synERJY
- an Object-oriented Synchronous Language -

Reinhard Budde[1] Axel Poigné[2] Karl-Heinz Sylla[3]

Fraunhofer Institute "Autonomous Intelligent Systems"
Schloß Birlinghoven
D-53754 Sankt-Augustin

Abstract

The programming language *synERJY* is presented. It integrates object-orientation and synchronous formalisms in the spirit of Esterel, Lustre, and Statecharts.

Key words: synchronous programming, object-oriented design, hierarchical state machines, data flow

1 Introduction

synERJY is a programming language and a design environment for embedded systems. It combines two paradigms:

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

Highlights are that

- *synERJY* provides a deep embedding of reactive behaviour into an object-oriented data model.
- *synERJY* offers fine-grained integration of synchronous formalisms such as Esterel [4], Lustre [7], Signal [2], and Statecharts [6].[4]

The programming environment supports compilation, configuration, simulation, and testing, as it provides input to model checkers. 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.

This paper sketches the language, its design decisions, and its semantics.

[1] Email:reinhard.budde@ais.fraunhofer.de

[2] Email:axel.poigne@ais.fraunhofer.de

[3] Email:karl-heinz.sylla@ais.fraunhofer.de

[4] We assume familiarity with both, object-oriented design and synchronous programming.

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

2 Reactive Classes, Sensors, and Signals

Reactive classes. *synERJY* extends (a subset of) Java$^{\text{TM}}$ by reactive classes. A class is *reactive* if its constructor ends with the statement

```
active { ... }
```

that embeds the synchronous reactive code. This code is executed at every instant. A simple reactive class is

```
class Signals {
  Sensor<int> sensor   = new Sensor<int>(new SimInput());
  Signal<int> actuator = new Signal<int>(new SimOutput());

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

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

Both sensors and signals may be *present* or *absent*. A sensor or signal is present at an instant if and only if it is updated at an 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($sensor + 1); };` checks for the presence of the sensor `sensor`. If `sensor` is present the signal `actuator` is emitted with a new value being the value `$sensor` of the sensor increased by one.

The types

`Sensor<`*T*`>` and `Signal<`*T*`>`

are a new kind of built-in reference types where T is a primitive or class type. There are *pure* sensors and signals of type `Sensor` resp. `Signal` that do not have a value. Operators related to a sensors are

`?`*s* **:** checks whether the sensor s is *present* or *absent*.

`$`*s* **:** yields the *value* of the sensor s.

`@`*s* **:** yields a *time stamp* (in terms of the system clock) of when the sensor s has been present for the last time.

Signals can be 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. `emit` s is used for pure signals.

Hence, `Signal<`*T*`>` may be considered as a subtype of `Sensor<`*T*`>`.

Interfacing. Signals are always private, as are all fields and methods of an reactive object. 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 its constructor. If the constructor has no argument the signal is local. It is an output signal if the signal constructor has a parameter of interface type `Output`. Constructors of sensors must always have an argument of interface type `Input`.

The interface types `Input` and `Output` are so-called marker interfaces meaning that they act as a place holder lacking any semantic content. Implementations, however, must provide appropriate callback methods.

- *input sensors*: a method `new_val`, and a method `get_val` with result type T if the sensor is valued.
- *output signals* a method `put_val` with a parameter of type T if the signal is valued.

According to the synchronous execution model, input sensors are set at the beginning of an instant: the method `new_val` is called. If it returns the value `true`, the sensor is set to be present, and, in case it is valued, its value is updated by the value obtained as a result of `get_val`. Output signals are communicated to the environment at the end of an instant: 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.[5]

3 Reactive Control

Reactive statements. The reactive statements of *synERJY* are[6]

assignment	x `=` E`;`	*method call*	$m(E_1, ..., E_n)$`;`
emittance	`emit` s`(`v`);`	*nothing*	`nothing;`

sequential composition	P_1 `...` P_n
parallel composition	`[[` P_1 `|| ... ||` P_n `]];`
conditional	`if (`E`) {` P `} else {` Q `};`
loop	`loop {` P `};`
(weak) preemption	`cancel {` P `} when (E);`
activate	`activate { P } when (E);`

A method call may either be the call of a void data method, or the call of a reactive

[5] The classes `SimInput` and `SimOutput` are builtin for convenience. They organize the interaction with the *synERJY* simulator. "Real" applications need "real" adaptors to the environment.

[6] The notation differ from that of Esterel on purpose since operators differ in meaning. For instance, the various preemption operators of Esterel are combined in the `cancel` operator.

method. A method is reactive if its body contains a reactive statement. Reactive method are expanded in-line, i.e. the method call is replaced by its body.

The **next** statement is the only statement to consume time: if started it terminates only in the next instant. Sequential composition, parallel composition, loop, and conditional behave as to be expected in a synchronous language.

Preemption is the most prominent reactive statement. The format is

```
cancel [ strongly ] [ next ] {
  P
} when (E_1) [ { P_1 } ]
[  else when (E_1) [ { P_1 } ]
    ...
[  else when (E_n) [ { P_n } ];
```

with the clauses enclosed by [...] being optional. We distinguish weak and strong preemption: for weak preemption, the body P is executed at an instant before the conditions E_1,...,E_n are evaluated successively. If E_i is the first condition to hold, the statement P_i is evaluated if defined. Further evaluation of the body P is cancelled. For strong preemption the conditions are evaluated before executing the body. The latter is indicated by the modifier `strongly`. Preemption may only start to be effective in the next instant after starting to execute a cancel statement. This is indicated by the modifier `next`[7]

A simple example may illustrate the style of presentation (where `await` E is a shorthand for `cancel {halt;} when (`E`);`)).

```
class Counter {
  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
```

[7] In that "`cancel strongly next` P `when (`E`)`", for instance, corresponds to Esterel's "`do` P `watching` E".

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

Processes for semantics. The semantics corresponds – with minor modifications – to that specified in [9]. The general idea is that each reactive statement P denotes a semantic entity p we refer to as a *synchronous process*. A synchronous process is presented in terms of "assembler" statements of the form

```
s <= φ            (set the wire s if φ holds)
s <= φ { f }      (set the wire s and execute f if φ holds)
r <- φ            (set the register r if φ holds)
```

The distinction of wires and registers is that, if the condition ϕ evaluates to true, the wire s is set to be "up" (and f is executes) at the present instant. In contrast, the register r is set for the *next* instant.

The translation of statements is denotational, i.e. the behaviour of a statement is synthesised from that of its sub-statements within an environment. Consider, for instance, the loop statement:

$$\begin{aligned}[\![\, \texttt{loop}\ \{P\}\ ]\!]\ \alpha\ \beta\ \tau = \texttt{let}\ \gamma = \mathit{new\_wire}()\ \texttt{and}\ p = [\![\ P\ ]\!]\ \gamma\ \beta\ \tau\ \texttt{in} \\ \gamma\ \texttt{<=}\ \alpha\ \texttt{|}\ p.\omega \\ p\end{aligned}$$

The environment consists of the "system wires" α, β, and τ. The wire α is up only at the instant when the process is started, β in all later instants. The wire τ is used for preemption.

This is the interpretation of the denotational equation above: the process p is obtained by translating the loop body P within the new environment γ, β, and τ. The new wire γ is set if either the wire α, or the wire $p.\omega$ is up. The latter is a particular (synthesised) wire that is up if and only if the process p terminates. The denotational semantics of the loop statement is comprised of the process p together with the definition of the wire γ. Hence, if the loop statement is started, the process p is started. If p terminates it is restarted instantaneously. This scheme for the loop is used in the compiler but the actual implementation additionally takes care of reincarnation [4]. In general, the compiler exactly mimics this kind of denotational semantics.

If translated the compiler generates the following intermediate code for the counter example above: [8]

```
signals:
  Sensor  I10 is  Counter.start
  Sensor  I11 is  Counter.incr
  Signal  S1  is  Counter.elapsed
equations:
  G1     <= ((Beta & R4) | Alpha)
  G3     <= (((Beta & R1) | G1) & I10)
```

[8] where & stands for logical and, and | for logical or.

```
    A1     <= G3 { reset() }
    G5     <= (CC(A1) | G3)
    G7     <= ((Beta & R3) | G5)
    G9     <= (((Beta & R2) | G7) & I11)
    A2     <= G9 { increment() }
    G6     <= (CC(A2) | (Beta & (R3 | R2)) | G5)
    A3     <= G6 { D1 <= isElapsed() }
    G13    <= (CC(A3) | (G6 & (CC(A3) | D1)))
    A4     <= G13: Sv1 <= Val: null
    S1     <= G13
  memorisations:
    R4     <- (CC(A4) | G13)
    R3     <- ((CC(A2) | G9) & not(G13))
    R2     <- ((R2 | G7) & not((G9 | G13)))
    R1     <- ((R1 | G1) & not(G3))
```

At every instant, this sequential code is executed. The wire `Alpha` is up only in the very first instant of, the wire `Beta` at all later instants. One should note that execution of actions links the reactive with data code. A data action may affect the reactive behaviour in that, for instance, the wire `D1` is set if the data action `isElapsed()` executes to true.

Wavefront Computation and causality. The translation scheme sketched above generates a sequence of assembler statements which need to be sorted according to the "write-before-read" strategy of the synchronous paradigm. This strategy guarantees that signals have a consistent status – being either being present with a certain value, or being absent – at an instant. *synERJY* uses topological sorting.

One should note that the control structure of the *synERJY* program is encoded in the generated assembler code. This applies as well to data actions. The control dependencies of data actions are encoded using the `CC` operator. For instance, tracing the example above one can see that the data action `increment()` must take place before the data action `isElapsed()` since the wire `G6`, which triggers the latter, depends on `A2`. This correctly implements weak preemption.

In that topological sorting is only an approximation of the constructive semantics of [3]. But we believe that detecting any kind of cycle within the control and signal flow is a simple and reasonable criterion for the user to decide whether a program features a causality cycle or not.

Time Races and precedences. Execution of data actions may be conflicting, for instance, if two data actions access the same variable for reading or writing at an instant. *synERJY* checks for such conflicts we refer to as *time races*. Whenever a time race is possible at an instant, The compiler raises an error message since a time race may possibly cause non-deterministic behaviour. As with causality the analysis is on syntactical level. Typically time races occur between actions that are called in different branches of a parallel statement. Otherwise potential conflicts are resolved by the control flow.

synERJY offers several facilities to schedule conflicting actions using a precedence statement such as

```
precedence {
        isElapsed() < increment();
    };
```

This implies that, at an instant, any call of the data action `isElapsed()` must be scheduled before a call of the data action `increment()`.

Scheduling actions by name is a rather coarse strategy. *synERJY* provides a finer-grained scheduling mechanism using labels. Labels refer to individual statements within a reactive program. 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 `l1::` and `l2::` 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.

Actually, the example shows that labels may be used to resolve a second source of non-determinism: multiple emits. *synERJY* has abandoned using combinators as in Esterel since users in practice tend to resolve multiple emits by some ad-hoc combinator, e.g. by projecting to – typically – the first argument. This often results in unforeseen behaviour.

4 State Machines

Textual Presentation The textual syntax for automata is very simple. The statement

```
    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. The processes P_{entry}, P_{during}, and P_{exit} must be instantaneous.

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} is executed. 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* is of the form init $\{P\}$ with P being instantaneous. Finally, the statement next state *state_name* ; denotes the (instantaneous) jump to the next state.

An example. The example demonstrates how a hierarchical automaton can be specified.

```
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; };
};
```

Note that each “branch” should end with a next state statement. Note further that the do clause can hold any reactive statement. Here the parallel statement and the automaton statement are used to generate a typical hierarchical state machine.[9]

5 Embedding Data Flow - Hybrid Systems

Flow equations and modes. *synERJY* supports a Lustre-like sub-language for presenting data flow. Flow equations are of the form, e.g.,

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

with count being a signal.

Flow equations are allowed to occur within a variant of the sustain statement

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

Its body consists of a sequence of flow equations (and of local signal declarations). When started, the sustain statement never terminates; the flow equations 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

[9] *synERJY* provides an equivalent graphical notation as well as a graphical editor.

instance from a start-up mode to a working mode, or from a working mode to an error mode or maintenance mode.

Being a process like any other, the sustain statement may be preempted and (re-) started as in the following rather artificial example (which is similar to the automaton discussed earlier)

```
class CountingUpAndDown {
  Sensor reverse = new Sensor(new SimInput());
  Signal<int> count = new Signal<int>(new SimOutput());

  public CountingUpAndDown () {
    active {
      emit count(0);
      next;
      cancel {
          sustain {| count := pre(count) + 1; |}; counting up mode
        } when (?reverse);
      next;
      cancel {
          sustain {| count := pre(count) - 1; |}; counting down mode
      } when (?reverse);
    };
  };
}
```

There are two modes: counting up and down. The modes are switched when the signal `reverse` is present.[10]

Signals revisited. One may have noted that the signal `count` is emitted as well as constrained by a flow equation. *synERJY* promotes a uniform view of signals: signals may be updated either by using the `emit` statement or by applying a flow constraint. Both statements behave equivalently as far as signals are concerned:

- if a signal is updated, either by emitting or by applying a flow equation, it is present with a new value.

Both `emit` statements and flow equations are different means to specify the semantics of signals which is given in terms of traces

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	...
reverse	.	.	.	*	.	.	*	.	.	...
count	**0**	**1**	**2**	**3**	**2**	**1**	**0**	*0*	*0*	...

where the index i ranges over instants. The convention is that boldface indicates value and presence while italics indicate the value and absence.[11] Note that signals have a value even if absent.

Of course, there are conceptual differences between using the `emit` statement or a flow equation, one being more appropriate for control, the other signal processing.

[10] Since the `sustain` statement may be used in automata we achieve the effect of *mode automata* as defined in [8].

[11] For pure signals, we just use an asterisk for presence.

We assume the reader to be aware of these differences, hence skip a discussion.

Clocks, flow types, and signal types. *Flow expressions* (those used on the right hand side of a flow equation) follow the syntax of Lustre [7] using operators

```
pre  (previous)            when     (down-sampling)
->   (initialisation)      current  (up-sampling)
```

Clocks are considered as part of the type information. Flow expressions have clocks as defined in Lustre. Sensor and signal types are enhanced to have the general format

$$\texttt{Sensor}\{C\}\texttt{<}T\texttt{>} \quad \text{resp.} \quad \texttt{Signal}\{C\}\texttt{<}T\texttt{>}$$

where the “clock” C is a Boolean flow expression. Signals that are “emitted” always have clock `true`. Hence the type `Sensor<`T`>` is a shorthand for `Sensor{true}<`T`>`, and `Signal<`T`>` for `Signal{true}<`T`>`.

The only difference between the emit statement and a flow equation is that only for a flow equations clocks are checked according to the rules of Lustre. In that flow equations are more restricted, the reason being that, in case of down-sampling and up-sampling, the restriction provides better semantic control.

Note that, in contrast to traditional data flow languages, updating of a signal is not restricted to a single flow equation. In case of signals of clock `true` in particular, the signals may be both, emitted *and* constrained by data flow equations. Note further that, at an instant, a signal may be neither, nor emitted nor constrained. This is the very basis of of the unification of synchronous formalisms supported by *synERJY*.

Hybrid systems. Hybrid systems switch between modes where each mode is governed by its own characteristic dynamic laws. Mode transitions are, for instance, 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.

As a typical presentation of a hybrid system we consider a bouncing ball:

- Motion is characterised by *height* (x_1) and *vertical* velocity (x_2),
- Continuous changes between bounces.
- Discrete change at bounce time.
- Dynamics summarised by
  - · one *mode* q with a continuous behaviour specified by the equations

    $$\dot{x_1} = x_2$$
    $$\dot{x_2} = -g$$

  - · 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 $-c * x_2$.

This behaviour is captured by the automaton

```
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; };
};
```

We comment on the program:

- An equation such as $\dot{x} = e$ is replaced by an integral $x = x_0 + \int edx$, and the integral is computed by the difference equation $x_n = x_{n-1} + e(n) * dt$ with initial condition x_0.

- `dt` is a predefined signal of primitive type `time` the value of which is the amount of "real time" passed between two instants.[12]

- The `during { ...}` is a second pattern in which flows may occur. The flow equations are only executed if control is in the respective states (not when jumping into it).

- The initialization `->` operator is defined relative to a flow context: initialization always takes place at the instant the flow context is started. Hence, in case of the example, whenever the value `x1` is smaller than 0.0, control reenters the state, and in the next instant the value of `x2` is initialized by the previous value of `x2` reduced by the factor c. Then the dynamic law $\dot{x_2} = -g$ applies upto the next bounce.

  This "localized" version of initialisation exceeds the standard semantics as defined in Lustre where initialisation refers to the very first instant of running a system. Local (re-) initialisation, however, comes handy for hybrid systems.[13]

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`.[14]

[12] "Real time" is specified in terms of the system clock. *synERJY* sports several other useful features related to real time, for instance a statement `await 3sec` with the obvious connotation. This is handled within the framework of the synchrony paradigm since "time" is handled like an input signal, always being updated at the beginning of an instant. Hence the resolution of real time is determined by the frequency of instants.

[13] There is a similar effect for the `pre` operator; it is set to the default value at the instant when entering a flow context except if its argument is a signal field as in case of the example.

[14] 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 .

6 Signal Bus for Interfacing Reactive Objects

Parameterizing reactive classes. 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 above: the sensors and signals become parameters of the constructor

```
class Counter {
   public Counter (int d,Sensor start,Sensor clock,Signal elapsed )  {
     latch = d;
     active {
        ...
     };
  };
  ...
}
```

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.

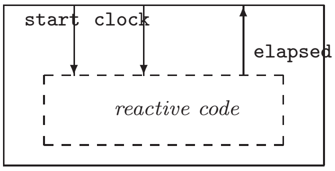
```
class PulseWidthModulation {
  static final int high = 5;  // constants for counting
  static final int  low = 15;

  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;
        };
    };
  };
}
```

The signal `wave` is emitted with value `true` if the value `toLowPhase` 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 semantics of object composition is that, when generating an instance of class `PulseWidthModulation`,

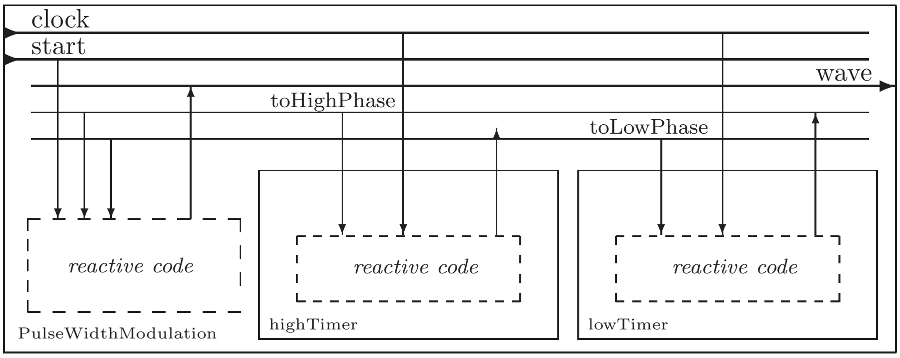
- signal parameters are substituted by arguments, e.g. the signal parameter `start` of a counter is substituted by the signal argument `toHighTimer` when initialising of the variable `highTimer`.
- the reactive code of the object `PulseWidthModulation` and of all its reactive sub-objects – here the two counters `highTimer` and `lowTimer` – are put in parallel.

In terms of pictures. 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. 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 then be presented by



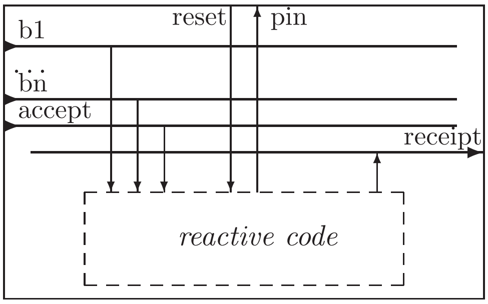
The picture suggests that the reactive codes of the objects involved are executed in parallel and that the different fragments of code communicate via a bundle of signals. We speak of a *signal bus* to refer to this bundle. The signal bus is comprised of all signal (fields) specified in a class. We distinguish local signals such as `toHighPhase` and `toLowPhase`, input signals such as `clock` and `start`, and output signals such as `wave`. In general

- reactive objects and signal busses form a static hierarchy
- 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.

Input and output signals reconsidered. 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.

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 (*information hiding* in other terms).

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 a 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.

7 Related Work

synERJY inherits its reactive concepts from Esterel and Lustre. Argos [8] has been the first language to integrate data flow with automata, while SyncCharts [1] has added automata to Esterel. There are several approaches of adding "synchronous behaviour" to standard programming languages. Typically add-ons are provided in terms of libraries that allow to specify the notion of an instant (e.g. [5]). In general the embedding is more shallow in that, for instance, a compile time analysis of causality and time races is not provided. Causality if often avoided by changing the synchronous model. In comparison *synERJY* faithfully implements the synchronous execution model.

8 Concluding Remarks

Designing *synERJY* we have spent much effort on a smooth integration of the concepts presented. It took many iteration to achieve a presentation that – we hope – is acceptable to both, JAVA^{TM}programmers as well as adepts of synchronous programming.

Since *synERJY* in particular targets micro controllers, efficiency of code is a major aim. The compiler generates standard C as an intermediate code that can be deployed using a cross compiler. Libraries are provided for some standard micro controllers that encapsulate the operations of the controller. In that case all the development up to register and bit level using the interrupts and timers can be done in *synERJY*. Future work will focus on extending the number of target architectures, and on further improving efficiency.

The language has been used in several student courses, and in in-house applications in robotics. It is freely available at www.ais.fraunhofer.de/∼budde.

References

[1] André, C.. "Representation and analysis of reactive behaviours: A synchronous approach," in: Proc. CESA'96, Lille, France, July 1996.

[2] Benveniste, A., P. Le Guernic, and C. Jaquemot, "Synchronous Programming with Events and Relations: the SIGNAL Language," Science of Computer Programming **16**(2) (1991), 103 – 149.

[3] Berry, G., *The Constructive Semantics of Pure Esterel*, Draft book, www-sop.inria.fr/meije/esterel/doc/main-papers.html, 1999

[4] Berry, G. and G. Gonthier. "The ESTEREL synchronous programming language: design, semantics, implementation," Science of Computer Programming **19**(2) (1992), 87–152.

[5] Boussinot, F., and J.-F. Susini, "Java threads and SugarCubes," Software - Practice and Experience **30**(5) (2000), 545-566.

[6] Harel, D., "Statecharts: A visual approach to complex systems," Science of Computer Programming, **8** (1987),231–274.

[7] Halbwachs, N., P. Caspi, P. Raymond, and D. Pilaud. "The synchronous dataflow programming language Lustre," Proceedings of the IEEE **79**(9) (1991), 1305–1320.

[8] Maranchini, F., Y. Rémond. "Mode-Automata: About Modes and States in Reactive Systems," Proc. European Symposium on Programming, Lisbon, Portugal, 1998

[9] A. Poigné, and L. Holenderski, "On the Combination of Synchronous Languages," in: W.P. de Roever (ed.), "Workshop on Compositionality, The Significant difference," Lecture Notes in Computer Science 1536 , Springer, Heidelberg, 1998, 490 – 514.