Automated Temporal Verification for Preemptive Asynchronous Programs

Anonymous
Null

Abstract

To make reactive programming more concise and expressive, it is promising to combine two approaches to concurrency that integrates *synchronous preemption* with *asynchronous promises*. Existing temporal verification techniques have not been designed to handle such a marriage of two execution models. This work presents a solution that integrates a modular Hoare-style forward verifier with a novel term rewriting system (TRS) on *(A)Synchronous Effects* (*ASyncEffs*). We use the full-featured Esterel as our target language, generalizing the preemptive asynchronous abstraction. We propose *ASyncEffs*, a new effect logic that extends *Synchronous Kleene Algebra* with a *waiting* operator. We establish an effect system for preemptive and asynchronous primitives. Lastly, we present a purely algebraic TRS to efficiently check language inclusions between *ASyncEffs*. We prototype the verification system, prove its correctness, and report case studies and experimental results.

2012 ACM Subject Classification Theory of computation → Semantics and reasoning; Security and privacy → Logic and verification

Keywords and phrases Synchronous Preemption, Asynchronous Promises, Hoare-style Temporal Verification,Term Rewriting System

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

Synchronous programming [1] has found success in many safety-critical applications, such as fly-by-wire systems and nuclear power plant control software[1]. It exhibits a high concurrency but calls for deterministic and predictable execution, which has been considered a clean formalism for modeling, specifying, validating, and implementing reactive systems. Languages based on this paradigm – such as Esterel [3], Lustre [4] and Signal [5] – assume that time is partitioned into discrete instants (or reactions) and the computation/communication for processing all events that occur within one time instant happen instantaneously.

Many mainstream languages, such as C#, Java, JavaScript, and Python, have recently added support for asynchronous promises, also known as *futures* or *tasks* [6]. These features support basic asynchronous operators, such as *yield* pauses/resumes generator functions asynchronously; *async/await* simplify the blending of asynchronous executions into sequential programming. However, most these languages offer a small set of preemption primitives, often inadequate for concisely modeling interruptions or control-driven computations.

To make reactive programming more concise and expressive, recent innovations are dedicated to integrating synchronous features to asynchronous infrastructures. For example: the language HipHop.js [7] is a mixture of JavaScript and Esterel for reactive web applications, which facilitates JavaScript with preemptions like *every* and *abort*; the Scala library ZIO [8] is for type-safe asynchronous and concurrent programming with rudimentary preemptive operators, such as *fiber interruption* and *racing*; similarly, Microsoft's durable function [9]

[1] Concretely, it has been used in the creation and verification of fuel control systems; landing gear control functions; virtual display systems at Dassault Aviation [2]; the control software of the N4 nuclear power plants; the Airbus A320 fly-by-wire system; and the specification of part of Texas Instrument's digital signal processors [1].

deploys blended asynchrony/synchrony, including pause, parallel composition for deterministic orchestration functions, which is available as libraries for C#, JavaScript, Python, etc.

There is a growing need to reason about such asynchronous reactive programs with multifarious preemptions. In particular, we are interested in the techniques for specifying and verifying temporal behaviors of such a mixed execution model, which has not been extensively studied. Therefore, this paper studies the challenges of synchronous preemptions and asynchronous promises and attempts to provide a practical solution accordingly.

```
1 module Main (in login, inout connected, out connState){
2  par{Identity(...)}  // enables the login button
3     {every(login) { // implements a preemptive loop
4         Authenticate(...);  // preempts the previous Session
5         present(connected){Session(...)}   // then branch
6                                {emit connState("err")}}}}// else ...
```

Figure 1 A preemptive program written in Esterel, for a simple web login procedure, from [7].

The power of preemption appears in Figure 1. Module `Main` makes use of three submodules: `Identity` reads the GUI and enables the login button when the input username and passwords are both longer than two characters; `Authenticate` calls the authorization service and output the signal `connected` when authorized; `Session` establishes an active communication session between the authorized user and the server.

Statement `par{...}{...}` (in lines 2 and 3) runs branches in parallel. The signal `login` is *present* when the login button is pressed in the first thread. Then the presence of `login` makes the `every` statement restart the sequence of tasks (in lines 4-6), so the current session is preempted and `Authenticate` begins execution. When `Authenticate` terminates, the status of `connected` is tested (in line 5). If *present*, `Session` starts running a new session until the next time the login button is pressed. When `Session` terminates, the `every(login){...}` statement merely waits for a new login. If after `Authenticate`, the status of `connected` is *absent*, then the output signal `connState` is emitted with an error message.

Although simple, Figure 1 shows that preemption statements allow a clean, hierarchical description of temporal behaviors. While flexible and expressive, preemption primitives have fairly complex semantics, which in turn, makes reasoning difficult. In this paper, we study the subtle operational semantics of various preemptions, including: *interrupt*; *abort*; *suspend*; *every*; and the *label-based escape*. Furthermore, we show that our approach supports a comprehensive foundation for verifying preemptions with different keywords, including *strong* or *weak*, *immediate* or *delayed*.

With asynchronous promises, we can perform long-lasting tasks without blocking the main thread. The keywords *async* and *await* allow sequential-style code to capture concurrent executions with explicit dependencies via asynchronous signals succinctly. However, promises are complex and error-prone in their own right. Prior works [10, 11] display a set of broken promises chain *anti-patterns* shown in asynchronous JavaScript, and propose to detect the anti-patterns by constructing a promise dependency graph. In this paper, we focus on the *async/await* related anti-pattern, where *unreachable promises* may have registered reactions that will never be executed. Different from prior works, we show that our purely algebraic approach detects such unreachable promises without any constructions of graphs.

This work achieves a modular verification - where modules can be replaced by their already verified properties - for preemptive and asynchronous programs. We propose to use

ASyncEffs to be the temporal specification language and deploy a compositional verification strategy via a forward verifier and a term rewriting system (TRS). Specifically, *ASyncEffs* enrich the Synchronous Kleene Algebra (SKA) [12, 13] with a new operator to add the abstraction for *"block waiting across threads"* into classic linear temporal verification.

This work is a significant extension of [14], which solely targeted the temporal verification for a subset of Esterel without preemptions or asynchronous primitives. This work inherits the verification framework proposed in [14] but tackles challenges for the full-featured Esterel. Our main contributions are:

1. **Language Abstraction:** we formally define the operational semantics for the full-featured Esterel and use it to generalize the preemptive asynchronous execution model.
2. **Specification Logic:** we propose *ASyncEffs*, by defining its syntax and semantics. We show that *ASyncEffs* subsume the expressiveness power of classic linear temporal logics.
3. **Forward Verifier:** we establish a Hoare-style forward verifier, which is an effect system (or axiomatic semantics per se) to reason the temporal behaviors of target programs. It deploys a TRS to check the actual behaviors against their annotated specifications.
4. **An Efficient TRS:** we present rewriting rules to prove/disprove the entailments between *ASyncEffs*. The TRS is a back-end solver deployed by the front-end forward verifier.
5. **Implementation and Evaluation:** we prototype our proposal, validate our implementation, prove the correctness, and report on case studies and experimental results.

2 Overview

An overview of our automated verification system is given in Figure 2. The system consists of a forward verifier and a TRS, shown in the rounded boxes. The inputs of the forward verifier are target programs annotated with temporal specifications written in *ASyncEffs*. The inputs of the TRS are pairs of effects, LHS and RHS, referring to the inclusion LHS $\sqsubseteq$ RHS[2] to be checked *(LHS and RHS refer to left-hand-side effects and right-hand-side effects respectively)*. The forward verifier calls the TRS to solve trace-based proof obligations for assertions. The TRS can also be used as a solver independently in its rights. We now highlight our primary methodology using examples.

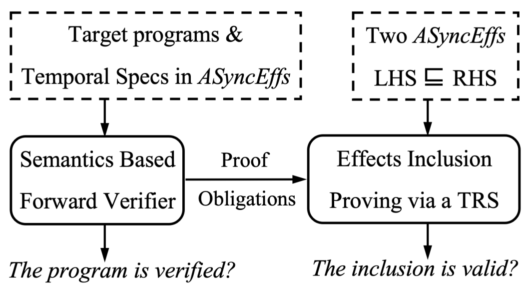

Figure 2 System Overview.

2.1 Target Programs and *ASyncEffs*

Specifications are annotated in `/*@...@*/` for each module, which leads to a compositional verification strategy, where temporal verification can be done locally. *ASyncEffs* use curly braces {} to enclose each logical-time instant (reaction). A time instant is a set of signals with status happening conceptually simultaneously.

As shown in Figure 3, module `Read` asynchronously loads and processes a JSON file. In line 4, the statement `async` is enriched with a completion signal, here `loaded`. When started,

---

[2] We formally define the *ASyncEffs* inclusion relation $\sqsubseteq$ in Definition 4.

async immediately emits loading, and calls fs.readFile, that is expected to take time in terms of reactions, i.e., not to complete during the current reaction. Statement async forks a new thread to execute its body and thus does not block the main execution. Therefore in line 7, the program can do other computations (i.e., compOther) while loading the file. In line 8, the program waits for the signal loaded to be emitted, i.e., to synchronize with the child thread spawned by async. Then, it does data processing by emitting logData and waits for the environment to close the file.

```
1  module Read (in open, close, out loading, loaded, compOther, logData)
2  /*@ requires {}^*.{open} @*/
3  /*@ ensures {loading,compOther}.{loaded}.{logData}.close? @*/ {
4    async loaded {  //"loaded" is emitted after the body is completed
5        emit loading;
6        fs.readFile(open.value);}
7    emit compOther;  //do things that do not depend on the loading file
8    await loaded;    //block waiting for the signal "loaded" to be emitted
9    emit logData;  //data processing and logging
10   await close; }
11
12 module Main (out open, close, loading, loaded, compOther, logData)
13 /*@ requires {} @*/
14 /*@ ensures {open}.{}^*.{close} @*/ {
15   emit open("filePath");
16   par { Read (open, close, loading, loaded, compOther, logData); }
17       { await logData;    //await for the signal "logData" to be emitted
18          emit close("filePath"); }}    //close the file
```

Figure 3 Asynchronously reading a file, using async/await in Esterel.

Module Read's precondition $\{\}^\star \cdot \{\texttt{open}\}$ requires that when Read is called, the signal open should be emitted in the latest reaction, indicating that the file is opened before the current method call[3]. Module Main firstly opens the file then creates two child threads via a par statement. The first thread calls Read, while another thread waits for the data processing to be done and closes the file. Note that *ASyncEffs* is an *affine logic* that only describes the signals we are concerned about, regardless of the non-mentioned signals.

2.2 Forward Verification

To reason about program implementations, we deploy a Hoare-style forward verifier. Given a statement p and the current program state $\langle\Phi\rangle$, we compute p's effects by transforming the program states, based on the forward rules defined in Sec. 4. We use Φ to denote *ASyncEffs* formulae (defined in Figure 8). Next, we use Figure 4 to demonstrate the verification for module Read. To facilitate this illustration, we mark the deployed forward rules in [gray].

At the beginning of the verification, state (1) is initialized with an empty instant. States (2)(4)(6) are obtained by the rule [FV-Emit], which adds the emitted signal to the latest

[3] We support parameterized signals, so this abstraction works for manipulating multiple files as well.

```
module Read (in open, close, out loading, loaded, compOther, logData){
```

(1) $\langle\{\}\rangle$ *(– initialize the current effects using an empty instant –)*

```
  async loaded {
      emit loading; fs.readFile(open.value);
```

(2) $\langle\{loading\}\rangle$ [FV-Emit]

`}` $\langle\{loading\}\cdot\{loaded\}\rangle$ [FV-Async]

(3) $\langle\{\}\rangle$ *(– inherited from state (1), because statement async is non-blocking –)*

```
  emit compOther;
```

(4) $\langle\{compOther\}\rangle$ [FV-Emit]

```
  await loaded;
```

(5) $\langle\{compOther\}\cdot loaded?\rangle$ [FV-Await]

```
  emit logData;
```

(6) $\langle\{compOther\}\cdot loaded?\cdot\{logData\}\rangle$ [FV-Emit]

```
  await close; }
```

(7) $\langle\{compOther\}\cdot loaded?\cdot\{logData\}\cdot close?\rangle$ [FV-Await]

(8) $\langle(\{loading\}\cdot\{loaded\})\ ||\ (\{compOther\}\cdot loaded?\cdot\{logData\}\cdot close?)\rangle$ [FV-Async]

$\Phi_{final}{=}\langle\{loading, compOther\}\cdot\{loaded\}\cdot\{logData\}\cdot close?\rangle$ [Effects-Parallel-Merge]

(9) $\Phi_{final} \sqsubseteq \Phi^{Read}_{post}$ [FV-Decl] *(-TRS: check the postcondition of Read; Succeed. -)*

Figure 4 A demonstration of the forward verification for the module `Read`.

instant. States (5)(7) are obtained by the rule [FV-Await], which concatenates a blocking signal (with a question mark) to the current effects. Since the `async` statement is non-blocking and forks a new thread, state (3) is the same as the state (1). Step (8) parallel composes the effects from the `async` thread and the main thread and normalizes the final effects. After these state transformations, step (9) invokes the TRS to check the postcondition. Besides, before each function call, the verifier invokes the TRS to check whether the current effect state satisfies the precondition of the callee, cf. [FV-Call].

Case Study I: Detecting Unreachable Promises.

Having the actual program behavior expressed in *ASyncEffs*, and the parallel merge algorithm (defined in Sec. 4.1) to eliminate the waiting operators as much as possible based on the local environment, we can easily capture the *unreachable promises* via a lexical checking for *ASyncEffs*. For example, in step (8), the final trace contains a dangling waiting for the signal `close` because there is no locally emitted `close`.

▶ **Definition 1** (Well-Synchronized Effects). *After parallel merging, we call effects without any waiting operators well-synchronized effects. Given any effect Φ, $well(\Phi)$ returns a Boolean value, defined recursively:*

$$well(\bot){=}well(\epsilon){=}well(I) = false \qquad well(\mathbb{S}?) = true \qquad well(\Phi^{\star}) = well(\Phi)$$
$$well(\Phi_1\cdot\Phi_2){=}well(\Phi_1\vee\Phi_2){=}well(\Phi_1||\Phi_2) = well(\Phi_1)\vee well(\Phi_2)$$

Definition 1 defines *well-synchronized effects*. Any not *well-synchronized effects* indicates the existence of registered reactions for unreachable promises. However, not well-synchronized effects can be further parallel composed to other threads, i.e., a bigger context, and become well-synchronized. For example, the final effects of module `Main` is well-synchronized after parallel composing module `Read` with another thread, shown in Appendix A at step (16).

2.3 The TRS

Having *ASyncEffs* to be the specification language, we are interested in the following verification problem: *Given a program $\mathcal{P}$, and a temporal property Φ', does $\Phi^{\mathcal{P}} \sqsubseteq \Phi'$ hold? In a typical verification context, checking the inclusion between the actual program effects $\Phi^{\mathcal{P}}$ and the valid traces Φ' proves that: the program $\mathcal{P}$ will never lead to unsafe traces which violate Φ'.*

We deploy a purely algebraic term rewriting system (TRS), to check language inclusions between *ASyncEffs*. Our TRS is inspired by Antimirov and Mosses's algorithm [15], whose rewriting system decides inequalities of regular expressions (REs) through an iterated process of checking the inequalities of their *partial derivatives* [16], as defined in Definition 2. There are two basic rules: [*Disprove*], which infers false from trivially inconsistent inequalities; and [*Unfold*], which applies Definition 3 to recursively generate new inequalities.

▶ **Definition 2** (Partial Derivatives). *Given any formal language L over an alphabet Σ and any alphabet $a \in \Sigma$, the partial derivative of L with respect to a is defined as:*

$$a^{\text{-}1}L = \{w \in \Sigma^* \mid aw \in L\}.$$

▶ **Definition 3** (REs Inequality). *For two REs r and s, $r \preceq s \Leftrightarrow \forall(\mathtt{A} \in \Sigma).\ \mathtt{A}^{\text{-}1}(r) \preceq \mathtt{A}^{\text{-}1}(s)$.*

▶ **Definition 4** (*ASyncEffs* Inclusion). *For two* ASyncEffs *Φ_1, Φ_2,*

$$\Phi_1 \sqsubseteq \Phi_2 \Leftrightarrow \forall I.\ I^{\text{-}1}(\Phi_1) \sqsubseteq I^{\text{-}1}(\Phi_2).$$

Similarly, we defined Definition 4 for *ASyncEffs*' inclusion relation, where $I^{\text{-}1}(\Phi)$ is the partial derivative of Φ w.r.t the instant I. Termination of the rewriting is guaranteed because the sets of alphabets and derivatives to be considered are finite, and possible cycles are detected using *memorization* [17].

We use Table 1 to illustrate the rewriting system, which proves that $\{\mathtt{A}\} \cdot \{\mathtt{C}\} \cdot \mathtt{B}? \cdot \{\mathtt{D}\}$ entails $\{\mathtt{A}\} \cdot \mathtt{B}? \cdot \{\mathtt{D}\}$. We define "waiting for the signal $\mathtt{B}$" as: $\mathtt{B}? \equiv \exists n, n{\geq}0 \wedge \{\overline{\mathtt{B}}\}^n \cdot \{\mathtt{B}\}$, where $\{\overline{\mathtt{B}}\}$ refers to the instants containing $\mathtt{B}$ to be absent. Therefore intuitively $\{\mathtt{C}\} \cdot \mathtt{B}?$ is a special case of $\mathtt{B}?$.

Table 1 A demonstration of rewriting *ASyncEffs* inclusions with waiting operators.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\epsilon \sqsubseteq \epsilon}{\cancel{\{D\}} \sqsubseteq \cancel{\{D\}} \vee \bot}\text{⑤[Prove]}}{\cancel{\{B\}} \cdot \{D\} \sqsubseteq (\cancel{\{B\}} \cdot \{D\}) \vee (\cancel{\{\overline{B}\}} \cdot B? \cdot \{D\})}\text{④} \qquad \dfrac{\dfrac{B? \cdot \{D\} \sqsubseteq B? \cdot \{D\}\ (\ddagger)}{B? \cdot \{D\} \sqsubseteq \bot \vee (B? \cdot \{D\})}\text{⑦[Reoccur]}}{\cancel{\{\overline{B}\}} \cdot B? \cdot \{D\} \sqsubseteq (\cancel{\{B\}} \cdot \{D\}) \vee (\cancel{\{\overline{B}\}} \cdot B? \cdot \{D\})}\text{⑥}}{B? \cdot \{D\} \sqsubseteq \bot \vee (B? \cdot \{D\})\ (\ddagger)}\text{③}}{\cancel{\{C\}} \cdot B? \cdot \{D\} \sqsubseteq (\cancel{\{B\}} \cdot \{D\}) \vee (\cancel{\{\overline{B}\}} \cdot B? \cdot \{D\})}\text{②[Unfold]}}{\cancel{\{A\}} \cdot \{C\} \cdot B? \cdot \{D\} \sqsubseteq \cancel{\{A\}} \cdot B? \cdot \{D\}}\text{①[Unfold]}$$

Steps ① ② ④ ⑥ unfolds the inclusions by eliminating the first instants from both sides. Step ③ observes a disjunction on the left-hand side (because $\mathtt{B}? \cdot \{\mathtt{D}\}$ is a shorthand for $(\{\mathtt{B}\} \cdot \{\mathtt{D}\}) \vee (\{\overline{\mathtt{B}}\} \cdot \mathtt{B}? \cdot \{\mathtt{D}\})$), thus creates two sub-trees, and the original inclusion is proved only when both branches succeed. The first sub-tree is proved at step ⑤ by the *frame* rule. The second sub-tree is proved by the *reoccur* rule, at step ⑦ where we observe the proposition is isomorphic with one of the previous step (before ③), marked with (‡).

3 Language and Specifications

3.1 The Target Language

Prior works [18, 14] on program analysis for Esterel tend to focus on the *perfect synchrony*[4] perspective in Esterel, without dedicated reasoning for asynchronous concurrency or preemptions. We argue that our work cannot be subsumed by prior works, because of the additional modeling for *block waiting*, which is essential for both asynchronous concurrency and preemptions. We summarize a full-featured Esterel in Figure 5 to be our target language, which provides the infrastructure for the preemptive asynchronous abstraction. The statements marked as purple are generalized from the preemptive statements in (reactive) synchronous programming, while the statements marked as blue provide the *async/await* constructs for asynchronous programming.

$$
\begin{array}{llcl}
(\mathit{Program}) & \mathcal{P} & ::= & \overrightarrow{meth} \\
(\mathit{Signal\ Types}) & \tau & ::= & in \mid out \mid inout \\
(\mathit{Module\ Def.}) & meth & ::= & mn\ \ (\overrightarrow{\tau\ S(x)})\ \ \langle \mathbf{req}\ \ \Phi_{pre}\ \ \mathbf{ens}\ \ \Phi_{post}\rangle\ \ p \\
(\mathit{Values}) & v & ::= & () \mid i \mid b \mid x \\
(\mathit{Parametrized\ Signal}) & \mathbb{S} & ::= & S(v) \\
(\mathit{Statements}) & p,q & ::= & v \mid yield \mid emit\ \mathbb{S} \mid p;q \mid p||q \mid call\ mn\ (\overrightarrow{\mathbb{S}}) \\
 & & \mid & loop\ p \mid present\ S\ then\ p\ else\ q \mid async\ S\ p\ q \mid await\ [\kappa_1]\ S \\
 & & \mid & trap\ p \mid exit[\kappa_2]\ \ d \mid [\kappa_2]\ abort\ p\ S \mid [\kappa_2]\ suspend\ p\ S \\
(\mathit{Preemption\ Keywords}) & & & \kappa_1 ::= immediate \mid delayed \qquad \kappa_2 ::= weak \mid strong
\end{array}
$$

$$
(\mathit{Signal\ Variables}) S \in \Sigma \qquad i \in \mathbb{Z} \qquad b \in \mathbb{B} \qquad mn, x \in \mathbf{var} \qquad (\mathit{Depth}) d \in \mathbb{N} \cup \{0\}
$$

Figure 5 Syntax of the target language.

Here, S, x are meta-variables ranging over signal variables and constants. Signal types are: *in* for input signals, *out* for output signals and *inout* for both. **var** represents the countably infinite set of arbitrary distinct identifiers. A program $\mathcal{P}$ comprises a list of module definitions $\overrightarrow{meth}$[5]. Each *module* has a name mn, a list of well-typed arguments $\overrightarrow{\tau\ S(x)}$, a statement-oriented body p, associated with a precondition Φ_{pre} and a postcondition Φ_{post}. We here present the intuitive semantics of the basic statements.

A thread of execution suspends itself for the current instant using the *yield* construct and resumes when the next instant started[6]. Statement *emit* $\mathbb{S}$ broadcasts the signal $\mathbb{S}$ to be *present*. The emission of $\mathbb{S}$ is valid for the current instant only. The sequence statement $p; q$ starts p and instantaneously passes the control flow to q when p terminates. Statement q is never started if p always yields. Parallel statement $p||q$ runs p and q in parallel. The branches can terminate in different instants, and the parallel statement waits for the last one to terminate. Statement *call* mn $(\overrightarrow{\mathbb{S}})$ is a call to module mn, providing the list of IO signals. Statement *present* $\mathbb{S}$ p q immediately starts p if S is present in the latest instant; otherwise it starts q instead. Statement *loop* p implements an infinite loop of executing p.

[4] Perfect synchrony is a high-level language abstraction where all reactions of a system are executed in (conceptually) zero time. Hence, outputs are generated simultaneously when the inputs are read.
[5] Here, we use the $\rightarrow$ script to denote a finite vector (possibly empty) of items.
[6] For a better *cooperative multitasking* [19], processes voluntarily yield control periodically or when idle or logically blocked.

Keywords *immediate* and *delayed* are for *await* statements, indicating to wait for a signal from the latest instant or the next instant, respectively. Keywords *weak* or *strong* are for preemptive statements, indicating to allow or not allow, respectively, the latest instant to execute when the preemption condition is met. In this work, we use the most commonly used *immediate* waiting and *weak* preemptions by default, with discussions on how to cooperate with the *delayed* waiting and *strong* preemptions in detail.

3.2 Structural Operational Semantics of the Target Language

Figure 6 provides the operational semantics of the preemptive and promise-related statements and leaves the rest standard semantics rules in Appendix B.

The reduction rules are in the form of $p \xrightarrow[\mathcal{E}]{\alpha,k} p'$, meaning that a process p performs an action α then becomes a process p' with a completion code k. $\mathcal{E}$ stands for all the signals produced at the instant by the whole program of which p is part, which gives the global information about the presence and absence of signals. In particular, $\alpha \subseteq \mathcal{E}$. The *completion code* k is a non-negative integer: when $k{=}0$, the reduction completes without exits or yields; when $k{=}1$, the reduction completes without exits but with a yield, i.e., starting a new instant; when $k{=}2$, the reduction completes with an exit that escapes the nearest trap; when $k{>}2$, the reduction completes with an exit which escapes a further enclosing *trap*. Such an encoding for preemptions was first advocated by Gonthier in [20].

Statement $async\ \mathbb{S}\ p\ q$ is a *syntactic sugar* which spawns a long-lasting background computation for p, which will join back to the main thread later. It essentially performs p and q in parallel, and emits $\mathbb{S}$ when p completes, i.e., $(p; yield; emit\ \ \mathbb{S})||q$. Statement $await\ \mathbb{S}$ blocks the local thread and waits for $\mathbb{S}$ to be emitted in the environment.

$$\frac{(\mathbb{S} \mapsto present) \in \mathcal{E}}{await\ \mathbb{S} \xrightarrow[\mathcal{E}]{\emptyset,1} ()}[\texttt{Await-1}] \qquad \frac{(\mathbb{S} \mapsto present) \notin \mathcal{E}}{await\ \mathbb{S} \xrightarrow[\mathcal{E}]{\emptyset,1} await\ \mathbb{S}}[\texttt{Await-2}] \qquad \frac{}{exit\ \ d \xrightarrow[\mathcal{E}]{\emptyset,d+2} ()}[\texttt{Exit}]$$

$$\frac{p \xrightarrow[\mathcal{E}]{\alpha,k} p' \quad (k{\leq}1)}{trap\ \ p \xrightarrow[\mathcal{E}]{\alpha,k} trap\ \ p'}[\texttt{Trap-1}] \qquad \frac{p \xrightarrow[\mathcal{E}]{\alpha,k} p' \quad (k{=}2)}{trap\ \ p \xrightarrow[\mathcal{E}]{\alpha,0} ()}[\texttt{Trap-2}] \qquad \frac{p \xrightarrow[\mathcal{E}]{\alpha,k} p' \quad (k{>}2)}{trap\ \ p \xrightarrow[\mathcal{E}]{\alpha,k\text{-}1} p'}[\texttt{Trap-3}]$$

$$\frac{(\mathbb{S}{\mapsto}present) \in \mathcal{E}}{abort\ \ p\ \mathbb{S} \xrightarrow[\mathcal{E}]{\emptyset,0} ()}[\texttt{Abort-1}] \qquad \frac{(\mathbb{S}{\mapsto}present) \notin \mathcal{E} \quad p \xrightarrow[\mathcal{E}]{\alpha,k} p'}{abort\ \ p\ \mathbb{S} \xrightarrow[\mathcal{E}]{\alpha,k} abort\ \ p'\ \mathbb{S}}[\texttt{Abort-2}] \qquad \overset{[\texttt{Abort-3}]}{\frac{}{abort\ \ ()\ \mathbb{S} \xrightarrow[\mathcal{E}]{\emptyset,0} ()}}$$

$$\overset{[\texttt{Suspend-1}]}{\frac{(\mathbb{S}{\mapsto}present) \in \mathcal{E}}{suspend\ \ p\ \mathbb{S} \xrightarrow[\mathcal{E}]{\emptyset,1} suspend\ \ p\ \mathbb{S}}} \qquad \overset{[\texttt{Suspend-2}]}{\frac{(\mathbb{S}{\mapsto}present) \notin \mathcal{E} \quad p \xrightarrow[\mathcal{E}]{\alpha,k} p'}{suspend\ \ p\ \mathbb{S} \xrightarrow[\mathcal{E}]{\alpha,k} suspend\ \ p'\ \mathbb{S}}} \qquad \overset{[\texttt{Suspend-3}]}{\frac{}{suspend\ \ ()\ \mathbb{S} \xrightarrow[\mathcal{E}]{\emptyset,0} ()}}$$

$$\overset{[\texttt{Par-Base-0}]}{\frac{p \xrightarrow[\mathcal{E}]{\alpha,0} p'}{p||q \xrightarrow[\mathcal{E}]{\alpha,0} p'||q}} \qquad \overset{[\texttt{Par-Base-1}]}{\frac{p \xrightarrow[\mathcal{E}]{\alpha_1,1} p' \quad q \xrightarrow[\mathcal{E}]{\alpha_2,1} q'}{p||q \xrightarrow[\mathcal{E}]{\alpha_1 \cup \alpha_2,1} p'||q'}} \qquad \overset{[\texttt{Par-Preemption}]}{\frac{p \xrightarrow[\mathcal{E}]{\alpha_1,k_1} p' \quad q \xrightarrow[\mathcal{E}]{\alpha_2,k_2} q' \quad (max(k_1,k_2){>}1)}{p||q \xrightarrow[\mathcal{E}]{\alpha_1 \cup \alpha_2,max(k_1,k_2)} ()}}$$

Figure 6 Operational semantics of promise-related and preemptive statements in Esterel.

We omit the labeling of the traps but use the exact depth value to refer to the nested level of *trap* statements. Statement *exit d* instantaneously exits the trap with depth d. For example, when $d{=}0$, the execution exits the nearest *trap*; when $d{=}1$, the execution exits one outer layer of *trap*; and so on. Statement *trap p* installs a trap and behaves like p until any *exit* occurs. Statement *abort p* $\mathbb{S}$ performs p and terminates when $\mathbb{S}$ occurs. Statement *suspend p* $\mathbb{S}$ suspends p for one instant whenever $\mathbb{S}$ is present in the environment and resumes p from the successive instants.

The rules for parallel statements execute the branches independently, then merge their output events accordingly. If one branch exits with code k, then both threads are preempted with the exception depth k. If both statements exit distinct traps with k_1 and k_2 in the same instant, then the execution exits with the larger value.

Case Study II: Derived Statements.

Figure 7 shows how to construct the derived statements [21] via the primitives. In particular, *every* $\mathbb{S}$ *p* implements a preemptive loop that checks the presence of $\mathbb{S}$. An *every* loop starts its body when $\mathbb{S}$ is present; and whenever $\mathbb{S}$ is present again in some further instants, it kills the current execution instantly and restarts a new iteration. Besides, *await [delayed]* $\mathbb{S}$ implements a *delayed* waiting, which starts as early as the next instant, as opposite to the default *immediate* waiting.

(1) $\mathit{halt} \triangleq \mathit{loop}\ (\mathit{yield})$

(2) $\mathit{loop}\ p\ \mathit{each}\ \mathbb{S} \triangleq \mathit{loop}\ (\mathit{abort}\ (p; \mathit{halt})\ \mathbb{S})$

(3) $\mathit{every}\ \mathbb{S}\ p \triangleq \mathit{await}\ \mathbb{S}; (\mathit{loop}\ p\ \mathit{each}\ \mathbb{S})$

(4) $\mathit{await}\ [\mathit{delayed}]\ \mathbb{S} \triangleq \mathit{yield}; \mathit{await}\ \mathbb{S}$

Figure 7 Expansion of derived preemptions.

3.3 An Effect Logic for the Temporal Specification, *ASyncEffs*

$$
\begin{array}{rccl}
(\mathit{Effects}) & \Phi & ::= & \bot \mid \epsilon \mid I \mid \mathbb{S}? \mid \Phi_1 \cdot \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid \Phi_1 || \Phi_2 \mid \Phi^\star \\
(\mathit{Instant}) & I & ::= & \{\} \mid \{\mathbb{S} \mapsto \alpha\} \mid I_1 \cup I_2 \\
(\mathit{Parametrized\ Signal}) & \mathbb{S} & ::= & S(v) \\
(\mathit{Signal\ Statuses}) & \alpha & ::= & \mathit{present} \mid \mathit{absent} \mid \mathit{undef}
\end{array}
$$

$(\mathit{Signal\ Variables})\, S \in \Sigma \qquad (\mathit{Values})\, v \qquad (\mathit{Waiting})? \qquad (\mathit{Kleene\ Star})\star$

Figure 8 Syntax of the *ASyncEffs*.

As shown in Figure 8, *ASyncEffs* comprise *false* ($\bot$); the empty trace ϵ; the singleton instant I; waiting for a parametrized signal $\mathbb{S}?$; trace concatenation $\Phi_1 \cdot \Phi_2$; trace disjunction $\Phi_1 \vee \Phi_2$; synchronous parallelism $\Phi_1 || \Phi_2$. *ASyncEffs* can also be constructed by $\star$, representing zero or more times of repetition of a trace. There are three possible statuses for a signal: present, absent and undefined. The default status of signals in a new instant is *undefined*. An instant I is a set of mappings from signals to their statuses; and instants can be empty sets $\{\}$, indicating no signal constraints for the instant.

3.4 Semantics of *ASyncEffs*

To define the semantic model, we use φ (*a trace of instants*) to represent the concrete computation execution. Let $\varphi \models \Phi$ denote the model relation, i.e., the execution trace φ satisfies the temporal effects Φ, with φ from the following concrete domain: $\varphi \triangleq \mathit{list}(I)$.

Figure 9 defines the semantics of *ASyncEffs*. $[]$ represents an empty trace; $++$ appends two traces; $[I]$ represents a singleton trace contains one instant I. Here I is a set of mappings from signals to statuses. We use $\{\mathbb{S}\}$ and $\{\overline{\mathbb{S}}\}$ to shorthand $\{\mathbb{S} \mapsto \mathit{present}\}$ and $\{\mathbb{S} \mapsto \mathit{absent}\}$ respectively. The pairings shown in one instant represent the *minimal* set of constraints for signals that are required/guaranteed to be true. Any instant contains contradictions, such as $\{\mathbb{S}, \overline{\mathbb{S}}\}$, will lead to *false*, as the signal $\mathbb{S}$ can not be both present and absent in the same instant.

$\varphi \models \epsilon$	*iff*	$\varphi = []$
$\varphi \models I$	*iff*	$\varphi = [I]$
$\varphi \models \mathbb{S}?$	*iff*	$\exists n{\geq}0.\ \varphi{=}\{\overline{\mathbb{S}}\}^n{++}[\{\mathbb{S}\}]$
$\varphi \models \Phi_1 \cdot \Phi_2$	*iff*	$\exists \varphi_1, \varphi_2,\ \varphi{=}\varphi_1{++}\varphi_2$ *such* that $\varphi_1{\models}\Phi_1$ and $\varphi_2{\models}\Phi_2$
$\varphi \models \Phi_1 \vee \Phi_2$	*iff*	$\varphi \models \Phi_1$ or $\varphi \models \Phi_2$
$\varphi \models \Phi_1 \vert\vert \Phi_2$	*iff*	$\varphi \models \Phi_1$ and $\varphi \models \Phi_2$
$\varphi \models \Phi^\star$	*iff*	$\varphi \models \epsilon$ *or* $\varphi \models (\Phi \cdot \Phi^\star)$
$\varphi \models \bot$	*iff*	*false*

Figure 9 Semantics of the *ASyncEffs* Logic.

Case Study III: Expressiveness of *ASyncEffs*.

As shown in Table 2, we are able to recursively encode event-based LTL operators into *ASyncEffs*, making it more intuitive and readable, mainly when nested operators occur. By putting effects in the postcondition, they restrict *future traces*; whereas in the precondition, they naturally encode *past-time* temporal specifications. The basic modal operators are: $\Box$ for "globally"; $\Diamond$ for "finally"; $\bigcirc$ for "next"; $\mathcal{U}$ for "until", and their past time reversed versions: $\overleftarrow{\Box}$; $\overleftarrow{\Diamond}$; and $\ominus$ for "previous"; $\mathcal{S}$ for "since". Besides, the implication operator is expressed as $\mathbb{A} \rightarrow \mathbb{B} \equiv \{\overline{\mathbb{A}}\} \vee \{\mathbb{A}, \mathbb{B}\}$. Apart from the high compatibility with standard first-order logic, *ASyncEffs* make the temporal verification more flexible to incorporate with other logics.

Table 2 Examples for converting LTL formulae into Effects. ($\{\mathbb{A}\}, \{\mathbb{B}\}$ represent different instants which contain signal $\mathbb{A}$ and $\mathbb{B}$ to be present.)

Φ_{post}	$\Box\mathbb{A} \equiv \{\mathbb{A}\}^\star$	$\Diamond\mathbb{A} \equiv \{\overline{\mathbb{A}}\}^\star \cdot \{\mathbb{A}\}$	$\bigcirc\mathbb{A} \equiv \{\} \cdot \{\mathbb{A}\}$	$\mathbb{A}\ \mathcal{U}\ \mathbb{B} \equiv \{\mathbb{A}\}^\star \cdot \{\mathbb{B}\}$
Φ_{pre}	$\overleftarrow{\Box}\mathbb{A} \equiv \{\mathbb{A}\}^\star$	$\overleftarrow{\Diamond}\mathbb{A} \equiv \{\mathbb{A}\} \cdot \{\overline{\mathbb{A}}\}^\star$	$\ominus\mathbb{A} \equiv \{\mathbb{A}\} \cdot \{\}$	$\mathbb{A}\ \mathcal{S}\ \mathbb{B} \equiv \{\mathbb{B}\} \cdot \{\mathbb{A}\}^\star$

4 Automated Forward Verification

Here, we present the forward rules, i.e., an axiomatic semantics model for the target language. These rules transfer program states and accumulate the effects syntactically. To define the rules, we introduce an environment $\mathcal{E}$ and the single program state (h, k), where h represents the trace of *history*; k is the *completion code*. Concretely: $\mathcal{E} \triangleq \overrightarrow{(\mathbb{S}{\mapsto}\alpha)},\ h \triangleq \Phi,\ k \in \mathbb{N}{\cup}\{0\}$. The forward rules are in the form: $\mathcal{E} \vdash \langle H, K\rangle\ p\ \langle H', K'\rangle$, where p is the given statement; $\langle H, K\rangle$ refers to a set of disjunctive program states, i.e., $\overrightarrow{(h, k)}$. The meaning of the transition rules can be described as follows:

$$\langle H', K'\rangle = \bigcup_{i=0}^{|\langle H,K\rangle|-1} \langle H'_i, K'_i\rangle \ \ \mathit{where}\ \ \mathcal{E} \vdash \langle h_i, k_i\rangle\ \ p\ \ \langle H'_i, K'_i\rangle.$$

[*FV-Value*] obtains the next state by inheriting the current state. [*FV-Emit*] extents the latest instant with $\mathbb{S}$ being present (the notation $+$ unions two instants). [*FV-Yield*] concatenates an empty instant to the tail of the history trace. [*FV-Seq*] computes p's effects first, and if the completion code $K_1{\leq}1$, i.e., there are no exits, then continuously computes the effects of q; otherwise, it discards q and propagates $\langle H_1, K_1\rangle$ directly.

$$\dfrac{}{\mathcal{E} \vdash \langle h, k\rangle\ v\ \langle h, k\rangle}\,[FV\text{-}Value] \qquad \dfrac{}{\mathcal{E} \vdash \langle h \cdot I, k\rangle\ emit\ \mathbb{S}\ \langle h \cdot (I{+}\{\mathbb{S}\}), k\rangle}\,[FV\text{-}Emit] \qquad \dfrac{}{\mathcal{E} \vdash \langle h, k\rangle\ yield\ \langle h \cdot \{\}, k\rangle}\,[FV\text{-}Yield]$$

$$\dfrac{\begin{array}{cc}\mathcal{E} \vdash \langle h, k\rangle\ \ p\ \ \langle H_1, K_1\rangle & \mathcal{E} \vdash \langle H_1, K_1\rangle\ \ q\ \ \langle H_2, K_2\rangle \\ \langle H', K'\rangle = \langle H_2, K_2\rangle\ \ when\ \ K_1{\le}1 & \langle H', K'\rangle = \langle H_1, K_1\rangle\ \ when\ \ K_1{>}1\end{array}}{\mathcal{E} \vdash \langle h, k\rangle\ seq\ p\ q\ \langle H', K'\rangle}\,[FV\text{-}Seq]$$

$[FV\text{-}Present]$ computes the effects of p and q by extending the latest instant with $\mathbb{S}$ being *present* and *absent*, respectively. The final state is a union of the results (the notation $\cup$ unions two program states).

$$\dfrac{\mathcal{E} \vdash \langle h \cdot I{+}\{\mathbb{S}\}, k\rangle\ \ p\ \ \langle H_1, K_1\rangle \qquad \mathcal{E} \vdash \langle h \cdot I{+}\{\overline{\mathbb{S}}\}, k\rangle\ \ q\ \ \langle H_2, K_2\rangle}{\mathcal{E} \vdash \langle h \cdot I, k\rangle\ present\ \mathbb{S}\ p\ q\ \langle H_1, K_1\rangle \cup \langle H_2, K_2\rangle}\,[FV\text{-}Present]$$

$[FV\text{-}Async]$ desugars the asynchronous primitive into a parallel program. $[FV\text{-}Await]$ concatenates $\mathbb{S}?$ to the tail of the history trace. $[FV\text{-}Exit]$ updates the value of k using $d{+}2$. $[FV\text{-}Trap]$ computes p's effects. When $K{\le}1$ – there is no exit to be handled – the final state is $\langle H, K\rangle$. When K equals to 2, it means there is an *exit* that needs to be handled by the current *trap*. When K is greater than 2, it means that there is an *exit* needs to be handled by an outer *trap* statement, therefore it propagates the program state $\langle H, K\text{-}1\rangle$.

$$\dfrac{\mathcal{E} \vdash \langle h, k\rangle\ \ (p; yield; emit\ \ \mathbb{S})||q\ \ \langle H', K'\rangle}{\mathcal{E} \vdash \langle h, k\rangle\ async\ \mathbb{S}\ p\ q\ \langle H', K'\rangle}[FV\text{-}Async] \qquad \dfrac{k'{=}d{+}2}{\mathcal{E} \vdash \langle h, k\rangle\ exit\ d\ \langle h, k'\rangle}[FV\text{-}Exit]$$

$$\dfrac{\langle \Delta\rangle = \langle h \cdot \mathbb{S}?, k\rangle}{\mathcal{E} \vdash \langle h, k\rangle\ await\ \mathbb{S}\ \langle \Delta\rangle}\,[FV\text{-}Await] \qquad \dfrac{\mathcal{E} \vdash \langle \epsilon, k\rangle\ \ p\ \ \langle H, K\rangle \quad \begin{array}{ll}\langle \Delta\rangle{=}\langle H, K\rangle & when\ (K{\le}1) \\ \langle \Delta\rangle{=}\langle H, 0\rangle & when\ (K{=}2) \\ \langle \Delta\rangle{=}\langle H, K\text{-}1\rangle & when\ (K{>}2)\end{array}}{\mathcal{E} \vdash \langle h, k\rangle\ trap\ p\ \langle h \cdot \Delta\rangle}[FV\text{-}Trap]$$

▶ **Definition 5** (Prepend Program States). *Given a history trace h', and program states $\Delta{=}(H, K)$, we define that: $h' \cdot \Delta = \{(h' \cdot h, k) \mid (h, k) \in \langle H, K\rangle\}$.*

$[FV\text{-}Call]$ triggers the back-end solver TRS to check if the instantiated precondition of the callee is satisfied by the current state. If it holds, the final state is obtained by concatenating the instantiated postcondition to the current effect state. Otherwise, the verification fails.

$$\dfrac{mn(\overrightarrow{\tau\ S(x)})\ \langle \mathbf{req}\ \Phi_{pre}\ \mathbf{ens}\ \Phi_{post}\rangle p \in \mathcal{P} \qquad h{\sqsubseteq}\Phi_{pre}[\overrightarrow{\mathbb{S}}/\overrightarrow{S(x)}] \qquad \langle H, K\rangle{=}\Phi_{post}[\overrightarrow{\mathbb{S}}/\overrightarrow{S(x)}]}{\mathcal{E} \vdash \langle h, k\rangle\ call\ mn\ (\overrightarrow{\mathbb{S}})\ \langle h \cdot H, K\rangle}\,[FV\text{-}Call]$$

$$\dfrac{\begin{array}{c}\mathcal{E} \vdash \langle \epsilon, k\rangle\ \ p\ \ \langle H, K\rangle \\ \langle \Delta\rangle{=}\langle h \cdot (H)^{\star}, K\rangle\ \ when\ \ (K{\le}1) \qquad \langle \Delta\rangle{=}\langle h \cdot H, K\rangle\ \ when\ \ (K{>}1)\end{array}}{\mathcal{E} \vdash \langle h, k\rangle\ loop\ p\ \langle \Delta\rangle}\,[FV\text{-}Loop]$$

$[FV\text{-}Loop]$ computes p's effects with ϵ to be the history trace. If the completion code $K{\le}1$, it appends a repeated trace to the history $h \cdot (H)^{\star}$. Otherwise, it exits the loop.

$$\frac{\begin{array}{c}\mathcal{E} \vdash \langle \epsilon, k\rangle\ p\ \langle H, K\rangle \\ \langle \Delta\rangle {=} \aleph^{Abort(\mathbb{S},K)}_{Interleave}(H, \epsilon)\end{array}}{\mathcal{E} \vdash \langle h, k\rangle\ abort\ p\ \mathbb{S}\ \langle h \cdot \Delta\rangle}[FV\text{-}Abort] \quad \frac{\begin{array}{c}\mathcal{E} \vdash \langle \epsilon, k\rangle\ p\ \langle H, K\rangle \\ \langle \Delta\rangle {=} \aleph^{Suspend(\mathbb{S},K)}_{Interleave}(H)\end{array}}{\mathcal{E} \vdash \langle h, k\rangle\ suspend\ p\ \mathbb{S}\ \langle h \cdot \Delta\rangle}[FV\text{-}Suspend]$$

[FV-*Abort*] and [FV-*Suspend*] compute the effects of p with ϵ to be the history traces; then calculate their corresponding interleaves[7]; lastly, prepend the original history to the final results.

Algorithm 1 presents the interleaving algorithm for *weak abortion* (cf. Definition 8 and Definition 10 for *First*(*fst*) and *Derivative*(D) functions respectively).

In strong preemption, the latest instant does not run when the preemption condition holds. In weak preemption, the latest instant is allowed to run even when the preemption condition holds but is terminated thereafter [22, 23]. Therefore, to implement the strong abortion from Algorithm 1, line 7 should be revised to $[(\Phi_{his} \cdot \{\mathbb{S}\}, \mathit{0})]$. We present the *interleaving* algorithm for *suspension* in Appendix C.

Algorithm 1 Weak Abort Interleaving

Input: $\mathbb{S}, (\Phi, k), \Phi_{his}$
Output: Program States Δ

```
1  rec function ℵ^{Abort(𝕊,K)}_{Interleave}(Φ, Φ_his)
2    if Φ=ε then
3      return [(ε, k)]
4    else
5      Δ ← []
6      foreach f ∈ fst(Φ) do
7        φ ← [(Φ_his · (f+{𝕊}), 0)]
8        Φ' ← D_f(Φ)
9        Φ'_his ← Φ_his · (f+{𝕊̄})
10       Δ' ← ℵ^{Abort(𝕊,K)}_{Interleave}(Φ', Φ'_his)
11       Δ ← Δ ∪ φ ∪ Δ'
12     end
13     return Δ
14 end
```

The rule [FV-*Par*] computes p and q's effects independently, then *parallel merges* the results. Notation $\vdash_{pm}$ refers to the *parallelMerge* algorithm, detailed in Sec. 4.1.

$$\frac{\mathcal{E} \vdash \langle \epsilon, k\rangle\ \ p\ \ \langle H_1, K_1\rangle \quad \mathcal{E} \vdash \langle \epsilon, k\rangle\ \ q\ \ \langle H_2, K_2\rangle \quad \vdash_{pm}\langle H_1, K_1\rangle || \langle H_2, K_2\rangle {\rightsquigarrow} \langle \Delta\rangle}{\mathcal{E} \vdash \langle h, k\rangle\ p||q\ \langle h \cdot \Delta\rangle}\ [FV\text{-}Par]$$

4.1 Parallel Merge Algorithm

The parallel merging[8] rules are in the form: $\vdash_p\langle H_1, K_1\rangle\ ||\ \langle H_2, K_2\rangle \rightsquigarrow \langle H', K'\rangle$. Given two sets of program states $\langle H_1, K_1\rangle$ and $\langle H_2, K_2\rangle$, the rule [PM-*Union*] obtains $\langle \Delta\rangle$ by combining the parallel merged states of their cartesian products.

$$\frac{\begin{array}{c}[PM\text{-}Union] \\ \forall(h_1, k_1){\in}\langle H_1, K_1\rangle \qquad \forall(h_2, k_2){\in}\langle H_2, K_2\rangle \qquad \langle \Delta\rangle{=}\bigcup(\vdash_{pm} \langle h_1, k_1\rangle||\langle h_2, k_2\rangle \rightsquigarrow \langle h', k'\rangle)\end{array}}{\vdash_{pm} \langle H_1, K_1\rangle||\langle H_2, K_2\rangle \rightsquigarrow \langle \Delta\rangle}$$

[7] The interleaving comes from the over-approximation of all the possible effect traces. For example, for trace $\{\mathtt{A}\} \cdot \{\mathtt{B}\}$, the (weak) abort preemption with condition signal $\mathbb{S}$ creates three possibilities: $(\{\mathtt{A}, \overline{\mathbb{S}}\} \cdot \{\mathtt{B}, \overline{\mathbb{S}}\}) \vee (\{\mathtt{A}, \overline{\mathbb{S}}\} \cdot \{\mathtt{B}, \mathbb{S}\}) \vee (\{\mathtt{A}, \mathbb{S}\})$.

[8] To help with the understanding, concrete examples are:
- $\langle\{\mathtt{A}\} \cdot \{\mathtt{B}\} \cdot \{\mathtt{C}\}, \mathit{0}\rangle\ ||\ \langle\{\mathtt{X}\} \cdot \{\mathtt{Y}\} \cdot \{\mathtt{Z}\}, \mathit{0}\rangle \rightsquigarrow \langle\{\mathtt{A}, \mathtt{X}\} \cdot \{\mathtt{B}, \mathtt{Y}\} \cdot \{\mathtt{C}, \mathtt{Z}\}, \mathit{0}\rangle$;
- $\langle\{\mathtt{A}\} \cdot \{\mathtt{C}\}, \mathit{2}\rangle\ ||\ \langle\{\mathtt{X}\} \cdot \{\mathtt{Y}\} \cdot \{\mathtt{Z}\}, \mathit{0}\rangle \rightsquigarrow \langle\{\mathtt{A}, \mathtt{X}\} \cdot \{\mathtt{C}, \mathtt{Y}\}, \mathit{2}\rangle$; and
- $\langle\{\mathtt{A}\} \cdot \{\mathtt{C}\}, \mathit{0}\rangle\ ||\ \langle\{\mathtt{X}\} \cdot \{\mathtt{Y}\} \cdot \{\mathtt{Z}\}, \mathit{2}\rangle \rightsquigarrow \langle\{\mathtt{A}, \mathtt{X}\} \cdot \{\mathtt{C}, \mathtt{Y}\}, \{\mathtt{Z}\}, \mathit{2}\rangle$.

$[PM\text{-}Unfold]$ applies to deductive steps, which deploys auxiliary functions, $\mathit{fst}(\Phi)$ and $D_I(\Phi)$, to compute the first instants and partial derivatives, cf. Definition 5.1. The first step is to get the *first* set from h_1 and h_2, respectively. For each pair (f_1, f_2) from the cartesian products of the *first* sets, it merges f_1 and f_2 to be the paralleled first instant, denoted as I; then gets the derivatives of h_1 and h_2 w.r.t I respectively; Finally, it prepends I to the parallel merged derivatives by recursively calling the parallel merge algorithm.

$$\frac{\begin{array}{ccc} F_1{=}\mathit{fst}(h_1) & F_2{=}\mathit{fst}(h_2) & \forall f_1{\in}F_1.\forall f_2{\in}F_2.\ I{=}f_1{\cup}f_2, \\ der_1{=}D_I(h_1) & der_2{=}D_I(h_2) & \langle H', K'\rangle{=}\bigcup(\vdash_{pm}\langle der_1, k_1\rangle||\langle der_2, k_2\rangle) \end{array}}{\vdash_{pm} \langle h_1, k_1\rangle||\langle h_2, k_2\rangle \rightsquigarrow \langle I\cdot H', K'\rangle}[PM\text{-}Unfold]$$

The next rules deal with the base cases and terminate the merging process. $[PM\text{-}EqLen]$ is used when two effects have the same length. $[PM\text{-}Cut]$ is used when one of the effects is shorter than the other and raises an exit. $[PM\text{-}Absorb]$ is used when one of the effects are shorter than another yet without any exits.

$$\frac{[PM\text{-}EqLen] \quad k'{=}max(k_1, k_2)}{\vdash_{pm} \langle\epsilon, k_1\rangle||\langle\epsilon, k_2\rangle{\rightsquigarrow}\langle\epsilon, k'\rangle} \quad \frac{[PM\text{-}Cut] \quad k_1{>}1}{\vdash_{pm} \langle\epsilon, k_1\rangle||\langle h_2, k_2\rangle{\rightsquigarrow}\langle\epsilon, k_1\rangle} \quad \frac{[PM\text{-}Absorb] \quad k_1{\leq}1}{\vdash_{pm} \langle\epsilon, k_1\rangle||\langle h_2, k_2\rangle{\rightsquigarrow}\langle h_2, k_2\rangle}$$

4.2 Soundness Theorem For the Forward Rules

▶ **Theorem 6** (Soundness of the Forward Rules). *$\forall p, \mathcal{E}$, if $\mathcal{E} \vdash \langle h, k\rangle\ p\ \langle H', K'\rangle$, and $\varphi \models h$, and $p \xrightarrow[\mathcal{E}]{e_0,0}{}^{*} p' \xrightarrow[\mathcal{E}]{\emptyset,1} p_1 \xrightarrow[\mathcal{E}_1]{e_1,0}{}^{*} p'_1 \xrightarrow[\mathcal{E}_1]{\emptyset,1} p_2 \xrightarrow[\mathcal{E}_2]{e_2,0}{}^{*} p'_2 \xrightarrow[\mathcal{E}_2]{\emptyset,1} \ldots p_n \xrightarrow[\mathcal{E}_n]{e_n,0}{}^{*} p'_n \xrightarrow[\mathcal{E}_n]{\emptyset,k_f} ()$, then it implies that $\exists(h', k') \in \langle H', K'\rangle$ such that $\varphi \mathbin{++} [e_0; e_1; ...; e_n] \models h'$ and $k'{=}k_f$. (Note that, $p \xrightarrow{e,0}{}^{*} p'$ denotes the reflexive, transitive closure of $p \xrightarrow{e,0} p'$.)*

Proof. By induction on the structure of p. detailed in Appendix D. ◀

5 Temporal Verification via a TRS

The TRS is inspired by Antimirov and Mosses' algorithm [15] but solving the language inclusions between *ASyncEffs*. It is triggered i) prior to module calls for the precondition checking; and ii) at the end of verifying a module for the post condition checking. More specifically, given two effects Φ_1, Φ_2, TRS decides if the inclusion $\Phi_1 \sqsubseteq \Phi_2$ is valid. During the effects rewriting process, the inclusions are in the form of $\Gamma \vdash \Phi_1 \sqsubseteq^{\Phi} \Phi_2$, a shorthand for: $\Gamma \vdash \Phi \cdot \Phi_1 \sqsubseteq \Phi \cdot \Phi_2$. To prove such inclusions is to check whether all the possible effect traces in the antecedent Φ_1 are legitimately allowed in the possible effects traces from the consequent Φ_2. Γ is the proof context, i.e., effects inclusion hypotheses, Φ is the history effects from the antecedent that have been used to match the effects from the consequent. The inclusion checking is initially invoked with $\Gamma{=}\{\}$, $\Phi{=}\epsilon$.

5.1 Auxiliary Functions: Nullable, First and Derivative

We provide definitions and implementations of auxiliary functions $\mathit{Nullable}(\delta)$, $\mathit{First}(\mathit{fst})$ and $\mathit{Derivative}(D)$ respectively. Intuitively, the Nullable function $\delta(\Phi)$ returns a boolean value indicating whether Φ contains the empty trace; the First function $\mathit{fst}(\Phi)$ computes a set of possible head instants of Φ; and the Derivative function $D_I(\Phi)$ computes a next-state effects after eliminating one instant I from the head of current effects Φ. Here marks the novel definitions – opposite to the existing ones in [15] – using '$=_{\clubsuit}$' in Definitions 6, 7, 9.

▶ **Definition 7** (Nullable). *Given any effect Φ, $\delta(\Phi){=}true \Leftrightarrow (\epsilon \in \Phi)$, where:*

$$\delta(\bot){=}false \qquad \delta(\epsilon){=}true \qquad \delta(I){=}false \qquad \delta(\mathbb{S}?){=_{\clubsuit}}false \qquad \delta(\Phi^{\star}){=}true$$
$$\delta(\Phi_1 \cdot \Phi_2){=}\delta(\Phi_1){\wedge}\delta(\Phi_2) \quad \delta(\Phi_1 \vee \Phi_2){=}\delta(\Phi_1) \vee \delta(\Phi_2) \quad \delta(\Phi_1||\Phi_2){=_{\clubsuit}}\delta(\Phi_1){\wedge}\delta(\Phi_2)$$

▶ **Definition 8** (First). *Let $fst(\Phi){:=}\{I \mid (I \cdot \Phi') \in [\![\Phi]\!]\}$ be the set of first instants derivable from effect Φ. ($[\![\Phi]\!]$ represents all the traces contained in Φ)*

$$fst(\bot){=}fst(\epsilon){=}\{\} \qquad fst(I){=}\{I\} \qquad fst(\mathbb{S}?){=_{\clubsuit}}\{\{\mathbb{S}{\mapsto}present\};\{\mathbb{S}{\mapsto}absent\}\}$$
$$fst(\Phi^{\star}){=}fst(\Phi) \qquad fst(\Phi_1 \cdot \Phi_2){=}\begin{cases} fst(\Phi_1) \cup fst(\Phi_2) & if\ \ \delta(\Phi_1){=}true \\ fst(\Phi_1) & if\ \ \delta(\Phi_1){=}false \end{cases}$$
$$fst(\Phi_1 \vee \Phi_2){=}fst(\Phi_1) \cup fst(\Phi_2) \qquad fst(\Phi_1||\Phi_2){=_{\clubsuit}}\{(f_1{+}f_2) \mid f_1 \in fst(\Phi_1), f_2 \in fst(\Phi_2)\}$$

▶ **Definition 9** (Instants Subsumption). *Given two instants I and J, we define the subset relation $I{\subseteq}J$ as: the set of present signals in J is a subset of the set of present signals in I, and the set of absent signals in J is a subset of the set of absent signals in I, as in having more constraints refers to a smaller set of satisfying instants. Formally,*

$$I \subseteq J \Leftrightarrow \{\mathbb{S} \mid (\mathbb{S}{\mapsto}present) \in J\} \subseteq \{\mathbb{S} \mid (\mathbb{S}{\mapsto}present) \in I\}$$
$$and\ \{\mathbb{S} \mid (\mathbb{S}{\mapsto}absent) \in J\} \subseteq \{\mathbb{S} \mid (\mathbb{S}{\mapsto}absent) \in I\}$$

▶ **Definition 10** (Partial Derivatives for *ASyncEffs*). *The partial derivative $D_I(\Phi)$ of effects Φ w.r.t. an instant I computes the effects for the left quotient $I^{-1}[\![\Phi]\!]$* [9].

$$D_I(\bot){=}\bot \quad D_I(\epsilon){=}\bot \quad D_I(J){=}\begin{cases} \epsilon & if\ \ I{\subseteq}J \\ \bot & if\ \ I{\not\subseteq}J \end{cases} \quad D_I(\mathbb{S}?){=_{\clubsuit}}\begin{cases} \epsilon & if\ \ I{\subseteq}\{\mathbb{S}{\mapsto}present\} \\ \bot & if\ \ I{\not\subseteq}\{\mathbb{S}{\mapsto}absent\} \\ \mathbb{S}? & otherwise \end{cases}$$
$$D_I(\Phi^{\star}){=}D_I(\Phi) \cdot \Phi^{\star} \qquad D_I(\Phi_1 \cdot \Phi_2){=}\begin{cases} D_I(\Phi_1) \cdot \Phi_2 \vee D_I(\Phi_2) & if\ \ \delta(\Phi_1){=}true \\ D_I(\Phi_1) \cdot \Phi_2 & if\ \ \delta(\Phi_1){=}false \end{cases}$$
$$D_I(\Phi_1 \vee \Phi_2){=}D_I(\Phi_1) \vee D_I(\Phi_2) \qquad D_I(\Phi_1||\Phi_2){=_{\clubsuit}}D_I(\Phi_1)||D_I(\Phi_2)$$

5.2 Rewriting Rules

Given the well-defined auxiliary functions above, we now discuss the key steps and related rewriting rules that we may use in effects inclusion proofs.

1. Axiom rules. Analogous to the standard propositional logic, $\bot$ (referring to *false*) entails any effects, while no *non-false* effects entails $\bot$.

$$\frac{}{\Gamma \vdash \bot \sqsubseteq \Phi}\,[\texttt{Bot-LHS}] \qquad \frac{\Phi \neq \bot}{\Gamma \vdash \Phi \not\sqsubseteq \bot}\,[\texttt{Bot-RHS}] \qquad \frac{\delta(\Phi_1) \wedge \neg\delta(\Phi_2)}{\Gamma \vdash \Phi_1 \not\sqsubseteq \Phi_2}\,[\texttt{Disprove}] \qquad \frac{fst(\Phi_1) = \{\}}{\Gamma \vdash \Phi_1 \sqsubseteq \Phi_2}\,[\texttt{Prove}]$$

[9] For example, $\{\mathtt{A}\}^{-1}[\![\{\mathtt{A}\} \cdot \{\mathtt{B}\}]\!]{=}[\![\{\mathtt{B}\}]\!]$, and $\{\mathtt{A}\}^{-1}[\![\{\mathtt{A}\} \vee \{\mathtt{B}\}]\!]{=}[\![\epsilon \vee \bot]\!]$, cf Definition 4.

2. Disprove (Heuristic Refutation). We [`Disprove`] the inclusions when the antecedent is nullable, while the consequent is not. Intuitively, the antecedent contains at least one more trace, i.e., ϵ, than the consequent.

3. Prove. We use two rules to prove an inclusion: (i) [`Prove`] is used when the fst set of the antecedent is empty; and (ii) [`Reoccur`] to prove an inclusion when there exist inclusion hypotheses in the proof context Γ, which are able to soundly prove the current goal. One of the special cases of this rule is when the identical inclusion is shown in the proof context, we then terminate the procedure and prove it as a valid inclusion.

$$\frac{(\Phi_1 \sqsubseteq \Phi_3) \in \Gamma \qquad (\Phi_3 \sqsubseteq \Phi_4) \in \Gamma \qquad (\Phi_4 \sqsubseteq \Phi_2) \in \Gamma}{\Gamma \vdash \Phi_1 \sqsubseteq \Phi_2} \quad [\texttt{Reoccur}]$$

4. Unfolding (Induction). This is the inductive step of unfolding the inclusions. Firstly, we make use of the auxiliary function *fst* to get a set of instants F, which are all the possible initial instants from the antecedent. Secondly, we obtain a new proof context Γ' by adding the current inclusion, as an inductive hypothesis, into the current proof context Γ. Thirdly, we iterate each element $I \in F$, and compute the partial derivatives (*next-state* effects) of both the antecedent and consequent w.r.t I. The proof of the original inclusion succeeds if all the derivative inclusions succeeds.

$$\frac{F = fst(\Phi_1) \quad \Gamma'{=}\Gamma, (\Phi_1 \sqsubseteq \Phi_2) \quad \forall I \in F.(\Gamma' \vdash D_I(\Phi_1) \sqsubseteq D_I(\Phi_2))}{\Gamma \vdash \Phi_1 \sqsubseteq \Phi_2} \quad [\texttt{Unfold}]$$

▶ **Theorem 11** (TRS Termination). *The rewriting system TRS is terminating.*

▶ **Theorem 12** (TRS Soundness). *Given an inclusion $\Phi_1 \sqsubseteq \Phi_2$, if the TRS returns TRUE when proving $\Phi_1 \sqsubseteq \Phi_2$, then $\Phi_1 \sqsubseteq \Phi_2$ is valid.*

Proof. See Appendix E and Appendix F. ◀

6 Implementation and Evaluation

To show the feasibility of our approach, we prototype our automated verification system using OCaml; prove soundness for both the forward verifier and the TRS; validate and evaluate the implementation for conformance using a microbenchmark [24]. The microbenchmark is constructed by manually annotating *ASyncEffs* specifications, including both succeeded and failed cases. The validation tests are synthetic examples to test the main contributions, including the preemption interleaving computation and the inclusion checking for the parallel composition and the waiting operator.

Table 3 presents the evaluation results. We select 16 target programs, varying from 15 lines to 300 lines, and annotate *ASyncEffs* specifications with a 1:1 ratio for succeeded/failed cases. The results record: **No.** for the index of the program; **LOC** for lines of code; **Infer(ms)** for effects inference time; **#Prop(✓)**for the number of valid properties; **Avg-Prove(ms)**for the average proving time for the valid properties; **#Prop(✗)**for the number of invalid properties; and **Avg-Dis(ms)**for the average disproving time for the invalid properties. Times are counted using milliseconds, and the experiment is done on a MacBook Pro with a 2.6 GHz 6-Core Intel Core i7 processor.

Discussion: Generally, the inference time increases with a linear complexity. We notice that the disproving times for invalid properties are constantly low. This finding echoes

Table 3 Experimental Results.

No.	LOC	Infer(ms)	#Prop(✓)	Avg-Prove(ms)	#Prop(✗)	Avg-Dis(ms)
1	18	0.037	5	0.7634	5	0.0116
2	33	0.145	5	1.3074	5	0.045
3	55	0.34	5	6.0766	5	1.1682
4	84	0.098	5	3.0678	5	0.1058
5	110	0.191	7	1.7544	7	0.5031
6	124	0.323	7	4.0114	7	0.3957
7	138	0.321	7	3.8399	7	0.4261
8	163	0.594	7	6.1009	7	1.5019
9	178	0.941	9	10.7758	9	0.5769
10	185	1.921	9	13.9332	9	0.04422
11	202	3.434	9	27.4447	9	0.0561
12	220	6.439	9	59.2226	9	0.745
13	250	3.6	11	29.5766	11	0.0662
14	261	7.552	11	64.2137	11	0.6121
15	293	14.896	11	115.9795	11	0.5462
16	304	30.889	11	237.2522	11	0.07164

the insights from prior TRS-based works [25, 15, 26, 27, 28], which suggest that TRS is a better average-case algorithm than those based on the comparison of automata. That is because *it only constructs automata as far as it needs*, which makes it more efficient when disproving incorrect specifications, as we can disprove it earlier without constructing the whole automata. In other words, the more incorrect specifications are, the more efficient our solver is. Our proposed effect logic and the abstract semantics for Esterel not only tightly capture the behaviors of an preemptive asynchronous execution models but also help to mitigate the programming challenges in both worlds.

7 Related Work

7.1 Verification framework

This work is a significant extension of [14], which proposes a novel temporal verification framework - a forward verifier with a TRS - for pure synchronous languages; while [14] mainly tackles the causality checking problem for signal status concerning the *perfect synchrony* assumption. Although based on the same verification framework, this work is dedicated to the verification for the mixture of synchronous preemptions and asynchronous primitives, which have not been covered by [14] or any other prior works.

7.2 Synchronous Preemptions and Asynchronous Promises.

Our semantics model for Esterel, closely follows [23, 29, 22, 30], where [29, 22] define the operational semantics of synchronous language Esterel and [23, 30] provide general perspectives for preemptions. However, the existing state-based operational semantics are not ideal for compositional reasoning for preemptive programs at the source level, our work proposes novel axiomatic semantics, which can help meet this requirement for better modularity.

In particular, [30] also provides a solution for verifying synchronous preemptions, but using classic LTL formulas as the specifications and limited with only *immediate* preemptions. We are of the opinion that i) our *ASyncEffs* is more flexible in terms of the expressiveness power; ii) our TRS is more efficient by avoiding the complex translation into automata; iii) and our forward verifier provides more comprehensive reasoning for preemption primitives.

ECMAScript 6 [31] supports primitives *async* and *await* serving for *promises*-based asynchronous programs, which can be written in a sequential style, leading to more concise code. However, the ECMAScript 6 standard specifies the semantics of promises informally and in operational terms, which is not suitable for formal reasoning or program analysis. Prior works [10, 11], in order to understand promise-related bugs, present the λ_p calculus, which provides a formal semantics for JavaScript promises. Based on these, our work defines the semantics of *async* and *await* in the event-driven synchronous concurrency context.

To the best of our knowledge, this work is the first to combine the semantics of synchronous preemptions and asynchronous promises, building the theoretical foundation for analyzing such a blending of two distinct execution models.

8 Conclusion

We demonstrate how to give axiomatic semantics for the full-featured Esterel by trace processing functions, and use *ASyncEffs* to capture reactive program behaviors and temporal properties. Our proposal enables a Hoare-style forward verifier, which computes the program effects constructively. The proposed modular analysis of preemptions and asynchronous interactions are new and potentially useful for prior constructiveness analysis. We present an efficient TRS to prove the annotated *ASyncEffs* properties. We prototype the verification system and show its feasibility. In summary, our work is the first that formulates the semantics of an preemptive asynchronous execution model; that automates modular temporal verification for reactive programs using an expressive effect logic.

References

1. A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, "The synchronous languages 12 years later," *Proc. IEEE*, vol. 91, no. 1, pp. 64–83, 2003. [Online]. Available: https://doi.org/10.1109/JPROC.2002.805826
2. G. Berry, A. Bouali, X. Fornari, E. Ledinot, E. Nassor, and R. de Simone, "ESTEREL: a formal method applied to avionic software development," *Sci. Comput. Program.*, vol. 36, no. 1, pp. 5–25, 2000. [Online]. Available: https://doi.org/10.1016/S0167-6423(99)00015-5
3. G. Berry and G. Gonthier, "The esterel synchronous programming language: Design, semantics, implementation," *Sci. Comput. Program.*, vol. 19, no. 2, pp. 87–152, 1992. [Online]. Available: https://doi.org/10.1016/0167-6423(92)90005-V
4. C. Ratel, N. Halbwachs, and P. Raymond, "Programming and verifying critical systems by means of the synchronous data-flow language LUSTRE," in *Proceedings of the conference on Software for citical systems, SIGSOFT 1991, New Orleans, Louisiana, USA*, M. Moriconi, Ed. ACM, 1991, pp. 112–119. [Online]. Available: https://doi.org/10.1145/125083.123062
5. A. Benveniste, P. Le Guernic, and C. Jacquemot, "Synchronous programming with events and relations: the SIGNAL language and its semantics," *Sci. Comput. Program.*, vol. 16, no. 2, pp. 103–149, 1991. [Online]. Available: https://doi.org/10.1016/0167-6423(91)90001-E
6. G. Bierman, C. Russo, G. Mainland, E. Meijer, and M. Torgersen, "Pause n play: Formalizing asynchronous c sharp," in *European Conference on Object-Oriented Programming*. Springer, 2012, pp. 233–257.

7 G. Berry and M. Serrano, “Hiphop.js: (a)synchronous reactive web programming,” in *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, A. F. Donaldson and E. Torlak, Eds. ACM, 2020, pp. 533–545. [Online]. Available: https://doi.org/10.1145/3385412.3385984

8 Z. Maintainers, https://zio.dev/, 2022.

9 D. F. patterns and technical concepts, https://docs.microsoft.com/en-us/azure/azure-functions/durable, 2019.

10 M. Madsen, O. Lhoták, and F. Tip, “A model for reasoning about javascript promises,” *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 86:1–86:24, 2017. [Online]. Available: https://doi.org/10.1145/3133910

11 S. Alimadadi, D. Zhong, M. Madsen, and F. Tip, “Finding broken promises in asynchronous javascript programs,” *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 162:1–162:26, 2018. [Online]. Available: https://doi.org/10.1145/3276532

12 C. Prisacariu, “Synchronous kleene algebra,” *J. Log. Algebraic Methods Program.*, vol. 79, no. 7, pp. 608–635, 2010. [Online]. Available: https://doi.org/10.1016/j.jlap.2010.07.009

13 S. Broda, S. Cavadas, M. Ferreira, and N. Moreira, “Deciding synchronous kleene algebra with derivatives,” in *Implementation and Application of Automata - 20th International Conference, CIAA 2015, Umeå, Sweden, August 18-21, 2015, Proceedings*, ser. Lecture Notes in Computer Science, F. Drewes, Ed., vol. 9223. Springer, 2015, pp. 49–62. [Online]. Available: https://doi.org/10.1007/978-3-319-22360-5_5

14 Y. Song and W. Chin, “A synchronous effects logic for temporal verification of pure esterel,” in *Verification, Model Checking, and Abstract Interpretation - 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17-19, 2021, Proceedings*, ser. Lecture Notes in Computer Science, F. Henglein, S. Shoham, and Y. Vizel, Eds., vol. 12597. Springer, 2021, pp. 417–440. [Online]. Available: https://doi.org/10.1007/978-3-030-67067-2_19

15 V. M. Antimirov and P. D. Mosses, “Rewriting extended regular expressions,” *Theor. Comput. Sci.*, vol. 143, no. 1, pp. 51–72, 1995. [Online]. Available: https://doi.org/10.1016/0304-3975(95)80024-4

16 V. Antimirov, “Partial derivatives of regular expressions and finite automata constructions,” in *Annual Symposium on Theoretical Aspects of Computer Science.* Springer, 1995, pp. 455–466.

17 J. Brotherston, “Cyclic proofs for first-order logic with inductive definitions,” in *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference, TABLEAUX 2005, Koblenz, Germany, September 14-17, 2005, Proceedings*, ser. Lecture Notes in Computer Science, B. Beckert, Ed., vol. 3702. Springer, 2005, pp. 78–92. [Online]. Available: https://doi.org/10.1007/11554554_8

18 S. P. Florence, S. You, J. A. Tov, and R. B. Findler, “A calculus for esterel: if can, can. if no can, no can,” *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 61:1–61:29, 2019. [Online]. Available: https://doi.org/10.1145/3290374

19 C. multitasking, https://en.m.wikipedia.org/wiki/Cooperative_multitasking, 2022.

20 G. Gonthier, “Sémantiques et modèles d’exécution des langages réactifs synchrones: application à esterel,” Ph.D. dissertation, Paris 11, 1988.

21 Wikipedia, https://en.wikipedia.org/wiki/Esterel, 2022.

22 G. Berry, “Preemption in concurrent systems,” in *Foundations of Software Technology and Theoretical Computer Science, 13th Conference, Bombay, India, December 15-17, 1993, Proceedings*, ser. Lecture Notes in Computer Science, R. K. Shyamasundar, Ed., vol. 761. Springer, 1993, pp. 72–93. [Online]. Available: https://doi.org/10.1007/3-540-57529-4_44

23 S. Pinchinat, É. Rutten, and R. K. Shyamasundar, “Preemption primitives in reactive languages (A preliminary report),” in *Algorithms, Concurrency and Knowledge: 1995 Asian Computing Science Conference, ACSC ’95, Pathumthani, Thailand, December 11-13, 1995, Proceedings*, ser. Lecture Notes in Computer Science, K. Kanchanasut

and J. Lévy, Eds., vol. 1023. Springer, 1995, pp. 111–125. [Online]. Available: https://doi.org/10.1007/3-540-60688-2_39

24 Anonymous, https://zenodo.org/record/7071374#.Yx8oU-xBydY, 2022.

25 Y. Song and W. Chin, "Automated temporal verification of integrated dependent effects," in *Formal Methods and Software Engineering - 22nd International Conference on Formal Engineering Methods, ICFEM 2020, Singapore, Singapore, March 1-3, 2021, Proceedings*, ser. Lecture Notes in Computer Science, S. Lin, Z. Hou, and B. P. Mahony, Eds., vol. 12531. Springer, 2020, pp. 73–90. [Online]. Available: https://doi.org/10.1007/978-3-030-63406-3_5

26 M. Almeida, N. Moreira, and R. Reis, "Antimirov and mosses's rewrite system revisited," *Int. J. Found. Comput. Sci.*, vol. 20, no. 4, pp. 669–684, 2009. [Online]. Available: https://doi.org/10.1142/S0129054109006802

27 M. Keil and P. Thiemann, "Symbolic solving of extended regular expression inequalities," in *34th International Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2014, December 15-17, 2014, New Delhi, India*, ser. LIPIcs, V. Raman and S. P. Suresh, Eds., vol. 29. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2014, pp. 175–186. [Online]. Available: https://doi.org/10.4230/LIPIcs.FSTTCS.2014.175

28 D. Hovland, "The inclusion problem for regular expressions," *J. Comput. Syst. Sci.*, vol. 78, no. 6, pp. 1795–1813, 2012. [Online]. Available: https://doi.org/10.1016/j.jcss.2011.12.003

29 G. Berry, "The constructive semantics of pure esterel-draft version 3," *Draft Version*, vol. 3, 1999.

30 M. Gesell, A. Morgenstern, and K. Schneider, "Lifting verification results for preemption statements," in *Software Engineering and Formal Methods - 11th International Conference, SEFM 2013, Madrid, Spain, September 25-27, 2013. Proceedings*, ser. Lecture Notes in Computer Science, R. M. Hierons, M. G. Merayo, and M. Bravetti, Eds., vol. 8137. Springer, 2013, pp. 91–105. [Online]. Available: https://doi.org/10.1007/978-3-642-40561-7_7

31 E. Ecma, "262: Ecmascript language specification," *ECMA (European Association for Standardizing Information and Communication Systems), pub-ECMA: adr,*, 1999.

A Forward verification for the module `Main`

```
module Main (out open, close, loading, loaded, compOther, logData){
```
(10) $\langle \{\} \rangle$ *(– initialize the current effects using precondition's last instant –)*
```
emit open("filePath");
```
(11) $\langle \{open\} \rangle$ [FV-Emit]
```
fork{ Read (open, close, loading, loaded, compOther, logData); }
```
(12) $\{\}^\star \cdot \{open\} \sqsubseteq \Phi^{Read}_{pre}$ [FV-Call] *(-TRS:Read's precondition is satisfied when called-)*
$\langle \{open\} \cdot \{loading, compOther\} \cdot \{loaded\} \cdot \{logData\} \cdot close? \rangle$ [FV-Call]
(13) $\langle \{open\} \rangle$ *(– inherited from state (11) –)*
```
par { await logData;
```
(14) $\langle \{open\} \cdot logData? \rangle$ [FV-Await]
```
        emit close("filePath"); }}
```
(15) $\langle \{open\} \cdot logData? \cdot \{close\} \rangle$ [FV-Emit]
(16) $\langle (\{open\} \cdot \{loading, compOther\} \cdot \{loaded\} \cdot \{logData\} \cdot close?)$
$|| (\{open\} \cdot logData? \cdot \{close\}) \rangle$ [FV-Fork-Par]
$\langle \{open\} \cdot \{loading, compOther\} \cdot \{loaded\} \cdot \{logData\} \cdot \{close\} \rangle$ [Effects-Parallel-Merge]
(17) *(-TRS: check the postcondition of module Main; Succeed. -)*
$\{open\} \cdot \{loading, compOther\} \cdot \{loaded\} \cdot \{logData\} \cdot \{close\} \sqsubseteq \{open\} \cdot \{\}^\star \cdot \{close\}$

Figure 10 A demonstration of the forward verification for the module `Main`.

B Operational Semantics Rules for the Basic Statements

We start with axioms: statement () terminates without emitting any signals and $k{=}0$; statement *yield* terminates without emitting any signals and $k{=}1$; and statement *emit* $\mathbb{S}$ sets the signal $\mathbb{S}$ to be present and terminates with $k{=}0$.

$$() \xrightarrow[\mathcal{E}]{\emptyset,0} ()\ [\texttt{Axiom-Nothing}] \qquad yield \xrightarrow[\mathcal{E}]{\emptyset,1} ()\ [\texttt{Axiom-Yield}] \qquad emit\ \mathbb{S} \xrightarrow[\mathcal{E}]{\{\mathbb{S}\},0} ()\ [\texttt{Axiom-Emit}]$$

The rules for sequences vary based on the completion code k: when p terminates with $k{=}0$, (`Seq-0`) executes q immediately; when p produces a yield, so does the whole sequence; when p raises an exception with depth k, (`Seq-n`) discards the rest of the code. The rule (`Loop`) performs an instantaneous unfolding of the loop into a sequence.

$$\frac{p \xrightarrow[\mathcal{E}]{\alpha,0} () \quad q \xrightarrow[\mathcal{E}]{f,k} q'}{p;\, q \xrightarrow[\mathcal{E}]{e\cup f,k} q'}[\texttt{Seq-0}] \qquad \frac{p \xrightarrow[\mathcal{E}]{\alpha,k} p' \quad (k{\le}1)}{p;\, q \xrightarrow[\mathcal{E}]{\alpha,k} p';\, q}[\texttt{Seq-1}] \qquad \frac{p \xrightarrow[\mathcal{E}]{\alpha,k} p' \quad (k{>}1)}{p;\, q \xrightarrow[\mathcal{E}]{\alpha,k} ()}[\texttt{Seq-n}] \qquad \frac{p;\, loop\ p \xrightarrow[\mathcal{E}]{\alpha,k} p'}{loop\ p \xrightarrow[\mathcal{E}]{\alpha,k} p'}[\texttt{Loop}]$$

The rule (`Call`) retrieves the function body p of mn from the program, and executes p.

$$\frac{x\ (\overrightarrow{\tau\ S})\ \langle \mathbf{req}\ \Phi_{pre}\ \mathbf{ens}\ \Phi_{post} \rangle\ p \in \mathcal{P} \qquad p \xrightarrow[\mathcal{E}]{\alpha,k} p'}{call\ x\ (\overrightarrow{\mathbb{S}}) \xrightarrow[\mathcal{E}]{\alpha,k} p'}\ [\texttt{Call}]$$

C Preemption Interleaving Algorithms

We present the weak suspend interleaving in Algorithm 2. Furthermore, to implement the strong suspend from Algorithm 2, in line 9 should be: $\Delta_2 \leftarrow \{\mathbb{S}\} \cdot \{\} \cdot \Delta'$.

Algorithm 2 Weak Suspend Interleaving

Input: $\mathbb{S}, (\Phi, k)$
Output: Program States, Δ

1 **rec function** $\aleph^{Suspend(\mathbb{S})}_{Interleave}(\Phi)$
2   **if** $\Phi{=}\epsilon$ **then**
3   **return** $[(\epsilon, k)]$
4   **else**
5         $\Delta \leftarrow []$
6   **foreach** $f \in fst(\Phi)$ **do**
7             $\Delta' {\leftarrow} \aleph^{Suspend(\mathbb{S})}_{Interleave}(D_f(\Phi))$
8             $\Delta_1 \leftarrow (f{+}\{\overline{\mathbb{S}}\}) \cdot \Delta'$
9             $\Delta_2 \leftarrow (f{+}\{\mathbb{S}\}) \cdot \{\} \cdot \Delta'$
10             $\Delta \leftarrow \Delta \cup \Delta_1 \cup \Delta_2$
11   **end**
12   **return** Δ
13 **end**
14 ▷ *Notation + unions two instants*
15 ▷ *Notation ∪ unions two program states*

▶ **Lemma 13** (Soundness of Weak Abort Interleaving)**.** *For function $\aleph^{Abort}_{Interleave}$, $\forall \mathbb{S}, \Phi, k, \Phi_{his}$, if $\Phi{=}\epsilon$, then $\Delta = [(\epsilon, k)]$, else $\Delta = \bigcup_0^{|F|}(\Phi_{his}, f{+}\{\mathbb{S}\}, 0) :: \aleph^{Abort}_{Interleave}(D_f(\Phi), \Phi_{his} \cdot (f{+}\{\overline{\mathbb{S}}\}))$ where $F{=}fst(\Phi)$.*

Proof. By induction on Φ, with Algorithm 1. ◀

▶ **Lemma 14** (Soundness of Weak Suspend Interleaving)**.** *For function $\aleph^{Suspend}_{Interleave}$, $\forall \mathbb{S}, \Phi, k$, if $\Phi{=}\epsilon$, then $\Delta = [(\epsilon, k)]$, else $\Delta = \bigcup_0^{|F|}((f{+}\{\overline{\mathbb{S}}\}) \vee (f{+}\{\mathbb{S}\}) \cdot \{\})) \cdot \Delta'$, where $F{=}fst(\Phi)$ and $\Delta'{=}\aleph^{Suspend}_{Interleave}(D_f(\Phi))$.*

Proof. By induction on Φ, with Algorithm 2. ◀

D Soundness of the Forward Rules

$\forall p, \mathcal{E}$, if $\mathcal{E} \vdash \langle h, k\rangle\ p\ \langle H', K'\rangle$, and $\varphi \models h$,
and $p \xrightarrow[\mathcal{E}]{e_0, 0}{}^{*}\ p' \xrightarrow[\mathcal{E}]{\emptyset, 1} p_1 \xrightarrow[\mathcal{E}_1]{e_1, 0}{}^{*}\ p'_1 \xrightarrow[\mathcal{E}_1]{\emptyset, 1} p_2 \xrightarrow[\mathcal{E}_2]{e_2, 0}{}^{*}\ p'_2 \xrightarrow[\mathcal{E}_2]{\emptyset, 1} \ldots p_n \xrightarrow[\mathcal{E}_n]{e_n, 0}{}^{*}\ p'_n \xrightarrow[\mathcal{E}_n]{\emptyset, k_f} ()$,
then it implies that $\exists (h', k') \in \langle H', K'\rangle$ such that $\varphi +\!\!+ [e_0; e_1; ...; e_n] \models h'$ *and* $k'{=}k_f$.
(Note that, $p \xrightarrow{e, 0}{}^{*}\ p'$ denotes the reflexive, transitive closure of $p \xrightarrow{e, 0} p'$.)

Proof. By induction on the structure of p:

1. Emit: $\mathcal{E} \vdash \langle h, k\rangle\ emit\ \mathbb{S}\ \langle h, c{+}\{\mathbb{S}\}, k\rangle$, $\varphi \models h$ and
$emit\ \mathbb{S} \xrightarrow[\mathcal{E}+\{\mathbb{S}\}]{\{\mathbb{S}\}, 0} ()$, implying $\varphi +\!\!+ [\{\mathbb{S}\}] \models h \cdot (c{+}\{\mathbb{S}\})$, is proved.

2. Yield: $\mathcal{E} \vdash \langle h, k\rangle\ Yield\ \langle h \cdot c, \{\}, k\rangle$, $\varphi \models h$ and
$Yield \xrightarrow[\emptyset]{\{\}, 1} () \xrightarrow[\emptyset]{\{\}, 0} ()$, implying $\varphi +\!\!+ [\{\}] +\!\!+ [\{\}] \models h \cdot c \cdot \{\}$, is proved.

3. Present: $\mathcal{E} \vdash \langle h, k\rangle$ *present* $\mathbb{S}$ p q $\langle H_1, K_1\rangle \cup \langle H_2, K_2\rangle$, $\varphi \models H$, where $\mathcal{E} \vdash \langle h, c{+}\{\mathbb{S}\}, k\rangle\ \ p\ \ \langle H_1, K_1\rangle$ and $\mathcal{E} \vdash \langle h, c{+}\{\overline{\mathbb{S}}\}, k\rangle\ \ q\ \ \langle H_2, K_2\rangle$.
- When $(\mathbb{S}) \in c$, *present* $S\ \ p\ \ q \rightsquigarrow p$, the inclusion is proved by inductive hypothesis.
- When $(\overline{\mathbb{S}}) \notin c$, *present* $S\ \ p\ \ q \rightsquigarrow q$, the inclusion is proved by inductive hypothesis.

4. Sequence: $\mathcal{E} \vdash \langle h, c, k\rangle$ *seq* p q $\langle H', K'\rangle$, $\varphi \models H$, where $\mathcal{E} \vdash \langle h, k\rangle\ \ p\ \ \langle H_1, K_1\rangle$, $\mathcal{E} \vdash \langle H_1, K_1\rangle\ \ q\ \ \langle H_2, K_2\rangle$ *and* $\langle H', K'\rangle{=}\langle H_2, K_2\rangle(K_1{\leq}1)$ or $\langle H', K'\rangle{=}\langle H_1, K_1\rangle(K_1{>}1)$
- When $K_1{=}0$, *seq* $p\ \ q \xrightarrow[\mathcal{E}]{e_0,0}{}^{*} \ldots \xrightarrow[\mathcal{E}_n]{e_n,0}{}^{*}\ q \xrightarrow[\mathcal{E}']{e_n\cup f_0,0}{}^{*} \ldots \xrightarrow[\mathcal{E}'_n]{f_m,0}{}^{*}\ q_n \xrightarrow[\mathcal{E}'_n]{\emptyset,k} ()$;
therefore φ ++ $[e_0; \ldots; (e_n \cup f_0); \ldots; f_m \models H_2 \cdot C_2$.
- When $K_1{=}1$, *seq* $p\ \ q \xrightarrow[\mathcal{E}]{e_0,0}{}^{*} \ldots \xrightarrow[\mathcal{E}_n]{e_n,0}{}^{*}\ p_n \xrightarrow[\mathcal{E}]{\emptyset,1}\ q \xrightarrow[\mathcal{E}']{f_0,0}{}^{*} \ldots \xrightarrow[\mathcal{E}'_n]{f_m,0}{}^{*}\ q_n \xrightarrow[\mathcal{E}'_n]{\emptyset,k} ()$;
therefore φ ++ $[e_0; \ldots; e_n; f_0; \ldots; f_n \models H_2 \cdot C_2$.
- When $K_1{>}1$, *seq* $p\ \ q \xrightarrow[\mathcal{E}]{e_0,0}{}^{*} \ldots \xrightarrow[\mathcal{E}_n]{e_n,0}{}^{*}\ p_n \xrightarrow[\mathcal{E}_n]{\emptyset,k} ()$, therefore φ ++ $[e_0; \ldots; e_n \models H_1 \cdot C_1$.

5. Exit: $\mathcal{E} \vdash \langle h, k\rangle$ *exit* d $\langle h, c, d+2\rangle$, $\varphi{\models}H$ and *exit* $d \xrightarrow[\mathcal{E}]{\{\},d+2} ()$, implying φ ++ $[\{\}] \models h \cdot c$, is proved .

6. Trap: $\mathcal{E} \vdash \langle h, k\rangle$ *trap* p $\langle h \cdot \Delta\rangle$ and $\varphi \models H$, and $\mathcal{E} \vdash \langle \epsilon, k\rangle p \langle H, C, K\rangle$, where $\langle\Delta\rangle{=}\langle H, C, K\rangle$*when*$(K{\leq}1)$; $\langle\Delta\rangle{=}\langle H, C, 0\rangle$*when*$(K{=}2)$; $\langle\Delta\rangle{=}\langle H, C, K{-}1\rangle$*when*$(K{>}2)$
- When $K{\leq}1$: *trap* $p \rightsquigarrow p$, the entailment is proved by inductive hypothesis.
- When $K{=}2$: by [`Trap-2`], *trap* $p \xrightarrow[\mathcal{E}]{\{\},0} ()$, implying φ ++ $[\{\}] \models H \cdot C$, is proved.
- When $K{>}2$: by [`Trap-3`], *trap* $p \xrightarrow[\mathcal{E}]{\{\},K-1} ()$, implying φ ++ $[\{\}] \models H \cdot C$, is proved.

7. Await: $\mathcal{E} \vdash \langle h, k\rangle$ *await* $\mathbb{S}$ $\langle h \cdot (c||\mathbb{S}?), \{\}, k\rangle$ and $\varphi \models H$,
- When $\mathbb{S} \in \mathcal{E}$: by [`Await-1`] *await* $S \xrightarrow[\mathcal{E}]{\{\},1} () \xrightarrow[\mathcal{E}_1]{\{\},0} ()$, φ++$[\{\}]$++$[\{\}] \models h \cdot c \cdot \{\}$, is proved.
- When $(\overline{\mathbb{S}}) \notin \mathcal{E}$: by by [`Await-2`] let $\varphi' \models \mathbb{S}?$, φ ++ $[\{\}]$ ++ $\varphi' \models h \cdot c \cdot \mathbb{S}?$, is proved .

8. Async: $\mathcal{E} \vdash \langle h, k\rangle$ *async* $\mathbb{S}$ p q $\langle H', K'\rangle$ and $\varphi \models H$ where $\mathcal{E} \vdash \langle h, k\rangle\ \ (p;$ *emit* $\mathbb{S})||q\ \ \langle H', K'\rangle$ and *async* $S\ \ p\ \ q \rightsquigarrow (p;$ *emit* $\mathbb{S}||q)$, therefore the entailment is proved by inductive hypothesis.

9. Parallel: $\mathcal{E} \vdash \langle h, k\rangle$ $p||q$ $\langle h \cdot \Delta\rangle$ and $\varphi \models H$ where $\mathcal{E} \vdash \langle \epsilon, c, k\rangle\ \ p\ \ \langle H_1, K_1\rangle$ $\mathcal{E} \vdash \langle \epsilon, c, k\rangle\ \ q\ \ \langle H_2, K_2\rangle$ and $\vdash_{pm} \langle H_1, K_1\rangle || \langle H_2, K_2\rangle \rightsquigarrow \langle\Delta\rangle$.
-When p and q exit at the same instant: the entailment is proved by [*PM-Unfold*] and [*PM-EqLen*].
-When p exits with an exception, and earlier than q: the entailment is proved by [*PM-Unfold*] and [*PM-Cut*].
-When p exits earlier than q without any exceptions: the entailment is proved by [*PM-Unfold*] and [*PM-Absorb*].

10. Abort: $\mathcal{E} \vdash \langle h, k\rangle$ *abort* p $\mathbb{S}$ $\langle h \cdot \Delta\rangle$ and $\varphi \models H$, where $\mathcal{E} \vdash \langle \epsilon, c, k\rangle p \langle H, C, K\rangle$ and $\langle\Delta\rangle{=}\aleph^{Abort(\mathbb{S},C,K)}_{Interleave}(H, \epsilon)$.
By semantics rules [`Abort-1`] and [`Abort-2`]; and Lemma 13.

11. Suspend: $\mathcal{E} \vdash \langle h, k\rangle$ *suspend* p $\mathbb{S}$ $\langle h \cdot \Delta\rangle$ and $\varphi \models H$, where $\mathcal{E} \vdash \langle \epsilon, c, k\rangle\ \ p\ \ \langle H, C, K\rangle$ and $\langle\Delta\rangle{=}\aleph^{Suspend(\mathbb{S},C,K)}_{Interleave}(H)$. By semantics rules [`Suspend-1`] and [`Suspend-2`]; and Lemma 14.

◀

E Termination Proof

Proof. Let Set[$\mathcal{I}$] be a data structure representing the sets of inclusions.

We use $\mathbb{S}$ to denote the inclusions to be proved, and H to accumulate "inductive hypotheses", i.e., $S, H \in Set[\mathcal{I}]$.

Consider the following partial ordering $\succ$ on pairs $\langle S, H\rangle$:

$$\langle S_1, H_1\rangle \succ \langle S_2, H_2\rangle \text{ iff } |H_1| < |H_2| \vee (|H_1| = |H_2| \wedge |S_1| > |S_2|).$$

where $|X|$ stands for the cardinality of a set X. Let $\Rightarrow$ donate the rewrite relation, then $\Rightarrow^*$ denotes its reflexive transitive closure. For any given S_0, H_0, this ordering is well founded on the set of pairs $\{\langle S, H\rangle | \langle S_0, H_0\rangle \Rightarrow^* \langle S, H\rangle\}$, due to the fact that H is a subset of the finite set of pairs of all possible derivatives in initial inclusion.

Inference rules in our TRS given in Sec. 5.2 transform current pairs $\langle S, H\rangle$ to new pairs $\langle S', H'\rangle$. And each rule either increases $|H|$ (Unfolding) or, otherwise, reduces $|S|$ (Axiom, Disprove, Prove), therefore the system is terminating.

◀

F Soundness Proof

Proof. For each inference rules, if inclusions in their premises are valid, and their side conditions are satisfied, then goal inclusions in their conclusions are valid.

1. Axiom Rules:

$$\frac{[\texttt{Bot-LHS}]}{\Gamma \vdash \bot \sqsubseteq \Phi} \qquad \frac{\overset{[\texttt{Bot-RHS}]}{\Phi \neq \bot}}{\Gamma \vdash \Phi \not\sqsubseteq \bot} \qquad \frac{\overset{[\texttt{Disprove}]}{\delta(\Phi_1) \wedge \neg\delta(\Phi_2)}}{\Gamma \vdash \Phi_1 \not\sqsubseteq \Phi_2} \qquad \frac{\overset{[\texttt{Prove}]}{fst(\Phi_1) = \{\}}}{\Gamma \vdash \Phi_1 \sqsubseteq \Phi_2}$$

- It is easy to verify that antecedent of goal entailments in the rule [`Bot-LHS`] is unsatisfiable. Therefore, these entailments are evidently valid.
- It is easy to verify that consequent of goal entailments in the rule [`Bot-RHS`] is unsatisfiable. Therefore, these entailments are evidently invalid.

2. Disprove Rules:

- It's straightforward to prove soundness of the rule [`Disprove`], Given that Φ_1 is nullable, while Φ_2 is not nullable, thus clearly the antecedent contains more traces than the consequent. Therefore, these entailments are evidently invalid.

3. Prove Rules:

$$\frac{(\Phi_1 \sqsubseteq \Phi_3) \in \Gamma \qquad (\Phi_3 \sqsubseteq \Phi_4) \in \Gamma \qquad (\Phi_4 \sqsubseteq \Phi_2) \in \Gamma}{\Gamma \vdash \Phi_1 \sqsubseteq \Phi_2} \; [\texttt{Reoccur}]$$

- For the rule [`Prove`], we consider an arbitrary model, φ such that: $\varphi \models \Phi_1$. Given the side conditions from the promises, we get $\varphi \models \Phi_1$. When the *fst* set of Φ_1 is empty, Φ_1 is possible $\bot$ or ϵ; and Φ_2 is nullable. For both cases, the inclusion is proved .

- For the rule [`Reoccur`], we consider an arbitrary model, φ such that: $\varphi \models \Phi_1$. Given the promises that $\Phi_1 \sqsubseteq \Phi_3$, we get $\varphi \models \Phi_3$; Given the promise that there exists a hypothesis $\Phi_3 \sqsubseteq \Phi_4$, we get $\varphi \models \Phi_4$; Given the promises that $\Phi_4 \sqsubseteq \Phi_2$, we get $\varphi \models \Phi_2$. Therefore, the inclusion is proved .

4. Inductive Unfolding Rule:

$$\frac{F = \mathit{fst}(\Phi_1) \quad \Gamma'{=}\Gamma, (\Phi_1 \sqsubseteq \Phi_2) \quad \forall I \in F.(\Gamma' \vdash D_I(\Phi_1) \sqsubseteq D_I(\Phi_2))}{\Gamma \vdash \Phi_1 \sqsubseteq \Phi_2} \quad [\texttt{Unfold}]$$

- For the rule [`Unfold`], we consider an arbitrary model, φ_1 and φ_2 such that: $\varphi_1 \models \Phi_1$ and $\varphi_2 \models \Phi_2$. For an arbitrary instant I, let ${\varphi_1}' \models \mathtt{I}^{-1}[\![\Phi_1]\!]$; and ${\varphi_2}' \models \mathtt{I}^{-1}[\![\Phi_2]\!]$.
Case 1), $\mathtt{I} \notin F$, ${\varphi_1}' \models \perp$, thus automatically ${\varphi_1}' \models D_{\mathtt{I}}(\Phi_2)$;
Case 2), $\mathtt{I} \in F$, given that inclusions in the rule's premise is proved , then ${\varphi_1}' \models D_{\mathtt{I}}(\Phi_2)$.
By Definition 4, since for all I, $D_{\mathtt{I}}(\Phi_1) \sqsubseteq D_{\mathtt{I}}(\Phi_2)$, the conclusion is proved .

All the inference rules used in the TRS are sound, therefore the TRS is sound. ◀