kiltera language reference

Ernesto Posse

December 8, 2006

Contents

1 Introduction **2**

2 Execution **2**
2.1 Interaction . . . 3
2.2 Mobility . . . 3
2.3 Time . . . 4

3 Data **4**

4 Lexical analysis **5**
4.1 Tokens . . . 5
4.2 Comments . . . 6

5 Expressions **6**
5.1 Atomic expressions . . . 7
5.2 Arithmetic expressions . . . 9
5.3 Boolean expressions . . . 9
5.4 String expressions . . . 9
5.5 Tuple (sequence) expressions . . . 9
5.6 Channel expressions . . . 10

6 Pattern-matching **10**

7 Processes **12**
7.1 The nil process . . . 12
7.2 Basic processes . . . 12
7.3 Conditional processes . . . 15
7.4 Local variable definitions . . . 16
7.5 Match processes . . . 16
7.6 Channel definitions . . . 17
7.7 Sequential processes . . . 18
7.8 Parallel processes . . . 19

7.9 Local process definitions 20
7.10 Process instantiation 21
7.11 Local function definitions 21
7.12 Interrupt processes 22
7.13 Wait processes 22
7.14 Timeout processes 22

8 Modules **23**
8.1 Toplevel process definitions 23
8.2 Toplevel function definitions 24

9 Built-in functions **24**

References **24**

1 Introduction

kiltera is a language for describing concurrent, interacting, mobile processes which "live" over time. We refer to kiltera as a *modelling* language as its terms can be seen as *models* of processes. We use the terms "model," "term," "process," "component" and "system" interchangeably.

kiltera is inspired by several languages, but prehaps the most relevant is Milner's π-calculus [8], which forms the basis of its semantics. Other languages that have inspired kiltera's design include Erlang [1], CCS [7], Timed CSP [3], [9], LOTOS [4], Real-time ACP [2], OCCAM [5], Python [10] and ML [6].

2 Execution

A kiltera model consists of one or more dynamic processes. Each process is an independent computational unit and proceeds concurrently with all other processes. Processes execute a number of different actions. The most important kind of actions performed by processes are communication actions.

Processes are dynamic entities, this is, at any given point in time a process is in a particular *state*. Each action executed by a process changes its state. The state determines the set of possible actions which a process can execute.

A process is not necessarily a purely sequential computation, as it may be itself composed of parallel subprocesses. The state of a process with several concurrent subprocesses is the combination of the state of the subprocesses. Hence, a process in a given state may have more than one possible action to execute.

The order in which actions of concurrent processes are executed is irrelevant, except when they are interactions between concurrent processes.

2.1 Interaction

Processes communicate through channels by message-passing. Processes have ports. A channel connects two or more processes through their ports. There are two basic communication operations: sending a message and receiving a message through some channel. When a process sends or receives a message it does so by specifying the port connected to the relevant channel.

There are two kinds of channels: synchronous and asynchronous.

Communication through a synchronous channel means that the sender and the receiver of a message engage in the interaction simultaneously. In other words, execution of a communication action (sending or receiving a message) is a blocking operation: the process executing the action will be blocked until some other process is ready to engage in the interaction. If a process attempts a receiving operation on a syncrhonous channel, it will be blocked until some other process sends a message through that channel, and dually, if a process attempts a sending operation on a synchronous channel, it will also be blocked until some other process executes a receiving action on that channel.

An alternative way of viewing synchronous communication is in terms of acknowledgement: a send operation waits for acknowledgment from the receiver before completion.

Communication through an asynchronous channel means that the sender and receiver do not need to engage in the interaction simultaneously. In other words, only the receiving operation is blocking, while the sender of a message can proceed with execution without waiting for acknowledgement from the receiver.

Channels can connect more than two processes, but communication is by "unicasting" or "two-way" communication rather than "multicasting" or "multi-way" communication. This is, when a process sends a message through a channel connected to two or more receivers, only one of the receivers will get the message and the rest will remain blocked. The selection is non-deterministic: the receivers are considered to be competing for the message. The same is true if there are many senders and one receiver.

Processes can also communicate through shared variables, and this gives rise to the usual complications of the shared-memory style of process communication.

2.2 Mobility

A *process network* is a set of processes connected through channels. A *configuration* is a particular topology of this network. Channels are first-class values and therefore can be communicated between processes. This means that the configuration of a network can change dynamically when processes execute. In particular processes can acquire access to channels and therfore to other processes to which they didn't have access. This is known as *mobility*.

2.3 Time

kiltera processes execute over time (whether it is logical or physical time.) The execution of a process occurs with respect to a global clock. The time base is the real numbers.

The execution of each action is an *event* which takes place at a particular point in time. Most normal actions do not take (logical) time to complete.

Synchronization actions (blocking actions such as waiting to receive a message, or sending a message over a synchronous channel) might take some time: from the point in time when the action is initiated or attempted until the point in time when the synchronization (exchange of information) actually occurs: for instance, if process P_1 attempts to receive a message through some channel a at time t_0 but no input is available on the channel at that time, it will block and wait until some other process sends data. When another process P_2 sends data through a at time $t_1 > t_0$ then synchronization occurs, and the receiving action of P_1 is said to have taken $t_1 - t_0$ time units. The actual synchronization is not considered to take any time itself.

Processes can be made to wait for a given amount of time, of equivalently, a process may schedule events in the future. Processes can also specify *timeouts*: the scheduling of future events which cancels another process which has not finished. Processes can also meassure the passage of time and change their behaviour accordingly.

3 Data

kiltera models manipulate several types of data. Each data value can be communicated between processes, assigned to variables or used in expressions.

The data types[1] are the following:

- The unit type which consists of a unique value $\perp$.
- The boolean type which consists of two values $true$ and $false$.
- The integer type which consists of the integers.
- The float type which consists of the rational numbers.
- The string type which consists of arbitrary character strings.
- The channel type which consists of channel objects,

and

- The tuple type which consists of tuples of the form $(v_1, v_2, ..., v_n)$ where each v_i is a data value from any type.

[1] Typing is not enforced in the current version of kiltera.

4 Lexical analysis

A kiltera interpreter, simulator, compiler or analysis tool takes as input some text file or string and uses a *lexer* to produce a stream of *tokens* which in turn is fed to a *parser* in order to generate an *abstract syntax tree* which can be processed.

This section describes the set of tokens that must be recognized by any kiltera lexer.

4.1 Tokens

The tokens recognized by kiltera are divided into literals, identifiers, keywords, operators, brackets, indentation and others.

4.1.1 Literals

There are several types of literals:

1. The unit literal: `unit`
2. Boolean constants: `true`, and `false`
3. Number literals: integers and rational numbers in decimal format: `0`, `1`, `-2`, `1.618`, ...
4. String literals: an arbitrary sequence of characters enclosed in double quotes: `"this is a string literal"`

4.1.2 Identifiers

An identifier is an alphanumeric sequence of characters which may include the underscore character '_' and which begins with a letter.

4.1.3 Keywords

Keywords cannot be used as identifiers. The following lists all kiltera's keywords.

```
after  and  at  async  channel  channels  do  else  false  for
from  function  if  import  in  interrupt  let  main  match
module  nil  not or  par  print  process  receive  send
seq  sync  then  timeout  to  true  unit  wait  with
```

4.1.4 Operators

Table 1 lists kiltera's operators in order of precedence, from highest to lowest.

Precedence	Operators
1	Unary `-`, `not`
2	`*`, `/`, `%`
3	`+`, `-`
4	`<`, `>`, `<=`, `>=`
5	`=`, `!=`
6	`and`
7	`or`

Table 1: Operators

4.1.5 Brackets

kiltera uses the following brackets:

```
( )  [ ]
```

4.1.6 Indentation

kiltera is sensitive to indentation. Indentation is used to describe nesting. Indentation can be given in terms of white spaces or tab characters.

Whenever the lexer finds a change of indentation it produces an appropriate number of "`Indent`" and "`Dedent`" tokens which are used by the parser to delimit nested blocks. If the indentation level is increased, one "`Indent`" token is generated. If the indentation level is decreased, as many "`Dedent`" tokens are generated as required to match the new indentation level.

4.1.7 Other symbols

Other symbols used in kiltera are

```
->  ,  :  |  :=
```

4.2 Comments

Comments are indicated with the pound character (#). A comment begins with the pound character (#) and ends with a new line character.

The lexer does not produce any token for comments, and thus, comments are ignored by the parser.

5 Expressions

An expression is a syntactic element which can be evaluated to yield a data value from a data type described in section 3.

There are six types of expressions:

1. Atomic expressions
2. Arithmetic expressions
3. Boolean expressions
4. String expressions
5. Tuple (sequence) expressions
6. Channel expressions

5.1 Atomic expressions

An atomic expression is one of the following:

1. A constant
2. A variable
3. A tuple access
4. A function call
5. A channel constructor

5.1.1 Constants

Constants are literals as described in section 4.1.1. The value of a constant is the corresponding data value as described in section 3.

5.1.2 Variables

A variable is an identifier as described in section 4.1.2. The value of a variable depends on the environment in which it occurs. Its value is whatever value the environment associates with the variable's name.

Variables must be declared, either as parameters of a process (see sections 7.9 and 8.1,) parameters of a function (see sections 7.11 and 8.2,) or by a local variable definition (see section 7.4.)

Operationally, the environment is a stack of frames. Each frame is a table associating a name with a value. The value of a variable is obtained by looking up its name in the environment from the topmost frame downwards.

Frames are created by different language constructs, such as function calls, process instantiation, local variable declaration, pattern matching, etc.

5.1.3 Tuple access

A tuple access is an expression used to access an element of a tuple (see section 5.5) by index. It has one of the following forms:

<variable> `[` *<arithmetic_expression>* `]`

or

<tuple_expression> `[` *<arithmetic_expression>* `]`

Its value is the value of the element $n+1$ of the tuple t where n is the value of the arithmetic expression and t is the value of the variable or tuple expression. This is, the first element of the tuple has index 0 and the last has index $k-1$ where k is the size of the tuple.

5.1.4 Function calls

A function call is of the form

<variable> `(` *<expr_1>*`,` *<expr_2>*`,` ..., *<expr_n>* `)`

Its value is the value of the expression which results from replacing v_1, v_2, ..., v_n in the body of f where each v_i is the value of *<expr_i>*, and f is the value of *<variable>* in the current environment, and which must be a function as defined in section 7.11 or section 8.2.

Operationally this results in creating a frame on top of the current environment, associating each parameter of the function with the value of the corresponding argument and executing the body of the function in such extended environment[2].

There are some built-in functions described in section 9.

5.1.5 The channel constructor

The channel constructor is an expression of the form:

```
channel
```

or

```
async channel
```

or

```
sync channel
```

Its value is a new asynchronous (resp. synchronous) channel. If no channel class is specified (the first form) the default is an asynchronous channel.

[2] This could also be implemented by inlining, since functions here are pure functions with no side-effects.

5.2 Arithmetic expressions

An arithmetic expression is an expression which evaluates to a number. It can have atoms, parenthesis and the arithmetic operators `+`, `-`, `*`, `/`, `%` (modulo) with the usual meaning and precedence rules.

5.3 Boolean expressions

A boolean expression is an expression which evaluates to a boolean value *true* or *false*. It can have atoms, parenthesis and the boolean operators `and`, `or`, `not`, and the comparison relations `<`, `<=`, `>`, `>=`, `=`, `!=` (different) with the usual meaning and precedence rules.

5.4 String expressions

A string expression is an expression which evaluates to a string. Currently only atoms can be string expressions.

5.5 Tuple (sequence) expressions

A tuple expression is an expression whose value is a tuple, this is, an ordered sequence of values (which themselves may be any legal kiltera value.)

There are two kinds of tuple expressions:

1. Explicit or extensive (called explicit tuples)
2. Implicit or comprehensive (called comprehensions)

5.5.1 Explicit tuples

An explicit tuple has the form

`(` *<expr_1>*`,` *<expr_2>*`,` ..., *<expr_n>* `)`

The value of an explicit tuple is the sequence $(v_1, v_2, ..., v_n)$ where each v_i is the value of *<expr_i>* in the current environment.

5.5.2 Comprehensions

A comprehension has the form

`(`*<expr>* `for` *<pattern>* `in` *<tuple_expression>*`)`

or

`(`*<expr>* `for` *<pattern>* `in` *<tuple_expression>* `if` *<boolean_expression>*`)`

where *<pattern>* is an expression, typically a variable or a tuple expression with free variables.

The value of the second form of comprehension is the tuple $(v_1, v_2, ..., v_n)$ where each v_i is the value of <*expr*> such that for each item w_j in the value of <*tuple_ expression*> that matches the pattern <*pattern*> and the condition <*boolean_ expression*> is true.

More precisely, the value of the second form of comprehension is the tuple $(v_1, v_2, ..., v_n)$ where each v_i is the value of <*expr*> with all its free variables replaced according to the substitution that results from matching[3] the <*pattern*> with an element w_j of t (the value of <*tuple_ expression*>) as long as the value of <*boolean_ expression*> is true.

Intuitively, the tuple expression <*tuple_ expression*> is evaluated, yielding a tuple $t = (w_1, w_2, ..., w_m)$, and each data value w_k is matched against the pattern <*pattern*>. If the data does not match the pattern, it is ignored, and the next item is considered. If the data value matches the pattern, it yields a *substitution*, i.e. a set of variable bindings, associating each free variable in the pattern with the corresponding data in w_k. If, in the environment extended with these new bindings, the <*boolean_ expression*> evaluates to *true*, then the expression <*expr*> is evaluated in this extended environment, and its value v_i is added to the resulting tuple.

The first form of comprehension

(<*expr*> `for` <*pattern*> `in` <*tuple_ expression*>)

is equivalent to

(<*expr*> `for` <*pattern*> `in` <*tuple_ expression*> `if true`)

5.6 Channel expressions

Channel expressions are expressions whose value is a channel. Currently the only kind of channel expressions are atomic expressions.

6 Pattern-matching

Pattern matching is an action performed in certain expressions and processes, particularly in comprehensive tuple expressions (see section 5.5.2) and in the *match* process (see section 7.5,) which is used to extract information from data.

Intuitively the idea is this: a *pattern* is an expression with free variables; a *datum* is a ground expression i.e. an expression with no free variables. A datum is said to "match" a pattern if it has the same structure, and all the corresponding constants agree. For instance the datum `(5, ("abc",true), (unit, 1.618))` matches the pattern `(5, ("abc", x), y)` but it does not match the pattern `(5, ("abc", x), false)`. If a datum matches a pattern, it yields a *substitution*, i.e. a mapping associating each free variable in the pattern with the corresponding

[3] See section 6 on pattern-matching.

piece of data occurring in the datum. In the previous example, for the succesful match, the resulting substitution is $\{x \mapsto true, y \mapsto (unit, 1.618)\}$.

A variable may occur more than once in the pattern. If this is the case, all occurrences of the variable must match the same data. For example, the datum

```
(2, (3, "abc" ), (false, (3, "abc")))
```

matches the pattern is

```
(2, z, (false, z))
```

with the substitution $\{z \mapsto (3, \text{“}abc\text{”})\}$, but the datum

```
(2, (3, "abc" ), (false, (3, "qwe")))
```

does not match the same pattern.

More precisely, we define pattern-matching as follows: A substitution is a mapping of variables to data values. If s is a substitution, and x is a variable, we denote $s(x)$ for the value of x in the substitution s.

A datum d *matches* a pattern p with respect to a subsitution s and yielding a new substitution s' if one of the following cases is true:

- Both d and p are constants and they are the same constant, irrespective of s. In this case $s' = s$.
- p is a variable which does not appear in the substitution s, and d is any datum. In this case, s' is the result of extending s by adding the association $\{p \mapsto d\}$
- p is a variable which does appear in the substitution s, and $s(p) = d$. In this case, $s' = s$.
- p is a tuple expression of the form $(p_1, p_2, ..., p_n)$ and d is a tuple of the form $(d_1, d_2, ..., d_n)$ of the same length, and
  - d_1 matches p_1 with respect to s yielding s_1,
  - d_2 matches p_2 with respect to s_1 yielding s_2,
    ...
  - d_n matches p_n with respect to s_{n-1} yielding s_n

  In this case, $s' = s_n$.

If none of these cases succeeds, there is no match and the resulting substitution is empty.

7 Processes

There are several kinds of processes in kiltera.

1. The nil process
2. Basic processes
3. Conditional processes
4. Local variable definitions
5. Match processes
6. Channel definitions
7. Sequential processes
8. Parallel processes
9. Local process definitions
10. Interrupt processes
11. Wait processes
12. Timeout processes

7.1 The nil process

The nil process is written

```
nil
```

It represents the dead process: it does nothing, and cannot interact with any process.

7.2 Basic processes

Basic processes are processes that perform a single action and then proceed to execute some process (called the *continuation*.) Actions are divided into two kinds: external and internal. External actions are those which involve interaction, i.e. communication through some channel. These actions are also called *observable* actions. Internal actions are those which are not observable.

The following are the possible actions:

1. Output: send actions (with or without timing)
2. Input: receive actions (with or without timing)
3. Assignment actions

4. Print actions

The input (receive) and output (send) actions are external actions. The wait and print actions are internal. The assignment action is normally considered an internal action, but since it can be used to modify the value of a shared variable it can also be considered an external action, since it provides an alternative approach to inter-process communication.

In general a basic process has the form

```
<action> ->
    <continuation>
```

where <*continuation*> is any process, or

```
<action> at <variable> ->
    <continuation>
```

for the actions with timing. Note that the continuation process *must* be indented with respect to the action.

If the contination is the nil process, there is no need to write it (or the arrow:)

```
<action>
```

is equivalent to

```
<action> ->
    nil
```

7.2.1 Send actions (with or without timing)

Without timing A send action without timing is of the form

```
send <expression> to <channel_reference> ->
    <continuation>
```

where <*channel_reference*> is any expression whose value is a channel.

The meaning of a send action is to send the value of the expression through the given channel. If the channel is synchronous, the process will be blocked until some other process receives the message. Otherwise the send action finishes immediately and the process continues with the <*coontinuation*>.

With timing A send action without timing is of the form

```
send <expression> to <channel_reference> at <variable> ->
    <continuation>
```

where <*channel_reference*> is any expression whose value is a channel.

The meaning of a send action with timing is as without timing, but once the action is completed, the variable is bound to the amount of time it took to complete, from when it was initially attempted. In other words, it is assigned the value of the time elapsed since the process arrived at the current state where it attempts to send the message. As before, if the channel is synchronous, the process will be blocked until some other process receives the message, and therefore the variable will record the amount of time the process remains blocked. Otherwise the send action finishes immediately and the process continues with the <*coontinuation*> in which case the variable's value is 0.

7.2.2 Receive actions (with or without timing)

Without timing A receive action without timing is of the form

```
receive <variable> from <channel_reference> ->
    <continuation>
```

where <*channel_reference*> is any expression whose value is a channel.

The meaning of a receive action is to wait for a message comming from the given channel. Once the message arrives, the variable is bound to the value of that message. A receive action is always blocking. Once the message is received, the process proceeds with the <*coontinuation*>.

With timing A receive action without timing is of the form

```
receive <variable> from <channel_reference> at <vari-
able> ->
    <continuation>
```

where <*channel_reference*> is any expression whose value is a channel.

The meaning of a receive action with timing is as without timing, but once the action is completed, the variable is bound to the amount of time it took to complete, from when it was initially attempted. In other words, it is assigned the value of the time elapsed since the process arrived at the current state where it attempts to receive the message. As before, the process will be blocked until some other process sends a message through that channel, and therefore the variable will record the amount of time the process remains blocked.

7.2.3 Assignment actions

An assignment action it of the form

```
<variable> := <expression> ->
    <continuation>
```

Its meaning is to evaluate the expression in the current environment and assign the resulting value to the variable in the current environment. Since the language has lexical scoping, this might mean looking up the variable in the environment's frame stack. Since two or more processes may share an environment, the effect of assigning a value to a variable may be observed by other processes, and thus may influence their behaviour.

A variable can be assigned a value only if it was declared with a local definition (let, see section 7.4,) or if it is a parameter of a process (see sections 7.9 and 8.1.)

7.2.4 Print actions

A print action is of the form

```
print <expression> ->
    <continuation>
```

Its meaning is to print the value of the expression to standard output.

7.3 Conditional processes

Conditional processes have the form

```
if <boolean_expression> then
    <process_1>
else
    <process_2>
```

or

```
if <boolean_expression> then
    <process_1>
```

which is equivalent to

```
if <boolean_expression> then
    <process_1>
else
    nil
```

The meaning of a conditional process is to evaluate the boolean expression in the current environment and if it is true, continue with *<process_1>*, otherwise continue with *<process_2>*.

7.4 Local variable definitions

Local variable definitions have the form

```
let <variable> = <expression> in
    <some_process>
```

or

```
let <variable_1> = <expression_1>
and <variable_2> = <expression_2>
and ...
...
and <variable_n> = <expression_n> in
    <some_process>
```

The meaning of this construct is to bind each variable to the value of the corresponding expression and execute the process *<some_process>* in an environment enhanced with these bindings. This defines a lexical scope for variables, so the proces *<some_process>* is the scope of the variables, and when the process finished, the frame with the local bindings is discarded.

7.5 Match processes

A match process has the form

```
match <expression> with
    <pattern_1> ->
        <process_1>
|   <pattern_2> ->
        <process_2>
| ...
...
|   <pattern_n> ->
        <process_n>
```

Note that there may be only one alternative.

The meaning is to evaluate the expression *<expression>*, and match that value with each pattern from top to bottom according to the pattern-matching procedure described in section 6. If the value matches some pattern *<pattern_i>* yielding a substitution s then *<process_i>* is executed in an environment enhanced with the bindings specified by the substitution s. Only the first match found executes the corresponding process. If no pattern matches, then the process is equivalent to the nil process and nothing is done.

7.6 Channel definitions

Channel definitions are the mechanism used to create channels local to a process, or equivalently, to hide channels from the environment.

There are two general forms of channel definitions: single channels and channel arrays.

7.6.1 Single channels

Single channel definitions have the form

```
channel <variable_1>, <variable_2>, ..., <variable_n> in
    <some_process>
```

or

```
async channel <variable_1>, <variable_2>, ..., <vari-
able_n> in
    <some_process>
```

or

```
sync channel <variable_1>, <variable_2>, ..., <vari-
able_n> in
    <some_process>
```

The meaning of the first two forms is the same as that of

```
let <variable_1> = async channel
and <variable_2> = async channel
...
and <variable_n> = async channel in
    <some_process>
```

and respectively, for the last form (syncronous channels:)

```
let <variable_1> = sync channel
and <variable_2> = sync channel
...
and <variable_n> = sync channel in
    <some_process>
```

7.6.2 Channel arrays

Channel array definitions have the form

```
channels <variable> [ <arithmetic_expression> ] in
    <some_process>
```

or

```
async channels <variable> [ <arithmetic_expression> ] in
    <some_process>
```

or

```
sync channels <variable> [ <arithmetic_expression> ] in
    <some_process>
```

The meaning of the first two forms is the same as that of[4]

```
let <variable> = (async channel for i in range(0, <arith-
metic_expression>)) in
    <some_process>
```

and for the last form (synchronous channels:)

```
let <variable> = (sync channel for i in range(0, <arith-
metic_expression>)) in
    <some_process>
```

7.7 Sequential processes

Sequential processes are processes which, as the name implies, execute subprocesses in sequence.

There are two forms of sequential processes: normal and indexed. The normal form is used to specify a fixed and usually small number of processes to execute in sequence. The indexed form is used to specify loops, i.e. the repetition of a process.

7.7.1 Normal sequence

A normal sequence process has the form

```
seq
    <process_1>
    <process_2>
    ...
    <process_n>
```

There must be at least two sub-processes.

The meaning of this construct is to execute each process in turn one after the other. In particular, *<process_(i+1)>* begins only after *<process_i>* has finished. This implies that if *<process_i>* is a parallel process (see section 7.8) then the sequence construct acts as a *join* operator, i.e. all the subprocesses must finish before continuing.

[4] See section 9 for the meaning of the built-in function `range`.

7.7.2 Indexed sequence

An indexed sequence has the form

```
seq
    <some_process>
for <pattern> in <tuple_expression>
```

Note that there is only one process in the body of this construct.

The meaning is to execute *<some_process>* in an environment enhanced with the bindings produced by matching v the pattern *<pattern>*, for each item v of t that succesfully matches[5] the pattern, where t is the value of *<tuple_expression>*. Items of t that do not match the pattern are ignored. Each execution of *<some_process>* must finish completely (including all subprocesses,) before the next iteration begins.

7.8 Parallel processes

Parallel processes are processes that execute independently of other processes. The parallel construct introduces parallel processes. It is both a structural and a behavioural construct: it specifies some processes as subprocesses within the process, and it initiates them.

There are two forms of the parallel construct: normal and indexed. The normal construct is used to specify a fixed and usually small number of subprocesses to run in parallel. The indexed form is used to describe an array of processes, each of which is an instance of the same "process class." (see section 7.9 or section 8.1.)

7.8.1 Normal parallel processes

A normal parallel process has the form

```
par
    <process_1>
    <process_2>
    ...
    <process_n>
```

There must be at least two sub-processes.

The meaning of this construct is to execute all subprocesses in parallel. This construct finishes when all subprocesses have finished.

7.8.2 Indexed parallel processes

An indexed sequence has the form

[5] see section 6 on pattern-matching.

```
par
    <some_process>
for <pattern> in <tuple_expression>
```

Note that there is only one process in the body of this construct.

The meaning is to execute in parallel an instance of *<some_process>* in an environment enhanced with the bindings produced by matching v the pattern *<pattern>*, for each item v of t that succesfully matches[6] the pattern, where t is the value of *<tuple_expression>*. Items of t that do not match the pattern are ignored. In other words, as many instances of the process are created and executed in parallel, as items in t which match the pattern, and each such instance has variable bindings which resulted from the corresponding matching with the pattern.

7.9 Local process definitions

A process definition defines a *class*[7] of processes, this is, it defines the structure of a family of processes and can be *instantiated* to create individual representatives from it. It specifies a name for the class, its initial ports[8], optional parameters and the body which is an arbitrary process.

A process definition has the form

```
process <name>[<port_1>, ...,<port_m>]:
    <body>
```

or

```
process <name>[<port_1>,...,<port_m>](<param_1>,...,<param_n>):
    <body>
```

where *<body>* is any process.

A local process definition defines a set of processes within a limited scope. It has the form:

```
<process_definition_1>
<process_definition_2>
...
<process_definition_n>
in
    <some_process>
```

The meaning of this is analogous to that of the local variables definitions (see section 7.4.) The current environment is extended with a frame binding each process name to a *latent process*, which can be instantiated by *<some_process>* as described in section 7.10. The scope of the definitions is *<some_process>*.

[6] see section 6 on pattern-matching.

[7] A "class" of processes should not to be confused with the notion of "class" from Object-Oriented Programming.

[8] Ports may change due to mobility or may be "inherited" from the enclosing scope.

7.10 Process instantiation

Process instantiation has the form

<name> [*<ch_expr_1>*, ..., *<ch_expr_n>*]

or

<name> [*<ch_expr_1>*, ..., *<ch_expr_n>*](*<expr_1>*, ..., *<expr_k>*)

where each *<ch_expr_i>* is a *channel expression*, this is, an expression whose value is a channel. The *<name>* must be defined in the current scope, and introduced by a process definition (see section 7.9 or section 8.1.) The first form is used if the process definition has no additional parameters. The second is used if the definition has parameters. The number of arguments must be the same as the number of parameters expected.

The meaning is to create an instance of the process with each port connected to channel given by evaluating the corresponding channel expression, and passing as arguments the value of each argument expression. The body of the process is executed in an environment extended by a frame binding each port name to the corresponding channel and each parameter to the corresponding argument.

A process instantiation finishes when the body of the process finishes.

7.11 Local function definitions

A function definition defines *pure* functions, i.e. functions in the mathematical sense, with no side effects. Therefore, the body of a function cannot contain any state-based actions, only expressions.

A function definition has the form

```
function <name>(<param_1>, ..., <param_n>):
    <expression>
```

A local function definition defines a set of functions within a limited scope. It has the form:

```
<function_definition_1>
<function_definition_2>
...
<function_definition_n>
in
    <some_process>
```

The meaning of this is analogous to that of the local variables definitions (see section 7.4.) The current environment is extended with a frame binding each function name to a *latent function*, which can be called by an expression ocurring in *<some_process>* as described in section 5.1.4. The scope of the definitions is *<some_process>*.

7.12 Interrupt processes

The interrupt construct is used to describe processes that can be interrupted by another process.

An interrupt process has the form

```
do
    <process_1>
interrupt
    <process_2>
```

Its meaning is as follows: <*process_1*> and <*process_2*> are executed concurrently as long as <*process_2*> does not engage in any external interaction (receive or send.) If <*process_1*> finishes before <*process_2*> engages in external interaction then the whole process finishes. If, however, <*process_2*> engages in external interaction before <*process_1*> finishes then <*process_1*> is killed and control passes fully to <*process_2*>.

7.13 Wait processes

A wait process describes a process which blocks for a certain amount of time.

A wait process is of the form

```
wait <arithmetic_expression> ->
    <continuation>
```

The meaning of such process is to block for an amount of time d where d is the value of the arithmetic expression in the current environment.

This construct is equivalent to (see 7.14.)

```
timeout
    nil
after <arithmetic_expression> ->
    <continuation>
```

7.14 Timeout processes

The timeout construct is used to describe processes whose behaviour depends on the clock.

A timeout process has the form

```
timeout
    <process_1>
after <arithmetic_expression> ->
    <process_2>
```

Its meaning is as follows: <*process_1*> is executed, and if it does not engage in a purely external interaction (send or receive actions) before the given timeout t (the value of <*arithmetic_expression*>) is due, then it is eliminated, along with all of its subprocesses, and <*process_2*> is executed. If, on the other hand, <*process_1*> does engage in an external interaction before t time units, then it completes its execution, the timeout and <*process_2*> are ignored and the whole process finishes when <*process_1*> finishes.

8 Modules

Modules are the toplevel unit in kiltera. A module is given by a (text) file with a `.klt` extension.

A module has the form

```
module <name>

<top_level_function_definition_1>
<top_level_function_definition_2>
...
<top_level_function_definition_n>

<top_level_process_definition_1>
<top_level_process_definition_2>
...
<top_level_process_definition_m>
main
    <main_process>
```

Its meaning is to evaluate each top-level process and function definitions, add them to an initial, empty environment, and execute the main process in this environment.

8.1 Toplevel process definitions

A top-level process definition has the form

```
process <name>[<port_1>, ..., <port_m>]:
    <body>
```

or

```
process <name>[<port_1>,...,<port_m>](<param_1>,...,<param_n>):
    <body>
```

where <*body*> is any process.

Its meaning is the same as in section 7.9 but the scope of the definition is the entire module.

Name	Parameters	Returns	Description
`range`	`start`, `end`	`int tuple`	This returns a tuple of all integers from `start` to `end-1`.
`len`	`tuple`	`int`	This return the length of a tuple.
`zip`	`tuple` list	`tuple tuple`	This takes a list of tuples and returns a tuple where the i-th element is a tuple of the i-th element of each tuple in the list respectively.
`random`	`-`	`float`	This returns a number between 0 and 1, including 0 but excluding 1.

Table 2: Built-in functions.

8.2 Toplevel function definitions

A top-level function definition has the form

```
function <name>(<param_1>,...,<param_n>):
    <expression>
```

Its meaning is the same as in section 7.11 but the scope of the definition is the entire module.

9 Built-in functions

Table 2 lists the supported built-in functions.

References

[1] J. L. Armstrong and R. Virding. Erlang – an experimental telephony switching language. In *International Switching Symposium*, Stockholm, Sweden, May-June 1991.

[2] J. C. M. Baeten and J. A. Bergstra. Real time process algebra. Technical report, Amsterdam, 1990.

[3] C. A. R. Hoare. Communicating sequential processes. *Comm. ACM*, 21(8):666–677, August 1978.

[4] ISO. LOTOS - language of temporal ordering specification. Technical Report ISO DP 8807, 1987.

[5] David May. OCCAM. In *SIGPLAN '83*, pages 69–79, 1983.

[6] R. Milner. Standard ML proposal. *Polymorphism - The ML/LCF/Hope Newsletter*, 1(3), 1983.

[7] Robin Milner. *A Calculus of Communicating Systems.* Springer-Verlag, 1980.

[8] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. Reports ECS-LFCS-89-85 and 86, Computer Science Dept., University of Edinburgh, March 1989.

[9] Steve Schneider. *Concurrent and Real-time Systems: The CSP Approach.* John Wiley & Sons, Ltd., 2000.

[10] Guido van Rossum. *Python Reference Manual.* Stichting Mathematisch Centrum, Amsterdam, 1996.