The CPAL Programming Language

An introduction

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
1.01	October 2015	First public release with a description of the core of CPAL.	LF & NN
1.02	November 2015	Update samples syntax. Describe parser and interpreter commands. First description of I/Os on embedded targets.	LF & NN
1.03	December 2015	Major update with new material in Chapter 2 and 3, creation of Chapter 4 on simulation. Creation of an index.	LF & NN
1.04	April 2016	Additions: tasks with jitters, new scheduling policies, new command-line options, scientific notations for floating point literals, new Hz time unit.	LF & NN
1.05	October 2016	Updates: type hierarchy, timing annotations, time units, cpal_interpreter command-line options, UDP. Additions: cpal_lint and cpal2x tools, unsized arrays, float64 type.	LF & NN
1.06	January 2017	Updates: scheduling policies to reflect the policy annotations introduced in release 1.17, priority scheme for FPNP policy. Additions: event-triggered activations, CPAL marshalling.	LF & NN
1.07	April 2017	Additions: executing several states in a row (self.continue), post-state common code (finally block), currying, Process_State type, observing automaton.	LF & NN
1.08	July 2017	Additions: CPAL grammar in BNF, wakeup_at(), use of wakeup_in() and next_activations queue for event-triggered processes, how to manipulate strings.	NN

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
1.09	December 2018	Updates: Improvements to scenario and code-coverage documentation. Additions: durations64, writing to files, multi-interpreter mode.	NN
1.10	February 2020	Updates: section "timeline of the project". Additions: document durations64 type, --progress option for multi-interpreter.	NN

Contents

Chapter 1

Overview of CPAL

1.1 CPAL — the Cyber Physical Action Language

CPAL® is an acronym for the Cyber-Physical Action Language. CPAL is a low-code language meant to model, simulate, verify and program industrial Cyber-Physical Systems (CPS) which are the types of embedded systems that can be found in cars, planes, robots, UAV, medical devices, factory and home automation, power production and distribution, etc.

CPAL serves to describe both the functional behaviour of the functions, that is their code, as well as the functional architecture of the system (i.e., the set of functions, how they are activated, and the data flow among the functions). CPAL is a formal language in the sense that it has well defined concepts of states and transitions, and has been designed with the requirement to remain a small, simple and unambiguous language. CPAL's aims is to offer high-level abstractions that help to speed-up the development of embedded systems such as:

- Real-time scheduling mechanisms: processes can be activated with a user-defined period and offset relationships, or upon the occurrence of some external events. Active processes are scheduled according to a chosen scheduling policy,
- Finite State Machines (FSM): the logic of a process is defined as a Finite State Machine (FSM) — possibly organized in a hierarchical manner — where code can be executed in the states, or upon the firing of transitions. The semantics that is implemented in CPAL is the Mealy semantics which enables the control program to react faster on external events,
- Communication channels: to support control and data flow exchanges between processes, and read/write operations to hardware I/O ports with well-defined policies (FIFO or LIFO buffering, data overwriting).

CPAL supports two use-cases:

- a design exploration platform for CPS with main features being the formal description, the edition, graphical representation and simulation of CPS models,
- a real-time execution engine: the vision behind CPAL is that programs can be executed and verified in simulation mode on a workstation and the exact same code can be later run on an embedded board with the same run-time timing behaviour.

CPAL at a glance:

- Model, simulate and execute embedded systems

- Both a design exploration platform and a real-time execution environment
- Same temporal behavior in simulation mode as in real-time mode on target
- Concurrent programming: processes, communication channels, etc
- Support for Finite State Machines (FSM): StateFlow®-like FSM with conditional and timed transitions
- Memory allocation free
- Code that humans can easily understand
- A low-code platform that frees the developer from unnecessary low-level programming

1.2 Hello, world

The code below shows a process writing "Hello, world" on the console each time it is activated, here every 100ms. The process is first defined, like a class would be in OO languages, then it is instantiated.

```
processdef Hello_World()
{
  state Main {
    IO.println("Hello, World");
  }
}

process Hello_World: a_task[100ms]();
```

Run in Playground[1]

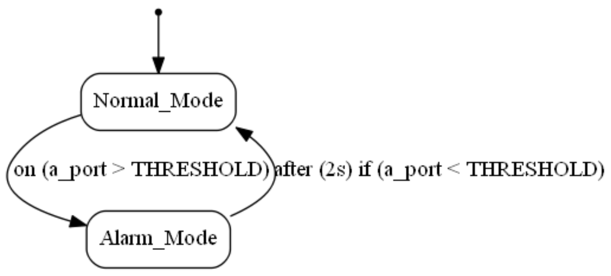
Although not everything from the program snippet below is to be understood at this point, it illustrates that more involved CPAL programs can remain both concise and rather intuitive.

```
processdef Monitor_Proc(
  in uint8: a_port,
  out bool: alarm)
{
  const uint8: THRESHOLD = 30;

  state Normal_Mode {
  /* ... */
  }
  on (a_port > THRESHOLD)
    {
      alarm = true;
    }
    to Alarm_Mode;

  state Alarm_Mode {
    /* ... */
  }
  after (2s) if (a_port < THRESHOLD)
    to Normal_Mode;
}
```

[1] http://www.designcps.com/cpal-playground?path=talks/cpal-intro/hello-world.cpal

```
var uint8: sensor#1; /* mapped to some I/O port */
var uint8: sensor#2; /* and updated upon activation of the processes */
var bool:  first_alarm  = false;
var bool:  second_alarm = false;

/* Instantiation of periodic monitoring processes*/
process Monitor_Proc: p1[500ms](sensor#1, first_alarm);
/* Second process is only executed when first_alarm is true */
process Monitor_Proc: p2[100ms][first_alarm](sensor#2, second_alarm);
```

The FSM embedded in a process can be visualized in the CPAL-Editor during the edition as shown below:



1.3 Timeline of the project

CPAL is jointly developed by RealTime-at-Work and the Critical Real-Time Embedded Systems (CRTES) research group at the University of Luxembourg. More information on the project can be found in an overview paper about CPAL and its intended use-cases that can be downloaded here[2]. Another good starting point is this paper published at DSM2016 about the high-level abstractions offered to the developer in CPAL (paper[3], slides[4]).

- 2011-2012 - initial language development at RealTime-at-Work[5] and feasibility assessment of a real-time model interpretation engine through a prototype implementation.
- From 2013 - CPAL is used in embedded system courses at the University of Luxembourg as a language to learn the principles of model-based design (MBD) and practice it by developing case-studies such as a capsule coffee machine, a simplified programmable floor robot and an elevator control system, etc.
- 2014 - CPAL is integrated in the network simulator RTaW-Pegase[6] as a language to describe the application's functional behaviour and new protocol layers. Development of the SOME/IP SD automotive protocol on top of Ethernet in CPAL which bas been used in a study with Daimler Cars published at Date2015[7].
- 2015 - CPAL is used in two applications: a smart parachute for UAVs (paper[8]) and for solving the Thales FTMV challenge (paper[9], slides[10]).

[2] https://www.designcps.com/wp-content/uploads/CPAL-ERTSS-2016.pdf
[3] https://www.designcps.com/wp-content/uploads/DSM2016_CPAL.pdf
[4] https://www.designcps.com/wp-content/uploads/DSM2016_CPAL_slides.pdf
[5] http://www.realtimeatwork
[6] http://www.realtimeatwork/software/rtaw-pegase
[7] http://www.realtimeatwork.com/wp-content/uploads/Date2015-SomeIP.pdf
[8] https://www.designcps.com/wp-content/uploads/UAV-ERTSS-2016.pdf
[9] http://hdl.handle.net/10993/21250
[10] https://www.designcps.com/wp-content/uploads/FMTV-challenge-slides.pdf

- 2016 - ongoing work to support IoT applications through gsm and IoT network connectivity. CPAL has been used for the design of a smart mobility system for two and three-wheelers implemented on an ARM mbed IoT board, with IBM Watson IoT platform as cloud backend (more information here[11]).
- 2016-2018 - CPAL is used by ONERA to simulate and prototype a code-upload protocol for the avionics domain. CPAL is used to simulate TTEthernet networks used in space launchers and study by fault-injection their robustness to transient failures (paper[12], slides[13]).
- 2018 - first CPAL to C code code generator available for Posix1003.1c and Zephyr platforms.
- 2018 - a CPAL parser and a CPAL model-transformation framework are open sourced (AGPL V3.0) (github repository[14], paper[15]).
- 2020 - major speed improvements (factor 5 to 10) in simulation mode to handle large models (hundreds of CPAL processes running in parallel).

1.4 Why a new language?

Innovation crucially relies on software today and the amount of software is growing exponentially. But the productivity gains in software development are much slower. Model-Driven Development is certainly an enabling technology to fill this gap but in our view programming languages still lack the high-level abstractions and automation features (e.g., “state the what, not the how”) that would make them more productive.

In addition, the design and development of embedded systems, especially ones with dependability constraints, necessitates the use of many best practices. None of the programming languages we are aware of are well suited to make the development of *safe* and *provably correct* embedded systems as quick and easy as possible:

- General purpose programming languages do not offer the right abstractions for today’s real-time embedded systems: programming periodic activities, real-time scheduling, time as a first class citizen, safe inter-process communication, native support for finite-state machines, high-level interfaces to I/Os, support for timing and formal verification, etc.
- Synchronous languages (Lustre, Signal, etc) are certainly great research outcomes but they impose many constraints and a programming style that does not suit everyone. In addition, the learning curve is steep.
- Development environments do not have to be complex and expensive, and turnover time (regeneration/rebuild/retest/redeploy steps) can be shortened to gain in productivity.
- Correctness starts with simplicity: there is no need to rely on millions of lines of code (code generator, compiler, linker, OS) to develop most embedded systems.
- Interpretation, though slower, brings benefits in terms of turnover time, security, error monitoring at run-time, updates at run-time, etc. For applications that cannot afford the slowdown of interpretation, CPAL includes a C code generator.

We believe that major productivity and quality improvements are still ahead of us through better programming languages and environments – CPAL is a contribution in that direction for the domain of CPS.

[11] https://www.designcps.com/news-events/
[12] https://www.realtimeatwork.com/wp-content/uploads/TTE-ERTSS2018.pdf
[13] https://www.realtimeatwork.com/wp-content/uploads/ERTS2018_TTE_color.pdf
[14] https://github.com/minimap-xl/nhc
[15] https://orbilu.uni.lu/bitstream/10993/41575/1/FT_Augmentation_preprint.pdf

Note

The abstract and concrete syntax of CPAL has been inspired by a number of diverse languages such as Eiffel, MISRA C and Erlang, model-based design products such as Matlab/Simulink® and Scade®, verification frameworks such as Promela/Spin (see this paper[a] for a comparison with Promela), simulation frameworks as Opnet®, and more generally what is usually referred to as the synchronous programming approach, such as the Giotto description language. Importantly, CPAL has been designed with the requirement to remain a small, simple and unambiguous language, easy to start with for the C or Java programmer, and less constraining than synchronous languages (e.g., multiple I/O operations per process activation are possible).

[a] https://www.designcps.com/wp-content/uploads/WFCS2017_Promela_vs_CPAL.pdf

Chapter 2

A tour of CPAL

2.1 Syntax of CPAL

CPAL is case-sensitive and its syntax is close to the syntax of the C language, as it is shown in the example below that illustrates the creation of a new type and the declaration of a variable of that type.

```
const uint32: N = 1024; /* a constant unsigned number on 32bits */

var float32: x;
var int8:    X; /* case-sensitive */
/* A name can contain underscores, sharps and numbers
   but not start with them */
var bool: does#it_4;

/* Define and use a Type T */
struct T
{
  uint32: a;
  int64:  b;
};
var T: t0 = {2, -4};

/* Only C-Style comment, no // */
```

The syntax of CPAL and the rules for naming identifiers are ones commonly used in programming languages:

- CPAL is a case-sensitive language, thus identifiers are case sensitive too,
- An identifier must be a word made-up of characters starting with an underscore or a letter,
- Identifiers' name can contain numbers and the # character but cannot start with these symbols,
- Keywords of the language cannot be used as identifiers,
- Statements must be terminated by a semicolon and can extend over several lines,
- CPAL programs can contain an arbitrary number of single-line and multi-line comments. Single-line or multi-line comments start with `/*` and end with `*/` (no comments starting with // and no nested comments).

The complete syntax of the language is described in Backus-Naur form in Chapter 7.

Note
CPAL, in the spirit of the MISRA C, does not include features that are neither desirable nor needed in the development of embedded systems with dependability constraints such as pointers and dynamic memory allocation.

2.2 Program structure

In CPAL there are functions and processes. Processes, also referred to as tasks, runnables or threads in other contexts, have their own dynamics: they are automatically activated at a specified rate or when specific conditions are met. Functions on the other hand are called whenever necessary from within processes or other functions.

A special function is `init()` which is executed only once at the startup of the program. The use of `init()` is optional but it is convenient in some cases for executing initialization code (e.g., hardware, random number generator initialization or scheduling event-triggered processes, see Section 2.9.2).

```
/* Definition of functions */
my_function(in uint32: a, out bool: flag)
{
/* ... */
}

/* Definition of processes */
processdef My_Process(
  in bool: doesX,
out uint32: aValue)
{
/* ... */
}

/* Global variables */
var uint32: a_global_variable;
var bool:   user_driven_var = true;

/* Instantiation of processes, aka Tasks or Runnables. */

/* Periodic process */
process My_Process: task1[500ms](userDrivenVar, aGlobalVariable);
/* Periodic with boolean condition */
process My_Process: task2[0.2Hz][userDrivenVar](userDrivenVar, aGlobalVariable);
```

Note
The `init()` function, if defined, will be executed once at the startup of the program. Through a command line option, it is possible to specify a function that is executed when the program exits (see Section 3.5).

2.3 Functions

Functions can be called from anywhere in CPAL: from a process state, a process transition (see Section 2.7.2) or another function like the `init()` function.

Functions can take zero, one or several arguments of any simple or complex types. In CPAL functions parameters are more generally referred to as *ports*. Parameters of a function whose value cannot be modified during the execution of the function are specified as `in` parameters. Parameters flagged as `out` can have their value changed in the function as illustrated in the code snippet below:

```
myfunc(in Item: an_item, out uint32: price)
{
  if (an_item.f == ORANGE) {
    price = an_item.quantity * 300;
  } else {
    price = an_item.quantity * 150;
  }
}
```

Note

CPAL does not allow functions with the same name and different arguments.

Functions, except the ones built in the language such as `time64.time()`, do not return a value and there is no `return` statement in CPAL. Data should be returned as `out` parameters. CPAL implements a mechanism called currying which allows to assign or test the output of a function as shown below:

```
/*
Licence Creative Commons CC0 - "No Rights Reserved"
@keywords function; structure; currying
*/

double(
    in  uint8: x,
    out uint8: y)
{
    y = 2 * x;
}
struct complex
{
    float64: real;
    float64: imag;
};


complex_add(
    in  complex: a,
    in  complex: b,
    out complex: res)
{
    res.real = a.real + b.real;
    res.imag = a.imag + b.imag;
}

init()
{
    var uint8:   result =     double(5);
    var complex: A      =     {
            3.0,
            2.5
    };
    var complex: B      =     {
            6.5,
            -2.3
    };
    var complex: res    =     complex_add(A, B);
    IO.println("res.real: %f  res.imag: %f", res.real, res.imag);
}
```

Run in Playground[1]

The currying mechanism works also for testing equality of native types:

```
/*
Licence Creative Commons CC0 - "No Rights Reserved"
@keywords equality; function; currying; time64
*/

add_1ms(
    in  time64: a,
    out time64: res)
{
    res = a + 1ms;
}

init()
{
    var time64: t1       = 999ms;
    var time64: expected = 1s;
    assert(expected == add_1ms(t1));
}
```

Run in Playground[2]

Important
A function must be either defined, or the prototype of the function be declared, before it can be used.

2.4 Data types

CPAL is a **strongly type language** with the following main characteristics:

- There are no untyped data and no pointers in CPAL,
- No memory is dynamically allocated or freed at run-time,
- Basic types: `bool`, `uint8`, `int64`, `float32`, `time64`, etc,
- User-defined types: arrays, enums and structures,
- Collections: stacks and queues, that allow to the addition and supression of elements at run-time,
- `process` is a built-in type for a recurrent activity of the system,

[1] http://www.designcps.com/cpal-playground?path=talks/cpal-intro/curry.cpal
[2] http://www.designcps.com/cpal-playground?path=talks/cpal-intro/curry_equality.cpal

- `Process_State` is a native type to save or test the current state of a process.

Note

There is no dynamic memory allocation. Thus, in the spirit of MISRA, a program cannot fail because of stack overflow after initialization.

2.4.1 Basic data types

The table below lists the basic data types available in CPAL:

Type	#Bytes	Range of values	Print format
uint8	1	0.. 255	%u
uint16	2	0.. 65535	%u
uint32	4	0..2^{32} -1	%u
uint64	8	0..2^{64}-1	%u
int8	1	-128.. 127	%d
int16	2	-65536..65535	%d
int32	5	-2^{31}..2^{31}-1	%d
int64	8	-2^{63}..2^{63}-1	%d
float32	4	-3.4e38..3.4e38	%f
float64	8	-1.7e308..1.7e308	%f
time64	8	0..2^64-1 ps	%t
duration64	8	-2^{63}..2^{63}-1 ps	%t
bool	1	false, true	%b

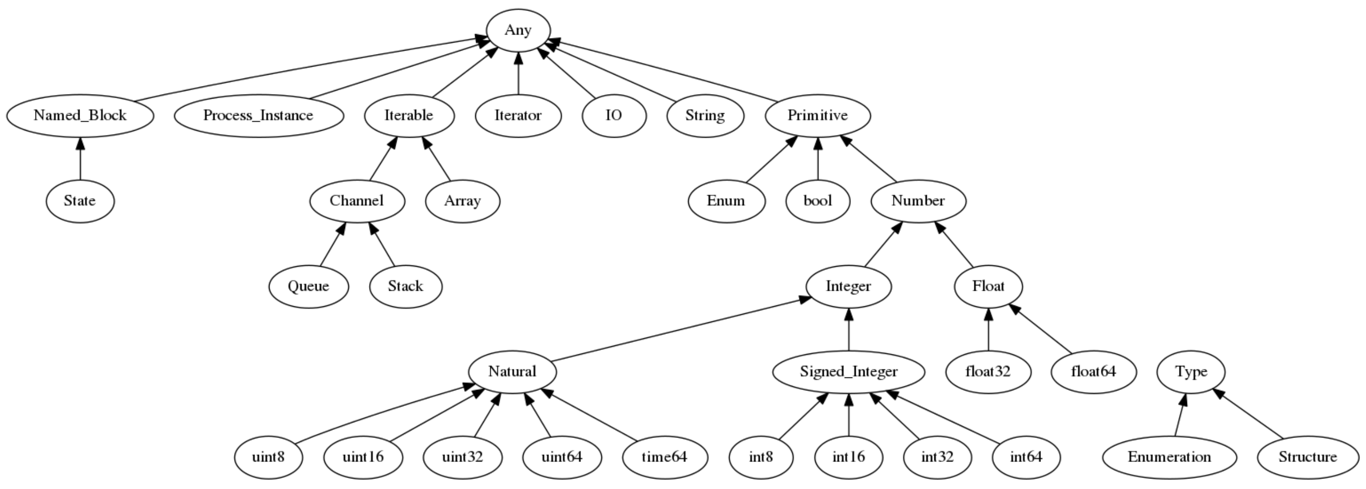
With the %u and %d format specifiers integer variables are printed out in decimal. They can be also displayed as ascii characters with %c and in hexadecimal with %x.

Note

Minimum and maximum value of each type is `type.FIRST` and `type.LAST`. Minimum and maximum between two variables is `type.min(a,b)` and `type.max(a,b)`. These two functions support an arbitrary number of arguments.

CPAL allows the use of scientific notation for floating-point data. Supported formats are `3.14e5`, `3.14E5`, `3.14e+5`, `3.14E+5`, `3.14e-5` and `3.14E-5`.

The figure below shows the hierarchy of the native types in CPAL. All leaf-types, plus `Named_block` (see Section 4.2.4) and `Channel` (see Section 2.6), are available in the language except `String` which is reserved for internal use (it is possible to manipulate strings through collections of characters, see Section 2.13.2). In the tree below, first letters in uppercase identify enumerations, structures or abstract types, while lowercase letters are for primitive types.



2.4.2 Conversion between basic types

CPAL provides the built-in functions listed in the table below to convert between any two primitive data types:

`uint8.as(x)`	`uint16.as(x)`	`uint32.as(x)`	`uint64.as(x)`
`time64.as(x)`	`bool.as(x)`	`int8.as(x)`	`int16.as(x)`
`int32.as(x)`	`int64.as(x)`	`float32.as(x)`	`float64.as(x)`
`duration64.as(x)`			

The function `type.as(x)` will convert the variable `x`, whatever its type, into a `type` variable. The result is saturated if destination type is not big enough. Examples of conversions are given in the program below:

```
init() {
  var uint64:  an_uint  = 52;
  var float64: a_float  = 52.36;
  /* time64 variables are positive while duration64 can be negative as well */
  var duration64: t1 = duration64.rand_uniform(-10s, 10s);
  /* non-zero values translates to true, positive or negative */
  assert(bool.as(an_uint) == bool.as(a_float * 1.0) == true);
  /* granularity of time is ps */
  assert(time64.as(an_uint) == time64.as(a_float) == 52ps);
  /* truncation to integer value */
  assert(uint64.as(time64.as(a_float)) == uint64.as(float32(a_float)) == 52);
  /* casting different sizes */
  assert(uint8.as(an_uint) == 52);
  /* saturation */
  assert(uint8.as(uint16.LAST) == uint8.LAST);
  assert(uint8.as(-2) == 0);
  assert(uint8.as(int8.FIRST) == 0);

}
```

Built-in functions are also available to re-interpret a data field storing a variable into another type. Such functions are for instance useful for interpreting data received from communication interfaces. The functions available are the following:

`uint8.cast(x)`	`uint16.cast(x)`	`uint32.cast(x)`	`uint64.cast(x)`
`time64.cast(x)`	`bool.cast(x)`	`int8.cast(x)`	`int16.cast(x)`

`int32.cast(x)`	`int64.cast(x)`	`float32.cast(x)`	`float64.cast(x)`
`duration64.cast(x)`			

The function `type.cast(x)` will read the memory area where `x` is stored and re-interpret it as a `type` variable. It should be noted that `x` can be an enumeration, although no assumption should be made on the value of the first element. Examples of binary re-interpretation are given in the program below:

```
enum Color
{
  BLUE,
  GREEN
};

init()
{
  var Color: enum1 = Color.BLUE;
  var Color: enum2 = Color.GREEN;
  var int8: minus_one = -1;
  assert(uint8.cast(time64.FIRST) == 0);
  assert(int32.cast(uint64.FIRST) == 0);
  assert(time64.cast(true) == 1ps);
  assert(time64.cast(false) == 0ps);
  assert(bool.cast(0) == false);
  assert(bool.cast(int64.LAST) == true);
  assert(uint8.cast(enum1)+1 == uint8.cast(enum2));
  assert(uint8.cast(minus_one) == uint8.LAST);
}
```

Warning
`type.cast()` semantics is binary re-interpretation and not conversion, the latter can be performed with `type.as()`.

2.4.3 Variables and constants

A variable holds a value which is compatible with the type of the variable, be it a basic type or a user-defined type. A variable can be global to the program, local to a function or local to a process. Local variables can only be declared at the beginning of a scope (i.e., beginning of the function/process or block of code enclosed by `{` and `}`) and are only known in their scope of creation. Variables of both basic types and user-defined types are by default initialized to zero at their creation (i.e., all bits are set to zero).

A variable that is local to a process can be declared as `static var` in which case it will retain its value accross successive executions of the process. If not static, the variable will be re-initialized at each execution of the process. The program below illustrates the declaration and initialization of variables of basic types and arrays thereof.

```
processdef A_Process() {
  static var uint8: current_state = 2;
  state Main{
    current_state = (current_state + 1) mod uint8.LAST;
  }
}
```

```
init() {
  var uint8: x;
  var int16: B_1[3] = {-1, 12}; /* partial init of an array */
  var bool: aFlag = true;
  var time64: t1 = time64.time() - 12ms5ps;
}
```

In addition to variables, it is possible to declare constant data of any types (all primitive types, arrays, structures, etc) except communication channels (queues and stacks) for which this does not make sense. Constants can be declared both at the global and local scope (i.e., within a function or a process).

```
const bool: VRAI = true;

processdef My_Proc()
{
  const uint32: V1 = 0xA1E;
  const uint32: V2 = 2590;
  state Main {
    assert(V1 == V2);
  }
}

const uint32: ARRAY_SIZE = 3;
var   uint32: my_array[ARRAY_SIZE];

const   time64: A_PERIOD = 100ms;
process My_Proc: a_task[A_PERIOD]();
```

Run in Playground[3]

Note
Although the language does not forbid it explicitly, it is a good practice to not use in a process (see Section 2.7) a global variable, such as a constant, not passed as an argument. This facilitates modularity, re-use and enables to trace and visualize the data flows between processes within the CPAL development environment.

2.4.4 Complex types: struct and enum

CPAL allows the definition of complex types such as enumerations and structures. Structures can be made up of primitive types, enumerations, complex types (including other structures thus) and collections.

```
enum Fruit
{
  APPLE,
  BANANA,
  ORANGE
};

struct Item
{
  uint32: quantity;
  Fruit:  f;
};
```

[3] http://www.designcps.com/cpal-playground?path=talks/cpal-intro/const.cpal

Note
There is no typedef in CPAL.

Warning
Enumerators must be prefixed by the name of the enum, e.g. `Fruit.APPLE`.

2.5 Basic constructs and operators

2.5.1 Comparison operators

The following comparison operators returning a boolean value are available in CPAL:

`==`: equal to	`!=`: not equal to	`>`: greater than
`<`: less than	`>=`: greater than or equal to	`<=`: less than or equal to

CPAL allows multiple comparisons such as `if (0 < a < 5)`.

Important
CPAL is a strongly typed language and comparing operands that do not have the same type is not allowed. Explicit conversions between types is however possible (see Section 2.4.2).

Note
CPAL does not allow to test the equality of two floating point values since testing equality of floating-point real numbers is error-prone.

2.5.2 Arithmetic operators

The following standard arithmetic operators are available in CPAL:

`=`: assignment	`+`: addition	`-`: subtraction	`mod`: modulo
`*`: multiplication	`/`: division	`-`: unary minus, e.g. `-x`	

Important
CPAL is a strongly typed language and arithmetic operations on operands that do not have the same type is not allowed. Explicit conversion between types is however possible (see Section 2.4.2).

Note
CPAL does not provide the prefix/postfix increment `++x` and decrement `--x` operators of the C/C++ language.

2.5.3 Boolean operators

CPAL provides a native bool type with the logical values true and false and the boolean operators and, or and not. The binary boolean operators associate left to right, like it is usual in programming languages.

```
var bool: a_boolean = true;
var bool: another_boolean = (not a_boolean) and (a_boolean or not a_boolean);
IO.println("Value of a_boolean is %b", a_boolean);
```

It should be noted that the and operator possesses the semantics of the “and then" operator in the Eiffel language, which means that a term is only evaluated if all preceding terms evaluates to true. For instance, the transition in the code snippet below cannot lead to a division by zero because the evaluation of the formula stops if the first term happens to be false:

```
state Main {
  /* ... */
}
on ((a != 0) and ((b / a) > 1))
  to Another_State;
```

In a similar manner, the or operator possesses the semantics of the “or else” operator in Eiffel, which means that the second term will not be evaluated if the first term evaluates to true.

2.5.4 Bitwise operators

CPAL includes the following standard bitwise operators listed in the table below:

&: bitwise and	|: bitwise or	^: bitwise xor
~: bitwise not	<<: right shift	>>: left shift

```
init()
{
    var uint8: a = 0x10;
    var uint8: b = 0x11;
    assert((a >> 4) == 1);
    assert((a << 4) == 0x100);
    assert((a & b) == 0x10);
    assert((a ^ b) == 1);
    assert((a | b) == 0x11);
    assert((~a) == 0xEF);
}
```

Note
all binary bitwise operators require operands of same type except shift operators which are allowed between any integer types (signed and unsigned).

2.5.5 if, for and while

CPAL provides the if/else conditional execution statement and 3 loop constructs:

- `for(init-statement; condition; post-statement)`,
- `while(condition) {...}`,
- `loop over`, which is a safe and efficient way to iterate over a collection (see Section 2.6.5).

The `break` and `continue` statements can be used in loops with the usual C/C++ semantics, as illustrated in the program below:

```
init()
{
  var uint8: i;
  for (i = 0; i < 10; i = i + 1) {
    if (i == 1) { /* skip the end of the loop */
      continue;
    }
    IO.println("loop1: i=%u",i);
  }

  i = 0;
  while (true) {
   if (i == 5) { /* exit the loop */
      break;
   }
   IO.println("loop2 i=%u",i);
   i = i + 1;
  }
}
```

Warning
The code executed in the `if` and `else` branches of an `if` statement must be enclosed by curly braces even if there is a single statement.

Note
CPAL does not provide a `switch` construct.

2.6 Arrays and collections

Arrays in CPAL possess the usual semantics of a group of elements identified by an index ranging from `0` to `a.max_size-1` where `a.max_size` is the size of the array `a`. Although CPAL natively supports only one-dimensional arrays, it is possible to create multidimensional arrays using the following idiom:

```
struct Array
{
  int32: y[3];
};
var Array: x[3];
```

The elements of the arrays can now be accessed with `x[i].y[j]`.

CPAL natively supports data structures grouping elements together with a specific internal structure, called collections, dedicated to store elements and exchange data between processes. On the contrary to arrays, true collections allow the addition and supression of elements at run-time. The collections available in CPAL are:

- `queue`: first-in first-out (FIFO) data buffers,
- `stack`: last-in first-out (LIFO) data buffers.

The types `queue` and `stack` both inherit from the native `channel` type (see Section 2.4.1). Prototypes of functions can thus have arguments of type `channel`, allowing at run-time the collection which is passed on to the function to be either a `queue` or a `stack` (see the code snippet in Section 2.6.5).

Note
An `array`, just like a `queue` and a `stack`, can be iterated with the `loop over` construct (see Section 2.6.4) but, unlike a `queue` and a `stack`, its size is fixed during the entire program execution.

2.6.1 Declaration of arrays and collections

The snippet of code below shows the instantiations of an array and the two supported types of collections. Collections and arrays can hold basic types, such as `uintX` here, but also user-defined types such as enums and structures.

```
init()
{
  /* A FIFO queue of maximum 10 unsigned integers */
  var queue<uint32>: a_queue_of_uint32[10];

  /* A LIFO queue of maximum 10 unsigned integers */
  var stack<uint32>: a_stack_of_uint32[10];

  /* A uni-dimensional array of uint8 */
  var uint8: data[20];

  assert(a_queue_of_uint32.max_size == a_stack_of_uint32.max_size == 10);
  assert(data.max_size==20);

}
```

Partial and empty initialization of arrays and collections are possible. Empty initialization of a collection is for instance needed to initialize a structure containing a collection as illustrated by the program below:

```
/*
Licence Creative Commons CC0 - "No Rights Reserved"
@keywords partial init; empty init; collection; stack; queue
*/

/* Partial init of queue, stack and arrays */
var queue<uint8>: q1[5]  = {1, 2};
var stack<uint8>: s1[10] = {3, 4, 5};
var uint8: a1[10] = {1, 2, 3};

/* Empty init of a collection needed in structure initialization */
```

```
struct A_Struct {
  stack<uint8>: s2[10];
  uint8: x;
};

var A_Struct: foo = { {}, 2 };

init() {
  assert(q1.count() == 2);
  assert(s1.count() == 3);
  assert(foo.s2.is_empty());
}
```

Run in Playground[4]

2.6.2 Operations on queues and stacks

The following primitives operate on queues and stacks:

- `push(item)`: insert a new element at the end of the collection,
- `pop()`: return the top element (i.e., head for a queue or tail for a stack) and remove it from the collection,
- `is_full()`, `is_empty()` and `count()`: test if the collection is empty, full and how many items it contains. Similarly, `not_full()` and `not_empty()` are also available,
- `peek()`: return the same element as `pop()` but does not change the content of the collection,
- `last_peek()`: return the last element of the collection (i.e., last element stored for a queue or first element stored for a stack) but does not change the content of the collection,
- `clear()`: remove all the elements from the collection,
- `choice_uniform()`: returns one of the elements at random with equiprobability.

```
assert(a_queue_of_uint32.is_empty());
assert(a_queue_of_uint32.not_full());
a_queue_of_uint32.push(10);
a_queue_of_uint32.push(11);
assert(a_queue_of_uint32.not_empty());
assert(a_queue_of_uint32.not_full());
assert(10 == a_queue_of_uint32.pop());

a_stack_of_uint32.push(10);
a_stack_of_uint32.push(11);
assert(11 == a_stack_of_uint32.pop());
```

[4] http://www.designcps.com/cpal-playground?path=talks/cpal-intro/partial_init.cpal

2.6.3 Sorted collections

Insertions in collections holding simple scalar types (integers and floating point values) can be performed in a sorted manner, using the `insert_sorted()` function. Precisely the new element is placed in the first place (starting from the head of the collection) such that the next element in the collection is strictly larger. Sorted insertions can be used to implement sorted queues, also known as priority queues. The use of `insert_sorted()` is shown in the program below:

```
/*
Licence Creative Commons CC0 - "No Rights Reserved"
@keywords queue; sorted queue; partial init
*/

init()
{
  var queue <uint32> : q[6] = {3,7}; /* partial init */
  q.push(4); /* 4 is placed at the tail of the queue */
  q.insert_sorted(5);  /* sorted insertion from the head of the queue */
  q.insert_sorted(1);  /* first position */
  q.insert_sorted(10); /* last position */
  loop over q with it {
    IO.print("%u ", it.current); /* 1 3 5 7 4 10 */
  }
}
```

Run in Playground[5]

Warning
If all elements of a collection have not been added using `insert_sorted()`, the collection may not be globally sorted like in the example above. The different ways of inserting elements gives the programmer flexibility to organize the elements of a collection.

2.6.4 Unsized arrays

Unsized arrays, also referred to as flexible arrays, allow the use of variable length arrays as function and process arguments. They enable the signature of the functions and processes taking arrays as argument remains generic whatever the dimensions of the arrays are. Be they unsized or with a fixed size, it is possible to iterate over arrays in a process or a function using a loop counter as shown below or with the `loop over` construct (see Section 2.6.5).

```
/*
Licence Creative Commons CC0 - "No Rights Reserved"
@keywords array, unconstrained; argument, unconstrained array
*/

my_function(in uint32: unsized_array[]) {
  var uint32: i;
  for (i = 0; i < unsized_array.max_size; i = i + 1) {
    IO.print("%u ", unsized_array[i]);
  }
  loop over unsized_array with it {
    IO.print("%u ", it.current);
```

[5] http://www.designcps.com/cpal-playground?path=talks/cpal-intro/insert-sorted.cpal

```
  }
}

init() {
  var uint32: a[3] = { 0, 1, 2 };
  const uint32: b[4] = { 3, 4, 5, 6 };
  my_function(a);
  my_function(b);
}
```

2.6.5 Iterating arrays and collections

Iterators provide a way to access data stored in arrays, stacks and queues. Iterators are meant to iterate through the elements of a collection, in order to perform operations such as reading elements or removing elements.

The use of iterators instead of simple loop counters has a number of advantages such as providing the same simple way to access all kinds of collections, a more efficient implementation for some collections and, when an element is added or removed, the fact that the iterator is transparently updated (no need for instance to change the loop counter manually). In addition, the use of iterators prevents *off-by-one* errors (e.g., a loop that iterates one time too many).

The following constructs and operators on iterators are available in CPAL:

- `loop over collection with iterator { /* body of the loop here */ }`: go through each of the element of the collection, the iterator that is used to access the elements is automatically declared.
- `iterator.index`: return the index of the current element,
- `iterator.current`: return the value of the current element,
- `iterator.is_last`: true if the current element is the last of the collection, false otherwise.
- `iterator.remove_current(option)`: remove the current element and move the iterator to the next element of the collection. This function works on all collections except on arrays whose length cannot vary at run-time.

- `continue`: terminate the loop for the current element, and restart at the beginning of the loop with the next element (like in C),
- `break`: terminate the execution of the loop.

The excerpt of code below illustrates the use of iterators.

```
processdef Publisher(
  in uint32: sensor,
  in queue<uint32>: subscribers,
  out channel<Frame>: port)
{
  state Emitting {
    var Frame: frame;
    frame.kind = PUBLISH;
    frame.data = sensor;
    loop over subscribers with it {
      frame.destinator = it.current;
      port.push(frame);
    }
  }
}
```

Note
The use of the `loop over` construct is not imposed to the programmer, it is also possible to iterate over a collection `a` with a loop counter ranging from `0` to `a.max_size-1`.

2.7 Processes: recurrent tasks embedding Finite-State Machines

Processes can be seen as functions that are activated periodically, or when certain activation conditions are met. CPAL offers language constructs that makes it possible to implement the logic of a process as a Finite-State Machine (FSM) in a non-ambiguous and intuitive manner. For instance, each state of the FSM can correspond to a distinct *running mode* of the application. The features available in CPAL rely are based on mode-automata[6].

2.7.1 Process definition

A process, which can simply be called a task too, is a recurrent activity whose functioning logic is described under the form of an FSM, possibly reduced to a single state repeatedly executed. Several processes can be executed in parallel - this is called *parallel composition* of FSMs - in which case their actual execution order depends on the scheduling policy (see Section 2.10). The first step is to define the process, that is, its list of input and ouput parameters, and the code itself. Then, one or several instances of the process can be created. These instances will be automatically executed at run-time by the interpreter at the right points in time. The program below shows the definition of a process and the creation of an instance of this process that is executed every 100ms.

Note
Although the language does not forbid it explicitly, it is a good practice to not use in a process a global variable passed as argument. This facilitates modularity and re-use and enables to trace and visualize the data-flows between processes within the CPAL development environment.

```
/*
Licence Creative Commons CC0 - "No Rights Reserved"
@keywords transition, timed; transition, conditional; transition, timed and ↩
   conditional
*/

processdef My_Proc(
  in uint32: arg1,
  in bool: reset,
  out bool: arg3)
{
  state Init {
    arg3 = false;
  }
  on (true)
    to Main;

  state Main {
  }
  after(5ms) if (reset)
    to Init;
```

[6] http://www-verimag.imag.fr/~Florence.Maraninchi/spip.php?rubrique22&lang=en

```
  after(1s)
    to Idle;
  on (arg1 > 5)
    {
      arg3 = true;
    }
    to Main;

  state Idle {
    arg3 = false;
  }
  on (reset)
    to Init;
}

var uint32: a = 5;
var bool:   b = false;
var bool:   c;

process My_Proc: p1[100ms](a, b, c);
```

Run in Playground[7]

Note
The first time a process is executed, the code of the first state that is declared in the code is executed and the process then yields the CPU. The subsequent executions of the process start with the execution of the first transition that can be fired, if any, then moves on to the current state. Then, the CPU is yielded except if otherwise stated (see Section 2.8.4).

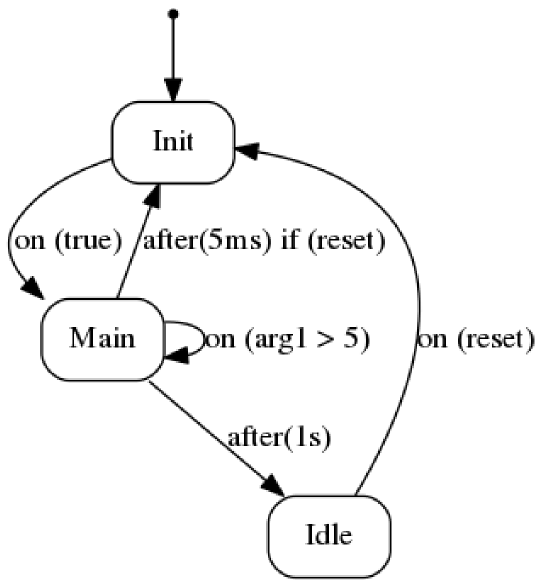
2.7.2 The Finite-State Machine implementing the logic of the process

Unlike in C or any other general purpose language, specific languages constructs are available in CPAL to ease the implementation of Finite-State Machines (FSMs). Each process embeds an FSM to describe its logic. The process initial state is the first state defined. Transitions (to states of the FSM) are evaluated in the order of their declaration. There are 3 possibilities for a transition to be triggered:

- On a Boolean condition being true:
  - `state State1 {...} on (cond) /*{ optional code }*/ to State2;`
- After a certain time spent in the active state:
  - `state State1 {...} after (2s) /*{ optional code }*/ to State2;`
  - `state State1 {...} after (500ms) /*{ optional code }*/ to State2;`
  - `state State1 {...} after (2*3s) /*{ optional code }*/ to State2;`
- Both a condition on the time spent in the current active state, and a Boolean condition:
  - `state State1 {...} after (500ms) if (cond) /*{ optional code }*/ to State2;`

[7] http://www.designcps.com/cpal-playground?path=talks/cpal-intro/sm-sample.cpal

As illustrated above, like states, transitions can optionally embed code that is executed each time the transition is taken.



Note
The time that allows an `after` transition to be triggered can be dynamically changed at run-time through a `time64` variable as illustrated in the code snippet below.

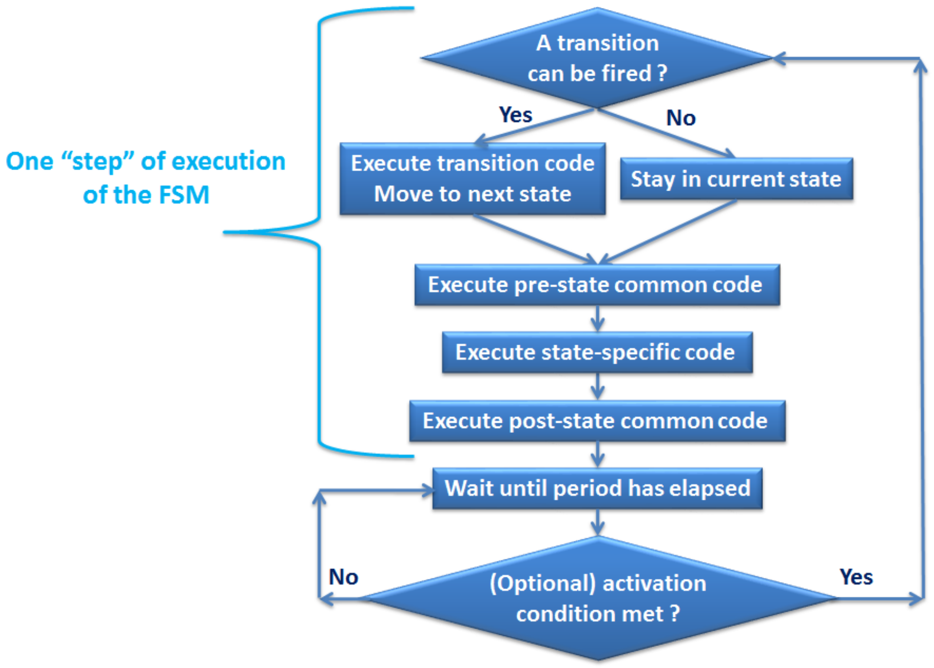
```
processdef A_Process()
{
    static var time64: trigger_t = 50ms;

    state Main {
      if (a_condition) {
        trigger_t = 50ms;
      } else {
        trigger_t = 150ms
      }
      /* body of the state */
    }
    after (trigger_t)
      to Another_State;
    /* ... */
}
```

2.8 Process execution

2.8.1 Elementary execution step

A process has a memory in the sense that, each time it starts to execute, it resumes in the state in which it was at the end of the previous execution. The code of the state is however not executed at this point. It is first checked whether a transition can be triggered (stage 1). If yes, the active state of the FSM is changed according to the transition (stage

2). The transition that is taken is always the first in the code that can be triggered. Then, optionally, a block of code common to all states of the process is executed (*pre-state common code* at stage 3, see Section 2.8.3), then comes the execution of the code of the current state (stage 4) and, finally, an optional block of code common to all states in executed (i.e., *post-state common code* at stage 5). The process then relinquishes the CPU to other processes that may be waiting to be executed. These 5 stages executed in sequence, as illustrated on the figure below, are called **an elementary execution step** of the FSM.


One "step" of execution of the FSM
A transition can be fired ?
Yes
No
Execute transition code Move to next state
Stay in current state
Execute pre-state common code
Execute state-specific code
Execute post-state common code
Wait until period has elapsed
No
(Optional) activation condition met ?
Yes

Note
CPAL implements Mealy semantics for FSM: first a transition is executed if any can be, then the current state of the FSM is executed. This allows for faster response to events.

2.8.2 Execution order: an example

Let us consider the following program:

```
/*
Licence Creative Commons CC0 - "No Rights Reserved"
@keywords execution order; state, common; finally; code in transition
*/

processdef A_Process()
{
  static var bool: toggle = true;
  static var uint64: step = 0;

  common{
    IO.println("pre-state common - step: %u", step);
  }

  state A{
    IO.println("state A - step: %u", step);
  }
  on (toggle) {
    IO.println("transition A->B - step: %u", step);
    toggle=false;
  } to B;

  state B{
    IO.println("state B - step: %u", step);
  }
  on (not toggle) {
    IO.println("transition B->A - step: %u", step);
    toggle=true;
  } to A;

  finally{
    IO.println("post-state common - step: %u", step);
    step = step + 1;
  }

}

process A_Process: p1[100ms]();
```

Run in Playground[8]

The output of the program above for the first 3 execution steps will produce in verbose mode (see Section 3.5) the following execution trace. The execution follows the 5 identified stages. It is should be noted that, at the very first execution of the process, it is assumed that a dummy transition doing nothing leads to the process initial state.

```
[0.000000000000:STATE] process "A_Process", instance "p1", state "A"
[0.000000000000:PRINTLN] pre-state common - step: 0
[0.000000000000:PRINTLN] state A - step: 0
[0.000000000000:PRINTLN] post-state common - step: 0
[0.100000000000:STATE] process "A_Process", instance "p1", transition 1
[0.100000000000:PRINTLN] transition A->B - step: 1
[0.100000000000:STATE] process "A_Process", instance "p1", state "B"
[0.100000000000:PRINTLN] pre-state common - step: 1
[0.100000000000:PRINTLN] state B - step: 1
[0.100000000000:PRINTLN] post-state common - step: 1
[0.200000000000:STATE] process "A_Process", instance "p1", transition 1
[0.200000000000:PRINTLN] transition B->A - step: 2
[0.200000000000:STATE] process "A_Process", instance "p1", state "A"
[0.200000000000:PRINTLN] pre-state common - step: 2
[0.200000000000:PRINTLN] state A - step: 2
[0.200000000000:PRINTLN] post-state common - step: 2
```

Important
The conditions of the transitions are evaluated in the order of declaration of the transitions and the first valid transition is taken.

[8] http://www.designcps.com/cpal-playground?path=talks/tutorial/samples/tut-execution-order.cpal

2.8.3 Sharing code between states

In order to avoid code duplication, the different states of a process can share *common code*. Precisely a block of common code can be executed before the code of the state (*pre-state common code* starting with keyword `common`) and another after the code of the state (*post-state common code* starting with keyword `finally`).

This pre-state, resp. post-state, common code has to be declared before, resp. after, the declaration of the states as illustrated in the example program below. At run-time, the pre-state common code will be executed after a transition, if one can be taken, but before the code of the active state. Then follows the execution of the post-state common code. At the startup of the program (i.e., first invocation of the process), since no transition is taken, the sequence of execution is first pre-state common code then the code of the initial state.

```
processdef A_Proc()
{

  /* pre-state common code executed first whatever the current state */
  common {
    /* do something such as evaluating pre-conditions */
  }

  state First {
    /* state-specific code goes here */
  }
  after (1000ms)
    to Second;

  state Second {
    /* state-specific code goes here */
  }
  after (1000ms)
    to First;

  /* post-state common code executed last whatever the current state */
  finally {
    /* do something such as evaluating post-conditions */
  }
}

process A_Proc: p1[50ms]();
```

Note
Pre-state common code, resp. post-state common code, are most often the right place to test that pre- and post-conditions are verified typically using the `assert()` function.

2.8.4 Executing several states in a row

By default, a process will execute a transition, if one can be executed, and a single state. Sometimes it is however necessary, or just convenient, to execute several states in a row before yielding back the CPU to the other processes. This can be done using the `self.continue` attribute:

```
/*
Licence Creative Commons CC0 - "No Rights Reserved"
@keywords self.continue; time64.time()
```

```
*/

processdef Simple()
{
    state First {
        IO.println("Time %t  - executing state First", time64.time());
        self.continue = true;
    }
    on (true)
      to Second;

    state Second {
        IO.println("Time %t - Executing state Second", time64.time());
    }
    on (true)
      to Third;

    state Third {
        IO.println("Time %t - Executing state Third", time64.time());
    }
    on (true)
      to First;
}

process Simple: p1[1s]();
```

Run in Playground[9]

In the program above, state `First` and `Second` will always be executed in a row since `self.continue` is set to true in `First` while state `Third` will be executed alone. Upon the activation of process `self.continue` is by default set to `false` but it can be changed to true anywhere in the code, that is in the `common` and `finally` blocks, the code of a state or the code of a transition.

Note
If several states are executed one after the other by setting `self.continue` to `true` the pre-state and post-state common code will be executed for each of the state.

2.9 Process creation

Once a new type of process has been defined through a `processdef` block code, it is possible to create, what is also called to "instantiate" or to "declare", one or several processes of this type. It is similar to creating an object of a certain class in C++ or Java.

One distinguishes processes with a periodic activation pattern, or more generally processes with a time-triggered activation, and the processes whose executions are triggered by events, called event-triggered processes. The activation pattern is specified at the creation of the process. By default, a process will remain active during the entire execution of the application but it is possible to stop its execution using an activation condition.

[9] http://www.designcps.com/cpal-playground?path=talks/cpal-intro/self_continue.cpal

2.9.1 Time-triggered activations

The code statement to create a time-triggered process starts with the keyword `process` and obeys the syntax: `process type:name[period,offset][option. exec. condition] (process args);`
The first parameter in brackets after the process name is its period, that is the time between two successive *releases* of that process. Here release time means the time at which a process becomes ready to execute, and not the time at which it first gets the CPU since there might be higher priority processes pending execution or being executed. Several processes of the same type can have a different *period* as specified at their instantiation. Except if otherwise stated with an offset (see below for an example), the first release of each process takes place at time 0, that is at the startup of the program. Then, the successive release times will be separated by exactly a period.

The code snippet below illustrates the three ways to create a process:

- `task1` is a **periodic process** activated every 500ms,
- `task2` is a **periodic process with offset**, here 100ms, which means that the first release of the process will take place 100ms after the startup of the program, then the next at 600ms, etc,
- `task3` is a **periodic process with activation condition**, this means that a periodic instance will only be executed if the activation condition is true, otherwise the instance is just skipped. Optionally, an offset can be specified too. This execution pattern is often referred to as "guarded execution" in the litterature.

Offsets are typically useful to spread the CPU load over time and to enforce an order of execution in the case where, for instance, a process produces data for another processes.

The activation condition execution pattern is useful to implement different functioning modes and to execute activities only if specific conditions are met (e.g., occurrence of an event such as an alarm).

```
/*
Licence Creative Commons CC0 - "No Rights Reserved"
@keywords activation, time-triggered; periodic process; guarded execution; offset
*/
processdef My_Process()
{
  state Main {
  }
}

var bool: a_trigger_condition = true;

/* Periodic process */
process My_Process: task1[100ms]();
/* Periodic process with initial offset */
process My_Process: task2[200ms, 100ms]();
/* Periodic with boolean condition, offset is optional */
process My_Process: task3[600ms][a_trigger_condition]();
```

Run in Playground[10]

Warning
A process will not obtain the CPU immediately upon its release if there are processes of "higher priority" that can be executed. How the relative priority of the processes is determined depends on the scheduling policy and the associated scheduling parameters (see Section 2.10).

[10] http://www.designcps.com/cpal-playground?path=talks/cpal-intro/proc-inst.cpal

Through the use of conditional execution it is easy to implement state-dependent executions. This is illustrated in this example program[11] where either of two functions is executed depending on the current state of the application.

Note
It is possible to depart from strictly periodic process release times and implement varying inter-arrival times between successive releases of a process. This is achieved using timing annotations as explained in Section 4.2.1.

2.9.2 Event-triggered activations

It is very common in embedded systems that activities must be executed in reaction to the occurrence of events. CPAL provides the event-triggered activation pattern to execute a process when a boolean condition evaluates to true. The condition can reflect the occurrence of the event, for instance that an I/O lies in a given range of values.

In a very similar manner as for time-triggered processes, the code statement to create an event-triggered process starts with the keyword `process` and obeys the syntax:
`process process_type:process_name[][triggering condition](process args);`
The event-triggered process will be executed as long as the triggering condition will evaluate to true (e.g., queue of incoming messages not empty), which means possibly several times in a row. In the example below, a process must be executed one time when the global variable `Event` is true, and thus it resets the value of `Event` to `false` during its execution.

```
/*
Licence Creative Commons CC0 - "No Rights Reserved"
@keywords activation, triggered
*/

var bool: Event = false;

processdef Event_Source(out bool: event) {
  state Main{
    /* e.g., variable or an I/O exceeding a certain threshold,
    a frame just having being received, etc */
    event = bool.rand_uniform();
  }
}

processdef Triggered_On_Event(out bool: event) {
  common {
    event = false;
  }
  state Main {
    IO.println("Triggered process execution");
  }
}

process Event_Source: source[5Hz](Event);
process Triggered_On_Event: trigger[][Event](Event);
```

Run in Playground[12]

[11] http://www.designcps.com/cpal-playground?path=talks/cpal-intro/control-flow.cpal
[12] http://www.designcps.com/cpal-playground?path=talks/cpal-intro/triggered-activation.cpal

An event-triggered process is scheduled for execution as soon as its triggering condition becomes true. It might however not get executed immediately because of higher-priority processes. The triggering condition will be re-tested before the execution of the process, and if the condition at this time evaluates to `false` the process will not be executed.

Note

The triggering condition of an event-triggered process, as well as the optional activation condition for a periodic process, can be defined with a function whose last argument is a `boolean` defined as `out` parameter, see the examples below.

Implementing complex process activation patterns (e.g., bursty) can be done by accessing the internal queue of the scheduled activations of a process named `next_activations`. Precisely, this is done by "pushing" new activation dates into the queue , which is automatically sorted by decreasing process activation times. The activation times are internally stored in relative time with respect to the current time, which implies than some activation due dates may become negative in some cases thus the use of `duration64` type instead of `time64`. In the example below, the helper function `require_activation(p)` returns `true` when an activation of `p` has been scheduled for execution earlier or at the current point in time. In the example below, two instances of the process are released 5ms apart every 50ms. It should be noted that at least the first activation must be scheduled in an other process or in the `init()` function as here.

```
/*
Licence Creative Commons CC0 - "No Rights Reserved"
@keywords activation, triggered; next_activations; init() function
*/

const time64: period = 50ms;

processdef My_Process()
{
  state Main {
    IO.println("Current activation released at %t", self.current_activation);
    if (self.current_activation mod period == 0ms) {
      IO.println("Creating two new activations");
      /* Schedule another activation 5ms after the period, the process is
         thus released at times 0ms, 50ms, 55ms, 100ms, 105ms, ...
      */
      self.next_activations.push(duration64.as(period));
      self.next_activations.push(duration64.as(period+5ms));
    }
    loop over self.next_activations with it {
        IO.println("An activation scheduled in %t", time64.as(it.current));
    }
  }
}

process My_Process: task1[][require_activation(task1)]();

init() {
  /* Create first activations in init() to bootstrap the process */
  task1.next_activations.push(0s);
  task1.next_activations.push(5ms);
}
```

Run in Playground[13]

In addition to `require_activation()`, the CPAL standard library (located under `pkg` folder) provides in source code other helper functions to trigger the execution of processes at specific points in time expressed in relative or absolute time: `wakeup_at()`, `wakeup_also_at()`, `wakeup_in()` and `wakeup_also_in()`. The example program below illustrates the use of these functions:

```
/*
Licence Creative Commons CC0 - "No Rights Reserved"
@keywords activation, triggered; next_activations; init() function; wakeup_at()
*/

const time64: period = 50ms;

processdef My_Process()
{
  state Main {
    IO.println("Current activation released at %t", self.current_activation);
    loop over self.next_activations with it {
        IO.println("An activation scheduled in %t", time64.as(it.current));
    }
  }
}

process My_Process: task1[][require_activation(task1)]();

init() {

  /* Schedule a process activation at time 150ms*/
  wakeup_at(task1, 150ms);
  /* This call is without effect since an activation is already planed at 150 ms*/
  wakeup_at(task1, 150ms);
  /* Schedule another activation, at time 300ms */
  wakeup_at(task1, 300ms);
  /* A second activation is released at time 300ms */
  wakeup_also_at(task1, 300ms);

  /* Scheduling a process activation in relative time */
  wakeup_in(task1, 400ms);
  /* This call is ignored as an activation is already planed at t=500ms */
  wakeup_in(task1, 400ms);
  /* This call will add another instance at t=500ms */
  wakeup_also_in(task1, 400ms);

}
```

Run in Playground[14]

[13] http://www.designcps.com/cpal-playground?path=talks/cpal-intro/require-activation.cpal
[14] http://www.designcps.com/cpal-playground?path=talks/cpal-intro/wakeup-at.cpal

Warning

The workload induced by event-triggered activations must be carefully assessed so as to not overload the CPU. This requires that it is possible to bound the number of events in any time interval, or that a control mechanism is implemented within the application. Event-triggered processes should thus be used with precaution for the programming of real systems. For simulation models however, this is usually not an issue, and the event-triggered activation pattern offers a simple and natural way to express chain of events across a system. The alternative to triggered executions would be periodic executions (i.e., "polling"), with the drawback of unecessary executions and an accuracy that depends on the frequencies of the processes.

2.9.3 Sub-processes

It is possible within a process to trigger the execution of one or several other processes, called *sub-processes* or *embedded processes*. By doing so, one can define hierarchical FSMs which helps to reduce the complexity of the individual FSMs and facilite the re-use of code. Sub-processes maintain their internal state, in particular the current state, from one activation to another. Sub-processes must be declared as local variables of the calling process without specifying an activation pattern and activation condition. They inherit the scheduling parameters of the calling process. The use of sub-processes is illustrated in the program below:

```
/*
Licence Creative Commons CC0 - "No Rights Reserved"
@keywords subprocess
*/

processdef Sub(in uint8: a) {
  common {
    IO.println("Sub - common");
  }
  state First {
    IO.println("Sub - First - a: %u", a);
  }
  on (true) to Second;
  state Second {
    IO.println("Sub - Second");
  }
  on (true) to First;
  finally {
    IO.println("Sub - finally");
  }
}

processdef Master() {
  process Sub: p1; /* create a sub-process of type `Sub` */
  common{
    IO.println("Master - common");
  }
  state Main {
    p1(uint8.rand_uniform(uint8.FIRST,uint8.LAST)); /* triggers execution of p1 */
  }
  finally {
    IO.println("Master - finally");
  }
}
```

```
process Master: p0[100ms]();
```

Run in Playground[15]

2.10 Process scheduling

The scheduling policy determines the order of execution the active processes. If processes have real-time constraints such as deadlines, these constraints can be verified by monitoring at run-time (see Section 2.11) or by schedulability analysis (see this technical paper[16]). The scheduling policy to be used is specified to the execution engine through a timing annotation in the code (see Section 2.10.2).

2.10.1 FIFO scheduling

The default scheduling policy in CPAL is FIFO (First-In First-Out): the order of execution is the order in which the processes have been released. This means that when a process becomes active, it will have to wait that all processes which have been released before have terminated. If several processes are released at the same time, the order of execution of the processes is their order of instantiation in the code. For example, in the program of Section 2.9.1, `task1` will be executed before `task3` when they happen to be activated at the same time. A more general way to break ties under FIFO, not relying on the order of instantiation, is to set the `priority` attribute as done for the non-preemptive Fixed Priority policy (see Section 2.10.2).

Note
The default CPAL scheduling policy is FIFO: processes are executed in the order of their activation. The FIFO policy ensures that whatever the execution time of the code, and thus whatever the execution platform, the execution order of the processes will remain identical. This property is called *event-order determinism*. FIFO is however largely outperformed by non-preemptive Earliest Deadline First and non-preemptive Fixed Priority (NPFP) scheduling in terms of its ability to meet deadline constraints. Scheduling under FIFO with the CPAL task model is studied in this technical paper[a].

[a] http://orbilu.uni.lu/bitstream/10993/24935/1/FIFO_scheduling_TR.pdf

2.10.2 EDF and Fixed-Priority scheduling policies

Two other scheduling policies are available in CPAL: non-preemptive Earliest Deadline First (NPEDF) and non-preemptive Fixed Priority (NPFP) scheduling.

In NPEDF, when the CPU becomes idle and several processes are pending execution, the process with the shortest deadline is chosen to be executed next. By default, the relative deadline of a process (i.e., how much time the process has to complete at the time of its release) is set to the process period. It is possible to set NPEDF as the scheduling policy and set the deadlines values through a timing annotation, executed once at the startup, as illustrated below.

```
/*
Licence Creative Commons CC0 - "No Rights Reserved"
@keywords timing annotation, scheduling; NPEDF; cpal_system
*/
```

[15] http://www.designcps.com/cpal-playground?path=talks/cpal-intro/subprocess.cpal
[16] http://orbilu.uni.lu/bitstream/10993/24935/1/FIFO_scheduling_TR.pdf

```
processdef Simple()
{
  state Main {
    IO.println("relative deadline: %t", self.deadline);
  }
}

process Simple: p1[10ms]();
process Simple: p2[15ms]();

@cpal:time
{
  cpal_system.sched_policy = Scheduling_Policy.NPEDF;
  p1.deadline = 8ms; /* relative deadline set to 0ms would be the most stringent  ↩
      one */
  p2.deadline = 12ms;
}
```

In NPFP it is the priorities of the processes which decide the execution order, `0` being the lowest possible priority for a process. If the priority of a process is not specified, its default priority is `0`. Processes of same priority are executed in the FIFO order. The NPFP and the process priorities can be set by a dedicated annotation as shown below.

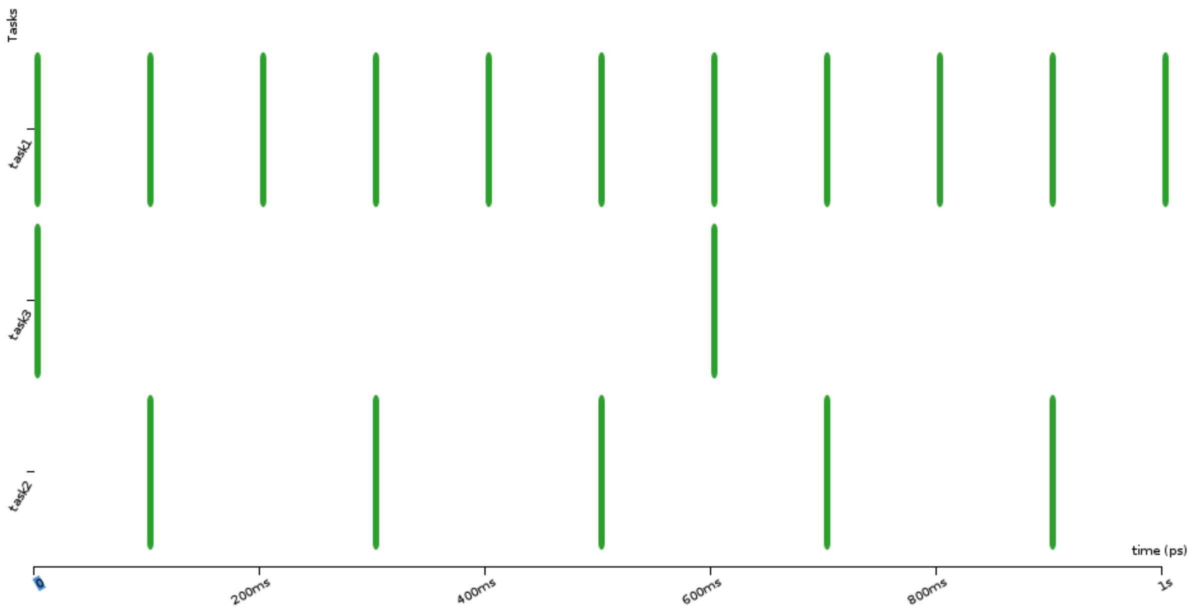
```
/*
Licence Creative Commons CC0 - "No Rights Reserved"
@keywords timing annotation, scheduling; NPFP; cpal_system
*/

processdef Simple()
{
  state Main {
    IO.println("process priority: %u", self.priority);
  }
}

process Simple: p1[10ms]();
process Simple: p2[15ms]();

@cpal:time
{
  cpal_system.sched_policy = Scheduling_Policy.NPFP;
  p1.priority = 1;
  p2.priority = 0; /* lowest priority process */
}
```

Note

`system` is a structure whose field `sched_policy` defines the scheduling policy among the processes. The set of active processes can be accessed through `system` as well, providing additional means of introspection as shown in Section 2.11.4.

2.10.3 Gantt diagram of the process executions

The Gantt diagram below shows the successive process activations of the example program in Section 2.9.1, as can be seen in the CPAL-Editor or in the CPAL Playground. The width of the bars represents the execution times of the processes. It should be noted that the program was here executed **in simulation mode** with the assumption that the code executes in no time. An execution in simulation mode with timing annotations for execution times (see Section 4.2.3), or an execution **in real-time mode** would led to different results.



Run in Playground[17]

The Gantt chart is obtained by monitoring the execution of the program for a certain duration, each time the source code is saved. If the default value for the monitoring duration is not suitable, it can be set by the user through an annotation in the source code: `/*cpal:tasks time=100ms*/`.

2.11 Process introspection

2.11.1 Process querying its own characteristics

At run-time it is possible for a process instance to query its pid, period, offset, period (see Section 4.2.1), jitter (see Section 4.2.2), priority, deadline as well as the activation time of the current and previous instance. This is illustrated in the example program below.

```
/*
Licence Creative Commons CC0 - "No Rights Reserved"
@keywords introspection; activation jitter; self, pid; self, period; self, offset
*/
processdef A_Process()
{
  state Main {
    IO.println("pid %u", self.pid);
    IO.println("period %t",self.period);
    IO.println("offset %t",self.offset);
    /* Time at which the current instance of the process obtained the CPU */
    IO.println("curr %t",self.current_activation);
    /* Time at which the previous instance of the process obtained the CPU */
    IO.println("last %t",self.previous_activation);
```

[17] http://www.designcps.com/cpal-playground?path=talks/cpal-intro/proc-inst.cpal

```
    /* Would not hold for the first instance released at time self.offset */
    if (self.current_activation > self.offset) {
      /* Warning: this only holds in simulation mode */
      assert((self.current_activation-self.previous_activation) == self.period);
    }
  }
}

process A_Process: p1[100ms]();
```

Run in Playground[18]

Note

Process introspection enables processes to query at run-time their execution characteristics such as their activation rate and activation jitters. This is helpful for instance to implement control algorithms that adapt to their frequency of execution or their execution jitters, and more generally to make decision at run-time. This can be used as well to identify abnormal timing behaviours during the execution.

Excessive start-of-execution jitters can be detected with the following code pattern where the value of factor defines the permissible threshold:

```
if (self.current_activation - self.previous_activation > self.period * factor){
/* ... */
}
```

2.11.2 Knowing the current scheduling policy

The current scheduling policy, and the parameters of a process for this policy, can also queried at run-time as illustrated below:

```
if (system.sched_policy==Scheduling_Policy.NPFP) {
  IO.println("Process priority under NPFP: %u", self.priority);
  } else {
    if (system.sched_policy==Scheduling_Policy.NPEDF) {
      IO.println("Process relative deadline under NPEDF: %t", self.deadline);
    } else {
        IO.println("System scheduled under FIFO policy");
      }
    }
```

2.11.3 Knowing the current state of a process

The native type `Process_State` makes it possible to query the current state of the FSM embebdded in any process and save/test it. This can be done internally by the process itself using `self.process_state`, for instance to record the previous state in static variable of type `Process_type`, or detect a particular execution path. It can also be done from the outside, by an "observing" process as illustrated below:

[18] http://www.designcps.com/cpal-playground?path=talks/tutorial/samples/tut-introspection.cpal

```
/*
Licence Creative Commons CC0 - "No Rights Reserved"
@keywords Process_State; observing automata; finally
*/

processdef Simple()
{
    static var bool: change_state = false;
    common {
        change_state = bool.rand_uniform();
    }

    state First {
    }
    on (change_state)
      to Second;

    state Second {
    }
    on (change_state)
      to First;
}

process Simple: p1[1s]();

processdef Observer()
{
    static var Process_State: previous;
    state Single {
        IO.print("Current state of p1: ");
        if (p1.process_state == Process_State.First) {
            IO.println("First");
        } else {
              IO.println("Second");
            }
    }
}

process Observer: p2[1s]();
```

Run in Playground[19]

Note
The use of observing automata is a powerful technique to check correctness properties on the execution of a program.

2.11.4 Iterating over the list of processes

The `system` global variable exposes to the programmer the list of running processes and their characteristics. As the program below shows, `system` can be used to compute at run-time the CPU load, for instance to implement adaptive behaviors (e.g., execute additional functions) or detect overload situations.

[19]http://www.designcps.com/cpal-playground?path=talks/cpal-intro/observer.cpal

```
/*
Licence Creative Commons CC0 - "No Rights Reserved"
@keywords cpal_system; introspection; process characteristics; execution time; pid
*/

processdef A_Proc()
{
    state Main {
    }
}

process A_Proc: p1[100ms]();
process A_Proc: p2[200ms]();

@cpal:time
{
    p1.execution_time = 50ms;
    p2.execution_time = 50ms;
}

init()
{
    var float64: load = 0.0;
    loop over cpal_system.processes with it {
        IO.println("id: %u period %t\n", it.current.pid, it.current.period);
        load = load + float64.as(it.current.execution_time) / float64.as(it. ↩
           current.period);
    }
    IO.println("cpal_system load %f\n", load);
}
```

Run in Playground[20]

The characteristics of a process that can be queried and possibly modified at run-time are members of the structure `Process_Instance` shown below:

```
struct Process_Instance {
/* User-defined execution time enforced in simulation mode */
    time64: execution_time; /* Read - Write through global level annotations */
/* Worst-Case Execution Time recorded at run-time on embedded boards */
    time64: wcet; /* Read Only */
/* Best-Case Execution Time recorded at run-time on embedded boards */
    time64: bcet; /* Read Only */
/* Time between two subsequent releases of the process instances */
    time64: period; /* Read - Write through annotations */
/* Time at which the first instance is released */
    time64: offset; /* Read - Write through annotations */
/* Priority under FPNP policy */
    uint8: priority; /* Read - Write through annotations */
/* Release jitter of the process instance */
    time64: jitter;  /* Read - Write through annotations */
/* Deadline of the process instance relative to its release time*/
    time64: deadline; /* Read - Write through annotations */
/* Start of execution of the previous instance */
    time64: previous_activation; /* Read Only */
```

[20] http://www.designcps.com/cpal-playground?path=talks/cpal-intro/cpuload.cpal

```
/* Start of execution of the current instance */
    time64: current_activation; /* Read Only */
/* Identifier of the process */
    uint32: pid; /* Read Only */
/* Current state of the FSM embedded in the process */
    Process_State: process_state; /* Read Only */
/* Sorted queue with the release times of the upcoming instances of the process */
    queue<int64>: next_activations[5]; /* Read - Write through push operations */
};
```

2.12 Inter-process communication

Passing data to a process, and receiving back data from a process, is done by through process arguments (resp. declared as `in` and `out`). One can distinguish two types of arguments:

- *global variables* others than communication channels, be they of a basic type or a user-defined type, and pass them as arguments of the processes. In that case, the variables are passed by value to the processes, which means that the processes will work on copies of the variables,
- *communication channel*, that is either a `stack` or a `queue`. In that case, the channel is not copied but a reference to it is provided to the process. This is transparent for the programmer and has not impact on the process code. Passing data by reference is of course more efficient in terms of speed and memory in most cases. Another interest of communication channels is that they allow data buffering when the production rate does not match the consumption rate.

2.12.1 Communication by global variables

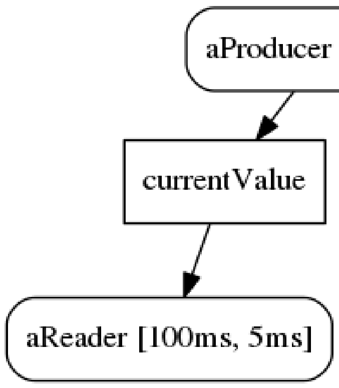
The code snippet below illustrates how a global variable can be used to exchange information between a producer process and a consumer process. As often done, an activation's offset ensures that the consumer process becomes executed after the producer process. It should be noted that this is not required in this CPAL program since the order of declaration would have been sufficient to enforce the right order of execution at run-time.

```
processdef Uint32_Producer(out uint32: n)
{
/* ... */
}

processdef Uint32_Reader(in uint32: n)
{
/* ... */
}

var uint32: current_value;

process Uint32_Producer: a_producer[100ms](current_value);
process Uint32_Reader:   a_reader[100ms, 5ms](current_value);
```

Run example[21]

The figure below shows the functional architecture of the application which is the set of processes, how they are activated, and the data flow among the processes. This view is available in the CPAL-Editor when editing a program.

[21] `http://www.designcps.com/cpal-playground?path=talks/cpal-intro/read-uint.cpal`



2.12.2 Communication by channels

A communication channel allows to pass indifferently a `stack` or a `queue` to a process in an efficient manner (i.e., without duplicating the complete data-structure in memory). The code of a process can be independent of the actual collection that is passed on to it, which is interesting in terms of re-use. The programmer can choose whatever collection has the right semantics given the context: a `queue` if the older data must be processed first (First-In First-out policy - FIFO) or a `stack` if it is the newer data that must be processed first (Last-In First-Out policy - LIFO).

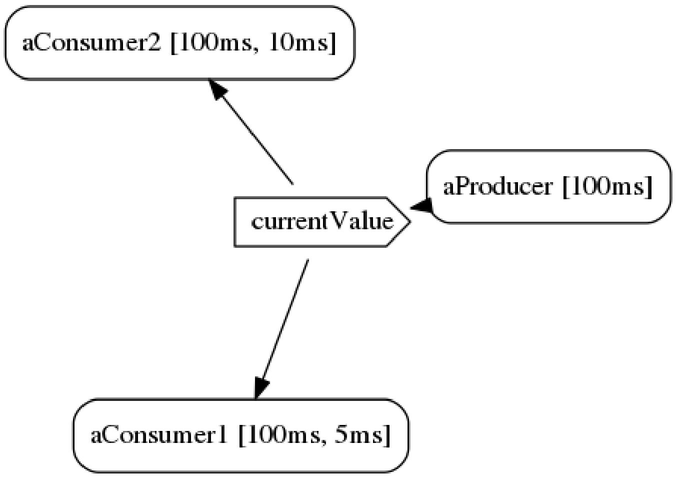
The snippet of code below shows the skeleton of a program where one process produces data that are provided to 2 consumer processes through a communication channel, here a queue.

```
processdef Uint32_Producer(out channel<uint32>: n)
{
/* ... */
}

processdef Uint32_Reader(in channel<uint32>: n)
{
/* ... */
}

var queue<uint32>: current_value;

process Uint32_Producer: a_producer[100ms](current_value);
process Uint32_Consumer: a_consumer1[100ms, 5ms](current_value);
process Uint32_Consumer: a_consumer2[100ms, 10ms](current_value);
```

Run in Playground[22]

The figure below shows the functional architecture of the application for the example of inter-process communication through channels.

[22] http://www.designcps.com/cpal-playground?path=talks/cpal-intro/consume-uint.cpal



2.13 Manipulation time and strings, built-in functions

2.13.1 Working with time quantities

Performing actions at certain time points and being able to express time quantities is often needed in embedded programming, this is why CPAL provides a dedicated data type for handling physical time. The `time64` type is meant to measure and manipulate time quantities. The available time units are `s`, `ms`, `us`, `ns`, `ps` and `Hz` (Hertz). The print format specifier for `time64` variables is `%t`. CPAL provides addition, subtraction and modulo operators between two `time64` values. It also provides multiplication and division between `time64` and `uint64`. The use of these binary operators is illustrated in the program below.

```
/*
Licence Creative Commons CC0 - "No Rights Reserved"
@keywords time64, type; time units
*/

init()
{
  /* Internal granularity of time type is the picosecond (ps) */
  var time64: a_duration = 5s + 150ms + 3ns + 1ps;
  var time64: same_duration = 5s150ms3ns1ps;
  var time64: another_duration = 2 * a_duration - 1ps;
  var time64: a = 15ms;
  var time64: b = 6ms;
  var time64: c = 0.1Hz;
  var uint64: number_of_b_in_a = a / b;
  var uint64: time_integration_in_ms_square = b * uint64.as(a) / 1ms;
  /* 5s150ms3ns1ps */
  IO.println("%t", a_duration);
  assert(a_duration == same_duration);
  /* 30ms 7ms 500us 3ms */
  IO.println("%t %t %t", a * 2, a / 2, a mod b);
  assert(1s == 1000ms);
  assert(1ms == 1000us);
  assert(1us == 1000ns);
  assert(1ns == 1000ps);
}
```

Run in Playground[23]

Warning
Time quantities are integral values, e.g. writing `1.5s` is not allowed but it can be expressed as `1s500ms`, or alternatively `1500ms` or `1s + 500ms`.

It is usually better to use a `time64` variable when manipulating time, however sometimes it is convenient and leads to more readable code to do it with `uintX`. The code snippet below shows how to convert a `uintX` into a `time64` of any time units. For example, converting a `uintX a` to `ms` with the `(a)ms` syntax is equivalent to executing `time64.as(a) * 1ms`.

```
init() {
 var uint64: a = 1100;
 /* 18mn20s 1s100ms 1us100ns 1ms100us 1ns100ps */
 IO.println("%t %t %t %t %t", (a)s, (a)ms, (a)ns, (a)us, (a)ps );
}
```

The example program below shows the use of the `time64.time()` function to obtain the current time with the startup of the execution engine being the origin of time.

```
/*
Licence Creative Commons CC0 - "No Rights Reserved"
@keywords time64.time(); mode, simulation; mode, real-time
@brief the assert statement will only succeed in simulation mode where all ↩
   instructions execute in zero time (except otherwise stated with a timing ↩
   annotation). In real-time mode, the execution time of the code will make it ↩
   fail.
*/

const time64: set_period = 200ms;

processdef Proc_Def()
{
    static var uint64: i = 0;
    state A {
        var time64: current_time;
        current_time = time64.time();
        IO.println("%t", current_time);
        assert(current_time == set_period * i);
        i = i + 1;
    }
}

process Proc_Def: p[set_period]();
```

Run in Playground[24]

An important function related to time is `sleep(time64)` which suspends the execution of a process during the specified duration.

```
/*
Licence Creative Commons CC0 - "No Rights Reserved"
```

[23]http://www.designcps.com/cpal-playground?path=./talks/tutorial/samples/time-units.cpal
[24]http://www.designcps.com/cpal-playground?path=./talks/tutorial/samples/tut-time.cpal

```
@keywords time64.time(); time64, type; sleep()
*/

const time64: DELAY0 = 3ms;

processdef P_Def() {

  var time64: timer0 = time64.time();
  var time64: timer1;

  state A {
    sleep(DELAY0);
    timer1 = time64.time();
    assert(timer1 - timer0 >= DELAY0);
    IO.println("Current time is %t", time64.time());
  }
}

process P_Def: p[100ms]();
```

Run in Playground[25]

Hertz is a convenient and natural unit in many applications to express the rate of activation of processes. It can only be used for assignment in CPAL and defining the frequency of a process and not for doing arithmetic on time quantities like the other time units. The maximum possible frequency in CPAL is `1000000000000Hz` and the minimum is `0.0000001Hz`. It should be noted that since the clock is of limited accuracy, there can be a loss of accuracy when using Hertz, as illustrated below.

```
init() {
  var time64: f = 3Hz;
  assert(f == 333333333333ps);
}
```

2.13.2 Working with strings

Strings made up of ASCII characters can be created and manipulated for instance to output information on the console or read/write data from/to devices connected on serial ports like a modem or gsm module. Through arrays and collections, CPAL provides support for manipulating strings and passing them as function arguments:

```
/*
Licence Creative Commons CC0 - "No Rights Reserved"
@keywords string, size; string, as function argument; channel
*/

foo(in channel<uint8>: ch, out uint8: r)
{
  r = uint8.as(ch.count());
}

init()
{
  var queue<uint8>: a_string[5] = "abcd";
  assert(4 == foo(a_string));
```

[25] http://www.designcps.com/cpal-playground?path=talks/tutorial/samples/sleep.cpal

```
  assert(4 == foo("abcd"));
}
```

Run in Playground[26]

The `IO.print()` function, in a similar way as its C language `sprintf()` counterpart, makes it possible to write formatted data into strings as shown in this example:

```
/*
Licence Creative Commons CC0 - "No Rights Reserved"
@keywords string, format data into string; time64
*/

init()
{
  var uint8: sfloat[10];
  const float32: f1 = 1.0/3.0;
  var uint8: stime[50];
  const time64: t1 = 123456789101112ps; /* 0d0h2mn3s456ms789us101ns112ps */
  float_to_string: {
    uint8.print(sfloat,"%f", f1);
    IO.println(sfloat);
  }
  time_to_string: {
    uint8.print(stime,"%t", t1);
    IO.println(stime);
  }
}
```

Run in Playground[27]

2.13.3 Mathematical functions

A set of standard mathematical functions is built-in the language. These functions are listed below for the `float32` type but also exist for `float64`:

`float32.cos(float32 x)`	`float32.sin(float32 x)`	`float32.atan(float32 x)`
`float32.sqrt(float32 x)`	`float32.pow(float32 x, float32 y)`: x to the power of y	

The following functions return the absolute value of a number as an unsigned value:

`uint8.abs(int8 x)`	`uint16.abs(int16 x)`	`uint32.abs(int32 x)`
`uint64.abs(int64 x)`	`float32.abs(float32 x)`	`float64.abs(float64 x)`

2.13.4 Exiting a CPAL program

The `exit(uint8)` function terminates a CPAL program and returns an exit code to the OS. When CPAL is used on bare-hardware, that is without an OS, then the program is halted and can only be re-started after a reboot of the micro-controller. A typical idiom to exit a CPAL program is shown in the code snippet below.

[26] http://www.designcps.com/cpal-playground?path=talks/cpal-intro/string-as-argument.cpal
[27] http://www.designcps.com/cpal-playground?path=talks/cpal-intro/format-data-into-string.cpal

```
processdef A_Proc()
{
  state Main {
    /* do something */
  }
  after (5s) to Stop;
  /* after (a_duration) if (a_certain_condition) to Stop;
  /* on (a_certain_condition) to Stop; */
  state Stop {
    exit(0);
  }
}

process A_Proc: p1[100ms]();
```

Note
A call to the `exit()` function terminates the complete application, it is also possible to stop the execution of processes on an individual basis using the conditional activation mechanism, e.g.: `process MyProcess: task3[600ms][aConditionThatMustRemainTrue]();` (see Section 2.9.1).

2.14 CPAL naming convention

The names of native CPAL data types are all lowercase, e.g. `uint8`, `time64`, `queue<uint32>`, `channel<int16>`. Following a naming convention for source codes is a good practice of software engineering. Although nothing is imposed to the user for user-defined names, we propose the following naming convention for CPAL programs:

- Names of variables, arguments, functions, and tasks shall be **lower_case_with_underscores**.
- Names of user-defined process, structures, states, and enumerations shall be **Mixed_Case_With_Underscores**.
- Names of enumerations values, and constant shall be **UPPER_CASE_WITH_UNDERSCORE**.

The code snippet below shows the skeleton of a program respecting this naming convention.

```
enum My_Enum
{
  OPTION_A,
  OPTION_B
}
struct My_Structure
{
  uint8:   field_a;
  My_Enum: field_b;
}

var uint8: the_global_variable;

my_function(
  in bool: a_flag,
  out uint16: the_result)
{
  /* ... */
}

processdef The_Process_Def()
{
  state Main {
  /* ... */
  }
}

process The_Process_Def: the_task[100ms]();
```

The `cpal_lint` utility program, see Section 3.4.2, verifies the conformance of source files to this naming convention, while `cpal2x` formats source files according to the CPAL indentation standard, see Section 3.4.1.

Chapter 3

Development and run-time environment

There are several ways to develop and execute CPAL programs:

- The `CPAL-Editor` is a development environment that can be downloaded from https://www.designcps.com/-binaries/. The best programming experience in CPAL is probably obtained using the `CPAL-Editor` since it offers an all-in-one environment with graphical representation of the functional architecture and the automata describing the logics of the tasks, as well as syntax highlighting.
- The CPAL-Playground[1] is a web service that allows to write and execute CPAL code in a browser without any installation. It is not meant to develop complete applications but it is a nice tool to quickly discover CPAL and experiment with it, for instance on the basis of the sample programs available on-line[2]. There are however limitations to what is possible with the CPAL playground: the `include` directive is not allowed, and there is no visualisation of the functional architecture and the automata embedded in the code. Also, execution in real-time mode is not possible.
- The use of the command-line parser and execution engine, described in the rest of this chapter, offers another way to develop in CPAL. Knowledge of the possibilities and options offered by the command-line tools is recommended to have a good understanding and command of the CPAL execution environment.
- Generate C code from a CPAL program from within the `CPAL-Editor`, compile the C code with the generated makefile, and execute the resulting binary file. This feature, that should be considered in a prototypical stage, is available for Posix1003.1c compliant OS and for the Zephyr.

3.1 Simulation versus real-time mode

In this section we explain how to execute CPAL programs with the CPAL execution engine, that is not using code generation. This latter possibility is discussed in Section 3.7.

CPAL programs can be executed in two modes:

- **Simulation mode** which is suited for use in the design phase:
  - Execution is as fast as possible: activation frequency of the processes are not respected, `sleep(time64)` executes in zero-time. Typically, executing in simulation mode is several orders of magnitude faster than in the real-time mode.

[1] `http://www.designcps.com/cpal-playground/`
[2] `https://www.designcps.com/cpal-code-examples-index/`

  - Code executes in zero-time (except if stated otherwise through timing annotations, see Section 4.2.3).
  - CPAL interpreter is hosted by an OS.

- **Real-time mode** for the actual execution of the program:
  - Execution happens in real-time: activation frequency of the processes are respected, `sleep(time64)` actually suspends the execution.
  - Code execution (instructions, read/write I/Os) takes a certain time which depends on the platform.
  - CPAL can be executed on bare hardware or hosted by an OS (depending on the platform).

The *execution mode* depends on the binary version of the CPAL interpreter that is used to run a program (see Section 3.5) and whether the `--realtime` option is set in the command line.

3.2 Supported platforms

CPAL runs on a number of platforms that have different capabilities in terms of execution mode and ability to access hardware Input/Output ports (e.g., General Purpose Input/Output, I2C, serial communication, etc). The table below summarizes the capabilities of all supported platforms as well as the name of the corresponding executables. Basically, one distinguishes between binaries meant for simulation and binaries meant for actual execution that can access the I/O ports and run in real-time mode. The binaries for simulation are named `cpal_interpreter`, or `cpal_multi_interpreter` for the multi-interpreter version (see Section 4.3), whatever the platform.

Platforms	**Execution mode**	**Access to HW?**	**Executable**
Windows 32/64bit	Simulation	No	cpal_interpreter
Embedded Windows 32/64bit	Real-Time and Simulation	No	cpal_interpreter_winmbed
Linux 64bit	Simulation	No	cpal_interpreter
Embedded Linux 64bit	Real-Time and Simulation	Yes	cpal_interpreter_linuxmbed
Mac OS X	Simulation	No	cpal_interpreter
NXP FRDM-K64F	Real-Time	Yes	NA, no OS, image is uploaded
Raspberry Pi	Real-Time and Simulation	Yes	cpal_interpreter_raspberry

Warning
Raspberry Pi is a good target to experiment with CPAL but it is not suited for executing real-time applications due to large timing variabilities (e.g., jitters in task release times). The best supported platform with respect to timing predictability is the NXP FRDM-K64F[a] SOC on which the CPAL execution engine runs on the bare hardware, thus without any interference and latency from an OS.

[a] http://www.nxp.com/products/software-and-tools/hardware-development-tools/freedom-development-boards/freedom-development-platform-for-kinetis-k64-k63-and-k24-mcus:FRDM-K64F

Note
You are interested to use CPAL on a platform that is not supported yet? Please contact us[a].

[a] https://www.designcps.com/contact/

3.3 Parsing source files

The `CPAL parser` takes as input the source code file and, if parsing succeeds, produces a more compact binary equivalent of it which will be loaded and executed by the `CPAL interpreter`. If a syntax error is detected in the source code, a message indicates both the error found and the line number.

If no output file is specified in the command line, the default output file is `out.ast`.

In CPAL, the `include "file.cpal"` directive inserts the external source file indicated at the point where the `include` is invoked. There are several ways to define the location of the files to be included listed below per priority order:

1. through the absolute path, e.g. on Windows: `"c:/src/FRDM/config.cpal"`
2. relatively to the including source file, e.g.: `include "../FRDM/config.cpal"`,
3. relatively to the current directory, e.g.: `include "config.cpal"` if `config.cpal` is in the directory from which the parser is called,
4. the path to the included source files is indicated through the `-I` parser command-line option, e.g.: `-I ./pkg -I ../FRDM`.

Warning
CPAL ships with default libraries for process scheduling and interactions with I/Os that are located in the `pkg` directory included in all distributions. These libraries must be included at parse time, this can be done through the parser option: `-I<path_to_pkg_folder>` or by setting the `CPAL_PKG_PATH` shell variable to the pkg folder location. For instance, on Linux this is done using the command `CPAL_PKG_PATH=/home/user/cpal/pkg`, or, better, by adding `export CPAL_PKG_PATH=/home/user/cpal/pkg` in `.bashrc` file.

```
CPAL Parser
Version 1.21
Usage:
    cpal_parser [<options>] <source file> [<output file>]

            --Werror             Make all warnings into errors
        -I  --include <path>     Specify include path (separated with semi-colon ↩
            ';')
        -m  --memory <size>      Specify maximum model size (default 1000000)
        -q  --quiet              Quiet parsing if no error.

        if <output file> is not specified output is in out.ast
```

Examples of parser command lines:

- `./cpal_parser -I ./pkg my_program.cpal`: the binary file `out.ast` is created
- `./cpal_parser -I ./pkg:../src:./subfolder my_program.cpal my_program.ast`: the binary file `my_program.ast` is created, include files were looked for in two directories

Warning
The include directories in the parser command line should be separated by `:` on Linux and `;` on Windows systems. On both Linux and Windows, blanks in paths must be protected by " " when used on the command line, e.g.: `-I ./"include folder"`. Alternatively it is possible to have several `-I` options in the same command line.

3.4 Enforcing a naming and formatting standard

3.4.1 cpal2x tool

`cpal2x` is a sort of Swiss army knife to manipulate and extract information out of CPAL source files. `cpal2x` serves to:

- format source code according to an indentation standard,
- instrument the code to evaluate the code coverage of tests,
- extract the timing annotations executed along with the functional code,
- generate the functions needed for serialization/deserialization (aka marshalling/unmarshalling) for data exchange over a network.

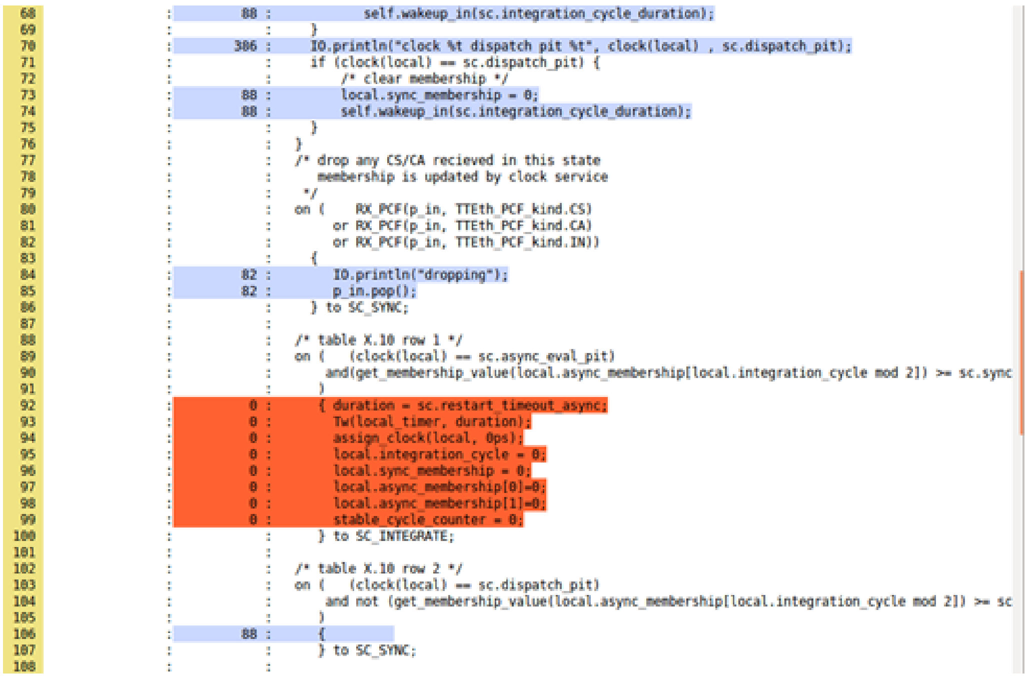
```
$cpal2x
RTaW CPAL Extractor
Version 1.17
Usage :
  ./cpal2x/cpal2x source [output]
  -I --include specify include path
    paths must be separated with colon(:)
  -T --lang       sets the output format.
                  One of json, s-expression, ast-dot, tff, rt-format, c-api-h, c- ↩
                     api-c, cpal, cpal-marshal, arch-dot. Most are experimental!
  --code-coverage When lang is 'cpal' then it also generates interpreted line  ↩
     tracker to do code coverage.
  --discover      When lang is 'cpal' then it also add comments displaying  ↩
     implicit variables.
  --flatten       When lang is 'cpal' then it replace field 'super' of structure  ↩
     by their definition (recursively).
  --annotation    When lang is 'cpal' then it extracts only given annotation  ↩
     namespace.
```

Warning
`cpal2x` should be considered in beta stage.

Here are the instructions to measure code coverage of an execution:

- The CPAL program should be instrumented with the following command: `cpal2x -Tcpal --code-coverage model.cpal`.

- To get the actual line coverage data for an execution, the program should be run with the following options: `--finally-func output_line_coverage --finally-output model.info`.
- To produce an html report of the coverage as below, the generated .info file can be processed by the standard gcov/lcov tools, e.g. `genhtml --branch-coverage -o coverage model.info`.



```
:       88 :            self.wakeup_in(sc.integration_cycle_duration);
:          :     }
:      306 :     IO.println("clock %t dispatch pit %t", clock(local) , sc.dispatch_pit);
:          :     if (clock(local) == sc.dispatch_pit) {
:          :         /* clear membership */
:       88 :         local.sync_membership = 0;
:       88 :         self.wakeup_in(sc.integration_cycle_duration);
:          :     }
:          :   }
:          :   /* drop any CS/CA recieved in this state
:          :      membership is updated by clock service
:          :    */
:          :   on (     RX_PCF(p_in, TTEth_PCF_kind.CS)
:          :         or RX_PCF(p_in, TTEth_PCF_kind.CA)
:          :         or RX_PCF(p_in, TTEth_PCF_kind.IN))
:          :     {
:       82 :        IO.println("dropping");
:       82 :        p_in.pop();
:          :     } to SC_SYNC;
:          :
:          :   /* table X.10 row 1 */
:          :   on (    (clock(local) == sc.async_eval_pit)
:          :       and(get_membership_value(local.async_membership[local.integration_cycle mod 2]) >= sc.sync
:          :      )
:        0 :      { duration = sc.restart_timeout_async;
:        0 :        Tw(local_timer, duration);
:        0 :        assign_clock(local, 0ps);
:        0 :        local.integration_cycle = 0;
:        0 :        local.sync_membership = 0;
:        0 :        local.async_membership[0]=0;
:        0 :        local.async_membership[1]=0;
:        0 :        stable_cycle_counter = 0;
:          :      } to SC_INTEGRATE;
:          :
:          :   /* table X.10 row 2 */
:          :   on (    (clock(local) == sc.dispatch_pit)
:          :       and not (get_membership_value(local.async_membership[local.integration_cycle mod 2]) >= sc
:          :      )
:       88 :      {
:          :      } to SC_SYNC;
:          :
```

3.4.1.1 cpal2cpal beautifier

`cpal2x -T cpal` automatically "beautifies" a CPAL source file according to the CPAL formatting standard. Once formatted in a standard manner, the code is:

- easier to write: you do not need to care about minor formatting issues while programming,
- easier to read: when all sources looks the same, you can focus on the meaning and not the aspect,
- easier to maintain: control version systems would not be tricked and only the true changes will lead to new version,
- uncontroversial: never have a debate about spacing or brace position ever again!

A typical command line for beautifying a source file is as follows:

```
cpal2x/cpal2x -I~/samples -T cpal ~/hello-world.cpal
```

Note
The code beautifier can also be executed from within the `CPAL-Editor` under the “Edit” menu.

3.4.1.2 CPAL Marshalling

`cpal2x` can also generate some helper functions to convert back and forth from communication channels like queues. These helper functions are convenient for exchanging messages (possible made up of complex data structures) over a network, or a model of a network if the CPAL program is a simulation model.

Let us for instance consider a communication network, or a model thereof, that requires as input from the sending node `uint8` variables stored in a queue, and provides the receiving node(s) with `uint8` variables again stored in a queue. For every structure `My_Struct` in a source file, `cpal2x` will generate two functions with the following signatures:

```
My_Struct_serialise_in_queue(
    in My_Struct: s,
    out channel<uint8>: q)
{
    /* ... */
}
My_Struct_deserialise_from_queue(
    in channel<uint8>: q,
    out My_Struct: s)
{
    /* ... */
}
```

These functions can then be used to encode the outgoing messages as a flow of raw `uint8` quantities, as expected by the network, and re-interpret the flow of incoming messages as the data structures manipulated at the application level.

3.4.2 cpal_lint tool

Formatting and naming are like the two sides of the same coin, thus in addition to `cpal2x` we provide another utility tool called `cpal_lint` to provide an automated way to check that a source file complies with the naming convention described in Section 2.14.

```
$cpal_lint
CPAL Lint
Version 1.21
Usage :
        cpal_lint source [output]
        -q --quiet      does not report errors, but count them and return in  ↩
           status code.
        -V --verbose    increase verbosity.
        -I --include    specify include path
                        paths must be separated with semi-colon(;)
        -H              Always print filename headers with output lines.
```

Typical usage and ouput looks like:

```
$cpal_lint src/rand-gauss.cpal
Type declaration "process_state"  does not match CPAL style.
Processdef declaration "GaussianRandomGenerator"  does not match CPAL style.
Variable declaration "currentMax" does not match CPAL style.
```

3.5 Executing CPAL programs

3.5.1 Interactive versus non-interactive mode

Once in binary form CPAL programs can be executed by the `CPAL interpreter`. This can be done in command-line or from within the CPAL-Editor (see Section 3.5.6). First, the user has decide which is the right execution mode between real-time and simulation mode (see Section 3.1). Then the interpreter offers two *functioning modes* which differ with respect to whether the user can enter commands and code statements during the execution:

- the **interactive mode**: the user drives the execution of the CPAL program by giving commands to the interpreter (e.g.: step-by-step execution, execution without interruption during 500ms then break, etc). The user can list and change the values of global variables at run-time, as well as execute additional code statements (see Section 3.5.2 for the set of possibilities). This is the best mode for debugging applications.
- the **non-interactive mode**: the program is executed without asking for user inputs, it runs indefinitely or during a duration specified on the command line. It is however possible to execute a set of commands (a "scenario" of execution) written in a file given as input to the interpreter (with the `--scenario` option, see Section 3.5.3).

```
CPAL Interpreter
Version 1.25
Usage:
    cpal_interpreter.exe [options] <cpal ast>

        <cpal ast>                  the CPAL program in its AST version
        -t --time <limit>           run until time 'limit' in milliseconds is  ↩
           reached
        -v --version                show this help
        -s --scenario <file>        send scenario 'file' commands during the  ↩
           interpretation
        -e --echo                   repeat received commands on stdout
        -i --interactive            enable interactive mode of interpreter
        -q --quiet                  disable verbose mode
        -z --silent                 disable all console outputs
        --finally-func <funcname>   execute no-argument function called 'funcname ↩
           ' before exiting
        --finally-output <filename> redirect output to 'filename' before  ↩
           executing final function
        -l --tasks-log              record processes activity per node in html  ↩
           format
        -m --states-log             record processes states per node in html  ↩
           format
```

Note
On embedded targets without an OS the CPAL interpreter is uploaded in memory (e.g. flash) alongside the program to execute. The interpreter boots up and executes the program at the next reboot of the board. In that case, the real-time execution mode is the only available execution mode.

Warning
The name of the interpreter in the command lines below should be adapted depending on the platform and execution mode according to the information in the table of Section 3.2. Importantly, the real-time execution mode is only available with the executables `cpal_interpreter_linuxmbed`, `cpal_interpreter_winmbed` and `cpal_interpreter_raspberry` and on bare-metal targets. Except for bare-metal targets which only offer the real-time mode, the real-time mode must be explicitly set through the `--realtime` option. If not, the interpreter will run in simulation mode that me with an "as fast as possible" execution (e.g., activation periods of the processes are not respected). The `--realtime` option used with an interpreter only capable to run in simulation mode (i.e., all cpal_interpreter executables) will have no effect.

Examples of interpreter command lines:

- `./cpal_interpreter my_program.ast`: execute indefinitely in simulation mode and non-interactive mode.
- `./cpal_interpreter_linuxmbded -r -q my_program.ast`: execute indefinitely in real-time mode, non-interactive mode and non-verbose mode. This is the command line to deploy CPAL programs on a Linux target.
- `./cpal_interpreter_winmbded -r -s scenario.sce my_program.ast`: execute on Windows in real-time mode the scenario defined in the file `scenario.sce` then exit the interpreter.
- `./cpal_interpreter --silent --time 5000 my_program.ast`: execution in simulation and non-interactive mode during 5000ms, with no outputs to the console.
- `./cpal_interpreter_raspberry -r -v --stats --time 5000 my_program.ast`: execute on Raspberry in real-time, non-verbose and non-interactive mode during 5000ms with the monitoring of the Worst-Case Execution Times (WCET) of the processes. The WCETs are printed on the console at the end of execution. The `--stats` option is only meaningful in real-time mode.

Warning
In real-time mode, `IO.print()` statements take a substantial amount of time to execute and their use should be minimized in time-constrained applications, when measuring code execution times or when performing long simulations. The `--silent` interpreter option goes farther than `--quiet` by suppressing all outputs to the console, including `IO.print()` calls. The arguments of `IO.print()` are however normally executed to avoid side-effects that would for instance happen with `IO.println("%u", some_queue.pop())`.

3.5.2 Working with the interpreter in interactive mode

The interpreter launched with `--interactive`, or `-i` for short, will run in interactive mode. In this mode, it is up to the user to drive the execution of the program. This can be done through the `step` and `run` commands. For instance `run +15us` tells the interpreter to execute the program until current time + 15us, and then stop the execution and wait for further instructions. Any time unit defined in CPAL can be used (see Section 2.13.1).

Note
When specified on the command line of the interpreter through the `--time` option, the time during which a program will be executed is always expressed in milliseconds. In the interactive mode, it is possible to use whatever time units to express time quantities in the `run` command.

The interactice mode is typically useful for debugging a program since it is possible to query and change the value of the variables, and execute the program step-by-step.

```
>> help
Commands:
   step                 Run the process(es) released at the next activation time
   run                  Run without time limit
   run <cpal time>      Run until absolute time (if greater than current time)
   run +<cpal time>     Run for a relative period of time
   list                 Display all global variables, their values, and all  ↩
      processes status
   time                 Display current time (in seconds)
   verbose              Disable quiet mode
   quiet                Enable quiet mode (enabled by default)
   quit                 End the execution and exit the interpreter

Execute a statement such an assignment:
   <global variable> = <value>;
```

Warning
Do not forget `;` at the end of a statement executed in interactive mode.

Let us consider the small CPAL program below:

```
processdef A_Process(in uint8: data)
{
  state Main {
    IO.println("data: %u", data);
  }
}

var uint8: g_variable = 5;
process A_Process: p1[100ms][g_variable > 3](g_variable);
```

Run in Playground[3]

Here is an example interactive execution of the above program in simulation and non-verbose mode:

```
./cpal_interpreter -q -i out.ast
Verbose mode disabled
>> step
[0.000000000000:PRINTLN] data: 5
>> step
[0.100000000000:PRINTLN] data: 5
>> g_variable=6;
>> list
[0.100000000000:VAR] g_variable=6
[0.100000000000:VAR] p1={0,0,0,100ms,0,0,0,100ms,0,Main}
[0.100000000000:STATE] process "A_Process", instance "p1", state "Main"
>> step
```

[3]http://www.designcps.com/cpal-playground?path=talks/cpal-intro/simple-interactive-session.cpal

```
[0.200000000000:PRINTLN] data: 6
>> g_variable=3;
>> step
>> step
>> time
[0.400000000000]
>> g_variable=4;
>> step
[0.500000000000:PRINTLN] data: 4
>> run +200ms
[0.600000000000:PRINTLN] data: 4
[0.700000000000:PRINTLN] data: 4
>> quit
```

The first two `step` commands execute the first two instances of the process respectively at time 0 and 100ms. Then, at time 200ms, the value of `g_variable` is set to 6, as the output of the `list` command confirms. At time 200ms, `g_variable` is assigned 3. As a result of that, the process will not execute anymore since the activation condition (see Section 2.9.1) requires `g_variable` to be larger than 3. At time 400ms, `g_variable` is set to 4 and the next instance of the process is executed at time 500ms. Finally, the command `run +200ms` executes the program without interruption during the next 200ms and then stops it.

3.5.3 Executing scenarios

A scenario is a set of commands written in a file that are executed one by one by the interpreter. Scenarios of execution are for instance useful to run test cases and check that the application behaves as expected. Typically one or several scenarios will test the fulfillment of a certain requirement of the system. A scenario is written in a text file with the extension `.sce` and executed with the command line option `--scenario` (see Section 3.5). In this file, each command given to the interpreter is placed on a single line. The scenario file corresponding the interactive session shown in the previous paragraph is the following:

```
step
step
g_variable=6;
step
g_variable=3;
step
step
time
g_variable=4;
step
run +200ms
quit
```

Lines starting with # are treated as comments. Other possible commands in scenarios are:

- `run 150ms`: execute program until absolute time 150ms, whereas `run +150ms` would execute program during 150ms starting from the current time,
- `@10ms9ns a_function();` execute specified function at an absolute time, any instructions manipulating global variables is accepted too (e.g., "g_value = 4.158;"). If other statements are to be executed at the same time point, the ones declared in the scenario file are executed first, in their order of declarations. Command of the form `@time CPAL-instructions;` is a shortcut is equivalent to issuing the command `run time` followed by the command `CPAL-instructions`.

Note
The interpreter can also accept commands through file redirection or pipes on platforms with an OS and a filesystem.

3.5.4 Final function

It is often useful to execute a function right before the program terminates to perform some specific actions such releasing I/Os or writing information as statistics. Such a function is called a `final function`. Let us consider the following program:

```
var uint64: instances_count = 0;

processdef A_Process(out uint64: count) {
  state Main_State {
    count = count + 1;
  }
  on (uint8.rand_uniform(1,4) == 1)
    to Exit;
  state Exit {
    exit(0);
  }
}

output_count() {
  IO.println("Process executed %u times", instances_count);
}

process A_Process: p1[100ms](instances_count);

init() {
  seed();
}
```

The function `output_count` will be executed after the call to `exit(0)` with the following command line:

- `./cpal_interpreter -q -t 500ms --finally-func output_count out.ast`: execute in simulation mode, non-verbose and non-interactive mode during 500ms then execute function `output_count()` before exiting.

On platforms supporting writing to files (all but embedded boards), it is also possible to redirect the output to a file with the following command-line:

- `./cpal_interpreter -q -t 500ms --finally-func output_count --finally-output log.txt out.ast`: the ouput is here written into `log.txt` file instead of being written to the console. The writing mode is append, which means that the outputs of successive executions can be stored in the same file. In multi-interpreter mode, all interpreters write successively to the same output file.

Note
A final function cannot possess arguments but it can access global variables.

3.5.5 File I/O operations

Using annotations, CPAL allows structured and unstructured data to be written to files. The program below illustrates how to write structures, possibly nested, to a file. CPAL automatically handles file opening and file closing operations.

```
/*
Licence Creative Commons CC0 - "No Rights Reserved"
@keywords annotation, hardware; file, csv; file I/O; IO.sync
*/
include "annotations/hardware/hw_file.cpal"

struct data {
  time64: time;
  uint8: u8;
};

var queue<data>: log[1];
/* `log` stores the data that will be "pushed" into the file */

@cpal:hardware:log
{
  var CPAL_File: logger = {File_Access.WRITE, File_Format.CSV, "csv_text.txt" };
  /* CSV format is reserved for data stored as structures */
  /* opening and closing of the file is handled automatically */
}

processdef foo() {
  static var data: item;
  state Main {
    item.time = time64.time();
    item.u8 = item.u8+1;
    log.push(item);
  }
  after (1s) to Exit;

  state Exit {
      exit(0);
  }
  /* Actual writing to the file takes place upon the (implicit) IO.synch()
  execution immediately after a process activation has finished */
}

process foo: bar[100ms]();
```

It is also possible to write unstructured data to a file as a string of characters as shown in the program below.

```
/*
Licence Creative Commons CC0 - "No Rights Reserved"
@keywords annotation, hardware; file, raw; file I/O; IO.sync
*/
include "annotations/hardware/hw_file.cpal"

var queue<uint8>: log[1];
/* characters are stored as their ascii value */

@cpal:hardware:log
```

```
{
  var CPAL_File: logger = {File_Access.WRITE, File_Format.RAW, "text_file.txt" };
}

processdef foo() {
  static var uint8: item = 50;
  state Main {
    item = item + 1;
    log.push(item);
  /* characters are here written one by one but it could be several in
  a row, in which case the size of 'log' must be adjusted accordingly */
  }
  after (1s) to Exit;

  state Exit {
      exit(0);
  }
}

process foo: bar[100ms]();
```

The other possibility to write to a file is through the `--finally-output` option (see Section 3.5) that allows to redirect all outputs on the console to the same file.

3.5.6 Execution from within the CPAL-Editor

For the majority of the use-cases, the CPAL-Editor offers the easiest way to execute CPAL programs. Indeed, all parameters for an execution (time, input scenario file, output folders, options, etc) can be saved through a dialog as a "launch configuration" file, which can be executed from within the CPAL-Editor. Output files can be included in the workspace and, provided they are in html format, shown inside the CPAL-Editor.

The reader is referred to Section 4.3.3 for instructions on how to execute programs from within the CPAL-Editor.

3.6 Embedded platforms

The interpreter has been currently ported to the following embedded platforms:

- Raspberry
- Embedded Linux
- NXP FRDMK64F

The reader is referred to Section 3.2 for the capabilities of each platform.

3.6.1 IO synchronisation

Programming embedded systems implies interactions with the environment through sensors and actuators piloted by I/O ports. In CPAL, global variables and collections can be mapped to I/O ports. By default, the I/O mapped variables used by a process are updated in reading when the process starts to execute, and in writing at the end of its execution. Sometimes, it is however needed to update I/O mapped variables during the execution of the process, this can be done by an explicit call to `IO.sync()` as illustrated in the program below.

```
processdef My_Proc(in bool: button, out bool: led)
{
  /* IO.sync() implicitly called right before each activation of the process */
  state Main {
    /* Some bit banging */
    if (button) {
      led = true;
      IO.sync(); /* Explicitly synchronizes I/Os */
      sleep(250ms);
      led = false;
    }
  }
  /* IO.sync() implicitly called at the end of execution of the process*/
}

process My_Proc: blinker[500ms](pin2_in, pin0_out);
```

3.6.2 Embedded Linux

Under Embedded Linux, CPAL programs can access any `/dev/tty` port in reading and writing through queues of `uint8` declared as global variables:

```
var queue<uint8>: tty0_in[10]; /* Reading from /dev/tty0 */
var queue<uint8>: tty0_out; /* Writing to /dev/tty0 */
```

As illustrated above, the queues can be of size 1 or dimensioned to hold several data if buffering is needed with respect to the data production production rate and application consumption rate.

3.6.3 Raspberry Pi

CPAL programs on Raspberry can access the following I/Os:

- GPIO pins: read and write
- Files: write only

Note
The Raspberry Pi dedicated interpreter must be launched as root (eg. with `sudo`).

3.6.3.1 Digital Pin

The code snippet below illustrates how to read and write GPIOs on the raspberry.

```
var bool:   pin0_out = false; /* Declare that GPIO 0 is a digital output pin. */
var bool:   pin2_in; /* Declare that GPIO 2 is a digital input pin */
var uint32: pin1_out; /* Declares GPIO 1 as (soft) PWM output with range 0-100 */
```

Note

Not all combinaisons are available. See the documentation of WiringPi[a].

[a] http://wiringpi.com/pins/special-pin-functions/

Warning

Remember that GPIOs of the Raspberry Pi are 3.3V (for reference, Arduino is 5V).

3.6.3.2 File output

The code snippet below shows how to write data to a file on the Raspberry. The actual filename is derived from the name of the queue holding the data to be written to the file.

```
struct A_Record
{
  time64: timestamp;
  uint32: someValue;
};

/* Declares a file where to serialize the queue contents. */
var queue<A_Record>: file_myfilename_csv[10];

/* On IO.sync(), the queue will be emptied and records will be
serialized as CSV into myfilename.csv file. */
```

Note

time64 variables are serialised in picosecond units.

3.6.4 NXP FRDMK64f

CPAL programs on the NXP FRDMK64f board can access the following I/Os:

- GPIO
- DAC (Digital-to-Analogic converter)
- accelerometer
- I2C bus
- serial communication

The program below illustrates how to use a DAC on the FRDMK64f.

```
compute_output_value(in uint16: percentage, out uint16: new_value)
{
    new_value = (4095 * percentage) / 100;
}
```

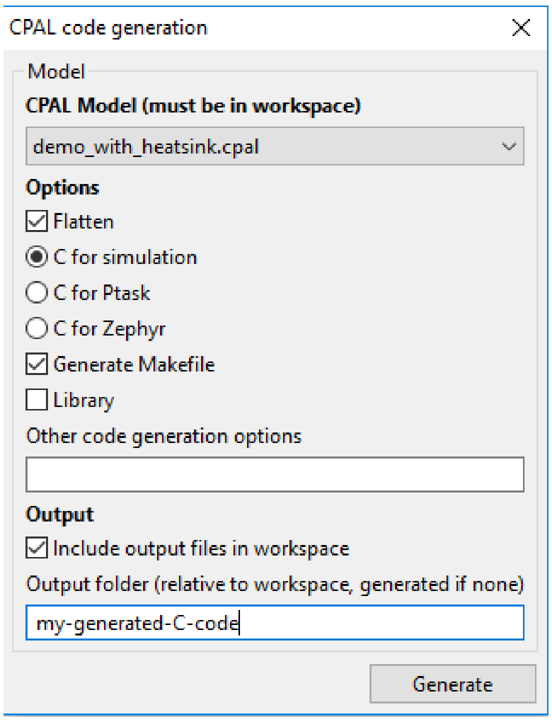
```
processdef DAC(out uint16: pin)
{
    static var uint16: output_value = 0;
    static var uint16: percentage = 0;

    state Increase {
        pin = output_value;
        percentage = percentage + 10;
        compute_output_value(percentage, output_value);
    }
    on (percentage == 100) to Decrease;

    state Decrease {
        pin = output_value;
        percentage = percentage - 10;
        compute_output_value(percentage, output_value);
    }
    on (percentage == 0) to Increase;
}

var uint16: pin_dac0 = 0;

process DAC: p_dac[100ms](pin_dac0);
```

3.7 Generation of C code

CPAL code can be converted into C code by executing the "Generate C code" function located under the "Run" menu of the `CPAL-Editor`. The makefile necessary to compile the C code will be created as well. The options for code generation are:

- "flatten" (mandatory for now): means that a single large C file will be generated including the C code of the external dependencies (i.e., the included CPAL files such as the CPAL standard library).
- "C for simulation": generate a CPAL program that runs in simulation mode (see Section 3.1).
- "C for Ptask": generate C code for the PTASK library (available on GitHub[4]) that provides high-level scheduling primitives (e.g., periodic tasks, scheduling policies, synchronization) on top of the standard Posix pthread library.
- "C for Zephyr": generate C code for the Zephyr OS.
- Library: generate C code without process instantiation that can be used in a library or on non-supported platforms.
- other code generation options: currently only "--verbosity" is possible to obtain more debugging information during code generation process.

[4] https://github.com/glipari/ptask


CPAL code generation

Model

CPAL Model (must be in workspace)

demo_with_heatsink.cpal

Options

- [✓] Flatten
- [● C for simulation]
- C for Ptask
- C for Zephyr
- [✓] Generate Makefile
- [ ] Library

Other code generation options

Output

- [✓] Include output files in workspace

Output folder (relative to workspace, generated if none)

my-generated-C-code

Dependencies needed to compile (e.g., scheduler, C version of the CPAL built-in functions), as can be seen in the generated makefile, are located under:

- `Documents/RTaW/CpalEditor/Samples/generated-code` under Windows,
- `<install folder>/samples/generated-code` under Linux and MacOS X.

Warning
Though C code generation will work fine for a majority of CPAL programs, this feature should be considered in alpha stage. In particular, C code generation is not available for the multi-interpreter execution mode.

Chapter 4

CPAL for simulation

CPAL has been designed to support the formal description, the editing, graphical representation and simulation of cyber-physical systems. Simulation is a key activity for the understanding, debugging of models and for the verification of correctness properties such as timing constraints. This chapter describes the features offered by CPAL to support the development of simulation models.

4.1 Pseudo-random numbers generation

In order to simulate the environment of use of the system that is being developed, or to test different path of executions of a same program, it is often convenient to use pseudo-random numbers. CPAL provides a set of functions to work with random numbers:

- `seed()`: initialize the pseudo-random number generator with the current time of the machine (precision of time is the microsecond) and return the seed as a `uint32` value.
- `seed(uint32 value)`: the pseudo-random number generator is initialized with the parameter value. This enables two runs of the same program to work with similar random values, and reproduce identical executions.
- `X.rand_uniform(X lowerBound, X upperBound)`: return a uniformly distributed `X` random number between lowerBound and upperBound, with `X` being all primitive types (`bool`, `uint32`, `float32`, `time64`, etc).
- `float32.rand_gauss(float32 mu, float32 sigma)`: generate a `float32` random number from the normal distribution with mean parameter `mu` and standard deviation parameter `sigma`.
- `float32.rand_exponential(float32 lambda)`: generate a `float32` random number from the exponential distribution with rate parameter `lambda` (i.e., expected value will be `1/lambda`).
- `X.rand_pareto(X scale, float32: shape)`: generate a random number of type `X` from the Pareto distribution, with `X` being a float32 or an integral type (uintX or intX).
- `An_Enum.choice_uniform()`: return one the values of the `An_Enum` enumeration at random, with equiprobability.
- `A_Collection.choice_uniform()`: return one the values currently stored in the collection at random, with equiprobability.

Pseudo-random number generation is illustrated in the program below:

```
/*
Licence Creative Commons CC0 - "No Rights Reserved"
@keywords random; random, gaussian distribution; random, exponential distribution; ↩
    random, pareto distribution; random, collection; random, enum; random, time64; ↩
    random, seed
*/

enum Cardinal_Points
{
  NORTH,
  SOUTH,
  EAST,
  WEST,
};

var stack<int8>: a_stack_of_int[10];

processdef Simple()
{
  state Main {
    /* random generation of a time quantity */
    IO.println("%t", time64.rand_uniform(0ms, 100ms));
    /* generation interval can span over the negative numbers when type allows */
    IO.println("%d", int16.rand_uniform(-64,64));
    IO.println("%f", float32.rand_pareto(10.0,0.5));
    IO.println("%f", float32.rand_exponential(1.0/50.0));
    IO.println("%f", float32.rand_gauss(0.0, 1.0));
    /* random selection over an enum */
    IO.println("%u", uint32.cast(Cardinal_Points.choice_uniform()));
    /* random selection over a collection */
    IO.println("%d", a_stack_of_int.choice_uniform());
  }
}

process Simple: p1[500ms]();

init() {
  /* Initialize the random generator with the current time of the cpal_system,
  alternatively, it is possible to initialize the generator with a fixed seed */
  seed();

  /* Populate the stack */
  a_stack_of_int.push(7);
  a_stack_of_int.push(0);
  a_stack_of_int.push(-4);
}
```

Run in Playground[1]

[1] http://www.designcps.com/cpal-playground?path=talks/cpal-intro/simple-random.cpal

4.2 Time-varying behaviours of processes

Using specific directives called *annotations* it is possible to implement processes with time-varying characteristics. All annotations can be used in simulation mode, but only some of them can be used in real-time. For instance, it is possible to change the period of a process in real-time mode but not to set its execution time.

4.2.1 Processes with varying interarrival times

The example program below shows a process with a randomly chosen interarrival times between two successive instances. It should be noted that the interarrival times may change in a non-random manner, for instance they may alternate between several values. Changing the interarrival time can be used both in real-time mode and simulation mode.

```
/*
Licence Creative Commons CC0 - "No Rights Reserved"
@keywords timing annotation, period; timing annotation, deadline; simulation;  ↩
    random; introspection
*/

processdef Time_Varying_Period()
{
  state Main {
    IO.println("period: %t",self.period);
    IO.println("offset: %t",self.offset);
    IO.println("current activation: %t",self.current_activation);
    IO.println("previous_activation: %t",self.previous_activation);
  }
}

/* The first instance of the process is executed at time 3ms,
  the subsequent instances with an interarrival time randomly
  chosen in [8,13]ms */

process Time_Varying_Period: p1[10ms, 3ms]();
@cpal:time:p1{
  p1.period = time64.rand_uniform(8ms,13ms);
}
```

Run in Playground[2]

Important
The annotation is executed upon each activation of process `p1`, immediately before the flow of execution enters the body of the process. This is the meaning of the suffix `:p1` in the annotation. Without this suffix, the annotation would be executed only once at the startup of the program.

4.2.2 Processes with release jitters

In many systems, due for instance to limited time accuracy or runtime overhead, tasks may not be released and ready to execute exactly when they are supposed to be. A certain delay, called release jitter, may happen and vary from a task instance to the next. This can be modeled in CPAL using a timing annotation as illustrated in the program below.

[2] http://www.designcps.com/cpal-playground?path=./samples/timing-annotations/random-period.cpal

```
/*
Licence Creative Commons CC0 - "No Rights Reserved"
@keywords timing annotation, jitter; simulation; random; introspection
*/
const time64: min_jitter = 0ms;
const time64: max_jitter = 10ms;
const time64: period = 100ms;

processdef Simple() {
  state Main {
    IO.println("%t %t %t", self.jitter, self.current_activation, self. ↩
        previous_activation);
  }
}

process Simple: p1[period] ();

@cpal:time:p1{
  p1.jitter = time64.rand_uniform(min_jitter, max_jitter);
}
```

Run in Playground[3]

4.2.3 Processes with varying execution times

The example program below illustrates how it is possible to simulate the time it takes to execute the code of the states of a process. The execution time of a state can be static or dynamic, based on a condition as the second process shows or it can be the result of a computation.

```
/*
Licence Creative Commons CC0 - "No Rights Reserved"
@keywords timing annotation, time; annotation, state; annotation, priority; timing ↩
    -accurate simulation
*/

processdef Varying_Execution_Time()
{
  state State1 {
    @cpal:time {
      State1.execution_time = 15ms;
    }
  }
  on (true) to State2;

  state State2 {
    @cpal:time {
      State2.execution_time = 35ms;
    }
  }
  on (true) to State1;
}
```

[3]http://www.designcps.com/cpal-playground?path=./samples/timing-annotations/release-jitter.cpal

```
processdef Conditional_Execution_Time()
{
  state Main {
    @cpal:time {
      if (uint16.rand_uniform(0,2)==0) {
        Main.execution_time = 1ms;
      } else {
        Main.execution_time = 15ms;
      }
    }
  }
}

process Varying_Execution_Time: p1[70ms]();
process Conditional_Execution_Time: p2[200ms]();
```

Run in Playground[4]

Note

The worst-case execution time (WCET) of a process can be derived by monitoring at run-time the program execution on the target using the `--stats` option of the interpreter (see Section 3.5). The WCETs can then be re-injected into the program so to perform more timing accurate simulation.

It is also possible to define an execution time that holds whatever the current state of a process. This has to be done at the global scope as follows:

```
@cpal:time {
      p1.execution_time = 50ms;
 }
```

in which case the execution time will be 50ms at each executions of `p1`, whatever the current state of the process is. Alternatively, one may want the annotation to be executed before each execution of the process:

```
@cpal:time:p1 {
      p1.execution_time = time64.rand_uniform(15ms,30ms);
 }
```

Using named blocks, we can define timing annotations in transitions and combine them with annotations in states as illustrated in the program below:

```
/*
Licence Creative Commons CC0 - "No Rights Reserved"
@keywords timing annotation, time; annotation, state; annotation, transition;  ↩
   timing-accurate simulation; named block
*/

processdef A_Process()
{
  state First {
    @cpal:time {
      First.execution_time = 5ms;
    }
```

[4] http://www.designcps.com/cpal-playground?path=./samples/timing-annotations/two-executions-time.cpal

```
  }
  on (true) {
    A_Named_Block: {
      @cpal:time {
        A_Named_Block.execution_time = 10ms;
      }
    }
  } to Second;

  state Second {

  }
  on (true) {
    A_Second_Named_Block: {
      @cpal:time {
        A_Second_Named_Block.execution_time = 50ms;
      }
    }
  } to First;
}

process A_Process: p1[100ms]();
```

4.2.4 A recap on timing annotations

Timing annotations are dedicated statements inserted in `@cpal:time` blocks to specify the behavior a program must have with respect to timing and scheduling. The timing behavior, along with safety and power consumption for instance, belongs to the set of non-functional properties of a program which are in many embedded systems not of secondary importance to the functional properties. There are two distinct types of timing annotations:

- **execution-time annotations** used in simulation mode only that are meant to reproduce the timing behavior the program would have on a target. These annotations can be defined for states, transitions or at the global scope (see Section 4.2.3). Typically what can be done is to measure on a specific target the maximum execution times of critical portions of the code with an oscilloscope or of complete states of the processes with interpreter option `--stats`, and re-inject them in simulation. The granularity of the execution-time annotations can be state-level down to block-level. Once a program is validated in simulation mode, the timing annotations are ignored when the program is deployed and runs in real-time mode.
- **scheduling annotations** used both in real-time and simulation mode to set
  - process interarrival times (see Section 4.2.1),
  - process activation jitters (see Section 4.2.2),
  - scheduling policy is set through a timing annotation as illustrated in Section 2.10.2),
  - scheduling parameters: priorities for NPFP policy and deadlines for NPEDF (see Section 4.2.1).

An annotation at the global scope is executed only once at the startup of the system with the `@cpal:time` statement, or it is executed right before the start of each execution of the process `p1` with the statement `@cpal:time:p1`. An annotation that is embedded in a process, be it in a state, transition or a named block is executed before the execution of the process.

4.3 Simulating several computational resources with the multi-interpreter

Most systems, even just single computers, are made up of several computational resources such as processors, co-processors, network interface, communication links, etc. These resources may be asynchronous with respect to each other, in the sense that they possess their own local clock and their behavior is governed by independent processes. A modeling facility in CPAL for such systems is to allocate each node to a dedicated interpreter and let the interpreters share information through communication channels (LIFO or FIFO) or simple buffers.The use of multiple interpreters, one per node or, more generally, one per computational unit, is the **multi-interpreter** feature of CPAL.

Note
The multi-interpreter feature is meant for simulation on workstations, not for the actual execution on embedded platforms.

4.3.1 A dedicated interpreter to simulate each resource

A multi-interpreter simulation implies simulating multiple "threads of control/execution" in parallel. The multi-interpreter provides discrete-event simulation with a local time on each interpreter and a global simulation time. The local times of the interpreters progress independently up to a time point where an event in the simulation necessitates a synchronisation point between the interpreters, such as a process reading input data and a process terminating and making updated data available to the other processes. At such time points, the input and output communication channels between interpreters are updated.

Underlying, the multi-interpreter relies on a synchronous semantics where the processes executing on the interpreters can only read input data upon the start of their execution and write output data upon their termination, but not at any other times. Building on this synchronous semantics, the multi-interpreter simulation engine iteratively increases the global simulation time, by jumping to the next scheduled event on one of the interpreter, in such a way as to enforce the precedence constraints induced by the data flows between processes (e.g., no data can be read before it has been produced).

Data can be exchanged between the interpreters through buffers and communication channels. There are two types of such channels, input and output channels, which can be either LIFO or FIFO queues. When buffers are used to communicate, data can be overwritten and is copied, instead of moved as in the case of channels, from one interpreter to the others.

4.3.2 Producer-consumer multi-interpreter example

We here illustrate the multi-interpreter feature of CPAL with the example of a simple producer-consumer model, where the producer and the consumer are executing on distinct processors and communicating via queues.

Each multi-interpreter configuration includes a cpal source file, called "master file", defining in the `init()` function:

- the different interpreters: structures of type `Cpal_Node` specify the names given to the interpreters, the CPAL model running on each interpreter, and the process scheduling policy that is local to each interpreter,
- the data interfaces on each interpreter: structures of type `Node_Port` declare the name of the ccommunication channels or variables used to resp. read/write data to/from other interpreters. It should be noted that in the example below, the `Node_Port` variables are passed as arguments in the declaration of the `Cpal_Flow` without being explicitely declared,
- the flows of data and their direction: structures of type `Cpal_Flow` declare the source `Node_Port` and the destination `Node_Port` of a flow,

- the complete configuration, stored in a structure of type `System_Multi`, that is populated with the `Cpal_Flows` and `Cpal_Nodes`.

The types used to declare a multi-interpreter configuration are all defined in the `annotations/simu/multi.cpal` file, that is shipped with the multi-interpreter and that must be included in the master file. For our simple producer-consumer model with two distinct flows of execution, the master file can be as follows:

```
include "annotations/simu/multi.cpal"

/* The whole configuration is stored in structure "config".
   Type System_Multi is defined in the CPAL standard library. */
var System_Multi: config;

init()
{
  /* Definition of the models executed on the different threads
     of simulation: name, model and local process scheduling policy */
  const Cpal_Node: producer = {"producer","producer.ast",Scheduling_Policy.FIFO};
  const Cpal_Node: consumer = {"consumer","consumer.ast",Scheduling_Policy.FIFO};

  /* Definition of the data flow between simulation threads */
  /* Producer will write data to queue "tokens_out", those data will be received
  by the consumer interpreter in the "tokens_in" queue. */
  const Cpal_Flow: flow_1 = {{producer,"tokens_out"},{consumer,"tokens_in"}};
  config.flows.push(flow_1);

  /* Store configuration in the appropriate data structures.
     All models will then start to execute at the same time. */
  config.nodes.push(producer);
  config.nodes.push(consumer);
}
```

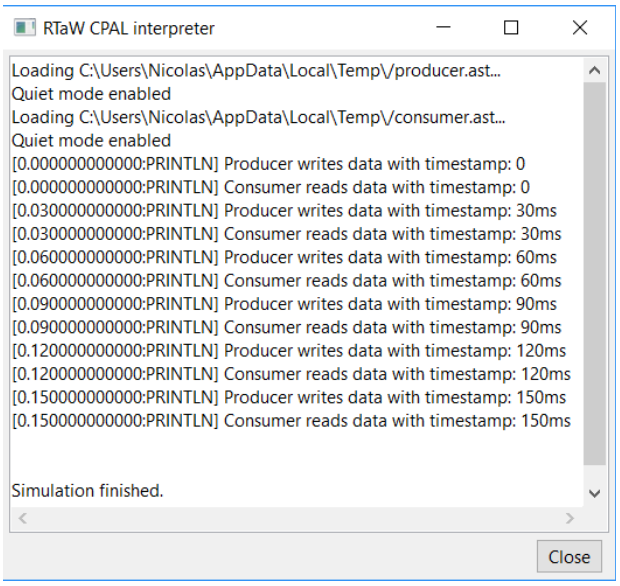
The producer, shown in the program below, writes periodically `time64` data to the queue `tokens_out` that is used to share data with the consumer.

```
processdef producer(out queue<time64>: tokens)
{

  state Main {
    var time64: t = time64.time();
    IO.println("Producer writes data with timestamp: %t", t);
    tokens.push(t);
  }
}

var queue<time64>: tokens_out[20];

/* The producer process "p" is executed every 30ms */
process producer: p[30ms](tokens_out);
```

The consumer execution is here triggered by the reception of a data in the queue. It will thus inherit its execution period from the one of the producer.

```
processdef consumer(in queue<time64>: tokens)
{
  state Main {
    var time64: t = tokens.pop();
    IO.println("Consumer reads data with timestamp: %t", t);
  }
}

var queue<time64>: tokens_in[20];

/* Execution of process "c" of type consumer will be triggered
by the reception of a data in the "tokens_in" queue. */
process consumer: c[][tokens_in.not_empty()](tokens_in);
```

The results of a simulation `150ms` simulation run of the producer-consumer model is as shown below:



```
Loading C:\Users\Nicolas\AppData\Local\Temp\/producer.ast...
Quiet mode enabled
Loading C:\Users\Nicolas\AppData\Local\Temp\/consumer.ast...
Quiet mode enabled
[0.000000000000:PRINTLN] Producer writes data with timestamp: 0
[0.000000000000:PRINTLN] Consumer reads data with timestamp: 0
[0.030000000000:PRINTLN] Producer writes data with timestamp: 30ms
[0.030000000000:PRINTLN] Consumer reads data with timestamp: 30ms
[0.060000000000:PRINTLN] Producer writes data with timestamp: 60ms
[0.060000000000:PRINTLN] Consumer reads data with timestamp: 60ms
[0.090000000000:PRINTLN] Producer writes data with timestamp: 90ms
[0.090000000000:PRINTLN] Consumer reads data with timestamp: 90ms
[0.120000000000:PRINTLN] Producer writes data with timestamp: 120ms
[0.120000000000:PRINTLN] Consumer reads data with timestamp: 120ms
[0.150000000000:PRINTLN] Producer writes data with timestamp: 150ms
[0.150000000000:PRINTLN] Consumer reads data with timestamp: 150ms


Simulation finished.
```

It does not have to be that the behaviour of the consumer is driven by the producer of data. For instance, the consumer can be activated periodically and checks, whenever active, if new data has arrived as illustrated below.

```
processdef consumer(in queue<time64>: tokens)
{
  state Main {
    var time64: timestamp;
    if (not tokens.is_empty()) {
      timestamp = tokens.pop();
      IO.println("Consumer read data with timestamp: %t at time %t", timestamp, ↩
          time64.time());
    } else {
      IO.println("Consumer does not read any data at time %t:", time64.time());
    }
  }
}

var queue<time64>: tokens_in[20];

process consumer: c[30ms](tokens_in);
```

4.3.3 Configuring and executing multi-interpreter simulation

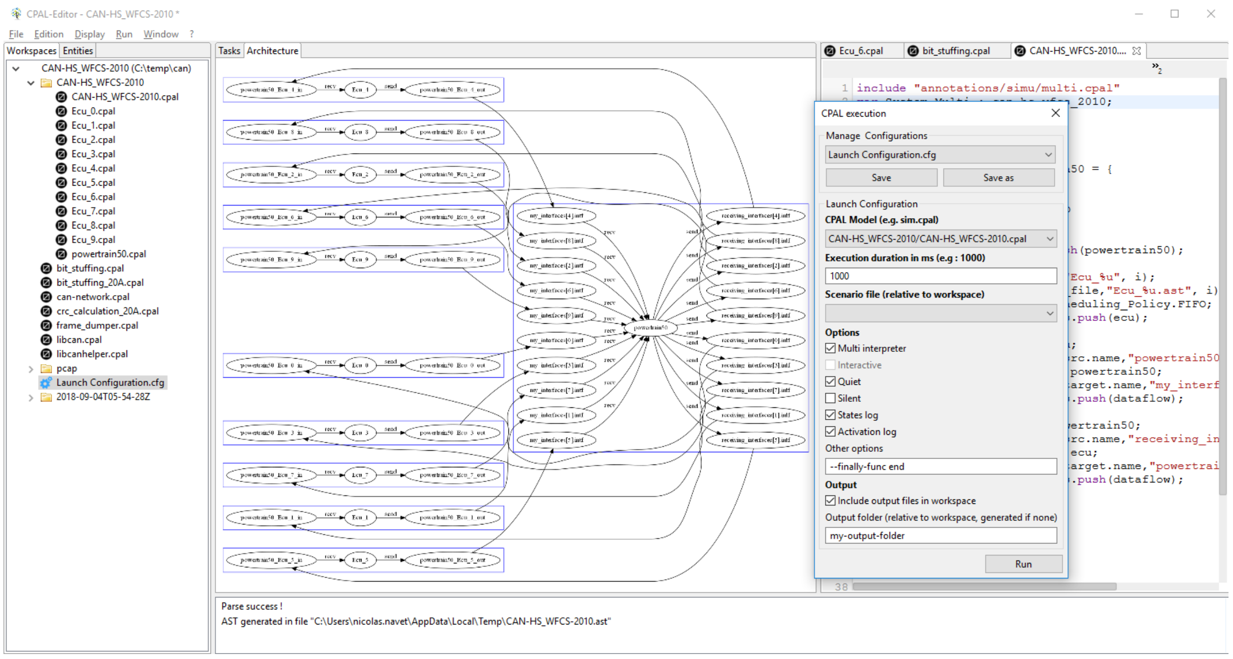
There are two ways to execute multi-interpreter simulation: command-line scripts and using the CPAL-Editor.

The script below is an example batch file for Windows platforms that parses then executes the multi-interpreter configuration used as illustration before.

```
REM sample multi-interpreter batch file

SET CPAL_INSTALL_DIR="C:\Program Files\RTaW\CPAL-Editor-v1.25"
SET CPAL_BINARIES=%CPAL_INSTALL_DIR%\tools
SET CPAL_LIB=%CPAL_INSTALL_DIR%\tools\pkg
REM alternatively, add CPAL_INSTALL_DIR\tools to the path and use -Ipkg

SET MODELDIR=%~dp0

REM step1: parse models
%CPAL_BINARIES%\cpal_parser -I%CPAL_INSTALL_DIR%\tools\pkg "%MODELDIR%\multi.cpal" ↩
     "%MODELDIR%\multi.ast"
%CPAL_BINARIES%\cpal_parser -I%CPAL_INSTALL_DIR%\tools\pkg "%MODELDIR%\producer. ↩
    cpal" "%MODELDIR%\producer.ast"
%CPAL_BINARIES%\cpal_parser -I%CPAL_INSTALL_DIR%\tools\pkg "%MODELDIR%\consumer. ↩
    cpal" "%MODELDIR%\consumer.ast"

REM step2: execute simulation
REM -z option to remove all outputs and speed up simulation
REM -q to disable verbose mode
REM -t simulation time in ms
%CPAL_BINARIES%\cpal_multi_interpreter  -q -I%CPAL_LIB% -t 1000 "%MODELDIR%\multi. ↩
    ast"
```

When it is not needed to automate the execution of simulation runs through scripts, the CPAL-Editor offers a convenient way to run simulations through "launch configuration" files. The CPAL-Editor below shows the functional architecture of a multi-interpreter configuration system (here a CAN communication network) and the dialog that allows to execute CPAL models from within the editor. Checking the boxes “States log” and “Activation log” will create a log in html format of the successive states of the processes and the times at which they execute during the simulation (i.e., start-end of execution). The `--finally-func` option specifies that the (user-defined) `end()` function will execute upon the end of execution (see Section 3.5.4), here to generate output files with statistics and graphical representation of the transmission schedule. Those output files will be included in the workspace by checking the right box in the dialog. Output files in html format can be displayed in the CPAL-Editor. All these choices can be saved in “launch configuration” file (.cfg file), which can be provided along with the model in the worskspace. Through launch configurations, it becomes possible to re-execute the programs in a similar manner in terms of execution time, command-line options, output directory, input scenario file, etc.

The `--progress`, or `-p` for short, allows to monitor the progress of a simulation executed in multi-interpreter mode.



```
include "annotations/simu/multi.cpal"
```

CPAL execution

Manage Configurations: Launch Configuration.cfg

Launch Configuration

CPAL Model (e.g. sim.cpal): CAN-HS_WFCS-2010/CAN-HS_WFCS-2010.cpal

Execution duration in ms (e.g : 1000): 1000

Scenario file (relative to workspace):

Options

- [✓] Multi interpreter
- [ ] Interactive
- [✓] Quiet
- [ ] Silent
- [✓] States log
- [✓] Activation log

Other options: --finally-func end

Output

- [✓] Include output files in workspace

Output folder (relative to workspace, generated if none): my-output-folder

```
Parse success !
AST generated in file "C:\Users\nicolas.navet\AppData\Local\Temp\CAN-HS_WFCS-2010.ast"
```

Note
Launch configuration files are especially convenient for large programs, e.g. simulation models run in multi-interpreter mode, but they can be used also for simpler programs in mono-interpreter mode. In some advanced use-cases, the same source program can be executed either in mono or multi-interpreter mode. Choosing between the two execution modes can be done through the multi-interpreter box of the "CPAL execution" dialog.

Note
Output files resulting from the execution of a CPAL program can be included in the workspace by checking the appropriate box in the "CPAL execution" dialog. If output files are in html format, they can be displayed inside the CPAL-Editor, which embeds a web browser.

4.3.4 Scenarios of execution

Whether a script or the CPAL-Editor is used, a simulation can be driven by a scenario file (with `.sce` extension) that defines a set of actions to execute at certain points in time. If the model is executed from the command line, then option `--scenario <scenario-file>` should be used (see Section 3.5), while the scenario can be specified as input through the "Run" dialog in the CPAL-Editor. Scenarios can be edited in the CPAL-Editor, or with any text editors.

A scenario file contains a sequence of instructions that follows the general syntax: `@time interpreter-name CPAL-instructions`. If time is omitted, then the specified instruction is executed at the current time of simulation. With respect to scenarios in mono-interpreter mode (see Section 3.5.3), the main difference is that the name of the interpreter which is going to execute the statement must be indicated. The scenario below, used in the simulation of a switched Ethernet network, illustrates the possibilities of scenario files.

```
# set a global variable (structure element) to true on interpreter sw0 at time 0
sw0 cm.sync_to_stable_enabled = true;
# other init. on other threads of simulation
sm1 active=false;
sc1 active=false;
```

```
# execute the startup() function on two interpreters at specific time points
@100ms9ns sm1 startup();
@400ms sc1 startup();

# cut a link to a bridge at a precise instant and observe what happens
@500ms link_sw1_sw0 cut_link_state=true;
# next instruction will be executed at 500ms as well
link_sw0_sw1 cut_link_state=true;

#restore link later on in the execution
@700ms link_sw0_sw1 cut_link_state=false;
link_sw1_sw0 cut_link_state=false;

# execute shutdown() on sm1 then sm2
@850ms10ps sm1 shutdown();
@900ms100us sm2 shutdown();
```

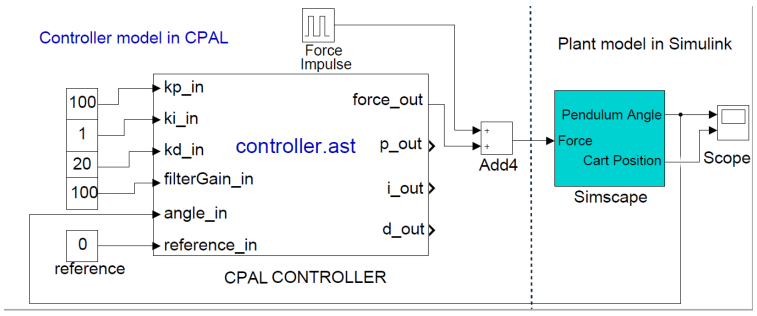
Important
The instructions defined in a scenario will be executed before any other instructions due for execution at the same time point.

Note
if no time unit is indicated in a `@time` statement in a scenario, the default unit is `ms`.

4.4 Co-simulation in MATLAB/Simulink

CPAL can be used in co-simulation with MATLAB/Simulink. Typically the controller model is programmed and simulated in CPAL while the plant model is programmed and simulated in Simulink. If controllers can be designed in Simulink too, Simulink out-of-the-box is not offering possibilities to study the performances of control loops subject to scheduling and networking delays. Indeed, varying execution times, preemption delays, blocking delays, kernel overheads cannot be captured in the standard Simulink environment while they can be accounted for in CPAL through timing annotations (see Section 4.2.4). Moreover, this co-design technique allows to execute the same controller model on the target hardware, without changing a single line of code, with the benefits of reducing development time and not introducing any distortion (i.e., semantic gaps) between the simulated and executed control program.

The figure below shows a controller model for an inverted pendulum integrated within the Simulink environment. The controller model is written in CPAL and executed by an interpreter embedded in the controller block. Input data of the controller are visible in the design window and can be changed without need to access CPAL model. The output data are updated by the CPAL controller.



In the co-simulation of CPAL within MLSL, Simulink acts as the primary simulator while CPAL executes the controller model as an S-function, which is being called by the Simulink engine. S-functions (system-functions) are high-level programming language description of a Simulink block written in C, C++ etc. The CPAL control library is implemented as a *mex* (Matlab Executable) file which executes the CPAL controller model. This CPAL controller is a generic execution engine that can run any CPAL model in .ast form.

More information can be found in a technical paper available here[5]. The CPAL control library needed to execute models written in CPAL in MLSL, as well as example models, is freely available upon request here[6].

4.5 Co-simulation with RTaW-Pegase network simulator

A CPAL program can serve to describe the functional behavior of any distributed application ranging from simple producer-consumer applications to complex functions such as automotive driving assistance functions. CPAL is also well suited to program high-level network protocol layers. A CPAL model is for instance used in this study[7] to simulate the SOME/IP Service Discovery protocol in an automotive Ethernet network.

It is often effective in terms of development time, whenever possible, to re-use the simulation models readily available in other simulation environments, typically network simulators. For instance, CPAL programs can be "coupled" with RTaW-Pegase[8], a tool to predict the performances of embedded communication architectures, which includes a network simulator. More precisely, CPAL processes can send data over the network simulating in RTaW-Pegase and receive data from the simulated network. The interest of using RTaW-Pegase are the following:

- RTaW-Pegase comes with a rich library of models of network components (switches, end-systems, transmission channels, gateways, etc) and communication protocols (Ethernet IEEE802.1Q, AFDX, TTEthernet, Ethernet AVB/CBS, Ethernet TSN/TAS, CAN, CAN FD, ARINC429/825, AUTOSAR Socket Adaptor, etc) that do not have to be programmed in CPAL,
- RTaW-Pegase provides a design space exploration environment with the possibility to compare alternative design solutions ("design variants"), collect and graphically visualize statistics of the quantities of interest,
- simulation will run faster since RTaW-Pegase models execute as binary code and simulations can be executed in parallel on an arbitrary number of CPU cores to collect large data samples,
- the CPAL model remains simple and are not "polluted" with lower-level models.

Through CPAL models and plugins written in JAVA, RTaW-Pegase supports user-defined simulation models, including the applicative level software components.

Note
A CPAL simulation model can be executed with no changes on an embedded target or a workstation in a testbed or a prototype of the system under development. The lower-level layers simulated in RTaW-Pegase, or in CPAL, are simply replaced by their physical counterparts while keeping the same communication interfaces (i.e., communication channels). This supports the rapid-prototyping of a system.

As an illustration, the CPAL program below implements the transmission of a video stream with segmented messages on an Ethernet network. The model hands over the frames once created to the simulation kernel of RTaW-Pegase.

[5] http://www.mdpi.com/1424-8220/18/2/628/pdf
[6] https://www.designcps.com/contact/
[7] http://www.realtimeatwork.com/wp-content/uploads/Date2015-SomeIP.pdf
[8] http://www.realtimeatwork.com/software/rtaw-pegase/

```
/* Copyright (C) 2015, RealTime-at-Work, All Rights Reserved
 *
 * You may use, copy, reproduce, and distribute this sample cource code for any ↩
    non-commercial purpose.
 * Some purposes which can be non-commercial are teaching, academic research,
 * public demonstrations and personal experimentation. You may also distribute ↩
    this Software
 * with books or other teaching materials, or publish the Software on websites,
 * that are intended to teaching for academic or other non-commercial purposes.
 *
 * In return, we simply require that you agree:
 * 1. That you will not remove any copyright or other notices from the Software.
 * 2. THAT THE SOFTWARE COMES "AS IS", WITH NO WARRANTIES. THIS MEANS NO EXPRESS,
 *    IMPLIED OR STATUTORY WARRANTY.
 */

/*cpal:tasks time=1ms*/

const uint32: cam_width = 640;
const uint32: cam_height = 480;
const uint32: cam_bits_per_pixel = 24;
const uint32: cam_fps = 30;
const time64: cam_period = (1000/cam_fps)ms;

const time64: comstack_latency_min = 25us;
const time64: comstack_latency_max = 50us;

const uint32: MTU = 1472; /* bytes for UDP payload.*/

const uint32: image_size_bytes = cam_width * cam_height * cam_bits_per_pixel / 8;

init()
{
  const uint64: network_frames_per_image = uint64.as(image_size_bytes / MTU);
  const time64: max_required_time_for_an_image = comstack_latency_max * ↩
     network_frames_per_image;
  const time64: min_required_time_for_an_image = comstack_latency_min * ↩
     network_frames_per_image;

  IO.println("Bandwidth: %u kBytes/s", image_size_bytes * cam_fps / 1000);
  IO.println("Size of image: %u bytes", image_size_bytes);
  IO.println("Min Required time to send an image: %t", ↩
     min_required_time_for_an_image);
  IO.println("Max Required time to send an image: %t", ↩
     max_required_time_for_an_image);
  IO.println("Period of camera process: %t", cam_period);
  if (min_required_time_for_an_image > cam_period) {
    IO.println("ERROR: period of camera does not allow to send all data for an ↩
       image.");
  }
  if (max_required_time_for_an_image > cam_period) {
    IO.println("WARNING: period of camera may not allow to send all data for an ↩
       image.");
  }
}
```

```
struct Raw_Video_Frame
{
  time64: origin;
  uint32: size;
};

Raw_Video_Frame_payload_in_bytes(in Raw_Video_Frame: msg, out uint32: size)
{
  size = msg.size;
}

processdef Raw_Camera(out channel<Raw_Video_Frame>: port)
{
  state Main {
    var uint32: remaining_bytes = image_size_bytes;
    var Raw_Video_Frame: frame;
    while (remaining_bytes > 0) {
      frame.origin = self.current_activation;
      frame.size = uint32.min(remaining_bytes, MTU);
      /* IO.println("%t %u", frame.origin, frame.size); */
      sleep(time64.rand_uniform(comstack_latency_min, comstack_latency_max));
      port.push(frame);
      remaining_bytes = remaining_bytes - frame.size;
      IO.sync();
    }
  }
}

var queue<Raw_Video_Frame>: pegase_ECU1_Switch#1_REQ1_output[2];

process Raw_Camera: cam1[cam_period](pegase_ECU1_Switch#1_REQ1_output);
```

Chapter 5

Distributed applications in CPAL

CPAL supports the design and programming of applications distributed over a communication network. A CPAL program can be first executed in simulation mode, where the frame transmissions are simulated, and then - without changing the code - in real-time mode where the frames are actually transmitted over the network. This allows to shorten the time to get a prototype of the application, or even to deliver the final ready-to-deploy application.

Version 1.15 of CPAL provides a first support for communication over UDP/Ethernet with the following limitations that will be lifted in later releases:

- each UDP datagram of a stream are of fixed size,
- there is no feedback of possible communication errors to the application (e.g., ICMP messages signaling non-existent UDP port number or UDP port overflow on the receiving end).

Note
Actual frame transmissions over UDP are supported under Linux64 in real-time execution mode. The interpretation engine to use is thus `cpal_interpreter_linuxmbed` (see table in Section 3.2).

The support of CAN (Controller Area Network) is currently under development and will be shortly available.

5.1 Communication over UDP

The UDP datagrams are sent and received through standard CPAL queues (see Section 2.6) that are mapped onto UDP ports. The UDP ports are assigned statically in the CPAL program, be it a UDP port for incoming datagrams and for outgoing datagrams. The UDP ports in use in a CPAL programs are read (to fetch incoming datagrams), resp. written (for outgoing datagrams), through an explicit call to `IO.sync()` (see Section 3.6.1) as well as at the very start and very end of execution of each process. In most cases, an explicit call to `IO.sync()` will not be needed.

Communication over UDP requires a source port, a source IP address, a destination port and destination IP address. A CPAL structure defines a new data type that contains the destination IP address and the data field of a datagram for a sending node, and the source IP address and data field for a receiving node. The other fields needed for communication are transparently taken care of by the CPAL execution engine or the communication stack. The Ethernet interface that will be used is implicitly defined through the IP address.

Note

UDP port numbers range from `0` to `65535` but many are reserved for common protocols or applications. However, port numbers from `49152` to `65535` are considered as private and can safely be used.

A naming convention for the communication queues has to be respected to correctly configure communication over UDP:

- queue name must start with `udp_`,
- then include the socket number to be used,
- then `in_` or `out_` depending on whether the queue serves to send or receive datagrams,
- finally the last part of the name is unconstrained.

Communication over UDP is illustrated with a client-server connection. The program below shows how to send a stream of UDP datagrams to a server and then read back the replies from the server. It is possible to broadcast a UDP datagram by using the destination IP address `255.255.255.255`.

```
/* Copyright (C) 2015, RealTime-at-Work, All Rights Reserved
 *
 * You may use, copy, reproduce, and distribute this sample cource code for any ←
   non-commercial purpose.
 * Some purposes which can be non-commercial are teaching, academic research,
 * public demonstrations and personal experimentation. You may also distribute ←
   this Software
 * with books or other teaching materials, or publish the Software on websites,
 * that are intended to teaching for academic or other non-commercial purposes.
 *
 * In return, we simply require that you agree:
 * 1. That you will not remove any copyright or other notices from the Software.
 * 2. THAT THE SOFTWARE COMES "AS IS", WITH NO WARRANTIES. THIS MEANS NO EXPRESS,
 *    IMPLIED OR STATUTORY WARRANTY.
 * @keywords udp, network; linuxmbed
 */

/*
 * This is a simple UDP client: it sends a request to the server
 * and wait for the server answer
 */

/*
 * A CPAL UDP packet is a struct made up of two UINT8 arrays:
 * - the first 4-byte array is reserved for either the sender
 *   or destination IP address,
 * - the rest is the data payload field.
 */
struct UDP_Datagram
{
    uint8: address[4];
    uint16: size;
    uint8: data[100];
};

/* helper function to display address */
```

```
display_sender(
  in uint8: m0,
  in uint8: m1,
  in uint8: m2,
  in uint8: m3)
{
  IO.print("%u.%u.%u.%u", m0, m1, m2, m3);
}

/* client process definition */
processdef PID_Sender(
  in channel<UDP_Datagram>: rcv,
  out channel<UDP_Datagram>: snd)
{
  static  var bool: wait_for_answer = false;
  static var uint8: alive_counter = 0;

  /* send request */
  state Requesting {
    var UDP_Datagram : p;
    /* server's IP address - broadcast would be 255.255.255.255 */
    p.address[0] = 127;
    p.address[1] = 0;
    p.address[2] = 0;
    p.address[3] = 1;
    p.size = 2;
    p.data[0] = uint8.as(self.pid);
    p.data[1] = alive_counter;
    snd.push(p);
    alive_counter = alive_counter + 1;
    wait_for_answer = true;
  }
  on (true) to Waiting_Answer;

  /* wait for answer */
  state Waiting_Answer {
    while(not rcv.is_empty()) {
      /* received something from server */
      var UDP_Datagram: p =  rcv.pop();
    IO.print("rcv from ");
    display_sender(p.address[0], p.address[1], p.address[2], p.address[3]);
    IO.println(" : %u", p.data[0]);
    wait_for_answer = false;
    }
    /* keep waiting till answer received */
  }
  on (not wait_for_answer) to Requesting;
}


/*
 * A CPAL UDP stream is a queue of packets
 */
/* comm port to server */
var queue<UDP_Datagram>: udp_out_client[20];
/* comm port from server */
```

```
var queue<UDP_Datagram>: udp_in_client[20];

/* client instanciation */
process PID_Sender: p_client[10ms](udp_in_client, udp_out_client);

include "annotations/hardware/hw_udp.cpal"

@cpal:hardware:udp_in_client
{
  var UDP : udp1;
  udp1.port = 12346;
  udp1.direction = Hw_Direction.HW_INPUT;


}
@cpal:hardware:udp_out_client
{
  var UDP : udp1;
  udp1.port = 12345;
  udp1.direction = Hw_Direction.HW_OUTPUT;
}
```

The code of the server is similar to the client, but the server has to first wait for a message from the client before replying. The IP address of the client is read from the queue.

```
/* Copyright (C) 2015, RealTime-at-Work, All Rights Reserved
 *
 * You may use, copy, reproduce, and distribute this sample cource code for any ↩
    non-commercial purpose.
 * Some purposes which can be non-commercial are teaching, academic research,
 * public demonstrations and personal experimentation. You may also distribute ↩
    this Software
 * with books or other teaching materials, or publish the Software on websites,
 * that are intended to teaching for academic or other non-commercial purposes.
 *
 * In return, we simply require that you agree:
 * 1. That you will not remove any copyright or other notices from the Software.
 * 2. THAT THE SOFTWARE COMES "AS IS", WITH NO WARRANTIES. THIS MEANS NO EXPRESS,
 *    IMPLIED OR STATUTORY WARRANTY.
 * @keywords udp, network
 */

/*
  * This is a simple UDP server: this server scans the input port periodically
  * and echoes back a message to the sender on the output port.
*/

/*
 * A CPAL UDP packet is a struct made up of two UINT8 arrays:
 * - the first 4-byte array is reserved for either the sender
 *   or destination IP address,
 * - the rest is the data payload field.
 */
struct UDP_Datagram
{
  uint8: address[4];
  uint16 : size;
  uint8: data[100];
};

/* helper function to display address */
display_sender(
  in uint8: m0,
  in uint8: m1,
  in uint8: m2,
  in uint8: m3)
{
  IO.print("%u.%u.%u.%u", m0, m1, m2, m3);
}

/* Server process */
processdef Echo1_Server(
  in channel<UDP_Datagram>: rcv,
  out channel<UDP_Datagram>: snd)
{
  state Main {
    /* scan input */
    while(not rcv.is_empty()) {
      /* received something from client */
      var UDP_Datagram: p =  rcv.pop();
      IO.print("rcv from ");
      display_sender(p.address[0],p.address[1],p.address[2],p.address[3]);
      IO.println(" : %u",p.data[0]);
      /* echo back message */
      snd.push(p);
    }
  }
}

/*
 * A CPAL UDP stream is a queue of packets
 */
var queue<UDP_Datagram>: udp_out_server[20];
var queue<UDP_Datagram>: udp_in_server[20];

/* server instanciation */
process Echo1_Server: p_server[10ms](udp_in_server, udp_out_server);

include "annotations/hardware/hw_udp.cpal"
@cpal:hardware:udp_in_server
{
  var UDP : udp1;
  udp1.port = 12345;
  udp1.direction = Hw_Direction.HW_INPUT;
}
@cpal:hardware:udp_out_server
{
  var UDP : udp1;
  udp1.port = 12346;
  udp1.direction = Hw_Direction.HW_OUTPUT;
}
```

Chapter 6

Next steps: how to get started with CPAL?

- Want to access more resources to learn CPAL? Get them from our web site[1]
- Want to try CPAL on-line? Start with the example programs that can be executed in the playground[2]
- Want to try it out on your machine? Download from https://www.designcps.com/binaries/
- Interested to use CPAL in your own projects? Please contact us[3]
- Updates about CPAL on Twitter at http://www.twitter.com/#DesignCPS

[1] https://www.designcps.com/resources-to-learn-cpal/
[2] http://www.designcps.com/cpal-code-examples-index/
[3] http://www.designcps.com/contact/

Chapter 7

Syntax of the language in BNF

```
anything::= program

program::= external_definitions

external_definitions::= external_definition
| external_definitions external_definition
| %empty

external_definition::= function_definition
| variable_declaration
| process_definiton
| enum_definition
| struct_definition
| global_process_instanciation
| annotation
| block_name scope

annotation::= '@' hierarchical_symbol scope

hierarchical_symbol::= SYMBOL
| hierarchical_symbol ':' SYMBOL

function_prototype::= SYMBOL '(' parameter_list ')' ';'

process_definiton::= process_prototype
| processdef_key_word SYMBOL '(' parameter_list ')' '{'  ↩
    process_implicit_declaration subprocess_declaration_list  ↩
    process_variables_declaration_list common_optional_block states_list  ↩
    finally_optional_block '}'

processdef_key_word::= 'processdef'

process_implicit_declaration::= %empty

common_optional_block::= %empty
| 'common' scope

finally_optional_block::= %empty
| 'finally' scope

process_variables_declaration_list::= %empty
```

```
| process_variable_declaration
| process_variables_declaration_list process_variable_declaration

process_variable_declaration::= 'static' variable_declaration
| variable_declaration

process_prototype::= processdef_key_word SYMBOL '(' parameter_list ')' ';'

states_list::= state
| states_list state

enum_definition::= enum_start SYMBOL '{' symbol_list '}' ';'
| enum_start SYMBOL '{' symbol_list ',' '}' ';'

enum_start::= 'enum'

struct_declaration_list::= struct_declaration
| struct_declaration_list struct_declaration

struct_declaration::= type_spec ':' SYMBOL ';'
| type_spec ':' 'continue' ';'
| type_spec ':' SYMBOL '[' expression ']' ';'
| channel_type '<' type_spec '>' ':' SYMBOL '[' expression ']' ';'
| channel_type '<' type_spec '>' ':' SYMBOL ';'

type_spec::= SYMBOL

variable_declaration::= 'var' type_spec ':' SYMBOL ';'
| 'var' type_spec ':' SYMBOL '=' initializer ';'
| 'const' type_spec ':' SYMBOL '=' initializer ';'
| 'const' type_spec ':' SYMBOL ';'
| buffer_declaration

scope_variables_declaration_list::= scope_variable_declaration
| scope_variables_declaration_list scope_variable_declaration

scope_variable_declaration::= variable_declaration
| 'static' variable_declaration

buffer_declaration::= 'var' type_spec ':' SYMBOL '[' expression ']' ';'
| 'var' type_spec ':' SYMBOL '[' expression ']' '=' initializer ';'
| 'const' type_spec ':' SYMBOL '[' expression ']' '=' initializer ';'
| 'const' type_spec ':' SYMBOL '[' expression ']' ';'
| 'var' channel_type '<' type_spec '>' ':' SYMBOL '[' expression ']' ';'
| 'var' channel_type '<' type_spec '>' ':' SYMBOL '[' expression ']' '='  ↩
   initializer ';'
| 'const' channel_type '<' type_spec '>' ':' SYMBOL '[' expression ']' '='  ↩
   initializer ';'
| 'var' channel_type '<' type_spec '>' ':' SYMBOL ';'
| 'var' channel_type '<' type_spec '>' ':' SYMBOL '=' initializer ';'

function_definition::= function_prototype

parameter_list::= %empty
| parameter
| parameter_list ',' parameter
```

```
parameter::= in_out type_spec ':' SYMBOL
| in_out channel_type '<' type_spec '>' ':' SYMBOL
| in_out type_spec ':' SYMBOL '[' ']'

in_out::= 'in'
| 'out'

scope::= '{' scope_variables_declaration_list statements_list '}'
| '{' statements_list '}'
| '{' scope_variables_declaration_list '}'
| '{' '}'

block::= block_name scope
| annotation
| scope

block_name::= SYMBOL ':'

subprocess_declaration_list::= %empty
| subprocess_declaration
| subprocess_declaration_list subprocess_declaration

subprocess_declaration::= 'process' SYMBOL ':' SYMBOL ';'

time_trigger::= '[' ']'
| '[' expression ']'
| '[' expression ',' expression ']'
| '[' frequency ']'
| '[' frequency ',' expression ']'

cond_trigger::= %empty
| '[' expression ']'

state::= 'state' state_name block triggers_optional_list

state_name::= SYMBOL

symbol_list::= SYMBOL
| symbol_list ',' SYMBOL

statements_list::= statement
| statements_list statement

loop_command::= B'break' ';'
| 'continue' ';'

iterator_declaration::= postfix_expression 'with' SYMBOL

statement::= standard_statement
| block
| loop_command

triggers_optional_list::= %empty
| trigger
| triggers_optional_list trigger
```

```
trigger::= 'on' '(' expression ')' block transition
| 'on' '(' expression ')' transition
| 'after' '(' expression ')' block transition
| 'after' '(' expression ')' transition
| 'after' '(' expression ')' 'if' '(' expression ')' block transition
| 'after' '(' expression ')' 'if' '(' expression ')' transition

transition::= 'to' SYMBOL ';'

assignment::= unary_expression '=' expression
| unary_expression '=' initializer

initializer::= frequency
| expression
| '{' '}'
| '{' initializer_element_list '}'

initializer_element_list::= initializer_element
| initializer_element_list ',' initializer_element

initializer_element::= frequency
| expression
| '{' initializer_element_list '}'
| '{' '}'

expression::= logical_or_expression

logical_or_expression::= logical_and_expression
| logical_or_expression 'or' logical_and_expression

logical_and_expression::= bitwise_or_expression
| logical_and_expression 'and' bitwise_or_expression

bitwise_or_expression::= bitwise_xor_expression
| bitwise_or_expression '|' bitwise_xor_expression

bitwise_xor_expression::= bitwise_and_expression
| bitwise_xor_expression '^' bitwise_and_expression

bitwise_and_expression::= multiple_equality_expression
| bitwise_and_expression '&' equality_expression

multiple_equality_expression::= equality_expression
| multiple_equality_expression '==' multiple_relational_expression
| multiple_equality_expression '!=' multiple_relational_expression

equality_expression::= multiple_relational_expression
| multiple_relational_expression '==' multiple_relational_expression
| multiple_relational_expression '!=' multiple_relational_expression

multiple_relational_expression::= relational_expression
| multiple_relational_expression comparison_token bitwise_shift_expression

relational_expression::= bitwise_shift_expression
| bitwise_shift_expression comparison_token bitwise_shift_expression
```

```
bitwise_shift_expression::= additive_expression
| bitwise_shift_expression bitwise_shift_token additive_expression

additive_expression::= multiplicative_expression
| additive_expression '+' multiplicative_expression
| additive_expression '-' multiplicative_expression

multiplicative_expression::= unary_expression
| multiplicative_expression multiplicative_token unary_expression

unary_expression::= postfix_expression
| '-' unary_expression
| 'not' unary_expression
| '~' unary_expression
| custom_expression

postfix_expression::= primary_expression
| postfix_expression '[' expression ']'
| '(' expression ')' SYMBOL
| TIME

primary_expression::= call
| SYMBOL
| INTEGER
| STRING
| FLOAT
| BOOLEAN
| '(' expression ')'
| frequency

call_operation::= SYMBOL '(' argument_expression_list ')'
| postfix_expression '.' SYMBOL '(' argument_expression_list ')'

call_data::= postfix_expression '.' SYMBOL
| postfix_expression '.' 'continue'

call::= call_operation
| call_data

standard_statement::= assignment ';'
| call_operation ';'
| if_statement
| WHILE '(' expression ')' scope
| 'for' '(' assignment ';' expression ';' assignment ')' block
| 'loop' 'over' iterator_declaration scope
| assignment
| call_operation

if_statement::= 'if' '(' expression ')' scope
| 'if' '(' expression ')' scope 'else' scope
| 'if' '(' expression ')' scope 'else' if_statement

argument_expression_list::= %empty
| expression
| argument_expression_list ',' expression
```

```
custom_expression::= 'state' SYMBOL

channel_type::= 'queue'
| 'stack'
| 'channel'

comparison_token::= '<'
| '>'
| '>='
| '<='

bitwise_shift_token::= '<<'
| '>>'

multiplicative_token::= '*'
| '/'
| 'mod'

frequency::= FREQ
```

Chapter 8

CPAL's User Licence

Hereafter, “software product” means all executables and accompanying files (example programs and documentation) parts of the CPAL distribution on all execution platforms.

CPAL is copyrighted by the University of Luxembourg and RealTime-at-Work, hereafter respectively referred to as UL and RTaW. The software product is provided as is without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of title, non-infringement, merchantability and fitness for a particular purpose.

No oral or written information or advice given by UL and RTaW, its agents or employees shall create a warranty and user may not rely on such information or advice.

- You may NOT resell, charge for, sub-license, rent, lease, loan or distribute the software product without our prior written consent.
- You may NOT repackage, translate, adapt, vary, modify, alter, create derivative works based upon the software product in whole or in part.
- You may NOT use the software product to engage in or allow others to engage in any illegal activity.
- You may NOT transfer or assign your rights or obligations under this License to any person or company, or authorize all or any part of the software product to be copied on to another user's computer.
- You may NOT decompile, disassemble, reverse engineer or otherwise attempt to discover the source code of the software product except to the extent that you may be expressly permitted to reverse engineer or decompile under applicable law.

UL, RTaW and any third party software vendor or provider shall have no liability to users or any customers of users for any claim, loss or damage of any kind or nature whatsoever, arising out of or in connection with (a) the deficiency or inadequacy of the software product for any purpose, whether or not known or disclosed to the user, (b) the use or performance of the software product, (c) any interruption or loss of service or use of the software product, or (d) any loss of business or other consequential loss or damage whether or not resulting from any of the foregoing.

In no event shall UL and RTaW, or any third party software vendor or provider, be liable to users or any customers of user for any special, indirect, incidental or consequential damages, even if the user has been advised of the possibility of such damages.

Appendix A

Reserved keywords

The following keywords are reserved and cannot be used as identifiers. The list is broken down into categories that indicate where the keywords come from.

A.1 Core language keywords

after	break	channel	common	const	continue
else	enum	finally	for	if	in
loop over	on	out	process	process_state	processdef
self	state	static	struct	to	var

A.2 Data types

bool	int8	int16	int32	int64
queue	stack	time64	uint8	uint16
uint32	uint64	float32	float64	duration64

A.3 Time units

Hz	s	ms	us	ns	ps

A.4 Operators

and	mod	not	or

A.5 Standard libraries

Hw_Direction	UDP	UDP_Group
system	System	Scheduling_Policy

Chapter 9

Index