Project 22703: SYRF

Deliverable 2.2

Combination of Formalisms

1 Introduction

The deliverable reports on three activities in all of which significant progress has, in particular, been made on the level of tool development.

- The smooth integration of formalisms on the semantic level of compiling schemes.
- The combination of formalisms on the common format DC.
- The compilation of mode automata into DC

2 Smooth Integration of Languages

2.1 Semantics and Implementation

The activities centred on the SYNCHRONIE Workbench (see [PMMBH98, HP98, HP98]), had to go through a major revision. The scheme used in the version of the workbench demonstrated at the last review had a few shortcomings which surfaced when being used in projects external to the SYRF-consortium. Beside technical details, the handling of data was considered as the major weakness of the overall approach.

In consequence, the decision was taken to go a step ahead by combining the synchronous programming paradigm with object-oriented design style, and to embed/extend the SYNCHRONIE workbench into the programming environment *synchronousEifel* (*sE*).[1] The inherent advantage of a proper handling of data is accompanied by the more flexible and robust design style generally attributed to object-oriented programming.

Objects in *sE* may be reactive meaning that a behaviour component is specified in a synchronous formalism. As usual objects encapsulate data operations. Reactive objects communicate via signals only, and are executed in parallel. Quite clearly, the design paradigm is very different to that of synchronous languages which use functional decomposition, and where the data space is global.

On the technical side, the translation schemes for the synchronous formalisms have been reconstructed to have a much leaner and more flexible character. The translation schemes are based on a few "semantic functions" which are an abstract implementation of concepts such as "preeemption", and which provide an abstract intermediate layer. These semantic functions behave very much like an action semantics in that the semantics of a specific language like (a

[1] The transition is somewhat similar to that from the Statemate tool to the Rhapsody tool of iLogix.

derivate of) Esterel is expressed in terms of these functions. The semantic function operate on synchronous automata which is the internal layer of behaviour very similar to DC but enhanced with additional semantical information. This architecture is extremely flexible in that it is easy to change notation on the syntactic level as well as it easy to add semantic constructs if needed, for instance in order to enhance efficiency. The details of the translation schemes are presented in [PH98].

The present version of synchronousEifel (*sE*1.1 [sE]) is based on an Esterel-dialect and an extended version of Argos. The next version (*sE*2.0) to be shipped by end of January will have Lustre as a declarative sublanguage. The *sE*-programming environment offers a graphical editor for Argos and stepper environment for debugging. It generates either C-code or hardware-formats such as Verilog or Blif. Version *sE*2.0 should be able to generate DC. The *sE*-environment at present provides interfaces to verification with **ctl**, **ptl**, and synchronous observers. For *sE*2.0 an interface to NP-Tools will be provided.

2.2 Application: The SAAB Case study

Starting from the documents provided by SAAB and an earlier implementation in the SYNCHRONIE Workbench, the SAAB case study has been reconstructed in *sE*1.1 by a combination of Argos and Esterel style programming. A report on the case study is to be found in [MM98]. The upshot is that, even without the *Lustre* being implemented yet, the combination of *Argos/Esterel* with object-oriented methods (or "features" in the idiom of *sE*) are highly useful in that it becomes a mere triviality to verify some of the questions raised in the specification of the case study.

3 Language Connection through DC

Significant progress has been done in the implementation of tools allowing languages to be combined through the common format DC [CS95]. In this section, we present the state of these tools, together with the first experiments of languages combination on the SAAB case study.

3.1 The Lustre front-end lus2dc

A first version of the new LUSTRE-V5 front-end, called lus2dc, is available. It performs the static verifications (types, clocks, unique definition, absence of dependence loops) and translate a LUSTRE program into DC. This first version does not deal with arrays.

Imported nodes have been added to LUSTRE, to allow DC nodes coming from another language to be called from a LUSTRE program. The syntax of the declaration of an imported node only gives its interface, its body being replaced by the keyword "imported". In the dependence analysis, when an imported node is called, its output parameters are supposed not to instantly depend on its input parameter: this is a permissive option, the exact dependence analysis being only possible when the node body is known.

Fig. 1 shows the DC code generated from the following very simple LUSTRE program:

```
%version: DC_1.0 %
%generated by: LUS2DC_5.0.1 %

package: import1
nodes: 2
0:  imp %file: import1.lus% %lc: 1 6 0% %lc_end: 1 36 0%
    flows: 2
    inflows: 1
    0:  x $1 ; %lc: 1 10 0% %lc_end: 1 11 0%
    end: – inflows
    outflows: 1
    1:  y $1 ; %lc: 1 27 0% %lc_end: 1 28 0%
    end: – outflows
    extern:
1:  import1 %file: import1.lus% %lc: 3 6 0% %lc_end: 6 1 0%
    flows: 5
    inflows: 2
    0:  a $1 ; %lc: 3 14 0% %lc_end: 3 15 0%
    1:  b $1 ; %lc: 3 16 0% %lc_end: 3 17 0%
    end: – inflows
    outflows: 1
    2:  c $1 ; %lc: 3 33 0% %lc_end: 3 34 0%
    end: – outflows
    locflows: 2
    3:  $1 ;%part: (0) % %lc: 5 7 0% %lc_end: 5 12 0%
    4:  $1 ;%part: (0) % %lc: 5 16 0% %lc_end: 5 21 0%
    end: – locflows
    definitions: 3
    0:  nodecall: 0(0)(3) at: @$1 ; %lc: 5 7 0% %lc_end: 5 12 0%
    1:  nodecall: 0(1)(4) at: @$1 ; %lc: 5 16 0% %lc_end: 5 21 0%
    2:  equ: 2 $13(3,4) at: @$1 ; %lc: 5 3 0% %lc_end: 5 21 0%
    end: – definitions
end: – nodes
endpackage: – import1
```

Figure 1: DC code of the LUSTRE node import1

```
node imp(x: int) returns (y: int); imported;

node import1(a,b: int) returns (c: int);
let
    c = imp(a) + imp(b);
tel
```

Notice that the DC code in instrumented with *pragmas* (enclosed between "%"), most of which give reference to objects in the source program: for instance, the pragmas

```
%lc: 1 6 0% %lc_end: 1 36 0%
```

associated with the declaration of the node imp indicate that, in the source file (given by the pragma "%file"), the corresponding declaration starts at line 1, column 6, and ends at line 1, column 36.

3.2 The translator scdc

3.2.1 Overview

We describe here the `scdc` translation, which allows ESTEREL programs in intermediate SC form to be compiled into DC format, and then combined at DClevel with other software modules originated from other synchronous reactive formalisms.

The SC format is a proprietary format designed as natural target for compilation from ESTEREL. ESTEREL, being a control-dominated imperative language, enjoys a straightforward interpretation into *boolean equation systems* with boolean registers, at least for its control part. Data are handled elsewhere in tables, and the SC format contains specific instructions so that external data operations are triggered exactly when control demands.

On the other hand the DC common exchange format does not separate control from data, the former being mostly represented as boolean activity conditions merged with other typed variables. It is a natural compilation target for data-flow languages such as LUSTRE and SIGNAL.

So the basic principle of the `scdc` translation consists in reintroducing data variables and values into the (synchronous) equation system, unifying control and data variables. In practice there are many more semantic issues due to the specifics of both formats and the need for precise correspondance.

3.2.2 How scdc works?

The SC and DC formats share a number of definitional tables : types, constants and external functions are essentially stored identically.

A main difference holds in operations and assignments, which are stored as external tables in SC, but represented as equational definitions in DC. As a consequence, a data variable can assume several successive values in the same instant in SC, while it should be assigned a unique instantaneous value in DC. `scdc` duplicates variables from SC representations in enough distinct occurrences so that each instance only gets a single value at each instant. This requires a preprocesing algorithm with a precise dependency analysis on variable assignments.

For each SC boolean equation triggering an action we generate two entries in the definition table of the target DC file:

1. the DC form of the original equation (on control variables);
2. a data handling definition implementing the actual data action.

Meanwhile, we generate a *dependency* DC table from the *access* dependency informations present in the SC file.

Also signal presence conditions, retained in SC from ESTEREL, are implemented in DC through `at` statements on clocks.

Using `scdc` we have been able to treat the whole SAAB case study, combining a control part for the various modes written in ESTEREL with a data part computing boundary conditions in LUSTRE. The combined code could enter all DC tools, developed either at Inria/Spectre or at Inria/Ep-atr.

3.3 The DC compiler drac

drac is a general DC translator, developped by Yannick Raoul at Inria/Spectre. It is invoked by a command of the following form drac <files > -main <node > <options >

It can translate a set of DC files either into a transformed DC file, or into a target language (presently C, C++, or Java). Here are examples of combined options:

-merge -DC takes several DC files and merges them into a single DC package. Note that such a merge involves a complete renumbering of objects (constants, types, functions, procedures, and nodes)

-DCE -pty expands a program, given by several DC files, into a single node. The "-pty" option asks for the resulting code to be printed in a readable form. During the expansion, since each node can have several instances, a "table of instances" is built to record the suitable relation with the source code.

-C -loop compiles a program, given by several DC files, into a C function. The "-loop" option produces a standard main program, which is a single loop, getting the inputs from stdin, calling the function, and writing its outputs on stdout.

For the time being, the generated code is not optimized. DC arrays and windows are not yet implemented. drac has been experimented on DC code produced from LUSTRE (by lus2dc) and from ESTEREL (through scdc), and on the result of compiling mode automata (see §4).

3.4 The DC+ format and compiler

The DC+ format, issued from european projects Synchron, Sacres and Syrf, serves as a common representation for programs (and properties) completely or partially described with *synchronous languages* such as SIGNAL, LUSTRE, ESTEREL, or STATECHARTS, and destined to verification and code generation tools. The DC+ representation of languages of the IEC 1131 norm concerning industrial automatisms is under study.

3.4.1 Organization in sub-formats

Different levels of DC+, or sub-formats, have been identified (see figure 2). The objective of the characterization of these different levels, and of the inter-level (or inter-format) transformations, is to adapt a given representation to the functions that can be applied to it.

Thus, the bDC+ (for "boolean DC+") sub-format, in which clocks, represented as boolean flows, are organized in a hierarchy for which there exists a master clock, is the right entry point for tools based on the clock hierarchy, such as for example, code generators.

The STS (for "Symbolic Transition Systems") sub-format of bDC+, in which the hierarchy of clocks is flat (the present/absent status of a flow is defined at any instant by some boolean), is used as an input for verification tools.

The inter-format transformations, DC+ $\rightarrow$ bDC+ and bDC+ $\rightarrow$ STS, have been defined and implemented.

The transformation DC+ $\rightarrow$ bDC+ consists in:

- the addition of a root clock, *tick,* if the clock calculus has not synthesized one; in that case, boolean flows are added as new inputs for the program (reparameterization of the system of equations);

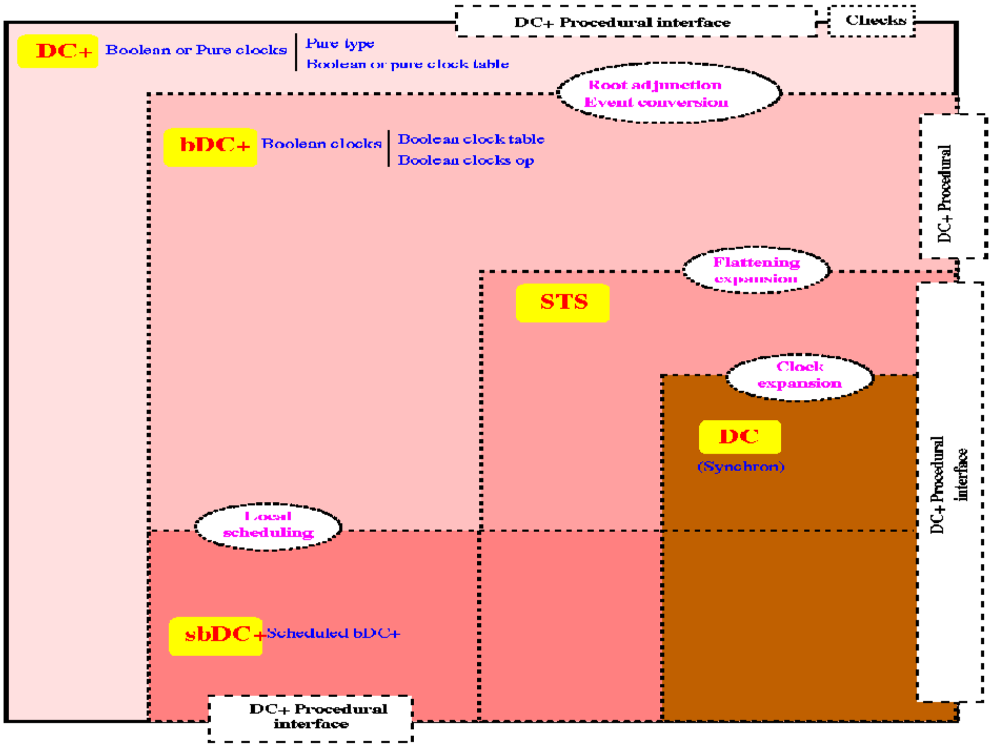

Figure 2: *The DC+ architecture*

- the replacement of pure flows by boolean flows (including the inputs–outputs); in case a pure input flow is constrained inside the program, it is suppressed from the inputs and the boolean representing its clock becomes an output of the program.

Then the transformation bDC+ $\rightarrow$ STS consists in:

- the memorization at the clock *tick* of all the delayed flows (state clock);
- the extension of all the boolean clocks at *tick* (complemented by the value *false*).

3.4.2 DC+ tools

A translator from Signal to DC+ is available. Besides that, an important work has been done to restructure the Signal compiler so that the internal graph representation is now the same one for Signal programs and for DC+ ones[2]. This gives access to the same functionalities for DC+ and for Signal programs: clock calculus, graph calculus, inter-format transformations, interface calculus, code generation, access to proof systems (Sigali...)... In particular, graph manipulation algorithms allowing separate compilation and distribution of Signal and DC+ programs have been implemented.

3.4.3 Further work

Since DC can be considered as the monochronous sub-format of DC+, and some very useful tools are available at the DC level, it would be nice to have a translation from DC+ to DC (not yet

[2]However, windows and arrays are not yet implemented in DC+.

available). For that, the transformation STS → DC has to be defined. Note that DC+ tools are already available for DC programs.

On the other hand, the handling of LUSTRE programs at a higher level than that of DC could be interesting. A prototype of translator bDC+ → LUSTRE has been implemented, and it would be very useful to have a translator LUSTRE → bDC+.

3.5 Experiments on SAAB Case Study

The progress achieved this year concerning practical combinaison of synchronous formalisms allows us to fully implement the SAAB case study in a mixed LUSTRE/ESTEREL version.

Already last year we provided a mixed description where the control part of the climatic chamber regulation was implemented in ESTEREL and the data part in LUSTRE. For the time being, both ESTEREL modules and LUSTRE nodes are separately compiled into DC format (with an intermediate step through SC format for the ESTEREL part. The resulting DC objects are linked into a single DC package. This experiment uses several software tools developed inside SYRF:

- the `lus2dc` front end of LUSTRE language which now support imported nodes. This new feature allows us to stay at the DC level and no more uses the `dc2lustre` "decompiler".
- the `scdc` front end of ESTEREL language which now expands the action table into the equation system.
- the `drac` tool which now has a linking functionality for DC nodes.

3.5.1 SAAB case study overview

This case study models the control of the climatic chamber in an aircraft. The purpose of the system is to keep the selected temperature and ventilation within given bounds. There is a control panel to indicate the state of the system and to give warnings when error occurs. The system continuously reacts with its environment through sensors catching the actual temperature and flow.

The system is started with an *on_off* switch and it can be in three modes : *wait*, *active* and *block*. From the current mode and appropriate sensor values, the actual temperature and flow are updated and also shown on the control panel.

The application is split in three parts:

1. one component computes the current mode of the system according to various temporal conditions. Each time a mode is reached , a lamp (*Panwait*, *Panoper* or *Panblock*) is highlighted on the control panel. There is also a *Panaudio* warning to be emitted when the temperature overpasses its allowed boundaries.
2. a second component regulates the actual temperature of the chamber(*Conheat* signal) inside the prescribed interval. The *Confan* signal is displayed on the control panel.
3. a third component similarly regulates the ventilation of the chamber as required from the sensors of the airflow (*Confan* signal). The *Confan* signal is displayed on the control panel.

3.5.2 Mixed LUSTRE/ESTEREL version

The three parts above are represented as finite state machines and their behavourial steps form the control part of the application. It is encoded using ESTEREL). The data flow computation of boundary conditions from sensor values remains purely combinational, and well-suited to LUSTRE. We provide code fragments in annex.

3.5.3 Experimental results

Combinaison of formalisms at DC level allows to apply the DC back-end tools on files from various sources. For instance, we ran C code for our mixed LUSTRE/ESTEREL application obtained as target code from the DC to C compiler developed in the Spectre site, with the associated simulator, on a typical simulation data file. We got identical results as compared to another simulation, this time with the application fully implemented in ESTEREL, under the ESTEREL language simulator XES.

Moreover, since DC+ tools developed at Ep-Atr site can also deal with plain DC format, the SIGNALback-end compiler was run on the mixed DC file, producing C code. The generation took two steps:

1. the individual DC codes were expanded prior to C code generation.
2. separate C code files were independently produced for each component of the application, then compiled and linked at the C level.

The resulting C code again coincides for simulation on our testbench with the previous experiments.

On the other hand, specific ESTEREL back-end verification and analysis tools were used to check correctness properties on the control part of the application. Using our FC2TOOLS package we proved that no two lamps out of *Panwait*, *Panoper* and *Panblock* lamps can be simultaneously highlighted. We checked also that the warning signal *Panaudio* is emitted only when the system is active and the temperature is out of bounds. To establish these results, first of all the ESTEREL control program was fully expanded into an explicit finite state machine. Then , we restricted observation to only the significant signals, and performed bisimulation minimisation reducing the whole system to a small automaton, displayed and laid out using AUTOGRAPH.

3.6 Source code fragment

This subsection provides most of the mixed code for the first part of the application, which sets the current mode of the system.

3.6.1 LUSTRE modeselect node

```
---------------- imported node declaration
node compute_mode
   ( _con_onoff__bool : bool;
     _dix_solution_time_con_onoff__bool : bool;
     _In2deltaTemp__bool : bool;
     _OutdeltaTemp__bool : bool;
     _TempdLess5Tgrad__bool : bool;
     _newpantreq__bool : bool)
returns
```

```
   ( _Wait__bool : bool;
     _Sol__bool : bool;
     _Work__bool : bool;
     _Block__bool : bool;
     _Panaudio__bool : bool;
     _Panwait__bool : bool;
     _Panoper__bool : bool;
     _Panblock__bool : bool;
     _Warntemp__bool : bool); imported;
---------------------------------------------
const
  deltatemp = 50 ;
  tgrad = 50 ;         -- gradient in 1/10oC per 1/2 sec
  solution_time = 600; -- time in sec

node modeselect (con_onoff: bool; pantreq: int; chamtact: int)
returns (Wait, Sol, Work, Block,
         Panaudio, Panwait, Panoper, Panblock, Warntemp: bool);
var temp0, temp1, temp2, temp3, temp4, temp5: int;
    newpantreq, In2deltaTemp, OutdeltaTemp, TempLess5Tgrad: bool;
let
    newpantreq = false -> if (pre(pantreq) = pantreq) then false else true;
    temp0 = chamtact->pre(temp1); temp1 = chamtact->pre(temp2);
    temp2 = chamtact->pre(temp3); temp3 = chamtact->pre(temp4);
    temp4 = chamtact->pre(temp5); temp5 = chamtact;
    TempLess5Tgrad = true-> if (con_onoff) then
                                if (temp5-temp0 < 5*tgrad) then true else false
                            else pre(TempLess5Tgrad);
    In2deltaTemp = true ->
                    if (con_onoff) then
                       if ((pantreq-chamtact <= 2*deltatemp) and
                           (chamtact-pantreq <= 2*deltatemp)) then true else false
                    else pre(In2deltaTemp);
     OutdeltaTemp = false ->
                    if (con_onoff) then
                       if ((pantreq-chamtact > deltatemp) or
                           (pantreq-chamtact < deltatemp)) then true else false
                    else pre(OutdeltaTemp);
     (Wait,Sol,Work,Block,Panaudio,Panwait,Panoper,Panblock,Warntemp) =
        compute_mode(con_onoff, timer(con_onoff),
                     In2deltaTemp, OutdeltaTemp, TempLess5Tgrad, newpantreq);
tel -- modeselect
```

3.6.2 ESTEREL **compute_node module**

```
module compute_mode:

input con_onoff, dix_solution_time_con_onoff;
output Wait, Sol, Work, Block;  %modes
output Panaudio, Panwait, Panoper, Panblock, Warntemp; %as panelout
input  In2deltaTemp, OutdeltaTemp, TempdLess5Tgrad, newpantreq;

suspend
[  trap block in
      await tick;
      loop                     %wait state
         weak abort
```

```
            [sustain Wait || sustain Panwait]
         when immediate In2deltaTemp;
         weak abort           %sol state
            weak abort
               [  weak abort
                     await tick;
                     [
                        sustain Sol
                      ||
                        await immediate [not (In2deltaTemp)]
                           do emit Warntemp end
                     ]
                     when dix_solution_time_con_onoff
                     do await tick;   %work state
                     [
                        sustain Work
                      ||
                        await immediate OutdeltaTemp
                           do emit Panaudio;emit Warntemp end
                     ]
                  end weak abort
                 ||
                  await tick;sustain Panoper
               ]
            when [not (In2deltaTemp and TempdLess5Tgrad)]
            do exit block
            end weak abort
          when [OutdeltaTemp and newpantreq]
          do  await tick
          end weak abort
       end loop
   handle block do
      await tick; %block state
      [sustain Block || sustain Panblock || sustain Warntemp]
   end trap
]
when immediate [not con_onoff]
end module
```

4 Compiling Mode Automata into DC

Significant progress has been done, this year, about the compilation of "Mode Automata" [MR98a], a model that was defined last year. In [MR98b], we investigate several solutions:

1. Translating Mode-Automata into Lustre: this solution is not completely satisfactory, because modes are imperative concepts, that must be translated by means of Lustre *clocks*. On one hand, this translation is heavy, and on the other hand, the Lustre compiler will translate back these clocks into the more imperative "*activation conditions*" available in DC. Moreover, interesting information is lost, concerning the exclusivity of clocks and activation conditions corresponding to distinct modes. It seems more efficient — concerning both the compilation and the resulting code — to translate the control structure directly into DC.

2. Translating Mode-Automata into DC: the problem, here, is that the Lustre code attached with modes has to be translated, and that we don't want to write a new Lustre compiler. Our goal is to use the existing translator from Lustre to DC.

3. As a consequence, the proposed solution consists of

   - encoding the control part (automaton) directly into DC
   - encapsulating each system of Lustre equations into syntactically correct Lustre programs, then passed to the Lustre-to-DC compiler.
   - analysing the resulting DC code, extracting the relevant parts, and merging them into the automaton encoding.

   A prototype compiler is available, and has been used to compile the mode automaton corresponding to the SAAB case study. Moreover, this compilation scheme has been proven correct, using PVS.

A Example Program of *sE*

The following example program is a complete *sE*-application. It demonstrates some features of the language. The source text is presented together with some remarks.

The configuration class initializes the application and connects it to the environment. The worst case reaction time is specified: The application must respond faster than 1/10 second to react in a perfect synchronous manner.

```
configuration class Rc
timing 1/10 sec;

creation
  run (e : ExceptionEnum) is
    do
      sat.make;
      reactive loop ;
    end ;
```

The creation feature `run` must be supplied for initalization of the whole application. The very first time it is called with actual parameter `startup`. Typically this takes place on power on or effecting a full reset. The statement `reactive loop` either executes forever or is interrupted, when a run-time error occurs. In the latter case `run` is (re-)called with an enumeration value denoting the error.

```
input
  SELECT  = sat.SELECT;
  TOGGLE = sat.TOGGLE;
output
  RED_LED_ON   = red_led.ON;
  GREEN_LED_ON = green_led.ON;

local
  red_led.COMMAND   = sat.RED_LED   [LedState] with mk_off;
  green_led.COMMAND = sat.GREEN_LED [LedState] with mk_off;

reactive sat : SelectAndToggle ;
reactive red_led, green_led : LedControl ;
```

```
feature
  mk_off( l1,l2 : LedState ) : LedState is
    do
      if l1 = l2 then
        result := l1;
      else
        result := [LedState].off;
      end;
    end;

  emit_RED_LED_ON_this_instant is
      external "red_on"
    end;

  emit_GREEN_LED_ON_this_instant is
      external "green_on"
    end;

end class Rc
```

The input and output signals of the configuration class are those, which are connected to the environment of the system. As shown in the example, all kinds of signals may be mapped to signals of reactive (sub-) objects. The local signals of the example carry an enumeration value of class `LedState`. The `with`-clause introduces a combination function: if the signal is emitted twice (or more), the function `mk_off` is called automatically to combine two values to a new one. Three reactive sub-objects are declared. Each of the two `LedControl`-object has its own set of signals declared in its class: `red_led.ON` and `green_led.ON` are different signals, for instance.


```
feature
  mk_off( l1,l2 : LedState ) : LedState is
    do
      if l1 = l2 then
        result := l1;
      else
        result := [LedState].off;
      end;
    end;

  emit_RED_LED_ON_this_instant is
      external "red_on"
    end;

  emit_GREEN_LED_ON_this_instant is
      external "green_on"
    end;

end class Rc
```


The feature part of the configuration class defines the combination function for the two `COMMAND`-signals, and, for each output signal it defines a procedure, which is called when the signal is emitted. A function or procedure may be declared external, if it is not implemented in *sE*, but in C for instance.

```
enum class LedState
feature
```

```
  on, off;
end class LedState
```

Enumerations are a common case in controller applications.

```
reactive class SelectAndToggle

creation
  make is
    do
      set_off;
    end;

input
  SELECT;
  TOGGLE;
output
  RED_LED[LedState];
  GREEN_LED[LedState];
```

The reactive class `SelectAndToggle` reacts to two input signals: `SELECT` selects between a red and a green LED. TOGGLE switches the selected LED on and off. This behavior is defined in the reactive implementation `reactive`, that follows.

```
reactive is
    do
      loop
        cancel
          set_off;
          loop
            await TOGGLE;
            toggle;
            emit RED_LED(led_state);
            next;
          end ; -- loop
        when SELECT
        end ; -- cancel
        emit RED_LED([LedState].off);
        next;

        cancel
          set_off;
          loop
            await TOGGLE;
            toggle;
            emit GREEN_LED(led_state);
            next;
          end ; -- loop
        when SELECT
        end ; -- cancel
        emit GREEN_LED([LedState].off);
        next;
      end ; -- loop
    end ; -- reactive
```

The outermost loop is executed forever, because there is no surrounding cancel- or terminate-statement. Initially the red LED is selected. The first group of statements is executed until a

SELECT-signal is present. In that case, the LED is switched off, and starting with the next instant the green LED is serviced. When the inner part of the first cancel is started, the procedure set_off is called and a loop is entered. The inner part of the loop waits for a TOGGLE-signals and emits the RED_LED with the appropriate value. The handling of the green LED is the same.

```
feature
  set_off is
    do
      led_state := [LedState].off;
    end;

  led_state : LedState;

  toggle is
    do
      if led_state = [LedState].on
      then
        led_state := [LedState].off;
      else
        led_state := [LedState].on;
      end;
    end;
```

The feature part declares and defines the (abstract) data needed in the reactive part.

```
sequence
  write_before_read: set_off* < toggle* < led_state*;

end class SelectAndToggle
```

In the sequence-part the designer declares his or her assumptions about a schedule of data accesses. In the example the assumption is expressed, that set_off must be scheduled before toggle, and toggle before led_state. The compiler checks, whether the reactive implementation is consistent with all assumptions.

```
reactive class LedControl

input
  COMMAND[LedState];
output
  ON;

reactive is
  do
    await COMMAND;
    loop
      if value(COMMAND) = [LedState].on
      then emit ON;
      end;
      next;
    end;
  end;

end class LedControl
```

The class LedControl transforms the switching commands ON and OFF to a "continous" output signals ON.

```
enum class ExceptionEnum
feature
  startup,
  memory_exc,
  require_exc,
  ensure_exc,
  constraint_exc,
  watchdog_exc,
  check_exc ;
end class ExceptionEnum
```

The class `ExceptionEnum` is required to declare the values for the startut-procedure `run` of the configuration class. In this example the minimal set is shown. The programmer may add own exception-values and raise them.

sE may run stand-alone or embedded in a small C-program. To demonstrate this, the complete C-program is given below, that is needed for a executable application. The C-wrapper includes needed header-files, one of which is part of the sE-installation, and contains the function `main`, which initializes the reactive engine and calls the procedure `run` of class `Rc`. `run` (written in *sE*) in turn will execure `reactive loop`, which calls the C-function `schedule`. `schedule` reads lines from stdin, produces input signals, triggers an instant, and writes the generated output-signals to stdout.

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>
#include "/home/budde/bin/sE/Misc/sedef.h"

main () {
  do_init ();
  f_Rc_run(get_configuration (),1);
  return;
}

static char zLine[21];

void schedule (void) {
  while ( fgets(zLine,20,stdin) ) {
    if (zLine[0] == 'q') {
       fprintf (stdout,"quit of schedule\n"); fflush (stdout);
       return;
    } else if (zLine[0] == 't') {
        f_Rc_set_TOGGLE_next_instant ();
    } else if (zLine[0] == 's') {
        f_Rc_set_SELECT_next_instant ();
    } else if (zLine[0] == 'I') {
        do_instant (); fprintf (stdout,"\n");
    } else {
        fprintf (stdout,"input ignored\n"); fflush (stdout);
    };
  };
  fprintf (stdout,"end of schedule\n"); fflush (stdout);
  return;
}
```

```
red_on () {
  fprintf (stdout,"RED "); fflush (stdout);
  return;
}

green_on () {
  fprintf (stdout,"GREEN "); fflush (stdout);
  return;
}
```

References

[CS95] C2A-SYNCHRON. The common format of synchronous languages – The declarative code DC. Technical report, Eureka-SYNCHRON Project, October 1995.

[HP98] L. Holenderski and A. Poigné. The Multi-Paradigm Synchronous Programming Language LEA. Presentation at: Formal Techniques for Hardware and Hardware-like Systems (FTH'98), Marstrand, 1998 , Draft Paper: `ftp://set.gmd.de/pub/vers/papers/LEA.ps.gz`

[HP98] L. Holenderski and A. Poigné. Synchronie Workbench. in: R.Berghammer, Y.Lakhnech (eds), Tools'98, Malente to appear: LNCS, 1998 (`ftp://set.gmd.de/pub/vers/papers/TOOLS98.ps.gz`)

[MR98a] F. Maraninchi and Y. Rémond. Mode-Automata: About Modes and States for Reactive Systems. in: European Symposium On Programming, Lisbon (Portugal), Springer Verlag, LNCS 1381, March 1998.

[MR98b] F. Maraninchi and Y. Rémond. Running modes in a data-flow language: Mode-automata and their implementation. Submitted for Publication, 1998.

[MM98] M.Müllerburg. Designing and Validating the SAAB example with *sE*. Draft,1998.

[PMMBH98] A. Poigné, O. Maffeis, M. Morley, R. Budde and L. Holenderski. The Synchronous Approach to Designing Reactive Systems. in: Formal Methods in System design, 12, 163 187, 1998 (`ftp://set.gmd.de/pub/vers/papers/FMSD.pdf.gz`)

[PH98] A. Poigné and L. Holenderski. On the Combination of Synchronous Languages. in:W.P.de Roever (ed), Workshop on Compositionality, Malente 1997, LNCS 1536, `ftp://set.gmd.de/pub/vers/papers/COMPOS98.ps.gz`

[sE] The *synchronousEifel* Site, `http://set.gmd.de/EES/synchronie/swb.html`