

Towards a Coq-verified Chain of Esterel Semantics

Lionel Rieg 

Université Grenoble Alpes, CNRS, Grenoble INP – UGA, VERIMAG, Grenoble, France
Collège de France, Paris, France

Gérard Berry 

Collège de France, Paris, France

Abstract

This article focuses on formally specifying and verifying the chain of formal semantics of the Esterel synchronous programming language using the Coq proof assistant. In particular, in addition to the standard logical (LBS) semantics, constructive semantics (CBS) and constructive state semantics (CSS), we introduce a novel microstep semantics that gets rid of the Must/Can potential function

pair of the constructive semantics and can be viewed as an abstract version of Esterel’s circuit semantics used by compilers to generate software code and hardware designs. The article also comes with formal proofs in Coq of the equivalence between the CBS and CSS semantics and of the refinement of the CSS by the microstep semantics, except for the loop construct of Esterel.

2012 ACM Subject Classification Theory of computation → Operational semantics; Hardware → Theorem proving and SAT solving; Security and privacy → Logic and verification; Computer systems organization → Real-time languages

Keywords and phrases Esterel programming language, formal verification, Coq proof assistant

Digital Object Identifier 10.4230/LITES.10.1.2

Related Version *Preprint version:* <https://arxiv.org/abs/1909.12582>

Supplementary Material *Software:* <https://zenodo.org/records/15199439>

Received 2023-07-18 **Accepted** 2025-03-11 **Published** 2025-04-16

1 Introduction

The long-term goal of the research presented here is twofold: first, formally validate the chain of semantics of the synchronous reactive language Esterel [3] that leads from its definition by formal semantics to its implementation; second, build a formally proven compiler from Esterel to clocked digital circuits or C code, in the spirit of the CompCert verified C compiler [35]. The tool we use for this is the chain of Plotkin-style Structural Operational Semantics (SOS) developed for Esterel between 1984 and 2000 and published in the web book [4], which was not submitted to an editor because it was lacking formal proofs. The subject being quite tricky, the second author thought that the proofs should be done on computer – which is what we contribute to here.

We also introduce here a new operational semantics that gets closer to the circuit semantics of [4], on which the Esterel v5 compiler and later the industrial v7 compiler were based. We use the Coq proof assistant [21] as the formal verification environment.

This work is still partial: we limit ourselves to Kernel Esterel [3], the core language that only deals with pure signaling but still gathers most of the technical difficulties. The Coq proofs accompanying this article also exclude loops. The full Esterel language has loops and involves data-valued signals and variables. We already know how to efficiently handle loops with a method presented in [4], used in the Esterel v5 and v7 compilers, or alternatively with Oliver Tardieu’s method published in [53]. Data handling should not add extra conceptual difficulties since the



© Lionel Rieg and Gérard Berry;
licensed under Creative Commons License CC-BY 4.0

Leibniz Transactions on Embedded Systems, Vol. 10, Issue 1, Article No. 2, pp. 2:1–2:54

Leibniz Transactions on Embedded Systems

LITES Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

only addition in the semantics and compilers is the insertion of additional dependencies in the control flow, which can be technically handled almost in the same way as Kernel Esterel's simpler pure signal dependencies. Nevertheless, it may still add technical difficulties to the Coq proofs.

1.1 Synchrony, determinism, concurrency: a short history of Esterel

Esterel, born in 1983 [9], is the oldest member of the family of synchronous and deterministic concurrent programming languages. It was initially dedicated to programming discrete control systems found in industry such as airplanes, automotive, process control in factories, etc., for which determinism is a key feature. While being concurrent, Esterel strongly differs from classical asynchronous and non-deterministic concurrent languages because it is synchronous and fully deterministic. It was introduced just before Lustre [31] and Signal [30], two other synchronous deterministic languages also dedicated to industrial process control but with a different style: they are data-flow oriented functional languages dedicated to controlling processes with fairly constant control laws, while Esterel was dedicated to control-intensive processes whose behavior keeps changing over time. All these synchronous languages are based on well-defined and formal semantics.

The Esterel language and formal semantics were developed together in a series of steps, from a first semantics used in 1984 for the Esterel v2 compiler [22] to the final constructive semantics in 2000 that lead to the academic Esterel v5 compiler [2]. This compiler was used in academic or industrial research for safety-critical embedded systems in a wider variety of domains compared to the initial goal: avionics [5], robotics [26], communication protocols [41], and the synthesis of efficient synchronous digital circuits [54, 8, 49]. Another very efficient but more limited compiler was developed for Esterel by Stephen Edwards at Columbia University [25]. All this work is described in detail in the *Compiling Esterel* book [44].

From 2001 to 2009, an enriched v7 version of Esterel was developed by the second author and his team at the Esterel Technologies company, with the same kernel semantics but many language extensions and a compiler able to generate both C programs for software applications and Verilog/VHDL circuits for hardware applications. Esterel v7 was used in R&D and production by companies such as Intel, Texas Instruments, ST microelectronics, NXP, etc. But the 2008 crisis unfortunately canceled its development and usage.

The Esterel v5 academic compiler and debugging environment as well as the Edwards Columbia compiler are still freely available.¹ Furthermore, the Esterel v5 semantics and compiling technology has been transported to HipHop [10], a reactive extension of Serrano's Hop language [50] which is itself an extension of JavaScript that specifies client/server computations in a single source program and simplifies the design of web pages. HipHop is dedicated to program complex behaviors of autonomous devices and human/machine Web-based interfaces.

In parallel, Lustre was industrialized by the French company Verilog under a graphical form called SCADE,² used for avionics by Airbus, for the Lyon automatic subway control, for the French nuclear plants safety systems, etc. In 2006, SCADE and Esterel (with some restrictions) were unified by Esterel Technologies into the SCADE 6 language and toolset [20] with ideas borrowed from Lucid Synchrone [17]; the SCADE 6 compiler was certified according to the DO-178C standard, the highest certification standard for avionics. Esterel Technologies was bought in 2012 by Ansys, and SCADE 6 is now widely used by several hundred industrial customers for certified embedded systems and other safety-critical applications.

¹ Respectively from <https://esterel.org/files/Html/Downloads/Downloads.htm> and <http://www1.cs.columbia.edu/~sedwards/cec/>.

² For Safety Critical Applications Development Environment

1.2 Signals in Esterel

Let us first focus on how and why concurrency and determinism are reconciled by Esterel. The shared objects between concurrent statements are called *signals*. A signal carries a presence/absence *status* and optionally a unique *data value*. The execution of a program consists of a series of discrete *reactions*, where each reaction handles an input signal vector from the environment and generates an output signal vector to the environment in a deterministic way and conceptually in zero time. The program can also use an arbitrary number of local signals declared by a specific “`signal S in ... end`” scoping statement.

In this article, as in [4], we only deal with *pure signals* that carry a Boolean *present / absent status* but no data. In each reaction, a pure signal *S* is *absent* by default; it is set *present* only if the environment says so for an input signal or if it is explicitly emitted within the program by an “`emit S`” statement that broadcasts this status in the signal scope. Valued signals would introduce no major difficulties in the semantics and compiling technology since they simply add data dependencies that can be handled in almost the same way as signal dependencies, see [44], but they would make the Coq formalization more complex.

1.3 Causality issues

Synchrony immediately leads to causality issues we explain with the following three examples:

```
P1: if S then emit S end
P2: if S then nothing else emit S end
P3: if S then emit S else emit S end
```

P1 looks non-deterministic: if *S* is present, it is emitted, which is logically fine; if *S* is absent, it is not emitted, logically fine as well. P2 is clearly nonsensical: if *S* is present, it is not emitted, contradicting the aforementioned basic rule; if *S* is absent, it is emitted, which sets it present, again a contradiction. But P3 is more problematic: in classical logic, one would say that *S* present is not contradictory since it makes it emitted, while *S* absent is contradictory since it makes it emitted as well. Thus, classical logic would accept *S* present as the unique solution, by the excluded middle rule. But we do not accept this viewpoint for Esterel because it is neither natural nor understandable in big programs. We want *causal and constructive* reasoning.

An easy way to reject all three cases above is to perform a topological sorting of signals, forbidding a signal to depend on itself even by a long chain of other signals that ends up with itself. This was adopted by the first formal semantics of Esterel in 1984 [6] and remains the rule for many aforementioned synchronous languages. But Esterel’s main industrial user in the 1990’s, a Dassault Aviation team, complained because it was developing large modular Esterel program for safety-critical avionics that sometimes led for good reasons to the following dependency structure:

```
if S1 then
  if S2 then emit S3 end
else
  if S3 then emit S2 end
end
```

Here, the *S2* and *S3* signals cannot be topologically ordered since the two `if` statements generate a trivially cyclic dependency graph, but there is no causality problem since the `then` and `else` branches cannot be executed together in a single reaction.

Georges Gonthier proposed a clever but partial solution [28] that was implemented by the Esterel team together with many other improvements in the Esterel v3 compiler he designed with the second author. Gonthier’s solution did handle many cases but not all of them, sometimes not being able to tell the programmer whether her program is correct or incorrect – this definitely had to be improved for industrial usage.

Finally, the causality problem has been fully solved by the *constructive semantics*, first presented in [4], the one detailed and proven correct here. It is used by the Esterel v5 and v7 compilers.

► **Remark 1.** One could think that P2 above provokes some sort of deadlock, as found when programming in asynchronous concurrent languages. Asynchronous deadlock are relatively easy to make and hard to detect at runtime, especially if they are partial, letting other threads of the program run normally. Therefore, in the asynchronous domain, it is highly recommended to stick to provably deadlock-free asynchronous protocols. The situation is different and much simpler in synchronous languages, since deadlocks are always detected and forbidden at compile-time. But the goals of both kinds of language are quite different.

1.4 Switching to the circuit semantics and implementation

In 1989, the second author was invited by Jean Vuillemin to the Digital Equipment Paris (DEC) Research Lab to work on the control logic of computation-oriented circuits implemented on the first Xilinx FPGAs (rewirable-by-software hardware circuits). They realized that Esterel’s synchrony hypothesis was well-known for acyclic synchronous hardware circuits: electrical fronts propagate in wires in an asynchronous way, but with a deterministic result computed in a predictable and very short time; at the end of each clock cycle, all wires are electrically stable to voltages that exactly correspond to the solution of the circuit’s Boolean equations.

Since this is exactly Esterel’s viewpoint, they could directly implement Esterel on synchronous circuits or FPGAs, each reaction lasting one clock cycle with data computations attached to particular gates. Furthermore, the generated circuits could be easily and reasonably efficiently simulated to build software code usable either as a circuit simulator or directly in applications. The Esterel v4 compiler followed, with the benefit of definitively solving the generated code-size explosion problem often encountered with Esterel v3. The software generation could now scale up to very large problems. Then, very efficient optimizers and verification methods for the generated circuits were developed based on Boolean Decision Diagrams (BDDs) [15], the optimized circuits turned out to be often smaller and more efficient than when designed as usual in Verilog or VHDL, and the generation of software code was improved in the same way.

Esterel v4 was limited to the acyclic case, a natural condition in the hardware field. However, a seminal article by Sharad Malik [37] studied cyclic hardware circuits and showed that some of them did compute deterministically and in bounded time, as for acyclic ones. In that case, they could even be much more symmetric and natural than any equivalent acyclic circuit.

The second author then realized that the Esterel cycle analysis problem could also be stated on the Boolean equations of a cyclic circuit and that it could be solved at that level by replacing classical Boolean logic by *constructive Boolean logic*, which is a logic that only propagates known Boolean values through logic operators, without ever using excluded-middle reasoning. The adaptation of this constructivity idea to Esterel led him to the final *constructive semantics* of Esterel described in the web book [4]. A new circuit generation backend able to generate correct cyclic circuits was built for the Esterel v4 compiler, which was then renamed Esterel v5, and a BDD-based tool was added to statically determine if a cyclic circuit is constructive and optionally to generate an acyclic (but possibly much bigger) equivalent one. The circuit implementation was later used for the Esterel v7 industrial compiler, which targets C, Verilog and VHDL, thus unifying software and hardware applications.

In 2012, Michael Mendler fully justified this viewpoint by proving the converse result: a cyclic circuit yields the expected result in finite time for all gates and wires delay *if and only if* this result is computable from the Boolean equations by using only constructive Boolean logic [40], a fact long conjectured by the second author; in our opinion, this definitely shows that constructive circuits exactly correspond to Esterel’s constructive semantics.

1.5 Summary of the final Esterel semantics chain

Each of the formal semantics we analyze in this article is presented in Plotkin’s SOS (Structural Operational Semantics) style, which is based on logical rules that are very versatile and quite naturally suited to formal analysis with systems such as Coq. All but the last one are detailed in [4]. The last *microstep semantics* is new and was created by the first author to mimic the behavior of circuits in an SOS-style. It gets very close to the level of synchronous circuits, which serve as the final implementation of Esterel. All Coq analysis and proofs in this article are due to the first author.

The *behavioral semantics*, introduced in [6], defines logical constraints that ensure reactivity, i.e., the existence of a solution, but not determinism. It is presented as a set of SOS rules, where each SOS transition computes the output signals and transforms a program text into a new program text ready for the next reaction. This semantics is based on natural constraints, but it is still too loose in the sense that it accepts some programs that happen to be deterministic “just by chance”, i.e., in a non-causal way such as for P3 above. (A slightly different version due to Olivier Tardieu [52] does ensure determinism, but adding some technical complexity. We do not find it relevant for this work, for which the logical semantics is just a historical starting step). Nevertheless, it was enough to build an interpreter and then the first Esterel v2 academic compiler [23] by adding a partial dependency ordering on signals that ensures determinism in quite a strict way. This v2 compiler was based on the transformation of a program into a finite automaton, following Brzozowski derivative-based construction [14] of finite automata from regular expressions, later improved in [11].

The *potentials-based semantics* [7], due to Georges Gonthier, adds sufficient conditions to accept many more programs, including cyclic ones, but not all those that should be considered as correct – which was a problem for users. The *state semantics*, also due to Gonthier, refines it by replacing rewriting the full program text done in the potentials-based semantics by moving state markers in the program’s source code. It led to a much faster implementation with the Esterel v4 compiler, the first one to be industrialized. But users rightly complained that when a program was rejected, the compiler could not tell whether it was the program’s insufficiency or a real programming error. These semantics are now obsolete.

The final solution came with the *constructive semantics* [4], which refines the logical semantics by imposing proof-theoretic constructivity constraints that ensure a causal and disciplined flow of information in the program; this implies determinism in a natural way. It is presented as a richer set of SOS rules, and has become the reference semantics of the language. It was similarly paired with a *state constructive semantics* that moves markers in the source code.

The *constructive circuit semantics* goes further by transforming a Kernel Esterel program into a flat Boolean circuit (i.e., system of equations) with the exact same behavior, provided one uses constructive logic when evaluating the circuit’s equations (no excluded middle). As said before, using constructive Boolean logic is equivalent to letting electricity find the right solution even in the presence of cycles, as proved in [40]. The state constructive semantics is the basis of the industrial v5 and v7 compilers [27] to hardware circuit designs and software code, by implementing the circuits Boolean equations in C for software. Furthermore, the circuit and the C code that simulates it can be heavily optimized using modern circuit CAD tools.

Finally, the *microstep semantics* presented here and due to the first author precisely describes the fine-grained step-by-step propagation of information in an Esterel program during one reaction, according to the current state computed from the last reaction and the current input. This is done at the level of Esterel statements unlike the second author’s unpublished microstep semantics, which relies on the circuit as an intermediate to define the microstep semantics. A first attempt at a microstep semantics at the level of Esterel statements was published in [44], but the one we present here is technically much closer to the circuit semantics. It is presented in an SOS-style amenable to Coq proofs, while circuits are graphs, which are harder to manipulate in Coq.

This article only deals with the behavioral, constructive, state constructive, and new microstep semantics. We prove the expected refinement or equivalence relations between these semantics. The Coq formalization only handles Kernel Esterel without loops, since loops create a schizophrenia problem (see Section 9.1) that was solved in a way we have not yet transported to Coq. This work is enough to deduce a correct-by-construction interpreter for loop-free Kernel Esterel (actually one for each semantics). But going to an efficient compiler will require handling the circuit semantics, which will be the goal of a subsequent article.

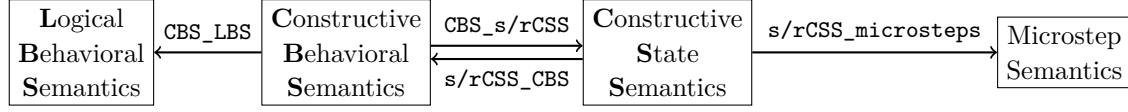
1.6 Related work

After 1985, research inspired by Esterel took place for other synchronous languages in several labs: Reactive C [13] by François Boussinot, in the same lab as Esterel; Argos [39] by Florence Maraninchi, a graphical formalism developed in the Lustre group and inspired by both Esterel and David Harel’s Statecharts (which were not quite synchronous in our sense); SyncCharts by Charles André at Nice University, an Argos-inspired graphical language that generated Esterel code [1]; Reactive ML [38] by Louis Mandel and Marc Pouzet at Ecole Normale Supérieure Paris, which adds Esterel-like statements to the functional OCaml language; Quartz [48] by Klaus Schneider’s team in Germany, now at the core of the Averest System³; SCL [56] and SCCharts [55] by Reinhard von Hanxleden and Michael Mendler also in Germany; finally, some Ptolemy II domains [45] and more recently Lingua Franca [36] by Edward A. Lee and his team at Berkeley University.

Considering that synchronous languages target particularly safety-critical systems, formal verification has been carried out since their beginning. The first formal verification efforts were targeted toward verifying program properties, to ensure the absence of bugs in the program but not in the compiler. For instance, the verification of properties of Esterel programs was done with the Auto/Autograph verifier [46], and for Lustre programs with the Lesar [32] and Kind2 [18] model-checkers; the Averest toolset for Quartz supports formal verification with BDD and SAT techniques.

Verification of the compilation from Signal to C was done by Van Chan Ngô *et al.* [42, 19] using translation validation, whereas the compilation from Lustre to C was verified by Bourke *et al.* [12] using a direct proof. Closer to our purpose, the first preliminary attempt [34] at formalizing the Esterel v5 compiler in Coq only considered the first instant. It also disregarded loops because of schizophrenia, just like we do here (see future work in Section 9.1). Nevertheless, it was enough to uncover bugs in the initial constructive semantics attempt. A more complete formal proof of a compiler of the Quartz language to circuits was done in the proof assistant HOL4 by Schneider *et al.* [48]. Their semantics is not defined in Plotkin’s SOS-style, as used by Esterel, but is instead based on logical predicates, which are closer to automata and circuits. They handle data, which we do not.

³ <https://www.averest.org>.



■ **Figure 1** The chain of Kernel Esterel semantics. Arrows represent simulation.

Two microstep semantics have previously been defined for Esterel in the literature. The first one is in the “Compiling Esterel” book [44]. It is slightly higher-level than our microstep semantics as it only propagates control and does not perform a detailed computation of completion codes. It handles data but still uses the *Can* function for the local signal rules. Overall, it may be more suited for reasoning and explaining reactions but it is less suited to a translation to digital circuits. The other one [47] is defined on the Quartz language, a variant of Esterel. It relates the original Quartz semantics given in terms of logical predicates to a new SOS semantics for Quartz, in the spirit of the SOS semantics of Esterel. Its objective is to explain the reactions of Quartz programs, not to be closer to the circuit semantics. Considering its focus and level of detail, we do not consider it to be a microstep semantics, as it is much closer to the state semantics of Section 6 than to our microstep semantics or the one in [44].

1.7 The contents of this article

This article proves the correctness of the Esterel semantics chain presented in [4] and described above, using the Coq proof assistant. The Coq proofs currently do not handle loops. This is essentially the work of the first author. After describing Kernel Esterel and its informal semantics in Section 2, and the Coq proof assistant and high-level features of the formalization in Section 3, the following semantics are described: first the behavioral semantics, namely the logical one (Section 4) and the constructive one (Section 5); then the constructive state semantics (Section 6), and finally a new fine-grained microstep semantics (Section 7). We also prove the relations between these semantics (simulation, equivalence), see Figure 1. Finally, we detail some aspects of the Coq formalization in Section 8 before concluding in Section 9.

The final operational microstep semantics presented at the end of the chain is not yet the original circuit semantics, but it is technically very close to it. As said before, the difference lies in the difficulty of dealing with graph manipulation with Coq. This very last step remains to be completed. Once this final simulation to the circuit semantics is completed, this chain of simulations will prove that the translation of Esterel programs to circuits [4], that is, the compiler, is correct: it preserves the semantics of Esterel programs down to the generated digital circuits. Nevertheless, this article is a major step towards the final justification and publication of the book [4] with formal proofs included as Coq programs executable by anybody who wants to check them.

2 Kernel Esterel

2.1 Signals, instants, and synchrony

Kernel Esterel is the core concurrent reactive language that deals with *pure signals*, carrying a *present/absent* status but not data, denoted by identifiers and abbreviated into *signals* in this article. An *event* is a set of present signals. A Kernel Esterel program P is defined by an *interface*, which defines the set $\mathcal{I}$ of its *input signals* and the set $\mathcal{O}$ of its *output signals*, and a *body* which is an executable statement that may declare locally scoped signals.

The execution of a Kernel Esterel program consists of successive *reactions* to *input events* E that generate *output events* E' according to the behavior of the body. The *synchrony hypothesis* stipulates that reactions have conceptually no duration: inputs do not vary and outputs are produced instantly in a deterministic way, local signals have a unique status within their scope. To stress that point, we call reactions *instants*. The synchrony hypothesis is the key to reconciling concurrency and determinism.

At each instant, each signal s carries a unique *present* or *absent status*. A signal s is set *present* either if it is an input set present by the program environment for the current instant or if it is a local or output signal emitted by some “`emit s`” statement executed during the current instant. If it is neither a *present* input nor emitted by some “`emit s`” executed statement, a signal s is declared *absent* for the instant; thus, a pure signal behaves as a Boolean hardware *or* gate that takes value 1 when any of its inputs has value 1, or value 0 where all its inputs take value 0. Each signal status in the input event is instantaneously broadcast to all active statements, which means that all of them see the same status for the signal at an instant, be they in sequence, in parallel, or embedded in any active statement anywhere in the program.

The status of a signal can be tested by an “`if s then p else q end`” statement (the keyword is `present` in [4] and Esterel v5, but we use `if` here as in Esterel v7). In this article, we shall also allow testing for signal absence, which simply exchanges the `if` branches; this is not indispensable but simplifies the presentation.

In actual implementations, instants are externally determined by the environment when it sends input; what is important is that the real time of a reaction is sufficiently small with respect to the timing constraints of the controlled process for the semantics to be preserved. Implementations may rely on atomic calls of a C function implementing the global program behavior (as for Esterel v5), on circuit clock cycles for translation to hardware (as for Esterel v7), on execution of atomic-by-construction JavaScript code (as for HipHop), or even on appropriately synchronized execution of a distributed program provided interference between I/O and execution is avoided, as in Lingua Franca [36]. Since they do not interfere with the semantics, these practical implementations are outside the scope of this article.

► **Remark 2.** In Full Esterel, signals can carry data values that are not available in the kernel. This is not really a limitation: the development of Esterel semantics and compilers has shown that dealing with shared data values can be done in much the same way as dealing with the pure signals that carry them [44]. The only difference is that knowing the value of a valued signal requires the resolution of all emitters to combine the individual values emitted by the active emitters using a user-specified associative and commutative function. This means that the value of a valued signal depends on the status of all emitters, which is exactly the condition for determining the absence of a pure signal. Of course, one must also add data dependencies, but this is as for any classical language. Nevertheless, as said before, we have not yet handled valued signals in the Coq proofs.

2.2 Kernel statements

Kernel Esterel contains a small number of statements, from which one can easily define the richer statement set of the user-friendly full language [2, 27]. A statement starts at some instant and may execute either instantaneously, i.e., entirely within the instant, or up to some further instant where reactions produce empty results from then on, or even indefinitely. Its starting and ending (if any) instants define its *lifetime*.

The statements can be presented in two equivalent forms: with keywords, which make reading easier, or with mathematical symbols, a compact notation for semantic rules and proofs. We use the latter symbolic form in all technical developments. Here are the kernel textual statements,

where s is a signal, T is a trap name, k is an integer and p and q are kernel statements. The statement order is not the same as in the original Esterel articles, but it will be technically more convenient here:

Textual form	Symbolic form
<code>nothing</code>	0
<code>pause</code>	1
<code>emit s</code>	$!s$
<code>await immediate s</code>	$@s$
<code>if s then p else q end</code>	$s ? p, q$
<code>suspend s when p</code>	$s \supset p$
<code>trap T in p end</code>	$\{p\}$
<code>exit T^k</code>	k (with $k \geq 2$)
	$\uparrow p$
<code>$p; q$</code>	$p; q$
<code>loop p end</code>	p^*
<code>$p \parallel q$</code>	$p \mid q$
<code>signal s in p end</code>	$p \backslash s$



We slightly depart from [4] in two ways: first, we rename the `present` test for signals of [2] and Esterel v5 into `if _ then _ else _ end` as in Esterel v7 [27]; second, we add an “`await immediate`” wait statement to the kernel, written as $@s$, which can be defined from the other primitives as follows:

```

trap T in
  loop
    if  $s$  then exit T else pause end
  end loop
end

```

The reason to include “`await immediate  $s$` ” here is that the simulation proof in Coq between the constructive state semantics (Section 6) and the microstep semantics (Section 7) does not handle loops yet, because they lead to the signal instantaneous reincarnation problem, explained and solved in the last chapter of [4] (see Section 9.1). But, in order to express suspension “`suspend  $p$  when  $s$` ” in the logical and constructive semantics (see Figure 2), we need “`await immediate  $s$` ” to wait until s becomes absent before performing another step of p , current instant included, which is what `await immediate` does. In Esterel, the “`await  $s$` ” statement is more familiar than “`await immediate  $s$` ” and can be easily defined as “`pause; await immediate  $s$` ”. Nevertheless, the fact that it cannot terminate in its starting instant makes it less suitable for our purpose.

2.3 Completion codes and the trap encoding

The `trap-exit` lexically scoped statement is unique to Esterel (although a fairly similar construct exists in David Harel’s graphical Statecharts, but with a different semantics) and needs more explanations. It is akin to an explicit user-driven error-handling statement in a sequential language, as for `try-throw` in Java or `try-raise` in Python. Furthermore, it is fully compatible with synchronous concurrency and fully deterministic: executing “`exit T`” anywhere means asking to terminate the body of the enclosing “`trap T`” statement as soon as it has been completely evaluated in the instant, even if it has many parallel components, but respecting a scope order for traps if the body is concurrent: if several traps are simultaneously exited by concurrent statements,

only the outermost one matters and the other ones are discarded. If no trap is exited, the trap terminates or pauses as its body does. Such a statement is essential for constructing derived statement from the kernel and is heavily used when programming large applications.

In the symbolic forms, this seemingly complex behavior and more generally the whole control propagation is realized using a simple *completion code* integer encoding due to Georges Gonthier and used in all compilers. Code 0 means termination, thus **nothing** simply becomes code 0 in the symbolic form; code 1 means pausing for the instant and waiting to be resumed at the next instant, thus **pause** becomes 1. A trap is just a marker, written $\{.\}$, and an “**exit T**” becomes code k with $k \geq 2$, which encodes exiting the enclosing **trap** reached by traversing $k - 2$ nested traps. This encoding greatly simplifies the semantic rules. Although it is not necessary, we add the code as an exponent to the trap name in our textual **trap-exit** examples for more clarity.

This is how the encoding works: at each instant, each executed statement returns a completion code, and the composition of these codes determines the control flow of the program in a deterministic way. In concurrent traps, each parallel statement waits for the completion codes returned by all branches and returns the *maximum* of these codes. This means that a parallel statement terminates if and only if all its branches terminate, pauses if at least one branch pauses and no branch exits a trap, and exits the outermost trap exited by branches if at least one exits a trap; terminating, pausing, or other traps exits are ignored. Determinism is therefore enforced.

For example, consider the statement

```

trap T in
  exit T2;
  trap U in
    exit T3
  ||
    exit U2
  end
end
end

```

where “**exit T**” is successively decorated by 2 and 3 because the second one is enclosed in “**trap U**”. This textual code simply becomes $\{2; \{3 \mid 2\}\}$ in the symbolic form.

► **Remark 3.** This encoding of traps is akin to the De Bruijn encoding of bound variables in the λ -calculus [24], but in a concurrent setting and adding 2 to account for termination (0) and pausing (1).

2.4 A running example

We now present a slight variant, called **ABROi**, of the **ABRO** program, which can be considered as the “Hello World!” of Esterel that starts most Esterel descriptions [4]. It features the most striking specificities of Kernel Esterel: deterministic parallelism, instantaneous reactions to presence/absence of signals, simultaneous reception of signals (and emission, not illustrated here), and using a trap statement capable of instantaneously canceling parallel activities. The reader can check that its behavior is much harder to write in sequential languages (including Javascript), and even with asynchronous threads or asynchronous parallel languages.

► **Example 4 (ABROi program).** The **ABROi** program involves three input signals **A**, **B**, and **R** and one output signal **O**. Its behavior can be specified as follows: as soon as signals **A** and **B** have been both received, either in succession or simultaneously, emit **O** and do nothing from then on; restart the same behavior afresh whenever **R** is received.

Here is `ABROi` in plain Esterel:

```
loop
  [ await immediate A || await immediate B ];
  emit 0
each R
```

The difference with classical `ABRO` is that `0` is also emitted if `A` and `B` are simultaneously received when `R` occurs. The original `ABRO` behavior, where `0` is never emitted in any instant where `R` is present, would be obtained by adding a `pause` in sequence right before the parallel.

Expanding with kernel statements the “`loop-each R`” loop that restarts its body each time `R` occurs, as first done by compilers, one gets:

```
loop
  trap T in
    loop
      pause;
      if R then exit T2 else pause end
    end
  ||
  [ await immediate A || await immediate B ];
  emit 0;
  loop pause end;
  exit T2
end
end
```

The last “`exit T2`” is generated by the macro-expansion of `loop – each` but is unreachable here, which is detected by compilers. We keep it in the sequel anyway. The much more compact symbolic form will be useful for semantics rules: $\{(1; (R?2, 1)*) \mid ((@A \mid @B); !O; (1*); 2)\}^*$, noting that $(1*)$ is simply called `halt` in Esterel. The reader is referred to [2, 4] for additional examples.

2.5 Intuitive Semantics of Kernel Esterel

The intuitive semantics is defined by cases over the statements.

- The “`nothing`” or `0` statement instantly terminates, returning completion code 0.
- The “`pause`” or `1` statement waits for the next instant. At its starting instant, it stops control propagation and returns completion code 1; at its next execution instant, it terminates and returns code 0. This is not necessarily the instant that follows the starting instant since the `pause` statement could be preempted or suspended.
- At a given instant, the status of a signal s is shared by all program components within its scope. By default s is absent in a reaction. Any executed “`emit s`” or `!s` statement sets s present for the instant. The `emit` statement terminates instantly.
- The “`await immediate s`” statement blocks control propagation until s is present. At every instant including the starting one, it terminates instantly if s is present, or it pauses and continues waiting if s is absent.
- The presence test statement “`if s then p else q end`” or $s?p, q$ instantly tests the status of s . According to the presence/absence of s , it selects p or q for immediate execution and behaves as it from then on. Note that the test is only performed at the instant in which the statement is started.

2:12 Towards a Coq-verified Chain of Esterel Semantics

- The delayed suspension “**suspend** p **when** s ” or $s \supset p$ statement behaves as p in its starting instant. In all subsequent instants during the lifetime of p , it executes p whenever s is absent in the instant or freezes the state of p until the next instant if s is present. Notice that the status of s is tested at all instants except for the starting one. The completion code at the first instant is that of p ; for subsequent instants (if any) it is that of p if s is absent or 1 if s is present. Thus, termination and trap exits of p are only propagated at the first instant or if s is absent.
- The loop “**loop** p **end**” or p^* statement instantly restarts its body p when this body terminates. The body p is not allowed to terminate instantly when started. Notice that exiting an enclosing trap is the only way to exit a loop. The **abort** statement of the full language is definable in the kernel language using traps.

In the Coq formalization presented here, we shall not handle loops because this would involve additional technical complications. The book [4, ch. 12] explains how to handle them by using decreasing ordinals that should be implemented in the Coq semantics. This is left as future work (see Section 9.1).

- At each instant, the “**trap** T **in** p **end**” or $\{p\}$ statement executes p for the instant; it terminates or pauses if p does, terminates if p returns completion code 2 (i.e., catches its exits), or returns completion code $k - 1$ if the completion code of p is $k > 2$ (thus propagating exits to the appropriate outer trap).
- The textual “**exit** T^k ” or symbolic k statement with $k \geq 2$ simply returns completion code k .
- The $\uparrow p$ symbolic statement is necessary to define the macro-based full-language statements that place p in a “ $\{_ \}$ ” trap context. It simply adds 1 to any trap completion code $k \geq 2$ returned by p . It is not needed in the textual form since traps are named.
- For a sequence “ $p; q$ ”, the statement q instantly starts if and when p terminates. Trap exits by p or q are propagated. Note that both p and q are executed in the instant when p terminates.
- The textual “ $p \parallel q$ ” or symbolic $p \mid q$ parallel statement terminates instantly as soon as both branches have terminated; it pauses if at least one branch pauses and no branch raises an exit; it exits a trap instantly if one of its branches does. If several branches concurrently exit traps, it only exits the outermost exited trap. In the symbolic formal semantics, this behavior simply reduces to the parallel statement $p \mid q$ returning as completion code the maximum $\max(k_p, k_q)$ of the completion codes of its branches at each instant.
- Finally, a local signal declaration “**signal** s **in** p **end**” or $p \setminus s$ declares a signal s local to its body p , with the usual lexical binding and shadowing.

► **Example 5** (Intuitive execution of the **ABROi** program). Consider the execution of the **ABROi** program (Example 4) during five consecutive instants where the following inputs are received:

1. $\{B\}$: only **B** is received, so **ABROi** keeps waiting on **A**;
2. $\{A, B\}$: as **A** is received, **0** is instantly emitted and **ABROi** reaches the **halt** statement – a natural name for the “**loop pause end**” or 1^* statement in the code of **ABROi**;
3. $\{B\}$: the **halt** statement stalls execution, as the outer loop has not yet been restarted; **B** is ignored;
4. $\{R\}$: receiving **R** kills the **halt** statement and restarts the loop so that **ABROi** again waits for **A**, **B**, and **R**;
5. $\{A, B, R\}$: receiving **R** kills and restarts the whole body of the loop, but **0** is emitted since **A** and **B** are both present.

3 Esterel semantics in Coq

3.1 The Coq proof assistant

A proof assistant is software that automates the verification of proofs, thus building increased confidence in their correctness. This is of particular importance and interest for critical system software as well as domains where proofs are notoriously difficult, such as distributed systems. Unlike for other formal methods such as model checking, writing a proof is not fully automated but assistance is usually provided in the form of tactics (“proof steps”) and decision procedures for some domains.

Coq is a proof assistant providing a unified formal language where both pure functional programs and proofs can be written, type-checked and run. It has been successfully used to prove both mathematical results as involved as the odd-order theorem [29] and verify software such as the CompCert C compiler [35]. The logical foundation of the Coq proof assistant is the Calculus of Inductive Construction [43], a powerful extension of the higher-order typed λ -calculus in which types and logical formulas are unified, so that writing a proof amounts to providing a λ -term of a given type.

The computational part of these terms can be extracted to functional programming languages (currently, Haskell, OCaml, and Scheme) and thus integrated as verified components in bigger programs. For instance, the CompCert compiler can be extracted from its Coq code, thus ensuring that the executed code is indeed the verified one. The compiler and runtime environment of the functional language must still be trusted.

The trust one may have in proofs accepted by such software is limited by the trust in the software itself. In order to reduce the trusted code base, Coq and other proof assistants are built around the so-called *de Bruijn principle*, in which only a reasonably small kernel that checks proof validity must be trusted, while the rest (tactics, automation, etc.) need not be. Other ways to increase confidence in a proof assistant are to provide an independent proof checker or even to aim at building formal proofs in the tool itself⁴ of the correctness of its logic and its software implementation [51].

3.2 The formalization of Kernel Esterel in Coq

Restriction to loop-free Esterel

The Coq formalization is currently restricted to Kernel Esterel without the looping construct p^* . Indeed, loops can interact with local signals in a subtle way, requiring a specific treatment for the simulation proof between the constructive state semantics of Section 6 and the microstep semantics of Section 7 which we do not handle yet, see [4, chap. 12] and Section 9.1 for details. In order to keep a coherent body of proofs, loops were entirely removed from Esterel in the whole Coq formalization, although they are easy to handle in the semantics.

High-level view of the formalization

The syntax of Kernel Esterel is represented in Coq as an inductive type, making it possible to perform proofs by induction on the syntax tree of a program. Similarly, each of our SOS semantics will be represented by an inductive type, making it possible to perform proof by induction on derivation trees. In this setting, proving simulation, respectively, equivalence, between two

⁴ Notice that this only increases confidence and is not a definitive answer because of Gödel’s second incompleteness theorem.

semantics amounts to proving that the existence of a derivation tree in the source semantics implies, respectively, is equivalent to, the existence of a derivation tree in the target semantics. Therefore, a typical proof goes by induction on the derivation tree in the source semantics and builds an adequate derivation tree in the target semantics.

The full Coq code for the syntax, semantics, and proofs presented in this article is available as supplementary material at the following address: <https://zenodo.org/records/15064811>.

Most proofs present no technical difficulty and rely mostly on a straightforward induction. They will not be detailed here, where we will only focus on the interesting points. We refer the interested reader to the Coq development. This article contains links to the html documentation generated from the Coq development, permitting a more comfortable browsing of the formalization. If you put the file corresponding to this article in the root directory of the Coq development, you

will be able to directly use these links, represented as  in the margin, to access the relevant documentation.

3.3 Representation choices in Coq

The Coq formalization tries to be faithful to the symbolic notation of Kernel Esterel, in particular to [4], but there are still two syntactic differences. First, the parallel composition $p \mid q$ is written with the textual notation $p \parallel q$ for technical reasons: the single pipe $|$ is used in Coq for pattern matching. Second, borrowing standard notation for lists, extending an event E by mapping the signal s to status b (with possible overshadowing of a previous signal s) is written $s^b ++ E$ rather than $E * s^b$.

There are also less superficial changes compared to the literature that we describe now. Other remarks or specificities of the Coq formalization will be signaled in this article as **Coq Remarks**. If the reader is not interested in the technical details of the formalization, they can be safely ignored.

Inputs and outputs events

Signal inputs and outputs are described with events (written $\mathcal{I}$ and $\mathcal{O}$ respectively) that may range over different sets of signals. (We usually have $\mathcal{O} \subseteq \mathcal{I}$ to represent the fact that any emitted signal can be instantaneously read.) In this article, input and output events E and E' are maps from the set of signals in scope to statuses, which makes it possible to prove that the set of visible signals is not modified during execution (see property “Domain invariance” in Section 8.1). This is different from the presentation of this article and of [4] where E' is a set of emitted signals, which amounts to the set of signals mapped to $+$ in our representation.

Setoid of signals

Signals are represented by a setoid, that is, a type equipped with a dedicated equality. In particular, signals do not use the equality of Coq and may use any coarser equivalence. The downside of this choice is that every semantics must have an extra rule **compat** ensuring compatibility with this equivalence, that is, a rule stating that one can replace a signal by any equivalent one, both in programs and in events. This is the last rule in all the semantics in the formalization. Furthermore, the **inversion** tactic is no longer useful as this compatibility case always applies. Nevertheless, it is straightforward to prove dedicated inversion lemmas and write a tactic mimicking **inversion**. The benefit of this choice is that we can use extensional equality for events and we can optimize signal expressions (once we introduce them in the formalization): instead of structural equality, we can choose propositional equality, so that for instance $s \& s'$ becomes equal to $s' \& s$. Another solution would have been to use Coq equality for everything (signals, events, statements and so on) and

either add axioms ensuring that this equality is extensional on events or to use an implementation of events featuring a unique canonical representative (for instance, ordered associative lists without duplicate keys).

Delta function

Whenever the completion code k of a statement is not 1, there is nothing left to execute, either because we finished execution ($k = 0$) or because a trap killed the remaining state ($k \geq 2$), hence the statement for the next instant, called the *derivative*, should be 0. The usual rules for Esterel in the literature do not have this property: for instance the trap rule from Figure 2 always keeps the trap. This makes rules easier to write and read but is sometimes inconvenient. Although we stick to this tradition in this article, the Coq formalization performs this normalization of the derivative to 0 whenever $k \neq 1$ by introducing a function δ defined as follows:

$$\delta(k, p) := \begin{cases} p & \text{if } k = 1 \\ 0 & \text{otherwise} \end{cases}$$



Then, we replace each derivative p' with $\delta(k, p')$ (some rules do not require this, namely the ones for k , $!s$, $s ? p$, q , $@s$, and $p ; q$ when $k_p = 0$). Although it is not mandatory, this normalization technically and conceptually simplifies the calculations by setting all terminated terms to 0 which makes termination checks simpler. The same choice is made for the state semantics, the only difference being that the derivative is not 0 but the inert form of the state.

Well-formed programs



When writing a Kernel Esterel program, there are implicit well-formation rules. For instance, it is not allowed to emit or read a signal that is not in scope. In compiler implementations, this is ensured by typing checks. Here, all semantic rules using signals ensure this property with a premise $s \in E$ which implies that s is in scope.

In Coq, we also define the $\text{valid_dom}(E, p)$ predicate expressing that all signals free in p are in the domain of E , thus making sure that the evaluation of p is properly defined. The definition is made by a straightforward case analysis and amounts to requiring $s \in E$ every time a signal is explicitly used, that is, for statements $!s$, $@s$, $s \supset p$, and $s ? p$, q , but not for local signal declaration in which it is added to the domain of E instead. See the Coq code for details.

4 The logical semantics



The SOS-style logical (behavioral) semantics (LBS) defines constraints that ensure reactivity of a program to a given set of inputs, i.e., absence of deadlock and presence/absence of all signals. In the version given here, it does not guarantee determinism; Tardieu [52] proposed a version that also guarantees determinism, but we find it too heavy since it amounts to verifying that all non-deterministic executions yield the same result, which may require considering an exponential number of executions with nested local signals. Determinism will be guaranteed for much more compelling conceptual reasons by the constructive semantics detailed in the next section.

The logical semantics defines reactions as behavioral transitions represented by SOS rules of the form

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

where E, E' are *events*. The event E denotes the *inputs* of p while E' denotes its *outputs*, that is, the signals emitted by p . The integer k is the *completion code* of p for the instant, and p'

is the *derivative* of p by the transition, i.e., the statement replacing p for the next instant (this terminology is inspired by the work of Brzozowski and Seger on translating regular expressions to automata [16]). The rules are given in Figure 2.

Note that the rule $k \xrightarrow[E]{\emptyset, k} 0$ covers three statements: first, the trivial termination of 0 (**nothing**); second, the pausing case for 1 (**pause**) that returns completion code 1 and has derivative 0 (**nothing**) that will terminate instantly at the next reaction; third, the k (**exit** T^k) exit statement which returns completion code k and has derivative 0 (**nothing**).

Except for the rules sig^+ and sig^- that deal with signal declaration, our rules are a straightforward implementation of the intuitive semantics. The trap statement catches the innermost exit (the one with code 2) and turns it into termination (code 0). This is the meaning of the $\downarrow$ function, with (pseudo-)inverse $\uparrow$ used in shift.

$$\downarrow k := \begin{cases} 0 & \mapsto 0 \\ 1 & \mapsto 1 \\ 2 & \mapsto 0 \\ n \ (n \geq 3) & \mapsto n - 1 \end{cases} \quad \uparrow k := \begin{cases} 0 & \mapsto 0 \\ 1 & \mapsto 1 \\ n \ (n \geq 2) & \mapsto n + 1 \end{cases}$$

We have $\downarrow(\uparrow n) = n$ for any integer n but not $\uparrow(\downarrow n) = n$ because $\uparrow(\downarrow 2) = \uparrow 0 = 0$. The union $E'_p \cup E'_q$ in the seq_0 rule conclusion directly expresses the synchrony hypothesis within the reaction: control passes from p to q in the very same reaction. The same holds for the parallel statement (rule par).

The sig^+ and sig^- rules for local signals first extend the input event E with s mapped to a status b (possibly shadowing any upper-scoped signal s), written $E * s^b$, then execute p using this new input event, and finally either restore the status of the shadowed signal s if any or remove s from the output event E' , before returning it. This restoration or removal is written $E' \setminus s$. The completion code and derivative statement comes from the execution of p . The side condition $s^+ \in E'$ or $s^+ \notin E'$ ensures the correct status for s in p : in rule sig^+ we assume that s is received, so we check that it is indeed emitted ($s^+ \in E'$), and conversely for rule sig^- with s not received and not emitted. This is called the (*logical*) *coherence law* [4, chap. 3]: a signal s is present in an instant if and only if an “**emit** s ” statement is executed in this instant. The local signal rules intuitively mean that, at each instant, one needs to guess a coherent valuation of local signals, in a way that should satisfy the coherence law for all local signals. Satisfying the coherence law avoids some logical contradictions but is not enough because the guess may not be causally justified, a problem that will be solved by the constructive semantics presented in the next section.

As an illustration of the LBS, the execution of the **ABROi** program is given in Figure 3.



5 The Constructive Semantics

The introduction showed that synchrony comes with some causality issues, in particular the program $P3 = s ? !s, !s$ (“**if** s **then** **emit** s **else** **emit** s **end**”) is only valid in a classical setting as s is present because it is emitted but also emitted (**then** branch) because it is present. In order to remove such causality loops as well as non-determinism, the constructive semantics was introduced in [4]. We now focus on this semantics, which has become the reference for the language because it solves these issues and also because it provides a much better match with the circuit semantics [40].

Altogether, the constructive (behavioral) semantics (CBS) we now present differs from the logical one on two main aspects: the statuses of signals and the signal rules. Signals may take a third status $\perp$ representing the absence of information: we do not yet know whether a signal is

$$\begin{array}{c}
\frac{}{k \xrightarrow[E]{\emptyset, k} 0} \text{ k} \\
\frac{}{!s \xrightarrow[E]{\{s^+\}, 0} 0} \text{ emit} \\
\frac{s^+ \in E}{@s \xrightarrow[E]{\emptyset, 0} 0} \text{ awimm}^+ \qquad \frac{s^- \in E}{@s \xrightarrow[E]{\emptyset, 1} @s} \text{ awimm}^- \\
\frac{s^+ \in E \quad p \xrightarrow[E]{E', k} p'}{s ? p, q \xrightarrow[E]{E', k} p'} \text{ then} \qquad \frac{s^- \in E \quad q \xrightarrow[E]{E', k} q'}{s ? p, q \xrightarrow[E]{E', k} q'} \text{ else} \\
\frac{p \xrightarrow[E]{E', k} p'}{s \supset p \xrightarrow[E]{E', k} @\neg s; s \supset p'} \text{ suspend} \\
\frac{k \neq 0 \quad p \xrightarrow[E]{E', k} p'}{p * \xrightarrow[E]{E', k} p'; p *} \text{ loop} \\
\frac{p \xrightarrow[E]{E', k} p'}{\{p\} \xrightarrow[E]{E', \downarrow k} \{p'\}} \text{ trap} \qquad \frac{p \xrightarrow[E]{E', k} p'}{\uparrow p \xrightarrow[E]{E', \uparrow k} \uparrow p'} \text{ shift} \\
\frac{k \neq 0 \quad p \xrightarrow[E]{E', k} p'}{p; q \xrightarrow[E]{E', k} p'; q} \text{ seq}_k \qquad \frac{p \xrightarrow[E]{E'_p, 0} p' \quad q \xrightarrow[E]{E'_q, k} q'}{p; q \xrightarrow[E]{E'_p \cup E'_q, k} q'} \text{ seq}_0 \\
\frac{p \xrightarrow[E]{E'_p, k_p} p' \quad q \xrightarrow[E]{E'_q, k_q} q'}{p \mid q \xrightarrow[E]{E'_p \cup E'_q, \max(k_p, k_q)} p' \mid q'} \text{ par} \\
\frac{p \xrightarrow[E*s^+]{E', k} p' \quad s^+ \in E'}{p \setminus s \xrightarrow[E]{E' \setminus s, k} p' \setminus s} \text{ sig}^+ \qquad \frac{p \xrightarrow[E*s^-]{E', k} p' \quad s^+ \notin E'}{p \setminus s \xrightarrow[E]{E' \setminus s, k} p' \setminus s} \text{ sig}^-
\end{array}$$

■ **Figure 2** Logical (Behavioral) Semantics (LBS) rules.

$$\begin{aligned}
\text{ABROi} &= \{ (1; (R?2, 1)*) \mid ((@A \mid @B); !O; (1*); 2) \} * \\
&\xrightarrow[\{B\}]{\emptyset, 1} \{ (0; (R?2, 1)*) \mid ((@A \mid 0); !O; (1*); 2) \}; \text{ABROi} \\
&\xrightarrow[\{A, B\}]{\{O\}, 1} \{ (0; (R?2, 1)*) \mid (0; (1*); 2) \}; \text{ABROi} \\
&\xrightarrow[\{B\}]{\emptyset, 1} \{ (0; (R?2, 1)*) \mid (0; (1*); 2) \}; \text{ABROi} \\
&\xrightarrow[\{R\}]{\emptyset, 1} \{ (0; (R?2, 1)*) \mid ((@A \mid @B); !O; (1*); 2) \}; \text{ABROi} \\
&\xrightarrow[\{A, B, R\}]{\{O\}, 1} \{ (0; (R?2, 1)*) \mid (0; (1*); 2) \}; \text{ABROi}
\end{aligned}$$

■ **Figure 3** Execution of **ABROi** in the LBS.



emitted or not. This means in particular that no rule can apply to $s?p, q$ (if s then p else q end) if s 's status is $\perp$: the execution is blocked on the test. Only the signal declaration rule deals with the $\perp$ status. For this, it uses two auxiliary mutually recursive functions *Must* and *Can* to compute respectively the set of signals that *must* and *can* be emitted within an instant using only the information contained in the event E . They intuitively represent the information we can gather from the body of p *by making no assumptions on the status of the declared signal s* , that is, by setting its status to $\perp$ (unknown). This restriction ensures that the justification of the status of signal s does not rely on itself.

The union of events is now defined pointwise using Scott's disjunction on $\{\perp, +, -\}$, defined by the following equations.

$$\begin{aligned}
+ \vee x &= x \vee + = + & - \vee \perp &= \perp \vee - = \perp \vee \perp = \perp & - \vee - &= -
\end{aligned}$$

In essence, non-emission must be agreed upon everywhere.

As most of the rules of the logical semantics do not deal with signal statuses, we do not need to modify them. Moreover, the rules for $!s$, $@s$ and $s?p, q$ can also be kept unmodified, as they only refer to statuses $+$ and $-$. In fact, only the two rules for signal declaration (rules sig^+ and sig^- rules on Figure 2) need to be reworked, where handling $\perp$ is deferred to two auxiliary functions, *Must* and *Can*. Thus, the CBS contains exactly the same rules as the LBS, except for the sig^+ and sig^- rules which are replaced by the following C- sig^+ and C- sig^- rules:

$$\begin{aligned}
&\frac{p \xrightarrow[E * s^+]{E', k} p' \quad s \in \text{Must}(p, E * s^\perp)}{p \setminus s \xrightarrow[E]{E' \setminus s, k} \delta(k, p' \setminus s)} \text{C-sig}^+ & \frac{p \xrightarrow[E * s^-]{E', k} p' \quad s \notin \text{Can}^+(p, E * s^\perp)}{p \setminus s \xrightarrow[E]{E' \setminus s, k} \delta(k, p' \setminus s)} \text{C-sig}^-
\end{aligned}$$

5.1 The *Can* and *Must* functions

The auxiliary functions *Must* and *Can* are respectively an under- and an over-approximation of the final set of emitted signals. They coincide on most statements, differing only on four of them: $s?p, q$, $@s, p; q$, and $p \setminus s$. For instance, when the status of s is $\perp$ in a presence test $s?p, q$, we do not know which branch to execute so that nothing *must* be done; on the contrary, both branches *can* be executed, hence the difference between *Must* and *Can*.

Technically, we need to compute approximations for both signals and completion codes at the same time because the sequential composition $p; q$ needs to know if p must/can terminate to decide if q has to be considered. The computed signal and completion code sets are denoted by

adding a subscript to the *Must* and *Can* functions: s for signals and k for completion codes, as in $Must_s(p, E)$ for signals and $Must_k(p, E)$ for completion codes. We usually drop the subscript as the ambiguity is easily resolved from the context. Furthermore, in the case of *Must*, the set of completion codes is either empty or a singleton because of determinism, whereas it can be an arbitrary non-empty set for *Can*.

When $Must(p \setminus s, E)$ concludes that the status of the local signal s must be $+$, we want to propagate this information inside the evaluation of p to get more precise information. This can only be done when we are sure that $p \setminus s$ is executed, not merely that it *could* be executed as *Can* expresses. Therefore, *Can* needs to carry a $+$ tag telling whether the statement must be executed or a $\perp$ tag if we do not have this information.

The $\uparrow$ and $\downarrow$ functions, used for computing the completion codes of statements $\{p\}$ and $\uparrow p$, are extended pointwise to sets of completion codes, that is: $\downarrow K = \{\downarrow k \mid k \in K\}$ and $\uparrow K = \{\uparrow k \mid k \in K\}$. The maximum of two completion code sets K_1 and K_2 , written $\text{Max}(K_1, K_2)$ and used in the $p \mid q$ case, is the set of pointwise maxima: $\{\max(k_1, k_2) \mid k_1 \in K_1 \wedge k_2 \in K_2\}$. We abbreviate by $X \setminus e$ the removal of element e from set X , instead of the more standard but more cumbersome $X \setminus \{e\}$. Operations on sets are extended pointwise to pairs of sets by applying them on each component where meaningful. For example:

$$\begin{aligned} Can^\perp(p, E) \cup Can^\perp(q, E) &= (Can_s^\perp(p, E) \cup Can_s^\perp(q, E), Can_k^\perp(p, E) \cup Can_k^\perp(q, E)) \\ Can^m(p, E) \setminus 0 &= (Can_s^m(p, E), Can_k^m(p, E) \setminus 0) \\ Must(p, E) \setminus s &= ((Must_s(p, E)) \setminus s, Must_k(p, E)) \end{aligned}$$

Here, removing the integer 0 from a set of signals does not make sense, hence the second example only applies it to the second component of the pair; similarly, removing a signal s from a set of completion codes does not make sense, hence the third example only applies it to the first component of the pair. We also extend inclusion pointwise to these pairs by writing $Must(p, E) \subseteq Can^+(p, E)$ to mean both $Must_s(p, E) \subseteq Can_s^+(p, E)$ and $Must_k(p, E) \subseteq Can_k^+(p, E)$.

Must and *Can* are defined by mutual induction (because of the signal declaration case) over the statement p , see Figure 4.

► **Coq Remark** (The *Must* and *Can* functions in Coq). *There are two small differences between the presentation of *Must* and *Can* above and their Coq description. First, as we know that $Must_k(p, E)$ is either a singleton or the empty set, it is more convenient to use an option type to enforce this invariant: in Coq, $Must_k(p, E)$ is either **Some** k for some $k \in \mathbb{N}$ or **None**. Second, the tag m carried by $Can^m(p, E)$ to express whether p must be evaluated or not can take values $+$ (surely executed) and $\perp$ (maybe executed). In Coq, we use a Boolean instead with **true** representing $+$ and **false** representing $\perp$.*

5.2 Properties of *Must/Can* and of the constructive behavioral semantics

The constructive restriction ensures that the result is deterministic and that undefined statuses cannot be created during execution: they are only temporary statuses, meant to be used for local signals. More precisely, if we define a *total event* as a constructive event where no signal is mapped to $\perp$, we have the following lemmas:

► **Lemma 6** (Output events are total). *For all p , E , E' , k , and p' , if $p \xrightarrow[E]{E', k} p'$, then E' is total. (Notice that we make no assumption on E).*



► **Lemma 7** (Determinism of the constructive semantics). *For all p , E , E'_1 , k_1 , p'_1 , E'_2 , k_2 , and p'_2 , if $p \xrightarrow[E]{E'_1, k_1} p'_1$ and $p \xrightarrow[E]{E'_2, k_2} p'_2$, then $E'_1 = E'_2$, $k_1 = k_2$, and $p'_1 = p'_2$.*



$$\begin{aligned}
Must(k, E) &= Can^m(k, E) = (\emptyset, \{k\}) \\
Must(!s, E) &= Can^m(!s, E) = (\{s\}, \{0\}) \\
Must(@s, E) &= \begin{cases} (\emptyset, \{0\}) & \text{if } s^+ \in E \\ (\emptyset, \{1\}) & \text{if } s^- \in E \\ (\emptyset, \emptyset) & \text{otherwise} \end{cases} \quad Can^m(@s, E) = \begin{cases} (\emptyset, \{0\}) & \text{if } s^+ \in E \\ (\emptyset, \{1\}) & \text{if } s^- \in E \\ (\emptyset, \{0, 1\}) & \text{otherwise} \end{cases} \\
Must(s \supset p, E) &= Must(p, E) \quad Can^m(s \supset p, E) = Can^m(p, E) \\
Must(p *, E) &= Must(p, E) \quad Can^m(p *, E) = Can^m(p, E) \\
Must(\{p\}, E) &= (Must_s(p, E), \downarrow (Must_k(p, E))) \quad Can^m(\{p\}, E) = (Can_s^m(p, E), \downarrow (Can_k^m(p, E))) \\
Must(\uparrow p, E) &= (Must_s(p, E), \uparrow (Must_k(p, E))) \quad Can^m(\uparrow p, E) = (Can_s^m(p, E), \uparrow (Can_k^m(p, E))) \\
Must(s ? p, q, E) &= \begin{cases} Must(p, E) & \text{if } s^+ \in E \\ Must(q, E) & \text{if } s^- \in E \\ (\emptyset, \emptyset) & \text{otherwise} \end{cases} \\
Can^m(s ? p, q, E) &= \begin{cases} Can^m(p, E) & \text{if } s^+ \in E \\ Can^m(q, E) & \text{if } s^- \in E \\ Can^\perp(p, E) \cup Can^\perp(q, E) & \text{otherwise} \end{cases} \\
Must(p ; q, E) &= \begin{cases} Must(p, E) & \text{if } 0 \notin Must_k(p, E) \\ (Must_s(p, E) \cup Must_s(q, E), Must_k(q, E)) & \text{if } 0 \in Must_k(p, E) \end{cases} \\
Can^m(p ; q, E) &= \begin{cases} Can^m(p, E) & \text{if } 0 \notin Can_k^m(p, E) \\ (Can^m(p, E) \setminus 0) \cup Can^+(q, E) & \text{if } 0 \in Must_k(p, E) \text{ and } m = + \\ (Can^m(p, E) \setminus 0) \cup Can^\perp(q, E) & \text{otherwise} \end{cases} \\
Must(p \mid q, E) &= (Must_s(p, E) \cup Must_s(q, E), \text{Max}(Must_k(p, E), Must_k(q, E))) \\
Can^m(p \mid q, E) &= (Can_s^m(p, E) \cup Can_s^m(q, E), \text{Max}(Can_k^m(p, E), Can_k^m(q, E))) \\
Must(p \setminus s, E) &= \begin{cases} Must(p, E * s^+) \setminus s & \text{if } s \in Must_s(p, E * s^\perp) \\ Must(p, E * s^-) \setminus s & \text{if } s \notin Can_s^+(p, E * s^\perp) \\ Must(p, E * s^\perp) \setminus s & \text{otherwise} \end{cases} \\
Can^m(p \setminus s, E) &= \begin{cases} Can^m(p, E * s^+) \setminus s & \text{if } s \in Must_s(p, E * s^\perp) \text{ and } m = + \\ Can^m(p, E * s^-) \setminus s & \text{if } s \notin Can_s^m(p, E * s^\perp) \\ Can^m(p, E * s^\perp) \setminus s & \text{otherwise} \end{cases}
\end{aligned}$$

■ **Figure 4** Definitions of the functions *Must* and *Can*. The first component is for signals, the second one for completion codes.

As *Must* and *Can* represents under- and over-approximation of the behavior of statements under the CBS semantics, we have the following inclusions:

► **Lemma 8** (Properties of *Must* and *Can*). *For all m , p , and E , we have:*

- *When p is known to be executed, Can is more precise: $Can^+(p, E) \subseteq Can^\perp(p, E)$;*
- *Everything that must be done can be done: $Must(p, E) \subseteq Can^m(p, E)$;*



How large is the gap between the under- and over-approximations described by *Must* and *Can*? The answer is that they coincide on constructive statements, that is, on statements that can reduce under the constructive semantics. Conversely, whenever they are equal, the statement is constructive.

► **Theorem 9.** *For all p, E , there exists a transition $p \xrightarrow[E]{E', k} p'$ for some E', k and p' if and only if we have $\text{Must}(p, E) = \text{Can}^+(p, E)$.*



► **Remark 10.** This is only true for loop-safe p [4], that is, for statements in which the *Can* function on all loop bodies does not contain 0. This notion could be refined more, but it does not seem very useful.

What about non-constructive statements? The gap can be arbitrarily large in general, for instance if s does not appear in p , then for any event E , $\text{Must}_s((s? p, p) \setminus s, E) = \emptyset$ whereas $\text{Can}_s^+((s? p, p) \setminus s, E) = \text{Can}_s^+(p, E)$. Nevertheless, the *Must* and *Can* functions are well-named, as they indeed describe what must/can be observed, even on non-constructive statements:

► **Lemma 11** (Inclusion between LBS and *Must* and *Can*). *For all p, E, E', k , and p' , if $p \xrightarrow[E]{E', k} p'$ then we have $\text{Must}(p, E) \subseteq (E', \{k\}) \subseteq \text{Can}^+(p, E)$.*



Notice that we use the logical behavioral semantics (LBS) here. If we have $\text{Must}_k(p, E) \neq \emptyset$, then the statement is actually constructive and these inclusions become equalities.

At this point, the reader may wonder how the non-causal programs P1 and P3 mentioned in the introduction are rejected. The technical answer is quite simple: no constructive rule can be applied to them. Why is that? Because of the definitions of *Can* and *Must*. For instance, to evaluate $P1 = s? !s, 0$ when s is a local signal, we need to check whether we can apply the rules for local signals, namely C-sig⁺ or C-sig⁻. To apply C-sig⁺, we have to compute $\text{Must}((s? !s, 0), E * s^\perp)$, which is $(\emptyset, \emptyset)$ as signal s has status $\perp$ and thus, rule C-sig⁺ cannot be applied. To apply C-sig⁻, we need to compute $\text{Can}^+((s? !s, 0), E * s^\perp) = (\{s\}, \{0\})$ and rule C-sig⁻ cannot be applied either. The exact same reasoning applies to $P3 = s? !s, !s$.

The reader may also wonder whether these definitions of *Must* and *Can* accurately represent constructiveness. In our opinion, what justifies it is the direct correspondence with the constructiveness of sets of Boolean equations logically defining digital circuits. More precisely, [40] proves that Boolean constructiveness is equivalent to electrical stability of the result computed by a physical mapping of the equations, under a reasonable gate and wire delay model.

5.3 Refinement between the logical and constructive semantics



The constructive semantics is a refinement of the logical one that precludes some unwanted behaviors due to non-causality. Therefore, we expect to have a theorem stating that any valid transition in the constructive semantics is also valid in the logical one.

In order to state this theorem in Coq, we need to convert constructive events into classical ones. We name C2K the function performing this conversion. Because of the status $\perp$ (unknown), there is no canonical way to do so. Therefore, we choose to restrict the equivalence theorem to *total (constructive) events*, that is, to constructive events E in which no signal is mapped to $\perp$. Thus, the precise statement we prove is the following:

► **Theorem 12.** *For all p, E, E', k, p' , if $\text{Total}(E)$ and $p \xrightarrow[E]{E', k} p'$ then $p \xrightarrow[\text{C2K}(E)]{\text{C2K}(E'), k} p'$.*



► **Coq Remark.** *As we already know that the constructive semantics produces total output events, we could replace $\text{C2K}(E')$ with E' . In Coq, it is still required for typing reasons.*

Proof of Theorem 12. By induction on the proof of $p \xrightarrow[E]{E', k} p'$.

As both semantics are the same except for signal declaration, only the $C\text{-sig}^+$ and $C\text{-sig}^-$ rules are interesting; the other ones are handled using properties of **C2K**. These two cases rely on Lemma 11 above which expresses that the functions *Must* and *Can* indeed compute what their names imply: any signal (resp. completion code) computed by $\text{Must}(p, E)$ is indeed emitted (resp. reached) and any emitted signal or reached completion code indeed belongs to the corresponding set computed by $\text{Can}^+(p, E)$. ◀



6 The Constructive State Semantics

Even though the constructive semantics is well-suited as the starting semantics of Esterel, it is quite far from the circuit one of [4], mostly because the models are very different. On the Esterel side, the semantics transforms the source code into a derivative by keeping only the currently active parts of the program, i.e., what is left to execute. On the contrary, circuits are fixed hardware where only the state of registers can be modified between reactions. To bridge this gap, one can present the constructive semantics in a different way, keeping the source code intact and adding tags on top of it to represent where the execution currently is: these tags correspond to a distributed program counter that precisely encodes the states of the hardware registers, and the program text will be preserved, exactly as in the circuit wiring. This leads to the *constructive state semantics* (CSS) we study now.

Let us consider the following program: $!o_1 ; 1 ; s ? (1 ; !o_2) \mid (!o_3 ; 1), (1 ; !o_4)$. In the constructive semantics, executing it in the environment $E = \{s\}$ would lead to the following execution chain:

$$\begin{aligned} !o_1 ; 1 ; s ? (1 ; !o_2) \mid (!o_3 ; 1), (1 ; !o_4) &\xrightarrow[\{s\}]{\{o_1\}, 1} 0 ; s ? (1 ; !o_2 \mid !o_3 ; 1), (1 ; !o_4) \\ &\xrightarrow[\{s\}]{\{o_3\}, 1} (0 ; !o_2) \mid 0 \xrightarrow[\{s\}]{\{o_2\}, 0} 0 \end{aligned}$$

Here, executed statements keep disappearing until the whole program becomes 0. In the state semantics, we instead keep the program intact and move tags $\hat{\cdot}$ to indicate where execution stops at each reaction. One way to start the overall execution is to add an activated pause $\hat{1}$ in sequence before the program, called the *boot* statement.

$$\begin{aligned} \hat{1} ; !o_1 ; 1 ; s ? (1 ; !o_2) \mid (!o_3 ; 1), (1 ; !o_4) &\xrightarrow[\{s\}]{\{o_1\}, 1} 1 ; !o_1 ; \hat{1} ; s ? (1 ; !o_2) \mid (!o_3 ; 1), (1 ; !o_4) \\ &\xrightarrow[\{s\}]{\{o_3\}, 1} 1 ; !o_1 ; 1 ; s ? (\hat{1} ; !o_2) \mid (!o_3 ; \hat{1}), (1 ; !o_4) \\ &\xrightarrow[\{s\}]{\{o_2\}, 0} 1 ; !o_1 ; 1 ; s ? (1 ; !o_2) \mid (!o_3 ; 1), (1 ; !o_4) \end{aligned}$$

In particular, one can directly observe two crucial invariants of the execution of Esterel statements: **(i)** execution only stops on 1 and @s statements, which are the only statements that can carry a $\hat{\cdot}$ tag; **(ii)** consider the two branches of a presence test or of a sequential composition: only one of them can be executed in a given reaction, unlike for parallel composition $p \mid q$ which has parallel threads. These two properties are enforced by the very definition of states below.

6.1 Formal definition of the state semantics

Formally, states (written $\widehat{p}, \widehat{q}$) are defined by the following grammar:



$$\begin{array}{l} \text{States} \quad \widehat{p}, \widehat{q} := \\ \quad \widehat{1} \\ \quad \widehat{@s} \\ \quad s ? \widehat{p}, q \quad s ? p, \widehat{q} \\ \quad s \supset \widehat{p} \\ \quad \{\widehat{p}\} \\ \quad \uparrow \widehat{p} \\ \quad \widehat{p}; q \quad p; \widehat{q} \\ \quad \widehat{p} * \\ \quad \widehat{p} | q \quad p | \widehat{q} \quad \widehat{p} | \widehat{q} \\ \quad \widehat{p} \setminus s \end{array}$$

The only elementary states are the active $\widehat{1}$ (**pause**) and the active $\widehat{@s}$ (**await immediate s**) statements, written $\widehat{1}$ and $\widehat{@s}$, and states propagate through all other compound statements of the language. The decorated statements $\widehat{1}$ and $\widehat{@s}$ represent the points where execution should restart in the next instant, in other words, the aforementioned distributed program counter since execution always stops only on “1” or “@s” statements in Kernel Esterel. For example, the states of “1; 1” are “ $\widehat{1}; 1$ ” and “1; $\widehat{1}$ ”, never “ $\widehat{1}; \widehat{1}$ ” as two states cannot be in sequence; the only state of “ $!(s_1 | 1); !s_2$ ” is “ $!(s_1 | \widehat{1}); !s_2$ ” as there is only one occurrence of 1 or @s. The **ABR0i** program of Section 2.2 has four possible states: waiting on both signals A and B , waiting on only one of them, and waiting on the **halt** ($1*$) statement.

The constructive state semantics is a rewording of the constructive semantics where we replace the derivative statement p' inside the rule $p \xrightarrow[E]{E', k} p'$ by a derivative *term* $\overline{p'}$. Such terms $\overline{p}, \overline{q}$ are either active states $\widehat{p}, \widehat{q}$ or inactive statements p, q . When a derivative term $\overline{p'}$ is a state $\widehat{p'}$, it means that execution has paused within $\widehat{p'}$ for the current reaction and can be resumed later. On the contrary, when this term $\overline{p'}$ is a statement p' , it means that execution is over, either by normal termination or raising a trap. Technically, the state semantics is still a rewriting semantics where the tags are moved around, but the core idea is that *the underlying statement never changes*. Execution of an instant starts with p and yields a term $\overline{p}$. From one reaction to the next, one resumes from $\widehat{p}$ but not from p itself, so that there is no implicit toplevel loop restarting p .



Tagged statements (states) are called *active*, and pauses are *activated* in a reaction if they are executed and not killed until the end of the reaction. Because Esterel has parallelism, there can be multiple active and activated statements inside a program: all active “1”s and “@s”s are used to resume execution and some “1”s and “@s”s are activated during the execution (possibly the same ones).

There are two kinds of state semantics: the *start* semantics for untagged statements used when we start execution and the *resumption* semantics for active states used when we resume execution. We distinguish them by subscripts.

$$\text{Start semantics} \quad p \xrightarrow[E]{E', k} \overline{p'} \quad \text{Resumption semantics} \quad \widehat{p} \xrightarrow[E]{E', k} \overline{p'}$$

To define these constructive state semantics, we need to extend *Must* and *Can* to states. Following [4], we define the expansion $\mathcal{E}(\widehat{p})$ of a state $\widehat{p}$ into a statement by erasing the already executed parts, thus translating $\widehat{p}$ into its corresponding statement in the constructive semantics.

$$\begin{array}{lll}
\mathcal{E}(\widehat{1}) := 0 & \mathcal{E}(\widehat{p}*) := \mathcal{E}(\widehat{p}); p* & \mathcal{E}(\widehat{p} \mid \widehat{q}) := \mathcal{E}(\widehat{p}) \mid \mathcal{E}(\widehat{q}) \\
\mathcal{E}(\widehat{@s}) := @s & \mathcal{E}(\{\widehat{p}\}) := \{\mathcal{E}(\widehat{p})\} & \mathcal{E}(\widehat{p} \mid q) := \mathcal{E}(\widehat{p}) \mid 0 \\
\mathcal{E}(s? \widehat{p}, q) := \mathcal{E}(\widehat{p}) & \mathcal{E}(\uparrow \widehat{p}) := \uparrow \mathcal{E}(\widehat{p}) & \mathcal{E}(p \mid \widehat{q}) := 0 \mid \mathcal{E}(\widehat{q}) \\
\mathcal{E}(s? p, \widehat{q}) := \mathcal{E}(\widehat{q}) & \mathcal{E}(\widehat{p}; q) := \mathcal{E}(\widehat{p}); q & \mathcal{E}(\widehat{p} \setminus s) := \mathcal{E}(\widehat{p}) \setminus s \\
\mathcal{E}(s \supset \widehat{p}) := @ \neg s; s \supset \mathcal{E}(\widehat{p}) & \mathcal{E}(p; \widehat{q}) := \mathcal{E}(\widehat{q}) &
\end{array}$$

Recalling that a term $\bar{p}$ is either an active state $\widehat{p}$ or an inert statement p , this function is extended to terms $\bar{p}$ by setting $\mathcal{E}(p) = 0$: in an inactive statement, nothing is pending execution.

► **Remark 13.** For the $\widehat{p} \mid q$ and $p \mid \widehat{q}$ cases, keeping the seemingly useless $0 \mid _$ is actually technically useful to avoid a mismatch between the derivatives of both semantics. Indeed, this simplifies the equivalence between the constructive and state semantics: if we instead set $\mathcal{E}(\widehat{p} \mid q) = \mathcal{E}(\widehat{p})$, the parallel statement disappears and we would need to prove that $p \mid 0$ and p are equivalent, which would require introducing bisimulation.

With this expansion function $\mathcal{E}$, we let $Must(\widehat{p}, E)$ be $Must(\mathcal{E}(\widehat{p}), E)$ and similarly for Can , so that we can directly reuse all the already proved properties about $Must$ and Can and have for free the following equivalences:



$$s \in Must_s(\widehat{p}, E) \iff s \in Must_s(\mathcal{E}(\widehat{p}), E) \quad s \in Can_s^m(\widehat{p}, E) \iff s \in Can_s^m(\mathcal{E}(\widehat{p}), E)$$

The rules of the state semantics are given in Figure 5 for the start semantics rules and in Figure 6 for the resumption semantics rules.

► **Remark 14.** The transformation done to turn the constructive semantics into the constructive state semantics can also be done on the logical semantics, yielding a logical state semantics as done in [4]. The only difference with the constructive version would be the local signal rules.

We illustrate the execution of the **ABR0i** program in the CSS semantics in Figure 7.

Finally, we can now prove that our initial idea of preserving the underlying statement actually works. Let us define an erasing function $\mathcal{B}$ that converts a term into its untagged underlying statement, which amounts to erasing active tags by turning all active “ $\widehat{1}$ ”s and “ $\widehat{@s}$ ”s into inactive ones. Then, the following lemma holds:

► **Lemma 15** (Invariance of the base statement). *For all p, E, E', k, p' , if $p \xrightarrow[E]{E', k} \bar{p}'$ then*

$$p = \mathcal{B}(\bar{p}').$$

For all $\widehat{p}, E, E', k, p'$, if $\widehat{p} \xrightarrow[E]{E', k} \bar{p}'$ then $\mathcal{B}(\widehat{p}) = \mathcal{B}(\bar{p}')$.



Being a rewording of the constructive semantics, the state semantics enjoys the same properties, in particular the state start and resumption semantics are deterministic.

► **Coq Remark** (The constructive state semantics in Coq). *The functions $\mathcal{E}$ and $\mathcal{B}$ are written respectively **expand** and **base**. Furthermore, in order to have a more self-contained definition of the state semantics, the Coq development defines the $Must$ and Can functions on states from scratch. Then, the equivalences $s \in Must_s(\widehat{p}, E) \iff s \in Must_s(\mathcal{E}(\widehat{p}), E)$ and $s \in Can_s^m(\widehat{p}, E) \iff s \in Can_s^m(\mathcal{E}(\widehat{p}), E)$ are proved rather than used as definitions. From these equivalences, the properties of $Must$ and Can on states are imported from the versions on statements.*

$$\begin{array}{c}
\frac{}{0 \xrightarrow[E]{\emptyset, 0} 0} \text{ nothing} \qquad \frac{}{1 \xrightarrow[E]{\emptyset, 1} \widehat{1}} \text{ pause} \qquad \frac{k \geq 2}{k \xrightarrow[E]{\emptyset, k} k} \text{ exit} \\
\\
\frac{}{!s \xrightarrow[E]{\{s^+\}, 0} !s} \text{ emit} \\
\\
\frac{s^+ \in E}{@s \xrightarrow[E]{\emptyset, 0} @s} \text{ awimm}^+ \qquad \frac{s^- \in E}{@s \xrightarrow[E]{\emptyset, 1} \widehat{@s}} \text{ awimm}^- \\
\\
\frac{s^+ \in E \quad p \xrightarrow[E]{E', k} \overline{p'}}{s ? p, q \xrightarrow[E]{E', k} s ? \overline{p'}, q} \text{ then} \qquad \frac{s^- \in E \quad q \xrightarrow[E]{E', k} \overline{q'}}{s ? p, q \xrightarrow[E]{E', k} s ? p, \overline{q'}} \text{ else} \\
\\
\frac{p \xrightarrow[E]{E', k} \overline{p'}}{s \supset p \xrightarrow[E]{E', k} s \supset \overline{p'}} \text{ suspend} \\
\\
\frac{k \neq 0 \quad p \xrightarrow[E]{E', k} \overline{p'}}{p * \xrightarrow[E]{E', k} \overline{p'} *} \text{ loop} \\
\\
\frac{p \xrightarrow[E]{E', k} \overline{p'}}{\{p\} \xrightarrow[E]{E', \downarrow k} \{\overline{p'}\}} \text{ trap} \qquad \frac{p \xrightarrow[E]{E', k} \overline{p'}}{\uparrow p \xrightarrow[E]{E', \uparrow k} \uparrow \overline{p'}} \text{ shift} \\
\\
\frac{k \neq 0 \quad p \xrightarrow[E]{E', k} \overline{p'}}{p ; q \xrightarrow[E]{E', k} \overline{p'} ; q} \text{ seq}_k \qquad \frac{p \xrightarrow[E]{E'_p, 0} p \quad q \xrightarrow[E]{E'_q, k} \overline{q'}}{p ; q \xrightarrow[E]{E'_p \cup E'_q, k} \overline{p'} ; \overline{q'}} \text{ seq}_0 \\
\\
\frac{p \xrightarrow[E]{E'_p, k_p} \overline{p'} \quad q \xrightarrow[E]{E'_q, k_q} \overline{q'}}{p \mid q \xrightarrow[E]{E'_p \cup E'_q, \max(k_p, k_q)} \overline{p'} \mid \overline{q'}} \text{ parallel} \\
\\
\frac{s \in \text{Must}(p, E * s^\perp) \quad p \xrightarrow[E * s^+]{E', k} \overline{p'}}{p \setminus s \xrightarrow[E]{E' \setminus s, k} \overline{p'} \setminus s} \text{ C-sig}^+ \qquad \frac{s \notin \text{Can}^+(p, E * s^\perp) \quad p \xrightarrow[E * s^-]{E', k} \overline{p'}}{p \setminus s \xrightarrow[E]{E' \setminus s, k} \overline{p'} \setminus s} \text{ C-sig}^-
\end{array}$$

■ **Figure 5** Start rules of the state semantics.

$$\begin{array}{c}
 \frac{}{\widehat{1} \xrightarrow[E]{\emptyset, 0} 1} \text{ pause} \\
 \\
 \frac{\frac{s^+ \in E}{\widehat{\text{@}}s \xrightarrow[E]{\emptyset, 0} \widehat{\text{@}}s} \text{ awimm}^+}{\widehat{p} \xrightarrow[E]{E', k} \overline{p'}} \text{ then} \quad \frac{\frac{s^- \in E}{\widehat{\text{@}}s \xrightarrow[E]{\emptyset, 1} \widehat{\text{@}}s} \text{ awimm}^-}{\widehat{q} \xrightarrow[E]{E', k} \overline{q'}} \text{ else} \\
 \frac{}{s ? \widehat{p}, q \xrightarrow[E]{E', k} s ? \overline{p'}, q} \quad \frac{}{s ? p, \widehat{q} \xrightarrow[E]{E', k} s ? p, \overline{q'}} \\
 \\
 \frac{s^+ \in E}{s \supset \widehat{p} \xrightarrow[E]{\emptyset, 1} s \supset \widehat{p}} \text{ suspend}^+ \quad \frac{s^- \in E \quad \widehat{p} \xrightarrow[E]{E', k} \overline{p'}}{s \supset \widehat{p} \xrightarrow[E]{E', k} s \supset \overline{p'}} \text{ suspend}^- \\
 \\
 \frac{\widehat{p} \xrightarrow[E]{E', k} \overline{p'} \quad k \neq 0}{\widehat{p} * \xrightarrow[E]{E', k} \overline{p'} *} \text{ loop}_k \quad \frac{\widehat{p} \xrightarrow[E]{E', 0} p \quad p \xrightarrow[E]{E', k} \overline{p'} \quad k \neq 0}{\widehat{p} * \xrightarrow[E]{E', k} \overline{p'} *} \text{ loop}_0 \\
 \\
 \frac{\widehat{p} \xrightarrow[E]{E', k} \overline{p'}}{\{\widehat{p}\} \xrightarrow[E]{E', \downarrow k} \{\overline{p'}\}} \text{ trap} \quad \frac{\widehat{p} \xrightarrow[E]{E', k} \overline{p'}}{\uparrow \widehat{p} \xrightarrow[E]{E', \uparrow k} \uparrow \overline{p'}} \text{ shift} \\
 \\
 \frac{\widehat{p} \xrightarrow[E]{E', k} \overline{p'} \quad k \neq 0}{\widehat{p}; q \xrightarrow[E]{E', k} \overline{p'}; q} \text{ seq}_k \quad \frac{\widehat{p} \xrightarrow[E]{E'_p, 0} p \quad q \xrightarrow[E]{E'_q, k} \overline{q'}}{\widehat{p}; q \xrightarrow[E]{E'_p \cup E'_q, k} p; \overline{q'}} \text{ seq}_0 \quad \frac{\widehat{q} \xrightarrow[E]{E'_q, k} \overline{q'}}{p; \widehat{q} \xrightarrow[E]{E'_q, k} p; \overline{q'}} \text{ seq}_q \\
 \\
 \frac{\widehat{p} \xrightarrow[E]{E'_p, k_p} \overline{p'} \quad \widehat{q} \xrightarrow[E]{E'_q, k_q} \overline{q'}}{\widehat{p} | \widehat{q} \xrightarrow[E]{E'_p \cup E'_q, \max(k_p, k_q)} \overline{p'} | \overline{q'}} \text{ both} \\
 \\
 \frac{\widehat{p} \xrightarrow[E]{E'_p, k_p} \overline{p'}}{\widehat{p} | q \xrightarrow[E]{E'_p, k_p} \overline{p'} | q} \text{ left} \quad \frac{\widehat{q} \xrightarrow[E]{E'_q, k_q} \overline{q'}}{p | \widehat{q} \xrightarrow[E]{E'_q, k_q} p | \overline{q'}} \text{ right} \\
 \\
 \frac{s \in \text{Must}(\mathcal{E}(p), E * s^\perp) \quad \widehat{p} \xrightarrow[E * s^+]{E', k} \overline{p'}}{\widehat{p} \setminus s \xrightarrow[E]{E' \setminus s, k} \overline{p'} \setminus s} \text{ C-sig}^+ \\
 \\
 \frac{s \notin \text{Can}^+(\mathcal{E}(p), E * s^\perp) \quad \widehat{p} \xrightarrow[E * s^-]{E', k} \overline{p'}}{\widehat{p} \setminus s \xrightarrow[E]{E' \setminus s, k} \overline{p'} \setminus s} \text{ C-sig}^-
 \end{array}$$

 ■ **Figure 6** Resumption rules of the state semantics.

$$\begin{aligned}
\widehat{1}; \text{ABR0i} &= \widehat{1}; (\{ (1; (R?2, 1) *) \mid (((@A \mid @B); !O; (1 *)); 2) \}) * \\
&\xrightarrow[\{B\}]{\emptyset, 1} \widehat{1}; (\{ (\widehat{1}; (R?2, 1) *) \mid (((@A \mid @B); !O; (1 *)); 2) \}) * \\
&\xrightarrow[\{A, B\}]{\{O\}, 1} \widehat{1}; (\{ (1; (R?2, \widehat{1}) *) \mid (((@A \mid @B); !O; (\widehat{1} *)); 2) \}) * \\
&\xrightarrow[\{B\}]{\emptyset, 1} \widehat{1}; (\{ (1; (R?2, \widehat{1}) *) \mid (((@A \mid @B); !O; (\widehat{1} *)); 2) \}) * \\
&\xrightarrow[\{R\}]{\emptyset, 1} \widehat{1}; (\{ (\widehat{1}; (R?2, 1) *) \mid (((@A \mid @B); !O; (1 *)); 2) \}) * \\
&\xrightarrow[\{A, B, R\}]{\{O\}, 1} \widehat{1}; (\{ (\widehat{1}; (R?2, 1) *) \mid (((@A \mid @B); !O; (\widehat{1} *)); 2) \}) *
\end{aligned}$$

■ **Figure 7** Execution of ABR0i in the CSS.

6.2 Equivalence between the constructive behavioral semantics and the constructive state semantics

To compare the constructive semantics and the state one, we need the expansion function $\mathcal{E}$ to relate the meaning of the derivatives of both semantics. Then, we can state the equivalence between the constructive behavioral semantics and the constructive state semantics as follows.

► **Theorem 16** (Equivalence between the constructive and state semantics).

$$\text{For all } p, E, E', k, \overline{p'}, \quad p \xrightarrow[E]{E', k} \overline{p'} \implies p \xrightarrow[E]{E', k} \mathcal{E}(\overline{p'})$$

$$\text{For all } \widehat{p}, E, E', k, \overline{p'}, \quad \widehat{p} \xrightarrow[E]{E', k} \overline{p'} \implies \mathcal{E}(\widehat{p}) \xrightarrow[E]{E', k} \mathcal{E}(\overline{p'})$$

$$\text{For all } p, E, E', k, p', \quad p \xrightarrow[E]{E', k} p' \implies \exists \overline{p''}, p \xrightarrow[E]{E', k} \overline{p''} \wedge p' = \mathcal{E}(\overline{p''})$$

$$\text{For all } \widehat{p}, E, E', k, p', \quad \mathcal{E}(\widehat{p}) \xrightarrow[E]{E', k} p' \implies \exists \overline{p''}, \widehat{p} \xrightarrow[E]{E', k} \overline{p''} \wedge p' = \mathcal{E}(\overline{p''})$$

Proof. The proofs of these four implications all go by induction on the derivation tree in the source semantics. They are straightforward, they simply require to match the source semantics rules by case distinction on p' and k . ◀

► **Coq Remark.** The actual Coq statement and proof are slightly different because we need to insert $\delta(k, _)$ and use the fact that $\delta(k, \delta(k, p)) = \delta(k, p)$.

6.3 Going beyond a state

In the circuit translation of [4], the state semantics provides a perfect match between states $\widehat{p}$ and active circuit states at the end of a clock cycle, that is, those in which at least one register is 1. We can extend this correspondence to untagged statements p and inactive circuits. In particular, the registers in the circuit exactly correspond to 1 (**pause**) and @s (**await immediate s**) statements.⁵ Nevertheless, the computation of semantics states is abstract: we go from one state to the next, without explaining precisely how information flows between these states. Furthermore, the state

⁵ Technically, the circuit translation of $s \supset p$ also contains a register (the @¬s appearing in the CBS rule). We do not add it explicitly to the state semantics because it is only active whenever p is.

semantics still uses the *Must* and *Can* functions recursively to compute the statuses of local signals. This is very different from the computation of wire values in digital circuits: instead of performing a two-step recursive evaluation using *Must/Can*, a digital circuit propagates electrical values in a forward way, following causality chains.

To solve these issues, we now introduce the *microstep semantics*, detailing how computation works *within* an instant as a *sequence* of microsteps, implementing a progressive evolution from one state to the next. This is the semantic equivalent of the propagation of electrical fronts in a digital circuit, which is internally asynchronous but where each step and the full reaction have predictable time and result.



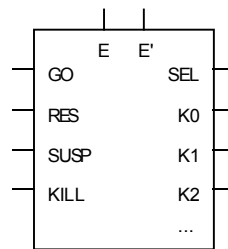
7 Microstep Semantics

With the microstep semantics, our goal is to have an Esterel semantics as precise as the generated circuit, meaning that its resolution should be (roughly) at the gate level, and that it should avoid the mutually recursive definitions of *Must* and *Can*. Furthermore, we want it to be defined directly on the source code, so that the equivalence with the previous semantics is easier to express. Because of that, like the state semantics, the microstep semantics tags Esterel programs and execution consists in moving tags. Like synchronous digital circuits in which electricity propagates within a clock cycle, we restrict the microstep semantics to a single reaction, meaning that there will be no rule to move past an electrical register and that execution is performed starting either from the boot of the generated circuit (in the starting instant) or from active 1 (*pause*) or *@s* (*await immediate s*) statements (in resumption instants) to either termination or reaching an activated 1 or *@s* statement. This matches the Register Transfer Level (RTL) evaluation in circuit design.

7.1 Microstate Definition

7.1.1 Relation with the circuit translation

The core idea in designing the tags propagated by the microsteps is taken from a simplification of the circuit translation of [4]. Actually, the microstep semantics can be seen as a simulation of this circuit translation on top of the Kernel Esterel statements. As Kernel Esterel does not feature data, the key point is to transfer control. In the circuit translation, this is done through a structural translation respecting the circuit interface depicted in Figure 8.



■ **Figure 8** Circuit Translation Interface, taken from [4].

This interface has:

- input and output events for signals (E and E' respectively);
- four input wires (Go, Res, Susp, Kill);
- a finite number of output wires (Sel and the K_i).

All are single wires except for E and E' which are bundles of wires (one per signal). Their meaning is as follows:

- Go is set to start the execution of the (currently inactive) circuit;
- Res (for Resume) is set to continue execution of the (currently active) circuit;
- Susp (for Suspend) is set to freeze the circuit registers for the current instant, but the combinational logic is still executed, in particular signals may be emitted;
- Kill is used to reset the circuit into its inactive state, that is, reset all registers to 0;
- Sel represents whether the circuit is active or not, that is, whether at least one of its registers has value 1;
- the K_i represent the completion code in one-hot encoding: completion code k is represented by wire K_k being one and all other wires K_i ($i \neq k$) being zero.

Looking ahead at Figure 11 (p. 34), we can see that among these wires from the circuit translation of [4], only Go and Res are used to propagate control within an instant. Since they only determine the next values of registers, Susp and Kill do not interact with the rest of control propagation and can be treated independently. As our microstep semantics aims at explaining the execution *within* one instant, we can omit Susp and Kill for simplicity. Their effect is instead obtained through the use of the auxiliary function to_term (see Section 7.2).

The Sel wire defines which parts of a circuit are active, implementing the $\hat{\cdot}$ tag of the state semantics. Since its value does not evolve during an instant, it can be computed once at the beginning of the instant, before the circuit's control and signal inputs are set. This should allow us to remove it from the microstep semantics. Unfortunately, we must keep it because it is used in the control propagation of $s \supset \hat{p}$ (see Figure 11c, p. 34). As for Susp and Kill, the computation of Sel is performed in our microstep semantics by auxiliary functions from_stmt and from_state , which create a microstate from a statement or a state (see Section 7.2). Thus, the reader may think of Sel as a constant input and we keep it separate from the input and output colors (see Figure 9), which evolve within an instant.

This analysis allowing us to remove the computation of Susp, Kill and Sel is done in order to simplify the microstep semantics and reduce the number of SOS rules. Nevertheless, it is by no means essential: one can certainly design a microstep semantics which keeps them, at the cost of more rules.

Remember that a statement can only return a single integer completion code; here it is technically simpler to encode this integer by the one-hot encoding represented by the K_i . This is a classical technique in circuits.

7.1.2 Microstate Interface

From the previous analysis, we design a *microstate interface* to annotate the source command p : the notation is $\blacksquare p \bullet$, defined as follows.



Inputs $\blacksquare$ consist in the Go and Res wires (as constructive Booleans that can take values $\perp$, $+$, or $-$); signals are read from an (external) event E but no output event E' is produced (instead it will be read from the evaluated microstate); outputs $\bullet$ consist in the completion code represented as either $\bullet_k$ with k an integer, meaning that wire K_k is 1 and all other wires are 0, or $\circ_K$ with K a finite set of integers, meaning that wires K_i with $i \in K$ are $\perp$ and all other wires are 0 (and in particular, no wire is 1). This completion code representation directly denotes the 1-hot encoding, thus avoiding maintaining it as an invariant or dealing with impossible cases.

In fact, $\bullet_k$ represents the fact that the k -th wire is $+$ (*Must* propagation) whereas $\circ_K$ represents a finite set K of completion code wires whose value is still unknown, the others being $-$ (*Can* propagation). The information about Sel is assumed to be already computed so we do not need to consider the $\perp$ case and can use a simple Boolean.

These $\blacksquare$ and $\bullet$ annotations are called *colors* and are recursively inserted into the bodies and branches of compound microstates. For example, $s \supset p$ becomes $\blacksquare(s \supset (\blacksquare p \bullet)) \bullet$. We usually drop parentheses when ambiguity can be resolved from the context. The formal definition of microstates is given in Figure 9.

sel	$:=$	$+$	$ $	$-$	sel Boolean		
cb	$:=$	$\perp$	$ $	$+$	$ $	$-$	constructive Boolean
$\blacksquare$	$:=$	$\{Go : cb ; Res : cb\}$				input color	
$\bullet$	$:=$					output color	
	$ $	$\bullet_k$	$k \in \mathbb{N}$		k -th wire is 1		
	$ $	$\circ_K$	$K \subset_{fin} \mathbb{N}$		wires outside K are 0		
$mstate$	$:=$	sel	$\blacksquare$	$mstmt$	$\bullet$	microstate	
$mstmt$	$:=$	0					
	$ $	1					
	$ $	$!s$	s signal				
	$ $	$@s$	s signal				
	$ $	$s ? mstate, mstate$	s signal				
	$ $	$s \supset mstate$	s signal				
	$ $	$\{mstate\}$					
	$ $	k	$k \geq 2$				
	$ $	$\uparrow mstate$					
	$ $	$mstate ; mstate$					
	$ $	$mstate *$					
	$ $	$mstate mstate$					
	$ $	$mstate \setminus s$	s signal				

■ **Figure 9** Definition of microstates.

7.1.3 Notations

We use the $\circ_{K \setminus i}$ notation to denote setting wire i to 0 inside $\circ_K$ instead of the more standard but cumbersome $\circ_{K \setminus \{i\}}$. We abbreviate $val = b$ as val^b , both when used as an equality or as an assignment. We let $in(p)$, $out(p)$, $Go(p)$, and $Res(p)$ denote respectively the input color, output color, Go wire, and Res wire of a microstate p ; and let $Go(\blacksquare)$ and $Res(\blacksquare)$ denote the Go and Res wires of an input color $\blacksquare$. To change the input color of a microstate p into $\blacksquare$, we write $[\blacksquare]p$. We extend this notation to single wires: $[Go^+]p$, $[Res(q)]p$. For a signal s , we define $E(s)$ to be the value of s in E if $s \in \text{dom}(E)$ and $\perp$ otherwise.

We color inputs as follows to have a visual representation of the various values of Go, Res and Sel (whether the square is filled or not).

Colors	Sel	Go	Res	Intuition
■	?	?	?	We know nothing
□	−	?	?	We know only Sel^-
□ ⁺	−	+	?	Starting an inactive statement
□ [−]	−	−	?	Keeping inactive an inactive statement
■	+	?	?	We know only Sel^+
■ ⁺	+	?	+	Resuming an active statement
■ [−]	+	?	−	Not executing an active statement

When we have Sel^- (that is, in the □, □⁺, and □[−] cases), Theorem 24 will formally justify that the value of Res is irrelevant. Similarly, an invariant of the circuit translation, called *input_invariant*($Sel, \blacksquare$) and defined as $Go(\blacksquare)^+ \implies Sel^-$, ensures that Sel and Go can never be true simultaneously.⁶ Intuitively, only inert statements can be started, active states are resumed instead. Thus, the “?”s for Go in the last three cases are actually either $\perp$ or $-$ but they cannot be $+$. Nevertheless, even for active microstates (having Sel^+) for which Go can only become Go^- , having Go^- still provide more information than $Go^\perp$ and may be necessary for some microstates. For instance, in the microstate built from $\hat{p} \mid q$, even though the overall statement has Sel^+ , we need Go^- to know that q is not executed and that its completion code will become $\circ_\emptyset$.

We define two colored input notations as convenient shorthands for several cases:

- ⁺: either □⁺ or ■⁺, that is, $Go^+ \vee (Sel^+ \wedge Res^+)$

The intuition is that the microstate is executed, either started fresh (if inactive) or resumed (if active). The input invariant expresses that Go^+ entails Sel^- , so that we do not need to add Sel^- in the left disjunct.

- [−]: either □[−] or ■[−], that is, $Go^- \wedge (Sel^- \vee Res^-)$

The intuition is that the microstate is not executed: neither started nor resumed. This definition is a bit more restrictive than $(Go^- \wedge Sel^-) \vee (Res^- \wedge Sel^+)$, which is the direct translation of □[−] or ■[−], because we also require Go^- in the Sel^+ case. Nevertheless, in that case we know that Go can only become Go^- and having Go^- may be useful as discussed previously.

7.2 Intuition of the Microstep Semantics

Synchronous digital circuits work by propagating electric potentials through wires and logical gates. Here, we only consider stabilizing synchronous digital circuits where all wires get a stable 0 or 1 value in finite time. They are exactly characterized as computable by constructive logic by Michael Mendler in [40]. Similarly, our microstep semantics works by propagating control information through Esterel statements until reaching some maximal state of information, that is, a state where all Go and Res wires are no longer $\perp$ and where all completion codes are either $\circ_\emptyset$ (not executed, thus no completion code) or $\bullet_k$ for some integer k (executed and returning completion code k). After the microstep execution chain completes, the resulting state $\hat{p}'$ can be read from this maximal state of information. This state is unique as we prove the microstep semantics to be confluent (Theorem 21, p. 42). To measure information and account for its propagation, we use a Scott ordering that we define now.

⁶ This requires careful handling of loops to maintain. For instance, the circuit translation of $1*$ will have both Sel and Go wires set to 1 after the first cycle. As the Coq formalization currently does not handle loops, this issue does not arise in the code accompanying this article.



Scott ordering on microstates



The Scott order $p \leq q$ between two microstates p and q intuitively means that p contains no more information (is not more defined) than q . It is defined recursively, by requiring that p and q have the same base statement and Sel values and that any input or output color in p is no larger (contains no more information) than the corresponding one in q . On input colors, we use a component-wise ordering of the constructive Booleans Go and Res (that is, $\perp < +$ and $\perp < -$).

The definition of Scott ordering on output colors stems from the intuition that $\bullet_k$ and $\circ_K$ represent respectively the fact that *Must* is k and *Can* is K .

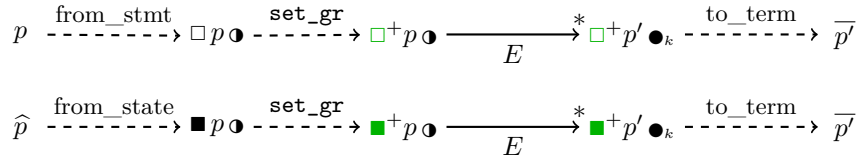
$\circ_K \leq \circ_L$	$:=$	$L \subseteq K$	with more information, more wires are set to 0
$\circ_K \leq \bullet_k$	$:=$	$k \in K$	the wire set to 1 must be among the ones that are not 0
$\bullet_k \leq \bullet_l$	$:=$	$k = l$	only a single wire can be 1
$\bullet_k \leq \circ_K$	$:=$	False	$\bullet_k$ cannot contain less information than $\circ_K$

We require the Sel values to be the same to enforce that $p \leq q$ entails that p and q have the same starting term, in particular the same base statement. Indeed, on microstates from different terms or different instants, the information available in both has no reason to be compatible and the comparison is not meaningful.

We define a *total* microstate as a microstate where the information is complete, that is, no input color still contains $\perp$ and all output colors are either $\circ_\emptyset$ or $\bullet_k$ for some k . In terms of circuits, this corresponds to having all wires stabilized to 0 or 1.

Connection between states and microstates

The microstep semantics can be related to the state semantics as follows: starting from a term $\bar{p}$ (that is, an inactive statement p or an active state $\hat{p}$) we build a starting microstate representing the state of computation at the beginning of the instant. Then, after setting its input color, we let the microstep semantics make this microstate evolve until reaching maximal information. Finally, we convert the final microstate back into a term. This is shown on Figure 10, where dashed arrows are function application and full arrows represent the microstep semantics.



■ **Figure 10** Links between the state and microstep semantics.



The difference between `from_stmt` and `from_state` only lies in how Sel is computed: in `from_stmt`, it is always Sel^- whereas in `from_state` the active parts of a state $\hat{p}$ have Sel^+ . In particular, the overall structure is the same (the one of p) and the input and output colors are identical. These output colors represent the statically-computed possible completion codes for each statement and sub-statement and thus, give the number of completion wires in the final circuit. Notice that the `from_stmt` and `from_state` functions perform the computation of the Sel wire.

The `to_term` function converts a microstate into a term. It is well-defined only for microstates with maximal information, since otherwise we do not know which parts are active or not. For example, $\square^+ \text{pause} \bullet_1$ should become $\hat{1}$ whereas $\square^- \text{pause} \circ_\emptyset$ should become 1, so that we cannot translate $\blacksquare \text{pause} \bullet$ in a meaningful way. This function also performs the effect of the `Susp` and

Kill wires of [4] by either freezing subterms (that is, reverting back to the state at the beginning of the instant) or killing a pausing parallel branch (that is, replacing it with its base statement) whenever the other branch raises an exit.

► **Remark 17.** As a microstate does not contain information about suspension, the `to_term` function must take the input event E in order to decide whether a suspension $s \supset p$ is triggered or not. A different solution could be to add a `Susp` component to the control part of input colors, to track the value of `Susp` in the circuit translation of [4].

Unlike the previous semantics, the microstep semantics produces neither a completion code k nor an output event E' . Although this would be technically possible, it is of little practical interest because microsteps are too fine-grained. Indeed, in most microstep rules, the output event and the completion code are undefined, that is, no signal would be emitted and the completion code would be $\perp$. Furthermore, the completion code k and the output event E' can be read off the final microstate. The completion code is simply given by the output color and the output event is computed by a function `to_event` by scanning the `emit` statements of the microstate:

- (i) if there is an executed emitter of s , that is, a subterm $\square^+!s \bullet_0$, s is emitted so we set s 's status to $+$;
- (ii) if all emitters of s are not executed, that is, they are all $\square^-!s \circ_\emptyset$, s is not emitted and its status is $-$;
- (iii) otherwise, some emitter of s is neither executed nor not executed, that is, it is neither $\square^+!s \bullet_0$ nor $\square^-!s \circ_\emptyset$ (hence it is $\square!s \circ_{\{0\}}$), and the status of s is $\perp$.

To distinguish these three cases, we can ignore input colors and only look at output colors: $\bullet_0$ means case (i); $\circ_\emptyset$ means case (ii); $\circ_{\{0\}}$ means case (iii). The `to_event` function can be applied to any microstate, not only final and total ones. In particular, it is used to compute the status of local signals (see Section 7.3.2.7), thus replacing the *Must* and *Can* functions.

7.3 Formal Definition of the Microstep Semantics

This long and technical section details the 51 rules of the microstep semantics and their meaning. We give in Figure 11 the circuit translation of [4], as it is a strong inspiration for the microstep rules presented in this section.

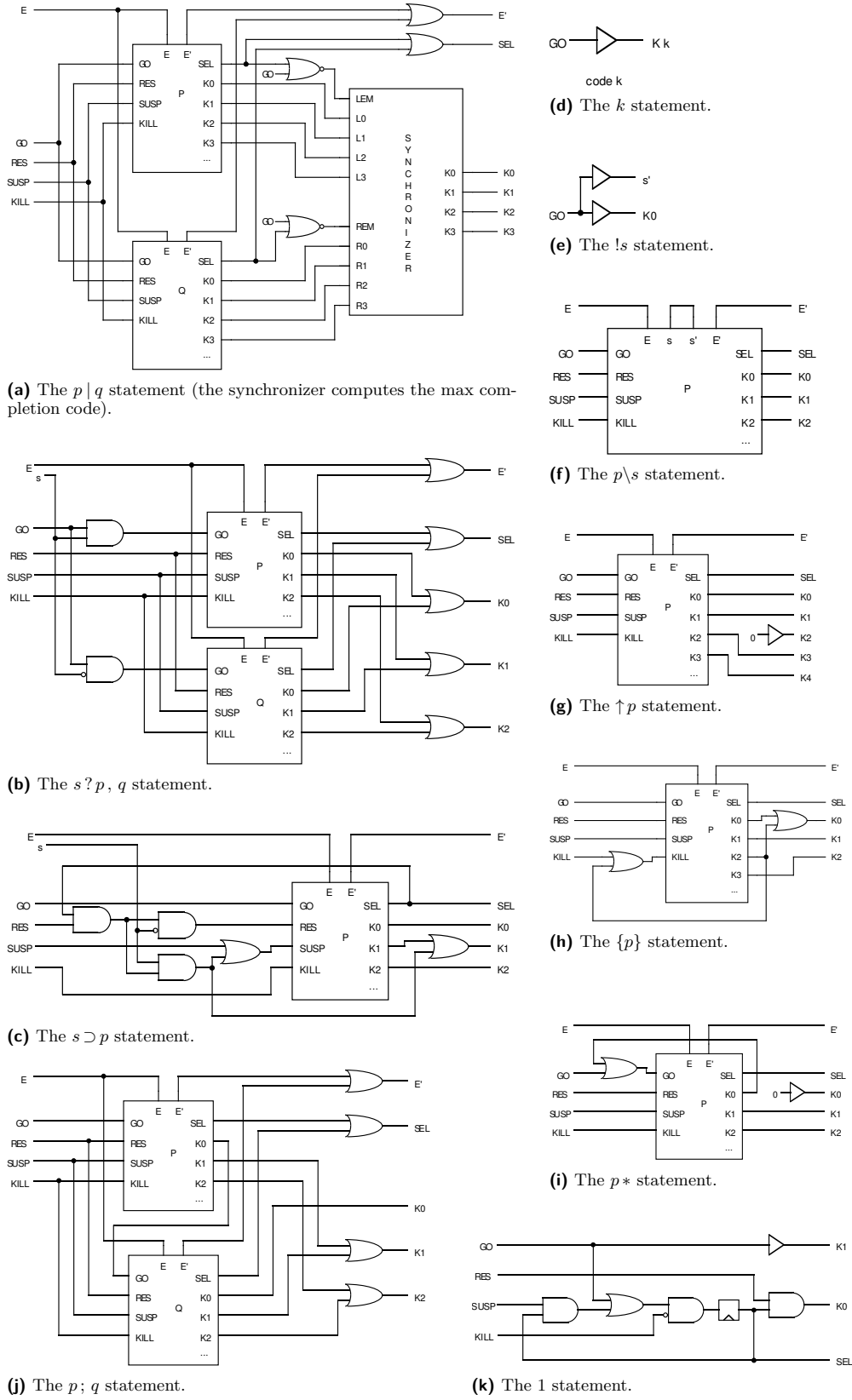
7.3.1 Elementary microstates

The statements 0 , 1 , $!s$, k and $@s$ only have input and output colors, and no sub-statement. Thus, their microstep rules simply compute their output color depending on their input color and, for $@s$, the status of s .

7.3.1.1 The 0 , k , $!s$ rules

These statements being instantaneous, we know that `Sel` is always false and that in the circuit translation only the `Go` wire is useful (and actually represented). Thus, the only possible input colors are $\square$, $\square^+$ and $\square^-$. If the statement is started (that is, the input is $\square^+$), we execute the statement. If the statement is not started (that is, the input is $\square^-$), we do not execute it. Otherwise, the statement is neither executed nor not executed (that is, the input is $\square$ or, in the circuit translation, `Go` is $\perp$) and no rule applies.

$$\begin{array}{ll}
 \frac{\square^+0 \circ_{\{0\}}}{E} \square^+0 \bullet_0 & \text{nothingGo} \\
 \frac{\square^+n \circ_{\{n+2\}}}{E} \square^+n \bullet_{n+2} & \text{exitGo} \\
 \frac{\square^+!s \circ_{\{0\}}}{E} \square^+!s \bullet_0 & \text{emitGo} \\
 \frac{\square^-0 \circ_{\{0\}}}{E} \square^-0 \circ_\emptyset & \text{nothingNoGo} \\
 \frac{\square^-n \circ_{\{n+2\}}}{E} \square^-n \circ_\emptyset & \text{exitNoGo} \\
 \frac{\square^-!s \circ_{\{0\}}}{E} \square^-!s \circ_\emptyset & \text{emitNoGo}
 \end{array}$$



■ **Figure 11** The Esterel circuit translation, as given by [4]. The missing $@s$ statement can be recovered as a macro: $\{(s ? 2, 1) *\}$.

7.3.1.2 The 1 rules

If the $\blacksquare 1 \bullet$ statement is started, that is, we have $Go(\blacksquare)^+$, then the output should become $\bullet_1$. In order to trigger this rule only when it increases information, we add the precondition $\bullet < \bullet_1$.

$$\frac{Go(\blacksquare)^+ \quad \bullet < \bullet_1}{\blacksquare \text{pause } \bullet \xrightarrow{E} \blacksquare \text{pause } \bullet_1} \text{pauseGo}$$

► Remark 18. Thanks to the input invariant (that is, $Go(\blacksquare)^+ \implies Sel^-$), $Go(\blacksquare)^+$ actually means $\square^+$. We use the former version because the circuit does not check (rightfully) that Sel is $-$.

If the **pause** statement is not started, then its completion code cannot be 1:

$$\frac{Go(\blacksquare)^- \quad 1 \in K}{\blacksquare \text{pause } \circ_K \xrightarrow{E} \blacksquare \text{pause } \circ_{K \setminus 1}} \text{pauseNoGo}$$

Similarly, for Res, we have:

$$\frac{\bullet < \bullet_0}{\blacksquare^+ \text{pause } \bullet \xrightarrow{E} \blacksquare^+ \text{pause } \bullet_0} \text{pauseRes} \quad \frac{Res(\blacksquare)^- \vee Sel^- \quad 0 \in K}{\blacksquare \text{pause } \circ_K \xrightarrow{E} \blacksquare \text{pause } \circ_{K \setminus 0}} \text{pauseNoRes}$$

7.3.1.3 The @s rules

If the statement is executed, depending on the presence or absence of the signal s , the completion code of @s will be either 0 or 1:

$$\frac{\blacksquare = \blacksquare^+ \quad s^- \in E \quad \bullet < \bullet_1}{\blacksquare @s \bullet \xrightarrow{E} \blacksquare @s \bullet_1} \text{awimmM}$$

$$\frac{\blacksquare = \blacksquare^+ \quad s^+ \in E \quad \bullet < \bullet_0}{\blacksquare @s \bullet \xrightarrow{E} \blacksquare @s \bullet_0} \text{awimmP}$$

When the statement is not executed or the signal is present, respectively, absent, we know that the rule for the absence, respectively, presence, cannot be triggered. Hence, we can remove the corresponding completion code from the output color:

$$\frac{\blacksquare = \blacksquare^- \vee s^- \in E \quad 0 \in K}{\blacksquare @s \circ_K \xrightarrow{E} \blacksquare @s \circ_{K \setminus 0}} \text{awimmNoGo0} \quad \frac{\blacksquare = \blacksquare^- \vee s^+ \in E \quad 1 \in K}{\blacksquare @s \circ_K \xrightarrow{E} \blacksquare @s \circ_{K \setminus 1}} \text{awimmNoGo1}$$

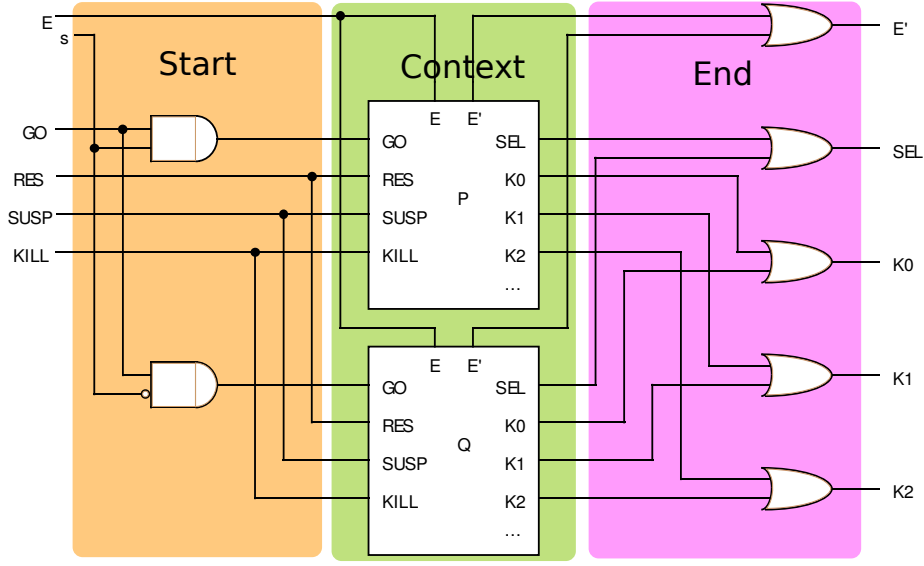
These two rules permit to make progress using the status of s , even before knowing whether the statement is executed or not.

7.3.2 Compound microstates

Compound microstates contain input and output colors and some sub-microstates. Their microstep rules are split into three parts⁷:

- a *start* part which propagates control: it converts the input color of the compound microstate into input colors for its sub-microstates;
- a *context* part where sub-microstates evolve on their own;
- an *end* part where the output colors of sub-microstates are combined into an output color for the compound microstate.

⁷ This distinction is actually violated by a single case: in a sequence $p; q$, starting q is decided depending on the output color of p , so that this rule is neither a start rule (which it should intuitively be) nor an end rule.



■ **Figure 12** The start, context and end delimitation in the circuit translation of the $s?p, q$ statement.

In SOS-style semantics, the context rules simply express that execution can happen inside a subpart: if p executes into p' , then any term containing p can execute into the same term where p is replaced by p' . They have the same role here, except in the $p \setminus s$ construction where they additionally update the value of the local signal s .

The level of detail we want to reach is roughly the gate level, but not always. More precisely, we want to express the functional dependency of each wire on the values of other wires, without being tied to a particular implementation. For instance, in $p \mid q$, the output color is the maximum of the output colors of p and q , but we do not want to commit to a given implementation of this maximum. This will permit to try other implementations without changing the specification.

► **Coq Remark.** In the Coq formalization, some input rules are split into two: one for *Go* and another for *Res*. This is merely for convenience: in this case, each input rule only transmits a single input wire (*Go* or *Res*), making proofs easier to follow. On paper, it seems unnecessary to add more rules, so we keep a single input rule. This is the case for rules *trapI*, *shiftI*, *seqIL*, *parIL*, *parIR*, *signaIL*.

7.3.2.1 The $s?p, q$ rules

The distinction between start, context and end rules is very clear here, see Figure 12. The start part contains two AND gates (one for p and one for q), and the end part contains several OR gates for the K_i . (Remember that we ignore the computation of *Sel*.) The start rules for p merely propagate the *Res* wire from $s?p, q$ to p :

$$\frac{Res(p) < Res(\Box)}{\Box(s?p, q) \bullet \xrightarrow{E} \Box(s?[Res(\Box)]p, q) \bullet} \text{ifResL}$$

The use of preconditions containing $<$ ensures that a rule can only be triggered if it increases information inside the microstate.

The statement p is started ($Go(p)^+$) if the overall statement is started ($Go(\Box(s?p, q) \bullet)^+$) and s is present ($E(s) = +$). More generally, $Go(p)$ is the conjunction of $Go(\Box(s?p, q) \bullet) = Go(\Box)$ and $E(s)$.

$$\frac{Go(p) < Go(\sqsubseteq) \wedge E(s)}{\sqsubseteq(s ? p, q) \bullet \xrightarrow{E} \sqsubseteq(s ? [Go^{Go(\sqsubseteq) \wedge E(s)}]p, q) \bullet} \text{ifGoL}$$

The start rules for q are identical, except that we negate the value of s in the rule for Go:

$$\frac{Go(q) < Go(\sqsubseteq) \wedge \neg E(s)}{\sqsubseteq(s ? p, q) \bullet \xrightarrow{E} \sqsubseteq(s ? p, [Go^{Go(\sqsubseteq) \wedge \neg E(s)}]q) \bullet} \text{ifGoR}$$

$$\frac{Res(q) < Res(\sqsubseteq)}{\sqsubseteq(s ? p, q) \bullet \xrightarrow{E} \sqsubseteq(s ? p, [Res(\sqsubseteq)]q) \bullet} \text{ifResR}$$

The two context rules permit executing either p or q ; the end rule combines the completion codes of p and q by taking their union. Remember that p and q are sub-microstates and thus contain their own input and output colors, so that p , and not $\sqsubseteq p \bullet$, is the correct microstate to use as a premise.

$$\frac{p \xrightarrow{E} p'}{\sqsubseteq(s ? p, q) \bullet \xrightarrow{E} \sqsubseteq(s ? p', q) \bullet} \text{ifCL} \quad \frac{q \xrightarrow{E} q'}{\sqsubseteq(s ? p, q) \bullet \xrightarrow{E} \sqsubseteq(s ? p, q') \bullet} \text{ifCR}$$

$$\frac{\bullet < \text{out}(p) \cup \text{out}(q)}{\sqsubseteq(s ? p, q) \bullet \xrightarrow{E} \sqsubseteq(s ? p, q)(\text{out}(p) \cup \text{out}(q)) \bullet} \text{ifE}$$

7.3.2.2 The $s \supset p$ rules

When started, the $s \supset p$ statement starts its sub-statement p . Thus, Go is simply transmitted to the sub-statement. When resumed, the $s \supset p$ statement resumes its sub-statement p only when s is absent. In order not to resume an inactive statement, $Sel(p)$ is also checked.

$$\frac{Go(p) < Go(\sqsubseteq)}{\sqsubseteq s \supset p \bullet \xrightarrow{E} \sqsubseteq s \supset ([Go(\sqsubseteq)]p) \bullet} \text{suspendGo}$$

$$\frac{Res(p) < \overbrace{Res(\sqsubseteq) \wedge Sel(p) \wedge \neg E(s)}^{res}}{\sqsubseteq(s \supset p) \bullet \xrightarrow{E} \sqsubseteq s \supset ([Res^{res}]p) \bullet} \text{suspendRes}$$

The context rule executes p :

$$\frac{p \xrightarrow{E} p'}{\sqsubseteq(s \supset p) \bullet \xrightarrow{E} \sqsubseteq(s \supset p') \bullet} \text{suspendC}$$

For the end rule, we first define an auxiliary function $\text{SuspNow}(bo, \bullet)$ to compute the completion code:

$$\text{SuspNow}(bo, \bullet) = \begin{cases} \bullet_1 & \text{if } bo = + \\ \bullet & \text{if } bo = - \\ \bullet_{\{1\}} \cup \bullet & \text{if } bo = \perp \end{cases}$$

Essentially, bo represent suspension of the sub-statement p : if bo is $+$, p is suspended and the completion code is 1; if bo is $-$, p is executed normally; if bo is $\perp$, it adds the possibility of returning 1 to the output color of p . The suspension of p only happens when p should have been resumed, that is, $Res(\sqsubseteq) \wedge Sel(p) = +$, but s was present, that is, $E(s) = +$.

$$\frac{\bullet < \text{SuspNow}(\overbrace{\text{Res}(\Box) \wedge \text{Sel}(p) \wedge E(s)}^{susp}, \text{out}(p))}{\Box(s \supset p) \bullet \xrightarrow{E} \Box(s \supset p) \text{SuspNow}(susp, \text{out}(p))} \text{suspendE}$$

For simplicity, we introduce a particular case of this rule, where bo is set to $\perp$:

$$\frac{\bullet < \text{SuspNow}(\perp, \text{out}(p))}{\Box(s \supset p) \bullet \xrightarrow{E} \Box(s \supset p) \text{SuspNow}(\perp, \text{out}(p))} \text{suspendEKO}$$

Without this rule, we cannot ignore the value of s in E , so that compatibility with ordering on events (last property of Theorem 19) is lost. More importantly, the context rule for $p \setminus s$ becomes false.

7.3.2.3 The $\{p\}$ and $\uparrow p$ rules

The $\{p\}$ and $\uparrow p$ statements only have an effect on the completion code of p , thus the input color is simply transmitted.

$$\begin{array}{c} \frac{in(p) < \Box}{\Box\{p\} \bullet \xrightarrow{E} \Box\{\Box p\} \bullet} \text{trapI} \\ \frac{p \xrightarrow{E} p'}{\Box\{p\} \bullet \xrightarrow{E} \Box\{p'\} \bullet} \text{trapC} \\ \frac{\bullet < \downarrow \text{out}(p)}{\Box\{p\} \bullet \xrightarrow{E} \Box\{p\} \downarrow \text{out}(p)} \text{trapE} \end{array} \quad \begin{array}{c} \frac{in(p) < \Box}{\Box(\uparrow p) \bullet \xrightarrow{E} \Box(\uparrow \Box p) \bullet} \text{shiftI} \\ \frac{p \xrightarrow{E} p'}{\Box(\uparrow p) \bullet \xrightarrow{E} \Box(\uparrow p') \bullet} \text{shiftC} \\ \frac{\bullet < \uparrow \text{out}(p)}{\Box(\uparrow p) \bullet \xrightarrow{E} \Box(\uparrow p) \uparrow \text{out}(p)} \text{shiftE} \end{array}$$

7.3.2.4 The $p; q$ rules

In the start rules, Res is forwarded to both p and q whereas Go is transmitted only to p .

$$\frac{in(p) < \Box}{\Box(p; q) \bullet \xrightarrow{E} \Box(\Box p; q) \bullet} \text{seqIL} \quad \frac{\text{Res}(q) < \text{Res}(\Box)}{\Box(p; q) \bullet \xrightarrow{E} \Box(p; [\text{Res}(\Box)]q) \bullet} \text{seqResR}$$

The sub-statement q is started depending on the completion code of p , that is, the value of $\text{Go}(q)$ is set to $+$ or $-$ respectively when $\text{out}(p)$ is $\bullet_0$ or cannot become $\bullet_0$.

$$\frac{\text{out}(p) = \bullet_0 \quad \text{Go}(q) = \perp}{\Box(p; q) \bullet \xrightarrow{E} \Box(p; [\text{Go}^+]q) \bullet} \text{seqGoR} \quad \frac{\text{out}(p) \not\leq \bullet_0 \quad \text{Go}(q) = \perp}{\Box(p; q) \bullet \xrightarrow{E} \Box(p; [\text{Go}^-]q) \bullet} \text{seqNoGoR}$$

In the second rule, we know that q cannot be started as soon as the completion code of p cannot be 0, regardless of other information we might know about $\text{out}(p)$. Hence, we use $\text{out}(p) \not\leq \bullet_0$. This amounts to having $\text{out}(p) = \bullet_k$ with $k \neq 0$ or $\text{out}(p) = \circ_K$ with $0 \notin K$.

As for all statements, the context rules permit execution of sub-statements.

$$\frac{p \xrightarrow{E} p'}{\Box(p; q) \bullet \xrightarrow{E} \Box(p'; q) \bullet} \text{seqCL} \quad \frac{q \xrightarrow{E} q'}{\Box(p; q) \bullet \xrightarrow{E} \Box(p; q') \bullet} \text{seqCR}$$

For the end rule, we remove 0 from the possible completion codes of p , written $\text{out}(p) \setminus 0$, before taking the union with the ones from q . More precisely, if $\text{out}(p)$ is $\circ_K$, we return $\circ_{K \setminus 0}$, if $\text{out}(p)$ is $\bullet_0$, we return $\circ_\emptyset$, and if $\text{out}(p)$ is $\bullet_k$ with $k \neq 0$, we return it unchanged.

$$\frac{\bullet < (\text{out}(p) \setminus 0) \cup \text{out}(q)}{\boxed{s ? p, q} \bullet \xrightarrow[E]{} \boxed{s ? p, q}(\text{out}(p) \setminus 0) \cup \text{out}(q)} \text{seqE}$$

► **Coq Remark.** In the Coq development, the operation $\bullet \setminus 0$ is written *SEQrestrict*($\bullet$).

7.3.2.5 The $p *$ rules

A loop $p *$ starts its body either when the loop itself starts or when the body finishes an iteration, that is, when $\text{out}(p) = \bullet_0$. Getting inspiration from the circuit translation, we use an OR gate between the Go wire of $p *$ and the 0-th component of the output color of p , written $\text{out}(p)[0]$. Resumption is simply propagated. The end rule propagates the output color of p after forcing the 0-th component to false.

$$\begin{aligned} & \frac{Go(p) < Go(\boxed{}) \vee \text{out}(p)[0]}{\boxed{p} * \bullet \xrightarrow[E]{} \boxed{([Go(\boxed{}) \vee \text{out}(p)[0])p} * \bullet} \text{loopGo} \\ & \frac{Res(p) < Res(\boxed{})}{\boxed{p} * \bullet \xrightarrow[E]{} \boxed{([Res(\boxed{})]p) * \bullet} \text{loopRes} \\ & \frac{p \xrightarrow[E]{} p'}{\boxed{p} * \bullet \xrightarrow[E]{} \boxed{p'} * \bullet} \text{loopC} \\ & \frac{\bullet < \text{out}(p) \setminus 0}{\boxed{p} * \bullet \xrightarrow[E]{} \boxed{p} * (\text{out}(p) \setminus 0)} \text{loopE} \end{aligned}$$

7.3.2.6 The $p \mid q$ rules

Input and context rules simply propagate control.

$$\begin{aligned} & \frac{in(p) < \boxed{}}{\boxed{p \mid q} \bullet \xrightarrow[E]{} \boxed{(\boxed{}p \mid q) \bullet} \text{parIL} & \frac{in(q) < \boxed{}}{\boxed{p \mid q} \bullet \xrightarrow[E]{} \boxed{p \mid (\boxed{}q) \bullet} \text{parIR} \\ & \frac{p \xrightarrow[E]{} p'}{\boxed{p \mid q} \bullet \xrightarrow[E]{} \boxed{p' \mid q) \bullet} \text{parCL} & \frac{q \xrightarrow[E]{} q'}{\boxed{p \mid q} \bullet \xrightarrow[E]{} \boxed{p \mid q'} \bullet} \text{parCR} \end{aligned}$$

For the end rule, the synchronizer computes the max of the completion codes of p and q , except when one of them is inactive (that is, its Sel is $-$), in which case it returns the other one since this corresponds to the case when one branch is dead and the other one is alive.

$$\frac{\bullet < \overbrace{\text{synchronizer}_{Sel(p), Sel(q)}(\text{out}(p), \text{out}(q))}^{synch}}{\boxed{p \mid q} \bullet \xrightarrow[E]{} \boxed{p \mid q} synch} \text{parE}$$

with

$$\text{synchronizer}_{Sel(p), Sel(q)}(\bullet_p, \bullet_q) = \begin{cases} \text{Max}(\bullet_p, \bullet_q) & \text{when } Sel(p) = Sel(q) \\ \bullet_p & \text{when } Sel(q) = + \text{ and } Sel(p) = - \\ \bullet_q & \text{when } Sel(p) = - \text{ and } Sel(q) = + \end{cases}$$

and

$$\begin{aligned} \text{Max}(\circ_K, \circ_{K'}) &:= \circ_{\text{Max}(K, K')} & \text{Max}(\bullet_k, \circ_K) &:= \circ_{\text{Max}(\{k\}, K)} \\ \text{Max}(\bullet_k, \bullet_{k'}) &:= \bullet_{\text{max}(k, k')} & \text{Max}(\circ_K, \bullet_k) &:= \circ_{\text{Max}(K, \{k\})} \end{aligned}$$

7.3.2.7 The $p \setminus s$ rules

The input and output colors are simply propagated by the start and end rules.

$$\frac{\text{in}(p) < \blacksquare}{\blacksquare(p \setminus s) \bullet \xrightarrow{E} \blacksquare(\blacksquare p) \setminus s \bullet} \text{signalI} \qquad \frac{\bullet < \text{out}(p)}{\blacksquare(p \setminus s) \bullet \xrightarrow{E} \blacksquare(p \setminus s) \text{out}(p)} \text{signalE}$$

For the context rule, we first check the emitters of s inside p to deduce the status of s . In the circuit, this only amounts to wire propagation and a big OR gate to combine all emitters into the signal status. Here, since we do not model signal propagation explicitly, we use instead the `to_event` function (see the end of Section 7.2) which scans a microstate and identifies emitters and whether they are executed, not executed or still pending.

$$\frac{b := (\text{to_event}(p, s^\perp * E))(s) \quad p \xrightarrow{E * s^b} p'}{\blacksquare(p \setminus s) \bullet \xrightarrow{E} \blacksquare(p' \setminus s) \bullet} \text{signalC}$$

7.3.3 ABROi in the microstep semantics

As the microstep semantics requires a lot more steps than previous semantics, for the sake of space, we only illustrate the first instant (where only B is present), which still requires 46 microsteps! We combine several microsteps at once when this does not hinder understanding too much and, to improve readability, we underline the parts that change and we define two abbreviations:

$$\begin{aligned} \text{check} &= \square 1_{\{0,1\}} ; \square (\square (R ? \square 2_{\{2\}} , \square 1_{\{0,1\}})_{\{0,1,2\}}) * \square_{\{1,2\}} \\ \text{body} &= \square (\square (\square @A_{\{0,1\}} \mid \square @B_{\{0,1\}})_{\{0,1\}} ; \square !O_{\{0\}})_{\{0,1\}} ; \square (\square 1_{\{1\}}) * \square_{\{1\}} \end{aligned}$$

Let us start by executing `check` and `body`, and seeing how this fits into the execution of `ABROi`.

$$\begin{aligned} \text{check} &= \square 1_{\{0,1\}} ; \square (\square (R ? \square 2_{\{2\}} , \square 1_{\{0,1\}})_{\{0,1,2\}}) * \square_{\{1,2\}} \\ &\xrightarrow[\{B\}]{}^2 \square 1_{\{1\}} ; \square (\square (R ? \underline{\square 2}_{\{2\}} , \square 1_{\{0,1\}})_{\{0,1,2\}}) * \square_{\{1,2\}} \\ &\xrightarrow[\{B\]}{} \square 1_{\{1\}} ; \square (\square (R ? \underline{\square 2}_{\emptyset} , \square 1_{\{0,1\}})_{\{0,1,2\}}) * \square_{\{1,2\}} \\ &\xrightarrow[\{B\}]{}^2 \square 1_{\{1\}} ; \square (\square (R ? \underline{\square 2}_{\emptyset} , \square 1_{\{0,1\}})_{\{0,1\}}) * \underline{\square_{\{1\}}} = \text{check}' \\ \text{body} &= \square (\square (\square @A_{\{0,1\}} \mid \square @B_{\{0,1\}})_{\{0,1\}} ; \square !O_{\{0\}})_{\{0,1\}} ; \square (\square 1_{\{1\}}) * \square_{\{1\}} \\ &\xrightarrow[\{B\}]{}^2 \square (\square (\square @A_{\{1\}} \mid \square @B_{\{0\}})_{\{0,1\}} ; \square !O_{\{0\}})_{\{0,1\}} ; \square (\square 1_{\{1\}}) * \square_{\{1\}} = \text{body}' \\ \text{ABROi} &= (\square \{ \square (\square \text{check}_{\{1,2\}} \mid \square (\square \text{body}_{\{1\}} ; \square 2_{\{2\}})_{\{1,2\}})_{\{1,2\}} \}_{\{0,1\}}) * \\ &\xrightarrow[\{B\}]{}^7 (\square \{ \square (\square \underline{\text{check}}'_{\{1,2\}} \mid \square (\square \underline{\text{body}}'_{\{1\}} ; \square 2_{\{2\}})_{\{1,2\}})_{\{1,2\}} \}_{\{0,1\}}) * \\ &\xrightarrow[\{B\}]{}^2 (\square \{ \square (\square \underline{\text{check}}'_{\{1\}} \mid \square (\square \text{body}'_{\{1\}} ; \underline{\square 2}_{\{2\}})_{\{1,2\}})_{\{1,2\}} \}_{\{0,1\}}) * \\ &\xrightarrow[\{B\]}{} (\square \{ \square (\square \underline{\text{check}}'_{\{1\}} \mid \square (\square \text{body}'_{\{1\}} ; \underline{\square 2}_{\emptyset})_{\{1,2\}})_{\{1,2\}} \}_{\{0,1\}}) * \\ &\xrightarrow[\{B\}]{}^3 (\square \{ \square (\square \underline{\text{check}}'_{\{1\}} \mid \square (\square \text{body}'_{\{1\}} ; \underline{\square 2}_{\emptyset})_{\{1\}})_{\{1\}} \}_{\{1\}}) * = \text{ABROi}' \end{aligned}$$

Notice that at this point, we have used the statuses of signals (A , B and R) but the input color of `ABROi` has not been set yet. Let us use now the fact that execution is started.

$$\begin{aligned}
\Box^+ \text{ABROi}' \circ_{\{1\}} &\xrightarrow[\{B\}]{5} \Box^+ (\Box^+ \{ \Box^+ (\Box^+ \text{check}' \circ_{\{1\}} \mid \Box^+ (\Box^+ \text{body}' \circ_{\{1\}} ; \Box^- 2 \circ_{\emptyset}) \circ_{\{1\}} \} \circ_{\{1\}}) * \circ_{\{1\}} \\
\Box^+ \text{check}' \circ_{\{1\}} &\xrightarrow[\{B\}]{1} \Box^+ (\Box^+ 1 \circ_{\{1\}} ; \Box (\Box (R ? \Box^- 2 \circ_{\emptyset} , \Box 1 \circ_{\{0,1\}}) \circ_{\{0,1\}}) * \circ_{\{1\}}) \circ_{\{1\}} \\
&\xrightarrow[\{B\}]{1} \Box^+ (\Box^+ 1 \bullet_1 ; \Box (\Box (R ? \Box^- 2 \circ_{\emptyset} , \Box 1 \circ_{\{0,1\}}) \circ_{\{0,1\}}) * \circ_{\{1\}}) \circ_{\{1\}} \\
&\xrightarrow[\{B\}]{3} \Box^+ (\Box^+ 1 \bullet_1 ; \Box (\Box (R ? \Box^- 2 \circ_{\emptyset} , \Box^- 1 \circ_{\{0,1\}}) \circ_{\{0,1\}}) * \circ_{\{1\}}) \circ_{\{1\}} \\
&\xrightarrow[\{B\}]{3} \Box^+ (\Box^+ 1 \bullet_1 ; \Box (\Box (R ? \Box^- 2 \circ_{\emptyset} , \Box^- 1 \circ_{\emptyset}) \circ_{\emptyset}) * \circ_{\emptyset}) \circ_{\{1\}} \\
&\xrightarrow[\{B\}]{1} \Box^+ (\Box^+ 1 \bullet_1 ; \Box (\Box (R ? \Box^- 2 \circ_{\emptyset} , \Box^- 1 \circ_{\emptyset}) \circ_{\emptyset}) * \circ_{\emptyset}) \bullet_1 \\
\Box^+ \text{body}' \circ_{\{1\}} &\xrightarrow[\{B\}]{4} \Box^+ (\Box^+ (\Box^+ @A \circ_{\{1\}} \mid \Box^+ @B \circ_{\{0\}}) \circ_{\{0,1\}} ; \Box !O \circ_{\{0\}}) \circ_{\{0,1\}} ; \Box (\Box 1 \circ_{\{1\}}) * \circ_{\{1\}}) \circ_{\{1\}} \\
&\xrightarrow[\{B\}]{2} \Box^+ (\Box^+ (\Box^+ @A \bullet_1 \mid \Box^+ @B \bullet_0) \circ_{\{0,1\}} ; \Box !O \circ_{\{0\}}) \circ_{\{0,1\}} ; \Box (\Box 1 \circ_{\{1\}}) * \circ_{\{1\}}) \circ_{\{1\}} \\
&\xrightarrow[\{B\}]{1} \Box^+ (\Box^+ (\Box^+ @A \bullet_1 \mid \Box^+ @B \bullet_0) \bullet_1 ; \Box !O \circ_{\{0\}}) \circ_{\{0,1\}} ; \Box (\Box 1 \circ_{\{1\}}) * \circ_{\{1\}}) \circ_{\{1\}} \\
&\xrightarrow[\{B\}]{1} \Box^+ (\Box^+ (\Box^+ @A \bullet_1 \mid \Box^+ @B \bullet_0) \bullet_1 ; \Box^- !O \circ_{\emptyset}) \circ_{\{0,1\}} ; \Box (\Box 1 \circ_{\{1\}}) * \circ_{\{1\}}) \circ_{\{1\}} \\
&\xrightarrow[\{B\}]{1} \Box^+ (\Box^+ (\Box^+ @A \bullet_1 \mid \Box^+ @B \bullet_0) \bullet_1 ; \Box^- !O \circ_{\emptyset}) \bullet_1 ; \Box (\Box 1 \circ_{\{1\}}) * \circ_{\{1\}}) \circ_{\{1\}} \\
&\xrightarrow[\{B\}]{2} \Box^+ (\Box^+ (\Box^+ @A \bullet_1 \mid \Box^+ @B \bullet_0) \bullet_1 ; \Box^- !O \circ_{\emptyset}) \bullet_1 ; \Box (\Box^- 1 \circ_{\{1\}}) * \circ_{\{1\}}) \circ_{\{1\}} \\
&\xrightarrow[\{B\}]{2} \Box^+ (\Box^+ (\Box^+ @A \bullet_1 \mid \Box^+ @B \bullet_0) \bullet_1 ; \Box^- !O \circ_{\emptyset}) \bullet_1 ; \Box (\Box^- 1 \circ_{\emptyset}) * \circ_{\emptyset}) \circ_{\{1\}} \\
&\xrightarrow[\{B\}]{1} \Box^+ (\Box^+ (\Box^+ @A \bullet_1 \mid \Box^+ @B \bullet_0) \bullet_1 ; \Box^- !O \circ_{\emptyset}) \bullet_1 ; \Box (\Box^- 1 \circ_{\emptyset}) * \circ_{\emptyset}) \bullet_1
\end{aligned}$$

We observe that the final microstates for `check'` and `body'` are total, so cannot execute further. Let us call them `check''` and `body''` respectively. Finally, we get for `ABROi`:

$$\begin{aligned}
\Box^+ \text{ABROi}' \circ_{\{1\}} &\xrightarrow[\{B\}]{5} \Box^+ (\Box^+ \{ \Box^+ (\Box^+ \text{check}' \circ_{\{1\}} \mid \Box^+ (\Box^+ \text{body}' \circ_{\{1\}} ; \Box^- 2 \circ_{\emptyset}) \circ_{\{1\}} \} \circ_{\{1\}}) * \circ_{\{1\}} \\
&\xrightarrow[\{B\}]{24} \Box^+ (\Box^+ \{ \Box^+ (\Box^+ \text{check}'' \bullet_1 \mid \Box^+ (\Box^+ \text{body}'' \bullet_1 ; \Box^- 2 \circ_{\emptyset}) \circ_{\{1\}}) \circ_{\{1\}} \} \circ_{\{1\}}) * \circ_{\{1\}} \\
&\xrightarrow[\{B\}]{1} \Box^+ (\Box^+ \{ \Box^+ (\Box^+ \text{check}'' \bullet_1 \mid \Box^+ (\Box^+ \text{body}'' \bullet_1 ; \Box^- 2 \circ_{\emptyset}) \bullet_1) \circ_{\{1\}} \} \circ_{\{1\}}) * \circ_{\{1\}} \\
&\xrightarrow[\{B\}]{3} \Box^+ (\Box^+ \{ \Box^+ (\Box^+ \text{check}'' \bullet_1 \mid \Box^+ (\Box^+ \text{body}'' \bullet_1 ; \Box^- 2 \circ_{\emptyset}) \bullet_1) \bullet_1 \} \bullet_1) * \bullet_1
\end{aligned}$$

From this final microstate, we can read that:

- `O` is not emitted because its only emitter inside `body''` is `Box^- !O \circ_{\emptyset}`,
- the activated `1` and `@s` statements are `Box^+ 1 \bullet_1` inside `check''` and `Box^+ @A \bullet_1` inside `body''`.

7.4 Properties of the Microstep Semantics



As it is lower-level than previous semantics, the reader may wonder which properties of the previous semantics are preserved by the microstep semantics. In fact, it enjoys most of the properties of other Kernel Esterel semantics we have studied so far:

► **Theorem 19.** *The microstep semantics enjoys the following properties:*

- *The base statement and Sel values are unchanged by execution:*
 $\forall E, p, p', p \xrightarrow{E} p' \implies \mathcal{B}(p') = \mathcal{B}(p) \wedge \text{Sel}(p') = \text{Sel}(p);$
- *The input color is unchanged by execution:* $\forall E, p, p', p \xrightarrow{E} p' \implies \text{in}(p) = \text{in}(p');$
- *Each microstep increases information:* $\forall E, p, p', p \xrightarrow{E} p' \implies p < p';$
Corollary: Total microstates cannot execute.



Since there is no output event E' , the event domain preservation property does not make sense.

The reader may have noticed that an important property of previous semantics is missing: determinism. And in fact, we do lose it, because microsteps are local and may happen in several places in parallel, for instance when starting both branches of $p \mid q$ (rules `parIL` and `parIR`). This may seem like a serious issue as the Esterel language is meant for programming critical systems, in which determinism is often paramount. Thankfully, we have the next best results: confluence and termination.

Determinism, confluence, and termination

The reason why determinism is so important for a semantics is to ensure uniqueness of the execution path and therefore of the result. *Confluence* is a property that allows to recover uniqueness of the result for a non-deterministic semantics: it expresses that for any two execution paths, we can execute them further to reach the same state. In particular, the final result, if any, must be the same since it cannot execute further. When all possible execution paths are finite, Newman’s lemma [33] shows that *local confluence*, where only single step executions need to be considered, is enough.



For Esterel, termination of the microstep semantics is quite easy to prove as this semantics is meant to mimic the circuit execution in which every execution step sets the value of (at least) one more wire. And indeed, if we define the $\mathbb{N}$ -valued measure $Mmeasure$ over microstates counting the number of wires still having a value $\perp$, we can prove that the measure strictly decreases with every microstep. Thus, from any microstate, only a finite number of microsteps are possible.

► **Remark 20.** This measure is actually a different way to define a partial order between microstates but it is coarser than the Scott ordering of Section 7.2 as it allows comparison between any two microstates, including microstates not having the same underlying base statement or where information evolves in different locations.

On the other hand, local confluence does not hold for arbitrary microstates. For example, the (meaningless) microstate “ $s ? \square^+ 0 \circ_{\{0\}}, \square^+ 1 \circ_{\{1\}}$ ” can produce output color $\bullet_0$ and $\bullet_1$ depending on which branch we execute, and cannot be made confluent. Nevertheless, the execution of an Esterel program will never exhibit such meaningless microstates. Thus, we need to restrict the proof of local confluence to microstates satisfying a well-formation invariant. Once we have that invariant called *valid_coloring*(E, p) (see below), we can prove that the microstep semantics is confluent:

► **Theorem 21 (Confluence).** *The microstep semantics is locally confluent from any well-formed starting microstate. As it is also terminating, by Newman’s lemma [33], it is confluent from any well-formed starting microstate.*

The *valid_coloring*(E, p) predicate

The predicate *valid_coloring*(E, p) represents the fact that microstates appearing along the evaluation of an Esterel program are not arbitrary: they satisfy some invariants coming from the circuit semantics and from the circuit translation of Esterel. More precisely, it expresses that

1. the input and output colors inside the microstate p are coherent with the information they have access to, that is, the output value of a gate contains no more information than its input values allow (circuit semantics invariant);

2. Sel and Go are exclusive and so are branches of a test or a sequence (Esterel circuit translation invariant);
3. all signals used in p are declared in E (well-formation invariant).

The last point is added only to ensure the well-formation condition $\text{valid_dom}(E, p)$ without which the execution of an Esterel program does not makes sense. In particular, $\text{valid_coloring}(E, p)$ then implies $\text{valid_dom}(E, \mathcal{B}(p))$.

The set of constraints for a microstate $\blacksquare p \bullet$ is defined by case analysis over p and is given in Figure 13. The interested reader is invited to look at the Coq code, which is more explicit and commented and less likely to contain a mistake.

For illustration purposes, we detail the case $s ? p, q$ and how it should be understood.

- the input color must satisfy its Esterel invariant: $\text{input_invariant}(\text{sel}, \blacksquare);$
- the signal s must exist in the environment E ; let b be its value: $s^b \in E;$
- $s ? p, q$ is active iff one of p or q is: $\text{sel} = \text{Sel}(p) \parallel \text{Sel}(q);$
- both p and q cannot be active at the same time: $\text{Sel}(p) \& \text{Sel}(q) = \text{false};$
- $\text{Go}(p)$ and $\text{Go}(q)$ are computed from $\blacksquare$ and b : $\text{Go}(p) \leq \text{Go}(\blacksquare) \& b$ and $\text{Go}(q) \leq \text{Go}(\blacksquare) \& \neg b;$
- the rest of the input color is transmitted to p and q : $\text{Res}(p) \leq \text{Res}(\blacksquare)$ and $\text{Res}(q) \leq \text{Res}(\blacksquare);$
- out contains at least the static information: $\text{out}(\text{from_stmt}(\mathcal{B}(p))) \cup \text{out}(\text{from_stmt}(\mathcal{B}(q))) \leq \bullet;$
- ... and at most the information of p and q : $\bullet \leq \text{out}(p) \cup \text{out}(q);$
- recursively, p and q must be well-colored: $\text{valid_coloring}(E, p)$ and $\text{valid_coloring}(E, q).$

To ensure that the $\text{valid_coloring}(E, p)$ property is indeed an invariant, we prove that it holds at the start of execution and is preserved by the microstep semantics:

► **Theorem 22** (Invariant of the circuit translation).

- For any well-formed statement p , $\text{valid_coloring}(E, \text{from_stmt}(p))$ holds ;
- For any well-formed state $\hat{p}$, $\text{valid_coloring}(E, \text{from_state}(\hat{p}))$ holds ;
- The valid_coloring property is preserved by the microstep semantics.

This invariant directly entails the following properties of the control flow in the microstep semantics, which are rather intuitive and increase our confidence in its definition:

- Control is never created, only propagated:
 - $\text{valid_coloring}(E, \blacksquare p \bullet_k) \implies \blacksquare = \blacksquare^+;$
 - $\text{valid_coloring}(E, \blacksquare p \circ_\emptyset) \implies \blacksquare = \blacksquare^-;$
- Branches in a test and a sequence are not simultaneously active:
 - $\text{valid_coloring}(E, \blacksquare(s ? p, q) \bullet) \implies \text{in}(p) \neq \blacksquare^+ \vee \text{in}(q) \neq \blacksquare^+;$
 - $\text{valid_coloring}(E, \blacksquare(p ; q) \bullet) \implies \text{out}(p) \setminus 0 \neq \bullet_k \vee \text{in}(q) \neq \blacksquare^+;$

Useful properties for optimization and reasoning

Using our microstep semantics, we can prove properties of control flow which can lead to better simulation or reasoning, as well as justify some design choices. We illustrate this on two examples.

First, dead code does not create a completion code:

► **Lemma 23** (Execution of non-executed microstates). *A non-executed microstate gets output color $\circ_\emptyset$. Formally, for all $p, \bullet$ and E :*

- $\blacksquare^- p \bullet \xrightarrow[E]{*} \blacksquare^- p \circ_\emptyset ;$
- For $\blacksquare^-$ (that is, when Set^+ holds), we merely have $\blacksquare^- p \bullet \xrightarrow[E]{*} \blacksquare^- p' \circ_K$, as the Go value may be required to eliminate some completion codes (e.g., 1 for pause). If we additionally have Go^- , then we recover $\blacksquare^- p \bullet \xrightarrow[E]{*} \blacksquare^- p \circ_\emptyset$.

The $\text{valid_coloring}(E, \blacksquare p \bullet)$ property is defined by induction over p . For readability, we omit two parts in these cases:

- all cases contain the input invariant $\text{input_invariant}(\text{sel}, \blacksquare)$ and
- all compound microstates contain recursive calls for sub-microstates.

This recursive call is explicitly shown for $p \setminus s$ because of the change in the environment E .

0	$\mapsto \text{sel} = \text{false} \wedge \text{WIRE}(0, \blacksquare, \bullet)$
1	$\mapsto \circ_{\{0,1\}} \leq \bullet \wedge \text{PAUSE}(\text{sel}, \blacksquare, \bullet)$
k	$\mapsto \text{sel} = \text{false} \wedge \text{WIRE}(k+2, \blacksquare, \bullet)$
$!s$	$\mapsto s \in E \wedge \text{sel} = \text{false} \wedge \text{WIRE}(0, \blacksquare, \bullet)$
$@s$	$\mapsto s \in E \wedge \circ_{\{0,1\}} \leq \bullet$ $\wedge (\blacksquare = \text{red}^- \implies \bullet \leq \circ_\emptyset) \wedge (\blacksquare \neq \text{red}^- \wedge s^\perp \in E \implies \bullet = \circ_{\{0,1\}})$ $\wedge (\blacksquare = \text{green}^+ \wedge s^+ \in E \implies \bullet \leq \bullet_0) \wedge (\blacksquare \neq \text{green}^+ \wedge \blacksquare \neq \text{red}^- \wedge s^+ \in E \implies \bullet \leq \circ_{\{0\}})$ $\wedge (\blacksquare = \text{green}^+ \wedge s^- \in E \implies \bullet \leq \bullet_1) \wedge (\blacksquare \neq \text{green}^+ \wedge \blacksquare \neq \text{red}^- \wedge s^- \in E \implies \bullet \leq \circ_{\{1\}})$
$\{p\}$	$\mapsto \text{in}(p) \leq \blacksquare \wedge \text{sel} = \text{Sel}(p) \wedge \downarrow \text{out}(\text{from_stmt}(\mathcal{B}(p))) \leq \bullet \leq \downarrow \text{out}(p)$
$\uparrow p$	$\mapsto \text{in}(p) \leq \blacksquare \wedge \text{sel} = \text{Sel}(p) \wedge \uparrow \text{out}(\text{from_stmt}(\mathcal{B}(p))) \leq \bullet \leq \uparrow \text{out}(p)$
$s \supset p$	$\mapsto s \in E \wedge \text{sel} = \text{false} \wedge \text{Go}(p) \leq \text{Go}(\blacksquare) \wedge \text{Res}(p) \leq \text{Res}(\blacksquare) \& \text{Sel}(p) \& \neg E(s)$ $\wedge \{1\} \cup \text{out}(\text{from_stmt}(\mathcal{B}(p))) \leq \bullet \leq \text{SuspNow}(\text{Res}(\blacksquare) \& \text{Sel}(p) \& E(s), \text{out}(p))$
$s ? p, q$	$\mapsto s^b \in E \wedge \text{sel} = \text{Sel}(p) \parallel \text{Sel}(q) \wedge \text{Sel}(p) \& \text{Sel}(q) = \text{false}$ $\wedge \text{Go}(p) \leq \text{Go}(\blacksquare) \& b \wedge \text{Go}(q) \leq \text{Go}(\blacksquare) \& \neg b$ $\wedge \text{Res}(p) \leq \text{Res}(\blacksquare) \wedge \text{Res}(q) \leq \text{Res}(\blacksquare)$ $\wedge \text{out}(\text{from_stmt}(\mathcal{B}(p))) \cup \text{out}(\text{from_stmt}(\mathcal{B}(q))) \leq \bullet \leq \text{out}(p) \cup \text{out}(q)$
$p ; q$	$\mapsto \text{sel} = \text{Sel}(p) \parallel \text{Sel}(q) \wedge \text{Sel}(p) \& \text{Sel}(q) = \text{false}$ $\wedge \text{in}(p) \leq \blacksquare \wedge \text{Go}(q) \leq (\text{out}(p))[0] \wedge \text{Res}(q) \leq \text{Res}(\blacksquare)$ $\wedge (\text{out}(\text{from_stmt}(\mathcal{B}(p))) \setminus 0) \cup \text{out}(\text{from_stmt}(\mathcal{B}(q))) \leq \bullet \leq (\text{out}(p) \setminus 0) \cup \text{out}(q)$
$p \mid q$	$\mapsto \text{sel} = \text{Sel}(p) \parallel \text{Sel}(q) \wedge \text{in}(p) \leq \blacksquare \wedge \text{in}(q) \leq \blacksquare$ $\wedge \text{out}(\text{from_stmt}(\mathcal{B}(p))) \cup \text{out}(\text{from_stmt}(\mathcal{B}(q))) \leq \bullet \leq \text{synch}_{\text{Sel}(p), \text{Sel}(q)}(\text{out}(p), \text{out}(q))$
$p \setminus s$	$\mapsto \text{sel} = \text{Sel}(p) \wedge \text{in}(p) \leq \blacksquare \wedge \text{out}(\text{from_stmt}(\mathcal{B}(p))) \leq \bullet \leq \text{out}(p)$ $\text{valid_coloring}(E * s^{\text{to_event}(p, E * s^\perp)}, p)$

where

- $\parallel$ and $\&$ are respectively the Boolean disjunction and conjunction
- $\text{WIRE}(k, \blacksquare, \bullet)$ expresses in terms of input/output relation that $\text{Go}(\blacksquare)$ is directly fed into the k -th component of $\bullet$, which is the case for the 0 , k , and $!s$ statements:

$$\text{WIRE}(k, \blacksquare, \bullet) = \begin{cases} \bullet = \circ_{\{k\}} & \text{when } \text{Go}(\blacksquare) = \perp \\ \bullet \leq \circ_\emptyset & \text{when } \text{Go}(\blacksquare) = - \\ \bullet \leq \bullet_k & \text{when } \text{Go}(\blacksquare) = + \end{cases}$$

- $\text{PAUSE}(\text{Sel}, \blacksquare, \bullet)$ expresses the input/output relation for the 1 (**pause**) statement:

$$\text{PAUSE}(\text{Sel}, \blacksquare, \bullet) = \begin{cases} \bullet \leq \bullet_1 & \text{when } \text{Go}(\blacksquare) = + \\ \bullet \leq \bullet_0 & \text{when } \text{Res}(\blacksquare) = + \text{ and } \text{Sel} = + \\ \bullet \leq \circ_\emptyset & \text{when } \blacksquare = \text{red}^- \\ \bullet \leq \circ_{\{0\}} & \text{when } \text{Go}(\blacksquare) = - \text{ and } \text{Sel} = + \text{ and } \text{Res}(\blacksquare) \neq + \\ \bullet \leq \circ_{\{1\}} & \text{when } \text{Go}(\blacksquare) = \perp \text{ and } (\text{Res}(\blacksquare) = - \text{ or } \text{Sel} = -) \\ \bullet \leq \circ_{\{0,1\}} & \text{otherwise} \end{cases}$$

■ **Figure 13** The $\text{valid_coloring}(E, p)$ invariant.

This property is the (partial) converse of the fact that control is never created, only propagated and can speed up reasoning about $s?p, q$ statements. Indeed, it means that there is no need to simulate the branch not taken in a branching test, we already know that all wires, both control and completion codes, will eventually be set to 0.

Second, when the starting microstate is not active (that is, it has Sel^-), the Res wire has no impact on the execution of the current reaction. Said otherwise, in inactive microstates, only Go matters, Res does not.

► **Lemma 24** (Invariance by Res). *In an inactive microstate, the Res wire does not matter. In other words, an inactive microstate is invariant by changes to the Res wires: any two well-formed microstates equal up to Res values have the same execution steps up to Res value computation.*



If we denote by $\equiv_{\text{Res}}$ the equality of microstates up to Res wires, we can write this property formally as:

$$\begin{aligned} \forall p, q, p', \text{valid_coloring}(E, p) \implies \text{valid_coloring}(E, q) \implies \\ \text{Sel}(p) = - \implies p \equiv_{\text{Res}} q \implies \\ p \xrightarrow{E} p' \implies p' \equiv_{\text{Res}} q \vee \left(\exists q', q \xrightarrow{E} q' \wedge p' \equiv_{\text{Res}} q' \right). \end{aligned}$$

We do not need the hypothesis $\text{Sel}(q) = -$ because $p \equiv_{\text{Res}} q$ entails $\text{Sel}(p) = \text{Sel}(q)$. The disjunction in the conclusion can be understood as follows: either the microstep from p to p' computed a Res value, in which case we have $p' \equiv_{\text{Res}} p \equiv_{\text{Res}} q$ (left disjunct), or it computed something else and q can mimic it (right disjunct). This result justifies why the notations introduced in Section 7.1.3 do not consider Res when $\text{Sel} = -$.

7.5 Refinement between the constructive state semantics and the microstep semantics

In order to reduce the proof burden and focus on control and completion codes, some parts of the circuit are not modeled in the microstep semantics, namely the computation of signal values and the synchronizer for the parallel statement. In the first case, it amounts to a big OR gate over the Go component of the input color of all emitters. In the second case, we work with the specification of the synchronizer rather than a given implementation, thus permitting to reason about what it *should* do and allowing us to swap and compare implementations.

Unlike the previous semantics which tackle reactions, the microstep one deals with steps *within* one reaction. This discrepancy makes it harder to relate both kinds of semantics because the microstep semantics is much more precise and can distinguish circuit states that cannot be expressed in the state semantics and whose translations in the CSS would be identical. Nevertheless, the biggest obstacle when connecting the constructive state semantics and the microstep semantics comes from the *Can* and *Must* functions.

The state semantics uses the auxiliary functions *Can* and *Must* to compute the value of a local signal then performs the evaluation using this value. Thus, the body of the statement is evaluated twice: first partially to get the value of the local signal and later from scratch again to compute the full step using the value of the local signal. On the contrary, in the microstep semantics, once the value of the local signal is computed, we continue the evaluation of the microstate using this value *without restarting from scratch*, exactly as a circuit does. This is the main difference with the microstep semantics presented in the *Compiling Esterel* book [44].

In order to complete the simulation, we need to translate the condition for applying a rule for local signals, $s \in \text{Must}_s(p, E)$ or $s \notin \text{Can}_s^+(p, E)$, into the corresponding condition in the microstep semantics, that is: there exists a microstep sequence $p \xrightarrow{E}^* p'$ for some p' emitting s or

some p' surely not emitting it. This is actually the hardest part about the simulation proof, which makes sense as it is also an essential difference between the semantics. Because the definitions of *Must* and *Can* are mutually recursive with recursive calls for $p; q$ requiring information about completion codes and recursive calls for $s?p, q$ changing the tag $+$ or $\perp$ on *Can*, we actually need six mutually recursive properties: three for signals and three for completion codes.



► **Lemma 25** (Interpretation of *Must/Can* as microsteps).

For any statement p and any event E containing all signals used by p , i.e., satisfying the $\text{valid_dom}(E, p)$ predicate, we have the following properties. (Recall that $[\square^+]p$ denotes replacing the input color of p by $\square^+$ or, more intuitively, starting the execution of p .)

- Any signal that must be emitted is indeed emitted after enough microsteps:
 $\forall s, s \in \text{Must}_s(p, E) \implies \exists p', [\square^+] \text{from_stmt}(p) \xrightarrow[E]{*} p' \wedge s^+ \in \text{to_event}(p', E) ;$
- Any signal that cannot be emitted is indeed not emitted after enough microsteps:
 $\forall s, s \notin \text{Can}_s^+(p, E) \implies \exists p', [\square^+] \text{from_stmt}(p) \xrightarrow[E]{*} p' \wedge s^- \in \text{to_event}(p', E) ;$
- Any signal that can never be emitted is indeed not emitted no matter what the input color is:
 $\forall s, s \notin \text{Can}_s^\perp(p, E) \implies \forall \blacksquare, \exists p', [\blacksquare] \text{from_stmt}(p) \xrightarrow[E]{*} p' \wedge s^- \in \text{to_event}(p', E) ;$
- If the completion code must be k , then it is indeed k after enough microsteps:
 $\forall k, k \in \text{Must}_k(p, E) \implies \exists p', [\square^+] \text{from_stmt}(p) \xrightarrow[E]{*} p' \wedge \text{out}(p') = \bullet_k ;$
- After enough microsteps, the possible completion codes are a subset of what they can be:
 $\exists p', [\square^+] \text{from_stmt}(p) \xrightarrow[E]{*} p' \wedge \text{out2C}(\text{out}(p')) \subseteq \text{Can}_k^+(p, E) ;$
- Regardless of input color, after enough microsteps, the possible completion codes are a subset of what they can be when not knowing if the statement is evaluated or not:
 $\forall \blacksquare, \exists p', [\blacksquare] \text{from_stmt}(p) \xrightarrow[E]{*} p' \wedge \text{out2C}(\text{out}(p')) \subseteq \text{Can}_k^\perp(p, E).$

where out2C is a function converting an output color to a set of completion codes, defined by $\text{out2C}(\bullet_k) = \{k\}$ and $\text{out2C}(\circ_K) = K$.



Six similar properties hold when resuming an active state rather than starting an inactive statement.

Once we can translate *Must* and *Can* into a sequence of microsteps, we can prove the simulation theorem: any step in the constructive state semantics can be performed by a sequence of steps in the microstep semantics. Furthermore, the final microstate is total and produces the same completion code and the same output event:

► **Theorem 26.** For all p, E, E', k , and $\bar{p}'$, if $p \xrightarrow[E]{E', k} \bar{p}'$ then there exists a total microstate p'' such that



$$[\square^+](\text{from_stmt}(p)) \xrightarrow[E]{*} p'', \quad \text{to_term}(p'', E) = \bar{p}', \quad \text{out}(p'') = k \quad \text{and} \quad \text{to_event}(p'', E) = E'.$$

For all $\hat{p}, E, E', k$, and $\bar{p}'$, if $\hat{p} \xrightarrow[E]{E', k} \bar{p}'$ then there exists a total microstate p'' such that



$$[\blacksquare^+](\text{from_state}(\hat{p})) \xrightarrow[E]{*} p'', \quad \text{to_term}(p'', E) = \bar{p}', \quad \text{out}(p'') = k \quad \text{and} \quad \text{to_event}(p'', E) = E'.$$

Proof. The proof amounts to finding an evaluation order leading from the microstate at the beginning of the instant to the one at the end of the instant. This order corresponds to input-to-output evaluation in circuits: first using start rules, then context ones,⁸ and finally end ones. Lemma 25 and confluence are critical for the $p \setminus s$ case. We use Lemma 23 (Execution of non-executed microstates) to handle the parts of the terms which are not executed. ◀

⁸ For $p; q$, the rule transmitting the output of p to the input of q is naturally used between the evaluations of p and q .

■ **Table 1** Properties of the various Esterel Semantics.

Semantics Properties	Logical Semantics	Constructive Semantics	Constructive State Semantics	Microstep Semantics
Deterministic	No	Yes	Yes	No
Confluent	No	Yes	Yes	Yes
Total output	Yes	Yes	Yes	N/A
Inactive derivative	weak	weak	strong	N/A
Statement invariance	No	No	Yes	Yes

Notice that this theorem is not an equivalence between the semantics, only a simulation. The converse direction cannot hold because the microstep semantics is local whereas the state one is global. For example, if we take a valid statement p and a non constructive one q (say, $(s?0, !s) \setminus s$), then $p \mid q$ cannot execute under the state semantics whereas the microstep one can execute p independently of q , so that the final microstate would correspond to some $p' \mid q$ in the state semantics.

8 Properties and interpreters of Esterel semantics in Coq

8.1 Summary of the properties of Esterel semantics

Most semantics satisfy a common body of properties, inherent to the Kernel Esterel language, that we recall now. These properties are presented below only for the constructive semantics but can be adapted to other semantics in a straightforward way (see the Coq files of each semantics). Table 1 gives an overview of which properties are satisfied by which semantics.

Determinism The semantics is deterministic:

$$\forall p E E'_1 E'_2 k_1 k_2 p'_1 p'_2, \quad p \xrightarrow[E]{E'_1, k_1} p'_1 \implies p \xrightarrow[E]{E'_2, k_2} p'_2 \implies p'_1 = p'_2 \wedge E'_1 = E'_2 \wedge k_1 = k_2$$

Notice that this property does not hold for the logical semantics! For example, the program $P1 = s? !s, 0$ presented in the introduction is non-deterministic. The microstep semantics is also non-deterministic because execution can happen in various places of a microstate. Nevertheless, it is confluent so that in practice this non-determinism is not really an issue.

Total output Output events are always total, that is, they contain only mappings to $+$ or $-$ and none to $\perp$. This is trivially true for the logical semantics, as there is no $\perp$ involved.

$$\forall p E E' k p', p \xrightarrow[E]{E', k} p' \implies \text{Total}(E') \quad \text{with} \quad \text{Total}(E') := \forall s \in \text{dom}(E'), E'(s) \neq \perp$$

This property does not make sense for the microstep semantics as there is no output event.

Inactive derivative If the completion code is not 1, then the derivative is 0:

$$\forall p E E' k p', p \xrightarrow[E]{E', k} p' \implies k \neq 1 \implies p' = 0$$

This result means that if a statement is not pausing, then its execution has finished (either by normal termination or raising an exit). This result is the motivation for introducing the $\delta(k, p)$ function.

Underlying statement invariance The underlying statement does not change during execution:

$$\forall p E E' k p', p \xrightarrow[E]{E', k} p' \implies \mathcal{B}(p) = \mathcal{B}(p')$$

This property is true only for the state and microstep semantics.



Of course, not all properties are shared between all semantics, some properties are still specific to each semantics. For example, the result “Inactive derivative” above can be strengthened into an equivalence for the constructive state semantics, as the derivative of 1 (**pause**) is no longer an inactive statement but the active state $\hat{1}$.



► **Lemma 27** (Strong inactive derivative). *For all p, E, E', k , and p' , if $p \xrightarrow[E]{E', k} \overline{p'}$ then we have $k \neq 1 \iff \overline{p'} = p$.*



For all $\hat{p}, E, E', k$, and p' , if $\hat{p} \xrightarrow[E]{E', k} \overline{p'}$ then we have $k \neq 1 \iff \overline{p'} = \mathcal{B}(\hat{p})$.

8.2 Interpreters

For all semantics, we have defined interpreters in Coq and proven their correctness and completeness with respect to the corresponding SOS semantics. These interpreters can be used as an executable semantics of Kernel Esterel programs. They take as input a program (either as a statement, a state, or a microstate depending on the interpreter) and an input event; they output either nothing (**None**) when the input program cannot execute, or the output event, completion code and derivative of a possible execution step.

As an illustration, below is the correctness statement for the interpreter of the constructive behavioral semantics. The same type of result holds for both variants (start and resumption) of the constructive state semantics.



► **Theorem 28** (Correctness of the CBS interpreter). *For all p, E, E', k , and p' , if all signals of p belong to E (i.e., $\text{valid_dom}(p, E)$ holds), then we have $\text{CBS_interp}(p, E) = \text{Some}(E', k, p') \iff p \xrightarrow[E]{E', k} p'$.*

Proof. All such proofs are straightforward (if sometimes long) and follow the structure of the interpreter or of the semantics in the premise. See the Coq files of each semantics for details. ◀

Since the microstep semantics may execute in several parts of a microstate, we need to choose an evaluation order, even though the actual order does not matter because of confluence. We use the start-context-end order: first start rules, then context rules, then end rules. Because of this choice, the completeness result no longer holds (one could pick a different order that the interpreter does not follow) but instead we have a weaker result: if a microstep execution step is possible, then the interpreter is not stuck.

► **Lemma 29** (Correctness and partial completeness of the microstep interpreter).



$$\begin{aligned} \forall p, E, p', \text{micro_interp}(p, E) = \text{Some } p' &\implies p \xrightarrow[E]{} p' \\ \forall p, E, p', p \xrightarrow[E]{} p' &\implies \text{micro_interp}(p, E) \neq \text{None} \end{aligned}$$

We can recover a complete interpreter `micros_interp` by using the iterated microstep semantics, that is, the reflexive transitive closure of the microstep semantics. Then, all microstates can execute (thanks to reflexivity) so we can drop **Some/None** and have a simpler correctness statement:



$$\forall p, E, p \xrightarrow[E]^* \text{micros_interp}(p, E) \quad \wedge \quad \forall p', \neg(\text{micros_interp}(p, E) \xrightarrow[E]{} p')$$

This statement means that any microstate p can execute up to `micros_interp(p, E)` and no further. Confluence ensures that `micros_interp(p, E)` is the final microstate of any execution chain starting from p in the environment E .

Finally, the interpreter for the logical semantics is only correct but not complete as the logical semantics is non deterministic and non confluent. Indeed, given a statement p that can non-deterministically pause or terminate and a statement q that cannot execute, then on $p; q$ the interpreter will choose to make p terminate and fail to execute q making the overall $p; q$ statement non executable. On the contrary, the LBS semantics allows p to pause, which results in a valid reaction as q is not considered. For instance, one may take p to be $(s ? !s, 1) \backslash s$ and q to be $(s ? 0, !s) \backslash s$.

► **Remark 30.** It would be possible to create a complete interpreter for the LBS semantics by returning the set of possible executions instead of just choosing one. This seems to be of little practical interest as the LBS semantics is not very useful anyway.

8.3 Proof effort

Table 2 presents the size of the Coq formalization. The code is organized into three sub-directories:

Util	auxiliary definitions, in particular notations and the representation of events;
Semantics	definitions of all semantics with basic utilitarian proofs; and
Proofs	main proofs, notably equivalence/simulation between semantics and confluence of microsteps.

In the semantics definitions, we can observe that the first three semantics (LBS, CBS, CSS) are defined with their main properties in a few hundred lines (files `Semantics/LBS.v`, `Semantics/CBS.v` and `Semantics/CSS.v`), the CSS being a bit bigger because it features two sets of rules: start and resumption. On the other hand, the microstep semantics (file `Semantics/Microstep.v`) is twice as big, and even the definition of microstates (files `Semantics/InputColor.v`, `Semantics/OutputColor.v` and `Semantics/Microstate.v`) is three-times bigger than the definitions of statements and states (file `Definitions.v`). For each semantics, about half of these numbers are due to the interpreters and their proofs of correctness with respect to the SOS semantics.

Among the proofs, the largest ones are by far local confluence of the microstep semantics (file `Proofs/MicroConfluence.v`, 1400 l.) and the link between *Must/Can* and microsteps for the start and resumption cases (file `Proofs/MicroMustCan.v`, 1500 l. for each). They are used in the simulation proof between the CSS and the microstep semantics, which is comparably short (file `Proofs/CSS_Micro.v`, 450 l. for each of the start and resume variants). The proof of local confluence is done naively by a big case analysis on each possible execution step, in total there are 280 cases to consider. It is very likely that it could be considerably shortened by a smarter decomposition: half of the cases are symmetric and many rules simply commute.

9 Conclusion and future work

This article has presented new work on the Coq-based formal verification of the Kernel Esterel semantics chain presented in the web draft book [4] written by the second author 20 years ago, augmented by the definition of a new fine-grained operational semantics developed and formally verified by the first author. Unlike the coarser-grained operational semantics introduced by Dumitru Potop in [44], this new operational semantics closely mimics the circuit translation introduced in [4], which was the basis of both the academic Esterel v5 compiler to C and the industrial Esterel v7 compiler to hardware circuits or software code. Its main advantage compared to circuits is its SOS inductive logical rules presentation, which makes the formal proof of its adequacy with respect to the constructive semantics much simpler with current Coq technology.

■ **Table 2** Size of the Coq formalization, as given by `coqwc`, ordered by dependency within each folder.

Spec	Proof	Comments	File
141	104	13	Util/Coqlib.v
28	130	25	Util/MapSig.v
274	651	9	Util/MapList.v
45	0	5	Util/Notations.v
171	201	22	Util/Events.v
198	251	38	Util/SemanticsCommon.v
857	1337	112	Total for Util/
262	200	40	Semantics/LBS.v
60	605	118	Semantics/MustCan.v
163	515	94	Semantics/CBS.v
52	160	21	Semantics/StateMustCan.v
318	812	113	Semantics/CSS.v
58	85	11	Semantics/InputColor.v
248	213	43	Semantics/OutputColor.v
552	740	88	Semantics/Microstate.v
396	380	138	Semantics/Microstep.v
158	983	118	Semantics/Microsteps.v
2267	4693	784	Total for Semantics/
58	348	51	Proofs/CBS_LBS.v
9	274	61	Proofs/CBS_CSS.v
245	1101	191	Proofs/ValidColoring.v
23	1382	81	Proofs/MicroConfluence.v
17	27	0	Proofs/MicrostepsFacts.v
80	3032	211	Proofs/MicroMustCan.v
11	821	97	Proofs/CSS_Micro.v
98	200	15	Proofs/SurfaceIgnoresRes.v
541	7185	707	Total for Proofs/
311	98	28	Definitions.v
3976	13313	1631	Total

Let us present a short history of this work. When he completed the book [4], the second author did not even include sketches of correctness proofs, for two reasons. First, he was completely involved in the industrial development and applications of the Esterel v7 compiler, which handles a much more expressive and modular language than v5. The v7 language and compiler have been successfully used by several hardware and software companies, before being taken over by Synopsys (they are no longer available). Second, he thought that only machine-checked correctness proofs would be fully convincing, but the formal verifiers were not yet mature enough at the beginning of the 2000's. The real game changer in the semantics/compiling field was the Coq-based construction and formal verification of the CompCert compiler by Xavier Leroy and his team [35]. This large-scale work was itself inspired by the Coq-based formal proof of a major mathematical theorem by Georges Gonthier and his team [29] (Gonthier himself had played a major role in the early development of Esterel). Our formal proofs, technically developed by the first author, followed a similar track. No mistakes were found in the book's claims, apart from the occasional typo, although it did not even contain informal correctness proofs.

Nevertheless, it must be noted that this article does not finish the job, for three reasons: first, the current Coq proofs limit the kernel language to loop-free programs; second, the new operational semantics is not yet formally related to the Boolean circuits generation described in [4], which is the heart of the v5 and v7 compilers. We briefly detail these points below. Third, but less importantly, we did not yet study the full Esterel language, which also supports data definitions and calculations. As shown by the way the v5 and v7 compilers handle data, this only requires adding data dependencies to the signal dependency structure, which should raise no conceptual issues.

9.1 Future work: statement and signal reincarnation

We chose to postpone the study of loops because they lead to a subtle phenomenon: *statement and signal reincarnation*. When embedded in possibly nested loop statements, a given Esterel statement can be reincarnated (i.e., re-executed) several times in the same instant, but in a clean way: a new incarnation can only occur when the previous one has terminated. Consequently, a locally scoped signal may take several simultaneous incarnation statuses in the same instant when its declaration is embedded in loops. But these are taken in strict succession of scope entering and exiting, as are the reincarnations of its declaration statements; this makes only one signal incarnation status visible at a time in any statement. Thus, several incarnations can appear but each will disappear before the next one appears. Reincarnation occurs naturally when programming, and raises no difficult problems in practice.

Three quite different technical solutions have been developed to deal with reincarnation. The first naive one consists of syntactically duplicating the body of each loop in an inside-out way, which completely suppresses reincarnation since Esterel requires that the body of a loop cannot terminate instantaneously. In p^* , the loop body p may terminate and be restarted at the same instant. But in $(p;p)^*$, since p cannot terminate instantaneously, the first and second p 's cannot terminate in the same instant. But this trivial solution is impractical since inside-out copying may exponentially increase the source code size.

In [4], the second author used a more semantic way based on *incarnation indices* that tag in a unique way each occurrence of a given statement or signal in execution proof trees. Since restarting a statement p can only execute its *surface*, i.e., the statements instantaneously reachable when p starts, such indices only lead to worst-case proof tree and generated circuit sizes that are quadratic compared to the statement size. Furthermore, the size increase is only quasi-linear in practice for most useful programs.

In [53], Olivier Tardieu introduced an alternative approach based on a semantically-guided partial syntactic duplication of statements, which copies the surfaces of source statements also in a quadratic worst-case and most often quasi-linear way. The price to pay is the addition of a **gotopause** statement, i.e., a **goto** statement limited to **pause** targets. Both approaches are compatible, since Tardieu's approach can also be viewed as a way of duplicating statements based on a static conservative calculation of possible incarnation indices.

It is not yet clear which approach should be preferred to complete our Coq-based verification. Incarnation indices have a simple and clean algebraic structure, which should make them easy to add to the current semantic rules. But the proofs will become heavier since they will require the addition of an extra decreasing incarnation index-related ordinal. Tardieu's approach may look simpler at first glance, because the proof might require less changes, but it has another drawback: the addition of **gotopause** breaks locality of programs because a **gotopause** may activate a **pause** arbitrarily far away, making SOS semantics and inductive reasoning less adequate.

9.2 Relating the operational semantics to the circuit translation

The relation between the microstep semantics and the circuit semantics is the only missing step to prove the compiler correct: by combining the simulation results of this article and this last missing simulation, one gets a proof of semantic preservation between the original Esterel program seen in the constructive semantics and its translation as a synchronous digital circuit seen in the circuit semantics.

Let us now finally analyze the technical difficulty of relating SOS semantics to circuits. Intuitively, the circuit generated by an Esterel program by the translation of [4] can be viewed as a folding of all possible proof trees of the new microstep operational semantics into a single and possibly cyclic graph of Boolean gates and wires (note that static cycles yield no dynamic electrical problems for constructive circuits, see [40]). Registers implement `pause` statements, with value 1 when a pause is active in a given program state or 0 when it is inactive. Specific gate clusters represent proof-tree nodes, and specific wire groups represent proof-tree branches, taking value sets that precisely encode the input and output colors attached to statements in the new operational semantics. In theory, making formal this informal correspondence should raise no real problem. But, as the saying goes: “In theory, theory and practice are the same and in practice, they are not”. The unfortunate practical problem is that finding a circuit representation in Coq well-suited to this formalization work is critical to its success.

References

- 1 Charles André. Representation and analysis of reactive behaviors: a synchronous approach. In *Proc CESA'96, IEEE-SMC, Lille, France*, 1996. URL: https://www-sop.inria.fr/members/Charles.Andre/CA%20Publis/Cesa96/SyncCharts_Cesa96.pdf.
- 2 Gérard Berry. The Esterel language primer, version v5_91. Technical report, Ecole des mines de Paris and Inria, June 2000. URL: https://www.college-de-france.fr/media/gerard-berry/UPL8106359781114103786_Esterelv5_primer.pdf.
- 3 Gérard Berry. The foundations of Esterel. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000. URL: <ftp://ftp-sop.inria.fr/meije/esterel/papers/foundations.pdf>.
- 4 Gérard Berry. The Constructive Semantics of Pure Esterel - Draft Version 3, 2002. URL: <https://hal.science/hal-04963343>.
- 5 Gérard Berry, Amar Bouali, Robert de Simone, Xavier Fornari, Emmanuel Ledinot, , and Éric Nassor. Esterel: a formal method applied to avionic software development. *Science of Computer Programming*, 36:5–25, 2000. doi:10.1016/S0167-6423(99)00015-5.
- 6 Gérard Berry and Laurent Cosserat. The ESTEREL synchronous programming language 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. doi:10.1007/3-540-15670-4_19.
- 7 Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science Of Computer Programming*, 19(2):87–152, 1992. doi:10.1016/0167-6423(92)90005-V.
- 8 Gérard Berry, Michael Kishinevsky, and Satnam Singh. System level design and verification using a synchronous language. In *2003 International Conference on Computer-Aided Design, ICCAD 2003, San Jose, CA, USA, November 9-13, 2003*, pages 433–440. IEEE Computer Society / ACM, 2003. doi:10.1109/ICCAD.2003.1257813.
- 9 Gérard Berry, Sabie Moisan, and Jean-Paul Rigault. Towards a synchronous and semantically sound high level language for real-time applications. In *IEEE Real Time Systems Symposium*, pages 30–40. IEEE Catalog 83 CH 1941-4, 1983.
- 10 Gérard Berry and Manuel Serrano. Hiphop.js: (a)synchronous reactive web programming. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 533–545. ACM, 2020. doi:10.1145/3385412.3385984.
- 11 Gérard Berry and Ravi Sethi. From regular expressions to deterministic automata. *Theoretical computer science*, 48:117–126, 1986. doi:10.1016/0304-3975(86)90088-5.
- 12 Timothy Bourke, Léo Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. A formally verified compiler for Lustre. In *PLDI 2017: Programming Language Design and Implementation*, pages 586–601. ACM, 2017. doi:10.1145/3062341.3062358.
- 13 Frédéric Boussinot. Reactive C: an extension of C to program reactive systems. *Softw. Pract. Exp.*, 21(4):401–428, 1991. doi:10.1002/spe.4380210406.

- 14 Janusz A. Brozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, 1964. doi:10.1145/321239.321249.
- 15 Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 100(8):677–691, 1986/8. doi:10.1109/TC.1986.1676819.
- 16 Janusz A. Brzozowski and Carl-Johan H. Seger. *Asynchronous Circuits*. Monographs in Computer Science. Springer, 1995. doi:10.1007/978-1-4612-4210-9.
- 17 Paul Caspi and Marc Pouzet. Lucid Synchrone: une extension fonctionnelle de Lustre. In *Journées Francophones des Langages Applicatifs (JFLA)*, Morzine-Avoriaz, France, February 1999. INRIA. URL: <https://hal.science/hal-01574464>.
- 18 Adrien Champion, Alain Mebsout, Christoph Stickel, and Cesare Tinelli. The kind 2 model checker. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, pages 510–517, Cham, 2016. Springer International Publishing. doi:10.1007/978-3-319-41540-6_29.
- 19 Van Chan Ngo, Jean-Pierre Talpin, and Thierry Gautier. Translation validation for synchronous data-flow specification in the signal compiler. In Susanne Graf and Mahesh Viswanathan, editors, *Formal Techniques for Distributed Objects, Components, and Systems*, pages 66–80, Cham, 2015. Springer International Publishing. doi:10.1007/978-3-319-19195-9_5.
- 20 Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. SCADE 6: A formal language for embedded critical software development (invited paper). In Frédéric Mallet, Min Zhang, and Eric Madelaine, editors, *11th International Symposium on Theoretical Aspects of Software Engineering, TASE 2017, Sophia Antipolis, France, September 13-15, 2017*, pages 1–11. IEEE Computer Society, 2017. doi:10.1109/TASE.2017.8285623.
- 21 The Coq Development Team. *The Coq Reference Manual, version 8.15.2*, May 2022. Available electronically at <http://coq.inria.fr/doc/V8.15.2/refman/>.
- 22 Laurent Cosserat. Sémantique opérationnelle du langage synchrone Esterel. Thèse de doctorat-ingénieur, Université de Nice, 1985.
- 23 Philippe Couronné. *Conception et implémentation du système Esterel v2*. Thèse d'informatique, Université Paris VII, 1988.
- 24 Nicolaas Govert de Bruijn. Lambda calculus notation with nameless dummies: A tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972. doi:10.1016/1385-7258(72)90034-0.
- 25 Stephen A. Edwards. An esterel compiler for large control-dominated systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(2):169–183, 2002. doi:10.1109/43.980257.
- 26 Bernard Espiau and Ève Coste-Manière. A synchronous approach for control sequencing in robotics applications. In *Proc. IEEE International Workshop on Intelligent Motion, Istanbul*, pages 503–508, 1990.
- 27 Esterel EDA Technologies. The Esterel v7_60 reference manual. Technical report, Esterel EDA Technologies, 2008. URL: https://www.college-de-france.fr/media/gerard-berry/UPL681247605558671166_Esterelv7.60_ReferenceManual.pdf.
- 28 Georges Gonthier. Sémantique et modèles d'exécution des langages réactifs synchrones; application à Esterel. Thèse d'informatique, Université d'Orsay, 1988.
- 29 Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A machine-checked proof of the odd order theorem. In *International Conference on Interactive Theorem Proving (ITP)*, pages 163–179. Lectures Notes in Computer Science 7998, Springer Verlag, 2013. doi:10.1007/978-3-642-39634-2_14.
- 30 Paul Le Guernic, Thierry Gautier, Michel Le Borgne, and Claude Le Maire. Programming real-time applications with SIGNAL. *Proc. IEEE*, 79(9):1321–1336, 1991. doi:10.1109/5.97301.
- 31 Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language LUSTRE. *Proc. IEEE*, 79(9):1305–1320, 1991. doi:10.1109/5.97300.
- 32 Nicolas Halbwachs, Fabienne Lagnier, and Christophe Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *IEEE Trans. Software Eng.*, 18(9):785–793, 1992. doi:10.1109/32.159839.
- 33 Gérard P. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems: Abstract properties and applications to term rewriting systems. *J. ACM*, 27(4):797–821, 1980. doi:10.1145/322217.322230.
- 34 Delphine Kaplan-Terrasse. Vers la certification du compilateur v5 d'esterel dans coq. Technical report, Inria, 2000. in French.
- 35 Xavier Leroy. A formally verified compiler backend. *J. Autom. Reason.*, 43(4):363–446, February 2009. doi:10.1007/s10817-009-9155-4.
- 36 Marten Lohstroh, Christian Menard, Soroush Bateni, and Edward A. Lee. Toward a lingua franca for deterministic concurrent systems. *ACM Trans. Embed. Comput. Syst.*, 20(4):36:1–36:27, 2021. doi:10.1145/3448128.
- 37 Sharad Malik. Analysis of cyclic combinational circuits. In Michael R. Lightner and Jochen A. G. Jess, editors, *Proceedings of the 1993 IEEE/ACM International Conference on Computer-Aided Design, 1993, Santa Clara, California, USA, November 7-11, 1993*, pages 618–625. IEEE Computer Society / ACM, 1993. doi:10.1109/ICCAD.1993.580150.
- 38 Louis Mandel and Marc Pouzet. ReactiveML, a reactive extension to ML. In *Proceedings of 7th ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming (PPDP'05)*, Lisbon, Portugal, 2005. doi:10.1145/1069774.1069782.

- 39 Florence Maraninchi. The Argos language: Graphical representation of automata and description of reactive systems. In *Proc. IEEE Conf. on Visual Languages, Kobe, Japan*, 1991.
- 40 Michael Mendler, Thomas Shiple, and Gérard Berry. Constructive Boolean circuits and the exactness of timed ternary simulation. *Formal Methods in System Design*, 40(3):283–329, 2012. doi:10.1007/s10703-012-0144-6.
- 41 Gary Murakami and Ravi Sethi. Terminal call processing in Esterel. In *Proc. IFIP 92 World Computer Congress, Madrid, Spain*, 1992. URL: https://www.researchgate.net/publication/264871837_Terminal_Call_Processing_in_Esterel.
- 42 Van Chan Ngo. *Formal Verification of a Synchronous Data-flow Compiler : from Signal to C. (Vérification Formelle d'un Compilateur Synchrone: de Signal vers C)*. PhD thesis, University of Rennes 1, France, 2014. URL: <https://tel.archives-ouvertes.fr/tel-01058041>.
- 43 Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the calculus of constructions. In *Mathematical Foundations of Programming Semantics*, pages 209–228, 1989. doi:10.1007/BFb0040259.
- 44 Dumitru Potop-Butucaru, Stephen A. Edwards, and Gérard Berry. *Compiling Esterel*. Springer, 2007. doi:10.1007/978-0-387-70628-3.
- 45 Claudius Ptolemaeus, editor. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, available as a free PDF download and low-cost paperback, 2014. URL: ptolemy.org.
- 46 Valérie Roy and Robert de Simone. Auto/autograph. In Edmund M. Clarke and Robert P. Kurshan, editors, *Computer Aided Verification, 2nd International Workshop, CAV '90, New Brunswick, NJ, USA, June 18-21, 1990, Proceedings*, volume 531 of *Lecture Notes in Computer Science*, pages 65–75. Springer, 1990. doi:10.1007/BFb0023720.
- 47 Klaus Schneider. Proving the equivalence of microstep and macrostep semantics. In Victor Carreño, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, 15th International Conference, TPHOLs 2002, Hampton, VA, USA, August 20-23, 2002, Proceedings*, volume 2410 of *Lecture Notes in Computer Science*, pages 314–331. Berlin, Heidelberg, 2002. Springer. doi:10.1007/3-540-45685-6_21.
- 48 Klaus Schneider. The synchronous programming language quartz. Technical report, Internal Report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, 2009. URL: <http://es.cs.uni-kl.de/publications/datarsg/Schn09.pdf>.
- 49 Ellen Sentovich, Horia Toma, and Gérard Berry. Latch optimization in circuits generated from high-level descriptions. In *Proc. IEEE International Conference on Computer Aided Design*, pages 428–435, 1996. doi:10.1109/ICCAD.1996.569833.
- 50 Manuel Serrano and Gérard Berry. Multitier programming in Hop. *Communications of the ACM*, 55(8):53–59, 2012. doi:10.1145/2240236.2240253.
- 51 Matthieu Sozeau, Simon Boulrier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. Coq coq correct! verification of type checking and erasure for coq, in coq. *Proc. ACM Program. Lang.*, 4(POPL):8:1–8:28, 2020. doi:10.1145/3371076.
- 52 Olivier Tardieu. *Loops in Esterel: From Operational Semantics to Formally Specified Compilers*. PhD, École Nationale Supérieure des Mines de Paris, September 2004. URL: <https://pastel.archives-ouvertes.fr/pastel-00001336>.
- 53 Olivier Tardieu and Robert de Simone. Loops in Esterel. *ACM Transactions on Embedded Computing Systems (TECS)*, 4(4):708–750, 2005. doi:10.1145/1113830.1113832.
- 54 Hervé Touati and Gérard Berry. Optimized controller synthesis using esterel. In *Proc. International Workshop on Logic Synthesis IWLS'93, Lake Tahoe*, 1993. URL: <ftp://ftp-sop.inria.fr/meije/esterel/papers/iwls93.pdf>.
- 55 Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O'Brien. Sccharts: sequentially constructive statecharts for safety-critical applications: Hw/sw-synthesis for a conservative extension of synchronous statecharts. In Michael F. P. O'Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 372–383. Edinburgh, UK, June 2014. ACM. doi:10.1145/2594291.2594310.
- 56 Reinhard von Hanxleden, Michael Mendler, Joaquín Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, and Owen O'Brien. Sequentially constructive concurrency - a conservative extension of the synchronous model of computation. In *Proc. Design, Automation and Test in Europe Conference (DATE'13)*, 2013. doi:10.7873/DATE.2013.128.