# Leibniz Transactions on Embedded Systems

**Volume 10 | Issue 1 | December 2025**

**ISSN 2199-2002**

*ACM Classification 2012*
Computer systems organization → Embedded software; Computer systems organization → Real-time languages; Computer systems organization → Real-time systems; Hardware → Theorem proving and SAT solving; Security and privacy → Logic and verification; Software and its engineering → Operational analysis; Software and its engineering → Real-time schedulability; Software and its engineering → Software testing and debugging; Software and its engineering → Syntax; Theory of computation → Operational semantics

*Published online and open access by*
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany.
Online available at
`https://www.dagstuhl.de/dagpub/2199-2002`.

*Publication date*
December 2025

*Bibliographic information published by the Deutsche Nationalbibliothek*
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at https://dnb.d-nb.de.



*Aims and Scope*
LITES aims at the publication of high-quality scholarly articles, ensuring efficient submission, reviewing, and publishing procedures. All articles are published open access, i.e., accessible online without any costs. The rights are retained by the author(s).

LITES publishes original articles on all aspects of embedded computer systems, in particular: the design, the implementation, the verification, and the testing of embedded hardware and software systems; the theoretical foundations; single-core, multi-processor, and networked architectures and their energy consumption and predictability properties; reliability and fault tolerance; security properties; and on applications in the avionics, the automotive, the telecommunication, the medical, and the production domains.



*Editorial Office*
Schloss Dagstuhl – Leibniz-Zentrum für Informatik
LITES, Editorial Office
Oktavie-Allee, 66687 Wadern, Germany
lites@dagstuhl.de

*Digital Object Identifier*
10.4230/LITES.10.1.0

`https://www.dagstuhl.de/lites`

# Contents

## Regular Papers

# List of Authors


Mohammad Ibrahim Alkoudsi (4)
RPTU Kaiserslautern-Landau, Kaiserslautern, Germany

Sanjoy Baruah (3)
Washington University, St. Louis, MO, United States

Gérard Berry (2)
Collège de France, Paris, France

Thidapat Chantem (3)
Virginia Tech, Blacksburg, VA, USA

Max Eisele (1)
Robert Bosch GmbH, Stuttgart, Germany; Saarland University, Saarbrücken, Germany

Nathan Fisher (3)
Wayne State University, Detroit, MI, USA

Gerhard Fohler (4)
RPTU Kaiserslautern-Landau, Kaiserslautern, Germany

Christopher Gill (3)
Washington University, St. Louis, MO, United States

Christopher Huth (1)
Robert Bosch GmbH, Stuttgart, Germany

Johannes Hägele (1)
Robert Bosch GmbH, Stuttgart, Germany

Damir Isovic (4)
Mälardalen University, Västerås, Sweden

Fatima Raadia (3)
Wayne State University, Detroit, MI, USA

Lionel Rieg (2)
Université Grenoble Alpes, CNRS, Grenoble INP - UGA, VERIMAG, Grenoble, France; Collège de France, Paris, France

Benjamin Standaert (3)
Washington University, St. Louis, MO, United States

Marion Sudvarg (3)
Washington University, St. Louis, MO, United States

Andreas Zeller (1)
CISPA Helmholtz Center for Information Security, Saarbrücken, Germany

# GDBMiner: Mining Precise Input Grammars on (Almost) Any System

**Max Eisele**
Robert Bosch GmbH, Stuttgart, Germany
Saarland University, Saarbrücken, Germany

**Johannes Hägele**
Robert Bosch GmbH, Stuttgart, Germany

**Christopher Huth**
Robert Bosch GmbH, Stuttgart, Germany

**Andreas Zeller**
CISPA Helmholtz Center for Information Security, Saarbrücken, Germany

## Abstract

If one knows the *input language* of the system to be tested, one can generate inputs in a very efficient manner. Grammar-based fuzzers, for instance, produce inputs that are syntactically valid by construction. They are thus much more likely to be accepted by the program under test and to reach code beyond the input parser.

Grammar-based fuzzers, however, need a *grammar* in the first place. *Grammar miners* are set to extract such grammars from programs. However, current grammar mining tools place huge demands on the source code they are applied on, or are too imprecise, both preventing adoption in industrial practice.

We present GDBMiner, a tool to mine input grammars for binaries and executables in *any* (compiled) programming language, on *any* operating system, using *any* processor architecture, even without source code. GDBMiner leverages the GNU debugger (GDB) to step through the program and determine which code locations access which input bytes, generalizing bytes accessed by the same location into grammar elements.

GDBMiner is slow, but versatile – and precise: In our evaluation, GDBMiner produces grammars as precise as the (more demanding) *Cmimid* tool, while producing more precise grammars than the (less demanding) *Arvada* black-box approach. GDBMiner can be applied on any recursive descent parser that can be debugged via GDB and is available as open source.

**2012 ACM Subject Classification** Software and its engineering → Syntax; Software and its engineering → Operational analysis; Software and its engineering → Software testing and debugging; Computer systems organization → Embedded software

**Keywords and phrases** program analysis, testing, input grammar, fuzzing, grammar mining

**Digital Object Identifier** 10.4230/LITES.10.1.1

**Supplementary Material** *Software (Source Code and Experiment Data)*: `https://github.com/boschresearch/gdbminer`, archived at `swh:1:dir:0f3e6bf68a0005a1ce1f93cf7389ad4f23f2ec6e`

**Funding** This work was funded by the German Federal Ministry of Education and Research (BMBF, project CPSec - 16KIS1565 and 16KIS1564K).
*Max Eisele*: Max Eisele conducted this research as part of his PhD thesis at Saarland University.
*Johannes Hägele*: Johannes Hägele conducted this research as part of his Bachelor thesis at Saarland University.

**Received** 2024-05-28 **Accepted** 2025-01-10 **Published** 2025-04-16

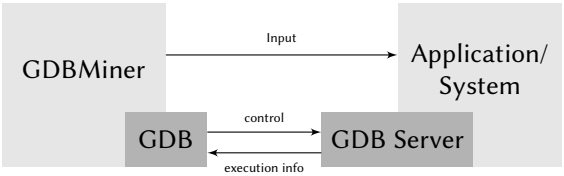



**Figure 1** GDBMiner connects to a *GDB*-compliant *GDB Server* in order to set data watchpoints and trace the execution of *inputs* within an *Application* or *System*.

## 1 Introduction

In industry, proper software testing is the number one technique to satisfy safety, security, and privacy requirements. As manual testing is laborious and code constantly grows [42], there is a huge demand for automated software testing solutions.

Having the *input specification* of a program under test can help to automate testing. A *context-free grammar,* for instance, can be used to produce large amounts of valid and diverse test inputs [20]. However, formal input specifications are rarely available in practice; and writing grammars by hand is cumbersome.

Recent research has demonstrated that input grammars can be *mined* from inputs and programs:

- **Autogram** [23] traces data flow of different parts of the input into functions and variables and compiles resulting rules to a context-free grammar. However, it is implemented for Java programs only, and therefore, unsuited for a variety of systems and programs.
- **Mimid** [18] leverages control flow, as well as data flow instrumentation of the target program to reconstruct derivation trees, and subsequently uses a number of active learning steps – basically testing for interchangeability of subtrees – to translate these into a grammar. Mimid requires the program code to be written in Python or C (Cmimid). However, to be used with Cmimid, the C code cannot have *gotos, macros,* or *enumerations;* case statements without *braces* or with *fall-throughs* are not allowed. While such restrictions pose little problems for the proof-of-concept of a prototype, they make Cmimid impractical for industrial use.
- **Arvada** [28] and **TreeVada** [5] are *black-box* approaches that only require oracle requests (tests whether an input is accepted by the target program) to approximate a grammar. They therefore offer an approach to grammar mining that only requires the ability to execute a program. However, by construction, Arvada, and TreeVada have less information available than code-based approaches, and the resulting grammars may thus be less precise.

To mine input grammars in our context, we present a novel approach that does not suffer from the above limitations – and actually is set to be applicable for binaries and executables in *any* programming language, on *any* operating system, using *any* processor architecture, even without source code. The realm of embedded systems is not well definable, but it is clear that they are computerized systems build for specific purposes [48]. Compared to traditional computing systems, the hardware of embedded systems is generally speaking more constrained and diverse. However, the program execution on embedded systems is often controllable via debuggers, which is why the key ingredient of our approach is to use the *GNU debugger (GDB)* [46] as interface to the program under test, allowing a grammar miner to:

1. *Step* through the program, identifying the code executed.
2. Use *watchpoints* to *track* data accesses during execution.

These features give a grammar miner the ability to *associate* input data with *code locations* that process them. This, in turn, allows a grammar miner to *group* input bytes that are processed by the same code into *equivalence classes* and hence *grammar elements.* Also taking the call stacks of the processing code into account, we can produce precise and human-readable input grammars.

Using GDB to step through executions is time-consuming – but once a grammar is mined from a system, it can be used again and again for high-speed generation of valid inputs, offsetting any effort spent to mine the grammar in the first place. The main advantage of using GDB, however, is that it is *versatile,* providing a *unified interface* for most software and systems. GDB works with Assembly, C, and C++, popular languages for programming embedded systems, as well as Ada, Objective-C, Rust, Pascal, Modula-2, FORTRAN, Go, and many other compiled programming languages; it can also be applied on binaries without source code. It supports most microprocessor instruction sets, including ARM, x86, Motorola 68000, MIPS, PA-RISC, PowerPC, and RISC-V.

Furthermore, GDB supports *remote debugging*, where GDB runs on one machine, and the program under test runs on another. GDB then communicates via TCP with a remote *GDB Server* running on a debug probe or the host computer, e.g. to control the debug unit on a microcontroller that is part of an embedded system (Figure 1).

We have implemented the above grammar mining approach in an open source tool named GDBMiner, bringing input grammar mining to any recursive descent parser that can be debugged with GDB – from C-compiled executables on general-purpose PCs to read-only binaries on embedded systems.

With GDBMiner, we contribute a *unified, language- and architecture-agnostic approach* for mining input grammars on *any* system with debugging capabilities. As we show in our evaluation, GDBMiner produces input grammars that are precise, well-structured, and very suitable for test generation, especially for black-box testing. The versatility of GDBMiner and the universality of the resulting grammars enables efficient software testing under adverse conditions, including embedded systems, and thus, specifically addresses the needs of software engineering in practice.

The remainder of this paper is organized as follows. Section 2 summarizes known concepts and techniques, followed by our approach in Section 3. We cover implementation details in Section 4. Section 5 evaluates GDBMiner against the state of the art and contains a walkthrough of a demo application. Section 6 discusses limitations of our approach and Section 7 closes with conclusion and future work.

## 2 Background

The following background knowledge is essential for this paper.

### 2.1 Context-free Grammars

Context-free grammars are mathematical descriptions for context-free languages. They can serve for generating inputs that belong to the corresponding language or for verifying if an input belongs to that language. For instance, the language of correctly parenthesized expressions is a popular example of a context-free language. In contrast to regular languages, context-free languages can take care of these matching parentheses, which makes them ideal for describing mathematical expressions, but also the syntax of programming languages. Generating strings from a context-free grammar is easy, and deciding whether a string is producible by a given context-free grammar is efficient [11]. Limits of context-free languages are context-sensitive properties, for instance, arbitrary matching tags as they are used in the XML format, or checksums over the content.

Formally, a context-free grammar is a four-tuple $G = (V, T, P, S)$, with variable (or non-terminal) symbols $V$, terminal symbols $T$, the set of productions $P$, and the start symbol $S$ [22]. A production rule $p \in P$ is a relation $(v, x)$ with variable $v \in V$ mapped to a string of variables and terminals $x \in (V \cup T)^*$.

As an example, consider this grammar $G$ for correctly parenthesized expressions: $\langle S \rangle ::= \langle S \rangle\langle S \rangle \mid (\langle S \rangle) \mid \epsilon$. Applying $G$ on the string *'(()())'* yields the parse tree in Figure 2.

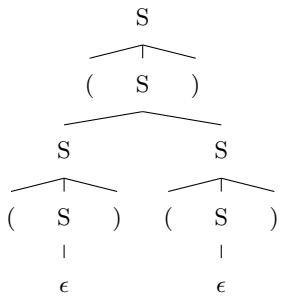



**Figure 2** Grammar parse tree for *'(()())'*.

**Listing 1** A recursive decent parser for correctly parenthesized expressions

```
1 void parse_S(const char *input, int *position) {
2   if (input[*position] == '(') {
3       (*position)++;
4       parse_S(input, position);
5       if (input[*position] != ')')
6           error("Expected␣closing␣parenthesis");
7       (*position)++;
8       parse_S(input, position);
9   }
10 }
```

## 2.2 Recursive Descent Parsers

Recursive descent parsers are a popular way to implement parsers for context-free grammars in practice [31]. They operate in a top-down manner, recursively calling parser subroutines that match non-terminals, and can precisely parse $LL(1)$ grammars [26] whose production rules are unambiguous when reading a string one by one. Having a context-free grammar, it is possible to generate sound recursive descent parsers automatically, for instance, with ANTLR [41].

Listing 1 shows a recursive descent parser written in C for the parenthesis language. The one single production rule is called recursively. Thus, the parser does not require loops for parsing, while it still can parse strings of arbitrary length (as long as the stack memory is not exhausted during execution).

## 2.3 Fuzzing with Grammars

Fuzzing is testing programs with huge amounts of randomly generated inputs while monitoring them for unwanted behavior, such as crashes, hangs, or other fault signals [30]. In conjunction with memory sanitizers, like AddressSanitizer [44], fuzzing can reveal real bugs in a wide variety of programs in an unsupervised way. For instance, Google's OSS-Fuzz initiative found more than 36,000 bugs in over 1,000 projects in the last seven years with fuzzing [17]. Since fuzzing is a dynamic software testing technique, it can only find bugs in code that it executes, which implies that a test engineer wants to achieve high code coverage during testing. Coverage-guided fuzzing therefore mutates known inputs and grows this collection with newly found code coverage-increasing inputs.

As an alternative, several approaches showed that grammars can be used to generate structured random inputs for fuzzing [45, 20], which have much higher chances of triggering deeper code in the target program. Grammar-based fuzzing does not need a coverage feedback mechanism, but it does need a grammar. As a result, grammar-based fuzzing is, in particular, interesting for testing embedded systems, where retrieving coverage feedback is hard [35, 49, 13].

## 2.4 Debuggers

Programmers use debuggers to investigate a program's behavior at runtime. Debugging tools therefore can externally control the execution of a program at the programmer's pace, as well as examine memory values, such as variables or the execution stack. Specifically,

- Breakpoints on instruction addresses interrupt the execution when reached.
- Watchpoints on memory addresses interrupt the execution on memory accesses.
- Single-stepping executes a single instruction at a time only.

The de facto standard protocol for debugging tools is the GNU Debugger (GDB) remote serial protocol [16], which allows connecting different debug backends to various debugging frontends. GDB distinguishes between *hardware breakpoints* that correspond to actual registers on the chip and *software breakpoints* that are realized by patching the binary with interrupt inducing instructions. The number of available hardware breakpoints is limited, but they can be set and reset easily. In contrast, software breakpoints are not limited in their amount, but require frequent rewriting of the program, which can be slow, can eventually wear out flash memory, or in case of read-only memory, may simply be impossible.

The same is true for *watchpoints* through which the debugger checks specific variables or memory regions for read and/or write access. *Hardware watchpoints* are efficient, but come in small numbers, if at all, whereas *software watchpoints* require interrupting execution with every step, with an even higher performance impact than software breakpoints.

Using debuggers, and particularly GDB, to enable dynamic software analysis for embedded systems has been shown to be a versatile and powerful approach, for instance by *Avatar 2* [34] and *GDBFuzz* [12]. The alternative to analyzing the execution on the target system itself, is to move the execution of the program into an emulator, where it can be instrumented and analyzed. However, this so-called re-hosting has its own drawbacks and difficulties, mainly due to the challenge of emulating not only the processor, but also all the hardware peripherals involved [15, 49].

## 2.5 Mining Input Grammars

Mining input grammars refers to deriving grammars from programs that match their input space best possible. Mimid [18], Arvada [28], and TreeVada [5] are currently, to the best of our knowledge, the three most effective methods to do so. Both require a set of valid seed inputs as a starting point. Seeds for a program are obtained by collecting example inputs that are available or by intercepting messages to the program during runtime.

Mimid [18] applies comprehensive instrumentation in terms of control and data flow instrumentation to the target code to find at which point of the execution the program *consumes* which bytes of the input data. It determines the consumption of a character using dynamic tainting to even track when the program accesses eventual copies of input bytes or extracted tokens of a lexer stage. For the control flow instrumentation, Mimid introduces a stack that, similar to a call stack, keeps track of active control flow scopes, such as *loops*, *if/else branches*, and *function calls*. From tracking the execution while parsing a set of valid seed inputs and in conjunction with the documented input data accesses, Mimid recovers derivation trees, which are already reminiscent of grammar parse trees. Mimid then searches for compatible nodes in these trees by swapping their subtrees and probing if the resulting new input is accepted by the target program. Mimid's source code aware approach enables the extraction of meaningful and human-readable labels from symbols, but it requires exhaustive instrumentation.

In contrast, Arvada [28] is a *black-box* approach which only requires oracle requests – tests whether an input is accepted by the target program or not. Also, based on a set of valid seed inputs, it tries to condense symbols in the input to *bubbles*, meaning they are assigned a new non-terminal

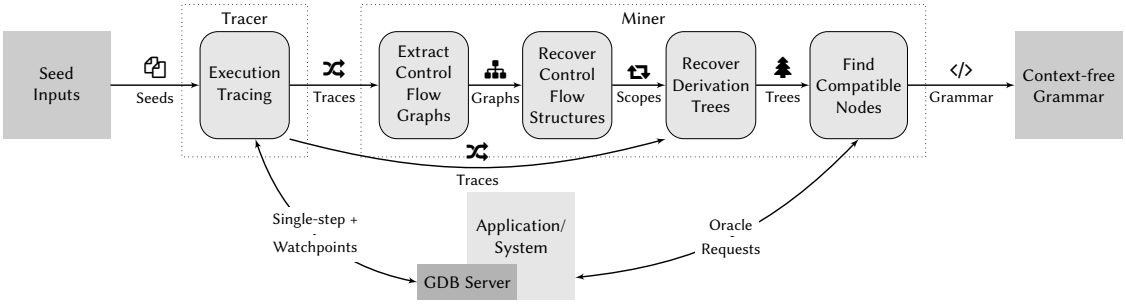



**Figure 3** How artifacts emerge through the different stages of GDBMiner. By tracing the execution of the *Seeds*, GDBMiner obtains *Traces*, from which it derives *Control Flow Graphs*. From the graphs, GDBMiner recovers *Control Flow Scopes*, which are required to reconstruct the *Derivation Trees*. Finally, GDBMiner extracts a *Grammar* through active learning.

symbol, and subsequently tries to *merge* different bubbles when they turn out to be compatible. Arvada chooses candidates for new bubbles randomly, which makes it a non-deterministic approach with respect to the seed inputs.

TreeVada [5] enhances Arvada's approach by building "on two soft assumptions, i.e., that many languages (1) use ' " quotes to wrap strings and (2) use ( ) [ ] { } brackets for nesting.". It prestructures the seed inputs, for instance by treating parts with matching brackets as a single unit, and then applies Arvada's approach to merge these units into bubbles. For the generalization step, TreeVada also creates only candidates with matching brackets or string quotes, which reduces the search space and increases the likelihood of finding compatible bubbles.

Polyglot [10] does not learn grammars from programs, but structures of binary protocol formats by leveraging execution tracing and dynamic taint analysis. It detects context-sensitive properties, such as length fields, but also keywords and seperator characters. Since Polyglot does not aim to learn context-free grammars, we consider it out of scope for this work.

## 3 Concept of GDBMiner

Similar to the Mimid algorithm, we exploit that the call stack of a recursive descent parser should express a branch of the derivation tree when a symbol is *consumed*; that is when the program processes a character of the input buffer last. The call stack in this case does not only contain the called functions but additionally all taken control flow decisions like conditional branches and loops. In contrast to Mimid, we relax the condition to consider a symbol to be consumed and just take the point where the program accesses the character in the input buffer last. This allows us to use data watchpoints for tracking accesses to the input buffer and therefore a programming language independent and source code agnostic approach. Moreover, it allows us to run the method on arbitrary systems that offer standard debugging capabilities.

Figure 3 shows the different stages of GDBMiner for mining grammars. In short, we use the *single-step* feature of the debugger to trace the execution of the targeted program and additionally log accesses to the input buffer with watchpoints. Based on the tracing data, we first reconstruct all control flow graphs of the involved functions, detect loops and conditional branches, and reconstruct derivation trees for each input. If the system under test offers only a limited amount of hardware watchpoints, we trace the same input multiple times with a sliding window of available watchpoints. Finally, we take the recovered derivation trees and apply the Mimid mining algorithm, consisting of multiple generalization steps.

## 3.1 Tracing

Before we can trace the parser of the target program, we need to determine the location of the input buffer in memory that the program reads from. The location of the input buffer depends on the concrete program and the execution environment, which is why we require the user to specify a symbol name or the actual address. To trace only the parsing stage rather than tracing the entire program, it is advisable to provide an entry point and optionally an exit point. Good candidates are parser functions, which usually have "*parse*" in the name and take a character buffer pointer as a parameter, which in turn holds the input buffer location. We explain more details in Sections 4.1 and 5.3.

Having the address of the input buffer and an entry function, we can start tracing the processing of the input. First, we use a breakpoint to run the program to the input function. On reaching the entry point, we byte-wise assign watchpoints on all addresses of the input buffer. For the actual tracing, we use the single-step functionality of GDB to walk through the program under test instruction by instruction. At each interrupt, we document the current program counter address and the current stack trace. In addition, we document eventual watchpoint hits (accesses to the input buffer) and reference them to the latest trace entry. We repeat this procedure until we step out of the entry function or we reach a defined exit point. Consequently, each trace element consists of a program address, a list of return addresses on the stack, and a list of watchpoints hits. To obtain a complete set of traces, we execute these steps for each seed input once.

## 3.2 Recovering Control Flow Graphs

Since we need to detect loops and conditional branches in our target program, we first need to recover the control flow graphs of all involved functions. The control flow graph expresses possible control flow sequences of a function as a directed graph from the entry to the exit node. Statically recovering control flow graphs from binary programs is its own discipline in academia [40]. However, since we have single-step traces available, we can dynamically recover the relevant parts of the control flow graph from a binary program.

We therefore iterate over the list of executed instructions and stack traces. If the stack length remains equal between two subsequent instructions, we add the transition as edge to our graph. When we observe an increase of the stack length, the last instruction must have been a function call. We ignore the call edge and connect the call instruction with the return address that is saved on the stack. When we observe a decrease of the stack length, we do nothing. This way the control flow graph of a function stays consistent and, most importantly, within the function itself.

Recovering the control flow graphs is the only stage of GDBMiner where we rely on a valid stack structure at every point of the execution to detect function calls and return addresses. Unfortunately, when using compiler optimizations, the consistency of the stack structure is often temporarily violated during runtime, making recovery of the control flow graph impossible. Using reverse engineering tools like Ghidra [36] would allow GDBMiner to operate on optimized or even obfuscated binaries, but that is out of scope for this work. We opt for recovering the control flow graph from instruction traces, because it is independent from the underlying instruction set, we only need to operate on instruction addresses, and we need the traces anyway for the derivation tree recovery.

## 3.3 Recognizing Control Flow Structures

Based on a control flow graph (CFG) $G = (V, E)$, the predominator relation describes the following dependency [1]:

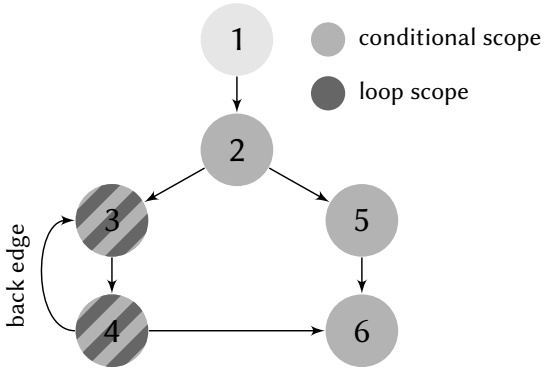



**Figure 4** Conditional and loop scopes in a simple control flow graph.

▶ **Definition 1** (Predomination). *A node $u \in V$* predominates *another node $v \in V$ ($u \xrightarrow{pre} v$ ), if every path from the entry node to $v$ contains $u$.*

The predominator relation of all nodes in a control flow graph can be represented in the predominator tree, starting from the entry node with dominated nodes as children, and can be computed efficiently. Based on the dominator relation, we can locate back edges in the control flow graph [2].

▶ **Definition 2** (Back Edge). *An edge $(u, v) \in E$ is a* back edge*, if the head $v$ predominates its tail $u$, namely $v \xrightarrow{pre} u$.*

Given a back edge $(u, v)$ and a function $reachable(G, u, v)$ telling whether $v$ is reachable from $u$ in $G$, the corresponding natural loop is the set of nodes that can reach the tail $u$ without going through the head $v$ [2]. In other words, these are the nodes that still have a path to $u$, when we remove $v$ from the graph.

▶ **Definition 3** (Natural Loop). *The* natural loop *of back edge $(u, v)$ consists of the nodes: $\{w \mid w \in V$ and reachable $(G \setminus \{v\}, w, u)\}$.*

Having these efficiently computable properties, we can find all loops in control flow graph $G$ by iterating over all edges, checking whether it is a back edge, and, if that is the case, associate the corresponding natural loop with the head of the back edge.

Given the set of successors of node $u$, $G[u]$, a node with multiple successors ($|G[u]| \geq 2$) is a conditional branch. We define the scope of a conditional branch as

▶ **Definition 4** (Conditional Scope). *The* conditional scope *of node $u$, where $|G[u]| \geq 2$ is: $\bigcup_{v \in G[u]} \{w \mid v \xrightarrow{pre} w\}$.*

The scope of a conditional node therefore is the union of all dominated nodes of its successors. Figure 4 shows loop and conditional scopes in a relatively simple control flow graph with a single conditional branch and a single loop. We can easily identify the back edge that closes the loop between nodes 3 and 4.

## 3.4 Recovering Derivation Trees

Based on the single-step traces, the control flow graphs of all functions, the detected natural loops, and the scopes of conditional branches, we can now start to recover a *derivation tree* for each traced seed input. We want to represent the execution flow of a parser in these derivation trees, particularly which function stack is responsible for which input character. Like grammar parse trees, derivation trees therefore have only non-terminal symbols as leaf nodes.

**Listing 2** JSON array parser excerpt. Adopted from [43]

```
1 int parse_val(char **cursor ) {
2   ...
3   switch (**cursor) {
4   ...
5   case '[': {
6     ++(*cursor);
7     valid = parse_array(cursor);
8     break;
9   }
10  default: {
11    double number = strtod(*cursor);
12    ...
13 }
14 int parse_array(char** cursor) {
15  ...
16  int valid = true;
17  if (**cursor == ']') {
18    ++(*cursor);
19    return ;
20  }
21  while (valid) {
22    valid = parse_val(cursor);
23    if (!valid) break;
24    ...
25    if (has_char(cursor, ']')) break;
26    else if (has_char(cursor, ',')) continue;
27    else valid = false;
28  }
29  ...
30 }
```

Let us consider the JSON array parser in Listing 2 as an example for typical parser code. The function `parse_val` consecutively tries to match the character at the current cursor position within a large switch statement. When the value starts with a square bracket, it calls the `parse_array` function, if nothing matches it parses the current value as a number. The function `parse_array` first checks if the character at the current cursor ends the array with a closing squared bracket and returns if that is the case. If not, it repeatedly calls `parse_val` as long as values are followed by a comma. If it reads a closing squared bracket, it ends the loop and returns.

Table 1 shows some important trace entries from a single-step trace with the input string *'[1,2,3]'* of the program Listing 2. Besides the program addresses and watchpoint hits, each trace entry contains the return addresses of all called functions on the stack, which already indicate at which point of the program the input is processed. However, the Mimid algorithm requires not only function scopes, but also loop and conditional scopes. Mimid obtains these finer-grained scopes by a customized instrumentation of the target program. Since we cannot profit from this comprehensive instrumentation, we need an additional processing step to recover the scopes from the traces, using Algorithm 1.

First, the algorithm initializes the list that represents the current call stack (Line 1), as well as the dictionary that represents the derivation tree we want to build from the trace (Line 2). Furthermore, it initializes the dictionaries that contain the scopes of functions, loops, and conditional branches (Line 3-5). Starting from Line 11 the algorithm processes each single-step trace element iteratively. For each element, it first checks if the program address belongs to the start of a function (Line 12), and if so, it appends that function and its scope to the stack and the currently active tree branch. For instance, the first trace entry in Table 1 opens the scope of function *json_parse*, the second opens the scope of function *parse_val*. Next, in Line 15, the algorithm removes all scopes from the scope stack that the current address is not part anymore.

**Table 1** Parts of a single-step trace of program Listing 2 with input *'[1,2,3]'*.

| Address | Function Name | Stack | WPs |
|---|---|---|---|
| 0x401370 | json_parse | [0x4017cd, 0x0] | [] |
| | ... | | |
| 0x4013f0 | parse_val | [0x401385, 0x4017cd, 0x0] | [] |
| | ... | | |
| 0x401417 | parse_val | [0x401385, 0x4017cd, 0x0] | [0] |
| | ... | | |
| 0x401a30 | parse_array | [0x4015cc, 0x401385, 0x4017cd, 0x0] | [] |
| | ... | | |
| 0x401a4e | parse_array | [0x4015cc, 0x401385, 0x4017cd, 0x0] | [1] |
| | ... | | |
| 0x4013f0 | parse_val | [0x401a96, 0x4015cc, 0x401385, 0x4017cd, 0x0] | [] |
| | ... | | |
| 0x401417 | parse_val | [0x401a96, 0x4015cc, 0x401385, 0x4017cd, 0x0] | [1] |
| | ... | | |
| 0x401a96 | parse_array | [0x4015cc, 0x401385, 0x4017cd, 0x0] | [] |
| | ... | | |
| 0x401b20 | has_char | [0x401ac8, 0x4015cc, 0x401385, 0x4017cd, 0x0] | [] |
| | ... | | |
| 0x401b42 | has_char | [0x401ac8, 0x4015cc, 0x401385, 0x4017cd, 0x0] | [2] |
| | ... | | |
| 0x401ac8 | parse_array | [0x4015cc, 0x401385, 0x4017cd, 0x0] | [] |
| | ... | | |
| 0x401b20 | has_char | [0x401ae4, 0x4015cc, 0x401385, 0x4017cd, 0x0] | [] |
| | ... | | |
| 0x401b42 | has_char | [0x401ae4, 0x4015cc, 0x401385, 0x4017cd, 0x0] | [2] |
| | ... | | |

This happens when a function returns, the control flow leaves the scope of a loop, or the control flow leaves the scope of a conditional branch. Similar to entered functions, it subsequently checks if the current address enters a new loop scope (Line 18) or a new conditional scope (Line 22) and also appends the respective scope addresses to the stack and the tree. An example for a loop scope is in line 21 of Listing 2, a conditional scope starts with the switch statement in line 3. Finally, if the current trace element contains watchpoint hits, the algorithm adds the corresponding index as a leaf node to the current active tree branch in Line 25. Overall, Algorithm 1 rebuilds the stack of scopes (functions, loops, conditions) that were entered during execution and attaches the watchpoint hits to the top scope in the tree as they occur.

After exercising the whole trace, the tree might contain some input buffer indices multiple times and branches that do not have such indices at all. We keep only the last access to each byte of the input buffer and discard others. In a separate step, we remove all branches in the tree that do not end with an input buffer index, resulting in our desired derivation tree. Figure 5 shows the derivation tree that our approach recovers with the explained algorithm from the single-step trace in Table 1.

All inner nodes of the tree represent functions or control flow structures as part of a function. Since we cannot simply distinguish between switch statements and if/else constructs in a trace, all conditional branches are named as "*if*" e.g. the case statement of function *parse_val* is identified

**Algorithm 1** Recover derivation trees from traces.

**Input:** A single-step trace
**Output:** A derivation tree to the trace

```
1  STACK ← [(0, [])]                                                  # init stack
2  TREE ← {}                                                          # init tree
3  FUNCTION_SCOPE ← {...}        # function entry address → addresses of function
4  LOOP_SCOPE ← {...}  # loop entry address → addresses in loop scope (natural loop)
5  COND_SCOPE ← {...}       # condition entry address → addresses in condition scope
6
   # helper function to add a scope to the tree
7  def add_scope(id, scope):
8      TREE[STACK[-1].id].append(id)              # add to currently active tree branch
9      STACK.append ((id , scope))                                    # push to stack
10
   # iterate over trace addresses
11 foreach id, addr ∈ trace do
12     if addr ∈ FUNCTION_SCOPE then                  # check for function start
13         add_scope(id , FUNCTION_SCOPE[addr])
14     end
15     while addr ∉ STACK[-1].scope do                                # check scope
16         STACK.pop()                                  # pop exited scope
17     end
18     if addr ∈ LOOP_SCOPE then                        # check for loop start
19         add_scope(id , LOOP_SCOPE[addr])
20     end
21     S ← CFG[addr]                          # get successors of addr from CFG
22     if |S| ≥ 2 ∧ S ⊆ STACK[-1].scope then   # if node has multiple successors
23         add_scope(id , COND_SCOPE[addr])      # start of a conditional scope
24     end
25     foreach idx ∈ watchpoint_hits(id) do   # attach eventual watchpoint hits
26         TREE[STACK[-1].id].append(idx)
27     end
28 end
```

as *parse_val:if1*. Similarly, we name label all flavors of loops, e.g. *while*, *do while*, and *for* loops, as "*loop*". Mimid puts each new iteration of a loop on the same level in the tree. Our approach differs from Mimid in treating multiple loop iterations. Notably, we can see that the program processes digits followed by a comma in the loop *parse_array:loop1* and that our approach attaches subsequent loop iterations as a child branch of the previous one. From our observation, loops in parsers are usually not limited by a constant value, but consume input until a certain character is reached. For instance, in the case above, the *parse_array:loop1* loop ends reading the character *']'*. Nesting the loop iterations leads to more generalizing grammars, as we explain in the next section. The resulting tree expresses at what point the program uses which characters of the input buffer.

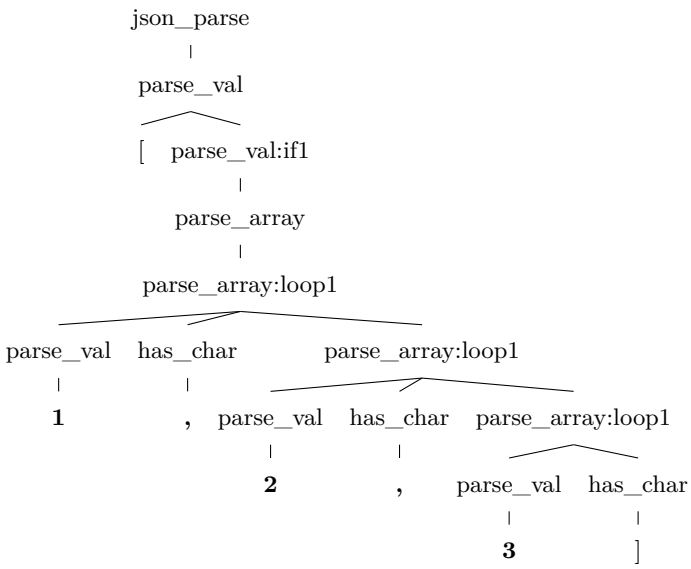



**Figure 5** Recovered derivation tree from *JSON* with input *'[1,2,3]'* using Algorithm 1.

## 3.5 From Derivation Trees to a Grammar

Given a derivation tree, we can derive a context-free grammar by treating all leaf nodes as terminals, all inner nodes as non-terminals, and create production rules for each inner node to the concatenation of all its child nodes. We can merge multiple grammars by union the sets of variables, terminals, and productions. The resulting grammar matches the input language of the program if all inner nodes of the derivation tree with the same name are compatible with each other. Hence, interchanging them still results in accepted inputs. In practice, however, this is rarely the case.

Like Mimid, we use an *active learning* stage, where we examine exactly this compatibility among identically named nodes. For each pair of identically named nodes $(a, b)$, we:

1. Replace the subtree of node $b$ with the one of node $a$.
2. Extract the resulting input string.
3. Check if the target program can parse the input.

Additionally, we cross-check if node $a$ is replaceable by node $b$. If in both cases the program accepts the input, we consider the nodes compatible and assign them a common name, if not, we assign different names. Since we process all possible pairs of nodes, the worst case complexity of this approach is quadratic.

Applying this process to the derivation tree in Figure 5, we figure out that all *parse_val* nodes are interchangeable. In contrast, the first *has_char* node is not compatible with the last *has_char* node, and we need to distinguish them in the resulting grammar.

Figure 6 shows the preliminary grammar derived from the derivation tree in Figure 5. We can see that the *has_char* node appears as ⟨*has_char1*⟩ and ⟨*has_char2*⟩ in the resulting grammar, depending on whether another loop iteration ⟨*parse_array:loop1*⟩ follows. The grammar can already serve for generating valid *JSON* arrays of arbitrary length, including nested arrays and the digits *1*, *2*, and *3*.

Usually, we have multiple seed inputs and therefore multiple derivation trees to learn a grammar from. For a correct grammar, we have to apply the aforementioned active learning steps on all equally named nodes across all obtained derivation trees. Then, we simply merge the individual grammars into a single one.

⟨*START*⟩ ::= ⟨*json_parse*⟩
⟨*json_parse*⟩ ::= ⟨*parse_val*⟩
⟨*parse_val*⟩ ::= '['⟨*parse_val:if1*⟩ | '1' | '2' | '3'
⟨*parse_val:if1*⟩ ::= ⟨*parse_array*⟩
⟨*parse_array*⟩ ::= ⟨*parse_array:loop1*⟩
⟨*parse_array:loop1*⟩ ::= ⟨*parse_val*⟩⟨*has_char1*⟩⟨*parse_array:loop1*⟩ | ⟨*parse_val*⟩⟨*has_char2*⟩
⟨*has_char1*⟩ ::= ','
⟨*has_char2*⟩ ::= ']'

**Figure 6** The grammar derived from the tree in Figure 5.

However, the mined grammar is still cluttered and by far not minimized. To clean the grammar, we remove non-terminals with single rules and replace them with their children accordingly. Additionally, we use the Mimid algorithm to generalize terminal symbols by replacing them with predefined dictionaries of symbols (digits, letters, punctuation) and verifying that the program still accepts generated inputs. Finally, we check if we can replace a symbol with multiple characters and insert appropriate replication rules [1].

## 4 Implementation

Our implementation consists of the *Tracer* component that takes care of generating traces for different systems and the *Miner* component that exercises the recovering of the control flow graphs, the control flow structures, the derivation trees, as well as performing active learning steps adopted from Mimid.

### 4.1 Tracer

We use the Python *pygdbmi* package to control the execution of the target system or program via GDB debug commands. Common instruction set architectures support one to sixteen hardware watchpoints [19], forcing us to trace each seed input repeatedly while sliding a window of available watchpoints over the input bytes. For Linux user programs, Valgrind [37] offers the usage of an unlimited amount of virtual watchpoints. It does so by letting the target program run in an emulator leading to precise control over all its memory accesses. This allows us to cover each byte of the input buffer individually with a watchpoint to recognize accesses during tracing on Linux. Another option would be, e.g., QEMU's user mode emulation [7].

To save time, we only single-step through functions we are interested in. Therefore, we start tracing at a manually defined entry function and stop tracing when we step out of the entry function or at a manually defined exit point. Additionally, we offer to blacklist functions that get skipped during tracing. When a called function $f$ matches the blacklist regular expression, the tracer skips $f$ including any subroutines using the GDB *finish* command, which continues the execution until $f$ exits (step out). Frequent blacklist candidates, for instance, are functions like `malloc` or `free`. Defining the location of the input buffer in form of a symbol name or memory address also is a manual task. While this sounds like tedious work, the proceeding therefore is quite simple when using the *find* function of GDB to locate input bytes in memory or by reading

[1] A full example of a mined JSON grammar is in the appendix.

function signatures when there is source code available. A typical workflow for finding the input buffer location and the entry point might look like this:

1. Start the program with GDB, set a breakpoint to the `main` function, and continue the execution.
2. Step through the program until the *find* function of GDB finds the input we execute the program with. Note the address of the found input buffer.
3. Set a watchpoint to the first address of the input buffer.
4. Restart the program, wait for a watchpoint hit, and take the first address of the current function or the function on the stack as entry function.
5. Take the address of the last instruction of the chosen entry function as exit point.

The entrypoint, the blacklisted functions, as well as the input buffer location are defined via the symbol name or alternatively as actual addresses. The latter is of interest when tracing a binary program without debug symbols. A concrete trace consists of a list of trace entries that each logs the current *address*, *function name*, *function arguments*, *stack*, and eventual *watchpoint hits*.

Since we rely on single stepping for generating the traces, we require the watchpoint implementation to trigger interrupts even during that operation mode, which unfortunately is not true for the ARMv7 debug unit [6]. We therefore have to investigate the state of the watchpoint registers after every single-step. On ARMv7 processors the corresponding bit is in the `DWT_FUNCTION` register, which we can simply read with the GDB *examine* command. Therefore, after every single step, we examine the corresponding bits of the debug unit and attach eventually triggered watchpoint events to the current trace entry.

## 4.2 Miner

The miner component starts with a set of raw traces and first recovers the control flow graphs of functions in the target program, as explained in Section 3.2. Next, it extracts all *natural loops*, as explained in Section 3.3. With these prerequisites, we now exercise Algorithm 1 to obtain a derivation tree for each of the traces.

As detailed in Section 3.5, the next task for the miner component is to discover compatible subtrees from the set of derivation trees. Therefore, we require the program to reflect whether the input was valid or not during or after execution. On Linux programs, we simply consider the input to be valid if the *exit code* is zero, which is the common convention. For embedded programs, however, we require a protocol-specific feedback mechanism. Error codes or behavior is common for most protocols. However, we simply stick to a serial connection that responds with zero when the input is valid or a negative number if parsing fails for our evaluation setup. This feedback oracle suffices to exercise active learning steps, as explained in Section 3.5.

# 5 Evaluation

For evaluating GDBMiner, we first consider the eleven Linux programs listed in Table 2 as a case study for our evaluation, along with handwritten *golden grammars* that cover the program's input specifications. We obtain these from the published replication packages from Mimid, Arvada, and the referenced open source repositories. The programs thereby use language-specific control flow structures, such as `setjmp` and `longjmp` in C, `std::exception` and `std::visit` in C++, and Rust's `match` and `std::result` mechanisms.

**Table 2** Programs for our case study.

| Program | Accepted input | PL | Origin |
|---|---|---|---|
| | *From Cmimid replication package* | | |
| cgidecode | CGI-style escaped strings. | C | [18] |
| json | JSON format. | C | [18] |
| mjs | Micro Java Script programs. | C | [18] |
| tinyc | TinyC programs. | C | [18] |
| | *Additional case study target programs* | | |
| calc | Arithmetic operations. | C | [9] |
| xml | XML format. | C | [21] |
| calcrs | Arithmetic operations. | Rust | Original |
| jsonrs | JSON format. | Rust | [29] |
| calccpp | Arithmetic operations. | C++ | [14] |
| jsoncpp | JSON format. | C++ | [38] |
| xmlcpp | XML format. | C++ | [25] |

XML is not context-free because it requires matching opening and closing tags of arbitrary length [2]. However, we adhere to the subset of XML from [28] for our evaluation, which limits possible tags to the characters *a, b, c, d*. In practice, programs usually consume inputs with such a limited set of possible tags, making them context-free and suitable for our approach.

We compile all programs in debug mode and without optimizations (`-O0`) to force the compiler to keep the stack consistent at all times, as explained in Section 3.2.

All four approaches use a set of seed inputs as a starting point for mining grammars. Consequently the quality of the seed inputs, respectively their input space coverage affects the quality of the mined grammars. However, we provide the same set of seed inputs for all approaches to ensure a fair comparison. As in [28], we randomly generate 20 seed inputs for each trial from the golden grammar using the fuzzingbooks's *GrammarCoverageFuzzer* [50], skipping duplicates.

We measure the quality of the mined grammars as *precision* and *recall* values, like practiced in [18, 28, 5]. Accordingly, we generate 1,000 inputs from the mined grammars and test how many generated inputs the target program accepts. The resulting *precision* value therefore is the number of accepted inputs divided by the number of tested inputs. Subsequently, we generate 1,000 inputs from the golden grammar and test how many of them are parsable by the mined grammars, using the Earley parsing algorithm [11]. The corresponding *recall* value is the number of parsable inputs divided by the number of generated inputs.

Achieving a high precision score alone is easy, because one can trivially construct a grammar that just generates the seed inputs leading to 100% precision. On the other hand, simply getting a high recall value is easy as well, because a grammar that can generate any string gets a recall value of 100%. The harmonic mean between precision and recall values, known as $F_1$*-score*, condenses the two performance values into a single accuracy value we use to compare the different approaches.

Additionally, we evaluate whether the differences between GDBMiner, Arvada, and Treevada are significant according to the Mann-Whitney U test, as recommended in [3]. Since the Mann-Whitney U test is a pairwise comparison, we compare the results of GDBMiner to the respective best black-box approach (Arvada or Treevada), and highlight the result if the corresponding $p$ value is lower than 0.05. We exclude Cmimid from this comparison, because of the practical infeasibilities described in Section 2.5.

[2] Can be shown with the Pumping Lemma.

⟨*START*⟩ ::= ⟨*parse_sum.0-1*⟩
⟨*parse_sum.0-1*⟩ ::= ⟨*parse_mult.0-1*⟩ | ⟨*parse_mult.0-1*⟩ ⟨*parse_sum.1-0-c*⟩ ⟨*parse_sum.0-1*⟩
⟨*parse_mult.0-1*⟩ ::= ⟨*parse_primary.0-c*⟩ | ⟨*parse_primary.0-c*⟩ ⟨*parse_mult.0-0-c*⟩ ⟨*parse_mult.0-1*⟩
⟨*parse_sum.1-0-c*⟩ ::= '+'|'-'
⟨*parse_primary.0-c*⟩ ::= '('⟨*parse_sum.1-1*⟩ | ⟨*DIGIT_s*⟩
⟨*parse_mult.0-0-c*⟩ ::= '*'|'/'
⟨*parse_sum.1-1*⟩ ::= ⟨*parse_mult.0-1*⟩ ⟨*parse_sum.1*⟩
⟨*parse_sum.1*⟩ ::= ')' | ⟨*parse_sum.1-0-c*⟩ ⟨*parse_sum.1-1*⟩
⟨*DIGIT_s*⟩ ::= ⟨*DIGIT*⟩ | ⟨*DIGIT*⟩ ⟨*DIGIT_s*⟩
⟨*DIGIT*⟩ ::= '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'

**Figure 7** Mined grammar for valid math expressions.

## 5.1 Resulting Grammars

First, we have a look at grammars that GDBMiner creates. Figure 7 shows a grammar mined by GDBMiner from the *calc* program using 20 seed inputs. We can see that GDBMiner recovered the recursive nature of mathematical expressions and that it can generate arbitrarily deep expressions, e.g., with the sequence of non-terminals ⟨*START*⟩ → ⟨*parse_sum.0-1*⟩ → ⟨*parse_mult.0-1*⟩ → ⟨*parse_primary.0-c*⟩ → ⟨*parse_sum.1-1*⟩ → ⟨*parse_mult.0-1*⟩. With ten non-terminals and 26 production rules, the mined grammar comes close to the handwritten grammar, which requires six non-terminals and 21 production rules. More importantly, the resulting grammar is correct and has reasonable names for non-terminals.

Table 3 shows the average number of non-terminals and production rules for the mined grammars from GDBMiner and the handwritten golden grammars. We can see that the number of non-terminals and production rules of the resulting grammars is often close to the golden grammars. A drastic outlier is tinyc, which we will discuss in detail in the next section. Also, the resulting grammars for yxml seem to contain significantly more non-terminals and production rules than the golden grammar, which impacts readability and usability. More important, however, is the precision and recall values of the mined grammars, which we will discuss in the next section.

**Table 3** Number of non-terminals and production rules for GDBMiner averaged from 50 runs.

| | Non-Terminals | | Production Rules | |
|---|---|---|---|---|
| Target | GDBMiner | Golden Grammar | GDBMiner | Golden Grammar |
| cgidecode | 5.00 | 5 | 106.00 | 99 |
| json | 33.36 | 30 | 143.46 | 162 |
| mjs | 60.25 | 609 | 263.44 | 1859 |
| tinyc | 273.22 | 12 | 550.42 | 59 |
| calc | 10.00 | 6 | 26.00 | 21 |
| yxml | 129.18 | 18 | 363.68 | 83 |
| calcrs | 40.04 | 6 | 90.32 | 21 |
| jsonrs | 59.68 | 30 | 187.22 | 162 |
| calccpp | 24.04 | 6 | 54.66 | 21 |
| jsoncpp | 57.72 | 30 | 182.32 | 162 |
| xmlcpp | 21.88 | 18 | 194.06 | 83 |

**Table 4** Timing and seed statistics averaged from 50 runs.

| Target | Seed Lengths | Mimid | GDBMiner | Arvada | Treevada |
|---|---|---|---|---|---|
| cgidecode | 2.19±1.66 | 34$s$±7 | 165$s$±53 | 6$s$±3 | 5$s$±2 |
| json | 9.94±8.14 | 34$s$±8 | 210$s$±78 | 70$s$±28 | 36$s$±16 |
| mjs | 25.78±17.74 | 1583$s$±357 | 8712$s$±3598 | 264$s$±267 | 155$s$±93 |
| tinyc | 69.14±35.59 | 14997$s$±38658 | 70609$s$±18923 | 6652$s$±3556 | 3852$s$±2208 |
| calc | 24.47±10.35 | 303$s$±48 | 1485$s$±385 | 64$s$±18 | 31$s$±8 |
| yxml | 17.72±11.92 | N/A | 1585$s$±678 | 284$s$±226 | 132$s$±43 |
| calcrs | 24.89±9.82 | N/A | 5196$s$±1255 | 57$s$±17 | 29$s$±8 |
| jsonrs | 10.37±9.75 | N/A | 548$s$±378 | 63$s$±27 | 31$s$±13 |
| calccpp | 25.11±10.14 | N/A | 5124$s$±1153 | 65$s$±19 | 30$s$±7 |
| jsoncpp | 10.09±8.60 | N/A | 551$s$±208 | 64$s$±36 | 31$s$±8 |
| xmlcpp | 17.44±11.50 | N/A | 492$s$±169 | 1331$s$±641 | 713$s$±339 |

**Table 5** Precision-scores in percentage averaged from 50 runs. Bold values show significant differences between GDBMiner and the best black-box approach.

| Target | Mimid | GDBMiner | Arvada | Treevada |
|---|---|---|---|---|
| cgidecode | 100.00±0.00 | 100.00±0.00 | 100.00±0.00 | 100.00±0.00 |
| json | 100.00±0.00 | **100.00**±0.00 | 81.81±7.99 | 82.68±7.05 |
| mjs | 95.99±5.63 | 90.83±6.93 | 78.98±13.35 | 87.63±10.81 |
| tinyc | 100.00±0.00 | 59.81±12.76 | 73.84±15.98 | **75.25**±18.70 |
| calc | 100.00±0.00 | 100.00±0.00 | 100.00±0.00 | 98.92±0.07 |
| yxml | N/A | **98.63**±0.57 | 29.86±14.98 | 53.77±26.72 |
| calcrs | N/A | 100.00±0.00 | 100.00±0.00 | 100.00±0.00 |
| jsonrs | N/A | **96.29**±4.03 | 77.60±6.97 | 87.01±9.58 |
| calccpp | N/A | 99.59±1.40 | **100.00**±0.00 | 98.90±0.11 |
| jsoncpp | N/A | **100.00**±0.00 | 76.95±7.10 | 83.41±9.15 |
| xmlcpp | N/A | **96.42**±2.22 | 52.86±23.56 | 53.07±30.62 |

*Grammars from GDBMiner reuse symbol names to label non-terminals.*

## 5.2 Comparison against State of the Art

Next, we compare GDBMiner against the current state-of-the-art grammar miners Mimid [18], Arvada [28], and TreeVada [5]. We therefore compile the programs as *Linux user applications* that read input from *stdin* and return a non-zero value if the parsing fails. Our test hardware for this part of the evaluation is a server with four Intel Xeon Gold 6144 CPUs and 1.48 TB of RAM.

As previously mentioned, we generate 20 seeds for each trial and let all grammar miners operate on exactly the same set of seed inputs. We repeat each experiment 50 times to compensate for the probabilistic nature of seed generation and the approaches itself, average the achieved precision and recall values, and calculate the standard deviation. Table 4 shows the average seed lengths generated for each program, as well as the average elapsed time for each approach. We can see that GDBMiner is drastically slower than all other approaches. However, its advantages lie in the quality of the mined grammars, as we will see in the following investigation.

Table 5 shows the achieved precision values. Cmimid achieves the highest precision on all the five programs it can compile. Deriving input grammars from control flow is very precise, but Cmimid has high requirements that the *xml* C source code does not fulfil. Cmimid does not support C++ and Rust at all. Because of these impracticalities, we exclude Mimid from direct comparisions to the other approaches.

**Table 6** Recall-scores in percentage averaged from 50 runs. Bold values show significant differences between GDBMiner and the best black-box approach.

| Target | Mimid | GDBMiner | Arvada | Treevada |
|---|---|---|---|---|
| cgidecode | 100.00±0.00 | **100.00**±0.00 | 95.48±2.47 | 95.48±2.47 |
| json | 53.97±6.61 | 65.57±8.22 | **74.42**±9.47 | 66.19±7.46 |
| mjs | 92.15±7.01 | 93.47±6.86 | 58.54±43.65 | 81.81±24.77 |
| tinyc | 45.73±6.44 | 2.50±1.05 | 68.71±32.17 | **79.67**±14.21 |
| calc | 4.24±1.21 | 100.00±0.00 | 100.00±0.00 | 100.00±0.00 |
| yxml | N/A | 78.31±8.25 | 90.00±15.23 | **90.64**±14.99 |
| calcrs | N/A | 100.00±0.00 | 100.00±0.00 | 100.00±0.00 |
| jsonrs | N/A | 58.00±9.43 | **65.31**±9.81 | 54.39±8.67 |
| calccpp | N/A | 100.00±0.00 | 100.00±0.00 | 100.00±0.00 |
| jsoncpp | N/A | 67.12±9.90 | **76.45**±6.99 | 64.14±6.70 |
| xmlcpp | N/A | 95.27±5.04 | **98.00**±6.86 | 88.84±16.82 |

As black-box approaches, *Arvada* and *Treevada* are easily applicable but deliver mixed precision results only. GDBMiner obtains almost perfect precision values on most targets and significant better results on five out of the eleven programs according to the Mann-Whitney U test. This is not surprising, as it is similar to the Mimid approach but much more versatile.

*GDBMiner generates more precise grammars than Arvada and Treevada.*

Looking at the averaged recall results in Table 6, Arvada and Treevada achieve better results, while GDBMiner is often close behind. Interestingly, Treevada does not show a clear advantage to it predecessor Arvada on our case study programs. One outlier is GDBMiner on the *tinyc* program. Looking into the implementation, we can see that *tinyc* employs a *lexer* stage, which first translates input characters into predefined tokens. The actual parsing operates not the input buffer, but on the stream of tokens. Since GDBMiner can only track accesses to the input buffer, it looses track between input characters and their point of consumption in the program. Consequently, the derivation tree gets flawed, and the subsequent mining step can not generalize reasonably. Cmimid works around that limitation by explicitly following token compares on programs with preprocessing lexer stages, using dynamic taint tracking. A generic taint tracking stage that works under our limited assumptions would be required to fix this inability of GDBMiner. However, only mining on a single of our case study programs would benefit from such a step, and therefore we keep GDBMiner's tracing stage simple and refrain from more complex analysis.

Compiling the averaged $F_1$-scores in Table 7, we can see that GDBMiner achieves the highest score on nine out of the eleven programs of our case study, with five of them beeing significantly better than the second-best approach. Mined grammars with high $F_1$-scores signify that generated inputs are most likely valid but also cover a large portion and wide variety of available input features. Not surprisingly, we found that using more input seeds leads to higher recall values, because they have higher chances of covering more input features. Using 20 seeds, as done in [28], seems to be a reasonable amount for all our case study programs, but an optimal value highly depends on the target program and diversity of the seeds.

*GDBMiner yields most of the best accuracy scores.*

**Table 7** F1-scores in percentage averaged from 50 runs. Bold values show significant differences between GDBMiner and the best black-box approach.

| Target | Mimid | GDBMiner | Arvada | Treevada |
|---|---|---|---|---|
| cgidecode | 100.00±0.00 | **100.00**±0.00 | 97.67±1.29 | 97.67±1.29 |
| json | 69.86±5.64 | 78.92±6.01 | 77.46±7.11 | 73.07±5.22 |
| mjs | 93.78±4.63 | **91.86**±5.03 | 54.57±38.10 | 81.83±15.81 |
| tinyc | 62.49±6.19 | 4.76±1.91 | 64.32±25.83 | **75.63**±14.34 |
| calc | 8.11±2.24 | 100.00±0.00 | 100.00±0.00 | 99.46±0.04 |
| yxml | N/A | **87.07**±5.22 | 42.67±16.15 | 62.38±21.40 |
| calcrs | N/A | 100.00±0.00 | 100.00±0.00 | 100.00±0.00 |
| jsonrs | N/A | 71.85±8.13 | 70.43±7.21 | 66.27±7.57 |
| calccpp | N/A | 99.79±0.72 | **100.00**±0.00 | 99.45±0.06 |
| jsoncpp | N/A | **79.91**±7.19 | 76.33±4.87 | 72.12±5.72 |
| xmlcpp | N/A | **95.76**±2.91 | 65.73±20.70 | 60.33±25.13 |

The time needed to mine the grammars is rather unimportant as long as it remains within a usable frame. Nevertheless, we would like to concede that Arvada and Cmimid require only a few minutes for the runs, while GDBMiner can take several hours, as presented in Table 4 This is taken to the extreme when mining grammars on embedded hardware, which we examine in one of the next sections.

## 5.3 Full-stack Case Study

Before running GDBMiner on embedded hardware, we go through the steps necessary to run GDBMiner on a real-world program and show what kind of analysis we can do with the resulting grammar. Industry extensively uses open source programs and to make all steps reproducible we will also demonstrate GDBMiner on an open source program. We want to emphasize again that GDB is available for most platforms and architectures, and hence these steps work on any program with a parser. The SVG++ library [32] processes scalable vector graphics (SVG) images using different XML parser backends and includes an application that renders given vector images into a pixel-based image format. The followings steps are required to apply GDBMiner:

1. Configure to build SVG++ in debug mode.
2. Looking at the main function of the render application and ensuring that it reads input data from a char buffer.
3. Adopt a demo SVG image from Mozilla's SVG docs [33] as seed, depicted in Figure 8.
4. Define entry and exit points by source file line numbers and the char buffer name.

On our laptop with an Intel i7-10610U CPU, tracing and mining took about 14 hours. The resulting grammar is compatible with the fuzzingbook's grammar fuzzer [50]. A collection of generated images is shown in Figure 9 to visually express the diversity possible with grammar-generated inputs.

The grammar allows us to perform the following types of static and dynamic program analysis.

1. We can generate an unlimited number of random, but valid SVG images to test the SVG renderer looking for inputs causing timeouts or crashes. Indeed, many of our generated inputs cause hangs when defined shape sizes are too huge. A robust render application should cope with any input image, in our opinion, which is why we consider this behavior as buggy. For the following analysis, we alter the grammar to let numbers consist of one to three digits only.

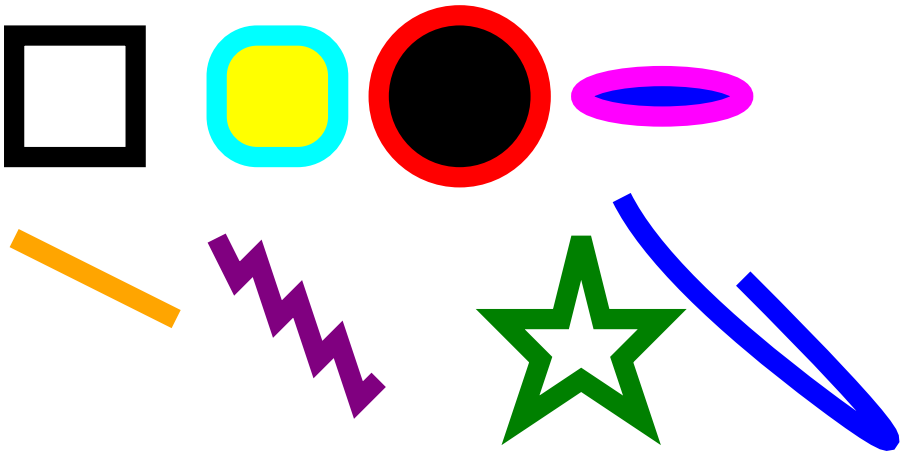


**Figure 8** SVG seed image.

2. We analyze the grammar alone and reveal that any attribute within a node can be redefined arbitrarily many times, such as multiple `fill=<color>` for the same shape node. The alternative XML parser backend of SVG++ does not allow attribute redefinitions, resulting in a different grammar. However, it is the programmer's decision whether this poses valid behavior.
3. We also compare the result of various SVG renderers to reveal differences. Again, we found that the default XML parser of the SVG++ library accepts redefinitions of node attributes, while the alternative XML backend, as well as Mozilla Firefox's SVG parser throw an error on redefinitions.

## 5.4 Evaluation on Embedded Programs

Finally, we evaluate GDBMiner under the restricted conditions of *embedded systems*. We therefore choose the *B-L475E-IOT01A Discovery kit for IoT* from *STMicroelectronics* as a representative platform. It features an ARM Cortex-M4 core, several sensors and interfaces, six hardware breakpoints, four hardware data watchpoints, and an on-board debug probe. For our case study, we build programs for evaluating GDBMiner on embedded hardware using the cross-platform framework platformIO [27]. We use libraries from its dependency management system for parsing *cgidecode* [39], *json* [4], and *xml* [24] data. GDBMiner can potentially feed input via any interface approachable by the host computer. For simplicity, the applications are build to fetch input via the serial interface, run the corresponding parser functions, and return zero if the input was valid or a negative number else.

Tracing on embedded hardware with a limited amount of hardware watchpoints is significantly slower than on Linux applications mainly for three reasons:

1. The ARM Cortex-M4 core on our embedded hardware runs at a frequency of 120 MHz compared to 4.20 GHz of the Intel Xeon Gold 6144 we used in the experiments before.
2. The overhead of the debugging mechanism itself. GDBMiner needs to transfer each debug command first via TCP to a GDB server application, which transmits it via USB to the debug probe on the development board, and finally the debug probe forwards commands via the SWD (Single Wire Debug) protocol to debug unit on the processor.
3. The limited amount of hardware watchpoints forces us to trace each seed input multiple times. The exact number of traces we need to exercise per seed is determined by the length of the seed divided by the number of available watchpoints. Tracing a seed with 20 characters and the four available watchpoints requires five repetitions.

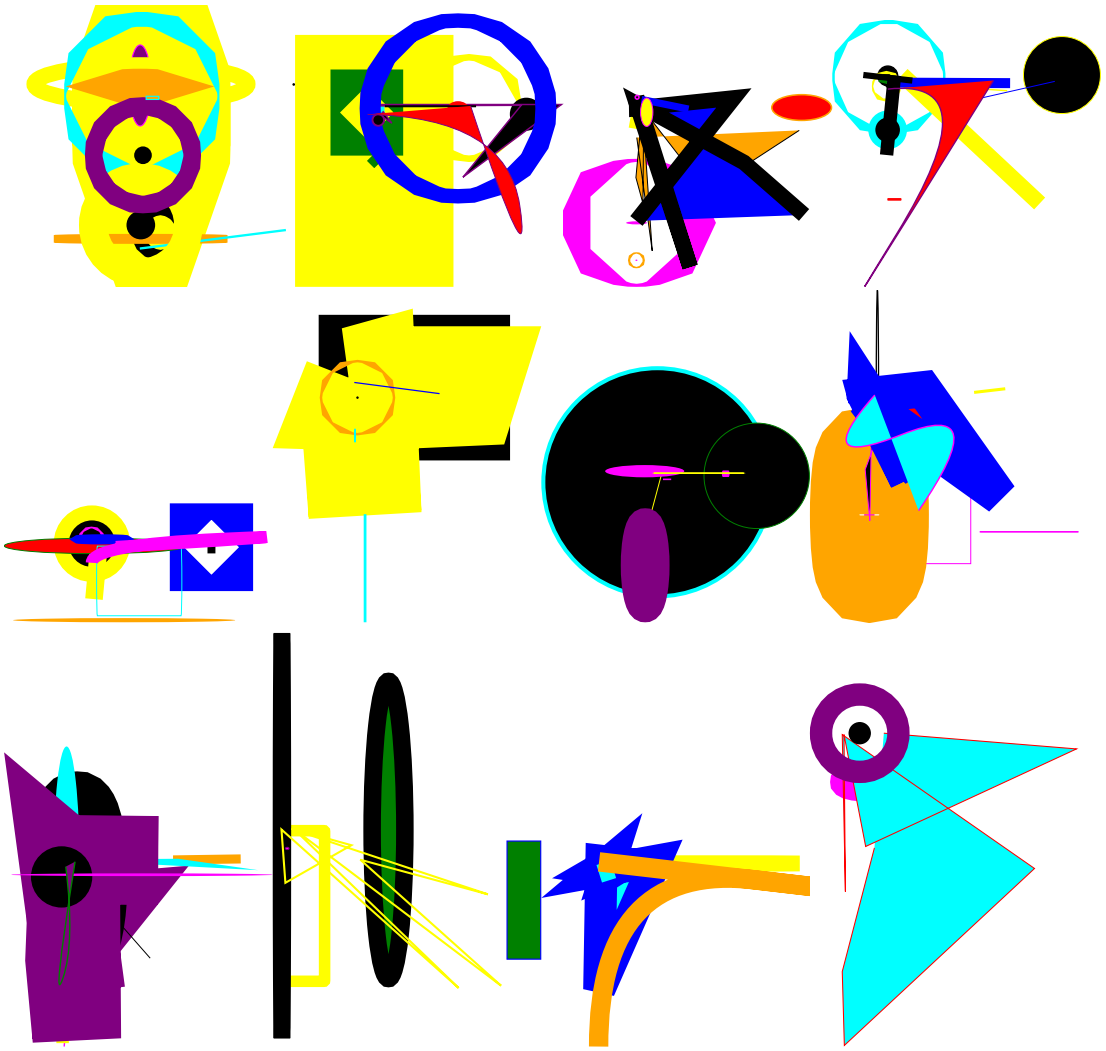


**Figure 9** Fuzzed images from mined SVG grammar.

For this part of the evaluation, we create a single high quality set of 20 seeds that cover most of the input features of the target programs. Again, we compute precision and recall values of the resulting grammar from 1,000 evaluation inputs derived from the golden grammars. Additionally, we measure the time elapsed for tracing and mining, and also the number of accepted inputs the fuzzingbook's mutation fuzzer [50] generates from the seeds. Table 8, shows that GDBMiner achieves high precision and recall values across all programs from our case study within a reasonable amount of time, even under the challenging conditions on our embedded hardware.

*GDBMiner effectively mines input grammars with limited numbers of hardware watchpoints by merging multiple traces.*

In Table 8, we can also see that the tracing part is the most time-intense stage in our setting, presumably from the aforementioned debug overhead and multiple analysis runs required. For the XML program, tracing took even multiple days. Nevertheless, we believe that the required time is practically reasonable. We want to emphasize that the required time is rather secondary since mining a grammar is a one-time task, and if available, it can be used to generate an unlimited number of valid test inputs subsequently.

**Table 8** Performance of GDBMiner on embedded hardware.

| Program | Precision | Recall | Tracing Time | Mining Time | Mutation Fuzzer |
|---|---|---|---|---|---|
| cgidecode | 93.9% | 97.5% | 01:09:50 | 00:00:48 | 79.5% |
| json | 99.3% | 88.1% | 03:29:14 | 00:02:24 | 20.8% |
| xml | 99.4% | 93.5% | 33:35:21 | 01:12:37 | 10.1% |

When fuzzing applications with preconnected parsing stages, it is important to maximize the number of inputs that make it through the parsing stage, that is they get accepted. In the last column of Table 8, we can see that a mutation-based fuzzer generates mostly invalid inputs that are rejected by the parser and do not reach *deeper* parts of the application. Using the grammars from GDBMiner, however, yields almost always to valid inputs, potentially reaching these deep parts.

## 5.5 Threats to Validity

Like any empirical study, ours is subject to threats to validity.

**External validity** refers to the generalizability of research findings beyond the specific study context. While our subjects cover a number of features in input languages, they in no way can be representative for all input languages used – a data set that is not known.

**Internal validity** refers to the degree to which a study provides causal conclusions about the relationship between variables. To minimize the risk of systematic errors, we verified that our miners produce the required results on a small set of sample programs before applying it to the full set of targets.

**Construct validity** refers to the extent to which the variables and measurements accurately represent the underlying theoretical constructs. For accuracy, we use the standard measures of precision and recall that have been used in the literature on grammar mining before [23, 18, 28]. As it comes to *readability,* we measure the number of non-terminals and production rules in the mined grammars. We are not aware of established metrics that would capture the readability of grammars further. We leave it to the readers to decide whether they can comprehend the structure.

# 6 Limitations

Since GDBMiner is designed to work only with generic debugging features and does not perform any static analysis of the software, it has some limitations.

1. GDBMiner relies on data watchpoints to trace input buffer accesses and does not track the input buffer across multiple stages of input processing. Hence, GDBMiner can not efficiently mine grammars from programs that tokenize input data before parsing, as we showed with the tinyc program.
2. GDBMiner requires a consistent stack and ideally a well structured program, which both is not always given in compiler-optimized code. It is also beneficial to have access to symbol names for assigning reasonable names to the non-terminals. These properties might not be available for closed-source programs.
3. GDBMiner needs seed inputs for the target program that ideally do cover most of its input features. These seeds can originate from capturing real-world inputs to the program, from test cases (as shown in our SVG example), or manually crafted. An approach for mining grammars

without seeds was proposed in [8], using symbolic execution to enumerate possible program paths. It shows near perfect results on the four C programs it was evaluated on, and the approach might be adapted to other programming languages in future.

Despite these limitations, we are confident that GDBMiner is a versatile and practical approach to mine input grammars. Moreover, we demonstrate that a sophisticated program analysis is possible with the limited features GDB offers, which in turn leads to an approach working on almost any system.

## 7 Conclusion and Future Work

In this paper, we presented a practical debugger-driven grammar mining approach, called GDBMiner, which is able to derive context-free input grammars from any system that can be debugged with GDB. We explained how we leverage the debugger single-step functionality to trace programs and use limited amounts of hardware watchpoints to trace input buffer accesses. Additionally, we explained how we recover control flow graphs, reveal control flow structures, and presented an algorithm to recover derivation trees from just the obtained traces. Finally, we showed that we can transform the recovered derivation trees to receive human-readable and highly precise input grammars.

GDBMiner generates near-perfect precise grammars and compared to state-of-the-art approaches reaches mostly higher accuracy ($F_1$) scores. We demonstrated its practicality by walking through the application on an SVG renderer step-by-step and GDBMiner achieved similar results on embedded hardware. We find that GDBMiner is a versatile solution to mine precise input grammars from programs small and large and recommend considering GDB as a robust and unified interface to a large variety of programs and architectures. Our future work will focus on the following topics:

**Handle tokenizers.** On one of our eleven case study programs, the point of input byte consuming mismatches the point of reading it due to a tokenizing stage. We will devise static and dynamic mechanisms to detect such copying, and thus track bytes accurately in such cases regardless.

**Binary formats.** Knowing the proper syntax for input formats is helpful for fuzzing; yet, there are also *semantic constraints* to be fulfilled. We are investigating higher order input specifications like ISLa [47] to additionally specify semantic constraints across inputs, as well as *learning* such constraints from given inputs, using ISLearn [47].

## 8 Data Availability

To foster open research and academic exchange, GDBMiner and requirements to reproduce the evaluation results are open-sourced under:

`https://github.com/boschresearch/gdbminer`

## References


1 Hiralal Agrawal. Dominators, super blocks, and program coverage. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–34, 1994. doi:10.1145/174675.175935.

2 Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques, and tools*, volume 2. Addison-wesley Reading, 2007. URL: https://www.worldcat.org/oclc/12285707.

3 Andrea Arcuri and Lionel Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 1–10, 2011. doi:10.1145/1985793.1985795.

4 Arduino Libraries . Arduino_json, 2022. Accessed: 2023-10-01. URL: https://registry.platformio.org/libraries/arduino-libraries/Arduino_JSON.

5 Mohammad Rifat Arefin, Suraj Shetiya, Zili Wang, and Christoph Csallner. Fast deterministic black-box context-free grammar inference. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–12, 2024. doi:10.1145/3597503.3639214.

6 Arm. ARMv7-M Architecture Reference Manual, 2021. Accessed: 2023-05-05. URL: https://developer.arm.com/documentation/ddi0403/ee/?lang=en.

7 Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, page 46. California, USA, 2005. URL: http://www.usenix.org/events/usenix05/tech/freenix/bellard.html.

8 Leon Bettscheider and Andreas Zeller. Look ma, no input samples! mining input grammars from code with symbolic parsing. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 522–526, 2024. doi:10.1145/3663529.3663790.

9 Franck Bui. Implement basic parsers for parsing trivial arithmetic expressions, 2010. Accessed: 2023-05-02. URL: https://github.com/fbuihuu/parser/blob/master/calc.c.

10 Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 317–329, 2007. doi:10.1145/1315245.1315286.

11 Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970. doi:10.1145/362007.362035.

12 Max Eisele, Daniel Ebert, Christopher Huth, and Andreas Zeller. Fuzzing embedded systems using debug interfaces. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1031–1042, 2023. doi:10.1145/3597926.3598115.

13 Max Camillo Eisele, Marcello Maugeri, Rachna Shriwas, Christopher Huth, and Giampaolo Bella. Embedded fuzzing: a review of challenges, tools, and solutions. *Cybersecurity*, 2022. doi:10.1186/s42400-022-00123-y.

14 Björn Fahller. A variant of recursive descent parsing, 2017. Accessed: 2023-05-02. URL: https://github.com/rollbear/variant_parse.

15 Andrew Fasano, Tiemoko Ballo, Marius Muench, Tim Leek, Alexander Bulekov, Brendan Dolan-Gavitt, Manuel Egele, Aurélien Francillon, Long Lu, Nick Gregory, et al. SoK: Enabling security analyses of embedded systems via rehosting. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, pages 687–701, 2021. doi:10.1145/3433210.3453093.

16 Bill Gatliff. Embedding with GNU: the GDB remote serial protocol. *Embedded Systems Programming*, 12:108–113, 1999.

17 Google. OSS-Fuzz, 2021. Accessed: 2021-12-20. URL: https://google.github.io/oss-fuzz/.

18 Rahul Gopinath, Björn Mathis, and Andreas Zeller. Mining input grammars from dynamic control flow. In *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 172–183, 2020. doi:10.1145/3368089.3409679.

19 Joseph L Greathouse, Hongyi Xin, Yixin Luo, and Todd Austin. A case for unlimited watchpoints. *ACM SIGPLAN Notices*, 47(4):159–172, 2012. doi:10.1145/2248487.2150994.

20 Nikolas Havrikov and Andreas Zeller. Systematically covering input structure. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 189–199. IEEE, 2019. doi:10.1109/ASE.2019.00027.

21 Yoran Heling. Yxml - a small, fast and correct* xml parser, 2013. Accessed: 2023-05-02. URL: https://dev.yorhel.nl/yxml.

22 John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. Introduction to automata theory, languages, and computation. *Acm Sigact News*, 32(1):60–65, 2001. doi:10.1145/568438.568455.

23 Matthias Höschele and Andreas Zeller. Mining input grammars with AUTOGRAM. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 31–34. IEEE, 2017. doi:10.1109/ICSE-C.2017.14.

24 Ioulianos Kakoulidis. Platformio yxml, 2021. Accessed: 2023-10-01. URL: https://registry.platformio.org/libraries/julstrat/LibYxml.

25 Marcin Kalicinski. C++ xml parser, 2006. Accessed: 2023-05-02. URL: https://rapidxml.sourceforge.net/.

26 Donald E Knuth. Top-down syntax analysis. *Acta Informatica*, 1:79–110, 1971. doi:10.1007/BF00289517.

27 Ivan Kravets. Platformio, 2014. Accessed: 2023-05-02. URL: https://platformio.org/.

28 Neil Kulkarni, Caroline Lemieux, and Koushik Sen. Learning highly recursive input grammars. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 456–467. IEEE, 2021. doi:10.1109/ASE51524.2021.9678879.

29 Linda_pp. Simple json parser/generator for rust, 2016. Accessed: 2023-05-02. URL: https://crates.io/crates/tinyjson.

30 Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Trans. Software Eng.*, 47(11):2312–2331, 2021. doi:10.1109/TSE.2019.2946563.

31 Björn Mathis, Rahul Gopinath, Michaël Mera, Alexander Kampmann, Matthias Höschele, and Andreas Zeller. Parser-directed fuzzing. In *Proceedings of the 40th ACM sigplan conference on programming language design and implementation*, pages 548–560, 2019. doi:10.1145/3314221.3314651.

32 Oleg Maximenko. Svg++ documentation, 2014. Accessed: 2024-01-23. URL: http://svgpp.org/.

33 MDN contributors. Svg tutorial - basic shapes, 2023. Accessed: 2024-01-23. URL:

https://developer.mozilla.org/en-US/docs/Web/SVG/Tutorial/Basic_Shapes.

34 Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. Avatar 2: A multi-target orchestration platform. In *Proc. Workshop Binary Anal. Res.(Colocated NDSS Symp.)*, volume 18, pages 1–11, 2018. URL: https://s3.eurecom.fr/docs/bar18_muench.pdf.

35 Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *NDSS*, 2018. URL: https://www.ndss-symposium.org/wp-content/uploads/2018/07/bar2018_1_Muench_paper.pdf.

36 National Security Agency. Ghidra, 2019. Accessed: 2021-12-20. URL: https://ghidra-sre.org/.

37 Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007. doi:10.1145/1273442.1250746.

38 Kazuho Oku. A header-file-only, json parser serializer in c++, 2009. Accessed: 2023-05-02. URL: https://github.com/kazuho/picojson.

39 Kazuki Ota. Arduino percent, 2023. Accessed: 2023-10-01. URL: https://registry.platformio.org/libraries/dojyorin/percent_encode/.

40 Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. SoK: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 833–851. IEEE, 2021. doi:10.1109/SP40001.2021.00012.

41 Terence J. Parr and Russell W. Quong. Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995. doi:10.1002/spe.4380250705.

42 Goldman Sachs. Average number of lines of codes per vehicle globally in 2015 and 2020, with a forecast for 2025, 2022. Accessed: 2023-05-02. URL: https://www.statista.com/statistics/1370978/automotive-software-average-lines-of-codes-per-vehicle-globally/.

43 Harald Scheirich. Jsonparser, 2017. Accessed: 2023-10-01. URL: https://github.com/HarryDC/JsonParser.

44 Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Address-Sanitizer: A fast address sanity checker. In *2012 USENIX annual technical conference (USENIX ATC 12)*, pages 309–318, 2012. URL: https://dl.acm.org/doi/abs/10.5555/2342821.2342849.

45 Prashast Srivastava and Mathias Payer. Gramatron: Effective grammar-aware fuzzing. In *Proceedings of the 30th acm sigsoft international symposium on software testing and analysis*, pages 244–256, 2021. doi:10.1145/3460319.3464814.

46 Richard Stallman, Roland Pesch, Stan Shebs, et al. Debugging with GDB. *Free Software Foundation*, 675, 1988. URL: https://sourceware.org/gdb/current/onlinedocs/gdb.pdf.

47 Dominic Steinhöfel and Andreas Zeller. Input invariants. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 583–594, 2022. doi:10.1145/3540250.3549139.

48 Elecia White. *Making embedded systems*. O'Reilly Media, Inc., 2024.

49 Christopher Wright, William A Moeglein, Saurabh Bagchi, Milind Kulkarni, and Abraham A Clements. Challenges in firmware re-hosting, emulation, and analysis. *ACM Computing Surveys (CSUR)*, 54(1):1–36, 2021. doi:10.1145/3423167.

50 Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. The fuzzing book, 2019. URL: https://www.fuzzingbook.org/.

# A Appendix

```
⟨START⟩ ::= ⟨json_parse.0-1⟩
⟨json_parse.0-1⟩ ::= ⟨json_parse_value.0-0-c⟩ | ⟨json_parse_value.0-0-c⟩ ⟨json_parse.0⟩
⟨json_parse_value.0-0-c⟩ ::= ⟨json_parse_value.0-10⟩ | ⟨json_parse_value.0-6⟩ | ⟨json_parse_value.0⟩ | ⟨skip_whitespace-.0-1⟩ ⟨json_parse_value.0-12-c⟩ | [ ⟨json_parse_value.0-1⟩
⟨json_parse.0⟩ ::= ⟨skip_whitespace_d520-.0-1⟩ | ⟨skip_whitespace_d520-.0-1⟩ ⟨json_parse.0⟩
⟨json_parse_value.0-10⟩ ::= ⟨json_parse_value_94a1.1-1⟩ ⟨json_parse_value_f180.0-0-c⟩
⟨json_parse_value.0-6⟩ ::= { ⟨json_parse_value.0-4⟩
⟨json_parse_value.0⟩ ::= ⟨json_is_literal_94a0.0-0-c⟩ | ⟨json_parse_value.0-7⟩
⟨skip_whitespace-.0-1⟩ ::= ⟨skip_whitespace_d520-.0-1⟩ | ⟨skip_whitespace_d520-.0-1⟩ ⟨skip_whitespace-.0-1⟩
⟨json_parse_value.0-12-c⟩ ::= ⟨json_parse_value.0-10⟩ | ⟨json_parse_value.0-6⟩ | ⟨json_parse_value.0⟩
⟨json_parse_value.0-1⟩ ::= ⟨json_parse_array.0-0-c⟩ | ⟨skip_whitespace-.0-1⟩ ⟨json_parse_array.0-0-c⟩
⟨json_parse_value_94a1.1-1⟩ ::= "
⟨json_parse_value_f180.0-0-c⟩ ::= ⟨json_parse_value_94a1.1-1⟩ | ⟨___ASCII_ALPHANUM_PUNCT_s___⟩ " | ⟨___ASCII_ALPHANUM_PUNCT_s___⟩ ) "
⟨json_parse_value.0-4⟩ ::= ⟨json_parse_object.0⟩ | ⟨skip_whitespace-.0-1⟩ ⟨json_parse_object.0⟩
⟨json_parse_object.0⟩ ::= ⟨has_char_94a0.2-1⟩ | ⟨json_parse_value.0-0-c⟩ ⟨has_char.3-0-c⟩ ⟨json_parse_value.0-0-c⟩ ⟨json_parse_object.0-4⟩
⟨has_char_94a0.2-1⟩ ::= }
⟨has_char.3-0-c⟩ ::= ⟨has_char_94a0.3-1⟩ | ⟨skip_whitespace-.0-1⟩ ⟨has_char_94a0.3-1⟩
⟨json_parse_object.0-4⟩ ::= ⟨has_char_94a0.2-1⟩ | ⟨json_parse_object.0-5⟩ | ⟨skip_whitespace-.0-1⟩ ⟨json_parse_object.0-7⟩
⟨has_char_94a0.3-1⟩ ::= :
⟨json_parse_object.0-5⟩ ::= ⟨has_char_94a0.0-1⟩ ⟨json_parse_object.0⟩
⟨json_parse_object.0-7⟩ ::= ⟨has_char_94a0.2-1⟩ | ⟨json_parse_object.0-5⟩
⟨has_char_94a0.0-1⟩ ::= ,
⟨json_is_literal_94a0.0-0-c⟩ ::= f a l s e | n u l l | t r u e
⟨json_parse_value.0-7⟩ ::= ⟨json_is_literal_94a0.0-0-c⟩ | ⟨json_parse_value.0-8⟩
⟨json_parse_value.0-8⟩ ::= ⟨json_is_literal_94a0.0-0-c⟩ | ⟨json_parse_value_6700.0-0-c⟩
⟨json_parse_value_6700.0-0-c⟩ ::= - ⟨___DIGIT_s___⟩ . ⟨___DIGIT_s___⟩ e - ⟨___DIGIT_s___⟩ | - ⟨___DIGIT_s___⟩ . ⟨___DIGIT_s___⟩ e ⟨___DIGIT_s___⟩ | - ⟨___DIGIT_s___⟩ E ⟨___DIGIT_s___⟩ | ⟨___DIGIT_s___⟩ |
                                  ⟨___DIGIT_s___⟩ . ⟨___DIGIT_s___⟩ | ⟨___DIGIT_s___⟩ E + ⟨___DIGIT_s___⟩ | ⟨___DIGIT_s___⟩ E ⟨___DIGIT_s___⟩
⟨skip_whitespace_d520-.0-1⟩ ::= ⟨___WHITESPACE_s___⟩
⟨json_parse_array.0-0-c⟩ ::= ⟨json_parse_array.0⟩ | ⟨json_parse_array_94a0.0-1⟩
⟨json_parse_array.0⟩ ::= ⟨json_parse_value.0-0-c⟩ ⟨json_parse_array.0-1⟩
⟨json_parse_array_94a0.0-1⟩ ::= ]
⟨json_parse_array.0-1⟩ ::= ⟨json_parse_array.0-2⟩ | ⟨json_parse_array_94a0.0-1⟩ | ⟨skip_whitespace-.0-1⟩ ⟨json_parse_array.0-4⟩
⟨json_parse_array.0-2⟩ ::= ⟨has_char_94a0.0-1⟩ ⟨json_parse_array.0⟩
⟨json_parse_array.0-4⟩ ::= ⟨json_parse_array.0-2⟩ | ⟨json_parse_array_94a0.0-1⟩
⟨___DIGIT_s___⟩ ::= ⟨___DIGIT___⟩ | ⟨___DIGIT___⟩ ⟨___DIGIT_s___⟩
⟨___DIGIT___⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
⟨___ASCII_ALPHANUM_PUNCT_s___⟩ ::= ⟨___ASCII_ALPHANUM_PUNCT___⟩ | ⟨___ASCII_ALPHANUM_PUNCT___⟩ ⟨___ASCII_ALPHANUM_PUNCT_s___⟩
⟨___ASCII_ALPHANUM_PUNCT___⟩ ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | 0 | 1 | 2
                                  | 3 | 4 | 5 | 6 | 7 | 8 | 9 | ! | # | $ | % | & | ’ | ( | ) | * | + | , | - | . | / | : | ; | < | = | > | ? | @ | [ | ] | ^ | _ | ‘ | { | | | } | ~
⟨___WHITESPACE_s___⟩ ::= ⟨___WHITESPACE___⟩ | ⟨___WHITESPACE___⟩ ⟨___WHITESPACE_s___⟩
⟨___WHITESPACE___⟩ ::= ␣ | \t | \n | \r | \u000b | \f
```

**Figure 10** JSON grammar mined from the *json* program.

# 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

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 System[3]; 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`.

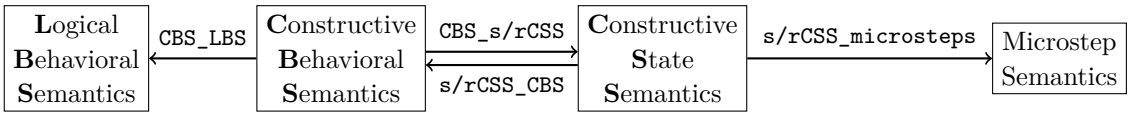



**Figure 1** The chain of Kernel Esterel semantics. Arrows represent simulation.

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** | |
|---|---|---|
| `nothing` | $0$ | |
| `pause` | $1$ | |
| `emit` $s$ | $!s$ | |
| `await immediate` $s$ | $@s$ | |
| `if` $s$ `then` $p$ `else` $q$ `end` | $s\,?\,p\,,\,q$ | |
| `suspend` $s$ `when` $p$ | $s \supset p$ | |
| `trap` $T$ `in` $p$ `end` | $\{p\}$ | |
| `exit` $T^k$ | $k$ | (with $k \geq 2$) |
| | $\uparrow p$ | |
| $p\,;\,q$ | $p\,;\,q$ | |
| `loop` $p$ `end` | $p*$ | |
| $p \,\|\|\, q$ | $p \,\|\, q$ | |
| `signal` $s$ `in` $p$ `end` | $p\backslash s$ | |

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 T^2;
   trap U in
      exit T^3
   ||
      exit U^2
   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 $\lambda$-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 O
each R
```

The difference with classical `ABRO` is that `O` is also emitted if `A` and `B` are simultaneously received when `R` occurs. The original `ABRO` behavior, where `O` 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 O;
      loop pause end;
      exit T2
   end
end
```

The last "`exit` $\mathtt{T}^2$" 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: $\big\{\big(1\,;\,(R\,?\,2\,,\,1)\,*\big)\mid\big((@A\mid@B\,)\,;\,!O\,;\,(1\,*)\,;\,2\big)\big\}\,*$, 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.

- 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 "$p \,||\, q$" 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 $p\backslash s$ 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, `O` 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 `O` 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 $\lambda$-calculus in which types and logical formulas are unified, so that writing a proof amounts to providing a $\lambda$-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 itself[4] 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 $p \,||\, q$ 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 $\delta$ 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 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 $k \xrightarrow[E]{\varnothing, k} 0$ 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 $\text{sig}^+$ and $\text{sig}^-$ 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.

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

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 $\text{seq}_0$ 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 $\text{sig}^+$ and $\text{sig}^-$ 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 $\text{sig}^+$ we assume that $s$ is received, so we check that it is indeed emitted ($s^+ \in E'$), and conversely for rule $\text{sig}^-$ 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 is it 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

$$\frac{}{k \xrightarrow[E]{\varnothing, k} 0}\ \mathrm{k}$$

$$\frac{}{!s \xrightarrow[E]{\{s^+\}, 0} 0}\ \texttt{emit}$$

$$\frac{s^+ \in E}{@s \xrightarrow[E]{\varnothing, 0} 0}\ \mathrm{awimm}^+ \qquad \frac{s^- \in E}{@s \xrightarrow[E]{\varnothing, 1} @s}\ \mathrm{awimm}^-$$

$$\frac{s^+ \in E \qquad p \xrightarrow[E]{E', k} p'}{s\,?\,p\,,\,q \xrightarrow[E]{E', k} p'}\ \mathrm{then} \qquad \frac{s^- \in E \qquad q \xrightarrow[E]{E', k} q'}{s\,?\,p\,,\,q \xrightarrow[E]{E', k} q'}\ \mathrm{else}$$

$$\frac{p \xrightarrow[E]{E', k} p'}{s \supset p \xrightarrow[E]{E', k} @\neg s\,;\, s \supset p'}\ \mathrm{suspend}$$

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

$$\frac{p \xrightarrow[E]{E', k} p'}{\{p\} \xrightarrow[E]{E', \downarrow k} \{p'\}}\ \mathrm{trap} \qquad \frac{p \xrightarrow[E]{E', k} p'}{\uparrow p \xrightarrow[E]{E', \uparrow k} \uparrow p'}\ \mathrm{shift}$$

$$\frac{k \neq 0 \qquad p \xrightarrow[E]{E', k} p'}{p\,;\, q \xrightarrow[E]{E', k} p'\,;\, q}\ \mathrm{seq}_k \qquad \frac{p \xrightarrow[E]{E'_p, 0} p' \qquad q \xrightarrow[E]{E'_q, k} q'}{p\,;\, q \xrightarrow[E]{E'_p \cup E'_q, k} q'}\ \mathrm{seq}_0$$

$$\frac{p \xrightarrow[E]{E'_p, k_p} p' \qquad q \xrightarrow[E]{E'_q, k_q} q'}{p \mid q \xrightarrow[E]{E'_p \cup E'_q, \max(k_p, k_q)} p' \mid q'}\ \mathrm{par}$$

$$\frac{p \xrightarrow[E*s^+]{E', k} p' \qquad s^+ \in E'}{p\backslash s \xrightarrow[E]{E'\backslash s, k} p'\backslash s}\ \mathrm{sig}^+ \qquad \frac{p \xrightarrow[E*s^-]{E', k} p' \qquad s^+ \notin E'}{p\backslash s \xrightarrow[E]{E'\backslash s, k} p'\backslash s}\ \mathrm{sig}^-$$

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

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

**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 $\bot$: the execution is blocked on the test. Only the signal declaration rule deals with the $\bot$ 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 $\bot$ (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 $\{\bot, +, -\}$, defined by the following equations.

$$+ \vee x = x \vee + = + \qquad - \vee \bot = \bot \vee - = \bot \vee \bot = \bot \qquad - \vee - = -$$

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 $\bot$ 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:

$$\frac{p \xrightarrow[E*s^+]{E',\,k} p' \qquad s \in \mathit{Must}(p, E * s^\bot)}{p\backslash s \xrightarrow[E]{E'\backslash s,\,k} \delta(k, p'\backslash s)}\ \text{C-sig}^+ \qquad \frac{p \xrightarrow[E*s^-]{E',\,k} p' \qquad s \notin \mathit{Can}^+(p, E * s^\bot)}{p\backslash s \xrightarrow[E]{E'\backslash s,\,k} \delta(k, p'\backslash s)}\ \text{C-sig}^-$$

## 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 $p\backslash s$. For instance, when the status of $s$ is $\bot$ 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 $Must(p\backslash s, E)$ 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 $p\backslash s$ 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 $\bot$ 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 $\mathrm{Max}(K_1, K_2)$ 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 $X\backslash e$ 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:

$$\begin{aligned} Can^{\bot}(p, E) \cup Can^{\bot}(q, E) &= \big(Can^{\bot}_s(p, E) \cup Can^{\bot}_s(q, E), Can^{\bot}_k(p, E) \cup Can^{\bot}_k(q, E)\big) \\ Can^{m}(p, E)\backslash 0 &= (Can^{m}_s(p, E), Can^{m}_k(p, E)\backslash 0) \\ Must(p, E)\backslash s &= ((Must_s(p, E))\backslash s, Must_k(p, E)) \end{aligned}$$

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 $\bot$ (maybe executed). In Coq, we use a Boolean instead with true representing $+$ and false representing $\bot$.*

## 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 $\bot$, 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 $p'_1 = p'_2$.*

$$Must(k, E) = Can^m(k, E) = (\varnothing, \{k\})$$

$$Must(!s, E) = Can^m(!s, E) = (\{s\}, \{0\})$$

$$Must(@s, E) = \begin{cases} (\varnothing, \{0\}) & \text{if } s^+ \in E \\ (\varnothing, \{1\}) & \text{if } s^- \in E \\ (\varnothing, \varnothing) & \text{otherwise} \end{cases} \qquad Can^m(@s, E) = \begin{cases} (\varnothing, \{0\}) & \text{if } s^+ \in E \\ (\varnothing, \{1\}) & \text{if } s^- \in E \\ (\varnothing, \{0, 1\}) & \text{otherwise} \end{cases}$$

$$Must(s \supset p, E) = Must(p, E) \qquad Can^m(s \supset p, E) = Can^m(p, E)$$

$$Must(p*, E) = Must(p, E) \qquad Can^m(p*, E) = Can^m(p, E)$$

$$Must(\{p\}, E) = (Must_s(p, E), \downarrow(Must_k(p, E))) \qquad Can^m(\{p\}, E) = (Can^m_s(p, E), \downarrow(Can^m_k(p, E)))$$

$$Must(\uparrow p, E) = (Must_s(p, E), \uparrow(Must_k(p, E))) \qquad Can^m(\uparrow p, E) = (Can^m_s(p, E), \uparrow(Can^m_k(p, E)))$$

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

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

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

$$Can^m(p\,;\, q, E) = \begin{cases} Can^m(p, E) & \text{if } 0 \notin Can^m_k(p, E) \\ (Can^m(p, E) \backslash 0) \cup Can^+(q, E) & \text{if } 0 \in Must_k(p, E) \text{ and } m = + \\ (Can^m(p, E) \backslash 0) \cup Can^\perp(q, E) & \text{otherwise} \end{cases}$$

$$Must(p \mid q, E) = (Must_s(p, E) \cup Must_s(q, E), \mathrm{Max}(Must_k(p, E), Must_k(q, E)))$$

$$Can^m(p \mid q, E) = (Can^m_s(p, E) \cup Can^m_s(q, E), \mathrm{Max}(Can^m_k(p, E), Can^m_k(q, E)))$$

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

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

**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 $Must(p, E) = Can^+(p, E)$.*

▶ 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 $E$, $Must_s((s\,?\,p\,,\,p)\backslash s, E) = \varnothing$ whereas $Can^+_s((s\,?\,p\,,\,p)\backslash s, E) = Can^+_s(p, E)$. 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 $Must(p, E) \subseteq (E', \{k\}) \subseteq Can^+(p, E)$.*

Notice that we use the logical behavioral semantics (LBS) here. If we have $Must_k(p, E) \neq \emptyset$, 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 $\texttt{P1} = s\,?\,!s\,,\,0$ 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 $Must((s\,?\,!s\,,\,0), E * s^\perp)$, 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 $Can^+((s\,?\,!s\,,\,0), E * s^\perp) = (\{s\}, \{0\})$ and rule C-sig$^-$ cannot be applied either. The exact same reasoning applies to $\texttt{P3} = s\,?\,!s\,,\,!s$.

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 $\texttt{Total}(E)$ and $p \xrightarrow[E]{E', k} p'$ then $p \xrightarrow[\texttt{C2K}(E)]{\texttt{C2K}(E'), k} p'$.*

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

**Proof of Theorem 12.** By induction on the proof of $p \xhookrightarrow[E]{E', k} p'$.
As both semantics are the same except for signal declaration, only the C-sig$^+$ and C-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 $Must(p, E)$ is indeed emitted (resp. reached) and any emitted signal or reached completion code indeed belongs to the corresponding set computed by $Can^+(p, E)$. ◀

## 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:

$$!o_1\,;\,1\,;\,s\,?\,(1\,;\,!o_2) \mid (!o_3\,;\,1)\,,\,(1\,;\,!o_4) \xhookrightarrow[\{s\}]{\{o_1\},\,1} 0\,;\,s\,?\,(1\,;\,!o_2 \mid !o_3\,;\,1)\,,\,(1\,;\,!o_4)$$
$$\xhookrightarrow[\{s\}]{\{o_3\},\,1} (0\,;\,!o_2) \mid 0 \xhookrightarrow[\{s\}]{\{o_2\},\,0} 0$$

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

$$\widehat{1}\,;\,!o_1\,;\,1\,;\,s\,?\,(1\,;\,!o_2) \mid (!o_3\,;\,1)\,,\,(1\,;\,!o_4) \xhookrightarrow[\{s\}]{\{o_1\},\,1} 1\,;\,!o_1\,;\,\widehat{1}\,;\,s\,?\,(1\,;\,!o_2) \mid (!o_3\,;\,1)\,,\,(1\,;\,!o_4)$$
$$\xhookrightarrow[\{s\}]{\{o_3\},\,1} 1\,;\,!o_1\,;\,1\,;\,s\,?\,(\widehat{1}\,;\,!o_2) \mid (!o_3\,;\,\widehat{1})\,,\,(1\,;\,!o_4)$$
$$\xhookrightarrow[\{s\}]{\{o_2\},\,0} 1\,;\,!o_1\,;\,1\,;\,s\,?\,(1\,;\,!o_2) \mid (!o_3\,;\,1)\,,\,(1\,;\,!o_4)$$

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 $\widehat{\cdot}$ 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:

$$
\begin{array}{llll}
\text{States} & \widehat{p}, \widehat{q} & := & \widehat{1} \\
 & & & \widehat{@s} \\
 & & & s\,?\,\widehat{p}\,,\, q \qquad s\,?\,p\,,\,\widehat{q} \\
 & & & s \supset \widehat{p} \\
 & & & \{\widehat{p}\} \\
 & & & \uparrow \widehat{p} \\
 & & & \widehat{p}\,;\, q \qquad p\,;\,\widehat{q} \\
 & & & \widehat{p}\,* \\
 & & & \widehat{p}\,|\,q \qquad p\,|\,\widehat{q} \qquad \widehat{p}\,|\,\widehat{q} \\
 & & & \widehat{p}\backslash s
\end{array}
$$

The only elementary states are the active 1 (`pause`) and the active $@s$ (`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 "$(!s_1 \mid 1)\,;\,!s_2$" is "$(!s_1 \mid \widehat{1})\,;\,!s_2$" as there is only one occurrence of 1 or $@s$. The `ABROi` 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 $p \xhookrightarrow[E]{E',\,k} p'$ 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.

$$
\textit{Start}\ \text{semantics} \quad p \xhookrightarrow[E]{E',\,k}_{\mathrm{s}} \overline{p'} \qquad\qquad \textit{Resumption}\ \text{semantics} \quad \widehat{p} \xhookrightarrow[E]{E',\,k}_{\mathrm{r}} \overline{p'}
$$

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.

$$\mathcal{E}(\widehat{1}) := 0 \qquad \mathcal{E}(\widehat{p}*) := \mathcal{E}(\widehat{p}); p* \qquad \mathcal{E}(\widehat{p} \mid \widehat{q}) := \mathcal{E}(\widehat{p}) \mid \mathcal{E}(\widehat{q})$$
$$\mathcal{E}(\widehat{@s}) := @s \qquad \mathcal{E}(\{\widehat{p}\}) := \{\mathcal{E}(\widehat{p})\} \qquad \mathcal{E}(\widehat{p} \mid q) := \mathcal{E}(\widehat{p}) \mid 0$$
$$\mathcal{E}(s\,?\,\widehat{p}\,,\, q) := \mathcal{E}(\widehat{p}) \qquad \mathcal{E}(\uparrow \widehat{p}) := \uparrow \mathcal{E}(\widehat{p}) \qquad \mathcal{E}(p \mid \widehat{q}) := 0 \mid \mathcal{E}(\widehat{q})$$
$$\mathcal{E}(s\,?\,p\,,\, \widehat{q}) := \mathcal{E}(\widehat{q}) \qquad \mathcal{E}(\widehat{p}\,;\, q) := \mathcal{E}(\widehat{p}); q \qquad \mathcal{E}(\widehat{p} \backslash s) := \mathcal{E}(\widehat{p}) \backslash s$$
$$\mathcal{E}(s \supset \widehat{p}) := @\neg s\,;\, s \supset \mathcal{E}(\widehat{p}) \qquad \mathcal{E}(p\,;\, \widehat{q}) := \mathcal{E}(\widehat{q})$$

Recalling that a term $\overline{p}$ is either an active state $\widehat{p}$ or an inert statement $p$, this function is extended to terms $\overline{p}$ 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 $\mathit{Must}(\widehat{p}, E)$ be $\mathit{Must}(\mathcal{E}(\widehat{p}), E)$ and similarly for $\mathit{Can}$, so that we can directly reuse all the already proved properties about $\mathit{Must}$ and $\mathit{Can}$ and have for free the following equivalences:

$$s \in \mathit{Must}_s(\widehat{p}, E) \iff s \in \mathit{Must}_s(\mathcal{E}(\widehat{p}), E) \qquad s \in \mathit{Can}^m_s(\widehat{p}, E) \iff s \in \mathit{Can}^m_s(\mathcal{E}(\widehat{p}), E)$$

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 `ABROi` 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 $p \xrightarrow[E]{E', k}_s \overline{p'}$ then $p = \mathcal{B}(\overline{p'})$.*
*For all $\widehat{p}$, $E$, $E'$, $k$, $p'$, if $\widehat{p} \xrightarrow[E]{E', k}_r \overline{p'}$ then $\mathcal{B}(\widehat{p}) = \mathcal{B}(\overline{p'})$.*

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 $s \in \mathit{Must}_s(\widehat{p}, E) \iff s \in \mathit{Must}_s(\mathcal{E}(\widehat{p}), E)$ and $s \in \mathit{Can}^m_s(\widehat{p}, E) \iff s \in \mathit{Can}^m_s(\mathcal{E}(\widehat{p}), E)$ 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.*

$$\frac{}{0 \xhookrightarrow[E]{\varnothing, 0}_{\mathrm{s}} 0}\ \text{nothing} \qquad \frac{}{1 \xhookrightarrow[E]{\varnothing, 1}_{\mathrm{s}} \widehat{1}}\ \text{pause} \qquad \frac{k \geq 2}{k \xhookrightarrow[E]{\varnothing, k}_{\mathrm{s}} k}\ \text{exit}$$

$$\frac{}{!s \xhookrightarrow[E]{\{s^+\}, 0}_{\mathrm{s}} !s}\ \text{emit}$$

$$\frac{s^+ \in E}{@s \xhookrightarrow[E]{\varnothing, 0}_{\mathrm{s}} @s}\ \text{awimm}^+ \qquad \frac{s^- \in E}{@s \xhookrightarrow[E]{\varnothing, 1}_{\mathrm{s}} \widehat{@s}}\ \text{awimm}^-$$

$$\frac{s^+ \in E \qquad p \xhookrightarrow[E]{E', k}_{\mathrm{s}} \overline{p'}}{s\,?\,p\,,\, q \xhookrightarrow[E]{E', k}_{\mathrm{s}} s\,?\,\overline{p'}\,,\, q}\ \text{then} \qquad \frac{s^- \in E \qquad q \xhookrightarrow[E]{E', k}_{\mathrm{s}} \overline{q'}}{s\,?\,p\,,\, q \xhookrightarrow[E]{E', k}_{\mathrm{s}} s\,?\,p\,,\, \overline{q'}}\ \text{else}$$

$$\frac{p \xhookrightarrow[E]{E', k}_{\mathrm{s}} \overline{p'}}{s \supset p \xhookrightarrow[E]{E', k}_{\mathrm{s}} s \supset \overline{p'}}\ \text{suspend}$$

$$\frac{k \neq 0 \qquad p \xhookrightarrow[E]{E', k}_{\mathrm{s}} \overline{p'}}{p* \xhookrightarrow[E]{E', k}_{\mathrm{s}} \overline{p'}*}\ \text{loop}$$

$$\frac{p \xhookrightarrow[E]{E', k}_{\mathrm{s}} \overline{p'}}{\{p\} \xhookrightarrow[E]{E', \downarrow k}_{\mathrm{s}} \{\overline{p'}\}}\ \text{trap} \qquad \frac{p \xhookrightarrow[E]{E', k}_{\mathrm{s}} \overline{p'}}{\uparrow p \xhookrightarrow[E]{E', \uparrow k}_{\mathrm{s}} \uparrow \overline{p'}}\ \text{shift}$$

$$\frac{k \neq 0 \qquad p \xhookrightarrow[E]{E', k}_{\mathrm{s}} \overline{p'}}{p\,;\, q \xhookrightarrow[E]{E', k}_{\mathrm{s}} \overline{p'}\,;\, q}\ \text{seq}_k \qquad \frac{p \xhookrightarrow[E]{E'_p, 0}_{\mathrm{s}} p \qquad q \xhookrightarrow[E]{E'_q, k}_{\mathrm{s}} \overline{q'}}{p\,;\, q \xhookrightarrow[E]{E'_p \cup E'_q, k}_{\mathrm{s}} p\,;\, \overline{q'}}\ \text{seq}_0$$

$$\frac{p \xhookrightarrow[E]{E'_p, k_p}_{\mathrm{s}} \overline{p'} \qquad q \xhookrightarrow[E]{E'_q, k_q}_{\mathrm{s}} \overline{q'}}{p \mid q \xhookrightarrow[E]{E'_p \cup E'_q, \max(k_p, k_q)}_{\mathrm{s}} \overline{p'} \mid \overline{q'}}\ \text{parallel}$$

$$\frac{s \in \mathit{Must}(p, E * s^\perp) \qquad p \xhookrightarrow[E*s^+]{E', k}_{\mathrm{s}} \overline{p'}}{p\backslash s \xhookrightarrow[E]{E'\backslash s, k}_{\mathrm{s}} \overline{p'}\backslash s}\ \text{C-sig}^+ \qquad \frac{s \notin \mathit{Can}^+(p, E * s^\perp) \qquad p \xhookrightarrow[E*s^-]{E', k}_{\mathrm{s}} \overline{p'}}{p\backslash s \xhookrightarrow[E]{E'\backslash s, k}_{\mathrm{s}} \overline{p'}\backslash s}\ \text{C-sig}^-$$

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

$$\frac{}{\widehat{1} \xhookrightarrow[E]{\varnothing, 0}_{\mathrm{r}} 1}\ \text{pause}$$

$$\frac{s^+ \in E}{\widehat{@s} \xhookrightarrow[E]{\varnothing, 0}_{\mathrm{r}} @s}\ \text{awimm}^+ \qquad \frac{s^- \in E}{\widehat{@s} \xhookrightarrow[E]{\varnothing, 1}_{\mathrm{r}} \widehat{@s}}\ \text{awimm}^-$$

$$\frac{\widehat{p} \xhookrightarrow[E]{E', k}_{\mathrm{r}} \overline{p'}}{s\,?\,\widehat{p}\,,\ q \xhookrightarrow[E]{E', k}_{\mathrm{r}} s\,?\,\overline{p'}\,,\ q}\ \text{then} \qquad \frac{\widehat{q} \xhookrightarrow[E]{E', k}_{\mathrm{r}} \overline{q'}}{s\,?\,p\,,\ \widehat{q} \xhookrightarrow[E]{E', k}_{\mathrm{r}} s\,?\,p\,,\ \overline{q'}}\ \text{else}$$

$$\frac{s^+ \in E}{s \supset \widehat{p} \xhookrightarrow[E]{\varnothing, 1}_{\mathrm{r}} s \supset \widehat{p}}\ \text{suspend}^+ \qquad \frac{s^- \in E \qquad \widehat{p} \xhookrightarrow[E]{E', k}_{\mathrm{r}} \overline{p'}}{s \supset \widehat{p} \xhookrightarrow[E]{E', k}_{\mathrm{r}} s \supset \overline{p'}}\ \text{suspend}^-$$

$$\frac{\widehat{p} \xhookrightarrow[E]{E', k}_{\mathrm{r}} \overline{p'} \qquad k \neq 0}{\widehat{p}* \xhookrightarrow[E]{E', k}_{\mathrm{r}} \overline{p'}*}\ \text{loop}_k \qquad \frac{\widehat{p} \xhookrightarrow[E]{E', 0}_{\mathrm{r}} p \qquad p \xhookrightarrow[E]{E', k}_{\mathrm{s}} \overline{p'} \qquad k \neq 0}{\widehat{p}* \xhookrightarrow[E]{E', k}_{\mathrm{r}} \overline{p'}*}\ \text{loop}_0$$

$$\frac{\widehat{p} \xhookrightarrow[E]{E', k}_{\mathrm{r}} \overline{p'}}{\{\widehat{p}\} \xhookrightarrow[E]{E', \downarrow k}_{\mathrm{r}} \{\overline{p'}\}}\ \text{trap} \qquad \frac{\widehat{p} \xhookrightarrow[E]{E', k}_{\mathrm{r}} \overline{p'}}{\uparrow \widehat{p} \xhookrightarrow[E]{E', \uparrow k}_{\mathrm{r}} \uparrow \overline{p'}}\ \text{shift}$$

$$\frac{\widehat{p} \xhookrightarrow[E]{E', k}_{\mathrm{r}} \overline{p'} \qquad k \neq 0}{\widehat{p}\,;\ q \xhookrightarrow[E]{E', k}_{\mathrm{r}} \overline{p'}\,;\ q}\ \text{seq}_k \qquad \frac{\widehat{p} \xhookrightarrow[E]{E'_p, 0}_{\mathrm{r}} p \qquad q \xhookrightarrow[E]{E'_q, k}_{\mathrm{s}} \overline{q'}}{\widehat{p}\,;\ q \xhookrightarrow[E]{E'_p \cup E'_q, k}_{\mathrm{r}} p\,;\ \overline{q'}}\ \text{seq}_0 \qquad \frac{\widehat{q} \xhookrightarrow[E]{E'_q, k}_{\mathrm{r}} \overline{q'}}{p\,;\ \widehat{q} \xhookrightarrow[E]{E'_q, k}_{\mathrm{r}} p\,;\ \overline{q'}}\ \text{seq}_q$$

$$\frac{\widehat{p} \xhookrightarrow[E]{E'_p, k_p}_{\mathrm{r}} \overline{p'} \qquad \widehat{q} \xhookrightarrow[E]{E'_q, k_q}_{\mathrm{r}} \overline{q'}}{\widehat{p} \mid \widehat{q} \xhookrightarrow[E]{E'_p \cup E'_q,\, \max(k_p, k_q)}_{\mathrm{r}} \overline{p'} \mid \overline{q'}}\ \text{both}$$

$$\frac{\widehat{p} \xhookrightarrow[E]{E'_p, k_p}_{\mathrm{r}} \overline{p'}}{\widehat{p} \mid q \xhookrightarrow[E]{E'_p, k_p}_{\mathrm{r}} \overline{p'} \mid q}\ \text{left} \qquad \frac{\widehat{q} \xhookrightarrow[E]{E'_q, k_q}_{\mathrm{r}} \overline{q'}}{p \mid \widehat{q} \xhookrightarrow[E]{E'_q, k_q}_{\mathrm{r}} p \mid \overline{q'}}\ \text{right}$$

$$\frac{s \in \mathit{Must}(\mathcal{E}(p), E * s^{\perp}) \qquad \widehat{p} \xhookrightarrow[E*s^+]{E', k}_{\mathrm{r}} \overline{p'}}{\widehat{p}\backslash s \xhookrightarrow[E]{E'\backslash s, k}_{\mathrm{r}} \overline{p'}\backslash s}\ \text{C-sig}^+$$

$$\frac{s \notin \mathit{Can}^+(\mathcal{E}(p), E * s^{\perp}) \qquad \widehat{p} \xhookrightarrow[E*s^-]{E', k}_{\mathrm{r}} \overline{p'}}{\widehat{p}\backslash s \xhookrightarrow[E]{E'\backslash s, k}_{\mathrm{r}} \overline{p'}\backslash s}\ \text{C-sig}^-$$

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

$$
\begin{aligned}
\widehat{1}\,;\,\texttt{ABROi} \quad &= \quad \widehat{1}\,;\,(\{(1\,;\,(R?2\,,\,1)*)\mid(((@A\mid@B)\,;\,!O\,;\,(1*))\,;\,2)\})* \\
&\xrightarrow[\{B\}]{\emptyset,1}_{\mathrm{r}} \quad 1\,;\,(\{(\widehat{1}\,;\,(R?2\,,\,1)*)\mid(((\widehat{@A}\mid@B)\,;\,!O\,;\,(1*))\,;\,2)\})* \\
&\xrightarrow[\{A,B\}]{\{O\},1}_{\mathrm{r}} \quad 1\,;\,(\{(1\,;\,(R?2\,,\,\widehat{1})*)\mid(((@A\mid@B)\,;\,!O\,;\,(\widehat{1}*))\,;\,2)\})* \\
&\xrightarrow[\{B\}]{\emptyset,1}_{\mathrm{r}} \quad 1\,;\,(\{(1\,;\,(R?2\,,\,\widehat{1})*)\mid(((@A\mid@B)\,;\,!O\,;\,(\widehat{1}*))\,;\,2)\})* \\
&\xrightarrow[\{R\}]{\emptyset,1}_{\mathrm{r}} \quad 1\,;\,(\{(\widehat{1}\,;\,(R?2\,,\,1)*)\mid(((\widehat{@A}\mid\widehat{@B})\,;\,!O\,;\,(1*))\,;\,2)\})* \\
&\xrightarrow[\{A,B,R\}]{\{O\},1}_{\mathrm{r}} \quad 1\,;\,(\{(\widehat{1}\,;\,(R?2\,,\,1)*)\mid(((@A\mid@B)\,;\,!O\,;\,(\widehat{1}*))\,;\,2)\})*
\end{aligned}
$$

**Figure 7** Execution of `ABROi` 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)**.**

$$\textit{For all } p,E,E',k,\overline{p'},\quad p \xhookrightarrow[E]{E',k}_{s} \overline{p'} \implies p \xhookrightarrow[E]{E',k} \mathcal{E}(\overline{p'})$$

$$\textit{For all } \widehat{p},E,E',k,\overline{p'},\quad \widehat{p} \xhookrightarrow[E]{E',k}_{r} \overline{p'} \implies \mathcal{E}(\widehat{p}) \xhookrightarrow[E]{E',k} \mathcal{E}(\overline{p'})$$

$$\textit{For all } p,E,E',k,p',\quad p \xhookrightarrow[E]{E',k} p' \implies \exists \overline{p''}, p \xhookrightarrow[E]{E',k}_{s} \overline{p''} \wedge p' = \mathcal{E}(\overline{p''})$$

$$\textit{For all } \widehat{p},E,E',k,p',\quad \mathcal{E}(\widehat{p}) \xhookrightarrow[E]{E',k} p' \implies \exists \overline{p''}, \widehat{p} \xhookrightarrow[E]{E',k}_{r} \overline{p''} \wedge p' = \mathcal{E}(\overline{p''})$$

**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 $@\neg s$ 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 predicable 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.

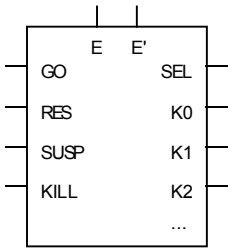



**Figure 8** Circuit Translation Interface, taken from [4].

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 $\mathrm{K}_i$ represent the completion code in one-hot encoding: completion code $k$ is represented by wire $\mathrm{K}_k$ being one and all other wires $\mathrm{K}_i$ ($i \neq k$) 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 $\widehat{\cdot}$ 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 $s \supset \widehat{p}$ (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 satement 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 $\mathrm{K}_i$. 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 ◪ $p$ ◑, defined as follows.

Inputs ◪ consist in the Go and Res wires (as constructive Booleans that can take values $\bot$, $+$, 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 ◑ consist in the completion code represented as either $\bullet_k$ with $k$ an integer, meaning that wire $\mathrm{K}_k$ is 1 and all other wires are 0, or $\circ_K$ with $K$ a finite set of integers, meaning that wires $\mathrm{K}_i$ with $i \in K$ are $\bot$ 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 $\bot$ case and can use a simple Boolean.

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

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

**Figure 9** Definition of microstates.

### 7.1.3 Notations

We use the $\circ_{K\backslash i}$ notation to denote setting wire $i$ to 0 inside $\circ_K$ instead of the more standard but cumbersome $\circ_{K\backslash\{i\}}$. We abbreviate $val = b$ as $val^b$, both when used as an equality or as an assignment. We let $\mathrm{in}(p)$, $\mathrm{out}(p)$, $Go(p)$, and $Res(p)$ denote respectively the input color, output color, Go wire, and Res wire of a microstate $p$; and let $Go(◪)$ and $Res(◪)$ denote the Go and Res wires of an input color ◪. To change the input color of a microstate $p$ into ◪, we write $[◪]p$. 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 $s \in \mathrm{dom}(E)$ and $\bot$ 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 $\text{Sel}^-$ |
| $\square^+$ | − | + | ? | Starting an inactive statement |
| $\square^-$ | − | − | ? | Keeping inactive an inactive statement |
| ■ | + | ? | ? | We know only $\text{Sel}^+$ |
| $\blacksquare^+$ | + | ? | + | Resuming an active statement |
| $\blacksquare^-$ | + | ? | − | Not executing an active statement |

When we have $\text{Sel}^-$ (that is, in the □, $\square^+$, and $\square^-$ cases), Theorem 24 will formally justify that the value of Res is irrelevant. Similarly, an invariant of the circuit translation, called $\mathit{input\_invariant}(\mathit{Sel}, ◪)$ and defined as $\mathit{Go}(◪)^+ \implies \mathit{Sel}^-$, 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 $\bot$ or $-$ but they cannot be $+$. Nevertheless, even for active microstates (having $\text{Sel}^+$) for which Go can only become $\text{Go}^-$, having $\text{Go}^-$ still provide more information than $\text{Go}^\bot$ and may be necessary for some microstates. For instance, in the microstate built from $\widehat{p} \mid q$, even though the overall statement has $\text{Sel}^+$, we need $\text{Go}^-$ to know that $q$ is not executed and that its completion code will become $\circ_\varnothing$.

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

- $◪^+$: either $\square^+$ or $\blacksquare^+$, that is, $\mathit{Go}^+ \vee (\mathit{Sel}^+ \wedge \mathit{Res}^+)$
  The intuition is that the microstate is executed, either started fresh (if inactive) or resumed (if active). The input invariant expresses that $\mathit{Go}^+$ entails $\mathit{Sel}^-$, so that we do not need to add $\mathit{Sel}^-$ in the left disjunct.
- $◪^-$: either $\square^-$ or $\blacksquare^-$, that is, $\mathit{Go}^- \wedge (\mathit{Sel}^- \vee \mathit{Res}^-)$
  The intuition is that the microstate is not executed: neither started nor resumed. This definition is a bit more restrictive than $(\mathit{Go}^- \wedge \mathit{Sel}^-) \vee (\mathit{Res}^- \wedge \mathit{Sel}^+)$, which is the direct translation of $\square^-$ or $\blacksquare^-$, because we also require $\mathit{Go}^-$ in the $\mathit{Sel}^+$ case. Nevertheless, in that case we know that $\mathit{Go}$ can only become $\mathit{Go}^-$ and having $\mathit{Go}^-$ 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 $\bot$ and where all completion codes are either $\circ_\varnothing$ (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 $\widehat{p'}$ 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, $\bot < +$ and $\bot < -$).

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 $\bot$ and all output colors are either $\circ_\varnothing$ 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 $\overline{p}$ (that is, an inactive statement $p$ or an active state $\widehat{p}$) 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.

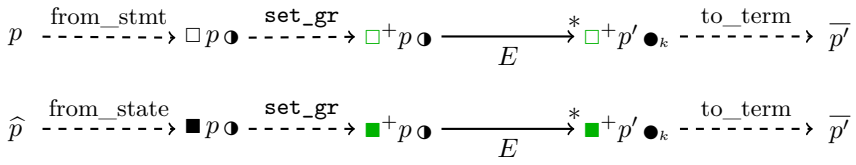



**Figure 10** Links between the state and microstep semantics.

The difference between from_stmt and from_state only lies in how Sel is computed: in from_stmt, it is always $\text{Sel}^-$ whereas in from_state the active parts of a state $\widehat{p}$ have $\text{Sel}^+$. 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, $\square^+\texttt{pause}\,\bullet_1$ should become $\widehat{1}$ whereas $\square^-\texttt{pause}\,\circ_\varnothing$ should become 1, so that we cannot translate $\blacklozenge\texttt{pause}\,\lozenge$ 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 $\bot$. 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 $\boxplus!s\,\bullet_0$, $s$ is emitted so we set $s$'s status to $+$;

***(ii)*** if all emitters of $s$ are not executed, that is, they are all $\boxminus!s\,\circ_\varnothing$, $s$ is not emitted and its status is $-$;

***(iii)*** otherwise, some emitter of $s$ is neither executed nor not executed, that is, it is neither $\boxplus!s\,\bullet_0$ nor $\boxminus!s\,\circ_\varnothing$ (hence it is $\boxslash!s\,\circ_{\{0\}}$), and the status of $s$ is $\bot$.

To distinguish these three cases, we can ignore input colors and only look at output colors: $\bullet_0$ means case ***(i)***; $\circ_\varnothing$ 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 $s$.

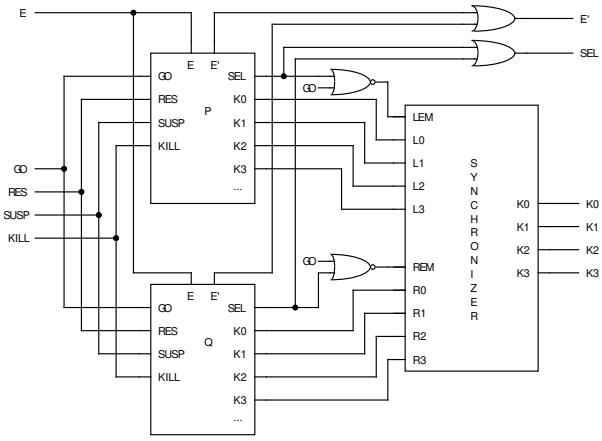
#### 7.3.1.1 The 0, $k$, $!s$ rules

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 $\square$, $\boxplus$ and $\boxminus$. If the statement is started (that is, the input is $\boxplus$), we execute the statement. If the statement is not started (that is, the input is $\boxminus$), 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 $\bot$) and no rule applies.

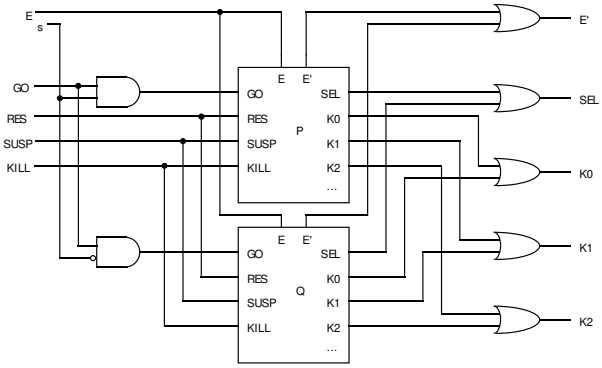
$$\frac{}{\boxplus 0\,\circ_{\{0\}} \xrightarrow[E]{} \boxplus 0\,\bullet_0}\ \text{nothingGo} \qquad \frac{}{\boxminus 0\,\circ_{\{0\}} \xrightarrow[E]{} \boxminus 0\,\circ_\varnothing}\ \text{nothingNoGo}$$

$$\frac{}{\boxplus n\,\circ_{\{n+2\}} \xrightarrow[E]{} \boxplus n\,\bullet_{n+2}}\ \text{exitGo} \qquad \frac{}{\boxminus n\,\circ_{\{n+2\}} \xrightarrow[E]{} \boxminus n\,\circ_\varnothing}\ \text{exitNoGo}$$

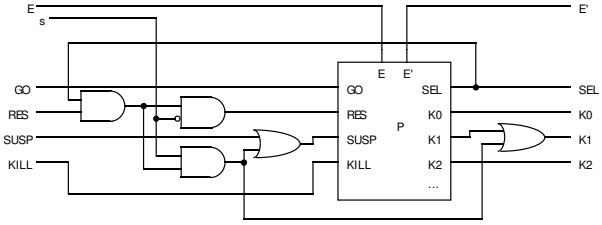
$$\frac{}{\boxplus !s\,\circ_{\{0\}} \xrightarrow[E]{} \boxplus !s\,\bullet_0}\ \text{emitGo} \qquad \frac{}{\boxminus !s\,\circ_{\{0\}} \xrightarrow[E]{} \boxminus !s\,\circ_\varnothing}\ \text{emitNoGo}$$



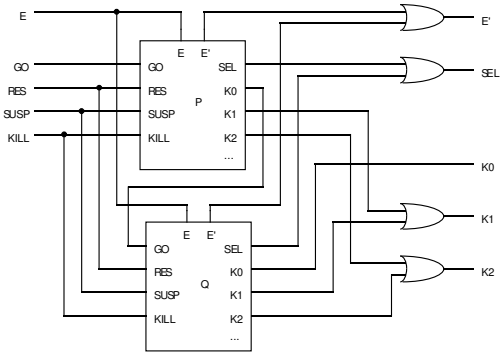

**(a)** The $p \mid q$ statement (the synchronizer computes the max completion code).



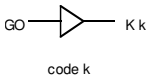

**(b)** The $s\,?\,p\,,\,q$ statement.



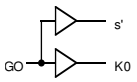

**(c)** The $s \supset p$ statement.



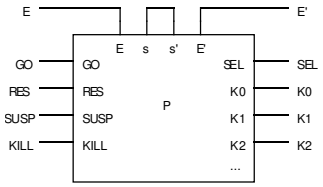

**(j)** The $p\,;\,q$ statement.



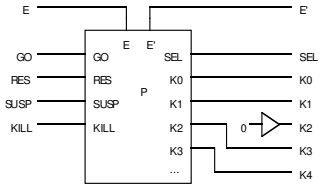

**(d)** The $k$ statement.



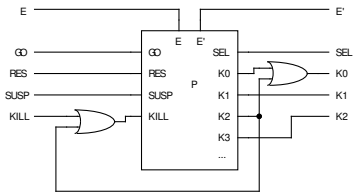

**(e)** The $!s$ statement.



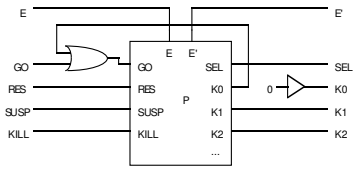

**(f)** The $p \backslash s$ statement.



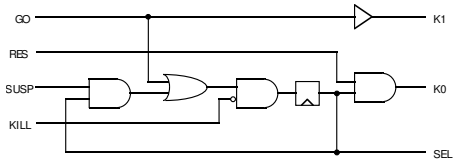

**(g)** The $\uparrow p$ statement.




**(h)** The $\{p\}$ statement.




**(i)** The $p*$ statement.




**(k)** The 1 statement.

**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 ◪1◑ statement is started, that is, we have $Go(◪)^+$, then the output should become $●_1$. In order to trigger this rule only when it increases information, we add the precondition $◑ < ●_1$.

$$\frac{Go(◪)^+ \qquad ◑ < ●_1}{◪\texttt{pause}\,◑ \underset{E}{\longrightarrow} ◪\texttt{pause}\,●_1}\ \text{pauseGo}$$

▶ Remark 18. Thanks to the input invariant (that is, $Go(◪)^+ \implies Sel^-$), $Go(◪)^+$ actually means $□^+$. 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:

$$\frac{Go(◪)^- \qquad 1 \in K}{◪\texttt{pause}\,○_K \underset{E}{\longrightarrow} ◪\texttt{pause}\,○_{K\setminus 1}}\ \text{pauseNoGo}$$

Similarly, for Res, we have:

$$\frac{◑ < ●_0}{■^+\texttt{pause}\,◑ \underset{E}{\longrightarrow} ■^+\texttt{pause}\,●_0}\ \text{pauseRes} \qquad \frac{Res(◪)^- \vee Sel^- \qquad 0 \in K}{◪\texttt{pause}\,○_K \underset{E}{\longrightarrow} ◪\texttt{pause}\,○_{K\setminus 0}}\ \text{pauseNoRes}$$

#### 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:

$$\frac{◪ = ◪^+ \qquad s^- \in E \qquad ◑ < ●_1}{◪\,@s\,◑ \underset{E}{\longrightarrow} ◪\,@s\,●_1}\ \text{awimmM}$$

$$\frac{◪ = ◪^+ \qquad s^+ \in E \qquad ◑ < ●_0}{◪\,@s\,◑ \underset{E}{\longrightarrow} ◪\,@s\,●_0}\ \text{awimmP}$$

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:

$$\frac{◪ = ◪^- \vee s^- \in E \qquad 0 \in K}{◪\,@s\,○_K \underset{E}{\longrightarrow} ◪\,@s\,○_{K\setminus 0}}\ \text{awimmNoGo0} \qquad \frac{◪ = ◪^- \vee s^+ \in E \qquad 1 \in K}{◪\,@s\,○_K \underset{E}{\longrightarrow} ◪\,@s\,○_{K\setminus 1}}\ \text{awimmNoGo1}$$

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 parts[7]:

- 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.

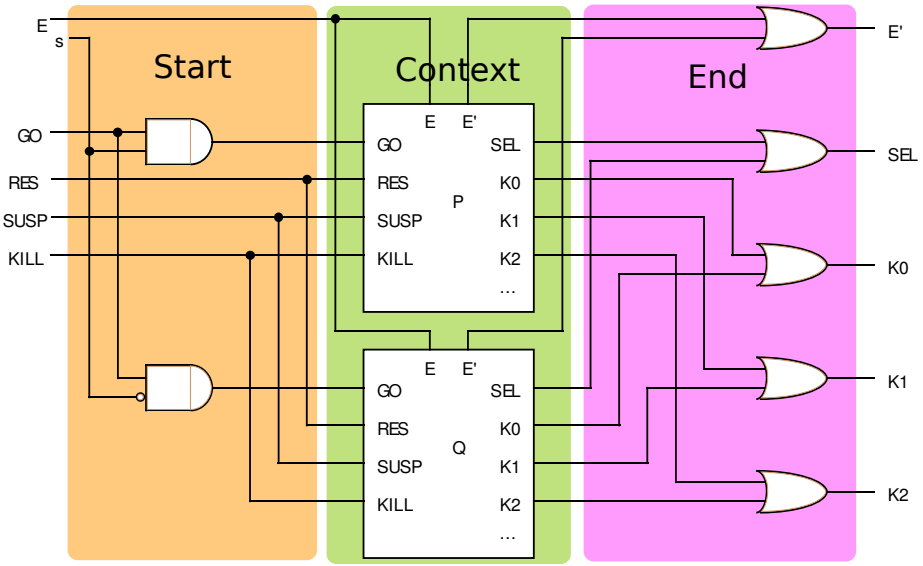



**Figure 12** The start, context and end delimitation in the circuit translation of the $s\,?\,p\,,\,q$ statement.

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 $p\backslash s$ 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, signalI.*

#### 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 $\mathrm{K}_i$. (Remember that we ignore the computation of Sel.) The start rules for $p$ merely propagate the Res wire from $s\,?\,p\,,\,q$ to $p$:

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

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 ($Go(\blacksquare(s\,?\,p\,,\,q)\,\blacklozenge)^+$) and $s$ is present ($E(s) = +$). More generally, $Go(p)$ is the conjunction of $Go(\blacksquare(s\,?\,p\,,\,q)\,\blacklozenge) = Go(\blacksquare)$ and $E(s)$.

$$\frac{Go(p) < Go(◪) \wedge E(s)}{◪(s\,?\,p\,,\,q)\,◗ \underset{E}{\longrightarrow} ◪(s\,?\,[Go^{Go(◪)\wedge E(s)}]p\,,\,q)\,◗}\ \text{ifGoL}$$

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

$$\frac{Go(q) < Go(◪) \wedge \neg E(s)}{◪(s\,?\,p\,,\,q)\,◗ \underset{E}{\longrightarrow} ◪(s\,?\,p\,,\,[Go^{Go(◪)\wedge \neg E(s)}]q)\,◗}\ \text{ifGoR}$$

$$\frac{Res(q) < Res(◪)}{◪(s\,?\,p\,,\,q)\,◗ \underset{E}{\longrightarrow} ◪(s\,?\,p\,,\,[Res(◪)]q)\,◗}\ \text{ifResR}$$

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 $◪\,p\,◗$, is the correct microstate to use as a premise.

$$\frac{p \underset{E}{\longrightarrow} p'}{◪(s\,?\,p\,,\,q)\,◗ \underset{E}{\longrightarrow} ◪(s\,?\,p'\,,\,q)\,◗}\ \text{ifCL} \qquad \frac{q \underset{E}{\longrightarrow} q'}{◪(s\,?\,p\,,\,q)\,◗ \underset{E}{\longrightarrow} ◪(s\,?\,p\,,\,q')\,◗}\ \text{ifCR}$$

$$\frac{◗ < \text{out}(p) \cup \text{out}(q)}{◪(s\,?\,p\,,\,q)\,◗ \underset{E}{\longrightarrow} ◪(s\,?\,p\,,\,q)(\text{out}(p) \cup \text{out}(q))}\ \text{ifE}$$

#### 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.

$$\frac{Go(p) < Go(◪)}{◪\,s \supset p\,◗ \underset{E}{\longrightarrow} ◪\,s \supset ([Go(◪)]p)\,◗}\ \text{suspendGo}$$

$$\frac{Res(p) < \overbrace{Res(◪) \wedge Sel(p) \wedge \neg E(s)}^{res}}{◪(s \supset p)\,◗ \underset{E}{\longrightarrow} ◪\,s \supset ([Res^{res}]p)\,◗}\ \text{suspendRes}$$

The context rule executes $p$:

$$\frac{p \underset{E}{\longrightarrow} p'}{◪(s \supset p)\,◗ \underset{E}{\longrightarrow} ◪(s \supset p')\,◗}\ \text{suspendC}$$

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

$$\text{SuspNow}(bo, ◗) = \begin{cases} ●_1 & \text{if } bo = + \\ ◗ & \text{if } bo = - \\ ○_{\{1\}} \cup ◗ & \text{if } bo = \bot \end{cases}$$

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 $\bot$, 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, $Res(◪) \wedge Sel(p) = +$, but $s$ was present, that is, $E(s) = +$.

$$\frac{\blacklozenge < \mathrm{SuspNow}(\overbrace{Res(\blacksquare) \wedge Sel(p) \wedge E(s)}^{susp}, \mathrm{out}(p))}{\blacksquare(s \supset p)\blacklozenge \underset{E}{\longrightarrow} \blacksquare(s \supset p)\,\mathrm{SuspNow}(susp, \mathrm{out}(p))}\ \text{suspendE}$$

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

$$\frac{\blacklozenge < \mathrm{SuspNow}(\bot, \mathrm{out}(p))}{\blacksquare(s \supset p)\blacklozenge \underset{E}{\longrightarrow} \blacksquare(s \supset p)\,\mathrm{SuspNow}(\bot, \mathrm{out}(p))}\ \text{suspendEKO}$$

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 $p\backslash s$ 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.

$$\frac{in(p) < \blacksquare}{\blacksquare\{p\}\blacklozenge \underset{E}{\longrightarrow} \blacksquare\{[\blacksquare]p\}\blacklozenge}\ \text{trapI} \qquad \frac{in(p) < \blacksquare}{\blacksquare(\uparrow p)\blacklozenge \underset{E}{\longrightarrow} \blacksquare(\uparrow[\blacksquare]p)\blacklozenge}\ \text{shiftI}$$

$$\frac{p \underset{E}{\longrightarrow} p'}{\blacksquare\{p\}\blacklozenge \underset{E}{\longrightarrow} \blacksquare\{p'\}\blacklozenge}\ \text{trapC} \qquad \frac{p \underset{E}{\longrightarrow} p'}{\blacksquare(\uparrow p)\blacklozenge \underset{E}{\longrightarrow} \blacksquare(\uparrow p')\blacklozenge}\ \text{shiftC}$$

$$\frac{\blacklozenge < \downarrow \mathrm{out}(p)}{\blacksquare\{p\}\blacklozenge \underset{E}{\longrightarrow} \blacksquare\{p\}\downarrow \mathrm{out}(p)}\ \text{trapE} \qquad \frac{\blacklozenge < \uparrow \mathrm{out}(p)}{\blacksquare(\uparrow p)\blacklozenge \underset{E}{\longrightarrow} \blacksquare(\uparrow p)\uparrow \mathrm{out}(p)}\ \text{shiftE}$$

#### 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 $p$.

$$\frac{in(p) < \blacksquare}{\blacksquare(p\,;\,q)\blacklozenge \underset{E}{\longrightarrow} \blacksquare(([\blacksquare]p)\,;\,q)\blacklozenge}\ \text{seqIL} \qquad \frac{Res(q) < Res(\blacksquare)}{\blacksquare(p\,;\,q)\blacklozenge \underset{E}{\longrightarrow} \blacksquare(p\,;\,[Res(\blacksquare)]q)\blacklozenge}\ \text{seqResR}$$

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

$$\frac{\mathrm{out}(p) = \bullet_0 \qquad Go(q) = \bot}{\blacksquare(p\,;\,q)\blacklozenge \underset{E}{\longrightarrow} \blacksquare(p\,;\,[Go^+]q)\blacklozenge}\ \text{seqGoR} \qquad \frac{\mathrm{out}(p) \not\leq \bullet_0 \qquad Go(q) = \bot}{\blacksquare(p\,;\,q)\blacklozenge \underset{E}{\longrightarrow} \blacksquare(p\,;\,[Go^-]q)\blacklozenge}\ \text{seqNoGoR}$$

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 $\mathrm{out}(p)$. Hence, we use $\mathrm{out}(p) \not\leq \bullet_0$. This amounts to having $\mathrm{out}(p) = \bullet_k$ with $k \neq 0$ or $\mathrm{out}(p) = \circ_K$ with $0 \notin K$.

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

$$\frac{p \underset{E}{\longrightarrow} p'}{\blacksquare(p\,;\,q)\blacklozenge \underset{E}{\longrightarrow} \blacksquare(p'\,;\,q)\blacklozenge}\ \text{seqCL} \qquad \frac{q \underset{E}{\longrightarrow} q'}{\blacksquare(p\,;\,q)\blacklozenge \underset{E}{\longrightarrow} \blacksquare(p\,;\,q')\blacklozenge}\ \text{seqCR}$$

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

$$\frac{\blacklozenge < (\mathrm{out}(p) \setminus 0) \cup \mathrm{out}(q)}{\blacksquare(s\,?\,p\,,\,q)\blacklozenge \underset{E}{\longrightarrow} \blacksquare(s\,?\,p\,,\,q)(\mathrm{out}(p) \setminus 0) \cup \mathrm{out}(q))}\ \mathrm{seqE}$$

▶ **Coq Remark.** *In the Coq development, the operation* $\blacklozenge \setminus 0$ *is written* `SEQrestrict`$(\blacklozenge)$.

#### 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 $\mathrm{out}(p) = \bullet_0$. 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 $\mathrm{out}(p)[0]$. Resumption is simply propagated. The end rule propagates the output color of $p$ after forcing the 0-th component to false.

$$\frac{Go(p) < Go(\blacksquare) \vee \mathrm{out}(p)[0]}{\blacksquare p*\blacklozenge \underset{E}{\longrightarrow} \blacksquare\left([Go(\blacksquare) \vee \mathrm{out}(p)[0]]p\right)*\blacklozenge}\ \mathrm{loopGo}$$

$$\frac{Res(p) < Res(\blacksquare)}{\blacksquare p*\blacklozenge \underset{E}{\longrightarrow} \blacksquare\left([Res(\blacksquare)]p\right)*\blacklozenge}\ \mathrm{loopRes}$$

$$\frac{p \underset{E}{\longrightarrow} p'}{\blacksquare p*\blacklozenge \underset{E}{\longrightarrow} \blacksquare p'*\blacklozenge}\ \mathrm{loopC}$$

$$\frac{\blacklozenge < \mathrm{out}(p) \setminus 0}{\blacksquare p*\blacklozenge \underset{E}{\longrightarrow} \blacksquare p*(\mathrm{out}(p) \setminus 0)}\ \mathrm{loopE}$$

#### 7.3.2.6 The $p \mid q$ rules

Input and context rules simply propagate control.

$$\frac{in(p) < \blacksquare}{\blacksquare(p \mid q)\blacklozenge \underset{E}{\longrightarrow} \blacksquare([\blacksquare]p \mid q)\blacklozenge}\ \mathrm{parIL} \qquad \frac{in(q) < \blacksquare}{\blacksquare(p \mid q)\blacklozenge \underset{E}{\longrightarrow} \blacksquare(p \mid [\blacksquare]q)\blacklozenge}\ \mathrm{parIR}$$

$$\frac{p \underset{E}{\longrightarrow} p'}{\blacksquare(p \mid q)\blacklozenge \underset{E}{\longrightarrow} \blacksquare(p' \mid q)\blacklozenge}\ \mathrm{parCL} \qquad \frac{q \underset{E}{\longrightarrow} q'}{\blacksquare(p \mid q)\blacklozenge \underset{E}{\longrightarrow} \blacksquare(p \mid q')\blacklozenge}\ \mathrm{parCR}$$

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.

$$\frac{\blacklozenge < \overbrace{\mathrm{synchronizer}_{Sel(p),Sel(q)}(\mathrm{out}(p),\mathrm{out}(q))}^{synch}}{\blacksquare(p \mid q)\blacklozenge \underset{E}{\longrightarrow} \blacksquare(p \mid q)\,synch}\ \mathrm{parE}$$

with

$$\mathrm{synchronizer}_{Sel(p),Sel(q)}(\blacklozenge_p,\blacklozenge_q) = \begin{cases} \mathrm{Max}(\blacklozenge_p,\blacklozenge_q) & \text{when} \quad Sel(p) = Sel(q) \\ \blacklozenge_p & \text{when} \quad Sel(q) = + \ \text{ and } \ Sel(q) = - \\ \blacklozenge_q & \text{when} \quad Sel(p) = - \ \text{ and } \ Sel(q) = + \end{cases}$$

and

$$\begin{array}{llll} \mathrm{Max}(\circ_K,\circ_{K'}) & := & \circ_{\mathrm{Max}(K,K')} & \qquad \mathrm{Max}(\bullet_k,\circ_K) := \circ_{\mathrm{Max}(\{k\},K)} \\ \mathrm{Max}(\bullet_k,\bullet_{k'}) & := & \bullet_{\max(k,k')} & \qquad \mathrm{Max}(\circ_K,\bullet_k) := \circ_{\mathrm{Max}(K,\{k\})} \end{array}$$

#### 7.3.2.7 The $p\backslash s$ rules

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

$$\frac{\mathrm{in}(p) < \blacksquare}{\blacksquare(p\backslash s)\,\blacklozenge \xrightarrow[E]{} \blacksquare([\blacksquare]p)\backslash s\,\blacklozenge}\ \text{signalI} \qquad \frac{\blacklozenge < \mathrm{out}(p)}{\blacksquare(p\backslash s)\,\blacklozenge \xrightarrow[E]{} \blacksquare(p\backslash s)\,\mathrm{out}(p)}\ \text{signalE}$$

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.

$$\frac{b := (\mathrm{to\_event}(p, s^{\perp} * E))(s) \qquad p \xrightarrow[E*s^b]{} p'}{\blacksquare(p\backslash s)\,\blacklozenge \xrightarrow[E]{} \blacksquare(p'\backslash s)\,\blacklozenge}\ \text{signalC}$$

### 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:

$$\mathrm{check} = \Box 1\,\circ_{\{0,1\}} ;\ \Box\left(\Box(R\,?\,\Box 2\,\circ_{\{2\}}\,,\ \Box 1\,\circ_{\{0,1\}})\,\circ_{\{0,1,2\}}\right) * \circ_{\{1,2\}}$$

$$\mathrm{body} = \Box\left(\Box(\Box\,@A\,\circ_{\{0,1\}} \mid \Box\,@B\,\circ_{\{0,1\}})\,\circ_{\{0,1\}} ;\ \Box!O\,\circ_{\{0\}}\right)\circ_{\{0,1\}} ;\ \Box\,(\Box\,1\,\circ_{\{1\}}) * \circ_{\{1\}}$$

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

$$\begin{aligned}
\mathrm{check} &= \Box 1\,\circ_{\{0,1\}} ;\ \Box\left(\Box(R\,?\,\Box 2\,\circ_{\{2\}}\,,\ \Box 1\,\circ_{\{0,1\}})\,\circ_{\{0,1,2\}}\right) * \circ_{\{1,2\}} \\
&\xrightarrow[\{B\}]{}{}^{2}\ \Box 1\underline{\circ_{\{1\}}} ;\ \Box\left(\Box(R\,?\,\underline{\Box^{-}}2\,\circ_{\{2\}}\,,\ \Box 1\,\circ_{\{0,1\}})\,\circ_{\{0,1,2\}}\right) * \circ_{\{1,2\}} \\
&\xrightarrow[\{B\}]{}\ \Box 1\,\circ_{\{1\}} ;\ \Box\left(\Box(R\,?\,\Box^{-}2\underline{\circ_{\varnothing}}\,,\ \Box 1\,\circ_{\{0,1\}})\,\circ_{\{0,1,2\}}\right) * \circ_{\{1,2\}} \\
&\xrightarrow[\{B\}]{}{}^{2}\ \Box 1\,\circ_{\{1\}} ;\ \Box\left(\Box(R\,?\,\Box^{-}2\,\circ_{\varnothing}\,,\ \Box 1\,\circ_{\{0,1\}})\underline{\circ_{\{0,1\}}}\right) * \underline{\circ_{\{1\}}} = \mathrm{check'} \\
\mathrm{body} &= \Box\left(\Box(\Box\,@A\,\circ_{\{0,1\}} \mid \Box\,@B\,\circ_{\{0,1\}})\,\circ_{\{0,1\}} ;\ \Box!O\,\circ_{\{0\}}\right)\circ_{\{0,1\}} ;\ \Box\,(\Box\,1\,\circ_{\{1\}}) * \circ_{\{1\}} \\
&\xrightarrow[\{B\}]{}{}^{2}\ \Box\left(\Box(\Box\,@A\underline{\circ_{\{1\}}} \mid \Box\,@B\underline{\circ_{\{0\}}})\,\circ_{\{0,1\}} ;\ \Box!O\,\circ_{\{0\}}\right)\circ_{\{0,1\}} ;\ \Box\,(\Box\,1\,\circ_{\{1\}}) * \circ_{\{1\}} = \mathrm{body'} \\
\texttt{ABROi} &= \left(\Box\left\{\Box\left(\Box\,\mathrm{check}\,\circ_{\{1,2\}} \mid \Box(\Box\,\mathrm{body}\,\circ_{\{1\}} ;\ \Box\,2\,\circ_{\{2\}})\,\circ_{\{1,2\}}\right)\circ_{\{1,2\}}\right\}\circ_{\{0,1\}}\right) * \\
&\xrightarrow[\{B\}]{}{}^{7}\ \left(\Box\left\{\Box\left(\Box\,\underline{\mathrm{check'}}\,\circ_{\{1,2\}} \mid \Box(\Box\,\underline{\mathrm{body'}}\,\circ_{\{1\}} ;\ \Box\,2\,\circ_{\{2\}})\,\circ_{\{1,2\}}\right)\circ_{\{1,2\}}\right\}\circ_{\{0,1\}}\right) * \\
&\xrightarrow[\{B\}]{}{}^{2}\ \left(\Box\left\{\Box\left(\Box\,\mathrm{check'}\underline{\circ_{\{1\}}} \mid \Box(\Box\,\mathrm{body'}\,\circ_{\{1\}} ;\ \underline{\Box^{-}}2\,\circ_{\{2\}})\,\circ_{\{1,2\}}\right)\circ_{\{1,2\}}\right\}\circ_{\{0,1\}}\right) * \\
&\xrightarrow[\{B\}]{}\ \left(\Box\left\{\Box\left(\Box\,\mathrm{check'}\,\circ_{\{1\}} \mid \Box(\Box\,\mathrm{body'}\,\circ_{\{1\}} ;\ \Box^{-}2\underline{\circ_{\varnothing}})\,\circ_{\{1,2\}}\right)\circ_{\{1,2\}}\right\}\circ_{\{0,1\}}\right) * \\
&\xrightarrow[\{B\}]{}{}^{3}\ \left(\Box\left\{\Box\left(\Box\,\mathrm{check'}\,\circ_{\{1\}} \mid \Box(\Box\,\mathrm{body'}\,\circ_{\{1\}} ;\ \Box^{-}2\,\circ_{\varnothing})\underline{\circ_{\{1\}}}\right)\underline{\circ_{\{1\}}}\right\}\underline{\circ_{\{1\}}}\right) * = \texttt{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}
\boxplus^{+}\texttt{ABROi'}\circ_{\{1\}} &\xrightarrow[\{B\}]{}{}^{5} \boxplus^{+}\Big(\underline{\boxplus^{+}}\,\big\{\underline{\boxplus^{+}}\big(\underline{\boxplus^{+}}\text{check'}\circ_{\{1\}} \mid \underline{\boxplus^{+}}(\underline{\boxplus^{+}}\text{body'}\circ_{\{1\}}\,;\,\boxplus^{-}2\circ_{\varnothing})\circ_{\{1\}}\big)\circ_{\{1\}}\big\}\circ_{\{1\}}\Big)*\circ_{\{1\}}\\
\boxplus^{+}\text{check'}\circ_{\{1\}} &\xrightarrow[\{B\}]{} \boxplus^{+}\Big(\underline{\boxplus^{+}}1\circ_{\{1\}}\,;\,\boxplus\big(\boxplus(R?\boxplus^{-}2\circ_{\varnothing}\,,\,\boxplus 1\circ_{\{0,1\}})\circ_{\{0,1\}}\big)*\circ_{\{1\}}\Big)\circ_{\{1\}}\\
&\xrightarrow[\{B\}]{} \boxplus^{+}\Big(\boxplus^{+}1\underline{\bullet_1}\,;\,\boxplus\big(\boxplus(R?\boxplus^{-}2\circ_{\varnothing}\,,\,\boxplus 1\circ_{\{0,1\}})\circ_{\{0,1\}}\big)*\circ_{\{1\}}\Big)\circ_{\{1\}}\\
&\xrightarrow[\{B\}]{}{}^{3} \boxplus^{+}\Big(\boxplus^{+}1\bullet_1\,;\,\underline{\boxplus^{-}}\big(\underline{\boxplus^{-}}(R?\boxplus^{-}2\circ_{\varnothing}\,,\,\underline{\boxplus^{-}}1\circ_{\{0,1\}})\circ_{\{0,1\}}\big)*\circ_{\{1\}}\Big)\circ_{\{1\}}\\
&\xrightarrow[\{B\}]{}{}^{3} \boxplus^{+}\Big(\boxplus^{+}1\bullet_1\,;\,\boxplus^{-}\big(\boxplus^{-}(R?\boxplus^{-}2\circ_{\varnothing}\,,\,\boxplus^{-}1\underline{\circ_{\varnothing}})\underline{\circ_{\varnothing}}\big)*\underline{\circ_{\varnothing}}\Big)\circ_{\{1\}}\\
&\xrightarrow[\{B\}]{} \boxplus^{+}\Big(\boxplus^{+}1\bullet_1\,;\,\boxplus^{-}\big(\boxplus^{-}(R?\boxplus^{-}2\circ_{\varnothing}\,,\,\boxplus^{-}1\circ_{\varnothing})\circ_{\varnothing}\big)*\circ_{\varnothing}\Big)\underline{\bullet_1}\\
\boxplus^{+}\text{body'}\circ_{\{1\}} &\xrightarrow[\{B\}]{}{}^{4} \boxplus^{+}\Big(\underline{\boxplus^{+}}\big(\underline{\boxplus^{+}}(\underline{\boxplus^{+}}@A\circ_{\{1\}}\mid\underline{\boxplus^{+}}@B\circ_{\{0\}})\circ_{\{0,1\}}\,;\,\boxplus!O\circ_{\{0\}}\big)\circ_{\{0,1\}}\,;\,\boxplus(\boxplus 1\circ_{\{1\}})*\circ_{\{1\}}\Big)\circ_{\{1\}}\\
&\xrightarrow[\{B\}]{}{}^{2} \boxplus^{+}\Big(\boxplus^{+}\big(\boxplus^{+}(\boxplus^{+}@A\underline{\bullet_1}\mid\boxplus^{+}@B\underline{\bullet_0})\circ_{\{0,1\}}\,;\,\boxplus!O\circ_{\{0\}}\big)\circ_{\{0,1\}}\,;\,\boxplus(\boxplus 1\circ_{\{1\}})*\circ_{\{1\}}\Big)\circ_{\{1\}}\\
&\xrightarrow[\{B\}]{} \boxplus^{+}\Big(\boxplus^{+}\big(\boxplus^{+}(\boxplus^{+}@A\bullet_1\mid\boxplus^{+}@B\bullet_0)\underline{\bullet_1}\,;\,\boxplus!O\circ_{\{0\}}\big)\circ_{\{0,1\}}\,;\,\boxplus(\boxplus 1\circ_{\{1\}})*\circ_{\{1\}}\Big)\circ_{\{1\}}\\
&\xrightarrow[\{B\}]{} \boxplus^{+}\Big(\boxplus^{+}\big(\boxplus^{+}(\boxplus^{+}@A\bullet_1\mid\boxplus^{+}@B\bullet_0)\bullet_1\,;\,\underline{\boxplus^{-}}!O\circ_{\{0\}}\big)\circ_{\{0,1\}}\,;\,\boxplus(\boxplus 1\circ_{\{1\}})*\circ_{\{1\}}\Big)\circ_{\{1\}}\\
&\xrightarrow[\{B\}]{} \boxplus^{+}\Big(\boxplus^{+}\big(\boxplus^{+}(\boxplus^{+}@A\bullet_1\mid\boxplus^{+}@B\bullet_0)\bullet_1\,;\,\boxplus^{-}!O\underline{\circ_{\varnothing}}\big)\circ_{\{0,1\}}\,;\,\boxplus(\boxplus 1\circ_{\{1\}})*\circ_{\{1\}}\Big)\circ_{\{1\}}\\
&\xrightarrow[\{B\}]{} \boxplus^{+}\Big(\boxplus^{+}\big(\boxplus^{+}(\boxplus^{+}@A\bullet_1\mid\boxplus^{+}@B\bullet_0)\bullet_1\,;\,\boxplus^{-}!O\circ_{\varnothing}\big)\underline{\bullet_1}\,;\,\boxplus(\boxplus 1\circ_{\{1\}})*\circ_{\{1\}}\Big)\circ_{\{1\}}\\
&\xrightarrow[\{B\}]{}{}^{2} \boxplus^{+}\Big(\boxplus^{+}\big(\boxplus^{+}(\boxplus^{+}@A\bullet_1\mid\boxplus^{+}@B\bullet_0)\bullet_1\,;\,\boxplus^{-}!O\circ_{\varnothing}\big)\bullet_1\,;\,\underline{\boxplus^{-}}(\underline{\boxplus^{-}}1\circ_{\{1\}})*\circ_{\{1\}}\Big)\circ_{\{1\}}\\
&\xrightarrow[\{B\}]{}{}^{2} \boxplus^{+}\Big(\boxplus^{+}\big(\boxplus^{+}(\boxplus^{+}@A\bullet_1\mid\boxplus^{+}@B\bullet_0)\bullet_1\,;\,\boxplus^{-}!O\circ_{\varnothing}\big)\bullet_1\,;\,\boxplus^{-}(\boxplus^{-}1\underline{\circ_{\varnothing}})*\underline{\circ_{\varnothing}}\Big)\circ_{\{1\}}\\
&\xrightarrow[\{B\}]{} \boxplus^{+}\Big(\boxplus^{+}\big(\boxplus^{+}(\boxplus^{+}@A\bullet_1\mid\boxplus^{+}@B\bullet_0)\bullet_1\,;\,\boxplus^{-}!O\circ_{\varnothing}\big)\bullet_1\,;\,\boxplus^{-}(\boxplus^{-}1\circ_{\varnothing})*\circ_{\varnothing}\Big)\underline{\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`:

$$
\begin{aligned}
\boxplus^{+}\texttt{ABROi'}\circ_{\{1\}} &\xrightarrow[\{B\}]{}{}^{5} \boxplus^{+}\Big(\underline{\boxplus^{+}}\,\big\{\underline{\boxplus^{+}}\big(\underline{\boxplus^{+}}\text{check'}\circ_{\{1\}} \mid \underline{\boxplus^{+}}(\underline{\boxplus^{+}}\text{body'}\circ_{\{1\}}\,;\,\boxplus^{-}2\circ_{\varnothing})\circ_{\{1\}}\big)\circ_{\{1\}}\big\}\circ_{\{1\}}\Big)*\circ_{\{1\}}\\
&\xrightarrow[\{B\}]{}{}^{24} \boxplus^{+}\Big(\boxplus^{+}\,\big\{\boxplus^{+}\big(\boxplus^{+}\underline{\text{check"}\bullet_1}\mid\boxplus^{+}(\boxplus^{+}\underline{\text{body"}\bullet_1}\,;\,\boxplus^{-}2\circ_{\varnothing})\circ_{\{1\}}\big)\circ_{\{1\}}\big\}\circ_{\{1\}}\Big)*\circ_{\{1\}}\\
&\xrightarrow[\{B\}]{} \boxplus^{+}\Big(\boxplus^{+}\,\big\{\boxplus^{+}\big(\boxplus^{+}\text{check"}\bullet_1\mid\boxplus^{+}(\boxplus^{+}\text{body"}\bullet_1\,;\,\boxplus^{-}2\circ_{\varnothing})\underline{\bullet_1}\big)\circ_{\{1\}}\big\}\circ_{\{1\}}\Big)*\circ_{\{1\}}\\
&\xrightarrow[\{B\}]{}{}^{3} \boxplus^{+}\Big(\boxplus^{+}\,\big\{\boxplus^{+}\big(\boxplus^{+}\text{check"}\bullet_1\mid\boxplus^{+}(\boxplus^{+}\text{body"}\bullet_1\,;\,\boxplus^{-}2\circ_{\varnothing})\bullet_1\big)\underline{\bullet_1}\big\}\underline{\bullet_1}\Big)*\underline{\bullet_1}
\end{aligned}
$$

From this final microstate, we can read that:

- O is not emitted because its only emitter inside body" is $\boxplus^{-}!O\circ_{\varnothing}$,
- the activated 1 and @$s$ statements are $\boxplus^{+}1\bullet_1$ inside check" and $\boxplus^{+}@A\bullet_1$ inside body".

## 7.4 Properties of the Microstep Semantics

As it is lower-level than previous semantics, the reader my 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:*
  $\forall E,p,p',\ p \xrightarrow[E]{} p' \implies \mathcal{B}(p')=\mathcal{B}(p)\wedge Sel(p')=Sel(p)$;
- *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 $\bot$, 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 valid_dom$(E, p)$ without which the execution of an Esterel program does not makes sense. In particular, valid_coloring$(E, p)$ then implies valid_dom$(E, \mathcal{B}(p))$.

The set of constraints for a microstate $◪\, p \,◑$ 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: $\mathit{input\_invariant}(\mathit{sel}, ◪)$;
- 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: $\mathit{sel} = \mathit{Sel}(p) \,||\, \mathit{Sel}(q)$;
- both $p$ and $q$ cannot be active at the same time: $\mathit{Sel}(p)\ \&\ \mathit{Sel}(q) = \text{false}$;
- $\mathit{Go}(p)$ and $G(q)$ are computed from $◪$ and $b$: $\mathit{Go}(p) \leq \mathit{Go}(◪)\ \&\ b$ and $\mathit{Go}(q) \leq \mathit{Go}(◪)\ \&\ \neg b$;
- the rest of the input color is transmitted to $p$ and $q$: $\mathit{Res}(p) \leq \mathit{Res}(◪)$ and $\mathit{Res}(q) \leq \mathit{Res}(◪)$;
- *out* contains at least the static information: $\text{out}(\text{from\_stmt}(\mathcal{B}(p))) \cup \text{out}(\text{from\_stmt}(\mathcal{B}(p))) \leq ◑$;
- ...and at most the information of $p$ and $q$: $◑ \leq \text{out}(p) \cup \text{out}(q)$;
- recursively, $p$ and $q$ must be well-colored: valid_coloring$(E, p)$ and valid_coloring$(E, q)$.

To ensure that the 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 $p$,* valid_coloring$(E, \mathit{from_stmt}(p))$ *holds ;*
- *For any well-formed state $\widehat{p}$,* valid_coloring$(E, \mathit{from_state}(\widehat{p}))$ *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:
  - valid_coloring$(E, ◪\, p \,●_k) \implies ◪ = ◪^+$;
  - valid_coloring$(E, ◪\, p \,○_\varnothing) \implies ◪ = ◪^-$;
- Branches in a test and a sequence are not simultaneously active:
  - valid_coloring$(E, ◪(s\,?\,p\,,\, q)\,◑) \implies \text{in}(p) \neq ◪^+ \vee \text{in}(q) \neq ◪^+$;
  - valid_coloring$(E, ◪(p\,;\, q)\,◑) \implies \text{out}(p) \setminus 0 \neq ●_k \vee \text{in}(q) \neq ◪^+$;

## 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 $○_\varnothing$. Formally, for all $p$, $◑$ and $E$:*

- $□^-\, p \,◑ \xrightarrow[E]{}{}^* □^-\, p \,○_\varnothing$ ;
- *For $■^-$ (that is, when $\mathit{Sel}^+$ holds), we merely have $■^-\, p \,◑ \xrightarrow[E]{}{}^* ■^-\, p' \,○_K$, as the Go value may be required to eliminate some completion codes (e.g., 1 for pause). If we additionally have $\mathit{Go}^-$, then we recover $■^-\, p \,◑ \xrightarrow[E]{}{}^* ■^-\, p \,○_\varnothing$.*

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

- all cases contain the input invariant *input_invariant*($sel, \blacksquare$) and
- all compound microstates contain recursive calls for sub-microstates.
  This recursive call is explicitly shown for $p\backslash s$ because of the change in the environment $E$.

$$
\begin{array}{lcl}
0 & \mapsto & sel = \text{false} \quad \wedge \quad \text{WIRE}(0, \blacksquare, \blacklozenge) \\
1 & \mapsto & \circ_{\{0,1\}} \leq \blacklozenge \quad \wedge \quad \text{PAUSE}(sel, \blacksquare, \blacklozenge) \\
k & \mapsto & sel = \text{false} \quad \wedge \quad \text{WIRE}(k+2, \blacksquare, \blacklozenge) \\
!s & \mapsto & s \in E \quad \wedge \quad sel = \text{false} \quad \wedge \quad \text{WIRE}(0, \blacksquare, \blacklozenge) \\
@s & \mapsto & s \in E \quad \wedge \quad \circ_{\{0,1\}} \leq \blacklozenge \\
 & & \wedge \quad (\blacksquare = \blacksquare^- \implies \blacklozenge \leq \circ_\varnothing) \quad \wedge \quad (\blacksquare \neq \blacksquare^- \wedge s^\perp \in E \implies \blacklozenge = \circ_{\{0,1\}}) \\
 & & \wedge \quad (\blacksquare = \blacksquare^+ \wedge s^+ \in E \implies \blacklozenge \leq \bullet_0) \quad \wedge \quad (\blacksquare \neq \blacksquare^+ \wedge \blacksquare \neq \blacksquare^- \wedge s^+ \in E \implies \blacklozenge \leq \circ_{\{0\}}) \\
 & & \wedge \quad (\blacksquare = \blacksquare^+ \wedge s^- \in E \implies \blacklozenge \leq \bullet_1) \quad \wedge \quad (\blacksquare \neq \blacksquare^+ \wedge \blacksquare \neq \blacksquare^- \wedge s^- \in E \implies \blacklozenge \leq \circ_{\{1\}}) \\
\{p\} & \mapsto & \text{in}(p) \leq \blacksquare \quad \wedge \quad sel = Sel(p) \quad \wedge \quad \downarrow\text{out}(\text{from\_stmt}(\mathcal{B}(p))) \leq \blacklozenge \leq \downarrow\text{out}(p) \\
\uparrow p & \mapsto & \text{in}(p) \leq \blacksquare \quad \wedge \quad sel = Sel(p) \quad \wedge \quad \uparrow\text{out}(\text{from\_stmt}(\mathcal{B}(p))) \leq \blacklozenge \leq \uparrow\text{out}(p) \\
s \supset p & \mapsto & s \in E \quad \wedge \quad sel = \text{false} \quad \wedge \quad Go(p) \leq Go(\blacksquare) \quad \wedge \quad Res(p) \leq Res(\blacksquare)\ \&\ Sel(p)\ \&\ \neg E(s) \\
 & & \wedge \quad \{1\} \cup \text{out}(\text{from\_stmt}(\mathcal{B}(p))) \leq \blacklozenge \leq \text{SuspNow}(Res(\blacksquare)\ \&\ Sel(p)\ \&\ E(s), \text{out}(p)) \\
s\,?\,p\,,\,q & \mapsto & s^b \in E \quad \wedge \quad sel = Sel(p) \mid\mid Sel(q) \quad \wedge \quad Sel(p)\ \&\ Sel(q) = \text{false} \\
 & & \wedge \quad Go(p) \leq Go(\blacksquare)\ \&\ b \quad \wedge \quad Go(q) \leq Go(\blacksquare)\ \&\ \neg b \\
 & & \wedge \quad Res(p) \leq Res(\blacksquare) \quad \wedge \quad Res(q) \leq Res(\blacksquare) \\
 & & \wedge \quad \text{out}(\text{from\_stmt}(\mathcal{B}(p))) \cup \text{out}(\text{from\_stmt}(\mathcal{B}(p))) \leq \blacklozenge \leq \text{out}(p) \cup \text{out}(q) \\
p\,;\,q & \mapsto & sel = Sel(p) \mid\mid Sel(q) \quad \wedge \quad Sel(p)\ \&\ Sel(q) = \text{false} \\
 & & \wedge \quad \text{in}(p) \leq \blacksquare \quad \wedge \quad Go(q) \leq (\text{out}(p))[0] \quad \wedge \quad Res(q) \leq Res(\blacksquare) \\
 & & \wedge \quad (\text{out}(\text{from\_stmt}(\mathcal{B}(p))) \setminus 0) \cup \text{out}(\text{from\_stmt}(\mathcal{B}(q))) \leq \blacklozenge \leq (\text{out}(p) \setminus 0) \cup \text{out}(q) \\
p \mid q & \mapsto & sel = Sel(p) \mid\mid Sel(q) \quad \wedge \quad \text{in}(p) \leq \blacksquare \quad \wedge \quad \text{in}(q) \leq \blacksquare \\
 & & \wedge \quad \text{out}(\text{from\_stmt}(\mathcal{B}(p))) \cup \text{out}(\text{from\_stmt}(\mathcal{B}(p))) \leq \blacklozenge \leq \text{synch}_{Sel(p),Sel(q)}(\text{out}(p), \text{out}(q)) \\
p\backslash s & \mapsto & sel = Sel(p) \quad \wedge \quad \text{in}(p) \leq \blacksquare \quad \wedge \quad \text{out}(\text{from\_stmt}(\mathcal{B}(p))) \leq \blacklozenge \leq \text{out}(p) \\
 & & \text{valid\_coloring}(E * s^{\text{to\_event}(p, E*s^\perp)}, p)
\end{array}
$$

where

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

$$
\text{WIRE}(k, \blacksquare, \blacklozenge) = \begin{cases} \blacklozenge = \circ_{\{k\}} & \text{when} \quad Go(\blacksquare) = \bot \\ \blacklozenge \leq \circ_\varnothing & \text{when} \quad Go(\blacksquare) = - \\ \blacklozenge \leq \bullet_k & \text{when} \quad Go(\blacksquare) = + \end{cases}
$$

- PAUSE($Sel, \blacksquare, \blacklozenge$) expresses the input/output relation for the 1 (`pause`) statement:

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

**Figure 13** The 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 $\mathrm{Sel}^-$), 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 $\equiv_{Res}$ the equality of microstates up to Res wires, we can write this property formally as:*

$$\begin{aligned}\forall p, q, p', \mathrm{valid\_coloring}(E, p) \implies \mathrm{valid\_coloring}(E, q) \implies \\ Sel(p) = - \implies p \equiv_{Res} q \implies \\ p \underset{E}{\longrightarrow} p' \implies p' \equiv_{Res} q \vee \left(\exists q', q \underset{E}{\longrightarrow} q' \wedge p' \equiv_{Res} q'\right) .\end{aligned}$$

We do not need the hypothesis $Sel(q) = -$ because $p \equiv_{Res} q$ entails $Sel(p) = Sel(q)$. 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 $p' \equiv_{Res} p \equiv_{Res} q$ (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 $Sel = -$.

## 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, $s \in Must_s(p, E)$ or $s \notin Can_s^+(p, E)$, into the corresponding condition in the microstep semantics, that is: there exists a microstep sequence $p \underset{E}{\longrightarrow}^* p'$ 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 $\bot$ 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* 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:*
  $\forall s, s \in \mathit{Must}_s(p,E) \implies \exists p', [\square^+]\,\mathit{from\_stmt}(p) \underset{E}{\longrightarrow}{}^{*}\ p' \wedge s^+ \in \mathit{to\_event}(p',E)$ ;
- *Any signal that cannot be emitted is indeed not emitted after enough microsteps:*
  $\forall s, s \notin \mathit{Can}^+_s(p,E) \implies \exists p', [\square^+]\,\mathit{from\_stmt}(p) \underset{E}{\longrightarrow}{}^{*}\ p' \wedge s^- \in \mathit{to\_event}(p',E)$ ;
- *Any signal that can never be emitted is indeed not emitted no matter what the input color is:*
  $\forall s, s \notin \mathit{Can}^\bot_s(p,E) \implies \forall \blacksquare, \exists p', [\blacksquare]\,\mathit{from\_stmt}(p) \underset{E}{\longrightarrow}{}^{*}\ p' \wedge s^- \in \mathit{to\_event}(p',E)$ ;
- *If the completion code must be $k$, then it is indeed $k$ after enough microsteps:*
  $\forall k, k \in \mathit{Must}_k(p,E) \implies \exists p', [\square^+]\,\mathit{from\_stmt}(p) \underset{E}{\longrightarrow}{}^{*}\ p' \wedge \mathrm{out}(p') = \bullet_k$ ;
- *After enough microsteps, the possible completion codes are a subset of what they can be:*
  $\exists p', [\square^+]\,\mathit{from\_stmt}(p) \underset{E}{\longrightarrow}{}^{*}\ p' \wedge \mathit{out2C}(\mathrm{out}(p')) \subseteq \mathit{Can}^+_k(p,E)$ ;
- *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:*
  $\forall \blacksquare, \exists p', [\blacksquare]\,\mathit{from\_stmt}(p) \underset{E}{\longrightarrow}{}^{*}\ p' \wedge \mathit{out2C}(\mathrm{out}(p')) \subseteq \mathit{Can}^\bot_k(p,E)$.

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

*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 $\overline{p'}$, if $p \xhookrightarrow[E]{E',k}_s \overline{p'}$ then there exists a total microstate $p''$ such that*

$$[\square^+](\mathit{from\_stmt}(p)) \underset{E}{\longrightarrow}{}^{*}\ p'', \quad \mathit{to\_term}(p'',E) = \overline{p'}, \quad \mathit{out}(p'') = k \quad \textit{and} \quad \mathit{to\_event}(p'',E) = E'.$$

*For all $\widehat{p}$, $E$, $E'$, $k$, and $\overline{p'}$, if $\widehat{p} \xhookrightarrow[E]{E',k}_r \overline{p'}$ then there exists a total microstate $p''$ such that*

$$[\blacksquare^+](\mathit{from\_state}(\widehat{p})) \underset{E}{\longrightarrow}{}^{*}\ p'', \quad \mathit{to\_term}(p'',E) = \overline{p'}, \quad \mathit{out}(p'') = k \quad \textit{and} \quad \mathit{to\_event}(p'',E) = E'.$$

**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 $p\backslash s$ 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, $(s\,?\,0\,,\,!s)\backslash s$), 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 $\texttt{P1} = s\,?\,!s\,,\,0$ 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 $\bot$. This is trivially true for the logical semantics, as there is no $\bot$ involved.

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

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:

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

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 $\widehat{1}$.

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

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

## 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.,* valid_dom$(p, E)$ *holds), then we have* $\mathit{CBS\_interp}(p, E) = \mathit{Some}(E', k, p') \iff p \xrightarrow[E]{E',k} p'$.

**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).

$$\forall p, E, p',\ \mathit{micro\_interp}(p, E) = \mathit{Some}\, p' \implies p \underset{E}{\longrightarrow} p'$$
$$\forall p, E, p',\ p \underset{E}{\longrightarrow} p' \implies \mathit{micro\_interp}(p, E) \neq \mathit{None}$$

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:

$$\forall p, E, p \underset{E}{\longrightarrow}{}^{*} \texttt{micros\_interp}(p, E) \quad \wedge \forall p',\ \neg\Big(\texttt{micros\_interp}(p, E) \underset{E}{\longrightarrow} p'\Big)$$

This statement means that any microstate $p$ can execute up to `micro_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, Lélio 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 Sticksel, 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 docteur-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, Istambul*, 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 Boulier, 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. Scharts: 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.

# Limited-Preemption EDF Scheduling for Multi-Phase Secure Tasks

**Benjamin Standaert**
Washington University, St. Louis, MO, United States

**Fatima Raadia**
Wayne State University, Detroit, MI, USA

**Marion Sudvarg**[1]
Washington University, St. Louis, MO, United States

**Sanjoy Baruah**
Washington University, St. Louis, MO, United States

**Thidapat Chantem**
Virginia Tech, Blacksburg, VA, USA

**Nathan Fisher**
Wayne State University, Detroit, MI, USA

**Christopher Gill**
Washington University, St. Louis, MO, United States

## Abstract

Safety-critical embedded systems such as autonomous vehicles typically have only very limited computational capabilities on board that must be carefully managed to provide required enhanced functionalities. As these systems become more complex and inter-connected, some parts may need to be secured to prevent unauthorized access, or isolated to ensure correctness.

We propose the *multi-phase secure* (MPS) task model as a natural extension of the widely used sporadic task model for modeling both the timing and the security (and isolation) requirements for such systems. Under MPS, task phases reflect execution using different security mechanisms which each have associated execution time costs for startup and teardown. We develop corresponding limited-preemption EDF scheduling algorithms and associated pseudo-polynomial schedulability tests for constrained-deadline MPS tasks. In doing so, we provide a correction to a long-standing schedulability condition for EDF under limited-preemption. Evaluation shows that the proposed tests are efficient to compute for bounded utilizations. We empirically demonstrate that the MPS model successfully schedules more task sets compared to non-preemptive approaches.

**2012 ACM Subject Classification** Computer systems organization → Real-time systems

**Keywords and phrases** real-time systems, limited-preemption scheduling, trusted execution environments

**Digital Object Identifier** 10.4230/LITES.10.1.3

**Funding** This research was supported in part by NSF Grants CPS-1932530, CNS-2141256, CNS-2229290, IIS-1724227, CNS-2038609, CCF-2118202, CNS-2211641, and CPS-2038726.

**Acknowledgements** We, the authors, would like to thank the reviewers for their constructive remarks.

**Received** 2024-08-19 **Accepted** 2025-03-31 **Published** 2025-05-15

**Editor** Björn B. Brandenburg

[1] Corresponding author





# 1 Introduction

In today's interconnected world, the security of real-time systems has emerged as a primary concern, e.g., [41, 23, 26, 20, 37, 44, 31], given the widespread integration of electronic devices into various aspects of daily life. However, the implementation of security measures often introduces additional resource requirements, such as increased computational overhead, or imposes specific constraints on application behaviors; for example, this could involve necessitating computation that requires isolation or cannot be preempted.

For example, control flow integrity (CFI) checks may be needed to ensure correct program execution. However, such checks, which require CPU time in addition to normal code execution, must be carried out at specific time points (e.g., after branching) and allowing for preemption may result in an arbitrary computation being performed but not detected. As another example, a task that is responsible for taking sensor readings may need to execute in isolation in order to ensure that another task cannot deduce when an event of interest occurs [31].

Since implementing security measures requires some of the same resources that the real-time tasks need to advance their execution, a co-design approach that explicitly considers security cost/requirements along with real-time requirements is potentially more effective at managing limited computational resources. For instance, *trusted execution environments* (TEEs) provide isolation of code and data in hardware at the expense of startup and teardown costs. A scheduling approach that does not consider this specific security-driven overhead may elect to switch between the secure world (i.e., executing in TEE) and the normal world (no TEE) indiscriminately. This may result in an excessive amount of overhead incurred by, e.g., hardware mode switching, saving and restoring the stack, and copying data, resulting in deadline misses. A security-cognizant scheduler, on the other hand, would make judicious decisions based on both security and real-time requirements, e.g., by bundling up multiple TEE executions and executing them one immediately after the other so as to have to pay for startup and teardown cost only once [32].

A recent ISORC paper [5] proposed and developed algorithms that are able to provide provable correctness of both the timing and some security properties. We believe that such a scheduling-based approach to achieving security in safety-critical systems is possible, and indeed, necessary in embedded systems that are particularly cost- and SWaP-constrained and hence need to be implemented in a resource-efficient manner. However, we consider it unlikely that a "one size fits all" solution exists; instead, security-cognizant scheduling must first explicitly identify the kinds of threats that are of concern by precisely defining a threat model, and design scheduling strategies that can be proved to be resistant to attacks under the identified threat model. We consider the research in [33] to be particularly noteworthy in this regard, in their explicit and methodical modeling of different threats in the context of the sporadic task model, and their analysis of vulnerabilities of current strategies (including security-agnostic fixed-priority scheduling [1] and the randomization-based schedule obfuscation approaches, e.g., the one in [44]) to such threats.

**Security-Cognizant Scheduling.** We believe the methodology formalized and used in [33] holds great promise as a means of integrating security and timing correctness concerns within a common framework. This methodology was articulated in [5] as follows: security for safety-critical real-time embedded systems can be achieved by *(i)* explicitly representing specific security considerations within the same formal frameworks that are currently used for specifying real-time workloads, thereby extending notions of correctness to incorporate both the timing and the security aspects; and *(ii)* extending previously-developed techniques for achieving provable timing correctness to these models, thus assuring that both timing and security properties are correct.

**This Work.** This paper **extends our prior work** in [3], which applied the methodology articulated in [5] to the following problem in system design for real-time + security. We consider computer platforms upon which *multiple different security mechanisms* (such as TEEs, encryption/decryption co-processors, FPGA-implemented secure computations, etc.) co-exist. Depending upon their security requirements, different pieces/parts of the (real-time) code may need to use different security mechanisms at different times. We therefore assume that the code is broken up into *phases*, with different consecutive phases needing to use different sets of security mechanisms – the security mechanisms used by each phase are specified for the phase. We assume that there is a startup/teardown overhead cost (for data communication, initialization, etc.) expressed as an execution duration, associated with switching between different security mechanisms. In other words, *there is a time overhead associated with switching between the execution of different phases.*

**Contributions.** As in our prior work that this paper extends [3], we formalize the workload model discussed above as the *Multi-Phase Secure* (**MPS**) task model, with multiple independent recurrent processes of this kind that are to execute upon a single shared preemptive processor. We start out in Section 4 assuming that each recurrent process is represented using the widely-used 3-parameter sporadic task model [7]. For this model, we represent the problem of ensuring timeliness plus security as a schedulability analysis problem, which we then solve by adapting results obtained in prior work (e.g., [4, 9, 13, 34, 14]) on limited-preemption scheduling. Later in Section 5, we propose a generalization that models conditional execution within each recurrent process. Its original treatment in [3] makes a simplifying assumption that was later addressed in another extension; we therefore present the conditional model and the simplifying assumption in this paper, but leave the analysis for conditional execution to the more accurate treatment in [38].

We emphasize that although the designs of these models are motivated by security considerations – they arose out of some security-related projects that we are currently working on – we are proposing a *scheduling model* and associated algorithms, not a complete solution to a particular security problem. That is, although our model draws inspiration from security concerns, it *(i)* does not claim a perfect match to all security requirements; and *(ii)* it should have applicability beyond the security domain – indeed, we suggest that the results presented in this paper be looked upon as a generalization of the rich body of real-time scheduling theory literature on limited-preemption scheduling.

**Extensions to the Prior Work.** This paper **extends, corrects, and clarifies** our prior results in [3], making the following new contributions. Section 3.2 presents a correction to the condition in [4, 9] for EDF schedulability of limited-preemption tasks. Section 4 leverages the corrected condition to introduce pseudo-polynomial schedulability analysis for sets of MPS tasks, improving the execution time of the associated algorithms originally proposed in [3], especially for tasks with implicit deadlines. With these improvements, we are able to evaluate larger sets of tasks with more realistic ranges of periods. We also address inconsistencies in [3] between the theoretical model and its implementation by using a continuous-time representation of the algorithms. These are reflected in the new results in Section 6, which more clearly demonstrate the advantages of our approach over non-preemptive schedulers.

**Organization.** The remainder of this manuscript is organized as follows. After briefly discussing some related scheduling-theory results in Section 2, Section 3.2 presents the correction to the limited-preemption EDF schedulability condition of [4, 9]. We then motivate and formally define the MPS sporadic tasks model in Section 4, and provide both pre-runtime analysis and a run-time scheduling algorithm for MPS sporadic task systems upon preemptive uniprocessor platforms. In Section 5 we further generalize the workload model to be able to represent conditional execution.

We have performed schedulability experiments to evaluate both the effectiveness of our algorithms and the performance improvements over their original implementations in [3]. We report the results in Section 6. We conclude in Section 8 by pointing out some directions in which we intend to extend this work, and by placing our results within a larger context on the timing- and security-aware synthesis of safety-critical systems.

# 2 Some Real-Time Scheduling Background

## 2.1 The Sporadic Task Model [7]

In this model, recurrent processes are represented as sporadic tasks $\tau_i = (C_i, D_i, T_i)$. Each task has three defining characteristics: worst-case execution requirement (WCET) $C_i$, relative deadline $D_i$, and period (minimum inter-arrival duration) $T_i$. The sporadic task $\tau_i$ generates a series of jobs, with inter-arrival times of at least $T_i$. Each job must be completed within a scheduling window, which starts at the job's release time and ends $D_i$ time units later, and the job's execution time is limited to $C_i$ units. A sporadic task system $\Gamma$ is made up of multiple independent sporadic tasks. We assume without loss of generality that tasks are indexed in non-decreasing relative deadline order (i.e., if $i < j$ then $D_i \leq D_j$).

**Processor Demand Analysis (PDA).** A sporadic task system can be scheduled optimally by the Earliest Deadline First (EDF) [24] scheduling algorithm, given a preemptive uniprocessor. To determine whether a specific task system can be correctly scheduled by EDF, Processor Demand Analysis (PDA) [8] can be utilized. PDA is a necessary and sufficient algorithm that is also optimal. The key idea of PDA is built upon the *demand bound function* (DBF). Given an interval length of $L$ such that $L \geq 0$, the DBF for a sporadic task $\tau_i$ can be represented by $\text{DBF}_i(L)$: the maximum possible aggregate execution time required by jobs of task $\tau_i$ such that they arrive in $L$ and have deadlines before $L$. The following equation was derived in [7] to compute its value:

$$\text{DBF}_i(L) = \max\left(\left\lfloor \frac{L - D_i}{T_i} \right\rfloor + 1, 0\right) \times C_i \tag{1}$$

For a task system $\tau$ to be correctly scheduled by EDF, the following was derived in [7] as a necessary and sufficient condition for all $L \geq 0$:

$$\left[\left(\sum_{\tau_i \in \Gamma} \text{DBF}_i(L)\right) \leq L\right] \tag{2}$$

**The Testing Set.** A naïve application of PDA requires testing the validity of Equation 2 for all intervals. However, a more efficient approach, outlined in [7], involves checking only values of $L$ that follow the pattern $L \equiv (k \times T_i + D_i)$ for some non-negative integer $k$ and some $\tau_i \in \Gamma$.

Furthermore, it suffices to test such values that are less than the least common multiple of all the $T_i$ parameters. The collection of all such values of $t$ for which it is necessary to verify that Condition 2 holds true in order to confirm EDF-schedulability is referred to as the *testing set* for the sporadic task system $\Gamma$, often denoted as $\mathcal{T}(\Gamma)$.

It is worth noting [7] that, in general, the size $|\mathcal{T}(\Gamma)|$ of the testing set $\mathcal{T}(\Gamma)$ can be exponential in the representation of $\tau$. However, it has been proven [8, Theorem 3.1] that for *bounded-utilization* task systems – i.e., systems $\Gamma$ that fulfill the additional requirement that $\sum_{\tau_i \in \Gamma} U_i \leq c$ for some fixed constant $c$ strictly less than 1 – it is sufficient to check a smaller testing set with pseudo-polynomial cardinality relative to the representation of $\Gamma$, consisting of all values of the form $L \equiv (k \times T_i + D_i)$ not exceeding

$$\min\left(P, \max\left(D_{\max}, \frac{1}{1-U} \cdot \sum_{i=1}^{n} U_i \cdot (T_i - D_i)\right)\right) \tag{3}$$

where $P$ is the least common multiple of all $T_i$, and $D_{\max}$ is the maximum of all $D_i$ parameters. We note that for bounded-utilization *implicit-deadline* tasks – those for which $T_i = D_i$ for every task $\tau_i$ – this bound reduces to $D_{\max}$. In this extension, we apply this smaller testing set to our scheduling algorithms for MPS tasks.

Since we can check Condition 2 in linear ($\Theta(n)$) time for any given value of $t$, these observations imply that we can perform an exponential-time EDF-schedulability test for general task systems and a pseudo-polynomial-time one for bounded-utilization systems.

Unfortunately, the general problem is NP-hard in the strong sense [15, 16, 18], and the bounded-utilization variant is NP-hard in the ordinary sense [17]. Therefore, it is unlikely that we will discover more efficient schedulability tests.

## 2.2 Limited-Preemption Scheduling

The limited-preemption sporadic task model, as introduced by Baruah et al. [4], adds to the task specification $\tau_i = (C_i, D_i, T_i, \beta_i)$ a *chunk-size* parameter $\beta_i$ in addition to the regular parameters $C_i$, $D_i$, and $T_i$. This parameter $\beta_i$ indicates that each job of task $\tau_i$ may need to execute non-preemptively for up to $\beta_i$ time units.

To schedule tasks in the limited-preemption sporadic task model, the limited-preemption EDF scheduling algorithm was proposed [4, 9]. Like its preemptive counterpart, the limited-preemption EDF algorithm prioritizes jobs based on their (absolute) deadlines. If a job of task $\tau_i$ with remaining execution time $e$ is executing and a new job with an earlier deadline arrives, then $\tau_i$'s job may execute for an additional $\min(e, \beta_i)$ time units before incurring a preemption.

Baruah and Bertogna [4, 9] showed that a task system is *not* schedulable under the limited-preemption EDF model if and only if either of the following conditions are true:

$$\exists L : L \geq 0 : \sum_{\tau_i \in \Gamma} \text{DBF}_i(L) > L \tag{4}$$

or

$$\exists \tau_i : \exists L : 0 \leq L < D_i : \beta_i + \sum_{\tau_j \in \Gamma, j \neq i} \text{DBF}_j(L) > L \tag{5}$$

Noting that $\text{DBF}_i(L) = 0$ when $L < D_i$, we combine and invert the two conditions, giving a necessary and sufficient condition for successfully scheduling a limited-preemption sporadic task system $\Gamma$ upon a single preemptive processor using the limited-preemption EDF algorithm:

$$\forall L, \quad \left[\left(\sum_{\tau_i \in \Gamma} \text{DBF}_i(L)\right) + \overbrace{\max_{\{\tau_i | D_i > L\}}\left\{\beta_i\right\}}^{\text{(blocking due to limited preemption)}} \leq L\right] \tag{6}$$

Unlike the exact test for preemptive uniprocessor EDF-schedulability (Equation 2), Equation 6 contains an additional term on the left-hand side of the inequality that accounts for blocking due to later-deadline (and hence lower-priority) jobs. Specifically, the $\max_{\tau_i | D_i > L} \beta_i$ term is a *blocking term* that captures the potential delay caused by lower-priority jobs that were already executing at the start of the interval, for a duration of up to their chunk size. Since we assume that all tasks have non-negative execution time, this blocking term is always non-negative.

In this extension to the prior work in [3], we use a continuous-time representation of the blocking term in Equation 6. The prior work used a discrete-time representation, $\max_{\tau_i|D_i>L} \beta_i - 1$, which requires both task periods *and execution times* to be represented as integers. This introduces several challenges. First, it is more difficult to reason about the effect of inserting preemption points; blocking times (chunk sizes) must also be represented as integers, but the number of "chunks" might not evenly divide the execution time. Second, there is a tradeoff when choosing the precision at which to represent execution time units. If the unit of time is coarse, then the execution time of each phase and its corresponding startup/teardown time must be rounded up. If the unit of time is very short – such as a single processor tick – to achieve higher precision, then the testing set grows rapidly as the representations of the task periods become larger. By using continuous time, this extension removes these limitations, and modifies the expression to allow execution times to take any non-negative real value.

Based on this observation, the Processor Demand Analysis (PDA) algorithm has been extended to apply to limited-preemption systems as well [4]. The extension is straightforward: Equation 6 replaces Equation 2 in the algorithm, and the algorithm proceeds as usual.

However, we note that Expression 6 cannot hold true for values of $L$ of the form

$$0 \leq L < \min\left(D_{\min}, \max_{\{\tau_i\}}\{\beta_i\}\right) \tag{7}$$

where $D_{\min} = \min_i\{D_i\}$, suggesting **incorrectly** that *a set of tasks is not EDF schedulable if any task has non-zero blocking time.* In the next section of this extension, we present a corrected condition that addresses this issue, which we then apply to scheduling of MPS tasks in Section 4.

## 3 The Corrected Limited-Preemption EDF Schedulability Condition

In this section, we present a correction to the condition of Baruah and Bertogna [4, 9] for EDF schedulability of limited-preemption tasks.

### 3.1 The Problem

As discussed in the prior section, in [4, 9], Baruah and Bertogna claimed as a necessary and sufficient condition for scheduling a limited-preemption sporadic task system upon a single preemptive processor using EDF that

$$\forall L, \quad \left[\left(\sum_{\tau_i \in \Gamma} \text{DBF}_i(L)\right) + \overbrace{\max_{\{\tau_i|D_i>L\}}\left\{\beta_i\right\}}^{\text{(blocking due to limited preemption)}} \leq L\right] \tag{8}$$

where $\beta_i$ is the blocking due to limited preemption induced by task $\tau_i$.

From the definition of the demand bound function $\text{DBF}_i(L)$ in Equation 1, we observe that $\text{DBF}_i(L) = 0$ for $L < D_i$. Then for $L < D_{\min}$ (i.e., $L < D_i$ for all tasks $\tau_i$), the above condition requires $L \geq \max_{\tau_i|D_i>L}\{\beta_i\}$; as we are already considering the case that $L < D_{min}$, this can be simplified to $L \geq \max_{\tau_i}\{\beta_i\}$.

This implies that for values of $L$ such that $L < D_{\min}$, the above condition cannot hold true if $L$ does not also exceed $\beta_i$ for all tasks $\tau_i$. More simply, the condition cannot hold true for values of $L$ of the form shown in Expression 7. Because processor demand analysis requires that the condition hold true for *all* values of $L \geq 0$, this would deem **any set of limited-preemption tasks with non-zero blocking time to be unschedulable by EDF.**

## 3.2 The Correction

As this is obviously not true – i.e., there **are** limited-preemption task sets that are schedulable by EDF, we now set out to correct the above condition. We do so by pointing out a subtlety to the **proof** in [4] of the condition in Expression 6.

That proof constructs the blocking time condition by defining a minimal unschedulable set of jobs for which $t_a$ represents the earliest arrival time of those jobs and $t_f$ is the time at which the first deadline miss occurs. Additionally, for each job $\tau_i$, it defines $q_i$ as representing an upper bound on the time for which $\tau_i$ can execute non-preemptively. In this system, there is exactly one job with a deadline after $t_f$; we let $\tau_j$ be the task that generates this job. If $t_1$ is the time at which that job begins to execute non-preemptively and $t_2$ is when it stops executing, then [4, Equation 4] states that

$$\sum_{i=1, i\neq j}^{n} \text{DBF}(\tau_i, t_f - t_1) > t_f - t_2$$

As $q_j$ is a bound on the non-preemptive execution time, the proof in [4] then claims that $t_2 - t_1 \leq q_j$. We observe that since $t_2 \leq t_f$, **it also follows that** $t_2 - t_1 \leq t_f - t_1$. We can therefore combine these inequalities to make the statement that $t_2 - t_1 \leq \min(q_j, t_f - t_1)$. In light of this, we modify the rest of the proof in [4]. Now, it follows that

$$\sum_{i=1, i\neq j}^{n} \text{DBF}(\tau_i, t_f - t_1) + \min(q_j, t_f - t_1) > t_f - t_2 + (t_2 - t_1)$$

We then replace the expression $t_f - t_1$ with a time $L$. Since $t_f < D_j$, it follows that $L < D_j$; the DBF of $\tau_j$ will therefore be zero at $L$ and we can rewrite the condition as:

$$\sum_{i=1}^{n} \text{DBF}(\tau_i, L) + \min(q_j, L) > L$$

This condition checks whether some task $\tau_j$ causes excessive blocking at times $L < D_j$; to determine if the system is schedulable, we can therefore test whether the following condition holds for any task $\tau_i$ at any time $L \geq 0$, considering blocking times from just those tasks for which $L < D_i$:

$$\left(\sum_{\tau_i \in \Gamma} \text{DBF}_i(L)\right) + \min\left(L, \max_{\{\tau_i | D_i > L\}} \{\beta_i\}\right) \leq L \tag{9}$$

Then when $L < D_{\min}$, the DBF for each task is 0, so the condition will always be satisfied. Furthermore, when $L \geq D_{\min}$, $\sum_i \text{DBF}(L_i) > 0$, and so the condition cannot be satisfied when $\min(L, \max_i \beta_i) \geq L$. We can therefore begin the testing set of our implementation at $D_{\min}$, and only check the blocking time when determining whether the condition is violated.

# 4 Systems of Multi-Phase Secure (MPS) Sporadic Tasks

In this section we extend the sporadic task model to consider the setting where the workload of each task comprises an ordered sequence of different *phases*, with each phase required to use a different security mechanism.[2] Therefore, switching between phases or between jobs incurs

[2] Note that we consider a unique *combination* of security mechanisms to be its own security mechanism. Thus, a portion of a task subject to multiple overlapping security mechanisms would execute in its own phase.

some *teardown/startup overhead*, which translates to an additional execution duration. We are given a system of multiple such independent tasks that are to be scheduled upon a shared preemptive processor. If an executing task is preempted within a phase, the assumption that the tasks in the system are independent of one another implies that we must conservatively assume that the preempting task may be executing using a different security mechanism; hence, the teardown/startup overhead may be incurred again. In some systems, the cost of a preemption may be less than the full startup/teardown cost of a phase; this can be accounted for by adding any additional cost to the execution time of the phase. When a phase of a task is selected for execution we assign it responsibility for taking care of both the startup that must happen at that point in time, and the subsequent teardown that occurs when it either completes execution or is preempted by a higher-priority task. Hence a phase with execution duration $c$ that is preempted $k$ times is responsible for (and hence should have been budgeted for) a total execution duration of

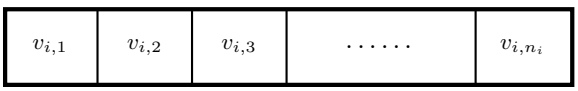
$$c + (k+1) \times (\text{startup cost} + \text{teardown cost}) \tag{10}$$

**A Motivating Example.** To motivate the MPS task model, consider the case of *trusted execution environments* (TEEs), a hardware feature that provides strong isolation of code and data. Although this isolation has significant security benefits, broadly-used TEE implementations such as OP-TEE [35] are sometimes limited to using only a small fraction of the available system memory [36] and code running in a TEE has limited access to common system APIs. In addition, placing more functionality inside a TEE increases the probability that an unidentified vulnerability compromises the isolated TEE environment; a number of works based on TEEs have cited minimization of the trusted computing base (TCB) as a key design concern [25, 29, 39]. A fine-grained minimization of the code placed in a TEE can lead to the existence of tasks that span multiple "worlds" – for example, a task may begin execution in the normal world (i.e. no TEE), switch to the secure world (executing in the TEE) to perform a sensitive cryptographic operation, and then return to the normal world to complete its work. In some cases, switching across worlds can also be used as a workaround for TEE memory limitations. For example, large neural networks may be executed securely by copying a single layer into the TEE, computing and storing an intermediate result, and then returning to the normal world to retrieve the next layer [2].

However, switching between worlds induces TEE startup and teardown costs, the impact of which varies depending on the choice of processor and TEE software implementation; one implementation based on an Arm Cortex-M development board saw costs on the order of microseconds [30], while an implementation using OP-TEE on a Raspberry Pi with a Cortex-A processor saw overheads ranging from hundreds of microseconds to tens of milliseconds [32]. A scheduling approach that does not consider these significant context-switching costs may preempt execution for higher-priority jobs indiscriminately, resulting in missed deadlines due to the overheads incurred by frequent startup/teardown.

## 4.1 Task Model

We have a task system $\Gamma$ comprising $N$ independent recurrent tasks $\tau_1, \tau_2, \ldots, \tau_N$, to be scheduled upon a single preemptive processor. The task $\tau_i$ is characterized by a period/inter-arrival separation parameter $T_i$ and a relative deadline $D_i \leq T_i$ . The body of the task – the work that must be executed each time the task is invoked – comprises $n_i$ *phases*, denoted $v_{i,1}, v_{i,2}, \ldots, v_{i,n_i}$, that must execute in sequence upon each job release of the task:



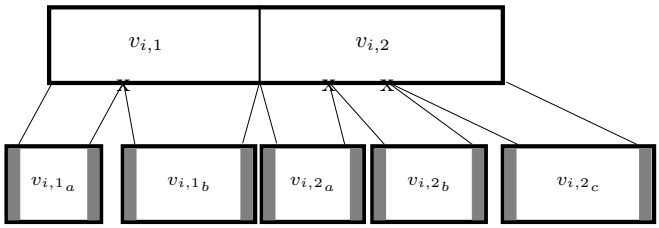
As previously discussed, we assume that successive phases are required to execute using different security mechanisms (i.e., $v_{i,j}$ and $v_{i,j+1}$ execute using different security mechanisms for all $j$, $1 \leq j < n_i$). Let $c(v_{i,j})$ denote the WCET of phase $v_{i,j}$, and $q(v_{i,j})$ denote the sum of the startup cost and the teardown cost associated with the security mechanism within which phase $v_{i,j}$ is to execute. The aggregate WCET of all phases of this task during its execution is thus given by the following expression

$$\sum_{j=1}^{n_i} \Big( c(v_{i,j}) + q(v_{i,j}) \Big)$$

However, suppose that during some execution of task $\tau_i$ the $j$'th phase is preempted $k_j$ times for each $j$, $1 \leq j \leq n_i$; as discussed above (Equation 10), the cumulative WCET of all phases of this task during this execution is then given by the expression

$$\sum_{j=1}^{n_i} \Big( c(v_{i,j}) + (k_j + 1) \times q(v_{i,j}) \Big)$$

The figure below depicts a 2-phase job, denoted by $v_{i,1}$ and $v_{i,2}$, which is preempted once in the first phase and twice in the second phase. The shaded region denotes the startup and teardown costs for each phase of the job.




## 4.2 Overview of Approach

Given a task system $\Gamma$ comprising multiple independent MPS tasks to be scheduled upon a single preemptive processor, we will first execute a schedulability analysis algorithm to determine whether this system is schedulable, i.e., whether we can guarantee to schedule it to always meet all deadlines, despite the costs incurred by startup/teardown. This schedulability analysis algorithm essentially constructs a limited-preemption task $\widehat{\tau_i}$ corresponding to each task $\tau_i$, and determines whether the resulting limited-preemption task system can be scheduled by the limited-preemption EDF scheduling algorithm [4, 9] to always meet all deadlines. If so, then during run-time the original task system is scheduled using the limited-preemption EDF scheduling algorithm, with chunk-sizes as determined for the corresponding constructed limited-preemption tasks. We point out that if a chunk-size $\beta_i$ is determined for the limited-preemption task $\widehat{\tau_i}$, then the $j$'th phase of $\tau_i$'s jobs will execute in no more than $\lceil c(v_{i,j})/(\beta_i - q(v_{i,j})) \rceil$ contiguous time-intervals (i.e., it would experience at most $(\lceil c(v_{i,j})/(\beta_i - q(v_{i,j})) \rceil - 1)$ preemptions); equivalently, the cumulative WCET of all the phases of each of task $\tau_i$'s jobs will be no more than

$$\sum_{j=1}^{n_i} \left( c(v_{i,j}) + \left\lceil \frac{c(v_{i,j})}{\beta_i - q(v_{i,j})} \right\rceil \times q(v_{i,j}) \right)$$

**Assumption of Fixed-Preemption Points.** We make the conservative assumption that once the chunk-size is determined for each task then the preemption points in the code are statically determined prior to run-time. That is, once the chunk-size $\beta_i$ is determined for a task $\tau_i$, a

preemption is statically inserted into the task's code after the code has executed non-preemptively for no more than $\beta_i$ time units; this is referred to as the *fixed-preemption point model* [42]. Once $\tau_i$'s program reaches this statically-placed preemption point, the security mechanism for the task's current phase must

1. complete a teardown (e.g., a flush of the cache, or ending a TEE session) to ensure task execution integrity during preemption;
2. invoke the operating system's scheduler to see if there are any high-priority tasks awaiting execution; and
3. upon resuming execution as the highest-priority task the security mechanism must perform a startup (e.g., starting a new TEE session from the task's last executed instruction).

Since the preemption points are statically inserted into the code, we must perform the teardown/startup for a phase each time a preemption point is encountered (even if there is no other task active in the system at that time). While clearly this approach suffers from potentially performing unnecessary preemptions, it is often used in safety-critical settings due the precise predictability that fixed-preemption points provide. In future work, we will explore the floating preemption point model and other models that would permit the system to avoid unnecessary preemptions and teardowns/startups.

## 4.3 The Schedulability Test

We now describe our schedulability test. As discussed above, our approach is to construct for each task $\tau_i$ a corresponding limited-preemption task $\widehat{\tau}_i$. This limited-preemption task is assigned the same relative deadline parameter value (i.e., $D_i$) and the same period parameter value (i.e., $T_i$) as $\tau_i$; its WCET $\widehat{C_i}$ and its chunk-size parameter $\beta_i$ are computed as described below, and in pseudo-code form in Algorithm 1.

We introduce integer variables $\text{cnt}(v_{i,j})$ for each $i$, $1 \leq i \leq n$, and for each $j$, $1 \leq j \leq n_i$, to denote the maximum number of contiguous time-intervals in which the $j$'th phase of $\tau_i$ may need to execute. Then $\widehat{C_i}$, the WCET of each job of task $\widehat{\tau}_i$, can be written as

$$\widehat{C_i} \stackrel{\text{def}}{=} \sum_{j=1}^{n_i} \Big( c(v_{i,j}) + \text{cnt}(v_{i,j}) \times q(v_{i,j}) \Big) \tag{11}$$

where the second term within the summation represents the maximum preemption overhead (the startup cost plus the teardown cost) that is incurred by the $j$'th phase of task $\tau_i$.

It remains to specify the values we will assign to the $\text{cnt}(v_{i,j})$ variables. We will start out assuming that each phase of each task $\tau_i$ executes non-preemptively – i.e., in one contiguous time-interval. We do this by initially assigning each $\text{cnt}(v_{i,j})$ the value 1; we will describe below how the $\text{cnt}(v_{i,j})$ values are updated if enforcing such non-preemptive execution may cause deadlines to be missed. With each $\text{cnt}(v_{i,j})$ assigned the value 1, it is evident that the largest duration for which task $\tau_i$ will execute non-preemptively is equal to $\max_{j=1}^{n_i} \{c(v_{i,j}) + q(v_{i,j})\}$; we initialize the chunk-size parameters – the $\beta_i$ values – accordingly: for each task $\tau_i$,

$$\beta_i \leftarrow \max_{j=1}^{n_i} \{c(v_{i,j}) + q(v_{i,j})\} \tag{12}$$

In this manner, we have instantiated the parameters for one limited-preemption task $\widehat{\tau}_i$ corresponding to each task $\tau \in \Gamma$. We must now check whether the limited-preemption task system $\widehat{\Gamma} = \{\widehat{\tau}_1, \widehat{\tau}_2, \ldots, \widehat{\tau_N}\}$ so obtained is schedulable using the limited-preemption EDF scheduling algorithm [4, 9]. We do so by checking whether Equation 9 holds for values of $t$ in the testing set $\mathcal{T}(\widehat{\Gamma})$, considered in *increasing order*; we motivate this ordering below in (5). First, we split

**Algorithm 1** The Preprocessing Algorithm for Systems of MPS Sporadic Tasks (see Section 4).

**Input:** $(\Gamma)$

**for** *each task* $\tau_i \in \Gamma$ **do** `//Initially, assume that no preemption is needed`
  **for** $j \leftarrow 1$ **to** $n_i$ **do**
    $\text{cnt}(v_{i,j}) \leftarrow 1$
  $\beta_i \leftarrow \max_{1 \le j \le n_i} (c(v_{i,j}) + q(v_{i,j}))$
`//The testing set` $\mathcal{T}(\Gamma)_1$ `is all` $t \equiv D_i + k \cdot T_i$, $t \le D_{\max}$ `for some task` $\tau_i$ `and some` $k \in \mathbb{N}$.
**for** $t_d$ *iterating in increasing order over* $\mathcal{T}(\Gamma)_1$ **do**
  Compute $\Delta(t_d)$ as per Eqn 13 `//This represents the` *slack* `in the schedule at` $t_d$
  **if** $\Delta(t_d) < 0$ **then** `//Check whether previously-assigned chunk sizes causes deadline miss`
    **return** THE SYSTEM IS NOT SCHEDULABLE
  **for** *each* $\tau_i$ *for which* $D_i > t_d$ **do**
    **if** $(\beta_i > \Delta(t_d))$ **then** `//Must reduce the value of` $\beta_i$
      $\beta_i \leftarrow \Delta(t_d)$
      **for** $j \leftarrow 1$ **to** $n_i$ **do** `//For each phase of` $\tau_i$, `ensure that it doesn't block too much`
        **if** $\beta_i > q(v_{i,j})$ **then** `//There is sufficient time in chunk to do task execution`
          $\text{cnt}(v_{i,j}) \leftarrow \min_{\kappa \in \mathbb{N}} \text{such that} \frac{c(v_{i,j})}{\kappa} + q(v_{i,j}) \le \beta_i$ `//Break` $c(v_{i_j})$ `into small enough pieces`
        **else**
          **return** THE SYSTEM IS NOT SCHEDULABLE
**if** *system utilization* $> 1$ **then**
  **return** THE SYSTEM IS NOT SCHEDULABLE
**if** *all tasks have implicit deadlines* $D_i = T_i$ **then**
  **return** THE SYSTEM IS SCHEDULABLE
`//For systems of constrained-deadline tasks, the testing set` $\mathcal{T}(\Gamma)_2$ `is all` $t \equiv D_i + k \cdot T_i$, $D_{\max} < t$ `and not exceeding the bound defined in Expression 3 for some task` $\tau_i$ `and some` $k \in \mathbb{N}$.
**for** $t_d$ *iterating in increasing order over the testing set* $\mathcal{T}(\Gamma)_2$ **do**
  Compute $\Delta(t_d)$ as per Eqn 13
  **if** $\Delta(t_d) < 0$ **then**
    **return** THE SYSTEM IS NOT SCHEDULABLE
**return** THE SYSTEM IS SCHEDULABLE

$\mathcal{T}(\widehat{\Gamma})$ into two sets, $\mathcal{T}(\widehat{\Gamma})_1$ containing all values up to and including $D_{\max} \equiv \max_{\tau_i} D_i$, and $\mathcal{T}(\widehat{\Gamma})_2$ containing all values thereafter. Then, we initialize $t_d$ to denote the smallest value in $\mathcal{T}(\widehat{\Gamma})_1$, and perform the following steps.

1. If Equation 9 is satisfied for $t_d$, we set $t_d$ to be the next-smallest value in $\mathcal{T}(\widehat{\Gamma})_1$, and repeat this step.
2. If Equation 9 is violated for $t_d$ and the first term in the LHS of Equation 9 is $> t_d$, then we conclude that THE SYSTEM IS NOT SCHEDULABLE and return.

   Suppose, however, that a violation of Equation 9 occurs due to the blocking term in Equation 9. That is, the first term in the LHS of Equation 9 is $\le t_d$ when Equation 9 is instantiated with $t \leftarrow t_d$, but the sum of the first and second terms exceeds $t_d$. For this to happen, it must be the case that some $\widehat{\tau}_i$ with $D_i > t_d$ is blocking 'too much;' we must reduce the amount of blocking each such task can cause (i.e., reduce its $\beta_i$ parameter). Below, we describe how to do so.

3. Let $\Delta(t_d)$ denote the amount of blocking that can be tolerated at $t_d$ without causing a deadline miss:

$$\Delta(t_d) \stackrel{\text{def}}{=} t_d - \sum_{\widehat{\tau_k} \in \widehat{\Gamma}} \text{DBF}(\widehat{\tau_k}, t_d) \tag{13}$$

As discussed above, each $\widehat{\tau_i}$ with $D_i > t_d$ must ensure that its blocking term, $\beta_i$, is no greater than $\Delta(t_d)$. For each such task with $\beta_i$ currently greater than $\Delta(t_d)$, we may need to *in*crease $\text{cnt}(v_{i,j})$, the number of contiguous time-intervals in which its $j$'th phase may execute for each of its phases $v_{i,1}, v_{i,2}, \ldots, v_{i,n_i}$, in the following manner:

$$\text{cnt}(v_{i,j}) \leftarrow \min_{\kappa \in \mathbb{N}} \text{such that} \frac{c(v_{i,j})}{\kappa} + q(v_{i,j}) \leq \Delta(t_d) \tag{14}$$

That is, we reduce blocking in order to satisfy Equation 9 for $t_d$ by potentially increasing the number of preemptions (and thereby incurring additional teardown/startup overhead). In prior work [3], we assumed that tasks had integer execution times and hence used $\beta_i - 1$ as the blocking term; here, we use continuous time and therefore do not subtract a time segment.

4. Such additional overhead must be accounted for; this requires that $\widehat{C_i}$, the WCET parameter of the limited-preemption task $\widehat{\tau_i}$, must be updated (i.e., potentially increased) by recomputing it using Equation 11 (reproduced below):

$$\widehat{C_i} \leftarrow \sum_{j=1}^{n_i} \Big( c(v_{i,j}) + \text{cnt}(v_{i,j}) \times q(v_{i,j}) \Big)$$

(Notice that since some of the $\text{cnt}(v_{i,j})$ values may have increased, the value of $\widehat{C_i}$ may also increase.)

5. Recall that we have been checking the validity of Equation 9 for values of $t_d$ in $\mathcal{T}(\widehat{\Gamma})_1$, the partial testing set of $\Gamma$, considered in increasing order. Since we are currently considering $t_d$, we have therefore already validated that Equation 9 previously held for values of $t < t_d$ in the testing set. The crucial observation now is that *the increase in the value of $\widehat{C_i}$ for any $i$ with $D_i > t_d$ does not invalidate Equation 9 for any $t < t_d$*, because the increased $\widehat{C_i}$ values only contribute to the cumulative demand (the first term in the LHS of Equation 9) for values of $t \geq t_d$. Hence, we do not need to go back and re-validate Equation 9 for values of $t$ smaller than $t_d$.

6. Having thus modified the $\text{cnt}(v_{i,j})$ variables (as in Equation 14) in order to ensure that Equation 9 is satisfied by $\widehat{\Gamma}$ for $t_d$, we update the value of $t_d$ to the next-smallest value in $\mathcal{T}(\widehat{\Gamma})_1$, and return to Step 1 above.

7. Once $t_d$ reaches $D_{\max}$, there are no remaining tasks with $D_i > t_d$, and so we are done updating the cnt variables. At this point, we check if the total utilization of the system, $\sum_{i=1}^{n} \frac{WCET(\tau_i)}{T_i} > 1$. If it is, the system cannot be scheduled.

8. For an implicit-deadline system with $U \leq 1$, we prove in Lemma 2 that the slack will never be $< 0$ at any later time. Therefore, we know that the system is schedulable and can return immediately.

9. For a constrained-deadline system, we check the slack at each remaining point $t_d$ in the testing set $\mathcal{T}(\widehat{\Gamma})_2$ up to and including the testing set's upper bound, described by Expression 3. If the slack is found to be $< 0$, the system is unschedulable, and we return immediately. Otherwise, is system is deemed to be schedulable.

**Computational Complexity.** The number of iterations of the *for-loops* of Lines 6 and 23 dominate the computational complexity; the other loops in the algorithm are polynomial in the number of tasks or number of vertices in a chain. The number of iterations of Line 6 is proportional to $D_{\max}$; for implicit-deadline systems, this is the only loop that executes. For constrained-deadline systems, the combined number of iterations for both loops is the number of testing set points. In general, for constrained-deadline sporadic task systems scheduled on a single processor, the testing set can be exponential in the number of tasks, but is psuedo-polynomial as long as the utilization is bounded by a fixed constant strictly less than 1 [8]. For MPS sporadic tasks, the utilization can change as we add preemption points. However, because we only continue to add preemption points as the testing set is traversed up to $D_{\max}$, the testing set remains pseudo-polynomial in size unless the utilization reaches exactly 1, in which case it becomes exponential (bounded by the least-common multiple of the task periods).

**Proof of Correctness/Optimality.** We now provide formal arguments that our approach for chains yields a correct assignment of $\beta_i$ values for all tasks (Theorem 3) and is optimal in the sense that if the approach returns THE SYSTEM IS NOT SCHEDULABLE, then there is no assignment of $\beta_i$s that would cause the system to become schedulable (Theorem 4).

Before we prove the two main theorems of the section, we prove a useful invariant for the *for-loop* in Line 6 of Algorithm 1. The remainder of this section assumes the fixed-preemption model where a preemption is always taken at a fixed-preemption point (i.e., incurring the teardown/startup costs). In this model, it can be shown that Equation 9 remains a necessary and sufficient condition for limited-preemption EDF schedulability. However, under other preemption models where we may skip/delay preemptions, Equation 9 is only a sufficient condition. Thus, only the correctness theorem will hold, and we leave an investigation of optimality for these more dynamic settings to future research.

▶ **Lemma 1.** *At the beginning of each iteration of the* for-loop *at Line 6 with $t_d$ being the current testing set interval considered, the following statements hold:*

1. *For any $t < t_d$, Equation 9 is satisfied for the current set of $\beta_i$ values.*
2. *For any $\tau_i \in \Gamma$ such that $D_i \leq t_d$, the value of $\beta_i$ set by the algorithm is maximum (over all possible schedulable chunk-size configurations of $\Gamma$).*

**Proof.** Lemma 1 is, as expected, proved by induction.

**Initialization.** Initially $t_d$ equals $D_{\min}$; the minimum deadline is the first non-zero value of the DBF function for all tasks. Thus, Statement 1 is true since at all prior timepoints the first term of Equation 9 is zero and Equation 9 reduces to $L \leq L$ per the arguments in Section 3.2. Statement 2 is also vacuously true since $\beta_1$ is set to its largest possible value $\max_{1 \leq j \leq n_1} (c(v_{1,j}) + q(v_{1,j}))$ by the previous loop and does not affect schedulability as $\tau_1$ cannot block any other task (by nature of having the smallest relative deadline).

**Maintenance.** Let us consider the current testing-set point $t_d$. Let $t_c$ be the testing-set point considered in the previous iteration of the *for-loop*. Assume that Statements 1 and 2 were true at the beginning of the *for-loop* for point $t_c$; we will show that the statements will hold for $t_d$.

For Statement 1, we must show that Equation 9 is not invalidated when we execute the *for-loop* for $t_c$. As was previously argued, for $t < t_c$, the DBF values are unchanged as any changes made to $\beta_i$ values in the iteration for $t_c$ only affect the $\widehat{C_i}$ of tasks with $D_i > t_c$. Thus, Statement 1 continues to hold for all $t < t_c$ by assumption. We only need to show that the previous iteration will set the corresponding $\beta$ values such that Equation 9 will also be true at $t_c$. However, this obviously holds since for each $\tau_i$ with $D_i > t_c$, either $\beta_i$ already satisfied Equation 9 (and does not change) or it is set by Line 12 of Algorithm 1 to the largest value that satisfies Equation 9 for the current values of $\beta$.

Statement 2 follows from the last observation of the previous paragraph and by the assumption that $\beta$ values for all $\tau_i$ with $D_i \leq t_c$ are set to their maximum value by assumption and cannot change at $t_c$ or after. Therefore, the $\widehat{C_i}$ values that contribute to the DBF in Equation 9 at $t_c$ are as small as possible. It therefore follows that any $\tau_j$ with $t_c \leq D_j \leq t_d$ has either already had its $\beta_j$ value set to the largest possible to satisfy Equation 9 for some $t < t_c$ or has its value set to the largest possible to satisfy Equation 9 at $t_c$.

**Termination.** Let $t_d$ be the last testing set interval considered by the for loop in Line 6. During the execution of the loop, the algorithm may return THE SYSTEM IS NOT SCHEDULABLE. In the case that not schedulable is returned, Statements 1 and 2 hold for all testing set intervals up to (and including) $t_d$, but not necessarily after $t_d$. If the algorithm completes execution of the loop without returning, the Statements 1 and 2 are guaranteed to hold for all testing set intervals in $\mathcal{T}(\widehat{\Gamma})_1$ (by properties of testing-sets for Equation 9 and that the last testing set point must be at least $D_N$). ◀

▶ **Lemma 2.** *In an implicit-deadline task system, Equation 9 will be satisfied at all points $L > D_{\max}$ if $U \leq 1$.*

**Proof.** By contradiction. Consider some $t_d > D_{\max}$ in the testing set at which Equation 9 fails to hold. At this point, there are no tasks where $D_i > t_d$, and so Equation 9 becomes $\sum_{\tau_i \in \Gamma} \text{DBF}_i(t_d) \leq t_d$. From the definition of the DBF:

$$\sum_{\tau_i \in \Gamma} \max\left(\left\lfloor \frac{t_d - D_i}{T_i} \right\rfloor + 1, 0\right) \cdot C_i > t_d$$

Since for an implicit-deadline task system, $D_i = T_i$ for all tasks $\tau_i$,

$$\sum_{\tau_i \in \Gamma} \max\left(\left\lfloor \frac{t_d}{T_i} \right\rfloor, 0\right) \cdot C_i > t_d$$

From which it follows that

$$\sum_{\tau_i \in \Gamma} \max\left(\frac{t_d}{T_i}, 0\right) \cdot C_i > t_d$$

As $t_d$ and $T_i$ are both positive:

$$\sum_{\tau_i \in \Gamma} \frac{t_d}{T_i} \cdot C_i > t_d$$

which implies that $U > 1$. ◀

▶ **Theorem 3.** *If the schedulability test returns* THE SYSTEM IS SCHEDULABLE*, then assignment of $\beta_i$ values by the algorithm in Algorithm 1 ensures that the task system meets all deadlines when scheduled by limited-preemption EDF.*

**Proof.** The termination argument of Lemma 1 argues that Statement 1 and therefore Equation 9 hold for all testing set points in $\mathcal{T}(\widehat{\Gamma})_1$. If the system is an implicit-deadline system, then by the check in line 18, the system utilization must be $\leq 1$; by Lemma 2, Statement 1 will continue to hold for all points in $\mathcal{T}(\widehat{\Gamma})_2$. If the system is a constrained-deadline system, the loop in Line 23 will complete and return THE SYSTEM IS SCHEDULABLE if and only if Equation 9 holds for all points in $\mathcal{T}(\widehat{\Gamma})_2$.

Since Equation 9 is sufficient (and necessary for the fixed-preemption model), the task system $\Gamma$ is schedulable by limited-preemption EDF when the assigned chunk-sizes $\beta_i$ are used. ◀

▶ **Theorem 4.** *If the schedulability test returns* THE SYSTEM IS NOT SCHEDULABLE*, then there does not exist an assignment of $\beta_i$ values such that Equation 6 is satisfied.*

**Proof.** If THE SYSTEM IS NOT SCHEDULABLE is returned while processing a testing-set interval $t_d$ in $\mathcal{T}(\widehat{\Gamma})_1$ then either $\Delta(t_d) < 0$ or $\beta_i < q(v_{i,j})$ for some $i$ and $j$. In either case, Statement 2 of Lemma 1 implies that the $\beta_i$'s set prior to $t_d$ are as large as possible with respect to $t < t_d$ for Equation 9. Therefore, if $\Delta(t_d) < 0$ is true, it is not possible to find another assignment of $\beta_i$'s to make this false. Otherwise, if $\beta_i < q(v_{i,j})$, the fact that $\beta_i$ was set to its maximum value means there does not exist a larger possible $\beta_i$ to successfully fit task execution into given the startup/teardown costs $q(v_{i,j})$ of the phase $v_{i,j}$.

If THE SYSTEM IS NOT SCHEDULABLE is returned while processing a testing-set interval $t_d$ in $\mathcal{T}(\widehat{\Gamma})_2$, then Equation 9 is false for some $t_d \in \mathcal{T}(\widehat{\Gamma})_2$. The right-side term of equation 9 considers only tasks where $D_i > L$; as $\mathcal{T}(\widehat{\Gamma})_2$ begins past $D_{\max}$, there can be no tasks matching this condition and therefore no assignment of $\beta_i$ that would reduce the right-side term. Per Statement 2 of Lemma 1, each $\beta_i$ has already been maximized when considering $\mathcal{T}(\widehat{\Gamma})_1$; reducing some $\beta_i$ can only increase the number of preemption points required and therefore increase the DBF of some task in the left-side term of Equation 9. Therefore, there is no alternative assignment of $\beta$'s that would reduce the left side of Equation 9 and cause the system to become schedulable. ◀

## 5 Systems of Conditional MPS Sporadic Tasks

In Section 4 we considered recurrent tasks representing "linear" workflows: each task models a piece of straight-line code comprising a sequence of phases that are to be executed in order. In many event-driven real-time application systems, however, the code modeled by a task may include conditional constructs ('if-then-else' statements) in which the outcome of evaluating a condition depends upon factors (such as the current state of the system, the values of certain external variables, etc.), which only become known at run-time, and indeed may differ upon different invocations of the task. Hence the precise sequence of phases that is to be executed when a task is invoked is not known a priori. It is convenient to model such tasks as directed acyclic graphs (DAGs) in which the vertices represent execution of straight-line code, and a vertex representing a piece of straight-line code ending in a conditional expression has out-degree $> 1$ – see Figure 1 for an example. In this figure the vertex $a$ denotes a piece of straight-line code that ends with the execution of a conditional expression. Depending upon the outcome of this execution, the code represented by either the vertex $b$ or the vertex $c$ executes, after which the code represented by the vertex $d$ is executed. In this section, we briefly explain how our proposed Multi-Phase Secure (MPS) workload model may be further extended (i.e., beyond the aspects discussed in Section 4) to accommodate recurrent tasks that may include such conditional constructs.

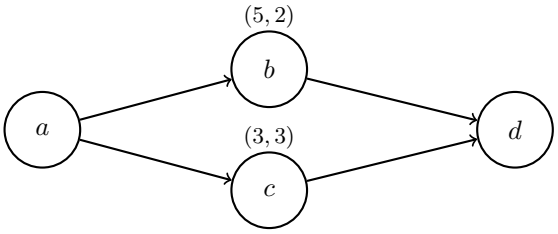



**Figure 1** Each vertex characterized by a pair of values; the former one representing the WCET of the node/phase, $c(v)$ and the latter one representing the sum of the startup and teardown cost, $q(v)$. We assume that the code represented by vertices $a$ and $d$ execute using the same security mechanism, whereas the code represented by vertices $b$ and $c$ each execute using a distinct different mechanism.

## 5.1 Model

We now provide a more formal description of the *conditional MPS sporadic task model*. Each task $\tau_i$ is characterized by a 3-tuple $(G_i, D_i, T_i)$ where $D_i$ and $T_i$ are the relative deadline and period, and $G_i$ is a DAG: $G_i = (V_i, E_i)$ with $E_i \subsetneq V_i \times V_i$. Each vertex $v_{i,j} \in V_i$ represents a phase of computation, which must execute using a specified security mechanism. The interpretation of each edge $(v_{i,j}, v_{i,k}) \in E$ depends upon the outdegree (i.e., the number of outgoing edges) of vertex $v_{i,j}$:

1. If this outdegree $= 1$, then the edge denotes a precedence constraint: vertex $v_{i,k}$ may only begin to execute after vertex $v_{i,j}$ has completed execution.
2. If this outdegree is $\geq 2$, then all the outgoing edges from $v_{i,j}$ collectively denote the choices available upon the execution of a conditional construct: after $v_{i,j}$ completes execution, exactly one of the vertices $v_{i,k}$ for which $(v_{i,j}, v_{i,k}) \in E$ becomes eligible to start executing. It is not known beforehand which one this may be, and different ones may become eligible upon different invocations of the task.

A WCET function $c : V_i \to \mathbb{N}$ is specified, with $c(v_{i,j})$ denoting the WCET of node $v_{i,j} \in V_i$. An overhead function $q : V_i \to \mathbb{N}$ is specified, with $q(v_{i,j})$ denoting the startup/teardown cost associated with the security mechanism using which vertex $v_{i,j}$ is to execute.

In the original model presented in [3], a simplifying assumption was made that teardown and setup costs are always incurred when transitioning between any two jobs, regardless of whether they use the same or different security mechanisms. This assumption was conservative, as it did not account for cases where both jobs use the same security mechanism, in which case the teardown and setup costs would not actually be incurred.

## 5.2 A Subtlety

A subtle issue arises when dealing with conditional code, which was not present in the consideration of linear workflows in Section 4. Specifically, during the execution of conditional code, it is not known at compile time which branch will be taken at runtime, and different invocations may take different paths. In hard real-time systems, it is necessary to ensure that tasks meet their deadlines under all possible runtime conditions. Thus, during pre-runtime schedulability analysis, a conservative approach is adopted, assuming that each invocation of the task follows the "longest" path – the one with the maximum cumulative execution requirement.

Standard algorithms are known for identifying the longest path through a DAG that have running time linear in the representation of the DAG. Under limited-preemption scheduling, however, identifying the path through the DAG that has maximum cumulative execution requirement is not entirely straightforward. Let us consider again the example DAG of Figure 1.

- If the chunk size $\beta_i$ for this task were $\geq 7$, then both branches can be executed non-preemptively. The upper branch incurs a cost of $5 + 2 = \mathbf{7}$ while for the lower branch the cost is $3 + 3 = \mathbf{6}$; therefore, the upper branch is the computationally more expensive one.
- Now suppose $\beta_i = 4$. Then the upper branch may need to execute in $\lceil 5/(4-2) \rceil$ or 3 contiguous pieces, for a cumulative cost of $5 + 3 \times 2 = \mathbf{11}$. The lower branch may need to execute in $\lceil 3/(4-3) \rceil$ or 3 contiguous pieces, for a cumulative cost of $3 + 3 \times 3 = \mathbf{12}$, and is hence the more expensive branch.

This example illustrates that the computationally most expensive path through a DAG that represents conditional execution depends upon the value of the chunk size parameter (the $\beta_i$ parameter of the task). As we saw in Section 4, the approach to scheduling MPS sporadic tasks has been to convert each task to a limited-preemption sporadic task. The procedure for doing so (Algorithm 1) repeatedly changes the values of the $\beta_i$ parameters of the tasks. As we adapt the techniques of Section 4 to schedulability analysis of conditional MPS tasks, it is important to note that the necessary improvements to handle conditional execution, including the correct analysis of teardown and setup costs, have already been addressed in a separate extension paper [38]. In that work, the unnecessary conservatism in the assumption that teardown and setup costs are always incurred during transitions between jobs, regardless of whether they use the same security mechanism is eliminated. This refinement leads to a more precise schedulability analysis, ensuring that only the relevant overheads are considered and optimizing the overall analysis of conditional MPS tasks. Thus, while this paper focuses on the linear task model and its optimization, the more accurate treatment of conditional execution is fully covered in [38].

## 6 Empirical Evaluation

In the previous sections, we have developed algorithms to introduce preemptions into the execution of the phases of multi-phase secure tasks. While these preemptions reduce the blocking that higher-priority phases/jobs experience at runtime, they come at a cost – increased overhead due to the additional teardown/startup costs that must be performed before and after every inserted preemption. Thus, while the limited-preemption approach proposed in this paper theoretically dominates the non-preemptive approach (i.e., each phase is executed non-preemptively and preemptions are permitted only between phases of a task), it is unclear how much schedulability improvement *on average* we can expect a system designer to obtain from using our proposed approach.

In this section, we provide an empirical analysis on that topic via the application of the schedulability tests of Section 4 over synthetically-generated MPS task systems. We compare the proposed schedulability tests with existing limited-preemption scheduling where each phase of a task is executed fully non-preemptively.

By using the refined algorithms presented in this work that permit pseudo-polynomial running times for bounded utilization task sets, and correcting implementation issues identified in our earlier work [3], we were able to extend our experiments to a broader parameter space and larger task sets. We also directly evaluate the running times of our algorithms and implementation, and compare them to the original versions from [3].

As before, we limit our scope to evaluating linear task sequences; we refer readers interested in further evaluation, analysis, and refinement of the conditional model to [38].

### 6.1 Experimental Setup

The evaluation was conducted using a C++ simulation. All tests were performed on a a server with two Intel Xeon Gold 6130 (Skylake) processors running at 2.1 GHz, and with 64GB of memory. Multiple task sets were evaluated in parallel; each task set was given a single thread

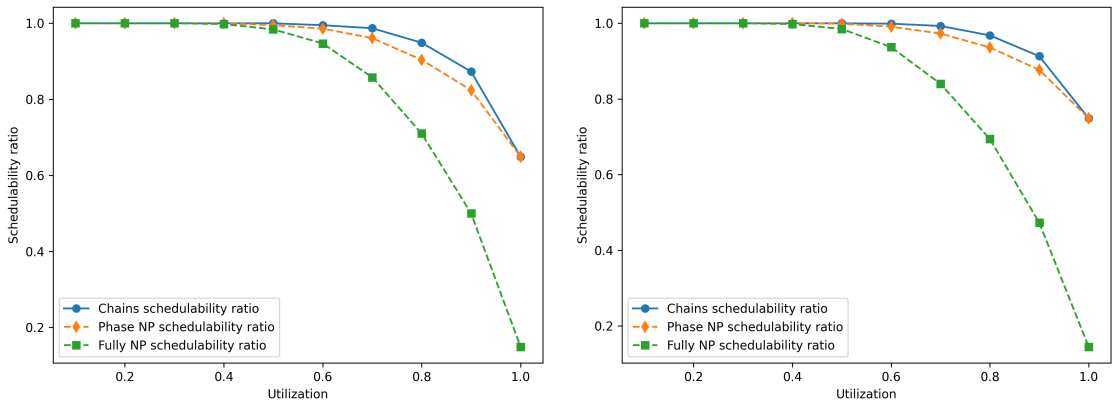


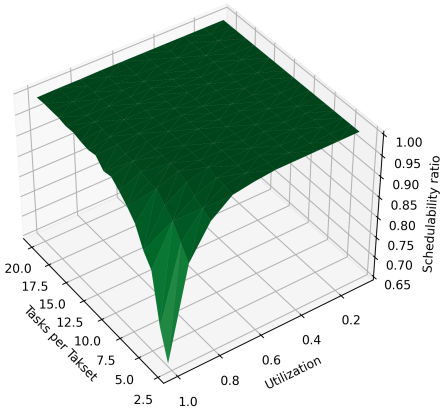

**Figure 2** Schedulability ratio of implicit-deadline tasksets with 3 tasks each and periods of 10–30 time units. On the left, tasks are randomly assigned 1–4 phases; on the right, 1–6.

on which to run. We evaluate task sets for many parameter variations, including the task count, number of phases per task, utilization, and task periods. In some cases, the number of phases per task is fixed; where it is not fixed, we select the number of phases for each task following a uniform distribution in the given range. Task periods are selected either from a uniform distribution within the period range specified for each test or, where specified below, a log-uniform distribution. For each combination of fixed parameters, we generate 1000 random task systems. For each system, we use the UUniFast algorithm [12], first to assign a total utilization $\frac{C_i}{T_i}$ to each task and then to distribute the task's total allotted time $C_i$ between the execution times $c_{(v_{i,j})}$ and startup/teardown overhead $q_{(v_{i,j})}$ for each of its phases. We note that compared to the prior work in [3] that this paper extends, we have adjusted this distribution to be more consistent and to be independent of the number of phases in the task. We then evaluate each task set against three algorithms: *chains*, the algorithm presented in section 4, *phase NP*, in which there is a static preemption point between each phase but no additional preemption points can be inserted, and *fully NP*, in which the entire task runs as a single non-preemptive chunk.

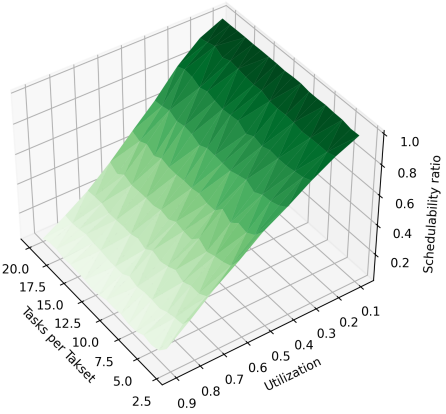
We evaluate both implicit-deadline and constrained-deadline task systems. In an implicit-deadline system, $D_i = T_i$ for each task. In a constrained-deadline system, we choose a random value for each $D_i$ that is uniformly distributed between the task's execution time and its period $T_i$. For implicit-deadline systems, we evaluate $U$ at increments of 0.1 in $[0, 1]$. We note that, compared to the prior version of this work in [3], the improvement to the testing set described in Section 4 allows us to evaluate these systems in a reasonable amount of time even for large numbers of tasks; to understand the impact of this optimization, we also test the exponential set presented in the original version on systems with 8 tasks or fewer and compare their execution times. For constrained-deadline systems, the testing set is bounded by a term that is determined in part by a factor $\frac{1}{1-U}$; as $U \to 1$, the testing set size becomes very large and becomes infeasible to evaluate. We therefore limit our evaluation of constrained-deadline systems to utilizations in $[0, 0.9]$.

## 6.2 Schedulability Analysis Results

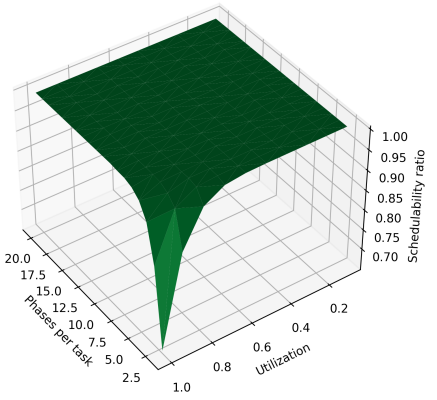
**Schedulability of Implicit-Deadline Systems.** Figure 2 shows the schedulability ratio of each of the three tests described above – chains, phase NP, and fully NP. For each test, tasks with low utilization are all schedulable, but schedulability decreases as the utilization of the system increases. By inserting additional preemption points, the chains algorithm is able to obtain a



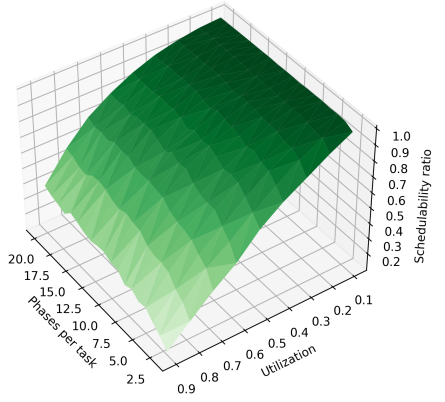

**(a)** Implicit deadlines, 1–4 phases.



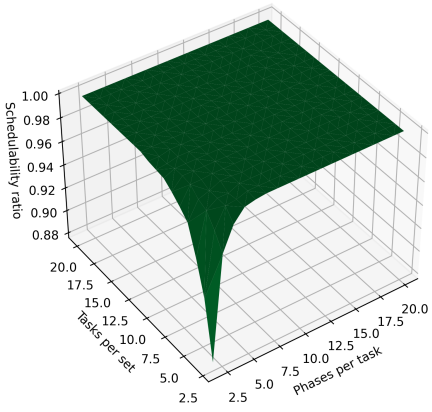

**(b)** Constrained deadlines, 1–4 phases.



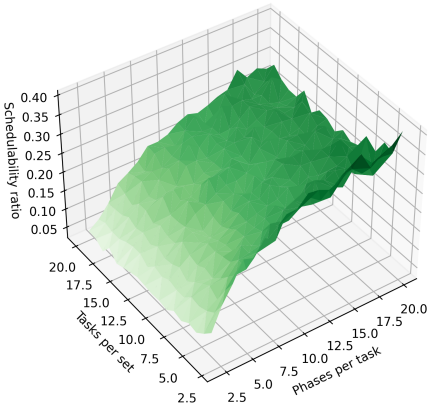

**(c)** Implicit deadlines, 3 tasks per set.




**(d)** Constrained deadlines, 3 tasks per set.




**(e)** Implicit deadlines, utilization 0.9.




**(f)** Constrained deadlines, utilization 0.9.

**Figure 3** Schedulability ratio of chains algorithm when varying taskset parameters. All tasks have periods selected uniformly from 10–30 time units.

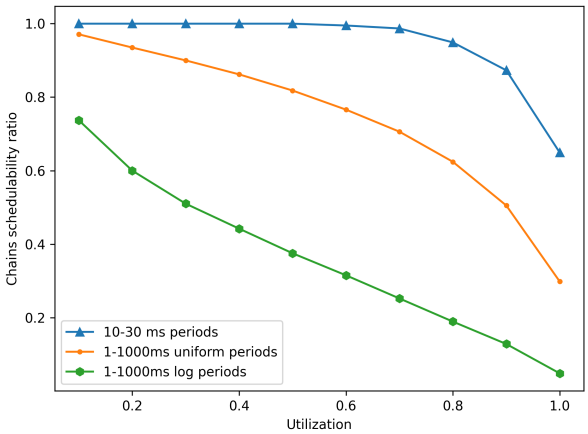


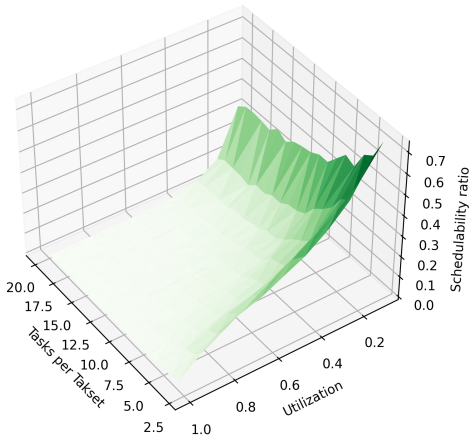

**Figure 4** Comparison of the effect on schedulability of changing the period range of the generated tasks. Each set contains 3 implicit-deadline tasks with 1–4 phases.

schedulability improvement over a phase NP approach on these higher-utilization tasks. Once utilization reaches 1, it is not possible to insert any preemption points, as inserting a preemption point would increase the startup/teardown overhead and cause the utilization to exceed 1; therefore, the chains algorithm does not lead to a schedulability improvement. Tasks with 1-6 phases have a slightly higher schedulability ratio than tasks with 1-4 phases; we explore this effect more in the paragraph below.

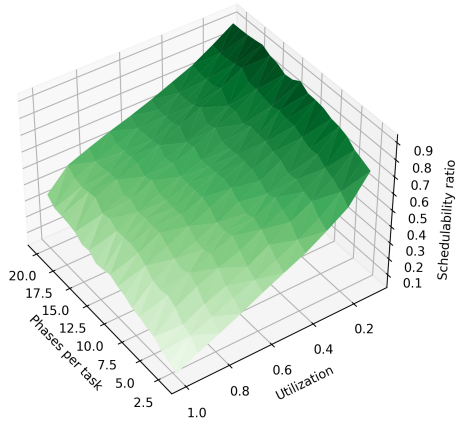
Figure 3 shows the impact of various parameters of the task system on the schedulability ratio using the chains algorithm and periods of 10–30 time units. In Figure 3a, tasks are randomly assigned 1-4 phases, while the utilization and tasks per taskset are varied. In Figure 3c, the number of tasks is held constant at 3, and each taskset is assigned a fixed number of phases, varying from 2 to 20 phases per task. In Figure 3e, the utilization is held constant at 0.9. As in Figure 2, increasing utilization reduces the schedulability ratio. Here, it is clearly visible that increasing the number of tasks or number of phases per task improves the schedulability ratio. When these parameters are increased, the system's total execution time is divided among more tasks or among more phases, so that each task has a lower blocking time. The reduction in blocking time makes the system more likely to be schedulable.

For completeness, we also evaluate the performance of the algorithm on a wider 1–1000 time unit period range, using both a uniform distribution of periods within this range, as well as the log-uniform distribution recommended in [19]. Figure 4 shows how the schedulability of these task sets compares to the tasksets with 10–30 unit periods. The schedulability of the sets with the wider period range is much lower; in particular, tasksets containing a task with a small period are less likely to be schedulable. We hypothesize that this is due to these high-frequency tasks having less ability to expand their execution time as preemption points are inserted. This trend is also apparent in Figure 5; unlike tasks with 10–30 unit periods, increasing the number of tasks increases the probability of generating a high-frequency task and therefore causes the schedulability ratio to decrease.

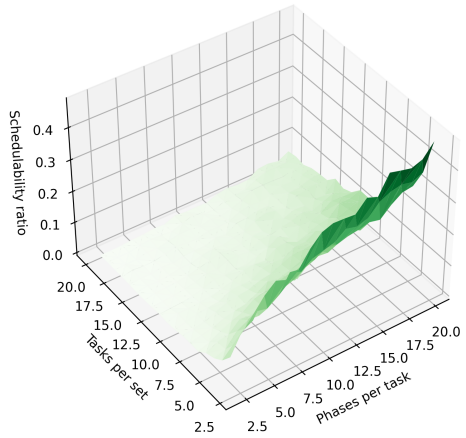
**Schedulability of Constrained-Deadline Systems.** Figure 6 shows the schedulability ratio for each of the three scheduling algorithms on constrained-deadline tasksets, in which all other task parameters follow the same configuration as Figure 2. In a constrained-deadline system, the chains algorithm is also able to obtain a schedulability improvement over a phase NP approach, by




**(a)** Implicit deadlines, 1–4 phases.




**(b)** Implicit deadlines, 3 tasks per set.




**(c)** Implicit deadlines, utilization 0.9.

**Figure 5** Schedulability ratio of chains algorithm when varying taskset parameters. All tasks have periods selected from 1–1000 time units using a log-uniform distribution.

inserting additional preemption points to reduce the blocking time. As the tasks in these systems have shorter deadlines than the implicit-deadline systems, all three algorithms obtain a lower schedulability ratio.

The right column of Figure 3 shows the effect of varying different taskset parameters for the constrained-deadline task sets, which otherwise have the same parameter configurations as the implicit-deadline systems. For constrained-deadline tasks, the schedulability ratio is generally lower, and the effect of increasing the phase count is smaller. Increasing the number of tasks has only a very small effect on the schedulability ratio. We hypothesize that, although increasing the task count still tends to reduce the system's blocking times, it also increases the probability that one or more tasks will have a tightly-constrained deadline, which reduces the chance that the system will be schedulable.

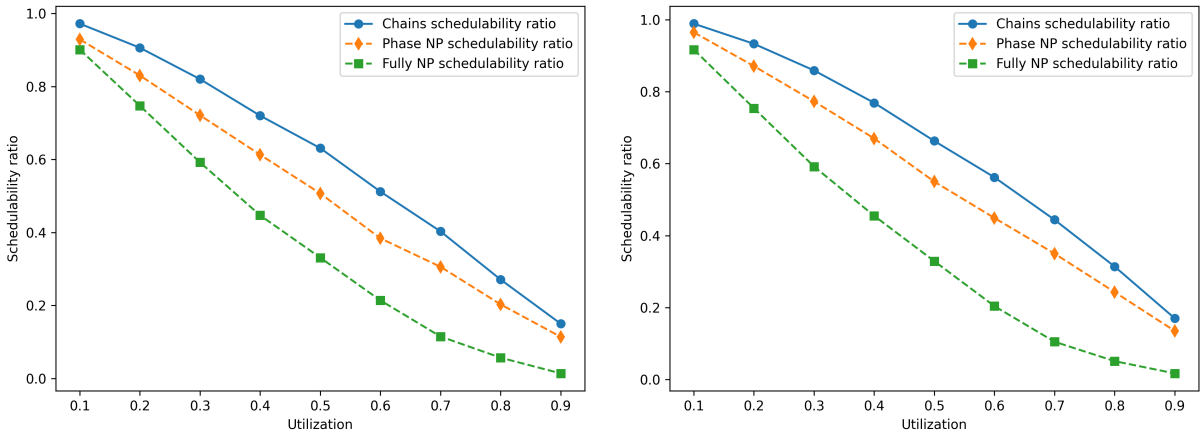



**Figure 6** Comparison of schedulability ratios. Each set has 3 constrained-deadline tasks with periods selected uniformly from 10–30 time units. On the left, tasks are randomly assigned 1–4 phases; on the right, 1–6.

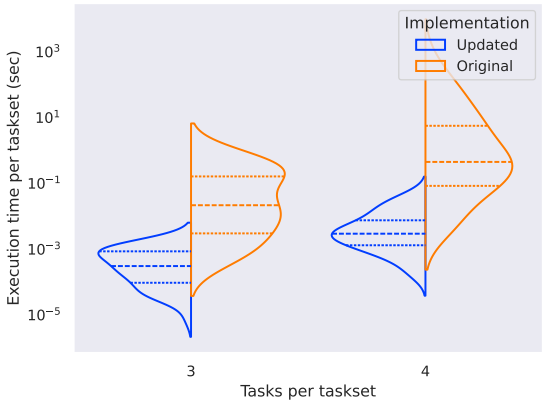



**Figure 7** Comparison of the time needed to determine schedulability for task sets with 1-4 phases, utilization of 0.9, and periods of 10-30. The left side is the rewritten C++ implementation; the right side is the original implementation from [3]. Note the log scale.

## 6.3 Runtime Performance

The implementation of the algorithm in our prior work [3] could only evaluate sets of a few tasks in a reasonable amount of time. In this work, we rewrite the original Python-based simulation using C++, and correct an implementation issue with the original construction of the hyperperiod-bounded testing set. These changes lead to significant performance improvements on their own. In Figure 7 we randomly generate and test 50 task sets with either 3 or 4 tasks each; the original implementation is compared to the rewritten one, which is configured to test points in the full, exponentially-sized testing set. For sets of 3 tasks, the median execution time improves by approximately 70× and the mean execution time improves by around 250×. For sets of 4 tasks, the median improves by around 150× and the mean execution time by more than 3500×. These improvements enable testing larger task sets, as shown in the following sections.

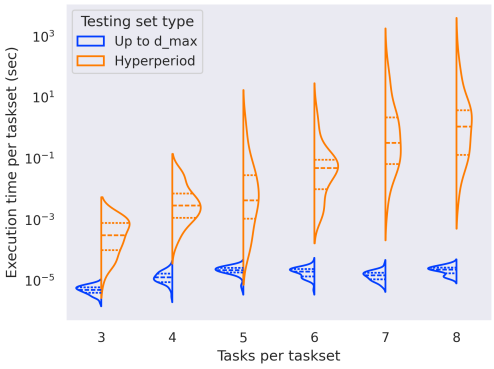



**Figure 8** Time to test schedulability of sets of implicit-deadline tasks of varying sizes with periods of 10–30 time units, 1–4 phases, and utilization of 0.9, either testing up to the hyperperiod or stopping at $D_{\max}$. Note the logarithmic scale.

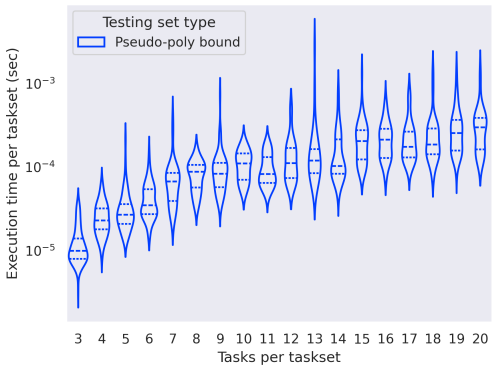



**Figure 9** Time to test schedulability of sets of constrained-deadline tasks of varying sizes with periods of 10–30 time units, 1–4 phases, and utilization of 0.9, using the psuedo-polynomial testing set bound from [7]. Note the logarithmic scale.

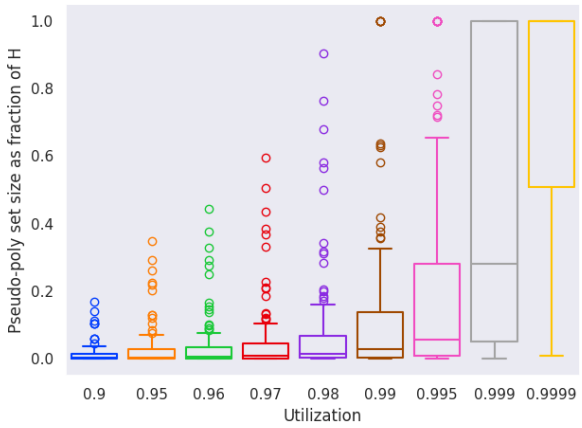



**Figure 10** Comparison of the sizes of the hyperperiod and pseudo-polynomial testing sets. For each utilization, we generated 100 sets of 5 constrained-deadline tasks with periods from 10–30, and 1–4 phases.

In this work, we also introduce an optimization for implicit-deadline task systems that allows us to avoid testing timepoints past $D_{\max}$. In Figure 8, we analyze the impact of this optimization. Using the full, exponential testing set from the prior work, our implementation is able to test tasksets with 6 tasks in only a few seconds, but the time needed still scales exponentially with the number of tasks; past eight tasks per set, the analysis once again takes an unreasonable amount of time to run. Using the optimization for implicit-deadline systems, evaluating schedulability is orders of magnitude faster, and the evaluation times also become more consistent for each taskset. This optimization allows us to determine the schedulability ratio for systems of up to 20 tasks, the results of which are displayed in Figure 3. As the time needed to determine schedulability now scales much more slowly with respect to the number of tasks, we believe that evaluating even larger task sets is also possible.

In Figure 9, we evaluate the performance of determining schedulability of constrained-deadline task sets using the pseudo-polynomial bound on the testing set identified in [7]. Compared to the full hyperperiod testing set, running the evaluation with this testing set is much faster. However,

we note that it is still slower than the optimized implicit-deadline bound from Figure 8. Moreover, the distribution of execution times exhibits high variability. Perhaps most importantly, our evaluation is on task sets where $U = 0.9$, at which the pseudo-polynomial bound is still reasonably small. As shown in Figure 10, when $U \to 1$, the size of the pseudo-polynomial set approaches the hyperperiod set, and it becomes increasingly less feasible to iterate over the entire testing set.

## 7 Related Work

The trade-off between security and timing constraints in real-time, IoT, and edge computing environments is crucial as it involves balancing robust security measures with the need for real-time performance. Several studies have explored these trade-offs. For instance, Leonardi et al. [22] and Lemieux-Mack et al [21] present mixed-integer linear programming approaches to optimize security and schedulability in real-time embedded systems under cyber-attacks. Wang et al. [40] consider the trade-off between security and overhead for pointer integrity checks. These papers attempt to maximize security without overloading the target system, but do not consider preemption-related overhead induced by the startup/teardown costs of the selected security mechanisms.

Limited-preemption scheduling, surveyed in 2013 by Buttazzo et al. [13], balances the increased blocking times of non-preemptive execution with the increased overheads caused by context switching. The original models of Baruah and Bertogna minimize context switching by finding the longest time each task may execute non-preemptively without compromising schedulability [4, 9], but do not account for the worst-case overhead due to the resulting preemptions.

Later approaches account for both blocking time and preemption overheads by defining instants that preemption can occur during task execution (called *preemption points*) to guarantee schedulability. In [10], Bertogna et al. develop an algorithm to find preemption points for tasks scheduled with EDF under the assumption that the preemption overhead for each task is constant. In contrast, in our MPS task model, individual task phases have unique preemption costs. A similar model to that of [10], but for fixed-priority (FP) scheduling, again with constant preemption overheads for each task, was developed by Yao et al. in [43].

In [11], Bertogna et al. model tasks as sequences of "basic blocks," and present algorithms for selecting preemption points between those blocks for both EDF and FP scheduling. Although less flexible than the earlier "floating" preemption point models where a preemption point can be placed anywhere, that model allows different preemption costs between blocks. The latter is similar to the algorithm in [10], but supports domain-specific preemption costs rather than a constant overhead per task.

Other work on limited-preemption scheduling, including cache-aware analysis [28], probabilistic [27] or "typical" execution models [6] are outside the scope of this paper.

## 8 Conclusions

We believe that the concurrent consideration of timing and security properties within a single unified framework is an effective means of extending the rigorous approach of real-time scheduling theory to guaranteeing appropriately-articulated security properties in resource-constrained embedded systems. In real-time scheduling theory, pre run-time verification of timing correctness is performed using models of run-time behavior; these models are carefully crafted for specific purposes: e..g, the sporadic task model [7] has been designed to represent recurrent processes for which it is safe to assume a minimum duration between successive invocations and for which timing correctness is defined as the ability to meet all deadlines.

In this work, as in [3], we have extended the sporadic task model in a security-cognizant manner, to deal with a particular kind of security-cognizant workload model. For the specific model that we have proposed, we have developed algorithms that are able to provide provable correctness of both the timing and the security properties that are considered.

In this extension to our original work in [3], we have corrected the long-standing condition of Baruah and Bertogna [4, 9] for EDF schedulability of limited-preemption tasks. Leveraging this, we reduced the execution time complexity of our algorithm, and demonstrated that in an implicit-deadline task system, it can run quickly even for large numbers of tasks.

We note that the improvements made to the algorithm in this extension also apply to the generalization of the model to conditional code that we presented in [3]. In an already-published extension to that work [38], we further improve on the conditional execution model by removing the assumption that startup/teardown costs are paid at every phase transition and analyze its performance.

The complexity improvements made in this extension, in combination with a more efficient implementation of the algorithm, allowed us to evaluate larger and more complex task systems. We have also illustrated the scaling properties of the algorithm's execution time for both constrained- and implicit-deadline systems, and shown how its ability to schedule tasks is affected by varying their properties. Finally, we have clarified several details of our algorithm and have provided a more complete demonstration of the improved schedulability offered by our algorithm over a phase or fully non-preemptive approach.

Although the work described in this manuscript arose out of our related projects in embedded systems security, we emphasize that we are not claiming that our algorithms solve any security problems; rather, they solve a scheduling problem that may arise from a class of security problems for which an adequate protective response gives rise to execution environments with bounded-cost startup/teardown operations. As future work, we intend to evaluate these scheduling models in conjunction with real-world attack/defense models.

On the other hand, we believe that our results are relevant beyond just security considerations: that they may, in fact, be considered to be further contributions to the real-time scheduling theory literature dealing with limited-preemption scheduling. They may also be extended to other limited-preemption scheduling frameworks, e.g., for multi-core platforms.

## References


1 N. C. Audsley, A. Burns, R. I. Davis, K. W. Tindell, and A. J. Wellings. Fixed priority preemptive scheduling: An historical perspective. *Real-Time Systems*, 8:173–198, 1995. doi:10.1007/BF01094342.

2 Mohammad Fakhruddin Babar and Monowar Hasan. Deeptrustˆ RT: Confidential deep neural inference meets real-time! In Rodolfo Pellizzoni, editor, *36th Euromicro Conference on Real-Time Systems, ECRTS 2024, July 9-12, 2024, Lille, France*, volume 298 of *LIPIcs*, pages 13:1–13:24. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPIcs.ECRTS.2024.13.

3 Sanjoy Baruah, Thidapat Chantem, Nathan Fisher, and Fatima Raadia. A scheduling model inspired by security considerations. In *2023 IEEE 26th International Symposium on Real-Time Distributed Computing (ISORC)*, pages 32–41, 2023. doi:10.1109/ISORC58943.2023.00016.

4 Sanjoy K. Baruah. The limited-preemption uniprocessor scheduling of sporadic task systems. In *17th Euromicro Conference on Real-Time Systems (ECRTS 2005), 6-8 July 2005, Palma de Mallorca, Spain, Proceedings*, pages 137–144. IEEE Computer Society, 2005. doi:10.1109/ECRTS.2005.32.

5 Sanjoy K. Baruah. Security-cognizant real-time scheduling. In *25th IEEE International Symposium On Real-Time Distributed Computing, ISORC 2022, Västerås, Sweden, May 17-18, 2022*, pages 1–9. IEEE, 2022. doi:10.1109/ISORC52572.2022.9812766.

6 Sanjoy K. Baruah and Nathan Fisher. Choosing preemption points to minimize typical running times. In *Proceedings of the 27th International Conference on Real-Time Networks and Systems, RTNS 2019, Toulouse, France, November 06-08, 2019*, RTNS '19, pages 198–208, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3356401.3356407.

7 Sanjoy K. Baruah, Aloysius K. Mok, and Louis E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of the Real-Time Systems Symposium - 1990, Lake Buena Vista, Florida, USA, December 1990*, pages 182–190, Orlando, Florida, 1990. IEEE Computer Society. doi:10.1109/REAL.1990.128746.

8 Sanjoy K Baruah, Louis E Rosier, and Rodney R Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Syst.*, 2(4):301–324, November 1990. doi:10.1007/BF01995675.

9 Marko Bertogna and Sanjoy K. Baruah. Limited preemption EDF scheduling of sporadic task systems. *IEEE Trans. Ind. Informatics*, 6(4):579–591, 2010. doi:10.1109/TII.2010.2049654.

10 Marko Bertogna, Giorgio Buttazzo, Mauro Marinoni, Gang Yao, Francesco Esposito, and Marco Caccamo. Preemption Points Placement for Sporadic Task Sets. In *2010 22nd Euromicro Conference on Real-Time Systems*, pages 251–260, 2010. doi:10.1109/ECRTS.2010.9.

11 Marko Bertogna, Orges Xhani, Mauro Marinoni, Francesco Esposito, and Giorgio Buttazzo. Optimal selection of preemption points to minimize preemption overhead. In *2011 23rd Euromicro Conference on Real-Time Systems*, pages 217–227. IEEE, 2011. doi:10.1109/ECRTS.2011.28.

12 Enrico Bini and Giorgio C Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2), 2005. doi:10.1007/s11241-005-0507-9.

13 Giorgio C Buttazzo, Marko Bertogna, and Gang Yao. Limited preemptive scheduling for real-time systems. a survey. *IEEE Trans. Industr. Inform.*, 9(1):3–15, February 2013. doi:10.1109/TII.2012.2188805.

14 John Cavicchio and Nathan Fisher. Integrating preemption thresholds with limited preemption scheduling. In *2020 IEEE 26th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, August 2020. doi:10.1109/RTCSA50079.2020.9203646.

15 Friedrich Eisenbrand and Thomas Rothvoß. EDF-schedulability of synchronous periodic task systems is coNP-hard. In Moses Charikar, editor, *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*, pages 1029–1034. SIAM, January 2010. doi:10.1137/1.9781611973075.83.

16 Pontus Ekberg. *Models and Complexity Results in Real-Time Scheduling Theory*. PhD thesis, Uppsala University, Sweden, 2015. URL: https://nbn-resolving.org/urn:nbn:se:uu:diva-267017.

17 Pontus Ekberg and Wang Yi. Uniprocessor feasibility of sporadic tasks remains conp-complete under bounded utilization. In *2015 IEEE Real-Time Systems Symposium, RTSS 2015, San Antonio, Texas, USA, December 1-4, 2015*, pages 87–95. IEEE Computer Society, 2015. doi:10.1109/RTSS.2015.16.

18 Pontus Ekberg and Wang Yi. Uniprocessor feasibility of sporadic tasks with constrained deadlines is strongly conp-complete. In *27th Euromicro Conference on Real-Time Systems, ECRTS 2015, Lund, Sweden, July 8-10, 2015*, pages 281–286. IEEE Computer Society, 2015. doi:10.1109/ECRTS.2015.32.

19 P. Emberson, R. Stafford, and R.I. Davis. Techniques for the synthesis of multiprocessor tasksets. In *WATERS workshop at the Euromicro Conference on Real-Time Systems*, pages 6–11, July 2010. 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems ; Conference date: 06-07-2010.

20 Ke Jiang, Adrian Alin Lifa, Petru Eles, Zebo Peng, and Wei Jiang. Energy-aware design of secure multi-mode real-time embedded systems with FPGA co-processors. In Michel Auguin, Robert de Simone, Robert I. Davis, and Emmanuel Grolleau, editors, *21st International Conference on Real-Time Networks and Systems, RTNS 2013, Sophia Antipolis, France, October 17-18, 2013*, pages 109–118. ACM, October 2013. doi:10.1145/2516821.2516830.

21 Cailani Lemieux-Mack, Kevin Leach, Ning Zhang, Sanjoy Baruah, and Bryan C Ward. Optimizing runtime security in real-time embedded systems. In *Proc. of Workshop on Optimization for Embedded and Real-time Systems (OPERA)*, December 2024.

22 Sandro Di Leonardi, Federico Aromolo, Pietro Fara, Gabriele Serra, Daniel Casini, Alessandro Biondi, and Giorgio C. Buttazzo. Maximizing the security level of real-time software while preserving temporal constraints. *IEEE Access*, 11:35591–35607, 2023. doi:10.1109/ACCESS.2023.3264671.

23 M. Lin, L. Xu, L.T. Yang, X. Qin, N. Zheng, Z. Wu, and M. Qiu. Static security optimization for real-time systems. *IEEEII*, 5(1):22–37, February 2009. doi:10.1109/TII.2009.2014055.

24 C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973. doi:10.1145/321738.321743.

25 Yin Liu, Siddharth Dhar, and Eli Tilevich. Only pay for what you need: Detecting and removing unnecessary TEE-based code. *Journal of Systems and Software*, 188:111253, 2022. Publisher: Elsevier. doi:10.1016/j.jss.2022.111253.

26 Yue Ma, Wei Jiang, Nan Sang, and Xia Zhang. ARCSM: A distributed feedback control mechanism for security-critical real-time system. In *10th IEEE International Symposium on Parallel and Distributed Processing with Applications, ISPA 2012, Leganes, Madrid, Spain, July 10-13, 2012*, pages 379–386. IEEE Computer Society, July 2012. doi:10.1109/ISPA.2012.56.

27 Filip Markovic, Jan Carlson, Radu Dobrin, Bjorn Lisper, and Abhilash Thekkilakattil. Probabilistic response time analysis for fixed preemption point selection. In *2018 IEEE 13th International Symposium on Industrial Embedded Systems (SIES)*, pages 1–10, 2018. doi:10.1109/SIES.2018.8442099.

28 Filip Marković, Jan Carlson, and Radu Dobrin. Cache-aware response time analysis for real-time tasks with fixed preemption points. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 30–42, 2020. doi:10.1109/RTAS48715.2020.00-19.

29 Jonathan M McCune, Bryan J Parno, Adrian Perrig, Michael K Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 315–328, 2008. doi:10.1145/1352592.1352625.

30 Tanmaya Mishra, Thidapat Chantem, and Ryan M. Gerdes. Teecheck: Securing intra-vehicular communication using trusted execution. In *28th International Conference on Real Time Networks and Systems, RTNS 2020, Paris, France, June 10, 2020*, RTNS 2020, pages 128–138, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3394810.3394822.

31 Sibin Mohan, Man Ki Yoon, Rodolfo Pellizzoni, and Rakesh Bobba. Real-time systems security through scheduler constraints. In *Proceedings of the 2014 Agile Conference*, AGILE '14, pages 129–140, 2014. doi:10.1109/ECRTS.2014.28.

32 Anway Mukherjee, Tanmaya Mishra, Thidapat Chantem, Nathan Fisher, and Ryan M. Gerdes. Optimized trusted execution for hard real-time applications on COTS processors. In *Proceedings of the 27th International Conference on Real-Time Networks and Systems, RTNS 2019, Toulouse, France, November 06-08, 2019*, RTNS '19, pages 50–60, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3356401.3356419.

33 M. Nasri, T. Chantem, G. Bloom, and R. M. Gerdes. On the pitfalls and vulnerabilities of schedule randomization against schedule-based attacks. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 103–116, April 2019. doi:10.1109/RTAS.2019.00017.

34 Mitra Nasri, Geoffrey Nelissen, and Gerhard Fohler. A new approach for limited preemptive scheduling in systems with preemption overhead. In *28th Euromicro Conference on Real-Time Systems, ECRTS 2016, Toulouse, France, July 5-8, 2016*, pages 25–35. IEEE Computer Society, July 2016. doi:10.1109/ECRTS.2016.15.

35 OP-TEE. URL: https://www.trustedfirmware.org/projects/op-tee/.

36 OP-TEE Documentation: Core: Pager. URL: https://optee.readthedocs.io/en/latest/architecture/core.html#pager.

37 Miroslav Pajic, Nicola Bezzo, James Weimer, Rajeev Alur, Rahul Mangharam, Nathan Michael, George J. Pappas, Oleg Sokolsky, Paulo Tabuada, Stephanie Weirich, and Insup Lee. Towards synthesis of platform-aware attack-resilient control systems. In Linda Bushnell, Larry Rohrbough, Saurabh Amin, and Xenofon D. Koutsoukos, editors, *2nd ACM International Conference on High Confidence Networked Systems (part of CPS Week), HiCoNS 2013, Philadelphia, PA, USA, April 9-11, 2013*, pages 75–76. ACM, April 2013. doi:10.1145/2461446.2461457.

38 Fatima Raadia, Nathan Fisher, Thidapat Chantem, and Sanjoy Baruah. An improved security-cognizant scheduling model. In *2024 IEEE 27th International Symposium on Real-Time Distributed Computing (ISORC)*, pages 1–8. IEEE, 2024. doi:10.1109/ISORC61049.2024.10551349.

39 Jinwen Wang, Ao Li, Haoran Li, Chenyang Lu, and Ning Zhang. Rt-tee: Real-time system availability for cyber-physical systems using arm trustzone. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 352–369. IEEE, 2022. doi:10.1109/SP46214.2022.9833604.

40 Yujie Wang, Cailani Lemieux-Mack, Thidapat Chantem, Sanjoy Baruah, Ning Zhang, and Bryan C Ward. Partial context-sensitive pointer integrity for real-time embedded systems. In *2024 IEEE Real-Time Systems Symposium (RTSS)*, pages 415–426. IEEE Computer Society, 2024. doi:10.1109/RTSS62706.2024.00042.

41 T. Xie and X. Qin. Scheduling security-critical real-time applications on clusters. *IEEE Transactions on Computers*, 55(7):864–879, July 2006. doi:10.1109/TC.2006.110.

42 Gang Yao, Giorgio Buttazzo, and Marko Bertogna. Feasibility analysis under fixed priority scheduling with fixed preemption points. In *2010 IEEE 16th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 71–80, 2010. doi:10.1109/RTCSA.2010.40.

43 Gang Yao, Giorgio Buttazzo, and Marko Bertogna. Feasibility analysis under fixed priority scheduling with limited preemptions. *Real-Time Systems*, 47(3):198–223, 2011. Publisher: Springer. doi:10.1007/s11241-010-9113-6.

44 Man-Ki Yoon, Sibin Mohan, Chien-Ying Chen, and Lui Sha. Taskshuffler: A schedule randomization protocol for obfuscation against timing inference attacks in real-time systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Vienna, Austria, April 11-14, 2016*, pages 111–122. IEEE, IEEE Computer Society, 2016. doi:10.1109/RTAS.2016.7461362.

# Revisiting Slot-Shifting's Offline Acceptance Test for Sporadic Tasks: A Technical Note

**Mohammad Ibrahim Alkoudsi**
RPTU Kaiserslautern-Landau, Kaiserslautern, Germany

**Damir Isovic**
Mälardalen University, Västerås, Sweden

**Gerhard Fohler**
RPTU Kaiserslautern-Landau, Kaiserslautern, Germany

## Abstract

The *Slot-Shifting* algorithm presents a solution to combine the benefits of offline and online scheduling in time-triggered systems. It dynamically adjusts the allocation of time slots to tasks in the scheduling tables to accommodate aperiodic tasks at runtime. In this note, we revisit an extension to Slot-Shifting that enables it to handle sporadic task sets. In particular, we clarify the assumptions required for the correct application of its offline acceptance test, identify sources of pessimism within it, and address its schedulability analysis interval.

**2012 ACM Subject Classification** Computer systems organization → Real-time systems; Software and its engineering → Real-time schedulability

**Keywords and phrases** real-time systems scheduling, time-triggered systems, offline acceptance test of sporadic tasks

**Digital Object Identifier** 10.4230/LITES.10.1.4

**Funding** The work presented in this paper was supported in part by the Fonds National de la Recherche Luxembourg (FNR) and the Deutsche Forschungs Gemeinschaft (DFG) through the Core-Inter project ReSAC.

**Acknowledgements** The authors gratefully acknowledge Anaïs Finzi, Silviu S. Craciunas, and Marc Boyer, whose evaluation presented in [3] inspired our review of the original work, and thank them for comments and valuable discussions. We also would like to thank the reviewers for their constructive comments that contributed to the quality of the text.

**Received** 2025-05-16 **Accepted** 2025-10-27 **Published** 2025-12-09

## 1 Introduction

The *time-triggered (TT)* paradigm of activation has gained acceptance in many safety critical systems by ensuring reliability and predictability. It is particularly well-suited for applications with complex timing constraints, e.g., precedence, mutual exclusion, and distributed end-to-end deadlines. By triggering activities at predetermined points in time and letting them execute exclusively in dedicated time intervals (*slots*), extra-functional properties w.r.t. timeliness and dependability, e.g., determinism, reliability, and temporal partitioning, can be efficiently guaranteed.

The strict adherence of native TT implementations to offline defined scheduling tables prevents runtime flexibility. Slot-Shifting [4] is a method to handle independent *event-triggered (ET)* tasks, not already included in the scheduling tables at runtime, while maintaining the feasibility of applications and TT's extra functional properties. It does so by dynamically adjusting the allocation of time slots to tasks in the scheduling tables.

In [5], Isovic and Fohler extended Slot-Shifting from guaranteeing individual aperiodic tasks to sporadic task sets, i.e., tasks with unknown arrival times, but known minimum separation between consecutive instances. The extension includes an offline acceptance test for sporadic tasks, along

with modified runtime scheduling rules and an updated online acceptance test for aperiodic tasks to account for the presence of sporadic tasks. In this note, we revisit the offline acceptance test of the extension, clarify the assumptions required for its correct application, identify sources of pessimism within it, and address its schedulability analysis interval.

## 2 Guaranteeing ET Tasks with Slot-Shifting

The TT scheduling approach can be split into two phases:

**(i)** offline: before runtime, a scheduling algorithm takes the complex timing constraints of tasks and their messages in the distributed system as input and generates scheduling tables for every node and communication controller in the system. Each scheduling table contains a sequence of time slots, each allocated exclusively to a given task or message on the respective resource. Notably, time slots align with task arrivals and deadlines, as all task parameters are configured to be integer multiples of the slot size.

**(ii)** online: the operating system at every resource (node or communication controller) invokes the local dispatcher at the beginning of every slot to execute the task or transmit the message allocated to it in the resource scheduling table.

### 2.1 Individual Aperiodic Tasks

The offline phase of Slot-Shifting extends that of the native TT approach by supplanting the fixed slot allocation with less rigid target execution windows, defined for each TT task on each node. It divides each scheduling table into disjoint *capacity intervals* created from the deadlines of the target execution windows of TT tasks. Each of these intervals is annotated with the as soon as possible (ASAP) available spare processing capacity, i.e. the number of free slots based on as late as possible (ALAP) execution of offline tasks. The spare capacity is derived from the length of the interval and the demand required by TT tasks within the interval for their safe execution. Since the offline scheduling algorithm successfully generated a schedule for the entire hyperperiod of TT tasks, the calculated ALAP spare capacity at the beginning of every hyperperiod is the same.

During runtime, the local dispatcher on each node uses and maintains these spare capacities for the safe handling and scheduling of hard and soft aperiodic tasks arriving into the system, in a slot by slot basis and based on the earliest deadline first (EDF) policy. An online acceptance test analyzes if aperiodic tasks can be safely included in the system while maintaining TT's extra functional properties. It verifies if the demand of an aperiodic task can be retrofitted in the free slots of capacity intervals that are located within its execution window.

### 2.2 Sporadic Tasks

In [5], Isovic and Fohler extended Slot-Shifting from guaranteeing individual aperiodic tasks to sporadic task sets. The extension includes an offline acceptance test for analyzing the feasibility of extending the TT task set with sporadic tasks, along with the necessary adaptations for runtime scheduling. This involves extending Slot-Shifting's runtime rules for the dynamic allocation of slots to tasks and modifying the admission test for aperiodic tasks to account for the presence of both TT and offline-guaranteed sporadic tasks.

The offline acceptance test considers the worst-case arrival pattern of sporadic tasks for schedulability analysis. In this scenario, all sporadic tasks arrive synchronously at the same slot, with subsequent instances released at their maximum frequency. Consequently, the sporadic task set can be modeled as a synchronous periodic task set during the test, where all tasks share a global offset and each task's period is equal to its minimum inter-arrival time, as proposed in [2].

The offline test analyzes whether the worst-case demand of sporadic tasks can be accommodated within the available capacity intervals, under the assumption that sporadic tasks may arrive synchronously at every slot in the schedule. In other words, each slot is treated as a potential global offset value for sporadic tasks. Isovic and Fohler proposed an optimization to reduce the runtime of the offline acceptance test by limiting the number of global offset candidates for sporadic tasks. They prove that selecting a single representative slot per capacity interval (termed the *critical slot* in [5]) is sufficient to analyze schedulability.

# 3 Revisiting Slot-Shifting's Offline Acceptance Test for Sporadic Tasks

## 3.1 Clarification of Assumptions

The offline acceptance test assumes that TT tasks experience maximum delay and are forced to complete their execution ALAP. Consequently, the identification of critical slots within each capacity interval for the simultaneous arrival of sporadic tasks, and in turn the correctness of the acceptance test, is only established under this assumption[1]. Every slot in the schedule represents a global offset candidate for sporadic tasks. The critical slots defined in [5] serve to reduce the number of candidate slots to a single slot per capacity interval. However, in [5], there is no proof of the applicability of critical slots to ASAP scheduling. Identifying critical slots for the ASAP scheduling case is beyond the scope of this note.

## 3.2 Identification of the Pessimism

We now turn to identifying the pessimism in the offline acceptance test, which analyzes the feasibility of adding the worst-case sporadic demand alongside the ALAP-serviced TT demand based on the offline computed capacity intervals, and assuming that sporadic tasks arrive synchronously at the critical slot of each capacity interval in the TT schedule.

The runtime mechanisms defined in the Slot-Shifting extension prioritize ready ET tasks whenever spare capacity is available in the current interval. Consequently, ET tasks can impose forced delays on the execution of TT tasks, even when they do not have the earliest deadlines. In the worst-case scenario, TT tasks execute ALAP, while still meeting their deadlines. However, when no ET tasks are ready, TT tasks execute ASAP on a slot-by-slot basis and according to EDF policy.

Since the considered acceptance test is performed offline and aperiodic tasks are only admitted online, analyzing the schedulability of the combined TT and sporadic task set offline does not need to take aperiodic tasks into account. This implies that, online, TT tasks execute ASAP prior to the synchronous arrival of sporadic tasks at a critical slot. Hence, the actual spare capacity values at a given slot deviate from those originally computed under the assumption of ALAP execution. Since we can precisely determine which TT tasks execute prior to the critical slot and for how many slots (based on their release times, worst-case execution times (WCET), and EDF-based priorities), we can precisely calculate the change in capacity values. Each slot used by a TT task reduces its remaining demand by one. This does not consume a free slot but may shift it to a later point in time, meaning that there can be more available spare capacity online compared to the offline calculated values based on ALAP execution in the test. Consequently, assuming less available spare capacity than actually present based on ALAP execution of TT

[1] The reasoning in the proof of Theorem 1 in [5] is grounded in ALAP reservation of slots for TT tasks. This is evidenced by the absence of any change in spare capacity within the considered arrival interval, expressed as $\alpha = 0$, when the arrival time of sporadic tasks is shifted forward.

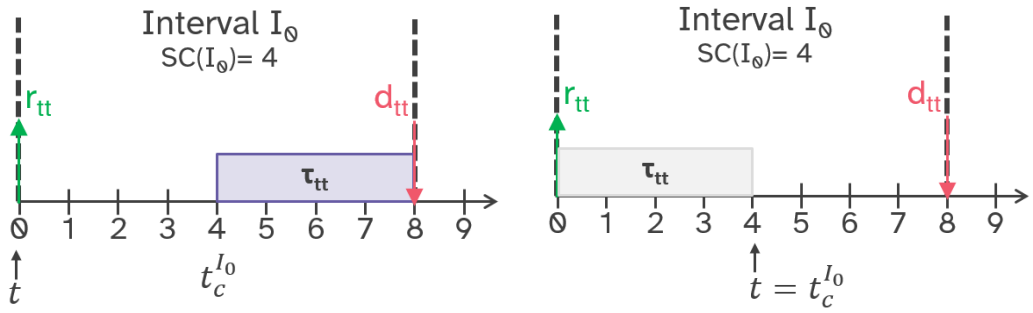
tasks introduces pessimism into the test. In particular, the test can produce false negatives by incorrectly classifying some schedulable task sets as unschedulable. However, this assumption does not lead to false positives, i.e., any task set deemed schedulable by the test is indeed schedulable. In Table 2, we list simple modifications to make the test exact.

**Detailed Description and Example.** In [5], Isovic and Fohler define, for each capacity interval $I_i$, a critical time slot $t_c^{I_i}$. They prove that if the sporadic task set is feasible when its tasks arrive synchronously at $t_c^{I_i}$, then it is also feasible when its tasks arrive synchronously at every other time slot $t_k$ located within the same interval $I_i$. Isovic and Fohler denote the start of capacity interval $I_i$ by $start(I_i)$ and the spare capacity within it by $sc(I_i)$. They calculate the critical slot of interval $I_i$ as follows: $t_c^{I_i} = start(I_i) + sc(I_i)$. In other words, since spare capacities in Slot-Shifting are calculated based on an ALAP execution of TT tasks in order to service aperiodic tasks ASAP, a critical slot of a capacity interval is the first time slot, where the spare capacity of the interval is completely used or wasted.

Let us consider the simple example task set system in Table 1, showing the parameters of a single TT task $\tau_{tt}$ after performing Slot-Shifting's offline phase (which resolves its complex constraints and transforms it into an independent task) and a single sporadic task $\tau_{sp}$ with which we seek to extend the task set. Figure 1a shows the schedule for $\tau_{tt}$ with ALAP execution, annotated with its single spare capacity interval $I_0$ and its spare capacity $sc(I_0) = 4$. The critical slot for this interval is $t_c^{I_0} = 4$.

**Table 1** Example task set, showing the parameters of a single TT task $\tau_{tt}$ after performing Slot-Shifting's offline phase and a single sporadic task $\tau_{sp}$.

| Task | Type | Offset $\phi$ | WCET $C$ | Deadline $D$ | Period/MIT $T$ |
|---|---|---|---|---|---|
| $\tau_{tt}$ | Periodic | 0 | 4 | 8 | 8 |
| $\tau_{sp}$ | Sporadic | - | 1 | 4 | 4 |




**(a)** ALAP Schedule following execution until $t = 0$. **(b)** ALAP Schedule following execution until $t = 4$.

**Figure 1** Schedule for task $\tau_{tt}$ from Table 1, illustrating ALAP execution relative to time slot $t$, under the assumption that tasks are executed ASAP prior to $t$. For both cases, the capacity interval $I_0$ is indicated and annotated with its spare capacity $sc(I_0)$.

Each sporadic task instance can be guaranteed depending on the availability of spare capacities within its execution window. If we let the sporadic task $\tau_{sp}$ arrive at the first critical slot, i.e., at $t = 4$, and attempt to guarantee it within the ALAP schedule in Figure 1a, we find that the first instance of $\tau_{sp}$ has no available free slots in its execution window interval $[4, 8]$. Therefore, it cannot be guaranteed according to the test in [5].

However, since Slot-Shifting's runtime scheduler executes tasks ASAP at each time slot, the corresponding ALAP schedule at $t_c^{I_0} = 4$ is shown in Figure 1b. By then, the task $\tau_{tt}$ has already finished executing, and all free slots have been relocated to the interval $[4, 8]$. Hence, the first instance of $\tau_{sp}$ can be guaranteed in the interval $[4, 8]$; there are 4 free-slots, and it needs only one. In fact, the task set in Table 1 follows the implicit deadline model, where the necessary and sufficient condition for EDF-schedulability is that the total utilization of the task set must be less than or equal to one [2]. Since this condition is satisfied, the task set is schedulable.

Therefore, the test in [5] is pessimistic. The ASAP execution of TT tasks, shifts and does not consume free time slots within capacity intervals. Removing this pessimism requires recalculating the spare capacity values of all intervals based on the ASAP execution of TT tasks prior to the candidate slot for the global arrival of sporadic tasks. A possible approach is by simulating Slot-Shifting's online scheduling behavior on TT tasks up to the critical slot, thereby ensuring that spare capacities are updated accordingly across the entire schedule. In Table 2, we list simple modifications to make the test exact.

## 3.3 Addressing the Analysis Interval

In [5], Isovic and Fohler perform the test relative to each critical slot for the duration of the hyperperiod of sporadic tasks $\mathcal{H}_S$, i.e., the least common multiple (LCM) of their periods. However, this is the proven analysis interval without the co-presence of TT periodic tasks [2]. Isovic and Fohler do not provide a formal proof for their proposed analysis interval and do not specify the deadline model assumed for sporadic tasks. Here, we propose adopting formally proven bounds within scheduling theory and have been shown to hold under ASAP scheduling: TT tasks are inherently periodic and may have offsets and complex timing constraints, such as end-to-end deadlines and mutual exclusion. As explained in Section 2, during the offline phase of Slot-Shifting, an offline scheduler resolves these constraints, effectively converting TT tasks into independent ones (see [4] for details). This transformation allows reusing established and safe schedulability bounds for periodic tasks with offsets [1, 2, 6] to modify the analysis interval of the offline acceptance test.

Let $\phi_{max}$ denote the largest offset among all periodic tasks, $\mathcal{H}_{TT}$ be their hyperperiod, $\Gamma$ be the combined set of time-triggered and sporadic tasks, and $\mathcal{C}_i$, $T_i$, and $D_i$ be the worst-case execution time, period, and relative deadline of task $\tau_i$, respectively. It was shown in [1] that it is sufficient to analyze schedulability over all possible release times $t_1$ and deadlines $t_2$, subject to the inequalities:

$$\phi_{max} \leq t_1 < \phi_{max} + \mathcal{H}_{TT} \tag{1}$$

$$t_1 < t_2 < t_1 + B_{TTS} \tag{2}$$

$$B_{TTS} = \frac{\sum_{\tau_i \in \Gamma} (T_i - D_i) \frac{\mathcal{C}_i}{T_i}}{1 - \sum_{\tau_i \in \Gamma} \frac{\mathcal{C}_i}{T_i}} \tag{3}$$

Since the bounds on $t_1$ and $t_2$ are established under the assumption of constrained deadlines, we adopt the same deadline model and adapt the terms to align with those used in Slot-Shifting's offline test as follows:

1. Set $t_1$ to the critical slot $t_c$, marking the beginning of an analysis iteration.
2. Omit $t_2$ as an explicit variable and have the test evaluate all capacity intervals starting from $t_c$ up to $t_1 + B_{TTS}$

With this modification, the acceptance test is expected to run longer, which is acceptable for offline use.

### 3.4 Properties Summery

We summarize the properties of the offline acceptance test, including its modified analysis interval, in Table 2. In addition, we outline minor adjustments that align the test with the exact formulation presented in [1], specifically by incorporating the ASAP scheduling of TT tasks prior to the synchronous release of sporadic tasks at the candidate global offset slot. Experimental results in [1] show that the exact test achieves comparable runtime performance to the original sufficient test, while offering substantial improvements in schedulability.

**Table 2** Properties of the offline acceptance test and simple modifications to the test to account for the ASAP scheduling of TT tasks.

| | **Acceptance test in [5]** | **Modifications** |
|---|---|---|
| Candidate slots as global offset of sporadic tasks | Critical slot at every interval $I$, calculated as $start(I) + sc(I)$ | Every slot that coincides with the release of a TT task in the schedule |
| During the test, available spare capacity is calculated | Once, based on ALAP execution of TT tasks | Once for every candidate slot, based on ASAP execution of TT tasks prior to the candidate slot |
| Analysis interval | Subject to the inequalities 1, 2 and 3 | |
| Test type | Sufficient | Exact |
| Runtime performance | Comparable (see [1]) | |

## 4 Conclusion

In this technical note, we identified pessimism in the offline test of Slot-Shifting for extending a feasible time-triggered task sets with a set of sporadic tasks, clarified assumptions for its use, and addressed the interval considered for schedulability analysis.

## References


1 Mohammad Ibrahim Alkoudsi, Sanjoy Baruah, Pontus Ekberg, Gerhard Fohler, Joël Goossens, and Maryam Moslehi. Extending periodic task sets with sporadic tasks: Computational complexity and exact feasibility tests. In *Proceedings of the 33rd International Conference on Real-Time Networks and Systems (RTNS '25)*, Pisa, Italy, 2025.

2 Sanjoy K. Baruah, Louis E. Rosier, and Rodney R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, 2, 1990. doi:10.1007/BF01995675.

3 Anaïs Finzi, Silviu S. Craciunas, and Marc Boyer. Integrating sporadic events in time-triggered systems via affine envelope approximations. In *2024 IEEE 30th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 15–28, 2024. doi:10.1109/RTAS61025.2024.00010.

4 G. Fohler. Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. In *Proceedings 16th IEEE Real-Time Systems Symposium*, pages 152–161, 1995. doi:10.1109/REAL.1995.495205.

5 Damir Isovic and Gerhard Fohler. Handling mixed sets of tasks in combined offline and online scheduled real-time systems. *Real-Time Syst.*, 43(3):296–325, November 2009. doi:10.1007/s11241-009-9088-3.

6 Joseph Y.-T. Leung and M.L. Merrill. A note on preemptive scheduling of periodic, real-time tasks. *Information Processing Letters*, 11(3):115–118, 1980. doi:10.1016/0020-0190(80)90123-4.