Synchronous reactive programming

Marcus Sundman
Department of Computer Science
Åbo Akademi University, FIN-20520 Åbo, Finland
e-mail: marcus.sundman@iki.fi
URL: http://www.iki.fi/marcus.sundman/

Abstract
The purpose of this paper is to be an introduction to the synchronous and the reactive programming paradigms. Different concepts will be presented in the form of examples in different existing programming languages. One simple, although not trivial, example is used in all of the presented languages to show the characteristics of them and to simplify a comparison of the principles through the different approaches taken.

ACM Classifications

CCS: D.1.3 Concurrent Programming, D.1.7 Visual Programming, D.2.2 Design Tools and Techniques, D.2.11 Software Architectures, D.3.2 Language Classifications, D.3.3 Language Constructs and Features, F.3.3 Studies of Program Constructs.

SIG: SIGPLAN.

1. Introduction

Reactive systems
Programming systems are generally divided into two categories, namely transformational and interactive. Programs of the former kind are executed with some input parameters, produce some result and then exit, while programs of the latter kind of systems can also interact with the environment and may not exit. Reactive systems fall into this latter category, but while interactive systems usually may drive the communication with their environment at their own pace, reactive systems react to stimuli immediately ([HP85], [Hal98], [Ber99]). Thus the reaction of reactive systems is passive, while in interactive systems it is active. In other words, reactive systems are completely driven by their environment whereas interactive systems operate more or less independently. Reactive programs with timing constraints are more commonly known as real-time programs.

Synchronous systems
The main hypothesis of the synchronism paradigm presented by A. Benveniste and G. Berry in [BB91] is that a) operations take no time at all (i.e. they are atomic in all regards), and b) time is only an abstract concept deduced from the order of events. Thus time is divided into a sequence of discrete instants, called ticks. Synchronous programming systems can then be used to control what happens in which tick. This translates very well to digital circuits where
the time between two consecutive ticks is regarded as none. The separation of time and operation also helps in creating real-time systems with other constraints than time, which they usually have.

The first formal language designed for reactive systems, named Statecharts and intended mainly as a specification and design formalism, was described in [H87]. It contains many features of the synchronous model, but also has some shortcomings, one of which is the lack of determinism. Nonetheless it greatly influenced the four programming languages of the reactive model that were developed at the same time; the imperative language Esterel ([Ber99], [BG92]), the data-flow languages Signal ([RG93], [LPM95]) and Lustre ([H93], [HCRP91]), and Argos ([M91], [M89]), a language very closely based on Statecharts.

Applications

Synchronous reactive systems are most commonly used in critical systems. This is partly because synchronous reactive programs are deterministic and therefore the correctness of a program can be verified both manually and automatically. Another reason is that most critical systems are reactive by nature, and thus using a reactive system comes as a natural choice.

Although computers are deterministic at heart the concurrent execution of multiple programs or threads leads to an apparent non-determinism. This is unfortunate since it is much easier to specify and verify deterministic systems than non-deterministic ones. A bug in a non-deterministic system may or may not appear in consecutive runs. Pinpointing a bug or finding out all possible side-effects of a program can be next to impossible because of the state space explosion that non-determinism brings. On the other hand the program flow can be easily tracked in deterministic programs, which makes it possible to make effective verification tools. Determinism also makes it possible to calculate maximum response times accurately, which is paramount in real-time applications.

Reactive programs are generally seen as suited only for critical systems and a few other niche areas (e.g. artificial intelligence) but there is really no reason why they could not be used in other systems as well. There may be reasons why the currently available reactive systems are not widely adopted by the general programming society, but that does not mean that the paradigm be reserved for critical systems only. As a matter of fact reactive systems are based on the most fundamental principle of science and our reasoning in general, namely causality. Therefore it would be logical to use this paradigm as the foundation when constructing a programming language.

2. Programming languages

An example program will be used when describing the programming languages below. The aim of the program is to make the robot depicted in Figure 1 follow a black line. The robot is equipped with two independently controllable wheels and two optical sensors. Each wheel controller accepts input as a floating point value in the range [-1, 1], representing the relative angular velocity. The sensors react to dark surfaces by emitting a signal until encountering a light surface again. Figures 1b-1d shows the required reactions to circumstances under normal
operation. Figure 1e shows an exception that might occur e.g. when the line is first found, and the shown reaction should continue until the left sensor no longer detects the line.

The imperative language Esterel

Esterel has been in development since 1983 at CMA¹ and INRIA in Sophia-Antipolis. The authors have adopted a Write Things Once philosophy throughout the language construction. The language is based on signals, of which there are two types, pure signals and valued signals. Pure signals only have a presence status, present or absent, while valued signals also have a value of an arbitrary, albeit specified, type. Result signals (i.e. outputs) are concurrent with the cause signals (i.e. inputs), as dictated by the synchronism paradigm. All signals are broadcasted and the propagation takes no time, which means that the resulting signals arrive at their destination the same instant as the signals that caused them. This leads to the possibility of causal paradoxes and non-determinism.

The inverter feedback loop in Figure 2b has a causal paradox. The synchronous inverter makes $I = \neg O$, but since signal propagation is instantaneous $I = O$, and thus we have the contradiction $O = \neg O$. The feedback loop in Figure 2c, on the other hand, has two stable states, which means that it is non-deterministic. We have $I_a = \neg O_a \land I_b = \neg O_b \land O_a = I_b \land O_b = I_a$, which holds for all values of $I_a$, $O_a$, $I_b$ and $O_b$. It is the responsibility of the compiler to discard all programs that contain causal paradoxes or non-deterministic signal values.

Being a reactive language Esterel is highly concurrent. Besides the implicit parallel execution caused by multiple reactions to one or more concurrent signals, the language also includes an explicit construct for parallel execution, $A \parallel B$, which immediately starts executing $A$ and $B$ in parallel and finishes at the first instance when both $A$ and $B$ have finished executing. As with

---

¹ Applied Mathematics Center, Ecole des Mines de Paris
all synchronous languages, time is multiform, i.e. the repetition of any signal can be considered as defining its own time measure.

Listing 1 shows an Esterel program for controlling the robot shown in Figure 1. The sensor inputs (line 2) are pure signals which are present whenever the line is visible to them. The wheel controlling outputs (line 3) are valued signals. The body of the module (lines 4-24) is an infinite loop that first waits for the next tick and then executes three parallel statements. The first one (lines 7-9) emits the $\text{LeftWheelSpeed}$ signal with the floating point value 1.0 (i.e. full speed forward) whenever $\text{LeftSensor}$ is not present (Figure 1b and Figure 1d). The second statement (lines 11-13) does the same for the right wheel (Figure 1b and Figure 1c). The third one (lines 15-22) continuously emits two signals (right wheel full speed ahead, left wheel half speed backward) in sequence (using the operator ';') each tick when

```
1 module ROBOT:
  2 input LeftSensor, RightSensor;
  3 output LeftWheelSpeed : float, RightWheelSpeed : float;
  4 loop
  5    await tick;
  6        [
  7            present LeftSensor else
  8                emit LeftWheelSpeed(1.0f)
  9            end present
 10        ||
 11            present RightSensor else
 12                emit RightWheelSpeed(1.0f)
 13            end present
 14        ||
 15            present [LeftSensor and RightSensor] then
 16                abort
 17                    loop
 18                        emit RightWheelSpeed(1.0f);
 19                        emit LeftWheelSpeed(-0.5f)
 20                    each tick
 21                        when [not LeftSensor]
 22                        end present
 23                 ]
 24    end loop
 25 end module
```

Listing 1 A program for a robot to follow a line, written in Esterel.
both sensor signals are present (Figure 1e) until LeftSensor is no longer present. Although the two operations on lines 18 and 19 are executed in sequence they still are executed at the same instant, i.e. within the same tick, since the emit operator returns immediately.

**The data-flow language Signal**

As its name suggests, Signal is a language for manipulating signals. A signal is an unbounded series of values of a specified type, and can either be present or absent in some instant. The temporal sequence of instants when a signal is present is defined as the *clock* of the signal. Like Esterel, also Signal has a kind of pure signals, called *event*, and they are given the boolean value *true* at each instant.

Programming languages based on the data-flow paradigm usually have a purely functional model, which means that operations have no side effects. The data-flow paradigm does not require this per se, but it follows from their equational nature. Purely functional programming languages also have other advantages, such as easy formal verification and enabling heavy code reuse.

Data-flow languages are based on a highly parallel model, going beyond even other types of reactive languages. The dependency of data is the only thing forcing synchronization and sequencing.

The Signal language is minimalistic. Its main purpose is to specify relations between values and clocks of signals. Elementary processes, which are simply systems of equations, are joined in parallel into larger processes, which in turn can be used in even larger processes, creating process hierarchies. The parallel composition construct, \( |\delta| \), operates on one or more processes, denoted by \( \delta \), separated by | (e.g. \( |A|B|C| \) defines \( A, B \) and \( C \) as concurrent). Besides the usual logical and arithmetic operations, there are a few operators defining elementary processes:

- the undersampling operator *when*: \((X := A \text{ when } B)\) assigns \(X\) the value of \(A\) if \(B\) is present and the value of \(B\) is *true*.

- the oversampling operator *default*: \((X := A \text{ default } B)\) assigns \(X\) the value of \(A\) if \(A\) is present, otherwise the value of \(B\) if \(B\) is present.

- the delay operator \(\$: (X := A\$B)\) assigns \(X\) the \(B\):th previous value of \(A\). The delay operator can take a default value that is used when requesting values from before the beginning of the stream. E.g. \((X := A\$1 \text{ init } 5)\) assigns \(X\) the previous value of \(A\) if there is one, or the value 5 otherwise.

Listing 2 shows two implementations of the robot program in Signal. They differ mainly in the way they handle the exceptional case shown in Figure 1e. The first implementation uses a boolean signal, Rotating, to simulate a state in much the same way as it is done in most non-reactive programming languages. It is always set to *true* when both sensor signals are present (line 1), and remains as *true* while LeftSensor is present (line 2). The wheels’
speeds are calculated in a straightforward manner (lines 4-6), but the calculations of the two states are mixed together in this solution, which thus does not scale well. This deficiency is eliminated in the second implementation. It consists of three concurrent processes. In the first one (lines 9-11) \( \text{Rotate} \) is defined as the constraint interval between when both sensor signals are present and when \( \text{LeftSensor} \) is no longer present (line 9), and \( \text{FollowLine} \) is defined as the complement to \( \text{Rotate} \) (line 10). The output calculations are divided into two separate processes, one for when following the line (lines 12-14) and one for when rotating (lines 15-17). This solution groups the calculations by function, making the maintenance easier, as well as making the calculations themselves simpler.

Listed below are the code implementations for two Signal processes, each of which makes a robot follow a line. The first implementation (lines 1-7) uses a delay on a boolean signal, \( \text{Rotating} \), to simulate a state. The second implementation (lines 9-18) uses a constraint interval to define two complementary states, \( \text{Rotate} \) and \( \text{FollowLine} \).

```
1 (| Rotate := true when (LeftSensor and RightSensor)
2     default ((Rotating$1 init false) when LeftSensor)
3     default false
4  | LeftWheelSpeed := -0.5 when Rotating
5     default (1.0 when (not LeftSensor))
6  | RightWheelSpeed := 1.0 when (Rotating or (not RightSensor))
7  |
8
9 (| (| Rotate := ])(LeftSensor and RightSensor), (not LeftSensor)]
10  | FollowLine := compl Rotate
11  |
12  | (| LeftWheelSpeed := 1.0 when (not LeftSensor)
13     | RightWheelSpeed := 1.0 when (not RightSensor)
14     |) on FollowLine
15  | (| LeftWheelSpeed := -0.5
16  | RightWheelSpeed := 1.0
17     |) on Rotate
18  |)
```

Listing 2 Two Signal processes, each of which makes a robot follow a line. The first implementation (lines 1-7) uses a delay on a boolean signal, \( \text{Rotating} \), to simulate a state. The second implementation (lines 9-18) uses a constraint interval to define two complementary states, \( \text{Rotate} \) and \( \text{FollowLine} \).

The data-flow language Lustre

One of the aims of the Lustre project was to develop techniques of formal verification based on the data-flow paradigm. The language is based on signal flows very much like Signal. The flow of instants that a variable or expression is present is called its clock, and the main clock of a program is called its basic clock. Any boolean variable or expression can be used as a clock, whose instances corresponds to its value being true. Since the variable or expression itself has a clock it follows that that it must be faster than the clock it defines. Thus all other clocks in a program are slower than the program’s basic clock. This differs from how clocks work in other data-flow languages, such as Signal, where faster clocks may be defined.

All variables must be explicitly declared, and the type of a variable is specified in its declaration. Each output or local variable must be given one and only one definition, which is done with an equation. The equation \( \text{A=E} \) defines the variable \( \text{A} \) to be equal to \( \text{E} \), which is either a variable or an expression. The equation is not symmetric, but although \( \text{A} \) and \( \text{E} \) are
defined to be equal it is only the variable $A$ that is actually defined by the equation. The definition of a variable, in this case $A$, is total and no other equation or operator can modify it, which implies that $A$ and $E$ are fully interchangeable everywhere else they are used. This is called the substitution principle and is a very fundamental part of the language.

<table>
<thead>
<tr>
<th>the basic clock</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>A that has the basic clock</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>a clock, $C$, based on $A$</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B that has the clock $C$</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a clock based on $B$</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Table 1 The relations between clocks and boolean signal flows.*

All operands of an operator must have the same clock. In addition to the normal operators there are also four temporal operators in Lustre:

- **pre($A$)**: the delay operator, which is equivalent to the expression $A$·$1$ in Signal.

- **$A$->$B$**: if $A$ has the sequence $(a_1, a_2, a_3, a_4, ...)$ and $B$ has the sequence $(b_1, b_2, b_3, b_4, ...)$ then $A$->$B$ has the sequence $(a_1, b_2, b_3, b_4, ...)$.

- **$B$ when $A$**: this is equivalent to *when* in Signal, i.e. a sequence whose clock is defined by $A$ and whose values are those of $B$ (see Table 2).

- **current $C$**: this is a sequence whose clock is the clock of the sequence defining $C$’s clock and whose values are those of $C$ (see Table 2).

<table>
<thead>
<tr>
<th>$A$</th>
<th>false</th>
<th>true</th>
<th>false</th>
<th>false</th>
<th>true</th>
<th>true</th>
<th>false</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B$</td>
<td>$x_1$</td>
<td>$x_2$</td>
<td>$x_3$</td>
<td>$x_4$</td>
<td>$x_5$</td>
<td>$x_6$</td>
<td>$x_7$</td>
</tr>
<tr>
<td>$C = B$ when $A$</td>
<td>$x_2$</td>
<td>$x_3$</td>
<td>$x_4$</td>
<td>$x_5$</td>
<td>$x_6$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$D = current C$</td>
<td>nil</td>
<td>$x_2$</td>
<td>$x_2$</td>
<td>$x_2$</td>
<td>$x_5$</td>
<td>$x_6$</td>
<td>$x_6$</td>
</tr>
</tbody>
</table>

*Table 2 An example demonstrating the when and current operators.*

The Lustre program for controlling the robot (Listing 3) is very similar to the first Signal program in Listing 2. This is hardly surprising considering the many similarities of the two languages. The whole program consists of only three equations. A local variable, $is\_rotating$ (line 3), is defined as the boolean flow of values denoting when the robot is rotating (Figure 1e). It will be true whenever both sensors are true (line 5), otherwise it will take its previous value as long as $LeftSensor$ is true (line 6). This equation is not really necessary since the substitution principle would allow $is\_rotating$ to be replaced with the expression anywhere, but it simplifies the other two equations, which now are trivial.
The Argos language is completely based on automata, which are described by states and labeled transitions between states. A transition is triggered by one or more events, and may in turn broadcast any number of events. All broadcasts are global and synchronous, which means that if a transition, $T_A$, from state $A_1$ to $A_2$, broadcasts an event that triggers another transition, $T_B$, from state $B_1$ to $B_2$, the states $A_2$ and $B_2$ will be entered simultaneously. Transitions are labeled triggers/events, so $A, B, C/D, E$ is a transition that is triggered either by $A$ or by $B$ and $C$, and when triggered it broadcasts the events $D$ and $E$.

The most critical problem with Statecharts is that there are no fully independent modules. Although states can be grouped together they are still accessible from outside the group. By contrast, components in Argos are completely independent. When there is a transition to a component with one or more subcomponents then all default transitions (i.e. unlabeled transitions without a source) within the component are executed. When there is a transition from a component with subcomponents then transitions are also made from all its active subcomponents. Thus the external behavior of a component does not change if subcomponents and subtransitions are added or removed. This makes Argos very modular as well as predictable. By comparison, other similar languages suffer from the problem where a module can unpredictably change its behavior as a result of a modification in an unrelated module.

The main component (Figure 3a) of the Argos robot program is very simple. It consists of three subcomponents, named Sensors, LeftWheel and RightWheel, which operate in parallel, which is denoted by the dashed lines. The Sensors component (Figure 3b) is by far the most complex of these although it still is quite simple. It only translates the two input sensor signals, LeftSensor and RightSensor, into four events, namely sNone, sLeft, sRight and sBoth. E.g. the transition RightSensor/sBoth from LeftSensorOnly to BothSensors is executed when the right sensor detects the line while the left sensor sees it, and in the same instant sBoth is broadcasted, denoting that there was a transition to the state depicted in Figure 1e. The RightWheel component (Figure 3d) is trivial. It broadcasts RW_STILL when it receives sRight while the right wheel is

```plaintext
Listing 3 A program for a robot to follow a line, written in Lustre.

1 node Robot(LeftSensor, RightSensor: boolean)
2 returns (LeftWheelSpeed, RightWheelSpeed: real);
3 var is_rotating: bool;
4 let
5    is_rotating = if (LeftSensor and RightSensor) then true
6    else (false -> (pre(is_rotating) and LeftSensor));
7    LeftWheelSpeed = if is_rotating then -0.5
8          else if LeftSensor then 0.0
9              else 1.0;
10   RightWheelSpeed = if is_rotating then 1.0
11       else if RightSensor then 0.0
12           else 1.0;
13 tel.
```

The language Argos

The Argos language is completely based on automata, which are described by states and labeled transitions between states. A transition is triggered by one or more events, and may in turn broadcast any number of events. All broadcasts are global and synchronous, which means that if a transition, $T_A$, from state $A_1$ to $A_2$, broadcasts an event that triggers another transition, $T_B$, from state $B_1$ to $B_2$, the states $A_2$ and $B_2$ will be entered simultaneously. Transitions are labeled triggers/events, so $A, B, C/D, E$ is a transition that is triggered either by $A$ or by $B$ and $C$, and when triggered it broadcasts the events $D$ and $E$.

The most critical problem with Statecharts is that there are no fully independent modules. Although states can be grouped together they are still accessible from outside the group. By contrast, components in Argos are completely independent. When there is a transition to a component with one or more subcomponents then all default transitions (i.e. unlabeled transitions without a source) within the component are executed. When there is a transition from a component with subcomponents then transitions are also made from all its active subcomponents. Thus the external behavior of a component does not change if subcomponents and subtransitions are added or removed. This makes Argos very modular as well as predictable. By comparison, other similar languages suffer from the problem where a module can unpredictably change its behavior as a result of a modification in an unrelated module.

The main component (Figure 3a) of the Argos robot program is very simple. It consists of three subcomponents, named Sensors, LeftWheel and RightWheel, which operate in parallel, which is denoted by the dashed lines. The Sensors component (Figure 3b) is by far the most complex of these although it still is quite simple. It only translates the two input sensor signals, LeftSensor and RightSensor, into four events, namely sNone, sLeft, sRight and sBoth. E.g. the transition RightSensor/sBoth from LeftSensorOnly to BothSensors is executed when the right sensor detects the line while the left sensor sees it, and in the same instant sBoth is broadcasted, denoting that there was a transition to the state depicted in Figure 1e. The RightWheel component (Figure 3d) is trivial. It broadcasts RW_STILL when it receives sRight while the right wheel is
rotating, and RW_FWD when it receives sNone, sLeft or sBoth while the right wheel is still. The LeftWheel component (Figure 3c) is somewhat more complicated. Besides the behavior analogous to RightWheel it broadcasts LW_REV when it receives sBoth and ignores all sLeft events while in the LeftWheelBackward state.

The global events LW_STILL, LW_FWD, LW_REV, RW_STILL and RW_FWD should be bound to the operations LeftWheel:=0.0, LeftWheel:=1.0, LeftWheel:=-0.5, RightWheel:=0.0 and RightWheel:=1.0, respectively, on the host system for this robot program to according to the specifications.

3. Conclusion

I have first introduced the synchronous paradigm and how reactive programs differ from other kinds of programs. Then I have described a programming task for controlling a simple robot that should follow a line. While presenting four different programming languages I have shown different approaches to solving the given problem. The four example programs, one for each language, demonstrated various characteristics of reactive and synchronous programming, the foremost of which is concurrency.
References


