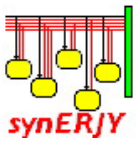


synERJY 5.1

An Introduction to the Language

Reinhard Budde
Axel Poigné
Karl-Heinz Sylla

Fraunhofer Institut

Autonome intelligente Systeme

Fraunhofer AiS

February 6, 2007

Preface

synERJY[1] is a programming language and a design environment for embedded reactive systems that combines two paradigms:

- *Object-oriented modelling* for a robust and flexible design.
- *Synchronous execution* for precise modelling of reactive behaviour.

Highlights are that

- *synERJY* provides a deep embedding of the reactive behaviour into the object-oriented data model.
- *synERJY* offers fine-grained integration of synchronous formalisms such as ESTEREL [4], LUSTRE [19], and STATECHARTS [18].[2]

The programming environment supports compilation, configuration, simulation, and testing, as well as verification by model checking. Behavioural descriptions may be edited in graphical or in textual form. Code generators for efficient and compact code in C and several hardware formats are available.

The *synERJY* language and its programming environment are developed at the Fraunhofer Institut Autonome Intelligente Systeme.[3] The programming environment is freely available with a public domain license.

Contact: {reinhard.budde,poigne,sylla}@ais.fraunhofer.de.

[1] *synERJY* may be read as **syn**chronous **e**mbedded **r**eactive **J**ava with a **y** for the sound.

[2] We recommend Halbwachs' book *Synchronous Programming of Reactive Systems* [16] as an excellent introduction.

[3] The development of *synERJY* has been partially supported by the ESPRIT LTR SYRF, "Synchronous formalisms", and the ESPRIT IIM Project CRISYS, "Critical Systems and Instrumentation".

Acknowledgements

We are grateful to all our former colleagues Agathe Merceron, Leazek Holenderski, Olivier Maffeis, Matthew Morley, Monika Müllergburg, Micele Pinna who substantially contributed to the genesis of *synERJY* (as outlined in the Appendix A.1).

Further we sincerly thank Albert Benveniste, Paul Caspi, Gerard Berry, Nicholas Halbwachs for being the "founding fathers" of synchronous programming, and for their co-operation and inspiration in the many activities we shared.

Contents

Chapter 1

Introduction

A priori. Synchronous Programming is a branch of concurrent programming. Concurrent programs are inherently more difficult to write and to debug than sequential programs. The sequential execution of each of the parallel branches is interlocked with communications between the parallel branches, resulting in a partial order according to which elementary statements are executed. Whether such a partial order is explicitly specified by the programmer (for instance using threads and synchronisations) or automatically scheduled by a compiler as in case of synchronous languages, the programmer will sometimes be surprised by unexpected behaviour. This one should keep in mind before getting despaired debugging a concurrent program.

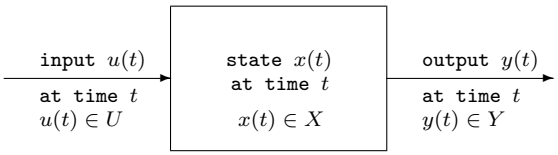
1.1 Systems, Models, and Programming

Models of physical systems. In the real world we encounter physical objects such as cars, electronic circuits, power stations, dogs, space ships, and computer programs. We refer to these as *physical systems*.

If we start to think about one of these objects we formulate a *model* of the object focusing upon some small number of properties we believe sufficient for capturing the behaviour we care about. If we wish to analyse the model we need some formal (mathematical) representation of it.

We may start by considering a system as a black box into which we feed inputs and out of which we receive outputs, inputs and outputs being properties we care about. It is quite common to refer to these properties as *signals* if we enter into the mathematical analysis of a system. In general we cannot expect the outputs to be completely determined by the inputs,

but rather to depend as well on a set of (internal) properties we refer to as a *state*. A state is some compact representation of the past activity of the system complete enough to predict, on the basis of the inputs, exactly what the output will be, and also to update the state itself.



Time scales. Given that we have decided what the sets of input, output and state variables are for our model, we must now determine on what *time scale* we shall analyse the system. For many physical systems we may think of input, output and state as changing continuously, then a model becomes a *continuous-time system*.

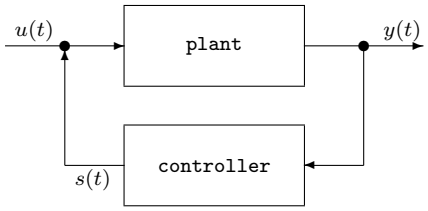
On the other hand, a continuous-time model of a computer system is inappropriate, and we treat them as a *discrete-time system*, since we can make only finitely many measurement at any time interval. Further, if we use a computer to control the behaviour in a desired way, we can generate distinct control signals only at discrete times.

Analyzing systems. Having modelled a physical system, the question arises whether we can *control* a system in a given state, providing inputs to bring it into its resting state, or, given that a system is in its resting state, can we apply inputs to which will force it to *reach* a desired state. The complimentary question is whether we can *observe* an unknown state of a system in such a way as to determine what the state was. In other words, we want to analyse the dynamic behaviour of a system. The differential and integral calculus provide the main tools for continuous-time systems, whereas algebra is the main tool for the analysis of discrete-time systems. Textbooks provide a vast body of material to deal with these questions.

All this contributes to the central task of *control theory* of controlling a system to behave in a desired way. We have to answer questions such as whether we are able to control a system at all, or further, if we are able to control a system in some optimal way, for instance spending a minimal amount of energy.

System control. Monitoring a system's performance, comparing it with some desired reference performance, and using any discrepancy to generate a correction input to control the system to approximate the reference, is

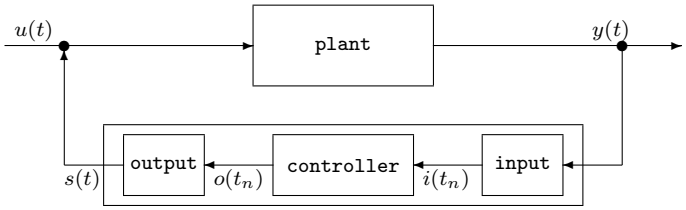
fundamental pervading all aspects of modern life. This is what *feedback control* is about



The most familiar example is that of thermostatically controlled heater. When the temperature is below a reference temperature the heater is turned on until the thermometer indicates that the temperature is higher than the reference point. Then the heater is turned off. In this system the heater is the *(dynamic) plant*. The *controller* converts performance data (the temperature) in a *control signal* (on-off switch). The feedback is here used to *regulate* the system's performance. There are other applications of feedback or *closed loop control*, for instance to improve the characteristics of a system making it less sensitive to changes to enhance stability. Not every control system, of course, uses feedback. Consider a light that is switched on at dusk and switched off in full daylight.

The task of the engineer is to design and implement the control system – i.e. to model the plant, to design an adequate controller, to analyze the overall system, and, finally, to "implement" the controller which today mostly means to run a computer program on a suitable piece of hardware, let it be a work station, a micro-controller, or some programmable hardware. One usually refers to such a combination of software and hardware as an *embedded system*.

Close-up. Embedded systems are usually comprised of three components: *input* and *output converters*, and a *digital controller* refining the diagram above



Obviously, the digital controller is a discrete-time system. The nature of the converters depend on whether the plant is continuous-time or discrete-time.

In the latter case, no converters may be needed. Otherwise inputs must be *sampled*, using A/D- and D/A-converters for example. The sampling may be periodic with a fixed length interval – being determined by some "clock cycle" or may depend on the input itself – in that a specific trigger signal is generated, for instance if the input changes substantially.

1.2 Synchronous Programming

Synchronous programming is just a way of specifying discrete-time (control) systems. Many formalisms do the same. However, synchrony as a paradigm avoids ad-hoc solutions and offers a mathematically precise and, as we do believe, also a simple programming model. Synchronous programming distinguishes between languages based on control flow and those based on data flow. Programmers seem to prefer the first, while engineers seem to prefer the latter.

The synchrony hypothesis. Synchronous behaviour is modeled as a machine that, on actuation, reads the input signals and is then *decoupled from the environment* to compute the subsequent *state* and the value of the output signals, dispatches the output signals to the environment, and waits for the next actuation. Such an execution step is called an *instant*.

Execution, of course, takes time. The delay does not cause harm as long as it is sufficiently small in that an instant always terminates before the next actuation takes place. Then we may use the hypothesis that input is sampled and output is generated "at the same time". The reaction appears to be "instantaneous" with regard to the observable time scale in terms of activations. Berry [2] speaks of the *synchrony hypothesis*. Since time cannot be measured "in between" activations one sometimes speaks of a "zero time model".

Whether or not the synchrony hypothesis holds needs checking in each particular case. Usually deadlines are set by real-time constraints: even in the worst case the execution of an instant must terminate before the deadline expires. Worst-case execution times depend on the particular implementation of a control system, while the real-time constraint depends on the system to be controlled. Thus system design is a compromise between system requirements in terms of real-time constraint and computation power offered by the implementation.

The benefits of synchronous systems are at hand. They are much easier to design, reason about, test, and implement if a computation proceeds in

discrete steps since, for instance, inductive arguments may be used. Further, the system behaves the same at whatever speed it is executed (up to the limit set by the worst case execution time). This property is particularly useful for purposes of testing: programs can be tested in an simulation environment on a slow time scale while running it operationally much faster having exactly the same behaviour.

Broadcast and determinism. Signals are broadcast in synchronous programming, i.e. at every instant there is a system-wide consistent view of all signals. In particular

- the value of a signal is never updated at an instant after the value has been read once (*write-before-read* strategy).

This substantially constrains scheduling as we will see below.

Synchronous programming additionally requires that

- synchronous execution is *deterministic*.

The subsequent state and the output signals must be uniquely determined by the input signals and the present state. Testing deterministic systems is substantially simpler than testing non-deterministic systems since system behaviour can be reproduced.

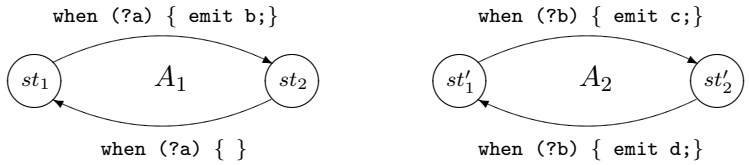
Synchronous languages nevertheless support concurrency even though it is a common belief that concurrency may generate non-determinism. But in the synchronous case non-determinism is resolved by the compiler. The elementary actions are scheduled according to the control flow with consistency as an additional constraint (cf. Section 1.2). If no deterministic scheduling can be found, either due to a *time race* (cf. Section 2.2) or due to a *causality cycle* (cf. Section 2.5), an error message is issued and the program is rejected.

Synchronous programming based on control flow. Imperative programming languages or finite state machines are typical formalisms for specifying the control flow. ESTEREL[4] is a prominent example for an imperative synchronous language as are STATECHARTS[18] (or SYNCCHARTS[1]) for state based formalisms.

All these languages share the idea that communication is based on *signals*. A signal may be emitted at an instant. Then it is *present*. Otherwise it is *absent*. This property is called *coherence* in [2]: there must be an explicit cause for a signal to be present, either it is emitted by the system at

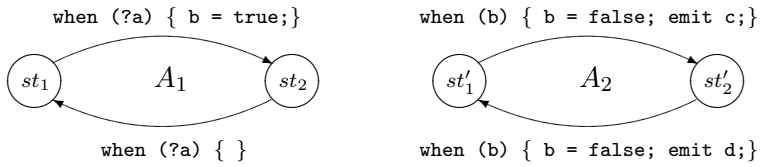
an instant, or it is present as an input. One may check for the presence of a signal.

Consider, for example, the following two simple automata that are supposed to run in parallel



The expression `?a` asks whether the signal `a` is present. Hence, if the automata are in state st_1 and state st'_1 and if the signal `a` is present the automaton A_1 moves from state st_1 to state st_2 emitting `b`. In parallel the automaton A_2 checks for the presence of `b` provided it is in state st'_1. Since `b` is present the A_2 moves from state st'_1 to state st'_2 emitting `c`. In that the signal `b` is part of the control flow.

At a first glance, the same effect may be achieved if `b` would be an ordinary Boolean variable which is set to true when changing state from st_1 to st_2, and which is checked by the second automaton (note that the variable `b` must be reset to false to emulate signal behaviour).



But then the behaviour of the two automata may be non-deterministic. If the value of `b` is checked before state is changed from st_1 to st_2, automaton A_2 would remain in state st'_1. Only the write-before-read strategy imposed on signals resolves the non-determinism: the emission must take place before reading the status of a signal. Obviously, it would be too restrictive if the write-before-read strategy would be imposed on all variables.

Synchronous programming based on data flow. LUSTRE[19] and SIGNAL[15] are synchronous programming languages based on data flow. These languages are inspired by difference equations such as

$$\begin{aligned} z(n) &= 0.5 * z(n-1) + 0.3 * y(n) + 0.2 * x1(n) \\ y(n) &= 0.5 * x1(n) + 0.5 * x2(n) \end{aligned}$$

Each of the equations is evaluated at time n to compute a new value of the respective variable. $x1$ and $x2$ are assumed to be inputs.

For difference equations, the order of evaluation is determined by the variables only (it does not depend on how equations are ordered when written down): the second equation must always be computed before the first one since the value $y(n)$ is used to compute the value of $z(n)$. The evaluation strategy is based on the "flow of data", and, by the way, exactly coincides with the write-before-read strategy of synchronous control flow as described in an earlier section.

It just takes a minor shift to rewrite the difference equations as *data flow equations*. Consider each of the variables as to refer to an indexed sequence of data, a *data flow*, and we assume that operations like addition and multiplication are defined point wise. Then we may abstract from the index by writing

$$y \quad = \quad 0.5 * x1 + 0.5 * x2$$

instead of

$$y(n) \quad = \quad 0.5 * x1(n) + 0.5 * x2(n)$$

For the other equation, we need what is called a *time shift operator*

$$pre(x)(n) \quad = \quad x(n-1)$$

to obtain

$$z \quad = \quad 0.5 * pre(z) + 0.3 * y + 0.2 * x1$$

Standard notation in control theory is z^{-1}. We here use the LUSTRE operator `pre`.

Hybrid systems. Originally the notion of *hybrid systems* refers to a combination of discrete-time and continuous-time behaviour. Typically this involves discrete transitions between continuous "modes of behaviour". In reference to the latter, we reinterpret hybrid systems as combining control flow with data flow: data flow typically models *periodic* behaviour while control flow typically models *sporadic* behaviour. A sporadic transition is used to switch from one periodic behaviour to another one.

It is a matter of background and preferences which models are used in which context. It seems that computer programmers prefer control-oriented languages as state machines are, while engineers prefer data flow languages but sometimes appreciate state machines as well.

synERJY smoothly integrates the different formalisms of synchronous programming. We believe that these should coexist in a language since, depending on the problem, one or the other formalisms tends to be more

appropriate. Our integration is fine-grained in that the formalisms are integrated on the level of statements offering a not yet preceded versatility.

Though we aim for a smooth interface, the complexity of interaction may be considerable. One should keep in mind that mastering one formalism (and its inherent paradigms of modeling) usually takes some effort. Mastering three such paradigms, even though familiar ones, takes a greater effort, as does the adequate combination of the paradigms in solving a particular problem. Nevertheless there should be some benefit at the end: the art of system design depends on using appropriate abstractions and paradigms. This is what we hope to support.

1.3 Outline of the Book

This textbook intends to introduce to the concepts of synchronous programming in general, and of *synERJY* in particular, at a reasonable level of detail. Note that several sections and paragraphs are *set in italics*. We recommend to skip these parts at a first reading since they treat somewhat more advanced concepts.

Chapter 2 presents the core ideas of synchronous programming in an imperative programming style close to that of ESTEREL that seems to appeal to computer programmers.

Chapter 3 consists of a number of examples.

Chapter 4 introduces hierarchical state machines that are particularly appreciated because of the visual presentation. Our approach is a mixture of that of STATECHARTS and SYNCCHARTS [1].

Engineering disciplines often base their models on differential or difference equations, "data flow" models in terms of computer science. Chapter 5 presents a data flow language closely related to LUSTRE.

Chapter 6 changes the focus relating reactive behaviour with object-oriented structuring principles. Objects are classified as *reactive* if some synchronous code is embedded. Reactive objects communicate via signals while being evaluated concurrently. All methods of a reactive object are private by demand. In that reactive objects behave like components that behave in the same way independently of the context.

Chapter 7 explains how to build and deploy applications that interact with the environment.

Many of the chapters are reasonably self-contained, hence can be read in any order. However, one should to start with Chapter 2 to get a grasp of the basics. If one is happy to use the simulator only for experiments, one may

continue with any of the Chapters 3 to 6. If one wants to build applications for some target platform, reading Chapter 7 is recommend.

A separate reference manual [39] and user guide for the programming environment [40] complements this exposition.

Remark. The documentation is complemented by a directory in which all the examples of the book are provided. These are referenced within the text: *(cf. chpt_2_delayed-emit.se)* refers to a file with the same name in the examples directory.

Chapter 2

The Reactive Core

2.1 Reactive Classes, Sensors, and Signals

Reactive classes. *synERJY* is comprised of a subset of JAVA™ extended by reactive classes.

- A class is *reactive* if it has only one constructor, and if that constructor has a tail of the form

  `active { ... },`

  Such an constructor is called an *reactive constructor*.

The operator `active` embeds the synchronous reactive code. This code is executed at every instant. A simple reactive class is

```
class PureSignals {

    Sensor   sensor = new Sensor(new SimInput());
    Signal actuator = new Signal(new SimOutput());

    public PureSignals() {
        active {
            if (?sensor) { emit actuator; };
        };
    };
}
```

Sensors and signals. Reactive objects communicate by *sensors* and *signals*. Sensors are read only whereas signals may be updated by the program. Sensors may be updated by the environment.

Both sensors and signals may be *present* or *absent*. A sensor or a signal is present at an instant if and only if it updated at the same instant. Otherwise it is absent. In the example above, there is one sensor `sensor` and one signal `actuator`. The reactive statement `if (?sensor) { emit actuator; };` checks for the presence of the signal `sensor`. If `sensor` is present the signal `actuator` is emitted.

The type `Sensor` indicates that `sensor` is a sensor while type `Signal` indicates that `actuator` is a signal. The objects of type `SimInput` and `SimOutput` interface the sensor resp. signal with the simulator. They will be discussed in Section 2.9.

The sensor `sensor` and the signal `actuator` are particular in that they do not carry for a value. We speak of *pure* sensors and *pure* signals in contrast to *valued* sensors and signals as in the fragment

```
class ValuedSignals {

  Sensor<int> sensor   = new Sensor<int>(new SimInput());
  Signal<int> actuator = new Signal<int>(new SimOutput());

  public ValuedSignals() {
     active {
       if (?sensor) { emit actuator($sensor + 1); };
     };
  };
}
```

Again we have a sensor and a signal. If the sensor `sensor` is present, the signal `actuator` is emitted with a new value that is obtained by increasing the value `$sensor` of the sensor `actuator`. Sensors and signal have a default value according to type, e.g. `0` for `int`.

Sensor and signal types. Sensor and signal types are a new kind of built-in reference types. Sensors have a type of the form

`Sensor<`T`>`

where T is a primitive or class type. Operators related to a sensors are

`?`s **:** checks whether the sensor s is `present` or *absent*. The result is of type Boolean.

`$`s `:` yields the *value* of the sensor s. The result is of type T.

Signals have a type of the form

`Signal<`T`>`.

`Signal<`T`>` is a subtype of `Sensor<`T`>` in that the `Sensor` operators are inherited. Signals are updated using the statement

`emit` $s(v)$ `:` The signal s is emitted to be present at an instant, and the value of s is updated to be v.

Sensors and signals have a default value depending on the type, e.g. `0` in case of `int`. This default applies if a sensor or signal has not been present yet.

Pure sensors and signals will be considered as being valued of null type. They have the only value `null`. We use

`Sensor` *resp.* `Signal`

as types for pure sensors and signals. The emit statement then is of the form `emit` s.

2.2 Reactive Control

Processes for semantics. The semantics of reactive statements is introduced inductively. For every reactive statement `P`, we define a semantic interpretation P. We refer to this entity as a *synchronous process* or *process* for short.[1] We may understand a synchronous process naively as some piece of executable code that behaves as required by the synchrony hypothesis. Properties of synchronous processes relevant for the subsequent discussion are that

- a process may be *active* or *inactive*,
- state is changed and output is generated only if a process is active.
- a process may be (*re*) *started* to turn active, and may *terminate* to turn inactive.

[1] We are aware that the term 'process' is heavily overloaded but use it for lack of a better alternative.

A process is called *instantaneous* if it terminates at the same instant it is started, otherwise it is active for more than one instant.

Since every statement will define a unique process we rather ambiguously speak of statements as processes mixing syntactic and semantic level, i.e. a syntactic statement such as `emit` s ambivalently denotes the corresponding process. This overloading avoids notational contortions such as $P_{\texttt{emit}s}$ to distinguish syntax and semantics. This applies as well to an operator such as `loop` { P }; it denote the syntactical construction (if P is considered as a piece of code) as well as the corresponding semantical operator on processes (if P is considered as a process).

Processes are defined inductively according to the syntactic structure of statements.

Elementary reactive statements. The language *synERJY* has two elementary reactive statements/processes: when started, the process

```
emit s;
```

(alternatively s`.emit()`) emits the signal s and terminates at the same instant. If s is a valued signal, the statement is

```
emit s(v);
```

(alternatively s`.emit(`v`)`) where v is a value of appropriate type.

The other process is

```
next;
```

Once started the process terminates only in the next instant. These processes are distinguished in that

- emitting is *instantaneous*, i.e. the process terminates at the same instant it is started.
- the process `next` consumes time, i.e. it starts at one instant but keeps control till the next instant.

Sequential composition. Composite reactive statements are the sequential and parallel composition, the conditional and the loop statement. The statements are familiar but the semantics may be less.

The ***sequential composition*** is given by a sequence of statements[2]

[2]Note that ";" is a terminator of statements as in C, i.e. ";" is part of P_1 and P_2.

$P_1 \;\; P_2$

Once the composite process is started, the subprocess P_1 is started. When P_1 terminates, the subprocess P_2 is started in the same instant. Hence, for

```
emit a;
emit b;
```

both signals, `a` and `b` are emitted at the same instant. This is very different to

```
emit a;
next;
emit b;
```

Once this process is started, the subprocess `emit a` is started, `a` is emitted, the subprocess terminates instantaneously, and the subprocess `next` is started. The latter keeps control till the next instant, i.e. it terminates at the beginning of the next instant. Only then the subprocess `emit b` is started. Hence `a` is emitted in the first instant and `b` in the second instant.

Traces. It is often convenient to use traces to illustrate the reactive behaviour. *Traces* are sequences of values variables or signals have at an instant. For instance,

```
emit x(1);
emit y(2);
```

has the following traces

i	*0*	*1*	*2*	*3*	*4*	...
x	**1**	*1*	*1*	*1*	*1*	...
y	**2**	*2*	*2*	*2*	*2*	...

Both `x` and `y` are present at the first instant, and then never again (the index i specifies the instants). We use a bold typeface to indicate that a signal is present, and italic to indicate that the value can be read only. To give another example, consider

```
emit x(1);
next;
emit y(2);
```

The behaviour is specified by the trace diagram

i	*0*	*1*	*2*	*3*	*4*	...
x	**1**	*1*	*1*	*1*	*1*	...
y	*0*	**2**	*2*	*2*	*2*	...

The signal `x` is present in the first instant, and its value set to 1. The signal `y` is present only in the second instant with its value being set to 2 In the first instant it has a default value (0 for integers).

For pure signals, only presence[3] is relevant. Hence we modify notation and indicate presence by an asterisk `*` and absence by a dot, as in

i	*0*	*1*	*2*	*3*	*4*	...
a	*	.	.	.	.	...
b	.	*	.	.	.	...

This trace diagram reflects the behaviour of

```
emit a;
next;
emit b;
```

Conditional. The ***conditional*** has the format

`if (`C`) {` P_1 `} else {` P_2 `};`

It behaves as to be expected. Once started, it evaluates the condition C and then, at the same instant, starts – according to the result – either P_1 or P_2. It terminates whenever either P_1 or P_2 terminate. The expression C must be Boolean.

Note that a conditional may terminate at different instants, depending on which branch has been chosen. Consider

```
if (?a) {
   emit b;
} else {
   next;
   emit c;
};
emit d;
```

In case that the signal `a` is present, the conditional terminates at the same instant when started. Hence the signals `b` and `d` are emitted when `a` is present.

[3] and not the value that is always `null`.

i	*0*	*1*	*2*	*3*	*4*	...
a	*	.	.	.	.	...
b	*	.	.	.	.	...
c	.	.	.	.	.	...
d	*	.	.	.	.	...

In contrast, if `a` is absent the signal `c` is emitted at the same instant, but `d` only at the next instant.

i	*0*	*1*	*2*	*3*	*4*	...
a	.	.	.	.	.	...
b	.	.	.	.	.	...
c	.	*	.	.	.	...
d	.	.	*	.	.	...

The loop. Finally, when started, the ***loop***

```
loop { P };
```

starts the body P. When P terminates, P is restarted *at the same instant.* A loop runs forever. It is required that the body P is *not* instantaneous, i.e. the body should not terminate in the same instant it starts. Otherwise, a infinite sequence of computations must be executed at an instant, which is not acceptable, of course.

A small example

```
loop {
   if (?a) {
      emit b;
   } else {
      emit c;
   };
   next;
   emit a;
};
```

may illustrate the behaviour:

i	*0*	*1*	*2*	*3*	*4*	...
a	.	*	*	*	*	...
b	.	*	*	*	*	...
c	*	.	.	.	.	...

In the first instant, `a` is absent, and `c` is emitted since the condition evaluates to false. At the next instant, the signal `a` is emitted, the body of the loop terminates, and the loop is restarted. Since `a` is present now, `b` is emitted. This behaviour persists forever.

Parallel composition. When the ***parallel composition***

`[[` P_1 `||` P_2 `]];`

is started, both P_1 and P_2 are started. The parallel composition terminates either when both P_1 and P_2 terminate in the same instant or when one process terminates and the other process has terminated earlier.

Here is an example where both the subprocesses terminate at the same instant

```
[[ next;
   emit a;
|| next;
   emit b;
]];
emit c;
```

The corresponding trace is

i	*0*	*1*	*2*	*3*	*4*	...
a	.	*	.	.	.	...
b	.	*	.	.	.	...
c	.	*	.	.	.	...

Both the subprocesses terminate at the same instant namely when the signals `a` and `b` are emitted.

The subprocesses in

```
[[ emit a;
|| next;
   emit b;
]];
emit c;
```

terminate at different instants: `emit a` instantaneously, whereas the other terminates at the next instant:

i	*0*	*1*	*2*	*3*	*4*	...
a	*	.	.	.	.	...
b	.	*	.	.	.	...
c	.	*	.	.	.	...

A parallel process never terminates if one of its subprocesses never terminates.

```
[[ emit a;
|| loop {
     emit b;
     next;
   };
]];
emit c;
```

The corresponding trace is

i	*0*	*1*	*2*	*3*	*4*	...
a	*	.	.	.	.	...
b	*	*	*	*	*	...
c	.	.	.	.	.	...

Wavefront computation and causality. The synchrony hypothesis demands that every signal has a unique status at an instant. This has some impact on the scheduling of computations. Let us consider a simple example

```
[[ if (?a) { emit b; }; // (1)
|| emit a;              // (2)
|| if (?a) { emit c; }; // (3)
]];
```

We assume that, when starting the process, the status of each signal is unknown. Because of the top-level parallel composition all the three subprocesses are started. Each of the conditionals instantaneously checks its condition. Since the status of the signals is not known the conditionals are frozen till sufficient information is available to evaluate the condition. Process (2) emits the signal `a` and terminates. The evaluation of the frozen conditionals is resumed since more information about the status of signals has been accumulated. The conditions (1) and (3) now evaluate to true, and the signals `b` and `c` are emitted. The conditionals terminate, hence the parallel composition as well.

This evaluation strategy is a consequence of the coherence condition (cf. Section 1.2).

A (sub-) process is suspended if required information about the status of signals is not available, and it resumes computation if the information is available. In that more information is gradually accumulated. If there are only suspended processes, we can assume that a signal, the status of which is

still unknown, can only be absent, since, by coherence, all sensors or signals that are not emitted yet or not present as input must be absent. With this additional information the wavefront computation is resumed. For instance, let us assume that, when starting

```
[[ if (?a) { emit b; };
|| if (?a) { emit c; };
]];
```

the status of `a` is still unknown. The both the conditionals are suspended. Now we conclude by coherence that `a` must be absent. Using this the conditionals evaluate to false, and the parallel process terminates instantaneously. Roughly, this is the *constructive semantics* of Berry [3].

This evaluation strategy has a – maybe – unforeseen consequence. Consider the statement

```
if (?a) { emit b; } else { emit a; };
```

If the status of `a` is unknown, the process is suspended till we can assume that `a` is absent. The condition is evaluated and `a` is emitted. Hence `a` becomes present – in contradiction to the assumption we made earlier. We violate the coherence condition in that the signal `a` is considered present and absent at the same instant. We speak of a *causality cycle* since presence of a signal depends on its own status.

For good reasons we use a more restrictive interpretation of causality:

- Once the status of a signal has been used, the status cannot be changed any more in the same instant.

According to this strategy

```
emit a;
if (?a) { emit b; } else { emit a; };
```

is rejected since, after checking the presence of `a` in the condition, `a` may be emitted. In contrast, the constructive semantics would see that `a` is present before evaluating the conditional, hence would emit `a`.

Our strategy, we tag as "write-before-read", strictly depends on the syntactical structure. Any causality cycle in any sub-statement will raise a causality error.[4]

Valued signals are another source for causality cycles as in

[4]Whether this strategy is too restrictive or not is a matter of discussion. It has the merit of being reasonably transparent while the constructive semantics may accept programs for which it is sometimes hard to grasp why there no causality cycle .

```
emit a($a + 1);
```

the value of `a` is read before it is written (emitted). Note that

```
emit a($a);
```

leads to a causality cycle as well; though the value of `a` is consistently the same, we have to read the value before we write it.

The analysis of causality is computationally expensive. The control and signal flow as well as the user specified precedence relations generate a partial order. All the reactive statements are *scheduled at compile time* according to this partial order relation thus guaranteeing deterministic evaluation of all programs accepted by the compiler. The compiler will reject programs that have a causal cycle raising an error message that indicates which parts of the program are involved in the cycle.

Weak and strong preemption. Apart from parallel composition and the infinitary loop synchronous programs are pretty conventional so far. The real power of synchronous programming stems from the preemption operators. Preemption means that execution of some process – an infinitary loop for instance – may be cancelled under some condition to start some other process instantaneously. We say that a process is *preempted*. There are two operators for preemption of a process.

The operator for *weak preemption*

`cancel` P `when (`C`);`

cancels the process P if the condition C becomes true, but only for the *next* instant, that is: evaluation of the condition C is scheduled after that of the process P. Hence P may influence the result of evaluating C.

The operator for *strong preemption*

`cancel strongly` P `when (`C`);`

cancels the process P at the *same* instant in which the condition C becomes true, that is: the evaluation of the condition C is scheduled before that of the process P. Hence P cannot influence the result of evaluating C.

A trivial example demonstrates the difference. If `b` is present in

```
cancel {
     emit a;
} when (?b);
emit c;
```

the cancel statement terminates, but `a` is emitted before:

a	$*$	.	.	.	.	...
b	$*$	.	.	.	.	...
c	$*$	.	.	.	.	...

In contrast, `a` is not emitted any more in

```
cancel strongly {
     emit a;
} when (?b);
emit c;
```

when `b` is present:

a	.	.	.	.	.	...
b	$*$	.	.	.	.	...
c	$*$	.	.	.	.	...

Weak preemption allows to preempt, for instance, a loop from the inside:

```
cancel {
    loop {
        emit a;
        next;
    };
} when (?a);
```

Since the `emit a` is executed before `?a` the loop is preempted instantaneously if started. Strong preemption causes a causality cycle here; since evaluation of the condition `?a` is scheduled before that of the loop there is a potential write before read of the signal `a`.

The general scheme for preemption operators is this:

```
cancel [ strongly ]
    P
when (C1) [ { P1 } ]
[ else when (C2) [ { P2 } ]
  ...
  else when (Cn) [ { Pn } ]
];
```

The conditions are executed in the obvious order. If the condition C_i holds, the process P_i is executed. The clauses in square brackets are optional.

The await statement. The statement `await` C; is a shorthand for

```
cancel {
   loop { next; };
} when (C);
```

The process waits, possibly for several instants, till the condition C becomes true. This obviously is a useful and often used construct – for instance when waiting for signal to become present (`await ?a`). Note that the await statement may terminate immediately when started.[5]

The `await` operator is often used when simulating automata-like behaviour as in

```
loop {
    next;      //  state 1
    await (?a);
    emit b;
    next;      //  state 2
    await (?c);
    emit d;
};
```

This encodes two "states"; if in the state 1 ,and if `a` is present, a "transition" from state 1 to state 2 takes place and the signal `b` is emitted. Similarly, a "transition" from state 2 to state 1 takes place if `a` is present, emitting `d`. A possible trace is

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	...
a	.	.	*	.	.	*	.	.	*	.	...
b	.	.	*	.	.	.	.	.	*	.	...
c	.	.	.	.	.	*	.	*	.	.	...
c	.	.	.	.	.	*	.	.	.	.	...

Note that the presence of `a` does not affect the behaviour when in "state" 2, resp. presence of `c` when in "state" 1.

Activation. Sometimes it is convenient to activate a process only if some condition holds. Then the *activation statement*

```
activate P when (C);
```

[5]The operator `await` in *synERJY* is equivalent to the operator `await immediate` of ESTEREL.

should be used where P being a process, and C being a Boolean expression. If started the process waits until the condition C becomes true. Then P is started and executes at every instant the condition C is true. Activation down-samples a process with a frequency specified by C.

In the example

```
activate {
  loop {
    emit a;
    next;
  };
} when (?b);
```

the loop process executes only if the signal `b` is present. A possible trace is

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	...	
a	.	.	*	*	.	*	*	*	*	.	...
b	.	.	*	*	.	*	*	*	*	.	...

Sustaining. Another convenient operator is sustaining an instantaneous process

```
sustain { P };
```

When started the the sustain statement executes at each instant the instantaneous process P (that is, P terminates at the instant it is started. The statement never terminates. The statement is an abbreviation of

```
loop {
   P
   next;
};
```

but is often handy, for instance to emit a signal continually

```
sustain { emit s; };
```

Delaying reactions. *We often distinguish between what happens at the first instant when a process P is started, and what happens at later instants. We use P_α to refer to the behaviour of P at the first instant, and P_β to refer to the behaviour at later instants. Given these preliminaries, a variation of the preemption and the activation operator can be defined that are useful at times, namely that either are not applied at the first, but only at later instants. These "delayed operators" indicated by a* `next` *are defined by*

```
cancel P next when (C);     ≅  Pα; cancel Pβ when (C);
activate P next when (C);   ≅  Pα; activate Pβ when (C);
```

We consider an example. Let $P =$ `loop {next;};`*. The* `next` *is executed at the first instant, and the loop is iterated at later instant, i.e.* $P_\alpha =$ `next` *and* $P_\beta =$ `loop {next;};`*. Thus*

```
cancel loop {next;}; next when (C);
    ≅ next; cancel loop {next;}; when (C);
```

Hence the process waits at least for one instant and then terminates if the conditions C *becomes true. We introduce the shorthand* `await next` C`;` *for this process. Note that this process is equivalent to* `next;await` C`;` *(though* `await next` C`;` *has a more efficient implementation).*

The ***`await next`*** *operator is particularly useful when simulating automata-like behaviour.*

```
loop {
    await next ?a;
    emit b;
    await next ?c;
    emit d;
};
```

is a somewhat more intuitive (and efficient) alternative for the example considered two paragraphs above.

The general scheme for delayed preemption is[6]

```
cancel [strongly]
    P
next when C1 [ { P1 } ]
   [ else when C2 { P2 }
     ...
     else when Cn { Pn };
   ]
```

A small example might highlight the differences

```
cancel strongly next {
    loop {
       emit a;
       next;
    };
} when (?b);
emit c;
```

[6]The construct `cancel strongly` P `next when` C is known as `do` P `watching` C in ESTEREL.

The signal `a` *is emitted when the process is started even if the signal* `b` *is present since preemption does not apply in the first instant, as shown in the trace*

	1	*2*	*3*	*4*	*5*	*6*	...
a	*	*	*	.	.	.	...
b	*	.	.	*	.	.	...
c	.	.	.	*	.	.	...

2.3 Local Signals and Reincarnation

Local signals. Local variables, in particular local signals, may be declared within a block, i.e. within a context of the form { ...}. The scope of a local variable declaration in a block is the rest of the block in which the declaration appears, starting with its own declaration. Local signals declarations are final, i.e. must be of the form

`Signal<`T`>` $identifier$ `= new Signal<`T`>();`
`Signal` $identifier$ `= new Signal();`

Due to this restriction, initialisation may be dropped for notational convenience, i.e. `Signal<`T`>` $identifier$ resp. `Signal` $identifier$ is sufficient to declare a local signal.

Reincarnation. Local declarations may result in a behaviour that may be unexpected. Consider

```
loop {
   Signal x = new Signal();
   if (?x) {
      emit a;
   } else {
      emit b;
   };
   next;
   emit x;
   if (?x) { emit c; };
};
```

(cf. `chpt_2_reincarnation.se`) If the loop is entered, a new incarnation x' of the local signal `x` is created. Next the conditonal is executed. Since `x` has not been emitted, the signal `a` is emitted. In the next instant `x` is emitted, the actual incarnation of `x` being x'. Hence x' is present, and `c` is emitted.

The loop terminates, and the incarnation x' is not accessible any more. The loop is restarted, a new incarnation x'' of `x` is created, and so on. The trace diagram illustrate this behaviour

	1	*2*	*3*	*4*	...
a	.	.	.	.	...
b	*	*	*	*	...
c	.	*	*	*	...
x'	.	*			...
x''		.	*		...
x'''			.	*	...
.....					

We note that two incarnations coexist *at the same instant*, one being present, the other not. This is a consequence of synchrony: restarting the loop is instantaneous, only the `next` statement "consumes time".

We observe further that incarnations of local signals have a restricted lifetime. In case of the example, each incarnation is accessible only for two instants: when it is created when (re-) entering a loop cycle, and when the respective cycle terminates. In the trace diagram, the lack of an entry indicates that a signal/incarnation is inaccessible.

Multiple evaluation of statements. Statements may even be evaluated more than once at one instant:

```
loop {
  cancel next {
     Signal x = new Signal();
     [[ loop {
           cancel next {
              Signal y = new Signal();
              [[ loop {
                    if (?x) {
                        if (?y) { emit a; } else { emit b; };
                    } else {
                        if (?y) { emit c; } else { emit d; };
                    };
                    next;
                 };
              || next;
                 emit y;
              ]];
           } when (true);
```

```
        };
      || next;
         emit x;
    ]];
  } when (true);
};
```

(cf. `chpt_2_gonthier.se`) The effect is thus: the conditional is evaluated three times at the same instant, depending on the incarnations of the signals `x` and `y`. Let us look at the second instant. Because cancelling is weak, we have to evaluate what is inside before we check for the cancellation condition. Hence `y` is emitted to make incarnation y' present, similarly for x' because it is in parallel. Now the conditional is evaluated with x' and y' being present. Hence `a` is emitted. Now the inner cancel statement is executed, the surrounding loop terminates. Then the second incarnation y'' of `y` is created. y'' is absent since there is no reason why it should be present. The conditional is evaluated for a second time, with x' being present, and y'' being absent. Now all the computations within the outer `cancel` statement are executed, the cancellation condition applies, and the outer loop is restarted. The new incarnation x'' of `x` is created as well as a third incarnation y''' of `y`, and the conditional is evaluated for a third time, now with both x'' and `y'''` being absent. Hence `d` is emitted.

	1	*2*	*3*	*4*	...
a	_	*	*	*	...
b	_	*	*	*	...
c	_	_	_	_	...
d	*	*	*	*	...
x'	.	*			...
x''		.	*		...
y'	.	*			...
y''		.	*		...
y'''		.	*		...
...	...				

2.4 *Breaking Causality Cycles.*

Delaying emittance. *Causality cycles sometimes are a nuisance when programming synchronously. They tend to raise unexpectedly, and they may be difficult to break. Using* delayed signals *is a means to break a causality*

cycle. Delayed signals are emitted at one *instant but are present with a new value at the* next *instant only.*

The signal types

`DelayedSignal` *and* `DelayedSignal<`T`>`

(or `Delayed`, `Delayed<`T`>` *for short) are subtypes of* `Sensor` *resp.* `Sensor<`T`>`*. The semantics of the emit statement changes for delayed signals.*

> `emit` $s(v)$ `:` *The signal s is emitted (at one instant) to be present at the next instant. The value of the signal is updated only at the next instant as well.*

Let is analyse a small example.

```
emit a;
if (?a) { emit b; };
next;
if (?a) { emit c; };
```

(cf. `chpt_2_delayed-emit.se`*) Since a is emitted for the next instant it is not present in the first instant, and b is not emitted. The* `next` *process is started to terminate in the next instant. Then* `a` *is present and* `c` *is emitted.*

A delayed signal cannot occur in a causality cycle since its status can be determined at the beginning of an instant (just like that of an input signal), having been emitted, if so, at the previous instant. Hence there is no causality cycle in

```
DelayedSignal a = new DelayedSignal(new SimOutput());
       Signal b = new Signal(new SimOutput());

public DelayedNoCausality () {
   active {
      if (?a) { emit b; } else { emit a; };
   };
};
```

(cf. `chpt_2_delayed-no-causality.se`*) The conditional process is suspended till the condition can be evaluated. Since the signal* `a` *is delayed, it is present only if it has been emitted at the previous instant. Let us assume that this is not the case, i.e. the signal* `a` *is absent at the present instant. Then the signal* `a` *is emitted, but only for the next instant. Hence it is still (consistently) absent at the present instant, and no causality cycle is generated. If is was present at the previous instant,* `b` *is emitted at the present instant.*

There has been a certain amount of discussion about the benefits of delayed signals (in particular with regard to STATECHARTS*, cf. Section 4.3). We believe that both signals and delayed signals have their merits and their shortcomings. Delayed signals get rid of causality cycles which are notoriously difficult to debug. On the other hand, a causality cycle possibly brings to light some misinterpretation of code already at compile time. In contrast, it is often difficult to compute the delays introduced by delayed signals. A mistake will become manifest only at run time, hence might be difficult to debug as well.*

2.5 Embedding Data

Calling data methods. Data methods may be called either as an atomic statement or as an atomic Boolean condition.

```
class Counter {

  // signals
  Sensor   start = new Sensor(new SimInput());
  Sensor    incr = new Sensor(new SimInput());
  Signal elapsed = new Signal(new SimOutput());

  public Counter (int d) {
      latch = d;
      active {
          loop {
            await ?start; // wait for start being present
            reset();      // reset the counter
            cancel {      // increment the counter when ..
              loop {      // .. signal incr is present
                await (?incr);
                increment();
                next;
              };          // incrementing is cancelled, when ..
            } when (isElapsed());
                        // .. isElapsed() is true ..
            emit elapsed;
                       // .. until the counter is elapsed
            next;
          };
      };
  };
```

```
// data fields and data methods
  int latch;
  int counter;

  static void        reset() { counter = 0; };
  static void    increment() { counter++; };
  static boolean isElapsed() { return (counter >= latch); };
}
```

(cf. `chpt_2_counter.se`) The loop increments a counter. The loop is preempted if the counter is incremented to 5 because the data method `isElapsed` then evaluates to true.

Calls of data method must be *atomic* and *instantaneous*, i.e. when started, the respective method is evaluated, and the process must terminate instantaneously. As a consequence, fields may have several values at an instant. For instance, if started

```
increment();
reset();
```

increments the field `counter` and then resets it at the same instant.

The requirement that data methods are considered as atomic and instantaneous in a reactive context imposes considerable constraints on data methods: the execution time of a data method should not depend on the parameter values, being e.g. a list or a stack. This would violate the synchrony hypothesis in that a worst execution time cannot be predicted at compile time.

Assignments may as well be used as elementary statements in reactive code. Using assignments the example above may be rewritten to (the somewhat simpler)

```
class Counter {

  public Counter (int d) {
      latch = d;
      active {
          loop {
            counter = 0;
            cancel {
              loop {
                counter = counter + 1;
                next;
              };
            } when (counter >= latch);
```

```
            next;
          };
        };
    };

    int latch;
    int counter;
}
```

(cf. `chpt_2_counting-assign.se`)

Time races. Calling data methods may result in a time race, i.e. the execution order of two or more calls of data methods at the same instant cannot be determined from the program structure. Consider the methods `increment` and `isElapsed` as defined above in

```
[[ increment();
|| if (isElapsed()) { emit a; } else { emit b; };
]];
```

In this fragment, the counter may first be incremented and then asked whether it has elapsed, or the other way round. The results will clearly be different. Since no order of execution is prescribed, evaluation may take place in any order resulting in non-deterministic behaviour.

The *synERJY* compiler rejects such programs with an error message. Such time races, of course, occur as well with regard to assignments

```
[[ counter = counter + 1;
|| if (counter >= latch) { emit a; } else { emit b; };
]];
```

Let us speak of a *potential time race* if several data methods or assignments are called at the same instant. Often the program structure provides enough information to resolve potential time races. For instance, in the fragment

```
cancel {
    loop {
       await (?incr);
       increment();
       next;
    };
} when (isElapsed());
```

the methods `increment` and `isElapsed` may be called at the same instant. However, the call of the method `increment` is always executed before that of `isElapsed` according to the definition of the weak cancel operator: the body of the cancel statement must be evaluated before the the condition.

As a general rule, time races occur between statements that reside in different branches of a parallel statement as above. However, the signal flow may restrict the order of computation so that even such potential time races are resolved as in, e.g.

```
[[ increment();
   emit incr;
|| if (?incr) { };
   decrement();
]];
```

According to the wavefront computation emittance takes place prior to evaluation of the conditional, hence the method `increment` is called before the method `decrement`.

As stated earlier, the compiler issues an error message if it cannot resolve a potential time race. In that case the user may add a *precedence statement* such as, e.g.,

```
precedence {
    increment() < isElapsed();
    (counter =) < counter;
    increment() < add(int,int);
};
```

It specifies that, whenever involved in potential time races,

- the parameterless method `increment` is always executed before the method `isElapsed`,
- the assignment to the field `counter` precedes the access to the field `counter` and that
- the method `increment` is always executed before the method `add` with two parameters of type `int`,

Note that the precedence is general: if involved in a potential time race, *any* call of the method `increment` is executed before any call of the method *isElapsed*. Similarly, any assignment to the field `counter` must be executed before *any* access of the field `counter`.

The precedence statement introduces a new kind of precedence relations on data methods, assignments, and field accesses. The precedence relations are specified using the following patterns.

- The *data method pattern* is determined by $m(T_1, \ldots, T_n)$, i.e. a method name m and the number and types T_1,...,T_n of arguments (i.e. the signature of the method).
- The *assignment pattern* is determined by $(f =)$ with f being a field name. The notation `(`f`=)` is meant to abstract the various assignment and increment operators that change the field f, e.g. `=`, `++`, `+=`, etc..
- The *field access pattern* is determined by the name f of a respective field.

Let a and b be two such patterns. The precedence may be specified using two kinds of relations:

`a < b` : all occurrences of the pattern `a` must be scheduled before any occurrence of the pattern `b`.

`a*` : calls of `a` may be scheduled in any order.

The precedence relations may be combined in a precedence clause. Combinations are, e.g.

`a* < b :`
all calls of `a` must be scheduled before `b` and all calls of `a` may be scheduled in any order.

Note that control flow and precedence statements may contradict each other as in the code fragment

```
...

if (counter > 0) { counter-- };

...

precedence {
    (counter =) < counter;
};
```

According to the control flow the field `counter` must be accessed before the value of field may eventually be decreased by one. On the other hand the

precedence close states that access to the field should take place only after all updates of the field have been executed. Obviously, we cannot satisfy both these scheduling requirements. Hence a causality error must be raised.

For a general strategy, the program structure will determine the causal dependencies most of the times. Only a parallel statement may cause a time race. In that case, precedence clauses should be added to eliminate non-determinism. A second choice, may be to add precedence clauses for semantic control; for instance, if some data actions should always be scheduled after some others, it may be a good idea to state this explicitly to exclude that an unwanted scheduling order is induced from the program structure. Be aware however, that, the more precedence clauses are introduced, the likelihood of a causality cycle increases. Hence precedence statements should be used with care.

Multiple emits for valued signals. Another type of "time race" occurs with regard to valued signals. A signal may be emitted several times at the same instant as in

```
[[ emit x(1); || emit x(2); ]];
```

Depending on the order of evaluation, the value of the valued signal `x` may be 1 or 2. The compiler again rejects such a program because of non-determinism except for one exception:

- if a signal is pure (i.e. of type `Signal`).

In that case no conflict can arise since due to the lack of values.

Note, however, that the signal `x` will have the value 2 after evaluating

```
emit x(1);
emit x(2);
```

The sequential composition imposes a causal order.

Labels for sequencing. *Labeling is a new concept for synchronous languages that supports a finer control of precedence that precedence statements do. Labels refer to individual statements within a reactive program. The notation is*

identifier`::`*statement*`;`

Labels are used for resolving time races as follows. Consider a fragment of code such as

```
[[ l1:: emit x(1); || l2:: emit x(2); ]];

...

precedence {
    l1:: < l2:: ;
};
```

The labels in the precedence statement quite neatly express that `emit x(1);` *should be executed before* `emit x(2);`*. Hence the value of* `x` *will be* 2 *after executing the code above.*

Labeling supports a much finer precedence control for resolving time races than using method names only. In the (semantically irrelevant) piece of code, for instance,

```
[[ l1:: reset(); || l2:: decrement(); || l3:: reset(); ]];

...

precedence {
    l1:: < l2:: < l3:: ;
};
```

a value is reset, then decremented, and reset again.

2.6 Elementary Examples: Counters

We discuss a number of elementary examples programming counters. We point out possible design decision and highlight some traps one may encounter.

Counting using control only. Designing counters using control only is a rather restricted exercise, but quite useful to demonstrate the power of parallel composition.

The most elementary counter probably is a two bit counter that emits the signal `elapsed` if the signal `incr` has been present twice.

```
class TwoBitCounter {

    Sensor    incr = new Sensor(new SimInput());
    Signal elapsed = new Signal(new SimOutput());

    public TwoBitCounter () {
```

```
        active {
            await ?incr;
            next;
            await ?incr;
            emit elapsed;
        };
    };
}
```

(cf. `chpt_2_two-bit-counter.se`) If started the process waits for the presence of the signal `incr`. In case that `incr` is present, the await process stops, and the second await process is started only in the next instant. Again presence of `incr` terminates the the await process. Hence in the same instant `elapsed` is emitted.

One should note that the rather similar process

```
await ?incr;
await ?incr;
emit elapsed;
```

(cf. `chpt_2_wrong-two-bit-counter.se`) fails to implement a two bit counter; the second await process is started at the same instant the first terminates, but terminates itself instantaneously since `incr` is present. Hence `elapsed` is emitted at the first instant the signal `incr` is present.

One may iterate the counting by adding a loop as in

```
loop {
    await ?start;
    await ?incr;
    next;
    await ?incr;
    emit elapsed;
    next;
};
```

(cf. `chpt_2_looping-two-bit-counter.se`) The signal `elapsed` is always emitted if the signal `incr` is present for the second time. We have added the signal `start` to start counting only when it is present.[7]

There is an alternative using the `await next` operator

```
loop {
    await next ?start;
    await ?incr;
```

[7] What happens if the second `next` is omitted?

```
    next;
    await next ?incr;
    emit elapsed;
};
```

(cf. `chpt_2_second-looping-two-bit-counter.se`) Note, however, that this counter does not start to count in the first instant.

Next step is to generate a four bit counter by parallel composition of two two bit counters.

```
[[ loop {
       await ?incr; // first two bit counter
       next;
       await ?incr;
       emit carry;
       next;
   };
|| loop {
       await ?carry; // second two bit counter
       next;
       await ?carry;
       emit elapsed;
       next;
    };
]];
```

(cf. `chpt_2_simple-four-bit-counter.se`) The first two bit counter emits the signal `carry` if the input signal `incr` has been present twice, the second emits the signal `elapsed` if `carry` has been present twice. `carry` is a local signal. Iteration generates 2^n bit counters.

We may add a tiny bit of sophistication by stipulating that the counter should be (re)started only if the signal `start` is present.

```
loop {
    await ?start;
    cancel {
        [[ loop {
               await ?incr;
               next;
               await ?incr;
               emit carry;
               next;
           };
        || loop {
               await ?carry;
```

```
                    next;
                    await ?carry;
                    emit elapsed;
                    next;
                };
            ]];
        } when (?elapsed) {};
        next;
    };
```

(cf. `chpt_2_four-bit-counter.se`) Since the parallel process never terminates we have to use preemption to restart.

One may wonder about the last `next` statement in the outer loop. One may omit it and find that the compiler issues an error message "`instantaneous loop ...`". Of course, analysis of the program proves that the signal `elapsed` is never emitted when entering the outer loop (provided that it is not emitted elsewhere). Hence the loop is not instantaneous. However, the effort to analyse such a fragment for absence of instantaneous loops is considerable.[8] Hence we use an algorithm of low complexity with the drawback that some ostensibly correct programs are rejected by the compiler.

Mixing control and data. A counter implemented using pure control has the obvious drawback that, for a counter of length n, a different program has to be designed for each n. A much more flexible solution is obtain if we use some integer attribute that is set to 0 when starting counting, and that is incremented whenever the signal `incr` is present.

```
class Counter {

  // signals
  Sensor   start = new Sensor(new SimInput());
  Sensor    incr = new Sensor(new SimInput());
  Signal elapsed = new Signal(new SimOutput());

  public Counter (int d) {
      latch = d;
      active {
          loop {
            await ?start; // wait for start being present
            reset();      // reset the counter
            cancel {      // increment the counter when ..
              loop {      // .. signal incr is present
```

[8] at worst as difficult as proving satisfiability of a set of Boolean equations.

```
                await (?incr);
                increment();
                next;
              };          // incrementing is cancelled, when ..
            } when (isElapsed());
                          // .. isElapsed() is true ..
            emit elapsed;
                        // .. until the counter is elapsed
            next;
          };
      };
    };

  // data fields and data methods
    int latch;
    int counter;

    static void        reset() { counter = 0; };
    static void    increment() { counter++; };
    static boolean isElapsed() { return (counter >= latch); };
  }
```

(cf. chpt_2_counter.se)

Using data methods with almost no control. There is a sort of contrived version of a counter where control is reduced to a minimum.

```
  class CounterWithDataOnly {

    // signals
    Sensor   start = new Sensor(new SimInput());
    Sensor    incr = new Sensor(new SimInput());
    Signal elapsed = new Signal(new SimOutput());

    public CounterWithDataOnly (int d) {
        latch = d;
        active {
            loop {
              if (isElapsed(?start,?incr)) {
                  emit elapsed;
                  reset();
              };
              next;
            };
        };
```

```
    };

    // data fields and data methods
    static int latch;
    static int counter = 0;
    static boolean hasStarted = false;

    static void reset () {
       counter    = 0;
       hasStarted = false;
    };

    static boolean isElapsed(boolean xstart, boolean xincr) {
            if (xstart) { hasStarted = true; };
            if (hasStarted && xincr) { counter++; };
            return (counter >= latch);
    };
  }
```

(cf. `chpt_2_counter-with-data-only.se`)

NOTES ON GOOD PRACTICE.

- Using data in reactive applications always is a cause of concern in embedded systems since resources are often constrained. For instance, the use of values of type float may be disastrous on a 8 bit micro controller because representation of floats and operations on floats consume a lot of memory and of computation time. This holds even if only one variable of type `float` is declared since respective software libraries must be loaded if no hardware support for floating point operations is provided.

- A similar argument applies for using fields and methods. Both should be declared `static` whenever feasible: then the additional overhead for implementing the indirection of calling methods relative to an object is avoided. In particular, all fields and methods in a configuration class should be static since the configuration object is a singleton, hence no indirection is needed.

- Finally, one should always keep in mind the statement

  Control is cheap, data are expensive

meaning that whenever one can model control using pure signals and the control structures such as loop, preemption, etc. the resulting code is extremely efficient, in particular for a micro controller.

2.7 *The Type* `time`

Sampling system time. *The synchrony hypothesis captures an abstract notion of time. As we have argued, the abstraction simplifies the task of the designer and programmer. However, one would like to refer to the passing of concrete time as well. Phrases such as "after x milliseconds do ..." come to mind immediately when dealing with applications. Such a concrete notion of time can coexist with abstract time if concrete time is considered as just another input that is sampled at the beginning of each instant.*

In fact, one is rather interested in the passing of time than in time in absolute terms, i.e. in statements of the form

- *for how much time a signal has not been present, or*
- *how much time is needed for computing an instant, or*
- *wait for a specified amount of time*

rather than a statement

- *stop the process in the year 2525 exactly at 12 pm, December, 31^{st}.*

A primitive built-in type

`time`

with constants such

```
1hour, 3sec, 100msec, ...
```

will help to specify timing conditions:

- *A statement such as*

  ```
  await 300msec;
  ```

  comes to mind when dealing with concrete time. The semantics is natural: it is recorded how much (real) time has passed since the waiting has started. If the difference is greater than 300 *milliseconds the wait process is preempted.*

  More generally, preemption operators may be enhanced in that the conditions C_i of

```
cancel [strongly]
   P
when (C1) { P1 }
[ else when (C2) { P2 }
  ...
  else when (C2) { Pn }
];
```

are required to be either Boolean expressions or expressions of type `time`*.*

- *Timeouts often depend on the concrete time that passed since some sensor or signal s was not present. Hence all sensors and signals are provided with an operator*

  `@`s

  that returns the time for which the sensor or signal s has not been present. We refer to `@` *as the* (a)wait-for-signal *operator.*

  A typical fragment of code is

  ```
  loop {
      await @actuator > 5sec;
      emit frozen;
      next;
  };
  ```

  This "watchdog" loop checks the signal `actuator` *has not been updated for more than* 5 *seconds. In that case an alarm* `frozen` *is issued.*

  Note that `@s` *equals* 0 *if the sensor or signal s is present at every instant.*

Resolution of time. *The finest resolution of time is naturally determined by the system clock. A work station may measure time in terms of micro- or even nanoseconds, while the time resolution of smaller micro controller will rarely exceed milliseconds.*

For synchronous programs time resolution must determined by instants in order to conform with the synchrony hypothesis. Hence we measure "passing of real time" in that we measure the delta of time between two instants. Hence the faster the execution of instants proceeds (as, for instance, specified by the field `timing` *in a configuration class) the finer the resolution of concrete time is. One should notice that this interpretation of real time*

conforms with the requirements of the synchrony hypothesis, and actually seems to be the only way how to reconcile "abstract" and "real" time in a synchronous setting.

For convencience, the delta of time of an instant can be accessed using the built-in input sensor `dt`*. Note that this sensor cannot be used in the first instant. If so, the run time system will throw an exception.*

2.8 Reactive Methods

Calling reactive methods. So far, we have avoided using methods for implementing behaviour. Now being on established grounds, we consider reactive methods as another means for reusing reactive code.

By definition, a method is *reactive* if a reactive statement occurs in its body, or if the modifier `reactive` is applied. By default, all other methods are referred to as *data methods*. Reactive methods denote processes that may have parameters.

For instance, the now familiar four bit counters of the example above may be embodied in a reactive method:

```
class FourBitCounter {

    Sensor start = new Sensor(new SimInput());
    Sensor  incr = new Sensor(new SimInput());
    Signal      carry = new Signal();
    Signal    elapsed = new Signal(new SimOutput());

    public FourBitCounter () {
        active {
            loop {
                await ?start;
                cancel {
                    [[ two_bit(incr,carry);
                    || two_bit(carry,elapsed);
                    ]];
                } when (?elapsed);
                next;
            };
        };
    };

    void two_bit (Sensor x,Signal y) {
        loop {
```

```
            await (?x);
            next;
            await (?x);
            emit y;
            next;
        };
    };

}
```

(cf. `chpt_2_two-times-two-bit-counter.se`)

Reactive methods can only be called within the body of an `active` statement or from reactive methods. In that data methods are "downward closed": the obvious argument is that the reactive code should control the data operation, and not vice versa.

The semantics of reactive method calls is defined by syntax expansion: the call is replaced by the code obtained from its body by replacing the formal by the actual parameters.

Labelling once again. *Method calls can be labelled. The advent of reactive methods considerably enhances the power of labels. Consider the following fragment of code:*

```
public LabelledMethods1 () {
    active { l1:: m1(); };
};

void m1 () {
    [[ l2:: data1();
    || l3:: data2();
    ]];
};

precedence {
    l1::l2:: < l1::l3::;
};
```

The label chain of `l1::l2` *exactly addresses the position of the reset statement as seen from level of the object as does the chain* `l1:l3` *for the decrement statement. The labelling extends to comparing calls of data methods made in different methods as in*

```
public LabelledMethods2 () {
```

```
        active { [[ l1:: m1(); || l2:: m2(); ]]; };
    };

    void m1 () { l3:: data1(); emit a; };

    void m2 () { l4:: data2(); emit a; };

    precedence {
        l1::l3:: < l2::l4::;
    };
```

One should note that precedence statements can only be specified at the level of an object.

2.9 Interfacing to the Environment

Interfaces for interfacing. Signals are always private, as all fields and methods of an reactive object are. However, sensors or signals may interface to the environment. We distinguish "input" sensors, "output" signals, and "local" signals. The kind of signal is determined by signal constructor. If the constructor has no argument the signal is local. If the signal constructor has one parameter of interface type `Output` the signal is an output signal. Constructors of sensors must always have an argument of interface type `Input`, e.g.

```
class InOutSignals {
    Input     sensor_input = new MyInput();
    Output actuator_output = new MyOutput();

    Sensor<int>   sensor = new Sensor<int>(sensor_input);
    Signal<int> actuator = new Signal<int>(actuator_output);

    public InOutSignals() {
        active {
            if (?sensor) { emit actuator($sensor + 1); };
        };
    };
}
```

The interface types `Input` and `Output` are built-in. The interfaces are so-called marker interfaces meaning that they act as a place holder lacking any semantic content. Implementations should provide the following callback methods depending on the nature of the input or output signal.

pure input sensors A parameterless method `new_val` with result type `boolean`.

valued input sensors A parameterless method `new_val` with result type `boolean` and a parameterless method `get_val` with result type T if the value type of the signal is T.

pure output signals A parameterless void method `put_val`.

valued output signals A void method `put_val` with a parameter of type T if the value type of the signal is T.

According to the synchronous execution model, input sensors are set at the beginning of an instant, while output signals are communicated to the environment at the end of an instant. Input sensors are set using the methods `new_val` and `get_val`. If `new_val` returns the value `true`, the sensor is set to be present, and its value is updated by the value obtained as a result of `get_val`. Output signals are communicated using the method `put_val`: if an output signal is present the method `put_val` will be called with the actual value of the signal, in case that the signal is valued.

In the example given, the interfaces may be implemented by the types

```
class MyInput  implements Input  {
   public MyInput()  {};

   native boolean new_val ();
   native int     get_val ();
}

class MyOutput implements Output {
   public MyOutput() {};

   native void put_val ();
}
```

Here the methods are assumed to be *native* , that is implemented in C as platform-dependent code. Alternatively, we may have used anonymous classes:

```
class AnonInOutSignals {
    Sensor<int>   sensor =
       new Sensor<int>(
              new Input(){ native boolean new_val ();
                           native int     get_val ();
                         }
       );
```

```
    Signal<int> actuator =
       new Signal<int>(
              new Output(){ native void put_val ();
                          }
       );

    public AnonInOutSignals() {
        active {
            if (?sensor) { emit actuator($sensor + 1); };
        };
    };
}
```

The marker interfaces are extended by the respective methods.

We summarise: the kind of a signal is determined by its constructor

- The type `Sensor<`T`>` has one constructor with a parameter of type `Input`.
- The type `Signal<`T`>` has two constructors,
  - one being parameterless (for "local" signals) , and
  - the other with a parameter of type `Output`.

Implementations of `Input` and `Output` are required to have (callback) methods `new_val` and `get_val`, resp. `put_val` the type of which must match with the type of the signal value. Mismatches raise an error message. We discuss the interfaces `Input` and `Output`, and their implementations at length in Chapter 7.

Running an example. We assume that the *synERJY* programming environment has been installed (cf. [40]). The following example can be found as file `firsttry.se` in the directory `Examples/Book` of the *synERJY* distribution (as all the other examples discussed further below). The program is very simple in that it reacts only at the first instant.

Example 2.9.1 (FirstTry)

```
class FirstTry {
    static final time timing = 100msec;

    public static void main (String[] args) {
       while (instant() == 0) {};  // control loop with 100msec
    };
```

```
    Sensor   sensor = new Sensor(new SimInput());
    Signal actuator = new Signal(new SimOutput());

    public FirstTry () {
        active {
            if (?sensor) { emit actuator; };
        };
    };
}
```

(cf. chpt_2_first-try.se)

The public static method `main` generates an application as usual. The procedure `main` is called at system startup and is designed to run forever as long as the method `instant` of the reactive machine returns without indication of an error (`!= 0`). The method `instant` is generated by the compiler. It executes an elementary execution step of the reactive machine, an instant.

The variable `timing` defines the minimal amount of time required to execute an instant properly by a real system for worst case behaviour.[9] Its value is used by the simulator for emulating the real system. In case that execution of any instant of the real system never exceeds the time specified by the timing clause, it will be guaranteed that the simulator exactly reflects the behaviour of the real system.

To interface to the simulator of the *synERJY* programming environment, there are two particular built-in classes `SimInput` and `SimOutput` that implement the interfaces `Input` resp. `Output`. These automatically provide proper methods `new_val` and `get_val`, respectively `put_val` for interacting with the simulator. In fact, when running to the simulator, all instances of an implementation of `Input` and `Output` are internally replaced by in instance of `SimInput` or `SimOutput`.

After launching *synERJY* the file `first_try.se` is loaded using the `Load` entry of the `File` menu. Loading the file should have compiled the example. Next pressing the button `Generate` generates the simulation code, and pressing the `Simulate` button opens the simulator.

Running the simulator, the sensor `sensor` can be set to present by clicking on it with the left button in the simulator panel. If the sensor `sensor` is set to be present at the first instant, the signal `actuator` will be emitted, and then nothing will happen any more. The presence of signals is indicated by a `$` in the trace browser.

[9] The type `time` is built in. Issues of timing will be discussed in Section 2.7.

Chapter 3

Examples

3.1 A Car Belt Controller

The task is to define a controller for a car belt with the subsequent behaviour:

- If ignition is started then an alarm beeper should be started after 5 seconds and keep on beeping for 10 seconds if the safety belt is not fastened.
- Additionally an alarm light should immediately start and keep on blinking as long as the safety belt is not fastened.

We assume that starting ignition is indicated by the presence of the sensor `keyOn` and cutting down the engine by presence of the sensor `keyOff`. The other sensors and signals have the obvious connotation.

```
class Belt {

Sensor  beltOn = new Sensor(new SimInput());
Sensor beltOff = new Sensor(new SimInput());
Sensor   keyOn = new Sensor(new SimInput());
Sensor  keyOff = new Sensor(new SimInput());

Signal  alarmLightOn = new Signal(new SimOutput());
Signal alarmLightOff = new Signal(new SimOutput());
Signal      beeperOn = new Signal(new SimOutput());
Signal     beeperOff = new Signal(new SimOutput());

public Belt () {
    active {
        loop {
```

```
                await ?keyOn;
                cancel {
                    emit alarmLightOn;
                    await 5sec;
                    emit beeperOn;
                    await 10sec;
                    emit beeperOff;
                    halt;
                } when (?beltOn || ?keyOff) {
                    emit alarmLightOff;
                    emit beeperOff;
                };
                next;
            };
        };
    };
    }
```

(cf. `chpt_3_belt.se`) The behaviour is thus:

- If the key is turned on the alarm routine (i.e. the parallel statement) is started.
- The alarm is immediately cancelled when the belt is put on, or the key is turned off.
- The alarm routine never terminates

For a proper behaviour we assume that neither the sensors `keyOn` and `keyOff` nor the sensors `beltOn` and `beltOff` can occur at the same instant. This assumption is not needed if we use valued signals as in

```
class BeltWithValuedSignals {

  Sensor<boolean>  belt
    = new Sensor<boolean>(new SimInput());
  Sensor<boolean>  key
    = new Sensor<boolean>(new SimInput());

  Signal<boolean>  alarmLight
    = new Signal<boolean>(new SimOutput());
  Signal<boolean>  beeper
    = new Signal<boolean>(new SimOutput());

  public BeltWithValuedSignals () {
```

```
      active {
          loop {
              await $key;
              cancel {
                  emit alarmLight(true);
                  await 5sec;
                  emit beeper(true);
                  await 10sec;
                  emit beeper(false);
                  halt;
              } when ( $belt || ! $key) {
                  emit alarmLight(false);
              };
              next;
          };
      };
    };
}
```

(cf. `chpt_3_belt-with-valued-signals.se`)

3.2 A Stopwatch

The stopwatch is a standard example for reactive control in Halbwachs' book [16]. We follow his presentation.

A simple stopwatch. The simple stopwatch has a sensor `start_stop` that alternatively sets the stopwatch in a running and a stopped state. Time resolution is 1/100 of seconds. The stopwatch computes an integer `the_time` the value of which is the total amount of time (in terms of 1/100 of seconds) spent in a running state.

```
class SimpleStopwatch {

static final time timing = 10msec;

public static void main (String [] args) {
 while (instant() == 0) { };
};

time the_time;
```

```
Sensor start_stop = new Sensor(new SimInput());

Signal<time> elapsed_time = new Signal<time>(new SimOutput());

public SimpleStopwatch () {
  active {
    the_time = 0msec;
    loop {
      await ?start_stop;
      emit elapsed_time(the_time);
      next;
      cancel {
        loop {
           the_time = the_time + $dt;
           emit elapsed_time(the_time);
           next;
        };
      } when (?start_stop);
      next;
    };
  };
};
}
```

(cf. `chpt_3_simple-stopwatch.se`)

The variable `the_time` is initialised to `0sec`. The signal `elapsed_time` is emitted with the value `the_time` whenever the time changes. Note that time constant `timing` is set to `10msec`. Thus an instant is required to take exactly a `1/100` of a second. The states "running" and "stopped" alternate. At the beginning the watch is stopped. If the `start_stop` signal is present for the first time the watch starts measuring time. At each instant the value of the variable `the_time` is increased by the value of the built-in signal `dt`. The value of `dt` is the "time delta" in between two instants (cf. Section 2.7).[1] The measuring is stopped if the signal `smart_stop` is present again. Note that the elapsed time is not increased at that instant since preemption is strong. The behaviour is iterated by the loop. The two next statements in the outer loop are necessary to distinguish between the start and the stop with regard to the presence of the signal `start_stop`.

Stopwatch with reset. In the next version we add a sensor `reset` that, if present, causes the time to be reset to 0. To this behalf we slightly modify

[1] Using `dt` the design becomes independent of a specific resolution in that the time constant may be set to any resolution.

the simple stopwatch program in that we use a method for the reactive behaviour. The method will be called withing a combination of a loop and a cancel statement.[2]

```
class StopwatchWithReset {

static final time timing = 10msec;

public static void main (String [] args) {
  while (instant() == 0) { };
};

time the_time;

Sensor start_stop = new Sensor(new SimInput());
Sensor      reset = new Sensor(new SimInput());

Signal<time> elapsed_time = new Signal<time>(new SimOutput());

public StopwatchWithReset () {
  active {
     stopwatch_with_reset();
  };
};

void stopwatch_with_reset () {
  loop {
      cancel strongly {
         simple_stopwatch ();
      } when (?reset);
      next;
  };
};

void simple_stopwatch () {
   the_time = 0msec;
   loop {
      await ?start_stop;
      emit elapsed_time(the_time);
      next;
      cancel {
        loop {
```

[2]If you compare with the solution in [16], Section 2.4, you should note that reactive methods compare to modules in ESTEREL.

```
            the_time = the_time + $dt;
            emit elapsed_time(the_time);
            next;
         };
      } when (?start_stop);
      next;
   };
};
}
```

(cf. `chpt_3_simple-stopwatch-with-reset.se`)

In anticipation of further developments, the reset mechanism is specified as method `stopwatch_with_reset` that is called within the active context.

Intermediate time handling. A next version of the stopwatch can record the intermediate time while continuing to measure the global time (for instance, to record the time spent on running one lap). At the presence of a signal `lap` time is frozen on the display while the internal stopwatch time proceeds. The next time `lap` is present the stopwatch again displays the running time.

This behaviour is achieved by putting a "lap filter" in parallel with the stopwatch behaviour. We discuss the following fragment.

```
Sensor start_stop = new Sensor(new SimInput());
Sensor      reset = new Sensor(new SimInput());
Sensor      lap = new Sensor(new SimInput());

Signal<time> display_time = new Signal<time>(new SimOutput());

Signal<time> elapsed_time = new Signal<time>();

public StopwatchWithIntermediateTime () {
  active {
    [[ stopwatch_with_reset(); || lap_filter(); ]];
  };
};

void lap_filter () {
   loop {
     cancel strongly next {
        loop {
           cancel next {
              sustain {  //  running time
                 if (?elapsed_time) {
```

```
                    emit display_time($elapsed_time); // (*)
                  };
               };
            } when (?lap);
            // frozen time
            await next ?lap;
            emit display_time($elapsed_time);
         };
      } when (?reset);
   };
};
```

(cf. `chpt_3_stopwatch-with-intermediate-time.se`)

Again the behaviour of the simple stopwatch with a reset is rephrased as a reactive method. The lap filter is specified in a second reactive method that is put in parallel with the stopwatch method. There are two states: when "running" the display time coincides with the elapsed time. The first instant the signal `lap` is present the display time is frozen – the emittance of the signal `display_time` is preempted – while the time still elapses. The next instant `lap` is present the display time coincides with the elapsed time again. This behaviour is iterated.

The general stopwatch. The actual stopwatch has only two buttons.

- The first button will correspond to the `start-stop` signal.
- The interpretation of the second button `freeze_reset`depends on the state of the stopwatch. Either it will correspond to the `reset` signal or to the `lap` signal.

The respective interpretation of the second button is realised by a method `button_interpreter` that emits the signals `reset` or `lap` according to the state of the stopwatch if the signal `freeze_reset` is present.

The "state" of the stopwatch is defined in terms of two "flip-flops". The first distinguishes if the stopwatch is running or not, the second if the display is frozen or not. The signal `reset` is emitted only if `freeze_reset` is present, the stopwatch is stopped, and the display is not frozen. Otherwise the signal `lap` is emitted if `freeze_reset` is present.

```
Sensor   start_stop = new Sensor(new SimInput());
Sensor freeze_reset = new Sensor(new SimInput());

Signal<time> display_time = new Signal<time>(new SimOutput());
```

```
Signal reset = new Signal();
Signal   lap = new Signal();

Signal<time> elapsed_time = new Signal<time>();

public Stopwatch () {
  active {
// [[CausalityCycle]]
    [[ button_interpreter();
    || lap_filter();
    || stopwatch_with_reset();
    ]];
  };
};

void button_interpreter () {
   Signal stopwatch_running = new Signal();
   Signal       frozen_time = new Signal();
   [[ loop {
         await ?freeze_reset;
         if (?stopwatch_running) {
            emit lap;
         } else if (?frozen_time) {
            emit lap;
         } else {
            emit reset;
         };
         next;
      };
   || // flip-flop "running/stopped"
      loop {
         await ?start_stop;
         cancel next {
            sustain { emit stopwatch_running; };
         } when (?start_stop);
         next;
      };
   || // flip-flop "running-time/frozen-time"
      loop {
         await ?lap;
         cancel next {
            sustain { emit frozen_time; };
         } when (?lap);
         next;
```

```
      };
   ]];
};

void lap_filter () {
   loop {
     cancel strongly next {
        loop {
           cancel strongly next {
              sustain {
                 if (?elapsed_time) {
                    emit display_time($elapsed_time);
                 };
              };
           } when (?lap);
           await next ?lap;
           emit display_time($elapsed_time);
        };
     } when (?reset);
  };
};
```

(cf. `chpt_3_stopwatch-general-causality.se`)

Unfortunately, the compiler raises a causality error involving the signals `lap` and `frozen_time`. Analysis proves that, if `freeze_reset` is present, and if the second flip-flop is in state "frozen_time", the signal `lap` is emitted, causing the process `sustain { emit frozen_time; };` to be cancelled. But the latter is a cause for the emittance of `lap`.

The remedy is subtle. If the signal `frozen_time` is redeclared to be a delayed signal

```
DelayedSignal frozen_time = new DelayedSignal();
```

the causality cycle is broken and the program behaves properly. One should note that there are other ways to break the causality cycle, for instance, to delay emittance of the signal `lap`. Then, however, the behaviour subtly changed.

3.3 A Train Example

The task is to design a control software for a single line that splits at both ends into two lines.[3] Trains can pass the single line from both sides. The control program has to ensure that no accident happens.

We consider the global system composed of:

physical part: the single line, the trains, tracks, pointings, presence sensors, actuators controlling the switches and traffic lights.

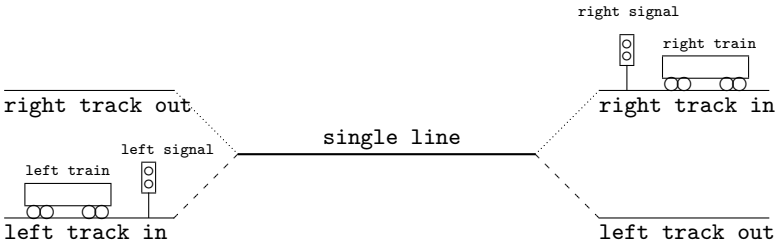
control part: an *synERJY* program which reads the sensors and controls actuators (traffic lights and switches).

The assumption is that trains behave properly, for instance they obey to the traffic lights and not to move backward.

Physical part. The layout consists of

- the single track,
- four other tracks: one in - track and one out track at each end of the single line,
- a pointing at each limit of the single line
- a sensor of train presence on the single line
- sensors for switch presence
- traffic lights on each side of the single line
- left trains and right trains coming respectively from the left and the right side of the single line.

A picture shows all the components



[3]The example was proposed by Paul Caspi and Rym Salem in context of the ESPRIT-CRISYS project.

The control system. Given the presence sensors, the control decides the position of the pointings and the state of the traffic lights.

One can imagine that the following properties should hold:

Safety: there must be no collision and no derailment. A collision occurs when two trains meet (if going in opposite directions) or reach (if going in the same direction) one another on the single line or on the same in - track. Derailment takes place if the physical path corresponding to the selected direction is not established while a train is moving.

Fairness: there must be no starvation, i.e it should not be the case that two successive trains go in one direction while another train is waiting in the opposite direction.

A control program. Here is a first simple design of the control program.

```
class SingleLine {

static final time timing = 300msec;

public static void main (String[] args) {
   while (instant() == 0) {};
};

public SingleLine () {
   active {
      emit leftSignalToRed;
      emit rightSignalToRed;
      loop {
          [[ if ( ?isLineFree && ?leftTrainWaiting ) {
                 emit leftPointingToIn;
                 emit rightPointingToOut;
                 await (?leftPointingIsIn &&
                        ?rightPointingIsOut);
                 emit leftSignalToGreen;
                 await (! ?isLineFree);
                 emit leftSignalToRed;
             };
          || if ( ?isLineFree && ?rightTrainWaiting
                  && ! ?leftTrainWaiting ) {
                 emit rightPointingToIn;
                 emit leftPointingToOut;
                 await (?rightPointingIsIn &&
```

```
                        ?leftPointingIsOut);
                  emit rightSignalToGreen;
                  await (! ?isLineFree);
                  emit rightSignalToRed;
                };
            ]];
            next;
        };
    };
};

Sensor isLineFree = new Sensor(new SimInput());

Sensor  leftTrainWaiting = new Sensor(new SimInput());
Sensor rightTrainWaiting = new Sensor(new SimInput());

Sensor   leftPointingIsIn = new Sensor(new SimInput());
Sensor  leftPointingIsOut = new Sensor(new SimInput());
Sensor  rightPointingIsIn = new Sensor(new SimInput());
Sensor rightPointingIsOut = new Sensor(new SimInput());

Signal  leftPointingToOut = new Signal(new SimOutput());
Signal   leftPointingToIn = new Signal(new SimOutput());
Signal rightPointingToOut = new Signal(new SimOutput());
Signal  rightPointingToIn = new Signal(new SimOutput());

Signal  leftSignalToGreen = new Signal(new SimOutput());
Signal    leftSignalToRed = new Signal(new SimOutput());
Signal rightSignalToGreen = new Signal(new SimOutput());
Signal   rightSignalToRed = new Signal(new SimOutput());
}
```

(cf. `chpt_3_train.se`) The signals have an obvious connotation. There are essentially two parallel branches; one for granting way for a left train, the other for a right train. We discuss the first. If the line is free and a left train is waiting then signals are emitted to set the pointings properly. Given that these are set, the left signal is set to green. If the line is busy the signal is reset to red. The other branch is symmetric except that a right train can only move if there is no left train waiting.

Concerning the environment interface. Note that, for proper operation we have to assume that the sensors `leftTrainWaiting` respectively `rightTrainWaiting` are continuously present if a train is waiting. Similarly

the sensor `lineFree` must always be present if the line is free. An alternative one may use Boolean valued signals instead

```
static final boolean   red = false; // constants for status ..
static final boolean green = true;  // ... of traffic lights

static final boolean  left = false; // constants for position ..
static final boolean right = true; // .. of pointings

public SingleLine () {
   active {
      emit leftSignal(red);
      emit rightSignal(red);
      loop {
          [[ if ( $lineFree && $leftTrainWaiting ) {
                 emit rightPointingToOut;
                 await (?leftPointingIsIn &&
                        ?rightPointingIsOut);
                 emit leftSignal(green);
                 await (! $lineFree);
                 emit leftSignal(red);
             };
          || if ( $lineFree && $rightTrainWaiting
                  && ! $leftTrainWaiting ) {
                 emit rightPointingToIn;
                 emit leftPointingToOut;
                 await (?rightPointingIsIn &&
                        ?leftPointingIsOut);
                 emit rightSignal(green);
                 await (! ?lineFree);
                 emit rightSignal(red);
              };
          ]];
          next;
      };
   };
};
```

(cf. `chpt_3_train-valued.se`) Here presence of, for instance, the signal `lineFree` implies a possible change of its value. Hence blocking the single line may be indicated by the presence of the sensor with value `false`, while setting the line free is indicated by its presence with a value `false`. Note, however, that one should not assume that the values alternate.

The discussion may stress the importance of choosing a proper interface with regard to the environment. Of course, sensors or signals are often

given a priori when programming an embedded system. However, an overall design should ideally reflect upon the choice of sensors and signals a priori depending on the control problem.

Toward a fairer solution. The control programs above have the obvious disadvantage that they show some preference to the trains on the left side. For a fairer solution one needs some kind of priority control. We add a boolean valued signal `priority` that gives priority to the trains on the left side if its value is true, and priority to the trains on the right if it is false.

```
        loop {
            [[ if (?leftTrainWaiting && ! ?rightTrainWaiting ) {
                  if ($priority) {
                      next; emit priority(false);
                  } else {
                      next; emit priority(true);
                  };
               };
            || if ( ($priority || !?rightTrainWaiting)
                     && ?isLineFree && ?leftTrainWaiting ) {
                   emit leftPointingToIn;
                   emit rightPointingToOut;
                   await (?leftPointingIsIn &&
                          ?rightPointingIsOut);
                   emit leftSignalToGreen;
                   await (! ?isLineFree);
                   emit leftSignalToRed;
               };
            || if ( ! ( $priority || ?leftTrainWaiting)
                    && ?isLineFree && ?rightTrainWaiting ) {
                   emit rightPointingToIn;
                   emit leftPointingToOut;
                   await (?rightPointingIsIn &&
                          ?leftPointingIsOut);
                   emit rightSignalToGreen;
                   await (! ?isLineFree);
                   emit rightSignalToRed;
                 };
            ]];
            next;
        };
    };
};
```

(cf. `chpt_3_train-with-priority1.se`) (Note that the delays avoid causality errors in the parallel branch dealing with priorities.)

Obviously, there is still some jitter of the priorities if two trains are waiting and the line is still busy. Priorities should change only if the status of the waiting trains or of the single line changes. We leave it as an exercise to modify the program accordingly.

3.4 A Robot Example

Let a wheeled robot being able to move forward and backward, and to turn left and right. Two bumper sensors on the left and right front end are used to detect obstacles. The task is to control the robot such that it travels around without being caught, for instance, in some corner.

The concrete robot to control is the LEGO^{TM}-Mindstorm robot "Rover". The two sensors are encoded by the sensors `sensorA` and `sensorC`. The rover has two servo motors (actuators) that drive the wheels on each side. The motors can turn forward or backward at a given speed. The valued signals `ldir` and `lspeed` determine the direction and the speed of left motor, the valued signals `rdir` and `rspeed` that of the right motor. The values are of type `uint8`.

The basic behaviour of the robot is to move straight forward at medium speed. For both the motor actuators `ldir` and `rdir` are set to move forward, and `lspeed` and `rspeed` are set to medium speed. The `halt` statement is a shorthand for `loop { next; };`, i.e. it keeps control forever. This behaviour is cancelled if one of the bumpers hits an obstacle, i.e. one of the sensors is present. If, for instance, `sensorA` is present, then the direction of both the motors is set to backward, the speed of the left motor is set to fast, and that of the right motor to slow. The rover moves backward with a left turn. Then a data method is called which computes the time the rover should move backward, and rover moves backward for exactly this time (due to the `await moveback_time`, cf. Section 2.7). The behaviour is iterated due to the outer control loop. Note that "move back time" is increased up to 4 seconds.

```
class TravelAround {

static final time timing = 500msec;

// constants for servo motor control
static final uint8 forward  = 1;
static final uint8 backward = 0;
static final uint8 slow     = 10;
```

```
static final uint8 medium   = 100;
static final uint8 fast     = 200;

public static void main (String[] args) {
   while (instant() == 0) {};
};

Sensor sensorA = new Sensor(new SimInput()); // bumper
Sensor sensorC = new Sensor(new SimInput());

Signal<uint8> ldir   =
          new Signal<uint8>(new SimOutput()); // servo motors ...
Signal<uint8> rdir   =
          new Signal<uint8>(new SimOutput()); // ... directions
Signal<uint8> lspeed =
          new Signal<uint8>(new SimOutput()); // servo motors ...
Signal<uint8> rspeed =
          new Signal<uint8>(new SimOutput()); // ... speed

public TravelAround () {
   active {
      loop {
         cancel {
            emit ldir(forward); // move straight forward ...
            emit lspeed(medium);
            emit rdir(forward);
            emit rspeed(medium);
            halt;                // ... until ...
         } when (?sensorA) {     // ... hit at bumper A
            emit ldir(backward); // backward with left turn
            emit rdir(backward);
            emit lspeed(fast);
            emit rspeed(slow);

         } else when (?sensorC) {// ... hit at bumper C
            emit ldir(backward); // backward with right turn
            emit rdir(backward);
            emit lspeed(slow);
            emit rspeed(medium);
         };

         upd_moveback_time ();
              // compute time to move backward
         await moveback_time;
             // move backward for 1,2,3,4 sec ...
```

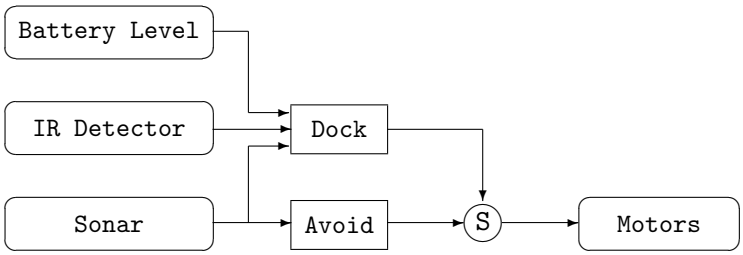
```
                // ... to avoid to be stuck in a corner ...
                // ... then return to forward move
          };
       };
    };

    static void upd_moveback_time() {
      if (moveback_time < 4sec) {
         moveback_time += 1sec;
      } else {
         moveback_time = 0sec;
      };
    };

    static time moveback_time = 0sec;

    }
```

(cf. `chpt_3_travel-around.se`)

3.5 Subsumption Architecture

Subsumption. Subsumption has been introduced by Rodney Brooks and the Mobile Robot Group at MIT as an alternative to the traditional modelling/planning approach used in AI. Subsumption combines real-time control with sensor-triggered behaviour. Behaviours are considered as layers of a control system that run in parallel driven by input provided by the sensors. Conflicting sensor information then results in conflicting behaviours. These are resolved by using priority rules to enable a dominant behaviour to take control.

Here is a simple example of a subsumption



A robot is supposed to follow an IR beacon to find a docking station for loading energy. This "docking" behaviour is in conflict with an "object avoidance" behaviour if the robot gets close to the docking station. In that

case the docking behaviour should dominate the object avoidance behaviour. This is achieved by adding a *suppressor node*, represented by the `S` in the circle, to the actuator wire for the motor. The idea is that messages may pass through the node from the object avoidance behaviour to the motor as long as this signal is not suppressed by the prioritised behaviour indicated by the arrow head, here the docking behaviour.

A rather simple realisation of a suppressor node uses valued signals `dock` and `avoid` that are issued by the respective behaviours with an appropriate value for the motor control. Subsumption is expressed by

```
 sustain { if (?avoid) { emit motor($avoid); };
if (?dock) { emit motor($dock); }; };
```

Note that by the order of evaluation the signal `motor` is emitted with value `$dock` if both the signals `avoid` and `dock` are present. The suppressor node is put in parallel with the behaviours.

The rug warrior. In [22] the design of a small robot, the *rug warrior* is presented, and some examples are given of how to program such a robot using a *subsumption*. We shall demonstrate how *synERJY* may be used to encode such a program using a cut-down version of the rug warrior (cf. [22], p. 259)

The rug warrior consists of several "behaviour modules" which will be implemented as reactive methods.

- The simplest behaviour is that of *cruising*.

  ```
  void cruise () {
     sustain {  // rug warrior goes forward
         emit cruise(forward);
     };
  };
  ```

  The output signal `cruise` is emitted with value `forward`.

- The *follow* behaviour implements light-source following is slightly more complicated.

  ```
  void follow () {
     sustain {
        if ($right_photo - $left_photo > photo_dead_zone) {
            if ($right_photo - $left_photo > 0) {
  ```

```
                                // light on left, turn left
                emit follow(left_turn);
            } else {
                                // light on right, turn right
                emit follow(right_turn);
            };
        };                      // else, no difference
    };
};
```

If a difference is detected between the left and the right photocell, and if the difference is above a threshold it will turn the robot in the direction of the brighter side. In that case it will emit the local signal `follow` with an appropriate value. Otherwise it will do nothing.

- The avoid behaviour gets input from an infrared proximity sensor.

```
void avoid () {
   sustain {
      if ($ir_detect == 0b11) {          // both IR see something
          emit avoid(left_arc);
      } else if ($ir_detect == 0b10) { // left IR sees something
          emit avoid(right_arc);
      } else if ($ir_detect == 0b01) { // right IR sees something
          emit avoid(left_arc);
      };                                // neither sees something
   };
};
```

- The escape behaviour is meant to allow the robot to escape when the bump sensors have found an obstacle.

```
void escape () {
  loop {
     if (?bump_left && ?bump_right) {
         emit escape(backward);
         await 2sec;
         emit escape(left_turn);
         await 4sec;
     } else if (?bump_left) {
         emit escape(right_turn);
         await 2sec;
     } else if (?bump_right) {
         emit escape(left_turn);
```

```
            await 4sec;
        } else if (?bump_back) {
            emit escape(left_turn);
            await 2sec;
        };
        next;
    };
  };
```

The only difference in terms of behaviour is that we have changed the timing constants for a simulation with fewer steps. Outputs and flags as specified in the original source code are subsumed by a signal, e.g.

```
follow_output & follow_output_flag -> follow_sgn
```

Arbitration is modelled as outlined above. The escape behaviour suppresses all other behaviours, the avoid behaviour suppresses all other behaviours except for the escape behaviour, and so on

```
void arbitrate () {
   sustain {
      if (?cruise) { emit motor_input($cruise); };
      if (?follow) { emit motor_input($follow); };
      if (?avoid)  { emit motor_input($avoid); };
      if (?escape) { emit motor_input($escape); };
   };
};
```

All these behaviours are executed in parallel

```
public RugWarrior () {
    active {
       [[ cruise();
       || follow();
       || avoid();
       || escape();
       || arbitrate();
       ]];
    };
};
```

Chapter 4

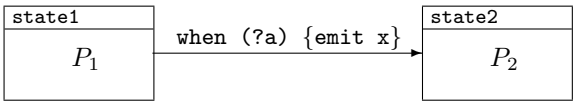
Hierarchical State Machines

STATECHARTS [18] is a well used visual formalism for hierarchical state machines which recently became the notation for presenting behaviour in UML. There has been some dispute about its semantics. We use the synchronous semantics of SYNCCHARTS [1] that coincides with that given earlier to ARGOS [26].

However, though we share the view that the synchronous semantics has its benefits, a minor trick allows us to extend the synchronous semantics to capture the original STATEMATE semantics of STATECHARTS: we introduce a particular kind of delayed signals that, if emitted at one instant, become present only at the next instant.

4.1 States and Preemption

States as processes. Automata is another familiar paradigm for both, engineers and computer programmers. Automata have states and transitions between these. A faintly new idea is to consider *states as processes*. A state may be *active* or *inactive* as is the corresponding process. The process is started when a state is entered, it is active as long as the automaton is in that state, and it is preempted when the state is left. For example, the behaviour of

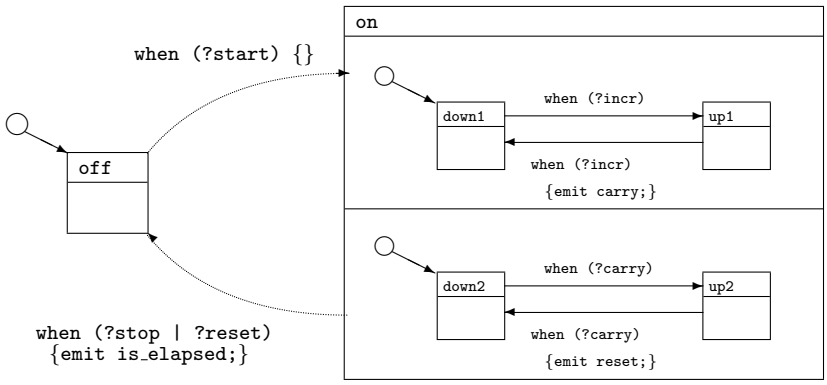


is this: assume that the process P_1 is running if control is in `state1`. This process is preempted if the signal `a` is present, the signal `x` is emitted, and,

at the same instant, control passes to `state2`, and the corresponding process P_2 is started.

In this interpretation, automata are just another notation for structuring processes. The implicit recursive definition justifies to speak of *hierarchical automata*; the process of a state can be again specified by an automaton. Obviously, any other specification of a process in terms of the any of other language constructs introduced so far might do as well.

Visual versus textual. A visual presentation is attractive if you deal with a customer. For a programmer, textual notation may be more condensed and more easily modified. Here is the same example in visual and textual presentation.



Here is the textual representation:

```
automaton {
  init { next state off; };
  state off
    when (?start) { next state on; };
  state on
    do {
      [[ automaton {
           init { next state down1; };
           state down1
             when (?incr) { next state up1; };
           state up1
             when (?incr) { emit carry; next state down1; };
         };
      || automaton {
           init { next state down2; };
           state down2
```

```
            when (?carry) { next state up2; };
          state up2
            when (?carry) { emit reset; next state down2; };
        };
      ]];
    }
    when (?stop || ?reset) { emit elapsed; next state off; };
};
```

(cf. `chpt_4_four-bit-automaton.se`) The automaton has two states `off` and `on`. The `init` statement is *instantaneous*, that is, it terminates in the same instant it is started. Hence, if the process/automaton is started it moves on to the state `off` and starts the process `halt`. In *synERJY*, automata have initial transitions rather than initial states. The equivalent in the visual notation is a transition with a blob rather than a state in its source.

A state can be left only when the condition of some outgoing transition becomes true, but a state can be left only in the next instant after it has been activated or after the next instant. This mean that a state is "active" for at least one instant.[1] The process of the state `off` is specified by the statement `do { nothing }` which does nothing (hence the statement may be omitted for convenience). Via the statement `do` $\{P\}$ any process P can be embedded.

If the `start` signal is present in the next instant or later, the automaton moves to state `on`, and starts two processes, again presented by automata. `when ?start { next state on}` is the textual phrase for the transition with obvious connotation.

Preemption by a transition is weak. Rather than to carry on, we focus on the particular situation when the automaton is in state `on`, the subautomata are in state `up1` resp. `up2` and the signal `incr` is present. Then the first subautomaton performs the transition and emits `carry`. At the same instant, the other subautomaton performs its transition as well since the signal `carry` is present, and emits `reset`. Still at the same instant, the process of state `on` is preempted, and the automaton moves to state `off` emitting `is_elapsed`. Note that, though the actions of the transitions are executed, the subautomata do not change state to `down1` resp. `down2` due to preemption of state `on`.

Preemption must be *weak* in that the process of state `on` must be allowed to do all its computation in this instant, otherwise the signal `reset` will

[1] In that preemption is delayed (cf. Section 2.2).

not be emitted. Preemption takes place from the inside, which would be impossible if preemption would be strong. Note further that the process can be preempted from the outside by the signal `stop`.

4.2 Textual Presentation

The textual syntax for automata is very simple. The keyword

```
automaton { P };
```

indicates that the process P is presented by an automaton. The specification of a *state* is of the form

```
state name
    [ do     { P } ]
    [ entry  { P_entry } ]
    [ during { P_during } ]
    [ exit   { P_exit } ]
           when C_1 { P_1 }
    [ else when C_2 { P_2 } ]
    ...
    [ else when C_n { P_n } ];
```

All the clauses in square brackets are optional. It is required that all processes except for P are instantaneous. [2]

The behaviour is as follows. When entering a state the processes P and P_{entry} are started. The processes P and P_{during} are active as long as the state is active. When a condition C_i becomes true, the process P is weakly preempted and the process P_{exit}. Finally the process P_i are started when P_{exit} terminates. At each instant, the process P executes before checking the conditions C_i that are checked in the obvious order.

The *initial transition* has a similar format

```
init {P}
```

with P being instantaneous.

Finally, the statement

```
next state state_name;
```

denotes the (instantaneous) jump to the next state. A particular form of the statement is the *next state exit* statement

[2] The pattern for is similar to that for preemption operators. This is not by chance. In the implementation there is a unique preemption operator used for all these cases.

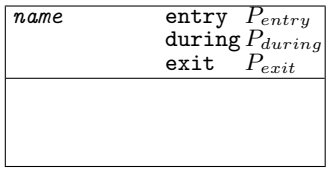
```
next state exit;
```

If this process is started the respective automaton terminates instantaneously.

4.3 Visual Presentation

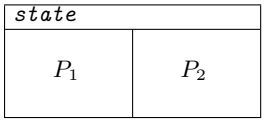
The visual syntax is close to that of STATECHARTS [18] and SYNCCHARTS [1]. Automata have nodes and transitions. Nodes are either persistent states or transient states. *Transient states* of the form of a circle are used for initial and terminal transitions, and for conditionals.

Persistent states are of the form

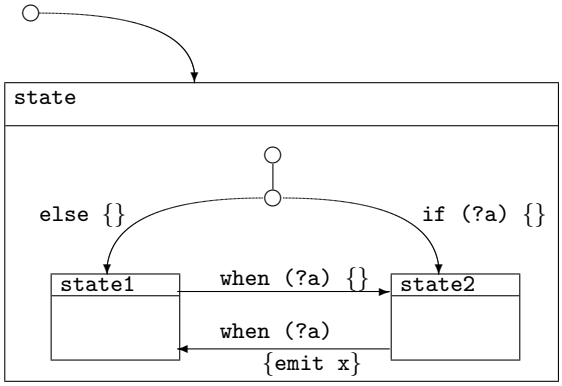


The same conditions apply as for the obvious textual equivalent. The body P of a visual state is (at present) restricted to a few formats (the restrictions apply recursively):

- the *parallel operator* indicated by horizontal or vertical lines (*and-states* in the terminology of statecharts)

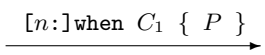


- (the visual presentation of) an automaton, e.g.



Transitions are of the form

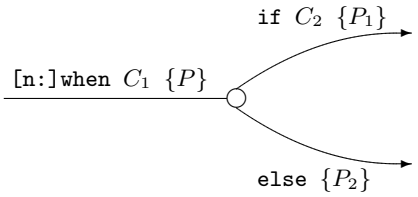
- A *simple* transition



  The priority n: is optional in case that only one transition leaves a node. Otherwise the number n indicates the priority of a transition. Highest priority is 1. The conditions of the transitions are evaluated according to priority. The textual equivalent is

```
     when C1 { P1 }  // prority 1
else when C2 { P2 }  // prority 2
...
else when Cn { Pn }; // prority n
```

- a *conditional transition*



  The conditional may be iterated. If the condition C_1 becomes true, the transition "fires"; first the action P is executed, then the condition C_2 is evaluated, and then one of the branches according to whether C_2 evaluates to true or false. This is equivalent to the textual clause

```
when C1 { if C2 {P1} else {P2} }
```

  The priority rules apply accordingly.

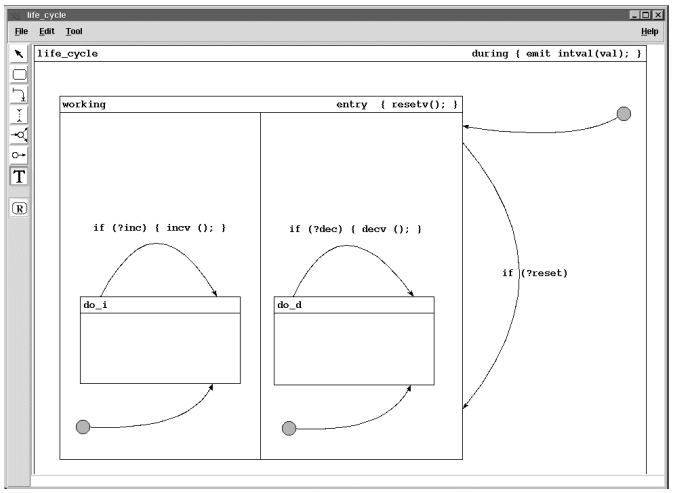
A *visual automaton* has exactly one transient state that is initial and at least one persistent state. There may be several transient states that are final. A transient state is *initial* if there is exactly one outgoing default transition and no ingoing transition. A transient state is *final* if there are only ingoing transitions.

Embedding visual presentations. Visual presentations of hierarchic automata are embedded using the keyword automaton followed by file name (as string), e.g.

```
class VisualIncrDecr {

    public VisualIncrDecr () {
      val = 2;
      active { automaton "chpt_4_inc_dec.sc"; };
    };

    Sensor inc   = new Sensor(new SimInput());
    Sensor dec   = new Sensor(new SimInput());
    Sensor reset = new Sensor(new SimInput());

    Signal<int> intval = new Signal<int>(new SimOutput());
    int val;

    void resetv () { val = 0; };
    void incv   () { val++;   };
    void decv   () { val--;   };
```

(cf. `chpt_4_visual-incr-decr.se`) It is recommended that all textual and all graphic files of an application share one directory. Graphic files should have the suffix "`.sc`".

Visual presentations are designed using the *synERJY* Graphic Editor. Here is a simple example



4.4 *Approximating* STATECHARTS *Semantics.*

Emittance of signals is always delayed in STATECHARTS. *Hence delayed signals as introduced in Section 2.2 may be used to approximate the* STATECHARTS *semantics.*

There has been a certain amount of discussion whether the statecharts interpretation is truely synchronous. Rather than to join the flame, we believe that both signals and delayed signals have their merits and their shortcomings. Delayed signals cut causality cycles but a "reaction" may take more than one instant. The delays caused by delayed signals may be difficult to compute. Consider, for example, the familiar four bit counter but replacing signals by delayed signals.

```
Sensor    start = new Sensor(new SimInput());
Sensor     incr = new Sensor(new SimInput());
Sensor     stop = new Sensor(new SimInput());
Delayed   reset = new Delayed();
Delayed elapsed = new Delayed(new SimOutput());
Delayed   carry = new Delayed();

public DelayedFourBitAutomaton () {
  active {
    automaton {
      init { next state off; };
      state off
         when (?start) { next state on; };
      state on
         do {
           [[ automaton {
                 init { next state down1; };
                 state down1
                    when (?incr) { next state up1; };
                 state up1
                    when (?incr) { emit carry; next state down1; };
              };
           || automaton {
                 init { next state down2; };
                 state down2
                    when (?carry) { next state up2; };
                 state up2
                    when (?carry) { emit reset; next state down2; };
              };
           ]];
         }
```

```
            when (?stop || ?reset) { emit elapsed; next state off; };
        };
      };
    };
```

(cf. chpt_4_delayed-four-bit-automaton.se) Let us assume that we are in the states `up1` *and* `up2`*, and let* `incr` *be present. Then* `carry` *is emitted for the next instant, and the first subautomaton moves to the state* `down1`*. In the next instant, the signal* `carry` *is present, the state of the second subautomaton is change to* `down2`*, and the signal* `reset` *is emitted for the next instant. At the same instant* `incr` *may be present causing a move to state emphppup1. Now in the next instant, the signal* `reset` *is present, the signal* `elapsed` *is emitted for the next instant, and toplevel automaton moves to state* `off`*.*

Of course, the process does not behave like a counter. We like to stress that the behaviour is not so easy to trace if delayed signals are involved, the main disadvantage being that mistakes will only show at run time.

4.5 Examples

4.5.1 The Stopwatch using Automaton

The simple stopwatch. We distinguish two states: the stopwatch being running, and the stopwatch being stopped. The stopwatch toggles between the states if the button `start_stop` is pressed.

```
automaton {
   init { the_time = 0msec;
          next state stopped;
   };

   state stopped
   when (?start_stop) { next state running; };

   state running
      during {
        the_time = the_time + $dt;
        emit elapsed_time(the_time);
      }
   when (?start_stop) { next state stopped; };
};
```

(cf. `chpt_4_simple-stopwatch-automaton.se`) One should note a distinction to the stopwatch as defined in Section 3.2. Since preemption is weak the elapsed

time is emitted if the signal `start_stop` is present in state `running`. It is a design choice whether this should be the case or not. For the latter, we must use strong preemption.

The general stopwatch Analysing the general stopwatch of Section 3.2 we realise that there are two states corresponding to the simple stopwatch, a further toggle between the `running` state, and a state for reset. These are used in the automaton below

```
automaton {
   init { next state reset; };

   state reset
      exit { the_time = 0msec;
             emit elapsed_time(the_time);
             emit display_time($elapsed_time);
      }
   when (?start_stop) { next state running; };

   state stopped
   when (?start_stop) { next state running; }
   else when (?freeze_reset) { next state reset; };

   state running
       during {
         the_time = the_time + $dt;
         emit elapsed_time(the_time);
         emit display_time($elapsed_time);
      }
   when (?start_stop) { next state stopped; }
   else when (?freeze_reset) { next state frozen; };

   state frozen
       during {
         the_time = the_time + $dt;
         emit elapsed_time(the_time);
      }
   when (?freeze_reset) { next state running; };
};
```

(cf. `chpt_4_stopwatch-general-automaton.se`) One should note that – quite unlike to the approach using the "imperative" sublanguage – designing in terms of automata ask for finding states. It seems that in case of the general stopwatch this results in a much clearer design. The transitions between

the states are obvious. however, the same remark applies as in case of the simple stopwatch above in that only weak preemption is used that results in a slightly different behaviour that that of the original stopwatch.

Chapter 5

Synchronous Data Flow

Engineers use difference equations for modelling discrete-time systems. Computer science speaks of synchronous data flow models. The synchronous programming language LUSTRE [19] is based on this model as is SIGNAL [15]. The data flow sub language of *synERJY* borrows much of the syntax and semantics of LUSTRE.

5.1 Discrete Data Flow

Difference equations. A low pass filter removes the higher frequencies in a signal while passing the lower frequencies. A simple low pass filter is, for instance, specified by a *difference equation* such as

$$y[n] = 0.1 * x[n] + 0.9 * y[n-1]$$

The index abstracts time since n really stands for $t = nT$ where T is an interval of time. Thus the equation is a shorthand for

$$y[nT] = 0.1 * x[nT] + 0.9 * y[(n-1)T]$$

which should be read as "at time nT the value of of y is equal to the sum of the the value of x at time nT multiplied by 0.1 and of the value of y at time $(n-1)T$ multiplied by 0.9". : Synchronous programming speaks of data flow equations rather than of difference equations. There is a shift in semantics as well: x and y are considered just as sequences of data with the value of y being determined from that of x in discrete steps by applying the difference equation. No timing is involved. In particular, no assumptions are made that the intervals in between two steps will be of equal length.

Flow equations abstract from indexes. The $n-1$ is replaced by an explicit operator (`pre` in *synERJY*)

```
y := 0.1 * x -> 0.1 * x + 0.9 * pre(y);
```

The operator `pre(x)` refers to the status of the signal `x` at the *previous* index.[1] The operator `->` (*arrow*) distinguishes between the *first* index – the value being determined by the initialisation `a0 * x` –, and *later* indexes – the value being determined by the expression `a0 * x + a1 * pre(x) + b1 * pre(y)`.

Traces. We will define the semantics of flow equations in terms of traces. Traces have been used in Chapter 2.1 to illustrate reactive behaviour. We will now start to use traces as a basis for the semantics of reactive behaviour.

Let a *trace* be a sequence of data (of the same type). At an instant, a trace may be *accessible* or *inaccessible*. If it is accessible it has a value. For visualisation, we use a diagram of the form

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	...
d		d_0	d_1			d_2		...

The trace d is accessible at an instant if there is an entry d_n. An empty slot marks inaccessibility. The index i indicates the instants.

Standard operators lift to the corresponding traces by defining the operations element-wise at every instant, e.g.

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	...
d		d_0	d_1			d_2		...
d'		d'_0	d'_1			d'_2		...
$d + d'$		$d_0 + d'_0$	$d_1 + d'_1$			$d_2 + d'_2$		...

The definition requires that d, d', and $d + d'$ are accessible at the same instant.

Literals and variables determine traces. For instance, the literal `3` determines a trace that has the value 3 at every instant

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	...
d	3	3	3	3	3	3	3	...

[1] Some may be reminded of the Z-tranform.

Shifting time. The operator $pre(d)$ shifts the trace d in that it pointwise refers to the *previous* element

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	...
d		d_0	d_1			d_2		...
`pre(`d`)`		δ	d_0			d_1		...

The first element is a default value since there is no previous element. δ is a (type dependent, e.g. 0 for integers, $false$ for booleans) default value. The trace `pre(`d`)` is accessible if and only if d is accessible.

Initialising a trace. The arrow operator distinguishes between the very first element of a trace, and all later elements.

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	...
d		d_0	d_1			d_2		...
d'		d'_0	d'_1			d'_2		...
d `->` d'		d_0	d'_1			d'_2		...

The first element of the trace d `->` d' is the first element d_0 of the trace d while the other elements are those of the trace d'. Note again that all the traces should be accessible at the same instants.

Flow equations. The traces corresponding to a flow equation are inductively defined folllowing the structure of flow expressions. Here are two examples. The first example is an incrementing counter defined by the data flow equation

```
count := 0 -> pre(count) + 1;
```

Here `count` is a signal of type `int`. At an instant, the signal may be constrained by the flow equation, i.e. its value is updated to be equal to the value of the expression on the right hand side. Hence the following behaviour is enforced.

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	...	
$count$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	...
$pre(count)$	**0**	*0*	*1*	*2*	*3*	*4*	*5*	*6*	...
$pre(count) + 1$	*1*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	...
0 `->` $pre(count) + 1$	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	...

Let us assume that the signal `count` is updated at every instant (implying that the signal is accessible). Let us say that a signal is present if it is constrained by a flow equation. Presence is indicated by the bold typeface.

In case of the (Boolean) flow expression

```
raising_edge :  = false -> x && !pre(x)
```

we may have the following table (f stands for *false*, and t for *true*)

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	...
x	f	t	t	f	f	t	f	t	...
$pre(x)$	f	f	t	t	f	f	t	f	...
$!pre(x)$	t	t	f	f	t	t	f	t	...
x && ! $pre(x)$	f	t	f	f	f	t	f	t	...
$false$ -> x && $!pre(x)$	f	t	f	f	f	t	f	t	...
$raising\_edge$	**f**	**t**	**f**	**f**	**f**	**t**	**f**	**t**	...

The expression detects what hardware people call a *raising edge*, i.e. the instants when the signal `x` changes from *false* to *true*.

A uniform view of updating. We note that signals my be updated

- either by emitting it as explained in the previous chapter,
- or by constraining it by a flow equation.

Though conceptually quite different, both methods of updating behave equivalently in that

- if a signal is updated, either by emitting or constraining, it is present with a new value (if valued), and in that
- multiple updating, either by emitting or constraining, of a signal at the same instant is a time race causing an error message.

However,

- *flow equations support a richer language* including the time shift operators (and sampling operations, as we will learn in Section 5.6).

5.2 Embedding Data Flow to Control

Flow contexts. Flow equations are only allowed to occur in a particular context of the form

```
{| ... |}
```

We speak of a *flow context*. Its body consists of a sequence of flow equations (and of local signal declarations, see below).

Restricting the occurrence of flow equations to a particular context has the advantage that ambiguity of typing can be avoided: we stipulate that *only flows can occur within a flow context*. Then an expression such as `3 + 5` then has different interpretations within or outside of a flow context.

- Outside of a flow context, the terms `3`, `5`, and `3 + 5` denote integers, and the addition operation operates on integers.
- Within a flow context, the terms `3`, `5`, and `3 + 5` are integer flows, and the addition operator operates on integer flows.
- *Note the operators `pre` and `->` can only be used within a flow context.*

The order of presentation of flow equations within a flow context is irrelevant for evaluation. The flow equation are evaluated by an order imposed by the equations: if a signal is used within a flow expression on some right its value must be computed before the expression can be evaluated (*write-before-read*), except if the signal is "guarded" by a `pre`. In that case, a value is used that has been computed at some previous instant. Here are some schematic examples for illustration:

- The two flow contexts

  ```
  {|  x := 0 -> pre(y);
      y := 2 * x;
  |}
  ```

  and

  ```
  {|  y := 2 * x;
      x := 0 -> pre(y);
  |}
  ```

  behave equivalently; the flow equation for signal `x` must be evaluated before that for signal `y` since it is used on the right hand side of the flow equation for signal `y`.

- The flow equations

```
{|  x := 0 -> y;
    y := 2 * x;
|}
```

  will raise a causality error. We have a cyclic dependency since the flow equation for `x` uses the value of `y` of the same instant, and vice versa. Note that guarding `y` with a `pre` in the equations above breaks the cycle, since the value of `y` of a previous instant is used.

The notation {| ... |} is meant to indicate that the order of presentation of flow equations is not relevant. Of course, for each signal, there should be at most one flow equation in any flow context.

The watchdog example. We rephrase the "watchdog" example of [16]. The watchdog is a device to manage deadlines. There are three Boolean signals `set`, `reset`, and `deadline`. The signals `set` and `reset` switch the watchdog "on" and "off". If the watchdog is "on", and if the signal `deadline` is true, the signal `alarm` is true.

```
{| alarm         := deadline && watchdog_is_on;
   watchdog_is_on := false -> if (set) { true;
                              } else if (reset) { false;
                              } else { pre(watchdog_is_on); }; |};
```

The formulation uses the conditional. A hardware designer might prefer to use Boolean operators only, which yields a (disputably) more elegant solution.

```
{| alarm := deadline && watchdog_is_on;
   watchdog_is_on := false ->
                     set || !reset && pre(watchdog_is_on); |};
```

Note that "setting the watchdog" has precedence over resetting. The precedence is reversed by using

```
{| alarm := deadline && watchdog_is_on;
   watchdog_is_on := false ->
                     !reset && (set || pre(watchdog_is_on)); |};
```

Modes. It is somewhat inherent that the evaluation of flow equations should be sustained for some time. This is achieved by using (a variant of) the `sustain` statement

```
sustain {|
   ...
|};
```

Once started, the `sustain` process never terminates; flow constraints are applied forever. We shall speak of a *mode* (of operation). The idea is that modes persist, usually for a long interval, but modes may be changed if necessary, for instance from a start-up mode to a working mode, or from a working mode to an error mode or maintenance mode, and vice versa.

Being a process like any other, the sustain statement may be preempted and (re-) started as in

```
loop {
    await ?start;
    cancel {
        sustain {|
           count := pre(count) + 1;
        |};
    } when (?stop);
    next;
};
```

A trace of this fragment may look like as follows

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	...
start	.	.	.	*	.	.	.	.	.	.	...
stop	.	.	.	.	.	.	*	.	.	.	...
count	*0*	*0*	*0*	**1**	**2**	**3**	**4**	**5**	*4*	*4*	...

If the signal `start` is present, the flow context starts to be sustained. The flow equation is evaluated at every instant till the signal `stop` is present. Whenever the flow equation is evaluated the signal `count` is updated, hence increased. Then the sustained process terminates instantaneously when the signal `stop` is present.

The presence of the pure signals `start` and `stop` is indicated by an asterisk. The signal `start` is present at the 4th instant, and the flow context is active from the 4th to the 8th instant.

Mode automata. *If we combine state machines with flow equations we may consider states as corresponding to modes. This is the idea of* mode automata *as presented in [25]. The following example has two moses/states, counting upward and counting downward.*

```
emit count(0);
automaton {
    init { next state up; };

    state up  // count upwards
        do { next;
             sustain {| count := pre(count) + 1; |};
        }
    when  ($count > 9) { next state down; };

    state down  // count downwards
        do { next;
             sustain {| count := pre(count) - 1; |};
        }
    when  ($count < 1) { next state up; };
};
```

(cf. chpt_5_up-and-down-counter-automaton.se)

The construction is somewhat clumsy in that we have to use the **next** *to avoid multiple constraints: since preemption is weak, the program may count upward, change state from* up *to* down*, and count downward at the same instant if the* next *would not guard the* sustain *clause. The idea, in fact, would be that the flow equations should be active only if a state is in control but not if control enters a state. Since this phenomenon is quite usual for mode automata, we allow to use flow equations in the* during *clause of a state as in*

```
automaton {
    init { next state up; };

    state up  // count upwards
        during {|  count := pre(count) + 1; |}
    when  ($count > 9) { next state down; };

    state down  // count downwards
        during {| count := pre(count) - 1; |}
    when  ($count < 1) { next state up; };
};
```

(cf. chpt_5_up-and-down-counter-during.se) Then the flow equations are active when being in a state, but not if entering a state. Thus multiple constraints are avoided.

5.3 Flow Contexts and Locality

Local Signals. Signals can be declared within a flow context as local variables. All the rules for local variables apply (cf. Section 2.3) except for exception: the initialisation may be dropped, i.e. the shorthand `Signal<`T`>` may be used instead of `Signal<`T`> = new Signal<`T`>();`. The scope of the signal are all the subsequent statements within the flow context.

Of course, using local variables may substantially change behaviour. We reconsider

```
Signal<int> count = new Signal<int>();
loop {
    await ?start;
    cancel {
        sustain {|
           count := pre(count) + 1;
       |};
    } when (?stop);
    next;
};
```

If the `sustain` statement is active, the signal `count` is increased, otherwise its value persists. A typical trace is

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	...
start	.	*	.	.	.	.	*	.	.	.	...
stop	.	.	.	*	.	.	.	*	.	.	...
count	*0*	**1**	**2**	**3**	*3*	*3*	**4**	**5**	*5*	*5*	...

If the signal `count` is declared locally within the flow context, a new incarnation of `count` is generated when control enters the flow context.

```
loop {
    await ?start;
    cancel {
        sustain {|
           Signal<int> count;
           count := pre(count) + 1;
       |};
```

```
    } when (?stop);
    next;
};
```

The corresponding trace is

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	...
start	.	*	.	.	.	.	*	.	.	.	...
stop	.	.	.	*	.	.	.	*	.	.	...
$count'$		**1**	**2**	**3**							...
$pre(count')$		*0*	*1*	*2*							...
$pre(count') + 1$		*1*	*2*	*3*							...
$count''$							**1**	**2**			...
$pre(count'')$							*0*	*1*			...
$pre(count'') + 1$							*1*	*2*			...

Why is this so? Whenever the flow context is started, a new incarnation of the signal `count` is created. Since this incarnation is clearly not accessible at previous instants, `pre(count)` is initialised with a default value (here 0).

The `pre` operator and locality. The operator `pre` may be applied to an expression as in

```
loop {
    await ?start;
    cancel {
        sustain {|
            count := pre(count+1);
        |};
    } when (?stop);
    next;
};
```

(cf. `chpt_5_counting-up-4.se`) We consider the expression `pre(count + 1)` as a shorthand for

```
Signal<int> aux;
aux   := count + 1;
count := pre(aux);
```

(cf. `chpt_5_counting-up-5.se`) with the local signal name `aux` being chosen appropriately. This implies that the value of the expression is set to a default value when starting the flow context. The corresponding traces look like this

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	...
start	.	*	.	.	.	.	*	.	.	.	...
stop	.	.	.	*	.	.	.	*	.	.	...
count	*0*	**0**	**1**	**2**	*2*	*2*	**0**	**1**	*1*	*1*	...
$count + 1$		*1*	*2*	*3*			*1*	*2*			...
$pre(count + 1)$		*0*	*1*	*2*			*0*	*1*			...

The result may be *surprising*. The general idea is that expressions within a flow context are accessible only if the flow context is active. Otherwise the expressions – hence the result of is evaluation – is inaccessible.

Since this may be confusing we issue a general warning:

- ***All usages of the operator*** `pre` ***should be properly guarded by an initialisation using the arrow operator.***

The arrow operator and locality. Initialisation by an arrow operator takes place *whenever a flow context is started*. Consider

```
loop {
    await ?start;
    cancel {
        sustain {|
           count := 0 -> pre(count) + 1;
        |};
    } when (?stop);
    next;
};
```

(cf. `chpt_5_counting-up-6.se`) with the signal `count` being globally defined. Whenever `start` is present the flow context is started, and the signal count is initialised to 0.

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	...
start	.	*	.	.	.	.	*	.	.	.	...
stop	.	.	.	*	.	.	.	*	.	.	...
count	*0*	**0**	**1**	**2**	*2*	*2*	**0**	**1**	*1*	*1*	...

Remember that without initialisation the counter is increased steadily whenever the flow context is active.

Remark. One may be seduced by as expression such as

```
x := 0 -> (1 -> pre(x) + 1 )
```

to believe that the value of `x` is 0 at the first instant when starting a flow context, and 1 at the second. However, the definition says that, whatever the trace on the right hand side of `0 -> ...` is, the value is 0 at the first instant. Hence the flow equation above behaves equivalently to

```
x := 0 -> pre(x) + 1
```

Hybrid systems. *We use "localised" versions of the pre and the arrow operator to support the specification of hybrid systems. Hybrid systems are characterized by the interaction of continuous parts, governed by differential or difference equations (difference equations only in our context), and by discrete parts, described by finite state machines, if-then-else rules, propositional and temporal logic. Hybrid systems switch between many operating modes where each mode is governed by its own characteristic dynamical laws. Mode transitions are triggered by variables crossing specific thresholds (state events), by the elapse of certain time periods (time events), or by external inputs (input events). Further it is usually required that each mode starts operating with defined initial conditions specified by a reset relation.*

In synERJY, all "continuous" modes are encapsulated by flow context, while all the other language constructs specify the discrete parts resp. the transitions. Now having local initialisation by the arrow operators provides the means to specify a reset relation. The initial condition can depend on the status of (globally declared) signals at a previous instant that is accessed by using the operator `pre`*. This is the sort of rationale for our "localised" interpretation of the operators* `->` *and* `pre`*.*[2]

To give a simple example of a typical presentation of a hybrid systems we consider a bouncing ball with the following properties:

- *Motion is characterised by* height *(x_1) and* vertical *velocity (x_2),*
- *Continuous changes between bounces.*
- *Discrete change at bounce time.*
- *Dynamics summarised by*

[2]Note that this generalises the use of these operators in LUSTRE. LUSTRE programs have – in our terminology – only one mode. Hence initialisation by the arrow operator can take place only in the very first instant, as well as the operator `pre` has a default value only in the first instant (forgetting about , see Section 5.6).

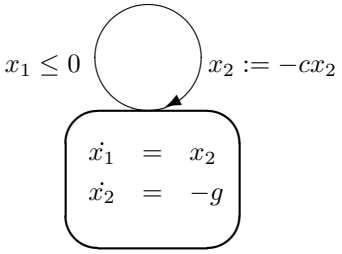
– *one* mode q *with a continuous behaviour specified by the equations*

$$\begin{aligned} \dot{x_1} &= x_2 \\ \dot{x_2} &= -g \end{aligned}$$

– *one* transition *from* q *to* q *guarded by the condition* $h \leq 0$,

– *a reset relation that keeps the height but reverses the direction of velocity and decreases it by a factor in that* x_2 *is set to* $-cx_2$.

The behaviour is graphically specified by



If we try to translate this to a program, the obvious is to replace an differential equation $\dot{x} = e$ *is replaced by an integral* $x = x_0 + \int e dx$, *and to compute the integral using difference equations such as, for instance:*

- *(forward) Euler*

$$\begin{aligned} x_1(n) &= x_1(n-1) + x_2(n-1) \times dt \\ x_2(n) &= \begin{cases} -c * x_2(n-1) & -g * dt & \text{if } x_1 \leq 0 \\ x_2(n-1) & -g * dt & \text{otherwise} \end{cases} \end{aligned}$$

- *or backward Euler*

$$\begin{aligned} x_1(n) &= x_1(n-1) + x_2(n) \times dt \\ x_2(n) &= \begin{cases} -c * x_2(n-1) & -g * dt & \text{if } x_1 \leq 0 \\ x_2(n-1) & -g * dt & \text{otherwise} \end{cases} \end{aligned}$$

Another choice maybe Runge-Kutta. For the example, backward Euler works. A corresponding synERJY program is

```
automaton {
   init { emit x1(height);
          emit x2(0.0);
          next state move; };
```

```
    state move
       during {| x1 :=  pre(x1) + x2*((double)dt);
                x2 :=  (-c*pre(x2) -> pre(x2)) - g*((double)dt);
       |}
    when ($x1 <= 0.0) { next state  move; };
};
```

(cf. `chpt_5_bouncing-ball.se`) *Whenever the state* `move` *is entered, the speed is reversed. Note that forward Euler would fail: whenever the direction switches, the speed is negative before the switch, and positive after. For backward Euler, we would compute* `x1 := pre(x1) + pre(x2) * dt` *after the switch, but with* `pre(x2)` *which is the negative value of speed before the switch. Hence, the height would decrease rather than increase after the switch, actully be less than* 0 *again, causing another switch. Hence the height toggles about* 0 *while the speed decreases, which does not quite reflect the physics.*

Warning *If compared with simulation tools such as Matlab/Simulink or Scilab/Scicos one should note that in case of "zero-crossings" resolution of computation cannot be improved for better results: the dt provides the finest resolution. One should keep here in mind that the language is build for hard real-time control. Hence we cannot artificially "stretch time" as can be achieved for offline simulators.*

Hence one should not use an exact value such as 0.0 *for changing direction but rather some* ϵ *ball, meaning replacing o.o by a small positive value.*

5.4 Examples

5.4.1 Using Flows for the Stopwatch

The simple stopwatch. The following fragment rephrases the simple stopwatch of Section 3.2 using flow equations

```
Sensor<bool> start_stop = new Sensor<bool>(new SimInput());
Sensor<bool>      reset = new Sensor<bool>(new SimInput());

Signal<time> display_time = new Signal<time>(new SimOutput());
Signal<bool>      running = new Signal<bool>(new SimOutput());

public SimpleStopwatch () {
  active {
```

```
      sustain {|
        display_time :=
           0msec -> running ?
                        pre(display_time) + 10msec
                      : reset ?
                           0msec
                         : pre(display_time) ;
           running := false ->
                      start_stop ? ! pre(running) : pre(running);
      |};
    };
  };
```

(cf. `chpt_5_flow-simple-stopwatch.se`) The signal `running` is toggled if the sensor `start_stop` is true. The displayed time is increased if in the running state, reset to 0 if the sensor `reset` is true, and otherwise kept unchanged.

The general stopwatch. The general stopwatch permits to record intermediate times. As in Section 3.2, we introduce a signal `elapsed_time` that behaves like the signal `display_time` of the simple stopwatch. For the general stopwatch the signal `display_time` remains constant if the stopwatch is in state "frozen". The signal `reset` has two functionalities: (i) if the stopwatch is stopped but not frozen then the time is reset to 0 (by the signal `actual_reset`, and (b) it toggles between the states "frozen" and "not froozen".

```
Sensor<bool> start_stop = new Sensor<bool>(new SimInput());
Sensor<bool>      reset = new Sensor<bool>(new SimInput());

Signal<time> display_time = new Signal<time>(new SimOutput());
Signal<bool>      running = new Signal<bool>(new SimOutput());
Signal<bool>       frozen = new Signal<bool>(new SimOutput());

Signal<time> elapsed_time = new Signal<time>();
Signal<bool> actual_reset = new Signal<bool>();

public SimpleStopwatch () {
  active {
    sustain {|
      elapsed_time :=
         0msec -> running ?
                      pre(elapsed_time) + dt
                    : actual_reset ?
                         0msec
```

```
                        : pre(elapsed_time) ;
      running := false ->
                      start_stop ? ! pre(running) : pre(running);
      frozen := false ->
                       if (reset && pre(running)) {
                          true;
                       } else if (reset && pre(frozen)) {
                          false;
                       } else {
                          pre(frozen);
                       };
      display_time :=
         0msec -> if (frozen) {
                        pre(display_time);
                     } else {
                        elapsed_time;
                     };
      actual_reset := reset && pre( !running && !frozen);
    |};
  };
};
```

(cf. `chpt_5_flow-simple-stopwatch-with-intermediate-time.se`)

The general stopwatch as a hybrid system. *The logic of the general stopwatch is quite intriguing and needs careful analysis for a proper understanding. This is due to the fact that the effect of sporadic control features such as pushing the* `start_stop` *button or the* `reset` *button are implemented using logic. Using hybrid systems provides a much clearer design*

```
Sensor   start_stop = new Sensor(new SimInput());
Sensor freeze_reset = new Sensor(new SimInput());

Signal<time> display_time = new Signal<time>(new SimOutput());
Signal<time> elapsed_time = new Signal<time>();

public HybridStopwatch () {
  active {
    automaton {
       init { next state reset; };

       state reset
          during {|
             elapsed_time := 0msec -> pre(elapsed_time);
             display_time := elapsed_time;
```

```
            |}
         when (?start_stop) { next state running; };

         state stopped
            during {|
               elapsed_time := pre(elapsed_time);
               display_time := elapsed_time;
            |}
         when (?start_stop) { next state running; }
         else when (?freeze_reset) { next state reset; };

         state running
             during {|
               elapsed_time := pre(elapsed_time) + dt;
               display_time := elapsed_time;
            |}
         when (?start_stop) { next state stopped; }
         else when (?freeze_reset) { next state frozen; };

         state frozen
             during {|
               elapsed_time := pre(elapsed_time) + dt;
               display_time := pre(display_time);
            |}
         when (?freeze_reset) { next state running; };
      };
   };
};
```

(cf. chpt_5_flow-hybrid-stopwatch.se) In state `running` *both the internal and the displayed times are increased. In state* `frozen` *only the internal time is increased. One realizes that the stopwatch can be reset to* `0sec` *only if the process is neither in state* `running` *nor in state* `frozen`*. Further if being stopped the* `start_stop` *button has precedence over the* `reset` *button. In the pure flow version this is rather implicit in that for evaluating the flow equation for the internal time the* `actual_reset` *condition is only checked if the system is not running.*

5.4.2 The Train Example Reconsidered.

We rephrase the single line train example 3.3 using data flow. We replace all the input and output signals by Boolean signals. There are two additional Boolean signals `change` and `priority`. The signal `change` computes any change situation, and signal `priority` the priority of trains as the name

suggests. The priority is computed only if a change of situation takes place in that a new train arrives or a train leaves the single track. To compute the change and the priority there are three more auxiliary Boolean signals `trainWaiting`, `trainArriving`, and `lineFreed`. The latter is true at the instant a train leaves the single line. The meaning of the other two are obvious.

```
sustain {|
  Signal<bool> change;
  Signal<bool> priority;
  Signal<bool> trainWaiting;
  Signal<bool> trainArriving;
  Signal<bool> lineFreed;
  priority       := if (change) {
                       leftTrainWaiting && rightTrainWaiting;
                    } else if (! pre(priority)) {
                       leftTrainWaiting;
                    } else {
                       pre(priority);
                    };
   change        :=   isLineFree && trainArriving
                    || lineFreed && trainWaiting;
   trainWaiting  := leftTrainWaiting || rightTrainWaiting;
   trainArriving := false -> trainWaiting && ! pre(trainWaiting);
   lineFreed     := false -> isLineFree && ! pre(isLineFree);
   leftPointingToIn   := priority;
   rightPointingToOut := priority;
   rightPointingToIn  := ! priority;
   leftPointingToOut  := ! priority;
   leftSignal   :=
       false -> leftTrainWaiting && isLineFree &&
                  leftPointingIsIn && rightPointingIsOut;
   rightSignal  :=
       false -> rightTrainWaiting  && isLineFree &&
                  rightPointingIsIn && leftPointingIsOut;
|};
```

The overall behaviour should be easy to grasp considering the explanations given in Section 3.3. The pointings are set according to the priority, and the traffic lights are set to green when the obvious conditions hold.

5.4.3 The LEGO Example using Flows

We present a flow variant of the robot example in Section 3.4. The wheeled robot is able to move forward and backward, and to turn left and right. Two

bumper sensors on the left and right front end are used to detect obstacles. The task is to control the robot such that it travels around without being caught, for instance, in some corner.

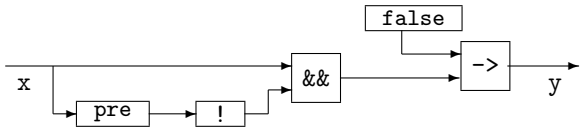
```
nothing;
sustain {|
   delta  := 0sec -> (pre(delta) > 2sec) ? 0sec : (pre(delta) + dt);
   swap   := false -> delta > 2sec;
   right  := true  -> swap ? (! pre(right)) : pre(right);
   left   := ! right;

   left_coll  := sensing & ? sensorA;
   right_coll := sensing & ? sensorB & (! left_coll);
   collision  := left_coll | right_coll;
   time_coll  := 0sec  -> collision ? now : pre(time_coll);
   duration   := 1sec  -> (collision ? 1sec : 0sec)
                                        + pre(duration);
   sensing    := false -> pre(time_coll + duration < now);

   l_speed := left_coll         ?  30 :
              right_coll        ? 180 :
              (sensing & right) ? 200 :
              (sensing & left)  ?  10 : pre(l_speed);
   r_speed := left_coll         ? 180 :
              right_coll        ?  30 :
              (sensing & right) ?  10 :
              (sensing & left)  ? 200 : pre(r_speed);
   l_dir   := sensing & (! collision) ? 1 : 2;
   r_dir   := l_dir;
|};
```

(cf. `chpt_5_travel-flow.se`)

The sensors are encoded by the sensors `lsensor` and `rsensor`. The motor actuators are `ldir`, `rdir`, `lspeed`, and `rspeed`.

5.5 Reusing Data Flow Equations

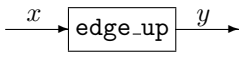
Block diagrams. Engineers often prefer a visual presentation for data flows, e.g.



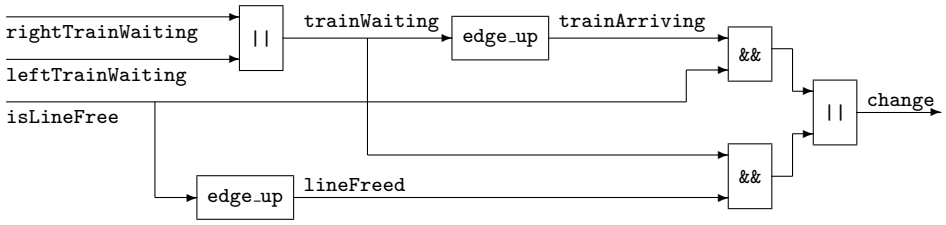
The boxes represent operators (with constants being an operator of arity 0), and arrows represent flows of data between these. The behaviour corresponds to that of the flow equation for finding a raising edge

```
y := false.. -> x && ! pre(x);
```

Typically such block diagrams are abstracted by boxing in the diagram as in



The new box may be used in other diagrams as a shorthand as in the following fragment of the train example in Section 3.3



Block diagrams provide a natural abstraction mechanism for structuring and for reuse in the world of data flow corresponding to method calls in an object-oriented world.

A textual presentation. Looking for a textual representation of block diagrams, we observe that

- a block is specified in terms of
  - ingoing and outgoing flows, and
  - a body being a flow context and that
- a block is embedded in a diagram by connecting the flows inside and outside of the block.

The mechanism we use resembles that of reactive methods, but the body only consist of a flow context, e.g

```
node edge_up (Sensor<boolean> _x,Signal<boolean> _y) {|
   _y := false -> _x && ! pre(_y);
|};
```

We speak of *nodes* rather than of blocks, hence the keyword. For connection, sensors and signals are passed as parameters as in

```
edge_up(isLineFree,LineFreed)
```

Calling the node within a flow context then "glues" the boxes.

```
change        :=    isLineFree && trainArriving
                 || lineFreed && trainWaiting;
trainWaiting  := leftTrainWaiting || rightTrainWaiting;
edge_up(isLineFree,lineFreed);
edge_up(trainWaiting,trainArriving);
```

When compiling, each node call is expanded by replacing the parameters by the arguments and by adding its flow equations to the enclosing flow context. In case of the train example the code to execute is

```
change        :=    isLineFree && trainArriving
                 || lineFreed && trainWaiting;
trainWaiting  := leftTrainWaiting || rightTrainWaiting;
lineFreed     := false -> isLineFree && ! pre(lineFreed);
trainArriving := false -> trainWaiting && ! pre(trainArriving);
```

Relational abstraction. The general scheme is to have boxes with m ingoing flows and n outgoing flows

The corresponding textual counterpart adds type information

```
node xxxx ( Sensor<T1>   x1 , ... , Sensor<Tm>   xm ,
            Signal<Tm+1> y1 , ... , Signal<Tm+n> yn ) {|

  ...

|};
```

The body is restricted to be a flow context.

A node specifies a relation for constraining flows, just as flow equations do. Hence a node may be only called within a flow context. If the flow context is active at an instant, the constraints of the flow equations apply as well as those that are specified by a node called.[3]

[3]Here, *synERJY* substantially differs from LUSTRE that uses functional abstraction.

The stopwatch again. The stopwatch of Section 5.4.1 is an example for using a flow method with multiple input and output flows. The simple stopwatch becomes the flow method

```
node simple_stopwatch ( Sensor<bool> _start_stop,
                        Sensor<bool> _reset,
                        Signal<time> _time,
                        Signal<bool> _running     ) {|
  _time := 0msec -> _running ?
                        pre(_time) + dt
                      : _reset ?
                        0msec
                      : pre(_time) ;
  _running := false ->
                _start_stop ? ! pre(_running) : pre(_running);
|};
```

This method is called within a flow context as follows

```
public Stopwatch () {
  active {
    sustain {|
      frozen := false ->
                      if (reset && pre(running)) {
                         true;
                      } else if (reset && pre(frozen)) {
                         false;
                      } else {
                         pre(frozen);
                      };
       displayed_time := 0msec -> if (frozen) {
                                     pre(displayed_time);
                                  } else {
                                     internal_time;
                                  };
      actual_reset := reset && pre( !running && !frozen);
      simple_stopwatch(start_stop,actual_reset,internal_time,running);
    |};
  };
};
```

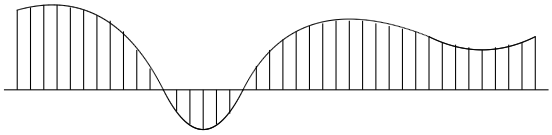
(cf. chpt_5_flow-stopwatch-general.se)

5.6 *On Clocks*

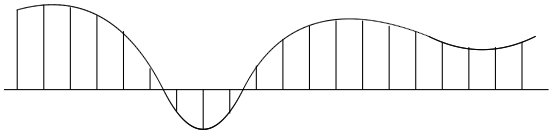
Sampling signals. *Execution of a flow equations consumes real time. Hence one may sometimes want to avoid executing a data flow equation at every instant. Let, for example, the input x of the filter equation*

$$y(n) = a_0 * x(n) + a_1 * x(n-1) - b_1 * y(n-1),$$

be given by a curve such as



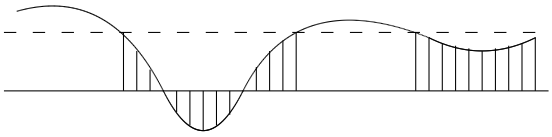
For reasons of efficiency, one want to compute the filter equation only every other instant



The filter equation turns into

$$y(2*n) = a_0 * x(2*n) + a_1 * x(2*(n-1)) - b_1 * y(2*(n-1))$$

The formalisation gets messy though if we want to sample the curve only if its amplitude is below a certain threshold as in



synERJY provides a very simple mechanism to deal with down-sampling. The operator

$$E \texttt{ when } C$$

states that the expression E is evaluated only if the condition C becomes true.

Hence the flow equations

```
y := (a0 * x + a1 * pre(x) + b1 * pre(y)) when c;
c := false -> !pre(c);
```

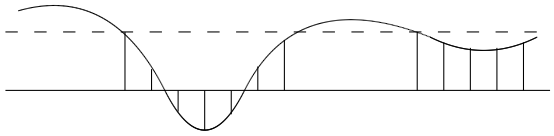
express precisely that the filter equation is evaluated only every other instant. Restricting sampling by a threshold is neatly expressed by

```
y := (a0 * x + a1 * pre(x) + b1 * pre(y)) when (x < up);
```

and both may be combined to

```
y := (a0 * x + a1 * pre(x) + b1 * pre(y)) when (c & x < up);
c := false -> !pre(c);
```

as visualised by



We used a bit of hand waving here. Precisely, the right hand side of the equation is down-sampled only using the operator `when`*, but not the left hand side. If we reconsider the down-sampled filter equation*

$$y(2*n) = a_0 * x(2*n) + a_1 * x(2*(n-1)) - b_1 * y(2*(n-1))$$

the down-sampling (multiplying by 2*) applies to both sides. We express down-sampling of signals as an annotation to the signal type, e.g.*

```
Signal{c & x < up}<double> y;
```

We refer to the expression `c & x < up` *as a* clock*. The idea is that a signal can only be updated by a flow constraint at an instant if the clock condition evaluates to true at that instant. Clearly, the signal and the expression on the right hand side of a flow equation should "be on the same clock" (this will be made precise further below).*

Sampling operators. *The operator* `when` down-samples *the trace* d *at a* frequency *specified by the Boolean "clock" flow* c *as in, e.g.,*

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	...
d		d_0	d_1			d_2		d_3	d_4		...
c		t	f			f		t	f		...
d `when` c		d_0						d_3			...

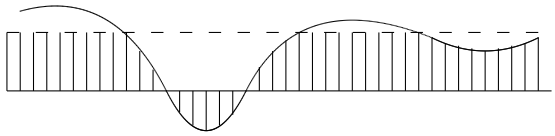
The flow d `when` b *is accessible at an instant if and only if the flow* b *has the value true at that instant.*

We expect up-sampling *as a counter part to down-sampling. The operator* `current` *latches the value of a trace till the next sampling.*

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	...
d		d_0	d_1			d_2		d_3	d_4		...
c		t	f			f		t	f		...
$e = d$ *when* c		d_0						d_3			...
current(e)		d_0	d_0			d_0		d_3	d_3		...

Up-sampling should generate a trace that is "as fast" as the trace originally down-sampled meaning that `current(`d'`)` *is accessible if and only if* d *(resp.* c*) is accessible.*

Upsampling of the curve above gives



The values stay constant when the clock condition does not hold.

An example. *The flow equation for the signal* `display_time`

```
display_time := 0msec -> if (frozen) {
                             pre(display_time);
                        } else {
                             elapsed_time;
                        };
```

can be rephrased quite elegantly using clocks

```
display_time := current(elapsed_time when !frozen);
```

The signal `display_time` *is constrained to the value of* `elapsed_time` *only if the stopwatch is not frozen, but otherwise keeps the value. The pattern is quite typical for the application of clocks.*

5.7 *Clocks for Typing*

Sensors and signal types enhanced by clocks. *Sensors, signals, and expressions may have a clock. A clock is a Boolean flow expression. Sensors, signals, and flow expressions are accessible at an instant only if the clock (expression) evaluates to true at that instant. Clocks are used for typing within a flow context.*

Sensors or signals with a clock are of a type of the form

$$\texttt{Sensor}\{C\}\texttt{<}T\texttt{>} \quad \textit{resp.} \quad \texttt{Signal}\{C\}\texttt{<}T\texttt{>}$$

where c is a Boolean flow expression, and where T is a (value) type.

Clocks are syntactic entities*: $\texttt{Sensor}\{C\}\texttt{<}T\texttt{>}$ and $\texttt{Sensor}\{C'\}\texttt{<}T'\texttt{>}$, for instance, are equal if and only if T and T', and C and C' are syntactically equal. Hence the types $\texttt{Sensor}\{C\}\texttt{<true>}$ and $\texttt{Sensor}\{C\}\texttt{<!false>}$ are different even if the clocks* `true` *and* `!false` *have the same semantics.*[4]

The familiar signal types

$$\texttt{Sensor<}T\texttt{>} \quad \textit{and} \quad \texttt{Signal}\{C\}\texttt{<}T\texttt{>}$$

are equivalent to

$$\texttt{Sensor}\{\texttt{true}\}\texttt{<}T\texttt{>} \quad \textit{resp.} \quad \texttt{Signal}\{\texttt{true}\}\texttt{<}T\texttt{>}.$$

Sensors ans signals with clock `true` *are said to be "on base clock".*

The following restrictions apply

- *Sensors and signals that have a clock different to* `true` *can only be updated within a flow context being constrained by a flow equation.*
- *Input sensors and output signals must have clock* `true`*.*
- *A local signal that is updated using the* `emit` *statement must have clock* `true`*.*

This implies that only those signals can be updated both within and outside of a flow context that are on base clock. Sensors and signals with another clock can be accessed outside of a flow context, though (by using the operators `?`*,* `$`*, or* `@`*).*

A sensor or signal is accessible *at an instant if and only if both*

[4] We check clocks only syntactically since checking for the equality of Boolean terms is potentially exponential which is not acceptable for a compiler.

- *its scope*[5] *is active and*
- *its clock evaluates to true at that instant.*

A sensor or signal can only be accessed resp. updated if it is accessible.

We have the following sub-typing

- `Sensor`$\{C\}$`<T>` *is a subtype of the flow type* $T\{C\}$.
- `Signal`$\{C\}$`<T>` *is a subtype of* `Sensor`$\{C\}$`<T>`

This embeds sensors and signals in flow expressions.

Typing flow expressions. *By definition, all expression that occur in a flow context are flow expressions. Flow expressions and their typing are inductively defined by:*

- *Expressions of primitive type lift to flow expressions.*[6]
  - c *is of type* `Sensor`$\{C\}$`<T>` *if* c *is a literal of type* T.
  - $f(E_1,\ldots,E_n)$ *is of type* `Sensor`$\{C\}$`<T>` *if*
    - f *is an operator or a data method with arguments of type* $T_1,\ldots,T_n$ *and with a result of type* T, *and if*
    - *the flow expressions* E_i *are of type* `Sensor`$\{C\}$`<`T_i`>`
- `pre(`E`)` *is a flow expression of typ* `Sensor`$\{C\}$`<T>` *if* E *is so.*
- E_1 `->` E_2 *is a flow expression of type* `Sensor`$\{C\}$`<T>` *if both* E_1 *and* E_2 *are so.*
- E `when` C *is a flow expression of type* `Sensor`$\{C\}$`<T>` *if, for some clock* C', E *is of type* `Sensor`$\{C'\}$`<T>`, *and* C *is of type* `Sensor`$\{C'\}$`<boolean>`.
- `current(`E`)` *is a flow expression of type* `Sensor`$\{C\}$`<T>` *if* E *is of type* `Sensor`$\{C\}$`<T>` *with* C *being of type* `Sensor`$\{C'\}$`<boolean>`.

Note that the arguments of all operators must have the same clock. The semantics of flow expressions will be given below in Section 5.8.

[5] As usually, the scope of a signal or sensor as a field is the whole object, and the scope of a local signal are the subsequent statements of the block.

[6] Infix, prefix, and postfix conventions are preserved.

Flow equations. *The signal that is constrained by a flow equation and the flow expression that is used to constrain the signal must have the same clock, i.e.*

- *A flow equation* s `:=` E *is well formed, if the flow expression is of type* `Sensor`$\{C\}$`<`T`>` *and if the signal* s *is of type* `Signal`$\{C\}$`<`T`>`*, for some type* T*, and some clock* C*.*

Clock dependency. *The declaration of a clocked signal may may depend on other declaration as in*

```
Sensor<int>            s1;
Signal{s1 >= 0}<boolean> s2;
Signal{s2}<int>          s3;
```

In such a case we speak of dependent declarations*. The condition* `s1 >= 0` *can only be evaluated if the value of the signal* `s1` *is known. In that the clock of* `s2`*, and hence the signal itself, depends on the signal* `s1`*. Similarly, the signal* `s3` *depends on the signal* `s3`*, and by transitivity on the signal* `s1`*.*

Note that flow declarations may cause a causality error*: the declarations*

```
Signal{s2}<boolean> s1;
Signal{s1}<boolean> s2;
```

are circular. Properly specified clocks should form a tree with the base clock as root.

Node calls revisited. *Flow expressions may be used as parameters for node calls*

5.8 *Flows for Semantics*

Clocks form a tree. *Clocks may form a hierarchy as we have seen in the previous section. Actually, if well defined, clock form a tree. This is due to the fact that the clock dependency of signals should not be cyclic and that all operators must have arguments on the same clock. Hence it is possible to up-sample any two expressions to run on the same clock. Up-sampling may, however, need to be iterated. This presumes a more sophisticated semantics than provided by traces.*

Up-sampling reinspected. *We have introduced up-sampling using a trace diagram such as*

i	0	1	2	3	4	5	6	7	8	9	...
d		d_0	d_1			d_2		d_3	d_4		...
c		t	f			f		t	f		...
$e = d$ `when` c		d_0						d_3			...
`current(`e`)`		d_0	d_0			d_0		d_3	d_3		...

The up-sampled trace `current(`e`)` *must be accessible if and only if* d *(resp.* c*) is accessible. Hence the definition of* `current(`e`)` *depends on the trace* e *and the frequency at which* d *is accessible. If we indicate accessibility of* d *by adding the* sampling frequency ν *in*

i	0	1	2	3	4	5	6	7	8	9	...
e		d_0						d_3			...
ν		.	.			.		.	.		...

this clearly provides sufficient information for constructing the up-sampled trace

i	0	1	2	3	4	5	6	7	8	9	...
`current(`e`)`		d_0	d_0			d_0		d_3	d_3		...

The sampling frequency abstracts the trace d *by indicating when* d *is accessible.*

Iterating up-sampling. *We generalise the idea in that we specify several sampling frequencies as in*

i	0	1	2	3	4	5	6	7	8	9	...
e		d_0						d_3			...
ν_2		.	.			.		.	.		...
ν_1	.	.	.		.	.	.	.	.		...
ν_0	.	.	.	.	.	.	.	.	.	.	...

Note that the ν_0 *is "faster" than* ν_1*, etc.*

Using these informations we can construct the up-samplings

i	0	1	2	3	4	5	6	7	8	9	...
e' `= current(`e`)`		d_0	d_0			d_0		d_3	d_3		...
e'' `= current(`e'`)`	δ	d_0	d_0		d_0	d_0	d_0	d_3	d_3		...
e''' `= current(`e''`)`	δ	d_0	d_0	d_0	d_0	d_0	d_0	d_3	d_3	d_3	...

The trace ν_0 *cannot be up-sampled any more since it is on base clock.*

Flows. *Obviously, traces are not sufficient to support up-sampling. A more sophisticated structure is needed that consists of the data trace plus all the sampling frequencies as illustrated by the diagram*

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	...
e		d_0						d_3			...
ν_2		.	.			.		.	.		...
ν_1	.	.	.		.	.	.	.	.		...
ν_0	.	.	.	.	.	.	.	.	.	.	...

We refer to such a structure as a flow.

Down-sampling revisited *All the operations on traces lift naturally from traces to flows except for the sampling operators* `when` *and* `current`*. These are the only ones to change the sampling frequencies. Since the* `current` *has been discussed it remains to reanalyse the operator* `when`*.*

Let us assume that we a flow d and a Boolean flow c that are on base clock

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	...
d	d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	...
c	t	t	t	f	t	t	t	t	t	f	...

Then the flow $d' = d$ `when` *c is defined by*

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	...
d'	d_0	d_1	d_2		d_4	d_5	d_6	d_7	d_8		...
ν_0	.	.	.	.	.	.	.	.	.	.	...

Again down-sampling d' by a Boolean flow

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	...
c'	f	t	t		f	t	f	t	t		...
ν_0	.	.	.	.	.	.	.	.	.	.	...

yields the flow $d'' = d'$ `when` *c' is defined by*

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	...
d'		d_1	d_2			d_5		d_7	d_8		...
ν_1	.	.	.		.	.	.	.	.		...
ν_0	.	.	.	.	.	.	.	.	.	.	...

and so on.

Mixing sampling and time shifting. *We illustrate the interaction of sampling with other operators by computing the semantics of a term of the form*

$$\begin{aligned} t \;=\; & \texttt{current}((0\ \texttt{->}\ \texttt{pre}(d))\ \texttt{when}\ c) \\ & + \texttt{current}((1\ \texttt{when}\ c')\ \texttt{->}\ \texttt{pre}(d'\ \texttt{when}\ c')) \end{aligned}$$

step by step where the flows d, d', c, and c' are specified by

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	...
d	d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	...
d'	d'_0	d'_1	d'_2	d'_3	d'_4	d'_5	d'_6	d'_7	d'_8	d'_9	...
c	f	t	t	f	t	t	f	t	t	t	...
c'	f	t	t	f	f	t	f	f	t	f	...

All the flows are on base clock. Hence we can include them in one diagram with the understanding that the frequency ν_0 is common.

We first consider the left argument of addition. The term $t_1 = (0\ \texttt{->}\ \texttt{pre}(d)$ *defines the flow*

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	...
t_1	0	d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	...

(Then 0 *at instant* 0 *is the default value for integers). Next we have that* $t_2 = (0\ \texttt{->}\ \texttt{pre}(d))\ \texttt{when}\ c$ *denotes*

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	...
t_2		d_0	d_1		d_3	d_4		d_6	d_7	d_8	...
ν_0	.	.	.	.	.	.	.	.	.	.	...

Finally the left argument $t_3 = \texttt{current}(0\ \texttt{->}\ \texttt{pre}(d))\ \texttt{when}\ c)$ *of the addition is determined by*

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	...
t_2	0	d_o	d_1	d_1	d_3	d_4	d_4	d_6	d_7	d_8	...

Though looking rather similar, the behaviour of the right hand argument is quite different. At first we include $t'_1 = 1\ \texttt{when}\ c'$, $t'_2 = 1\ \texttt{when}\ c'$, *and* $t'_3 = \texttt{pre}(d'\ \texttt{when}\ c')$ *in one diagram*

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	...
t'_1		1	1			1			1		...
t'_2		d'_1	d'_2			d'_5			d'_8		...
t'_3		0	d'_1			d'_2			d'_5		...
ν_0	.	.	.	.	.	.	.	.	.	.	...

Hence $t'_3 = 1$ `when` c' `->` `pre(`d' `when` c'`)` *denotes the flow*

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	...
t'_3		1	d'_2			d'_5			d'_8		...
ν_0	.	.	.	.	.	.	.	.	.	.	...

that is up-sampled to $t'_3 =$ `current(1 when` c' `->` d' `when` c'`)`

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	...
t'_4	0	1	d'_1	d'_1	d'_1	d'_2	d'_2	d'_2	d'_5	d'_5	...

Finally we add to obtain

i	*0*	*1*	*2*	*3*	*4*	*5*	...
t	$0+0$	d_0+1	$d_1+d'_1$	$d_1+d'_1$	$d_3+d'_1$	$d_4+d'_2$	...

We conclude that mixing sampling with other operators such as time shifting may result in quite complex – sometimes unexpected – behaviour. For instance, a structure such as

```
current(
  (d when c) ->
     pre(
       current(
         0 -> pre((pre(d' when c) when c') -> (d'' when c) when c'
       )
     )
)
```

needs some thought even of the experienced. However combinations such as

`current(`d `when` c`)`

are obviously useful.

The conclusion is that clocks should be used with care; clocks are a useful programming device, but they easily may be misused.

5.9 *Nodes Inherit Clocks*

The rationale. *If we consider a node such as*

```
node edge_up (Sensor<boolean> _x,Signal<boolean> _y) {|
   _y := false -> _x && ! pre(_y);
|};
```

the parameters formally are on base clock. This constitutes a severe restriction for reusing nodes: whenever we want to apply the node `edge_up` *to signals that have a different clock – say C – we would have to define a new node where only the type of the parameters are changed*

```
node edge_up_1 (Sensor{C}<boolean> _x, Signal{C}<boolean> _y) {|
   _y := false -> _x && ! pre(_y);
|};
```

This is quite annoying.

We overcome this restriction by assuming that the parameters may inherit the clocks from the arguments. Then we could use the original node `edge_up` *but still have a node call*

```
edge_up(x,y)
```

with `x` *and* `y` *being of type* `Sensor{`C`}<boolean>` *resp.* `Signal{`C`}<boolean>` *for some clock C. The node call will be expanded substituting the arguments for the parameters. One should note that both the signals must have the same clock, or – stated more generally – that the inherited clocks must such that the flow equations can still be properly typed.*

The clocks of a node must be consistent. *In order to achieve the latter the following restrictions must be satisfied*

(i) Sensors and signals used within the body of a node or in clocks of the parameters must be either a parameter of a local variable.

(ii) At least one parameter must have clock `true`*.*

(iii) Node parameters that have the same clock must be substituted by arguments that have the same clock.

The general idea is that a node is type checked for once and all independently of the environment. For a node call, then one has just to check whether the clocks of the arguments are consistent with regard to the clocks of the parameter.

The conditions (i) and (ii) guarantee that the clock `true` *is the "fastest" clock used within a node. All other clocks are defined relative to this base clock. Note in particular that condition (i) excludes that signals that are declared as a field can be used within a node. The reason is that these have an "absolute" clock, which would restrict the choice of clocks of arguments from within a node. Condition (iii) is obviously necessary for a type consistent substitution of parameters by argument.*

We consider some schematic examples to illustrate the points.

- *Let the clock of a parameter depend on another parameter*

  ```
  node aaaa (Sensor<int> x, Signal{?x}<int> y) {|
     y := x when ?x;
  |};
  ```

  Let `xx` *be of type* `Signal{`C`}<int>` *for an arbitrary clock* C *and* `yy` *be of type* `Signal{?xx}<int>`*. Then the node call* `aaaa(xx,yy)` *is correct.*

- *Let* `z` *be a signal of type* `Signal<int>` *being declared as a field, and let*

  ```
  Signal<int> z = new Signal<int>();
  node bbbb (Signal<int> y) {|
     y := 0 -> pre(y) + z;
  |};
  ```

  This is not allowed according to (i) since the node accesses a signal being a field. The argument is that the addition forces `x` *and* `y` *to be of type* `Signal<int>` *which prohibits a node call such* `bbbb(xx)` *with* `xx` *being of type* `Signal{`C`}<int>` *for an arbitrary clock* C*.*

- *On the other hand a node may access data fields. Let* `v` *be a field of type* `int`*. Then*

  ```
  node cccc (Signal<int> y) {|
     y := 0 -> pre(y) + v;
  |};
  ```

  is well defined. Data variables and constants have the (within a node) relative clock `true`*.*

Design of a watchdog using clocks. *We extend the watchdog example of Section 5.2. We remind that a watchdog is a device to manage deadlines. The Boolean sensors* `set` *and* `rest` *switch the watchdog "on" and "off". The simple watchdog of Section 5.2 is turned into a node*

```
node watchdog1 (Sensor<bool> _set,
                Sensor<bool> _reset,
                Sensor<bool> _deadline,
                Signal<bool> _alarm ) {|
    Signal<bool> watchdog_is_on;
    _alarm         := _deadline && watchdog_is_on;
    watchdog_is_on := _set ||
                      (false -> ! _reset && pre(watchdog_is_on));
|};
```

(cf. chpt_5_watchdog-node1.se) The alarm is raised if the watchdog is "on", and if the parameter _deadline *true.*

A second version of the watchdog (compare [16]) must raise the alarm if it has remained set for a given delay, counted in terms of instants.

```
node watchdog2 ( Sensor<bool> _set,
                 Sensor<bool> _reset,
                 Sensor<int>  _delay,
                 Signal<bool> _alarm ) {|
     Signal<bool> deadline_condition;
     Signal<bool> deadline;
     Signal<int>  remaining_delay;
     watchdog1(_set,_reset,deadline,_alarm);
     deadline_condition := remaining_delay == 0;
     edge_up(deadline_condition,deadline);
     remaining_delay := if (_set) {
                          _delay;
                        } else {
                          0 -> pre(remaining_delay) - 1;
                        };
|};
```

The deadline approaches when the delay has elapsed. This is computed by decreasing a counter at every instant. The counter is set to the given delay whenever the watchdog is set. We have reused the nodes watchdog1 *and* edge_up.

The next version of the watchdog behaves like the previous one, but delay must be counted relative to the occurrence of some event time_unit. *To reduce computation we only have to call* watchdog2 *if it perceive a set or reset command or a time unit event. This is achieved by down-sampling the arguments when calling the node* watchdog2, *and by up-sampling its "result" signal* alarm.

```
public Watchdog () {
  active {
    sustain {|
      Signal<bool> clock;
      Signal{clock}<bool> c_set;
      Signal{clock}<bool> c_reset;
      Signal{clock}<int>  c_delay;
      Signal{clock}<bool> c_alarm;
      clock   := true -> set || reset || time_unit;
      c_set   := set   when clock;
      c_reset := reset when clock;
```

```
        c_delay := delay when clock;
        watchdog2(c_set,c_reset,c_delay,c_alarm);
        alarm := current(c_alarm);
      |};
    };
};
```

5.10 Digital Signal Processing

The concepts presented in this section are preliminary and may be modified.

For motivation. Our first example in this chapter, the filter

$$y[n] = 0.1 * x[n] + 0.9 * y[n-1],$$

is a simple example for digital signal processing we so far have used for motivation. Although the data flow sub-language is sufficiently expressive to support digital signal processing in general, one may notice severe drawbacks concerning scalability. A general IIR filter of the form

$$y[n] = \sum_{i=0}^{N} a_i * x[n-i]$$

is easy to state as a data flow equation if N equal 2, but becomes cumbersome if N equals 17, and is quite unfeasible if N equals 1000. Now one may wonder whether the latter occur. They do, for instance when considering adaptive filters, a prominent example being the adaptive LMS filter

$$\begin{aligned} y[n] &= \sum_{i=0}^{N} w_i * x[n-i] \\ w_i[n] &= w_i[n-1] + mu * e[n] * x[n-i] \end{aligned}$$

where μ is a constant factor, and e is an error signal. Application of this algorithm for active noise cancellation sometimes demands for an N even bigger that 1000.

We may consider the IIR filter as a dot multiplication of linear algebra, of the vector $\bar{a}_i = a_i$ of coefficients and the vector $\bar{x}_i = x[n-i]$. Hence, we could succinctly state

$$\bar{a} * \bar{x} = \sum_{i=0}^{N} a_i * x[n-i].$$

In programming languages, vectors come as arrays, e.g.

```
double[] a = {0.1,0.2,0.3};
```

However we need extra notation to turn a signal in an array of buffered values. Since we cannot use an accent, we propose the notation `x...` The length of this vector depends on the number of coefficients in the equations above. It would be cute to deduce the length from the context. Then we tentatively write a flow equation

```
y := a * x..;
```

where we use `*` polymorphically to denote the scalar product of two vectors. This should be equivalent to

```
y := a[0] * x + a[1] * pre(x) + a[2] * pre(pre(x));
```

or in mathematical terms

$$y(n) = 0.1 * x[n] + 0.2 * x[n-1] + 0.3 * x[n-2].$$

Then the adaptive LMS algorithm would be concisely captured by

```
y := w * x..;
w := pre(w) + mu * e * x..;
```

provided we can specify the length of the vector `w`.

Vectors and matrices. Though being restricted as compared to JAVA™, arrays in *synERJY* are too flexible to serve our purposes in that an array variable may refer to arrays of arbitrary size, and the size may even vary at run time. Hence we introduce the new data type *vector*. A vector is an array of a fixed size. The notation is

```
double[3]  a = {0.1,0.2,0.3};
double[13] w = new double[13];
```

The size is part of the type. Hence checking the sizes of vectors is part of type checking. Vectors are a subtype of one-dimensional arrays thus inherit all the operators of these. The size of an array must be specified by an integer constant, i.e. either an integer, or an integer variable the value of which can be determined at compile time.

Corresponding to vectors we introduce *matrices* as a subtype of two-dimensional arrays. The obvious notation is

```
double[2,3]  a = {{0.1,0.2,0.3},{0.1,0.2,0.3}};
double[4,13] b = new double[4,13];
```

Operations on vectors and matrices. According to the LMS filter example, scalar multiplication of vectors would come handy as well as addition. Matrix multiplication, of course is needed. At present, *synERJY* supports addition, subtraction and multiplication.[7]

Since we distinguish between vectors and matrices as types, we gain some notational freedom using overloading. We discuss each the operators separately comparing it with its mathematical equivalent. Values must always be of the same type.

- ***Scalar product.*** The multiplication `a * b` of two vectors `a` and `b` is defined by
$$\bar{a} * \bar{b} = \sum_{i=0}^{N-1} a_i * b_i$$
where the vectors must both be of length N.

- ***Matrix multiplication.*** The multiplication `a * b` of two matrices `a` and `b` is defined by
$$\bar{\bar{a}} * \bar{\bar{b}} = \begin{pmatrix} \sum_{i=0}^{N-1} a_{0,k} * b_{k,0} & & \cdots \\ & \cdots & \\ \cdots & & \sum_{i=0}^{N-1} a_{M-1,i} * b_{i,L-1} \end{pmatrix}$$
where the matrices are of type $T[M, N]$ and $T[N, L]$. The result has type $T[M, L]$.

- ***Multiplication with a scalar.*** The multiplication `a * b` of a vector `a` and a scalar `b` is defined by
$$\bar{a} * b = (a_0 * b, \ldots, a_{N-1} * b)$$
and similarly for matrices
$$\bar{\bar{a}} * b = \begin{pmatrix} a_{0,0} * b & & \cdots \\ & \cdots & \\ \cdots & & a_{M-1,N-1} * b \end{pmatrix}$$
The multiplication `b * a` is defined in analogy.

- ***Point-wise operations.*** The point-wise vector multiplication `a .* b` of a vectors `a` and a vector `b` is defined by
$$\bar{a} \ .* \ \bar{b} = (a_0 * b_0, \ldots, a_{N-1} * b_{N-1})$$

[7] If other operations are required, please `mailto:axel.poigne@ais.fraunhofer.de`.

and similarly for matrices

$$\bar{\bar{a}} * \bar{\bar{b}} = \begin{pmatrix} a_{0,0} * b_{0,0} & & \cdots \\ & \cdots & \\ \cdots & & a_{M-1,N-1} * b_{M-1,N-1} \end{pmatrix}$$

Addition $\bar{a}$ `+` $\bar{b}$ and subtraction $\bar{a}$ `-` $\bar{b}$ are defined likewise.

- ***Transpose operator.*** The transpose operator `a^t` on a matrix `a` is defined by

$$\bar{\bar{a}}\texttt{\^{}t} = \begin{pmatrix} a_{0,0} & & a_{N,0} \\ & \cdots & \\ a_{0,M} & & a_{N,M} \end{pmatrix}$$

  where `a` is of type $T[M, N]$. In case of vectors the transpose operator is defined by

$$\bar{a}\texttt{\^{}t} = \begin{pmatrix} a_{0,0} \\ \cdots \\ a_{N,0} \end{pmatrix}$$

  hence `a^t` is of type $T[N, 1]$ if `a` is of type $T[N]$.

- ***Diagonal operator.*** The diagonal operator `D` generates a quadratic diagonal matrix $\texttt{D}(x)$ of size N such that

$$\texttt{D}(x) = \begin{pmatrix} x & & 0 \\ & \cdots & \\ 0 & & x \end{pmatrix}$$

  The dimension N is determined by the context.

- ***All operator.*** The all operator `A` generates a vector $\texttt{A}(x)$ of size N such that

$$\texttt{A}(x) = \begin{pmatrix} x \\ \cdots \\ x \end{pmatrix}$$

  The dimension N is determined by the context.

Vectors considered as matrices. For notational convenience vectors `a` of type $T(M)$ will be considered as a matrix of type $T(1, M)$ in case of matrix multiplication. To give an example, let `a` be a vector of type $T(M)$ and `b` be a matrix of type $T(M, N)$. Then

$$\bar{a} * \bar{\bar{b}}$$

will be well defined since `a` is considered as a matrix of type $T(1, M)$. On the other hand $\bar{\bar{b}} * \bar{c}$ is not well defined for a vector `c` of type $T(N)$. We have to use the transpose of `c` in the multiplication

$$\bar{\bar{b}} * \bar{c}\texttt{\^{}t}$$

Note that, in consequence, we have that $\bar{a} * \bar{b}$ and $\bar{a} * \bar{b}\hat{}t$ are equal if considered as matrices. This, however, does not cause inconsistencies since the first term denotes a vector, the second a matrix.

Slices. Slices of vectors and matrices may be used. The general definition of a vector slice

$$\bar{v}[i..j] = \{v_i, v_{i+1}, \ldots, v_j\}$$

provided that $\bar{v}[i..j] = \{v_0, v_{i+1}, \ldots, v_n\}$ such that $0 \leq i$, $i \leq j$, and $j \leq n$. Similar $\bar{m}[i_1..j_1, i_2..j_2$ denotes the submatrix as defined by the bounds $i_1 \leq j_1$, $i_2 \leq j_2$

We use the same notation within *synERJY*. The restriction (at present) is that the bounds as well of the size of slized vectors and matrices can be computed at compile time.

For typing, the slice $\bar{v}[i..j]$ is of $T[j-i]$ if $\bar{v}$ is of type $T[n]$, and $\bar{m}[i_1..j_1, i_2..j_2]$ is of type $T[j_1 - i_1, j_2 - i_2]$ if $\bar{m}$ is of type $T[m, n]$.

We consider $\bar{v}[i..i]$ as being equivalent to $\bar{v}[i]$, as well as $\bar{m}[i..i, j..j]$ as being equivalent to $\bar{m}[i, j]$. In the same way, the slice $\bar{m}[i_1..i_1, i_2..j_2]$ corresponds to $\bar{m}[i_1, i_2..j_2]$, etc. .

Extending flow equations. Assignments to slices are admitted in case of array-valued signals, e.g.: if the signal `y` is of type `int[5]` the flow equation

```
y[1..5] := some expression of appropriate type
```

is correct. Note the restriction that the array indeces must be "constant" expressions, i.e. an expression that can be evaluated to an integer value at compile time.

Example. An experiment[8] consists of a beam that that is attached to a piezo actuator at one end. At the other end one finds an accelerator sensor. The structure is put on top of a shaker that vibrates according to a given white noise input (cf. Figure 5.1). The goal is to control the actuator using

[8] developed at Fraunhofer LBF

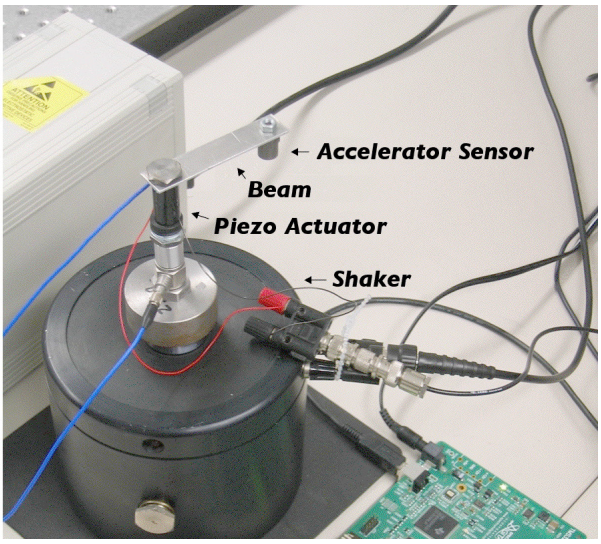
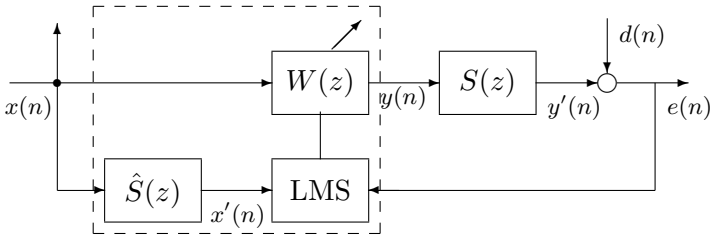


Figure 5.1: **Piezo experiment**

an adaptive FxLMS filter in order to dampen the resultant vibration of the beam.

An adaptive FxLMS filter is determined by the following components:



- The actor-sensor system S (secondary system) is comprised of the piezo actuator, the beam, and the accelerator sensor. Under the assumption that the filter $W(z)$ is linear and time invariant, the filter $W(z)$ and the secondary system may be switched. Replacing the secondary system by its model $\hat{S}$ results in the Figure above. The dashed line then include the FxLMS filter algorithm.

- The adaptive FIR filter $w(z)$ specified by the difference equation

$$y(n) = \sum_{i=0}^{N} w_i \cdot x(n-i)$$

,

- the stochastic LMS algorithm by the equation

$$w_i(n+1) = w_i(n) + \mu \cdot x(n-i) \cdot e(n)$$

where $e(n) = d(n) - y(n)$ is the error with $d(n)$ being the sensor signal.

This algorithm is active in state `controlling` of the subsequent program (with names being more meaningful).

```
class SysId {
  static final time timing = 0sec;

  Sensor<float> acc_sensor  = new Sensor<float>(new CodecRightInput());
  Signal<float> piezo_act  = new Signal<float>(new CodecRightOutput());
  Signal<float> shaker_act  = new Signal<float>(new CodecLeftOutput());

  Sensor          dip0 = new Sensor(new Dip(0));
  Sensor          dip1 = new Sensor(new Dip(1));
  Sensor          dip2 = new Sensor(new Dip(2));
  Signal<boolean> led0 = new Signal<boolean>(new Led(0));
  Signal<boolean> led1 = new Signal<boolean>(new Led(1));
  Signal<boolean> led2 = new Signal<boolean>(new Led(2));

  Signal<float>         y = new Signal<float>();
  Signal<float[256]> w_id = new Signal<float[256]>();
  Signal<float[400]>  w    = new Signal<float[400]>();

  static final float mu_id   = 1.0e-11f;
  static final float mu_ctrl = 1.0e-11f;
  static final float err     = 500.0f;

  public SysId () {
     active {
        automaton {
           init { next state identification; };

           state identification
              entry { emit led2(false); emit led0(true); }
              during {|
                 Signal<float> e;
                 piezo_act := noise_gen();
                 w_id      := pre(w_id + mu_id * e * piezo_act..);
                 y         := w_id * piezo_act..t;
```

```
                e          := acc_sensor-y;
                led0       := e*e < err;
             |}
          when (?dip0) { next state shaker_only; };

          state shaker_only
             entry  { emit led1(true); emit led0(false); }
             during {| shaker_act  := noise_gen(); |}
          when(?dip1) {next state controlling; };

          state controlling
             entry { emit led2(true); emit led1(false); }
             during {|
                Signal<float> z  = new Signal<float>();
                shaker_act := noise_gen();
                z          :=  w_id * shaker_act..t̂;
                w          := pre(w - mu_ctrl * acc_sensor * z..);
                piezo_act  := w * shaker_act..t̂;
             |}
          when  (?dip2) { next state identification; };
       };
    };
  };

  precedence {
     noise_gen() < mu_id;
     noise_gen() < mu_ctrl;
  };

  public static void main (String[] args) {
     while (instant() == 0) {};
  };

  static native private float noise_gen();
}
```

(cf. `chpt_5_fxlms.se`)

There are two other states:

- In state *identify* system identification of the secondary system takes place, and

- in state *shaker_only* only the shaker is excited by the native method `noise-gen` which generates white noise. This state is meant provide a

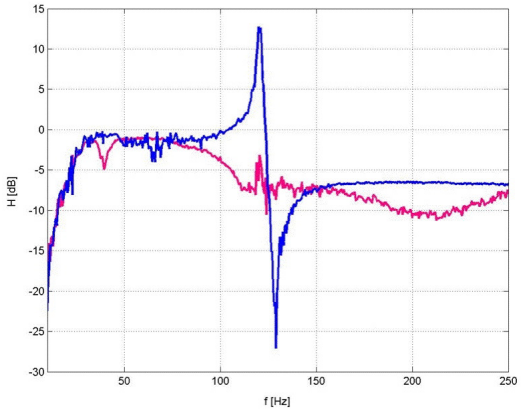


Figure 5.2: **Piezo experiment**

reference with regard controlled state in which the vibration is actively dampened by the piezo actuatuator.

The results can be seen in Figure 5.2. The blue line specifies the undampened, the red line the dampened behaviour.

The program runs on a DSP evaluation board by Texas Instrument to be seen in Figure 5.1. The board provides a Codec and several Dip switches and Led's that are encapsulated in respective input and output classes. The program is built and uploaded from the code as given.

5.11 Representing State Models

Dynamic systems a state model. A *state model* is a system of first-order coupled differential equations of the form

$$\begin{aligned} x_1' &= f_1(t, x_1, \ldots, x_n, u_1, \ldots, u_p) \\ x_2' &= f_2(t, x_1, \ldots, x_n, u_1, \ldots, u_p) \\ &\vdots \qquad\qquad \vdots \\ x_n' &= f_n(t, x_1, \ldots, x_n, u_1, \ldots, u_p) \end{aligned}$$

where x_i' denotes the derivative of x_i with regard to the time variable t, and $u_1, \ldots, u_p$ are specified input variables. The variables $\dot{(x)}_1, \ldots, \dot{(x)}_n$ are calle *state variables.*

Linear time-invariant systems can easily be reformulated to a state model. Consider for instance the equations

$$L\frac{di(t)}{dt} + Ri(t) + v_c(t) \quad = \quad v_i(t)$$

and

$$v_c(t) \quad = \quad \frac{1}{C}\int_{-\infty}^{t} d\tau$$

describing an RLC-circuit. Using the state variables $x_1(t) = i(t)$ and $x_2(t) = v_c(t)$ substitution plus a few algebraic laws, we obtain the corresponding state model:

$$\begin{aligned} x_1' &= \frac{R}{L}x_1 + \frac{1}{L}x_2 + \frac{v_i}{L} \\ x_2' &= \frac{1}{C}x_1 \end{aligned}$$

State models are quite a convenient representation for the numerical solution of a system of differential equations on a computer. The method appropriate for our purposes is the (forward) Euler method. Let

$$\mathbf{x}'(t) \quad = \quad f(t, \mathbf{x}, \mathbf{u})$$

be a state mode (where $\mathbf{x}$ and $\mathbf{u}$ are vectorts). We replace the derivative by the difference approximation

$$\mathbf{x}'(t) \quad \equiv \quad \frac{\mathbf{x}(t+dt) - \mathbf{x}(t)}{dt}$$

which yields the formula

$$\mathbf{x}(t+dt) \quad \equiv \quad \mathbf{x}(t) + f(t, \mathbf{x}, \mathbf{u})dt$$

We can compute the estimates using the scheme

$$\mathbf{x}_{n+1} \quad \equiv \quad \mathbf{x}_n + f(t_n, \mathbf{x}_n, \mathbf{u}_n)dt$$

which naturally tranlates to *synERJY*. This is the *(forward) Euler* method.

Similarly the *backward Euler* method using

$$\mathbf{x}'(t) \quad \equiv \quad \frac{\mathbf{x}(t) - \mathbf{x}(t-dt)}{dt}$$

for approximation can be used.

State models in *synERJY*. We adopt the notation in that we allow "primed" signals in flow equations, e.g.

```
x1' := R/L * x1 + 1/L * x2 + vi/L

x2' := 1/C * x1
```

These equations will be a convenient shorthand for

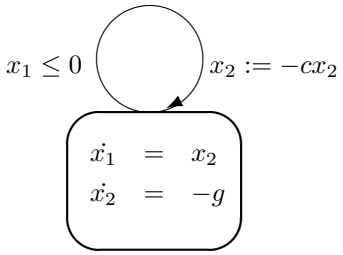
```
x1 := pre(x1) + (R/L * x1 + 1/L * x2 + vi/L) * dt;

x2 := pre(x2) + (1/C * x1) * dt;
```

Note that this corresponds to backwards Euler. This is the more flexible approach since we use the same scheme for

```
x1' := R/L * x1 + 1/L * pre(x2) + pre(vi)/L

x2' := 1/C * x1
```

to obtain forward Euler.

The bouncing ball example reconsidered. We remember that the behaviour of a bouncing ball can graphically specified by the hybrid system



Hence one is tempted to translate this to

```
automaton {
   init { x1 := height;
          x2 := 0.0;
          next state move; };

   state move
      during {| x1' := x2;
                x2' := -c*pre(x2) -> -g;
      |}
   when ($x1 <= epsilon) { next state change; };
};
```

(cf. chpt_5_bouncing-ball-state-model-1.se) However, this is not quite what we want since, according to the above, the state equations translate to

```
x1 := pre(x1) + x2 * dt;

x2 := pre(x2) + (-c * pre(x2) -> -g * dt;
```

But the equations should be (cf. 5.3)

```
x1 := pre(x1) + pre(x2)* dt;

x2 := (-c * pre(x2) -> x2) + -g * dt;  (*)
```

For convenience, we add some notation which achieves just what is wanted

```
automaton {
   init { emit x1(height);
          emit x2(0.0);
          next state move; };

   state move
      during {| x1' := x2;
                x2' := -c*pre(x2) => -g;
      |}
   when ($x1 <= 0.0) { next state  move; };
};
```

(cf. chpt_5_bouncing-ball-state-model-2.se) Here the => indicates that the switch of in the previous value of the signal. The corresponding equation (∗) is generated by preprocessing from `x2' := -c2 * x2 => -g`.

Chapter 6

Reuse

Up to this point, the aspect of control dominates the exposition. We have essentially been interested in the reactive code. The foundations of synchronous programming are settled by now, and we can proceed to a more structural, object-oriented view.

6.1 Interfacing Reactive Objects.

Parameterizing reactive classes. We reiterate that reactive objects communicate by sensors and signals only. The interface to the environment has been discussed in in Section 2.9. Now we focus on interfacing reactive objects.

Sensors and signals are passed to reactive object by calling its constructor. Note that reactive objects have only one constructor. To give an example, we modify the class `Counter` of Subsection 2.6 of Chapter 2: the sensors and signals become parameters of the constructor

```
class Counter {

// constructor
  public Counter (    int _latch,
                   Sensor start,
                   Sensor clock,
                   Signal elapsed )  {
     latch = _latch;
     active {
       loop {
         await ?start; // wait for signal start being present
         reset();      // reset the timer count
```

```
          next;         // reaction finished for this instant
          cancel {      // decrement the counter when ..s
            loop {      // .. signal clock is present
              await (?clock);
              decr();
              next;
            };     // decrementing is cancelled, when ..
          } when (isElapsed()) {};
                  // .. isElapsed() is true ..
          emit elapsed;
                  // .. tell that the timer elapsed
        };
      };
    };

    // data fields and data methods
    int latch;
    int counter;

    void       reset()  { counter = latch; };
    void         decr() { if (counter > 0) { counter--; }; };
    boolean isElapsed() { return (counter == 0); };

  }
```

(cf. `chpt_6_counter-object.se`) Instances of reactive classes are created using the operator `new` as usual. Counters are, for instance, used in the class `PulseWidthModulation` to modulate a signal `wave` to be "up" and "down" for a specified number of instants.

```
  class PulseWidthModulation {
    // constants for counting
    static final int high = 5;
    static final int  low = 15;

    Sensor start = new Sensor(new SimInput());
    Sensor clock = new Sensor(new SimInput());

    Signal<boolean>  wave = new Signal<boolean>(new SimOutput());

    Signal toHighPhase = new Signal(); // local signals
    Signal  toLowPhase = new Signal();

    // two counters as subojects
    Counter highTimer = new Counter(high,toHighPhase,clock,toLowPhase );
```

```
  Counter lowTimer  = new Counter(low ,toLowPhase ,clock,toHighPhase);

  public PulseWidthModulation () {
    // run the pulse width modulation
    active {
        await ?start;
        emit toHighPhase;
        loop {
            await ?toHighPhase;
            emit wave(true);
            next;
            await ?toLowPhase;
            emit wave(false);
            next;
       };
    };
  };
}
```

(cf. `chpt_6_pulse-width-modulation1.se`) The signal `wave` is emitted with value `true` if the value `toHighPhase` is present, and emits the signal `wave` with value `false` if the value `toHighPhase` is present. The counter `highTimer` counts the instants of the high phase, as specified by the actual value of the variable `high`, and the counter `lowTimer` counts the instants of the low phase, as specified by the actual value of the variable `low`.

The signals `clock`, `toHighPhase`, and to `toLowPhase` are arguments of the two constructors of the counters. One should note that the signals are constrained to be read-only sometimes. For instance, the signal `toHighPhase` is restricted to be read-only as argument of the counter initializing `highTimer` (since the parameter `start` of the counter if of type `ConstSignal`). In contrast, the counter initializing `lowTimer` emits the signal `toHighPhase` but can only read the signal `toLowPhase`.

Constructor invocations. The semantics of object composition is that, when generating an instance of class `PulseWidthModulation`,

- signal parameters are substituted by arguments, e.g. the signal parameter `start` of a counter is substituted by the signal argument `toHighTimer` when initialising of the variable `highTimer`.
- the reactive code of the object `PulseWidthModulation` and of all its reactive subobjects – here the two counters `highTimer` and `lowTimer` – are put in parallel.

The resulting reactive code is (more or less) equivalent to

```
[[ await ?start;          // reactive code of the constructor ...
   emit toHighPhase;      // ... of the configuration class ...
   loop {                 // ... PulseWidthModulation
      await ?toHighPhase;
      emit wave(true);
      next;
      await ?toLowPhase;
      emit wave(false);
      next;
   };
|| loop {                   // reactive code related the ...
      await ?toHighPhase;   // ... Counter object highTimer

      resetHigh();
      next;
      cancel {
        loop {
          await (?clock);
          decrHigh();
          next;
        };
      } when (isElapsedHigh()) {};
      emit toLowPhase;
    };
|| loop {                   // reactive code related the ...
      await ?toLowPhase;    // ... Counter object lowTimer
      resetLow();
      next;
      cancel {
        loop {
          await (?clock);
          decrLow();
    next;
        };
      } when (isElapsedLow()) {};
      emit toHighPhase;
    };
]];
```

(cf. `chpt_6_pulse-width-modulation2.se`) Some renaming has been used: for instance, the method `reset` of the class `Counter` has two "localised" versions `resetHigh` and `resetLow` to mimic the objects `highTimer` and `lowTimer`.

Objects and Multiple Emittance Since reactive objects evaluate in parallel, and since they may share signals, conflicts may arise due to multiple emittance. Consider the following (rather useless) object

```
class Simple {

int x;

  public Simple (int _x, Signal<int> _result )  {
    x = _x;
    active {
        emit _result(x);
    };
  };
}
```

(cf. `chpt_6_multiple-emittance1.se`) which is instantiated twice

```
Signal<int> result = new Signal<int>(new SimOutput());

Simple simple1 = new Simple(3,result);
Simple simple2 = new Simple(5,result);
```

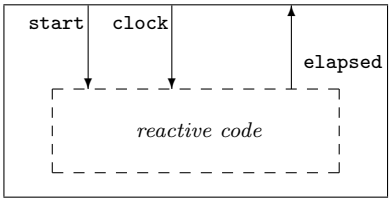
(cf. `chpt_6_multiple-emittance2.se`) Hence the signal `result` is emitted twice at the first instant, with the value 3 and with value 5. An error message `MultEmitInAppl` is raised. Since the error concerns several objects, precedence rules as defined so far will fail to cope. We introduce a new kind of precedence is introduced:

```
precedence {
  for result : simple1 < simple2;
};
```

(cf. `chpt_6_multiple-emittance3.se`) It states that if the signal `result` is emitted in both the objects `simple1` and `simple2`, the *every* emittance of the signal in `simple1` precedes any emmittance of the signal in `simple2`.

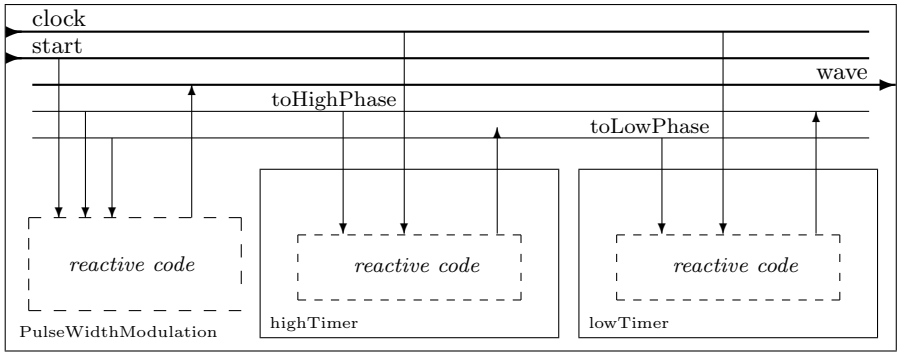
6.2 The Signal Bus

Pictorial presentation. If we focus on the reactive part of objects, a pictorial presentation may be illuminating. Let an instance of the class `Counter` be sketched by



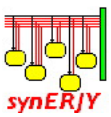
Its reactive code is indicated by the dashed box, and the object itself by the framed box (forgetting about data fields and methods) The parameter signals are presented by arrows going from the framed box to the dashed box and vice versa.

The reactive structure of an instance of class `PulseWidthModulation` may be then presented by



The picture suggests that the reactive codes of the objects involved are executed in parallel and that the different fragments of code communicate via a bundle of signals. We speak of a *signal bus* to refer to this bundle. The signal bus is comprised of all signal (fields) specified in a class. We distinguish local signals such as `toHighPhase` and `toLowPhase`, input signals such as `clock` and `start`, and output signals such as `wave`. We distinguish input and outputs to the environment by a thicker line, and indicate the interface to the environment by connecting them to the vertical sides of the framed box (while parameter signals are connected to the top of the box).

The *synERJY*-logo

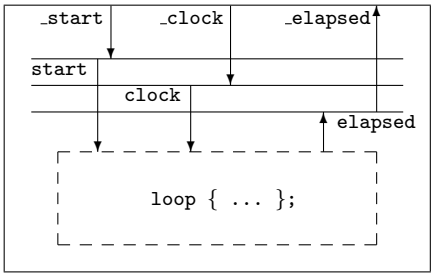


is meant to visualize the idea of a *signal bus*.

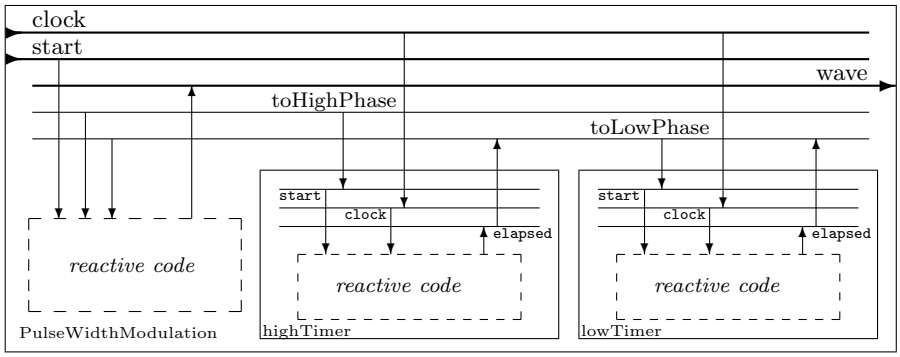
Signal bus hierarchy. Reactive objects and signal busses naturally form hierarchies. Consider, for instance, a variant of the `Counter` class.

```
class Counter {

  int    latch;
  Sensor start;
  Sensor clock;
  Signal elapsed;

// constructor
  public Counter ( int      _latch,
                   Sensor   _start,
                   Sensor   _clock,
                   Signal _elapsed )  {
     latch   = _latch;
     start   = _start;
     clock   = _clock;
     elapsed = _elapsed;
     active {
         // ... reactive behaviour ...
     };
  };
  // ... data fields and data methods ...
}
```

Here the parameters are used to initialize the signals `start`, `clock`, and `elapsed`. The corresponding graphics is



The signal bus is comprised of the signal fields that are used inside the reactive code and that are connected to the parameters. Using this kind of counter the composition of objects yields the following hierarchy where each object has a local signal bus.



The pictures are meant to suggest that

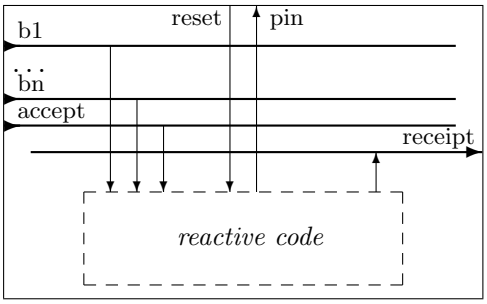
- reactive objects and signal busses form a static hierarchy, and that
- if signals of different busses are "wired" together, it is sufficient to generate only one signal (we refer to as *principal signal*) and to replace every signal by its principal signal.

We will analyse these requirements more deeply in Section 6.3.

On input and output signals. We like to stress that every reactive object may specify input sensors and output signals. This is in contrast to the more usual idea that input and output signals are defined only top-level by the configuration object (i.e. the only object generated by a class with a method `main`) .

There are good reasons: imagine an application with some component being a key pad for submitting a personal identification number. The design of such pads may vary, even in terms of the number of inputs. However, the number of inputs usually is irrelevant with regard to the overall application that may only depend on whether a correct pin has been submitted.

A schematic view of the key pad control in terms of the interface may be



Here `pin` is meant to be a integer valued signal. The box/reactive object analyses the sequence of pressed keys if an `accept` is submitted. If the sequence is submitted the pin is communicated to the application, and the `receipt` signal is emitted with an OK message, otherwise only the `receipt` signal is emitted with an reject message. The number of keys is irrelevant for the overall application. It depends on the actual pad. Typically it will have ten keys, for instance, for an electronic bank till but there might be other builds.

If input and output signals can only be specified at top-level one may have to touch many components of an application to pass the key signals down to the pin analyser and to pass the receipt signal back to top-level. In *synERJY*, these variations have only a local impact in that the component and the connectors have to be redesigned. In that the rationale of *synERJY* is component oriented in that reactive objects behave the same within an application even if the interface to the environment may differ.

6.3 Structural Constraints

Firewalls for separating concerns. A design goal of *synERJY* has been clearly to separate of different concerns as there are: reactive control, data operations, and interaction of objects. We speak of *firewalls* if addressing these separations of concern. The firewalls encapsulate some of the design decision of *synERJY*. There are three such firewalls:

Data Firewall – Control dominates data.

Signal Firewall – Reactive object only share signals.

The data firewall. The data firewall avoids dynamic changes of the reactive control. The data firewall restricts the use relation in that

- A reactive class can only be a subclass of a reactive class.
- data methods cannot invoke reactive methods.

The requirement reflects the standard two-level architecture for control applications: control dominates data.

The signal firewall. The signal firewall constrains time race analysis of data calls to reactive objects, both for reasons of transparency and for reasons of efficiency of the semantic analysis. The constraints are that

- constructor parameters are either primitive types and signal types, and that
- all fields and methods of a reactive class are private.

The combination avoids that two reactive objects can share data except for signals.

Note that time races may still occur between signal emittances. Like causality, time races can only partly be resolved on the level of a class but requires an analysis at the level of an application, at some cost in terms of computing time for the semantic analysis.

Control is static. The firewalls are effective due to an essential fact: the structure of reactive behaviour is statically determined at compile time. Only then our analysis of causality and of time races is feasible. Control is static because of several constraints.

- For each reactive class, the call graph of reactive methods is cycle-free.
- The call graph of reactive objects forms a tree with the configuration object at the root (i.e. the one having the method `main`).
- Subtrees of objects are downwards closed with regard to signals: an reactive object can access signals of a super object, but not vice versa, i.e. *the signal bus must be properly constructed.*
- Reactive objects do not share data except for signals.

Mapping the signal bus. Using constructor parameters for sharing signals is too general to implement the signal bus faithfully. For the signal bus, we have to guarantee that there is exactly one signal that corresponds to one wire.

For a proper implementation of the signal bus, a signal variable must be constrained to refer to a signal of the following kinds:

- a *local* signal, or
- an *input* or *output* signal,
- a *global* signal of a super object.

We speak of local and input/output signals as *principal signals* of the respective object. By induction, a global signal is a principal signal of some super object.

This constraint is syntactically enforced by requiring that

1. a signal can be assigned to only once.
2. the assignment to a signal only occurs in its declaration or in a constructor tail. A *constructor tail* consists of a sequence of (unconditional) signal assignments followed by the active statement.
3. an assignment to a signal variable is either of the forms

```
signal_variable = new signal_constructor(...);
signal_variable = formal_parameter;
```

4. a signal constructor only occurs in an assignment or initializer of the form

```
... signal_variable = new signal_constructor(...);
```

   In that case we refer to *signal_variable* as a *principal signal*.

Condition 1 states that at most one value can be assigned to a signal (variable). Conditions 2 and 3 state that

- a signal (variable) is either initialized using class instance generation (`new ...`) (this is a principal signal) *principal signal*, or that
- a signal being argument of the constructor is assigned to a signal variable.

By induction, every signal argument of a constructor is the principal signal of some super object. Hence every signal variable refers to principal signals, which faithfully reflects the signal wiring of the visual presentation. Finally, condition 4 guarantees that "all signals are principal"; all signal (objects) are refered to by a signal (variable).

6.4 *A Loophole in the Signal Firewall*

An example *In spite of the signal firewall, there might be a time race of data methods being invoked in different objects. Consider a configuration consisting of a "sender" and a "receiver"*

```
class Sender {
    Data data = new Data();

    public Sender (Signal<Data> sent) {
        active {
            data.set(0);
            emit sent(data);
            $sent.set(3);
        };
    };
}

class Receiver {
    int val;

    public Receiver (Sensor<Data> received, int x) {
        val = x;
        active { $received.set(val); };
    };
}
```

The configuration class just establishes a connection between a sender and two receivers.

```
class SignalBus {
   public SignalBus () { active { }; };

   Signal<Data>   sig = new Signal<Data>(new SimOutput());
   Sender      sender = new Sender(sig);
   Receiver receiver1 = new Receiver(sig,1);
   Receiver receiver2 = new Receiver(sig,2);
}
```

The class `Data` *has just a field* `value` *that can be updated.*

```
final class Data {
    public Data () { };

    int value;
    public void set (int val) { value = val; };
}
```

The behaviour is thus: at the first instant, `sender`

1. *updates the fields of* `data` *to* 0 ,
2. *emits the shared signal* `sig` *with value* `data`, *and*
3. *updates the fields* `val` *of* `data` *to* 3.

In parallel,

4. `receiver1` *updates the fields of* `data` *to* 1, *and*
4. `receiver2` *updates the fields of* `data` *to* 2.

In contrast to what we aim for, there are time races between the various invocations of the method `set`.

Dealing with the problem. *Signals with values of class type allow to transmit complex data which clearly is a useful feature. The prize to pay is time races may occur across object borders.*

*There are several strategies of how to avoid such time races. Remember once a signal is updated, its value should not change any more, but only be read (*write-before-read*). This trivially holds for values primitive type. For values of class type, one should only be able to access its fields but not to change them. The same holds for arrays.*

In other languages than JAVA^{TM} *this is achieved using a class modifier* `const` *that defines a superclass comprising only those methods that are read-only. Hence emittance of signal with a value of class type should emit a copy obtained by the respective "const" type. In that case the three applications of the method* `set` *would be illegal. However, it would be legal to apply to replace the line*

```
$sent.set(3):
```

of the class `Sender` *by the line*

```
data.set(3):
```

without changing semantic content.

An alternative strategy would be to guarantee by an in-depth data analysis that a signal value is not changed once the signal has been emitted/constrained.

Neither of these strategies is currently implemented in synERJY. Hence we must leave it as an ***obligation for the programmer*** *to pursue either of the strategies:*

- *Either explicitly to design "const" classes without side effects, and to restrict signal values to these,*
- *or otherwise to guarantee that signal value is not changed after emittance.*

Chapter 7

Building Applications

7.1 Embedding Software

Embedded programs is in general part of a system comprised of mechanical, electrical, and/or pneumatic components, and of one or many micro processors or even workstations possibly linked by a bus system such as CAN. The embedded program must interact with these components and it have have access to the resources of the overall system. In *synERJY*, input sensors and output signals provide the interface to the environment. Access to the resources is gained using native methods.

7.1.1 Interfacing to the Environment.

The interface of a *synERJY* program to the environment is defined in terms of *input sensors* and *output signals*. We recall the basic facts.

- A sensor is an input sensor if its constructor has an argument of type `Input`.
- A signal is an output signal if its constructor has an argument of type `Output`.
- `Input` and `Output` are marker interfaces.
- An implementation of the interface `Input` requires that two (callback) methods `new_val` and `get_val` are implemented.

  For each input sensor, the Boolean method `new_val` is called at the beginning of an instant. If it returns true, the respective signal is set to be present, otherwise it is set to absent. Then, if the sensor

is valued, the method `get_val` is called. It must return a value of appropriate type that is the new value of the signal.

- An implementation of the interface `Output` requires that a method `put_val`.

  For each output signal, the method `put_val` is called at the end of an instant. If the signal is valued it has an argument of appropriate type.

7.1.2 Native Methods

Native methods are implemented in a platform-dependent language. We use ANSI-C as a host language since cross-compilers are available for most platforms, in particular if restricted to the small subset of ANSI-C as used by the *synERJY* compiler.

We have broadened the concept in that not only methods may be native, but also constants, and in that *synERJY* methods may be exported to the host language. We have the following formats

- The standard format is

  ```
  void native method_name( ... );
  ```

  A method `method_name` is defined by a C function with the same name.

- The native method may be renamed

  ```
  void native("method_name") se_method_name( ... );
  ```

  The method `se_method_name` is defined by the C function `method_name`

- Methods defined in *synERJY* may be exported

  ```
  void export_to_C method_name( ... ) {
      ...
  };
  ```

  The function `method_name` may be called in C to execute the method.

- The exported method may be renamed

```
void export_to_C(method_name) se_method_name( ... ) {
...
};
```

The function `method_name` may be called in C to execute the method `se_method_name`

- Constants, i.e. fields that are static and final, may be imported from C or exported to C using the same modifiers as used for methods.

Whether methods or constants are imported or exported is often a matter of taste. However, if such constants or methods are often changed it is probably more convenient to export since development takes place in the same framework. In contrast, it might be useful to import in order to hide details of implementation.

Import and export of constants is useful in particular since JAVA™ lacks enumerations. since enumerations are typically defined by a list of constants the *synERJY* program and C environment should agree about such enumeriations which is best achieved if the enumerations are only defined once and then imported or exported.

7.2 Interfacing to the Environment

The following example demonstrates how interfaces may be implemented using native methods. The example can be found in
`$SE_HOME\target\examples\Blink4`.

Reactive behaviour. The reactive behaviour of the program is specified by the reactive method

```
void toggleLeds () {
  automaton {
    init { next state select_red; };

    state select_red
      do { automaton {
             init { next state red_on; };

             state red_on
             during { emit red_on; }
             when (?toggle) { next state red_off; };
```

```
            state red_off
            when (?toggle) { next state red_on; };
          };
        }
      when (?select) { next state select_green; };

    state select_green
      do { automaton {
             init { next state green_on; };

             state green_on
             during { emit green_on; }
             when (?toggle) { next state green_off; };

             state green_off
             when (?toggle) { next state green_on; };
          };
        }
      when (?select) { next state select_red; };
  };
};
```

There is an hierarchic automaton with two states `select_green` and `select_red`. The automaton changes state if the sensor `select` is present. Each of the states host an automaton. Each of these automata toggles between emittance and non-emittance of an output signal, `red_on` for the state `select_red`, and `red_green` for the state `select_green`. The toggle is applied if the input sensor `toggle` is present.

Interfaces of sensors and signals. We use a very simple idea for interfacing: each of input sensors and output signals is known in the environment by its name as a string. This name is passed on to the environment, either for checking for presence with `new_val`, or when calling `put_val`.

Here is the implementation for inputs

```
class InputButton implements Input {
  public InputButton(String s) { name = s; };
  String name;

  public boolean new_val() { return new_val(name); };

  private native("getPresence") boolean new_val(String _name);
}
```

Calling the constructor stores the "button name". The parameterless method `new_val` just calls a method `new_val` with the button name being parameter. The latter method is native, the implementing function being `getPresence`.

An object of class `InputButton` is then used to initialise the signal `toggle` in

```
Sensor toggle = new Sensor(new InputButton("toggle"));
```

Alternatively, one might have declared an object of type `InputButton` that is used in a sensor constructor as in

```
InputButton selbut = new InputButton("select");
Sensor select = new Sensor(selbut);
```

For output signals, we implement the interface `Output` similarly

```
class LedOutput implements Output {
  public LedOutput(String s) { name = s; };
  String name;

  public void put_val() { put_val(name); return; };

  native("printStdout") void put_val(String s);
}
```

The method `put_val` essentially calls the function `printStdout` of the environment.

A variation of the theme is define the method `stdOutput` on class level and to use an anonymous class

```
static native("printStdout") void    print_stdout(String s);
Signal red_on   =
       new Signal(
         new Output(){
           public void put_val(){print_stdout("red");};
         }
       );
```

The method `main`. The method `main` consists of a while loop.

```
public static void main (String[] args) {
  while (read_input() && (instant() == 0)) {
     print_stdout(". end of instant");
  };
};
```

The condition of the while loop is a conjunction of two conditions. The first condition consists of the method call `read_input`. The method is native. It "reads" the input sensors as we will see when analysing the corresponding C-code below. It is called at the beginning of an instant since it is executed before the second condition is evaluated. The second condition is the familiar call of the method `instant`.

The native code. The native code specifies all the native functions.

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>
#include "se.Blink4.h" /* as generated by the sE-compiler */
```

For checking the presence of input sensors we have the following code

```
Bool selectPresence, togglePresence;

Bool getPresence (char *s) {
  if (strcmp(s,"select") == 0) {
     return selectPresence;
  } else if (strcmp(s,"toggle") == 0) {
     return togglePresence;
  } else {
     fprintf (stdout,"invalid getPresence returned FALSE\n");
     fflush (stdout);
     return FALSE;
  };
}
```

For each input sensor, a Boolean variable is declared. The variables encode the status; if a variable is true, the sensor is present, otherwise it is absent. The function `getPresence` checks whether a sensor is present or not. Its arguments is a string that is matched with the names of the input sensors. If the argument matches a signal name its status is returned. If no signal name matches an error message is written to `stdout`.

```
Bool readInput (void) {
  selectPresence = FALSE;
  togglePresence = FALSE;
  while ( fgets(zLine,20,stdin) ) {
    if (zLine[0] == 'q') {
       fprintf (stdout,"exit (0)\n");
```

```
        fflush (stdout);
        return FALSE;
      } else if (zLine[0] == 't') {
        togglePresence = TRUE;
      } else if (zLine[0] == 's') {
        selectPresence = TRUE;
      } else if (zLine[0] == '.') {
        return TRUE;
      } else {
         fprintf (stdout,"input ignored\n");
         fflush (stdout);
      };
    };
    return FALSE;
  }
```

7.3 Interfacing with the Environment - continued

7.4 Building a Project

The *synERJY* environment supports a very elementary project handling. A *project* consists of a directory that includes all the files relevant to a particular application such as *synERJY* files, *synERJY* charts files, C files, and C libs, and a particular *project resource file* `.seprj` which is generated by using the *project panel* but which may be edited by hand. Here we

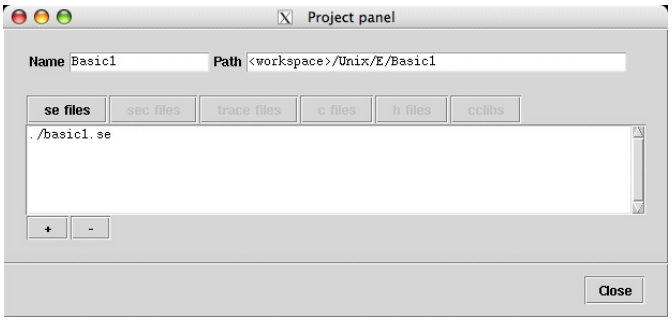


Figure 7.1: The synERGY project panel

consider the project `$SE_HOME\target\examples\Blink4` which consists of the files discussed above.

Building of an application is discussed in the *synERJY* user manual but a few additional remarks may be useful. For maximal flexibility we use makefiles for building and running an application. The `Build` button at first generates a file `Makefile.gen` in `$SE_HOME/tmp`

```
SE_MAKEINFO=makeinfo
SE_WORKSPACE=/Users/ap/Unix/E/Blink4
SE_BINARY=se.a.out
SE_CPROGRAM=/Users/ap/Unix/E/Blink4/se.Blink4.c
SE_CFILES=/Users/ap/Unix/E/Blink4/blink4.c
SE_CLIBS=
```

in case of the example.

This file is included in a generic `Makefile` in the directory `$SE_HOME/target/`*`host`*. The structure of the generic *Makefile* is as follows

```
-include $(SE_HOME)/tmp/Makefile.gen

sim:
        gcc $(SE_CPROGRAM) $(SE_CFILES) -o $(SE_BINARY) \
            -Dse_sim \
            -Dse__linux \
            -I $(SE_HOME)/include \
            -I include \
            -I $(SE_HOME)/target/linux/include \
        $(SE_HOME)/target/linux/lib/libse_rt.a \
        -lm -lpthread

tgt:
        gcc $(SE_CPROGRAM)  $(SE_CFILES) -o $(SE_BINARY) \
        -Dse__linux \
        -I $(SE_HOME)/include  \
        -I $(SE_HOME)/target/linux/include \
        $(SE_HOME)/target/linux/lib/libse_rt.a \
        -lm -lpthread

sfun:
        mex $(SE_CPROGRAM)  $(SE_CFILES) -output $(SE_BINARY) \
            -Dse__linux \
            -I$(SE_HOME)/include \
            -I$(SE_HOME)/target/linux/include \
            $(SE_HOME)/target/linux/lib/libse_rt.a

run:
        exec xterm -e $(SE_BINARY)
```

```
clean:
        rm -f $(SE_PREFIX)*
```

There are several rules according to target.

`sim` A rule to invoke the simulator.

`tgt` A rule to generate code for the host machine.

`sfun` A rule to generate an Sfunction for import to Simulink.

`run` A rule to run the application. Here it is executed in a terminal. For microprocessors, this typically implies upload.

The Makefiles may be replaced according to one's taste but be careful to use the same rules names if the *synERJY* environment is to be used. The better alternative is to include a self-defined Makefile in the project directory. This will then be called automatically instead of the predefined ones.

7.5 Using Makefiles only.

The final alternative for the friends of the hard-core approach is to use Makefiles exclusively forgetting about soft programming environments: there is a reasobly exprissive command language to deal with most problems on the level of command lines or - for comfort - in a Makefile. Here is an example

```
include $SE_HOME/target/unix/include/Makefile.inc

# example blink4.se needs external C
now: se.Blink4

se.Blink4.c: blink4.se
se -f "               set target=linux;set hfile=se_rt.h; \
              make C-code;quit;"

se.Blink4: se.Blink4.o blink4.o
$(CC) -o se.Blink4 se.Blink4.o blink4.o \
             $(SE_HOME)/target/unix/lib/libsert.a

# misc stuff ********************************************************
doc: clean
doxygen
```

```
clean:
rm -f se.* *.o
rm -rf html rtf man latex
rm -rf se.*
rm -rf sim/se.*
rm -rf target/se.*
```

The `Makefile.incl` sets several variables in a uniform way. The general strategy of writing Makefiles is to use one rule to execute two steps

- generate the `se.<ConfigurationClass>.c` file from the synERJY-sources (as it is done in the `se.Blink4.c` - rule)
- compile and link the application (as it is done in the `se.Blink4` rule)

Chapter 8

Validation

A Priori It is a matter of a long lasting discussion what benefits may be expected by validation techniques. We adhere to the pragmatic point of view that any kind of validation, let it be verification or testing, increases our confidence that a program might be reasonably well behaved.

Validation means that we demonstrate that the behaviour of a program complies with some sort of specification, let it be a formal specification in terms of some logic or a test set. Any inconsistency between specification and behaviour reaveals some missing understanding of what either the specification expresses, or how the program behaves. By experience, both may be wrong. Even if the program satisfies the specification absence of errors is not guaranteed. Both may follow the same lines of thought generating the same mistakes. However, independently designed specifications provide a different view of a problem, even if the program and the specification is designed by the same person. In that validation is added value.

synERJY supports formal verication of essentially the control structure of programs using model checking (Section 8.1), and it supports testing on a yet rudimentary basis. (Section 8.6).

8.1 Formal Verification by Model Checking

The principle. The goal of formal verification to prove an assertion ϕ for a given program. As an example we consider a part of a trivial controller for traffic light.

```
loop {
  [[ await ?button;    // demand to cross by a pedestrian
  || emit car_to_green;
```

```
   await 30sec;     // car has a green light for > 30sec
]];
emit car_to_yellow;
await 3sec;        // time for cars to clear the road
emit car_to_red;
await 3sec;        // safety delay
emit ped_to_green;
await 20sec;        // Time for pedestrians to cross
emit ped_to_red;
await 5sec;        // safety delay
emit car_to_yellow;
await 2sec;        // safety delay;
```

Now we would like to check, for example,

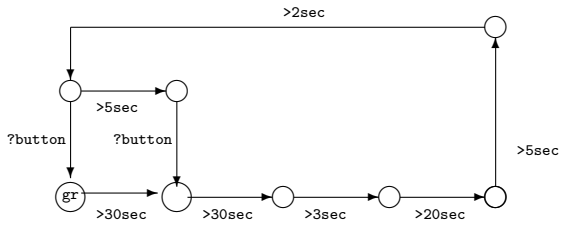
(i) that the traffic light for both cars and pedestrians are never green at the same time (*Safety condition*), or

(ii) whether the pedestrian light eventually turns green after the button has been pressed. (*Liveness condition*).

(iii) the light always eventually turns green. (*Liveness condition*).

There are a number of methods to provide answers to such questions. Pragmatically one has to consider that (i) such methods should be sufficiently efficient and that (ii) they should be applicable for a non-specialist. It seems that from today's viewpoint *Model Checking* [12] best complies with both these requirements.

Model checking checks whether an assertion holds with regard to a particular model. In case of a program as above this means that an assertion must hold in any state that can be reached when executing the program. The reachable states of the piece of code above are given by the following diagram (omitting the emitted signals)



We indicate the status of the light by the combination of "r" and "g" in the circles where car light is on the left and the pedestrian light on the right: e.g. "gr" states that the car light is green and the pedestrian light is red. If

we believe that that diagram above gives a fair account of what the piece of program does, then a simple inspection seems to show that never both the lights are green. Similarly, it seems to be obvious that pedestrians never get "starved". However, in a big system such inspection is hardly manageable and asks for automation.

Actually, if we would apply model checking we would find that assertion (i) is violated. Though the visual inspection seems to prove the assertion, the picture does not tell about initialisation: both traffic lights may be green when starting execution of the code fragment. Prefixing with

```
emit car_to_red;
emit ped_to_red;
```

fixes this problem. However then cars may get starved when the pedestrian button is never pressed. Changing the program structure slightly hopefully helps:

```
emit car_to_red;
emit ped_to_red;
loop {
  await 5sec;        // safety delay
  emit car_to_yellow;
  await 2sec;        // safety delay;
  [[ await ?button;    // demand to cross by a pedestrian
  || emit car_to_green;
     await 30sec;     // car has a green light for > 30sec
  ]];
  emit car_to_yellow;
  await 3sec;        // time for cars to clear the road
  emit car_to_red;
  await 3sec;        // safety delay
  emit ped_to_green;
  await 20sec;        // Time for pedestrians to cross
  emit ped_to_red;
```

Model checking automatically constructs the space of reachable states and checks whether a formula or a set of formulas holds in every state. If this is not the case, a counter example is provided in terms of a trace. Model checking really got off the ground since efficient symbolic techniques have been developed which allow to analyse systems of industrial size.

There are restrictions, however:

- Firstly, assertions are restricted to use Boolean variables only, though recently systems are on the market that deal with enumerations or even integers.

- Secondly, the complexity may grow exponentially with the size of the model and the assertions. Hence model does not guarantee that assertions can be shown to hold within acceptable costs, though model checking can be applied successfully in surprisingly many cases.

Model checking for *synERJY*. Since the control structure of *synERJY* is translated into a kind of sequential circuits ("Synchronous Automata", cf. Appendix B) model checking can easily be accommodated in the development cycle. We provide a binding to NuSMV [31] which can be called from the programming environment.

NuSMV supports *temporal logics* like CTL and LTL. These logics have temporal operators which explicitly refer to state information, e.g.

- **AG** ϕ - In all states the assertion ϕ holds, or
- **AF** ϕ - Eventually a state will be reached in which ϕ holds.

Using this we can rephrase the assertions (i) - (iii) above::

(i) **AG** `!?car_green & ?ped_green`

(ii) **AG** `?button` $\rightarrow$ **AF** `?ped_green`

(iii) **AG AF** `?car_green`

We refer to the NuSMV [31] for more details. The syntax of the temporal logics supported can be found in the reference manual [39].

8.2 Synchronous Observers

The concept of a synchronous observer offers an alternative. A *synchronous observer* [17]) consists of a program `O` that runs in parallel with the program `P` to be observed

```
[[ P || B ]];
```

The idea is now that the program and the observer share signals, but the observer only checks for presence and value of these signals, but never emits them. The observer can only emit a particular signal `error` which is emitted if and only if some error occurs (according to the observer). Absence of errors may be verified by model checking the assertion**AG**`(¬ ?error)`.

Using observers one can check safety properties such as (i)

```
await (?car_green and ?ped_green);
emit error;
```

The advantage is at hand. No second language is needed for assertions, and it may be easier for a programmer to use the familiar programming language rather than an unfamiliar logic.

As a second advantage, the observer may be executed together with the program at run time for redundant control, thus increasing reliability.[1]

Synchronous observers are constrained to cover safety conditions only, i.e. temporal formulas which do not use the operator **AF**. However, synchronous observers represent a good compromise if one is only interested in safety conditions, in particular since one can avoid using a temporal logic at all.

8.3 Handling Data

The restriction to propositional (temporal) logic (sometimes enhanced by some integer arithmetic on a finite subsets of integers) may raise the impression that model checking is not suitable for applications involving data. For instance, there is little hope to prove a property such as "the traffic light for pedestrians eventually turns green after the pedestrian button has been pressed" since this depends on the command `await 5sec` which depends on data.

Assume, howver, that we weaken the original proposition:

(1) The traffic light for pedestrians eventually turns green after the pedestrian button has been pressed if we *assume* that all commands of the form `await` x`sec` eventually terminate after being started.

This proposition abstracts from data and appears as acceptable since the original proposition was only qualitative ("eventually").

The question is how to embed such abstractions into the language. A simple example may provide some insight. Consider the statement

```
await 30sec
```

If we would replace it by the sequence

```
emit await_30sec_start;
await next ?await_30sec_end;
```

then and would extend the model such that the constraint

AG `?await_30sec_start` → **AF** `?await_30sec_end )`

[1] provided that the additional computational effort does not violate timing conditions.

is holds then we should be able to prove

$$\mathbf{AG}\ \texttt{?button} \rightarrow \mathbf{AF}\ \texttt{?ped\_green}.$$

The example provides the guideline of what subsequently will be worked out:

- Introduce additional signals - we refer to as *virtual signals* - to abstract the data flow, and
- extend the behavioural model in a way that some specified constraints are satisfied.

8.3.1 Virtual Signals

Of course, one does not want to rewrite the given program for doing proofs: (i) it is inconvenient, and (ii) one proves programs to be correct which do not run on the target machine. However, the idea is worthwile: try to use pure signals for data abstraction but without changing the behaviour of the program. In that these signals should be *virtual*. Here labels come handy.

Consider the following schematic example

```
[[ emit x(3);
|| await $x < 4; emit y;
|| emit x(5);
]];
```

Now let us attach labels

```
[[ emit x(l1::3);
|| await (l2::$x < 4); emit y;
|| emit x(5);
]];
```

and consider the labels as virtual signals. If we assume as constraint that `l2` is present whenever `l1` is present then we should, for instance, be able to prove **AF** `?y` for the given piece of code provided that

(i) the condition `l2::$x < 4` evaluates to true whenever the virtual signal `l2` is present, and that

(ii) the virtual signal `l1` is "emitted" whenever `emit x(l1::3);` is executed.

This kind of data abstraction naturally is independent of the data abstracted meaning that we could replace the integer and Boolean expressions by arbitrary term e and c without affecting the proof (but, of course, with affecting the semantics.

```
[[ emit x(l1::e);
|| await (l2::c); emit y;
|| emit x(5);
]];
```

Thus, the simple message is: be careful using such abstractions. If we, for instance, use

```
[[ emit x(3);
|| await (l2::$x < 4); emit y;
|| emit x(l1::5);
]];
```

with the proposition above we have the same result but the proven abstraction is semantically inconsistent with the real behaviour.

Data abstraction using virtual signals extends to statements which modify data as in

```
[[ l1::x = 3;
|| await (l2::x < 4); emit y;
]];
```

Here the label `l1` of the assignment can be used as virtual signal.

8.3.2 Constraints

Next we have to deal with constraints such as the above "the (virtual) signal `l2` is present whenever the (virtual) signal `l1` is present".

Let us at first justify the term "constraint". The virtual signal `l2` behaves like a sensor, thus is unconstrained. It behaves like an oracle with regard to the proof system. Hence the statement above should constrain the behaviour in that of `l2` must be present if `l1` is present, but otherwise it still behaves like an oracle: it may be present or absent.

This is sometimes called "Assume-Guarantee Model Checking" (cf. [33], for example). The constraints represent assumptions about the behaviour of the environment under which the guarantees hold. The problem is that the behaviour of the environment must be generated from the constraints and combined with the behaviour of the software to be verified. The generated behaviour should be the "most general" behaviour satisfying the constraints.

It is well known that such a kind of generation only works for the universally quantified ACTL version of CTL. *synERJY* supports such generation of behaviour from constraints for NuSMV.

8.4 Dealing with Time

Though being our motivating example, this kind of data abstraction does not work for a statement such as

```
await 30sec;
```

since we would have to split the command in order to use the data abstraction mechanism above.

Fortunately, modern model checkers usually support variables which range over finite intervals of integers. In case of the statement `await 30sec;` we can use a counter which, at every instant, is increased by the period time until 30 seconds have passed.

Now time resolution is microseconds which defines a finite interval, though: the size of intervals is reciprocal to the speed of the model checker. The bigger the intervals are the longer a proof may take, and the less likely it is that the model checker will terminate properly. Hence it is important to keep the intervals of integers to be used small. Now if a period is specified by the static field `timing`, than we may divide any time constant by the period without affecting the overall behaviour. For example, if the period is `1sec` then the statement `await 30sec;` needs a counter over the interval 0 to 30, which still is quite big with regard to a model checker but manageable.

Statements such as

```
await @button < 5sec;
```

are more difficult to handle. Again a counter is needed to accumulate the time since the signal `button` was present for the last time. In the example given, the context constrains the size of the counter naturally but in

```
await @button1 < @button2 + 5sec;
```

no such constraint can be deduced, one may wait forever for the condition to be satisfied. If this would be a safety condition which guards some emergency behaviour one would pleased never to start the emergency procedure, however one would like to check the behaviour for the case that the safety condition is violated. Hence it may be reasonable for proving to specify as an assumption for how long `button1` or `button2` may be absent at maximum. Our notation is

```
assumption {
    ax0 :: @button1 :> 20sec;
    ax1 :: @button2 :> 30sec;
}
```

meaning that `button1` can be absent for at most `20sec`, `button2` for at most `20sec`. If these times are exceeded model checking will result in an error message. As such this may be not too helpful though it may provide insights: it shows that in our - admittedly very artificial - example the "safety condition" is never really checked within thirty seconds, which one probably would not like to see for a safety condition.

The same game may be played if some fields of type `time` are used. Here restrictions are specified using the field name

```
assumption {
    ax0 :: field1 :> 20sec;
    ax1 :: field2 :> 30sec;
}
```

8.5 Syntactical Issues: by Example

to be written

8.6 Testing as alternative?

Formal verification often implicitly assumes that a program is verified against a *complete* specification. This appears not to be realistic in context of reactive systems since a complete specification - by experience - leaves little space for abstractions.

Most of the times only some crucial properties are checked where often it is more interesting if proofs fail rather than succeed, since design weaknesses or imprecisions are discovered.

If, however, once a property is satisfied, it should (in general) remain true when a program is developed further. Hence verification needs to be redone for every (even little) development step. We may speak of a *regression verification* in analogy to a regression test. Clearly, regression verification should be push-button and highly efficient to be acceptable. Unfortunately, model checking, though push-button, is not always sufficiently efficient so that one may rather rely on regression testing as alternative.

We consider testing and formal verification as complimentary. Even if we have proved a property of a system test cases may provide additional confirmation. Since formal verification proves properties that are abstractions of system behaviour these abstractions may be as error-prone as programs are. Additional test cases substantiate support the belief in such abstractions if the tests pass, or indicate what may be wrong with a program *and* the abstractions used for proving properties. The bottom line is that the more we do to confirm our belief in a program the better. Of course testing is a means in its own right.

By experience, regression tests (or regression verification) is a must when developing embedded software since the various uncertainties one encounters at hardware level should be counterbalanced by a solid belief in the functional correctness of the software. Synchronous programming supports systematic testing at least in three aspects:

- *Behaviours can be reproduced* since determinism of behaviour is guaranteed. This is almost a conditio-sine-qua-non for systematic testing.

- The generated code can easily be *instrumented*. This is due to the underlying semantic paradigm (cf. B) that translates the reactive code into the combination of a sequential circuit and elementary data actions triggered by signals. We indicate the idea using a familiar example

  ```
  emit car_to_red;
  await ?car_red;
  ```

  When instrumented the generated code will not only emit the signal to be present, but the position of the emitting command will be highlighted. Similarly, when waiting the position of the respective statement is highlighted. The highlight commands are additional "data operations" with no repercussions on the behaviour, hence "runtime" code and instrumented code behave equivalently. This holds even for the timing conditions provided that a periodicity is specified.

- This equivalence of behaviour should extend to the code running on the respective target hardware since the code used for internal representation corresponds to a simple subset of ANSI-C (of which we hope it is correctly translated by cross compilers).

Bibliography

[1] C. André. Representation and analysis of reactive behaviours: A synchronous approach, in: *Proc. CESA'96*, Lille, France, July 1996.

[2] G. Berry, The foundations of Esterel, in: G. Plotkin, C. Stirling, and M. Tofte (Eds.), *Proofs, Languages, and Interaction: Essays in Honour of Robin Milner*, MIT Press, 2000

[3] G. Berry, The Constructive Semantics of Pure Esterel, Draft book, `http://www-sop.inria.fr/meije/esterel/doc/main-papers.html`, 1999

[4] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[5] R. Budde, A. Poigné, K.-H- Sylla. synERJY - an Object-oriented Synchronous Language -. Workshop on Synchronous Languages and Applications, Barcelona, 2004,
also: *Electronic Notes in Theoretical Computer Science*, Volume 153

[6] R. Budde, K.-H. Sylla. Objektorientierte Echtzeit-Anwendungen auf Grundlage perfekter Synchronisation In: *OBJEKTspektrum* 0(2):54-60, 1995, Zeitschrift fr objektorientierte Technologien, Mnchen und SIGS Conferences GmbH Mnchen, Bd.Nr.7, 1995.

[7] R. Budde, K.-H. Sylla. ObjektorientiertBudde, R.; Merceron, A.; Sylla, K.H. Formal Verification as a Design Tool - The Transponder Lock Example In: Safe Comp'96, Vienna, Austria 23-25 October 1996, Proceedings of the 15th International Conference on Computer Safety, Reliability and Security, Springer Verlag, ISBN 3-540-76070-9 pp. 73-82.

[8] R. Budde, A. Poigné. Complex Reactive Control with Simple Synchronous Models ACM SIGPLAN Workshop on Languages, Compilers,

and Tools for Embedded Systems (LCTES'00), Vancouver, Canada, June 2000, Lecture Notes in Computer Science 1985, Springer, 2000

[9] R. Budde, P. Pl"o"ger, K.-H. Sylla. A Synchronous Object-Oriented design flow for Embedded Applications In: Proc. *2nd International Forum on Design Languages* (FDL), ECSI Verlag, Gires, 1999

[10] R. Budde, M. Pinna, A. Poigné Coordination of Synchronous Programs In: P. Ciancarini, A.I. Wolf (eds.), *3rd Int. Conf. on Coordination Languages and Models*, LNCS 1594, Springer, Heidelberg, 1999

[11] R. Budde, A. Poigné. On the synthesis of Distributed Synchronous Processes. in: F.Cassez, C.Jard, B.Rozoy, M.Ryan (Eds.) Proceedings of the summer School "Modelling and Verification of Parallel Processes" (MOVEP'2k), Nantes, June 2000

[12] E.M. Clarke, E.A. Emerson, A.P. Sistla, Automatic verification of finite-state concurrent systems using temporal logical specifications, in: *ACM Transaction on Programming Languages and Systems*, 8(2):244-263, 1986

[13] P. Caspi, C. Mazuet, R. Salem, D. Weber, Formal Design of Distributed control Systems with Lustre, in: Proc. Safecomp'99, 1999

[14] J. Gosling, B. Joy, G. Steele.*The* JAVA^{TM} *Language Specification*, Addison-Wesley, Nov. 1999

[15] P. Le Guernic, A. Benveniste, P. Bournaii, T. Gautier, Signal: a data-flow oriented language for signal processing, *IEEE-ASSP*, 34(2), 1986.

[16] N. Halbwachs. *Synchronous Programming of Reactive Systems*, Kluwer Academic Publishers, Dordrecht, 1993.

[17] N. Halbwachs, E. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In: Third Int. Conference on Algebraic Methodology and Software Technology, AMAST93, Workshops in Computing. Springer Verlag, June 1993.

[18] D. Harel. Statecharts: A visual approach to complex systems, *Science of Computer Programming*, 8:231–274, 1987.

[19] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[20] L. Holenderski. LUSTRE - a verified production cell controller In: *Formal development of reactive systems* / Lewerentz, Claus (Ed.), 101-112, 1995. NuSMV

[21] L. Holenderski, A. Poigné. Synchronous Automata for Synchronous Programming Languages In: Wolisz, A. (Hrsg.); Schieferdecker, I. (Hrsg.); Rennoch, A. (Hrsg.), *Formale Beschreibungstechniken fr verteilte Systeme GI/ITG-Fachgesprch*, 19.-20. Juni 1997, Berlin, (GMD-Studien Nr. 315). Sankt Augustin: GMD, 1997, S. 129-134. ISBN 3-88457-315-2 Full Paper in Postscript

[22] J.L. Jones, A.M. Flynn, *Mobile Robots - Inspiration to Implementation*, Wellesley,1993

[23] S. Kowalewski. Modellierungsmethoden aus der Informatik, at-Automatisierungstechnik, Heft 5(50): A1-A5, September 2001.

[24] O. Maffeis, A. Poigné. Synchronous Automata for Reactive, Real-Time or Embedded Systems (Arbeitspapiere der GMD Nr. 967). Sankt Augustin: GMD, Schlo Birlinghoven, D-53754 Sankt Augustin, ISSN 0723-0508, 51 S., 1996.

[25] F. Maranchini, Y .Rémond. Mode-Automata: About Modes and States in Reactive Systems, Proc. European Symposium on Programming, Lisbon, Portugal, 1998

[26] F. Maranchini. Operational and compositional semantics of synchronous automaton compositions, In *Proceedings of CONCUR'92*, volume 630 of *Lecture Notes in Computer Science*, Springer-Verlag, 550–564, Aug. 1992.

[27] A. Merceron, M. Müllerburg, M. Pinna. Verifying a time-triggered protocol in a multi-language environment In: Ehrenberger, W. (Hrsg.): *Computer Safety, Reliability and Security*. Proc. 17th Intern. Conf. SAFECOMP'98. Lecture Notes in Computer Science (LNCS) No. 1516. 1998. pp 185-195.

[28] M. Müllerburg. Systematic Testing: a Means for Validating Reactive Systems In: *Software Testing*, Verification and Reliability 5(3):163-179 (1995). Selected papers from EuroSTAR'94

[29] M. Müllerburg, L. Holenderski, O. Maffeis, A. Merceron, M. Morley. Systematic Testing and Formal Verification to Validate Reactive Programs In: *Software Quality Journal* 4(4):287-307 (1995)

[30] M. Müllerburg, R. Budde. Validation und Verifikation in Synchronie: synchronousEifel In: GMA-Kongress '98 Mess- und Automatisierungstechnik. VDI-Berichte 1397, S. 299-308. Dsseldorf : VDI Verlag GmbH, 1998.

[31] The NUSMV website, `http://nusmv.irst.itc.it/`

[32] L. Padulo, and M. Arbib. *System Theory - A Unified State-Space Approach to Continuous and Discrete Time Systems*, W.B. Saunders Company, Philadelphia - London - Toronto, 1974.

[33] C.S, Psreanu, M.B. Dwyer, M. Huth. Assume-Gurantee Model-Checking of Software: A Comparative Case study. *Theoretical and Practical Aspects of SPIN Model Checking*,Springer, 1999.

[34] A. Poigné, L. Holenderski. *Boolean Automata for Implementing Pure Esterel* Sankt Augustin: GMD, 1995, 47 S. (Arbeitspapiere der GMD Nr. 964).

[35] A. Poigné, M. Morley, O. Maffeis, L. Holenderski, R. Budde. The Synchronous Approach to Designing Reactive Systems In: *Formal Methods in System Design*, 12, pp. 163-187 (1998), Kluwer Academic Publishers, The Netherlands.

[36] A. Poigné, L. Holenderski. The Multi-Paradigm Synchronous Programming Language LEA In Proceedings of the Int. Workshop on Formal Techniques for Hardware and Hardware-like Systems, 1998.

[37] A. Poigné, L. Holenderski. On the Combination of Synchronous Languages In: W.P.de Roever (Hrsg.): Workshop on Compositionality: The Significant Difference, Malente September 8-12, 1997, LNCS 1536 Springer Verlag, pp. 490-514, 1998

[38] A. Poigné, L. Holenderski. Synchronie Workbench TOOLS 98, Y. Lakhnech, R. Berghammer (Hrsg.) Advances in Computer Science, 1999.

[39] *synERJY* Reference Manual

[40] *synERJY* Programming Environment User Guide

Appendix A

A.1 The genesis of synERJY: a personal recollection

The seed of what was to become *synERJY* was set when Gerard Berry and myself had a drink at a bar in San Gimignano somewhat about 1990. At his usual self, Gerard enthused about a new programming paradigm "synchronous programming", and his new programming language - ESTEREL.

This seed germinated when the then Embedded System Group at the then GMD was looking for new themes in 1993 and Michele Pinna and myself started to experiment with ESTEREL and more or less immediately got convinced that synchronous programming may should have a future.

For whatever reason Michele and Leslek Holenderski started to think of a new semantics for ESTEREL, probably they disliked the hardware semantics of ESTEREL [2]. This was a trigger for myself, and some weeks later on a spare afternoon I came up with the idea to use a hardware description like Berry but based on the assumption that every statement should yield a sequential circuit which is started if a system signal α is present and that issues a system signal ω when it terminates. Using this, I sat down and computed roughly 50 examples by hand, and got convinced that the approach was wothwile. Some other "system variables" emerged naturally, and I think it was Leslek who introduced a system variable τ for preemption.

Then Leslek wrote a compiler for pure ESTEREL within weeks in Ocaml (which still is the implementation language of *synERJY*). The compiler turned to be sound with regard to the original semantics. On the way, I found a very simple splitting technique to deal with the reincarnation problem. Thia approach was first presented at the C^2A autumn meeting in Paris in 1993. A formal account has then been given in [34] .

This was the starting point of many - sometimes competing - activities,

in which Reinhard Budde, Leslek Holenderski, Agathe Merceron, Olivier Maffeis, Matthew Morley, Monika Müllergburg, Michele Pinna, Axel Poigné, and Karl-Heinz Sylla took part. Everybody intensively contributed to the heated discussions, and it is difficult to give appropriate credits to everybody in retrospect.

In these discussions, we tried to improve and optimise the translation scheme. For instance, Matthew Morley and myself developed, and Matthew implemented another translation scheme in SML. In retrospect however, whatever came up did not really improve the translation scheme, and it is still - with minor changes - the basis of *synERJY*.

Almost immediately, the idea arose to extend the translation scheme to the other synchronous formalisms as ARGOS and LUSTRE. This integration was (almost) achieved with the *Synchrony Workbench* [35], that included a sophisticated front-end for ARGOS, also written by Leslek. Unfortunately, the approach had a semantic shortcoming, which was not so easy to fix, in particular since Leslek left the then GMD. Nevertheless, the *synERJY* inherited much of the simulator of the Synchrony Workbench then written by myself.

Based on the same translation scheme, Reinhard and Karl-Heinz developed a combination of synchronous programming with objected-oriented design starting in 1994. The object-oriented part was strongly influenced by Eiffel, hence the language was called *synchronousEifel* (Eifel being a reference to the Eifel mountains close to Sankt Augustin), or *sE* for short (still syntactically contained *synERJY*) [6].

All this lead to a vision of a smooth integration of all synchronous formalisms which has finally being stated in [35]. The vision then was even bolder, by far exceeding the given means; formal verification was tackled as well testing and the problem dtribution. Agathe, Reinhard, and Matthew worked on formal verification, Monika on testing, and Olivier on distribution, and but everybody contributed to all subjects. This resulted in several publications ([7], [8], [9], [20], [21], [24], [27], [28], [29], [30], [36], [37], [38]) but did not evolve into a programming and verification environment as originally envisaged.

To make our efforts visible, we joined the EUREKA project SYNCRON, and subsequently became partners in the ESPRIT projects SYRF and CRISYS. Further, in order to propagate the gospel, Willem-Paul de Roever and myself discussed the possibility of having a workshop for synchronous programming when he visited GMD in Spring 94. This started the SYNCHRON workshop series, which first took place in Dagstuhl in 1994. Next it was important to get industry interested, in particular in Germany. Hence

Albert Benveniste and myself started the series of the FEMSYS workshops, which was successfully staged in Munich in 1997, 1999, and 2001.

The ESPRIT project SYRF ("Synchronous Reactive Formalisms") boosted the development of *synERJY* in that Reinhard, Karl-Heinz, and myself joined forces to integrate and redevelop the different branches of development. Reinhard was in charge of the object-oriented data part, and he wrote the lexer, parser and typechecker as well as the code generator to C. Further he developed a new graphical editor for *synERJY* charts. Karl-Heinz programmed the control for the simulator, and took care of the backend compilers to diverse micro controllers, while myself focussed on the translation scheme of reactive part of the compiler, and on the gui aspects of the simulator and the programming environment.

In CRISYS ("CRitical Instrumentation and control SYStem") we extended *synERJY* to support distribution using a blackboard like architecture ([10], [11]), but this extension has remained experimental up to today.

Development of the language itself had many interations always aiming for simplifying the concepts and for improving usability. A major change was the shift to Java, or rather Java-like language) for the object-oriented data language, which was accomplished by Reinhard by implementing more or less a compiler from Java to C. Motivation was not to deter users by a combination of two relatively unknown languages, but to built on top of a fairly widely known language such as Jave. Since then *synERJY* is a syntacically stable language though minor changes at the fringes may still happen. A general overview over the present language is to be found in [5].

Within the Fraunhofer internal project FASPAS on adaptronics bwe have extended the language by components dealing with vectors and arrays so that it becomes simple to program typical signal processing applications such as, e.g., filters. Further, *synERJY* has been linked to MATLAB and scicos to support model-based design, and new backends for TI DSPs and Xilinx FPGAs have been added. Shakil Ahmed linked *synERJY* to TI's DSP family and Motorola's MPC555. Further we developed a proprietary DSP board based on TI's TMS329c6713 with four analogue inputs and four analogue outputs with AD/DA converters and anti-aliasing filters, and with *synERJY* as input language.

Axel Poigné

Appendix B

Semantics

The general layout. Our intention is to provide a (slightly simplified) version of the translation scheme for *synERJY*. The following schema explains the general setting

synERJY programs are translated to an internal representation of the reactive part with a control structure in terms of a sequential circuit and data part for the Java-like host language. Semantical checks for causality and time races are performed on this internal representation. Code for the different targets is generated from the internal representation. The target code is typically C to which back-end compilers are applied. For specific targets such as the simulator, Simulink, or Scicos the C code is instrumented, but in a way that the code kernel is exactly the one that runs on micro controllers, DSPs, or similar like. The situation is special for verification and FPGA generation. These translation demand for particular code formats such as Verilog. However, in both cases the translations of the internal code to these targets is of so elementary nature that we claim equivalence of the resulting target code. This may be proved by code inspection comparing the C code (which is annotated by readable version of the internal representation) with the other respective targets.

Thus we will only be concerned with the translation of the *synERJY* code to the internal representation only.

A hardware translation. We define synchronous behaviour in terms of "Synchronous Automata" which have commands of the form

```
s <= φ;          (Emittance of the signal s)
r <- φ;          (Setting a register r
if (s) { f };    (triggering a data operation f)
```

where ϕ is a Boolean expression respectively. The difference between signals and registers is thus:

- the status of a signal can be read only in the instant it is emitted
- the status of a register can be read only in the next instant after it has been set.

The difference is indicated by using the different symbols `<=` and `<-`. The data operation `f` is executed if the signal `s` is present.

A *synchronous automaton* consists of a sequence of such commands. The execution model is simple: the whole sequence of commands is executed exactly once at every instant. We refrain from spelling out a formal definition of behaviour or semantics at this point, but believe that everybody will have an intuitive understanding of how synchronous automata execute.

The translation scheme is compositional. Each statement generates a synchronous automaton, and the language operators of *synERJY* define operators on these automata.[1] If there is any substantial idea in the translation scheme it the use of particular *system signals* that are used in this translation scheme to glue together automata.

To give an example, we translate the elementary statement

```
emit s
```

The corresponding synchronous automaton is

```
s <= α;
ω <= α;
κ <= false;
```

Here α is the *start* signal, and ω the termination. Execution of the automaton starts if α is present, and terminates at the same instant. Then |*omega* will be present at the same instant. The signal κ stands for "is in control", meaning that the automaton does not terminate at the given instant. of course, the emit statement never gets control.

Somewhat more elaborated is the translation of `await ?s`:

```
r <- !s & (α | r);
ω <=  s & (α | r);
κ <= r;
```

[1] Actually, one should be careful with the term compositionality here. The translation scheme is compositional but not the behaviour. The causality analysis will reschedule the sequence of equations genersted.

Here a new *control register* `r` is generated. The automaton keeps control until the signal `s` is present. It then terminates at the same instant, and looses control.

Sequential composition now almost comes for free. Assume that A_1 and A_2 are synchronous automata. Then "A_1 A_2" denotes the synchronous automaton

$A_1[\gamma/\omega, \kappa_1/\kappa]$
$A_2[\gamma/\alpha, \kappa_2/\kappa]$
κ `<=` κ_1 `|` κ_2`;`

where γ, κ_1, κ_2 are new pure signals (γ/ω denotes the substitution of ω by γ). If started A_1 is executed. If A_1 terminates, A_2 is started. The sequential composition is in control if one of the sub-automata is in control.

We apply these rules to

```
emit car_to_red;
await ?car_red;
```

and obtain the synchronous automaton

```
car_to_red <= α;
       γ  <= α;
       κ1 <= false;
       r  <- !car_red & (γ or r);
       ω  <= car_red  & (γ or r);
       κ2 <= r;
       κ  <= κ1 | κ2;
```

The *parallel composition* "A_1 `||` A_2`;`" is slightly more complicated. If started, both the sub-automata start to execute. The parallel composition terminates if either both sub-automata terminate at the same instant, of if one of the sub-automata terminates while the other has terminated computation at an earlier instant. Here the control signal κ comes handy:

A_1`[`ω_1/ω`,`κ_1/κ`]`
A_2`[`ω_2/ω`,`κ_2/κ`]`
ω `<=` ω_1 `&` ω_2
`|` ω_1 `&` κ_1 `&` `not` κ_2
`|` ω_2 `&` κ_2 `&` `!`κ_1`;`
κ `<=` κ_1 `or` κ_2`;`

The condition "ω_1 `&` κ_1 `&` `!`κ_2" states: if the automaton A_1 is in control and terminates, and if the automaton A_2 is not in control (has terminated earlier), then the parallel composition terminates.

The *loop statement* "`loop` `{` A `};`" is again simple

```
γ <= α & ω1;
A[γ/α,ω1/ω]
ω <= false;
```

as is the *conditional* "`if (`ϕ`) {` A_1 `} else {` A_2 `}`"

```
γ1 <= φ & α;
γ2 <= not γ1 & α;
A1[γ1/α,ω1/ω,κ1/κ]
A2[γ2/α,ω2/ω,κ2/κ]
ω <= ω1 | ω2;
κ <= κ1 | κ2;
```

with `r` being a new control register. Note that A should always be in control for the loop statement because otherwise A would be executed infinitely often at one instant. The conditional can easily be iterated.

The power of the imperative part of the language stems from the *pre-emption statement*. In preparation, we take a closer look at the `halt` statement. It keeps control forever except if it pre-empted. We model pre-emption by introducing a new system signal τ which is supposed to trigger pre-emption. For each `halt` statement a control register r is generated

```
r <- α | !τ & r;
ω <= τ & r;
κ <= r;
```

Pre-emption is then modelled using the system signal τ. We consider a simple case of the cancel statement

```
cancel {
   A
} when (φ1) { A1 }
   ...
   else when (φn) { An }
```

that translates to

```
τ1 <= τ | κ & (φ1 | ... | φn);
A[τ1/τ]
γ1 <= τ1 & φ1;
A1[γ1/α,κ1/κ]
...
A1[γ1/α,κ1/κ]
```

The automaton A is pre-empted if it is in control, and if one of the conditions ϕ_i becomes true. Then the respective branch is executed. The other cases like strong pre-emption are considerably more complicated and omitted here. Note that the cancel statement may be pre-empted from the context, hence the τ | ....

Just to convince ourselves that the translation scheme operates properly, let us consider the statement

```
await (φ); = cancel {
                 loop {
                    next;
                 };
             } when ?s;
```

The next statement is defined by `next;` statement

```
r <- α;
ω <= τ & r;
κ <= !τ & r;
```

Not surprisingly, the loop generates

```
r <- α | γ;
γ <= τ & r;
κ <= !τ & r;
ω <= false;
```

which exactly is the semantics of the halt statement. The cancel statement then generates (modulo renaming and rearrangement, and using that A_1 is the nothing statement)

```
τ1 <= τ | κ & ?s;
r <- α | γ;
γ <= τ & r;
κ <= !τ & r;
ω <= false;
 ω <= τ & r;
ω <= τ & ?s;
```

This is equivalent to

```
r <- !s & (α | r);
ω <=  s & (α | r);
κ <= r;
```

and We omit translation of the other available constructs to the reader who also may want to check that the *await* statement does not need its own translation rule but can be synthesised

Dealing with data. Using `await x > 5` we demonstrate the interaction of data and control The obvious idea is that if started a time counter is set to 0 that is increased by the ddelta of time `dt` at every instant till the counter exceeds the time limit of 30 seconds. lässt sich die Interaktion von

```
r <- α | !δ & r;
if (r) { δ = x > 5 };
ω <= δ & r;
κ <= r;
```

The counter is initialized when `alpha` is present. The other data actions are executed whenever the register `r` is set. The register is set for the next instant when `alpha` is present. The Boolean signal δ interfaces the data with the control structure.

The general idea is thus: data actions like routine calls, assignments, etc. are embedded by the scheme

```
if (α) { data action };
ω <= α;
κ <= ff;
```

and atomic Boolean expressions by

```
if (α) { δ = atomic proposition };
ω <= α;
κ <= ff;
```

One notices a clear separation of control and data operations that are activated by signals and that provide feedback using pure signals.

to be continued