Insense Tutorial
Alan Dearle

1 Motivation

The aims of designing inSense for the DIAS-MC project were to produce a language which

1. is specifically designed for WSN applications
2. is easy to learn and use for non-Computer Scientists
3. allows worst-case space and time usage of programs to be determined statically

2 Design Guidelines

1. Easy to use language with familiar syntax
2. Can be used by unskilled programmers
3. Programmers don’t have to do any of the hard stuff – especially memory management and synchronisation
4. Basic building blocks are components
5. These have zero syntactic nesting (and therefore no implicit dependencies)
6. Fractal composition – one component can instantiate other components (which cannot be seen from outside them)
7. No sharing of any state
8. Need dynamicity – want to be able to replace components, perform rewiring etc.
9. Separation of concerns – don’t mix up policies
10. Dependency injection – supply components with what they need when they are constructed
11. Want to be able to reason about space and time

3 Overview

The basic computational unit in **Insense** is the **component**. Instances of components may be created using constructors that are defined in a similar manner to Java, C# or C++. Components are strongly encapsulated and interact with each other via typed directional channels. Conceptually, each component instance contains a single thread of control. The actions of this thread are called the component’s **behaviour** which is specified in a syntactic entity of the same name. The actions in behaviour repeat indefinitely until the component is stopped either by itself or another component.

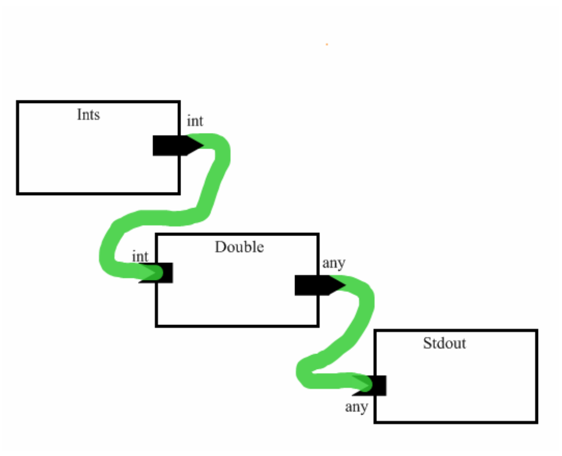

Figure 1

Figure 1 shows three component instances joined together with two channels. Three component instances exist in the figure each of a different type. In Insense, the type of a component is determined by the types of the channels it presents. In the example the component labelled *Ints* presents a single outgoing channel of type integer. The middle component labelled *Double* presents two channels an in channel of type integer and an out channel of type any. The component labelled *StdOut* presents a single input channel of type *any*.
In Insense the types of the interfaces presented by these components may be written as the following type declarations

```
type t1 is interface( out integer output )
type t2 is interface( in integer input; out any output )
type t3 is interface( in any input )
```

Code snippet 1

Note that these type declarations merely describe the types of components, they do not define components or their behaviour.
In order to define a component a component declaration is required. A component definition is needed. A component definition has four parts: a specification of the interface it presents definitions of component variables, at least one constructor and exactly one behaviour.

```
component Ints presents t1 {
  last = 0;

  constructor() {
  }

  behaviour {
          send last on output
          last := last + 1
  }
}
```

Code snippet 2

Code snippet 2 shows the definition of the component labelled *Ints* in Figure 1. A component definition is introduced by the keyword **component** which is followed by the component's name. In practice this name is only used to identify which component is being created when constructors are called. The name of the component is always followed by the keyword **presents** which must be followed by a (comma-separated) list of the interfaces it presents. In Code snippet 2 the component presents a single interface the type *t1* shown in Code snippet 1. The names of all channels in the presented interfaces are automatically implicitly declared. Thus in this example, the name *output* is implicitly brought into scope and is of type "*out integer*".
The component contains the definition of a component variable called *last*. Note that in Insense the types of declarations are inferred by the compiler and the "=" symbol is used to initialise the variable with a value. Component variables are local to the component in which they are declared and may not be accessed outwith their declaring component. The scope of component variables are from the point they are declared until the end of the component.
Component instances are created by calling one of the component's constructors. Like constructors in Java, they may perform arbitrary computation but are normally used to perform dependency injection by initialising component variables. You can have as many constructors as are required but there must be at least one,
The keyword **behaviour** introduces the behaviour of the component which repeats until the component is stopped either by itself or another component using the keyword **stop**. In Code snippet 2 the behaviour writes the current value of variable *last* on the channel *output* before incrementing it (using the update operator written ":="). Note that the channel *output* is declared implicitly as a result of the **presents** statement in the component definition.
Instances of components are created by calling a constructor, an instance of the *Ints* component shown in Code snippet 2 may be created as shown in Code snippet 3.

```
ints_instance = new Ints()
```

Code snippet 3

Components may be wired together by connecting the output of one component to the input of another. To illustrate we introduce a new components called *double* in Code snippet 4. *Double* reads integers from an input using the "**receive from**" construct. It wraps the values received from its input in an any and sends it on a channel called output.

```
component Double presents t2 {

  constructor() {
  }

  behaviour {
          next = receive from input
          send any( next ) on output
  }
}
```

Code snippet 4

To instantiate instances of *Ints* and *Double* and wire them up we write the following code shown in Code snippet 5.

```
ints_instance= new Ints()
double_instance = new Double()
connect ints_instance to double_instance.input
```

Code snippet 5

If we had an instance of a standard out component called *stdout* and with interface *t3*, we could write,

```
connect double_instance.output to stdout.input
```

Code snippet 6

This would complete a running system containing three processes shown in Figure 1.

4 Universe of Discourse and Type System

The InSense base types are: integer, real, bool, string, byte, any and enum. These are defined in sections 4.1 to 4.7 below. InSense also supports five constructed types: Arrays, Channels, Interfaces, Functions, and Structs.

4.1 Integer

The type **integer** defines the class of 16 bit 2s complement integers. These range from *minint* to *maxint* (TODO define literals) which have the values -32768 and 32767 respectively.

4.2 Real

The type **real** provides to 32-bit IEEE single precision floating point numbers.

4.3 Bool

The type **bool** is the class of Booleans – two Boolean literals are provided – *true* and *false*.

4.4 Byte

The type **byte** defines unsigned 8 bit values ranging from 0 to 255. In Insense byte literals are written with a proceeding ‘#’ e.g. #134.

4.5 String

The type string is the class of concatenations of UTF-8 characters.

4.6 Any

The type **any** is a base type containing values of any other type. Any may be thought of as a wrapper that permits an arbitrary type to be wrapped in an any and later unwrapped using projection. There are no literals of type any. Anys are created using the keyword **any** as follows:

```
anany =  any(3)
anotherany = any( true )
```

Code snippet 7

In the code fragment above both *anany* and *anotherany* are of type **any**.

4.7 Enum

Enums provide enumerated types (as recently discovered by Java).

4.8 Naming types

In Insense types are commonly named. For example an interface might be named as follows:

```
type myinterface is interface( in integer input; out any output )
```

Code snippet 8

4.9 Arrays

Arrays are fixed length data structures containing values of a single type. Arrays are indexed from zero. Array types are specified using square brackets as in Java. Thus the type of an array of integers is written integer[]. Two different syntactic forms for creating arrays exist as shown below.

```
intarray =  new integer[5] of 5
anotherarray = new { 2,3,4,5,6 }
```

Code snippet 9

Both arrays shown above have a lower bound of zero and an upper bound of four. Both are or type integer[]. The array *intarray* has all its elements intialised to the value 5 whereas the elements of *anotherarray* are initialised with the values 2,3,4,5,6. Two predefined functions are provided for arrays – size() and upb() which return the number of elements in an array and its upper bound respectively. (TODO define these).
Multi-dimension arrays may be formed as shown in Code snippet 10.

```
threeD =  new integer[2][3][4] of 5
```

Code snippet 10

This creates an array of array of arrays with all the elements initialised to 5 as shown in Figure 2.

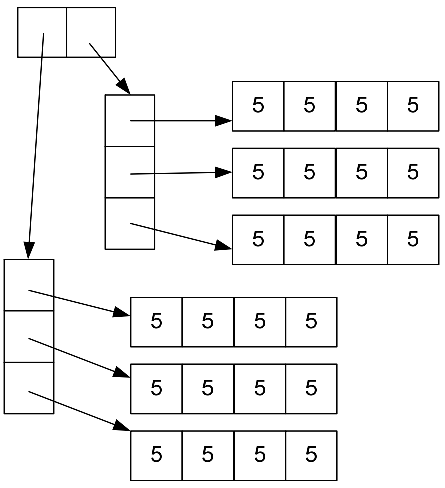

Figure 2

4.10 Channels

Channels are the communication mechanism in Insense. They are typed, directional and synchronous. That is, each channel has a buffer size of one. The type of an input channel of integers is written as follows:

```
in integer
```

Code snippet 11

where as the type of an output channel of anys is written:

```
out any
```

Code snippet 12

4.11 Interfaces

Interfaces are used to define the way in which component instances can interact. An interface is a collection of named channels. For example the following interface defines

```
interface( in integer input1, interface2; out any output )
```

Code snippet 13

4.12 Functions

In InSense functions are abstractions over expressions. The Syntax for function declarations is shown in Figure 3.

```
<function_decl> ::= function < identifier> <lrb> [<named_param_list>] <rrb>
                    <colon> <type>
                    <lcb> <expression> <rcb>
```

Figure 3

Functions may be declared in two places in an inSense *program*[1]: at the top level as a global declaration in the syntactic construct <global_decl> or as a component local declaration. Functions declared globally are global to all components in a *compilation unit*[2] whereas functions defined with a component may only be used in the component in which they are declared. An example function declaration is shown in Code snippet 14.

```
function domath( integer i, j, k; boolean unused ) : integer {
  i + j + k
}
```

Code snippet 14

Functions are called in the normal manner e.g. to execute the above function the programmer may write

```
myint = domath( 2,3,5,true )
```

Code snippet 15

4.13 Structs

Simple, labelled cross product data structures may be constructed using **struct**s in Insense. Unlike structs in other languages (notably C) you cannot create pointers to structs. Therefore structs may be formed from any type other than other structs. An example of a struct type definition is shown below.

```
type person is struct( string name; integer age )
```

Code snippet 16

Instances of structs are created using the keyword new followed by the name used to declare a struct type. Therefore, if the type *person* had been defined as shown above, an instance of that type may be created as follows.

```
al = new person( “al”, 47 )
```

Code snippet 17

4.14 Type equivalence

TODO TYPE EQUIVALENCE AND TYPE RULES

5 Components

Components are the unit of system construction in Insense. A component is a closed entity which presents an interface containing channels. Components may contain some local variable declarations, at least one constructor and a **behaviour**. Like Java classes, a component may have multiple constructors which each have a different type signature. An example component is shown below:

[1] Programs are not well defined at the moment.
[2] Compilation Units are not well defined either.

```
type inttobool is interface( in integer input, out boolean output )

component checker presents inttobool {
  threshold = 10

  constructor() { }

  constructor( integer sz ) {
          threshold := sz
  }

  behaviour {
          num = receive from input
          send num < threshold on output
  }
}
```

Code snippet 18

Note that the component presents the interface *inttobool* containing an input and an output channel. The channel names declared in an interface type are automatically declared in any component which presents that interface. Thus in the component *checker* shown above, the channel names *input* and *output* are brought into scope at the start of the component declaration. The component has a single component global declaration, *threshold* which is of type integer. Note that the type is inferred from the compulsory initialising expression.
Two constructors are provided. All constructors must be called constructor and if more than one constructor is provided they must all have distinct types. In this example, one constructor does nothing, the other sets the variable threshold to be *size*.
All components must contain a *behaviour*, which specifies what the component does. All behaviours loop forever until they are stopped using the ***stop*** construct. In this example, the behaviour reads integer values from the input channel and writes a boolean on the output channel indicating whether the value read was less than or greater than or equal to the *threshold*. Note that behaviours are blocking with respect to their channels. The behaviour will block until a value is available o be read from the input channel and will block on the output channel until it is free.

6 Clauses

There are 12 syntactic clause constructs: **if**, **for**, **switch**, **declaration**, **assignment**, **connect**, **disconnect**, **send**, **receive**, **select**, **try.. except** and **project**. These constructs are described in this section.

6.1 If clauses

The syntax of for clauses is the same as Java and C:

```
if <lrb> <clause> <rrb> <clause> [ else <clause> ]
```

Figure 4

The type of the first clause must be boolean, the two choice arms must be void.

6.2 Switch

TODO

6.3 For clauses

```
for <lrb> <identifier>
<equals> <expression> <dotdot> <expression>
[<colon>[+|-]<integer_literal> ]<rrb> <clause>
```

Figure 5

For clauses are used for iteration. The for loop declares a control variable whose value ranges from the value indicated by the first expression to that indicated by the second expression. By default the increment is one; however, a different increment or decrement may be specified using the optional [<colon> <integer_literal>] syntax.

```
evens = new boolean[10] of false
for( i = 0..9 : 2 )
  evens[i] = true
```

Code snippet 19

Thus, the example shown in Code snippet 19 will initialise the elements corresponding to the even indices to true and those corresponding to the odd indices to false.

6.4 Declarations

In InSense there are two types of declarations: local declarations and global declarations. Global declarations may only be made at the top level – that is not inside any other syntactic construct. Local declarations may be made inside any sequence. Global declarations are defined in the BNF as shown in Figure 6,

```
<global_decl>   ::= <component_decl> | <function_decl> | <type_decl>
```

Figure 6

and local declarations as shown in Figure 7.

```
<decl>   ::=      <value_decl> | <function_decl> | <type_decl>
```

Figure 7

Component declarations are described in the Section entitled *Components*; function declarations are described in the Section entitled *Functions* and type declarations are described in the Section entitled *Enum*. That leaves value declarations which are described here. The BNF for value declarations is shown in Figure 8.

```
<value_decl>   ::=      <identifier> [ <equals> <value> ]
```

Figure 8

As shown in Figure 8, value declarations are very simple, with names being introduced with the equals (=) symbol with a value on the right hand side of the equals as shown in Code snippet 20. Note that the type of the variable is inferred from the initialising value. Note also that there is no way of introducing an un-initialised identifier (this being 2007 and all).

```
myinteger = 7
abool = false
```

Code snippet 20

6.5 Assignment

Assignment to variables is made in the traditional manner using the := operator. Thus the value *myinteger* introduced in Code snippet 20 may be updated as shown in Code snippet 21. Note that equals makes declarations whereas := performs assignment.

```
myinteger := 7
```

Code snippet 21

6.6 Connect

Connect is used to connect an input and output channel together. Typically this is performed to permit communication to occur between two components. The syntax of the connect clause is shown in Figure 9.

```
<connect_clause>        ::=     connect <expression> to <expression>
```

Figure 9

Both expressions must evaluate to a channel value with the type of each being symmetric, that is, if one channel is of type “in integer” the other must be of type “out integer”. An example of connect being used is shown in Code snippet 6.

6.7 Disconnect

Disconnect is the opposite of connect; that is, it disconnects two previously connected channels. Either end of a pair of connected channels may be specified. Al is worried about this…

```
<disconnect_clause>     ::=     disconnect <expression>
```

Figure 10

6.8 Send

The **send** construct is used to send a value on a channel, the value is defined by the expression and the channel is specified using the name as shown in Figure 11. Send is a blocking operation. If the channel already contains a value (placed on it either by the caller of **send** or by another executing component) the execution will block until the channel is empty. If multiple component instances are attempting to write on a single channel the order in which they write on the channel (if at all) is non-deterministic.

<send_clause>	::=	**send** [<expression>] **on** <name>

Figure 11

An example of the **send** construct being used is shown in Code snippet 4.

6.9 Receive

Just as **send** is used to send a value on a channel, receive is used to obtain a value from a channel. Like send, receive is a blocking operation. If no value is on the channel, the operation will block until a value is available. An example of the receive construct being used is shown in Code snippet 4.

<receive_clause>	::=	**receive from** <name>

Figure 12

6.10 Select

Sometimes you do not want a component to block on a particular channel and instead want to read a value from any channel that has values available. This functionality is provided by the **select** construct shown in Figure 13.

```
<select_clause> ::=     select {
                                <select_list>
                                 [default : <clause>]
                        }

<select_list>   ::=     receive <identifier> from <name>
                        [when <clause> ] : <clause>
                        [ <select_list> ]
```

Figure 13

The **select** construct is the most complex and most powerful clause in the language. It provides non-deterministic, guarded channel selection over multiple channels. The semantics of the construct is perhaps best explained with the example shown in Code snippet 22. The behaviour of component *selectExample* contains a **select** clause. The clause non-deterministically reads from one of the three channels *chan1*, *chan2* and *chan3* presented in the interface *threeChans* presented by *selectExample*.

A select arm is eligible for execution only if input is available on the channel specified in the arm. If input is not available on any channel the (optional) default is executed. If no default is specified, the construct blocks until input is available on at least one of the arms. If input is available on an arm, the (optional) *when clause* associated with the arm is evaluated. If the *when clause* evaluates to true or if no *when clause* is specified, the arm is eligible for evaluation. Finally, if no arms are available for evaluation, the default is called or the execution blocks awaiting input on some arm, otherwise one of the clauses on the right hand side of one of the eligible arms are evaluated. If the construct blocks the re-evaluation of eligibility, the algorithm to determine which arm is selected restarts from the beginning following the description contained in this paragraph.

```
type threeChans is interface( in integer chan1, chan2; in bool chan3)
component selectExample presents foo {
  p = 8; q = false
  constructor() {
  }

  behaviour {
          select {
                  receive x from chan1 when q : { p := x }
                  receive y from chan2: p := y
                  receive z from chan3 when p < 7 : q := z
                  default: p := 1
          }
  }
}
```

Code snippet 22

Consider the example shown in Code snippet 22. Suppose that all three channels contain values to be read. Therefore all arms are eligible. Next the guards are evaluated, since *p* = 8, and *q* is false, the first and last arms are not eligible. The assignment of y to p will be executed in the second arm. The value of *y* assigned to *p* is the value waiting on *chan2*. An observant reader will note that this example is in fact poor since the default clause will never be executed. Hopefully the above description will be enough to see why – is it?

6.11 Try.. except

InSense contains a simple exception mechanism which is supported by the **try**..**except** construct shown in Figure 14. Any exceptions thrown in the execution of the first clause in the **try**..**except** construct causes flow of control to be passed to the second clause. Exceptions may be caused in the normal manner with out of bounds indexing operations, divide by zero etc.

```
<try_except>     ::= try <clause> except <clause>
```

Figure 14

6.12 Project

The project clause is used for projecting out of an **any**. The type **any** is described in Section 4.6 Any. The project clause is a type matching operation which permits the type of value wrapped in an any to be discovered and extracted.

```
<project_clause> ::=  project <clause> as <identifier> {
                            <project_list>
                            [default : <clause>]
                      }

<project_list>    ::= <type> : <clause> [ <project_list> ]
```

Figure 15

An example of the project clause being used is shown in Code snippet 23. In the example, a declaration is made of a variable called *anany* whose type is any. Note that the variable may hold a value of any type (hence the name **any**). The project statement projects out of an any by matching against the type of the value enclosed within it. In the example, two types (integer and bool) are tested for, if one of these types matches that of the enclosed value the code following the colon is executed. In the example, the boolean will match and the code "*b* := *avalue*" will be executed. The value wrapped in the any is extracted in the "**project** *anany* **as** *avalue*" and that value bound to the name *avalue*. Note that the type of avalue is different in each of the project arms i.e. it is integer in the first arm, boolean in the second and the name is not in scope in the default arm. In the example, since the boolean type is matched, the value associated with *avalue* will be true and true will be assigned to the *b*.

```
  behaviour {
        a = 7
        b = true
        anany = any( a )
        anany:= any( b )
        project anany as avalue {
              integer      : { a := avalue }
              bool         : { b := avalue }
              default      : { a := 32 }
        }
}
```

Code snippet 23

7 Expressions

7.1 Boolean Expressions

There are two boolean literals, **true** and **false** and three operators: not (written !) and two binary operators, **and** and **or**. The precedence ordering of these operators in descending order is: not, and, or. This is reflected in the BNF shown in Figure 16.

```
<expression>    ::=     <exp1> {<or> <exp1>}
<exp1>  ::=             <exp2> {<and> <exp2>}
<exp2>  ::=             [not_op] <exp3> [<rel_op> <exp3>]
```

Figure 16

The evaluation of non-strict, that is as soon as the result is found, no more computation is performed on the expression.

7.2 Comparison operators

The usual comparison operators found in other languages are provided in inSense. These are: equals, not equals, less than, greater than, greater than or equal to and less than or equal to. These are written in the C/Java style namely, ==, !=, <, >, <= and >=.

7.3 Arithmetic expressions

Arithmetic may be performed on the types byte, integer, real. No automatic coercion is provided by the language (check this). The syntax of arithmetic expressions is:

```
<exp3>   ::= <exp4> { <add_op> <exp4> }
<exp4>   ::= <exp5> { <mult_op> <exp4> }
<exp5>   ::= [add_op] <exp6>
<mult_op> ::= <int_mult_op> | <real_mult_op> | <string_mult_op>
<int_mult_op>   ::= * | / | %
<real_mult_op> ::= * | /
<string_mult_op>        ::= ++
<add_op>          ::= + | -
```

Figure 17

Where the operators have their normal C/Java meaning. Precedence todo

7.4 Identifiers

InSense identifiers start with a letter and are composed of letters and digits as defined in the BNF shown in Figure 18. Thus the following are legal identifers:

a, *anidentifier*, *aComponent*, *Component1243*;

and the following are not:

123a, *a.b*, *an_identifier*.

```
<identifier>      ::= <letter> [<id_follow>]
<id_follow>       ::= <letter> [<id_follow>] | <digit> [<id_follow>]
```

Figure 18

7.5 Literals

There are 5 types of literals in InSense: integer, byte, real, boolean and string literals as shown in Figure 19.

```
<literal>      ::=  <int_literal> | <real_literal> | <bool_literal> |
                    <string_literal> | <byte_literal>
<int_literal>      ::=      [add_op] digit {digit}
<byte_literal>     ::=      # [digit {digit}
<real_literal ::=  int_literal.{digit}[e <int_literal>]
<bool_literal>     ::=      true | false
<string_literal>   ::=      " {<char>} "
```

Figure 19

The literals are described in more detail in the corresponding sub sections of the section entitled *Universe of Discourse and Type System*.

7.6 Function application

<expression> <lrb> [<clause_list>] <rrb> | // application

Al is currently here

7.7 Sequences

<clause_block>
<clause_block>   ::= <lcb> <clause_sequence> <rcb>

7.8 Dereferencing operators

<dereference>   ::= <component_struct_dereference> | <array _dereference>
<component_struct_dereference> ::= <expression> { <dot> <identifier> }
<array_struct_dereference> ::= <expression> <lsb> [<clause_list>] <rsb>

8 Index

A

assignment, 11

C

- Clauses
  - assignment, 11
  - connect, 11
  - declarations, 11
  - disconnect, 12
  - for, 10
  - if, 10
  - project, 14
  - receive, 12
  - select, 12
  - send, 12
  - try, 14
- Clauses, 10
- Components, 9
- connect, 11

D

declarations, 11
Design Guidelines, 1
disconnect, 12

F

for, 10
Functions, 8

I

if, 10

O

Overview, 1

P

project, 14

R

receive, 12

S

select, 12
send, 12

T

- try, 14
- Types, 5
  - Any, 6
  - Array, 7
  - Bool, 6
  - Byte, 6
  - Channel, 7
  - Components, 9
  - Enum, 6
  - Functions, 8
  - Integer, 6
  - Interface, 8
  - Naming, 6
  - Real, 6
  - String, 6
  - Structs, 8

9 Appendix A: Context Free Grammar

9.1 Programs

<prog> ::= {<global_decl> <semi_colon>} <sequence>

9.2 Global Declarations

// deleted global_decls
<global_decl> ::= <component_decl> | <function_decl> | <type_decl>
<component_decl> ::= **component** <identifier> **presents** <identifier_list> <component_body>
<component_body> ::= <lcb> <local_decls> <constructors> <behaviour> <rcb>
<local_decls> ::= <decl> <semi_colon> [<local_decls>]
<constructors> ::= <constructor> <semi_colon> [<constructors>]
<constructor> ::= **constructor** <lrb> [<named_param_list>] <rrb> <block>
<named_param_list> ::= <type> <identifier_list> [<semi_colon> <named_param_list>]
<behaviour> ::= **behaviour** <block>

```
<function_decl>  ::=  function  <  identifier>  <lrb>  [<named_param_list>]  <rrb>
<colon> <type>
                         <lcb> <expression> <rcb>
<identifier_list>        ::=  <identifier> [<comma> <identifier_list>]
```

9.3 Type Declarations

```
<type_decl>              ::=  type <identifier> is <type>
<type>                   ::= <type1> {<lsb> <rsb>}
<type1>                  ::=  integer | real | bool | string | void | byte
                              <interface_type> | <function_type> |
                              <channel_type> | <struct_type> |
                              <enum_type> | <identifier>
<interface_type>         ::=  interface <lrb> <named_channel_list> <rrb>
<named_channel_list>     ::=  <named_channels [<semi_colon> <named_channel_list>]
<named_channels>         ::=  <channel_type> <identifier> [<comma> <identifier> ]
<channel_type>           ::=  <direction> <type>
<direction>              ::=  in | out
<struct_type>            ::=  struct <lrb> <named_param_list> <rrb>
<type_list>              ::=  <type> [<semi_colon> <type_list>]
<enum_type>              ::=  enum <lrb> <identifier_list> <rrb>
<function_type>          ::=  function <lrb> <type_list> <rrb> <colon> <type>
```

9.4 Sequencing

```
<sequence>               ::= <statement> [<semi_colon> <sequence>]
<statement>              ::= <decl> | <catch_clause>
<block>                  ::= <lcb> <sequence> <rcb>
<clause_sequence>        ::= <catch_clause> [ <semi_colon> <clause_sequence>]
<catch_clause>           ::= <try_except> | <clause>
```

9.5 Value Declarations

```
<decl>                   ::=  <value_decl> | <function_decl> |  <type_decl>
<value_decl>             ::=  <identifier> [ <equals> <value> ]
<value>                  ::=  <clause> | <value_constructor>
```

9.6 Value Constructors

```
<value_constructor>      ::=  new <value_def>
<value_def>              ::= <array_value> | <component_value> |
                              <channel_value> | <struct_value>
<array_value>            ::=  <type1> <dim_expr> of <expression>
<dim_expr>               ::=  <lsb> <int_literal> <rsb> [<dim_expr>]
<component_value>        ::= <identifier> <lrb> [ <clause_list> ] <rrb>
<channel_value>          ::=  <channel_type>
<value_decl_list>        ::=  <value_decl> [<comma> < value_decl_list>]
<struct_value>           ::=  <identifier> <lrb> [ <clause_list> ] <rrb>
```

9.7 Clauses

```
<try_except>             ::= try <clause> except <clause>
```

<clause>	::= <if_clause> | <for_clause> | <switch_clause> | <assign_clause> | <simple_decl> <connect_clause> | <disconnect_clause> | <select_clause> | <send_clause> | <receive_clause> | <project_clause> | | <any> **stop** [<name>] | <expression>
<if_clause>	::= **if** <lrb> <clause> <rrb> <clause> [**else** <clause>]
<for_clause>	::= **for** <lrb> <identifier> <equals> <expression> <dotdot> <expression> [<colon>[+|-]<integer_literal>]<rrb> <clause
<switch_clause>	::= **switch** <lrb> <expression> <rrb> <lcb> <switch_list> [**default** <colon> <clause>] <rcb>
<switch_list>	::= <clause_list> <colon> <clause> [<semi_colon> <switch_list>]
<clause_list>	::= <clause> [<comma> <clause_list]
<assign_clause>	::= <name> <assign_op> <expression>
<simple_decl>	::= <identifier> <equals> <expression>
<connect_clause>	::= **connect** <expression> **to** <expression>
<disconnect_clause>	::= **disconnect** <expression>
<select_clause>	::= **select** <lcb> <select_list> [**default** <colon> <clause>] <rcb>
<select_list>	::= **receive** <identifier> **from** <name> [**when** <clause>]<colon> <clause> [<select_list>]
<send_clause>	::= **send** [<expression>] **on** <name>
<receive_clause>	::= **receive from** <name>
<project_clause>	::= **project** <clause> **as** <identifier> <lcb> <project_list> [**default** <colon> <clause>] <rcb>
<project_list>	::= <type> <colon> <clause> [<project_list>]
<any>	::= **any**<lrb> <expression> <rrb>

9.8 Expressions

<expression>	::= <exp1> {<or> <exp1>}
<exp1>	::= <exp2> {<and> <exp2>}
<exp2>	::= [not_op] <exp3> [<rel_op> <exp3>]
<exp3>	::= <exp4> { <add_op> <exp4> }
<exp4>	::= <exp5> { <mult_op> <exp4> }
<exp5>	::= [add_op] <exp6>
<exp6>	::= <literal> | <lrb> <clause> <rrb> | <expression> <lrb> [<clause_list>] <rrb> | // application <name> | <clause_block>
<clause_block>	::= <lcb> <clause_sequence> <rcb>
<name>	::= <identifier> | <dereference>
<dereference>	::= <component_struct_dereference> | <array _dereference>

```
<component_struct_dereference> ::= <expression> { <dot> <identifier> }
<array_struct_dereference> ::= <expression> <lsb> [<clause_list>] <rsb>
```

9.9 Literals

```
<literal>              ::= <int_literal> | <real_literal> | <bool_literal> |
                           <string_literal> | <byte_literal>
<int_literal>          ::= [add_op] digit {digit}
<byte_literal>         ::= # [digit {digit}
<real_literal          ::= int_literal.{digit}[e <int_literal>]
<bool_literal>         ::= true | false
<string_literal>       ::= " {<char>} "
```

9.10 Misc

```
<assign_op>            ::= :=
<add_op>               ::= + | -
<and>                  ::= and
<or>                   ::= or
<not_op>               ::= !
<mult_op>              ::= <int_mult_op> | <real_mult_op> | <string_mult_op> |
<int_mult_op>          ::= * | / | %
<real_mult_op>         ::= * | /
<string_mult_op>       ::= ++
<rel_op>               ::= <eq_op> | <co_op> |
<eq_op>                ::= == | !=
<co_op>                ::= < | <= | > | >=
<dotdot>               ::= ..
<identifier>           ::= <letter> [<id_follow>]
<id_follow>            ::= <letter> [<id_follow>] | <digit> [<id_follow>]
<letter>               ::= a | b | c | d | e | f | g | h | i | j | k | l | m |
                           n | o | p | q | r | s | t | u | v | w | x | y | z |
                           A | B | C | D | E | F | G | H | I | J | K | L | M |
                           N | O | P | Q | R | S | T | U | V | W | X | Y | Z
<digit>                ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<char>                 ::= <UTF-8 character> except " and \ | <escape_sequence>
<escape_sequence>      ::= \\ | \n | …
<input character>      ::= UTF-8 character set
```

9.11 Index of keywords, literals and predefined names

and
any page 4, section 4.6
as page **Error! Bookmark not defined.**, section **Error! Reference source not found.**
behaviour page 7, section 5
bool page 4,section 4.3
byte page 4, section 4.4
component page 7, section 5
connect page 4, section 5; page 10, section 6.6
constructor page , section; page 7, section 5
default in *select*: page 1, section 6.10; in *switch*:
disconnect page 4, section 5; page 10, section 6.7
else page 8, section 6.1
enum page 5, section 4.7
except page 12, section 6.11
false page 4, section 4.3
for page 9, section 6.2
from page 11, section 6.9
function page 6, section 4.12
if page 8, section 6.1
integer page 4, section 4.1
interface page 6, section 4.11
new arrays: page , section; components: structs: …
of page 5, section 4.9
on page 10, section 6.8
or page 13, section 7.1
presents page 7, section 5
project page **Error! Bookmark not defined.**, section **Error! Reference source not found.**
real page 4, section 4.2
receive page 11, section 6.9
select page11 , section 6.10
send page 10, section 6.8
stop page 7, section 5 more??
string page 4, section 4.5
struct page , section c.f new
switch page , section
to page , section
true page 4, section 4.3
try page 12, section 6.11
type page 5, section 4.8
void page , section
when page 11, section 6.10

10 Appendix B: Type Rules

11 Appendix C: Predefined Environment

Standard functions to be defined