The Esterel v5 Language Primer
Version v5_91

Gérard Berry
Centre de Mathématiques Appliquées
Ecole des Mines and INRIA
2004 Route des Lucioles
06565 Sophia-Antipolis

berry@sophia.inria.fr

June 5, 2000

Contents

Chapter 1

Introduction

This document is the primer for the Esterel$^{\mathrm{TM}}$ synchronous programming language, which is devoted to programming control-dominated software or hardware reactive systems. The language version is that of the Esterel v5 system, version v5_91. This new language version extends the previous version v5_21 by the addition of new `pre` operators, which makes it possible to access the previous status and value of a signal. This has always been possible in data-flow languages such as Lustre [33] and Signal [31], but has long been missing in Esterel. This addition should simplify programming and reduce causality problems. The rest of the language is left unmodified.

The reader familiar with previous versions of this document should simply read Section 3.3, Section 3.5, Section 4.6.1, Section 4.6.2, Section 4.6.3, Section 4.7.12, Section 4.7.13, and Chapter 5.

We tried to write the primer in a precise but informal way that should make most users happy in their use of the language and system. However, the primer is not meant to be the reference document for the language and semantics. The language reference manual will be appended to the primer in a subsequent release and the formal semantics is given in the companion book "The Constructive Semantics of Pure Esterel" [9], which discusses all mathematical semantics issues in depth. The use of the Esterel v5 system is presented in the documentation delivered with the system.

1.1 Organization of the Primer

Chapter 2, *Deterministic Reactive Systems*, describes the systems to which Esterel is dedicated. Chapter 3, *The Esterel Programming Style*, illustrates Esterel programming by examples. Chapter 4, *A Tour of Esterel*, presents all the Esterel constructs. Chapter 5, *Constructive Causality*, studies semantical issues related to program correctness. Chapter 6, *Reflections on Perfect Synchrony*, reflects on the semantical model of Esterel and on its practical meaning. Chapter 7 presents the Esterel grammar and some old syntax that can still be used for backwards compatibility reasons.

1.2 The Evolution of Esterel

The Esterel language is both stable and in constant evolution. In its development, we always tried to conciliate two objectives:

- Keeping the language stable, to guarantee that already written programs compile without modification and that the compiled code can also be used without modification in its original execution environment.

- Making the language evolve, to follow the progress in the scientific knowledge on which it is grounded, to make it more more powerful, and, above all, to make it more user-friendly

To us, it is right to ask users to recompile their programs to benefit of new compiler versions that correct bugs or produce better code. It is much more questionable to ask them to rewrite or even adapt existing programs that have been extensively and carefully tested and verified, especially in the context of critical applications. All programs that have been written since 1985 (i.e. for the Esterel v2, v3, and v4 systems) still compile in a compatible way, but much more efficiently.

As far as evolution is concerned, new statements have been added over the years, such as the `exec` external task execution statement, and the syntax has been improved. For example, we now suggest to replace the "`do...watching`" statement by "`abort...when`" that is clearer and extends more naturally into "`weak abort...when`" that has been long missing. This document will always use the new syntax and it will mention the old syntax only for compatibility with the past.

The semantics has never "changed" in the sense that all programs that used to compile still do and keep the same semantics. However, the introduction of the *constructive semantics* has made the ground very firm and has solved many of the annoying problems encountered by users of previous versions that sometimes unduly rejected correct programs. The constructive semantics is informally presented here, see [9] for the formal presentation.

Our conservative politics has two obvious drawbacks: evolution is slowed down, and design mistakes must be faithfully carried over from a version to the next one. To reduce the number of mistakes, we introduced new constructs only when we thought we really understood all the transitive consequences of our choices and all the interactions between them. We were reasonably successful as far as executable statements are concerned, but much less for the module structure that will require a deep revision to be really usable for big programs[1].

Every language is a compromise; the best a language can do is to be transparent, i.e. to let the user reflect its way of thinking in the most direct and elegant way. Since this goal is highly unreachable, users usually *complain* about languages, even if they like them. Complaints about Esterel should be addressed to *esterel-users@cma.inria.fr*. Complaints about Esterel v5 system bugs should be addressed to *esterel-bugs@cma.inria.fr*.

1.3 A Brief History of Esterel

In 1982, Jean-Paul Marmorat and Jean-Paul Rigault, two researchers in Control Theory and Computer Science at CMA [2], were designing a robot car for a race organized by an early microcomputer journal. They rapidly understood that the tools they had for programming the car were very far from being satisfactory. Classical languages did not let them express control algorithms in the way they were thinking about them. They recognized the need for specific statements to deal with time, namely delays and preemption, and they made the point that the repetition of any signal should count as an autonomous time unit. They drafted a little language using intuitive keywords. At that time, the author was working on theoretical aspects of λ-calculus and denotational semantics. He found the application area and the new ideas interesting and challenging, and he tried to make mathematical sense out of the language proposal.

With Sabine Moisan — who found the name Esterel[3] — and Jacques Camerini, we sorted out the primitives and we tried to use the newly developed SCCS and Meije synchronous process calculi to give their semantics

[1] At the times where Esterel was designed, modules were not very well understood altogether, and we could not figure out what was the right choice. Then, causality issues attracted our attention and energy much more than a redesign of modules.

[2] Centre de Mathématiques Appliquées, Ecole des Mines de Paris, Sophia-Antipolis

[3] The Estérel is a small but beautiful mountain range of red rocks culminating at 619 m, between Cannes and St Raphaël; "-terel" sounds a little bit like "temps-réel", which is real-time in French.

(SCCS [44] is due to Robin Milner and Meije [3] to Gérard Boudol). The reference [14] is the oldest one about Esterel.

Process calculi turned out not to be well-adapted to Esterel. Fortunately, Gordon Plotkin published his seminal book on Structural Operational Semantics or SOS [49]. This new semantics style gave us much sharper tools to describe intrinsic semantics, and, above all, it freed us from the classical vision of concurrency as bound to interleaving and rendezvous communication, which was inappropriate in our control world. Laurent Cosserat and the author worked out a better set of primitives and made much better sense of instantaneous control propagation and communication, which is the key to get Esterel right[11, 26]. The author rediscovered the beautiful derivative algorithm of Brzozowski that translates any kind of regular expression into automata [21] and that can be extended to any finite-state language described by SOS rules. Using this algorithm, Laurent Cosserat wrote the first Esterel v1 prototype compiler to automata. That compiler was entirely rewritten by Philippe Couronné and the author in 1985-86, and the new Esterel v2 system was swiftly used for non-trivial academic and industrial applications.

Georges Gonthier's thesis [29] was a major step in the development of Esterel. He understood the fundamental distinction between the behavioral semantics, which is of a logical nature and based on mutual agreement about signal presence and values, and causality issues that deal with effective information propagation. He introduced the fundamental encoding of synchronization and exception by integers, and he designed much more efficient operational semantics and compiling algorithms. While the Esterel v2 compiler strictly used Brzozowski's original derivative algorithm in which automaton states are program texts, which is unreasonably memory-consuming, Gonthier's technique uses simple bit-sets as states, which is orders of magnitude more efficient (see [15] for the same kind of optimization for regular expressions). Finally, Gonthier designed the overall architecture of the Esterel v3 compiler, which was written in 1987-88 by Raphaël Bernhard, the author, Frédéric Boussinot, Annie Ressouche, Jean-Paul Rigault, and Jean-Marc Tanzi. The architecture and some intermediate codes have been kept basically unchanged since then. Esterel v3 has been used rather heavily. It worked well for small to medium-size programs, but blew up or big programs for which state space explosion turned out to be the rule.

The next progress came from interaction with Jean Vuillemin's group at Digital Equipment Paris Research Laboratory. This group was developing the PeRLe FPGA-based programmable hardware machine [16]. Many of the hardware designs involved controllers that are a pain in the neck to write with gates and registers, and the group thought that Esterel was very well adapted for direct controller specification. The author learned about logic and hardware and developed a structural translation of Esterel programs into gates that could be used to directly generate netlists, fully avoiding the state space explosion of Esterel v3 [6]. Hervé Touati brought Computer Aided Design technology in the picture and he showed how to deeply optimized the obtained netlists to make them practical [56]. The author then extended the logic translation to software implementation of general Esterel programs. Xavier Fornari extended the hardware optimization techniques to deal with software generation. Frédéric Mignard cleaned up many processors of the Esterel v3 compiler and built a bunch of new ones. Jean-Pierre Paris implemented external task control by the `exec` statement. Jean-Paul Marmorat and Jean-Pierre Paris developed the graphical simulator and symbolic debugger `xsimul`, which later became the current `xes` tool. The resulting Esterel v4 compiler was delivered in 1992.

Esterel v4 was much better than Esterel v3 since it avoided state space explosion. However, it required generated circuits to be acyclic. Although this condition is standard in hardware or data-flow systems design, it turned out to be too restrictive for Esterel. The older Esterel v3 compiler was quite smart about causality and was able to compile many correct but cyclic programs that were rejected by the more recent Esterel v4 compiler. This made our users unhappy, and we could not convince them that the problem lied in their bad programming habits. On the contrary, they convinced us that making safe cycles is natural when programming symmetric protocols or resource access strategies. The solution came from an encounter with a paper of Sharad Malik on cyclic circuits [41], which we later extended with Tom Shiple [54] by showing the equivalence between three points of views on Boolean circuits: the electrical point of view that deals with current propagation and delays, the constructive logic point of view that deals with proving values of Boolean variables in a constructive way[4], and the denotational point of view of Scott's semantics. This lead to the constructive semantics presented in [9] and to the current Esterel v5 compiler. Technically, the compiling algorithms for the constructive semantics are based on Binary Decision Diagrams. They were implemented by Tom Shiple and Horia Toma with help of Hervé Touati and Jean-Christophe Madre, using the Tiger BDD system of Olivier Coudert, J-C. Madre, and H. Touati. Ellen Sentovich made many contributions to the presentation

[4] i.e. without speculative computation or reasoning by contradiction.

of the semantics and to optimization techniques, that were developed and implemented by Horia Toma [51, 52]. Xavier Fornari is now in charge of the development of the Esterel v5 compiler, which has been heavily tested by Monica Robert.

Synchronous languages are not enough for complex systems programming and they must interact with other languages and communication styles, in particular with asynchronous ones. The CRP formalism (Communicating Reactive Processes) developed by the author, R.K. Shyamasundar from TIFR Bombay and S. Ramesh from IIT Bombay, India, is an attempt at marrying Esterel and CSP. See [28] for a presentation of CRP. The Polis hardware / software codesign system also uses Esterel in a mixed synchronous / asynchronous framework. See also [5] for a comparison between synchrony and asynchrony.

The verification tools for Esterel are developed by a group headed by Robert de Simone, who also deeply contributed to many aspects of the design of the language and tools. Didier Vergamini developed the early automata manipulation algorithms included in the Auto explicit verification system, Amar Bouali wrote the current Xeve BDD-based verification package using the Tiger library. Valérie Roy developed the Autograph automata visualization tools. Annie Ressouche wrote most of the programs that interface the compiler and the verifiers. These tools are accessible from the Esterel Web page. Carlos Puchol, from University of Texas, and Lalita Jagadeesan, from AT&T Bell Laboratories, developed the Tempest temporal logic verification system for Esterel [38]. Amar Bouali and Valérie Roy replaced the original explicit state enumeration technique by implicit traversal using BDDs in the Hurricane evolution of Tempest.

1.4 Other Synchronous Languages

Esterel is a member of a small community of synchronous languages born in the beginning of the 80's. Cooperation has been constant with the Lustre team [33] headed by Paul Caspi and Nicolas Halbwachs in Grenoble, France, and the early Esterel and Lustre tools shared intermediate languages and compilation tools. Cooperation was later extended to the Signal [31] team headed by Paul le Guernic and Albert Benveniste in Rennes, France, The design of Esterel was influenced by the independent design of the Statecharts visual formalism [35] introduced by David Harel in 1984 and by the Argos synchronous variant of Statecharts developed by Florence Maraninchi in Grenoble [42]. See [32] for a global survey of these synchronous languages.

Charles Andre's SyncCharts [1] graphical formalism is an extension of Argos that yields the power of Esterel. The Modecharts [47] formalism is another synchronous graphical language. The Reactive C language developed by Frédéric Boussinot [18] is a reactive extension of C that borrows its main concepts from synchronous languages and has evolved into reactive objects languages [20]. Frédéric Boussinot and Robert de Simone also developed a synchronous language called SL [19] that can be viewed as a syntactic restriction of Esterel where all causality problems are suppressed and that can be easily implemented using Reactive C.

1.5 Acknowledgements

Special thanks to Maurice Gherardi, from the Simulog company, who performed a very careful proof-reading and wrote the syntax chapter. Many thanks to Frédéric Boussinot (Ecole des Mines), Sylvan Dissoubray (Simulog) Xavier Fornari (Armines), Emmanuel Ledinot (Dassault Aviation), Eric Nassor (Dassault Aviation), Monica Robert (Armines), and Ellen Sentovich (Cadence) for their proof-reading and numerous improvement suggestions.

Please send report any mistake you find or send any improvement suggestion to berry@cma.inria.fr.

Chapter 2

Deterministic Reactive Systems

2.1 Transformational, Interactive, and Reactive Systems

Computerized systems can be divided into three broad categories:

- *Transformational systems* compute output values from inputs values, and then stop. Most numerical computation programs, payroll programs, and compilers are transformational.

- *Interactive systems* constantly interact with their environment in such a way that the computers can be viewed as the masters of the interaction. A user calls for services, the system listens to him when it can, and it delivers the services when they are available. Operating systems, centralized or distributed databases, and the Internet are interactive.

- *Reactive systems*, also called *reflex systems*, continuously react to stimuli coming from their environment by sending back other stimuli. Contrarily to interactive systems, reactive systems are purely input-driven and they must react at a pace that is dictated by the environment. Process controllers or signal processors are typical reactive systems.

Of course, a given large-scale computerized system never falls entirely in any of the three categories. A banking system involves transformational payroll programs, interactive access to data bases of clients, and reactive automatic teller machines and graphical user interfaces. Nevertheless, it is always useful to identify which *parts* of the system are transformational, interactive, or reactive, and to handle them with appropriate tools.

Interactive and reactive systems call for *concurrent programming*. First, they act concurrently with their environment. Second, they are themselves most often made of concurrent parts that communicate with each other, such as the bank and the banker teller in the banking system or the time-keeper, stopwatch, and alarm in a digital watch.

2.2 Control-Dominated Reactive Systems

In most reactive applications, there is a fairly clear distinction between *data handling* and *control handling*. Data handling is about continuously producing output values from input values. It is the typical activity of signal processing programs. Control handling is about producing discrete output signals from input ones. It is the typical activity of man-machine interface or supervision programs, among other examples listed below. As usual, the two aspects are not fully independent: for instance, in signal processing applications, there are usually several computation modes supervised by some discrete commands. ESTEREL is an imperative concurrent language specifically dedicated to *control-dominated* (parts of) reactive programs, which are found in the following application areas:

- *Real-time process control*, in manufacturing, transportation systems, etc. The input stimuli are physical parameters produced by sensors, operator commands, device failure signals, etc. The outputs are commands to motors, valves, etc. The program must react to its inputs within a short predefined time frame. See [10] for examples in avionic software development, and see [4] for the use of Esterel in the Polis hardware / software codesign system.

- *Embedded systems* that drive automated objects such as robots and domestic appliances. Sequencing tasks is the primary concern. See [27] for examples in robotics.

- *Supervision* of complex systems, where information coming from a variety of sources is gathered and handled by one or more supervisors, that can be programs, human beings, or a combination of those. Safety systems that detects anomalies in complex processes are typical supervision systems.

- *Communication protocols*, or more precisely the control part that concerns connection, disconnection, failure recovery, quality of service control, etc. The stimuli are either physical signals sent by line handler devices or logical signals generated by packet decoding. See [13, 48, 38, 39, 24] for examples.

- *Peripheral drivers* that control disks, printers, or other computer peripherals.

- *Hardware glue logic and controllers.* Glue logic is a common name for protocols and drivers in hardware design. A typical glue logic device is an interface between two busses. Controllers are finite state machines driving data-paths, usually opening and closing multiplexer according to control conditions. See [6] for examples.

- *Human-machine interface.* HMI is of course ubiquitous and is a basic tool for supervision. Basic GUIs (Graphical User Interfaces) use a fixed set of predefined interactors such as menubars, scrollbars, etc., see [25]. Modal HMIs fold a lot of commands into a small number of interactors and display a lot of values on a small number of display units. They are very common in industrial systems or in airplane cockpit. See [8] for the small but representative example of a digital wristwatch.

Esterel can be used in all these quite diverse domains that are usually handled in different and often separated technical communities. This poses a non-trivial terminology problem, since the same words can have different meanings in different communities, and since the same concepts can be named by different words. In this document, we try to do our best to define all the technical words we use and to relate them to other words used in the different fields whenever necessary.

2.3 Determinism Versus Non-Determinism

Determinism vs. *non-determinism* is a key difference between reactive and interactive systems. A system is said to be deterministic if the same sequence of inputs always produces the same sequence of outputs. It is non-deterministic otherwise.

Determinism is the rule for reactive systems. It is very clear that an airplane must be driven in a deterministic way, and the same holds for any controlled physical system governed by deterministic physical laws[1]. In man-machine interface systems, it is also clear that user commands have precise deterministic meanings. We leave it for the reader to check that all the aforementioned applications are deterministic.

On the contrary, behavioral non-determinism is the rule for interactive systems for which the interaction is driven by the computers. Even if the Unix kernel is internally deterministic, its scheduling and resource allocation policies are unknown to the user, who naturally perceives the system as non-deterministic: running twice the same sequence of commands can produce different results if there are other interacting users. There should be no need to convince the reader that Internet is and must be non-deterministic.

Non-determinism is much harder to handle than determinism. Non-deterministic systems are harder to specify, and it is not even trivial to define a good notion of behavior and equivalence for them, while execution traces are perfectly adequate for deterministic systems. Debugging non-deterministic systems can be a nightmare since transient bugs may not be reproduced. Analyzing systems is also much more difficult since the state space tends to explode. Therefore, it is important to reserve non-determinism for places where it is really mandatory, i.e. interactive systems, and to forget about it for reactive systems. Historically, it was long thought that concurrency and non-determinism had to go together. As we shall see in the sequel, this is wrong. The main merit of synchronous languages is probably to have reconciled concurrency and determinism.

2.4 Programming Tools

We are mostly interested in the programming part of reactive systems, that is, in writing *reactive programs* and making them work. Let us explain why we need specific tools for this purpose.

Transformational systems have been well-studied for very long, since they used to form the bulk of computing. Most programming constructs and languages were originally tailored for them. It is very clear that their extension to event handling in interactive or reactive systems is not obvious: classical languages lack concurrency and event-handling primitives.

Tools for interactive systems programming came afterwards because of the development of operating systems and distributed algorithms. They introduced the key notions of concurrency, communication, and synchronization between processes. In practice, interactive applications are built either

[1] more precisely, modeled by deterministic systems of mathematical equations.

by linking conventional transformational programs through synchronizing operating systems calls, or by using specific languages such as CSP [36], Occam [37], or Ada [22] that make concurrency and communication first-class citizens. The advantage of specific concurrent languages is clear for program behavior analysis: mathematical semantics can be given in an intrinsic way and proof rules can be derived from it [36]. Process calculi try to abstract away from the concrete programming issues and to address the essence of interactivity [45]. Large-scale networking has considerably extended the scope of interactive systems and unveiled new needs such as process migration, realized for example in Milner's π-calculus [46] and in the Java language [2]

Tools for reactive systems did not develop in the same way. The subject was long kept outside the main core of Computer Science. In our belief, there are three reasons. First, the control and embedded applications used to belong to the domain of control theory and electrical engineering, where the traditions and technical background were quite different[2]. Second, high-level reasoning and programming tools promoted by computer scientists were considered as quite useless by many control engineers, whose main problem was (and still is) to pack code in small ROM for cheap microprocessors. Third, it was long thought by computer scientists that tools designed for interactive systems would cover reactive systems equally well. This turned out to be false, since classical concurrent languages are non-deterministic, give no guarantee about response times, and give no fine control over life and death of reactive activities.

Reactive systems were identified as such in the early 80's by several authors [14, 30, 23, 35], the name being given by David Harel and Amir Pnueli. Synchronous languages were born from the recognition that instantaneous broadcasting was the way to handle communication in reactive systems, making it possible to handle together concurrency, determinism, and response time control, and yielding a programming style far more natural than the one enforced by conventional interactive communication mechanisms. Lustre [33], developed by Paul Caspi and Nicolas Halbwachs, is a synchronous functional data-flow language for data-dominated systems. Signal [31] is a powerful relational synchronous data-flow language developed by Paul le Guernic and Albert Benveniste. Statecharts [35] is a visual formalism introduced by David Harel in 1984 for hierarchical state machine design. Argos [42] is a purely synchronous variant of Statecharts developed by Florence Maraninchi. Electre [50] is an imperative

[2] Even the computers used to be different.

task scheduling language that borrows many concepts from synchronous languages. Some of these languages are surveyed in a book by Nicolas Halbwachs [32].

All the reactive languages are accompanied by automatic formal verification tools, which are now of common use in industrial environments. It is now recognized that *safety* is the main concern in reactive systems. This is why rigorous high-level languages and verification tools will gradually replace low-level hand programming and testing, as in all the other areas of Computer Science. Formal verification issues will not be covered in this language primer, see [17] for more information.

As of now, there is no completely satisfactory way of unifying the data-flow styles of LUSTRE and SIGNAL and the control flow styles of ESTEREL and STATECHARTS. The programming primitives one need for data-handling and control-handling are indeed quite different. This is somewhat unfortunate for the user. Research is active in this area, and we hope to come up with a reasonable theoretical and practical unification in a finite amount of time.

We mentioned that classical concurrent languages were inadequate for reactive programming. The converse is also true: reactive languages are largely inadequate for interactive programming as a whole, although they can be locally useful for reactive parts of interactive programs. Here also, there is no known way to unify the interactive and reactive styles [3]. A nice pragmatic superposition of both approaches could be the network of reactive objects studied for example in [28].

[3] In [7], we claim that reactive programming is akin to Newtonian mechanics while interactive programming is akin to chemistry. We shall not develop the argument here, but just remind the reader that there is no physical theory that unifies *in a simple way* mechanics and chemistry.

Chapter 3

The Esterel Programming Style

3.1 Pure Signal Handling: the ABRO example

Our first example is the following specification:

> Specification `ABRO`:
> *Emit an output* `O` *as soon as two inputs* `A` *and* `B` *have occurred.*
> *Reset this behavior each time the input* `R` *occurs.*

The `ABRO` specification can be implemented either in hardware or in software. We first assume that the implementation is to be done by a synchronous digital circuit, because circuits have the simplest timing model to start with. Software implementation of the same program will be explained later on.

The interface of a (sequential) digital circuit is defined by a list of input wires, a list of output wires, and a clock. In the electrical view, the wires carry voltages, say 0V and 3.3V, and the clock tells when outputs can be sampled and when values can be latched in registers. In terms of clock cycles, registers behave as unit-delay elements. In the logical view, the wires carry Booleans values and the clock determines the successive reactions instants. The logical values are called 0 and 1, or `true` and `false`, or *set* and *unset*, or *present* and *absent*, according to local usage. In ESTEREL, we use *present* and *absent*, reserving 0 and 1 for integers and `true` and `false` for Boolean data. Presence or absence defines the *status* of a signal. In the `ABRO` specification, we say that signals *occur*, which is the same as saying that they are present.

An *input event* is defined by a status for each input. Given an input event, the circuit computes an *output event* made of a status for each output. A new input event is processed at each clock cycle.

Being carried by independent wires, hardware input signals can be simultaneously present in the same clock cycle. The ABRO specification is actually ambiguous, because it does not talk about handling such simultaneous occurrences of signals. There is clearly no problem if A and B are simultaneously present: O must be emitted right away. What if A and B are present when R occurs? We assume that R takes priority and that A and B are ignored when R occurs. Other choices will be studied later on. There is also another minor ambiguity at *boot time*, i.e. for the initial transition. Here, we assume that booting ABRO is a blank step and that the real behavior starts only after boot. This is a common assumption.

3.1.1 Execution Traces

It is convenient to have a notation for execution traces of a specification or program. We use that of the Esterel v5 csimul simulator. Here is a correct trace w.r.t. the ABRO specification:

```
> ;
Output:
> A;
Output:
> B;
Output: O
> R;
Output:
> A B;
Output: O
> ;
Output:
> A B R;
Output:
```

A trace is an alternated sequence of *reactions*, each reaction being composed of an input event preceded by the prompt '>' and an output event preceded by 'Output:'. An input event is written as the list of present signals terminated by a semicolon. Only the present or *emitted* output signals are mentioned in the output list; for controller programming, the main goal of Esterel, this is usually economical since control signals tend to be more often absent than present. A blank event is one with no signal mentioned, i.e. one where all signals are absent. Here, the initial (boot) input event is blank and there is another blank input event before the second reset. The response to this event is a blank output event.

The trace notation implicitly defines the basic timing model. Time is logical and seen as generated by the sequence of reactions, also called *instants* or *ticks*, which directly correspond to hardware clock cycles. We say that a reaction occurs at time t if its rank in the trace is t. With that timing notion, the output part of an event occurs at the same logical time as the input: only bookkeeping actions are performed in a reaction, and such actions should not consume logical input time. This model is called the perfectly synchronous or *zero delay* model. It will be analyzed in depth in Chapter 6.

3.1.2 Mealy Machines

A common way of programming `ABRO` is to design a deterministic *Mealy machine* (in state graph form), which is a deterministic finite automaton in which each transition arrow bears an input / output label. A Mealy machine for `ABRO` is pictured in Figure 3.1. A transition label contains input signals preceded by either '`?`' for presence or '`#`' for absence and output signals preceded by '`!`'. A transition labeled "`?A.#B.!O`" is taken if `A` is present and `B` is absent in the input event, and it provokes emission of `O`. A label acts as a filter for input events. In "`?A.#R.!O`", the transition is taken if `A` is present, independently of the presence of `B`. In other words, a transition labeled "`?A.#R.!O`" abbreviates the pair of transitions "`?A.?B.#R.!O`" and "`?A.#B.#R.!O`".

Notice that a Mealy machine can be seen as a folding of all possible traces into a single graph, sharing states with identical futures. Hardware can be automatically synthesized from Mealy machine specifications.

3.1.3 ABRO in Esterel

Since we have Mealy machines, why should we bother building new languages? Look at the automaton in Figure 3.1. Each signal appears several times, unlike in the original specification. For example, `A` appears 3 times positively: once on the left, as the first input, once on the right, as the second input, and once in the middle, for the case where `A` and `B` are simultaneous. It also appears negatively on the right. The reset signal `R` appears 8 times, 3 times positively and 5 times negatively. The output `O` also appears 3 times, one for each possible sequencing of `A` and `B`. Consider now the prob-

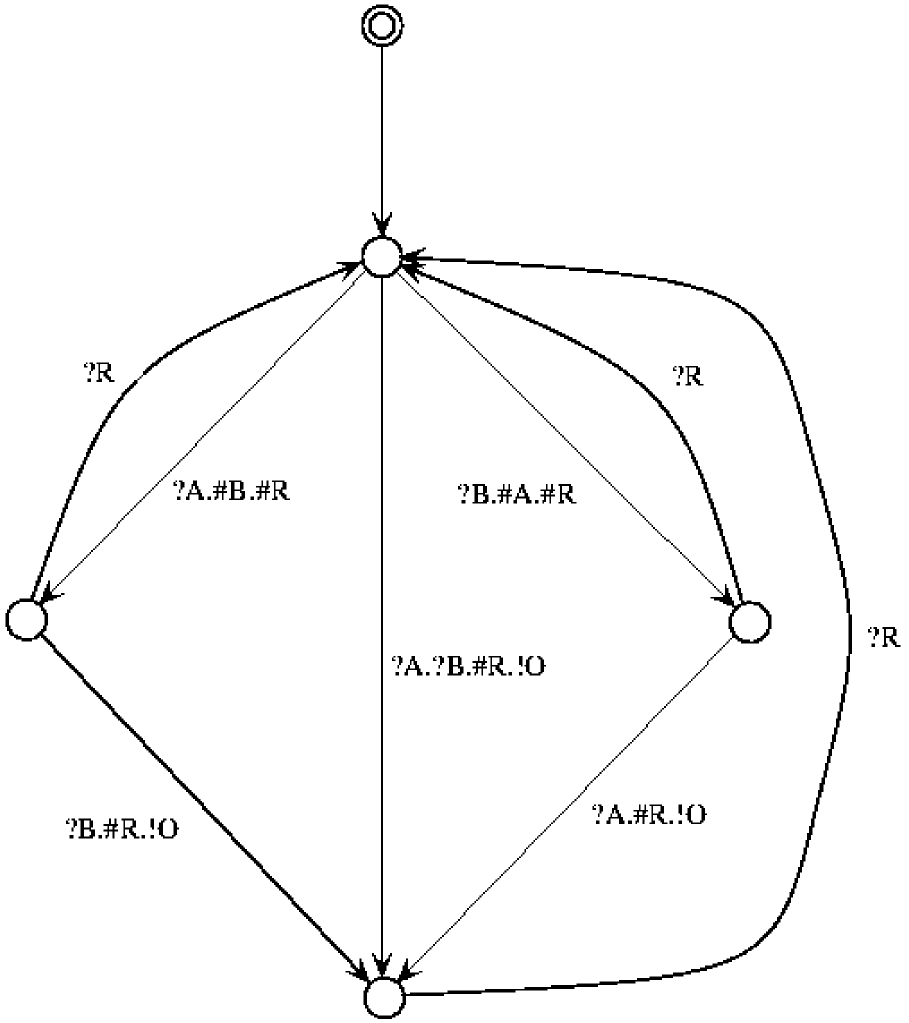


Figure 3.1: The ABRO Mealy machine

lem `ABCRO`, where there is one more signal `C` to wait for before emitting `O`. We leave the automaton drawing to the reader: the automaton core now has the shape of a 3-D cube with 8 vertices. Of course, the n-signal problem yields a n-cube with 2^n vertices. This is not for human beings. Furthermore, automata are hard to draw and read if not ridiculously small, and they are very sensitive to specification changes. In practice, Mealy machines are in no way good programs. In contrast, the Esterel code for `ABRO` is:

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

The declaration part is trivial. The body is written in an imperative way using control threads. It involves five ingredients: delay, signal emission, sequencing, concurrency, and abortion. It can be read either in an outside-in way, insisting first on the reset aspects, or in an inside-out way, insisting first in the basic behavior. We choose the inside-out way for this example.

Esterel statements are imperative. A statement *starts* in some instant t, remains *active* for a while, and may *terminate* in some instant $t' \geq t$. A statement is *instantaneous* if $t' = t$; hardware designers would say that the statement is *combinational.* If $t' > t$, we say that the statements *takes time*; hardware designers would say that the statement is sequential.

The delay statement "`await A`" always takes time; it *lasts one* `A`: if started at time t, it terminates at the least $t' > t$ such that `A` occurs at t', staying active in between. In terms of threads, the current control thread is stuck at the `await` statement until `A` occurs.

In "`await A || await B`", the parallel bars denotes explicit concurrency. When a parallel statement $p \,||\, q$ starts, it instantaneously starts its branches p and q, thus forking its incoming control thread. Here, both `await` statements start simultaneously and instantaneously when the parallel starts. The threads then evolve in lockstep compared to the environment: in each instant, each of them reacts to the current input event. Here, the processing is trivial: the first branch tests for `A` in each instant and terminates when `A` occurs, while the second one tests for `B` and terminates when `B` occurs. The parallel statement performs branch synchronization by terminating in the precise instant where both branches are terminated. The branches need not terminate at the same time, the parallel waiting for the last one. Therefore, our parallel terminates exactly when both `A` and `B` have been received, be they successive or simultaneous.

The brackets '`[`' and '`]`' are statement parenthesis used to solve syntactic priority conflicts. They are necessary here since the *sequencing* operator '`;`' binds tighter than the parallel operator '`||`'. Sequencing is as usual. In a sequence "$p\,;\,q$", the statement q starts when p terminates. However, in ESTEREL, sequencing is instantaneous: q starts in the instant where p terminates; there is no reason to consume logical time for trivial bookkeeping.

The "`emit O`" statement is instantaneous. It emits the signal `O` and terminates at the time it starts. Therefore, `O` is emitted exactly at the time where the last of `A` and `B` occurs, which is the best meaning we can give to the "as soon as" specification.

The last item is the handling of the reset condition by the *abortion loop* "`loop` p `each R`". The semantics is as follows. When the loop starts, it starts instantaneously its body p. The body runs freely until the next occurrence of `R`. At that time, the body is aborted if still active and it is immediately restarted afresh. If the body terminates before `R` occurs, one simply waits for `R` and restarts the body afresh. Abortion is *strong*: the active body of the "`loop...each`" statement does not receive control at abortion time to react from its current state, which is ignored and discarded. It is only restarted afresh.

Using strong abortion, signal priority is simply realized by statement nesting: the outermost abortion by `R` takes priority over the innermost delays waiting for `A` and `B` that are not watched for when `R` occurs. Therefore, priority is handled by structure, not by artifacts such as explicit numerical indices. Abortion is a strong form of statement *preemption.* A milder form of preemption called *suspension* will be presented in Section 3.8.

The ESTEREL code for `ABCRO` is

```
loop
   [ await A || await B || await C ];
   emit O
each R
```

The code grows linearly with the size of the specification.

3.2 Write Things Once

The `ABRO` example illustrates the fundamental difference between Mealy machines and ESTEREL programs. Esterel statements make it possible to replace *replication* by *structure*. This is the essence of language design: loops, functions, concurrency, objects, etc. all replace explicit code replication by structure. The real key to good programming is the *Write Things Once* or WTO principle: any violation of this principle means a replication that makes the program harder to understand and to maintain, and yields a potential source of bugs: modifying a copy without modifying the other ones. We do not claim that Esterel fully achieves WTO. We only claim that the Esterel primitives help finding the real structure of reactive applications, which is the prerequisite to WTO.

In `ABRO`, each construct contributes in its own way to WTO. Concurrency immediately saves an exponential. Sequencing is fundamental for using a single occurrence of `O` for all the termination cases of the parallel statement. The "`loop...each`" abortion statement makes it possible to preempt the body in any state using a single occurrence of `R`. All these constructs are *orthogonal*. This means that they can be freely mixed at any nesting depth without restriction. Here, the parallel statement appears within a sequence that itself appears within an abortion. Many languages limit concurrency to toplevel, therefore loosing orthogonality. This is a sure way of not achieving WTO.

Communication in Esterel is done by *broadcasting*. Input broadcasting was implicit when we said that concurrent statements evolve in lockstep in the same input environment. Broadcasting is extended to all signals. For examples, processes interested in knowing when `ABRO` emits `O` just wait for `O`. They do not have to signal their identity to `ABRO`, the code of which does not depend on the number of receivers. Broadcasting is essential for WTO in reactive systems programming.

There are other ways to obtain similar results. An important one is the definition of hierarchical automata in Statecharts [35], Argos [42, 32], or SyncCharts [1]. All these formalisms extend Mealy machine in a fully compatible way and use broadcasting, achieving WTO up to some extent. Their compilers are able to re-generate flat machines from structured source code (for Esterel v5, use the `-A` automaton code generation option).

3.3 First Valued Example: Counting

Specification `COUNT`:
Count the number of occurrences of the input `I` *seen so far, and broadcast it as the value of a* `COUNT` *signal at each new* `I`*.*

The Esterel implementation uses a *valued signal* to broadcast the count. In addition to its presence status, a valued signal carries a value, which can be of arbitrary type. The statement "`emit S(`*exp*`)`" emits the signal `S` with value that of the expression *exp*. The status is just as for a pure signal. The value is permanent: at any instant, the expression `?S` yields the current value of `S`, which is either the current value of `S` if `S` has been emitted in the instant or the previous value of `S` otherwise. The expression `pre(?S)` yields the value of `S` at previous instant[1]. An initial value can be given to a signal; it is then specified in its declaration, as shown below. This value is initial for both `?S` and `pre(?S)`.

3.3.1 First Solution, Using pre

The simplest Esterel code for counting is:

```
module COUNT:
input I;
output COUNT := 0 : integer;
every I do
   emit COUNT(pre(?COUNT) + 1)
end every
end module
```

The declaration of `COUNT` specifies that this output signal is of type `integer`, with initial value 0 for `?S` and `pre(?S)`. The `every` statement differs from the `loop...each` statement presented in Section 3.1.3 at initialization time. In "`loop` *p* `each S`", the body *p* is started right away. In "`every S do` *p* `end`", one waits for the first future occurrence of `S` to start *p*, which is what we want here. This could also be written

```
await S;
loop p each S
```

However, using `every` is better since it avoids duplicating `S` and achieves WTO.

The `pre(?COUNT)` expression yields the previous value of `COUNT`. This value is incremented and the result is emitted as the new value of `COUNT`.

Beware: writing "`emit COUNT(?COUNT+1)`" is tempting but incorrect. Since `?COUNT` is the *current* value of `COUNT`, it cannot be incremented and re-emitted right away as itself. It is necessary to use the *previous* value `pre(?COUNT)`. More on this in Section 3.5 and Chapter 5.

[1] New in Esterel v5_91.

Finally, notice that the module and signal have the same name. In Esterel, name spaces are disjoint, and no confusion can occur between object of different kinds such as modules and signals.

3.3.2 Handling the Initial Instant

In "`every S do` *p* `end`", the starting instant is ignored, i.e. an initial `S` does not start *p*. This can be changed by adding the `immediate` keyword. In our counting example, the statement

```
every immediate I do
   emit COUNT(pre(?COUNT));
end every
```

immediately increments `Count` if `I` is present at first instant.

3.3.3 Second Solution, using a variable

Before the introduction of `pre(?S)`, one had to use a local variable to hold the count. This is still possible, writing:

```
module COUNT:
input I;
output COUNT : integer;
var Count := 0 : integer in
   every I do
      Count := Count+1;
      emit COUNT(Count)
   end every
end var
end module
```

The `var` keyword declares a local variable `Count`, of type `integer`, initialized to 0. The variable is incremented when `I` occurs and serves as the value to be emitted. Since name spaces are disjoint, the variable could also be named `COUNT` just as the signal, without any confusion.

For this example, using `pre` is simpler than using a variable. When to use variables and when to use `pre` will be studied in Section 3.5.

3.4 Second Valued Example: Speed Measure

The next example is more on the software side, although it can obviously be implemented in hardware. It is used for bicycles, cars, etc.

> Specification `SPEED`:
> *Count the number of centimeters run per second, and broadcast that number as the value of a* `Speed` *signal every second.*

We assume that centimeters and seconds are received as discrete ticks of signals `Centimeter` and `Second`. The specification is somewhat ambiguous if `Centimeter` and `Second` can be simultaneous. Unlike for `ABRO`, we first neglect this case, assuming that our execution environment (operating system) does not support input event simultaneity, i.e. that it serializes the events it sees. This assumption is very common is event-queue based reactive software design. Then, the code of `Speed` is:

```
module SPEED:
input Centimeter, Second;
relation Centimeter # Second;
output Speed : integer;
loop
   var Distance := 0 : integer in
      abort
         every Centimeter do
            Distance := Distance+1
         end every
      when Second do
         emit Speed(Distance)
      end abort
   end var
end loop
end module
```

In the declaration part, the '`#`' symbol in the `relation` declaration asserts that `Centimeter` and `Second` are *exclusive*, which is the way to assert input serialization.

Let us read the executable body in an outside-in way. The "`loop...end`" statement performs an infinite loop, restarting its own body afresh as soon as it terminates. The `var` keyword declares a local variable called `Distance`, of type `integer`, initialized to 0. One could as well use a `DISTANCE` signal incremented using the `pre(?S)` operator, but a variable is equally simple here since the intermediate distance values need not be broadcast, being of no use in the rest of the module.

The `abort` statement preempts its body when the `Second` signal occurs. An `abort` statement can have a timeout clause following the '`do`' keyword.

Here, the timeout clause emits the speed; in general, it can be an arbitrary statement.

The body of the `abort` statement is an `every` temporal loop that increments the `Distance` variable every centimeter.

Since control transmission and arithmetic operations are instantaneous (i.e. of computation time negligible compared to the input timing), the speed is emitted right away when a second occurs: this is the only way to respect the mathematical definition of the speed, which would be broken by any extra delay. In Esterel, when we write `Speed`, we mean speed!

3.5 Valued Signals Versus Variables

Let us summarize the behavior of variables and signals, which are very different objects.

- A signal is shared throughout its scope, which is the whole program for an interface signal and the scope of its declaration for a local signal, see Section 3.7. A valued signal has one and only one status and one and only one value at a time. Both the status and the value are broadcast. Unlike the status, the value is permanent; if it is unchanged in an instant, the value is that of the previous instant. The writers of a signal are the environment for an input signal and the `emit` statements for an output or local signal. The value can be changed only when the status is present. The readers are the presence or preemption tests for the status, and the *value access* expressions `?S` and `?pre(S)` for the value. Presence tests and value access expressions can occur anywhere in the signal's scope.

- The value of a variable is written by an instantaneous assignment statement[2]. The value is read in expressions when the variable's identifier is mentioned, as usual. Unlike a signal, a variable can take several successive values in an instant. For example, in `SPEED`, the `Distance` variable takes two successive values in the same instant when a centimeter occurs: if its value is n before the assignment, it is $n + 1$ after the assignment. The order in which the values are taken is the internal control-flow order of the program, which we shall call the *constructive* order in Chapter 5.

[2] Or by an instantaneous external procedure call, see Section 4.3.4, or by a non-instantaneous `exec` remote task execution statements, see Section 4.7.17.

> Since Esterel has built-in parallelism, we have to keep variable values consistent between threads. The rule is simple and classical: a variable is local to a thread in case the thread writes it. If the thread forks on a '`||`' parallel statement, then only two cases are legal:
>
> - The variable is accessed in read-only mode in each subthread,
> - If the variable is written by some thread, then it can neither be read nor be written by concurrent threads.
>
> Therefore, it is forbidden to write concurrent assignments such as
>
> ```
>    X:=0;
>    X:= X+1
> ||
>    X:= 1
> ```
>
> There is no way to give decent meaning to such statements in a synchronous deterministic framework.

The sharing law of variables does not apply to signals, and a signal can be emitted from different threads. Furthermore, it is possible and sometimes useful to write simultaneous concurrent emissions such as

```
emit S(1) || emit S(2)
```

This involves using an associative and commutative function to combine the emitted values. The discussion of this feature is deferred to Section 4.4.

As mentioned above in Section 3.3, it is impossible to use a signal's value in its own computation, unlike for a variable. A frequent beginner's mistake is to write

```
emit S(?S+1)
```

to increment the current value of a signal `S`, just as one writes "`X := X+1`". This does not work, since the current value `?S` of `S` is precisely the one emitted by the `emit` statement. The value should satisfy the equation $?\mathtt{S} = ?\mathtt{S} + 1$, which is impossible. This phenomenon is called a *causality cycle*. It is analyzed in Chapter 5. One can either write

```
emit S(pre(?S)+1)
```

or use an auxiliary variable.

We suggest to use the `pre` operator whenever possible, and to introduce a variable in the two following cases:

- The value does not need to be broadcast every time it is changed. This is the case for `Distance` in `SPEED`, which is changed every centimeter but broadcast only every second.

- The value is modified by an external procedure call, as in

  ```
  call Increment(X)()
  ```

  see Section 4.3.4. Signal values or previous values cannot be used in this case. Notice that procedure calls can efficiently update variables in-place, which can be necessary for performance reasons.

3.6 Weak and Immediate Abortion

Let us now accept simultaneity of `Centimeter` and `Second`, which is natural for an hardware implementation and can also be considered useful for a software implementation. For example, in a polling implementation, the module is called at regular time intervals with the inputs buffered since last call. Then, `Centimeter` and `Second` are seen as simultaneous if they are close enough.

What should we do if a centimeter and a second occur simultaneously? The centimeter can be considered as belonging either to the second interval that ends in the current instant or to the one that starts in the current instant, the choice belonging to the specifier. Here, we show how to implement both choices. Beforehand, we need to refine the semantics description of delays and abortion. For "`abort` *p* `when S`", the exact behavior is as follows:

- In the starting instant, *p* is immediately started, the initial presence or absence of `S` being ignored.

- If *p* terminates before `S` occurs, then the whole `abort` statement terminates at the same time.

- If `S` occurs while *p* is not yet terminated, then the `abort` statement immediately terminates and *p* does not receive the control in the current instant.

Because of the first and third clauses respectively, we say that the `abort` statement is *delayed* and *strong*. It is clear that we may need different behaviors in the first and last instant. To make the `abort` statement sensitive to `S` in the first instant, we add the `immediate` keyword in the delay specification:

```
abort
   p
when immediate S
```

Then, an occurrence of `S` in the first instant provokes an immediate termination of the `abort` statement, without *p* being started at all.

To make the body have its "last wills" and receive the control for a last time at abortion time, we add the `weak` keyword:

```
weak abort
   p
when S
```

If *p* is non-instantaneous and if `S` occurs before *p* terminates, then the `weak abort` statement is terminated instantaneously as usual but *p* receives the control for a last time when `S` occurs. Here, the `abort` statement is delayed and an initial `S` is ignored.

If we want to take an initial `S` into account as well, we use both the `immediate` and `weak` keywords:

```
weak abort
   p
when immediate S
```

Then, if `S` occurs in the starting instant, the statement terminates but *p* is executed in the instant. The four possible cases are now covered.

The `every` statement performs strong abortion of its body and it is delayed by default. The statement "`every S do` *p* `end`" is actually not a primitive statement, but an abbreviation for

```
await S;
loop
   abort
      p; halt
   when S
end loop
```

where the `halt` primitive statement waits forever. The loop abbreviates the "`loop`...`each`" statement. The `every` statement also has an immediate form, written

```
every immediate S do p end
```

In the expansion, the initial await statement is made immediate[3].

Back to our SPEED problem, what happens if we simply remove the relation in the initial coding of SPEED? Well, things go wrong. Consider an instant where Centimeter and Second both occur. Since the abortion by Second is strong, the internal every statement is not executed, which implies that Centimeter is not counted in the currently finishing second. The external loop loops and the every statement is instantaneously restarted. Since it is a delayed every, the current centimeter is not taken into account either. Therefore, the centimeter is lost.

Let us now code the two possible consistent specifications. First, assume we want to count the centimeter in the finishing second. We just make the abort statement weak:

```
loop
   var Distance := 0 : integer in
      weak abort
         every Centimeter do
            Distance := Distance+1
         end every
      when Second do
         emit Speed(Distance)
      end abort
   end var
end loop
```

The "every Centimeter" statement is now executed if the two inputs are simultaneous, and the distance is incremented before the speed is emitted. Here, "before" refers to control propagation in threads. When the external loop loops, control reaches the "every Centimeter" statement again, which ignores the current centimeter since it is delayed.

To count the centimeter in the next second, we leave the abort strong and we make the every immediate:

[3] We do not allow weak every, which never proved that useful, but this is a lack of orthogonality that might be corrected some day.

```
loop
   var Distance := 0 : integer in
      abort
         every immediate Centimeter do
            Distance := Distance+1
         end every
      when Second do
         emit Speed(Distance)
      end abort
   end var
end loop
```

Then the centimeter is not counted in the finishing second interval but it is counted in the now starting interval since the immediate `every` statement executes its body.

Finally, notice that making both the `await` statement weak and the `every` statement immediate would lead to count the centimeter *twice*, another possible mistake[4].

Back to the `ABRO` example of Section 3.1, assume now that `R` should not take priority over `A` and `B` any more and that `O` must be emitted even if `R` occurs. This behavior can be coded by replacing the `abort` statement implicit in `loop`...`each` by `weak abort`:

```
loop
   weak abort
      [await A || await B];
      emit O;
      halt
   when R
end loop
```

3.7 Using Submodules: the REGUL Specification

Consider now the following specification of part of a car speed regulation system:

[4] These problems make the author remember the elementary school fencepost problems (*problèmes de poteaux*, in French): if a fence has n poles, how much wire should I buy? It terribly depends on the shape of the fence. Although there is apparently no more open problems in the field, it is still one of the most difficult part of engineering.

> Specification REGUL:
> *In each instant, emit the result of the function* Regfun *applied to the position of the gas pedal and the current speed as the value of the* Regul *signal.*

Of course, the emission of Regul should start only once the first value of Speed has been computed. For REGUL, we additionally require the implementation to reuse the SPEED module.

Here, we mention a function called Regfun. Such an object is external to ESTEREL and written in the *host language* in which the ESTEREL program will be compiled, for example C. The code of REGUL is:

```
module REGUL:
function Regfun (integer, integer) : integer;
input Centimeter, Second;
sensor GasPedal : integer;
relation Centimeter # Second;
output Regul : integer;
signal Speed : integer in
   run SPEED
||
   await Speed;
   sustain Regul(Regfun(?Speed, ?GasPedal))
end signal
end module
```

We first declare the type of the external Regul function, which is necessary for type-checking. We then declare GasPedal to be a *sensor*. A sensor is a valued signal without the presence status part. A gas pedal or a thermometer usually do not send interrupts, they just define numerical values that can be read at any time. For them, the status part is useless (equivalently, the status is always *present*). The value of a sensor is read by the '?' operator, just as for valued signals.

The body is a *local signal declaration* that declares the Speed local signal and whose own body is made of two parallel statements. The first statement is an instantiation of the SPEED module using the run keyword, which amounts to copying its body in place and binding the interface signals of SPEED to the signals bearing the same name in the instantiation scope. The second statement is the sequence of an "await Speed" delay and of a sustain statement that emits the Regul signal with the required difference value in each instant, unlike the emit statement that works only once.

The Speed local signal is generated by the SPEED module. It is instantaneously broadcast to all statements in its scope. Therefore, it is received by the sustain statement at the time where it is generated, which ensures that the value of the Regul signal is always exactly the required one. The "await Speed" delay is necessary to ensure that the value of the Speed signal is well-defined when the value of Regul is computed. Before the first emission of Speed by SPEED, the value ?Speed is undefined.

3.8 Suspension

We now present *suspension*, which is a mild form of instantaneous preemption. In Unix, typing ^C aborts a process, while typing ^Z suspends it; the suspended process is resumed when typing bg or fg. The ESTEREL suspend statement realizes a similar function. However, instead of using a ^Z-fg suspend-resume mechanism, suspension is determined in each instant by the presence of a signal.

Assume the speed regulation system should be suspended when the user keeps the Coast button pressed. This means that the Regul signal should not be emitted if the Coast signal is present. This goal is achieved by replacing the sustain statement by the following construct in the body of REGUL:

```
suspend
   sustain Regul(Regfun(?Speed, ?GasPedal))
when Coast
```

In this simple case, the body of the suspend statement has only one state. In the general case, one can suspend an arbitrary statement in any of its state. The state is frozen until next instant.

Suspension by suspend is delayed, just as abortion by abort: in the starting instant, the presence of the suspension signal is not tested for and the body is run anyway. Suspension can also be made immediate, as in

```
suspend
   sustain Regul(Regfun(?Speed, ?GasPedal))
when immediate Coast
```

The regulation user interface may be different, involving for example a CoastOn-CoastOff button pair with CoastOn # CoastOff, which calls for a suspend-resume way of doing things. In ESTEREL, this is easily done by synthesizing the Coast signal from CoastOn and CoastOff:

```
signal Coast in
   suspend
      sustain Regul(Regfun(?Speed, ?GasPedal))
   when Coast
||
   loop
      await CoastOn;
      abort
         sustain Coast
      when CoastOff
   end loop
end signal
```

3.9 Generic Behaviors and Modules

In the previous coasting example, the loop in the second branch of the parallel statement implements a very common two-states behavior. Therefore, to achieve WTO, it is useful to make a generic module for that behavior. For this, we just build a module with standard names for interface signals:

```
module TWO_STATES:
input On, Off;
output IsOn, IsOff;
loop
   abort
      sustain IsOff
   when On;
   abort
      sustain IsOn
   when Off
end loop
end module
```

Then the module can be instantiated as follows in place of the original loop:

```
signal IsOff in
   run TWO_STATES [ signal CoastOn / On,
                           CoastOff / Off,
                           Coast / IsOn ]
end signal
```

Explicit renaming using the '/' symbol overrides the default signal capture by name. The dummy declaration of the unused signal IsOff is necessary since IsOff is declared in TWO_STATES and must be bound to a signal in the caller. Notice that IsOff is captured by name since it does not appear in the renaming list.

An interesting way to program the ABCRO specification of Section 3.1 is to consider ABRO as a generic behavior to be reused twice in parallel:

```
module ABCRO:
input A, B, C, R;
output O;
signal AB in
   run ABRO [ signal AB / O]
||
   run ABRO [ signal AB / A, C / B]
end signal
end module
```

Notice that R will reset the two submodules simultaneously, thanks to the synchronous model.

The SPEED module should also be made generic, actually in a much broader sense. To measure the speed, it does not matter whether we count centimeters per second, meters per hour, or whatever. It does not matter either whether the speed type is integer, float, or whatever. The type, the initial value, the incrementation function, and the signals should be passed as parameters. In Esterel, the generic code is:

```
module GENERIC_SPEED;
type T;
constant Initial : T,
         Increment : T;
function Add (T, T) : T;
input A, B; % count how many A's per B
output Speed : T;
loop
   var NumberOfA := Initial : T in
      abort
         every immediate A do
            NumberOfA := Add(NumberOfA, Increment)
         end
      when B do
         emit Speed(NumberOfA)
      end abort
   end var
end loop
end module
```

The original SPEED module can be obtained by instantiating GENERIC_SPEED as follows:

```
module SPEED:
input Centimeter, Second;
output Speed : integer;
run GENERIC_SPEED [type integer / T;
                   constant 0 / Initial, 1 / Increment;
                   function + / Add;
                   signal Centimeter / A, Second / B]
end module
```

The run statement is conceptually replaced by the body of GENERIC_SPEED with the appropriate substitutions done. For example, the actual value 0 is substituted to the Initial constant. Renaming arguments are passed by name and not by position as in Lustre or Signal. Each technique has its advantages and drawbacks. Passing by name can be heavier, but passing by position is messy for long argument lists that often occur in practice. Moreover, we can use a very convenient abbreviation: if a name is kept unchanged in a substitution, we just don't mention it. This is what we did for REGUL, see Section 3.7. Finally, it is also possible to give another name to an instantiated submodule, see Section 4.7.16.

3.10 Multiform Time

In Esterel, there is no predefined time unit and `Second` or `Millisecond` act as standard signals. Conversely, any signal can be considered as defining an independent time unit, which gives a broader meaning to timing control issues. Consider the following elementary school reactive problem:

> *You are driving your car at 100 km/h. Suddenly, you see an obstacle on the road at a distance of 50 m. Your reflex time is 1/10 s. Your brakes are able to decelerate the car at 5 m/s*^{2}*. What is your timing constraint?*

The right answer is *50 meter*. This is indeed the only thing that matters to avoid hitting the obstacle. Read the question carefully: it is not "*Will you hit the obstacle*", which is a different much more difficult issue related to program verification. That issue can actually be rephrased as the generic sentence "*Will you meet your timing constraint?*". In Esterel, we want to express timing constraints, not necessarily to check them.

Using meters as "timing constraints" may look unconventional. We consider time as being multiform: the repetition of any signal can be considered as defining its own time measure. Signal periodicity may be of interest for physical modeling, but not for programming. Programming in Esterel mostly consists in understanding what are the time units of a problem and how they relate with each other. In `SPEED`, we measure the speed in the usual unit, centimeters per second. In some applications, it might be better to use the inverse of the speed expressed in seconds per centimeter. It is fundamental to understand that this quantity is computed by the very same program. Passing `Centimeter` for `A` and `Second` for `B` in `GENERIC_SPEED` yields the speed, passing `Second` for `A` and `Centimeter` for `B` yields the inverse of the speed.

3.10.1 The RUNNER example

The next `RUNNER` example makes heavy use of multiform time.

> Specification `RUNNER`:
> *Every morning, go the stadium and do the following for a fixed number of laps: walk for 100 meter, then, during 15 seconds, keep jumping at each step; finish the lap by running full speed.*

The inputs are `Second`, `Morning` (synchronous with `Second`), `Meter`, `Step`, and `Lap` (synchronous with `Meter`). The current action is determined by an output signal chosen among Walk, Jump, and Run. The Walk and Run signals must be continuously emitted, while the Jump signal should only be sent in response to a Step input. The code is as follows:

```
module RUNNER:
constant NumberOfLaps : integer;
input Morning, Second, Meter, Step, Lap;
relation Morning => Second,
         Lap => Meter;
output Walk, Jump, Run;
every Morning do
   repeat NumberOfLaps times
      abort
         abort
            sustain Walk
         when 100 Meter;
         abort
            every Step do
               emit Jump
            end every
         when 15 Second;
         sustain Run
      when Lap
   end repeat
end every
end module
```

The code is self-explanatory and it means exactly what it says. The implication relations written using the '=>' symbol express synchrony of Morning with Second and of Lap with Meter. Notice the following facts:

- If a lap is shorter than 100 meter, the runner will keep walking during that lap; the length of successive laps is not bound to be constant.
- If a lap is shorter than (i.e. occurs before) 100 meter plus 15 seconds, then the runner will never run during that lap.

provided of course that mornings do not occur too often compared to meters, seconds, and laps.

The same program written in an asynchronous language would have a quite different meaning. For example, there could be in principle any physical delay between the time a lap is ended and the Lap signal is actually perceived by the program. This is why asynchronous languages cannot handle reactive specifications.

3.11 Traps and Exception Handling

Let us add an extra specification item to REGUL: if the speed gets bigger than a constant MaxSpeed, the behavior should be stopped at once and an alarm should be emitted. This is programmed using the trap construct:

```
trap SpeedTooHigh in
   signal Speed : integer in
      run SPEED
   ||
      await Speed;
      sustain Regul(Regfun(?Speed, ?GasPedal))
   ||
      every Speed do
         if ?Speed > MaxSpeed then
            exit SpeedTooHigh
         end
      end every
   end signal
handle SpeedToHigh do
   emit Alarm
end trap
```

The if statement instantaneously tests its Boolean condition. If the condition is true, the then clause is instantaneously executed. Here, it is an exit statement. When such a statement is executed, control is instantaneously transferred to the corresponding handler if there is one, the trap simply terminating if there is no handler. At exit time, all statements in the trap body are weakly aborted, as for a weak abort statement (weak abort is actually a macro-statement built using traps).

Similarly, for the runner, the jumping phase is pretty strenuous. During that phase, one should monitor the heart and rush to hospital if there is any problem. The program structure is

```
every Morning do
   trap HeartAttack in
      repeat NumberOfLaps times
         abort
            abort
               sustain Walk
            when 100 Meter;
            abort
               every Step do
                  emit Jump
               end every
            ||
               <MonitorHeart>
            when 15 Second;
            sustain Run
         each Lap
      end repeat
   handle HeartAttack do
      <RushToHospital>
   end trap
end every
```

We leave it to the reader to fill in the details, i.e. to write the program fragments `<MonitorHeart>` that contains the "`exit HeartAttack`" statement and `<RushToHospital>`. In the way we wrote the program, the runner may have to rush to the hospital every morning. Swapping "`every Morning`" and "`trap HeartAttack`" would yield a different behavior: at first heart attack, the runner would abandon running for good.

3.12 Boolean Signal Expressions and the Present Test

So far, we have handled signals only by using preemption constructs. It is also useful to perform instantaneous presence tests of signals using the dedicated `present` statement. We give three examples of such tests.

Consider first the `TWO_STATES` module of Section 3.9. In this module, we clearly start in the off state. In practice, it is also useful to be able to start in the on state. One way of doing that is to invert on and off states at module instantiation time, writing for example

```
signal IsOn in
   run TWO_STATES [ signal CoastOff / On,
                           CoastOn / Off,
                           Coast / IsOff ]
end signal
```

This is quite clever, but probably a little bit too clever and somewhat misleading. A more pedestrian way is to use another signal meaningful only at module start time to determine in which state to start:

```
module TWO_STATES:
input StartOn;
input On, Off;
output IsOn, isOff;
loop
   present StartOn else
      abort
         sustain IsOff
      when On;
   end present;
   abort
      sustain IsOn
   when Off
end loop
end module
```

If `StartOn` is present, we skip the off state and start directly in the on state. Notice that the `then` clause of the `present` test is omitted; if `StartOn` is present, the `present` statement simply terminates, otherwise the `else` branch is immediately started. If one finds the omission of `then` misleading, one can write the test as follows:

```
present [not StartOn] then ...
```

To use the module with initial state the on state, just write

```
signal StartOn in
   emit StartOn;
   run TWO_STATES [...]
end signal
```

Remove the emit statement to start in off mode.

The second example illustrates the role of signal synchrony in program architecture. In the wristwatch program described in [8], the timekeeper broadcasts the time using a Time signal. The alarm manages the AlarmTime signal to know when to beep. Beeping should occur when both times are equal, *unless the current time is not a proper time but is a time being set by the user.* To distinguish between the two cases, the timekeeper sends a pure signal WatchBeingSet synchronously with Time whenever it is in set-time mode. The alarm tests for the absence of this signal when receiving the time:

```
every Time do
   present WatchBeingSet else
      if Equal(?Time, ?AlarmTime) then
         emit StartBeeping
      end if
   end present
end every
```

This control-based solution is much cleaner than polluting the Time type with an extra field telling whether the time is actual or set.

The third example illustrates Boolean signal expressions and case statements in tests. It is statement decoding in a microprocessor. Bus wires Bit0 and Bit1 carry the opcode: 11 for Load, 10 for Store, and 01 for Noop, the code 00 being forbidden. The decoding is written as follows, assuming there is an enclosing trap WrongOpCode

```
present
   case [Bit0 and Bit1] do
      emit Load
   case [Bit0 and not Bit1] do
      emit Store
   case [not Bit0 and Bit1] % Noop
   else
      exit WrongOpCode
end present
```

For Noop, the empty do clause is simply omitted.

Boolean signal expressions can be used in any temporal statement, abort, every, etc.

Chapter 4

A Tour of Esterel

4.1 Lexical Aspects

Lexical aspects are classical:

- Identifiers are sequences of letters, digits, and the underline character '_', starting with a letter.

- Integers are as in any language, e.g. `123`. and floating-point numerical constants are as in C++ and Java; the values `12.3`, `.123E2`, and `1.23E1` are constants of type `double`, while `12.3F`, `.123E2F`, and `1.23E1F` are constants of type `float`.

- Strings are written between double quotes, e.g., `"a string"`, with doubled double quotes as in `"a "" double quote"`.

- Keywords are reserved and cannot be used as identifiers. Many constructs are bracketed, like "`present` ... `end present`". For such constructs, repeating the initial keyword is optional; one can also write "`present` ... `end`".

- Simple comments start with '%' and end at end-of-line. Multiple-line comments start with '%{' and end with '}%'.

4.2 Modules

The Esterel programming unit is the module. A module has a name, an interface declaration part, and a body, which is an executable statement:

```
module  name :
 interface declaration
 statement
end module
```

A module can use submodules that are instantiated by the `run` statement. Instantiation cannot be recursive. An Esterel program is specified by a collection of modules and a designated module. All modules transitively referred to in the main module must be defined in the collection.

The Esterel v5 compiler translates an Esterel program into a program or circuit written in a *host language* that is chosen by the user, for example C.

The interface declaration part specifies which objects a module imports or exports. This is essential both for Esterel type-checking and for host language type-checking. There are two kinds of interface objects:

- Data objects, which are declared abstractly in Esterel. Their actual value is supposed to be given in the host language and linked to the Esterel compiled code in a way that depends on the compiler and target language.

- Signals and sensors, which are the primary objects the Esterel program deals with. Which host objects correspond to signal and sensors depends on the host language and code generation type.

The data and signal declarations can be mixed in an arbitrary way, provided that any item is declared before being used. The scope of interface objects is the whole module. Here are complete examples of module interface declarations, the components of which are explained below:

```
module WATCH :

input UL, UR, LL, LR;       % the four watch buttons
relation UL # UR # LL # LR; % they are incompatible

input S, HS;                % second and 1/100 second
relation S => HS;           % no S without an HS

type Time;
constant Noon : Time;
function CompareTime (Time, Time) : boolean;
procedure IncrementTime (Time) (integer);
```

```
output CurrentTime := Noon : Time;

type Beep;
constant WatchBeep : Beep, AlarmBeep : Beep;
function CombineBeeps (Beep, Beep) : Beep;
output Beeper : combine Beep with CombineBeeps;


module ROBOT:

type Coord, Rectangle;

function MakeRectangle (Coord, Coord) : Rectangle;
function InRectangle (Coord, Rectangle) : boolean;
procedure TranslateAndRotate (Rectangle) (Coord, integer);

task MoveRobotInsideRectangle (Coord) (Rectangle);
return RobotInRectangle;


module MISC :
constant WordLength = 16 : integer;
sensor Temperature : float;
inputoutput BusRequest;
output YesVotes := 0 : combine integer with +;
```

4.3 Data

Data objects are divided between primitive and user-defined objects. Since data handling is not a primary concern in control-dominated reactive programming, we kept the data definition facilities minimal, heavily relying on the host language capabilities. All data objects are global to the program. Each data object used within a module must be declared in that module. For a multi-module program, an data object declared in several submodules must be identically declared in all of them, see Section 4.7.16.

4.3.1 Types and Operators

There are only five primitive types in Esterel: `boolean`, `integer`, `float`, `double`, and `string`. The Boolean constants are `true` and `false`. The numerical and string constants were described in Section 4.1.

The operations are the usual ones. Equality is written `=` and difference is written `<>` for all types. The `boolean` type is equipped with `and`, `or`, and `not`, and the operations `+`, `-`, `*`, `/`, `<`, `<=`, `>`, and `>=` are available for `integer`, `float`, and `double`.

There is no implicit type conversion. In particular, the user must call explicitly declared external functions to convert integers to floats or floats to doubles and conversely. Remember that "`1 + 3.14`" and "`3.14 + 2.718F`" do not typecheck.

The user can define his own types by declaring their names. For Esterel, a user type is a completely abstract object. Its actual definition will be given only in the host language. Since there is no data-structuring mechanism in Esterel, explicit user types must be constructed for records, arrays, etc, with appropriate access functions and procedures. This is certainly heavy, but highly portable.

Equality '`=`' and unequality '`<>`' can be used for user types. If they are used, they must be adequately defined in the host language.

Here are the type declarations of the above module examples:

```
type Time;
type Beep;
type Coord, Rectangle;
```

4.3.2 Constants

Constants of any type can be declared as follows:

```
constant Noon : Time;
constant WatchBeep : Beep, AlarmBeep : Beep;
constant WordLength = 16 : integer;
```

There are two ways to declare a constant. In the implicit way, only the name and the type are given, see `Noon` above. The value is defined in the host language. In the explicit way, the name, the type, and the value are declared, see `WordLength` above. This is possible only for constants of predefined types.

4.3.3 Functions

Functions take a list of objects of arbitrary types and return a single object of arbitrary type:

```
function CompareTime (Time, Time) : boolean;
function CombineBeeps (Beep, Beep) : Beep;
function MakeRectangle (Coord, Coord) : Rectangle;
function InRectangle (Coord, Rectangle) : boolean;
```

Functions are defined in the host language. They are called in data expressions, and they must be side-effect free.

4.3.4 Procedures

Procedures have two lists of arguments of arbitrary types:

```
procedure IncrementTime (Time) (integer);
procedure TranslateAndRotate (Rectangle) (Coord, integer);
```

The first list is the list of reference arguments that are passed by reference and possibly modified by the call. The second list is that of value arguments that are passed by value and not modified. For example, in `TranslateAndRotate`, the rectangle is passed by reference and modified when translated and rotated, while the translation and rotation arguments are passed by value. Each of the lists can be empty. Procedures are called by the `call` statement, which is assumed to be instantaneous.

Procedures are defined in the host language. Their code should be side-effect free besides the obvious side-effect on reference arguments (see Section 5.2.6 for the evaluation ordering of procedure calls).

4.3.5 Tasks

Tasks are external computation entities syntactically similar to procedures but whose execution is assumed to be non-instantaneous. They are declared exactly as procedures:

```
task MoveRobotInsideRectangle (Coord) (Rectangle);
```

Tasks are executed by the `exec` statement and coupled with `return` signal as described in Section 4.7.17. Tasks are supposed to run concurrently with the Esterel program. The way this is implemented depends on the compiler, on the host language, and on the run-time system. The actual code of tasks is given in the host language.

4.4 Signals and Sensors

Signals and sensors are the logical objects received and emitted by the program or used for internal bookkeeping. Signals can be interface signals declared in the module interface or local signals declared by the `signal` local signal declaration statement, see Section 4.4.5 and Section 4.7.13 .

Signals are instantaneously broadcast throughout the program, which implies that all statements see each of them in a consistent way. Pure signals have a presence status, *present* or *absent*. In addition to their status which is as for pure signals, valued signals carry a value of arbitrary type. For single valued signals, only one statement can emit the signal in an instant. For combined valued signals, multiple emitters are allowed. Sensors have a value but no status. The broadcast value of a valued signal or of a sensor is unique in each instant. One can also access the previous status and value of a signal or sensor by using the `pre` operators.

There is one predefined signal, the special pure signal `tick` that represents the activation clock of the reactive program. Its status is *present* in each instant. The `tick` signal is declared implicitly and cannot be redeclared.

4.4.1 Interface Signal Declarations

Interface signals are either `input`, `output`, `inputoutput`, or `return`. The `return` signals are used to signal termination of external tasks, see Section 4.7.17. Here are the interface signals declarations of the above modules:

```
input UL, UR, LL, LR;       % the four watch buttons
input S, HS;                % second and 1/100 second
output CurrentTime := Noon : Time;
output Beeper : combine Beep with CombineBeeps;

return RobotInRectangle;

inputoutput BusRequest;
output YesVotes := 0 : combine integer with +;
```

Here, `UL`, `UR`, `LL`, `LR`, `S`, and `HS` are pure input signals. The signal `BusRequest` is a pure inputoutput signal. It is both received by the module and emitted by it.

The `CurrentTime` output signal is a valued signal of abstract time `Time`. The value of `CurrentTime` is accessed through the expression `?CurrentTime`, see Section 4.6.2. It is initialized to Noon. Similarly, the YesVotes signal is integer-valued with initial value 0. If no initial value is given, as for Beeper, the value is undefined until the first time the signal is received from the environment or emitted by the program itself. If an initial value is given to a signal S, it also serves as the initial value of the expression pre(?S).

The return signal RobotInRectangle is a special input signal used for signaling external task completion, see Section 4.3.5. A return signal can be valued just as a standard input signal.

4.4.2 Single and Combined Valued Signals

The above CurrentTime signal is called a single signal: it cannot be emitted twice in the same instant and it cannot be emitted by the program in an instant if it is received from the environment in that instant. This restriction holds for any valued signal not declared using the combine keyword. The signals declared with that keyword are called combined signals.

In the above declarations, Beeper and YesVotes are combined. For them, the values simultaneously emitted by several emitters or received from the environment are gathered and combined using the specified binary function or operator that must be commutative and associative. For Beeper, the Beep type can represent a set of sound frequencies and combining several sounds by CombineBeeps can be taking the union of their frequencies. In this way, one can hear the timekeeper, alarm, and stopwatch beep together. For YesVotes, using addition as a combination function makes it easy to count simultaneous yes votes if each participant broadcasts the number of voices he or she represents.

For the type boolean, the combination function can be and or or. For the types integer, float, and double, the combination function can be + or *. Any other combination function must be user-defined and declared prior to the signal declaration. The corresponding host language combination function is assumed to be commutative and associative, which obviously cannot be checked by Esterel.

4.4.3 Sensors

Sensors are valued input signals without presence information. A sensor is declared by giving its name and type:

```
sensor Temperature : integer;
```

Sensors differ from signals in the way they are interfaced with the environment. The value of a sensor is read by the program whenever needed. Therefore, the notion of an initial value is meaningless for sensors.

4.4.4 Input Relations

Input relations declare some Boolean condition about input or return signals that are assumed to be guaranteed by the environment. In the above WATCH example, the relations are:

```
relation UL # UR # LL # LR;
relation S => HS;
```

The first relation is called an incompatibility (or exclusion) relation. It asserts that the four input buttons are incompatible (or exclusive), i.e. that no two of them can be simultaneously present in the environment. The second relation is called an implication relation. It asserts that S can be present only if HS is, i.e. that a second is always synchronous with a 1/100 second.

Relations are useful to avoid specifying irrelevant behaviors. For example, in the WATCH module, the exclusion relation asserts that the user cannot simultaneously request to change to set-watch mode and to stopwatch mode; in practice, buttons are serialized by the low-level event handler. Relations are also useful to optimize automaton code generation, for circuit code optimization, and to speed-up program verification.

For the Esterel v5 compiler, only the relations of a program's main module matter. The relations declared in the submodules are discarded.

4.4.5 Local Signal Declaration

A local signal declaration is performed by the following construct:

```
signal Alarm,
       Distance : integer,
       Beep := OneBeep : combine Beep with CombineBeeps
in
   p
end signal
```

where p is any statement. A local signal declaration is an executable statement, and it can be placed wherever a statement can.

The individual declarations are the same as for interface signals, see Section 4.4.1. Types must be declared separately for each signal in a signal declaration list. Here, `Alarm` is a pure signal.

The scope of a local signal declaration is the body *p*. Scoping is lexical: any re-declaration of a signal hides the outer declaration.

Local signals are subject to reincarnation: a local signal placed within a loop can be executed several times in the same instant. Then, each execution declares a new copy or incarnation of the signal, see Section 4.7.13. Signal handling can also yield causality problems studied in Chapter 5. Section 4.7.13 studies the local signal declaration statement in more details, including issues about taking the `pre` of a local signal.

4.5 Variables

Variables are assignable objects that have a name and a type. Variables are declared by the local variable declaration construct, which has the form

```
var X : double,
    Count := ? Distance : integer,
    Deadline : Time
in
   p
end var
```

where *p* is any statement. A variable declaration is an executable statement, and it can be placed wherever a statement can.

A variable declaration declares the names of the variables, their types, and possibly their initial values. The scope of a variable declaration is the body *p*. Scoping is lexical: any re-declaration of a variable hides the outer declaration. The type must be declared individually for each variable. The declaration

```
var X, Y : integer in
```

is incorrect since `X` has no type (there is no "pure variable"). One must write

```
var X : integer, Y : integer in
```

A variable is modified by assignments, see Section 4.7.2, procedure calls, see Section 4.7.2, and the `exec` external task execution statement, see Section 4.7.17. An initial value can be assigned at declaration time, as for `Count` above. If no initial value is given, the variable's value is undefined until the first assignment is performed.

Unlike a signal, a variable can take several successive values in the same instant. For example, in the statement

```
X := 0;
emit S1(X);
X:= X+1;
emit S2(X)
```

the signals S1 and S2 are emitted simultaneously with respective values 0 and 1, the variable X taking these values in succession within the instant. This poses absolutely no problem in the constructive semantics of Esterel presented in Chapter 5, provided of course that variables cannot be shared in read-write mode between threads. More precisely, if a variable is written in a thread, then it can be neither read nor written in any concurrent thread.

4.6 Expressions

There are three kinds of expressions in Esterel: data expressions, signal expressions, and delay expressions.

4.6.1 Data Expressions

Data expressions are built as usual by combining basic objects using operators and function calls. Their evaluation is instantaneous. All expressions must type-check.

Constants and variables appear under their names. The current value of a valued signal or sensor S is written ?S. The previous value of a signal S is written pre(?S). There is currently no previous value operator for a sensor (this is due to the way sensors are interfaced in the current compiler: a sensor is read only in call-by-need; there is no way to know in the current instant if we have to read the sensor for the next instant, and it would be too expensive to read the sensor at any instant). Traps can carry values just as signals, see Section 4.7.14. However, the name space of traps is distinct from that of signals, and we must use a different symbol to access trap values. The current value of a valued trap T is written ??T. There is no ?pre operator for trap values. Accessing a yet undefined signal or trap value is an error. Function calls are written as usual. Here are some expressions: Delay

```
X * WordLength
FloatToInteger(?Temperature) * (??ExitCode+5) / pre(?Age)
```

4.6.2 Signal Expressions

Signal expressions are Boolean expressions over signal statuses. They are used in instantaneous `present` tests or in delay expressions. Signal expressions are obtained by combining signal names or the `tick` predefined signal using the `and`, `or`, `not`, and `pre` operators, viewing *present* as *true* and *absent* as *false*. In a signal expression, `S` means the current status of `S` and `pre(S)` the previous status of `S`, i.e. its status at previous instant. The binding conventions are the usual ones: `not` binds tighter than `and`, which binds tighter than `or`. Any signal can appear in a signal expression, including an output signal. Here are examples:

```
Meter and not Second
Bit1 and Bit2 and not (Bit3 or Bit4)
pre(Meter)
I and not pre(J)
I and pre(I or J)
```

The only restriction is that `pre` operators cannot be nested[1].

The interpretation of signal expressions is obvious, except for the `pre` operator. We define a *first instant of a signal* `S` as follows:

- If `S` is a program interface signal, the only first instant of `S` is the first instant of the program execution.

- If `S` is a local signal, a first instant of `S` is any instant where the local signal declaration that declares `S` is entered (local signal declarations are detailed in Section 4.7.13).

The signal expression `pre(S)` has value *false* in any first instant of `S`, following our convention that a signal is absent by default: a signal is also considered absent before it exists. Then, in subsequent instants, `pre(S)` is true if `S` was present in the previous instant.

To define `pre`(e) where e is a signal expression, we need two auxiliary semantical operators: pre_0(`S`), which is the previous status of `S` with initial value *absent* just as for `pre(S)`, and pre_1(`S`), which only differs by having

[1] This restriction may be lifted in subsequent versions if it turns out to be too stringent.

value *present* in any first instant of `S`[2]. Let $\neg 0 = 1$ and $\neg 1 = 0$. Then $\texttt{pre}(e)$ is defined by structural induction on e:

$$
\begin{array}{rcl}
\texttt{pre}(e) & = & \mathit{pre}_0(e) \\
\mathit{pre}_i(e \ \texttt{and}\ e') & = & \mathit{pre}_i(e)\ \texttt{and}\ \mathit{pre}_i(e') \\
\mathit{pre}_i(e \ \texttt{or}\ e') & = & \mathit{pre}_i(e)\ \texttt{or}\ \mathit{pre}_i(e') \\
\mathit{pre}_i(\texttt{not}\ e) & = & \texttt{not}\ \mathit{pre}_{\neg i}(e)
\end{array}
$$

The pre_1 operator needs not be primitive in Esterel since its effect can be obtained using two negations: `not(pre(not S))` is `pre` initialized to *present*. In any first instant of `S`, the initial value *absent* is negated by the outer `not` operator and become *present*. In all subsequent instants, the outer negation is canceled since the previous value was negated by the inner `not` operator.

Since `tick` is always true, the expression “`not tick`” is always false.

When dealing with complex expressions, some aspects related to reincarnation and constructive causality have to be well-understood, see Section 4.7.13 and Chapter 5 for details.

4.6.3 Delay Expressions

Delay expressions are used in temporal statements such as `await` or `abort`. There are three forms of delay expressions: standard delays, immediate delays, and count delays. A delay starts when the temporal statement that bears it starts, and it elapses in some later instant, possibly in the same instant for immediate delays.

Standard delays are defined by a signal expression. Standard delays never elapse instantaneously. For example, the standard delay

```
Meter and not Second
```

elapses in the *next* instant in which a meter occurs without a simultaneous second.

Immediate delays can elapse instantaneously. They are defined by the **`immediate`** keyword followed by a signal expression, which must appear within brackets ‘[ ]’ unless it is a single identifier. For example, the immediate delay

```
immediate [Meter and not Second]
```

[2] In Lustre [33], one would write `false->pre(S)` for $\mathit{pre}_0(\texttt{S})$ and `true->pre(S)` for $\mathit{pre}_1(\texttt{S})$.

elapses instantaneously if a meter and no second are present when the delay is initiated, and it behaves as a standard delay otherwise. Notice that there is only one layer of brackets '[]' and that standard parentheses are used inside delay expressions, as for `Meter and (not Second)`.

Count delays are defined by an integer count expression followed by a signal expression.

The signal expression must be bracketed using square brackets '[]' if it is not reduced to a single signal. Here are examples:

```
3 Second
3 pre(Second)
5*X Meter
3 [Second and not Meter]
```

The expression is evaluated only once when the delay is initiated. If the expression's value is 0 or less, it is set to 1. Therefore, a count delay never elapses instantaneously. This is fundamental for various kinds of static analysis including constructiveness analysis, see Chapter 5 and Section 4.7.6.

Notice two restrictions on delays: there is no immediate count delay, and counts cannot be intertwined with Boolean signal operators. This is a deliberate choice. Consider an immediate delay expression of the form

```
await immediate N Second
```

Then, if `N` is greater than 0, the statement cannot terminate instantaneously, while it can if `N` is 0. Appropriate information about possible instantaneous termination cannot be extracted from this kind of statement, which is why we forbid it[3]. Also, we think that expressions such as

```
immediate [ 3 [n Seconds or p [Meter and not Second]]]
```

are too difficult to understand unambiguously.

4.7 Statements

We describe here all statements except local signal declaration described in Section 4.4.5 and local variable declaration described in Section 4.5. All constructs but sequencing '`;`' and concurrency '`||`' use bracketing keywords: `abort`—`end abort`, etc. Repetition of the initial keyword is optional, so `abort`—`end` is also correct. To resolve the remaining syntactic ambiguities,

[3] We could use a `positive` clause as for `repeat`, see Section 4.7.6, but this has not been done yet.

any statement can be explicitly bracketed using square brackets '[]'. Esterel is fully orthogonal: statements can be freely mixed in an arbitrary way. One can sequence parallel statements or put sequences in parallel, one can subject any statement to an abortion, etc.

In the sequel, we do not add indentation for parallel statements. Therefore,

```
signal S in
   p
||
   q
end signal
```

is preferred to the more indented form

```
signal S in
      p
   ||
      q
end signal
```

4.7.1 Basic Control Statements

There are three basic pure control statements:

```
nothing
pause
halt
```

The `nothing` statement terminates instantaneously when started. The `pause` statement pauses for one instant. More precisely, it pauses when started, and it terminates in the next instant (`pause` can also be written "`await tick`", see Section 4.7.9). The `halt` statement pauses forever and never terminates.

4.7.2 Assignment and Procedure Call

See Section 4.5 for the definition of a variable. Assignments have the form

```
X := e
```

where `X` is a variable and e is a data expression. The variable and expression must have the same type. Assignments are instantaneous.

Procedure calls have the form

```
call P (X, Y) (e1, e2)
```

where `X` and `Y` are variables and the e_i are expressions. The types must match those of the declaration. The variables are modified by the call. The call is instantaneous.

4.7.3 Signal Emission

Instantaneous signal emission is realized by the `emit` statement, which has one of two forms:

```
emit S
emit S(e)
```

For a pure signal `S`, the `emit` statement simply emits `S` and terminates instantaneously. For a valued signal, the `emit S(`e`)` statement evaluates the data expression e, emits `S` with that value, and terminates instantaneously. Input signals can be internally emitted by the program, but return signals cannot be emitted.

For a single signal, if an `emit` signal is executed, it must be the only one in the instant; for an input or inputoutput single signal, no `emit` statement can be executed if the signal is received in the input event. Both requirements are statically checked by the Esterel v5 compiler using the `-Icheck` option. For a combined signal, the emitted value is combined with those emitted by other `emit` statements executed in the instant using the combination function. For an input or inputoutput combined signal that is received from the environment and locally emitted in the same instant, the received and emitted valued are combined.

Continuous emission of a signal is realized by the `sustain` statement:

```
sustain S
sustain S(e)
```

When started, the `sustain` remains active forever and it emits `S` in each instant. For a valued signal, the data expression e is re-evaluated in each instant. The "`sustain S`" statement abbreviates "`loop emit S each tick`", see Section 4.7.11.

4.7.4 Sequencing

Sequencing is done using the '`;`' sequence operator. In

p ; q

the first statement p is instantaneously started when the sequence is started, and it is executed up to completion or trap exit. If p terminates, q is immediately started and the sequence behaves as q from then on. If p exits some enclosing traps, the exits are immediately propagated and q is never started, see Section 4.7.14. For example, "`exit T; emit S`" does not emit `S`.

4.7.5 Looping

A simple loop has the following form:

```
loop
   p
end loop
```

The body p is instantaneously restarted afresh upon termination, this forever. If p exits some enclosing traps, the exits are propagated instantaneously and the loop stops. This is the only way to exit a loop from inside. Of course, a loop can also be killed by an external preemption statement, see Section 4.7.10 and Section 4.7.14.

The body of a loop is not allowed to be able to terminate instantaneously when started. This condition is static: there must be no potential direct path from start to termination in p, even if that path cannot be taken dynamically. For instance, the following loop is rejected:

```
loop
   present I then
      present J else
         p
      end present
   else
      q
   end present;
end loop
```

Even if p and q have delays, there is a potential instantaneous path in the loop body corresponding to the case where `I` and `J` are both present. If `I` and `J` are inputs declared incompatible by the input relation `I # J`, then the instantaneous path is a false one since it cannot be taken in any valid input configuration. The program is rejected nevertheless. One must add an extra clause involving a delay:

```
loop
   present I then
      present J then
         pause % unreachable since I # J
               % but ensures static non-termination
      else
         p
      end present
   else
      q
   end present
end loop
```

4.7.6 Repeat Loops

A `repeat` loop executes its body for a finite number of times. The body is not allowed to terminate instantaneously. The simplest form is

```
repeat e times
   p
end repeat
```

The expression e must be of type `integer`. It is evaluated only once at starting time. The body is not executed at all if e evaluate to 0 or to a negative number. Therefore, the `repeat` statement is considered as possibly instantaneous and it cannot be put in a loop if not preceded or followed by a delay, this even if its own body is non-instantaneous. Therefore, the following statement is rejected as being a potentially instantaneous loop, independently of the body '`...`':

```
loop
   repeat 3 times
      ...
   end repeat
end loop
```

Esterel compilers are not required to perform static analysis and discover that 3 is never null, because one can replace 3 by user-defined constants or complex expressions. To solve this problem, we ask the user to assert that the body will be executed at least once by adding the `positive` keyword:

```
loop
   positive repeat 3 times
      ...
   end repeat
end loop
```

In the `positive repeat` statement, the test for repetition is performed only after the first execution of the body. The body is not allowed to be able to terminate instantaneously, and the whole `positive repeat` statement inherits the same property.

4.7.7 The present Signal Test

The `present` statement branches according to the instantaneous values of signal expressions. The simplest form checks for one signal expression and performs binary branching. Each of the `then` and `else` branches can be omitted, but at least one of them must be specified. An omitted branch is implicitly `nothing`:

```
present S then p else q end present
present [Second and Meter] then p end present
present Meter then q end
present pre(Meter) else q end
```

The `case` form tests several signal expressions in sequence:

```
present
   case Meter do
      Distance := Distance+1;
      emit Distance(Distance)
   case Second do
      emit Speed(Distance)
end present
```

The tests are taken in order, and the first true expression starts immediately its `do` clause. If the `do` clause is omitted, the `present` statement simply terminates. If none of the expressions is true, the `present` statement terminates. One can add an `else` statement for that case:

```
present
   case [Bit0 and Bit1] do
      emit Load
   case [Bit0 and not Bit1] do
```

```
      emit Store
   case [not Bit0 and Bit1] % no-op
   else
      exit WrongOpCode
end present
```

4.7.8 The if Data Test

The `if` statement is used to test Boolean data expressions. In the basic binary form, either the `then` or the `else` clause can be omitted, as for the `present` statement:

```
if X>=0 then p else q end if
if X=Y and Y<>?Z then p end
if ?Flag else q end
```

Multiple cases can be checked in sequence using the `elsif` keyword, the `case` keyword being reserved for signal expressions:

```
if X > 5 then
   p
elsif X > 3 then
   q
else
   r
end if
```

The conditions are evaluated in sequence. The first true condition triggers the corresponding statement. If no condition is true, then the `if` statement executes the `else` statement if there is one and terminates otherwise.

4.7.9 The await Statement

The `await` statement is the simplest temporal statement. In its basic form, it simply waits for a delay:

```
await Second
await immediate Second
await immediate [Second and Meter]
await pre(Second)
await immediate pre(Second)
await 2 Second
await 2 tick
await 2 [Second and not pre(Meter)]
```

The delay is started when the await statement is started. The statement pauses until the delay elapses and terminates in that instant. An immediate await statement terminates instantaneously if the signal expression is true in the starting instant. Be careful: the sequence

```
await immediate Meter;
await immediate Meter
```

terminates instantaneously if Meter is present in the starting instant (this is why making immediate the default would be misleading).

A do clause can be used to start another statement when the delay elapses:

```
await 2 Second do
   emit Beep
end await
```

This is simply an abbreviation for "await 2 Second; emit Beep", but it makes the dependency of Beep on 2 Second more explicit. As for present, one can introduce a case list:

```
await
   case 2 Second do p
   case immediate Meter
   case Button do q
end await
```

The above statement immediately terminates if Meter occurs at start time. Otherwise, the first delay to elapse determines the subsequent behavior: p is started if 2 Second elapses first, the await statement simply terminates if Meter occurs first, and q is started if Button occurs first. If several delays elapse at the same time, the first one in the list takes priority. For example, if Meter and Button occur simultaneously, then the await statement terminates and q is not started.

Unlike for present, no else clause is allowed for an await statement. One can use "case tick" to achieve the same effect.

4.7.10 The abort Statements

An abortion statement kill its body when a delay elapses. For strong abortion, performed by abort, the body does not receive the control at abortion time. For weak abortion, performed by weak abort, the body receives the control for a last time at abortion time. The syntax is as follows:

```
abort p when 3 Meter
weak abort p when 3 Meter
```

For both constructs, the body p is run until termination or until the delay elapses. If p terminates before the delay elapses, so do the `abort` and `weak abort` statements. Otherwise, p is preempted when the delay elapses; in that instant, p is not executed with strong abortion, and it is executed for a last time with weak abortion (p has rights to its "last wills").

If the delay is immediate and elapses immediately at starting time, the body is not executed at all with strong abortion, and it is executed for one instant with weak abortion For example, in

```
abort
   sustain O
when immediate I
```

the `abort` statement terminates immediately without emitting `O` if `I` is present at starting time. If `abort` is replaced by `weak abort`, the whole statement also terminates instantaneously but `O` is emitted once.

As for `await`, one can add a `do` clause to execute a statement q in case of delay elapsing:

```
abort % or weak abort
   p
when 3 Meter do
   q
end abort
```

With both weak and strong abortion, q is executed if and only if p did not terminate strictly before delay elapsing. At abortion time, with strong abortion, p is not executed and q is immediately started. With weak abortion, the first instant of q is done in sequence after the last instant of p.

As for `await`, one can introduce an ordered list of abortion cases:

```
abort % or weak abort
   p
when
   case Alarm do r
   case 3 Second do q
   case immediate Meter
end abort
```

Here, *p* is immediately aborted if there is a `Meter` at starting time. Otherwise, *p* is run for at least one instant. The elapsing of any of the three delays aborts *p*. If there is a `do` clause for the delay, that statement is immediately started; otherwise, the `abort` statement simply terminates. If more than one of the delays elapses at abortion time, then the first one in the list takes priority as for the `await` statement.

Nesting `abort` statements also builds priorities. In the statement

```
abort
   abort
      p
   when I do
      q
   end abort
when J
```

the signal `J` takes priority over `I` if they occur simultaneously, and *q* is not started in that case. This is no special rule, but just a consequence of the strong abortion semantics of `abort`.

Finally, notice that "`await S`" can be defined as "`abort halt when S`".

4.7.11 Temporal Loops

Temporal loops are loops over strong abortion statements. The first form is

```
loop
   p
each d
```

where *d* is a non-immediate delay. At starting time, the body *p* is started right away, and it is restarted afresh whenever the delay *d* elapses. If *p* terminates before *d* elapses, then one waits for the elapsing of *d* to restart *p*. The "`loop each`" statement is simply an abbreviation for

```
loop
   abort
      p; halt
   when d
end loop
```

The delay cannot be immediate, otherwise the loop body would be instantaneous.

The second temporal loop has the form

```
every d do
   p
end
```

The difference is that d is initially waited for before starting the body p. The delay d can be immediate. In that case, in the starting instant, p starts immediately if the delay elapses immediately. The statement

```
every 3 Second do
   p
end every
```

abbreviates

```
await 3 Second;
loop
   p
each 3 Second
```

The statement

```
every immediate Centimeter do
   p
end
```

abbreviates

```
await immediate Centimeter;
loop
   p
each Centimeter
```

All temporal loops are infinite. The only way to terminate them is by exiting a trap, see Section 4.7.14 or by the elapsing of an enclosing abortion delay.

4.7.12 The suspend Statement

Abortion violently preempts a statement and kills it, in the same way as ˆC kills a process in Unix. Suspension has a milder action, like ˆZ in Unix. The basic syntax is

```
suspend
   p
when s
```

where s is any signal expression. When the `suspend` statement starts, p is immediately started. Then, in each instant, the following occurs:

- If the signal expression s is true, then p remains in its current state and the `suspend` statement pauses for the instant.
- If the signal expression s is false, then p is executed for the instant. If p terminates or exits a trap, so does the `suspend` statement. If p pauses, so does the `suspend` statement, and suspension is re-examined in the next instant.

Here is an example. The statement

```
suspend
   abort
      sustain O
   when J
when I
```

emits `O` in the first instant and in all subsequent instants where `I` is absent, until the first instant where `I` is absent and `J` present. Then the `suspend` statement terminates and `O` is not emitted.

The default `suspend` statement is delayed, in the sense that the signal expression is not tested for in the first instant. The immediate form performs that test:

```
suspend
   p
when immediate s
```

Here p is not started in the first instant if s is true. The immediate form can be rewritten as follows:

```
await immediate [ not s ];
suspend
   p
when s
```

4.7.13 Local Signal Declaration

In Section 4.4.5, we have seen the form of a local signal declaration and the way in which signals are scoped. We now present some important issues about local signals. We start by a macro-expansion definition of `pre`. Then, we study the interaction between `suspend` and `pre`. Finally, we study the *reincarnation* phenomenon.

The pre Operators for Local Signals

The `pre` status operator and the `pre(?s)` value operator need not be primitive in Esterel. For a pure local signal, one could directly define a signal `preS` that behaves just as `pre(S)`, by writing

```
trap T in
   signal S, preS in
      p;
      exit T
   ||
      loop
         present S then
            pause;
            emit preS
         else
            pause
         end
      end loop
   end signal
end trap
```

Within *p*, any occurrence of `pre(S)` can be changed into `preS`. The trap T serves in propagating termination of *p*. The loop in the second parallel branch computes the status of `pre(S)` using two `pause` statements; this code is less efficient than the direct implementation available in Esterel v5_91 since it uses two registers instead of one, except when using automaton code generation where it yields exactly the same automaton.

The same can be done for an interface signal, putting the `pre` code at toplevel (no trap is necessary in this case).

Similarly, the `pre(?S)` previous value operator need not be primitive, since `pre(?S)` could be defined as the value `?preS` of an auxiliary signal `preS`:

```
trap T in
   signal S : Type, preS : Type in
      p;
      exit T
   ||
      loop
         present S then
            var preval := ?S in
               pause;
               emit preS(preval);
            end var
         else
            pause
         end
      end loop
   end signal
end trap
```

If S has an initial value, then we specified in Section 4.4 that pre(?S) has the same initial value. In this case, we should write

```
signal S := v : Type, preS := v : T in ...
```

Local Signals and Suspension

Suspension interacts with taking the pre operators for a signal declared within the suspension body, as in

```
suspend
   signal S in
       ...
       present pre(S) then ...
       ...
   end signal
when I
```

The expression pre(S) refers to the status of S *in the previous instant where the* signal *statement was activated*, not to the status of S in the previous absolute instant or tick. Instants where I suspends the signal statement do not count for pre. In some sense, the suspend statement steals the tick to its body. This is obvious when expanding the pre operators as explained in Section 4.7.13.

Reincarnation

Because of instantaneous looping of loops, local signals can have several simultaneous instances that we call reincarnations. They pose no particular problem, but one has to be aware of their existence, in particular to understand causality issues. Here is an example:

```
loop
   signal S in
      present S then emit O1 else emit O2 end;
      pause;
      emit S
   end signal
end loop
```

In the first instant, the local signal S is declared. It is absent since there is no emitter for it. Therefore, the else branch of the present statement is taken and O2 is emitted. In the second instant, the pause statement terminates and S is emitted and set present. The loop body terminates and it is restarted afresh right away. The local signal declaration is immediately re-entered. It declares a *fresh* signal, distinct from the old one, whose status is lost. The fresh incarnation is absent, unlike the old one. The present statement tests the fresh incarnation and only O2 is emitted. Everything happens as if the loop body was duplicated:

```
loop
   signal S in
      present S else emit O end;
      pause;
      emit S
   end signal;
   signal S in
      present S else emit O end;
      pause;
      emit S
   end signal;
end loop
```

In this obviously equivalent statement, the old and fresh incarnations are split into two syntactically distinct signals that happen to bear the same name S and the present statement is duplicated. In the original form, the single S generates two distinct dynamic incarnations, and the present statement dynamically tests the current incarnation of the signal.

The `pre` and `pre(?S)` operators always refer to the current incarnation. For example, in

```
loop
   signal S in
      present pre(S) then emit O1 else emit O2 end;
      pause;
      emit S
   end signal
end loop
```

the `O2` signal is continously emitted. The `S` emitted at the end of the loop body is not matched by `pre`, which matches the new incarnation, with `pre(S)` initially absent, as specified in Section 4.4 and Section 4.4.5.

4.7.14 Traps

A trap defines an exit point for its body. The basic syntax is

```
trap T in
   p
end trap
```

The scope of `T` is the body *p* and scoping is lexical. A redeclaration of a trap hides the outer declaration.

The body *p* is immediately started when the `trap` statement starts. Its execution continues up to termination or trap exit, which is provoked by executing the "`exit T`" statement. If the body terminates, so does the `trap` statement. If the body exits the trap `T`, then the `trap` statement immediately terminates, weakly aborting *p*.

The `weak abort` statement can be defined using traps. The construct "`weak abort` *p* `when S`" is an abbreviation for

```
trap T in
   p;
   exit T
||
   await S;
   exit T
end trap
```

Nested Traps

When traps are nested, the outer one takes priority. Consider for example

```
trap U in
   trap T in
      p
   end trap;
   q
end trap;
r
```

If p exits `T`, then q is immediately started. If p exits `U`, then r is immediately started. If p exits simultaneously `T` and `U`, for example by executing "`exit T || exit U`", then `U` takes priority and only r is executed. From the point of view of the "`trap T`" statement, `T` is discarded and `U` is propagated.

Trap Handlers

A handler can be used to handle a trap exit, with the following syntax:

```
trap T in
   p
handle T do
   q
end trap
```

If p terminates, so does the trap statement. If p exits `T`, then p is weakly aborted and q is immediately started in sequence.

Concurrent Traps

Several traps can be declared using a single `trap` keyword:

```
trap T, U, V
   p
handle T do
   q
handle U do
   r
end trap
```

In this case, the traps are called concurrent traps and they must have distinct names. Concurrent traps are at the same priority level, and any of them can have a handler. If several traps are simultaneously exited, then the corresponding handlers are executed in parallel:

Here, *q* and *r* are executed in parallel if *p* exits `T` and `U` simultaneously. Since they are concurrent, the *q* and *r* handlers cannot share variables. The `trap` statement simply terminates if *p* exits `V` that has no handler.

Here is the translation of "`weak abort` *p* `when S do` *q* `end`" using concurrent traps:

```
trap Terminate, WeakAbort in
   p;
   exit Terminate
||
   await S;
   exit WeakAbort
handle WeakAbort do
   q
end trap
```

Valued Traps

Traps can be valued exactly as signals, except that `pre(?S)` is not available for traps (it would make no sense since an statement that exits a trap dies instantaneously). Value initialization and combined traps are allowed. This is useful to pass a value to the handler. The value is obtained as the result of the expression '`??S`', which is allowed only in the handler:

```
trap Alarm : combine integer with + in
   ... exit Alarm(3) ...
   ... exit Alarm(5) ...
handle Alarm do
   emit Report(??Alarm)
end trap
```

Of course, concurrent traps can be valued:

```
trap T, U := 0 : integer, V : combine integer with + in
   p
handle T do
   q
handle U and V do
   emit O(??U + ??V)
end trap
```

Here, the second handler starts if and only if `U` and `V` are exited in parallel, in which case `O` is emitted with values the sum of the values of `U` and `V`. Beware of uninitialized trap values, which could occur if `and` was replaced by `or` in the second handler above.

4.7.15 The Parallel Statement

The parallel operator puts statements in synchronous parallel. The signals emitted by any of its branches or by the rest of the program are instantaneously broadcast to all branches in each instant.

A parallel can be binary, as in p `||` q, ternary, as in p `||` q `||` r, or of any arity. Syntactically, the sequencing operator '`;`' binds tighter than the parallel operator '`||`'. Therefore, p`;` q `||` r means `[`p`;` q`]` `||` r, which is different from p`;` `[`q `||` r`]` where the brackets are mandatory.

A parallel statement forks its incoming thread when it starts, starting instantaneously one thread per branch. The parallel terminates when all its branches have terminated, waiting for the last one if some branches terminate earlier. The parallel propagates a trap `T` as soon as one of its branches exits `T`, weakly aborting all its branches at that time. See Section 4.7.14 for the case where several traps are simultaneously exited.

Variables can only be shared among parallel branches if they are read-only. If a branch can write a variable `X`, then no other branch can read or write `X`. Signals are the only truly shared objects.

4.7.16 The run Module Instantiation Statement

A module can be instantiated within another module using the `run` executable statement. In the simplest form, one simply writes

```
run SPEED
```

This amounts to syntactically replace the `run` statement by the body of the `SPEED` module. Recursive or mutually recursive submodule instantiation is forbidden.

All data objects (types, functions, procedures, tasks) are global to an Esterel program. Therefore, the data declarations of the instantiated submodule are exported to the parent module. If some data objects were already declared in the parent, the parent and child declarations must be the same.

The signal interface declarations of the instantiated module are simply discarded, as well as the relation declarations. This means that the interface signals of the instantiated submodule must exist in the parent module with the same type. Notice that a signal declared as input in the submodule is seen as global after instantiation. For instance, in

```
module M :
input I;
emit I
end module

module N :
output I;
run M
end module
```

the signal I is effectively emitted by `N` although it was declared as an input in `M`. This is an anomaly that should be corrected some day.

Any interface object can be renamed at module instantiation time using the following renaming syntax:

```
run GENERIC_SPEED [ type integer / T;
                    constant 0 / Initial,
                             1 / Increment;
                    function + / Add;
                    signal CarSpeed / Speed ]
```

A renaming `X / Y` is read "`X` renames `Y`". The renaming object `X` can be either an explicit constant or operator or an identifier. If it is an identifier, it must be declared in the parent module. The renamed object `Y` must be an identifier belonging to the data or signal interface of the instantiated module. The kinds and types must match.

Full renaming makes it possible to build generic modules. Partial renaming is also possible. In that case, any submodule interface object that is not renamed is captured by the parent object of the same name (and kind: a type is captured by a type, a signal by a signal, etc.).

The included module itself can be renamed:

```
   run CarSpeed / SPEED [...]
||
   run BicycleSpeed / SPEED [...]
```

This is useful for identifying submodule occurrences in symbolic debuggers.

4.7.17 The exec Task Execution Statement

External procedure calls performed using the `call` statement are supposed to be instantaneous. This does not fit with many practical applications where procedure computing times cannot be neglected. The `task-exec` mechanism we now describe makes it possible to control execution of external tasks that take time.

Roughly speaking, tasks behave as procedures that are executed asynchronously with the ESTEREL program. At the ESTEREL abstraction level, we take a logical view of tasks. We care about controlling them, and we do not care about how they are actually executed in the environment concurrently with the ESTEREL program. The only thing we are interested in is when external tasks start, when they terminate in ESTEREL sense, and when they should be suspended or aborted by other ESTEREL statements.

Tasks are not limited to computationally intensive ones. They can also be of a more physical nature. For instance, in Robotics, a task may be "grasp this object".

Tasks are declared in the data interface part of a module, see Section 4.3.

The exec Statement and the Return Signals

The statement that executes a task is the `exec` statement. It has the form

```
exec TASK (reference-params) (value-params) return R
```

where `R` is called the return signal. A return signal is a special input signal declared using the `return` keyword instead of the `input` keyword in the module signal interface:

```
return R1;
return R2 : integer;
return R3 : combine FOO with F;
```

Notice that a return signal can have a value as any input signal. Return signals can also appear in exclusion or implication relations together with input signals, see Section 4.4.4. Like any other input signal, a return signal can be tested for presence or awaited concurrently with the task execution. The use of this feature will be explained in Section 4.7.17. Unlike a standard input, a return signal cannot be internally emitted by the program.

External Task Execution

When an "`exec T return R`" statement starts, it signals to its environment that a fresh instance of the task `T` should start with parameters passed by reference and value just as for procedures. The signaling is instantaneous. The Esterel program does not wait for the task and continues reacting autonomously. More precisely, the thread that has started the `exec` statements waits for task completion, but the other threads continue reacting to inputs. In some instant in the strict future of the starting instant, the environment signals back task completion to the Esterel program by sending the return signal `R`. Within the Esterel program, receiving `R` provokes instantaneous update of reference arguments according to the values returned by the task and instantaneous termination of the `exec` statement.

During its execution, an `exec` statement can be suspended or aborted. This is signaled to the external task by sending appropriate suspension and abortion signals.

The task launching and signaling implementation mechanism entirely depends on the compiler and run-time system. The only implementation constraint is to respect the Esterel logical view.

Uniqueness of Return Signals

One may have several `exec` statements for a given task `T`; therefore, one may also have different concurrent instances of the same task in the environment. The return signal is used to tell the Esterel program which instance has terminated. For this to be possible, return signals must uniquely identify `exec` statements. Hence, we impose the following restriction:

> *No two* `exec` *statements in a program can have the same return signal.*

This condition must be verified after submodule expansion. Uniqueness of return signals may call for explicit renaming at submodule instantiation time:

```
module OneTask :
task TASK (integer) (integer);
return R;
var X := 0 : integer in
   exec TASK(X)(1) return R
end var
end module
```

```
module TwoTasks :
return R1, R2;
   run Task1 / OneTask [signal R1 / R]
||
   run Task2 / OneTask [signal R2 / R]
end module
```

Abortion of exec Statements

As any other Esterel statement, an `exec` statement is subject to abortion by `abort`, `weak abort`, or `trap` statements and to suspension by `suspend` statements. The simplest case of abortion is the weak one. Consider the example:

```
weak abort
   exec TASK (X) (1) return R;
when I
```

In the first instant, the task is started. Then, the behavior is as follows:

- If `R` occurs before `I` or if `R` and `I` occur simultaneously, then `X` is updated and the whole `weak abort` statement terminates.
- If `I` occurs before `R`, then execution of `TASK` is aborted and the external task is aborted. There is no update of `X`.

Strong abortion is a little bit more delicate. Consider the example:

```
abort
   exec TASK (X) (1) return R;
when I
```

After starting the task in the first instant, the behavior is as follows:

- If `R` occurs before `I`, then `X` is updated and the whole `abort` statement terminates.
- If `I` occurs before `R`, then execution of `TASK` is aborted and the external task is aborted. There is no update of `X`.
- If `I` and `R` occur simultaneously, then the `abort` statement terminates. Although the task did terminate, `X` is not updated since the body of the `abort` statement does not receive control. No abort signal is sent to the task either since it is terminated.

Notice the subtle difference between weak abortion by `weak abort` or `exit` and strong abortion by `abort` in the case where `R` and `I` are simultaneous: with strong abortion, update of reference variables is not performed, while it is performed with weak abortion.

Suspension of exec Statements

Consider a program fragment of the form

```
suspend
   exec TASK (X) () return R
when S
```

When `S` occurs after the starting instant, the `exec` statement is suspended. This is signaled to the environment by sending an implementation-dependent suspension signal. The signal is sent in every instant where the `exec` statement is suspended.

Termination of the `exec` statement can occur only when that statement is active. Assume that `R` and `S` occur simultaneously. Then, `R` does not provoke termination of the `exec` statement and its occurrence is lost. It is the environment's responsibility to sustain `R` until the `exec` statement is not suspended any more, which is easy using the abort and suspend signaling mechanism.

If needed, it is easy for the execution environment to convert the suspension information available in each instant from the Esterel program into a suspend-resume information that may be handier for operating systems. See the Esterel v5 task execution library for examples.

Testing for the Return Signal

When an `exec` statement is strongly aborted, one may need to know if the external task did terminate in the instant. This is easy using a `present` test on the return signal:

```
abort
   exec TASK (X) (1) return R
when I do
   present R then ... else ... end present
end abort
```

The same can be done for suspension

```
      suspend
         exec TASK (X) () return R
      when S
   ||
      await R do ... end await
```

Multiple exec

The multiple `exec` statement makes it possible to control several tasks simultaneously. It resembles the "`await...case`" statement:

```
   exec
      case T1 (...) (...) return R1 do p1
      case T2 (...) (...) return R2 do p2
      ...
      case Tn (...) (...) return Rn do pn
   end exec
```

Reference variables can be shared between the cases. As for the multiple await statement, "`do` *pi*" can be omitted if *pi* is just `nothing`.

When a multiple `exec` statement starts, all tasks are started simultaneously and concurrently. Then, one waits for the return signals. When at least one return signal occurs, the `exec` statement terminates instantaneously; in that instant, all non-terminated tasks are aborted, *only one reference argument update is performed*, the one corresponding to the *first terminated case* in the case list, and only the corresponding `do` continuation is taken. In case of abortion, all tasks are aborted simultaneously. In case of suspension, all tasks are suspended simultaneously.

A typical use of the multiple `exec` statement is to try several ways to perform a given computation in parallel, stopping when the first computation is done:

```
   exec
      case InvertMethod1 (Matrix) () return R1
      case InvertMethod2 (Matrix) () return R2
      case InvertMethod3 (Matrix) () return R3
   end exec
```

All necessary bookkeeping is nicely performed by the ESTEREL compiler.

Immediate Restart of an exec Statement

An exec statement may be aborted and restarted immediately. Consider for instance

```
loop
   exec TASK (X) (1) return R;
each I
```

If I occurs before task completion, the Esterel program signals to the environment that the current instance of TASK should be aborted and that a fresh instance should be started right away.

A slightly more difficult situation appears in the following somewhat artificial program fragment borrowed from [9]:

```
loop
   trap T1 in
      loop
         trap T2 in
            exec TASK (X) (1) return R
         ||
            await I do exit T2 end
         end trap
      end loop
   ||
      await I do exit T1 end
   end trap
end loop
```

In the first instant, a fresh instance of TASK is started. Then, if I occurs before R, the following happens instantaneously: the inner trap T2 is exited and an abort information is sent to the environment to abort the running instance of TASK; the inner loop loops, and another TASK is restarted immediately; however, the outer trap T1 is also exited, which implies that this new instance of TASK is aborted right away; since the "trap T1" statement terminates, the outer loop loops, and yet another instance of TASK is started, this time in a for-real way.

In such an intricate behavior, the intermediate launching of TASK by the inner loop does not provoke any signaling to the environment, the task being simply considered as stillborn by Esterel. Only the aborting of the current instance and the starting of the last instance are signaled.

Because of this special handling of stillborn tasks, we can guarantee the following property:

In any instant, at most two instances of a task launched by a given `exec` *statement can be active. The only possibility to have two instances active at the same time is when an already active and not yet terminated instance is aborted, while a fresh instance is started. In* ESTEREL*, this means that the* `exec` *statement is aborted and is instantaneously restarted.*

Chapter 5

Constructive Causality

The availability of instantaneous broadcasting and control transmission makes it possible to write syntactically correct but semantically non-sensical programs. The constructive semantics mathematically described in [9] characterizes sensible ESTEREL programs. It is the reference semantics of the language. In this chapter, we briefly present constructive correctness in terms of the intuitive operational semantics of ESTEREL programs, referring to [9] for the mathematical definition. Most of the examples already appeared in [9] and we keep the same names for them here. We also discuss the acyclicity condition that automatically guarantees constructiveness and is very easy to check at compile-time, unlike proper constructiveness.

Users should remember that many causality problems can now be solved by adding `pre` status or value operators at the right place, which makes causality issues much simpler than in previous versions.

5.1 Cyclic and Acyclic Programs

5.1.1 Non-Reactive and Non-Deterministic Programs

In our class of application, reactivity and determinism are the minimal requirements a program should obey. A program is *reactive* if it provides a well-defined output for each input. A program is *deterministic* if it produces only one output for each input.

Here is the simplest example of a non-reactive program:

```
module P3:
output O;
present O else emit O end
end module
```

Broadcasting means that a signal is present if and only if it is emitted. Here, O cannot be present, otherwise it would not be emitted and therefore absent; it cannot be absent either, otherwise it would be emitted and therefore present.

Here is the simplest example of a reactive non-deterministic program:

```
module P4:
output O;
present O then emit O end
end module
```

Here, O present can be seen as valid since it is justified by the emission of O, and O absent can also be seen as valid because it implies non-emission of O.

5.1.2 Signal Dependency Cycles

Both examples involve an instantaneous dependency cycle between O and itself. Similar examples can be constructed from dependency cycles between two signals O1 and O2. Here is one:

```
module P5:
output O1, O2;
   present O1 then emit O2 end
||
   present O2 else emit O1 end
end module
```

Dependency cycles can easily be constructed for signal values. Consider again the wrong COUNT example of Section 3.3:

```
module BAD_COUNT:
input I;
output COUNT := 0 : integer;
every I do
   emit COUNT(?COUNT+1)
end every
end module
```

The programmer's intention is clear: emit COUNT(n) at the n.-th occurrence of I. However, the program makes no sense. Let us call c the value of COUNT. By definition of broadcasting, c must satisfy the equation $c = c + 1$, which is impossible. The right way to write this program is to write "emit COUNT(pre(?COUNT) + 1)" or to use an auxiliary variable, as explained in Section 3.3.

5.1.3 Acyclic Programs

The problems we mentioned are generally called *causality problems*. They resemble deadlocks in asynchronous languages, and they are indeed "instantaneous deadlocks". To avoid them, most synchronous languages require signal dependency to be *acyclic*. In this case, it is obvious that any signal has one and only one status and one and only one value, i.e. that programs are reactive and deterministic.

Acyclicity is very easy to check and it is also quite natural in dataflow programs or in electronic circuits. Often, a program is cyclic instead of acyclic simply because some `pre` operator is missing, `pre` and `pre(?S)` cutting all instantaneous cycles as do all delay operators in all formalisms. Considering `P3` above, the programmer' intention was probably to toggle between presence and absence of `O`. Turning `P3` into an acyclic program is easy using `pre`:

```
module P3OK:
output O;
present pre(O) else emit O end
end module
```

Similarly, the `BAD_COUNT` program above is made correct by changing `?COUNT` into `pre(?COUNT)`; If you encounter an unintentional cycle, first check that you did not forget a `pre` or a `pre(?S)`.

5.1.4 Correct Cyclic Programs

However, acyclicity has been considered by Esterel users as being too restrictive a condition. Consider the following programs:

```
module P13:
input I;
output O1, O2;
present I then
   present O2 then emit O1 end
else
   present O1 then emit O2 end
end present
end module
```

```
module P14:
output O1, O2;
present O1 then emit O2 end;
pause;
present O2 then emit O1 end
end module
```

In both P13 and P14, there is a static cyclic dependency between O1 and O2. However, for both programs, it is clear that the cycle is a false one and that everything goes well at run-time. In P13, only one branch of the test can be taken at a time, according to the externally defined status of I. In P14, the dependency from O1 to O2 is valid in the first instant only while the reciprocal dependency is valid in the second instant only. The imperative syntax of Esterel makes the correctness of P13 and P14 obvious, which would not be true of their data-flow counterparts. The more practical example of a cyclic symmetrical bus arbiter will be presented in Section 5.2.7.

5.2 Constructiveness in Esterel

5.2.1 Logical Correctness

At first glance, it appears natural to simply require programs to be reactive and deterministic. This is what we call *logical correctness*. Logical correctness fits reasonably well with data-flow languages [34], but not with the imperative style of Esterel. Consider the following example:

```
module P9:
output O1, O2;
   present O1 then emit O1 end
||
   present [O1 and not O2] then emit O2 end
end module
```

Surprisingly enough, P9 is logically correct, with unique behavior O1 and O2 absent. Indeed, this hypothesis *self justifies*: O1 absent implies non-emission of O1 and non-emission of O2, which is consistent with the assumption. We leave it to the reader to check that no other hypothesis is consistent. The problem is that self-justification does not fit with the standard control propagation intuition of imperative language, where the evaluation of a test should *precede* the evaluation of its branches, at least in a causal sense.

Another interesting example is the Esterel analogue of the Boolean equation “O = O and not O”:

```
module P12:
output O;
present O then emit O else emit O end
end module
```

Here, `O` present is justified by `O` emitted and `O` absent is not justified since `O` would be emitted. Once more, the program is logically correct by self-justification, the flow of control going *backwards* from the `then` part to the test.

5.2.2 Constructiveness

The idea of the constructive semantics is to forbid self-justification and any kind of speculative reasoning, replacing them by pedestrian fact-to-fact propagation. Ignore values and signal expressions for a while, concentrating on pure signal tests of the form "`present S`". We use a three-valued logic for signals, where the status of a signal is *present*, *absent*, or *unknown*. In each instant, the statuses of the input signals are given by the environment and the statuses of the other signals are initially set to *unknown*. The only inferences we can perform are as follows:

1. An unknown signal can be set present if it is emitted.

2. An unknown signal can be set absent if no emitter can emit it.

3. The `then` branch of a test can be executed if the test is executed and the signal is present.

4. The `else` branch of a test can be executed if the test is executed and the signal is absent.

5. The `then` branch of a test cannot be executed if the signal is absent.

6. The `else` branch of a test cannot be executed if the signal is present.

The rules forbid speculative execution, since (3) and (4) can be applied only if it is already known that the `present` statement must be executed. The rules allow us to prune false paths, since (5) and (6) can be applied anywhere, see example `P2` below.The precise mathematical rules are given in [9].

We say that a program is *constructive* if the status of each local or output signal can be determined using these rules; it is then determined in a unique way. Let us work through an acyclic example:

```
module P1:
input I;
output O;
signal S1, S2 in
   present I then emit S1 end
||
   present S1 else emit S2 end
||
   present S2 then emit O end
end signal
```

We start with status *unknown* for S1, S2, and O. Assume I is present. Then, the first test takes its then branch and emits S1, which sets S1 present. The second test can proceed by terminating, which implies that S2 cannot be emitted since its only emitter has been discarded. Therefore, S2 can be set absent. Finally, the third present statement can proceed and terminate. Since the "emit O" statement is discarded, O can be set absent. Conversely, assume I absent. Then S1 cannot be emitted and is set absent, which triggers emission of S2, which itself triggers emission of O. Notice that present tests are locked until the status of the signal they test becomes known.

Well-behaved cyclic programs are handled without much difficulty. For example, in P13 above, if I is present, then the emitter of O1 is discarded by rule (6), O1 is set absent, the first present test terminates and discards "emit O2", and O2 is set absent. Here is a more sophisticated example:

```
module P2:
output O;
signal S in
   emit S;
   present O then
      present S then
         pause
      end present;
      emit O
   end present
end signal
end module
```

In P2, S is emitted and set present before the control reaches the test for O. Execution cannot proceed since the status of O is unknown. However, we can perform false path pruning using rule (6) and infer that the implicit else branch of the "`present S`" statement cannot be executed. Therefore, the "`emit O`" statement cannot be reached instantaneously because the `then` branch of the "`present S`" cannot terminate instantaneously, which implies that `O` can be set absent since it has no other emitter.

Logically incorrect programs are easily rejected. For example, in `P3` or `P4`, there is no way to make any progress from the unknown state. Logically correct programs that require self-justification or speculative computation are rejected as well. For example, in `P12`, the two "`emit O`" statements can neither be executed nor be discarded from the initial *unknown* status of `O`[1].

When a signal has several simultaneous incarnations, each of them must be handled independently. In practice, it is sufficient to reset the status to *unknown* when entering the signal declaration.

5.2.3 Constructiveness and Preemption

Preemption statements are easily handled, noticing that they behave just as tests for the guard in each instant where the guard is active. For example, the following program is not constructive:

```
module P3bis:
output O;
abort
   sustain O
when O
```

In the first instant, the guard is inactive and `O` is emitted. In the second instant, the guard becomes active, and it must be tested *before* the body is executed, in the constructive order. The body is neither found to be executed nor to be discarded, and the program is non-constructive. In the second instant, the program's body just behaves as

```
module P3ter:
present O else
   sustain O
end present
```

[1] In [54, 55], we prove that an electronic circuit that implements the Boolean equation "`O = O or not O`" indeed behaves in a constructive way rather than in a logical one. For some wire and gate delays, the output voltage won't stabilize. We show that constructiveness is the logical counterpart of delay-independent electrical stabilization, which gives strong physical roots to the constructive semantics.

which is a variant of P3. If abort is replaced by weak abort, the problem disappears, since in the second instant the statement behaves as

```
module P3bisWeak :
trap T in
   present O then exit T end
||
   sustain O
end trap
```

which is obviously constructive, emits O, and terminates.

5.2.4 Constructiveness of Signal Expressions

Signal expressions are evaluated as follows in the constructive semantics: "not e" evaluates to *false* if e evaluates to *true* and conversely; "e_1 or e_2" evaluates to *true* as soon as one of e_1 or e_2 evaluates to *true*, even if the other one is still unknown; it evaluates to *false* if both e_1 and e_2 evaluate to *false*. The evaluation of "e_1 and e_2" is dual. Notice that the evaluation is parallel: the evaluation of an expression does not require the evaluation of all its subexpressions.

5.2.5 Constructiveness for Valued Signals

Consider now a valued signal S. The most general case is that of a combined signal with combination function F. Since each emitter can contribute to a part of the final combined value, that value is known only when all emitters are either executed or discarded. Unlike the computation of the status that succeeds as soon as one emitter emits, the computation of the value cannot be lazy.

A reader of the value is an expression '?S'. The expression must lock the control until the value is defined. All data operators are strict, i.e. must evaluate all their arguments before giving their result. This is an important difference between pure and valued signals. Let X and Y be two Boolean-valued signals. For the status test

```
present [ X or Y ] then p else q end
```

the statement p is executed as soon as one of X or Y is emitted. For the value test

```
if ?X or ?Y then p else q end
```

the statement p is executed if one of `?X` or `?Y` is true, but only after the status of both `X` and `Y` is known[2].

Value handling combines nicely with status handling. A statement such as "`emit S(2)`" should be thought of as a sequence "`emit S; ?S:=2`" (this for a single signal; one should invoke the combination function for a combined signal). Consider the following toy example:

```
module OK :
output O1 : integer, O2: integer;
   emit O1(?O2)
||
   present O1 then emit O2(1) end
end module
```

This program is constructively correct, and both `O1` and `O2` are emitted with value 1. The constructive reasoning is as follows: The "`emit O1`" statement in the first branch is executed, hence, `O1` is present, but its value is still unknown. Since `O1` is present, the `then` branch of the `present` statement is executed, and `O2` is emitted with value 1. From then on, the value `?O2` becomes readable, and the value of `O1` is determined to be 1.

5.2.6 Constructiveness and Side-Effects

The constructive analysis determines the ordering in which statements will be executed for each state and each input. This ordering also determines in which order side-effecting host-language procedures are called. Consider for example the program

```
signal S1, S2 in
   present I then emit S1 else emit S2
||
   present S1 then
      call P1 () ();
      emit S2
   end present
||
   present S2 then
      call P2 () ();
```

[2] In the Esterel v5 C translator, the Boolean `or` value operator is implemented by the C '`|`' operator that evaluates both its arguments. It is *impossible* to define a disjunction operator that returns 1 as soon as one of its arguments is 1 in any sequential language such as C, see [12]. Statuses are handled in a very different way.

```
            emit S1
         end present
      end signal
```

In this example, it is guaranteed that P1 is called before P2 if I is present and that P2 is called before P1 if I is absent. For example, with I present, we must emit S1 and call P1 before emitting S2 that provokes the call to P2.

Generally speaking, any control dependency between calls is respected: in "call P1()(); call P2()()", the call to P1 always precedes the call to P2. Concurrent calls can be ordered by signal dependencies, as in the above example. If there is no control dependency and no signal dependency, as in "call P1()() || call P2()", the order is unspecified and it would be an error to rely on it.

5.2.7 Constructiveness vs. Acyclicity

Although compile-time constructiveness analysis is available, acyclic programs should be preferred whenever possible, since their compilation is much faster and generally more efficient. However, we mentioned that cyclic programs can be more natural. Let us show the example of a symmetric bus arbitration mechanism[3].

The bus is a ring on which a bunch of identical stations are hooked. In each instant, the user of the bus can request the bus and he can obtain it or not. A priority mechanism arbitrates simultaneous requests. A token defines the current initial station. At any time, the bus is granted to the first station that asks for it, starting from the initial station in clockwise order. To obtain fairness, the token is moved to the next station in each instant, so that each station is the initial one in turn.

[3] Thanks to R. de Simone for the example.

The ESTEREL code of one station is

```
module STATION :
input Request;          % from user
output Granted;         % to user
input PreviousPassed;   % from previous station
output Pass;            % to next station
input Token;            % from previous station
output PassToken;       % to next station
   loop
      present [ Token or PreviousPassed ] then
         present Request then
            emit Granted
         else
            emit Pass
         end present
      end present
   each tick
||
   loop
      present Token then
         await tick;
         emit PassToken
      else
         await tick
      end present
   end loop
end module
```

A bus with three stations is programmed in Figure 5.1.

The `Pass1`, `Pass2`, and `Pass3` signals form a static dependency cycle. At any time, the cycle is dynamically cut at the station that possesses the token. This is easily found by the constructive reasoning, that figures out in which order things must be done in each state. However, there is no *uniform* order in which to do things.

```
module BUS :
input Request1, Request2, Request3;
output Granted1, Granted2, Granted3;
signal Pass1, Pass2, Pass3,
       Token1, Token2, Token3
in
   emit Token1
||
   run Station1 /
          STATION [ signal Request1 / Request,
                           Granted1 / Granted,
                           Pass3 / PreviousPassed,
                           Pass1 / Pass,
                           Token1 / Token,
                           Token2 / PassToken ]
||
   run Station1 /
          STATION [ signal Request2 / Request,
                           Granted2 / Granted,
                           Pass1 / PreviousPassed,
                           Pass2 / Pass,
                           Token2 / Token,
                           Token3 / PassToken ]
||
   run Station1 /
          STATION [ signal Request3 / Request,
                           Granted3 / Granted,
                           Pass2 / PreviousPassed,
                           Pass3 / Pass,
                           Token3 / Token,
                           Token1 / PassToken ]
end signal
end module
```

Figure 5.1: The 3-stations `BUS` program

Finally, notice that dependencies can be somewhat hidden, since circuit generation from Esterel programs is non-trivial. Here is an example

```
trap T in
   loop
      emit O
   each A
||
   await I;
   exit T
end;
emit B
```

Unexpectedly, that statement builds a dependency from `A` to `B`, which may cause a cycle if the reverse dependency exists somewhere else. Understanding why unfortunately requires understanding the circuit translation presented in [9][4]. Nevertheless, if the program is constructive, the Esterel v5 compiler will compile it correctly using the full constructiveness analysis.

[4] Since it always pauses, the first branch always returns termination code 1 to the parallel, and the wire that carries this information depends on `A`. The second branch has a wire for `exit T` that enters the parallel synchronizer at code 2. The two wires join at the synchronizer `and` gate at code 2. The wire sourced at this gate reaches the `emit B` statement, hence the dependency.

Chapter 6

Reflection on Perfect Synchrony

We end this primer by a reflection on the zero-delay or perfectly synchronous model on which Esterel and the other synchronous languages are based. We discuss two essential questions: why the model is needed, and how it relates to practical hardware or software implementation.

Our experience has shown that the reaction of users to the synchronous model varies greatly according to their background. Users trained to control theory find the model standard. Users trained to digital circuit design already know it. Users of cycle-based programmable controllers use it implicitly or explicitly. Users trained to classical concurrent programming languages or process calculi may find the model very puzzling. We hope that the discussion will help making things clear for most users.

We start by discussing the two kinds of models we need for Esterel: the idealized zero-delay model and the delay-based low-level implementation models. We discuss the relations between these two kinds of models, first in general and then in the particular cases of hardware and software implementations of Esterel.

6.1 Reactive Programming Models

6.1.1 The Qualities of a Model

Any programming model is a compromise between various conflicting requirements. A model must be simple and intuitive to help the user understanding and solving his problem in the simplest and neatest possible way.

A model must be accurate enough to describe the physical reality one deals with in a sensible way. A model must be mathematically efficient to be useful for defining the semantics of programs and for performing program analysis, optimization, and verification. Finally, a model must be general enough to cope with different ways of realizing systems in software, hardware, or mixed architectures that obey different execution logics. Because of the tension between these requirements, it may be necessary to use several models corresponding to several levels of abstraction.

Notice that we use the word *mathematical* instead of the more usual word *formal*. It is now clear to most language designers and users that language semantics should be formally defined. However, almost anything can be formalized with enough sweat and enough Greek letters and funny symbols. We go a step further and claim that semantics should be mathematically relevant to be of any real use. What really matters in mathematical semantics is that the objects and object combinations obey deep combinatorial and algebraic properties.

6.1.2 The Models of Esterel

For Esterel, we use two basic kinds of models.

- The semantics of programs is given in the logical synchronous or zero-delay model. We deal with sentences such as "`A` occurs" or "control goes to p", explaining *why* things happen but neglecting *how* they are actually realized. In particular, we neglect the time it takes to emit signals or to propagate the control. The programmer is encouraged to think about the behavior the program he or she writes in that logical way. The underlying mathematical theories are the Theory of Automata and the Boolean calculus[1], which are very powerful and very well-studied. See [9] for mathematical details.

- To build real systems, which is our actual goal, we move to more detailed hardware or software implementation models. The signal `A` can be implemented by a bit in memory, an interrupt, a voltage, the sampling of a continuous signal, or whatever is available. Control propagation can be realized by propagating electrical signals in a circuit or by chaining assembly instructions in a CPU. At this level, one may have to deal with a rather wide variety of physically accurate but mathematically very difficult models.

[1] or more exactly its constructive version, which does not matter here

The translation from the logical model to the various implementation models is performed by the compiler and the optimizers.

Our approach is based on the standard principle of *separation of concerns*. First, write the program at the logical level and make use of all the mathematics available there. Then, implement the program using the best available automatic synthesis tools, and check that the result is practically OK.

6.1.3 Inter-Model Consistency

A key practical issue is to check that a physical implementation is consistent with a specification and a set of additional realization constraints. For Esterel-like languages, one must often check that actual reaction times are consistent with real-time implementation constraints. Timing analysis heavily depends on the target implementation model. It is relatively easy for hardware implementation, since circuit reaction times are computed by CAD tools. Accurate timing analysis is much harder for software implementation, especially for RISC processors because of cache misses and pipeline stalls. However, that problem is not particular to Esterel. Even handwritten assembly programs are hard to predict. The only things we propose for Esterel are to develop good optimizers and to use any analysis tool available for the generated circuits and or software codes.

6.1.4 High-Level vs. Low-Level Programming Models

Because they constantly care for performance, many reactive programmers tend to discard high-level languages and to work at once within fine-grain real-time models of gates or assembly statements. However, such detailed models are most often mathematically too intricate to be of any practical efficiency, they tend to make programs hard to understand, and they compromise portability. Furthermore, the automata and Boolean algebra techniques are inaccessible to such models, which makes automatic program analysis, verification, and optimization much harder. The end result can actually be *loss of efficiency*, since automatic techniques based on high-level source code can actually generate more efficient object codes than manual techniques (see for example [43]).

A comparison with mechanics is useful at this point. Consider the problem of computing planet trajectories. In a very detailed approach, one considers planets as being made of atoms themselves made of elementary particles with mass and charge and obeying the laws of quantum mechanics.

In the gross Newtonian approach, one considers planets as perfect spheres or even as points attracting each other in function of their masses and positions, and there is no propagation delay for gravity. The first model is more accurate, but it is useless since it makes it totally impossible to compute trajectories. The second model is grossly idealized but it has more mathematical power and it makes the computation possible, with a reasonable estimate of the error. The gross model is in fact more *efficient* than the fine one for most practical uses. Similarly, for reactive programs, it may be a mistake to work within a fine-grain model unless this is strictly necessary. Similarly, our gross approach will not be able to solve all the problems, but it may well make 90% of them much simpler.

6.2 Logical Time vs. real Time

We now explain in more details why we stick to a purely logical notion of time at Esterel level. The clearest fact about a reactive system is that there is an alternation between environment moves and program moves, like in a two-players game. The environment chooses the inputs of the program, the program replies by computing the outputs. The game is asymmetrical in the sense that the environment drives it by choosing the inputs and the timing. The reactive program is in a slave position and it must be always ready to accept any input. Therefore, the global input can be characterized as being an environment-provided sequence of input events. Since we are interested in discrete systems, the sequence is a discrete one and the events can be indexed by successive integers called logical instants. In this setting, it is natural to consider an event as defining the status and value of each input signal, making it possible for signals to be simultaneous. For example, all the bits of a bus are received simultaneously by the bus clients. Input relations can be used to restrict simultaneity if needed.

Formally, logical instants can be represented by integers $n \in N$. The equality $n' = n + 1$ means that instant n immediately follows instant n', and $n' > n$ means that n' is in the strict future of n. A pure signal S generates a sequence of statuses S_n, a valued signal additionally generating a sequence of values $?S_n$.

Should we place logical instants on a real-time axis, defining the actual "physical time" $t(n)$ of the instant n? This natural temptation should be taken with care. Is that useful for all applications? Yes for many real-time programs, no for simple man-machine interface drivers. What does it bring in terms of power? The relation with continuous control theory for control programs or with sampling theory for signal processing, the relation with actual timing delays in telecommunication or systems drivers, nothing for many other untimed reactive applications. Does real-time indexing bring unexpected annoying consequences? Yes, Zeno behavior being a good example (a Zeno behavior occurs when an infinite sequence of discrete events occurs within a finite amount of physical time). Since it makes the formalism heavier without necessarily adding power, real-time should be introduced only if necessary for the application and it should not be a mandatory part of the model.

When are the outputs generated by the reactive program? In the logical view, there is no problem. The output corresponding to the input event I_n receives the same index and is called O_n. From within the programs, the signals in I_n and the signals in O_n can be equally tested for presence or absence, as well as the local signals used within the program. There is no logical delay in generating the output. In a real-time model, the question is more delicate and one may adopt two main points of view:

- The perfectly synchronous point of view asserts that O_n occurs at time $t(n)$ if I_n occurs at time $t(n)$, thus neglecting the computation time. This is what mathematicians do when they write $x_t = y_t + z_t$ in control equations. In that case, the real number $t(n)$ is of no use.

- The delay point of view asserts that O_n occurs at some time $t'(n) > t(n)$, with the constraint $t'(n) < t(n+1)$ for the reactive system to behave properly. An additional bounded delay property asserting the existence of a maximum delay δ such that $t'(n) - t(n) < \delta$ for all n may be additionally required.

Clearly, the first point of view corresponds to Newtonian mechanics. It is the simplest one and also the mathematically most efficient one. It gives the clearest semantics to programs: in the `SPEED` program of Section 3.4, the value of the `Speed` signal is exactly the speed according to mathematics, which would not be true in the delay approach. Therefore, the logical model with only integer indices is the one of choice for reasoning about program behavior.

Conversely, the delay approach closely corresponds to generated code analysis. The favorable case is when the maximal delay d can be computed for a given implementation. In that case, if d makes us happy, there is no reason to worry and the synchronous model is as good an approximation as Newtonian mechanics is for planets. Otherwise, we are in trouble, and implementation must be improved. Of course, the value d depends more on

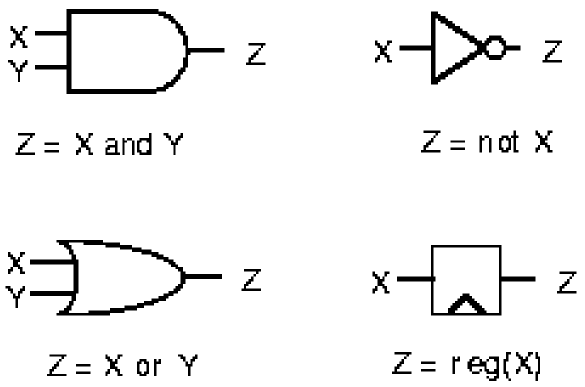


Figure 6.1: Circuit gates

the implementation target than on the source program and is hard to deal with. This is why it should not be introduced early in the program design.

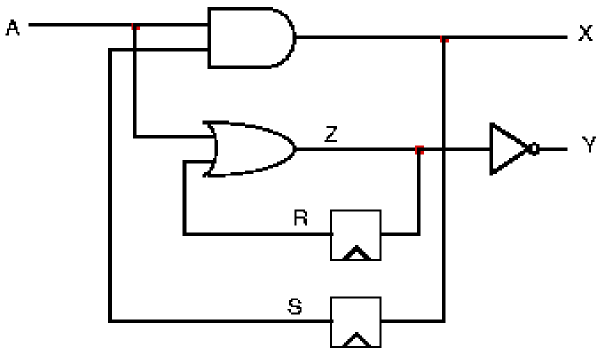
6.3 Implementation by Sequential Circuits

We now briefly analyze the logical and delay models for digital sequential circuits. In [9], we show how to translate an Esterel program into a sequential circuit. This translation is currently the basis of both the hardware and the software implementation of Esterel.

A sequential circuit is a graph of gates, drawn using conventional symbols pictured in Figure 6.1. One usually divides a circuit into a combinational part and a register part. The combinational part has `and`, `or`, and `not` gates. The register part contains elementary delay elements called registers. An example is pictured in Figure 6.2.

6.3.1 The Logical View of Circuits

In the zero-delay logical view of circuits, wires are Boolean variables and gates define Boolean equations to be solved given the current input and the current state of registers. Initially, the registers all have value 0. If inputs are given, the outputs variables of the combinational part determine the Boolean primary outputs and the values to be fed into the registers for next input. Like in Esterel, one solves equations without caring about physical time. If the combinational part has no cycle, then it is obvious that the Boolean outputs and next register state are uniquely determined



$$\begin{aligned}
X &= A \;\texttt{and}\; S \\
Y &= \texttt{not}\; Z \\
Z &= A \;\texttt{or}\; R \\
R &= \texttt{reg}(Z) \\
S &= \texttt{reg}(X)
\end{aligned}$$

Figure 6.2: A sequential circuit

by the inputs and current register state. For circuits with combinational cycles, outputs may not always be well-determined; the constructive logic propagation theory presented in [41, 9, 54] characterizes logically sensible (delay-independent) cyclic circuits, which exactly corresponds to constructive ESTEREL programs.

6.3.2 The Electrical View of Circuits

In the electrical view of circuits, we use a simple but reasonably accurate model called the *up-bounded inertial delay model* by J. Brzozowski. More sophisticated electrical models won't deeply modify the picture.

Wires can take values 0 (or 0 Volt) and 1 (or 5 Volt) at any real time t. A wire can switch instantaneously from 0 to 1 or conversely. A gate has a delay δ with the following properties:

- There is no spontaneous output switch: if the output switches at time t, then at least one input must have switched between time $t - \delta$ and time t.

- Gates switch according to their Boolean meanings: if all gate inputs

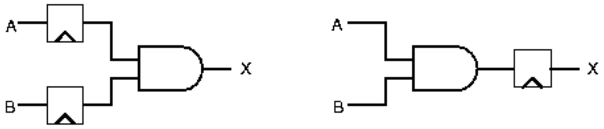


Figure 6.3: Retiming sequential circuits

> are kept stable to Boolean values between time $t - \delta$ and time t, then the output becomes stable at time t and its value is the one determined by the gate's Boolean function.

Delays can also be put on wires. Notice that short transient input switches may or may not show up at a gate output.

Registers are driven by a global clock, which is a signal that alternates between 0 and 1. A clock front time t, the register output is set to the current register input and it remains stable up to the next clock front.

Consider an acyclic combinational part. It the input and register wires are kept stable, then, after some input-independent bounded time, the outputs are guaranteed to be stable. The bound is obviously determined by the longest path delay from combinational inputs to combinational outputs, delays summing up along paths. Call d the bound. Considering now the registers, we require to use the circuit as follows: keep the inputs stable within each clock period, and ensure that the clock period is greater than d. Then the output and register new state wires are guaranteed to be stable at each clock front. At that time, the outputs can be sampled by the environment and the registers get updated. Notice that there is no need for the clock to be regular.

6.3.3 Connecting the Logical and Electrical Views

The connection between the logical and electrical models is simple: the circuit electrically computes the logical Boolean results in at most one clock cycle. This is not zero-delay, but the maximal reaction delay d can be measured and we can verify whether it makes us happy or not [2].

The delay d does not solely depend on the specification, for there are lots of ways to implement a Boolean circuit specification. At the zero-delay level, combinational optimization can be used to replace the original combinational

[2] The result extends to circuits having a cyclic combinational part, as explained in [9, 54]: electrical stabilization is equivalent to solvability in Constructive Boolean Logic.

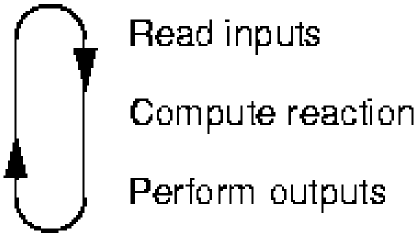


Figure 6.4: Cyclic reactive software

equations by equivalent but more efficient ones [53]. Sequential optimization can be used to improve the state encoding in registers [51, 52, 53]. At the electrical level, technology mapping makes it possible to use various electronic gate structures to realize different speed/area/power tradeoffs, and lots of target physical electronic technologies are available. Retiming [40] is a logical transformation with interesting electrical consequences. For instance, although the circuits pictured in Figure 6.3 have the same behavior, the one on the right is electrically preferable since its output is ready at the beginning of the clock cycle (it is truly zero-delay).

To summarize things, with the modern view of electronic circuit design, implementation issues are largely decoupled from specification issues. Because of the power of Boolean simplification techniques, the zero-delay model has become the model of choice to design synchronous circuits, and automatic synthesis is becoming the technique of choice for actual circuit implementation.

6.4 Software Implementation

Software implementation of reactive programs relies on the same basic principles as hardware implementation, but the details differ since event handling is usually not done in the same way in software. The primary idea is still that of cycle-based computation, pictured in Figure 6.4. A cycle consists in an input step, a reaction step, and an output step. Input can be performed in several ways, e.g., interrupt handling, polling and event sampling, shared memory reading, etc.

In the Esterel v5 C generated code, interfacing the generated code is done by calling automatically generated input and reaction functions. The input of a signal `A` to a module `M` is performed by calling a generated input function called `M_I_A()`. Calling this function only sets one status bit and stores the value in a temporary location for valued signals, which is very fast. Setting simultaneous inputs simply consists in calling their input functions in any order. The reaction of M to the input event consisting of the input signals received so far is performed by calling a generated C function called M(). The reaction updates the value store and the state, and it calls a user-supplied output function called M_O_X() for each emitted signal X. For example, assume we want to trigger a reaction with two inputs A and B present. We execute the following C code:

```
M_I_A();
M_I_B();
M();
```

If X is emitted by the reaction, the function M() will call an output function M_O_X() we have to supply.

The reaction is subject to an essential atomicity condition: during the execution of the main reaction function M(), it is forbidden to call any input function. This is the software counterpart to the hardware requirement that input voltages must be kept stable during a clock cycle.

As for hardware, there is lots of freedom for code generation. The Esterel v5 compiler can generate straight-line code implementing a sorted system of Boolean equations (default option), automata tables (-A option), etc. Different code structures correspond to different time/space tradeoffs. There is no obligation to implement the program in a way that resembles its logical structure. Unlike in conventional concurrent languages, writing source code concurrent threads does not mean executing matching run-time threads. The only thing that matters is preserving the behavior. As for circuits, the full power of Boolean calculus and automata theory can be used to optimize the generated code. Some transformations can entirely remove concurrency, i.e., perform scheduling at compile-time. Other can unravel some form of concurrency not explicit in the source program and usable at run-time. There is still an enormous potential of improvement in that area, which is outside the scope of this primer.

Chapter 7

The Esterel Grammar

7.1 Syntax Notation

The context-free syntax of the language is described using a simple variant of Backus-Naur-Form (BNF). In particular,

(i) Concatenated capitalized words are used to denote syntactic categories, for example:

> *Identifier*
> *ExceptionHandler*

The page reference of the description of a category can be found under the "Grammar Summary" index entry.

(ii) `Typewriter` type style words or characters are used to denote reserved words, delimiters or lexical elements of the language, other than identifiers. For example:

> `type` *TypeDeclList* `;`
> *Identifier* `=>` *Identifier*

(iii) Alternative items are on different lines. For example:

ConstantLiteral :

> *Identifier*
> `true`
> `false`
> *StringConstant*

A single item can extend over many lines. In that case, additional indentation is put on the lines following the first one. For example:

Abort :

`weak`$_{opt}$ `abort` *Statement* `when` *DelayExpression*

`weak`$_{opt}$ `abort` *Statement* `when` *DelayExpression*
  `do` *Statement* `end` `weak`$_{opt}$ `abort`$_{opt}$

`weak`$_{opt}$ `abort` *Statement* `when`
  AbortCaseList
  `end` `weak`$_{opt}$ `abort`$_{opt}$

(iv) Optional items have *opt* written in subscript. Thus the two following rules are equivalent:

AwaitCaseList :

AwaitCaseList$_{opt}$ *AwaitCase*

AwaitCaseList :

AwaitCase

AwaitCaseList AwaitCase

(vi) If the name of any syntactic category does not start with italicized style, it is equivalent to the category name of the italicized part. The unitalicized part is intended to convey some semantic information. For example Module*Identifier* or Signal*Identifier* are both equivalent to *Identifier* alone.

7.2 Modules

Module :

`module` Module*Identifier* :
  InterfaceDeclList$_{opt}$
  Statement
  `end` `module`$_{opt}$

7.3 Interface Declaration

InterfaceDeclList :

InterfaceDeclList$_{opt}$ *InterfaceDecls*

InterfaceDecls :

TypeDecls

ConstantDecls

FunctionDecls

ProcedureDecls

TaskDecls

InterfaceSignalDecls

SensorDecls

RelationDecls

OutputDecls

7.3.1 Type Declarations

TypeDecls :

`type` *TypeDeclList* ;

TypeDeclList :

TypeDecl

TypeDeclList , *TypeDecl*

TypeDecl :

Identifier

7.3.2 Constant Declarations

ConstantDecls :

`constant` *ConstantDeclList* ;

ConstantDeclList :

ConstantDecl

ConstantDeclList `,` *ConstantDecl*

ConstantDecl :

Identifier `:` Type*Identifier*

Identifier `=` *ConstantAtom* `:` Type*Identifier*

7.3.3 Function Declarations

FunctionDecls :

`function` *FunctionDeclList* `;`

FunctionDeclList :

FunctionDecl

FunctionDeclList `,` *FunctionDecl*

FunctionDecl :

Identifier `(` Type*IdentifierList*$_{opt}$ `)` `:` Type*Identifier*

IdentifierList :

Identifier

IdentifierList `,` *Identifier*

7.3.4 Procedure Declarations

ProcedureDecls :

`procedure` *ProcedureDeclList* `;`

ProcedureDeclList :

ProcedureDecl

ProcedureDeclList `,` *ProcedureDecl*

ProcedureDecl :

Identifier `(` Type*IdentifierList*$_{opt}$ `)` `(` Type*IdentifierList*$_{opt}$ `)`

7.3.5 Task Declarations

TaskDecls :

`task` *TaskDeclList* ;

TaskDeclList :

TaskDecl

TaskDeclList , *TaskDecl*

TaskDecl :

Identifier (Type*IdentifierList*$_{opt}$) (Type*IdentifierList*$_{opt}$)

7.3.6 Signal Declarations

InterfaceSignalDecls :

`input` *SignalDeclList* ;

`output` *SignalDeclList* ;

`inputoutput` *SignalDeclList* ;

`return` *SignalDeclList* ;

SignalDeclList :

SignalDecl

SignalDeclList , *SignalDecl*

SignalDecl :

Identifier

Identifier : *ChannelType*

Identifier `:=` *Expression* : *ChannelType*

ChannelType :

Type*Identifier*

`combine` Type*Identifier* `with` Function*Identifier*

`combine` Type*Identifier* `with` *PredefinedCombineFunction*

PredefinedCombineFunction :

`+`

`*`

`or`

`and`

7.3.7 Sensor Declarations

SensorDecls :

`sensor` *SensorDeclList* ;

SensorDeclList :

SensorDecl

SensorDeclList , *SensorDecl*

SensorDecl :

Identifier : Type*Identifier*

7.3.8 Input Relation Declarations

RelationDecls :

`relation` *RelationDeclList* ;

RelationDeclList :

RelationDecl

RelationDeclList , *RelationDecl*

RelationDecl :

ImplicationDecl

ExclusionDecl

ImplicationDecl :

Signal*Identifier* `=>` Signal*Identifier*

ExclusionDecl :

Signal*Identifier* `#` Signal*Identifier*

ExclusionDecl `#` Signal*Identifier*

7.4 Expressions

7.4.1 Data Expressions

Expression :

Constant

`(` *Expression* `)`

`?` Signal*Identifier*

`pre(?` Signal*Identifier* `)`

`??` Exception*Identifier*

`-` *Expression*

Expression `*` *Expression*

Expression `/` *Expression*

Expression `+` *Expression*

Expression `-` *Expression*

Expression `mod` *Expression*

Expression `=` *Expression*

Expression `<>` *Expression*

Expression `<` *Expression*

Expression `<=` *Expression*

Expression `>` *Expression*

Expression `>=` *Expression*

`not` *Expression*

Expression `and` *Expression*

Expression `or` *Expression*

FunctionCall

The precedence of operators is given below. Each line holds operators with the same precedence. Operators in higher lines have higher precedence than operators in lower lines. All operators are left-associative, except the minus unary operator which is right-associative.

`-` *(unary minus)*

`*`   `/`   `mod`

`+`   `-`

`=`   `<>`   `<`   `<=`   `>`   `>=`

`not`

`and`

`or`

Constant :

ConstantLiteral

UnsignedNumber

ConstantLiteral :

Constant*Identifier*

`true`

`false`

StringConstant

ConstantAtom :

ConstantLiteral

SignedNumber

SignedNumber :

UnsignedNumber

`-` *UnsignedNumber*

UnsignedNumber :

UnsignedIntegerConstant

UnsignedFloatConstant

UnsignedDoubleConstant

FunctionCall :

Function*Identifier* `(` *ExpressionList*$_{opt}$ `)`

ExpressionList :

Expression

ExpressionList `,` *Expression*

7.4.2 Signal Expressions

SignalExpression :

Signal*Identifier*

`pre(` Signal*Identifier* `)`

`not` *SignalExpression*

SignalExpression `and` *SignalExpression*

SignalExpression `or` *SignalExpression*

`(` *SignalExpression* `)`

7.4.3 Delay Expressions

DelayExpression :

BracketedSignalExpression

`immediate` *BracketedSignalExpression*

Expression *BracketedSignalExpression*

BracketedSignalExpression :

Signal*Identifier*

`[` *SignalExpression* `]`

7.5 Statements

Statement :

Parallel

NonParallel

Parallel :

NonParallel `||` *NonParallel*

Parallel `||` *NonParallel*

NonParallel :

AtomicStatement

Sequence

Sequence :

SequenceWithoutTerminator `;`$_{opt}$

SequenceWithoutTerminator :

AtomicStatement `;` *AtomicStatement*

SequenceWithoutTerminator `;` *AtomicStatement*

Notice that sequencing has priority over parallelism. Notice also that we make it possible to terminate a sequence by a trailing semicolon. This is to (partly) conform with C programming style where '`;`' is a terminator. It is therefore possible to write "`abort emit X; emit Y; when S`".

AtomicStatement :

`nothing`

`pause`

`halt`

Emit

Sustain

Assignment

ProcedureCall

Present

If

Loop

Repeat

Abort

Await

LoopEach

Every

Suspend

Trap

Exit

Exec

LocalVariableDecl

LocalSignalDecl

RunModule

`[` *Statement* `]`

7.5.1 Signal Emission

Emit :

`emit` Signal*Identifier*

`emit` Signal*Identifier* `(` *Expression* `)`

Sustain :

`sustain` Signal*Identifier*

`sustain` Signal*Identifier* `(` *Expression* `)`

7.5.2 Assignment and Procedure Call

Assignment :

Variable*Identifier* `:=` *Expression*

ProcedureCall :

`call` Procedure*Identifier* `(` Variable*IdentifierList*$_{opt}$ `)`
`(` *ExpressionList*$_{opt}$ `)`

7.5.3 The present Signal Test

Present :

PresentThenElse

PresentCaseElse

PresentThenElse :

`present` *PresentEvent* *ThenPart*$_{opt}$ *ElsePart*$_{opt}$ `end` `present`$_{opt}$

PresentEvent :

SignalExpression

`[` *SignalExpression* `]`

ThenPart :

`then` *Statement*

ElsePart :

`else` *Statement*

PresentCaseElse :

`present` *PresentCaseList* *ElsePart*$_{opt}$ `end` `present`$_{opt}$

PresentCaseList :

PresentCase

PresentCaseList *PresentCase*

PresentCase :

`case` *PresentEvent*

`case` *PresentEvent* `do` *Statement*

7.5.4 The if Data Test

If :

`if` *Expression* *ThenPart*$_{opt}$ *ElsifPartList*$_{opt}$ *ElsePart*$_{opt}$ `end if`$_{opt}$

ElsifPartList :

Elsif

ElsifPartList *Elsif*

Elsif :

`elsif` *Expression* *ThenPart*$_{opt}$

7.5.5 Looping

Loop :

`loop` *Statement* `end loop`$_{opt}$

7.5.6 Repeat Loops

Repeat :

`positive`$_{opt}$ `repeat` *Expression* `times` *Statement* `end repeat`$_{opt}$

7.5.7 The abort Statements

Abort :

`weak`$_{opt}$ `abort` *Statement* `when` *DelayExpression*

`weak`$_{opt}$ `abort` *Statement* `when` *DelayExpression*
`do` *Statement* `end` `weak`$_{opt}$ `abort`$_{opt}$

`weak`$_{opt}$ `abort` *Statement* `when`
AbortCaseList
`end` `weak`$_{opt}$ `abort`$_{opt}$

AbortCaseList :

AbortCaseList$_{opt}$ *AbortCase*

AbortCase :

> `case` *DelayExpression* `do` *Statement*
>
> `case` *DelayExpression*

A `weak abort` statement can be ended with either `end weak abort`, `end abort`, or just `end`. An `abort` statement can only be ended with either `end abort`, or `end`.

7.5.8 The await Statement

Await :

> `await` *DelayExpression*
>
> `await` *DelayExpression* `do` *Statement* `end await`$_{opt}$
>
> `await` *AwaitCaseList* `end await`$_{opt}$

AwaitCaseList :

> *AwaitCaseList*$_{opt}$ *AwaitCase*

AwaitCase :

> `case` *DelayExpression* `do` *Statement*
>
> `case` *DelayExpression*

7.5.9 Temporal Loops

LoopEach :

> `loop` *Statement* `each` *DelayExpression*

Every :

> `every` *DelayExpression* `do` *Statement* `end every`$_{opt}$

7.5.10 The suspend Statement

Suspend :

> `suspend` *Statement* `when` *DelayExpression*

7.5.11 Traps

Trap :

`trap` *ExceptionDeclList* `in` *Statement*
ExceptionHandlerList$_{opt}$
`end trap`$_{opt}$

ExceptionDeclList :

ExceptionDecl

ExceptionDeclList `,` *ExceptionDecl*

ExceptionDecl :

Identifier *ChannelType*$_{opt}$

Identifier `:` *ChannelType*

Identifier `:=` *Expression* `:` *ChannelType*

Exit :

`exit` Exception*Identifier*

`exit` Exception*Identifier* `(` *Expression* `)`

ExceptionHandlerList :

ExceptionHandlerList$_{opt}$ *ExceptionHandler*

ExceptionHandler :

`handle` *ExceptionEvent* `do` *Statement*

ExceptionEvent :

Exception*Identifier*

`(` *ExceptionEvent* `)`

`not` *ExceptionEvent*

ExceptionEvent `and` *ExceptionEvent*

ExceptionEvent `or` *ExceptionEvent*

7.5.12 The exec Task Execution Statement

Exec :

`exec` Task*Identifier* `(` Variable*IdentifierList*$_{opt}$ `)` `(` *ExpressionList*$_{opt}$ `)`
`return` ReturnSignal*Identifier*

`exec` Task*Identifier* `(` Variable*IdentifierList*$_{opt}$ `)` `(` *ExpressionList*$_{opt}$ `)`
`return` ReturnSignal*Identifier* `do` *Statement* `end` `exec`$_{opt}$

`exec` *ExecCaseList* `end` `exec`$_{opt}$

ExecCaseList :

ExecCaseList$_{opt}$ *ExecCase*

ExecCase :

`case` Task*Identifier* `(` Variable*IdentifierList*$_{opt}$ `)` `(` *ExpressionList*$_{opt}$ `)`
`return` ReturnSignal*Identifier* `do` *Statement*

`case` Task*Identifier* `(` Variable*IdentifierList*$_{opt}$ `)` `(` *ExpressionList*$_{opt}$ `)`
`return` ReturnSignal*Identifier*

7.5.13 Local Signal Declaration

LocalSignalDecl :

`signal` *SignalDeclList* `in` *Statement* `end` `signal`$_{opt}$

7.5.14 Local Variable Declaration

LocalVariableDecl :

`var` *VariableDeclList* `in` *Statement* `end` `var`$_{opt}$

VariableDeclList :

VariableDecl

VariableDeclList `,` *VariableDecl*

VariableDecl :

Identifier `:` Type*Identifier*

Identifier `:=` *Expression* `:` Type*Identifier*

7.5.15 The run Module Instantiation Statement

RunModule :

`run` *RunModuleNames*

`run` *RunModuleNames* `[` *RenamingList* `]`

RunModuleNames :

Module*Identifier*

Module*Identifier* `/` Module*Identifier*

RenamingList :

Renaming

RenamingList `;` *Renaming*

Renaming :

`type` *TypeRenamingList*

`constant` *ConstantRenamingList*

`function` *FunctionRenamingList*

`procedure` *ProcedureRenamingList*

`task` *TaskRenamingList*

`signal` *SignalRenamingList*

TypeRenamingList :

TypeRenaming

TypeRenamingList `,` *TypeRenaming*

TypeRenaming :

Type*Identifier* `/` Type*Identifier*

ConstantRenamingList :

ConstantRenaming

ConstantRenamingList `,` *ConstantRenaming*

ConstantRenaming :

ConstantAtom `/` Constant*Identifier*

FunctionRenamingList :

FunctionRenaming

FunctionRenamingList `,` *FunctionRenaming*

FunctionRenaming :

Function*Identifier* `/` Function*Identifier*

PredefinedFunction `/` Function*Identifier*

PredefinedFunction :

`and`

`or`

`not`

`+`

`-`

`*`

`/`

`mod`

`<`

`>`

`<=`

`>=`

`<>`

`=`

ProcedureRenamingList :

ProcedureRenaming

ProcedureRenamingList `,` *ProcedureRenaming*

ProcedureRenaming :

Procedure*Identifier* `/` Procedure*Identifier*

TaskRenamingList :

TaskRenaming

TaskRenamingList `,` *TaskRenaming*

TaskRenaming :

Task*Identifier* **/** Task*Identifier*

SignalRenamingList :

SignalRenaming

SignalRenamingList **,** *SignalRenaming*

SignalRenaming :

Signal*Identifier* **/** Signal*Identifier*

7.6 Old Syntax

Previous versions of Esterel used a different syntax for some constructs. For backward compatibility, we have chosen to still parse the old syntax, although we try to discourage its usage.

Valued signals used to be declared using parentheses as in `emit` statements:

```
output Speed (integer);
output Beeper (combine Beep with CombineBeeps);
```

the standard notation is now the colon '`:`' as for variables.

Abortion used to be written with the `watching` and `upto` keywords. For instance,

```
abort
   p
when S
```

used to be written

```
do
   p
watching S
```

and

```
abort
   p
when S do
   q
end abort
```

used to be written

```
do
   p
watching S timeout
   q
end timeout
```

The `upto` statement used to be written

```
do
   p
upto S
```

with the following meaning:

```
abort
   p; halt
when S
```

This statement turned out to be not fundamental and its name is not fully clear. It is still available.

Finally, it used to be possible to declare several constants or variables of a given type in a row, as in:

```
constant A, B : integer;
var X, Y : integer in ...
```

to be interpreted as

```
constant A : integer , B : integer;
var X : integer, Y : integer in ...
```

This syntax is now discouraged and not part of the Esterel definition, because it does not extend to signal declarations. The declaration

```
signal X, Y : integer in ...
```

means that `X` is a pure signal while `Y` is an integer-valued signal. Therefore, it is not equivalent to

```
signal X : integer, Y : integer in ...
```

Bibliography

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

[2] K. Arnold and J. Gosling. *The Java Programming Language.* Addison-Wesley, 1996.

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

[4] F. Balarin, M. Chiodo, A. Jurecska, H. Hsieh, A. L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara. *Hardware-Software Co-Design of Embedded Systems: The Polis Approach.* To be published by Kluwer Academic Press, 1997.

[5] 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.

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

[7] G. Berry. The foundations of Esterel. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction: Essays in Honour of Robin Milner.* MIT Press, 2000.

[8] G. Berry. Programming a digital wristwatch in Esterel. In *http://www.esterel.org*, 2000.

[9] G. Berry. *The Constructive Semantics of Pure Esterel.* Draft book, available at http://www.esterel.org, version 3, July 1999.

[10] G. Berry, A Bouali, X. Fornari E. Nassor, E. Ledinot, and R. de Simone. Esterel: a formal method applied to avionic development. *Science of Computer Programming*, 36:5–25, 2000.

[11] G. Berry and L. Cosserat. The synchronous programming languages Esterel and its mathematical semantics. In S. Brookes and G. Winskel, editors, *Seminar on Concurrency*, pages 389–448. Springer Verlag Lecture Notes in Computer Science 197, 1984.

[12] G. Berry, P-L. Curien, and J-J. Lévy. *Full Abstraction for Sequential Languages*, pages 89–132. Cambridge University Press, 1985.

[13] G. Berry and G. Gonthier. Incremental development of an HDLC entity in Esterel. *Comp. Networks and ISDN Systems*, 22:35–49, 1991.

[14] G. Berry, S. Moisan, and J-P. Rigault. Towards a synchronous and semantically sound high level language for real-time applications. In *IEEE Real Time Systems Symposium*, pages 30–40. IEEE Catalog 83 CH 1941-4, 1983.

[15] G. Berry and R. Sethi. From regular expressions to deterministic automata. *Theoretical Computer Science*, 48:117–126, 1987.

[16] P. Bertin, D. Roncin, and J. Vuillemin. Programmable active memories: a performance assessment. In G. Borriello and C. Ebeling, editors, *Research on Integrated Systems: Proceedings of the 1993 Symposium*, pages 88–102, 1993.

[17] A. Bouali. Xeve: an Esterel verification environment. In *Proc. Computer Aided Verification (CAV'98), Vancouver, Canada*, 1998.

[18] F. Boussinot. Reactive C: An extension of C to program reactive systems. *Software Practice and Experience*, 21(4):401–428, 1991.

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

[20] F. Boussinot, G. Doumenc, and J.B. Stefani. Reactive objects. *Annals of Telecommunications*, 51(9–10):459–473, 1996.

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

[22] A. Burns and A. Wellings. *Concurrency in Ada*. Cambridge University Press, 1995.

[23] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. LUSTRE: a declarative language for programming synchronous systems, 1987.

[24] I. Chrisment, D. Kaplan, and C. Diot. An alf communication architecture: Design and automated implementation. *To appear in JSAC*.

[25] D. Clément and J. Incerpi. Programming the behavior of graphical objects using Esterel. In *TAPSOFT '89, Springer-Verlag LNCS 352*, 1989.

[26] L. Cosserat. Sémantique opérationnelle du langage synchrone Esterel. Thèse de docteur-ingénieur, Université de Nice, 1985.

[27] Eve Coste-Manière and Nicolas Turro. The MAESTRO language and its environment: Specification, validation and control of robotic missions. In *IEEE/RSJ Intl Conf on Intelligent Robots and Systems, IROS'97*, pages vol2, p 836, Grenoble, France, September 1997.

[28] S. Ramesh G. Berry, R. K. Shyamasundar. Communicating reactive processes. In *Proc. 20th ACM Conf. on Principles of Programming Languages, Charleston, Virginia*, 1993.

[29] G. Gonthier. Sémantique et modèles d'exécution des langages réactifs synchrones; application à Esterel. Thèse d'informatique, Université d'Orsay, 1988.

[30] P. Le Guernic, A Benveniste, P. Bournai, and T. Gautier. SIGNAL: a data-flow oriented language for signal processing. Technical report, RR. 378, INRIA, 1985.

[31] 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.

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

[33] 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.

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

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

[36] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall International, 1985.

[37] C.A.R. Hoare. *Concurrent programming.* Addison-Wesley, 1990.

[38] L. Jagadeesan, J. Von Olnhausen, and C. Puchol. Safety property verification of Esterel programs and applications to telecommunications software. In *Proc. Conf. on Computer-Aided Verification (CAV'95), Liège, Belgium*, July 1995.

[39] L. Jagadeesan, J. Von Olnhausen, and C. Puchol. A formal approach to reactive system software: A telecommunications application in Esterel. *Journal of Formal Methods in Systems Design*, 8(2), March 1996.

[40] C.E. Leiserson and J.B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1), 1991.

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

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

[43] 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.

[44] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25(3), 1983.

[45] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[46] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part i and ii. *Information and Computation*, 100(1):1–77, 1992.

[47] A. K. Mok, C. Puchol, and D. Stuart. Compiling Modechart specifications. In *Proc. IEEE Real-Time Systems Symposium (RTSS '95), Pisa, Italy*, December 1995.

[48] G. Murakami and Ravi Sethi. Terminal call processing in Esterel. In *Proc. IFIP 92 World Computer Congress, Madrid, Spain*, 1992.

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

[50] O. Roux. Compilation of the Electre reactive language into finite transition systems. *Theoretical Computer Science*, 146:109–143, 1995.

[51] E. Sentovich, H. Toma, and G. Berry. Latch optimization in circuits generated from high-level descriptions. In *Proc. International Conf. on Computer-Aided Design (ICCAD)*, 1996.

[52] E. Sentovich, H. Toma, and G. Berry. Efficient latch optimization using exclusive sets. In *Proc. Digital Automation Conference (DAC)*, 1997.

[53] E.M. Sentovich, K.J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P.R. Stephan, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. Sis: A system for sequential circuit synthesis. Technical report, University of California at Berkeley, 1992. Memorandum No. UCB/ERL M92/41.

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

[55] Thomas R. Shiple, Vigyan Singhal, Gérard Berry, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Analysis of combinational cycles. Technical Report UCB/ERL M96, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, 1996.

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

Index