Object and Data Types

ConPro type system

1. TABLE OF CONTENT

2. INTRODUCTION

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

3. OBJECT TYPES

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

Table 1. Object Types

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

4. DATA TYPES

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

Table 2. Data Types

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

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

5. DATA STORAGE OBJECTS

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

rays and structure), and sum types (enumeration type), too, see section **8** to **10**.

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

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

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

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

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

Definition	Description
`reg name: DT[N];`	Defines a new storage object of type register with a specified data type *dt* and optional data width *N* bits.
`queue name: DT[N];` `queue name: DT[N] with depth=8;`	Defines a new interprocess communication data object of type queue with a specified data type *dt* and optional data width *N* bits. Optional parameters are passed using the `with` statement.

`block RAM;`
`var name: DT[N] in RAM;`
`var name: DT[N];`
	Defines a new storage object of type variable with a specified data type *dt* and optional data width *N* bits. The object is stored in a shared RAM block (optional).

`x ← y;`	Appearence of stoarge objects on left and right hand side of an assignment.
`x[J] ← y[I];`	Bit index selector used in expression with vector objects.

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

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

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

```
1  process xyz:
2  begin
3    reg x,y,z: int[5];
4    var v: int[10];
5    for i = 1 to 10 do
6    begin
7      x ← x + 1;
```

```
8       y ← x * z - 1;
9       if z < 10 and x > 100 then v ← v + z;
10    end;
11  end;
```

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

```
1  queue q1,q2: char;
2  process flip:
3  begin
4    reg c: char;
5    c ← 'a';
6    for i = 1 to 10 do
7    begin
8      q2 ← c + 1;
9      c ← q1;
10   end;
11 end;
12 process flop:
13 begin
14   reg c: char;
15   for i = 1 to 10 do
16   begin
17     c ← q2;
18     q1 ← c + 1;
19   end;
20 end;
```

6. SIGNALS

Signals are interconnection elements without a storage model. They provide an interface to external hardware blocks. Signals are used in component structures, too, shown in example **4**. Lines 1 to 7 define a signal port interface of a component, and line 8 instantiates a component of this type.

Signals can be used directly in expressions like any other storage object. Signals can be read in expressions, and a value can be assigned in assignments, shown in example **4** (for example line 11 using a static map statement, and line 33). Reading a signal returns the actual value of a signal, for example line 23, but writing to a signal assigns a new value only for the time the assignment is active, otherwise a default value is assigned to the signal, for example in line 34. Therefore, there may be only one assignment for a signal.

Signals are non-shared objects, and have no access scheduler. Only one process may assign values to a signal (usually using the `wait for` statement), but many processes may read a signal concurrently. Additionally, signals can be mapped to register outputs using the `map` statement.

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

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

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

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

7. ABSTRACT OBJECT TYPES

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

Definition 4. Opening of a module.

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

ADT objects can be accessed by their appropiate method set $\theta=\{\theta_1,\theta_2,...\}$. A method is applied using the selector `'.'` operator followed by a list of arguments passed to method parameters, with arguments separated by a comma list encapsulated between paranthesis, shown in definition **5**.

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

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

Methods which do not expect arguments are applied with an empty argument list

(). Table **4** summarizes the statements required for using abstract object types.

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

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

8. ARRAY TYPES

Arrays are product types and can be applied to data storage objects including queues and channels. Additionally, arrays can be applied to abstract objects, too, providing indexed object selection. Array elements can be accessed with static selectors (constant values or expressions foleded to a constant value), and with dynamic selectors (expressions referencing storage objects).

Arrays of variables are implented in memory blocks. The element access is performed by address calculation and access of the memory block. Arrays of any other object type are always implemented with single objects. If elements of such an array only accessed with static selectors, the array access is replaced by the appropiate object. If at least there is one element access with a dynamic selector expression, all objects of this array are accessed my multi- and demultiplexer blocks (simulated memory block implementation).

Arrays of processes can be defined, too. A process identifier symbol #=[0,N-1] can be used in expressions of each process created by the array definition.

Multi-dimensional arrays are supported, too. But in this case each dimension must be of power 2 (or will be aligned during the synthesis). Multi-dimensional arrays are mapped to one-dimensional arrays to simplify hardware synthesis. the one-dimensional idnex is calculated by the following equation:

Equation 1.

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

Formal syntax definitions **6** and **7** show definition and access of arrays, summa-

rized in table **5**. An extended example **5** demonstrates object and storage arrays.

Definition 6. Formal syntax specification for array definition

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

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

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

```
dim ::= ( number // ',' ) .
```

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

Statement	Description
`array A: OT[N] of DT;` `array A: OT[N,M,O] of DT;`	Defines an storage array of size N with object type OT and data type DT. Second line defines a multi-dimensional array (matrix).
`array A: OT[N] of DT` `with param=value;`	Defines an storage array of size N with object type OT and data type DT, with additional parameter settings.
`array A: object obj[N]` `with param=value;`	Defines an abstract object array of size N with object type obj, with additional parameter settings. Note: object parameters must be preceeded by the module name and the dot selector!

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

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

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

```
1  open Core;
2  open Process;
3  open Semaphore;
4
5  array fork: object semaphore[5] with
6              Semaphore.depth=8 and Semaphore.scheduler="fifo";
7  array eating,thinking: reg[5] of logic;
8
9  process init:
10 begin
11   for i = 0 to 4 do
12   begin
13     fork.[i].init (1);
14   end; -- with unroll;
```

```
15   ev.init ();
16 end;
17
18 function eat(n):
19 begin
20   begin
21     eating.[n] <- 1;
22     thinking.[n] <- 0;
23   end with bind;
24   wait for 5;
25   begin
26     eating.[n] <- 0;
27     thinking.[n] <- 1;
28   end with bind;
29 end with inline;
30
31 array philosopher: process[5] of
32 begin
33   if # < 4 then
34   begin
35    always do
36    begin
37      -- get left fork then right
38      fork.[#].down ();
39      fork.[#+1].down ();
40      eat (#);
41      fork.[#].up ();
42      fork.[#+1].up ();
43    end;
44   end
45   else
46   begin
47    always do
48    begin
49      -- get right fork then left
50      fork.[4].down ();
51      fork.[0].down ();
52      eat (#);
53      fork.[4].up ();
54      fork.[0].up ();
55    end;
56   end;
57 end;
58
59 process main:
60 begin
61   init.call ();
62   for i = 0 to 4 do
63   begin
64     philosopher.[i].start ();
```

```
65    end;
66 end;
```

9. STRUCTURE TYPES

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

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

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

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

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

Tip

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

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

Some care must be taken for the direction: if the component is in lower 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 top-level port interface of a ConPro module, it must be seen from the internal view. Formal syntax definitions for structure definitions and structure element access can be found in definitions **8** to **9**. An extended demonstration of the capabilities of structure types can be found in example **6**. Table **6** summarizes the definition and usage of structure types.

Definition 8. Formal syntax specification for structure type definition

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

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

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

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

```
1  -- Multi-type structure type definition
2  type registers : {
3    ax : logic[32];
4    bx : logic[32];
5    sp : logic[16];
6   };
7  type image : {
8    row: logic[32%4];
9    col: logic[32%4];
10  };
11 -- Component structure type defintion
12 type uart : {
13   port rx : input logic[2];
14   port tx : output logic[2];
15   port re : output logic;
16   port we : input logic;
17  };
18 -- Bit-type structure type defintion
```

```
19 type command : {
20   ack:  0;
21   cmd: 1 to 2;
22   data: 3 to 7;
23  };
24
25 block ram1;
26 reg regs : registers;
27 var vegs : registers in ram1;
28 var vim : image in ram1;
29 var after : logic[16] in ram1;
30 component dev1: uart;
31 reg cmd: command;
32 process p1:
33 begin
34   reg x: logic[2];
35   type cpu_regs : {
36    ax : logic[8];
37    bx : logic[8];
38    sp : logic[8];
39   };
40   var cpu : cpu_regs in ram1;
41   reg row: logic[32];
42   ...
43   regs.ax ⟵ row;
44   regs.ax ⟵ regs.ax + 1;
45   vegs.ax ⟵ vegs.ax + intern;
46   ...
47   wait for dev1.re = 1;
48   x ⟵ dev1.rx;
49   dev1.tx ⟵ x, dev1.we ⟵ 1;
50   cmd ⟵ 0;
51   cmd.ack ⟵ 1;
52   cmd.cmd ⟵ x;
```

```
53 end;
```

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

Statement	Description

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

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

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

reg *R*: ST; var *R*: ST; reg *R*; *BT*;	Defines scalar storage objects of structure type ST and bit type BT.
array AS: *OT*[*N*] of *ST* with *PARAMS*;	Defines an array of storage objects with object type OT and structure type ST.
component *C*: CT; export *C*;	Defines a new component structure and exports the component structure.
ST.e1 <- *ST.e2*;	Access of structure elements in assignments and expressions.

10. ENUMERATION TYPES

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

Definition 10. Formal syntax specification for enumeration type definition

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

object-definition ::= *object-type* (*identifier* // ',') ':' *enum-type* .

Example 7. Enumeration types

```
1  type states : {
2    S_START;
3    S_1;
4    S_2;
5    S_END;
6  };
7  reg state,next_state: states;
8  process fsm:
9  begin
10   while state <> S_END do
11   begin
12     match state with
13     begin
14       when S_START: state ← S_1;
15       when S_1: state ← S_2;
16       when S_2: state ← S_END;
17     end;
18   end;
19 end;
```