Project 22703: SYRF

Deliverable 2.1

Combination of Formalisms

---

1 Introduction

Data-flow formalisms for synchronous reactive programming, such as LUSTRE and SIGNAL, essentially compile into (definitional) equation systems with additional registers. This compilation process amounts to a great extend to a clock resolution calculus. Variables and operators in the equation system may belong to various types (while boolean, integer and floating point reals are predefined).

On the other hand imperative languages like ESTEREL (but also most graphical formalisms of STATECHART influence) produce *purely boolean* equation systems, with additional boolean variables inserted to drive memory operations on data, these operations being stored in a side *action table*. This corresponds to a separation between control and data flows.

Against this background several routes have been pursued to achieve a combination of synchronous formalisms:

1. combination on the source level by extending the data-flow model by "modes" of operation (Section 2),
2. combination on the level of an intermediate code
   (a) by cross-compiling between different existing intermediate codes, and by then linking on the level of one intermediate code (Section 3.1), and
   (b) by compiling into, and linking the same intermediate code (Section 3.2).

The achievements in '97 consist in research reports [2, 3] presenting the approaches and in prototypical implementations [1]. The implementations have been tested on the SAAB case study.

2 On Modes

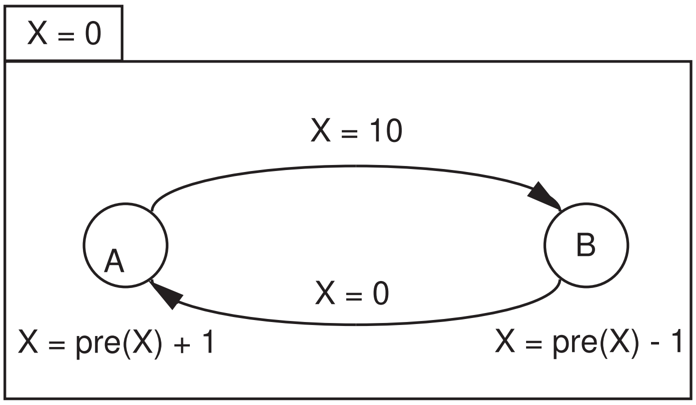
The dataflow style is particularly approprioate if the system behaviour is based on some regularity, like in signal processing.

The behaviour of a system, however, may switch between different regular or "continuous" patterns according to some external influence. We refer to such a pattern as a *running mode*. For instance, if some filter is used to dampen a rather steady movement, in extraordinary situation an additional filter needs to be added to keep some fast reaction under control, e.g. if electronically controlling the behaviour of a car suspension. Such changes of running modes are difficult to represent in the dataflow style, the resulting code is often no more readable, is difficult to modify, and error prone.

Technically, all reactive systems can be viewed as state machines ("automata") with a possibly huge set of states, and with transistions from one state to another. In such a setting each mode corresponds to a subautomaton, and a change of mode to a transition from a state of one subautomaton to a state of another subautomaton.

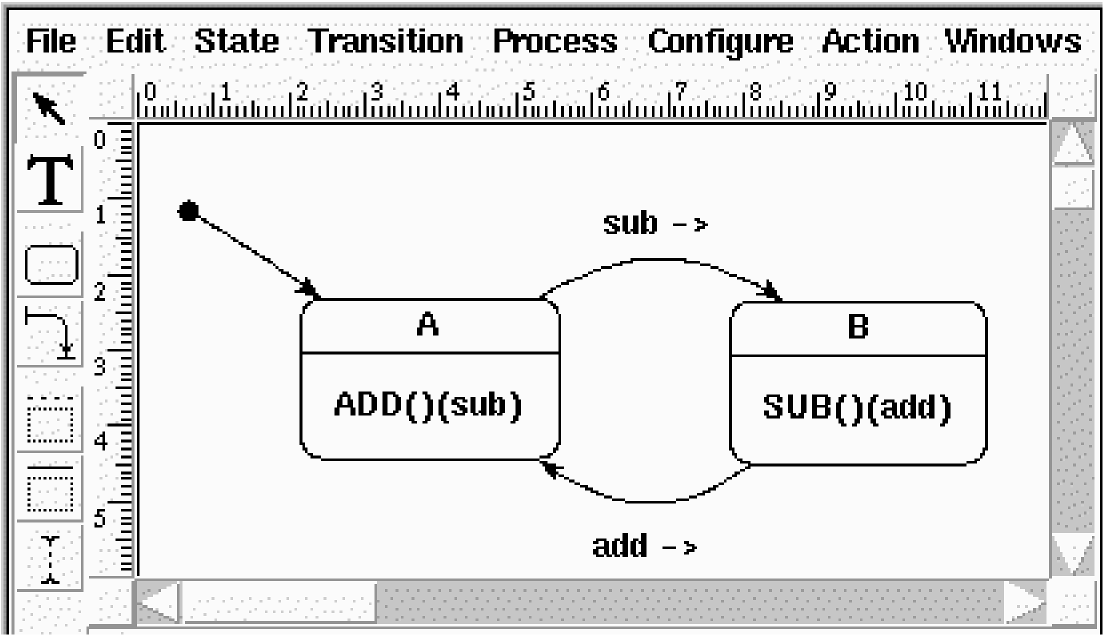
While these considerations are common to all approaches dealing with a combination of a state-based (imperative) formalism with a dataflow formalism, the approaches differ in the way modes are realised on the level of source formalism.

On one hand, one may consider modes as changing the computation scheme as in e.g.



where we start to add to the previous value of `X` if `X` becomes `0`, and where we subtract if `X` becomes `10`.

On the other hand, one may consider modes as subprocesses where all variables are (re-) initialized if started as in e.g.



where the subprocesses are specified in LUSTRE

File Edit Search Configure Action Windows

```
export ADD, SUB;

node ADD () returns (sub);
var x:int;
let
sub = pre(x) >= 10;
x = 0 -> pre(x) + 1;
tel;

node SUB() returns (add);
var x:int;
let
add = false -> x <= 0;
x = 10 -> pre(x) - 1;
tel;
```

Here, the local variable `X` is initialized to `0` resp. `10` if the processes `ADD` and `SUB` when entering the state `A` resp. `B`. The latter requires a full combination of the formalisms as explained in Section 3.

The semantics of the first solution have been investigated in [3] where *mode automata* are introduced. Mode automata translate elegantly into LUSTRE programs since the principle of uniqueness of definition for variables can be preserved: rather than to have a unique equation for each variable, uniqueness of definition is guaranteed at every instant (of time). Then all definitions can be integrated into a unique equation by case distinction with regard to the modes. The uniqueness at every instant is achieved by an automaton structure on the modes where only one state is in control at every instant. The approach avoids any complicated handling of preemption mechanisms. The paper [3] proves compositionality of this translational with regard to parallel composition. The still open question is whether the approach scales to hierarchical automata.

3 Combination of Processes

Here synchronous formalisms are conceived as defining processes which are started at a certain instant (of time), and which may be preempted at another instant. Two complimemtary lines of thought have been pursued so far:

- to use existing compilers to cross-compile into the common standard format **DC**, "declarative code", and linking on this level, and
- to base integration on a common semantical model, exploring the semantic concept needed for an orthogonal integration.

3.1 Crosscompiling

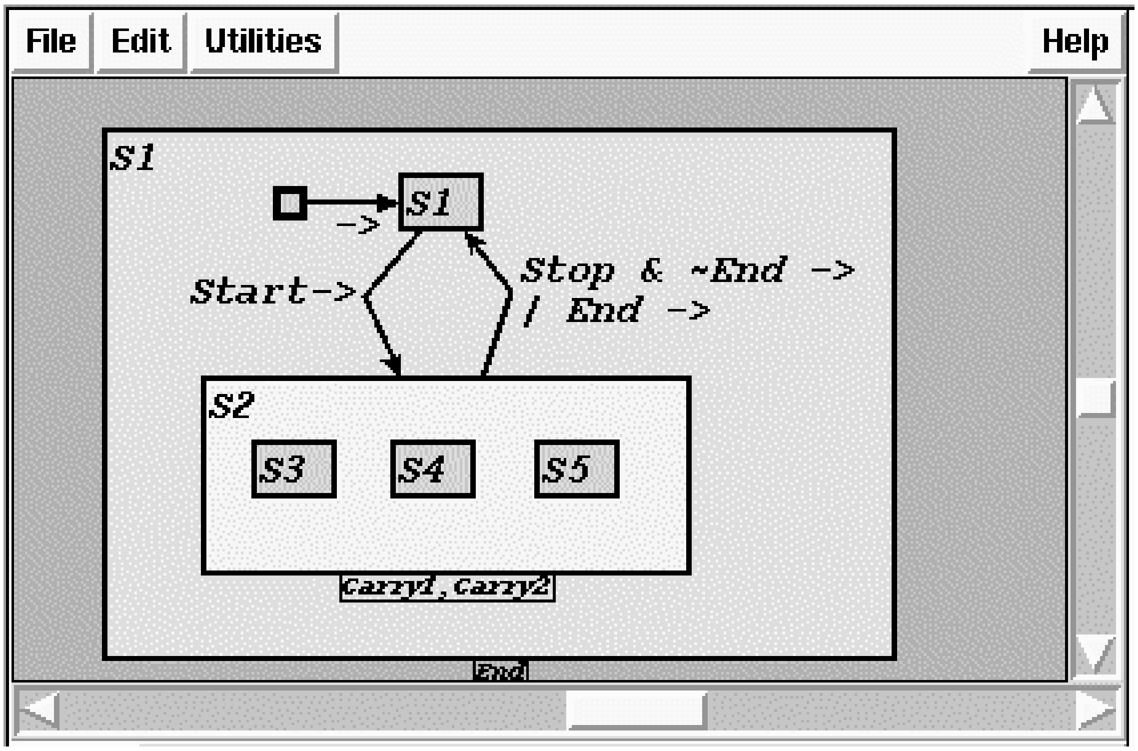
At present, a the ESTEREL compiler of INRIA (Sophia Antipolis) has been extended to generate DC-code by realizing a **SC** to **DC** translator. The translator was tested on the SAAB case study (see below) which have been specified both:

1. in an ESTEREL version, and
2. in a mixed LUSTRE/ESTEREL version: the control part of the aircraft has been implemented in ESTEREL, and the data part in LUSTRE.

Actually, the ESTEREL part has been translated to **DC**, then from **DC** to Lustre (by a "decompiler"). Linking is done by hand at LUSTRE level. The investigation aims for having structured **DC** linking at the end of the project.

3.2 Orthogonal Integration

By orthogonal integration we understand a combination of synchronous languages which allows freely to call a module of a synchronous language in another languages, languages being restricted so far to an extended version of ARGOS, and light versions of ESTEREL and LUSTRE. Integration is achieved by automatic linking on the level of of *synchronous components*. The code of synchronous components is similar to that of **DC**, though somewhat lighter, but with additional "glueing constructs" supporting the linkage. This has been prototypically implemented under the name of the SYNCHRONY WORKBENCH, SWB. At present, it provides compilers, linkers and a simple causality check based on sorting, as well as two graphical interfaces for ARGOS. One is a free-style drawing tool, a screenshot of which is displayed below



the other is syntax-directed (cf. the screenshot in Section 2). Further, a debugging/simulation environment is provided. The current version SWB-L ("light") is available via web under `http:set.gmd.de/EES/synchronie.swb.html`.

The underlying compilation schemes are described in [2] but a few specific comments may be appropriate here. One of the difficulties is that declarative languages do not have an inherent notion of *control*, once started they run "forever". If used within a control based laguage such as ESTEREL or ARGOS there must be some way to restart and to preempt such a program. Roughly, this is achieved within SWB-L by adding a "control register" to every declarative node, and by relating the clocks of the node to this control register: clocks are present if and only if the register gets control (i.e. is set to "true"), or if it is in control. Preemption of the register implies absence of all clocks, hence no computations takes place for the respective declarative node. The node is properly initialized if its control register is set to "true".

A second problem is related to the "uniqueness of definition" principle in declarartive languages. If a declarative node is imported to a control based language call-by-name parameter passing may result in unexpected behaviour, e.g. one would expect that in

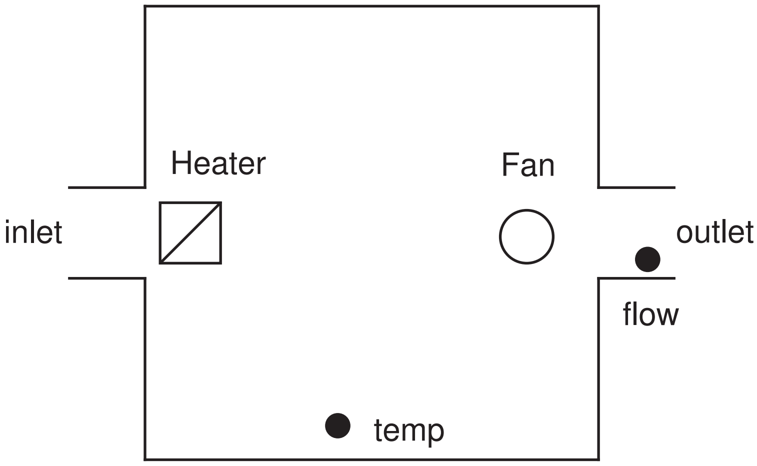
```
node FALSE () returns (z:bool);
let
   z = false -> pre(z)
tel
```

the output remains `false` for all instants. If we, however, apply the parameter passing scheme of ESTEREL we can replace `z` by an actual parameter `x` which may become `true` at some instant because of a emit statement `emit x(true)` in some other part of the code. Hence the variable `z` may become `true` in contradiction to what we expect. We resolve this problem by replacing call-by-name by call-by-value: the call `FALSE () (x)` will result in an additional assignment `x := z` where `z` becomes a local variable thus retaining "uniqueness" of definition (just to give the idea without bothering about all the details).

4 Experiences in Case Studies

SWB-L has been experimentally used to model the SAAB case study (sketched below) and for other industrial case studies not being part of the project. As a preliminary remark, the case studies demonstrate that a combination of ARGOS and LUSTRE allow for very succinct specifications (and implementations, since rerasonably efficient code is generated). Unexperienced users, however, (even with a background in synchronous languages) have some difficulty to use the abstractions in a proper way. For instance, automata and declarative code are put in parallel without taking advantage of the hierarchical structure offered by ARGOS. We clearly need a tutorial introduction which gives advice how to use the combination of formalisms properly, which demonstrates via examples how to exploit the combination, how appropriate abstractions are made, and which kind of formalism is most appropriate for particular patterns of behaviour.

The SAAB case study models a small part the control of a climatic chamber in an airplane. The system consists of a chamber with a given volume and its different parts are: inlet, outlet, heater, temperature sensor, fan for ventilation, flow sensor and a control panel. The chamber is to be ventilated via the inlet and outlet.



The purpose of the system is to keep the selected temperature and ventilation within given limits. There is a control panel with several light and sound which indicate the working mode and give warnings in case of malfunction, a main switch, and instruments to select temprature and the time for the mass flow rate.

We do not attempt to go into detail here but concentrate on the subject of combination of formalism: the system continuously interacts with the environment via the sensors and actuators. This part is best modeled by data flow code, e.g. we have some code checking for temprature conditions (the actual names being not of interest in this context)

```
node TEMP_COND (pantreq,chamtact,deltatemp:real)
returns (temp_in_scope,proper_temp:bool);
let
  temp_in_scope = ((pantreq-chamtact<=2.0*deltatemp)
                   & (chamtact-pantreq<=2.0*deltatemp));
  proper_temp = ((pantreq-chamtact<=deltatemp)
                 & (chamtact-pantreq<=deltatemp));
tel
```

Some of these continuous samplings are activated only in specific working modes. There are three such modes:

- *wait*: At start up the chamber is heated to the requested chamber temperature. Also the fan starts to work.
- *active*: The actual temperature of the chamber is to be automatically regulated by the system. There are two submodes *sol* and *work*. T
- *block*: if the regulation fails the system enters a blocking mode before switching off.

The system is in the submode *sol* at start or if the temperature requirement changes for a fixed amount of time within which the temperature must be within the required limits. The overall model has been specified in SWB-L as displayed on the next page. Here the process `CONTINUOUS_CONTROL` is actually a piece of ESTEREL code which puts in parallel all the continuous sensor/actuator routines. The timers in the *sol*-mode and the *block* mode are simple LUSTRE programs.

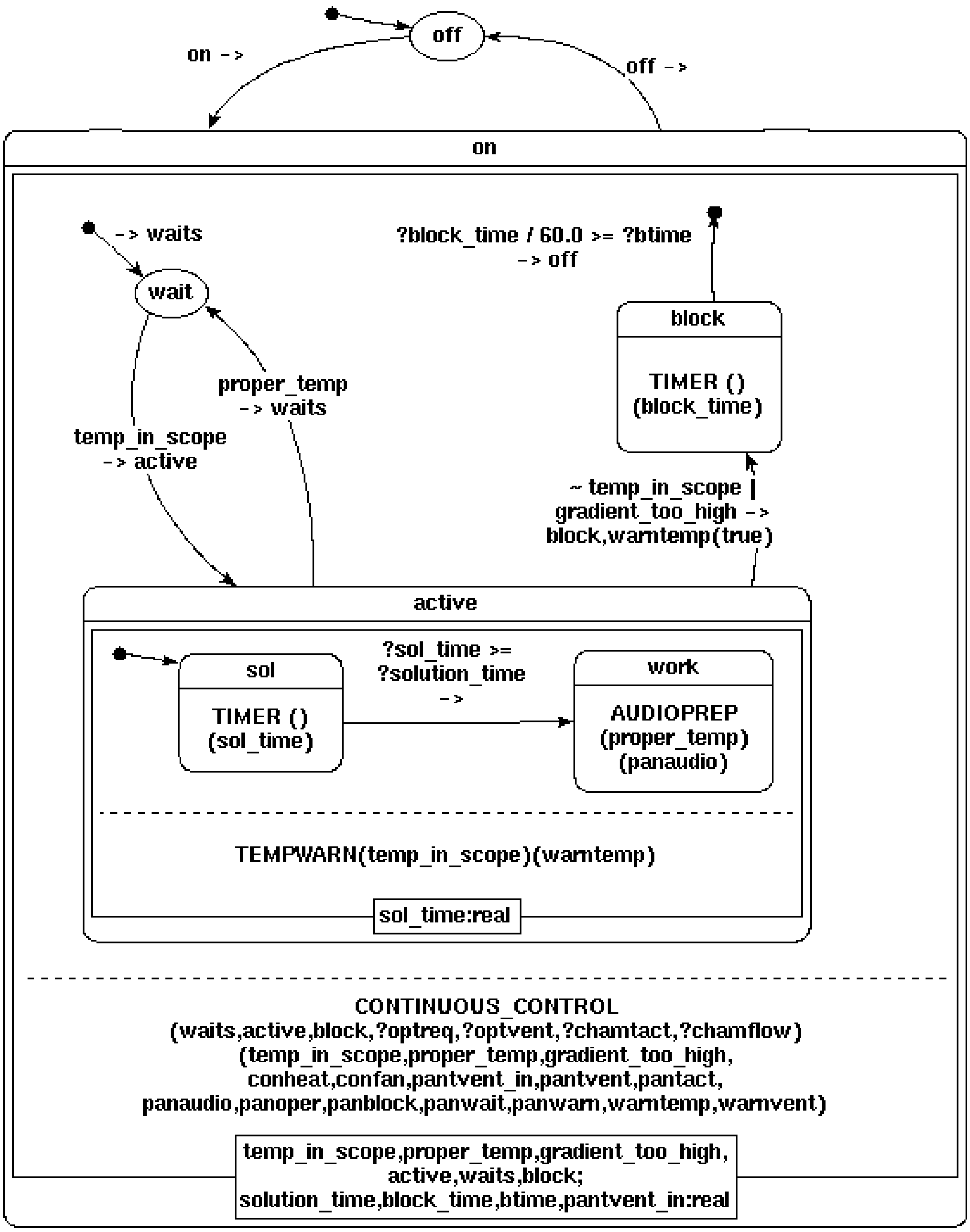
It should be remarked that the present modeling is an intermediate step which follows the original (about 50 page) specification in ASA (which is the language of an industrial tool) as closely as possible for reasons of comparison. A next step, starting from a system specification in plain English will probably result in an even simpler system.

References

[1] The SYNCHRONIE WORKBENCH, `http://set.gmd.de/EES/synchronie/swb.html`

[2] L.Holenderski, A.Poigné, The Multi-Paradigm Language *LEA*, Submitted, and L.Holenderski, Translating *LEA* to synchronous components, Arbeitspapiere der GMD, Nr. 1090, 1997

[3] F.Maraninchi, Y.Rémond, "Mode-Automata: About Modes and States in Reactive Systems", Research Report, Verimag


off
on ->
off ->
on
-> waits
wait
?block_time / 60.0 >= ?btime
-> off
block
TIMER ()
(block_time)
proper_temp
-> waits
temp_in_scope
-> active
~ temp_in_scope |
gradient_too_high ->
block,warntemp(true)
active
sol
TIMER ()
(sol_time)
?sol_time >=
?solution_time
->
work
AUDIOPREP
(proper_temp)
(panaudio)
TEMPWARN(temp_in_scope)(warntemp)
sol_time:real
CONTINUOUS_CONTROL
(waits,active,block,?optreq,?optvent,?chamtact,?chamflow)
(temp_in_scope,proper_temp,gradient_too_high,
conheat,confan,pantvent_in,pantvent,pantact,
panaudio,panoper,panblock,panwait,panwarn,warntemp,warnvent)
temp_in_scope,proper_temp,gradient_too_high,
active,waits,block;
solution_time,block_time,btime,pantvent_in:real