The Constructive Semantics of Pure Esterel
Draft Version 3

G. Berry
Centre de Mathématiques Appliquées
Ecole des Mines de Paris and INRIA
2004 route des Lucioles
06902 Sophia-Antipolis CDX, France

berry@sophia.inria.fr
http://www.inria.fr/meije/Personnel/Gerard.Berry.html

July 2, 1999

Contents

List of Figures

Preface

We present the reference semantical framework for the synchronous programming language Esterel, which we call the constructive framework. After a detailed explanation of the causality issues in the language and an intuitive presentation of constructiveness, we show the equivalence between three semantics: the constructive behavioral semantics, the constructive operational semantics, and the electrical semantics, which is based on a translation of Esterel programs into digital circuits. The main full abstraction result is that an Esterel program is constructive if and only if its possibly cyclic circuit reaches electrical stabilization in bounded time. The constructive semantics supersedes all previous semantical attempts for the definition of Esterel, while remaining perfectly compatible with them. The constructive semantics is compositional, unlike its predecessors.

The constructive semantics is implemented in the Esterel v5 compiler. This new compiler accepts all programs accepted by the previous Esterel v3 and v4 compiler, as well as all the constructive programs these compilers reject.

This draft book is directed to the users of Esterel who want to understand the underlying language and semantics design, and, more generally, to the readers who are interested in language semantics. It complements the general presentation of Esterel called "The Foundations of Esterel", see [8], the Esterel Primer, [7], and the Esterel v5 Systems Manual [12]. Our central aims are to explain the design and properties of the constructive semantics, which, to our belief, should be the final one, to make causality issues understandable, a thing that has long been missing, and finally to fully explain the translation of Esterel programs into circuits.

In the former releases 1.1 and 2.0 of this book, there were subtle bugs in the behavioral semantics rules. Please accept my apologies, and throw away copies. Fortunately, the circuit translation was already correct and so was the compiler. The bug in version 2.0 was found by Delphine Terrasse (now Kaplan), who tried to prove the correctness of the circuit translation using the COQ proof-checker [40]; many thanks to her. Some other minor mistakes have also been corrected.

The operational semantics was presented in version 1.0, but removed from version 1.1 and 2.0. This is because the author designed a much improved version but had no time to write it down yet; so far, he has chosen to favor the development of the v5 system over the writing of this book. Hopefully, the *operational semantics will be included in the next release. The other missing parts concern the extension to data-handling and the implementation issues.*

Acknowledgements: Our work on the constructive semantics owes much to Frédéric Boussinot, from École des Mines, Olivier Ploton and Robert de Simone, from INRIA, Tom Shiple, from Synopsys, and Ellen Sentovich, from Cadence. Many thanks to Ellen Sentovich, Emmanuel Ledinot (from Dassault Aviation), and John Plaice, who performed a careful proof-reading.

Comments from readers are very welcome. Please send them by e-mail to berry@sophia.inria.fr.

Chapter 1

Introduction

Esterel [11, 14, 4, 5] is an imperative synchronous parallel programming language dedicated to reactive systems [22]. Esterel is tailored for programming hardware or software synchronous controllers for which the control-handling aspects are predominant. Esterel programs are input-driven: they wait for inputs and compute corresponding outputs in a cyclic manner. An input-output computation is called a reaction. Synchrony conceptually means that reactions take no time, or, equivalently, that outputs become available as inputs become available. By abstracting away reaction times, synchrony reconciles concurrency and determinism, and it greatly simplifies controller programming. Furthermore, it becomes possible to use sophisticated implementation, optimization, and verification techniques commonly used in areas such as process calculi or hardware design, and to extend them to software applications [22, 41].

Other synchronous languages are the graphical languages Argos [29] and SyncCharts [1], also dedicated to controllers, the data-flow languages Lustre and Signal, and the TCCP timed concurrent constrained programming language of [35]. See [22] for a comprehensive survey of Esterel, Argos, Lustre, and Signal. Although it is not fully synchronous, Harel's Statecharts graphical formalism [25] clearly belongs to the same language family [1]

This book is devoted to the formal semantics of Esterel and to the translation of Esterel programs into Boolean circuits, upon which the current Esterel hardware or software compiling and verification technology is based.

[1] One can indeed see Maraninchi's Argos formalism as a simplification of Statecharts with a fully synchronous semantics and André's SyncCharts as an extension of Argos towards the greater expressive power of Esterel. Since SyncCharts can be translated into Esterel, its semantics is covered by this book.

The book should serve as a reference manual for the language semantics and for the Esterel compiling algorithms. We shall not discuss programming style or adequacy to practical applications, this being done elsewhere. See [8] for a general overview, see [7] for the Esterel Manual, and [22, 2, 13, 3, 10, 32, 27] for examples.

The book concentrates on Pure Esterel, which is restricted to pure signaling. Pure Esterel is enough to express all the semantical difficulties and their solution. The extension to the full Esterel language that handles data of arbitrary data types will only be outlined; if not entirely straightforward, it raises no particularly difficult problems.

The Constructive Semantics

The main semantics we present is called the *constructive semantics*. It is a major improvement over previous semantics discussed in [11, 5], since it solves the *causality problem* common to all synchronous formalisms, although particularly acute in Esterel. Many Esterel users have suffered from the insufficiencies of the previous attempts, and we hope that the constructive semantics will make them happy.

The constructive semantics is presented in several variants, which are shown to be equivalent. The variants serve different purposes. The *constructive behavioral semantics* is the simplest and the most abstract one. Its main use is to formally define what a program means. It is effective and it could be directly used to build interpreters and compilers; however, these would be very inefficient. The *constructive operational semantics* is a microstep semantics, which analyzes the fine structure of control and signal propagation in the reaction. It extends the original operational semantics of [11], which was used in the Esterel v3 compiler. The *circuit semantics* translates an Esterel program into a constructive Boolean circuit. It is the basis of the current Esterel v5 implementation.

The translation into a circuit avoids the state explosion problem, which occurred when translating Esterel programs into explicit automata, as in Esterel v3. Circuits can be directly implemented in hardware, or they can be simulated for software implementation. The vast amount of work on circuit optimization and verification can be used to optimize and verify hardware or software implementations of Esterel.

The path from the constructive behavioral semantics to the circuit semantics is not a trivial one, and half of the book is devoted to its formalization and correctness proof. The final result we obtain is that an Esterel program is constructive if and only if its circuit electrically stabilizes for all gate and wire delays, this even if the circuit has combinational cycles. This result deeply roots our constructive approach in the physics of electronic designs.

The semantical study of Esterel we present is mostly implementation-oriented and it does not cover all possible fields of interest. All our semantics explain what happens within one reaction and handle sequences of reactions using state changes. We do not discuss *denotational semantics* where a program is directly seen as a flow transformer, forgetting about internal states. Such semantics are useful when dealing with compositionality issues and when mixing Esterel programs with programs written in synchronous data-flow languages, which is outside the scope of this book. Denotational semantics can be indirectly derived from the circuit translation, since defining the denotational semantics of circuits is easy. The reader interested in a direct denotational semantics can refer to G. Gonthier's thesis [18]. We do not discuss either *axiomatic semantics* where one is interested in characterizing term equality using algebraic rewrite rules. Such semantics are useful for program transformation and for proofs by rewriting. The axiomatic theory of Esterel remains to be done.

A Short Overview

The book consists of four parts: informal presentation, structural operational semantics, circuit translation, and implementation.

Part I: Informal Presentation

Chapter 2 describes the kernel of Pure Esterel: no-op statement "`nothing`", signal broadcasting "`emit S`", signal test "`present S then` p `else` q `end`", suspension "`suspend` p `when S`", sequencing "p`;` q", looping "`loop` p `end`", explicit concurrency "$p \,||\, q$", unit delay "`pause`", a "`trap`–`exit`" exception mechanism compatible with concurrency, and the local signal declaration "`signal S in` p `end`". The intuitive semantics describes instantaneous signal broadcasting and control transmission. Its cornerstone is the *signal coherence law*, which expresses that a signal is present in a reaction if and only if it is explicitly emitted in this reaction.

Chapter 3 discusses *logical correctness* of programs, which is the conjunction of *reactivity* and *determinism*. Reactivity is the ability to react to any input, while determinism is uniqueness of the reaction. Both properties are essential for synchronous controller programming. Logical correctness can be established by making all possible hypotheses about signal presence or absence and checking that exactly one hypothesis is consistent with the signal coherence law. Not all programs are logically correct: using *instantaneous feedback*, which is available in synchronous frameworks, one can write non-reactive programs such as "`present S else emit S`", which internally deadlocks as does the liar paradox, or non-deterministic programs such as "`present S then emit S`", which does not properly define its output, since both `S` present and `S` absent respect the signal coherence law. Reactivity and determinism are not built into the signal coherence law and the logical behavioral semantics; they must be added as additional conditions to be satisfied at each instant, which is not very satisfactory. We end the chapter by showing that bare logical correctness is not a nice and robust consistency notion: we exhibit a logically correct program whose unique behavior is absolutely counter-intuitive (program `P9`, page 33). In this program, signal information flows *backwards*, compared to sequential control, which is unacceptable on intuitive grounds.

A cheap way of rejecting non-reactive and non-deterministic programs is to reject instantaneous feedback altogether. This is what is generally done in synchronous circuits design and in synchronous data-flow languages, and we did it in the Esterel v4 compiler. Technically, one requires acyclicity of the static signal instantaneous dependency relation. Both previous examples are rejected right away since `S` depends on itself. However, this method has been dismissed by Esterel users as being too restrictive. In Esterel, one can easily and naturally write cyclic feedback dependencies, which raise no problem at run-time, and users do that. For example, if an instantaneous dependency $A \rightarrow B$ and the reciprocal dependency $B \rightarrow A$ appear respectively in the `then` and `else` branches of a test or are separated by a delay, it is immediately visible that only one dependency matters at a time, and there is no reason to reject the program. Users ask us to *control* feedback, not to *restrict* it[2]. Examples of useful cyclic programs are given in [7].

Chapter 4 informally presents the constructive approach, which exactly characterizes sensible feedback using a dynamic analysis instead of a static one. The analysis is performed separately for every program state and every input. It computes what a program *must* do or *cannot* do, based solely on pedestrian fact-to-fact propagation. The analysis is recursive in the signals and control. Intuitively, a signal `S` must be emitted if some "`emit S`"

[2] This is also the aim of the authors of [35]; we have not yet compared our techniques with theirs.

statement must be executed, and it cannot be emitted if no such statement can be executed. A signal is set present if and only if it must be emitted, and it is set absent if and only if it cannot be emitted. Facts about signals feedback to control: for example, in a test "`present S then` p `else` q", if `S` is known to be present, then p must be executed and q cannot be executed. In turn, these facts make it possible to determine new signal statuses, and so on. The constructive analysis accepts all sensible cyclic programs. it rejects the aforementioned logically correct but counter-intuitive program `P9`, since the unique consistent signal status cannot be computed by fact-to-fact propagation but only as a self-justified hypothesis.

In the terminology of proof theory [17], the logical behavioral semantics is an extensional logic of presence or absence values, while its constructive version is an intensional logic of presence or absence proofs. This is very much in the spirit of constructiveness in the Curry-Howard isomorphism [17], but in a much simpler setting. This analogy explains why we call our new semantics a constructive one.

Using simple examples, Chapter 4 briefly presents the different technical approaches to constructiveness we shall deal with. It also includes a comparison with previous attempts to solve the causality problem.

Part II: Structural Operational Semantics

All the formal semantics presented in this part define what happens in an instant. They are given in Plotkin's Structural Operational Semantics [34] inference-rule style. To enhance the readability of semantic definitions, we replace the original keyword-based syntax by an equivalent but terser process-calculus syntax, presented in Chapter 5 and first defined in [5].

Chapter 6 presents the formal definition of the *logical behavioral semantics* and defines what it means for a program to be reactive and deterministic. The inference rules define a reaction of a program P to an input I as an atomic transition $P \xrightarrow[I]{O} P'$, where O is the output and P' is the program that will execute the next reaction. The signal coherence law is embodied in two rules for local signal declaration, one for presence and one for absence. Reactivity and determinism are not built in since the two signal rules are not disjoint; they are imposed as additional conditions. Although the logical behavioral semantics accepts too many programs, it still plays an important safeguard role since any other semantics must be a refinement of it.

Chapter 7 presents the *constructive behavioral semantics*, which also defines atomic transitions $P \xhookrightarrow[I]{O} P'$. It is obtained by augmenting the logical behavioral semantics by the *must* and *cannot* predicates, which are defined inductively on statements. All the rules of the logical behavioral semantics are kept unchanged, except the signal presence rule, which receives the side-condition "`S` must be present", and the signal absence rule, which receives the side-condition "`S` cannot be present". The accepted programs are called *constructive* ones. We show that any constructive program is reactive and deterministic.

The simplicity of the logical and constructive behavioral semantics is offset by the need to change the program text from P to P' to handle the next reaction. In the *state* logical and constructive behavioral semantics presented in Chapter 8, we replace the rewriting by a much simpler marking of active delays in the original program, thereby defining a program state. This is just what we do in the Esterel v5 symbolic debugger. The state semantics are more practical, but the number of rules is roughly doubled. Both state semantics are shown to be equivalent to their rewriting counterparts, which proves in passing that Pure Esterel programs are finite-state.

Chapter 9 is devoted to the *constructive state operational semantics*. This semantics defines a reaction as a sequence of microsteps, where each microstep is either an elementary control transmission or the setting of a signal status. Parallel statements interleave the microsteps of their branches. The recursive computation of the *must* predicate is replaced by the actual execution of statements, which makes it much more efficient and somewhat more natural. However, the operational semantics is technically much more complex than the behavioral one. In particular, because of interleaving, there are several possible microstep sequences for a reaction. The main theorem refers to confluence and strong normalization: all microstep sequences yield the same result, in which all signal statuses are determined if and only if the program is constructive. It follows that the constructive operational semantics is equivalent to the constructive behavioral one.

Part III: Circuit Translation

In Chapter 10, we define the notion of a constructive circuit. Unlike classical circuits [21], which may not have cycles in their combinational part, constructive circuits are allowed to have combinational cycles. Constructiveness is exactly the characterization of safe cycles. We characterize constructive circuits in three equivalent ways: proof-theoretic, denotational, and electrical. Building up on work by Brzozowski and Seger [15], Shiple [38, 39] has shown the following fundamental result guessed by the author: a possibly cyclic circuit is logically constructive if and only if, in any physical implementation into wires and gates, all wires stabilize in bounded time to a voltage encoding either 0 and 1, this for any electrical propagation delay in the wires and gates[3]. The result implies that constructive cyclic circuits can be used exactly as acyclic ones in synchronous circuit design.

In Chapter 11, we informally present the basic structural translation idea: to translate a statement, we first translate its substatements and we connect them by appropriate gates and wiring. The translation is explained by pictures. It improves the original translation of [4], which turns out to be inaccurate in the constructive setting.

The basic translation is relatively simple, but it does not work for all programs because of a rather intricate phenomenon we call *schizophrenia*. The problem is due to the possibility of reincarnating a statement several times in a reaction by instantaneously looping loops. We could avoid schizophrenia by restricting the class of programs we translate, but we do not like the idea of restricting the language. In Chapter 12, we carefully analyze schizophrenia and we find a cure for it: logic duplication.

The final translation is formalized and proved correct in Chapter 13. There, we use a textual presentation of (hierarchical) circuits, since logic duplication makes pictures too difficult to draw. The main theorem is that constructiveness of an Esterel program is equivalent to constructiveness of its circuit. The proof idea is that propagation of 1's in the circuit mimics the computation of the *must* predicate, while propagation of 0's mimics the computation of the *cannot* predicate. In other words, the translation directly implements the semantic rules of the behavioral semantics using simple Boolean gates.

Part ??: Implementation

Yet to be written. Will be shortly presented: an interpreter for constructive circuits. A compiler that uses BDD algorithms to check for constructiveness. Hardware circuit optimization. Extension to the full language. Software implementation and optimization.

[3] In the up-bounded inertial delay model of [15].

Part I

Informal Presentation

Chapter 2

The Pure Esterel Kernel Language

This section describes the kernel language and its intuitive semantics. It can be skipped by the experienced Esterel user.

2.1 Signals and Reactions

Esterel deals with *broadcast signals*. In most of this book, we limit ourselves to the Pure Esterel sublanguage, where the information carried by a signal is limited to a *presence / absence status*. In the full Esterel language, signal can also carry values of arbitrary types.

A Pure Esterel program or *module* has a input-output signal interface and an executable body, which is an imperative statement:

```
module M:
input  names;
output  names;
 statement
end module
```

An *input event* specifies the presence / absence status of each input signal. A Pure Esterel program reacts to an input event by computing an *output event*, i.e. by assigning a status to each output signal. The reaction is conceptually instantaneous, and a reaction is also called an *instant*.

Reacting instantaneously is done repeatedly for input event sequences, also called *input histories*, thus generating *output histories*. The reaction to an input history is "sequential" in hardware terminology: an Esterel program has a *state*, which is implicitly encoded in its executable body. The reaction to an input event provokes a state change, and the reaction to the next input in an input history is computed from the new state.

2.2 Kernel Esterel Statement

Pure Esterel statements are divided into *kernel statements*, which form the Kernel Esterel primitive core of the language and *derived statements*, which make programming more convenient but are definable as combinations of kernel ones. As far as semantic issues are concerned, only kernel statements matter. The kernel language we consider is that of [5]:

```
nothing
emit S
pause
present S then p else q end
suspend p when S
p; q
loop p end
p || q
trap T in p end
exit T
signal S in p end
```

By default, '`;`' binds tighter than '`||`'. One can use brackets '`[`' and '`]`' to group statements in arbitrary ways. Both the `then` and `else` parts are optional in a `present` statement. If omitted, they are assumed to be `nothing`.

The statements are imperative and manipulate control flow and signal status. Most of them are classical in appearance. The `trap`-`exit` combination defines an exception mechanism fully compatible with parallelism. Traps are lexically scoped. The local signal declaration "`signal S in` p `end`" declares a lexically scoped signal `S`, which can be used for internal broadcast communication within p.

2.3 The Intuitive Semantics

We describe how signals are emitted and how control is transmitted between statements.

The status of an input signal is only determined by the input event. Explicit emission of an input signal by "`emit I`" is disallowed.

The status of a local or output signal is determined on a per-instant basis. In each instant, the status is absent by default. The only way to set a signal `S` present in an instant is to execute an "`emit S`" statement.

A statement can *start* in some instant; it then remains *active* within the instant and possibly for some further instants until it relinquishes control, either by terminating or by exiting a trap. The only way for a statement to stay active from one instant to the next one is to explicitly execute a `pause` statement, which pauses for exactly one instant. For example, consider the following statement:

```
trap T in
   loop
      present I then
         emit O;
         pause
      else
         exit T
      end
   end
end
```

When started, the statement emits `O` if `I` is present, and it reproduces this behavior until the first instant where `I` is absent, where it terminates instantaneously. Termination is provoked by executing the "`exit T`" statement, which provokes instantaneous termination of the enclosing "`trap T`" statement. Notice that the statement terminates instantaneously when started if `I` is initially absent. Notice also that the internal state encoded by the activity of the `pause` statement propagates from one instant to the next one.

Besides `pause`, all constructs propagate control in an instantaneous (or combinational) way. Signals are also broadcast and tested instantaneously. A statement that terminates or exits a trap in the same instant it starts is said to be *instantaneous*.

The intuitive semantics is defined by structural induction on statements:

- `nothing` terminates instantaneously.

- `pause` pauses in the current instant and terminates in the next instant. The pause statements act as state variables.

- An "`emit S`" statement instantaneously broadcasts the signal `S`, i.e. sets its status to present and terminates instantaneously. The emission of `S` is transient, i.e. valid for the current instant only.

- When a "`present S then` p `else` q `end`" statement starts, it immediately starts p if `S` is present in the current instant; otherwise it starts q if `S` is absent.

- In the "`suspend` p `when S`" suspension statement, `S` is called the *guard*. The guard controls execution of the body in each instant except for the first one, where it is ignored (masked in common terminology). In the initial instant, the body p is started; if p terminates or exits a trap, so does the `suspend` statement, and the guard is transparent. Interesting things only happen in the following instants if p paused in the first instant. Then, as long as p remains active, the guard signal `S` is tested for presence.

  - If `S` is present, then p is not executed in the instant and it is kept frozen for the next instant. The whole `suspend` statement pauses. In this case, we say that p is *suspended* for the instant.
  - If `S` is absent, then p receives the control for the instant. We say that p is *activated* for the instant. If p terminates or exits a trap, so does the `suspend` statement. If p pauses, then the `suspend` statement also pauses.

  In other words, when `S` is present, we "steal the clock" from p. Notice that a `suspend` statement remains active until its body terminates or exits a trap, which can only occur in the first instant or in successive instants during which `S` is absent.

- When started, a sequence "p`;` q" immediately starts p and behaves as such as long as p remains active. If and when p terminates, control is passed instantaneously to q, which determines the behavior of the sequence from then on. If and when p exits a trap `T`, so does the whole sequence, q being discarded in this case. Notice that q is never started if p always pauses. Notice also that "`emit S1; emit S2`" emits `S1` and `S2` simultaneously and terminates instantaneously.

- When started, "`loop` p `end`" immediately starts its body p. If and when p terminates, it is immediately restarted. If p exits a trap, so does the whole loop. The body of a loop is not allowed to terminate instantaneously when started; it must execute either a `pause` or an `exit` statement. A `loop` statement never terminates, but it is possible to escape from the loop by enclosing it within a trap and executing an `exit` statement.

- When started, a parallel statement "p `||` q" immediately starts p and q in parallel. The parallel statement remains active as long as one of its branches remains active, unless some branch exits a trap. The parallel statement terminates instantaneously if and when both p and q are terminated. The branches can terminate in different instants, the parallel waiting for the last one to terminate. Parallel branches may simultaneously exits traps. If, in some instant, one branch exits a trap `T` or both branches exit the same trap `T`, then the parallel exits `T`. If both statements exit distinct traps `T1` and `T2` in the same instant, then the parallel only exits the *outermost* of these traps, the other one being discarded.

- The statement "`trap T in` p `end`" defines a lexically scoped exit point `T` for p. When the `trap` statement starts, it immediately starts its body p and behaves as p until termination or exit. If p terminates, so does the `trap` statement. If p exits `T`, then the `trap` statement terminates instantaneously. If p exits an enclosing trap `U`, this exit is propagated by the `trap` statement.

- An "`exit T`" statement instantaneously exits the trap `T`. The corresponding `trap` statement is terminated, unless an outermost trap is concurrently exited (see the parallel and `trap` statements above).

- When started, the statement "`signal S in` p `end`" immediately starts its body p with a fresh signal `S`, overriding any that might already exist. The statement behaves as its body until termination or exit, except that the status of the local signal `S` is not exported.

2.4 Comparison with Previous Kernels

There are slight technical differences with the original kernel language and semantics of [11, 4]. We describe the differences and show how to recover the old kernel from the new one. This is an interesting exercise in Kernel Esterel programming.

- In the above presentation, a statement can exit at most one trap. If both branches of a parallel statement exits traps, then the parallel statement exits only the outermost one. In [11], a statement could exit a set of traps and the `trap` statement did the sorting. Both solutions are equivalent, but the new one has the technical advantage of yielding a nice numerical encoding of traps, see Chapter 5.

- The original `halt` statement, which pauses forever, is replaced by the `pause` statement, which pauses for one reaction only; `halt` is recovered as "`loop pause end`".

- The original kernel used the "`do` p `watching S`" watchdog statement, which behaves as its body p until the first following instant where `S` occurs, where the whole watchdog statement terminates without transferring control to p in this instant. The `watchdog` construct is used to abort the body when an event occurs. In the kernel, it is now replaced by the semantically simpler and more powerful "`suspend` p `when S`" suspension statement, which suspends its body instead of aborting it, see [5]. Recovering the "`do` p `watching S`" statement is a good example of kernel expansion:

```
trap T in
      suspend p when S;
      exit T
   ||
      loop
         pause;
         present S then exit T end
      end
end
```

  The first parallel branch suspends the watchdog's body when `S` is present, and, otherwise, it exits `T` if p terminates to propagate termination. The second branch makes the whole expansion terminate when `S` occurs. The instantaneous combination of suspension and trap exit has the intended effect.

- In [11, 4], we assumed the existence of an always present `tick` signal. The new `pause` statement could be written "`await tick`", i.e. "`do halt watching tick`". If needed, the `tick` can now be derived as a local signal by executing

```
loop
   emit tick;
   pause
end
```

in parallel with the module's body.

Chapter 3

Logical Correctness

The intuitive semantics specifies what should happen when executing a program, but it does not guarantee that an execution actually exists and is unique. Indeed, we shall need extra criteria for this to happen. In this chapter, we study the apparently simplest possible criterion, *logical correctness.*

In the intuitive semantics, we said that a signal `S` is absent by default and present if an "`emit S`" statement is executed. Let us rephrase this requirement in a more logical style, getting rid of imprecise notions such as "default" behaviors:

> **The Logical Coherence Law:** *A signal* `S` *is present in an instant if and only if an "*`emit S`*" statement is executed in this instant.*

Logical correctness is simply the requirement that there exists exactly one status for each signal that respects the coherence law. Of course, there is a strong relationship between signal status and control propagation: a signal status determines which branch of a `present` test is executed, which in turn determines which `emit` statements are executed.

We shall see that paradoxical statements can be written between control and signals. All the problems we shall mention appear only within instantaneous reactions. Program states do not add extra complexity; they will be ignored until we present the formal treatment, from Chapter 6 on.

We begin by intuitively defining logical correctness of Esterel programs. We then give examples of logically correct and incorrect programs. Finally, we show that composing several logically incorrect programs may result in a logically correct one.

3.1 Logical Correctness

Consider a fixed program and a fixed input event. Given a global status, i.e. a status for each signal of the program respecting the given input event, the flow of control is entirely determined and each `emit` statement is known to be either executed or not executed. Therefore, one can check whether the coherence law is respected for each signal: the global status is logically coherent iff at least one `emit` statement is executed for each signal assumed present and no `emit` statement is executed for each signal assumed absent.

We say that the program is *logically reactive* (resp. *logically deterministic*) w.r.t. the input event if there is at least (resp. at most) one logically coherent global status. We say that the program is *logically correct* w.r.t. the input event if it is both logically reactive and deterministic. We say that a program is logically correct if it is logically correct w.r.t. all possible input events.

Pure Esterel programs can be analyzed for logical correctness by performing exhaustive case analysis. Given the status of each input signal, one can make all possible assumptions about the global status and check them individually. Therefore, logical correctness is decidable.

3.2 Examples of Logically Correct Programs

Our first example is the following Esterel program `P1`:

```
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
end module
```

The program `P1` has no internal state, and only its first reaction matters. It is easy to check that `P1` is logically correct for all inputs:

- If `I` is present, the unique logically coherent assumptions are `S1` present, `S2` absent, and `O` absent. According to these assumptions, the first `present` statement takes its `then` branch, the second `present` statement takes its empty `then` branch, and the third `present` statement takes its empty `else` branch. The "`emit S1`" statement is executed, justifying the assumption `S1` present. No "`emit S2`" and "`emit O`" statement are executed, justifying the assumptions `S2` absent and `O` absent. It is easy to check that no other assumptions are logically coherent.

- If `I` is absent, the unique logically coherent assumptions are `S1` absent, `S2` present, and `O` present. According to these assumptions, the first `present` statement takes its empty `else` branch, the second `present` statement takes its `else` branch, and the third `present` statement takes its `then` branch. The "`emit S1`" statement is not executed, justifying the assumption `S1` absent. The "`emit S2`" and "`emit O`" statement are executed, justifying the assumptions `S2` present and `O` present. It is easy to check that no other assumptions are logically coherent.

Our second example `P2` is slightly more intricate since it involves a `pause` statement:

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

Notice that `P2` is inputless. Inputless programs react on empty input events, i.e. on "clock ticks". The only logically coherent set of assumptions is `S` present and `O` absent. The "`emit S`" statement is executed, justifying the assumption `S` present. The "`present O`" statement takes its empty `else` branch, which implies that the "`emit O`" statement is not executed, justifying the assumption `O` absent. All other assumptions are incoherent as follows. Since the "`emit S`" statement is executed whatever assumption we make, an assumption where `S` is absent is incoherent. Now, assume `S` present and `O` present. Then both `present` statements take their `then` branches and the `pause` statement is executed, which implies that the "`emit O`" statement is not executed in the instant, contradicting the assumption `O` present.

3.3 Examples of Logically Incorrect Programs

We now turn to logically incorrect programs. The following inputless program `P3` is the simplest example of a non-reactive program :

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

Clearly, no logically coherent assumption can be made for `P3`:

- Assuming `O` present is not justified, since the "`emit O`" statement is not executed.
- Assuming `O` absent is not justified, since the "`emit O`" statement is executed.

The next program `P4` is the simplest example of a nondeterministic program:

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

For `P4`, there are two logically coherent assumptions:

- Assuming `O` present is justified since the "`emit O`" statement is executed.
- Assuming `O` absent is justified since the "`emit O`" statement is not executed.

To make examples shorter, we omit input-output declarations from now on. Inputs will be written `I`, `I1`, etc., and outputs will be written `O`, `O1`, etc.

Logical coherence problems may involve several signals. The following program `P5` is non-reactive:

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

The case analysis shows that no assumption is logically coherent.

The following program P6 is nondeterministic:

```
module P6:
   present O1 then emit O2 end
||
   present O2 then emit O1 end
```

For P6, simultaneous presence or absence of O1 and O2 are logically coherent assumptions. It is interesting to try all combinations of then and else in the above examples; we leave this to the reader.

The next example P7 involves a pause statement:

```
module P7:
present O then pause end;
emit O
```

This program is non-reactive:

- Assume O absent. Then, the present statement takes its empty else branch and terminates. The "emit O" statement is executed, violating the assumption.

- Assume O present. Then, the present statement takes its then branch and pauses. The "emit O" statement is not executed, violating the assumption.

The example P8 below shows that coherence analysis can be quite intricate when traps are involved:

```
module P8:
trap T in
   present I else pause end;
   emit O
||
   present O then exit T end
end trap;
emit O
```

The program P8 is logically correct for I present, but it is logically nondeterministic for I absent.

- Consider the case `I` present. The only logically correct assumption is `O` present. Under this assumption, the "`present I`" statement takes its empty `then` branch and terminates, and the first "`emit O`" statement is executed, which suffices in justifying the assumption. The second `present` statement takes its `then` branch, the trap `T` is exited, the `trap` statement terminates, and `O` is reemitted, which is no problem. The opposite assumption `O` absent is incoherent since the first "`emit O`" is executed.

- Consider the case `I` absent. Whichever assumption we take for `O`, the pause statement of the first branch is executed, the enclosing `present` statement pauses, and the first "`emit O`" is not executed, Surprisingly, both assumptions `O` present and `O` absent are logically coherent:

  – Assume `O` present. Then, the second parallel branch exits the trap `T`, the `trap` statement terminates, and the last "`emit O`" statement is executed, which justifies the assumption.

  – Assume `O` absent. Then, the second parallel branch terminates without executing the `exit` statement and trap `T` is not exited. The parallel statement does not terminate since its first branch does not, and the `trap` statement does not terminate since its body does not. This implies that the last "`emit O`" statement is not executed. Since neither "`emit O` " statement is executed, the assumption `O` absent is coherent.

Replacing "`then`" by "`else`" in the second parallel branch would yield a program with no logically coherent behavior.

3.4 A Strange Logically Correct Program

Our last example `P9` shows that composing programs can lead to counter-intuitive phenomena. The program is as follows:

```
module P9:
   present O1 then emit O1 end
||
   present O1 then
      present O2 else emit O2 end
   end
```

The first parallel branch is a copy of the nondeterministic program `P4`, while the second branch contains a copy of the non-reactive program `P3` enclosed in an apparently innocuous "`present O1`" statement. Surprisingly enough, `P9` is reactive and deterministic, since there is only one logically coherent assumption: `O1` absent and `O2` absent. With this assumption, the first `present O1` statement takes its empty `else` branch, which justifies `O1` absent. The second "`present O1`" statement also takes its empty `else` branch, and "`emit O2`" is not executed, which justifies `O2` absent. Disproving the other assumptions is left to the reader.

Chapter 4

The Constructive Approach

Logical correctness is a sound semantics for Esterel. However, we have two strong reasons to reject it as the basis of the language. The primary reason is lack of fidelity to the programmer's intuition. A secondary but practically important reason is computational complexity. In this chapter, we present the constructive semantics, which solves both problems.

4.1 External Justification Versus Self-justification

In practice, programming in Esterel consists of analyzing input events to generate appropriate output signals, and using concurrent statements and intermediate local signals to create modular, well-structured programs. The programmer's natural way of thinking is in terms of information propagation by *cause and effect*. For example, in the statement

```
present I then
   emit O
end
```

the presence of `I` causes that of `O`, and the absence of `I` causes that of `O`.

Clearly, the programs `P1`, page 28, and `P2`, page 29, are well behaved as far as information propagation is concerned. For example, in `P1` with `I` present, information is propagated as follows: the `then` branch of the first `present` statement is taken, `S1` is emitted and therefore present; the `else` branch of the second `present` is not taken, and the only "`emit S2`" statement is not executed, which implies that `S2` is absent; the only "`emit O`" statement is not executed, and `O` is absent. There is no need to study the other assumptions about `S1`, `S2`, and `O`.

On the other hand, in the logically correct program `P9` that ends Chapter 3 (page 33), there is no natural information propagation. The logically coherent set of assumptions `O1` absent and `O2` absent happens to be *self-justified*: for instance, the `O1` absent assumption is valid since "`emit O1`" is not executed, a fact that itself follows from the starting assumption `O1` absent. Logical correctness of `P9` only follows from the fact that no other set of assumptions is self-justified.

In our opinion, accepting `P9` as correct is logically possible, but not in accordance with the *intention* of the language, i.e. with its intuitive semantics and with the intended sequential[1] character of test statements. In an imperative language such as Esterel, when we write "`present S then` p `end`", we obviously mean "*first* test the status of `S`, *then* execute p if `S` is present", assuming that the status of `S` should not depend on what p might do. The ordering implicit in the `then` word is not that of time, since everything is conceptually instantaneous, but that of sequential causality. We want to allow actual computation of the form "since `S` is present, we take the `then` branch" and to forbid speculative computation such as "if we assume `S` present, then we take the `then` branch". Aside from the explicit concurrency '`||`', all Esterel statements are sequential, and this character should be preserved in the semantics. Here is an example involving an explicit sequence operator '`;`':

```
module P10:
present O then nothing end;
emit O
```

`P10` is logically coherent with `O` present, but we do want to reject it: in the logical semantics, the information that `O` is present flows *backwards* across the sequencing operator, contradicting the basic intuition about sequential execution.

In the *constructive semantics*, the idea of checking assumptions about signal statuses is replaced by the idea of *propagating facts* about control flow and signal statuses. Obviously, the name is borrowed from *constructive logic*, in which one handles fact-propagating *proofs*, instead of handling values as in classical logic (see [17]). The analogy will be made formal in Chapter 7 and Chapter 9, where we shall present the new semantics as a constructive version of logical correctness.

Technically speaking, there are three equivalent ways to present the constructive semantics. We begin with the *constructive behavioral semantics*,

[1] In classical programming terminology, not in hardware terminology!

derived from the logical behavioral semantics by adding constructive restrictions to the logical coherence rule. It is the simplest way of defining the language. Then, we present the *constructive operational semantics* , which is based on an interpretation scheme expressed by term rewriting rules defining microstep sequences. It is the simplest way of defining an efficient interpreter. Finally, we briefly present the *circuit semantics*, which is based on a translation of programs into constructive circuits and is the core of the Esterel v5 compiler.

4.2 The Constructive Behavioral Semantics

The *constructive behavioral semantics* retains the spirit of the logical coherence semantics, but it adds reasoning about what a program *must* or *cannot* do, both predicates being disjoint and defined in a constructive way. These disjoint predicates express the following facts respectively:

- A statement must terminate, must pause, must exit a trap `T`, or must emit a signal `S`.
- A statement cannot terminate, cannot pause, cannot exit a trap `T`, or cannot emit a signal `S`.

The *must* predicate determines which signals are present and which statements are executed. The *cannot* predicate determines when signals are absent and it serves in pruning out false execution paths. The logical coherence law splits into two constructive sublaws, which are exclusive of each other, thus ensuring determinism at once:

- A signal is declared present if and only if it must be emitted.
- A signal is declared absent if and only if it cannot be emitted.

The predicates are defined by structural induction on statements, in a way that respects the sequential character of all primitives besides concurrency. In the recursive definition of the predicates, a signal can have three statuses: $+$, i.e. known to be present, $-$, i.e. known to be absent, or $\perp$, i.e. yet unknown. Technically, it is easier to define the *cannot* predicate as the negation of a *can* predicate; there is no constructiveness problem in taking such a negation since we only deal with finite sets. Here are some examples of inductive definitions:

- In a sequence "p`;` q", one must (resp. can) execute q if p must (resp. can) terminate.
- For a test "`present S then` p `else` q `end`", there are three subcases:
  - If `S` is known to be present, the test behaves as p.
  - If `S` is known to be absent, the test behaves as q.
  - If `S` is yet unknown, the test can do whatever p or q can do, and there is nothing it must do.

The main subtlety (and novelty) appears in the analysis of output or local signals. The analysis being the same in both cases, we consider a local signal declared by "`signal S in` p `end`".

First consider the *must* predicate. The idea is as follows. Assume we already know that we must execute "`signal S in` p `end`" in some signal context E that defines the status of visible signals. To find in which signal context p must be executed, we must compute the final status of `S`. We first analyze p in E augmented by setting the unknown status $\perp$ for `S`. If we find that `S` must be emitted, we propagate this information by re-analyzing p in E with `S` present, which may generate more information about the other signals. Similarly, if we find that `S` cannot be emitted, we re-analyze p in E with `S` absent; this is the only place where the *can* predicate is used.

For the *can* predicate, we just recursively analyze p with status $\perp$ for `S`. We cannot do any better without performing a speculative computation, since the *can* predicate can be computed for statements that must not be executed.

We refer to Chapter 7 for the other statements and the formal definitions.

4.2.1 Accepting Programs

Programs are accepted as constructive ones if and only if fact propagation using the *must* and *can* (or *cannot*) predicates suffices in establishing presence or absence of all signals. For example, the constructive behavioral analysis of `P1`, page 28, is as follows:

- If `I` is present, then the first `present` statement must take its first branch, emit `S1`, and terminate. From this, we deduce that `S1` is present. Then, the second `present` statement must take its (empty) `then` branch and cannot take its `else` branch. Since the "`emit S2`" statement cannot be executed, `S2` cannot be emitted, which implies that S2 is absent. Finally, the third present statement cannot take its then branch, which implies that O cannot be emitted and is absent.

- If I is absent, then the first present statement cannot take its first branch, and the "emit S1" statement cannot be executed, which implies that S1 is absent. Therefore, the second present statement must take its then branch, the "emit S2" statement must be executed, which implies that S2 is present. Finally, the third present statement must take its then branch, and the "emit O" statement must be executed, setting O present.

Consider now the program P2, page 29, which we recall here for readability:

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

We first start analyzing what the "signal S" statement must do with status $\perp$ for O. For this, we analyze its body with status $\perp$ for O and S. We immediately find that S must be emitted since we must execute the "emit S" statement. Therefore, we redo the analysis with status $\perp$ for O and $+$ for S. We reach the test for O. Since the status of O is unknown, there is nothing we must do and we can make progress only by analyzing what we cannot do in the branches of the test. In the then branch, there is a present test for S. Since S is known to be present, we cannot take the implicit else branch that would terminate. Since the then branch is a pause statement, it cannot terminate. Summing up things, the "present S" test cannot terminate. Therefore, the "emit O" statement cannot be executed and O cannot be emitted. As a consequence we must set O absent and redo the analysis of the program with status $-$ for O. We now find that we must take the implicit else branch of the "present O" test that terminates execution. The program is constructive since we have fully determined the signal statuses.

In the analysis, we never performed speculative computation based on *assumptions* about what we could do. We just propagated already established positive or negative facts. Fact propagation is monotonic: when a fact is established, it can never be contradicted later on.

Of course, the analysis involves tedious recomputation. Once we have set a signal status, we re-analyze the body of its declaration (the whole program for an output), and we re-establish facts we already know. The goal of the operational and circuit semantics is precisely to avoid recomputing known facts.

4.2.2 Rejecting Programs

Programs are rejected when the *must* and *cannot* predicates bring no information about the status of some signal. This is the case for the programs `P3`, page 30, and `P4`, page 30. For both programs, we find that `O` can be emitted since there are potentially reachable "`emit O`" statements, but that it is not true that `O` must be emitted. Since we cannot make any constructive progress on the status of `O`, we reject the programs. Notice that `P3` and `P4` are rejected for the very same reason by the constructive semantics, while they were rejected for two different reasons in the logical behavioral semantics, respectively, non-reactivity and non-determinism.

Generally speaking, all logically incoherent programs are rejected. Logically coherent programs can also be rejected as being non-constructive. This is the case for the strange example `P9`, page 33. For `P9`, the *can* analysis finds that both `O1` and `O2` can be emitted, and the *must* analysis finds that no signal must be emitted; since we cannot make any progress, we reject the program.

To understand further which programs are rejected, consider the following variant `P11` of `P2`:

```
module P11:
output O;
signal S in
   present O then
      emit S;
      present S then
         pause
      end;
      emit O
   end
end signal
```

The "`emit S`" statement is now inside the `then` branch of the "`present O`" statement. The *must* analysis with status $\perp$ for `O` and `S` finds nothing we must do since we are not allowed to speculatively compute within the branches of the test for `O`. In the same status, the *cannot* analysis finds that both `S` and `O` can be emitted since it finds potentially reachable emitters. Therefore, we can make no progress and we reject `P11`.

The following program `P12` is also rejected:

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

The constructive analysis finds that `O` must not be emitted since it is not allowed to speculatively execute the branches of the test. We shall explain strong physical (electrical) reasons to reject `P12` in Section 10.

4.3 The Constructive Operational Semantics

The constructive operational semantics is defined by a rewriting-based interpretation scheme, which gives a dynamic vision of program execution. Instead of reasoning about what we must do, we just do it. *A priori*, this looks like a better idea. However, the formal definition and technical treatment of the constructive operational semantics is much heavier than that of the constructive behavioral semantics. This is why we take the latter as the primary semantics.

4.3.1 Accepting Programs

We begin the intuitive explanation with the example of `P1` with input `I` present. First, we decorate the declaration of each signal with a status in $+$ (present), $-$ (absent), or $\perp$ (unknown). Initially, all signals except inputs are unknown, and the body of the program is started, which is indicated by a bullet:

```
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
```

The ternary parallel statement forks three execution threads:

```
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
```

Such a move of bullets is called a *microstep*, following the terminology of [26].

When encountering a "`present S`" statement in a thread, we proceed as follows. If `S` is annotated with $+$, we transfer control to the `then` branch. If `S` is annotated with $-$, we transfer control to the `else` branch. If `S` is annotated by $\perp$, we block until the status of `S` becomes $+$ or $-$. Here, only the first thread can continue. Since `I` is known to be present, we can take the `then` branch of the first test, rewriting the program as follows:

```
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
```

In the next microstep, we execute "`emit S1`" that sets `S1` present and terminates the first branch:

```
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
```

Next, we take the implicit `then` branch of the second `present`:

```
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
```

Since there is no more occurrence of "`emit S2`", we cannot emit `S2` and we set `S2` absent:

```
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 now take the implicit `else` branch of the last `present` statement:

```
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
```

Now, there are no occurrences of "`emit O`" reachable by the threads, and we cannot emit `O` any more. We set `O` absent as expected. Finally, we synchronize the three terminated threads to terminate the whole program:

```
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
•
```

The execution of `P1` with `I` absent is similar and left to the reader.

Let us now execute `P2`, page 29. We start from the following decorated statement:

```
output O⊥;
• signal S⊥ in
   emit S;
   present O then
      present S then
         pause
      end;
      emit O
   end
end signal
```

In three microsteps, we traverse the local signal declaration, we execute "`emit S`" that sets `S` present and we reach the test for `O` by traversing the sequencing operator:

```
output O^⊥;
signal S^+ in
   emit S;
 • present O then
      present S then
         pause
      end;
      emit O
   end
end signal
```

Since the status of `O` is $\perp$, control is frozen at the "`present O`" test and we can perform no further trivial microstep. We now perform a *cannot* analysis just as in the constructive behavioral semantics. The analysis reports that `O` cannot be emitted. We use a microstep to set it absent:

```
output O-;
signal S^+ in
   emit S;
 • present O then
      present S then
         pause
      end;
      emit O
   end
end signal
```

and we terminate execution by taking the implicit `else` branch of the test:

```
output O^-;
signal S^+ in
   emit S;
   present O then
      present S then
         pause
      end;
      emit O
   end
end signal
•
```

In the operational semantics, we avoid most of the recomputation that take place in the constructive behavioral semantics. The reason why we can do this is that statuses evolve in a monotonic way.

4.3.2 Rejecting Programs

Programs are rejected very much in the same way as in the constructive behavioral semantics, i.e. whenever no progress can be made on signal statuses. Consider for example `P3`, page 30. There is no possible initial microstep, which prevents us from setting the status of `O` to $+$, and there is a potential path to "`emit O`", which prevents us from setting the status of `O` to $-$. Since we cannot make progress, we reject the program. All logically incorrect programs are rejected in the same way, as is the logically correct strange program `P9`, page 33.

4.3.3 Summary of the Constructive Interpretation Scheme

The interpretation scheme handles sequential threads of control forked by parallel statements. Signals are shared objects having a three-valued status in $\{+, -, \perp\}$ and initialized to $\perp$, except for input signals, which are initialized according to the input event. The status of a signal `S` changes from $\perp$ to $+$ as soon as an "`emit S`" statement is executed and from $\perp$ to $-$ as soon as all the "`emit S`" statements have been found unreachable by the *cannot* false path analysis. When a thread reaches a "`present S`" statement, it remains there, frozen, as long as the status of `S` is $\perp$, and it can resume by taking the appropriate branch as soon as `S` has a non-$\perp$ status. If several threads are enabled, any one of them can be chosen. Threads are stopped by termination or by executing `pause` or `exit` statements, and parallel statements synchronize stopped threads, as explained in the intuitive semantics. Finally, the false path analysis explores all possible instantaneous paths towards `emit` statements, taking into account all facts established so far and making no speculative reasoning.

Given an input, a program is accepted if the analysis succeeds in setting each signal status to a defined value $+$ or $-$. Logical correctness is guaranteed for accepted programs.

It is not obvious that the result is independent of the order in which threads are executed and that the constructive operational semantics is equivalent to the constructive behavioral one. This will be proved in Chapters 9.

4.3.4 Comparison With Previous Operational Semantics

The constructive operational semantics can be seen as an improved version of the former operational semantics presented in [11] and used in the Esterel v3 compiler. Technically the improvement is brought by the finer *cannot* analysis that allows us to handle P2, page 29, which was rejected by the operational semantics of [11]. Semantically, the constructive operational semantics is fully equivalent to the constructive behavioral semantics, while the previous operational semantics lacked proper semantical characterization.

4.4 The Circuit Semantics

In the circuit semantics, we translate Esterel programs into Boolean digital circuits, i.e. systems of equations between Boolean variables.

In circuits, we deal with a set of Boolean variables split into disjoint subsets of input variables, output variables, and local variables. A circuit must specify an equation for each output or local variables, using either a Boolean expression or a *register* unit-delay operator that we shall ignore in this informal presentation, sticking to what is usually called combinational circuits. A Boolean solution of the equations defines a logical behavior.

4.4.1 The Logical Interpretation of Circuits

Given a circuit and an input, there may be no logical behavior, one behavior, or several behaviors. Therefore, we can define logical reactivity and determinism in the same way as for Esterel programs.

Let us re-examine some of the examples of the previous chapter in terms of behavior of Boolean circuits. The program P1, page 28 can be rewritten as follows:

```
circuit C1:
S1 = I
S2 =¬S1
O = S2
```

Since C1 defines an acyclic dependency relation between variables, it is trivial that C1 is logically reactive and deterministic. The program P3, page 30, corresponds to the following circuit C2:

```
circuit C2:
O = ¬O
```

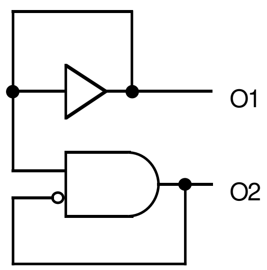


Figure 4.1: Circuit C9

This cyclic circuit is not reactive. Similarly, `P4`, page 30, corresponds to the non-deterministic cyclic circuit "`O = O`".

The program `P9`, page 33, and `P12`, page 41, corresponds to the cyclic circuits `C9` and `C12` below:

```
circuit C9:
O1 = O1
O2 = O1 ∧ ¬O2


circuit C12:
O = O ∨ ¬O
```

The circuits `C9` and `C12` are logically reactive and deterministic, as are `P9` and `P12`.

4.4.2 The Electrical Implementation of Circuits

Circuits are usually meant to be implemented using electrical wires and gates composed of appropriately wired transistors. The Boolean values 0 and 1 are represented by distinct voltages, say $0V$ and $5V$. Wires propagate voltages with some delay, and gates compute Boolean functions of voltages with some delay. Electrical circuits are generally pictured using conventional symbols[2]. For example, the circuits `C9` and `C12` are respectively pictured in Figure 4.1 and Figure 4.2. For an acyclic circuit, such as `C1`, it is trivial that the output voltage remain stable after some delay if the inputs are kept stable.

[2]See Figure 10.1, page 104.

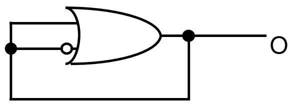


Figure 4.2: Circuit C12

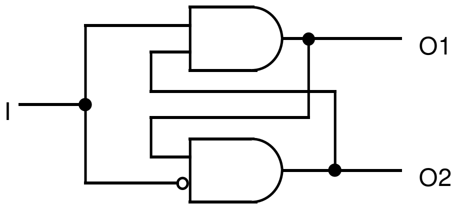


Figure 4.3: Circuit C13

Stabilization is much less trivial for cyclic circuits. For non-reactive or non-deterministic circuits, one can always find delays such that the outputs never stabilize. The worst case is that of `C2`, which never stabilizes. For the logically correct circuits `C9` and `C12`, one can find delays for which there is no output stabilization (left to the reader). This already means that *electrical stabilization is not the conjunction of reactivity and determinism.*

An example of a cyclic circuit that stabilizes for all delays is the following circuit `C13`, pictured in Figure 4.3:

```
circuit C13:
O1 = I ∧ O2
O2 = ¬I ∧ O1
```

In `C13`, the loop is electrically cut at one of the two `and` gates according to the value of `I`, since a single 0 on an `and` gate input is enough to generate a 0 on the output.

Notice that the non-stabilizing circuits `C9` and `C12` correspond to the non-constructive Esterel programs `P9` and `P12`, while the stabilizing circuit `C13` corresponds to the following constructive Esterel program `P13`:

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

This suggests to relate constructiveness and electrical stabilization. For this, we need a theory of constructive circuits parallel to that of constructive Esterel programs.

4.4.3 Constructive Circuits

The appropriate theory of constructive circuits is presented in [38]. There are three equivalent frameworks for defining circuit constructiveness:

- **Constructive Boolean Logic:** a circuit is constructive for an input if one can compute all the local and output variables by applying the following constructive Boolean laws:

$$\begin{gathered} \neg 0 = 1 \\ \neg 1 = 0 \\ 0 \wedge x = x \wedge 0 = 0 \\ 1 \wedge 1 = 1 \\ 1 \vee x = x \vee 1 = 1 \\ 0 \vee 0 = 0 \end{gathered}$$

  and by replacing a variable by its value when this value has been computed. Notice that the excluded middle law "$x \vee \neg x = 1$" is not valid in the constructive calculus unless x has been proved to be 0 or 1, which explains why `C12` is not constructive.

- **Denotational Semantics:** we interpret variables in Scott's ordered Boolean domain $B_\perp = \{\perp, 0, 1\}$. Given an input, the system of equation defines a monotonic (increasing) function f from tuples of local and output variables to tuples of local and output variables. The circuit is constructive for the input if the least fixpoint Y of f is fully defined, i.e. if the value of any variable in the tuple Y is different from $\perp$.

- **Electrical Semantics:** we say that a circuit is constructive if, for all gate and wire delays, all wire voltages stabilize in bounded time, provided that the input wires are kept stable at voltages encoding the input values.

The equivalence between these three semantics is a strong full abstraction result that relates logical, denotational, and operational views. Its real strength comes from the physical character of the electrical semantics.

4.4.4 From Esterel Programs to Constructive Circuits

The translation of Pure Esterel programs into circuits is a non-trivial process presented in Chapters 11 through 13. The translation we present extends the original translation of [4] by solving the schizophrenia problem, which we left unsolved in [4]. The solution involves performing appropriate logic duplication. All Esterel programs are now translated. To preserve constructiveness, the priority queue used in the original translation must also be modified in a quite subtle way, see Chapter 11.

The main result is that an Esterel program is constructive if and only if its circuit is. This implies that constructiveness in Esterel exactly corresponds to electrical stabilization: the full abstraction result for circuits also holds for Esterel.

4.5 Complexity Issues

In the application class towards which Esterel is targeted, namely hardware or software controllers, users definitely want fast simulation tools able to handle large programs involving hundreds or even thousands of signals. Given a program and an input, such a simulator should either perform the transition or report a semantical error such as non-reactivity or non-determinism. As far as full compilation goes, users can accept comparatively longer compilation and optimization times to generate efficient circuits or embedded software code, but they still want the compiler to take a reasonable amount of time.

Computing whether a program is constructively correct for a given input is efficient. For instance, the execution and false path pruning steps can be efficiently implemented using pointers from `emit` statements to signals and from signals to `present` statements. Because of the reincarnation problem mentioned in [6], the worst case cost of the analysis of a program of size n

is n^2. However, the squaring factors appears only for pathological examples, and the cost is roughly linear in practice.

In the Esterel v5 interpreter, we translate a program into a digital circuit and we run a linear-time constructiveness analysis algorithm for each input. Interpretation is very fast. In the Esterel v5 compiler, we analyze constructive correctness for all possible input. The easy case is when the circuit is acyclic, which is found in linear time. Then, the program is automatically correct. To analyze cyclic circuits, we use an algorithm based on Binary Decision Diagrams and presented in [38]. The algorithm is an extension of Malik's original algorithm [28] for sequential circuits. The algorithm checks for constructiveness, and, if it succeeds, returns an equivalent acyclic version of the circuit that can be implemented very efficiently. The current Esterel v5 compiler can handle cyclic programs of industrial size[3].

As far as logical correctness semantics is concerned, even the simplest problem of computing the reaction of a program to a given input is NP-complete. The exhaustive search method we have presented in Chapter 3 is obviously impractical since it is exponential in the number of signals. Much less naive and more efficient algorithms such as the one presented in [24] can be adapted to Esterel using the translation into circuits. Such heuristic algorithms can work reasonably well on medium-size examples, but their time and space resource requirements are somewhat unpredictable and they will not scale to real-size examples. This is not acceptable for an interpreter.

4.6 Constructive Symbolic Debugging

The constructive semantics fits well with symbolic debugging, a must for practical applications of the language. In the Esterel v5 symbolic debugger, we use a coloring scheme to show control and signal status propagation within a reaction. Executed statements and emitted signals are highlighted in green. Replacing green by underlining, the execution of `P1` with `I` present is pictured below:

[3]We are currently working on modular analysis of programs, but it is too early to report on the results.

```
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
```

The `signal` keyword is underlined since the signal statement is executed. The '`||`' statements are underlined since they fork the control. The three `present` statements are underlined since they are executed concurrently. The `then` keyword and branch of the first present are underlined since this branch is executed, while the branches of the other two `present` statements are in normal font since they are not executed. The signals `I` and `S1` are underlined since they are present. Since each control thread is strictly sequential, the picture gives complete information about what is executed and what is not.

With `I` absent, the execution is pictured as follows:

```
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
```

This form of symbolic debugging also applies to printing appropriate error messages for non-constructive programs. We can paint in red (here, italic) the frozen part of each thread and the signals with status $\perp$. For `P3`, page 30, the picture is

output O;
`present O` *else emit* `O end`

The `present` keyword is underlined since the `present` statement is executed. The declaration of `O` is in italic since the status of `O` is $\perp$, and the `then` and `emit` keywords are in italic to show where the thread is frozen.

4.7 Comparison with Previous Attempts

The Esterel v2 [9] and Esterel v4 compilers take a restricted topological approach to correctness. Write `S1` $\rightarrow$ `S2` if there is a potential direct control path from a "`present S1`" statement to an "`emit S2`" statement, and build a signal dependency graph by gathering all such arrows. The dependency graph is required to be acyclic[4]. All accepted programs are constructively correct.

The topological approach is usually considered adequate for data-flow synchronous languages [23, 20] and manually designed hardware circuits, but it is insufficient for Esterel since it does not take care of false control paths, of inputs, and of states. The correct program `P2`, page 29. is rejected since there is an arrow `O` $\rightarrow$ `O`. However, the control path from "`present O`" to "`emit O`" is a false one, which is detected by the constructive analysis. Program `P13`, page 50, is also rejected, since there is a static cycle between `O1` and `O2`. However, `P13` is constructive and accepted by Esterel v5.

The following constructive program `P14` is also rejected by both Esterel v2 and Esterel v4:

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

As in `P13`, there is a topological cycle between `O1` and `O2`, hence the rejection. However, the constructive analysis finds that this cycle is a false one, since the dependency `O1` $\rightarrow$ `O2` is only meaningful for the first reaction, while the dependency `O2` $\rightarrow$ `O1` is only meaningful for the second reaction; `P13` is accepted by Esterel v5.

In Esterel v3 [11], causality analysis is more elaborate and `P13` and `P14` are accepted. The analysis is performed on a per-state and per-input basis. Given a state and an input, an approximate analysis is run to detect what statements must and cannot do, just as in the constructive analysis. However, there is one difference: when analyzing a `present` statement that tests a yet unknown signal, the information known so far is not used to prune false branches (except for inputs). Because of this limitation, `P2`, page 29, is

[4]Esterel v4 actually builds a graph between signal *incarnations*, correctly handling the reincarnation problem presented in [6], which Esterel v2 did not handle.

rejected. All possible states and inputs are explored in a systematic way, building the whole state graph of the program. The main weakness of the technique is that the state graph can become exponential in the size of the program, limiting Esterel v3 to comparatively small programs.

Esterel v5 combines the advantages of Esterel v3 and Esterel v4, without having their drawbacks, while accepting more programs and accepting them for clear semantical reasons. There is no need to build an explicit state graph, as in Esterel v3. Nevertheless, the analysis is done on a per-input and per-state basis, thanks to symbolic BDD-based algorithms presented in [38]. Furthermore, the semantical characterization of accepted programs is constructiveness, which can be viewed in several natural and equivalent ways. Nothing more precise can be said without entering into the formal aspects, which we now do.

Part II

Structural Operational Semantics

Chapter 5

The Esterel Process Calculus Syntax

The keyword-based programming language syntax we have used so far is convenient for actual programming and for presenting examples, but we find it too heavy to be comfortably used in the framework of mathematical semantics. From now on, we use an equivalent terse syntax in which Esterel takes the look and feel of a process calculus. This syntax was first introduced in [5]. The kernel constructs are written as follows:

`nothing`	0
`pause`	1
`emit S`	$!s$
`present S then` p `else` q `end`	$s\,?\,p\,,q$
`suspend` p `when S`	$s \supset p$
$p; q$	$p\,;\,q$
`loop` p `end`	$p*$
$p \,\|\|\, q$	$p \mid q$
`trap T in` p `end`	$\{p\}$
	$\uparrow p$
`exit T`	k with $k \geq 2$
`signal S in` p `end`	$p \backslash s$

The main differences between the programming language syntax and the terse syntax are the removal of trap names, the introduction of an auxiliary *shift* operator $\uparrow p$ in conjunction with the trap operator $\{p\}$, and the use of integer *completion codes* $k \geq 0$ to encode the `nothing`, `pause`, and `exit` statements.

We first explain completion codes. The `nothing` statement is encoded by 0, the `pause` statement is encoded by 1, and the "`exit T`" statement is encoded by 2 if the closest trap declaration is that of `T`, and by $n+2$ if n trap declarations have to be traversed before reaching that of `T`. This is illustrated on the following example:

```
trap U in
      trap T in
            nothing
         ||
            pause
         ||
            exit T
         ||
            exit U
      end
   ||
      exit U
end
```

which is encoded as follows in the terse syntax:

$$\{\{0 \mid 1 \mid 2 \mid 3\} \mid 2\}$$

The first "`exit U`" statement is encoded by 3 since one must traverse the declaration of `T` to reach that of `U`, while the second "`exit U`" statement is encoded by 2 since, in its context, the declaration of `U` is the closest trap declaration.

The idea of the encoding is as follows. Each control thread returns an integer completion code $k \geq 0$ when it has completed its execution in the instant. The completion code is generated by executing a k statement, i.e. a `nothing`, `pause`, or `exit` kernel statement. Consider a parallel statement $p|q$. If p returns k and q returns l, the parallel statement returns the maximum $max(k,l)$ of k and l. This takes into account all the synchronization the parallel must perform:

- The parallel terminates if and when both branches have terminated, since $max(k,l) = 0$ is equivalent to $k = l = 0$.

- The parallel pauses if one branch pauses and the other one does not exit a trap, since $max(k,l) = 1$ implies $k = 1$ and $l \leq 1$ or vice-versa.

- The parallel exits a trap if one branch does and exits the outermost trap only if both branches exit traps. This follows from the encoding of exit statements that starts from 2 and increases by 1 for each enclosing trap.

Having lost trap names in the terse syntax, we need the auxiliary $\uparrow p$ shift operator, whose semantical effect is to increment by one all the completion codes greater than 1 in p, thus ensuring that $\{\uparrow p\}$ is behaviorally equivalent to p. To illustrate the need for that operator, consider the Esterel `watching` derived statement whose kernel expansion was given in Chapter 2, page 24. In the terse syntax, "`do` p `watching S`" is written $s >\!\!> p$, and its kernel expansion is rewritten as follows:

$$s >\!\!> p \quad = \quad \{(s \supset \uparrow p) \mid (1\,;\, s\,?\,2\,,0)*\}$$

Since the expansion adds an extra trap, it is necessary to write $\uparrow p$ instead of p to ensure correct propagation of the exit statements internally executed by p.

In addition to the above kernel statements, we shall use the *immediate suspension* derived statement written "`suspend p when immediate S`" in the language syntax and $s \mathbin{\dot{\supset}} p$ in the terse syntax. This statement differs from $s \supset p$ only for the starting instant. In this instant, the signal s is also tested and p does not receive the control if s is present. The kernel expansion is as follows:

```
trap T in
   loop
      present S then
         pause
      else
         exit T
      end present
   end loop
end trap;
suspend p when S
```

The loop encodes the full Esterel statement "`await immediate [not S]`". In terse syntax, the definition is

$$s \mathbin{\dot{\supset}} p \quad = \quad \{(s\,?\,1\,,2)*\}\,;\, s \supset p$$

which means "wait until the first instant where s is absent, then execute $s \supset p$". Finally, we impose a minor restriction on programs that makes semantical definitions simpler: an input signal cannot be internally emitted, i.e. one cannot write $!i$ if i is an input. This restriction ensures that input statuses are defined by the environment only. Any program that does not obey this restriction can be easily recoded using extra local signals (use one local signal per internally emitted input).

Chapter 6

The Logical Behavioral Semantics

This chapter is devoted to the formalization of the logical behavioral semantics. Although we reject this semantics as the basis of Esterel, we want to describe it formally for three reasons. First, it formally defines reactivity and determinism, which are the minimal requirements that any other semantics must obey. Second, it is mathematically simple and elegant. Third, in Chapter 7, we shall present the constructive behavioral semantics as a simple refinement of the logical one, leaving most rules unchanged.

6.1 Events

Given a set S of signals, also called a *sort*, an *event* E is the definition of a status b in $B = \{+, -\}$ for each signal. The sort of E is written $\mathcal{S}(E)$. Technically, we consider E either as the subset of S containing all signals having status $+$, or as a mapping from S to B. The status of s in E is written $E(s)$. We write $s^+ \in E$ (resp. $s^- \in E$) if the status of s in E is $+$ (resp. $-$). We write $E \subset E'$ if $s^+ \in E$ implies $s^+ \in E'$ for any signal s. Given a signal s, the *singleton event* $\{s^+\}$ is defined by $\{s^+\}(s) = +$ and $\{s^+\}(s') = -$ for $s' \neq s$.

Given a signal set S and a signal $s \in S$, we write $S \backslash s = S - \{s\}$. Given E and $s \in \mathcal{S}(E)$, we write $E \backslash s$ to denote the event of sort $\mathcal{S}(E) \backslash s$ which coincides with E on all signals but s.

Given an event E of sort S, a signal s possibly not in S, and a status b in B, we define $E * s^b$ as the event E' of sort $S \cup \{s\}$ defined by $E'(s) = b$

and $E'(s') = E(s')$ for $s' \neq s$. This construct is useful for signal scoping. Notice that the status of s in E is lost in $E * s^b$ if $s \in \mathcal{S}(E)$: although declarations of local signals having the same name can be nested, as in $(p\backslash s)\backslash s$, the event E does not need to be a stack since the outermost signal s is not visible in p.

6.2 Program and Statement Transitions

The behavioral semantics formalize a reaction of a program P as a *behavioral transition* of the form

$$P \xrightarrow[I]{O} P'$$

where I and O are respectively an *input event* and an *output event*. The resulting program P' is called the *derivative* of P by the reaction. It represents the new state reached by P after the reaction. Coding states by program texts is standard in process calculi definitions based on Structural Operational Semantic (SOS) rules [34, 31], which is the style we use here.

Reactions are computed using an auxiliary *statement transition* relation, which has the following form:

$$p \xrightarrow[E]{E', k} p'$$

Here, E is an event that defines the status of all signals declared in the scope of p, i.e. an assumption in the sense of Chapter 4, E' is an event composed of all the signals emitted by p in the reaction, k is the completion code returned by p, and the statement p' is called the *derivative* of p by the reaction. The statement transition relation is defined by structural induction on statements according to the rules given below.

Given a program P of body p and an input event I, the program transition of P is defined in terms of the statement transition of p in the following way:

$$P \xrightarrow[I]{O} P' \text{ iff } p \xrightarrow[I \cup O]{O, k} p' \text{ for some } k$$

This definition exactly expresses what we called the logical coherence of the global event $I \cup O$ in Chapter 3.

Definition: The program P is *logically reactive* (resp. *logically deterministic*) w.r.t. I if there exists at least (resp. at most) one program transition $P \xrightarrow[I]{O} P'$. It is *logically correct* if it is logically reactive and logically deterministic.

6.3 Trap Propagation and Completion Codes

To handle trap propagation, we introduce two operators on completion codes. The $\downarrow k$ operator is used to compute the completion code of $\{p\}$ from that of p, while the $\uparrow k$ operator is used to compute the completion code of $\uparrow p$ from that of p:

$$\downarrow k \quad = \quad \begin{cases} 0 & \text{if } k = 0 \text{ or } k = 2 \\ 1 & \text{if } k = 1 \\ k-1 & \text{if } k > 2 \end{cases}$$

$$\uparrow k \quad = \quad \begin{cases} k & \text{if } k = 0 \text{ or } k = 1 \\ k+1 & \text{if } k > 1 \end{cases}$$

The most important rule is $\downarrow 2 = 0$: it indicates that $\{p\}$ terminates when p exits the trap, i.e. returns code 2. If p exits an enclosing trap by returning a code $k > 2$, the exit is propagated as $k-1$ to the upper trap since one level of trap has been traversed.

6.4 The Logical Behavioral Semantics Rules

The behavioral rules are as follows:

$$k \xrightarrow[E]{\emptyset,\, k} 0 \qquad \textit{(compl)}$$

$$!s \xrightarrow[E]{\{s^+\},\, 0} 0 \qquad \textit{(emit)}$$

$$\frac{s^+ \in E \qquad p \xrightarrow[E]{E',\, k} p'}{s\,?\,p\,,q \xrightarrow[E]{E',\, k} p'} \qquad \textit{(present+)}$$

$$\frac{s^- \in E \qquad q \xrightarrow[E]{F',\, l} q'}{s\,?\,p\,,q \xrightarrow[E]{F',\, l} q'} \qquad \textit{(present}-\textit{)}$$

$$\frac{p \xrightarrow[E]{E',k} p' \qquad k \neq 0}{s \supset p \xrightarrow[E]{E',k} s \supset p'} \qquad \textit{(susp1)}$$

$$\frac{p \xrightarrow[E]{E',0} p'}{s \supset p \xrightarrow[E]{E',0} 0} \qquad \textit{(susp2)}$$

$$\frac{p \xrightarrow[E]{E',k} p' \qquad k \neq 0}{p\,;\,q \xrightarrow[E]{E',k} p'\,;\,q} \qquad \textit{(seq1)}$$

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

$$\frac{p \xrightarrow[E]{E',k} p' \qquad k \neq 0}{p* \xrightarrow[E]{E',k} p'\,;\,p*} \qquad \textit{(loop)}$$

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

$$\frac{p \xrightarrow[E]{E',k} p' \qquad k = 0 \text{ or } k = 2}{\{p\} \xrightarrow[E]{E',0} 0} \qquad \textit{(trap1)}$$

$$\frac{p \xrightarrow[E]{E',k} p' \qquad k = 1 \text{ or } k > 2}{\{p\} \xrightarrow[E]{E',\downarrow k} \{p'\}} \qquad \textit{(trap2)}$$

$$\frac{p \xrightarrow[E]{E',k} p'}{\uparrow p \xrightarrow[E]{E',\uparrow k} \uparrow p'} \qquad \textit{(shift)}$$

$$\frac{p \xrightarrow[E*s^+]{E'*s^+,k} p' \qquad \mathcal{S}(E') = \mathcal{S}(E)\backslash s}{p\backslash s \xrightarrow[E]{E',k} p'\backslash s} \qquad \textit{(sig+)}$$

$$\frac{p \xrightarrow[E*s^-]{E'*s^-,k} p' \qquad \mathcal{S}(E') = \mathcal{S}(E)\backslash s}{p\backslash s \xrightarrow[E]{E',k} p'\backslash s} \qquad \textit{(sig−)}$$

It is important to notice that all rules build a *single* statement transition, possibly as a function of several simultaneous statement transitions. This is how we handle synchrony.

If the return code k for a statement p is 0, i.e. if p terminates, then p' will always behave as 0 (`nothing`) in further instants. If k encodes a trap exit, i.e. if $k > 1$, the resulting statement p' is immaterial since it will always disappear by some application of rule *(trap1)*. The rules exactly formalize the intuitive semantics given in Chapter 2:

- The rules *(compl)*, *(emit)*, *(present+)*, and *(present-)* are trivial.
- In the *(susp1)* rule, we transfer control to p and generate an immediate suspension $s \,\underline{\supset}\, p'$ as the derivative for next instant if p does not terminate; remember that $s \,\underline{\supset} p$ is not a kernel statement but an abbreviation of $\{(s\,?\,1\,,2)*\}\,;\, s \supset p$, see Chapter 5, page 61. If p terminates, we use rule *(susp2)* to explicitly return the 0 derivative: we cannot return $s \,\underline{\supset} 0$ as in rule *(susp1)*, since this term would pause in the following instant if s were present, preventing termination of a term such as $(s \supset 0) \mid 1$.
- The *(seq1)* rule expresses that a sequence pauses if p pauses and that

the sequence propagates the traps exited by p. The *(seq2)* rule expresses that control is instantaneously transferred to q if p terminates.

- The *(loop)* rule expands a loop once and requires the loop's body not to terminate instantaneously.
- The *(parallel)* rule performs the synchronization by using the $max(k, l)$ maximum operation, as we explained in Chapter 5.
- The *(trap1)* rule expresses that a trap terminates if its body terminates or exits the trap. The *(trap2)* rule expresses that a trap pauses if its body pauses (case $k = 1$) and that a trap propagates exits to outer traps (case $k > 2$). In rule *(trap1)* with $k = 2$, we must explicitly return 0 as a derivative instead of $\{p'\}$, otherwise the term $\{2 \mid 1\} \mid 1$ would not terminate in the second instant.
- The *(shift)* rule is trivial.
- The *(sig+)* and *(sig-)* rules formalize the signal logical coherence law. In *(sig+)* and *(sig-)*, the additional sort condition expresses that the sort of E' does not contain s, which is handled specifically by an $E' * s^b$ operation. This is necessary to avoid propagating the local status of s outside the $p \backslash s$ statement.

6.5 Reactivity and Determinism

Here is the formal definition of reactivity and determinism:

Definition: A program P is *reactive* w.r.t. an input event I if there exists a output event O and a program P' such that $P \xrightarrow[I]{O} P'$. The program is *deterministic* w.r.t. I if there exits at most one such output event and program. The program is *logically correct* w.r.t. I if it is reactive and deterministic w.r.t. I. The program is *logically correct* if it is logically correct for all input events and if its derivatives are logically correct.

Input-output determinism leaves some room for internal non-determinism. For example, consider the program $(s\,?\,!s\,,0)\backslash s$. This inputless and outputless program is deterministic: it has no output and can only be rewritten into $0\backslash s$. However, it is internally non-deterministic since the local signal s can be either emitted or not emitted. In constructive semantics, we shall forbid internal non-determinism, in the same way one forbids type-check errors even in dead code in classical languages. For this, we define strong determinism as follows:

Definition: A program P is *strongly deterministic* for an input event I if it is reactive and deterministic for this event and if, furthermore, there exists a unique proof of the unique transition $P \xrightarrow[I]{O} P'$.

6.6 Loop-Safe Programs

In rule *(loop)*, there is a side condition $k \neq 0$ to prevent instantaneous loops that would execute infinitely often their body in the instant. It is often useful to put a static restriction on programs to make instantaneous loops impossible and the side condition superfluous. For this, we define the set $K(p)$ of *potential completion codes* of p. First, we extend the $max(k,l)$, $\downarrow k$, and $\uparrow k$ operations to sets K, L of completion codes as follows:

$$
\begin{array}{rcl}
Max(K,L) & = & \left\{ \begin{array}{ll} \emptyset & \text{if } K = \emptyset \text{ or } L = \emptyset \\ \{max(k,l) \mid k \in K,\ l \in L\} & \text{if } K, L \neq \emptyset \end{array} \right. \\[2ex]
\downarrow K & = & \{\downarrow k \mid k \in K\} \\[2ex]
\uparrow K & = & \{\uparrow k \mid k \in K\}
\end{array}
$$

(The cases $K = \emptyset$ and $L = \emptyset$ in the definition of Max are not needed here, but they will be needed in Chapter 7.) Write $K\backslash 0$ for the set $\{k \in K \mid k \neq 0\}$. Then, $K(p)$ is defined as follows:

$$
\begin{array}{rcl}
K(k) & = & \{k\} \\
K(!s) & = & \{0\} \\
K(s\,?\,p\,,q) & = & K(p) \cup K(q) \\
K(s \supset p) & = & K(p) \\
K(p\,;\,q) & = & \left\{ \begin{array}{ll} (K(p)\backslash 0) \cup K(q) & \text{if } 0 \in K(p) \\ K(p) & \text{otherwise} \end{array} \right.
\end{array}
$$

$$
\begin{aligned}
K(p*) &= K(p)\backslash 0 \\
K(p \mid q) &= Max(K(p), K(q)) \\
K(\{p\}) &= \downarrow K(p) \\
K(\uparrow p) &= \uparrow K(p) \\
K(p\backslash s) &= K(p)
\end{aligned}
$$

The next lemma shows that $K(p)$ is a superset of the set of completion codes a statement can return.

Lemma 1 *If* $p \xrightarrow[E]{E',k} p'$, *then* $k \in K(p)$.

We can now define loop-safe programs.

Definition: A program P of body p is *loop-safe* if, for each substatement $q*$ of p, one has $0 \notin K(q)$.

In practice, loop-safety is not a very restrictive condition and it makes life easier. However, we know of one case where users find it a little annoying. Assume that two input signals `I` and `J` are known to be incompatible, i.e. never present in the same instant. In full Esterel, this is asserted by writing

```
relation I#J;
```

Then, the following loop-unsafe program is obviously correct because the direct path from `loop` to "`end loop`" cannot be taken:

```
loop
   present I else
      p % non instantaneous
   end present;
   present J else
     q  % non instantaneous
   end present
end loop
```

The solution to make the program loop-safe is to add a `pause` statement that will never be reached:

```
loop
   present I else
      p % non instantaneous
   end present;
   present J then
      pause  % unreachable
   else
     q  % non instantaneous
   end present
end loop
```

Chapter 7

The Constructive Behavioral Semantics

In the logical behavioral semantics, given a program P of body p and an input event I, there are exactly two places where we must make signal status assumptions: when searching for an output event O such that $p \xrightarrow[I \cup O]{O,\,k} p'$, and when handling the local signal declarations that occur in p. In the constructive semantics presented in the sequel, we constructively enforce the choice of determinate statuses at the same places, in a way that ensures determinism by causal information propagation.

We shall present three different but equivalent views of the constructive semantics.

- The *constructive behavioral semantics* presented in this chapter uses the same semantic rules as the behavioral semantics, but it disambiguates the search for signal statuses by adding predicates telling whether signals *must* or *cannot* be emitted. Signals are declared present if they must be emitted, absent if they cannot be emitted. The predicates are computed recursively in a way that forbids speculative computation. Given an input event, if the calculation determines a status for each signal, then the program is said to be constructive w.r.t. the event. Then, reactivity holds by construction and determinism follows from the disjointness of the *Must* and *Cannot* predicates. If the calculation fails, the program is rejected as non-constructive.

- The *constructive operational semantics* presented in Chapter 9 is a more conventional microstep semantics acting on states. However, this semantics is technically much more intricate since the microsteps are relational and non-deterministic. A direct correctness proof would require proving very delicate confluence and strong normalization properties, such as the ones proved by Gonthier [18] for a former version of the operational semantics. Here, we avoid these difficulties by proving equivalence of the relational microstep semantics and of the functional constructive behavioral semantics.

- In the *constructive circuit semantics* presented in Chapter 11 and Chapter 13, we translate an Esterel program into a Boolean sequential circuit that we interpret in a constructive (i.e. electrical) way. We show that propagation of 1's and 0's in the circuit's wires exactly performs the *must* and *cannot* computations and that program states are appropriately encoded into memory states. The circuit semantics avoids all the computational redundancies of the constructive behavioral semantics and yields very efficient hardware and software implementations of Esterel, which will be discussed in Chapter **??**.

7.1 The Must and Cannot Analysis

We start by an intuitive presentation of the analysis. We then define the functions $Must(p, E)$ and $Cannot^m(p, E)$, where p is a statement and E is a (partial) event. We discuss other possible choices for *Must* and *Cannot* and we explain why we discard them. We present the constructive behavioral rules, which differ from the logical ones only by adding appropriate predicates in three places. Finally, we show that constructive programs are reactive and deterministic, i.e. that the constructive behavioral semantics is logically correct.

7.1.1 Intuitive Presentation

Consider the following statement `P15`:

```
    present I then
       emit O1;
    end
||
    present J then
       present O1 then emit O2 else emit O3 end
    end
```

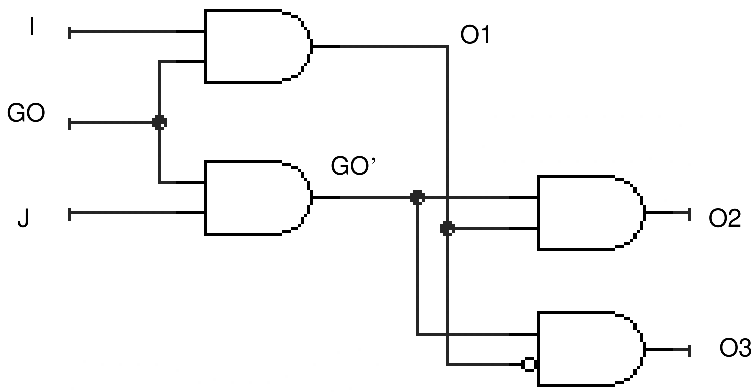


Figure 7.1: Circuit for `P15`

Assume first that the statement is a full program body with `I` and `J` inputs. Consider the case `I` and `J` present. Then, since we must execute "`present I`", we must execute "`emit O1`", which implies that `O1` must be present. We must execute "`present J`", "`present O1`", and "`emit O2`". Therefore, `O2` must be present. We cannot execute "`emit O3`", which implies that `O3` must be absent since it cannot be emitted. All the signal statuses are determined. In the case `I` absent, `J` present, `O1` and `O2` cannot be emitted and are absent while `O3` must be emitted and is present. The other cases are similar.

Things become more complex if we consider the `P15` statement as a fragment of a larger program, in a context where `I` and `J` are not inputs but local signals the statuses of which are also being computed (to avoid interference, we assume that there is no other emitter of `O1`, `O2`, `O3` in the global program). In this case, we must also explain how we propagate *must* and *cannot* information when the statuses of `I` and `J` are yet unknown. There are two subcases, depending on whether we already know from the recursive reasoning context whether we must execute the statement; the cases are *yes* or *unknown*, since we shall never need to analyze statements known not to be executed.

Assume first that we know that we must execute the statement. Assume `I` present and `J` unknown. Then, "`emit O1`" must be executed and `O1` must be present. However, since `J` is unknown, there is nothing we must do in the second parallel branch, and, in particular, we cannot deduce any status for `O2`. However, we have enough information to deduce that the "`emit O3`" statement is on a false path since `O1` is know to be present. Therefore, we safely deduce that we cannot emit `O3`. The circuit equivalent of `P15`, shown in Figure 7.1, gives another view of this reasoning. Saying that `P15` must be executed is saying $\mathtt{GO} = 1$. Since $\mathtt{I} = 1$, one has $\mathtt{O1} = 1$, which is just enough to deduce $\mathtt{O3} = 0$, although $\mathtt{GO}'$ remains unknown. No value propagates to the `O2` wire.

Assume now that we do not yet know whether `P15` statement must be executed. Then, there is nothing we must do since we may not compute speculatively. However, we can still prune out false paths and compute what we cannot do. If `I` is yet unknown, no information propagates. If `I` is known absent, then we can constructively deduce that `O1` cannot be emitted, which in turn implies that `O2` cannot be emitted. In the `P15` circuit, $\mathtt{I} = 0$ is enough to deduce $\mathtt{O1} = \mathtt{O2} = 0$, even when `GO` is unknown.

In the *cannot* analysis, knowing whether a statement must be executed in fundamental. For example, in

```
emit S;
present S then emit O1 else emit O2 end
```

we can deduce that `O2` cannot be emitted only if we know that the statement must be executed. Deducing the same while not knowing whether the statement must be executed would be speculatively reasoning about "`emit S`".

So far, we have only reasoned about *Must* and *Cannot*. Technically, it is often easier to reason about the complement $\overline{Cannot}$ of *Cannot*, which we call simply *Can*. Therefore, we say "p can emit `S`" for "we cannot prove that p cannot emit `S`". Beware, this shortcut can sometimes be slightly misleading. In case of doubt, always translate *Can* into the double negation $\overline{Cannot}$.

7.1.2 The Must, Cannot, and Can Functions

The *Must* function determines what must be done in a reaction $P \xrightarrow[I]{O} P'$, abstracting away the derivative P' that will be recovered using the behavioral semantics rewrite rules. The *Must* function has the following form:

$$Must(p, E) \quad = \quad \langle\, S \,,\, K \,\rangle$$

where E is a *partial event* that associates a status in $B_\perp = \{+, -, \perp\}$ with each signal, where S is the set of signals that p must emit, and where K is the set of completion codes that p must return. For *Must*, the set K is either empty, if we cannot derive any *Must* information, or a singleton $\{k\}$, if we can derive that p must return k. It is impossible that p must return more than one completion code.

To distinguish between the elements of the result pair, we use s and k subscripts:

$$Must(p,E) \quad = \quad \langle\, Must_s(p,E)\,,\, Must_k(p,E)\,\rangle$$

The inclusion predicate $\subseteq$ and the union operator $\cup$ are extended componentwise on pairs $\langle\, S\,,\, K\,\rangle$.

The function $Cannot^m(p,E)$ is used to prune out false paths. Its type is similar to that of *Must*, but with an extra argument m added in exponent:

$$Cannot^m(p,E) \;=\; \langle\, Cannot_s^m(p,E)\,,\, Cannot_k^m(p,E)\,\rangle \quad = \quad \langle\, S\,,\, K\,\rangle$$

Here, the results S and K are respectively the set of signals that p cannot emit and the set of completion codes that p cannot exit when the input event is E. The extra argument $m \in \{+,\perp\}$ tells whether it is known that the statement p must be executed in the event E, as explained before. It is recursively provided by the context when a statement is analyzed. The case $m = -$ will never occur in the recursion since *Cannot* will only be called for potentially executable statements.

Technically, it is simpler to define the set complement $Can^m(p,E)$ of $Cannot^m(p,E)$. The complementation is done componentwise, w.r.t. the set of visible signals for the signal part and w.r.t. the set of potential completion codes for the completion part. The set $Can_k^m(p,E)$ may be any subset of the potential completion set $K(p)$ defined in Section 6.6. It can be strictly smaller since we take signal information into account. In case of problems about the intuition of *Can*, always remember it really means $\overline{Cannot}$.

7.1.3 The Definition of Must and Can

The definitions are trivial and identical for completion codes and signal emissions. The m argument is unused here.

$$\begin{array}{rcl} Must(k,E) = Can^m(k,E) & = & \langle\, \emptyset\,,\, \{k\}\,\rangle \\ Must(!s,E) = Can^m(!s,E) & = & \langle\, \{s\}\,,\, \{0\}\,\rangle \end{array}$$

Consider now a signal test $s\,?\,p\,,q$ and a partial event E. First, for both *Must* and *Can*, the definitions are easy if the status of s in E is either $+$ or $-$:

we recursively analyze the first branch if the status is $+$ and the second branch if the status is $-$, with the same m argument. If the status of s in E is unknown, *Must* and *Can* behave very differently. For *Must*, we return the empty signal set and the empty completion code set since none of the branches must be taken. For *Can*, we return the union of the signals the branches can emit and the union of the completion codes they can return, computed with $m' = \perp$ since no branch must be taken.

$$\mathit{Must}\Big((s\,?\,p\,,q), E\Big) \;=\; \begin{cases} \mathit{Must}(p, E) & \text{if } s^+ \in E \\ \mathit{Must}(q, E) & \text{if } s^- \in E \\ \langle\, \emptyset\,,\, \emptyset\,\rangle & \text{if } s^\perp \in E \end{cases}$$

$$\mathit{Can}^m\Big((s\,?\,p\,,q), E\Big) \;=\; \begin{cases} \mathit{Can}^m(p, E) & \text{if } s^+ \in E \\ \mathit{Can}^m(q, E) & \text{if } s^- \in E \\ \mathit{Can}^\perp(p, E) \cup \mathit{Can}^\perp(q, E) & \text{if } s^\perp \in E \end{cases}$$

Suspension is trivial, since, in an instant, a suspension acts as its body:

$$\begin{aligned} \mathit{Must}(s \supset p, E) &= \mathit{Must}(p, E) \\ \mathit{Can}^m(s \supset p, E) &= \mathit{Can}^m(p, E) \end{aligned}$$

For a sequence $p\,;\,q$, we analyze q only if p must (resp. can) terminate, in which case the completion code 0 of p is discarded. For *Can*, we analyze q with argument $m' = +$ if $m = +$ and if p must terminate, with argument $m' = \perp$ otherwise:

$$\mathit{Must}(p\,;\,q, E) \;=\; \begin{cases} \mathit{Must}(p, E) \\ \qquad\quad \text{if } 0 \notin \mathit{Must}_k(p, E) \\ \langle\, \mathit{Must}_s(p, E) \cup \mathit{Must}_s(q, E)\,,\; \mathit{Must}_k(q, E)\,\rangle \\ \qquad\quad \text{if } 0 \in \mathit{Must}_k(p, E) \end{cases}$$

$$\mathit{Can}^m(p\,;\,q, E) \;=\; \begin{cases} \mathit{Can}^m(p, E) \\ \qquad\quad \text{if } 0 \notin \mathit{Can}_k^m(p, E) \\ \langle\;\; \mathit{Can}_s^m(p, E) \cup \mathit{Can}_s^{m'}(q, E)\,, \\ \quad\;\; \mathit{Can}_k^m(p, E)\backslash 0 \;\cup\; \mathit{Can}_k^{m'}(q, E) \;\;\rangle \\ \qquad\quad \text{if } 0 \in \mathit{Can}_k^m(p, E) \\ \qquad\quad \text{with } m' = + \text{ if } m = + \text{ and } 0 \in \mathit{Must}_k(p, E) \\ \qquad\quad \text{or } m' = \perp \text{ otherwise} \end{cases}$$

In the definition of $Must(p\,;\,q, E)$, we could also write the predicates using set comparisons, i.e. as $Must_k(p, E) \neq \{0\}$ and $Must_k(p, E) = \{0\}$. The result would be equivalent because $Must_k(p, E)$ is either empty or a singleton.

Notice that the set of signals that p must emit is not transmitted to q. We could do it, but we obtain the same effect using the simple rule above and a global iteration process: if p must emit s in a partial event E such that $s^\perp \in E$, the analysis will call it again in the partial event $E * s^+$, in which case s^+ will effectively reach q. The same holds for propagating s^- to q if p cannot emit s. The iteration is performed by the signal and global program rules described below.

For a loop, we analyze the body once. That the body cannot terminate will be ensured by the constructive semantic inference rules.

$$\begin{array}{rcl} Must(p*, E) & = & Must(p, E) \\ Can^m(p*, E) & = & Can^m(p, E) \end{array}$$

For a parallel, we take the union of the signal sets and the extension of max to sets of completion codes defined in Section 6.6:

$$\begin{array}{rcl} Must(p \mid q, E) & = & \langle\ Must_s(p, E) \cup Must_s(q, E)\,, \\ & & Max\Big(Must_k(p, E), Must_k(q, E)\Big)\ \rangle \\ Can^m(p \mid q, E) & = & \langle\ Can^m_s(p, E) \cup Can^m_s(q, E)\,, \\ & & Max\Big(Can^m_k(p, E), Can^m_k(q, E)\Big)\ \rangle \end{array}$$

Notice that $Must_k(p \mid q, E)$ is nonempty and is a singleton set if and only if $Must_k(p, E)$ and $Must_k(q, E)$ are singleton sets. As far as completion codes are concerned, one must be able to compute the codes that both branches must return to compute the code that the parallel statement must return. Using the Max set operation in the definition of $Can^m(p \mid q, E)$ is fundamental for adequate control propagation. For example, this operation ensures that a parallel cannot terminate if one of its branches cannot. This property would be lost if Max were replaced by a simple union, and this would lead to abnormally rejecting constructively correct programs such as $((o\,?\,0\,,0) \mid 1)\,;\,!o$, where the first parallel statement always pauses in the first instant because of its second 1 branch, breaking the potential cycle on o.

For trap and a shift, we apply the appropriate operators to the completion codes returned by the body:

$$Must(\{p\}, E) \;=\; \langle\, Must_s(p, E)\,,\, \downarrow Must_k(p, E)\,\rangle$$

$$\begin{array}{rcl}
Can^m(\{p\}, E) & = & \langle\, Can^m_s(p, E)\,,\, \downarrow Can^m_k(p, E)\,\rangle \\
Must(\uparrow p, E) & = & \langle\, Must_s(p, E)\,,\, \uparrow Must_k(p, E)\,\rangle \\
Can^m(\uparrow p, E) & = & \langle\, Can^m_s(p, E)\,,\, \uparrow Can^m_k(p, E)\,\rangle
\end{array}$$

The rules for the local signal declaration operator $p\backslash s$ are deeply different for *Must* and *Can*. For *Must*, because of the way the recursion works, the rule is used only if we already know that $p\backslash s$ must be executed. Since we have yet no information about the status of s, we first set this status to $\perp$ and we compute what we must and cannot do. If we find that p must emit s, we take that fact for granted, and we re-analyze p with status $+$ for s. If we find that p cannot emit s, we know that s must be absent and we re-analyze p with status $-$ for s. Otherwise, we cannot make progress. In each case, we remove the status of the local s from the emitted signal set, using the notation $\langle\, S\,,\, K\,\rangle\backslash s$ for $\langle\, S\backslash s\,,\, K\,\rangle$:

$$Must(p\backslash s, E) \;=\; \left\{\begin{array}{ll}
Must(p, E * s^{+})\backslash s & \text{if } s \in Must_s(p, E * s^{\perp}) \\
Must(p, E * s^{-})\backslash s & \text{if } s \notin Can^{+}_s(p, E * s^{\perp}) \\
Must(p, E * s^{\perp})\backslash s & \text{otherwise}
\end{array}\right.$$

For *Can*, we first analyze the body p with status $\perp$ for s, with the same m argument. If $m = +$ and if we find that the signal must be emitted, we re-analyze p with status $+$ for s. For both $m = +$ and $m = \perp$, if the signal cannot be emitted, we re-analyze p with status $-$ and with the same m. Otherwise, we return the result of the analysis of p with status $\perp$ for s.

$$Can^m(p\backslash s, E) \;=\; \left\{\begin{array}{l}
Can^{+}(p, E * s^{+})\backslash s \\
\quad \text{if } m = + \text{ and } s \in Must_s(p, E * s^{\perp}) \\
Can^m(p, E * s^{-})\backslash s \\
\quad \text{if } s \notin Can^m_s(p, E * s^{\perp}) \\
Can^m_s(p, E * s^{\perp})\backslash s \\
\quad \text{otherwise}
\end{array}\right.$$

In the *Can* analysis, notice that the signal status can be set to $+$ only if $m = +$. This is necessary to avoid speculative computation.

The constructively correct program `P2`, page 29, is a good example to try these definitions.

7.2 Possible Variants

There are two places where we made non-obvious choices: the *Must* rule for signal presence test and the *Can* rule for local signal declaration. We now discuss these choices in more details.

Consider first $s\,?\,p\,,q$. The reader familiar with static analysis might find the following *Must* rule less conservative than ours:

$$Must\Big((s\,?\,p\,,q), E\Big) \quad = \quad \begin{cases} Must(p,E) & \text{if } s^+ \in E \\ Must(q,E) & \text{if } s^- \in E \\ Must(p,E) \cap Must(q,E) & \text{if } s^\perp \in E \end{cases}$$

We reject this intersection rule as performing speculative computation. The rule would accept `P12`, page 41, "`present O then emit O else emit O end`". In this program, signal information flows *backwards* with respect to control, the typical thing we want to forbid.

For $p\backslash s$, one could think of making a perfect symmetry between *Can* and *Must*, writing

$$Can^m(p\backslash s, E) \quad = \quad \begin{cases} Can^m(p, E * s^+)\backslash s & \text{if } s \in Must_s(p, E * s^\perp) \\ Can^m(p, E * s^-)\backslash s & \text{if } s \notin Can^m_s(p, E * s^\perp) \\ Can^m(p, E * s^\perp)\backslash s & \text{otherwise} \end{cases}$$

Here again, we would perform speculative computation, since we would call *Must* even when $m = \perp$[1]. For instance, we would accept the program

```
present O then
   signal S in
      emit S
   ||
      present S else emit O end
   end
end
```

on the following grounds: since the body of the `signal` statement must emit `S`, the output `O` cannot be emitted and can be set absent. This reasoning speculatively executes the "`emit S`" statement.

[1]The first draft version of this book actually used this rule. This was a deep mistake, sorry!

Both variants above would ensure determinism. However, none of them obeys our intuition about fact-to-fact propagation and none of them is amenable to a natural operational and circuit semantics.

7.3 Elementary Properties of Must and Can

Remember that *Can* is an auxiliary tool to compute its complement *Cannot*, with

$$\mathit{Cannot}^m(p, E) \quad = \quad \overline{\mathit{Can}^m(p, E)}$$

The interesting properties are best expressed using *Cannot* since their counterpart with *Can* would be contravariant (antimonotonicity instead of monotonicity, etc.). The first property implies that the *Must* and *Cannot* predicates are disjoint:

Lemma 2 *For any statement* p *and partial event* E*, one has* $\mathit{Must}(p, E) \cap \mathit{Cannot}^m(p, E) = \langle\, \emptyset\,,\, \emptyset\, \rangle$*, i.e.* $\mathit{Must}(p, E) \subseteq \mathit{Can}^m(p, E)$*.*

The next property is that the *Must* and *Cannot* predicates are Scott-monotonic; it is essential to define the input/output function of a program, see Section 7.4.

Definition: The Scott ordering $\leq$ on statuses is defined by $\perp \leq +$ and $\perp \leq -$. The Scott ordering is extended to partial events of the same sort by $E_1 \leq E_2$ iff $E_1(s) \leq E_2(s)$ for each signal s in the sort.

Lemma 3 *If* $E_1 \leq E_2$*, then* $\mathit{Must}(p, E_1) \subseteq \mathit{Must}(p, E_2)$*, and* $\mathit{Cannot}^m(p, E_1) \subseteq \mathit{Cannot}^m(p, E_2)$*. Furthermore, for any partial event* E*, one has* $\mathit{Cannot}^{\perp}(p, E) \subseteq \mathit{Cannot}^{+}(p, E)$*.*

7.4 Definition of the Constructive Semantics

Given a program P of body p and an input event I, we compute the constructive behavioral semantics $P \xhookrightarrow[I]{O} P'$ of P for I in two steps. First, we compute the output event O using the *Must* and *Can* functions; this can fail if the status of some output or local signal cannot be determined to be either $+$ or $-$, in which case the program P is declared non-constructive for I.

Then, if all signal statuses have been determined, we compute a behavioral transition $p \xhookrightarrow[I \cup O]{O,k} p'$, which yields the next state p'; this can fail only if the body of some loop is found to terminate instantaneously.

We begin by computing the output event. The idea is to iteratively compute $\mathit{Must}(p, I \cup O)$ and $\mathit{Can}^m(p, I \cup O)$, starting from an undefined O where all output signal statuses are $\perp$, and repeatedly enriching O using the *Must* and *Can* information generated by a pass, this up to stabilization. This is just what we did for local signals, extended to sets of signals. Because of Lemma 3, stabilization is guaranteed by monotonicity. The simplest way to formalize this process is to notice that it is the computation of the least fixpoint of a monotonic function.

Definition: Given a program P and an input event I, we denote by $[\![P]\!](I)$ the function on partial output events defined by $[\![P]\!](I)(O) = O'$, where, for each output signal o, one has

$$O'(o) \quad = \quad \begin{cases} + & \text{if } o \in \mathit{Must}_s(p, I \cup O) \\ - & \text{if } o \in \mathit{Cannot}_s^+(p, I \cup O) \\ \perp & \text{otherwise} \end{cases}$$

(The restriction that an input signal cannot be internally emitted is essential here.)

Lemma 3 yields immediately monotonicity:

Lemma 4 *Given P and I, the function $[\![P]\!](I)$ is monotonic on output environments.*

Definition: Given P and I, the output event O constructively computed by P on I is the least fixpoint of the function $[\![P]\!](I)$. We say that P is *constructive* for I if $O(o) \neq \perp$ holds for any output signal o.

If P is constructive for I, we have determined the output event O, and we are left with determining the new state P' such that $P \xhookrightarrow[I]{O} P'$. As in the behavioral semantics, we use an inductive relation $p \xhookrightarrow[E]{E',k} p'$ and we determine p' by $p \xhookrightarrow[I \cup O]{O,k} p'$. The definition rules of the constructive relation are exactly those of $p \xrightarrow[E]{E', k} p'$ in the logical behavioral semantics, except for the local signal rules *(sig+)* and *(sig-)*, which are made constructive:

$$\frac{s \in Must_s(p, E * s^{\perp}) \quad p \xhookrightarrow[E*s^{+}]{E'*s^{+}, k} p' \quad \mathcal{S}(E') = \mathcal{S}(E)\backslash s}{p\backslash s \xhookrightarrow[E]{E', k} p'\backslash s} \qquad (csig+)$$

$$\frac{s \in Cannot^{+}_{s}(p, E * s^{\perp}) \quad p \xhookrightarrow[E*s^{-}]{E'*s^{-}, k} p' \quad \mathcal{S}(E') = \mathcal{S}(E)\backslash s}{p\backslash s \xhookrightarrow[E]{E', k} p'\backslash s} \qquad (csig-)$$

Notice that the inference rules involve only total events since the status of all signals is determined to be $+$ or $-$, either by the above computation of the output event or by the constructive local signal rules; this is why the other behavioral rules can be left unchanged.

7.5 Correctness of the Constructive Semantics

The main result shows that the constructive behavioral semantics is logically correct.

Theorem 1 *Let P be a program and I be an input event for P. If $P \xhookrightarrow[I]{O} P'$ is provable in the constructive behavioral semantics, then $P \xrightarrow[I]{O} P'$ is provable in the logical behavioral semantics and it has a unique proof. The program P is reactive and strongly deterministic w.r.t. I.*

Chapter 8

The State Behavioral Semantics

The logical and constructive behavioral semantics define the semantics of Esterel with a minimal number of rules. However, there is a drawback: in a reaction $p \xrightarrow[E]{E',k} p'$, the resulting term p' is obtained by a non-trivial rewriting process. Furthermore, when chaining reactions, the rewritings pile up, which makes it difficult to understand the structure of the resulting statement in function of the initial one. This does not fit with elementary programming intuition, where one prefers to deal with control points moving in a fixed program text. The *state semantics* we describe in this chapter realizes this goal, while computing reactions in the same way as the behavioral semantics. The price to pay is an extension of the syntax and an increase in the number of rules.

We first introduce the extended syntax used to denote the state of a statement after a reaction. Next, we give the logical behavioral rules of the extended language, and we extend the *Must* and *Can* predicates to define the constructive behavioral semantics. Finally, we show the correctness of the logical and constructive state semantics w.r.t. the original semantics.

8.1 The Extended Syntax

8.1.1 States as Decorated Terms

Consider the simple inputless term $p = 1\,;\,1\,;\,!s$. In the behavioral semantics, the sequence of reactions is:

$$1\,;\,1\,;\,!s \xrightarrow[\emptyset]{\emptyset,1} 0\,;\,1\,;\,!s \xrightarrow[\emptyset]{\emptyset,1} 0\,;\,!s \xrightarrow[\emptyset]{\{s\},0} 0 \xrightarrow[\emptyset]{\emptyset,0} 0 \cdots$$

In the state semantics, the idea is to keep the shape of the statement constant and to decorate it to indicate where control is pausing between reactions. Only occurrences of 1 are decorated[1]: we write $\hat{1}$ instead of 1 to indicate that execution has paused there and will resume from there in the following instant. The other statements need not be decorated since they are purely instantaneous. Using decoration, the above execution sequence becomes:

$$1\,;\,1\,;\,!s \xrightarrow[\emptyset]{\emptyset,1} \hat{1}\,;\,1\,;\,!s \xrightarrow[\emptyset]{\emptyset,1} 1\,;\,\hat{1}\,;\,!s \xrightarrow[\emptyset]{\{s\},0} 1\,;\,1\,;\,!s \xrightarrow[\emptyset]{\emptyset,0} \cdots$$

The intuition is clear, but there is a slight difficulty. In the above example, the fourth statement is $p_3 = 1\,;\,1\,;\,!s$, which is just the same as the initial statement p_0 from which execution starts. However, p_3 should be considered as being terminated, unlike p_0. To resolve this ambiguity, we replace the initial statement p_0 by the decorated term $\hat{1}\,;\,p_0$. Then, control initially resumes from the auxiliary head $\hat{1}$ statement, which is called the *boot* statement. Any standard undecorated program body such as p_3 is considered to be terminated.

With the boot $\hat{1}$ statement added, the execution sequence becomes:

$$\hat{1}\,;\,1\,;\,1\,;\,!s \xrightarrow[\emptyset]{\emptyset,1} 1\,;\,\hat{1}\,;\,1\,;\,!s \xrightarrow[\emptyset]{\emptyset,1} 1\,;\,1\,;\,\hat{1}\,;\,!s \xrightarrow[\emptyset]{\{s\},0} 1\,;\,1\,;\,1\,;\,!s \xrightarrow[\emptyset]{\emptyset,0} 1\,;\,1\,;\,1\,;\,!s$$

Instead of the boot $\hat{1}$ statement, we could have used a termination marker. Using a boot statement is more natural in the circuit translation and avoids introducing a new end marker symbol in the language.

[1] In the full user-level language, other statements can be decorated, for example `await`, `every`, and `loop-each`. The expansion of each of these user-level statements involves exactly one 1 statement, which makes decoration unambiguous (see [19]).

8.1.2 State Syntax

We call *standard statements* the basic kernel statements introduced in Chapter 5. We define a *state* $\hat{p}$, $\hat{q}$, etc., as a kernel statement where some part is decorated or *selected* for execution. Finally, we say that a *term* $\overline{p}$, $\overline{q}$, etc., is either a standard statement or a state. The grammar of states and terms is as follows:

$$
\begin{array}{rcl}
\hat{p} & ::= & \hat{1} \\
& | & s\,?\,\hat{p}\,,q \\
& | & s\,?\,p\,,\hat{q} \\
& | & s \supset \hat{p} \\
& | & \hat{p}\,;\,q \\
& | & p\,;\,\hat{q} \\
& | & \hat{p}* \\
& | & \overline{p} \mid \overline{q} \\
& | & \{\hat{p}\} \\
& | & \uparrow \hat{p} \\
& | & \hat{p}\backslash s \\
\overline{p} & ::= & p \\
& | & \hat{p}
\end{array}
$$

Notice that a subterm is selected if and only if it contains a selected pause statement $\hat{1}$. In a state such as $s\,?\,\hat{p}\,,q$, the only selected subterm is $\hat{p}$. Such a state occurs if the test has taken its left branch in some previous instant and if the execution of this branch is not yet terminated. A parallel branch can be either a standard statement or a state. In $\hat{p} \mid \hat{q}$, both $\hat{p}$ and $\hat{q}$ are selected: the parallel has been started in some past instant and both branches are still active. A state of the form $\hat{p} \mid q$ corresponds to the case where the second branch of the parallel is already terminated while the first one is still active. For example, the state $(\hat{1}\,;\,1) \mid \hat{1}$ becomes $(1\,;\,\hat{1}) \mid 1$ in the following instant. There is no state of the form $s\,?\,\hat{p}\,,\hat{q}$ or of the form $\hat{p}\,;\,\hat{q}$, since one cannot be pausing simultaneously in both arms of a test or of a sequence.

We now define the *base statement* $\mathcal{B}(\overline{p})$ of a term $\overline{p}$. For a standard statement p, we set $\mathcal{B}(p) = p$. For a state $\hat{p}$, we strip the decoration, i.e. we recursively replace the selected parts of $\hat{p}$ by their base statement, ending with $\mathcal{B}(\hat{1}) = 1$ (formal definition left to the reader). To make the notation lighter, given terms $\overline{p}$ and $\overline{q}$, we simply abbreviate $\mathcal{B}(\overline{p})$ and $\mathcal{B}(\overline{q})$ into p and q, always assuming that $\hat{p}$, $\hat{p}'$, $\overline{p}$, $\overline{p}'$, etc., are extended statements having p as base statement.

8.2 State Expansion

States encode derivatives of a program by decorating the program's body, which implies that understanding the state semantics essentially amounts to understanding how the decoration encoding works. Given a starting term p, a derivative p' obtained after the reaction to some input sequence, and a reaction $p' \xrightarrow[E]{E',k} p''$, our goal is to obtain an equivalent state reaction $\overline{p}' \xrightarrow[E]{E',k} \overline{p}''$ where $\overline{p}'$ encodes p' and $\overline{p}''$ encodes p'' with the same base statement p. For this, we define the *expansion function* $\mathcal{E}(\overline{p})$ from terms to standard statements. One more example will be helpful. Consider the following statement reaction sequence:

$$(!s \mid (s\,?\,(1\,;\,!o)\,,1))\backslash s \quad \xrightarrow[\emptyset]{\emptyset,1} \quad (0 \mid (0\,;\,!o))\backslash s$$

$$\xrightarrow[\emptyset]{\{o\},0} \quad (0 \mid 0)\backslash s$$

The corresponding state reaction sequence is:

$$\hat{1}\,;\,(!s \mid (s\,?\,(1\,;\,!o)\,,1))\backslash s \quad \xrightarrow[\emptyset]{\emptyset,1} \quad 1\,;\,(!s \mid (s\,?\,(\hat{1}\,;\,!o)\,,1))\backslash s$$

$$\xrightarrow[\emptyset]{\{o\},0} \quad 1\,;\,(!s \mid (s\,?\,(1\,;\,!o)\,,1))\backslash s$$

For the first expansion, we use the rules $\mathcal{E}(\hat{1}) = 0$ and $\mathcal{E}(\hat{p}\,;\,q) = \mathcal{E}(\hat{p})\,;\,q$. This yields the expanded statement $0\,;\,(!s \mid (s\,?\,(1\,;\,!o)\,,1))\backslash s$ which is not quite the original initial statement but is clearly equivalent since the initial 0 disappears in any reaction. For the second expansion, we use the following rules:

- $\mathcal{E}(p\,;\,\hat{q}) = \mathcal{E}(\hat{q})$, which expresses that p is already terminated and must be discarded if control is currently in q;
- $\mathcal{E}(\hat{p}\backslash s) = \mathcal{E}(\hat{p})\backslash s$, which is trivial;
- $\mathcal{E}(p \mid \hat{q}) = 0|\mathcal{E}(\hat{q})$: the first branch is a standard statement, which means that it is already terminated and that it must be expanded into 0 in the current instant.

- $\mathcal{E}(s\,?\,\hat{p}\,,q) = \mathcal{E}(\hat{p})$, since the test for s has already been performed and should not be performed again when executing the `then` branch.

This time, the expansion yields the exact second statement. For the third expansion, we use the rule $\mathcal{E}(p) = 0$ that turns any standard statement into 0. As for the first reaction, we do not exactly obtain the original third term $(0\,|\,0)\backslash s$, but the simpler term 0 has the same behavior in all reactions. Altogether, the expansion $\mathcal{E}(\overline{p})$ is identical to the corresponding original derivative up to some trivial arrangements of 0's.

Technically, the *expansion function* $\mathcal{E}(\overline{p})$ is defined as follows:

$$
\begin{array}{rcl}
\mathcal{E}(p) & = & 0 \\
\mathcal{E}(\hat{1}) & = & 0 \\
\mathcal{E}(s\,?\,\hat{p}\,,q) & = & \mathcal{E}(\hat{p}) \\
\mathcal{E}(s\,?\,p\,,\hat{q}) & = & \mathcal{E}(\hat{q}) \\
\mathcal{E}(s \supset \hat{p}) & = & s \mathbin{\hat{\supset}} \mathcal{E}(\hat{p}) \\
\mathcal{E}(\hat{p}\,;\,q) & = & \mathcal{E}(\hat{p})\,;\,q \\
\mathcal{E}(p\,;\,\hat{q}) & = & \mathcal{E}(\hat{q}) \\
\mathcal{E}(\hat{p}*) & = & \mathcal{E}(\hat{p})\,;\,(p*) \\
\mathcal{E}(\overline{p}\,|\,\overline{q}) & = & \mathcal{E}(\overline{p})\,|\,\mathcal{E}(\overline{q}) \\
\mathcal{E}(\{\hat{p}\}) & = & \{\mathcal{E}(\hat{p})\} \\
\mathcal{E}(\uparrow \hat{p}) & = & \uparrow \mathcal{E}(\hat{p}) \\
\mathcal{E}(\hat{p}\backslash s) & = & \mathcal{E}(\hat{p})\backslash s
\end{array}
$$

Notice that a selected suspension statement $s \supset \hat{p}$ is expanded into an immediate suspension $s \mathbin{\hat{\supset}} \mathcal{E}(\hat{p})$ as required by the behavioral semantics.

8.3 The State Behavioral Semantics

In this section, we directly define the logical behavioral semantics of the extended language. Then, we define the *Must* and *Can* predicates and the constructive behavioral semantics.

8.3.1 The Logical State Behavioral Semantics

The definition of the state behavioral semantics follows the same pattern as in Chapter 6. Let P be a program of body q, and let $p = 1\,;\,q$. An *extended program* $\overline{P}$ of base P has body an extended statement $\overline{p}$ of base p. The behavioral semantics formalize a reaction of an extended program $\overline{P}$ as a behavioral transition of the form

$$\overline{P} \xrightarrow[I]{O} \overline{P}'$$

If the body of $\overline{P}$ is the standard statement p, then $\overline{P}$ is considered to be terminated and it reacts to any input by producing no output. If the body of $\overline{P}$ is a proper state $\hat{p}$, the reaction is computed using an auxiliary inductive relation

$$\hat{p} \xrightarrow[E]{E',\,k} \overline{p}'$$

where the base statement p is left unchanged according to our convention. As in Section 6.2, we set

$$\hat{P} \xrightarrow[I]{O} \overline{P}' \;\text{ iff }\; \hat{p} \xrightarrow[I \cup O]{O,\,k} \overline{p}'$$

The logical behavioral rules immediately follow from the definitions of the statement behavioral semantics and of the expansion function: to find what a proper state $\hat{p}$ can do, just inspect what $\mathcal{E}(\hat{p})$ can do.

The rules are split into two categories: *s-rules* start execution of a fresh statement p, while *r-rules* resume execution from a proper state $\hat{p}$. When two similar rules apply to both a state $\hat{p}$ and a standard statement p, we group them into a single *sr-rule* acting on a term $\overline{p}$, see for example rules *(sr-seq1)* and *(sr-seq2)* below.

$$\frac{k \neq 1}{k \xrightarrow[E]{\emptyset,\,k} k} \qquad \textit{(s-term-exit)}$$

$$1 \xrightarrow[E]{\emptyset,\,1} \hat{1} \qquad \textit{(s-pause)}$$

$$\hat{1} \xrightarrow[E]{\emptyset,\,0} 1 \qquad \textit{(r-pause)}$$

$$!s \xrightarrow[E]{\{s^+\},\,0} !s \qquad \textit{(s-emit)}$$

$$\frac{s^+ \in E \qquad p \xrightarrow[E]{E',\,k} \overline{p}'}{s\,?\,p\,,q \xrightarrow[E]{E',\,k} s\,?\,\overline{p}'\,,q} \qquad \textit{(s-present}+\textit{)}$$

$$\frac{s^- \in E \qquad q \xrightarrow[E]{F',\,l} \overline{q}'}{s\,?\,p\,,q \xrightarrow[E]{F',\,l} s\,?\,p\,,\overline{q}'} \qquad \textit{(s-present}-\textit{)}$$

$$\frac{\hat{p} \xrightarrow[E]{E',\,k} \overline{p}'}{s\,?\,\hat{p}\,,q \xrightarrow[E]{E',\,k} s\,?\,\overline{p}'\,,q} \qquad \textit{(r-then)}$$

$$\frac{\hat{q} \xrightarrow[E]{F',\,l} \overline{q}'}{s\,?\,p\,,\hat{q} \xrightarrow[E]{F',\,l} s\,?\,p\,,\overline{q}'} \qquad \textit{(r-else)}$$

$$\frac{p \xrightarrow[E]{E',\,k} \overline{p}'}{s \supset p \xrightarrow[E]{E',\,k} s \supset \overline{p}'} \qquad \textit{(s-suspend)}$$

$$\frac{s^+ \in E}{s \supset \hat{p} \xrightarrow[E]{\emptyset,\,1} s \supset \hat{p}} \qquad \textit{(r-suspend}+\textit{)}$$

$$\frac{s^- \in E \qquad \hat{p} \xrightarrow[E]{E',\,k} \overline{p}'}{s \supset \hat{p} \xrightarrow[E]{E',\,k} s \supset \overline{p}'} \qquad \textit{(r-suspend}-\textit{)}$$

$$\frac{\overline{p} \xrightarrow[E]{E',k} \overline{p}' \qquad k \neq 0}{\overline{p}\,;\, q \xrightarrow[E]{E',k} \overline{p}'\,;\, q} \qquad \textit{(sr-seq1)}$$

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

$$\frac{\hat{q} \xrightarrow[E]{F',l} \overline{q}'}{p\,;\, \hat{q} \xrightarrow[E]{F',l} p\,;\, \overline{q}'} \qquad \textit{(r-seq3)}$$

$$\frac{\overline{p} \xrightarrow[E]{E',k} \overline{p}' \qquad k \neq 0}{\overline{p}* \xrightarrow[E]{E',k} \overline{p}'*} \qquad \textit{(sr-loop)}$$

$$\frac{\hat{p} \xrightarrow[E]{E',0} p \qquad p \xrightarrow[E]{E'',k} \overline{p}' \qquad k \neq 0}{\hat{p}* \xrightarrow[E]{E' \cup E'',k} \overline{p}'*} \qquad \textit{(r-do-loop)}$$

$$\frac{p \xrightarrow[E]{E',k} \overline{p}' \qquad q \xrightarrow[E]{F',l} \overline{q}'}{p \mid q \xrightarrow[E]{E' \cup F',\, max(k,l)} \overline{p}' \mid \overline{q}'} \qquad \textit{(s-both)}$$

$$\frac{\hat{p} \xrightarrow[E]{E',k} \overline{p}' \qquad \hat{q} \xrightarrow[E]{F',l} \overline{q}'}{\hat{p} \mid \hat{q} \xrightarrow[E]{E' \cup F',\, max(k,l)} \overline{p}' \mid \overline{q}'} \qquad \textit{(r-both)}$$

$$\frac{\hat{p} \xrightarrow[E]{E', k} \overline{p}'}{\hat{p} \mid q \xrightarrow[E]{E', k} \overline{p}' \mid q} \qquad \textit{(r-left)}$$

$$\frac{\hat{q} \xrightarrow[E]{F', l} \overline{q}'}{p \mid \hat{q} \xrightarrow[E]{F', l} p \mid \overline{q}'} \qquad \textit{(r-right)}$$

$$\frac{\overline{p} \xrightarrow[E]{E', k} \overline{p}' \qquad k = 0 \text{ or } k = 2}{\{\overline{p}\} \xrightarrow[E]{E', 0} \{p\}} \qquad \textit{(sr-term-trap)}$$

$$\frac{\overline{p} \xrightarrow[E]{E', k} \overline{p}' \qquad k \neq 0 \text{ and } k \neq 2}{\{\overline{p}\} \xrightarrow[E]{E', \downarrow k} \{\overline{p}'\}} \qquad \textit{(sr-prop-trap)}$$

$$\frac{\overline{p} \xrightarrow[E]{E', k} \overline{p}'}{\uparrow \overline{p} \xrightarrow[E]{E', \uparrow k} \uparrow \overline{p}'} \qquad \textit{(sr-shift)}$$

$$\frac{\overline{p} \xrightarrow[E * s^+]{E' * s^+, k} \overline{p}' \qquad \mathcal{S}(E') = \mathcal{S}(E) \backslash s}{\overline{p} \backslash s \xrightarrow[E]{E', k} \overline{p}' \backslash s} \qquad \textit{(sr-sig+)}$$

$$\frac{\overline{p} \xrightarrow[E * s^-]{E' * s^-, k} \overline{p}' \qquad \mathcal{S}(E') = \mathcal{S}(E) \backslash s}{\overline{p} \backslash s \xrightarrow[E]{E', k} \overline{p}' \backslash s} \qquad \textit{(sr-sig}-\textit{)}$$

8.3.2 The Constructive State Behavioral Semantics

To define the constructive behavioral semantics of the extended language, we add the *Must* and *Can* predicates to the local signal rules as before:

$$\frac{s \in Must_s(\overline{p}, E * s^{\perp}) \quad \overline{p} \overset{E' * s^+,\, k}{\underset{E * s^+}{\hookrightarrow}} \overline{p}' \quad \mathcal{S}(E') = \mathcal{S}(E) \backslash s}{\overline{p} \backslash s \overset{E',\, k}{\underset{E}{\hookrightarrow}} \overline{p}' \backslash s} \quad \textit{(sr-csig+)}$$

$$\frac{s \notin Can_s^+(\overline{p}, E * s^{\perp}) \quad \overline{p} \overset{E' * s^-,\, k}{\underset{E * s^-}{\hookrightarrow}} \overline{p}' \quad \mathcal{S}(E') = \mathcal{S}(E) \backslash s}{\overline{p} \backslash s \overset{E',\, k}{\underset{E}{\hookrightarrow}} \overline{p}' \backslash s} \quad \textit{(sr-csig−)}$$

the other rules of $\overline{p} \overset{E',\, k}{\underset{E}{\hookrightarrow}} \overline{p}'$ being those of $\overline{p} \overset{E',\, k}{\underset{E}{\longrightarrow}} \overline{p}'$.

We extend the definition of *Must* and *Can* in such a way that $Must(\hat{p}, E) = Must(\mathcal{E}(\hat{p}), E)$ and $Can^m(\hat{p}, E) = Can^m(\mathcal{E}(\hat{p}), E)$, retaining the basic definitions of *Must* and *Can* for standard statements as they were given in Chapter 7. The additional definitions are:

$$\begin{aligned}
Must(\hat{1}, E) = Can^m(\hat{1}, E) &= \langle\, \emptyset \,,\, \{0\} \,\rangle \\
Must\Big((s \,?\, \hat{p} \,,\, q), E\Big) &= Must(\hat{p}, E) \\
Can^m\Big((s \,?\, \hat{p} \,,\, q), E\Big) &= Can^m(\hat{p}, E) \\
Must\Big((s \,?\, p \,,\, \hat{q}), E\Big) &= Must(\hat{q}, E) \\
Can^m\Big((s \,?\, p \,,\, \hat{q}), E\Big) &= Can^m(\hat{q}, E) \\
Must(s \supset \hat{p}, E) &= \begin{cases} \langle\, \emptyset \,,\, \{1\} \,\rangle & \text{if } s^+ \in E \\ Must(\hat{p}, E) & \text{if } s^- \in E \\ \langle\, \emptyset \,,\, \emptyset \,\rangle & \text{if } s^{\perp} \in E \end{cases}
\end{aligned}$$

$$Can^m(s \supset \hat{p}, E) = \begin{cases} \langle\, \emptyset\,,\, \{1\}\,\rangle \\ \qquad \text{if } s^+ \in E \\ Can^m(\hat{p}, E) \\ \qquad \text{if } s^- \in E \\ \langle\, \emptyset\,,\, \{1\}\,\rangle \cup Can^m_s(\hat{p}, E) \\ \qquad \text{if } s^\perp \in E \end{cases}$$

$$Must(\hat{p}\,;\, q, E) = \begin{cases} Must(\hat{p}, E) \\ \qquad \text{if } 0 \notin Must_k(\hat{p}, E) \\ \langle\; Must_s(\hat{p}, E) \cup Must_s(q, E)\,, \\ \quad Must_k(q, E) \;\rangle \\ \qquad \text{if } 0 \in Must_k(\hat{p}, E) \end{cases}$$

$$Can^m(\hat{p}\,;\, q, E) = \begin{cases} Can^m(\hat{p}, E) \\ \qquad \text{if } 0 \notin Can^m_k(\hat{p}, E) \\ \langle\; Can^m_s(\hat{p}, E) \cup Can^{m'}_s(q, E)\,, \\ \quad Can^m_k(\hat{p}, E) \backslash 0 \,\cup\, Can^{m'}_k(q, E) \;\rangle \\ \qquad \text{if } 0 \in Can^m_k(\hat{p}, E) \\ \qquad \text{with } m' = + \\ \qquad\quad \text{if } m = + \text{ and } 0 \in Must_k(p, E) \\ \qquad \text{or } m' = \perp \text{ otherwise} \end{cases}$$

$$Must(p\,;\, \hat{q}, E) = Must(\hat{q}, E)$$

$$Can^m(p\,;\, \hat{q}, E) = Can^m(\hat{q}, E)$$

$$Must(\hat{p}*, E) = \begin{cases} Must(\hat{p}, E) \\ \qquad \text{if } 0 \notin Must_k(\hat{p}, E) \\ \langle\, Must_s(\hat{p}, E) \cup Must_s(p, E)\,,\, Must_k(p, E)\,\rangle \\ \qquad \text{if } 0 \in Must_k(\hat{p}, E) \end{cases}$$

$$Can^m(\hat{p}*, E) = \begin{cases} Can^m(\hat{p}, E) \\ \qquad \text{if } 0 \notin Can^m_k(\hat{p}, E) \\ \langle\; Can^m_s(\hat{p}, E) \cup Can^{m'}_s(p, E)\,, \\ \quad Can^m_k(\hat{p}, E) \backslash 0 \,\cup\, Can^{m'}_k(p, E) \;\rangle \\ \qquad \text{if } 0 \in Can^m_k(\hat{p}, E) \\ \qquad \text{with } m' = + \\ \qquad\quad \text{if } m = + \text{ and } 0 \in Must_k(\hat{p}, E) \\ \qquad \text{or } m' = \perp \text{ otherwise} \end{cases}$$

$$
\begin{array}{rcl}
Must(\hat{p} \mid \hat{q}, E) & = & \langle\ Must_s(\hat{p}, E) \cup Must_s(\hat{q}, E)\,, \\
 & & \quad Max\Big(Must_k(\hat{p}, E), Must_k(\hat{q}, E)\Big)\ \rangle \\
Can^m(\hat{p} \mid \hat{q}, E) & = & \langle\ Can^m_s(\hat{p}, E) \cup Can^m_s(\hat{q}, E)\,, \\
 & & \quad Max\Big(Can^m_k(\hat{p}, E), Can^m_k(\hat{q}, E)\Big)\ \rangle \\
Must(\hat{p} \mid q, E) & = & Must(\hat{p}, E) \\
Can^m(\hat{p} \mid q, E) & = & Can^m(\hat{p}, E) \\
Must(p \mid \hat{q}, E) & = & Must(\hat{q}, E) \\
Can^m(p \mid \hat{q}, E) & = & Can^m(\hat{q}, E) \\
Must(\{\hat{p}\}, E) & = & \langle\, Must_s(\hat{p}, E)\,,\ \downarrow Must_k(\hat{p}, E)\,\rangle \\
Can^m(\{\hat{p}\}, E) & = & \langle\, Can^m_s(\hat{p}, E)\,,\ \downarrow Can^m_k(\hat{p}, E)\,\rangle \\
Must(\uparrow \hat{p}, E) & = & \langle\, Must_s(\hat{p}, E)\,,\ \uparrow Must_k(\hat{p}, E)\,\rangle \\
Can^m(\uparrow \hat{p}, E) & = & \langle\, Can^m_s(\hat{p}, E)\,,\ \uparrow Can^m_k(\hat{p}, E)\,\rangle
\end{array}
$$

$$
Must(\hat{p}\backslash s, E) = \begin{cases}
Must(\hat{p}, E * s^+)\backslash s \\
\quad \text{if } s \in Must_s(\hat{p}, E * s^\perp) \\
Must(\hat{p}, E * s^-)\backslash s \\
\quad \text{if } s \notin Can^+_s(\hat{p}, E * s^\perp) \\
Must(\hat{p}, E * s^\perp)\backslash s \\
\quad \text{otherwise}
\end{cases}
$$

$$
Can^m(\hat{p}\backslash s, E) = \begin{cases}
Can^+(\hat{p}, E * s^+)\backslash s \\
\quad \text{if } m = +\text{and } s \in Must_s(p, E * s^\perp) \\
Can^m(\hat{p}, E * s^-)\backslash s \\
\quad \text{if } s \notin Can^m_s(\hat{p}, E * s^\perp) \\
Can^m_s(\hat{p}, E * s^\perp)\backslash s \\
\quad \text{otherwise}
\end{cases}
$$

8.4 Equivalence to the Standard Semantics

As expected, the main result is that the standard semantics of a program P of body p and the state semantics of the program of initial state $\hat{1}\,; p$ determine the same transition sequences and the same causality, in the sense that corresponding derivatives react logically or constructively to the same inputs by producing the same outputs and corresponding derivatives. However, the standard and state transition systems are not exactly isomorphic because of the fact that a standard term reduction can leave some innocuous terminated terms that do not appear in the state semantics: consider for example the reduction $1\,;\,0 \xrightarrow[\emptyset]{\emptyset,0} 0\,;\,0 \xrightarrow[\emptyset]{\emptyset,0} 0$, which yields a proper 0 only in two steps. Technically, the right notion is that of (strong) bisimilarity well-known in process calculi [31] and not recalled in this draft.

Before stating the main result, we need a tool to relate standard terms that differ only by terminated initial subterms, such as $0\,;\,0$ and 0. This is *immediate equivalence* defined below:

Definition: We say that two standard statements p and q of the same sort S are *immediately equivalent*, and we write $p \equiv q$, if, for all E, E', k, one has $p \xrightarrow[E]{E',k} p'$ if and only if $q \xrightarrow[E]{E',k} p'$. Similarly, we say that p and q are *constructively immediately equivalent*, and we write $p \equiv_c q$, if, for all E, E', k, one has $p \xhookrightarrow[E]{E',k} p'$ if and only if $q \xhookrightarrow[E]{E',k} p'$.

Lemma 5 *Let $\hat{p}$ be a state. One has $\hat{p} \xrightarrow[E]{E',k} \overline{p}'$ if and only if one has $\mathcal{E}(\hat{p}) \xrightarrow[E]{E',k} q$ for some q such that $\mathcal{E}(\overline{p}') \equiv q$. One has $\hat{p} \xhookrightarrow[E]{E',k} \overline{p}'$ if and only if one has $\mathcal{E}(\hat{p}) \xhookrightarrow[E]{E',k} q$ for some q such that $\mathcal{E}(\overline{p}') \equiv_c q$.*

Lemma 6 *Let p be a standard statement. Then one has $\mathcal{E}(\hat{1}\,;\,p) \equiv p$ and $\mathcal{E}(\hat{1}\,;\,p) \equiv_c p$.*

Theorem 2 *For any standard statement p, the transition systems determined by p in the logical (resp. constructive) behavioral semantics and by $\hat{1}\,;\,p$ in the logical (resp. constructive) state behavioral semantics are bisimilar.*

The next result expresses that any Pure Esterel program is finite state. It directly follows from the fact that a program body can be decorated only in finitely many ways.

Theorem 3 *In the state semantics, any program has only finitely many iterated derivatives.*

The Esterel v5 compiler can translate an Esterel program into a deterministic finite automaton by exhaustively computing all iterated derivatives of the initial state. In practice, a derivative is represented by the set of all selected occurrences of 1 in the current state, which can be efficiently implemented by a bitset.

Chapter 9

The Constructive Operational Semantics

This part of the book is currently being rewritten. It is not necessary to understand what follows.

Part III

Circuit Translation

Chapter 10

Constructive Circuits

This chapter presents constructive (sequential) Boolean circuits. Circuits react to inputs by producing outputs and changing state, as do Esterel programs. Circuits are physical objects, with much simpler semantics as Esterel's. This is why we choose them as implementation targets. Producing a circuit out of a program does not imply a hardware implementation, since it is easy to simulate circuit execution in software. This is what the Esterel v5 compiler to C does.

Below, we shall use two equivalent presentations of Boolean circuits. In Chapter 11, we use circuits diagrams. In Chapter 13, we use systems of Boolean equations. Both presentations have been informally used in Section 4 and will be defined here.

10.1 Boolean Circuits

10.1.1 Textual Definition

A Boolean circuit $\mathcal{C}$ is defined by a set $\mathcal{W}$ of *variables* or *wires* and a set of *wire definitions*. The set $\mathcal{W}$ is partitioned into a set $\mathcal{I}$ of *input* wires, a set $\mathcal{O}$ of *output* wires, and a set $\mathcal{L}$ of *local* wires.

Boolean expressions $e, e' \ldots$ are composed of wires w, constants 0 and 1, and connectives $\neg$, $\vee$, and $\wedge$. Each output or local wire must be defined by exactly one wire definition. There are two kinds of wire definitions.

- An *equality* definition $w = e$ defines the value of w to be that of e in each instant.

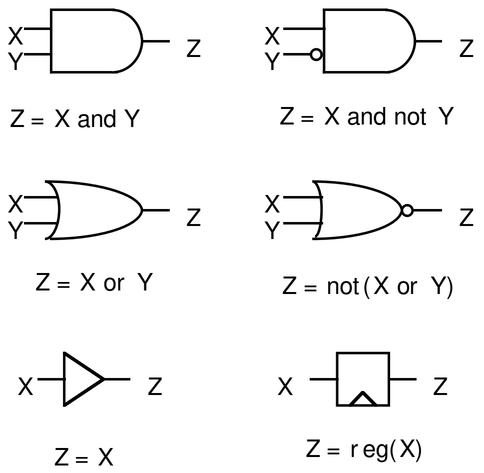


Figure 10.1: Gate Symbols

- A *register* definition $w := e$ defines the value of w to be initially 0; subsequently, it is the value of e in the previous instant.

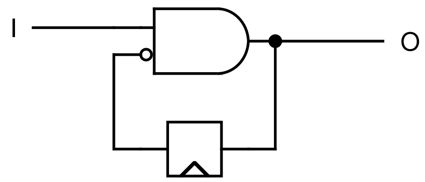
The expression that appears in the right-hand side of the definition of wire w is called $\mathcal{C}(w)$. A wire defined by an equality definition is called a *standard* or *combinational* wire, and $\mathcal{S}$ denotes the set of standard wires. A wire defined by a register definition is called a *register*, and $\mathcal{R}$ denotes the set of registers. An output wire may be either a standard wire or a register wire. Therefore, one has $\mathcal{S} \subset \mathcal{L} \cup \mathcal{O}$, $\mathcal{R} \subset \mathcal{L} \cup \mathcal{O}$, $\mathcal{S} \cap \mathcal{R} = \emptyset$, $\mathcal{S} \cup \mathcal{R} = \mathcal{L} \cup \mathcal{O}$, and $\mathcal{I} \cup \mathcal{S} \cup \mathcal{R} = \mathcal{W}$.

An *input event* I (resp. *output event* O) is an assignment of Boolean values to the input (resp. output) wires. A *state* R is an assignment of Boolean values to the register wires. The *initial state* assigns 0 to all registers.

10.1.2 Definition by Diagrams

In diagrams, a circuit is a graph of *gates* that represent Boolean operators linked by wires. Each gate may have several inputs and has only one output, which is the wire it defines. The conventional gates are pictured in Figure 10.1. The identity function is represented by the conventional *buffer* gate[1]; we could also use a unary *and* or *or* gate. Negation gates are pictured

[1] In hardware terminology, there is no memory in a buffer.



$$O = I \wedge \neg R$$

$$R := O$$

Figure 10.2: Diagrams vs. Definitions

by a little circle and placed either on an input or on the output of a gate.

The wires in a circuit must be either primary inputs or driven by a single gate. However, for convenience, we allow an output to have no defining gate, in which case it is assumed to always have value 0.

To translate a system of definitions into a circuit diagram, one simply creates as many gates as there are Boolean operators in expressions and one register gate per register definition. To translate diagrams into equations, one introduces one wire name for each unnamed gate output and one writes the corresponding equations. An example of a diagram and its equational form is given in Figure 10.2.

10.2 The Logical Semantics

In the logical semantics, we are interested in the solutions of the system of equations that defines the standard wires. Given an input I and a state R, a *logical solution* is a set S of assignments of values 0 or 1 to the standard wires such that all standard wire equations are valid when interpreted with wire values given by I, R, and S. We say that a wire is *set* in a logical solution if it has value 1 and that it is *unset* if it has value 0.

Reactivity and determinism w.r.t. I and R are defined as existence and uniqueness of a logical solution, just as in Chapter 3, and logical correctness is the conjunction of reactivity and determinism. If a circuit is logically correct, then the output O is defined by the value of the output wires and the new state R' is defined by the values of the right-hand-side expressions of the register definitions. We then write $R \xrightarrow[I]{O} R'$.

10.3 The Constructive Semantics

As we mentioned in Section 4.4, not all circuits are logically correct (see circuit `C2`, page 47), and not all logically correct circuits should be accepted (see circuits `C9`, page 48 and Figure 4.1, and `C12`, page 48 and Figure 4.2). We now define the constructive semantics in three different ways: a proof-theoretic way, by defining constructive information propagation, a denotational way, by considering least fixpoints of Scott-monotonic functions, and an electrical way, by delay-independent electrical stabilization. These three definitions turn out to be equivalent, which shows that constructiveness is a very robust notion.

10.3.1 Constructive Value Propagation

Given an input I, a state R, a wire expression e, and a Boolean value b, we define the *constructive evaluation* relation $I, R \vdash e \hookrightarrow b$, read "*for input I and state R, the expression e constructively evaluates to b*". The definition is:

$$
\begin{array}{lll}
 & I, R \vdash b \hookrightarrow b & \\
\text{for } w \in \mathcal{I} & I, R \vdash w \hookrightarrow b & \text{if } \ I(w) = b \\
\text{for } w \in \mathcal{R} & I, R \vdash w \hookrightarrow b & \text{if } \ R(w) = b \\
\text{for } w \in \mathcal{S} & I, R \vdash w \hookrightarrow b & \text{if } \ \mathcal{C}(w) = e \ \text{ and } \ I, R \vdash e \hookrightarrow b \\
 & I, R \vdash \neg e \hookrightarrow b & \text{if } \ I, R \vdash e \hookrightarrow \neg b \\
 & I, R \vdash e \vee e' \hookrightarrow 1 & \text{if } \ I, R \vdash e \hookrightarrow 1 \ \text{ or } \ I, R \vdash e' \hookrightarrow 1 \\
 & I, R \vdash e \vee e' \hookrightarrow 0 & \text{if } \ I, R \vdash e \hookrightarrow 0 \ \text{ and } \ I, R \vdash e' \hookrightarrow 0 \\
 & I, R \vdash e \wedge e' \hookrightarrow 1 & \text{if } \ I, R \vdash e \hookrightarrow 1 \ \text{ and } \ I, R \vdash e' \hookrightarrow 1 \\
 & I, R \vdash e \wedge e' \hookrightarrow 0 & \text{if } \ I, R \vdash e \hookrightarrow 0 \ \text{ or } \ I, R \vdash e' \hookrightarrow 0
\end{array}
$$

In the evaluation, the current value of a register w is determined by R and not by its definition expression $\mathcal{C}(w)$, which is only used to determine the next value of the register. Notice that the value of a standard wire can be determined only by determining the value of its definition expression; this is the essence of fact-to-fact propagation. The following lemma shows that evaluation is deterministic.

Lemma 7 *For any wire w, if $I, R \vdash w \hookrightarrow b$ and $I, R \vdash w \hookrightarrow b'$, then $b = b'$.*

We say that $\mathcal{C}$ is *constructive* w.r.t. I and R if, for any wire w, there exists a Boolean value b such that $I, R \vdash w \hookrightarrow b$. As in the logical semantics, the output O is such that the value of any output wire w is b if $I, R \vdash w \hookrightarrow b$ and the new state R' is such that the new value of any register w is b if $I, R \vdash \mathcal{C}(w) \hookrightarrow b$. We then write $R \underset{I}{\overset{O}{\hookrightarrow}} R'$.

By the lemmas, if $\mathcal{C}$ is constructive w.r.t. I and R, it is logically correct w.r.t. I, and $R \underset{I}{\overset{O}{\hookrightarrow}} R'$ implies $R \underset{I}{\overset{O}{\rightarrow}} R'$.

Registers do not interfere with instantaneous constructiveness analysis, but they are important for global constructiveness. We say that a circuit is *constructive* if it is constructive w.r.t. each input I and each reachable state R, where a reachable state is a state that can be reached from the initial state 0 by some input sequence.

See Section 4.4 for examples of constructive and non-constructive circuits.

10.3.2 Three-Valued Denotational Semantics

In the denotational semantics, we interpret wires in the three-valued Scott Boolean domain $B_\perp = \{\perp, 0, 1\}$, partially ordered by $\perp \leq 0$ and $\perp \leq 1$. The values 0 and 1 are called *defined* values, while $\perp$ is called the *undefined* value. The *or* and *and* operators are interpreted by the least monotonic functions that respect their pure Boolean outputs, which are classically called *parallel or* and *parallel and*, see [33]. Therefore, one has $\perp \vee 1 = 1 \vee \perp = 1$ and $\perp \wedge 0 = 0 \wedge \perp = 0$.

Consider a circuit $\mathcal{C}$ with n standard wires, and number these wires from 1 to n. Let $S \in B_\perp^n$. Given an input I and a state R, the circuit $\mathcal{C}$ defines a monotonic function $\mathcal{C}(I, R) : B_\perp^n \mapsto B_\perp^n$. For any assignment S of values in $B_\perp$ to standard wires, this function defines another assignment S', where the value of each standard wire w is that of its definition expression $\mathcal{C}(w)$ computed in the environment I, R, S. Let $Y(I, R) \in B_\perp^n$ be the least fixpoint of $\mathcal{C}(I, R)$.

Theorem 4 *Let $\mathcal{C}$ be a circuit, let I be an input, and let R be a state. Then, for any standard wire w_i, one has $I, R \vdash w_i \hookrightarrow b$ if and only if $Y(I, R)_i = b$.*

It follows that a circuit is constructive w.r.t. I and R if and only if all components of the least fixpoint $Y(I, R)$ are defined. Then, the new state is computed as usual as the value of the register definition expressions, which are all defined.

10.3.3 The Electrical Semantics

The electrical semantics deals with temporal propagations of idealized voltages through electrical gates that have delays. It is easier to explain with gates and wires, i.e. with diagrams. We use the *up-bounded inertial delay model* of [15]. We consider only the combinational part and ignore the registers, which play no role in the instantaneous analysis.

Wires are interpreted as functions from the positive reals (time) to $\{0,1\}$. With any gate that drives a wire w, we associate a positive real number $\delta(w)$ called the *delay* of the gate. The following property must hold for the gate:

(i) If, at time t, the value of w changes from 0 to 1 or from 1 to 0, then the value of some input wire of the gate must have changed at some time t' such that $t - \delta \leq t' \leq t$

(ii) If the values of the gate inputs are stable between time $t - \delta(w)$ and t included, then the value of w at time t is the Boolean value determined by the gate logical function and the input values.

We also allow for arbitrary wire delays by making it possible to insert delayed buffers anywhere on wires.

Notice that the delay model yields lots of freedom. For example, transient pulses shorter than $\delta(w)$ may or may not show up on the output. The only things that are required are that the output changes only because of inputs and that the output must receive its logical value if the inputs are kept stable long enough.

Given a circuit $\mathcal{C}$, a delay assignment to gates, and an input I kept stable from time 0 on, we say that the circuit *electrically stabilizes in time* t if all the wires keep a stable Boolean value after time t.

Theorem 5 *Let $\mathcal{C}$ be a circuit, let I be an input and R be a state. Then $\mathcal{C}$ if constructive if and only if, for any delay assignment, all wires stabilize after some time t. The stable electrical values of the wires are those determined by constructive Boolean propagation.*

Since the number of inputs and states is finite, there is a maximum stabilization time valid for all inputs and all states. This time is called the *clock period*. After it has elapsed, one can set the new values of the registers and process new inputs[2].

[2] Our model is not fully respective of electrical phenomena. One should wait for a setup

Therefore, the constructive semantics abstracts away gate delays. Cyclic constructive circuits behave synchronously, as do acyclic circuits.

10.4 Syntactic Extensions

We now discuss the two syntactic extensions we shall need for the formal translation from Esterel to circuits.

10.4.1 Hierarchical Circuits

In the translation of Esterel programs into circuits, we shall define circuits by structural induction over statements. For this, we shall include already defined circuits in the new circuits we build and connect them using appropriate wires and gates.

In diagrams, we simply represent an included subcircuit as a rectangle with inputs on the left-hand side and outputs on the right-hand side and consider it as a new kind of gate.

In equational definitions, we use the same technique as for submodule inclusion in the full Esterel language. To include a subcircuit C' in a circuit $\mathcal{C}$, we simply write the name $\mathcal{C}'$ in place of a definition in $\mathcal{C}$. This amounts to renaming the local and register wires of $\mathcal{C}$ by new names unknown in $\mathcal{C}$ to avoid name clashes and copying the renamed definitions of $\mathcal{C}'$ into $\mathcal{C}$. The input and output variables of $\mathcal{C}'$ are captured by the variables of $\mathcal{C}$ that have the same name. An input variable of $\mathcal{C}'$ can be captured by any variable of $\mathcal{C}$. Since it has a definition in $\mathcal{C}'$, an output variable of $\mathcal{C}'$ cannot be defined elsewhere in $\mathcal{C}$.

We can also rename input or output variables of $\mathcal{C}'$ before copying it, using the notation $\mathcal{C}'[w_1/w'_1,\, w_2/w'_2, \ldots]$. Then the input or output names $w'_1, w'_2, \ldots$ of $\mathcal{C}'$ are captured by the names $w_1, w_2, \ldots$ of $\mathcal{C}$. The formal definition is trivial and boring.

10.4.2 Definitions by Implications

The Esterel circuits will contain many *or*-gates used to gather signal emissions and completion codes. Instead of making these gates explicit in the equations, we shall leave them implicit and define the output of an *or*-gate

time when setting the new register values. Furthermore, we assume that standard wires are recomputed only from new inputs and that their old value has disappeared. See [39] for a more accurate discussion of the electrical model.

as being separately implied by each input. This will greatly shorten our formal translation in Chapter 13.

Syntactically, we use *implication* definitions of the form $w \Leftarrow e$, which are read "e implies w". There can be several implications $w \Leftarrow e_i$ per wire, and the value of w is then defined by the equality $w = \bigvee_i e_i$. This way, an equation

$$w = e \vee e'$$

can be replaced by two implications

$$\begin{aligned} w &\Leftarrow e \\ w &\Leftarrow e' \end{aligned}$$

which do not need to be syntactically related to each other. In hierarchical circuits, the implication definitions of a wire will actually be scattered in several subcircuits. This is the main advantage of the extension, which has some object-oriented flavor: $w \Leftarrow e$ can be read "send e to w as an input".

If a wire has one implication definition, it can only have other implication definitions; it is not allowed to mix the different definition forms. A local or output wire that has no definition in a circuit is assumed to be defined by an empty implication and to have value 0. This exactly corresponds to our previous graphical convention that an undriven output is implicitly unset.

Chapter 11

The Basic Circuit Translation

In this informal (but precise) chapter, we explain the basic principles of the translation of Esterel kernel programs into circuits. The translation we present is essentially that of [4], with a modification in the handling of the parallel statement w.r.t. constructiveness. The basic translation is only partially correct because of the schizophrenia problem already mentioned in [4]. This problem will be solved later on in Chapter 12.

11.1 Structure of the Esterel Circuit

The basic translation is purely structural, and it closely follows the state semantics rules. The substatements of a statement are first translated, and the obtained circuits are then combined using appropriate auxiliary gates and wiring.

Since only '1' (`pause`) statements are decorated in the state semantics, we associate a register with each '1' statement, all the other statements only generating combinational logic. Selected '$\hat{1}$' statements correspond to registers set in the current circuit state. An additional boot register is introduced to implement the added initial '$\hat{1}$' state. The reactions exactly correspond to clock cycles.

11.2 The subcircuit generated by a statement

We now describe the interface of the circuit generated by a statement and the way in which the circuit should behave.

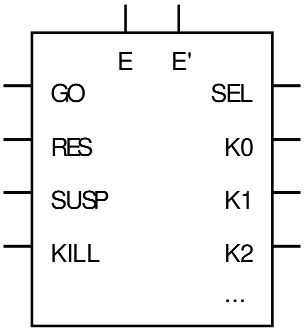


Figure 11.1: Circuit associated with a statement

11.2.1 Subcircuit Interface

The interface is pictured in Figure 11.1. The left pins and the leftmost top pin `E` are input ones, while the right pins and the rightmost top pin $\mathtt{E}'$ are output ones. The meaning of the pins is as follows:

- The `GO` input pin is used to start the statement afresh, i.e. to execute an s-rule in the state semantics. This occurs when `GO` is set.

- The `RES` input pin is used to resume the execution of a selected statement, i.e. to execute an r-rule in the state semantics. This occurs when `RES` is set.

- The `SUSP` input pin is used to suspend the execution of the statement, according to the state semantics rule *(r-suspend+)*. Suspension occurs when `SUSP` is set. Then, the registers keep their current value unless killed because of the `KILL` input below.

- The `KILL` input pin is used to unset the registers of the statement in case of a trap exit. This occurs when `KILL` is set by the translation of a trap statement $\{p\}$, as specified by rule *(sr-term-trap)*. The `KILL` signal is propagated by all statements towards the `pause` registers.

- The `SEL` selection output pin indicates that the statement is a state currently selected for resumption, i.e. that some internal `pause` register is set. The `SEL` signal is simply the disjunction of the internal registers.

- The output pins `K0`, `K1`, etc. correspond to completion codes. There are $n+2$ such pins if n is the number of traps in which the statement is enclosed. When the statement is either started or selected and resumed, the pin that corresponds to the completion code returned by the statement is set. If the statement is not executed, i.e. if it is not started, not selected, or not resumed if selected, then the statement circuit explicitly unsets all `K` pins. Notice that completion codes are unary encoded (one-hot in hardware terminology).

- The pins `E` and `E`$'$ correspond to the signal interface. They are not simple pins but compound pins or *buses*, containing one elementary pin per signal visible in the scope of the current statement. The `E` input bus is for input to the statement, corresponding to the input event E in the semantics. The `E`$'$ output bus pin is for output from the statement, corresponding to the output event E'. We shall freely extract specific signals `s` or `s`$'$ out of `E` or `E`$'$. As for the `K` pins, the `E`$'$ pins are explicitly unset when the statement is not executed, i.e. when $\neg(\texttt{GO} \vee (\texttt{RES} \wedge \texttt{SEL}))$.

11.2.2 Execution Scheme

The basic execution scheme consists in first setting `GO` to start execution, and then setting `RES` at each clock cycle. At each cycle, control propagates combinationally within the statement circuit; the completion wire corresponding to the returned completion code is set, and the registers corresponding to the reached `pause` statements are set to resume execution from the right state in the next cycle. The visible Esterel signals are received and emitted through `E` and `E`$'$.

To suspend the statement for the cycle, we set `SUSP` instead of `RES` (`RES` and `SUSP` are never set at the same time). If the statement is preempted by some internal or concurrent trap exit, we set `KILL` to unset the `pause` registers.

The selection wire `SEL` is propagated upwards in compound statements. It remains set as long as some pause registers are set, i.e. as long as the statement is selected. It becomes unset at the cycle that follows termination or trap exit, unless the statement has been immediately restarted by some loop. The `SEL` wire is necessary since `RES` may also be sent to currently unselected statements. When `RES` is set, unselected statements should remain silent, i.e. should unset all output signal and completion wires. This is done by internally anding `RES` and `SEL`.

Figure 11.2: Circuit for a completion code $k \neq 1$

Since it is possible to instantaneously loop back into a statement, several elementary behaviors of a subcircuit can be superimposed in the same instant. This will be neglected for the moment and fully analyzed in Chapter 12.

11.2.3 Constructive Semantics of the Program and Circuit

The circuit of a statement is carefully designed to achieve full correspondence between the constructive information propagation in the circuit and the constructive semantics of the source Esterel statements. The propagation of 1's implements the *Must* calculation and state change as specified by the semantic rules. The propagation of 0's exactly implements the *Cannot* calculation. The partial statuses $\{+, -, \perp\}$ of a signal s are encoded as follows: status is s^+ if the `s` wire has been proved to have value 1, status is s^- if the `s` wire has been proved to have value 0, and status is $s^\perp$ if nothing has been proved for `s`. See Section 11.5 for a concrete example.

11.3 Translating the Kernel Esterel Statements

We now show by pictures how each Esterel kernel statement is translated into a circuit. To make the pictures simpler, we use two conventions:

- Unused inputs are not pictured.
- Not all outputs are pictured. An omitted output is assumed to be explicitly unset, i.e. to be driven by a 0 constant (see Section 10.1.2).

11.3.1 Translation of Completion Codes

The translation of a completion code $k \neq 1$ (termination or trap exit) is obvious: the `GO` input is connected to the corresponding completion output,

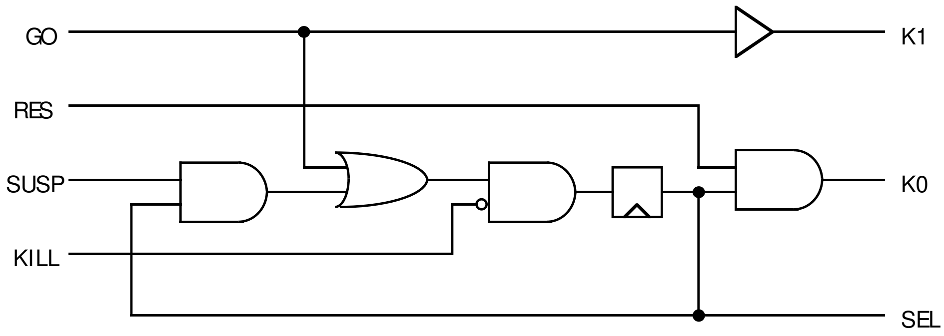


Figure 11.3: Circuit for `pause`, i.e. completion code 1

as shown in Figure 11.2. This implements rule *(s-term-exit)*, as well as *Must* and *Can* rules.

The translation of the `pause` statement, i.e. completion code 1, is pictured in Figure 11.3. The `pause` register drives the `SEL` selection wire. The register is potentially set in two cases:

- when `GO` is set, i.e. when the `pause` statement starts, according to rule *(s-pause)*;

- when `SUSP` and `SEL` set are set, i.e. when the `pause` statement is selected, according to rules *(r-suspend+)*.

According to rules *(sr-term-trap)* and *(sr-prop-trap)*, actual setting of the register occurs only if `KILL` is unset, i.e. only if the `pause` statement is not preempted by some trap exit.

Termination wire `K1` is set when `GO` is set, according to rule *(s-pause)*. Termination wire `K0` is set when the `pause` statement terminates, i.e. when both `SEL` and `RES` are set, according to rule *(r-pause)*. The *Must* and *Can* rules are also correctly implemented.

11.3.2 Translation of the Emit Statement

As shown in Figure 11.4, to translate !s, we just connect the input `GO` wire to the `K0` termination output pin and to the `s'` signal emission output pin. This implements rule *(s-emit)*.

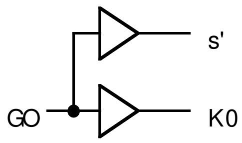


Figure 11.4: Circuit for $!s$

11.3.3 Translation of the Present Statement

The translation of a present statement $s\,?\,p\,,q$ is shown in Figure 11.5. We put the circuits of p and q in parallel, we build a switch for the `GO` wire according to the value of the `s` input that we extract from `E`, we send `RES`, `SUSP`, and `KILL` to both subcircuits, and we join their outputs using *or* gates.

Since the switch on `GO` makes the respective `GO` inputs of the subcircuits incompatible, only one of them can be started at a time. Therefore, in each following instant, only one of the subcircuits can be selected, as in the state semantics. Constructive information propagation works as follows:

- When `GO` is set, i.e. when the circuit is started, there are three cases.
  - The `s` wire is set, i.e. s is known to be present. The `GO` input of p is set, which implements rule *(s-present+)*. The `GO` input of q is unset, and the circuit of q unsets all its output pins. Since 0 is neutral for an *or* gate, the 1's and 0's generated by p propagate to the $\mathtt{E}'$ and `Ki` outputs, exactly as required by the definition of *Must* and *Can*.
  - The situation is symmetrical if `s` is unset, i.e. if s is known to be absent.
  - If s is yet unknown, then the `GO` inputs of the subcircuit are also unknown. No 1 propagates to outputs, yielding the $\langle\,\emptyset\,,\,\emptyset\,\rangle$ result required by the *Must* formula. As far as 0's are concerned, we must calculate

$$\overline{Can^{\perp}(p,E) \cup Can^{\perp}(q,E)} = \overline{Can^{\perp}(p,E)} \cap \overline{Can^{\perp}(q,E)}$$

  This is indeed what the output *or* gates implement, since 0 propagates through an *or* gate if and only if both inputs are 0. Correct computation of $\overline{Can}$ is propagated by induction from p and q to $s\,?\,p\,,q$.

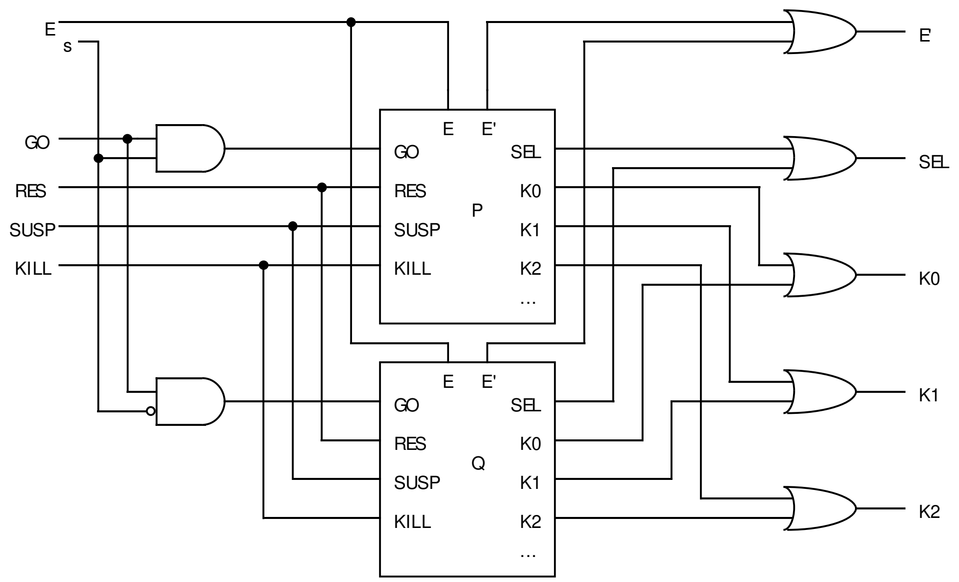


Figure 11.5: Circuit for $s\,?\,p\,,q$

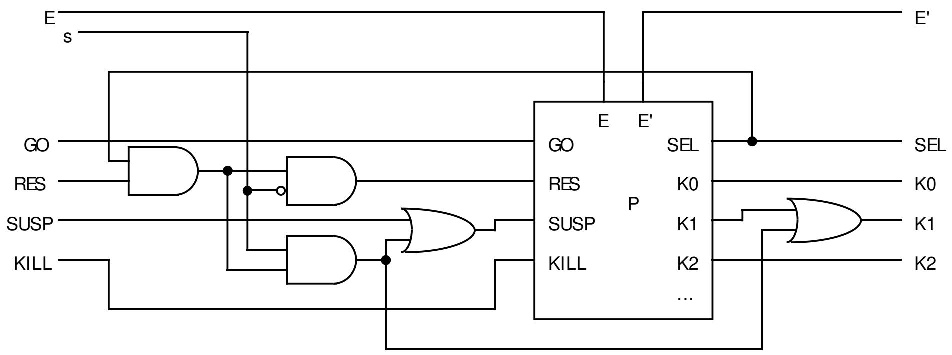


Figure 11.6: Circuit for $s \supset p$

- Assume RES is set. It is then broadcast to the circuits of p and q. There are two subcases:
  - If SEL is unset, the circuit is not currently selected. This is also true for the subcircuits of p and q, which unset all their outputs. The output *or* gates adequately unset the outputs.
  - If SEL is set, then only one of the subcircuits is selected. Assume it is that of p. Then, p is executed as required by rule *(r-then)*, and q unsets all its outputs, which ensures correct propagation of p's results and *Can* values by the output *or* gates.

11.3.4 Translation of suspension and strong preemption

The translation of the suspension statement $s \supset p$ is shown in Figure 11.6. Since suspension does not occur in the first instant, the `GO` wire is directly connected to the subcircuit's `GO` input pin. The resumption wire `RES` is *and*-ed with the selection wire `SEL` to avoid resuming an unselected subcircuit, and this conjunction drives a switch on the `s` signal, which is extracted from `E`. If `s` is absent, the `RES` input pin of the subcircuit is set for normal resumption. If `s` is present, the `SUSP` suspension pin of the subcircuit is set for suspension, and `K1` is set since we must return completion code 1. Suspension of the subcircuit also occurs if the input `SUSP` wire is 1. The `KILL` wire is directly fed into the subcircuit.

Although it is not a kernel statement, it is practically useful to directly implement the strong abortion statement $s >\!\!> p$ or "`do` p `watching` s", since

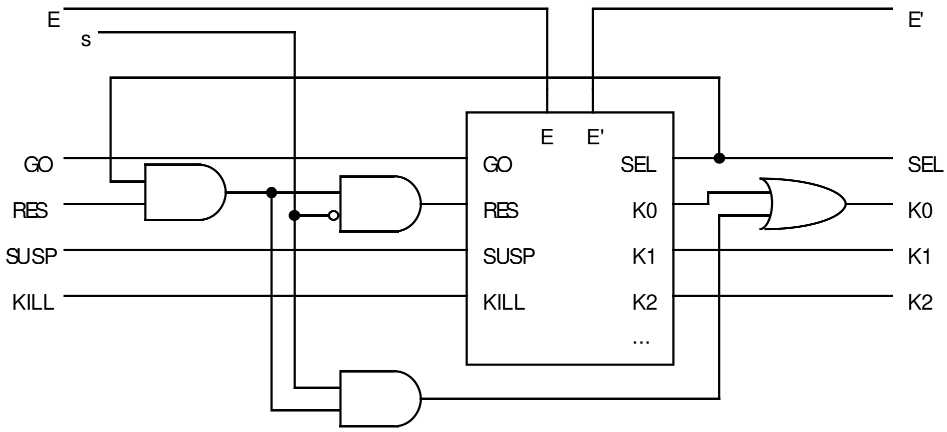


Figure 11.7: Circuit for $s \gg p$

its kernel expression given in Chapter 2.4 is quite heavy. The implementation is shown in Figure 11.7. Since preemption does not occur in the first instant, the `GO` wire is directly connected to the subcircuit's `GO` input pin. The `SUSP` and `KILL` wires are also directly connected. Once *and*-ed with `SEL`, the `RES` wire drives a switch on `s`. If `s` is absent, resumption is passed to the subcircuit. If `s` is present, the subcircuit `RES` pin receives 0 and the `K0` termination pin receives 1 to indicate termination.

11.3.5 Translation of Sequencing

The translation of a sequence $p\,;\,q$ is easy and shown in Figure 11.8. The input `GO` wire is sent to the `GO` pin of p, and the termination wire `K0` of p is sent to the `GO` input of q to ensure immediate sequencing. All the other input wires are broadcast, and the output wires are *or*-ed. Correctness of the translation relies on the fact that the subcircuits of p and q cannot be both simultaneously selected.

Constructive propagation of 1's and 0's calculates *Must* and $\overline{Can}$ in the same way as for $s\,?\,p\,,q$. This is left to the reader.

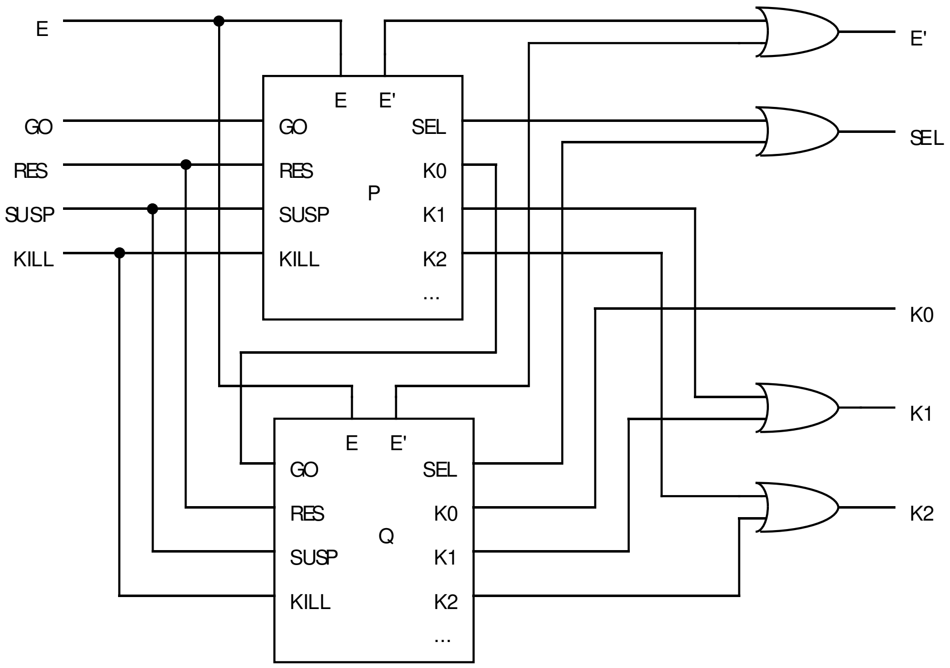


Figure 11.8: Circuit for $p\,;\,q$

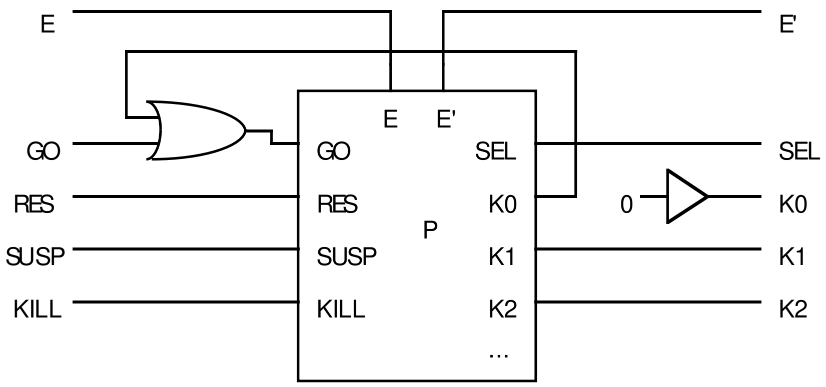


Figure 11.9: Circuit for $p*$

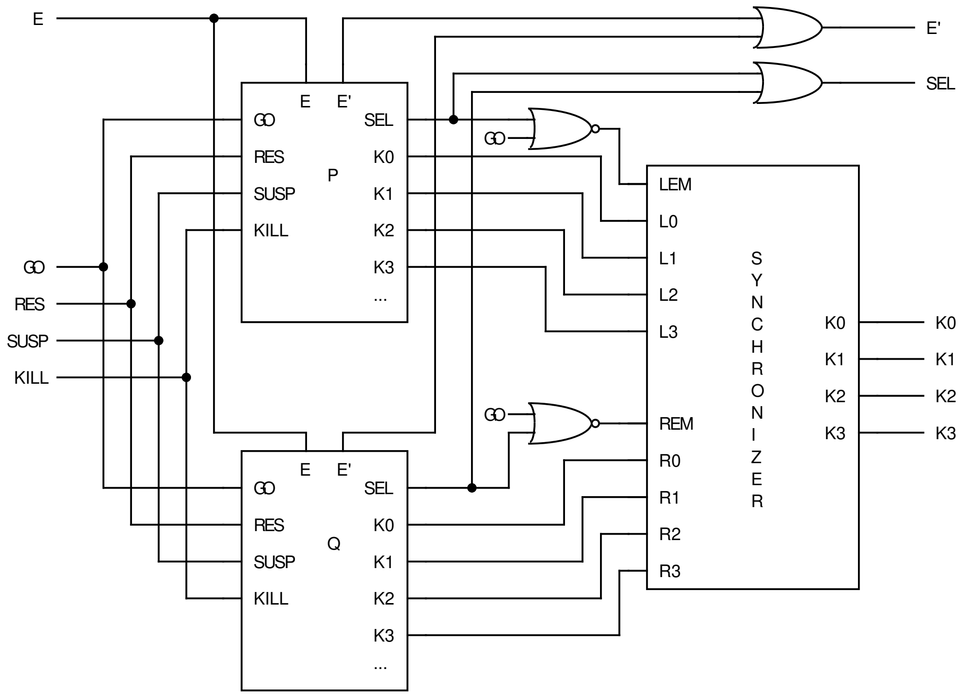


Figure 11.10: Circuit for $p \mid q$

11.3.6 Translation of Loop

As shown in Figure 11.9, to translate a loop $p*$, we simply feedback the `K0` termination wire into the `GO` input. Of course, this can create combinational cycles. A cycle occurs if there is a direct path from `GO` to `K0`. Requiring a program to be loop-safe is enough to make this problem harmless. Nastier cycles may appear through signal wires; their analysis deferred to Chapter 12. Since a loop cannot terminate, the output termination wire `K0` has value 0, which is made explicit here for clarity.

11.3.7 Translation of Parallel

The circuit of a parallel statement $p \mid q$ is pictured in Figure 11.10. It is built by putting the circuits of p and q in parallel and synchronizing their outputs using a synchronizer subcircuit to achieve the required semantical effect, i.e.

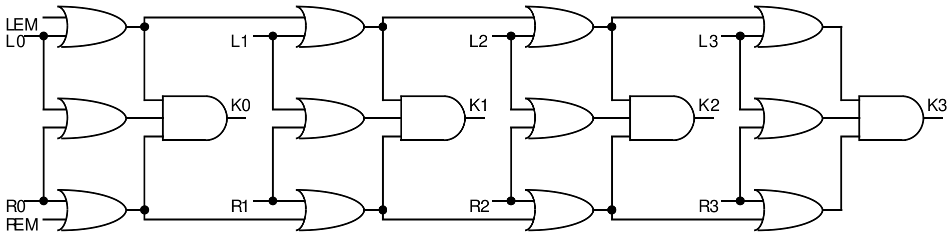


Figure 11.11: The constructive *Max* circuit

the *Max* computation of completion codes.

The `GO`, `RES`, `SUSP`, `KILL` and `E` wires are broadcast to the subcircuits of p and q. The global `SEL` output is the disjunction of the `SEL` output of the components, and the output signal wires are *or*-ed according to the semantic rules.

The heart of the synchronizer is the constructive *Max* circuit pictured in Figure 11.11. Given two sets of completion codes L and R, the *Max* circuit computes $Max(L, R)$ using the following formula[1].

$$\begin{aligned} Max(L,R) &= \{max(l,r) \mid l \in L, r \in R\} \\ &= \{i \in N \mid i \geq min(L)\} \ \cap \ (L \cup R) \ \cap \ \{j \in N \mid j \geq min(R)\} \end{aligned}$$

Each of the three components of the intersection is computed by a row of `or` gates in the *Max* circuit. The auxiliary `LEM` and `REM` inputs are used to control the cases where the left or right set is empty. Assume for example $L = \emptyset$. In the circuit, this is represented by `LEM` = 1 and `L0` = `L1` = ... = 0. In this case, the upper part of the circuit is innocuous and the result is R as required by the formula $Max(\emptyset, R) = R$.

In the parallel circuit, `LEM` and `REM` are controlled according to rules *(r-both)*, *(s-both)*, *(r-left)*, and *(r-right)*.

- When resuming $\hat{p} \mid \hat{q}$ according to rule *(r-both)* or when starting $p \mid q$ according to rule *(s-both)*, we know that each branch will return a termination code and we unset `LEM` and `REM`. Given the left and right

[1] The reader familiar with the application of 2-adic number theory to circuits‘ř [42] might prefer a beautiful arithmetico-logical formula due to G. Gonthier, where ‘$-$’ denotes arithmetic opposite: $Max(L, R) = (L \vee -L) \wedge (L \vee R) \wedge (R \vee -R)$.

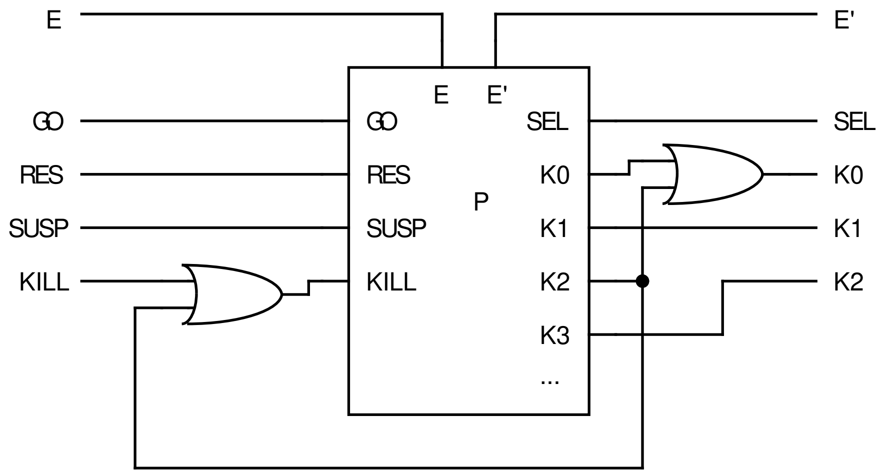


Figure 11.12: Circuit for $\{p\}$

completion codes l and r, represented by $\mathtt{L}l = 1$ and $\mathtt{R}r = 1$, the *Max* circuits compute the maximum $k = max(l, r)$, represented by $\mathtt{K}k = 1$. In addition, the *Max* circuits propagate 0's as required by the computation of $\overline{Can}$. For example, from $\mathtt{L0} = \mathtt{L1} = 0$ we deduce $\mathtt{K0} = \mathtt{K1} = 0$ without any knowledge of `R0` and `R1`. (In [4], we used a simpler circuit to compute $max(k, l)$; it turns out that this circuit did not propagate 0's in the same way and did not implement the constructive semantics.)

- When resuming $\hat{p}|q$ according to rule *(r-left)*, `REM` is set by the *nor*-gate in the parallel circuit, and the 0's and 1's of $\hat{p}$ are directly propagated to the outputs of the parallel circuit.

- When resuming $p|\hat{q}$ according to rule *(r-right)*, `LEM` is set symmetrically.

11.3.8 Translation of Trap and Shift

The circuit for $\{p\}$ is shown in Figure 11.12. As required by the definition of the $\downarrow k$ operator on completion codes, we *or* the termination wires of code 0 (termination) and 2 (exit of current trap), to build the new termination wire `K0`. The wire `K1` is left unchanged, and the other termination wires are shifted downwards.

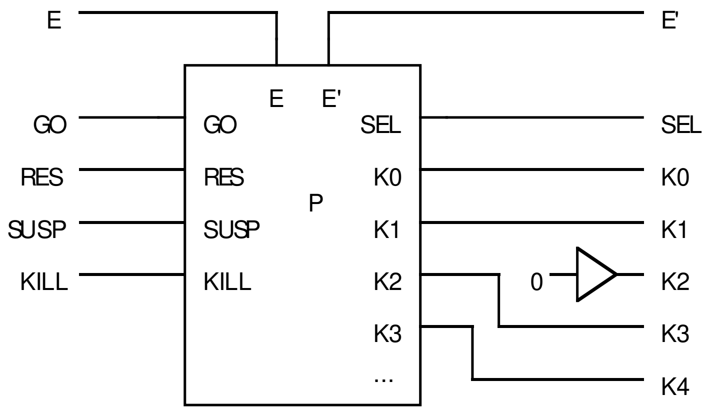


Figure 11.13: Circuit for $\uparrow p$

In the translation of $\uparrow p$, shown in Figure 11.13, the completion codes are simply shifted according to the definition of the $\uparrow k$ operator.

11.3.9 Translation of local signal declaration

To translate a local signal declaration $p \backslash s$, we simply close the signal s by equating the `s` and `s'` wires. This is pictured in Figure 11.14. The closure can create combinational loops to be handled by the constructiveness analysis of the circuit, see Chapter 4.

Constructive information is correctly propagated according to the recursive definitions of Chapter 7. If `s` is emitted by p, then p sets `s'`, a fact that is fedback into `s` as in the recursive definition of *Must*. If `s` cannot be emitted, then p unsets `s'`, which is fed back into `s`.

11.4 The Global Environment

To translate a full program of body p into a circuit, we must place the circuit of p in a suitable environment pictured in Figure 11.15. The `GO` wire is initially set and then unset at all following cycles by the *boot register*, which corresponds to the added initial $\hat{1}$ in the state semantics. Notice that a register with a negation added on each side acts as a register initialized to 1. The `RES` wire is permanently set; the initial value 1 is harmless since no sub-

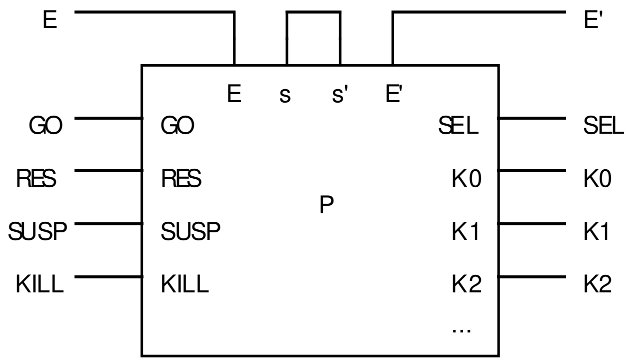


Figure 11.14: Circuit for $p\backslash s$

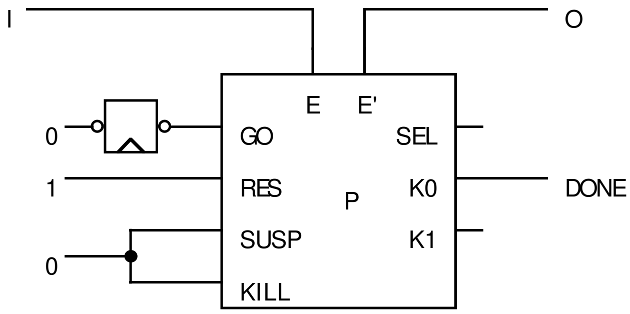


Figure 11.15: The global environment

circuit is initially selected. The global SUSP and KILL wires are permanently unset. The KO output is renamed into DONE to indicate that the body has terminated, an information that is not part of the semantics but convenient in practice. The global SEL and K1 wires are useless. Finally, the input and output wires are respectively connected to the I and O signal buses.

11.5 A Constructive Execution Example

Using example P2, page 29, we now show how the constructive evaluation of an Esterel program is adequately performed by the constructive evaluation of its circuit. We recall the code of P2:

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

The circuit obtained from P2 is pictured in Figure 11.16. To make the drawing simpler, we simplified the translation of pause, using the global environment definitions RES $= 1$ and SUSP $=$ KILL $= 0$: in this environment, a pause statement boils down to a register.

In the first instant, the boot register sends a 1 to the S$'$ wire, thus executing "emit S", and a 1 to the test for O, which is represented by the leftmost and gates. Since O's status is yet unknown, control cannot propagate further, neither to the O+ wire, which triggers the then case, nor to the O- wire, which triggers the else case. However, the information S$' = 1$ feeds back into the test for S, which is represented by the two middle and gates. From S $= 1$, we deduce S$- = 0$. Since the pause register is initially 0, we deduce O$' = 0$, i.e. that we cannot emit O. From O $= 0$, we now deduce O$+ = 0$ and O$- = 1$, which determines all the other wires.

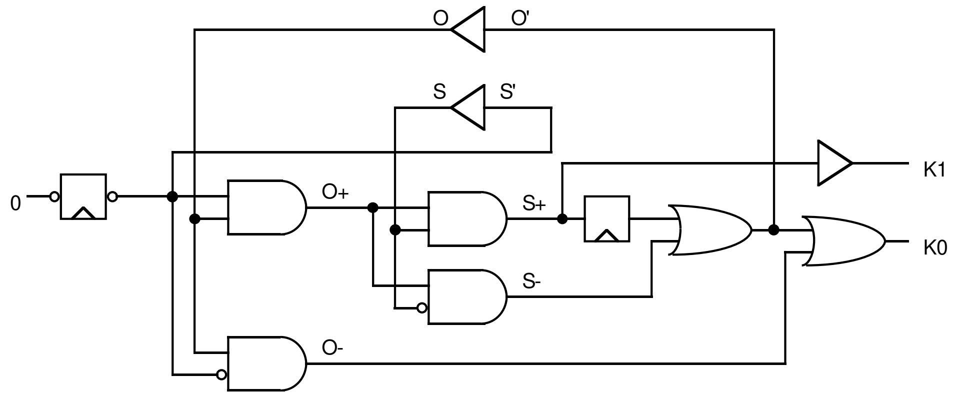


Figure 11.16: Circuit for program P2

Chapter 12

Schizophrenia

The relatively simple translation of a loop-safe Esterel program P into a hardware circuit C we have given in Chapter 11 is essentially linear in size: the size of C measured in number of gates is linearly bigger than the size of P measured in number of kernel statements. If the circuit C is constructive, then it can be shown that the program P is also constructive and that the behavior of C coincides with that of P. However, the basic translation is only partially correct: there exist constructive programs that yield non-constructive cyclic circuits, because of the schizophrenia problem we study in this section.

The problem is related to combinational feedback. In the circuit construction, there are exactly three places where an output of a subcircuit is combinationally fed back into the same subcircuit:

- The translation of a loop $p*$. This is the source of the schizophrenia problem.
- The translation of a trap statement $\{p\}$, where the `K2` output is fed back into the `KILL` input. This feedback is harmless since the `KILL` wire is propagated straight to the registers by all statement translations. No combinational loop can be created.
- The translation of a local signal statement $p \backslash s$. Creating combinational loops involving signals is normal, there as in Esterel proper. The whole point of constructive analysis is to determine whether signal loops are sensible.

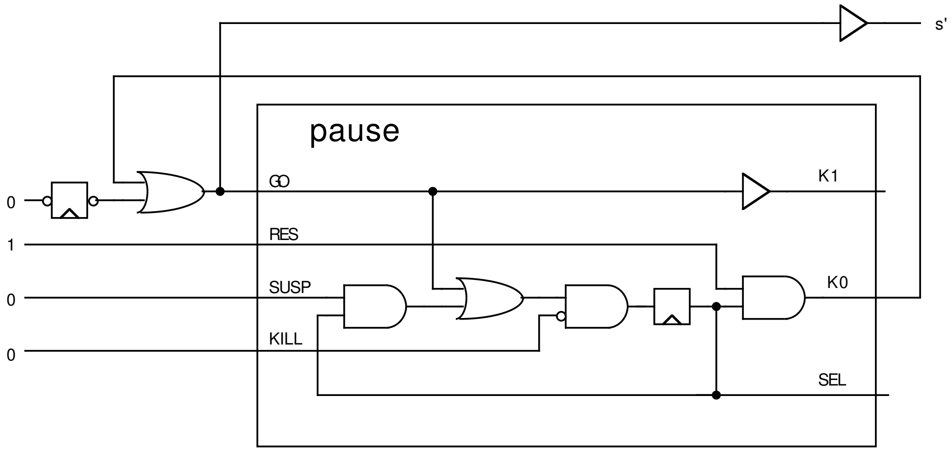


Figure 12.1: Circuit for `sustain s` = $(!s\,;\,1)*$

12.1 A Correctly Translated Loop

Consider the statement "`sustain S`", i.e. $(!s\,;\,1)*$. The circuit translation is pictured in Figure 12.1. The `K0` output of the `pause` subcircuit feeds back to the `GO` input, but no combinational cycle is created and the translation is correct. Of course, the circuit almost vanishes by constant propagation.

In the second instant, the *(r-pause)* and *(s-pause)* behaviors are harmlessly superimposed in the `pause` subcircuit: both `RES` and `GO` inputs of the subcircuit are set, and the subcircuit sets both its `K0` and `K1` completion outputs.

12.2 Schizophrenic Parallel Synchronizers

Consider now the following apparently trivial variant of "`sustain S`":

```
module P16:
loop
   emit S;
   [ nothing || pause ]
end loop
```

written $(!s\,;\,(0\,|\,1))*$ in terse syntax.

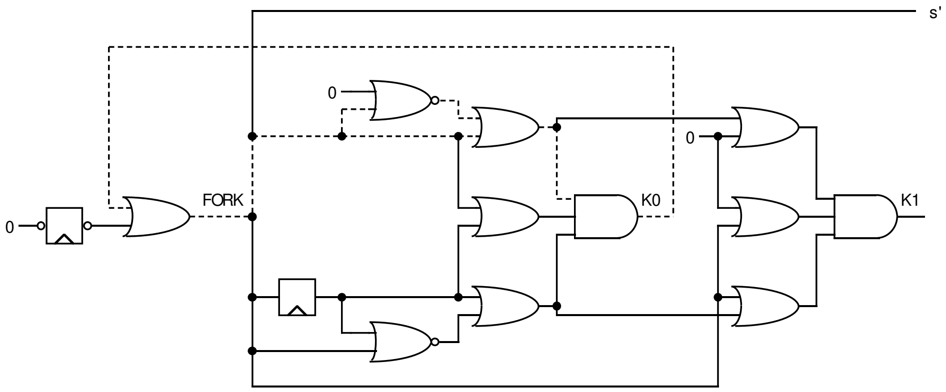


Figure 12.2: Incorrect basic circuit for $(!s\,;\,(0 \mid 1))*$

Semantically speaking, the added `nothing` statement is completely innocuous. However, the basic translation builds the non-constructive circuit pictured in Figure 12.2. The `FORK` label shows where the parallel statement starts. The loop feeds back the `K0` termination output to `FORK`. The unstable combinational loop is drawn in dotted lines. In the first instant, the circuit is constructive: `S` is emitted, the boot register is unset, and the `pause` register is set. In the second instant, the `pause` register sets the two lower inputs of the `K0` completion gate. Therefore, to compute the value of this gate, we need to compute the dotted top input. For this, we have to compute the value of the `FORK` wire, which itself requires computing the value of `K0`, hence the non-constructiveness.

Consider now the state semantics. Omitting the auxiliary boot statement, the state of interest is $(!s\,;\,(0 \mid \hat{1}))*$. The constructive state transition is

$$(!s\,;\,(0 \mid \hat{1}))* \xhookrightarrow[s^+]{s^+,1} (!s\,;\,(0 \mid \hat{1}))*$$

This transition is proved using rule *(r-do-loop)* in the following way:

$$\frac{!s\,;\,(0 \mid \hat{1}) \xhookrightarrow[s^+]{\emptyset,0} !s\,;\,(0 \mid 1) \qquad !s\,;\,(0 \mid 1) \xhookrightarrow[s^+]{s^+,1} !s\,;\,(0 \mid \hat{1})}{(!s\,;\,(0 \mid \hat{1}))* \xhookrightarrow[s^+]{s^+,1} (!s\,;\,(0 \mid \hat{1}))*}$$

The first premise is proved using rules *(r-seq3)*, *(r-right)*, and *(r-pause)*, while the second premise is proved using rules *(s-seq2)*, *(s-emit)*, *(s-both)*, *(s-term-exit)* for 0, and *(s-pause)* for 1.

The key point is that the parallel statement is *instantaneously reincarnated* by the loop: when resumption of state $0 \mid \hat{1}$ terminates, the parallel is reincarnated as $0 \mid 1$. Each incarnation performs a separate transition:

$$\frac{\hat{1} \xhookrightarrow[s^+]{\emptyset,\,0} 1}{0 \mid \hat{1} \xhookrightarrow[s^+]{\emptyset,\,0} 0 \mid 1} \qquad \textit{(r-right)}$$

$$\frac{0 \xhookrightarrow[s^+]{\emptyset,\,0} 0 \qquad 1 \xhookrightarrow[s^+]{\emptyset,\,1} \hat{1}}{0 \mid 1 \xhookrightarrow[s^+]{\emptyset,\,1} 0 \mid \hat{1}} \qquad \textit{(s-both)}$$

In the basic circuit of Figure 12.2, there is a single parallel synchronizer whose wires cannot be reincarnated dynamically. We ask the synchronizer to perform two distinct synchronizations at a time, which is impossible. This is what we call schizophrenia. Clearly, some logic duplication is necessary to get rid of schizophrenia.

12.3 Schizophrenic Signals

Local signals are also subject to instantaneous reincarnation. Consider the following statement:

```
module P17:
loop
   signal S in
      present S then emit O else nothing end;
      pause;
      emit S
   end signal
end loop
```

written $(((s\,?\,!o\,,0)\,;\,1\,;\,!s)\backslash s)*$ in terse syntax. In the first instant, `S` is absent, `O` is not emitted, and we reach the state $(((s\,?\,!o\,,0)\,;\,\hat{1}\,;\,!s)\backslash s)*$, omitting the dead boot statement. In the second instant, execution resumes from $\hat{1}$, the signal `S` is emitted, and the loop loops. Since the declaration of `S` is

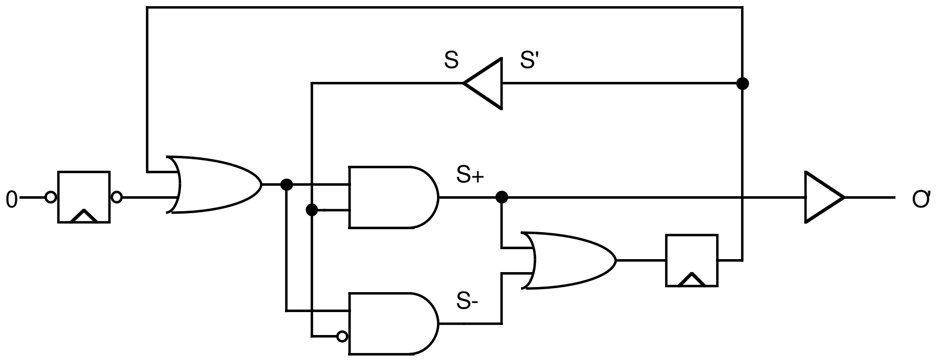


Figure 12.3: Incorrect basic circuit for $(((s\,?\,!o\,,0)\,;\,1\,;\,!s)\backslash s)*$

re-entered, the body is executed with a *fresh incarnation* of `S`, which is not emitted. Therefore, the test takes its `else` branch, `O` is not emitted, and we are back to the same state.

The (simplified) basic circuit translation is pictured in Figure 12.3. The circuit is constructive, but it behaves incorrectly in the second instant. The `S'` wire is set by the `pause` register, and it is directly fed back into the presence test. The test takes its `then` branch and provokes emission of `O` instead of taking its `else` branch as in the semantics.

The key point is that the basic translation does not take into account the scope of `S`. The statement is translated as if it were $((s\,?\,!o\,,0)\,;\,\hat{1}\,;\,!s)*\backslash s$, which indeed emits o in the second instant since s does not reincarnate any more. A finer translation is clearly needed to correctly handle the instantaneous reincarnation of `S` induced by scoping.

12.4 An Easy Solution

An easy solution is to duplicate the body of each loop, transforming $p*$ into $(p\,;\,p)*$, which is semantically equivalent. The parallel example becomes

```
loop
   emit S;
   [ nothing || pause ];
   emit S;
   [ nothing || pause ]
end loop
```

and the basic circuit now has two distinct synchronizers able to perform the two simultaneous synchronizations. The signal example becomes

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

and the basic translation now allocates two distinct wires for the two syntactically distinct local signals.

To correctly handle nested loops, duplication must be done in a recursive way: inner loops must be first expanded before duplicating a loop body. The circuit can become exponential in the size of the program, which is not practical. Furthermore, registers get duplicated, which is very bad for optimization and verification purposes. We need a better solution, which we shall find by analyzing more carefully the structure of proofs in the state semantics.

12.5 The Surface and Depth of a Statement

Schizophrenic conflicts are always created by rule *(r-do-loop)*. In this rule, the subjects of the premises are respectively a proper state $\hat{p}$ and the base standard statement p. Therefore, conflict occurs only between the *surface* and the *depth* of the circuit, which are defined as follows:

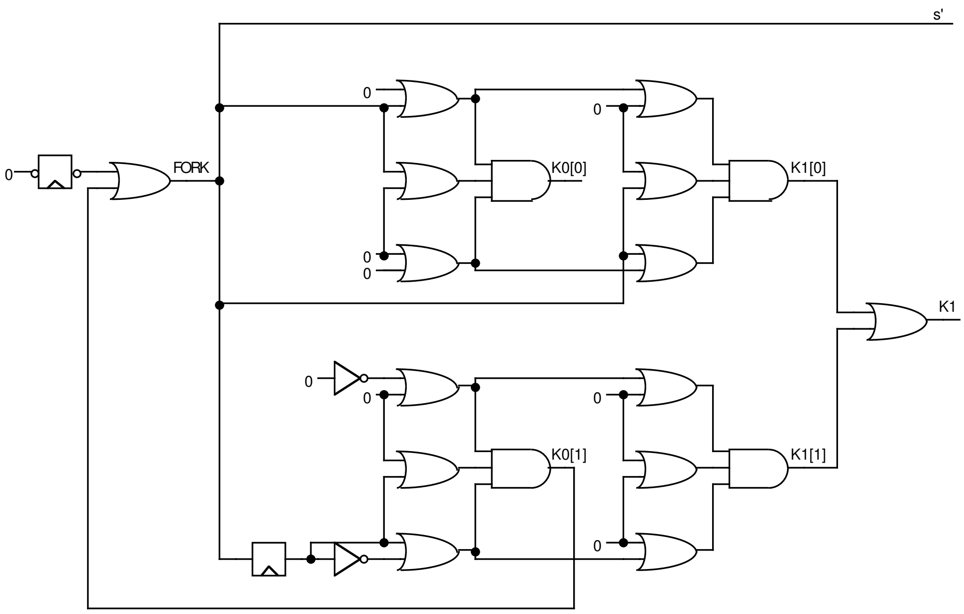


Figure 12.4: Correct circuit for $(!s\,;\,(0 \mid 1))*$

- The *surface* is the part that is driven by the `GO` input. The surface acts when the statement is started.
- The *depth* is the part of the circuit that is driven by the `RES` input and the `pause` registers at resumption time. The depth acts when the statement is selected and resumed.

In the basic translation, the surface and depth need not be disjoint. In "`sustain S`", Figure 12.1, the $!s$ statement is both in the surface and in the depth. In this case, everything works fine since it is harmless to emit a signal twice. In Figure 12.2, there is a single synchronizer for the surface and the depth. Since the surface and depth must simultaneously perform incompatible synchronization, the synchronizer is schizophrenic. In Figure 12.3, schizophrenia comes from the fact that there is a single signal buffer for the depth and surface signals, which should have distinct statuses at the same time.

The solution to avoid schizophrenia is to always keep the surface and depth entirely disjoint. This can be done without duplicating the registers,

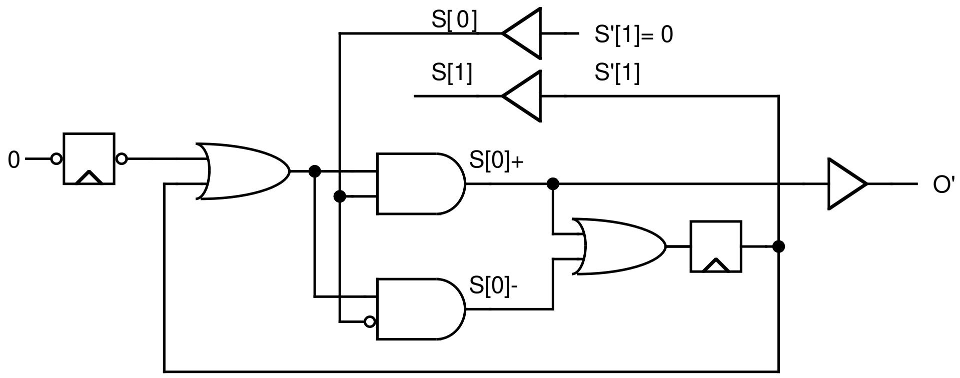


Figure 12.5: Correct circuit for $(((s\,?\,!o\,,0)\,;\,1\,;\,!s)\backslash s)*$

using a reasonable amount of logic duplication.

A correct translation of P16 is pictured in Figure 12.4. It uses separate synchronizers for the surface and the depth. The top surface synchronizer is called SYN[0], and the bottom depth synchronizer is called SYN[1]. The parallel statement is initially started using the surface synchronizer, which detects pausing and sets K1[0]. In the next instants, the selected pause register sets the termination input R0[1] of the depth synchronizer SYN[1], which reports immediate termination of the parallel statement by setting the output K0[1]. The parallel is immediately restarted and the inputs L0[0] and R1[0] of the surface synchronizer SYN[0] are set. The surface synchronizer reports pausing by setting K1[0].

At first glance, the new circuit looks much bigger than the old one. However, when constants are propagated, almost nothing is left of the circuit, and there is no practical penalty to the duplication. The LEM and REM inputs are also handled in a simpler way:

- In the surface synchronizer, we set LEM $=$ REM $= 0$ since we know that each branch will act and return a termination code when the parallel is started by setting GO.

- In the depth synchronizer, a branch will return a termination code if and only if it is selected. The negation of the branch selection wire is connected to the auxiliary input.

A correct translation of P17 is pictured in Figure 12.5. It uses two wires for S. The surface signal wire S[0] has input 0 since the surface incarnation of S has no emitter, and the depth signal wire S[1] is not connected to any gate since the depth incarnation is not tested for presence, as in the state semantics.

12.6 Multiple Reincarnation

So far, we have seen cases where a statement has two simultaneous incarnations. Nesting loops, traps, and parallel statements can lead to more complex situations with multiple reincarnation of parallels and signals. Consider program P18 in Figure 12.6. In the first instant, the three pause statements are selected, S1 and S2 are not emitted, and not_S1_and_not_S2 is emitted. In the second instant, the constructive behavior is as follows:

- The three pause statements terminate, S1 and S2 are emitted, and S1_and_S2 is emitted by the present statement.
- The trap T2 is exited by the "exit T2" statement, the inner loop loops, and a new signal S2 is declared. This new signal is absent since it cannot be emitted, while the current S1 is still present. The present statement is instantaneously re-executed with S1 present and S2 absent, and it emits S1_and_not_S2.
- The trap T1 is exited by the "exit T1" statement and the outer loop loops. A new S1 and a new S2 are declared. Since none of these new signals can be emitted, the present statement is executed again with S1 and S2 absent, and not_S1_and_not_S1 is emitted.

In this constructive behavior, S1 has two simultaneous incarnations, a present depth incarnation and an absent surface incarnation, while S2 has three incarnations, one present and two absent. In terms of depth and surface[1], the first one in constructive execution order can be called a *depth-depth* incarnation since it is both in the depth of "signal S1" and in the depth of "signal S2", the second one can be called a *depth-surface* incarnation since it is in the depth of "signal S1" and in the surface of "signal S2", and the third one can be called a *surface-surface* incarnation since it is in the surface of both signal declarations. Accordingly, the present statement has three different active incarnations with three different behaviors.

[1]Here, we ignore the parallel statements. See Section 12.7.4 for the accurate naming.

```
program P18 :
output S1_and_S2,
       S1_and_not_S2,
       not_S1_and_S2,
       not_S1_and_not_S2;
loop
   trap T1 in
      signal S1 in
         pause;
         emit S1;
         exit T1
      ||
         loop
            trap T2 in
               signal S2 in
                  pause;
                  emit S2;
                  exit T2
               ||
                  loop
                     present S1 then
                        present S2 then
                           emit S1_and_S2
                        else
                           emit S1_and_not_S2
                        end present
                     else
                        present S2 then
                           emit not_S1_and_S2
                        else
                           emit not_S1_and_not_S2
                        end
                     end present;
                     pause
                  end loop
               end signal
            end trap
         end loop
      end signal
   end trap
end loop
```

Figure 12.6: Multiple incarnations: program P18

The example can be extended to n signals, in which case the innermost signal has $n+1$ incarnations. Although such programs rarely occur in practice and can even look pathological, they are constructively correct and they must be correctly translated[2].

12.7 Curing Schizophrenia

To correctly translate examples such as `P18`, we must duplicate logic as many times as required by the number of possible incarnations. The solution we present here is based on the idea of always keeping the surface(s) and depth disjoint to be able to correctly translate loops by straightforward feedback. It has the drawback that logic is duplicated even if there are no loops, but it is semantically very clear. (A different solution based on duplicating logic only when translating loops is presented in [30].) In the worst case, the size of the logic will be square of the size of the source program. However, the squaring factor only shows up for artificial programs such as `P18`, and it is not a problem in practice. To minimize logic, the Esterel v5 compiler uses static analysis techniques to avoid duplication for non-schizophrenic parallels or local signals. We shall not present this optimization here.

12.7.1 Incarnation Indices

The key ingredient of our solution is the use of integer indices to name incarnations. In `P16`, page 130, and `P17`, page 132, we used indices 0 for the surface wires and 1 for the depth wires. We extend this numbering scheme to more complex examples such as `P18`, page 138. In the analysis of `P18`, we informally named the three incarnations of `S2` "depth–depth", "depth–surface", and "surface–surface", in the constructive order in which they appeared. We can abbreviate these names into words dd, ds, and ss. In this naming, we neglected the parallel statement that encloses "`signal S2`"; when taking this parallel into account, we get the exact naming ddd, dds, and sss. Notice that there can be no name such as sds where a d follows an s: by definition, the depth of a statement is included in the depth of all statements that enclose it. Therefore, we can further simplify the names and replace them by integer *incarnation indices* 2, 1, and 0, simply counting the number of occurrences of d in the name.

[2]The reader may question this point and wonder whether reincarnation is of any practical interest. The answer is yes, although we shall not try to prove this statement here. For an example, see [16] where reincarnation is nicely used to handle menus in a menubar.

In the final translation, we create separate logic for each incarnation index. The reason why this way of duplicating logic works is easy to explain: if a program P is constructive, then, for any substatement p, each indexed incarnation of p can be executed only once in a reaction. Intuitively, given a constructive proof of a reaction of P to an input, one can label each transition in the proof by the index at which the transition occurs. The only way to have two transitions of a substatement p at the same index would be to start a loop at this index and to loop the loop. This would imply that the body of the loop terminates instantaneously when started, which is forbidden by the semantics (see rules *(loop)* in the behavioral semantics and *(r-do-loop)* in the state semantics). Technically, this property is not trivial to prove, and the proof is deferred to Chapter 13. In `P18`, one notices that looping a loop always strictly decreases the incarnation level, as we shall explain in greater detail in Section 12.7.4.

12.7.2 Translation Sketch

Before giving the final translation in Chapter 13, let us explain it intuitively and revisit examples. Let P be a program and p be a substatement of P. We translate p as in the basic translation, except that we replace the original `GO`, `KILL`, `K`, `S`, and $\mathtt{S}'$ wires by arrays of wires indexed by incarnation indices. The `RES`, `SUSP`, and `SEL` wires need not be indexed since they concern only the depth of statements and are not reincarnated. For $\mathtt{GO}[i]$, $\mathtt{KILL}[i]$, and $\mathtt{Kk}[i]$ completion wires, the incarnation index i ranges from 0 to l, where the *level* l of p is the number of parallel statements and local signal declarations of P in which p is contained (the other statements cannot provoke schizophrenia and do not trigger logic duplication). For local signal input and output wires $\mathtt{S}[i]$ and $\mathtt{S}'[i]$, the index i ranges from 0 to l' if l' is the level of the body of the declaration of `S`, i.e. the level of the declaration plus one. Gates that receive wire arrays are replicated for each index. The wire driven by a `pause` register of level l is indexed by l since it is a typical depth wire.

Consider a parallel statement $p \mid q$ of level l. Then p and q are at level $l' = l+1$. A surface of the parallel statement is characterized by an index i such that $0 \leq i \leq l$, and it is made of a pair of surfaces of p and q synchronized by a specific synchronizer $\mathtt{SYN}[i]$. It is started by the wire $\mathtt{GO}[i]$, which is passed to p and q as in Figure 11.10. No surface is generated at index l'. The depth of the parallel statement is made of all the wires driven by the `pause` registers of p and p', which have index l', and of a depth synchronizer $\mathtt{SYN}[l']$. The surfaces and depth are kept disjoint as requested. Finally, we set $\mathtt{KILL}[l'] = \mathtt{KILL}[l]$

to propagate trap exits to the depth, and we union the completion wires of level l and l' to correctly propagate depth completion to the enclosing statement at level l.

For a local signal declaration of level l, the construction is similar. A surface has an index i such that $0 \leq i \leq l$ and uses a pair $\mathtt{S}[i]$, $\mathtt{S}'[i]$ of signal wires, while the depth has index $l' = l + 1$ and uses a distinct pair of wires $\mathtt{S}[l']$, $\mathtt{S}'[l']$.

12.7.3 Simple Examples

In P16, all wires in the top surface synchronizer have index 0, and all wires in the bottom depth synchronizer have index 1. Consider the following slightly more elaborate example P19.

```
module P19:
input I;
output O;
signal S in
   present I then emit O else pause end;
   [
      emit S
   ||
      pause; emit S
   ]
end signal
```

Let us first analyze control propagation. The "signal S" statement opens one level, and the parallel statement opens another level. The present statement is translated at indices 0 and 1. Here, the depth incarnation of index 1 is useless since there is no way to start the present statement at resumption time: since $\mathtt{GO}[1] = 0$, the incarnation of index 1 disappears by constant propagation[3]. The "emit O" statement in the then branch is translated with index 0 and its termination wire has the form K0[0]. The pause statement in the else branch generates two completion wires: the K1[0] wire, which reports pausing at start time, and the K0[1] wire, which reports termination at resumption time. The termination wire index is 1 since the pause

[3]In the Esterel v5 compiler, instead of generating and simplifying the depth incarnation, we do not generate it at all.

statement is at level 1. The parallel statement has three incarnations 0, 1, and 2, with two surface synchronizers `SYN[0]` and `SYN[1]` and one depth synchronizer `SYN[2]`.

- Incarnation 0 is of type surface-surface. It is active when both the `signal` and parallel statements are started. In our example, this happens at boot time if `I` is present. The `GO[0]` start wire of the parallel statement is the `K0[0]` termination wire of "`emit O`". The `GO[0]` wire is connected to the `L0[0]` entry of the `SYN[0]` synchronizer through the first "`emit S`" statement and to the `R1[0]` entry of `SYN[0]` through the `K1[0]` completion wire of the second `pause` statement, which indicates pausing. If `GO[0]` is set, `SYN[0]` reports pausing by setting its own completion wire `K1[0]`.

- Incarnation 1 is of type depth-surface. It is active when the "`signal S`" statement is resumed and the parallel statement is started. This happens when the first `pause` statement is selected and resumed, i.e. in the second instant if `I` was absent in the first instant. The `GO[1]` start wire is the termination wire `K0[1]` of the first `pause` statement. As before, the `GO[1]` wire is connected to the `L0[1]` and `R1[1]` entries of `SYN[1]`, which reports pausing by setting `K1[1]`.

- The `SYN[2]` synchronizer is of type depth-depth. It is active when the parallel statement is resumed, which can occur only when the `signal` statement is resumed. In our example, this happens in the second instant if `I` was present in the first instant, and in the third instant otherwise. There is no `GO` wire at index 2 since the depth is solely driven by the `RES` resumption wire. The termination wire of the second `pause` statement is `K0[2]` since this statement is at level 2. It is connected to the `R0[2]` entry of `SYN[2]`. The first branch is not connected to `SYN[2]` since its depth is empty.

The `Kk[1]` and `Kk[2]` completion wires of `SYN[1]` and `SYN[2]` are *or*-ed at level 1, since they are all in the depth of the enclosing `signal` statement. This way, the parallel statement reports pausing and termination as if it was not internally split.

Let us now analyze signals. The toplevel `I` and `O` signals have only one incarnation indexed by 0. The input wire `I[0]` is connected to the `present` statement *and* gates. The output wire `O'[0]` is equal to the `GO[0]` input of "`emit O`", i.e. to the output of the first *and* gate of the `present` statement.

The `S` local signal has two incarnations 0 and 1. The surface "`emit S`" in the first parallel branch is translated at indices 0 and 1. Its `GO[0]` and `GO[1]` start wires respectively reach the *or*-gates of `S'[0]` and `S'[1]`. The depth "`emit S`" in the second branch acts only at index 2 since it follows a `pause` statement of level 2. Its `GO[2]` wire is connected to the `S'[1]` *or*-gate: the index 2 is transformed into $1 = min(1, 2)$ since the level of `S` is 1.

12.7.4 Correct Translation of `P18`

Consider now the program `P18`, page 138. The maximum level is 4 since new levels are opened by the nested "`signal S1`", "`||`", "`signal S2`", and "`||`" statements. The first `pause` statement has level 2 since it is enclosed in the first signal declaration and the first parallel statement, while the second and third ones have level 4. The `S1` signal has two incarnations, the first parallel statement has three, the `S2` signal has four, and the second parallel statement has five.

The "`signal S1`" declaration generates two input-output wire pairs: `S1[0]` and `S1'[0]` for surface incarnation, and `S1[1]` and `S1'[1]` for depth incarnation. At boot time, no "`emit S1`" statement is executed, which implies that `S1'[0]` and `S1[0]` have constant value 0. Being driven by the first `pause` statement of level 2, the "`emit S1`" statement connects its `GO[2]` input to `S'[1]`, which is itself connected to `S[1]` by a buffer. Notice that the index 2 of the `GO` wire is decreased to `1`, which is the maximal index for `S1`.

The first parallel statement has three synchronizers, only `SYN[0]` and `SYN[2]` being of interest. The surface-surface `SYN[0]` synchronizer is used when entering the parallel statement at boot time or when looping the outermost loop. It then reports pausing. The depth-depth `SYN[2]` synchronizer is used in all instants except boot to synchronize the termination of the branch resumptions. It then reports exiting trap `T1` by setting `K2[2]`. Since there is no possibility to start the parallel from a `pause` statement of level 1, all inputs to the depth-surface `SYN[1]` synchronizer are 0, and `SYN[1]` disappears by constant propagation.

The "`signal S2`" declaration generates four pairs of wires corresponding to indices 0 to 3. No "`emit S2`" statement can be executed at indices 0, 1, and 2, which implies that `S2'[0]`, `S2[0]`, `S2'[1]`, `S2[1]`, `S2'[2]`, and `S2[2]` have constant value 0. The "`emit S2`" statement is started by a `pause` statement of level 4. It connects its `GO[4]` input to `S2'[3]`, and one has `S2[3]` = `S2'[3]` = `GO[4]`.

The innermost parallel statement has five synchronizers ranging from `SYN[0]` to SYN[4], two of them being useless. The surface-surface SYN[0] synchronizer is used when entering the parallel statement at boot time or when looping the outermost loop. It then reports pausing by setting K1[0]. The SYN[2] synchronizer is used when the second loop loops. It then reports pausing by setting K1[2]. The SYN[4] synchronizer is used for proper resumption of the parallel depth. It then reports exiting T2 by setting K2[4]. The SYN[1] and SYN[3] synchronizers are useless and disappear by constant propagation.

The present statement also has also five incarnations, the ones indexed by 1 and 3 being useless since GO[1] = GO[3] = 0.

At boot time, the SYN[0] synchronizers all report pausing by setting their K1[0] outputs. No local signal is emitted, and the surface 0 incarnation of the present statement executes "emit not_S1_and_not_S2", which sets not_S1_and_not_S2′[0]. In the next instants, execution is as follows.

- The three pause registers set their respective termination outputs K0[2], K0[4], and K0[4], which provokes setting S1′[1] and S2′[3] and executing the present statement at index 4. The present statement tests for the depth S1[1] and S2[3] incarnations, which are set. Therefore, "emit S1_and_S2" is executed and S1_and_S2′[0] is set. The first branch of the innermost parallel executes "exit T2" and sets the L2[4] input of SYN[4], while the second branch executes pause and sets the R1[4] input of SYN[4]. Therefore, SYN[4] propagates the exit by setting its K2[4] output wire. This wire reaches the "trap T2" statement output *or*-gate, which sets the termination wire K0[2] of the trap statement, which has depth 2.

- The surrounding loop sets GO[2] that traverses the "signal S2" declaration and starts the incarnation 2 of the innermost parallel. The first branch reports pausing to SYN[2] by setting L1[2]. The second branch executes the present test at index 2. This time, the signals tested for are S1[1] = 1 and S2[2] = 0, and S1_and_not_S2′[0] is set. The second branch reports pausing to SYN[2] by setting the R1[2] input, and SYN[2] sets its K1[2] output. Pausing is reported to the R1[2] input of the depth SYN[2] synchronizer of the first parallel.

- In parallel, the first branch of the outermost parallel asks for exiting T1 by setting the L2[2] input of the SYN[2] depth synchronizer. Since L2[2] and R1[2] are set, the SYN[2] synchronizer sets its completion wire K2[2], which sets the termination wire K0[0] of the "trap T1" statement.

- The outermost loop sets `GO[0]` and starts the surface incarnations. All `pause` statements report pausing to `SYN[0]` synchronizers. The incarnation 0 of the `present` statement is executed. It tests for `S1[0]` and `S2[0]`, which are both unset. The "`emit not_S1_and_not_S2`" circuit is executed and sets `not_S1_and_not_S2[0]`.

This all happens in one instant.

Chapter 13

The Formal Translation to Hardware

We now formally present the final correct translation of Esterel programs into circuits, using the textual presentation of circuits as sets of definitions and making heavy use of the syntactic extensions described in Section 10.4.

13.1 Translation Environments

For simplicity, we assume that all local signals have distinct names in the Esterel source program. This can be achieved by a suitable renaming.

The translation function takes as argument an *environment* ρ composed of several constants and wires. The environment is used to determine the input and output wires of the current statement and some numerical auxiliaries. It contains the following fields:

- An integer κ that denotes the *maximal completion code* available when translating the current statement, i.e. the number of traps in which the statement is included plus one.
- An integer l that denotes the level of the current statement, i.e. the number of local signal and parallel statements in which the statement is enclosed.
- A vector `GO` of *start wires* $\mathtt{GO}[i]$, indexed by incarnation indices i such that $0 \leq i \leq l$.
- A *resumption* wire `RES`.

- A *suspension* wire SUS.
- A vector KILL of *kill wires* $\mathtt{KILL}[i]$, indexed by incarnation indices i such that $0 \leq i \leq l$.
- A *selection* wire SEL.
- A matrix K of *completion* wires $\mathtt{K}[k, i]$, indexed by completion codes k such that $0 \leq k \leq \kappa$ and by incarnation indices i such that $0 \leq i \leq l$. In the formal translation, we write $\mathtt{K}[3, 2]$ instead of the K3[2] of Chapter 12.
- A set E of *input signal vectors* $\mathtt{S}[i]$ and a set $\mathtt{E}'$ of *output signal vectors* $\mathtt{S}'[i]$, each set containing one vector for each signal s visible in the current scope. For a vector $\mathtt{S}[i]$ or $\mathtt{S}'[i]$, the index i is an incarnation index such that $0 \leq i \leq l(s)$, where the *level* $l(s)$ of a signal s declared by $p \backslash s$ is the level of p, i.e. the level of $p \backslash s$ plus one.

We use the classical record field notation to retrieve environment components. For instance, $\rho.\mathtt{GO}[i]$ denotes an incoming control wire. Unless confusion is possible, we simply abbreviate $\rho.\mathtt{GO}[i]$ by $\mathtt{GO}[i]$.

Given an environment ρ, we shall often need to consider another environment ρ' that differs from ρ by the value of one field, say by changing $\rho.\mathtt{GO}[i]$ into some wire w. We then write $\rho' = \rho[w/\mathtt{GO}[i]]$. The notation extends naturally when changing several fields.

13.1.1 The Global Environment

To translate a program, we first allocate the following global wires:

- A global selection wire GSEL to be defined by the circuit.
- A global resumption wire GRES defined by $\mathtt{GRES} = 1$.
- A global suspension wire GSUS defined by $\mathtt{GSUS} = 0$.
- A global kill wire GKILL defined by $\mathtt{GKILL} = 0$.
- A global start GGO wire defined by $\mathtt{GGO} = \neg\mathtt{BOOTREG}$, where the auxiliary boot register BOOTREG is defined by $\mathtt{BOOTREG} := 1$.
- A matrix GK made of two global continuation wires $\mathtt{GK}[0, 0]$ and $\mathtt{GK}[1, 0]$, which respectively correspond to global termination and pause.

- A set $\mathcal{I}$ of input signal wires, and a set $\mathcal{O}$ of output signal wires. For each input signal i, there is an input wire $\mathtt{I}$, and for each signal o there is an output wire $\mathtt{O}$. Being an input, $\mathtt{I}$ has no definition. Since the body of a program is at level 0, we use $\mathtt{I}[0]$ and $\mathtt{O}[0]$ as synonyms for $\mathtt{I}$ and $\mathtt{O}$.

We translate the module body in the global environment

$$g\rho = (1, 0, \mathtt{GGO}, \mathtt{GRES}, \mathtt{GSUS}, \mathtt{GKILL}, \mathtt{GSEL}, \mathtt{GK}, \mathcal{I}, \mathcal{O})$$

13.2 Translation Rules

For a completion code different from 1, we replicate the design of Figure 11.2 at all incarnation indices.

$$\text{for } k \neq 1, \quad \mathcal{C}(k, \rho) \quad = \quad \mathtt{K}[k, i] \underset{0 \leq i \leq l}{\Longleftarrow} \mathtt{GO}[i]$$

In the translation of a `pause` statement, we replicate the surface part of Figure 11.3, and the register drives the $\mathtt{K}[0, l]$ depth termination output.

$$\mathcal{C}(1, \rho) \quad = \quad \left\{ \begin{array}{l} \text{local } \mathtt{REG} \\ \mathtt{REG} := \quad (\mathtt{SUS} \wedge \mathtt{REG} \wedge \neg\mathtt{KILL}[l]) \\ \qquad\quad \vee \bigvee_{0 \leq i \leq l} (\mathtt{GO}[i] \wedge \neg\mathtt{KILL}[i]) \\ \mathtt{SEL} \Leftarrow \mathtt{REG} \\ \mathtt{K}[1, i] \underset{0 \leq i \leq l}{\Longleftarrow} \mathtt{GO}[i] \\ \mathtt{K}[0, l] \Leftarrow \mathtt{REG} \wedge \mathtt{RES} \end{array} \right.$$

For a signal emission, we replicate Figure 11.4 at all indices. If $i > l(s)$, we are in the depth of the definition of s, and the wire to be set is $\mathtt{S}'[l(s)]$. We use the notation $m \downarrow n$ as a short-hand for $min(m, n)$.

$$\mathcal{C}(!s, \rho) \quad = \quad \left\{ \begin{array}{l} \mathtt{K}[0, i] \underset{0 \leq i \leq l}{\Longleftarrow} \mathtt{GO}[i] \\ \mathtt{S}'[i \downarrow l(s)] \underset{0 \leq i \leq l}{\Longleftarrow} \mathtt{GO}[i] \end{array} \right.$$

For a signal presence test, we replicate Figure 11.5 at all indices. If $i > l(s)$, we are in the depth of the definition of s, and the wire to be tested for is $\mathtt{S}[l(s)]$. All the *or*-gates that gather the outputs of the subcircuits in Figure 11.5 are implicitly created, since the subcircuits are instantiated in the same output environment and since the output wires are locally defined by implications in the subcircuits.

$$\mathcal{C}((s\,?\,p\,,q),\rho) \quad = \quad \begin{cases} \text{local } \mathtt{GO}_1[0..l],\ \mathtt{GO}_2[0..l] \\ \mathcal{C}(p,\rho\,[\mathtt{GO}_1\,/\,\mathtt{GO}]) \\ \mathcal{C}(q,\rho\,[\mathtt{GO}_2\,/\,\mathtt{GO}]) \\ \mathtt{GO}_1[i] \underset{0\leq i\leq l}{=} \mathtt{GO}[i] \wedge \mathtt{S}[i \downarrow l(s)] \\ \mathtt{GO}_2[i] \underset{0\leq i\leq l}{=} \mathtt{GO}[i] \wedge \neg\mathtt{S}[i \downarrow l(s)] \end{cases}$$

The translations of a suspension or of a watchdog are trivial since only the depth is of interest.

$$\mathcal{C}(s \supset p,\rho) \quad = \quad \begin{cases} \text{local } \mathtt{NRES}, \mathtt{NSUS} \\ \mathcal{C}(p,\rho\,[\mathtt{NRES}\,/\,\mathtt{RES},\ \mathtt{NSUS}\,/\,\mathtt{SUS}]) \\ \mathtt{NRES} = \mathtt{RES} \wedge \neg\mathtt{S}[l(s)] \\ \mathtt{NSUS} = \mathtt{SUS} \vee (\mathtt{RES} \wedge \mathtt{S}[l(s)]) \\ \mathtt{K}[1,l] \Leftarrow \mathtt{RES} \wedge \mathtt{S}[l(s)] \end{cases}$$

$$\mathcal{C}(s >\!> p,\rho) \quad = \quad \begin{cases} \text{local } \mathtt{NRES} \\ \mathcal{C}(p,\rho\,[\mathtt{NRES}\,/\,\mathtt{RES}]) \\ \mathtt{NRES} = \mathtt{RES} \wedge \neg\mathtt{S}[l(s)] \\ \mathtt{K}[0,l] \Leftarrow \mathtt{RES} \wedge \mathtt{S}[l(s)] \end{cases}$$

In the translation of a sequence $p\,;\,q$, we chain p and q at all incarnation indices, as in Figure 11.8. The output *or*-gates are implicitly generated by implications, as for $s\,?\,p\,,q$.

$$\mathcal{C}(p\,;\,q,\rho) \quad = \quad \begin{cases} \text{local } \mathtt{SEQ}[0..l] \\ \mathcal{C}(p,\rho\,[\mathtt{SEQ}[i]\,/\,\mathtt{K}[0,i]]) \\ \qquad\qquad\qquad\quad {\scriptstyle 0\leq i\leq l} \\ \mathcal{C}(q,\rho\,[\mathtt{SEQ}\,/\,\mathtt{GO}]) \end{cases}$$

For a loop, we feedback the depth termination wire to the depth $\mathtt{GO}[l]$ start wire.

$$\mathcal{C}(p*, \rho) \quad = \quad \begin{cases} \text{local } \mathtt{INLOOP}, \mathtt{OUTLOOP} \\ \mathcal{C}(p, \rho\,[\mathtt{INLOOP} \,/\, \mathtt{GO}[l],\ \mathtt{OUTLOOP} \,/\, \mathtt{K}[0,l]]) \\ \mathtt{INLOOP} = \mathtt{GO}[l] \vee \mathtt{OUTLOOP} \end{cases}$$

Notice that the wires $\mathtt{K}[0,i]$ are left pending for $i \neq l$. This works only for loop-safe programs defined in Section 6.6, for which these wires are guaranteed to always have value 0. A translation that also handles loop-unsafe programs will be presented in Section 13.4.

We now translate the parallel statement. We increment the level and declare the selection, left and right completion, and synchronizer wires. We translate the left and right substatements in the appropriate environments, and we build the selection wire and the synchronizers, as in Figures 11.10 and 11.11. The output *or*-gates are created implicitly by implications. To translate the substatements, we use an auxiliary environment ρ' such that $l' = l{+}1$, $\rho.\mathtt{GO}[l'] = 0$, since there should be no surface at depth level l', and $\rho'.\mathtt{KILL}[l'] = \rho.\mathtt{KILL}[l]$ since the input depth kill wire must be propagated to the new depth to handle exceptions. Given an array A of size n, call $A\cdot v$ the array of size $n+1$ obtained by adding v to A at position n. Then $\rho' = \rho\,[l' \,/\, l,\ \mathtt{GO}\cdot 0 \,/\, \mathtt{GO},\ \mathtt{KILL}\cdot\mathtt{KILL}[l] \,/\, \mathtt{KILL}]$. The formal translation is

$$
\mathcal{C}(p \mid q, \rho) \quad = \quad \left\{
\begin{array}{l}
\text{local} \quad l' = l + 1 \\
\qquad\quad \mathtt{LSEL},\, \mathtt{RSEL}, \\
\qquad\quad \mathtt{L}[0..\kappa, 0..l'],\, \mathtt{R}[0..\kappa, 0..l'], \\
\qquad\quad \mathtt{LMIN}[0..\kappa, 0..l'],\, \mathtt{RMIN}[0..\kappa, 0..l'],\, \mathtt{UNION}[0..\kappa, 0..l'] \\
\mathcal{C}(p, \rho' \,[\mathtt{LSEL} \,/\, \mathtt{SEL},\, \mathtt{L} \,/\, \mathtt{K}]) \\
\mathcal{C}(q, \rho' \,[\mathtt{RSEL} \,/\, \mathtt{SEL},\, \mathtt{R} \,/\, \mathtt{K}]) \\
\mathtt{SEL} \Leftarrow \mathtt{LSEL} \\
\mathtt{SEL} \Leftarrow \mathtt{RSEL} \\
\mathtt{UNION}[k, i] \quad \underset{\substack{0 \le k \le \kappa \\ 0 \le i \le l'}}{=} \quad \mathtt{L}[k, i] \vee \mathtt{R}[k, i] \\
\mathtt{LMIN}[0, l'] \Leftarrow \neg\mathtt{LSEL} \\
\mathtt{LMIN}[0, i] \underset{0 \le i \le l'}{\Longleftarrow} \mathtt{L}[0, i] \\
\mathtt{LMIN}[k+1, i] \quad \underset{\substack{0 \le k < \kappa \\ 0 \le i \le l'}}{=} \quad \mathtt{LMIN}[k, i] \vee \mathtt{L}[k+1, i] \\
\mathtt{RMIN}[0, l'] \Leftarrow \neg\mathtt{RSEL} \\
\mathtt{RMIN}[0, i] \underset{0 \le i \le l'}{\Longleftarrow} \mathtt{R}[0, i] \\
\mathtt{RMIN}[k+1, i] \quad \underset{\substack{0 \le k < \kappa \\ 0 \le i \le l'}}{=} \quad \mathtt{RMIN}[k, i] \vee \mathtt{R}[k+1, i] \\
\mathtt{K}[l \downarrow i, k] \quad \underset{\substack{1 \le i \le \kappa \\ 0 \le i \le l'}}{\Longleftarrow} \quad \mathtt{LMIN}[k, i] \wedge \mathtt{UNION}[k, i] \wedge \mathtt{RMIN}[k, i]
\end{array}
\right.
$$

Notice that the last implication implicitly performs the union of $\mathtt{K}[k, l]$ and $\mathtt{K}[k, l+1]$, which is necessary to report depth completion at the level at which the parallel is translated. Such a completion is either the completion of the surface of index l or a completion of the parallel depth proper.

Trap and shift are translated as in Figures 11.12 and 11.13 at all indices.

$$\mathcal{C}(\{p\},\rho) \quad = \quad \begin{cases} \text{local} \quad \kappa' = \kappa+1 \\ \qquad\quad \mathtt{NKILL}[0..l] \\ \qquad\quad \mathtt{NK}[0..\kappa', 0..l] \\ \mathcal{C}(p, \rho\,[\kappa' \,/\, \kappa,\ \mathtt{NKILL} \,/\, \mathtt{KILL},\ \mathtt{NK} \,/\, \mathtt{K}]) \\ \mathtt{NKILL}[i] \underset{0 \le i \le l}{=} \mathtt{KILL}[i] \vee \mathtt{NK}[2,i] \\ \mathtt{K}[0,i] \underset{0 \le i \le l}{\Longleftarrow} \mathtt{NK}[0,i] \vee \mathtt{NK}[2,i] \\ \mathtt{K}[1,i] \underset{0 \le i \le l}{\Longleftarrow} \mathtt{NK}[1,i] \\ \mathtt{K}[k,i] \quad \underset{\substack{2 \le k \le \kappa \\ 0 \le i \le l}}{\Longleftarrow} \quad \mathtt{NK}[k+1,i] \end{cases}$$

$$\mathcal{C}(\uparrow p,\rho) \quad = \quad \begin{cases} \text{local} \quad \kappa' = \kappa-1 \\ \qquad\quad \mathtt{NK}[0..\kappa', 0..l] \\ \mathcal{C}(p, \rho\,[\kappa' \,/\, \kappa,\ \mathtt{NK} \,/\, \mathtt{K}]) \\ \mathtt{K}[\uparrow k,i] \quad \underset{\substack{0 \le k < \kappa \\ 0 \le i \le l}}{\Longleftarrow} \quad \mathtt{NK}[k,i] \end{cases}$$

To translate a $p\backslash s$ statement, we increment l and feed back the signal wires at all incarnation indices. As for the parallel statement, we use the environment $\rho' = \rho\,[l' \,/\, l,\ \mathtt{GO}{\cdot}0 \,/\, \mathtt{GO},\ \mathtt{KILL}{\cdot}\mathtt{KILL}[l] \,/\, \mathtt{KILL}]$. The translation is:

$$\mathcal{C}(p\backslash s,\rho) \quad = \quad \begin{cases} \text{local} \quad l' = l+1 \\ \qquad\quad \mathtt{S}[0..l'],\ \mathtt{S}'[0..l'] \\ \mathcal{C}(p, \rho\,[\mathtt{E}{\cdot}\mathtt{S} \,/\, \mathtt{E},\ \mathtt{E}'{\cdot}\mathtt{S}' \,/\, \mathtt{E}']) \\ \mathtt{S}[i] \underset{0 \le i \le l'}{=} \mathtt{S}'[i] \end{cases}$$

13.3 Correctness of the Translation

Let P be a program of body q, let $p = 1\,;\, q$ be the base statement of the states of P, and let $\mathcal{C} = \mathcal{C}(P)$ be the generated circuit. Registers in $\mathcal{C}$ are in bijection with `pause` statements in p. With each state $\hat{p}$ of P we associate a state $R(\hat{p})$ of $\mathcal{C}$. In $R(\hat{p})$, a register has value 1 if and only if the corresponding `pause` statement is selected in $\hat{p}$, except for the boot register, which is inverted and has value 0 if the auxiliary boot statement is selected; this inversion is needed since all registers are assumed to have initial value 0 while the boot statement is originally selected. Note that the terminated extended statement p corresponds to the circuit state where the boot register has value 1 and all other `pause` registers have value 0, which is characterized by $\texttt{GSEL} = 0$ in $\mathcal{C}$.

The main equivalence theorem shows that our circuit translation exactly implements the constructive behavioral semantics for loop-safe programs.

Theorem 6 *For any state $\hat{p}$ of a loop-safe program P and for any input event I, one has $\hat{p} \xhookrightarrow[I]{O} \hat{p}'$ in the state constructive semantics if and only if one has $R(\hat{p}) \xhookrightarrow[I]{O} R(\hat{p}')$ in the constructive circuit semantics.*

13.4 Extension to Loop-Unsafe Programs

We briefly indicate how to extend the translation to all programs, including loop-unsafe ones.

Consider a loop $q*$ of level l in a program P. For $i < l$, the termination wire $\texttt{K}[0,i]$ of q is generated by the surface of q, which cannot terminate instantaneously if P is loop-safe. In this case, $\texttt{K}[0,i]$ always has value 0 and can be ignored. For loop-unsafe programs, we must dynamically check for $\texttt{K}[0,i] = 0$. One way to do this is to introduce an artificial combinational loop using an auxiliary wire $\texttt{INSTLOOP}[i]$ defined by

$$\texttt{INSTLOOP}[i] \quad = \quad \texttt{INSTLOOP}[i] \wedge \texttt{K}[0,i] \tag{13.1}$$

Then, the value of $\texttt{INSTLOOP}[i]$ can be constructively computed to be 0 if and only if $\texttt{K}[0,i] = 0$. If $\texttt{K}[0,i] = 1$, we get $\texttt{INSTLOOP}[i] = \texttt{INSTLOOP}[i]$ and the circuit is not constructive, which is detected by the Esterel compiler. This is fine for $i < l$ since $\texttt{K}[0,i]$ is a surface wire, and the extension of the translation is trivial. However, at level l, $\texttt{K}[0,l]$ handles both the surface of index l and the depth of q. To use the same trick, we have to distinguish between surface and depth completion wires and to use only the surface completion wire in the definition of $\texttt{INSTLOOP}[l]$. This requires an easy but tedious modification of the translation, which we leave to the reader.

Chapter 14

Conclusion

We have presented the new constructive semantics of Pure Esterel and we have shown the equivalence between three styles of semantics: the constructive behavioral semantics, where reactions are defined by single synchronous transitions, the constructive operational semantics, where reactions are defined by sequences of elementary microsteps, and the electrical semantics that uses a translation into constructive digital circuits.

On the theoretical side, the main result is the equivalence of constructiveness of a program and electrical stabilization of its circuit. This result is important since it relates a logical notion to a truly physical one. It gives us additional confidence in the relevance of the constructive semantics. On the practical side, the constructive semantics is implemented in the Esterel v5 compiler to software or hardware, and it is also used in other tools such as symbolic debuggers, see the Esterel v5 manual [12].

There are many more things to be studied. In this book, we have not studied compositionality issues. It turns out that the logical semantics has a weird behavior w.r.t. program composition, while the constructive semantics is compositional by nature, being a least fixpoint semantics (nevertheless, compositionality issues are never trivial in synchronous languages). We have also not discuss optimization issues. The interested reader can refer to [36, 37]. We are also lacking an axiomatic semantics, which should be useful for studying program equivalence. Finally, we are looking for very efficient algorithms to show constructiveness for all possible inputs. The ones we presently use are not that bad, but they can surely be dramatically improved.

Bibliography

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

[2] 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.

[3] G. Berry. Programming a digital watch in Esterel v3_2. Rapport de recherche 08/91, Centre de Mathématiques Appliquées, Ecole des Mines de Paris, Sophia-Antipolis, 1991.

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

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

[6] G. Berry. The semantics of pure Esterel. In M. Broy, editor, *Program Design Calculi*, volume 118 of *Series F: Computer and System Sciences*, pages 361–409. NATO ASI Series, 1993.

[7] G. Berry. *The Esterel Primer.* http://wwww.inria.fr/meije/esterel, 1998.

[8] 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, to appear.

[9] 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.

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

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

[12] G. Berry and the Esterel Team. *The Esterel v5 System Manual.* http://wwww.inria.fr/meije/esterel, 1998.

[13] F. Boussinot. Programming a reflex game in Esterel v3_2. Rapport de recherche 07/91, Centre de Mathématiques Appliquées, Ecole des Mines de Paris, Sophia-Antipolis, 1991.

[14] F. Boussinot and R. de Simone. The Esterel language. *Another Look at Real Time Programming, Proceedings of the IEEE*, 79:1293–1304, 1991.

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

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

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

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

[19] The Esterel Group. The Esterel v5 documentation. Technical report, Ecole des Mines de Paris / INRIA.

[20] 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.

[21] G. D. Hachtel and F. Somenzi. *Logic Synthesis and Verification Algorithms.* Kluwer Academic Publishers, 1996.

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

[23] 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.

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

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

[26] D. Harel, A. Pnueli, J.P. Schmidt, and R. Sherman. On the formal semantics of Statecharts. In *Proc. Logic in Computer Science*, 1986.

[27] L. Jategaonkar Jagadeesan, C. Puchol, and J. E. Von Olnhausen. A formal approach to reactive systems software: a telecommunications application in Esterel. *Formal Methods in Systems Design*, 8(2), 1996.

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

[29] Florence Maraninchi. The Argos language: Graphical representation of automata and description of reactive systems. In *Proc. IEEE Conf. on Visual Languages, Kobe, Japan*, 1991.

[30] F. Mignard. Compilation du langage Esterel en systèmes d'équations booléennes. Thèse d'informatique, Ecole des Mines de Paris, 1994.

[31] R. Milner. *Communication and Concurrency.* Series in Computer Science. Prentice Hall, 1989.

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

[33] G. Plotkin. Lcf considered as a programming language. *Theoretical Computer Science*, 5(3):223–256, 1977.

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

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

[36] 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.

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

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

[39] 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.

[40] The COQ Team. The COQ system. Technical report, INRIA, http://pauillac.inria.fr/coq.

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

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