Synchronous Programming[*]

Paul Caspi
CNRS, Verimag Laboratory[†]

Pascal Raymond
CNRS, Verimag Laboratory[‡]

Stavros Tripakis
CNRS/Verimag and Cadence Berkeley Labs[§]

September 11, 2006

1 Introduction

Synchronous programming is a school of thought in embedded software which has been in the air for a while but without being always given clear and thorough explanations. This has led to many misunderstandings as, for instance, opinions like: "we do not use synchronous programming because null execution time is not realistic and infinitely fast machines do not exist". At the same time many practitioners adopted (implicitely or not) the synchronous programming principles and, by now, synchronous programming is used in real-world safety-critical systems like the "fly-by-wire" systems of Airbus commercial aircrafts (A340, A380). This discrepancy between a valuable industrial usage and charges of unrealism clearly shows that better explanations are urgently needed. This chapter hopefully aims at contributing to a better understanding.

In order to provide these mandatory explanations, we propose to take a historical perspective by showing that synchronous programming has arisen from the interaction between:

1. the practices of control and electronic engineers in the early times of digital computers when they

[*] This work has been partially supported by the European project ARTIST2 (IST-004527).

[†] Centre Equation, 2, avenue de Vignate, 38610 Gières, France, caspi@imag.fr

[‡] Centre Equation, 2, avenue de Vignate, 38610 Gières, France, raymond@imag.fr

[§] 1995 University Ave, Suite 460, Berkeley, CA 94704, USA, tripakis@cadence.com

```
initialize state;
loop
   wait clock-tick;
   read inputs;
   compute outputs and state;
   emit outputs
end-loop
```

Table 1.1: Basic synchronous program structure

adopted this new technology in replacement of analog devices (section 2),

2. and the concerns of some computer theoreticians who were trying to better account for the modelling of reactive computing systems (section 3).

Indeed, synchronous languages arose from the (unlikely) convergence of these two very different concerns and we shall explain how this convergence took place. Then we describe some of the languages at section 4. As we shall see, these languages are both quite useful and yet with very restrictive implementation capabilities and we describe in section 5 the possible extensions that have been brought, through time, to the initial concepts.

2 From practice...

2.1 Programming practices of control and electronic engineers

In the early eighties, microprocessors appeared as very efficient and versatile devices, and control and electronic engineers were quickly moving to using them. Yet, these devices were very different from the ones they were used to, i.e., analog and discrete components coming along with a "data sheet" basically describing some "input-output" behavior. Here, on the contrary, the device was coming along with a larger user manual, encompassing a so-called "instruction set" that users had to get used to. It should be noted that these users were almost totally ignorant of elementary computer engineering concepts like operating systems, semaphores, monitors and the like. Consequently, after some trial and error, most of them finally settled with a very simple program structure, triggered by a single periodic real-time clock, as shown in Table 1.1.

2.2 The interest of the approach

Though this program structure may appear as very basic, it still has many advantages, in the restricted context it is applied to:

It requires a very simple operating system, if any. In fact, it uses a single interrupt, the real-time clock, and from the real-time requirement, this interrupt should occur only when the loop body is over, that is to say, when the computer is idle. Thus, there is no need for context switching and this kind of program can even be implemented on a "bare" machine.

Timing analysis is the simplest one. Timing analysis amounts to checking that the worst-case execution time (WCET) C of the loop body is smaller than the clock period T. In order to make this check even easier, people took care of forbidding the use of unbounded loop constructs in the loop body as well as forbidding the use of dynamic memory and recursion so as to avoid many execution errors. Despite these restrictions, the problem of assessing WCET has been constantly becoming harder because of more and more complex hardware architectures, with pipelines, caches, etc. Still, this is clearly the simplest possible timing analysis.

It perfectly matches the needs of sampled-data control and signal processing. Periodic sampling has a long lasting tradition in control and signal processing and comes equipped with well established theories like the celebrated Shannon-Nyquist sampling theorem or the sampled data control theory [1].

All this is very simple, safe and efficient. Safety is also very important in this context as many of these control programs apply to safety-critical systems like commercial aircraft flight control, nuclear plant emergency shutdown, railway signalling and so on. These systems could not have experienced the problems met by the Mars Pathfinder due to priority inversion [2] in programs based on tasks and semaphores. As a matter of fact, operating systems are extremely difficult to debug and validate. Certification authorities in safety-critical domains are aware of it and reluctantly accept to use them. Even the designers of these systems are aware of it and issue warnings about their use, like the warning about the use of priorities in Java [3].[1]

[1] ... use thread priority only to affect scheduling policy for efficiency purposes; do not rely on it for algorithm correctness.

2.3 Limitations of the approach

Thus, this very simple execution scheme should be used as much as possible. Yet, there are situations where this scheme is not sufficient and more involved ones must be used. We can list here some of these situations:

Multi-periodic systems. When the system under control has several degrees of freedom with different dynamics, the execution periods of the corresponding controls may be different too and, then, the single loop scheme is rather inefficient. In this case an organization of the software into periodic tasks (one task for each period) scheduled by a preemptive scheduler provides a much better utilization of the processor. In this case, checking the timing properties is somewhat harder than in the single-loop case, but there exist well established schedulability checking methods for addressing this issue [4].

Yet a problem remains when the different tasks need to communicate and exchange data. It is there where mixing tasks and semaphores raises difficult problems. We shall see in Section 5.1, some ways of addressing this problem in a way which is consistent with the synchronous programming philosophy.

Discrete-event and mixed systems. In many control applications discrete events are tightly combined with continuous control. A popular technique consists of sampling these discrete events in the same way as continuous control.[2] When such sampling is performed, software can be structured as described previously in the periodic and multi-periodic cases.

There are still more complex cases when some very urgent tasks are triggered by some non-periodic event. In these cases more complex scheduling methods need to be used as for instance deadline monotonic scheduling [5].

In every case but the single period one, the problem of communication remains and has to be addressed, see Section 5.1.

Distributed systems. Finally, most control systems are distributed, for several reasons, for instance, sensor and actuator location, computing power, redundancy linked with fault tolerance, etc. Clearly, most of the computing structures considered above for single computers get more complicated when one moves

[2] though we can remark that there does not exist a sampling theory of discrete event systems as well established as the one which exists for continuous control.

toward distribution. We shall present in Section 5.2.1 a solution to this problem for the case of a synchronous distributed execution platform.

3 To theory

3.1 Milner's synchronous calculus of communicating systems

In the late seventies, computer theorists were thinking of generalizing usual formalisms like the λ-calculus and language theory so as to encompass concurrency which was urgently needed in view of the ever growing interest in reactive systems. In this setting, C.A.R. Hoare proposed *rendez-vous* as the structuring concept for both communication and synchronization between concurrent computing activities [6]. Slightly later, R. Milner used this concept to build a general algebraic concurrent framework called *Calculus of Communicating Systems* generalizing both λ-calculus and the calculus of regular expressions of language theory [7]. This approach was very successful and yielded valuable by-products like the discovery of the bisimulation concept.

But it soon appeared that this framework was not expressive enough in that it did not allow the expression of such a current object of reactive systems as a simple *watch-dog*. Thus, Milner went back to work and invented the *Synchronous Calculus of Communicating Systems* [8] which could overcome this drawback. These two calculi are based on a different interpretation of the parallel construct, *i.e.,* of what takes place when two computing agents operate concurrently. This difference is shown in Figure 1.1: while in the asynchronous version of the calculus, at each time step, one agent is chosen non deterministically to perform a step, in the synchronous case, each agent performs a step at each time. Milner also showed that SCCS can simulate CCS and thus that CCS is a *sub-theory* of SCCS. This, in our opinion provides some theoretical support to control practices: this means that synchronous primitives are stronger than and encompass asynchronous ones.

3.2 Synchrony as a programming paradigm

To this landscape G. Berry and the Esterel team added several interesting remarks [9]:

- First they remarked that the synchronous product is *deterministic* and yields less states than the

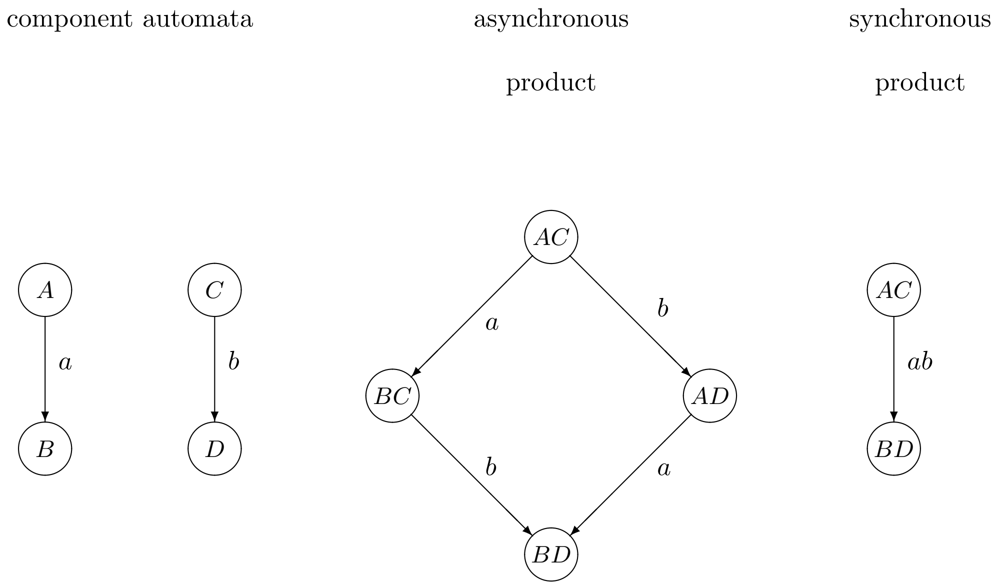


Figure 1.1: Asynchronous and synchronous products of automata

asynchronous one. This has two valuable consequences: programs are more deterministic and thus yield easier debugging and test and have less state explosion and thus yield easier formal verification.

- They also remarked that a step in the synchronous parallel composition corresponds exactly to one step of each of the component computing agents. This provides a "natural" notion of *logical time* shared by all components which allows an easier and clearer reasoning about time. In this setting, several activities can take place *simultaneously, i.e., at the same logical time instant*. This in turn, gave birth to the celebrated *infinitely fast machine* and *null execution time* mottos.

- Finally, they remarked that simultaneity provided for an important communication scheme, *instant broadcast*: in the same time step, an agent can emit a message and several other agents can receive it.

3.3 The causality problem in synchronous composition

These features were appealing. But they also yielded many problems because they allowed so-called *instant dialogs*: an agent can, at the same step, emit and receive messages and this can give rise to paradoxical situations when some events (receptions, emissions) are conditioned to some other events. Examples of these paradoxes are:

- an event occurs iff it does not (there is no solution),
- an event occurs iff it occurs (there can be two solutions: either the event occurs or it does not).

This was a general problem of synchronous parallel composition, the *causality problem.* This problem arose in many frameworks even for those which did not recognize themselves in the synchronous programming school, *i.e.,* each time concurrent communicating activities take place synchronously. Thus, in any of these frameworks, solutions had to be found and, indeed, many of them had been proposed. Let us list some of them:

Impose unit delay constraints on instant dialogs. This is the most conservative solution as it suppresses the problem. This solution has been adopted in Lustre [10], SL [11], and several execution modes of discrete-time Simulink.[3]

Rely on an arbitrary ordering of actions within a time step. This solution has often been taken in simulation based frameworks. This is the case of some Simulink simulation modes and, most notably, of Stateflow. The advantage comes from the fact that no restrictions need to be imposed on instant dialogs. The main drawback is that it provides an obscure semantic in which the behavior may depend on strange features such as, for instance, the lexicographic order of block names or the graphical position of blocks in a drawing [12].

Order activities within a time step according to a small-step semantic. This is the approach followed by the STATEMATE semantics of Statecharts [13].[4] The difference with the previous approach is that it does not garantee the uniqueness of solutions.

Reject non-unique solutions. This is the approach taken in Signal [14] and Argos [15]. At each instant, the different possible emission and receptions patterns are gathered and the design is rejected if there is no solution or if there are more than one solution. The drawback of this proposal is that it requires solving, at each time step, an NP-complete problem akin to solving boolean equations.

[3] http://www.mathworks.com/products/simulink/

[4] Note that many semantics have been proposed for Statecharts, showing the intrinsic difficulty of the problem.

Reject non-unique constructive solutions. This is the approach taken by the Esterel team. It is based on a deep criticism of the previous solution which, according to G. Berry, does not allow designers to understand and get insight on why their design is accepted or not: a designer does not solve boolean equations in his head when designing a system [16]. Moreover, constructiveness is easier to solve and is also consistent with the rules of electric stabilization in hardware design.

As we can see, there are many possible solutions to this problem and it is not yet clear whether a satisfactory one has already emerged.

4 Some languages and compilers

Although they are based on the same principles, different synchronous languages propose different programming styles:

- dataflow languages are inspired by popular formalisms like block diagrams and sequential circuits. Programs are described as networks of operating nodes connected by wires. The actual syntax can be textual, like in Lustre [10] or Signal [14, 17], or graphical like in the industrial tool SCADE [18, 19]. Lustre and SCADE are purely functional, and provide a simple notion of clock, while Signal is relational and provides a more sophisticated clock calculus;

- imperative languages are those where the program structure reflects the control flow. Esterel [9] is such a language, providing control structures resembling the ones of classical imperative languages (sequence, loops, etc.) but also interrupts and concurrency. In graphical languages, the control structure is described with finite automata. Argos [15] and SynchCharts [20], both inspired by Statecharts, are such languages, where programs consist of hierarchical, concurrent Mealy machines.

This section gives a flavor of two languages: Esterel as an example of an imperative language, and Lustre as an example of a data-flow language. It also presents some classical solutions for the causality problem, and the basic scheme for code generation.

4.1 Esterel

This section presents a flavor of Esterel, a more detailed introduction can be found in [21]. We use an example adapted from that paper: a *speedometer*, which indefinitely computes and outputs the estimated speed of a moving object.

Signals. Inputs/outputs are defined by means of *signals*. At each logical time instant, a signal can be either *present* or *absent*. In the speedometer example, inputs are *pure signals*, that is, events occurring from time to time without carrying any value:

- `sec` is an event which occurs each second; it is typically provided by some real-time clock;
- `cm` is an event which occurs each time the considered object has moved by one centimeter; it is typically provided by some sensor.

According to the synchrony hypothesis, the program runs on a discrete clock defining a sequence of logical instants. This clock can be kept abstract: the synchronous hypothesis simply states that this base clock is "fast-enough" to capture any occurrence of the inputs `sec` and `cm`.

The output of the program is the estimated speed. It can be present or absent, but when present, a numerical information must be provided (the computed speed). It it thus a *valued signal*, holding (for instance) a floating point value.

The header of the program (a module in Esterel terminology) is then:

```
module SPEEDOMETER:
input sec, cm;        % pure signals
output speed: float;  % valued signal
```

Basic statements. Basic statements are related to signals. The presence of a signal in the current logical instant can be tested with the `present S then ... else ...` statement. More sophisticated, the `await S` statement is used to wait for the next logical instant when `S` occurs.

An output signal is present in the current logical instant if and only if it is broadcast by the emit statement. Pure signals are emitted using `emit S`, while valued signals are emitted using `emit S(val)`, where `val` is a value of the suitable type.

Variables, sequence and loops. The other statements provided by Esterel resemble the ones of a classical imperative language. A program can use imperative variables, *i.e.*, memory locations. In the example, we use a numerical variable to count the occurrences of `cm`.

Statements can be put in sequence with the semicolon operator. This sequence is *logically* instantaneous, which means that both parts are occurring in the same logical instant. For instance, `emit X; emit Y` means that `X` and `Y` are both present in the current instant, and thus, it is equivalent to `emit Y; emit X`. However, the sequence reflects a *causality* precedence as soon as imperative variables are concerned: `x := 1; x := 2` results in `x = 2`, which is indeed different form `x := 2; x := 1` (result `x = 1`).

The unbounded loop is the (infinite) repetition of the sequence: `loop P end`, where `P` is a statement, is equivalent to `P; P; P; ...`. The loop statement may introduce instantaneous loops where the logical instant never ends. For instance `loop emit X end` infinitely emits `X` in the same logical instant. Such programs are rejected by the compiler: each control path in a loop must pass through a statement which "takes" logical time, typically an `await` statement.

Interrupts. Suppose for the time being that the event `cm` is more frequent than `sec`. Then, we can approximate the speed by the number of `cm` received between two occurrences of `sec`. As a consequence, the result will not be accurate if the speed is too low (less than one centimeter per second).

The normal behavior consists in waiting for a `cm`, then incrementing the counter and so on. This behavior is interrupted when a second occurs: the output speed is then emitted with the current value of the counter, and then, the whole process repeats (see Figure 1.2).

Parallel composition. As said before, the result of speedometer is not accurate if the speed is too low. In this case, an approximation is the inverse of the number of `sec` between two occurrences of `cm`.

In a new version of the program, two processes are running concurrently. One is based on a centimeter counter and is suitable for speeds > 1. The other is based on a `sec` counter and is suitable for speeds < 1. An actual null speed results in not emitting `speed` at all. This example illustrates a problem due to synchrony, since two concurrent processes are supposed to produce the same output (`speed`). In Esterel this problem is solved as follows: a signal is present if it is emitted by at least one concurrent process, and, in

```
module SPEEDOMETER:
input sec, cm;           % pure signals
output speed : double;   % valued signal
loop                        % Infinite behavior
  var cpt := 0 : double in  % initialize an imperative variable
    abort                   % Aborts the normal behavior ...
      loop                     % repeat
         await cm ;            % wait a centimeter,
         cpt := cpt + 1.0      % increment the counter
      end loop
    when sec do             % ... on the next occurrence of second, then
      emit speed(cpt)          % emit speed with the current value of cpt
    end abort
  end var
end loop.                 % and so on ...
```

Figure 1.2: First version of the speedometer in Esterel.

case of a valued signal of type τ, the several values emitted at the same logical instant are combined with an associative operator $\Gamma : \tau \times \tau \mapsto \tau$. The user has to specify this operator when declaring the signal, otherwise concurrent emission will be forbidden by the compiler.

This standard solution can be used in our example, since we know that, whenever both `sec` and `cm` are present, at most one counter is relevant (≥ 0), while the other is equal to 0. As a consequence, the values emitted on the signal `speed` can be safely combined with $+$ (see Figure 1.3). However, this solution is somewhat ad-hoc, and one may prefer to program a suitable "combinator" process, running concurrently with the other processes.

4.2 Lustre

Lustre programs as dataflow networks. A Lustre program denotes a network of operators connected by wires. Figure 1.4 shows both the graphical view and the textual (Lustre) version: `x` and `y` are input wires, that is, free variables. Other variables correspond to wires connected to some source. In Lustre, they are either output variables (`m`) or "hidden" local variables (`c`). Each linked variable must be defined by a unique equation. Note that is not necessary to name every wire: in the example, the local variable `c` can be avoided by simply writing `m = if x >= y then x else y`. Moreover, the order of the equations is not relevant; a Lustre equation is not a assignment: it expresses a global invariant (e.g. the value of `c` is always the sum of the values of `x` and `y`.

```
module SPEEDOMETER:
input sec, cm;
output speed : combine double with +;
[
loop
   var cpt := 0.0 : double in
      abort
         loop await cm ; cpt := cpt + 1.0 end loop
      when sec do
         if (cpt > 0.0) then  % null speed is no longer relevant
            emit speed(cpt)
         end if
      end abort
   end var
end loop
||
loop
  var cpt := 0.0 : double in
    abort
      loop await sec ; cpt := cpt + 1.0 end loop
    when cm do
      if (cpt > 0.0) then  % null speed is no longer relevant
         emit speed(1.0/cpt)
      end if
    end abort
  end var
end loop
].
```

Figure 1.3: Second version of the speedometer in Esterel.

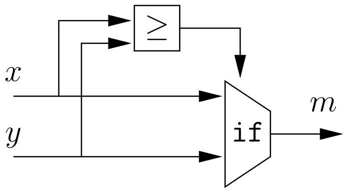
Semantics. In Lustre every variable denotes a synchronous *flow*, that is, a function from the discrete time ($\mathbf{N}$) to a declared domain of values. As a consequence, the semantics of the example above is obvious: the program takes two real flows `x` and `y`, and computes, step by step, the flow `m` such that:

$$m_t = \text{if } x_t \geq y_t \text{ then } x_t \text{ else } y_t \quad \forall t \in \mathbf{N}$$

Temporal operators. The Lustre language provides the basic data types `bool`, `int` and `real` and all the standard arithmetic and logic operators (`and`, `or`, `not`, `+`, `*`, etc.). Constants (e.g. `42`, `3.14`, `true`) are interpreted as constant functions over discrete time.

With this subset, one can write combinational programs, which are simply scalar functions extended point-wise on flows. In order to actually operate on flows, the language provides the `pre` operator:

$$(\texttt{pre X})\,(1) = \bot \qquad (\texttt{pre X})\,(t) = \texttt{X}(t-1)\;\forall t > 1$$



```
node max(x, y: real) returns (m:real);
var c : bool;
let
   m = if c then x else y;
   c = x >= y;
tel
```

Figure 1.4: A dataflow graph and the corresponding Lustre program.

This operator is similar to a non-initialized memory: its value is initially undefined (represented by $\perp$), and then, forever, it holds the previous value of its argument.

Some mechanism is necessary to properly initialize flows. This is achieved by the arrow operator:

$$(\texttt{X -> Y})\,(1) = \texttt{X}(1) \qquad (\texttt{X -> Y})\,(t) = \texttt{Y}(t)\ \forall t > 1$$

By combining pre and arrow, one can define recursive flows. For instance the alternating Boolean flow:

```
alt = false -> not pre alt
```

denotes false, true, false, true, $\cdots$; the counter:

```
i = 0 -> pre i + 1
```

denotes the sequence of integers $0, 1, 2, 3, \cdots$.

A simple example. In the dataflow paradigm, it is natural to represent events as infinite Boolean flows according to the equivalence present=true/absent=false. The program in Figure 1.5 counts the occurrences of its input `x` since the last occurrence of `reset`, or the beginning of time if `reset` has never occurred.

```
node counter(x, reset: bool) returns (c: int);
var lc : int;
let
   c = lc + (if x then 1 else 0);
   lc = if (true -> pre reset) then 0 else pre c;
tel
```

Figure 1.5: A simple counter with delayed reset in Lustre.

The speedometer in Lustre. Lustre is modular: user-defined nodes can be reused to program more and more complex systems. The syntax is functional-like, and the semantics is simple: instantiating a node is equivalent to inlining the definition of the node (provided local variables are renamed with fresh identifiers).

Note that this does not mean that inlining is necessary for code generation: under some restrictions the code generation can be modular and thus, reflect the structure of the source code (see Section4.4).

For instance, the node `counter` can be reused to program the speedometer. Just like in the Esterel program (Figure 1.3), we use two counters running concurrently. The results of the counters are used to compute the fast speed (`sp1`) and the slow speed (`sp2`). By construction, at most one of these speed is relevant (which means ≥ 0), as a consequence, the output can be obtained by computing the maximum value out of `sp1` and `sp2`.

```
node speedometer(sec, cm: bool) returns (speed: real);
var
   cpt1, cpt2 : int;
   sp1, sp2 : real;
let
   cpt1 = counter(cm, sec);
   sp1 = if sec then real(cpt1) else 0.0;
   cpt2 = counter(sec, cm);
   sp2 = if (cm and (cpt2 > 0)) then 1.0/(real(cpt2)) else 0.0;
   speed = max(sp1, sp2);
tel
```

Figure 1.6: A speedometer in Lustre.

4.3 Causality checking

Among classical static checks (type checking, references etc.), the compilation of synchronous languages requires to check causality. As explained in Section 3.3, the logical simultaneity of causes and consequences may introduce logical loops. This is the case:

- in Esterel, when the status of a signal depends on itself, e.g. `present X then emit X`;
- in Lustre, when the definition of a flow is instantaneously recursive, e.g. `X = X or Y`.

Whatever the language, the problem is in fact similar: instantaneous dependencies may contain loops that must be checked in some manner. We present here three solutions adopted in actual compilers.

Syntactic reject. This is somehow the most strict approach: it consists in statically rejecting any program containing a recursion in the instantaneous dependencies relation. This solution is adopted in Lustre (and

SCADE), and the criterion is purely syntactic. For instance, the following program is rejected, even if it is impossible to have a causality loop at run-time (if condition C is true then X depends on Y, if C is false then Y depends on X):

```
X = if C then f(Y) else T;

Y = if C then U else g(X);
```

This choice in Lustre is purely pragmatic: it is widely accepted that combinational loops in dataflow descriptions are error-prone and should be avoided, even in the case where the loop can actually be solved by some reasoning.

Boolean causality. This solution is the opposite of the previous one, in the sense that it aims at accepting any "safe" program, regardless to syntactic loops. Intuitively, this analysis accepts any program that can be proved to be both *reactive* and *deterministic*, i.e., to have exactly one possible reaction (pair of next state and outputs), for each set of inputs, no matter what the current state is.

More precisely, instantaneous dependencies are represented by a system of logical equations of the form $V = F(V, I)$, where V are the local and output variables and I are the inputs. The system is considered as correct if, for any valuation of I, the system admits a unique solution for V. For instance, supposing that $V = \{x, y\}$, $I = \{c\}$ and f, g are Boolean functions, the system:

$$\begin{array}{rcl} x & = & c \wedge f(y) \\ y & = & \neg c \wedge g(x) \end{array}$$

admits a unique solution for both $c = 1$ $(x = f(0), y = 0)$ and $c = 0$ $(x = 0, y = g(0))$, and thus, it is accepted.

This solution rejects programs that clearly make no sense. For instance, consider an Esterel program where the only emission of `X` appears in the statement `present X else emit X`. Instantaneous dependencies gives the equations $X = \neg X$, which has no logical solution; the program is not *reactive* and thus, it is rejected.

Consider now another example where the unique emission of `X` appears in "`present [X or A] then emit X end`". The underlying logical equation is then $X = X \vee A$, which has a unique solution if $A = 1$, but two solutions when $A = 0$. The corresponding program is then rejected as *non-deterministic*.

Boolean causality has been adopted in Argos and in early versions of the Esterel compiler. However, it suffers from several drawbacks:

- It accepts "dubious" programs. For instance, consider the system $(X = X,\ \ Y = X \wedge \neg Y)$; this system seems to have all the bad characteristics: X is not constrained, while Y depends on its own negation. However, the system admits a unique solution $X = Y = 0$, and thus, the corresponding program is accepted. Note that this system, if implemented as a combinational circuit in the straightforward way, yields an *unstable* circuit, that is, a circuit with races.
- Boolean causality requires satisfiability checking, which is NP-complete. As a consequence, the static check is likely to become intractable as the number of variables grows.

Constructive causality. This solution has been proposed [22, 23] to circumvent the problems of Boolean causality, which means: to reject the doubtful programs, and to allow a (much) more efficient static check. The idea is to solve recursive equations of the form $V = F(I, V)$ according to constructive (or *intuitionistic*) logic rather than classical Boolean logic.

Roughly speaking, constructive logic is classical logic without the *tertium non datur* principle (i.e. without the axiom $\forall x,\ x \vee \neg x$). In constructive logic, the system $(X = X,\ \ Y = X \wedge \neg Y)$ has no solution, but $x = c \wedge f(y),\ \ y = \neg c \wedge g(x)$ still admits a unique solution (as a function of c): $x = c \wedge f(0),\ \ y = \neg c \wedge g(0)$.

Concretely, constructive logic behaves as a propagation of facts in the four-values lattice $\{\top, 0, 1, \bot\}$, where $\top$ is the erroneous value (both true and false), and $\bot$ the unknown value (either true or false). If, during the evaluation, some variable becomes $\top$, the system has no solution. If the evaluation stops while a variable is still $\bot$, the system has several solutions.

Solving closed systems in constructive logics is similar to partial evaluation, and thus, has a polynomial cost. The complexity still grows exponentially by adding free variables such as inputs (the system must be solved for any valuation of free variables). However, efficient symbolic algorithms exist [23], and, in practice, constructive causality is much more efficient than Boolean causality.

We have shown that constructive causality rejects programs that are obviously wrong, or at least dubious. But what are exactly the right programs ? and, does the method accept them all ? G. Berry [22] pointed out a natural argument in favor of constructive causality, by stating that it reflects what actually happens in a digital circuit: recursive systems that are constructively correct are those which, when mapped to hardware, give circuits that eventually stabilize.

4.4 Code generation

The importance of this topic is due to the very peculiar position of synchronous languages, somewhere between modeling tools and programming languages. According to which definition is chosen, one would address the topic as "compilation" or as "program synthesis". The truth is somewhere in-between: code generation for synchronous languages is in general harder than usual compilation but still easier than program synthesis. It should be noted that code generation is still met with resistance: in many cases, even if people use synchronous modeling tools (e.g., Simulink/Stateflow), they still prefer to manually recode the models, mainly for efficiency reasons, related either to performance or code length or memory width. This situation reminds of the old times of assembly versus high level languages. There is little doubt that, sooner or later, code generation from high level models will become mainstream. This is why this question is addressed here.

We only consider here the basic problem, which is the generation of purely sequential code, suitable for a single-task, mono-processor implementation, as shown in Table 1.1.

Synchronous compilers do not actually produce the full program: they identify the necessary memory (and its initial value), and produce a procedure implementing a single step of execution (the `compute outputs and states` in Table 1.1). In other terms, compilers only provide the functionality of the system, and some main loop program should be written to define the actual execution rate (e.g., event-driven or periodically-sampled, depending on the application).

4.4.1 From data-flow to sequential code

Consider the example of the counter (Figure 1.5). Obtaining sequential code from the set of Lustre equations is rather simple. It requires: (1) to introduce variables for implementing the `pre` operators (in the example `pre_reset`, `pre_c`); (2) to sort equations in order to respect data-dependencies. Note that a suitable order exists as soon as the program has been accepted by the causality checking (in the example, `lc` must be

```
bool pre_reset = true;
int pre_c = 0, c; // c: output
bool x, reset;    // inputs

void counter_step() {
   int lc;
   lc = (pre_reset)? 0 : pre_c;
   c = lc + (x)? 1 : 0;
   pre_c = c;
   pre_reset = reset;
}
```

```
bool pre_reset = true;
int c;
bool x, reset;

void counter_step() {
   if (pre_reset) c = 0;
   if (x) c++;
   pre_reset = reset;
}
```

Figure 1.7: Simple and optimized C code for the Lustre counter.

computed before c).

Following those principles, the target code is a simple sequence of assignments, as shown in the left of Figure 1.7. The main advantage of this somehow naive algorithm is that it produces a code which is neither better nor worse than the source code: both the size and the execution time are linear with respect to the size of the source code. This one-to-one correspondence between source and target code is particularly appreciated in critical domain like avionics, and it has been adopted by the SCADE compiler.

However, some optimizations can be made in order:

- to minimize the number of buffers (in the example, pre_c is redundant),
- to speed-up execution by building a control structure.

An optimized code is shown in the right of Figure 1.7.

4.4.2 Explicit automaton

For imperative languages like Esterel, and even more for languages based on concurrent Mealy machines, it may appear a good idea to generate a code whose structure is an explicit automaton. The principle of automaton generation is illustrated in Figure 1.8: the program ABRO awaits in parallel one A and one B, then it emits Output, and waits for a Reset. For building the automaton, it is necessary to first identify the *halt-points*: halt-points are the statements where the logical time passes (the await statements in the example). A state of the automaton corresponds to a configuration of halt-points. Each transition is labeled by a condition on input signals ($\bar{A}$ means A is absent).

```
module ABRO;
inputs A, B, R;
outputs O;
loop
   [await A | await B];
   emit O;
   await R
end loop
```

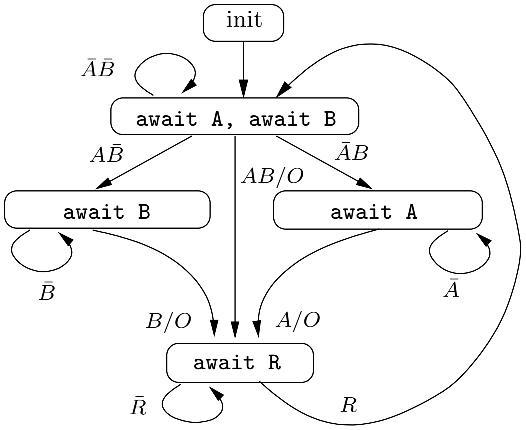


Figure 1.8: The ABRO system and its automaton.

```
enum AbroState = {Init, awaitAB, awaitA, awaitB, awaitR};
static AbroState state;
void AbroNext(){
  switch(state){
    case Init: state = awaitAB; break;
    case awaitAB:
     if (presentA()) {
        if (presentB()) { emitO(); state = awaitR; }
        else state = awaitB;
      } else if (presentB()) state = awaitA; }
    break;
    case awaitA:
      if (presentA()) { emitO(); state = awaitR; }
    break;
    case awaitB: if (presentB()) { emitO(); state = awaitR; } break;
    case awaitR: if (presentR()) { state = awaitAB; } break;
   }
}
```

Figure 1.9: A typical C code for the ABRO automaton.

Producing a "next" procedure from such an automaton is trivial. Figure 1.9 shows a possible C code: the procedure is a switch on the enumerated variable implementing the current state. In this example, inputs/outputs are performed using procedures that are not detailed.

Compilation into an explicit automaton produces code which is very efficient in terms of execution time: a typical call of the next procedure only requires a few tests and assignments. The drawback is indeed the size of the code: flattening a hierarchical, parallel composition of Mealy machines may result in code exponentially larger than the source program. As a consequence, this compilation scheme was soon abandoned for methods providing a more realistic compromise between code size and execution time.

```
next_Init = false
next_awaitA = Init or (awaitA and not A) or (awaitR and R)
next_awaitB = Init or (awaitB and not B) or (awaitR and R)
next_awaitR = not (next_awaitA or next_awaitB)
O = next_awaitR and not awaitR
```

Figure 1.10: Implicit automaton of ABRO, as a set of equations.

4.4.3 Implicit automaton

Several techniques for generating efficient and compact code for Esterel have been proposed [24, 25, 26, 27]. All these methods are far too sophisticated to be presented in a few lines. However, they all share the same characteristic, namely, that the state is encoded using several Boolean variables instead of a single enumerated variable. Note that an automaton with n states can be encoded using $log_2(n)$ Boolean variables. It is a bad idea to first generate the explicit automaton and then try to encode it with Booleans. On the other hand, a trivial encoding is obtained directly from the source program by taking one Boolean variable for each control point.

Consider the ABRO example. We first introduce four Boolean variables, one for each halt-point (Init, awaitA, awaitB, awaitR). Initially, all these variables are false except Init. Then, the next values of these variables are defined by a set of equations obtained by analyzing the program, as shown in Figure 1.10. Finally, we have obtained an implicit automaton which is equivalent to a sequential circuit, or, equivalently, a Lustre program.

One can note that, while the memory required is now clearly linear with respect to the source size, the number of required operations seems to have grown dramatically. Thus, some work remains in order to actually obtain concise and efficient code, for instance:

- identify common sub-expressions, in order to compute things once,
- embed the whole computation step into a local control structure to avoid useless computation,
- or even, try to build a new encoding giving a better compromise between the number of state variables and the size of code.

Figure 1.11 shows a possible optimized code for the ABRO system.

```
if (Init) {
   awaitA = awaitB = 1; Init = 0;
} else if (awaitR) {
   awaitA = awaitB = R;
   awaitR = !R;
} else {
   awaitA = (awaitA && !A);
   awaitB = (awaitB && !B);
   if (!awaitA and !awaitB)) {
      awaitR = 1; emitO();
   }
}
```

Figure 1.11: Optimized C code for ABRO.

In practice, efficient Esterel compilers [25, 28] are able to produce code whose size is reasonable (most of the times linear, quadratic in the worst case) and whose execution time is similar to what can be obtained by "hand-made" code.

5 Back to practice

We have seen several examples of synchronous languages. We have also seen how a *single-processor* and *single-task* (or, equivalently, *single-process* or *single-thread*) implementation can be automatically generated from a synchronous program. Such an implementation is simple: it consists of initialization code followed by a read-compute-write loop triggered by some external event (clock "tick" or other). Despite its advantages, discussed above, this "classical" implementation method also has limitations, as discussed in Section 2.3 above: it is not well-suited for multi-rate applications (multi-periodic or mixed time- and event-triggered) and it is not application to implementations on distributed execution platforms.

Multi-rate applications and distributed implementations are common in industrial practice. In order to deal with these cases, new methods have been developed which allow to implement synchronous programs under various software or hardware architectures, for instance, involving many tasks or many processors connected with a *bus*. We review some of these methods in the subsections that follow.

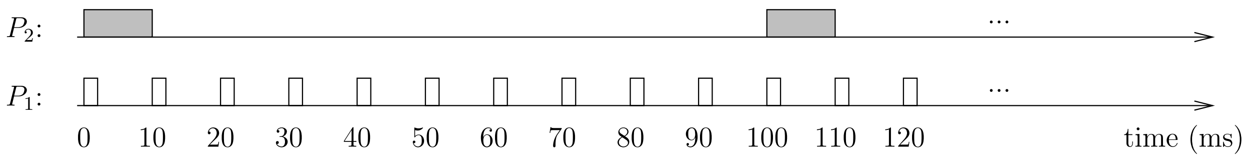


Figure 1.12: Two periodic tasks.

5.1 Single-processor, preemptive multi-tasking implementations

In this subsection we will explain how synchronous programs can be implemented on a single processor that is equipped with a *real-time operating system* (RTOS) employing some type of *preemptive scheduling* policy, such as *static-priority* or *earliest-deadline first* (EDF). In such a case, we can generate a *multi-task* instead of a single-task implementation. This means that there will be many tasks, each corresponding to a part of the synchronous program. The RTOS will schedule the tasks for execution on the processor.

To justify the interest behind multi-task implementations, let us provide a simple example. Consider a synchronous program consisting of two parts, or tasks, P_1 and P_2, that must be executed every 10 ms and every 100 ms, respectively. Suppose the worst-case execution time (WCET) of P_1 and P_2 is 2 ms and 10 ms, respectively, as shown in Figure 1.12. Then, generating a single task P that includes the code of both P_1 and P_2 would be problematic. P would have to be executed every 10 ms, since this is required by P_1. Inside P, P_2 would be executed only once every 10 times (e.g., using an internal counter modulo 10). Assuming that the WCET of P is the sum of the WCETs of P_1 and P_2 (note that this is not always the case), we find that the WCET of P is 12 ms, that is, greater than its period. In practice, this means that every ten times, the task P_1 will be delayed by 2 ms. This may appear harmless in this simple case, but the delays might be larger and much less predictable in more complicated cases.

Until recently, there has been no rigorous methodology for handling this problem. In the absence of such a methodology, industrial practice consists in “manually” modifying the synchronous design, for instance, by “splitting” tasks with large execution times, like task P_2 above. Clearly, this is not satisfactory as it is both tedious and error-prone.

When an RTOS is available, and the two tasks P_1 and P_2 do not communicate with each other (i.e., do not exchange data in the original synchronous design), there is an obvious solution: generate code for two *separate* tasks, and let the RTOS handle the scheduling of the two tasks. Depending on the scheduling policy used, some parameters need to be defined. For instance, if the RTOS uses a static-scheduling policy, as this is the case most of the times, then a priority must be assigned to each task prior to execution. During execution, the highest-priority task among the tasks that are ready to execute is chosen. In the case of *multi-periodic* tasks, as in the example above, the *rate-monotonic* assignment policy is known to be optimal in the sense of *schedulability* [4]. This policy consists in assigning the highest priority to the task with the highest rate (i.e., smallest period), the second highest priority to the task with the second highest rate, and so on.

This solution is simple and works correctly as long as the tasks do not communicate with each other. However, this is not a common case in practice. Typically, there will be data exchange between tasks of different periods. In such a case, some inter-task communication mechanism must be used. On a single processor, this mechanism usually involves some form of *buffering*, that is, shared memory accessed by one or more tasks. Different buffering mechanisms exist:

- simple ones, such as a buffer for each pair of writer/reader tasks, equipped with a *locking* mechanism to avoid corruption of data because of simultaneous reads and writes;
- same as above but also equipped with a more sophisticated protocol, such as a *priority inheritance* protocol to avoid the phenomenon of *priority inversion* [2], or a *lock-free* protocol to avoid blocking upon reads or writes [29, 30];
- other shared-memory schemes, like the *publish-subscribe* scheme used in the PATH project [31, 32], which allows *decoupling* of writers and readers.

None of these buffering schemes, however, guarantees preservation of the original synchronous semantics.[5] This means that the sequence of outputs produced by some task at the implementation may not be the same as the sequence of outputs produced by the same task at the original synchronous program. Small discrepancies between semantics and implementation can sometimes be tolerated, for instance, when the task

[5] Many of these schemes guarantee a *freshest-value* semantics, where the reader always gets the latest value produced by the writer. Freshness is desirable in some cases, in particular in control applications, where the more recent the data, the more accurate they are.

implements a *robust* controller which has built-in mechanisms to compensate for errors. In other applications, however, such discrepancies may result in totally wrong results, with catastrophic consequences. Having a method that guarantees equivalence of semantics and implementation is then crucial. It also implies that the effort spent in simulation or verification of the synchronous program need not be duplicated for the implementation. This is an extremely important cost factor.

To show why preservation of synchronous semantics is not generally guaranteed, consider the example shown in Figure 1.13. The timeline on the top of the figure shows the arrivals (releases) of three tasks: τ_i, τ_j and τ_q. Task τ_j (the *reader*) reads the output produced by τ_i (the *writer*). τ_q is a third task not communicating with the other two: its role will become clear in what follows. Task τ_i is released at times r^i_k and r^i_{k+1}, and produces outputs y^i_k, y^i_{k+1}, respectively. Task τ_j is released at time r^j_m and reads from τ_i a value x^j_m. According to the synchronous semantics, x^j_m must be equal to y^i_k, since all tasks execute in *zero time*.

The timeline on the bottom of Figure 1.13 shows a possible behavior of a simple implementation of the three tasks. We suppose that each task is executed as a separate process on a RTOS that uses static-priority preemptive scheduling. We suppose that task τ_q has higher priority than τ_i and τ_i has higher priority than τ_j. The execution periods of the different tasks are represented by the "boxes" shown in the figure. It can be seen that task τ_q, because of its higher priority, continues to execute when τ_j and τ_i are released. In that way, it "masks" the order of arrivals of τ_i and τ_j, which results in an inverse execution order: τ_i executes before τ_j, because it has higher priority. As a result, task τ_j reads the wrong output of τ_i: y^i_{k+1} instead of y^i_k.

In the rest of this subsection we will present a novel implementation method which permits to systematically build multi-task implementations of synchronous programs while preserving the synchronous semantics. This method has been developed in a number of recent works [33, 34, 35]. We only sketch the main ideas of the method here, and refer the reader to the above publications for details.

The method consists in a set of buffering schemes that mediate data between writer and reader tasks. Each of these schemes involves a set of *buffers* shared by the writer and the readers, a set of *pointers* pointing to these buffers and a *protocol* to manipulate buffers and pointers. The essential idea behind all protocols is

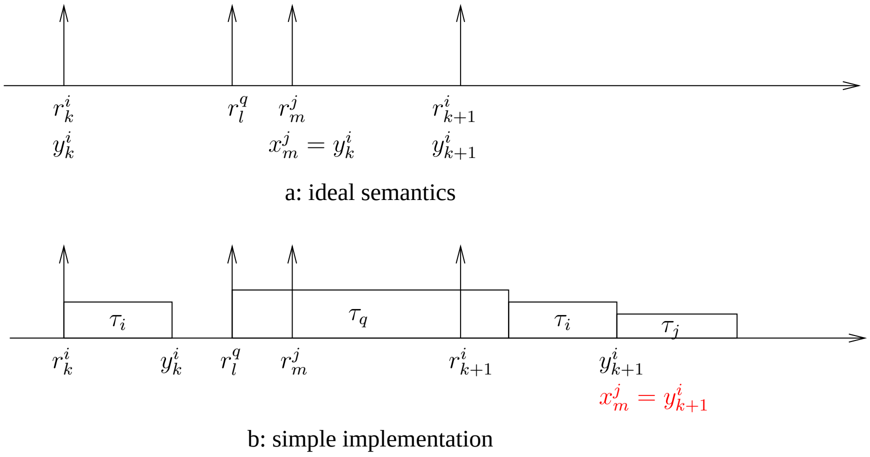


Figure 1.13: In the semantics, $x_m^j = y_k^i$, whereas in the implementation, $x_m^j = y_{k+1}^i$.

that *the pointers must be manipulated not during the execution of the tasks, but upon the arrival of the events triggering these tasks*. In that way, the order of arrivals can be "memorized" and the original semantics can be preserved.

Let us describe one of these protocols, called the *high-to-low buffering protocol*, and described in Figure 1.14. This protocol is used when the writer has higher priority than the reader (for simplicity we assume a single reader, but the protocol can be extended to any number of readers, with the addition of extra buffers, see [35]). In this protocol, the reader τ_j maintains a double buffer `B[0,1]`. The reader also maintains two boolean variables `current, next`. `current` points to the buffer currently being read by τ_j and `next` points to the buffer that the writer must use when it arrives next, in case it preempts the reader. The two buffers are initialized to a default value (which the reader reads if it is released before any release of the writer) and both bits are initialized to 0.

When the reader task is released, it copies `next` into `current`, and reads from `B[current]` during its execution. When the writer task is released, it checks whether `current` is equal to `next`. If they are equal, then a reader might still be reading from `B[current]`, therefore, the writer must write to the other buffer, in order not to corrupt this value. Thus, the writer toggles the `next` bit in this case. If `current` and `next` are not equal, then this means that one or more instances of the writer have been released before any reader was released, thus, the same buffer can be re-used. During execution, the writer writes to `B[next]`.

Setting: a high-priority writer task τ_i communicates data to a low-priority reader task τ_j.

Task τ_j maintains a double buffer `B[0,1]` and two boolean pointers `current, next`.

Initially, `current = next = 0` and `B[0] = B[1] =` some default value.

During execution:

- When τ_i is released: if `current = next`, then `next := not next`.
- While τ_i executes it writes to `B[next]`.
- When τ_j is released: `current := next`.
- While τ_j executes it reads from `B[current]`.

Figure 1.14: High-to-low buffering protocol.

Let us see how this protocol copes with the problem illustrated in Figure 1.13 above. Suppose that the release of the writer at time r_k^i is the first release. Then, since at that time we have `current = next = 0`, `next` will be toggled to `1`. Upon release of the reader at time r_m^j, `current` is set to `next`, that is, to `1` also. Upon the next release of the writer at time r_{k+1}^i, `next` is toggled to `0`. Thus, when the writer executes after the end of task τ_q, it writes to `B[next]=B[0]`. When the reader executes after the end of the writer, it reads from `B[current]=B[1]`, which holds the correct value.

5.2 Distributed implementations

In this subsection we will explain how synchronous programs can be implemented on distributed architectures, consisting of a number of computers connected via some type of network. Most embedded applications today involve such distributed architectures. For instance, high-end cars today may contain up to seventy electronic control units connected by many different busses. The implementation of a synchronous program consists of one or more tasks per computer, plus extra "glue code" that handles inter-task communication. In the rest of the section, we present two distributed implementation methods.

5.2.1 Implementation on the Time-Triggered Architecture

The Time Triggered Architecture [36] or TTA is a distributed, synchronous, fault-tolerant architecture. It includes a set of computers (or *nodes*) connected via a *synchronous bus*. The bus is synchronous in the sense

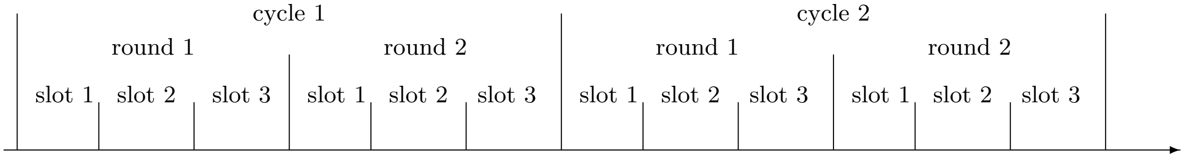


Figure 1.15: TTA bus schedule.

that it implements a clock-synchronization protocol. This implies a global time frame for all nodes. Access to the bus is done using a statically defined *time-triggered* schedule: each node is allowed to transmit during a specified time slot. The schedule repeats periodically as shown in Figure 1.15. The schedule ensures that no two nodes transmit at the same time (provided their clocks are synchronized, which is the responsibility of the TTA bus controller). Therefore, no on-line arbitration is required, contrary to the case of the CAN bus [37]. A set of tasks is executing at each TTA node. The latter is equipped with the *OSEKtime* operating system [38] which allows tasks to be also scheduled in a time-triggered manner.

Generating a TTA implementation from a synchronous program means the following:

- *Decomposing* the synchronous program into a set of tasks.
- *Assigning* each task to a TTA node where it is going to execute.
- *Scheduling* the tasks on each node and the messages on the TTA bus.
- *Generating* code for each task and glue code for the messages.

Notice that the first two steps are not particular to an implementation on TTA. Any distributed implementation of a synchronous program requires at least these two steps. The scheduling steps are specific to TTA. Implementations on other types of platforms will probably involve similar steps, however. For instance, instead of producing a time-triggered schedule of tasks and messages, priorities might need to be assigned to tasks (e.g., when the RTOS uses static-priority preemptive scheduling) or messages (e.g., when the bus is CAN).

In theory, the first three steps above can be formulated in terms of an optimization problem that can be solved automatically. In practice, the degrees of freedom (unknowns of the optimization problem) are far too many for the problem to be tractable (the problem is NP-hard). Thus, some choices must be made by the human user of the method. Fortunately, in practice the user has often a good intuition about many of these choices. For instance, the decomposition of the program into tasks and the assignment of tasks to processors is often dictated by the application and topology constraints (e.g., a task sampling a sensor must be executed on the computer directly connected to the sensor).

In [39], a method is proposed to generate TTA implementations from Lustre programs. In fact, the method uses a version of Lustre extended with annotations that allow the user to specify total or partial choices on the decomposition and assignment steps, as well as real-time assumptions (on worst-case execution times) and requirements (deadlines). The method uses a *branch-and-bound* algorithm to solve the scheduling problem. Decomposition is handled using the following strategy: start with a *coarse* decomposition; if this fails to produce a valid schedule then *refine* some of the tasks, that is, "split" them into more than one tasks. Which tasks should be refined is determined based on feedback from the scheduler and on heuristics (e.g., task with the largest WCET, task with longest blocking time, etc.). We refer the reader to the above paper for details.

5.2.2 Static scheduling of synchronous data-flow graphs on multi-processors

An older method for producing distributed implementations of synchronous designs is the one proposed in [40, 41]. This method concerns the *synchronous data flow* model (SDF). SDF can be viewed as a subclass of synchronous languages such as Lustre in the sense that only multi-periodic designs can be described in SDF. On the other hand, SDF descriptions are more "compact" and must generally be "unfolded" before being transformed into a synchronous program. Algorithms to perform this unfolding are provided in [40].

Before proceeding to review the implementation of SDF, let us give an example of an SDF graph and its translation into a synchronous language. The SDF graph is shown in Figure 1.16. It consists of two nodes A and B. The arrow from A to B denotes that A produces a sequence of *tokens* that are consumed by B. The number 2 in the source of the arrow denotes that A produces two tokens every time it is invoked. The number 1 at the destination of the arrow denotes that B consumes one token every time it is invoked. These numbers imply that, in a periodic execution of A and B, B must be executed twice as many times as A.

In Lustre, the above SDF graph could be described as follows:

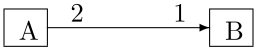


Figure 1.16: A simple SDF graph.

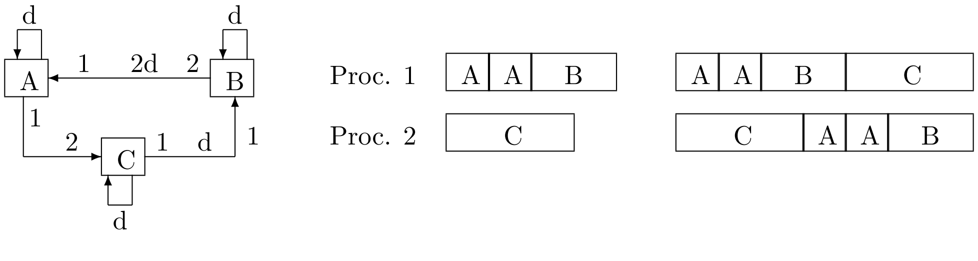
```
c = periodic_clock(0, 2);        /* clock 1/2 with phase 0 */

(a1,a2) = A(in when c);          /* A runs at 1/2 and produces two tokens */

(b1,b2) = current (a1,a2);       /* buffer made explicit */

out = B( if c then b1 else b2 );
```

The `periodic_clock(k,p)` macro constructs a boolean flow corresponding to a clock of period `p` and initial phase `k`.

Let us now turn to the implementation of SDF on multi-processor architectures proposed in [40]. This consists essentially in constructing a *periodic admissible parallel schedule*, or PAPS. The schedule is admissible in the sense that (1) node precedences are respected, that is, a node cannot execute before all required input tokens are available, (2) execution does not block, and (3) a finite amount of buffering is required to store tokens until they are consumed. The architecture considered in [40] consists of a number of *homogeneous* computers, in the sense that a node may be executed on any computer and it requires the same execution time on any computer. Communication delays between nodes are considered negligible. Finally, some synchronization mechanism is assumed available, which allows all processors to resynchronize at the end of a global period in order to restart the schedule.

Let us provide an example, taken from [40]. Figure 1.17 shows an SDF graph (a), and two PAPS (b) and (c). In the SDF graph there are three nodes A, B and C, with execution times 1, 2 and 3, respectively. The notation "d" means there is a *unit-delay* in the data-flow link (similarly, "2d" means there are two unit-delays). The schedules are for two processors.

The schedule shown in Figure 1.17(b) corresponds to a single unfolding of the SDF graph. Notice that A must be executed twice so that the two tokens needed by C are available. Also note that B may start before C finishes, since there is a unit-delay in the link from C to B. Finally, note that the period of this schedule cannot be less that 4, in other words, the rate of this implementation is $\frac{1}{4}$. The schedule shown in Figure 1.17(c) corresponds to two unfoldings of the SDF graph. This schedule achieves a better rate, namely,



(a) (b) (c)

Figure 1.17: An SDF graph and two periodic schedules on two processors (taken from [40]).

$\frac{2}{7}$. It also fully utilizes the two processors (whereas in the previous schedule Processor 2 remains idle for 1 time unit every 4 time units).

It should come as no surprise that constructing an optimal (i.e., rate maximizing) PAPS is a hard (combinatorial) problem. [40] proposes a heuristic algorithm, based on the so-called level-scheduling algorithm [42].

6 Conclusions and perspectives

Synchronous programming has been founded on the widely adopted practice of simple control-loop programs and on the synchrony hypothesis, which permits to abstract from implementation details by assuming that the program is sufficiently fast to "keep up" with its real-time environment. This spirit is in full accordance with modern thinking about system design, sometimes called model-based design, which advocates the use of high-level models with "ideal", implementation-independent semantics, and the separation of concerns between design and implementation. New methods that allow the high-level semantics to be preserved by different types of implementations on various execution platforms are constantly being developed and gaining ground in the industry.

A number of challenges remain for the future. To list only a few of them:

- There is currently no good way to describe execution platforms. Although the complete synthesis of semantics-preserving implementations starting from a high-level model (e.g., a synchronous program) plus a formal description of the execution platform seems too far-reaching today (because of the complexity involved), less ambitious goals such as reconfiguring the code when reconfiguring the platform ultimately require a well-defined description of execution platform or its variations.

- Regarding the preservation of semantics itself, many options are open. Preservation in the "strict" sense such as the one described in Section 5 may not always be necessary. For instance, a mixed discrete/continuous controller may require strict preservation of determinism in what concerns control-flow decision points and looser, freshest-value semantics in what concerns robust, continuous control. Finding ways to express the preservation requirements and methods to achieve them are challenging topics for future research.
- UML [43] is fastly moving toward establishing as a standard in embedded system design. Owing to the fact that synchronous programming is also a popular approach, this seems to call for a thorough comparison between these two approaches and, maybe attempts to make them converge.

References

[1] K. Aström and B. Wittenmark. *Computer Controlled Systems.* Prentice-Hall, 1984.

[2] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Computers*, September 1990.

[3] M. Campione, K. Walrath, and A. Huml. *The Java(TM) Tutorial: A Short Course on the Basics.* Sun Microsystems, 3rd edition, 2001.

[4] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[5] N. Audsley, A. Burns, M. Richardson, and A. Wellings. Hard Real-Time Scheduling: The Deadline Monotonic Approach. In *Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software*, Atalanta, 1991.

[6] C. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[7] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS.* Springer-Verlag, 1980.

[8] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25(3):267–310, 1983.

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

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

[11] F. Boussinot and R. de Simone. The SL synchronous language. *IEEE Transactions on Software Engineering*, 22(4):256–266, 1996.

[12] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi. Defining and translating a "safe" subset of Simulink/Stateflow into Lustre. In G. Buttazzo, editor, *4th International Conference on Embedded Software, EMSOFT04*. ACM, 2004.

[13] D.'Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

[14] P. Le Guernic, A. Benveniste, P. Bournai, and T. Gautier. Signal : a data flow oriented language for signal processing. *IEEE-ASSP*, 34(2):362–374, 1986.

[15] F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *Proc. of CONCUR'92*, volume 630 of *LNCS*. Springer Verlag, August 1992.

[16] G. Berry. *The foundations of Esterel*, pages 425–454. MIT Press, 2000.

[17] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with Signal. *Proc. of the IEEE*, 79(9):1321–1336, 1991.

[18] B. Dion and J. Gartner. Efficient development of embedded automotive software with IEC 61508 objectives using SCADE Drive. `http://www.esterel-technologies.com/files/Efficient-Development-of-Embedded-SW.pdf`.

[19] J. Camus and B. Dion. Efficient development of airborne software with SCADE Suite$^{\mathrm{TM}}$, 2003. `http://www.esterel-technologies.com/files/SCADE_DO-178B_2003.zip`.

[20] C. André. Representation and analysis of reactive behaviors: a synchronous approach. In *IEEE-SMC'96, Computational Engineering in Systems Applications*, Lille, France, July 1996.

[21] G. Berry. The Esterel v5 Language Primer. `ftp://ftp-sop.inria.fr/meije/esterel/papers/primer.pdf`.

[22] G. Berry. The constructive semantics of Esterel. Draft book available by ftp at `ftp://ftp-sop.inria.fr/meije/esterel/papers/constructiveness3.ps`, 1999.

[23] T. Shiple, G. Berry, and H. Touati. Constructive analysis of cyclic circuits. In *International Design and Testing Conference IDTC'96*, Paris, France, 1996.

[24] G. Berry. Hardware and software synthesis, optimization, and verification from Esterel programs. *Lecture Notes in Computer Science*, 1217, 1997.

[25] D. Weil, V. Bertin, E. Closse, M. Poize, P. Venier, and J. Pulou. Efficient compilation of Esterel for real-time embedded systems. In *CASES'2000*, San Jose, November 2000.

[26] S. Edwards. Compiling Esterel into sequential code. In *Proceedings of the 37th Conference on Design Automation (DAC-00)*, pages 322–327, NY, June 5–9 2000. ACM/IEEE.

[27] S. Edwards. An Esterel compiler for large control-dominated systems. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 21(2):169–183, 2002.

[28] CEC: The Columbia Esterel Compiler. `http://www1.cs.columbia.edu/ sedwards/cec/`.

[29] J. Chen and A. Burns. A three-slot asynchronous reader-writer mechanism for multiprocessor real-time systems. Technical Report YCS-286, Department of Computer Science, University of York, May 1997.

[30] H. Huang, P. Pillai, and K. Shin. Improving wait-free algorithms for interprocess communication in embedded real-time systems. In *USENIX'02*, 2002.

[31] A. Puri and P. Varaiya. Driving safely in smart cars. In *IEEE American Control Conference*, 1995.

[32] S. Tripakis. Description and schedulability analysis of the software architecture of an automated vehicle control system. In *Embedded Software (EMSOFT'02)*, volume 2491 of *LNCS*. Springer, 2002.

[33] N. Scaife and P. Caspi. Integrating model-based design and preemptive scheduling in mixed time- and event-triggered systems. In *Euromicro conference on Real-Time Systems (ECRTS'04)*, 2004.

[34] S. Tripakis, C. Sofronis, N. Scaife, and P. Caspi. Semantics-preserving and memory-efficient implementation of inter-task communication on static-priority or EDF schedulers. In *5th ACM International Conference on Embedded Software (EMSOFT'05)*, pages 353 – 360, 2005.

[35] C. Sofronis, S. Tripakis, and P. Caspi. A Dynamic Buffering Protocol for Preservation of Synchronous Semantics under Preemptive Scheduling. Technical Report TR-2006-2, Verimag Technical Report, 2006.

[36] H. Kopetz. *Real-Time Systems Design Principles for Distributed Embedded Applications.* Kluwer, 1997.

[37] ISO. Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling. Technical Report ISO 11898, 2003.

[38] OSEK/VDX. Time-Triggered Operating System – Version 1.0, 2001.

[39] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From Simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications. In *Languages, Compilers, and Tools for Embedded Systems (LCTES'03).* ACM, 2003.

[40] E. Lee and D. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, 36(1):24–35, January 1987.

[41] E. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[42] T. Hu. Parallel sequencing and assembly line problems. *Operations Research*, 9(6), 1961.

[43] Open Management Group. Response to the OMG RFP for schedulability, performance, and time revised submission. OMG Document number: ad/2001-06-14, June 18, 2001.