The Foundations of Esterel

Gérard Berry
Ecole des Mines de Paris and INRIA
2004 Route des Lucioles
06904 Sophia-Antipolis CDX

berry@cma.inria.fr
http://www.inria.fr/meije/personnel/Gerard.Berry.html
http://www.inria.fr/meije/esterel

1 Introduction

This paper informally presents the theoretical and practical foundations of synchronous programming of reactive systems, mostly focusing on the author's Esterel language. Synchronous languages are based on the perfectly synchronous concurrency model, in which concurrent processes are able to perform computation and exchange information in zero time, at least at a conceptual level. The synchronous model is well adapted to a very wide spectrum of computer applications, ranging from hardware circuit design to large-scale real-time process control, and including embedded systems, communication protocols, systems drivers, or user interfaces.

The synchronous model and languages are very different from models and languages well-known in the Computer Science community such as Petri Nets, CCS, CSP, or the π-calculus. Therefore, we find it useful to write a foundational paper explaining the application class, the model, the programming styles and languages based on it, their semantics, their implementation, and program verification. The development of synchronous languages was deeply influenced by the work of Robin Milner on process calculi and bisimulation. Since Robin Milner himself always expressed great interest in the subject, we find it natural to write that paper for a book dedicated to him. The paper is based on two invited lectures by the author: one at LICS'94, and the first Milner Lecture at Edinburgh University in 1996.

1.1 History

The perfectly synchronous model and languages appeared independently in the beginning of the 80's in different places. Esterel was defined by the author in

Sophia-Antipolis [11, 10]. Lustre was defined by P. Caspi and N. Halbwachs in Grenoble [27]. Signal was developed by A. Benveniste and P. Le Guernic in Rennes [24]. In Israel, D. Harel introduced the Statecharts quasi-synchronous graphical formalism [30]. In Grenoble, F. Maraninchi defined the Argos formalism [38] that makes (restricted) Statecharts drawings fully synchronous. More recently, in Nice, C. Andre extended Argos into the SyncCharts formalism [2] that has the same expressive power as Esterel. Synchronous programming was also introduced in the framework of concurrent constraint programming by V. Saraswat *et. al.* [46, 47]. See [26] for a joint presentation of Argos, Esterel, Lustre, and Signal.

R. Milner also introduced a form of synchrony primitive in his SCCS process calculus [40]; D. Austry and G. Boudol developed Milner's synchronous approach further in the Meije calculus [3]. The SCCS and Meije calculi are somewhat weaker than the aforementioned languages since they do not support negation, i.e. instantaneous test for signal absence. Nevertheless, they are useful to us for verification purposes.

The synchronous model and languages caught on quite easily in the automatic control community, where they did not fundamentally depart from models implicitly already in use in these areas. Esterel, Lustre, and Signal were actually designed and developed in mixed Control Theory and Computer Science teams[1]. The languages also entered the field of hardware design in the beginning of the 90's [5, 53], when it was realized that the synchronous model was identical to the zero-delay model of circuits[2]. Being somewhat unclassical compared to prevalent CSP or CCS based models, it took more time for the synchronous model to be accepted in the mainstream Computer Science community.

From the very beginning, the authors of synchronous languages developed or helped to develop software systems to support them and submitted them to industrial experimentation. The interest for synchronous languages in industry has grown steadily, and we think that their proper industrial career is about to start.

The development of synchronous languages has borrowed techniques from a number of usually disconnected fields. We already mentioned Control Theory. The semantics are given using Scott's fixpoint semantics and Plotkin's Structural Operational Semantics techniques [45]. The compilers are developed directly from the semantics, following the example of Robin Milner's ML language [42], itself in the line of Landin's viewpoint [35]. Automata theory techniques are used in the compilers [16, 12, 10]. Process calculi techniques such as bisimulation [41] or testing [31, 28] play a major role in program verification, as well as abstract

[1]The control-theory designers were Jean-Paul Rigault and Jean-Paul Marmorat for Esterel, Paul Caspi for Lustre, and Albert Benveniste for Signal.

[2]Thanks to Jean Vuillemin and Patrice Bertin at Digital Equipment Paris Research Laboratory; with them, the author also developed the 2z synchronous language based on 2-adic number theory [54], not presented here.

interpretation techniques [25]. Synchronous hardware design, optimization, and verification techniques based on logic simplification techniques or on Binary Decision Diagrams [21, 14, 15] are now of prominent use in implementation and verification. Finally, constructive logic techniques as well as asynchronous hardware analysis techniques [17] turned out to be fundamental for solving the particularly important semantical causality problem for Esterel [52].

1.2 Overview of the Paper

Section 2 presents the application area, namely, deterministic reactive systems. Section 3 presents an analysis of models of concurrent computing, insisting on the synchronous model and its adequacy for reactive systems programming. Section 4 presents the linguistic principles that underly synchronous languages, using the example of Esterel and Lustre. Section 5 presents the semantics and discusses the causality issues that are inherent in synchronous programming. In particular, we discuss the constructive semantics idea and its physical roots: the equivalence between propagation of electrical currents in circuits and proofs in constructive Boolean logic. Section 6 presents the techniques used to compile Esterel programs into automata, hardware circuits or conventional C programs, as well as optimization techniques. Finally, Section 7 discusses program verification.

2 Interactive and Reactive Systems

Instead of computing data outputs from data inputs, most modern computer-driven systems constantly interact with their environment and are themselves made of concurrent parts. Such systems fall into two distinct classes.

- In *interactive* systems, clients ask for accesses or resources that the system grants or allocates if and when possible. This class covers operating systems, data bases, networking, distributed algorithms, etc. The computer (network) is the leader of the interaction, and clients wait to be served. The main concerns are deadlock avoidance, fairness, and coherence of distributed information.

- In *reactive* or *reflex* systems, the computer rôle is to react to external stimuli by producing appropriate outputs in a timely way, the leader of the interaction being the environment. Reactive systems are prominent in industrial process control, airplane or automobile control, embedded systems, audio or video protocols, bus interfaces, systems or man-machine interfaces drivers, signal processing, etc. In reactive systems, the pace of the interaction is determined by the environment, not by the computers. Most often, clients cannot wait. The main concerns are correctness (safety) and timeliness.

The above terminology was introduced in [4] and we find it convenient to reuse it here, knowing of no better words. Of course, large scale systems can have components of both kinds. For instance, driving an airplane is mostly reactive, while communicating with the ground is mostly interactive. An automatic teller machine is reactive except for interactive communication with the bank.

Interactive and reactive systems deeply differ on the key issue of behavioral determinism. Interactive systems are naturally viewed as being non-deterministic. Being the master of the interaction, the system is allowed to make hidden internal choices about if and when requests are answered, and the answer to a sequence of inputs needs not be unique. On the other hand, behavioral determinism is a highly desirable and often mandatory property of slave reactive systems: the outputs of the system should be uniquely determined by its inputs and possibly by their timing. Think for example of airplane or automobile control, signal processing, or camera control.

Respecting either the non-deterministic or deterministic character of a system is mandatory for any formalism used to describe or program it. Since the behavior of a non-deterministic systems is far more complicated than that of a deterministic one (e.g., bugs may even be non-reproducible), the use of non-deterministic primitives should be reserved for interactive systems. In classical and well-studied concurrent formalisms such as Petri Nets or process calculi, non-determinism is built-in. This makes the formalisms well-suited to interactive systems and not well-suited to reactive ones. The synchronous languages we study here are concurrent and deterministic. This makes them well suited to reactive systems and inadequate for interactive ones. No formalism is yet able to encompass both characteristics in a smoothly unified way.

Most reactive systems involve two kinds of activities, data handling and control handling, with a rather varied balance between them. At one extreme, signal processing applications are mostly data-oriented: the data flow is quite complex but the control is often reduced to pipelining of operators. At the other extreme, a bus interface is control-intensive and manipulates data in a trivial way, filling and emptying buffers. Data-intensive and control-intensive applications call for different specification and programming techniques. As far as synchronous languages are concerned, Lustre and Signal are tailored to data-intensive applications while Esterel, Statecharts, and its descendants are tailored to control-intensive applications. Large applications can have both data-intensive parts and control-intensive parts. Unifying the corresponding styles at the programming language level is an active area of research.

3 Models of Concurrent Computations

To deal with reactive or interactive systems, our first task is to look for an adequate concurrency model. Here, we mean a naive model that one can explain to non-computer scientists, not a 26-tuple of sets and relations. Such a model should have four characteristics. First, it should be simple and intuitive. Second, it should be physically meaningful w.r.t. its application class. Third, it should be compositional, in the sense that a group of agents can be viewed as a single agent and a sequence of communications can be viewed as a single communication. Last, it should be mathematically powerful to serve as a basis for semantics and verification. In our view, there are three fundamental and radically different models that can be described by analogy with elementary physics:

- **The Chemical Model.** Computing agents are viewed as molecules floating in a soup which is stirred by a magical mechanism called Brownian motion. Communication (computation) can occur when two or more molecules enter in contact, and it results in the destruction of some old molecules and the generation of some new ones.
- **The Newtonian Model.** Computing agents are viewed as planets moving in space. In each instant, planets move in function of their current speed, their acceleration being determined by the positions and weights of all other planets. In terms of information, everything is as if each planet communicates its weight and position to every other planet in zero time.
- **The Vibration Model.** Computing agents are viewed as molecules organized in a crystal. When a molecule is kicked, it pushes its neighbors, which generates a wave traveling at some predefined speed (e.g., the speed of sound).

The three models obey our four requirements in different ways. The main difference is the time x it takes to establish a desired communication. Since sequencing communications sums up times, we can roughly express compositionality by the equation $x + x \sim x$ where $\sim$ is read as "homogeneous with". In the chemical model, x is *arbitrary*, and *arbitrary* + *arbitrary* $\sim$ *arbitrary* implies compositionality. In the Newtonian model, x is always 0, and $0+0=0$ holds trivially. In the vibration model, the communication time is fixed, or rather *bounded* if we allow for some non-determinism due to heat variations, and compositionality follows from *bounded* + *bounded* $\sim$ *bounded*. These are the three basic compositional models.

Chemistry is non-deterministic and asynchronous: there is no guarantee on the time it takes for two given molecules to interact, even if the interaction proper can be viewed as a synchronous act. The *Chemical Abstract Machine* or CHAM [9] is a mathematical version of chemistry that now routinely serves as a basis for the semantics of interactive process calculi or languages [43]. Being unable to express timeliness, the chemical model is obviously inappropriate for reactive systems.

In the Newtonian model, planets evolve in a deterministic and perfectly synchronous way. The Newtonian model will serve as a guideline for the definition and semantics of our synchronous languages, where we shall similarly assume

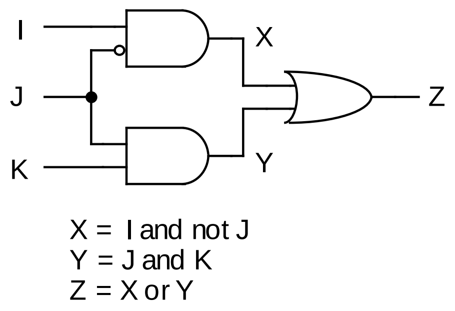

Figure 1: A synchronous circuit

that processes instantaneously exchange information in a deterministic way. For implementation, we shall use the more complex electrical vibration model, where information propagates with delay, where geometrical constraints may come in the picture, and where some (controllable) internal non-determinism may exist.

In physics, there is a well-known tension between the accuracy and the adequacy of a mathematical model. To compute planet or billiard ball trajectories, one can use either Einstein's generalized relativity theory or Newtonian mechanics. The former is more accurate but much harder, while the latter is less accurate but much easier and still adequate in most cases. The same happens in our field. The simplifying Newtonian assumption is adequate for programming, since it brings simplicity, determinism, and technology-independence at the language level. The more accurate vibration model that governs implementation is much harder, since actual response (propagation) times depend on implementation details and since a given system can be implemented in many different ways. To control how good the logical synchrony assumption is w.r.t. practical constraints, we have to estimate a bound on the actual reaction time for a given implementation. If the bound meets the specified timing constraints, we are happy. Otherwise, we either look for a better implementation or conclude that the intended system is infeasible with our technology. We would be satisfied to solve only 90% of the problems in this way and to leave the rest to more sophisticated or more manual techniques.

3.1 An Example: Synchronous Circuits

The simplest example of the joint use of Newtonian and vibration models is synchronous circuit design. An *acyclic combinational synchronous circuit* is pictured in Figure 1. In the *zero delay* (Newtonian) viewpoint, the circuit is viewed as a set of Boolean equations that has an acyclicity property: the equations can be ordered in such a way that any variable only depends on previously defined ones. Then, for any Boolean input, all the Boolean variables are uniquely defined by the equations. This view is used by the logic designer, who concentrates on the Boolean properties of the circuit, ignoring why and how the values are actually computed.

In the *electrical* vibration viewpoint, the circuit is viewed as an acyclic network of gates linked by wires. Boolean values are represented by voltages, say 0V and 5V, and wires and gates have bounded propagation delays. If the input voltages are kept stable to some Boolean voltages, then, after some predictable time, the output voltages stabilize at the right Boolean voltages. This is a physical fact, not a mathematical theorem. Since the number of input configurations is finite, it is possible to determine a maximum output stabilization time δ valid for all inputs. That view is taken by the electrical engineer, who does not care about what the circuit does and whose rôle is to minimize δ according to the current technology, using all possible tricks.

The technology-dependent value δ is the right and ideally the only interface between the logic designer the electrical engineer. Both know that waiting for δ time units ensures that the electrical circuit behaves as the zero-delay system of equations, which makes both of them happy.

An acyclic combinational circuit is memoryless. In *sequential circuits*, one adds elementary Boolean memories called *registers* to the combinational part to hold the state. A register is a delay element initialized to 0 and driven by a clock. The register input is an output of the combinational part, and the register output feeds back as an input to the combinational part. The output value of the register is initially 0, and then the value of its input at the previous clock tick. if δ is the stabilization time of the combinational part, a combinational reaction can be performed every δ time units. Changing the register output at clock tick consumes some additional time δ'. If the clock period is bigger than $\delta + \delta'$ time units, the sequential circuit adequately performs a series of reactions both in the logical and in the electrical model.

3.2 Pure Synchrony in Software

The software analogue of synchronous circuits is *cycle-based reaction*, a very common model in software process control. The implementation cyclically repeats a sequence of three actions: reading the inputs, computing the reaction, and producing the corresponding outputs. Input events occurring during a reaction are queued for the next reaction, which makes the reaction atomic and deterministic. The Newtonian and vibration viewpoints are as for circuits. In the Newtonian viewpoint, we neglect the reaction time and we consider a reaction to be instantaneous. In the vibration viewpoint, we measure the maximum reaction time δ for a given platform and check how good the Newtonian approximation is w.r.t. the actual problem to be solved.

Focusing on time yields of course a simplified picture. In practice, space is equally important and one must explore different time / space implementation tradeoffs. This will be discussed in Section 6. Pipelining and distributed implementation can also be necessary. They will not be discussed here.

4 Synchronous Styles and Languages

This section is devoted to synchronous programming styles. We start with the data-flow styles of Lustre and Signal. Then, we introduce the imperative style used in Esterel and in graphical formalisms à la Statecharts.

4.1 The Data-Flow Style

The data-flow style is well-adapted to steady process-control applications and to signal processing. Consider a dynamical system of equations of the form:

$$\begin{aligned} X_{t+1} &= U_{t+1} * sin(X_t + S_{t+1} - S_t) \\ S_{t+1} &= cos(S_t + U_{t+1}) \end{aligned}$$

where U is the input signal, X is the output signal, with $X_0 = 0$, and S is an internal state variable, with $S_0 = 1$. In such a system, there is already an implicit perfect synchrony assumption: the time taken by the arithmetical operations is 0. In Lustre [26, 27], the system is rewritten as follows:

```
node Control (U : float) returns (X : float);
var S : float;
let
   X = 0. -> (U*sin(pre(X)+S-pre(S)));
   S = 1. -> cos(pre(S)+U);
tel
```

or in an equivalent graphical form pictured in Figure 2.

The time indices are removed, and a variable such as X denotes the sequence or *flow* of values $\{X_t \mid t \in N\}$ where t is an integer discrete time index. Standard operators act synchronously: $X + Y = \{X_t + Y_t \mid t \in N\}$. The `pre` operator acts as an uninitialized delay: $\texttt{pre}(X)_{t+1} = X_t$ for $t > 0$ and $\texttt{pre}(X)_0 = nil$, where *nil* denotes uninitialization. Finally, the `->` operator provides flow initialization: $(X{-}{>}Y)_0 = X_0$ and $(X{-}{>}Y)_t = Y_t$ for $t > 0$.

Flows can be extracted from other flows using the `when` undersampling operator. If B is a boolean flow and X is a flow of type t, then "X `when` B" is a flow of type t that contains only the values of X whose indices t are such that $B_t =$ *true*, renumbered to form a proper flow. For instance, if $X = 0\ 1\ 2\ \ldots$ and $B =$ *true false true* $\ldots$, then $(X\ \texttt{when}\ B)_0 = 0$, $(X\ \texttt{when}\ B)_1 = 2$, $\ldots$. In "X `when` B", the Boolean flow B is called the *clock* of the result. The constant flow `true` acts as the master clock. Flows can be computed at different rates according to their clock. An important restriction is that only flows having

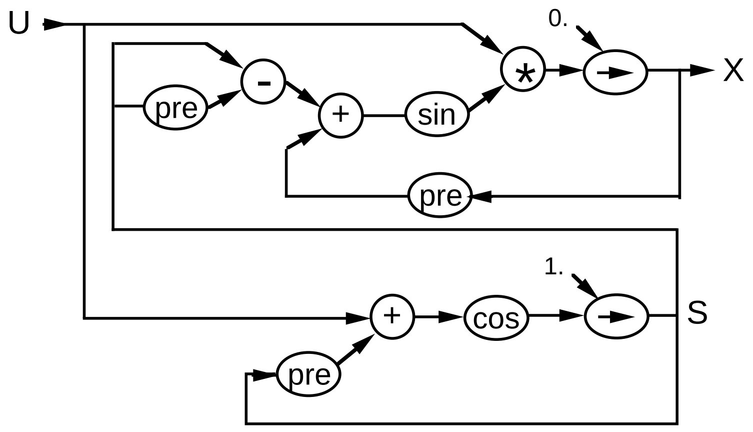

Figure 2: A graphical Lustre program

equal clocks can be combined by the operators. This ensures that any program can be computed with finite memory.

The Signal language [24] is similar, except that it also allows for *oversampling*, i.e. for creating flows that are faster than the inputs. Technically, Signal considers flow operators that define relations between flows instead of just functions in Lustre.

4.2 The Imperative Style

Consider now the following controller specification written in natural language:

> *Emit the output* `O` *as soon as both the inputs* `A` *and* `B` *have been received. Reset the behavior whenever the input* `R` *is received.*

As it stands, this simple specification is a little bit ambiguous. We additionally assume that nothing is to be done at initialization time and that the input signals can be simultaneous, as it is common in hardware. Furthermore, in the case where `R` occurs, the output should not be emitted and only the resetting should be performed.

A common way of making such a specification formal is to draw the picture of an automaton (also called a Mealy machine) as in Figure 3. The '.' operation in labels is the synchronous product of signals of SCCS [40] and Meije [3]. There are tools to analyze the behavior of such automata and to translate them into software programs or circuits. However, the direct specification of an automaton is not good programming style. In the automaton, the inputs and output names appear in many places, unlike in the specification. If we consider the same problem with n basic inputs `A`, `B`, `C`, ..., the automaton explodes exponentially.

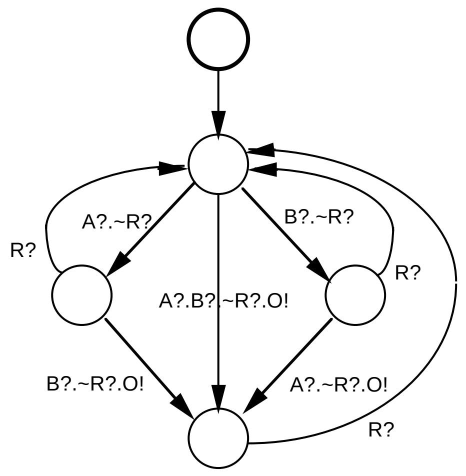

Figure 3: A Mealy machine

Even for automata of manageable size, a little change to the specification may incur a major change to the automaton, which often ends with a full rewriting. These facts are well-known in Language Theory, where regular expressions are usually preferred to automata pictures.

Synchronous imperative formalisms aim at providing modular ways of describing control-intensive reactive behaviors. The basic principle is to help the user to write things only once. Although it is not always made explicit, the *Write Things Once* or *WTO* principle is clearly the basis for loops, procedures, higher-order functions, object-oriented programming and inheritance, concurrency vs. choice between interleavings, etc. Reactive programming will call for even more structure. In Esterel, the controller is written as follows:

```
module ABRO:
input A, B, R;
output O;
loop
   [ await A || await B ];
   emit O
each R
end module
```

with only one occurrence of `A`, `B`, `R`, and `O`.

In each reaction, each signal has a unique presence / absence status. For input signals, the status is given by the environment. For other signals, the status is absent by default, and it is set present by executing an `emit` statement. The “`await A`” instruction waits for `A` and terminates when `A` occurs.

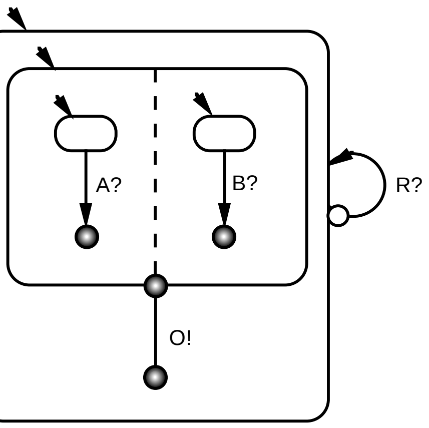

Figure 4: A chart for `ABRO`

A parallel combination of two statements terminates instantaneously as soon as both statements are terminated. The time needed for synchronization is conceptually 0 (Newtonian). Therefore, "`await A || await B`" terminates instantaneously as soon as both `A` and `B` have occurred. The sequencing operator "$p\,;\,q$" instantaneously transfers control to q when p terminates. Therefore, `O` is emitted as soon as both `A` and `B` have been received. The "`loop` p `each R`" operator is a *preemption* operator [6]. Its behavior is as follows: the body p is immediately started and it runs freely until the next instant where `R` occurs. At that instant, p is instantaneously killed, whatever its current state is, and p is immediately restarted afresh. In "`loop...each`", preemption is called *strong* because it has priority over body execution: at preemption time, the body is *not* executed in the instant. Therefore, if `A`, `B`, and `R` are simultaneously present, then `O` is not emitted, as requested by the specification. The behavior is exactly that of the automaton, but the writing is much better. *Write Things Once* is achieved using the cooperation of sequencing, concurrency, and preemption constructs, each of them being indispensable.

Synchrony expresses that the internal bookkeeping necessary to execute statements takes no time, i.e. that it should be performed entirely within an input-output cycle in an implementation (see Section 3.2). The only constructs that take time are the ones explicitly required to do so, here "`await`" and "`loop...each`". Notice that synchrony of all other constructs is necessary to obtain the required behavior with no spurious silent move.

A graphical program for `ABRO` is pictured in Figure 4 States are hierarchically decomposed. Sourceless arrows indicate initial states. Bullets indicate termination, and the `R` arrow has a circle to indicate strong preemption. The

SyncCharts formalism [2] is based on a similar graphical style and compiles into Esterel.

4.2.1 Nested Preemptions and Exceptions

In Esterel, the essence of programming consists of controlling the life and death of activities by using preemption structures. The nesting of preemption structures expresses preemption priority in a natural way. Here is the basic training of an athlete:

```
module Runner:
input Second, Meter, Lap;
output ...; % not given here
every Morning do
   abort
      loop
         abort RunSlowly when 15 Second;
         abort
            every Step do
               Jump || Breathe
            end every
         when 100 Meter;
         FullSpeed
      each Lap
   when 2 Lap
end every
end module
```

Here, the inputs are `Morning`, `Second`, `Step`, `Meter`, and `Lap`. The identifiers in italic represent statements not written here. In a lap, the full sequence is executed only if the lap is longer than `15 Second` plus `100 Meter`. If the lap is shorter than `15 Second`, one only runs slowly. If the lap is shorter than `15 Second` plus `100 Meter`, one never runs full speed. The same happens if mornings occurs very often.

Notice that any input can serve as a time unit in a preemption. In reactive programming, timing constraints should not be expressed only in seconds. When driving a car, if there is an obstacle at 30 meters, the timing constraint is "stop in less than 30 meters", no matter the time it takes to stop.

4.2.2 Exceptions

Esterel supports an exception mechanism that is fully compatible with concurrency. When the athlete is getting older, he should worry about his heart during the most strenuous part of a lap:

```
trap HeartAttack in
   abort
      loop
         abort RunSlowly when 15 Second;
         abort
            every Step do
              Jump || Breathe || CheckHeart
            end every
         when 100 Meter;
         FullSpeed
      each Lap
   when 2 Lap
handle HeartAttack do
   GoToHospital
end trap
```

In *CheckHeart*, one should execute an exception raising statement of the form "`exit HeartAttack`" if there is any problem with the heart, which can be detected using the aforementioned preemption constructs. Then, the concurrent processes *Jump* and *Breathe* are immediately preempted[3], and control immediately enters the *GoToHospital* statement.

4.2.3 Data Handling

Esterel programs can also manipulate data of arbitrary types. Here is a simple protocol:

```
module Sender :
input Message : Message; % from user
output Send : Message;   % to line
input Ack;               % from line
output Sent;             % to user
input Millisecond;       % from timer
loop
   await Message ;
   abort
      every 100 MilliSecond do
         emit Send(?Message)
      end every
   when Ack ;
   emit Sent
end loop
end module
```

[3]Technically, concurrent statements are *weakly preempted*, unlike with the `abort` statement: they are allowed to run for a last time in the reaction.

In addition to their presence / absence status which is as for pure signals, the signals `Message` and `Send` bear a value that belongs to an abstract type `Message`. In any reaction, a valued signal has exactly one value, which is determined by the environment for input signals and by the `emit` statements for other signals. The expression `?Message` yields the current value of `Message`. Esterel also supports non-shared variables that can be assigned to or passed to routines written in other languages.

By default, data-handling operations are assumed to be instantaneous, as in data-flow languages. However, Esterel also supports an `exec` primitive that makes it possible to call long external computations that do take time. This construct is very useful for computation tasks scheduling.

5 Semantics

The denotational semantics of data-flow languages is standard. Streams are modeled as infinite sequences, most often in Scott's classical cpo model, and sometimes using the p-adic metric $d(X, Y) = 2^{-n}$, where n is the least integer such that $X_n \neq Y_n$ [54]. Equations are solved using fixpoint techniques based either on the Knaster-Tarski or on the Banach fixpoint theorems. Acyclic programs have well-defined unique solutions.

The semantics of imperative languages is more difficult. We give some clues for Esterel, referring to [7, 6] for more details. The first step is to define a *kernel calculus* from which the other statements are derived by macro-expansion. The Esterel kernel contains primitives for terminating, pausing for the instant, and exiting a trap, respectively written 0, 1 and k with $k \geq 2$ (this numerical encoding follows an idea of Cousineau [22]). Signal emission is written $!s$, while $s \mathbin{?} p \mathbin{,} q$ instantaneously tests for the presence of a signal. Sequencing, looping, and synchronous concurrency are written $p \mathbin{;} q$, $p*$, and $p \,|\, q$. The preemption structures are suspension $s \supset p$, which freezes p for the instant if s is present, and trap declaration $\{p\}$. An auxiliary $\uparrow p$ operator is necessary for trap renumbering. Finally, local signals are declared using the classical hiding notation $p \backslash s$.

5.1 The Behavioral Semantics of Esterel

The primary semantics is the *behavioral semantics*. The reaction of a program P to an input event I is defined by a reaction $P \xrightarrow[I]{O} P'$ where O is the output event and P' is the derivative, i.e., the program that will perform the next reaction. The reaction is defined using an auxiliary inductive relation $p \xrightarrow[E]{E', k} p'$ where p is a statement, p' is its derivative, E is the context event that tells which signals are present, E' is the event made of the signals emitted by p in E, and k is a numerical completion code. Instantaneous signal broadcasting is obtained by imposing the invariant $E' \subset E$, which expresses that any statement hears what it is saying. Here are two of the rules:

$$\frac{p \xrightarrow[E]{E',0} p' \qquad q \xrightarrow[E]{F',l} q'}{p\,;q \xrightarrow[E]{E'\cup F',l} q'} \quad \textit{(seq2)}$$

$$\frac{p \xrightarrow[E]{E',k} p' \qquad q \xrightarrow[E]{F',l} q'}{p\,|\,q \xrightarrow[E]{E'\cup F',\,max(k,l)} p'\,|\,q'} \quad \textit{(parallel)}$$

Rule *(seq2)* is used for a sequence $p\,; q$ when p terminates in the instant, i.e., returns code 0. Then q is executed in the same rule premise, which models synchrony. Signal information flows from p to q because of the broadcasting invariant: one must have $E' \cup F' \subset E$, hence $E' \subset E$, which means that q receives the signals emitted by p. Rule *(parallel)* defines the semantics of concurrency. The statements p and q are executed simultaneously in the same context E, and each of them receives the signals emitted by the other because of the broadcasting invariant. Control synchronization between the branches is performed by returning the maximum of their completion codes; this is the essence of the numerical encoding, see [7, 6].

5.2 Cyclic Instantaneous Dependencies and Paradoxes

For reactive programs, the two basic requirements are reactivity, i.e., existence of reaction for all inputs, and determinism, i.e., uniqueness of the reaction. Not all Esterel programs are reactive and deterministic. With output `X`, the program "`present X else emit X`" is non-reactive since `X` should be present if and only if it is not emitted, which contradicts instantaneous broadcasting. The program "`present X then emit X`" is non-deterministic since `X` should be present if and only if it is emitted, which does not determine its status. Such paradoxical programs must be rejected at compile-time. An easy way to reject them is to forbid static self-dependency of signals, in the same way one usually requires circuits to be acyclic. The above programs indeed corresponds to the nonsensical cyclic circuits "`X = not X`" and "`X = X`". However, requiring acyclicity turns out to be inadequate to Esterel practice. Users do write cyclic but sensible programs such as the following one:

```
module  GoodCycle1 :
input I;
output X, Y;
present I then
   present X then emit Y end
else
   present Y then emit X end
end present
```

In GoodCycle1, X depends on Y and conversely, but it is immediately visible that any given status of I breaks the cycle. Assume for example I present. Then, the else branch of "present I" is not executed and X is not emitted. Therefore, X is absent, and Y is absent since it is not emitted either. The deduction is symmetrical if I is absent, with X and Y absent. Delays can also cut cycles:

```
module  GoodCycle2 :
output X, Y;
present X then emit Y end;
pause;
present Y then emit X end
```

In GoodCycle2, the dependency of X on Y is meaningful at first reaction only while the reverse dependency of Y on X is meaningful at second reaction only. The signals X and Y are both absent in any instant. As before, this is directly obvious on the source code.

Of course, these toy examples do not show why cycles are useful in practice. See [8] for the example of a naturally cyclic bus arbiter. In [36], it is shown that cyclic programs can be exponentially smaller than acyclic ones for the same behavior.

5.3 Logical Correctness and Further Paradoxes

An apparently simple way to deal with cycles is to require the programs to be reactive and deterministic, i.e., to have one and only one behavior for each input. This condition is also called *logical correctness*. It can be checked using BDD algorithms [29]. However, logical correctness also leads to somewhat paradoxical behavior. Consider the following Strange program:

```
module Strange:
output X, Y;
   present X then
      emit X
   end
||
   present [X and not Y] then
      emit Y
   end
```

An easy case inspection shows that there is only one logically consistent behavior, `X` and `Y` absent. That behavior is consistent since neither `X` nor `Y` is emitted. No other behavior is consistent. For instance, consider `X` present and `Y` absent. Then, "`emit Y`" is executed, which contradicts `Y` absent. Although `Strange` is logically correct, it is by no means understandable, which is more important to us. The reaction is not computed in a causal way.

The *constructive semantics* restricts the behavioral semantics by properly defining how information should causally propagate in programs, regardless of cycles. The foundations of the constructive semantics being much simpler to explain on circuits, let us first examine how Esterel programs are translated into Boolean circuits.

5.4 Translating Esterel Programs Into Circuits

The circuit semantics translates Esterel imperative programs into sequential circuits, see Section 3.1, or, equivalently, into Lustre data-flow programs. The basic idea is to associate a subcircuit with each statement, allocating gates and wires for control and signal propagation. Only the 1 or `pause` kernel unit-delay statement generates a register. All the other constructs only generate combinational logic. A first translation was presented in [5]. It rejected programs that can execute a given statement several times in different contexts in the same instant, which is possible (and useful) in Esterel. The translation has now been extended to cover that case as well, see [7].

If an Esterel program contains no cyclic instantaneous signal dependencies, then the circuit obtained by the translation has no combinational cycle. In that case, it is easy to see that both the Esterel program and the circuit are logically correct and that they have the same behavior. However, remember we also want to deal with dependency cycles in Esterel programs. Esterel cycles translate into combinational logic cycles, which implies that we also need to understand combinationally cyclic circuits. Consider first the circuit obtained by translating `GoodCycle1` above:

```
X = (not I) and Y
Y = I and X
```

The circuit is also logically correct: `I` $= 0$ implies `Y` $= 0$ and then `X` $= 0$, while `I` $= 1$ implies `X` $= 0$ and then `Y` $= 0$. Notice that the correctness of the compiled equational circuit is much less directly visible than that of the imperative Esterel program. This is why cycles are usually rejected in data-flow languages and digital circuit design. The (simplified) circuit for `GoodCycle2` is

```
R = reg(1)
X = (not R) and Y
Y = R and X
```

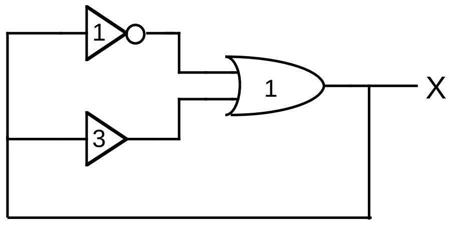

Figure 5: Delay assignment for unstable Hamlet

where `reg` is the register construct (`reg(X) = 0->pre(X)` in Lustre). In the first instant, $\texttt{R} = 0$ implies $\texttt{Y} = 0$ and then $\texttt{X} = 0$. In all further instants, $\texttt{R} = 1$ implies $\texttt{X} = 0$ and then $\texttt{Y} = 0$. The circuit has no behavioral problem.

Consider now the circuit for `Strange`:

```
X = X
Y = X and not Y
```

As for Esterel, the only Boolean solution is $\texttt{X} = 0$, $\texttt{Y} = 0$, but that solution seems to come from nowhere.

5.5 Electrical Propagation in Cyclic Circuits

Let us now switch from the Newtonian Boolean model to the electrical vibration model. What happens if we implement `GoodCycle1`, `GoodCycle2`, and `Strange` with gates and wires? It is easy to see that `GoodCycle1` and `GoodCycle2` stabilize in bounded time for any input, exactly as if they were acyclic. On the contrary, `Strange` does not stabilize, since the `X` wire is not driven by a gate. Things become clearer by considering the following simpler `Hamlet` circuit[4]:

```
X = X or not X
```

Obviously, the only Boolean solution is $\texttt{X} = 1$ according to the law of excluded middle. Unfortunately, electrons never heard of excluded middle, and the electrical circuit does not stabilize for all gate and wire delays. For example, consider the delay assignment of Figure 5, where the bottom triangle represents an identity gate with delay 3 and where wires are delay-free (we are not precise about the delay model for lack of room, see [51, 52] for details). Then, assuming that all wires have initial value 0, the value of `X` oscillates forever between 0 and 1.

[4]To understand the name, interpret `X` as "to be".

5.6 The Constructive Boolean Logic

For `Hamlet`, the Boolean solution $\mathtt{X} = 1$ is obtained by making a self-justifying guess and a proof by contradiction to reject $\mathtt{X} = 0$. Electrons are unable of performing such fancy speculative reasoning, which we must therefore reject to model circuits. The solution is to use *constructive logic*, in which all values must be computed by explicit proofs. The constructive Boolean logic for combinational circuits is very simple. It can be presented in three equivalent ways: as a proof calculus, as a term rewriting system, or as the Scott semantics of Boolean equations.

The proof calculus deals with sequents of the form $\mathcal{C}, \mathcal{I} \vdash e \rightarrow b$, where $\mathcal{C}$ is the circuit presented as a system of equations, $\mathcal{I}$ is an input function that defines a Boolean value 0 or 1 for each input variable, e is a Boolean expression written with the inputs and variables of $\mathcal{C}$, and b is a Boolean value. The sequent is read "for the circuit $\mathcal{C}$, with input values $\mathcal{I}$, the expression e evaluates to b". For expressions, we restrict ourselves to `not` and `or` operators; as usual, conjunction can be defined by x `and` $y =$ `not` (`not` x `or` `not` y). Let $\overline{0} = 1$ and $\overline{1} = 0$. The proof rules are as follows:

$$\frac{\mathcal{C}, \mathcal{I} \vdash e \rightarrow b}{\mathcal{C}, \mathcal{I} \vdash \mathtt{not}\ e \rightarrow \overline{b}} \quad (\textit{negation})$$

$$\frac{\mathcal{C}, \mathcal{I} \vdash e \rightarrow 1}{\mathcal{C}, \mathcal{I} \vdash e\ \mathtt{or}\ e' \rightarrow 1} \quad (\textit{left-or-1})$$

$$\frac{\mathcal{C}, \mathcal{I} \vdash e' \rightarrow 1}{\mathcal{C}, \mathcal{I} \vdash e\ \mathtt{or}\ e' \rightarrow 1} \quad (\textit{right-or-1})$$

$$\frac{\mathcal{C}, \mathcal{I} \vdash e \rightarrow 0 \qquad \mathcal{C}, \mathcal{I} \vdash e' \rightarrow 0}{\mathcal{C}, \mathcal{I} \vdash e\ \mathtt{or}\ e' \rightarrow 0} \quad (\textit{or-0})$$

$$\frac{\mathcal{I}(\mathtt{I}) = b}{\mathcal{C}, \mathcal{I} \vdash \mathtt{I} \rightarrow b} \quad (\textit{input})$$

$$\frac{\mathtt{X} = e \in \mathcal{C} \qquad \mathcal{C}, \mathcal{I} \vdash e \rightarrow b}{\mathcal{C}, \mathcal{I} \vdash \mathtt{X} \rightarrow b} \quad (\textit{variable})$$

A circuit $\mathcal{C}$ is said to be *constructive* for an input $\mathcal{I}$ if all variables can be evaluated to a Boolean value using the above rules. In this case, it is easy to see by induction on the length of the proof that the results form the unique solution of the Boolean system, establishing that constructiveness implies logical correctness and that the order of proof steps is immaterial.

It is easy to see that `GoodCycle1` and `GoodCycle2` are constructive for all inputs, while `Strange` and `Hamlet` are non-constructive. For `Hamlet`, there is no way to build a proof: a proof of $\mathtt{X} \rightarrow b$ must end by rule (*variable*), but no proof of "`X or not X`" can be constructed without first proving $\mathtt{X} \rightarrow b'$ for some b'.

In the term rewriting approach, the equations are oriented from right to left and the following constant-folding rules are added:

$$\begin{array}{c}\mathtt{not}\ 0 \rightarrow 1 \\ \mathtt{not}\ 1 \rightarrow 0 \\ 1\ \mathtt{or}\ x \rightarrow 1 \\ x\ \mathtt{or}\ 1 \rightarrow 1 \\ 0\ \mathtt{or}\ 0 \rightarrow 0\end{array}$$

A circuit $\mathcal{C}$ is constructive for an input if each variable in $\mathcal{C}$ can be rewritten into a Boolean value using the input value assignment $\mathcal{I}$, the oriented equations and the above rules.

In the Scott denotational semantics view, variables are interpreted over the Scott Boolean domain $B_\perp = \{\perp, 0, 1\}$ and the Boolean operators are interpreted as the least monotonic functions that satisfy the above equations. Notice that `or` is interpreted by Plotkin's *parallel or*, a function that cannot be defined in software programming languages [44]. Then, for each input $\mathcal{I}$, the circuit $\mathcal{C}$ defines a monotonic function $\mathcal{C}_\mathcal{I}$ from variable values to variable values, and the circuit is said to be constructive if the value of any variable in the least fixpoint of $\mathcal{C}_\mathcal{I}$ has no $\perp$-component.

It is easy to see that the three definitions of constructiveness coincide. The fact that constructiveness is a variant of Scott semantics immediately implies compositionality. The main full abstraction theorem shows that constructive logic exactly matches electrical current propagation:

Theorem 1 *Let $\mathcal{C}$ be a circuit and $\mathcal{I}$ be an input event. Then $\mathcal{C}$ with input $\mathcal{I}$ electrically stabilizes in bounded time for all gate and wire delays if and only if it is constructive for $\mathcal{I}$.*

In other words, a cyclic constructive circuit electrically stabilizes just as an acyclic one. It is natural to call constructive cyclic circuits combinational ones.

Notice that Theorem 1 is very much in the spirit of the Curry-Howard correspondence between computations and proofs [23]: an electrical computation performs a proof of a logical formula.

The proof of Theorem 1 is given by Shiple in [51]. For lack of room, we can only give a very brief proof sketch. The roots are in Brzozowski and Seger's analysis of asynchronous circuits [17]. Information propagation in the up-bounded inertial delay model is "asynchronous" because of gate and wire delays. Here, in the terminology of Section 3, asynchrony is vibrational rather than chemical since the delays are bounded from above. Given any circuit with delays, Brzozowski and Seger first show that, after a bounded time, only non-transient states of the circuit wires can be reached. Then, they present a technique called GWM

(Global Multiple Winner) analysis that makes it possible to directly compute the reachable non-transient states, using a state transition system semantics that abstracts away delays. Next, they show that a ternary analysis using Scott's Booleans can be used to easily compute the least upper bound of the reachable non-transient states. Finally, Shiple shows that a circuit is constructive if and only if this least upper bound contains only the Booleans 0 and 1 identifying a unique stable state, i.e., if it stabilizes for all gate and wire delays.

Constructiveness is extended to sequential circuits by requiring the combinational part to be constructive for any input and any reachable state. There is no added difficulty, see [52] for details. The set of legal inputs can also be restricted using input relations, see [8, 7]. In that case, constructiveness is required only for the legal inputs.

Constructiveness for combinational and sequential circuits is decidable. The BDD-based algorithms presented in [36, 52] perform an efficient fixpoint computation in a symbolic version of Scott's semantics. They synthesize the set of inputs that make a circuit constructive and yield an equivalent acyclic version that may be better for practical implementation purposes since conventional synthesis tools do not handle cycles.

5.7 The Constructive Semantics of Esterel

The constructive semantics of Esterel lifts the basic constructiveness idea to the imperative constructs. The idea is to control the logical behavioral rules by means of two auxiliary constructive predicates that determine for each input what a program *must* do or *cannot* do in terms of control and signal propagation. In the reactive system terminology, proof steps are called microsteps and they are used to define fine-grain operational semantics. See [7] for the rules. As for circuits, the constructive semantics can be presented in an equivalent denotational form that is directly synchronous and compositional. The operational semantics is adequate for studying the execution of a reaction, while the denotational semantics directly defines the input/output function of a module, abstracting away all possible microstep orderings.

With respect to the circuit translation, the main result is as follows:

Theorem 2 *An Esterel program is constructive for an input if and only if the translated circuit is.*

We are currently building a mechanically checked proof of that theorem using the COQ system [32].

Combining Theorem 1 and Theorem 2, we obtain the final result that an Esterel program is constructive if its circuit electrically stabilizes for all gate and wire delays. This final result shows that the constructive semantics is not only mathematical but also *physical*, which yields the most solid foundations to the language we can think of.

6 Implementation

Synchronous languages can be implemented on hardware or software centralized or distributed platforms. For simplicity, we concentrate on centralized software or hardware implementations. The reader interested in distributed implementation can refer to [18].

6.1 Control vs. Data

In imperative languages such as Esterel, the distinction between control and data is direct at source code level, see for instance the protocol in Section 4.2.3. A major property of Esterel is that control is *finite-state*. The implementation basically consists of building a deterministic control finite-state machine that schedules data-handling actions.

In data-flow languages, one can use the same implementation scheme by establishing a distinction between two kind of variables: actual data variables, to be computed at run-time, and control variables to be handled at compile-time using some kind of partial evaluation. Control variables are most often Boolean variables that express a property of the program state, e.g., some status is on or off; by extension, they can range over any finite set of values.

The control finite-state machine can be implemented in many different ways in software or hardware, with a variety of time-space tradeoffs. This gives us lots of freedom to meet application-dependent performance constraints. In the sequel, we detail the two main implementations, explicit automata and Boolean circuits, and we briefly discuss optimization issues. Implementation of data handling is comparatively easy provided one carefully analyzes the relationship between control dependencies and data dependencies.

6.2 Implementation By Explicit Automata

An explicit control automaton is given by a set of states and a transition from each state, which is a tree with unary or binary nodes. A unary node triggers a data action, a binary node is either an input signal presence test or a data test. The leaves of the transition are states. Reaction from a state follows the transition, executing the actions and performing the tests on the way, until reaching the state leaf from which the next reaction will start.

To compute the control automaton associated with an Esterel program, we adapt Brzozowski's *residual algorithm* originally introduced to translate regular expressions into automata [16, 12]. Given a program body p, we formally compute all derivatives p' for all input event sequences as specified by the formal semantics, but leaving data values uninterpreted. Then, we construct a finite automaton having the derivatives as states. That automaton is often close to minimal, for yet largely unknown reasons. Automata can also be constructed from data-flow programs using partial evaluation techniques.

The main advantage of automata is speed. Executing a transition is very fast and independent of program size. Since executing the data actions at run-time is necessary for any implementation, automata are close to time-optimal amongst centralized implementations. Local signals used for internal communication between statements disappear in the automaton, exactly as intermediate non-terminals disappear in parser generation. Therefore, local signals are truly zero-delay at run-time. Further optimizations concerning the orders of tests in transitions are analyzed in [19].

The drawback of automata is of course size. Only relatively small applications can be handled. Automata are usually appropriate for protocols, drivers, or man-machine interface systems. Large process-control applications most often lead to size explosion.

6.3 Implementation using Sequential Boolean Circuits

Sequential circuits were introduced in Section 3.1. Since n Boolean registers can hold 2^n states, a sequential circuit can denote an exponentially bigger automaton, which makes the state space explosion vanish. Direct implementation of acyclic sequential circuits in hardware is performed by sequential logic synthesis systems [50]. Software implementation in the cycle-based model of Section 3.2 is easy: sort the equations according to the variable dependency relation, print the equations as C assignments in order, then print the assignments of new values to the registers. Other more efficient software implementations are discussed in [39].

The translation of a data-flow program into a synchronous Boolean circuit is a simple process. Roughly, for Lustre, each Boolean `pre` delay operator generates a register, and one additional register is generated for all initializations by the `->` operator. The translation of an Esterel program into a sequential circuit was already mentioned in Section 5.4. The circuit's worst-case size is the square of that of the source program, but the squaring factor rarely shows up in practice. The translation can yield cyclic circuits, which are analyzed for constructiveness and made acyclic using algorithms presented in [36, 52].

6.4 Circuit Optimization

The direct translation of high-level programs into circuits usually yields rather fat circuits that must be optimized before actual implementation. Fortunately, circuit optimization has been extensively studied in the hardware community. For Esterel, we borrowed many existing algorithms and we also developed specific algorithms that give good practical results. Optimization can be split into two subproblems: combinational optimization, and sequential optimization

In combinational optimization, the game is as follows: given a network of combinational gates, build another network optimized w.r.t. size or speed criteria. There is a wide variety of academic and industrial tools for that purpose, see [14]. In the previously described software implementation of circuits, the reaction time is roughly proportional to the number of equations and operators. Therefore, size optimization is the issue even for speed. Hardware-directed tools can be used as well, provided one pretends to optimize the "area of silicon".

The sequential optimization problem can also be called the *state assignment problem*. A sequential circuit obviously denotes a finite automaton, the states of which are the register Boolean configurations reachable from the initial state by some input sequence. The mapping from states to configurations is called the state assignment. Given a circuit, the goal is to denote the same automaton using more efficient state assignments and fewer registers.

If the automaton has n states, it is clearly sufficient to use $log(n)$ registers. However, when changing the state assignment, one must change the combinational circuit accordingly. In the worst case, the new combinational size can be $2^{log(n)} = n$, which means that reducing the number of registers can make the combinational logic explode. The problem of finding the best $log(n)$ assignment is NP-complete and no good heuristics scale up for it. Furthermore, in many cases, adding a few registers can make the combinational logic much smaller. Therefore, the real problem is to find a good register / combinational logic tradeoff.

An Esterel program directly specifies such a tradeoff: a register is generated by each source Esterel temporal statement, and the combinational logic is generated by the other statements. Achieving *Write Things Once* ensures a good register assignment, which means that elegant programs have good implementation. However, there is often some unessential redundancy between the registers in the direct translation. In [48, 49], we present algorithms that reduce the number of registers without significantly changing the encoding, hence without making the logic explode.

For a simple example, consider the `ABRO` program of Section 4.2. Four registers are allocated: a boot register B, a register A for "`await A`", a register B for "`await B`", and a register R for "`each R`". The boot register B has initial value 0 and then value 1 at all cycles. The register A (resp. B) has value 1 while waiting for `A` (resp. `B`), and the register R has value 0 initially and 1 while waiting for R. Using BDD-based reachable states computation, one finds that R is always 1 except at start time, which implies $R = B$. Therefore, one can remove R and replace its output by that of B, without changing the logic. Combinational optimization then yields an optimal circuit. See [48] for more elaborate examples.

7 Verification

Since reactive software is often used for safety-critical applications, verification of program properties is fundamental. Here again, we directly benefit from work done in other areas such as process calculi and hardware circuit verification [34,

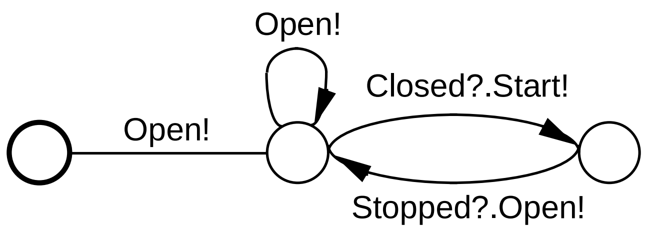

Figure 6: Bisimulation Reduction

13, 20]. We are interested in safety properties of the kind "wrong things never happen" and in bounded-response properties of the kind "something useful will happen before some time" that are in fact safety properties. Liveness properties of the kind "something useful will happen some day" are usually much less important for reactive systems.

Many useful properties are data-insensitive and can be proved or disproved using only the finite-state control structure of a program. We present the two techniques we use most often for such pure control properties, bisimulation reduction and verification using observers. As a running example, we use a lift controller and we show how to verify that the lift cannot travel with the door open. Data-dependent properties are analyzed in [1, 25]. They are usually much harder and will not be considered here.

7.1 Bisimulation Reduction

Bisimulation has been originally introduced by David Park and Robin Milner to define equivalence between process calculi terms [41]. The variant we use here is *weak bisimulation.*

Consider the lift controller. For the door, there is an output signal `Open` sent by the controller to open the door and an input signalsent by a door sensor when the door is closed. For lift motion, there is an output signal `Start` sent by the controller to start the engine and an input signal `Stopped` sent by a sensor when the lift is stopped. There are of course many other signals such as call buttons, bells, and whistles, which are not relevant to the property we want to prove.

The first step is to erase the useless signals and to keep only the four relevant signals `Open`, `Closed`, `Start`, and `Stopped`. If none of these signals appears on a transition, the transition is called a silent transition τ as in CCS [41]. After this erasure process, the automaton has the same number of states, the same number of transitions, and fewer transition labels. It can also be non-deterministic.

The second verification step is to perform *weak bisimulation reduction*, which

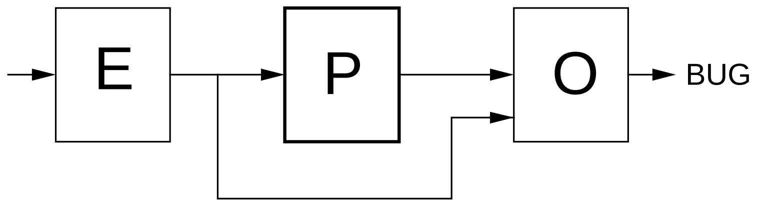

Figure 7: Verification by observers

consists of computing the smallest weakly bisimilar transition system. This is a very intuitive operation that can be explained to any user without mathematics: the reduct has the *same behavior* w.r.t. the observed signals, no spurious path is introduced, and no path disappears. Figure 6 shows the result for the lift controller. On such a three-state automaton, the property we want to verify is immediate.

Bisimulation reduction is performed by various tools. For Esterel, we mostly use the FcTools system described in [13]. The implementation is very efficient for explicit automata, but as yet much less efficient for sequential circuits since bisimulation is expensive to compute with BDDs.

7.2 Verification by Observers

Verification by observers is probably a folk technique. It has been made systematic for synchronous languages by Halbwachs *et. al.* [28]. A similar technique called *testing* has been extensively studied by M. Hennessy for process calculi [31].

The idea is described in Figure 7. The safety property to be verified for a program P is expressed as another reactive program called the *observer* O, which is put in synchronous parallel with P. The observer takes as inputs the inputs and outputs of P. Its only output is a signal called `BUG`. Since not all input sequences may be meaningful for P, another reactive program E called the *environment* can be put in synchronous parallel with P and O to only generate the useful input sequences. The outputs of E are the inputs of P, and the inputs of E are arbitrary signals acting as oracles. Notice that the synchronous observer O is purely passive: it listens to P without interfering with it, unlike in asynchronous formalisms where the observer interacts with the observed process and can restrict its behavior, which is quite unnatural for verification purposes.

The verification consists of checking that the signal `BUG` is never emitted by the triple $E \,||\, P \,||\, O$ for any input sequence of E. This can be done using standard reachability analysis techniques for explicitly or implicitly encoded finite-state machines [21, 33]. If the property is false, one can build counter-examples, i.e., sequences of useful inputs of P that violate the property.

Any synchronous language can be used to program the observer O and the environment E. For the lift example, we can write the observer in Esterel as follows:

```
Module Doors:
input Start, Open, Closed;
output BUG;
loop
   await Open;
   abort
      await Start;
      emit BUG
   when Closed
end loop
```

Temporal logic is also a well-known way of expressing properties [37]. Lustre can easily encode a temporal logic of the past, sufficient for most safety properties [28]. The TempEst system described in [33] allows the user to specify the observer as a temporal logic property, which is compiled into an Esterel program.

8 Conclusion

In this overview paper, we have tried to cover all aspects of synchronous programming, from theory to implementation. The kernel is of course the synchronous model of deterministic concurrency. Languages were grouped in two categories: data-flow and imperative. Smoothly unifying both styles is one of the important remaining challenges (we cannot explain here why this is non-trivial). The semantics are now well-understood, and major progress has been made recently in understanding causality issues through a somewhat unexpected use of constructive logic. Efficient implementation uses techniques from automata theory and hardware. Program verification is based on process calculi and finite-state machine verification techniques.

For all the synchronous languages designers, what really matters is the *use* of the languages and compilers. A lot of emphasis has been put on developing techniques and tools that scale to real-size programs. We hope that the synchronous tools will make their users happy, both in academia and in industry, and that application will foster new ideas and new research directions.

References

[1] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[2] C. André. Representation and analysis of reactive behaviors: A synchronous approach. In *Proc. CESA'96, Lille, France*, July 1996.

[3] D. Austry and G. Boudol. Algèbre de processus et synchronisations. *Theoretical Computer Science*, 30(1):91–131, 1984.

[4] G. Berry. Real-time programming: General purpose or special-purpose languages. In G. Ritter, editor, *Information Processing 89*, pages 11–17. Elsevier Science Publishers B.V. (North Holland), 1989.

[5] G. Berry. Esterel on hardware. *Philosophical Transactions Royal Society of London A*, 339:87–104, 1992.

[6] G. Berry. Preemption and concurrency. In *Proc. FSTTCS 93*, Lecture Notes in Computer Science 761, pages 72–93. Springer-Verlag, 1993.

[7] G. Berry. *The Constructive Semantics of Esterel.* Draft book, available at http://www.inria.fr/meije/esterel, 1996.

[8] G. Berry. *The Esterel Language Primer.* Draft book, available at http://www.inria.fr/meije/esterel, 1996.

[9] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.

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

[11] G. Berry, S. Moisan, and J-P. Rigault. Esterel: Towards a synchronous and semantically sound high-level language for real-time applications. In *Proc. IEEE Real-Time Systems Symposium, Arlington, Virginia, IEEE Catalog 83CH1941-4*, pages 30–40, 1983.

[12] G. Berry and R. Sethi. From regular expressions to deterministic automata. *Theoretical Computer Science*, 48:117–126, 1986.

[13] A. Bouali, A. Ressouche, V. Roy, and R. de Simone. The fc2tools set. In *AMAST'96*, volume 1101 of *LNCS*, Munich, Germany, 1996.

[14] R.K. Brayton, G.D. Hachtel, and A.L. Sangiovanni-Vincentelli. Multilevel Logic Synthesis. *Proceedings of the IEEE*, 78(2):264–300, February 1990.

[15] R.E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[16] J. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4), 1964.

[17] J. Brzozowski and C.-J. Seger. *Asynchronous Circuits.* Springer-Verlag, 1996.

[18] P. Caspi and A. Girault. Distributing reactive systems. In *Proc. Intnl. Conf. on Parallel and Distributed Computing Systems, Las Vegas*, 1994.

[19] C. Castelluccia, W. Dabbous, and S. O'Malley. Generating efficient protocol code from an abstract specification. In *Proc. ACM SIGCOMM, Stanford*, 1996.

[20] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 1(15):36–72, 1993.

[21] O. Coudert, C. Berthet, and J. C. Madre. Verification of Sequential Machines Based on Symbolic Execution. In *Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, 1989.

[22] G. Cousineau. An algebraic definition for control structures. *Theoretical Computer Science*, 12:175–192, 1980.

[23] J-Y. Girard, Yves Lafont, and Paul Taylor. *Proofs and Types.* Cambridge Univerity Press, 1989.

[24] P. Le Guernic, M. Le Borgne, T. Gauthier, and C. Le Maire. Programming real time applications with Signal. *Another Look at Real Time Programming, Proceedings of the IEEE, Special Issue*, Sept. 1991.

[25] N. Halbwachs. Delay analysis in synchronous programs. In *Proc. CAV'93*, pages 333–346, 1993.

[26] N. Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer, 1993.

[27] N. Halbwachs, P. Caspi, and D. Pilaud. The synchronous dataflow programming language Lustre. *Another Look at Real Time Programming, Proceedings of the IEEE, Special Issue*, Sept. 1991.

[28] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In *Proc. AMAST'93*, june 1993.

[29] N. Halbwachs and F. Maraninchi. On the symbolic analysis of combinational loops in circuits and synchronous programs. In *Euromicro'95*, Como (Italy), september 1995.

[30] D. Harel. Statecharts: a visual approach to complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[31] M. Hennessy. *Algebraic Theory of Processes.* MIT Press, Cambridge, Massachusetts, 1988.

[32] Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. The Coq proof assistant : A tutorial : Version 6.1. Technical Report RT-204, INRIA, 1997.

[33] L. Jategaonkar Jagadeesan, C. Puchol, and J. E. Von Olnhausen. Safety property verification of Esterel programs and applications to telecommunications software. In *Proc. CAV'95*, LNCS. Springer-Verlag, july 1995.

[34] R.P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach.* Princeton U. Press, Princeton, New Jersey, USA, 1995.

[35] P.J. Landin. The next 700 programming languages. *Comm. ACM*, 9:157–166, 1966.

[36] S. Malik. Analysis of cyclic combinational circuits. *IEEE Trans. Computer-Aided Design*, 13(7):950–956, 1994.

[37] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, 1992.

[38] F. Maraninchi. The Argos language: graphical representation of automata and description of reactive systems. In *International Conference on Visual Languages, Kobe, Japan*, 1991.

[39] P.C. McGeer, K.L. Mcmillan, A. Saldanha, A. Sangiovanni-Vincentelli, and P. Scaglia. Fast discrete function evaluation using decision diagrams. In *Proc. International Conf. on Computer-Aided Design (ICCAD)*, 1995.

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

[41] R. Milner. *Communication and Concurrency.* Prentice-Hall International, Englewood Cliffs, 1989.

[42] R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML.* MIT Press, 1991.

[43] Robin Milner. Functions as processes. *Journal of Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[44] G. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):452–488, september 1976.

[45] G. Plotkin. A structural approach to operational semantics. Technical Report report DAIMI FN-19, University of Aarhus, 1981.

[46] V. A. Saraswat, R. Jagadeesan, and V. Gupta. Foundations of timed concurrent constrained programming. In S. Abramsky, editor, *Proc. 9th Ann. IEEE Symp. on Logic in Computer Science*. IEEE Computer Press, 1994.

[47] V. A. Saraswat, R. Jagadeesan, and V. Gupta. Default timed concurrent constraint programming. In *Proc. POPL'95, San Francisco, USA*, pages 272–285, 1995.

[48] E. Sentovich, H. Toma, and G. Berry. Latch optimization in circuits generated from high-level descriptions. In *Proc. ICCAD'96*, 1996.

[49] E. Sentovich, H. Toma, and G. Berry. Efficient latch optimization using exclusive sets. In *Proc. DAC'97, Anaheim*, 1997.

[50] E.M. Sentovich, K.J. Singh, C. Moon, H. Savoj, R.K. Brayton, and A Sangiovanni-Vincentelli. Sequential circuits design using synthesis and optimization. In *Proc. ICCD'92*, pages 328–333, 1992.

[51] T. Shiple. *Formal Analysis of Cyclic Circuits.* PhD thesis, University of California at Berkeley, 1996.

[52] T. Shiple, G. Berry, and H. Touati. Constructive analysis of cyclic circuits. In *Proc. International Design and Test Conference ITDC 96, Paris, France*, 1996.

[53] H. Touati and G. Berry. Optimized controller synthesis using Esterel. In *Proc. International Workshop on Logic Synthesis IWLS'93, Lake Tahoe*, 1993.

[54] J. Vuillemin. On circuits and numbers. *IEEE Transactions on Computers*, 43:8:868:27–79, 1994.