init:
    lock ← false
    ∀ P ∈ Θ: ∴P
    Θ ← []
  METHOD unlock:
    if Θ ≠ [] then
      Θ ← TAIL(Θ), P' ← HEAD(Θ)
      ∴P'
    else
      lock ← false

  METHOD lock:
    P ← SELF
    if lock = false then
      lock ← true
    else
      Θ ← Θ @ [P]
      ⊥P
```

DEFINITION 18: **Interface of ADTO Module Mutex.**

```
MODULE Mutex
INTERFACE
  SIGNALS GD,INIT,UNLOCK,LOCK: logic
  METHOD init:
```

```
  Γ: σ ← σ | GD=1 //
     σ+ | GD=0
  Δ: INIT ← ¬GD
METHOD unlock:
  Γ: σ ← σ | GD=1 //
     σ+ | GD=0
  Δ: UNLOCK ← ¬GD
METHOD lock:
  Γ: σ ← σ | GD=1 //
     σ+ | GD=0
  Δ: UNLOCK ← ¬GD
```

EXAMPLE 8: **A mutex lock is used to protect the access of two global registers x and y.**

```
1:  open Core;
2:  open Process;
3:  open Mutex;
4:
5:  reg x,y: int[10];
6:  object mu: mutex;
7:
8:
9:  process p1:
10: begin
11:   for i = 1 to 10 do
12:   begin
13:     mu.lock ();
14:     x ← y-1;
15:     y ← y+1;
16:     mu.unlock ();
17:   end;
18: end;
19:
20: process p2:
21: begin
22:   for i = 1 to 10 do
23:   begin
24:     mu.lock ();
25:     x ← y+1;
26:     y ← y-1;
27:     mu.unlock ();
28:   end;
29: end;
30:
31: process main:
32: begin
33:   mu.init ();
34:   x ← 100;
35:   y ← 100;
36:   p1.start ();
```

```
37:     p2.start ();
38:   end;
39:
40:
```

Semaphore

The Semaphore module implements a guarded counter ω. A semaphore is used in produce-consumer applications. The counter can be incremented (up operation) and decremented (down operation). The value of the counter may never be negative. Thus a down operation with an actual semaphore value zero blocks the requesting process untill another processes increments the semaphore counter.

The following object methods are available:

METHODS

down

A process decrements the semaphore counter $\omega \leftarrow \omega$-1 iff $\omega > 0$. In the case the counter is alreayd zero, the calling process is blocked untill the semaphore was incremented by another process using the up method. In this case the calling process P is added to a wait-list Ξ.

up

The up method increments the semaphore counter $\omega \leftarrow \omega$+1. If there are blocked processes awaiting an increment the next waiting process is scheduled and ω is still zero (the increment compensates the decrement operation). If there are no blocked processes the sempahore counter is incremented.

init

Initialize the semaphore object with an initial counter value.

PARAMETERS

scheduler

Selects static or FIFO scheduler policy.

depth

The depth parameter specifies the bit width of the semaphore counter, thus the semaphore value can be in the range $[0,2^{\text{depth}}\text{-}1]$

Definitions **19** and **20** specifiy the signature and the implementation of the semaphore module, and **21** the interface. Two schedulers are available: static priority and dynmaic priority FIFO scheduler.

DEFINITION 19: **Signature of ADTO Module Semaphore.**

```
MODULE Semaphore
SIGNATURE
  TYPE semaphore
  TYPE constant: natural
  TYPE parameters = {
    scheduler: {static,fifo}
    depth: [4 to 16]
  }

  METHOD NEW ≡
    object name: semaphore [with parameters]: semaphore

  METHOD init: ↓init-val:(constant | storage-type × integer)
  METHOD down: ∅
  METHOD up: ∅
```

DEFINITION 20: **Implementation of ADTO Module Semaphore.**

```
MODULE Semaphore
IMPLEMENTATION
  VAR lock: bool
  VAL counter: natural [0,2^depth-1]
  VAR Θ,Φ: process list
  VAR P,P': process
  OBJ S: scheduler

  METHOD init:
    lock ← false
    ∀ P ∈ Θ: ∴P
    Θ ← []
    counter ← init-val
  METHOD up:
    if Θ = [] then
      incr counter
      lock ← false
    else
      Θ ← TAIL(Θ), P' ← HEAD(Θ)
      ∴P'

  METHOD down:
    P ← SELF
    if counter > 0 then
      decr counter
    else
      Θ ← Θ @ [P], lock ← true
      ⊥P
```

DEFINITION 21: **Interface of ADTO Module Semaphore.**

```
MODULE Semaphore
INTERFACE
  SIGNAL GD,INIT,UP,DOWN: logic
  SIGNAL WR: logic[depth]
  METHOD init:
```

```
  Γ: σ ← σ | GD=1 //
     σ+ | GD=0
  Δ: INIT ← ¬GD
     WR ← ARG1
METHOD up:
  Γ: σ ← σ | GD=1 //
     σ+ | GD=0
  Δ: UNLOCK ← ¬GD
METHOD down:
  Γ: σ ← σ | GD=1 //
     σ+ | GD=0
  Δ: UNLOCK ← ¬GD
```

EXAMPLE 9: **Semaphores are used to implement a resource negoatation algorithm: Dining philosophers problem using semaphores. Five philosophers sit around a circular table. Each philosopher spends his life alternately thinking and eating. In the center of the table is a large platter of spaghetti. Each philosopher needs two forks two eat. But there are only five forks for all. One fork is placed between each pair of philosophers, and they agree that each will use only the forks to the immeadiate left and right. [Andrews 2000, Multihtreaded, Parallel, and Distributed Programming]**

```
1:  open Core;
2:  open Process;
3:  open Semaphore;
4:  open System;
5:  open Event;
6:  object sys: system;
7:    sys.simu_cycles (500);
8:  object ev: event;
9:
10: array eating,thinking: reg[5] of logic;
11: export eating,thinking;
12:
13: array fork: object semaphore[5] with depth=8 and scheduler="fifo";
14:
15: process init:
16: begin
17:   for i = 0 to 4 do
18:     fork.[i].init (1);
19:   ev.init ();
20: end;
21:
22: function eat(n):
23: begin
24:   begin
25:     eating.[n] ← 1;
26:     thinking.[n] ← 0;
27:   end with bind;
28:   wait for 5;
```

```
29:   begin
30:     eating.[n] ← 0;
31:     thinking.[n] ← 1;
32:   end with bind;
33: end with inline;
34:
35: array philosopher: process[5] of
36: begin
37:   if # < 4 then
38:   begin
39:    ev.await ();
40:    always do
41:    begin
42:      - get left fork then right
43:      fork.[#].down ();
44:      fork.[#+1].down ();
45:      eat (#);
46:      fork.[#].up ();
47:      fork.[#+1].up ();
48:    end;
49:   end
50:   else
51:   begin
52:    always do
53:    begin
54:      - get right fork then left
55:      fork.[4].down ();
56:      fork.[0].down ();
57:      eat (#);
58:      fork.[4].up ();
59:      fork.[0].up ();
60:    end;
61:   end;
62: end;
63:
64: process main:
65: begin
66:   init.call ();
67:   for i = 0 to 4 do
68:   begin
69:     philosopher.[i].start ();
70:   end;
71:   ev.wakeup ();
72: end;
```

Event

The Event module implements an event handler (abstract signal) and is used for process control flow synchronization (control flow boundary). A group of processes can join the event handler and wait for the occurrence of this event. The processes are blocked untill the event occurs. The event is signaled by another process.

The following object methods are available:

METHODS

await
: A process waits for the event associated with the abstract object. The calling process is blocked untill the event occurs, and P is added to a wait-list Ξ.

wakeup
: The event associated with this abstract object is signaled. All blocked processes waiting for this event are released.

init
: Initialize the event object.

PARAMETERS

latch
: The latch=1 parameter setting provides a latched event, which causes if an event occured with empty blocked process list, this event is latched. If a process requests the await method, and the latch is set, it will immediately released.

Definitions **22** and **23** specifiy the signature and the implementation of the event module. The `latch=1` parameter setting provides a latched event, which means if an event occured with empty blocked process list, this event is latched. If a process requests the await method, and the latch is set, it will immediately released.

DEFINITION 22: **Signature of ADTO Module Event.**

```
MODULE Event:
SIGNATURE
  TYPE event
  TYPE parameters = {
    latch: {0,1}
  }

  METHOD NEW ≡
    object name: event [with parameters]: event

  METHOD init: ∅
  METHOD await: ∅
  METHOD wakeup: ∅
```

DEFINITION 23: **Implementation of ADTO Module Event.**

```
MODULE Event:
IMPLEMENTATION
  if $latch=1 then VAR latch: bool
  VAR Θ,Φ: process list
  VAR P,P': process
  OBJ S: scheduler

  METHOD init:
    if $latch=1 then latch ← false
    ∀ P ∈ Θ: ∴P
    Θ ← []
  METHOD wakeup:
    if Θ ≠ [] then
      ∀ P ∈ Θ: ∴P
    Θ ← []
    if $latch=1 then latch ← false
  METHOD await:
    P ← SELF
    if $latch=1 ∧ latch = true then
      latch ← false
    else
      Θ ← Θ @ [P]
      ⊥P
```

DEFINITION 24: **Interface of ADTO Module Event.**

```
MODULE Event
INTERFACE
  SIGNALS GD,INIT,WAKEUP,AWAIT: logic
  METHOD init:
    Γ: σ ← σ | GD=1 //
       σ+ | GD=∅
    Δ: INIT ← ¬GD
  METHOD wakeup:
    Γ: σ ← σ | GD=1 //
       σ+ | GD=∅
    Δ: WAKEUP ← ¬GD
  METHOD await:
    Γ: σ ← σ | GD=1 //
       σ+ | GD=∅
    Δ: AWAIT ← ¬GD
```

EXAMPLE 10: **An event synchronize the control flow of several processes. The processes of array p are started sequentially, but they are all suspended untill the event is signaled by the main process. This happens again in each loop iteration.**

```
1:
2:  open Core;
3:  open Process;
```

```
4:  open Event;
5:  open System;
6:
7:  object sys:system;
8:    sys.sim_cycles(300);
9:
10: object e: event;
11:
12: array d: reg[4] of int[8];
13:
14: export d;
15:
16: array p: process[4] of
17: begin
18:   for i = 1 to 5 do
19:   begin
20:     e.await ();
21:     d.[#] ← # + 1;
22:     d.[#] ← 0;
23:   end;
24: end;
25: process main:
26: begin
27:   e.init ();
28:   for i = 0 to 3 do
29:    p.[i].start ();
30:   for i = 1 to 5 do
31:   begin
32:     wait for 20;
33:     e.wakeup ();
34:   end;
35: end;
36:
```

Barrier

The barrier module implements a self synchonization event handler (abstract signal) and is used for process control flow synchronization (control flow boundary). A group of processes with defined number N can join the barrier and wait for the occurrence of the event. The processes are blocked untill the event occurs. The event is signaled by the last process N joining the barrier. Each time a proces join the barriers, a counter `join` is incremented.

The following object methods are available:

METHODS

await

A process waits for the barrier event associated with the abstract object. The calling process is blocked untill the event occurs, and P is added to a wait-list Ξ. The event happens when `size(Ξ)=join=N`.

init

Initialize the barrier object with size of joining process group. The process group size N is automatically determnined by the number of differernt processes calling the await method.

PARAMETERS

∅

Definitions **25** and **26** specifiy the signature and the implementation of the barrier module, and **27** the process interface.

DEFINITION 25: **Signature of ADTO Module Barrier.**

```
MODULE Barrier:
SIGNATURE
  TYPE barrier

  METHOD NEW ≡
    object name: barrier [with parameters]: barrier

  METHOD init: ∅
  METHOD await: ∅
```

DEFINITION 26: **Implementation of ADTO Module Barrier.**

```
MODULE Barrier:
IMPLEMENTATION
  VAR Θ,Φ: process list
  VAR P,P': process
  OBJ S: scheduler
  VAL size,join: natural

  METHOD init:
   size ← sizeof(Φ_await)
   join ← 0
   ∀ P in Θ: ∴P
   Θ←[]

  METHOD await:
    if (join+1) = size then
      join ← 0
      ∀ P in Θ: ∴P
      Θ←[]
    else
      P ← SELF
      Θ ← Θ @ [P]
      incr join
      ⊥P
```

DEFINITION 27: **Interface of ADTO Module Barrier.**

```
MODULE Barrier
INTERFACE
  SIGNALS GD,INIT,AWAIT: logic
  METHOD init:
    Γ: σ ← σ | GD=1 //
       σ+ | GD=0
    Δ: INIT ← ¬GD
  METHOD await:
    Γ: σ ← σ | GD=1 //
       σ+ | GD=0
    Δ: AWAIT ← ¬GD
```

EXAMPLE 11: **A group of processes defined in array p wait for the barrier b event. The loop iteration of each process starts at the same time.**

```
1:
2:  open Core;
3:  open Process;
4:  open Barrier;
5:  open System;
6:
7:  object sys:system;
8:    sys.sim_cycles(300);
9:
10: object b: barrier;
11:
12: array d: reg[4] of int[8];
13:
14: export d;
15:
16: array p: process[4] of
17: begin
18:   for i = 1 to 5 do
19:   begin
20:     b.await ();
21:     d.[#] ← # + 1;
22:     d.[#] ← 0;
23:   end;
24: end;
25: begin
26:   b.init ();
27:   - d ← 64;
28:   for i = 0 to 3 do
29:    p.[i].start ();
30: end;
31:
```

Timer

The Timer module implements an interval timer and is used for time constrained process control flow synchronization. A group of processes can join

the timer handler and wait for the occurrence of this time event. The processes are blocked untill the event occurs. The event is signaled by the timeout of the timer. The timer can operate one-time and continously.

The following object methods are available:

METHODS

await
A process waits for the timer event associated with the abstract object. The calling process is blocked untill the timer event occurs, and P is added to a wait-list Ξ.

wakeup
Wakeup all blocked processes.

init
Initialize the timer object.

time
Set interval time of timer in nano seconds (or in time unit specified).

start
Start the timer.

stop
Stop the timer.

sig_action
A signal can be used to check the actual state of the timer: enabled or disabled. Can be used in one-shot timer mode to check the timeout. The activity level and the default signal value must be specified, too.

PARAMETERS

time
The timer interval period time in nano seconds.

mode
The mode parameter specifies timer operation: one-time/shot (1) or continously (0).

Definitions **28** and **29** specifiy the signature and the implementation of the timer module.

DEFINITION 28: **Signature of ADTO Module Timer.**

```
MODULE Timer
SIGNATURE
```

```
TYPE timer
TYPE parameters = {
  time: natural
  mode: {0,1}
}

METHOD NEW ≡
  object name: timer [with parameters]: timer

METHOD init: ∅
METHOD await: ∅
METHOD wakeup: ∅
METHOD time: ↓period:natural
METHOD mode: ↓mode:natural
METHOD sig_action: ↓sig:(output signal × logic) ×
                   ↓action_level:logic ×
                   ↓def_level:logic
```

DEFINITION 29: **Implementation of ADTO Module Timer.**

```
MODULE Timer
IMPLEMENTATION
  VAR Θ,Φ: process list
  VAR P,P': process
  OBJ S: scheduler
  VAL time,timer: natural
  VAL mode: natural {0,1}
  VAL enabled: bool
  VAL count,counter: natural

  METHOD init:
    counter ← 0
    count ← clkcycles($time)
    enabled ← false
    mode ← $mode
    ∀ P ∈ Θ: ∴P
    Θ ← []
  METHOD start:
    counter ← count
    enabled ← true
  METHOD stop:
    counter ← 0
    enabled ← false
  METHOD time:
    count ← clkcycles($ARG1)
  METHOD mode:
    mode ← $ARG1
  METHOD wakeup:
    if Θ ≠ [] then
      ∀ P ∈ Θ: ∴P
    Θ ← []
  METHOD await:
    P ← SELF
    Θ ← Θ @ [P]
    ⊥P
  PROCESS timer:
```

```
    ∀ CLK'event:
      if enabled = true then
        if counter = 0 then
          ∀ P ∈ Θ: ∴P
          Θ ← []
          if mode = 0 then
            counter ← count
          else
            enabled ← false
        else
          decr counter
```

DEFINITION 30: **Interface of ADTO Module Timer.**

```
MODULE Timer
INTERFACE
  SIGNAL GD,INIT,WAKEUP,AWAIT,TIME_SET,START,STOP: logic
  SIGNAL TIME: natural
  METHOD init:
    Γ: σ ← σ | GD=1 //
       σ+ | GD=0
    Δ: INIT ← ¬GD
  METHOD start:
    Γ: σ ← σ | GD=1 //
       σ+ | GD=0
    Δ: START ← ¬GD
  METHOD stop:
    Γ: σ ← σ | GD=1 //
       σ+ | GD=0
    Δ: STOP ← ¬GD
  METHOD time:
    Γ: σ ← σ | GD=1 //
       σ+ | GD=0
    Δ: TIME_SET ← ¬GD
       TIME ← index($time,$ARG1) - List index of time value
  METHOD wakeup:
    Γ: σ ← σ | GD=1 //
       σ+ | GD=0
    Δ: WAKEUP ← ¬GD
  METHOD await:
    Γ: σ ← σ | GD=1 //
       σ+ | GD=0
    Δ: AWAIT ← ¬GD
```

EXAMPLE 12: **Timer example.**

```
1:  open Core;
2:  open Process;
3:  open Timer;
```

```
4:  open System;
5:
6:  object sys:system;
7:    sys.sim_cycles(300);
8:
9:  object t: timer;
10:   t.time (1 microsec);
11:
12: array d: reg[4] of int[8];
13:
14: export d;
15:
16: array p: process[4] of
17: begin
18:   d.[#] ← 0;
19:   for i = 1 to 5 do
20:   begin
21:     t.await ();
22:     d.[#] ← # + 1;
23:     d.[#] ← 0;
24:   end;
25: end;
26:
27: process main:
28: begin
29:   t.init ();
30:   t.time (2 microsec);
31:   for i = 0 to 3 do
32:     p.[i].start ();
33:   t.mode (0);
34:   t.start ();
35: end;
36:
```

Queue

The queue is both a core and abstract object data type. Queues can be used directly in expressions (read and write). The queue buffers data words written from processes to the queue for processes requesting a read operation. The data word order is FIFO.

The following object methods are available:

METHODS

read

A process reading a queue is blocked untill at least one data word is available.

write

A process writing to a queue is blocked untill at least one data word cell is free to hold the new value.

unlock

This method releases all blocked processes (both waiting for read and write completion).

PARAMETERS

depth

Number of cell of the data queue: 2^{depth}-1

DEFINITION 31: **Signature of ADTO Module Queue.**

```
MODULE Queue
SIGNATURE
  TYPE queue
  TYPE parameters = {
    depth: natural
  }

  METHOD NEW ≡
    queue name: data-type [with parameters]: object-type

  METHOD unlock: ∅
  METHOD read: ↑data:data-type
  METHOD mode: ↓data:data-type
```

DEFINITION 32: **Implementation of ADTO Module Queue.**

```
MODULE Queue
IMPLEMENTATION
  VAR Θ↑,Θ↓,Φ: process list
  VAR P,P': process
  OBJ S: scheduler
  VAL avail,free,size: natural
  VAL φ: data-type list
  VAL d: data-type

  METHOD RESET:
   size ← 2^depth-1
   free ← 2^depth-1
   avail ← 0
   φ ← []
   ∀ P in Θ↓ ∪ Θ↑: ∴P
   Θ↓←[], Θ↑←[]

  METHOD read:
    if avail > 0 then
      decr avail, incr free
      d ← HEAD(φ), φ ← TAIL(φ)
      if Θ↓ ≠ [] then
        P ← HEAD(Θ↓), Θ↓ ← TAIL(Θ↓)
        ∴ P
      $1 ← d
```

```
    else
      P ← SELF
      Θ↑ ← Θ↑ @ [P]
      ⊥P
  METHOD write:
    if free > 0 then
      incr avail, decr free
      φ ← φ @ [↓]
      if Θ↑ ≠ [] then
        P ← HEAD(Θ↑), Θ↑ ← TAIL(Θ↑)
        ∴ P
    else
      P ← SELF
      Θ↓ ← Θ↓ @ [P]
      ⊥P
  METHOD unlock:
   free ← 2^depth-1
   avail ← 0
   φ ← []
   ∀ P in Θ↓ ∪ Θ↑: ∴P
   Θ↓←[], Θ↑←[]
```

DEFINITION 33: **Interface of ADTO Module Queue.**

```
MODULE Queue
INTERFACE
  SIGNAL GD,RE,WE: logic
  SIGNAL RD,WR: data-type
  METHOD read:
    Γ: σ ← σ | GD=1 //
       σ+ | GD=0
    Δ: RE ← ¬GD
       $ARG1 ← RD
  METHOD write:
    Γ: σ ← σ | GD=1 //
       σ+ | GD=0
    Δ: WE ← ¬GD
       $ARG1 ← WR
```

Channel

The channel is both a core and abstract object data type. Queues can be used directly in expressions (read and write). A channel is a special case of a queue. The depth of the buffer is either 1 (buffered) or 0 (unbuffered). The channel provides a handshake for data transfer between processes.

The following object methods are available:

METHODS

read

A process reading a channel is blocked untill one data word is available.

write

A process writing to a queue is blocked untill there is a ready reader (unbuffered version) or the channel is empty (buffered version).

unlock

This method releases all blocked processes (both waiting for read and write completion).

PARAMETERS

depth

Number of cell of the data queue: {0,1}

DEFINITION 34: **Signature of ADTO Module Channel.**

```
MODULE Channel
SIGNATURE
  TYPE channel
  TYPE parameters = {
    depth: {0,1}
  }

  METHOD NEW ≡
    channel name: data-type [with parameters]: object-type

  METHOD unlock: ∅
  METHOD read: ↑data:data-type
  METHOD mode: ↓data:data-type
```

DEFINITION 35: **Implementation of ADTO Module Channel.**

```
MODULE Channel
IMPLEMENTATION
  VAR Θ↑,Θ↓,Φ: process list
  VAR P,P': process
  OBJ S: scheduler
  VAL avail,free,size: natural
  VAL φ: data-type list
  VAL d: data-type

  METHOD RESET:
   size ← 2^depth-1
   free ← 2^depth-1
   avail ← 0
   φ ← []
   ∀ P in Θ↓ ∪ Θ↑: ∴P
   Θ↓←[], Θ↑←[]

  METHOD read:
    if avail > 0 then
      decr avail, incr free
```

```
    d ← HEAD(φ), φ ← TAIL(φ)
    if Θ↓ ≠ [] then
      P ← HEAD(Θ↓), Θ↓ ← TAIL(Θ↓)
      ∴ P
    ↑d
  else
    P ← SELF
    Θ↑ ← Θ↑ @ [P]
    ⊥P
METHOD write:
  if free > 0 then
    incr avail, decr free
    φ ← φ @ [↓]
    if Θ↑ ≠ [] then
      P ← HEAD(Θ↑), Θ↑ ← TAIL(Θ↑)
      ∴ P
  else
    P ← SELF
    Θ↓ ← Θ↓ @ [P]
    ⊥P
METHOD unlock:
 free ← 2^depth-1
 avail ← 0
 φ ← []
 ∀ P in Θ↓ ∪ Θ↑: ∴P
 Θ↓←[], Θ↑←[]
```

DEFINITION 36: **Interface of ADTO Module Channel.**

```
MODULE Channel
INTERFACE
  SIGNAL GD,RE,WE: logic
  SIGNAL RD,WR: data-type
  METHOD read:
    Γ: σ ← σ | GD=1 //
       σ+ | GD=0 ∧ (avail > 0 ∨ ∴P)
    Δ: RE ← ¬GD
       $ARG1 ← RD
  METHOD write:
    Γ: σ ← σ | GD=1 //
       σ+ | GD=0 ∧ (free > 0 ∨ ∴P)
    Δ: WE ← ¬GD
       $ARG1 ← WR
```

External Communication

Link

The Link module implements a bidirectional asynchronous parallel communication interface using dual rail data encoding and a 4-phase handshake

protocol. A link is used to connect different circuit components with different or skewed clock domains.

Arbitrary data widths [1...64] bits are supported. There is an outgoing and incoming link, requiring each 2*datawidth data lines D and one acknowledge signal A: D↑ and A↓ for the outgoing, and D↓ and A↑ for the incoming link.

The following object methods are available:

METHODS

start
Enables the LINK component.

stop
Disables the LINK component.

read
Read one data byte from the incoming link. This method blocks the calling process untill a data word was received. There is no receive queue, therefore if more than one data byte was received the old ones are overriden!

write
Write one data byte to the outgoing link. The calling process is blocked untill the data word was processed and acknowledged by the receiving end.

interface
Binds external signals (data and acknowledge, both for incoming and outgoing link).

PARAMETERS

datawidth
Datawidth of link.

Definitions **37** and **38** specifiy the signature and the implementation of the link module.

DEFINITION 37: **Signature of ADTO Module Link.**

```
MODULE Link:
SIGNATURE
  TYPE link

  METHOD NEW ≡
    object name: link: link
  TYPE parameters = {
    datawidth: natural=[1,64];
```

```
  }

  TYPE data-type: logic[datawidth]
  TYPE data-type2: logic[2*data-width]
  TYPE storage-type: register | variable |
                     reg-array-selector |
                     var-array-selector |
                     reg-struct-selector |
                     var-struct-selector
  TYPE constant: data-type

  METHOD start: ∅
  METHOD stop: ∅

  METHOD read:
    ↓data:(storage-type × data-type) ×
    ↓err:(storage-type × bool)
  METHOD write:
    ↑data:(storage-type × data-type | constant) ×
    ↓err:(stoarge-type × bool)
  METHOD interface:
    ↓data-in:(input signal × data-type2) ×
    ↓data-in-ack:(output signal × logic) ×
    ↓data-out:(output signal × data-type2) ×
    ↓data-out-ack:(input signal × logic)
```

DEFINITION 38: **Implementation of ADTO Module Link.**

```
MODULE Link:
IMPLEMENTATION
  VAR Θ↑,Θ↓,Φ: process list
  VAR P,P': process
  OBJ S: scheduler
  VAL enable: bool
  VAL din_compl,din_empty: logic

  METHOD start:
    enable ← true
  METHOD stop:
    enable ← false
  METHOD read:
   wait for din_empty=1 - await empty set (data-in)
   wait for di_compl=1  - await data (data-in)
   d ← decode(data-in)
   data-in-ack ← 1 ← 0
   $ARG1 ← d
  METHOD write:
   d ← $ARG1
   data-out ← encode(∅)  - send empty set (data-out)
   data-out ← encode(d)  - send data (data-out)
   wait for data-out-ack=0 - await empty set ack. (data-out-ack 1→0),
   wait for data-out-ack=1 - await data acknowledge (data-out-ack 0→1)
  PROCESS:
   din_empty ← ∀ i ∈ [0,data_width-1]: Π ¬data-in.[2*i] ∧
¬data-in.[2*i+1]
```

```
    din_compl ← ∀ i ∈ [0,data_width-1]: Π data-in.[2*i] ⊕
data-in.[2*i+1]
   METHOD interface:
    data-in ⇐ $ARG1
    data-in-ack ⇐ $ARG2
    data-out ⇐ $ARG3
    data-out-ack ⇐ $ARG4
   FUN encode:
     ...
   FUN decode:
     ...
```

DEFINITION 39: **Interface of ADTO Module Link.**

```
MODULE Link
INTERFACE
  SIGNAL GD,RE,WE,START,STOP: logic
  SIGNAL RD,WR: data-type
  SIGNAL RD_ERR,WR_ERR: logic (bool)

  METHOD start:
    Γ: σ ← σ | GD=1 //
       σ+ | GD=0
    Δ: START ← ¬GD
  METHOD stop:
    Γ: σ ← σ | GD=1 //
       σ+ | GD=0
    Δ: STOP ← ¬GD
  METHOD read:
    Γ: σ ← σ | GD=1 //
       σ+ | GD=0
    Δ: RE ← ¬GD
       $ARG1 ← RD
       $ARG2 ← RD_ERR
  METHOD write:
    Γ: σ ← σ | GD=1 //
       σ+ | GD=0
    Δ: WE ← ¬GD
       WR ← $ARG1
       $ARG2 ← WR_ERR
```

EXAMPLE 13: **More complex example of a Link component used by two processes, one reading from the link, and one writing to the link.**

```
1:  open Core;
2:  open Process;
3:  open Link;
4:  open System;
5:
6:  type dev_type:{
```

```
7:    port ln_din: input logic[20];
8:    port ln_din_ack: output logic;
9:    port ln_dout: output logic[20];
10:   port ln_dout_ack: input logic;
11: };
12: component DEV: dev_type;
13: export DEV;
14:
15: object sys: system;
16:   sys.simu_cycles(100);
17:
18: object ln: link with datawidth=10;
19:   ln.interface(DEV.ln_din,DEV.ln_din_ack,DEV.ln_dout,DEV.ln_dout_ack);
20:
21: reg x,y: int[10];
22: reg xa,ya: int[8];
23: export x,y,xa,ya;
24: exception Exit;
25:
26: process p1:
27: begin
28:   reg err:bool;
29:   reg d:logic[10];
30:   try
31:   begin
32:     for i = 1 to 10 do
33:     begin
34:       xa ← 'r';
35:       ln.read (d,err);
36:       xa ← '.';
37:       if err = true then raise Exit;
38:       x ← to_int(d);
39:     end;
40:   end
41:   with
42:   begin
43:     when Exit: ln.stop ();
44:   end;
45: end;
46:
47: process p2:
48: begin
49:   reg err:bool;
50:   reg d:logic[10];
51:   try
52:   begin
53:     for i = 1 to 10 do
54:     begin
55:       d ← to_logic(i);
56:       y ← i;
57:       ya ← 'w';
58:       ln.write(d,err);
59:       ya ← '.';
60:       if err = true then raise Exit;
```

```
61:       end;
62:     end
63:     with
64:     begin
65:       when Exit: ln.stop ();
66:     end;
67:   end;
68:   process main:
69:   begin
70:     ln.init ();
71:     ln.start ();
72:     p1.start ();
73:     p2.start ();
74:   end;
```

Data Processing

[redacted] Random

...

Introduction

...

Modules and Processes

...

Modules

...

Processes

...

Block Strcutures

...

Data Objects and Data Types

...

Registers

...

Variables

...

Signals

...

Expressions and Assignments

...

Interprocess-Communication

...

Mutex

...

ALGORITHM 3: **The static priority mutex access scheduler with embedded mutex implementation.**

```
MUTEX_$O_SCHED: #process ($scheduler="static")
```

```
begin
  if $CLK then
  begin
    if $RES then
    begin
      MUTEX_$O_LOCKed <= '0';
      foreach $p in $P do
      begin
        MUTEX_$O_$p_GD <= '1';
      end;
      foreach $p in $P.lock do
      begin
        MUTEX_$O_$p_LOCKed <= '0';
      end;
    end
    else
    begin
      foreach $p in $P do
      begin
        MUTEX_$O_$p_GD <= '1';
      end;
      sequence
      begin
        foreach $p in $P.init do
        begin
          if MUTEX_$O_$p_INIT = '1' then
          begin
            MUTEX_$O_LOCKed <= '0';
            MUTEX_$O_$p_GD <= '0';
            foreach $l in $P.lock do
            begin
              if MUTEX_$O_$l_LOCKed = '1' then
              begin
                MUTEX_$O_$l_LOCKed <= '0';
                MUTEX_$O_$l_GD <= '0';
              end;
            end;
          end;
        end;
        foreach $p in $P.lock do
        begin
          if MUTEX_$O_$p_LOCK = '1' and MUTEX_$O_LOCKed  = '0' then
          begin
            MUTEX_$O_LOCKed <= '1';
            MUTEX_$O_$p_LOCKed <= '1';
            MUTEX_$O_$p_GD <= '0';
          end;
        end;
        if $model = "owner" then
          foreach $p in $P.unlock do
          begin
            if MUTEX_$O_$p_UNLOCK = '1' then
            begin
              MUTEX_$O_LOCKed <= '0';
              MUTEX_$O_$p_LOCKed <= '0';
              MUTEX_$O_$p_GD <= '0';
            end;
          end;
        if $model = "group" then
```

```
          begin
            foreach $p in $P.unlock do
            begin
              if MUTEX_$O_$p_UNLOCK = '1' then
              begin
                MUTEX_$O_LOCKed <= '0';
                if member($P.lock,$p) = true then
                  MUTEX_$O_$p_LOCKed <= '0';
                MUTEX_$O_$p_GD <= '0';
              end;
            end;
          end;
        end;
      end;
    end;
  end;
```

Semaphore

...

Event

...

Barrier

...

Timer

...

Queue

...

Channel

...

External Communication

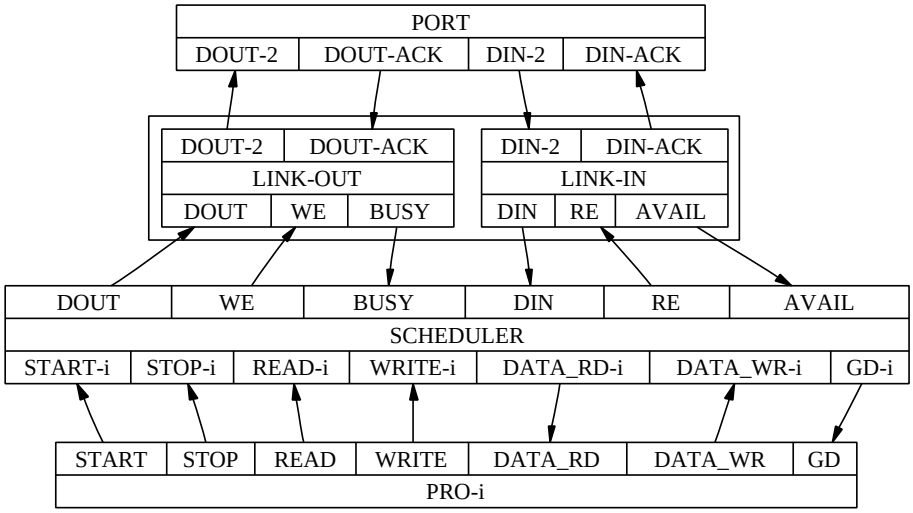
Link

A link belongs to the class of external IO components, providing handshaked data exchange with the outside world, requiring interconnect both with ConPro processes using internal signals $\wp_{\text{int}}$ and external signals $\wp_{\text{ext}}$. Because a link is a sheared resource, a scheduler is invoked.

Types

...

Data Types



GRAPH 2: **Link Block Interconnect**

...

Abstract Object Types

...

Product Types: Array

...

Product Types: Structure

...

Sum Types: Enumeration

...

Exceptions

...

Control Statements

Counting for-Loop

...

Conditional while-Loop

...

Endless always-Loop

...

- **Blocking wait-for-Loop**

...

- **Conditional if-else-Branch**

...

- **Multicase match-Branch**

...

- **Exception Handler**

...

I/O: Hardware Port Interface

- **Components: Interfacing HDL**

...

- **External Module Interface: Embedding HDL**

...

6 External Module Interface EMI

The External Module Interface (EMI) is used to connect and merge VHDL modelled components with the *CONPRO* process framework.

The purpose of the EMI module interface is to embedd and connect external VHDL code directly into a ConPro implementation with direct access from inside ConPro processes using the Abstract Data Type Object (ADTO) method interface and method calls. In contrast to external VHDL components requiring signal objects for interconnection with ConPro, here VHDL processes are linked invisible and transparent to the programmer view 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.

Each EMI module defines a new abstract object type. Objects can be created from this new type.

The EMI language is a modified subset of VHDL, aligned to ConPro programming language concepts. The language semantic is layered on hardware behaviour level using signal objects.

Parameter Section

Name

`#parameter`

Syntax

```
#parameter
begin
  $pname;           - (1)
  $pname <= m;      - (2)
  $pname[n1,n2,n3,...] <= m;  - (3)
  $pname[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).

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.

Example

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

Methods Section

Name

#methods

Syntax

```
#methods
begin
  mname(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 ADTO implementation is supported 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;
```

Interface Section

Name

#interface

Syntax

```
#interface
begin
  signal sname_$0 : dir datatype;     - (1)

  foreach $p in $P do                 - (2)
  begin
    signal sname_$0 : dir datatype;
    ...
  end;

  foreach $p in $P.meth do            - (3)
  begin
    signal sname_$0 : 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 implementation 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 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.

There are two ways for adding process port signals:

(1,2)

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 $0.

(3)

Signals depending on ConPro processes accessing this object and a specific method. The signal name can contain the object name variable `$0`. 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;
```

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 implementation 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;
```

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;
```

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 `$0`. 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 `$0`. 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;
```

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 for-loop, too.

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

constant values

Syntax is slightly modified:

```
...
```

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
```

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, on 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 an environment variable returns the top of the array (the last element stored). But there are builtin functions to access array elements.

```
$name            <=>
  scalar read access of an
  environment variable
size($array)    <=>
  returns number of array elements
  actually stored in the array
width($array)   <=>
  returns number of bits required
  for the maximal value in array
  and weighted binary coding
index_width($array)  <=>
  returns number of bits required
  for the index of an array
  and weighted binary coding
index($array,value)  <=>
  returns the binary coded index
  selector for value element
  contained in array
min($array)     <=>
  returns minimal value of all
  elements in the array
max($array)     <=>
  returns maximal value of all
  elements in the array
```

Printing

During object synthesis informational text lines can be printed to the standard output channel using the `print` function.

```
#print("Achieved baud rate accuracy [bit/s]: ",
```

```
                "[actual = ",$clock / (($clock / (16 * $ARG1))*16),"]
      ",        "[requested = ",$ARG1,"] ",
                "[error = ",(((($clock / (($clock /
                            (16 * $ARG1))*16))*1000)/$ARG1)-1000,
                          " %%]" );
```

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 expression) 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;
```

7 Tool Description Interface TDI

The Tool Description Interface (TDI) is used to emit project specific synthesis and technology tool scripts.

Using the TDI tool interface provides a unique and easy way to emit project tool script files for post hardware synthesis and simulation. Different target script languages are supported, for example bash and make.

The TDI tool file (file suffix .tool) defines

1. environment parameters (immutable) and variables (mutable),
2. target and auxilliary functions,
3. and a target scheduler.

Each TDI tool defines a new tool description. There are different output language targets. Actually only BASH scripts are supported.

Parameter section

Name

`#parameter`

Syntax

```
#parameter
begin
  $pname <= expr;     - (1)
  if expr then statement else statement;  - (2)
  foreach $iname in $array do  - (3)
    statement;

  ...
end;

statement ::= assign ';' | 'begin' assign-list 'end' ';'
assign-list ::= assign ';' assign ';' ...
assign ::= $pname '<=' expr
expr ::= value | function | arith-expr | bool-expr | relat-expr
value ::= string | char | int
string ::= "..."
char ::= '.'
int ::= [0..9]+
function ::= fname '(' arg-list ')'
arg-list ::= expr , expr , ...
```

Description

Environment parameter and variable definition section. Parameters are assigned only one time a (constant) value, variables are assigned different values. Parameters are replaced in expression with their actual value, whereby variables are used in expressions as real variables.

Global variables are always arrays. That means each time a new value is assigned to this variable a new array element is created and appended to the array.

Operations on environment parameters:

1. Assignments and definition of an environment parameter.
2. Conditional assignments.
3. Array iteration. The loop body execution is performed with the iteration variable $iname with actual value from the array element set.

There are some predefined parameters and parameter arrays:

$proj
: This project name

$vhdl
: Parameter array containg all synthesized and imported VHDL files of this project.

$port
: Parameter array containg all VHDL toplevel port signals of this project.

$target
: Parameter array containg all hardware target devices of this project.

$clock_edge
: System clock signal activity level.

$reset_state
: Reset signal activity level.

$simu_cycles
: Number of simulation clock cycles.

$simu_period
: Period of simulation clock cyclce in nanoseconds.

$encoding
: Default encoding format.

Example

```
#parameter
begin
  $TOP <= $proj;
  $OBJDIR <= "obj";
  $SRCDIR <= ".";
  $DESIGN <= $proj;
  $LOG <= $TOP + ".log";
  $DUP <= "tee";
  $PWD <= get_env($PWD,"/");
  $ALLIANCE_TOP <= get_env($ALLIANCE_TOP,"/opt/alliance-5.0");
  $BIN <= $ALLIANCE_TOP + "/bin";
  $CELLS <= $ALLIANCE_TOP + "/cells";
  $MBK_CATA_LIB <= $CELLS + "/sxlib";
  $SCRAM <= $BIN + "/scram";
  $VHD2 <= $BIN + "/vhd2vst";
  $SXLIB_COMP <= $CELLS +"/sxlib/sxlib_components.vhd";
  $EXEMPLAR <= get_env($EXEMPLAR,"/export/home/leonardo");
  $LEONARDO_TOP <= $EXEMPLAR;
  $SPECTRUM <= $LEONARDO_TOP + "/bin/spectrum";
  $LIBRARY <= "sxlib";
  $ENCODING <= get_opt($encoding,"binary");
  $res <= 0;
  $PATH <= get_env($PATH,"/bin");
  $PATH <= $PATH + ":" + "/usr/ccs/bin";
  $SIM_CYCLES <= get_opt($simu_cycles,"100");
  $SIM_PERIOD <= get_opt($simu_period,"100");
  $SIM_RES <= get_opt($simu_res,"5");
  $CLOCK_EDGE <= get_opt($clock_edge,"1");
  if $CLOCK_EDGE = "1" then
    $CLOCK_EDGE_NEG <= "0"
  else
    $CLOCK_EDGE_NEG <= "1";
  $RESET_STATE <= get_opt($reset_state,"1");
  if $RESET_STATE = "1" then
    $RESET_STATE_NEG <= "0"
  else
    $RESET_STATE_NEG <= "1";
end;

#parameter
begin
  - VHDL sources
  foreach $file in $vhdl do
  begin
```

```
    $VHDL.[$I] <= chop_extension($file);
  end;
end;
```

Function section

Name

#function

Syntax

```
label:#fun
begin
  statement;
  statement;
  ...
end;

statement ::= assign | conditional | loop | function | block
block ::= 'begin' statement-list 'end' ';'
statement-list ::= statement ';' statement;...

assign ::= $pname '<=' expr
conditional ::= 'if' expr 'then' statement ['else' statement] ';'
loop ::= for-loop | foreach-loop
foreach-loop = 'foreach' $iname 'in' $aname 'do' statement ';'
for-loop = 'for' a 'to' | 'downto' b 'do' statement ';'

expr ::= value | function | arith-expr | bool-expr | relat-expr
value ::= string | char | int
string ::= "..."
char ::= '.'
int ::= [0..9]+
function ::= fname(arg-list)
arg-list ::= expr , expr , ...
```

Description

This section defines a user function. User defined functions can be called from any other function and used inside the #target section. User defined functions actually expect no arguments!

foreach

This loop is used to iterate the set of all elements of the array $aname. Each loop iteration changes the iteration variable $iname to the actual array element value. This variable can be used in expressions. There is an additional variable $I which holds the actual loop iteration number, starting with value 1.

for

This loop is used with numerical ranges [a,b]. Each loop iteration changes the iteration variable $iname to the actual value of the set of range values. This variable can be used in expressions. After each loop iteration, the iteration variable is incremented by one (to-direction) or decremented by one (downto-direction).

if-then-else

The conditional expression expr is evaluated. If the result is true, the first then-statement is executed, if it is false the else-statement, which is optional.

Warnings and Limitations

Some target scripting languages don't support lokal variables. Lokal variable names therefore are prefixed with the function name. Global variables and parameters, especially shell environment variables, must be assigned a value outside functions on top level, at least using the get_env or get_opt builtin function!

Example

```
do_synth:#fun
begin
  print ("[Creating Leonardo Spectrum command file...]");
  $TCL <= $TOP  + ".tcl";
  $FILES <= "";
  foreach $file in $VHDL do
  begin
    $SRC <= $file + ".vhdl";
    $FILES <= "../" + $SRC + " " + $FILES;
  end;

  create_file ($TCL);
  $PWD <= get_env($PWD,"/");
  write_line ($TCL,"set_working_dir " + $PWD);
  write_line ($TCL,"set hierarchy_flatten TRUE");
  write_line ($TCL,"set output_file "+ $TOP + ".vhd");
  write_line ($TCL,"set novendor_constraint_file FALSE");
  write_line ($TCL,"set bubble_tristates FALSE");
  write_line ($TCL,"set encoding "+ $ENCODING);
```

```
    write_line ($TCL,"load_library "+ $LIBRARY);
    write_line ($TCL,"read -dont_elaborate {"+ $FILES + "}");
    write_line ($TCL,"pd");
    write_line ($TCL,"read -technology " + $LIBRARY + " {" + $FILES
+ "}");
    write_line ($TCL,"pre_optimize -common_logic -unused_logic
-boundary -xor_comparator_optimize");
    write_line ($TCL,"pre_optimize -extract");
    write_line ($TCL,"pd");
    write_line ($TCL,"optimize .work.MOD_"+ $TOP +
    $CMD <= $SPECTRUM + " -f "+$TCL;
    write_line($LOG,"["+$CMD+"]");

    $res <= exec($CMD);
    check();
    append_file ("leospec.log",$LOG);
  end;
```

Builtin Core Functions

Name

TDI core functions

Syntax

```
call-fun ::= fname '(' arg-list ')'
arg-list ::= par1 ',' par2 ',' ...
infix-fun ::= '+' '-' '/' '*' '=' '<>' '<' '>' '>=' '<='
prefix-fun ::= '-'

type file ::= string
type path ::= string
```

Description

There are a large set of builtin functions wich can be used within expressions. There are functions with a variable number of arguments indicated with dots '...'. All following arguments of same type as the last specified before the dots are optional. Files are always identified by their name.

The following function type declarations have some special notations: Functions without returning a result, commonly procedures, return the dummy type unit, polymorph functions are typed with type'a. All function arguments are placed within paranthesis pairs '()'.

Operators

```
fun +: type'a -> type'a -> type'a
```

If `type'a=int` this operation adds the two operands with result type integer.

If `type'a=string` this operation concatenates both string operands with result type string.

IO

```
fun print: (string,...) -> unit
```

Print string arguments concatenated to one line to standard output channel.

```
fun print_line: (string,...) -> unit
```

Print string arguments each in one line to standard output channel.

```
fun write: (file,string,...) -> unit
```

Print string arguments concatenated to one line to a file specified with the first argument.

```
fun write_line: (file,string,...) -> unit
```

Print string arguments each in one line to a file specified with the first argument.

File Management

```
fun create_file: (file) -> unit
```

Create a file whose file-name and path is specified with the first argument.

```
fun open_file: (file) -> unit
```

Open an existing file whose file-name and path is specified with the first argument.

Example

```
do_synth:#fun
begin
  print ("[Creating Leonardo Spectrum command file...]");
  $TCL <= $TOP  + ".tcl";
  $FILES <= "";
  foreach $file in $VHDL do
  begin
```

```
      $SRC <= $file + ".vhdl";
      $FILES <= "../" + $SRC + " " + $FILES;
    end;


    create_file ($TCL);
    $PWD <= get_env($PWD,"/");
    write_line ($TCL,"set_working_dir " + $PWD);
    write_line ($TCL,"set hierarchy_flatten TRUE");
    write_line ($TCL,"set output_file "+ $TOP + ".vhd");
    write_line ($TCL,"set bubble_tristates FALSE");
    write_line ($TCL,"set encoding "+ $ENCODING);
    write_line ($TCL,"load_library "+ $LIBRARY);
    write_line ($TCL,"read -dont_elaborate {"+ $FILES + "}");
    write_line ($TCL,"pd");
    write_line ($TCL,"read -technology " + $LIBRARY + " {" + $FILES
+ "}");
    write_line ($TCL,"pre_optimize -common_logic -unused_logic
-boundary -xor_comparator_optimize");
    write_line ($TCL,"pre_optimize -extract");
    write_line ($TCL,"pd");
    write_line ($TCL,"optimize .work.MOD_"+ $TOP +
                     ".main -target sxlib -macro -area -effort
quick -hierarchy flatten");
    write_line ($TCL,"optimize_timing .work.MOD_"+$TOP+".main");
    write_line ($TCL,"report_area -cell_usage -hierarchy
-all_leafs");
    write_line ($TCL,"report_delay  -num_paths 1 -critical_paths
-clock_frequency");
    write_line ($TCL,"auto_write -format VHD "+$TOP+".vhd");
    print_line ("[Forking Leonardo Spectrum...]");
    $CMD <= $SPECTRUM + " -f "+$TCL;
    write_line($LOG,"["+$CMD+"]");

    $res <= exec($  write_line ($TCL,"set novendor_constraint_file
FALSE");
    check();
    append_file ("leospec.log",$LOG);
  end;
```

...

9 Synthesis

Basic Scheduling Model

...

Synthesis RulesSynthesis Rules

...

Expression Modells and Allocation

...

μCode Transformation

...

Reference Stack Scheduler

...

Basic Block Scheduler

...

Expression Scheduler

...

...

[RU87] Steven M. Rubini
Computer Aids For VLSI Design
Addision Wesley 1987