A Synchronous Effects Logic for Temporal Verification of Pure Esterel

Yahui Song and Wei-Ngan Chin

School of Computing
National University of Singapore, Singapore
{yahuis,chinwn}@comp.nus.edu.sg

Abstract. Esterel is an imperative synchronous language that has found success in many safety-critical applications. Its precise semantics makes it natural for programming and reasoning. Existing techniques tackle either one of its main challenges: correctness checking or temporal verification. To resolve the issues simultaneously, we propose a new solution via a Hoare-style forward verifier and a term rewriting system (TRS) on *Synced Effects*. The first contribution is, by deploying a novel effects logic, the verifier computes the deterministic program behaviour via construction rules at the source level, defining program evaluation syntactically. As a second contribution, by avoiding the complex translation from LTL formulas to Esterel programs, our purely algebraic TRS efficiently checks temporal properties described by expressive Synced Effects. To demonstrate our method's feasibility, we prototype this logic; prove its correctness; provide experimental results, and a number of case studies.

1 Introduction

Esterel [6] is a synchronous programming language for the development of complex reactive systems. Its high-level imperative style allows the simple expression of parallelism and preemption, making it natural for programmers to specify and reason about control-dominated model designs. Esterel has found success in many safety-critical applications such as nuclear power plant control software.

The success with real-time and embedded systems in domains that need strong guarantees can be partially attributed to its precise semantics and computational model. There exist two main semantics for Esterel [4]: (i) the *operational semantics*: is a small-step semantics, a procedure for running a whole program defined by an interpretation scheme. It analyses control flow and signals propagation in the reaction; and (ii) the *circuit semantics*: translates Esterel programs into constructive boolean digital circuits, i.e., systems of equations among boolean variables. *Existing semantics are particularly useful for code compilation/optimization or tracking the execution, but not ideal for compositional reasoning in terms of the source program.*

Esterel treats computation as a series of deterministic reactions to external signals. All parts of a reaction complete in a single, discrete-time step called an *instant*. Besides, instants exhibit deterministic concurrency; each reaction

may contain concurrent threads without execution order affecting the result of the computation. Primitives constructs execute in zero time except for the pause statement. Hence, time flows as a sequence of logical instants separated by explicit pauses. In each instant, several elementary instantaneous computations take place simultaneously.

To maintain determinism and synchrony, evaluation in one thread of execution may affect code arbitrarily far away in the program. In another words, there is a strong relationship between signal status and control propagation: a signal status determines which branch of a present test is executed, which in turn determines which emit statements are executed (See Sec. 3.1 for the language syntax). In this paper, we tackle Esterel's *Logical Correctness* issue, caused by these non-local executions, which is simply the requirement that there exists precisely **one** status for each signal that respects the coherence law. For example:

```
1 signal S1 in
2 present S1 then nothing else emit S1 end present end signal
```

Consider the program above. If the local signal S1 were *present*, the program would take the first branch of the condition, and the program would terminate without having emitted S1 (nothing leaves S1 with its default value, *absent*). If S1 were absent, the program would choose the second branch and emit the signal. Both executions lead to a contradiction. Therefore there are no valid assignments of signals in this program. This program is logically incorrect.

```
1 signal S1 in
2 present S1 then emit S1 else nothing end present end signal
```

Consider the revised program above. If the local signal S1 were present, the conditional would take the first branch, and S1 would be emitted, justifying the choice of signal value. If the S1 were absent, the signal would not be emitted, and the choice of absence is also justified. Thus there are two possible assignments to the signals in this program, which is also logically incorrect.

```
1 present S1 then emit S1 else nothing end present
```

However, if S1 is an unbounded external input signal, then this program becomes logically correct, as given a certain status of the input signal, there is precisely one reaction, which satisfies the the coherence law. *Although logical correctness is decidable, there is a deep lack in the state-of-the-art semantics for Esterel [12], which is the ability to reason about unbounded input signals.* We show that our Effects logic resolves the above issues more systematically, by taking the signal statuses (both present and absent) explicitly as arithmetic path constraints and looking ahead of analyzing the whole program.

In this paper, we represent the program behaviours using Synced Effects. By deploying a novel fixpoint logic, the Hoare-style forward verifier computes all the possible execution traces. Logically incorrect programs, having none/multiple assignments for local/output signals w.r.t the same input set, will be rejected. Meantime, we present a term rewriting system (TRS) upon synced effects to support temporal verification, which is another research challenge of Esterel.

Safety properties are typically used to describe the desired properties of reactive systems. One of the widely used languages for specifying temporal behaviour and safety properties is linear-time temporal logic (LTL). Existing approaches to Esterel's temporal verification have neither achieved compositionality nor automation. One prior work [15], recursively translates LTL formula into an Esterel program whose traces correspond to the computations that violate the safety property. The program derived from the formula is then parallel composed with the given Esterel program to be verified. The composed program is compiled using Esterel tools. Lastly, an analysis of the compiler's output then indicates whether or not the property is satisfied by the original program.

In this work, we propose an alternative approach based on our effects logic, which enables a modular *local* temporal verification without any complex translation. More specifically, given a logical correct program $\mathcal{P}$, we compute its synced effects Φ, and express the temporal properties in Φ'; Our TRS efficiently checks the language inclusions $\Phi \sqsubseteq \Phi'$. To the best of the authors' knowledge, this work proposes the first algebraic TRS for Esterel and resolves the correctness checking and temporal verification at the same time. In addition, while existing works for Esterel's temporal verification have designed for a fixed set of temporal primitives such as *finally*, *next*, *until*, we show that our expressive synced effects provide us with more flexibility than existing temporal logics (Sec. 5.2).

We summarize our main contributions as follows:

1. **The Synced Effects:** We define the syntax and semantics of the Synced Effects, to be the specification language, which are sufficient to capture the Esterel program behaviours and non-trivial temporal properties (Sec. 3.2).
2. **Automated Verification System:** Targeting a pure Esterel language (Sec. 3.1), we develop a Hoare-style forward verifier (Sec. 4), to compositionally compute the program effects, and check the logical correctness with the presence of input signals. We present an effects inclusion checker (the TRS), to soundly prove temporal properties represented by synced effects (Sec. 5).
3. **Implementation and Evaluation:** We prototype the novel effects logic, prove the correctness, provide experimental results and case studies to show the feasibility of our method (Sec. 6).

Organization. Sec. 2 gives motivation examples to highlight the key methodologies and contributions. Sec. 3 formally presents a pure Esterel language, the syntax and semantics of our synced effects. Sec. 4 presents the forward verification rules and the logical correctness checking process. Sec. 5 explains the TRS for effects inclusion checking, and displays the essential auxiliary functions. Sec. 6 demonstrates the implementation and evaluation. We discuss related works in Sec. 7 and conclude in Sec. 8. Proofs can be found in the technical report [21].

2 Overview

We now give a summary of our techniques, using Esterel programs shown in Fig. 1. and Fig. 2. Our synced effects can be illustrated with the modules `close` and `manager`, which simulate the operations to constantly open and close a file.

Here, CLOSE and BTN are declared to be input/output signals. The module manager enters into a loop after declaring a local signal OPEN. Inside of the loop, it emits the signal OPEN, indicating the file is now opened; then tests on the status of signal BTN. Signals are absent by default, until they are explicitly emitted. If BTN is present, a function call to module close will be triggered, otherwise, it does nothing[1].

The input signal BTN denotes a button which can be pressed by the users, and its presence indicates the intention to close the file. Then before exiting from the loop, the manager pauses for one time instant.

The module close is obligated to simply emit the signal CLOSE after a pause, indicating the file is now closed.

```
1 module close:
2 output CLOSE;
3 /*@ requires {OPEN}
4     ensures {}.{CLOSE} @*/
5   pause; emit CLOSE
6 end module
```

Fig. 1. The close module

```
1  module manager:
2  input BTN;
3  output CLOSE;
4  /*@
5  requires {}
6  ensures ({BTN}.{CLOSE}\/{})*
7  @*/
8    signal OPEN in
9      loop
10       emit OPEN;
11       present BTN
12         then run close
13         else nothing
14       end present;
15       pause
16     end loop
17   end signal
18 end module
```

Fig. 2. The manager module

2.1 Synced Effects. We define Hoare-triple style specifications (marked in green) for each program, which leads to a compositional verification strategy, where temporal reasoning can be done locally.

Synced effects is a novel abstract semantics model for Esterel. The process control in such synchronous languages are event driven. Events, represented by signals, are emitted within the environment for instance by sensors or the users. The system generates signals in response which are either internal or external. Following this model, synced effects describe the program behaviours using *sequences of sets of signals* occurring in the macro level.

More specifically, the set of signals to be present in one logical time instance are represented within one $\{\}$. For example, the postcondition of module close, $\{\} \cdot \{\texttt{CLOSE}\}$, says that the execution leads to two time instances, and only in the second time instance, the signal CLOSE is guaranteed to be present.

Putting the temporal effects in the precondition is new, to represent the required temporal execution history. For example, the precondition of module close, $\{\texttt{OPEN}\}$ requires that before entering into this module, the signal OPEN should be emitted in the current time instance. Besides, to enhance the expressiveness, synced effects allow trace disjunctions via $\vee$ and trace repetitions via $\star$ and ω. For example, the postcondition in module manager ensures a repeating

[1] nothing is the Esterel equivalent of unit, void or skip in other languages.

pattern, in which it can be either $\{\texttt{BTN}\} \cdot \{\texttt{CLOSE}\}$ or just $\{\}$. See Sec. 3.2 for the syntax and semantics of synced effects[2].

2.2 Forward Verification. As shown in Fig. 3., we demonstrate the forward verification process of the loop in module `manager`. The current *effects state* of a program is captured in the form of $\langle\Phi\rangle$. To facilitate the illustration, we label the verification steps by 1), ..., 9). We mark the deployed verification rules in gray. The verifier invokes the TRS to check language inclusions along the way.

1) `loop`
   $\langle\{\}\rangle$
2) `emit OPEN;`
   $\langle\{\texttt{OPEN}\}\rangle$ **[FV-Emit]**
3) `present BTN then`
   $\langle\{\texttt{OPEN},\texttt{BTN}\}\rangle$ **[FV-Present]**
4) `run close`
   $\{\texttt{OPEN},\texttt{BTN}\} \sqsubseteq \{\texttt{OPEN}\}$ *(-TRS: check precondition, succeed-)*
   $\langle\{\texttt{OPEN},\texttt{BTN}\} \cdot \{\texttt{CLOSE}\}\rangle$ **[FV-Call]**
5) `else nothing`
   $\langle\{\texttt{OPEN}\}\rangle$ **[FV-Present]**
6) `end present;`
   $\langle\{\texttt{OPEN},\texttt{BTN}\} \cdot \{\texttt{CLOSE}\} \vee \{\texttt{OPEN}\}\rangle$ **[FV-Present]**
7) `pause`
   $\langle(\{\texttt{OPEN},\texttt{BTN}\} \cdot \{\texttt{CLOSE}\} \vee \{\texttt{OPEN}\}) \cdot \{\}\rangle$ **[FV-Pause]**
8) `end loop`
   $\langle(\{\texttt{OPEN},\texttt{BTN}\} \cdot \{\texttt{CLOSE}\} \vee \{\texttt{OPEN}\})^\star\rangle$ **[FV-Loop]**
9) $(\{\texttt{OPEN},\texttt{BTN}\} \cdot \{\texttt{CLOSE}\} \vee \{\texttt{OPEN}\})^\star \sqsubseteq (\{\texttt{BTN}\} \cdot \{\texttt{CLOSE}\} \vee \{\})^\star$ *(-TRS: check postcondition, succeed-)*

Fig. 3. The forward verification example for the loop in module `manager`.

The effects state 1) is the initial effects when entering into the loop. The effects state 2) is obtained by [`FV-Emit`], which simply adds the emitted signal to the current time instance. The effects states 3), 5) and 6) are obtained by [`FV-Present`], which adds the constraints upon the tested signal into the current state, and unions the effects accumulated from two branches at the end. The effects state 4) is obtained by [`FV-Call`]. Before each method call, it checks whether the current effects state satisfies the precondition of the callee method. If the precondition is not satisfied, then the verification fails, otherwise it concatenates the postcondition of the callee to the current effects. The effects state 7) is obtained by [`FV-Pause`]. It concatenates an empty time instance to the current effects, to be the new *current state*. The effects state 8) is obtained by [`FV-Loop`], which computes a deterministic fixpoint of effects, to be the invari-

[2] The signals shown in one time instance represent the *minimal* set of signals which are required/guaranteed to be there. An empty set $\{\}$ refers to any set of signals.

ant of executing the loop. After these states transformations, step 9) checks the satisfiability of the declared postcondition by invoking the TRS.

Table 1. The inclusion proving example from the postcondition checking in Fig. 3.

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{\Phi \sqsubseteq \Phi_{\texttt{post}}(\dagger)}{\mathcal{E} \cdot \Phi \sqsubseteq (\mathcal{E} \vee \bot) \cdot \Phi_{\texttt{post}}}\,[\text{REOCCUR}]
}{\{\texttt{CLOSE}\} \cdot \Phi \sqsubseteq (\{\texttt{CLOSE}\} \vee \mathcal{E}) \cdot \Phi_{\texttt{post}}}\,[\text{UNFOLD}]
}{\{\texttt{OPEN}, \texttt{BTN}\} \cdot \{\texttt{CLOSE}\} \cdot \Phi \sqsubseteq \Phi_{\texttt{post}}}\,[\text{UNFOLD}]
\qquad
\dfrac{
\dfrac{\Phi \sqsubseteq \Phi_{\texttt{post}}(\dagger)}{\mathcal{E} \cdot \Phi \sqsubseteq (\bot \vee \mathcal{E}) \cdot \Phi_{\texttt{post}}}\,[\text{REOCCUR}]
}{\{\texttt{OPEN}\} \cdot \Phi \sqsubseteq \Phi_{\texttt{post}}}\,[\text{UNFOLD}]\,[\text{UNFOLD}]
}{\Phi \sqsubseteq \Phi_{\texttt{post}}(\dagger)}
$$

where $\Phi = (\{\texttt{OPEN}, \texttt{BTN}\} \cdot \{\texttt{CLOSE}\} \vee \{\texttt{OPEN}\})^{\star}$; and $\Phi_{\texttt{post}} = (\{\texttt{BTN}\} \cdot \{\texttt{CLOSE}\} \vee \{\})^{\star}$

2.3 The TRS. Our TRS is obligated to check the inclusion among synced effects, an extension of Antimirov and Mosses's algorithm. Antimirov and Mosses [3] present a term rewriting system for deciding the inequalities of regular expressions, based on a complete axiomatic algorithm of the algebra of regular sets. Basically, the rewriting system decides inequalities through an iterated process of checking the inequalities of their *partial derivatives* [2]. There are two important rules: [DISPROVE], which infers false from trivially inconsistent inequalities; and [UNFOLD], which applies Theorem 1 to generate new inequalities. $\texttt{D}_{\texttt{a}}(\texttt{r})$ is the partial derivative of $\texttt{r}$ w.r.t the instance $\mathbf{a}$. (Σ is the whole set of the alphabet.)

Theorem 1 (Regular Expressions Inequality (Antimirov)). *For regular expressions* $\texttt{r}$ *and* $\texttt{s}$*,* $\texttt{r} \preceq \texttt{s} \Leftrightarrow (\forall \texttt{a} \in \Sigma).\ \texttt{D}_{\texttt{a}}(\texttt{r}) \preceq \texttt{D}_{\texttt{a}}(\texttt{s})$.

Extending to the inclusions among synced effects, we present the rewriting process by our TRS in Table 1., for the postcondition checking shown in Fig. 3. We mark the rules of the inference steps in gray. Note that time instance $\{\texttt{OPEN}, \texttt{BTN}\}$ entails $\{\texttt{BTN}\}$ because the former contains more constraints. We formally define the subsumption for time instances in Definition 3. Intuitively, we use [DISPROVE] wherever the left-hand side (LHS) is *nullable*[3] while the right-hand side (RHS) is not. [DISPROVE] is the heuristic refutation step to disprove the inclusion early, which leads to a great efficiency improvement.

Termination is guaranteed because the set of derivatives to be considered is finite, and possible cycles are detected using *memorization*. The rule [REOCCUR] finds the syntactic identity, as a companion, of the current open goal, as a bud, from the internal proof tree [9]. (We use (†) in Fig. 3. to indicate such pairings.)

3 Language and Specifications

In this section, we first introduce a pure Esterel language and then depict our Synced Effects as the specification language.

[3] If the event sequence is possibly empty, i.e. contains $\mathcal{E}$, we call it nullable, formally defined in Definition 1.

3.1 The Target Language: Pure Esterel

In this work, we consider the Esterel v5 dialect [4,5] endorsed by current academic compilers, shown in Fig. 4. Pure Esterel is the subset of the full Esterel language where data variables and data-handling primitives are abstracted away. We shall concentrate on the pure Esterel language, as in this work, we are mainly interested in signal status and control propagation, which are not related to data.

(*Program*)	$\mathcal{P}$ ::=	`meth`*	(*Basic Types*) τ ::= `IN` | `OUT` | `INOUT`
(*Module Def.*)	`module` ::=	`x` $(\tau\ \mathbf{S})^*$ $\langle$**requires** $\Phi_{\texttt{pre}}$ **ensures** $\Phi_{\texttt{post}}\rangle$ **p**	
(*Statement*)	**p** **q** ::=	`nothing` | `pause` | `emit` **S** | `present` **S p q**	
		| **p** ; **q** | `loop` **p** | **p** || **q** | `trap` **T p** | `exit` $\mathbf{T}_{\texttt{d}}$	
		| `signal` **S p** | `run x` $(\mathbf{S})^*$ | `assert` Φ	

S $\in$ signal variables x, **T** $\in$ **var** (*Finite List*) $*$ (Depth)$\texttt{d} \in \mathbb{Z}$

Fig. 4. Pure Esterel Syntax.

Here, we regard **S**, `x` and **T** as meta-variables. Basic signal types include `IN` (input signals), `OUT` (output signals), `INOUT` (the signals used to be both input and output). **var** represents the countably infinite set of arbitrary distinct identifiers. We assume that programs we use are well-typed conforming to basic types τ. A program $\mathcal{P}$ comprises a list of method declarations `meth`*.

Each module `meth` has a name `x`, a list of well-typed arguments $(\tau\ \mathbf{S})^*$, a statement-oriented body **p**, also is associated with a precondition $\Phi_{\texttt{pre}}$ and a postcondition $\Phi_{\texttt{post}}$. (The syntax of effects specification Φ is given in Fig. 6.)

Following the language constructs formally defined in Fig. 4., we describe how signals are emitted and how control is transmitted between statements [4]:
- The statement `nothing` terminates instantaneously.
- The statement `pause` pauses exactly one logical instant and terminates in the next instant.
- The statement `emit` **S** broadcasts the signal **S** to be set to present and terminates instantaneously. The emission of **S** is valid for the current instant only.
- The statement `present` **S p q** immediately starts **p** if **S** is present in the current instant; otherwise it starts **q** when **S** is absent.
- The sequence statement **p** ; **q** immediately starts **p** and behaves as **p** as long as **p** remains active. When **p** terminates, control is passed instantaneously to **q**, which determines the behaviour of the sequence from then on. If **p** exits a `trap`, so does the whole sequence, **q** being discarded in this case. **q** is never started if **p** always pauses. *(Notice that '*`emit` ***S1*** ; `emit` ***S2****' emits* ***S1*** *and* ***S2*** *simultaneously and terminates instantaneously.)*
- The statement `loop` **p** immediately starts its body **p**. When **p** terminates, it is immediately restarted. If **p** exits a trap, so does the whole loop. The body of a loop is not allowed to terminate instantaneously when started, i.e., it must execute either a pause or an exit statement. For example, '`loop emit` **S**' is not a

correct program. A loop statement never terminates, but it is possible to escape from the loop by enclosing it within a trap and executing an exit statement.
- The parallel statement **p** || **q** starts **p** and **q** in parallel. The parallel statement remains active as long as one of its branches remains active unless some branch exits a trap. The parallel statement terminates when both **p** and **q** are terminated. The branches can terminate in different instants, the parallel waiting for the last one to terminate. Parallel branches may simultaneously exit traps. If, in some instant, one branch exits a trap **T** or both branches exit the same trap **T**, then the parallel exits **T**. If both statements exit distinct traps **T** and **U** in the same instant, then the parallel only exits the higher prioritized one.
- The statement `trap` **T** **p** defines a lexically scoped exit point **T** for **p**. When the trap statement starts, it immediately starts its body **p** and behaves as **p** until termination or exit. If **p** terminates, so does the trap statement. If **p** exits **T**, then the trap statement terminates instantaneously. If **p** exits an inner trap **U**, this exit is propagated by the trap statement.
- The statement `exit` $\mathbf{T}_{\mathtt{d}}$ instantaneously exits the trap **T** with a depth `d`. The corresponding trap statement is terminated unless an outermost trap is concurrently exited, as an outer trap has a higher priority when being exited concurrently. For example, as shown in Fig. 5., such an encoding of exceptions for Esterel was first advocated for by Gonthier [13]. As usual, we make depths value `d` explicit. Here, **T1** has depth 1 because of the declaration of trap **U** in the middle; **U0** and **T0** have depth 0 because they are directly enclosed by the trap **U** and **T** respectively; **V3** has depth 3 corresponding to an outer trap, defined outside of this code segment. Therefore Fig. 5. ends up with exiting outermost trap **V3**.

```
1 trap T in
2   trap U in
3     [   exit T1
4      || exit U0
5      || exit V3]
6   end trap;
7   exit T0
8 end trap
```

Fig. 5. Nested Traps

- The statement `signal` **S** **p** starts its body **p** with a fresh signal **S**, overriding any that might already exist.
- The statement `run` `x`($\mathbf{S}^*$) is a call to module `x` providing the list of IO signals.
- The statement `assert` Φ is used to guarantee the temporal property Φ asserted at a certain point of the programs.

3.2 The Specification Language: Synced Effects

We present the syntax of our Synced Effects in Fig. 6. Effects Φ can be recursively constructed by *nil* ($\bot$); an empty trace $\mathcal{E}$; one time instant represented by `I`;

$$\begin{array}{rrcl} (\textit{Synced Effects}) & \Phi & ::= & \bot \mid \mathcal{E} \mid \mathtt{I} \mid \Phi \cdot \Phi \mid \Phi \vee \Phi \mid \Phi^{\star} \mid \Phi^{\omega} \\ (\textit{Time Instant}) & \mathtt{I} & ::= & (\mathbf{S} \mapsto \theta)^{*} \\ (\textit{Status}) & \theta & ::= & \mathtt{present} \mid \mathtt{absent} \end{array}$$

$(\textit{Omega})\ \omega \qquad (\textit{Kleene Star})\ \star \qquad (\textit{Finite List})\ *$

Fig. 6. Synced Effects.

effects concatenation $\Phi \cdot \Phi$; effects disjunction $\Phi \vee \Phi$; Kleene star $\star$, a multiple times repetition of the effects Φ (possibly $\mathcal{E}$); Omega ω, an infinite repetition of the effects Φ. One time instant is constructed by a list of mappings from signals to status, recording the current status of all the signals, and will be overwritten if there is a new status of a signal had been determined. The status of a signal can be either `present` or `absent`.

Semantic Model of Effects. To define the model, we use φ (*a trace of sets of signals*) to represent the computation execution, indicating the sequential constraint of the temporal behaviour. Let $\varphi \models \Phi$ denote the model relation, i.e., the linear temporal sequence φ satisfies the synced effects Φ.

As shown in Fig. 7., we define the semantics of our synced effects. We use [] to represent the empty sequence; ++ to represent the append operation of two traces; [`I`] to represent the sequence only contains one time instant.

`I` is a list of mappings from signals to status. For example, the time instance $\{\mathbf{S}\}$ indicates the fact that signal $\mathbf{S}$ is present regardless of the status of other non-mentioned signals, i.e., the set of time instances which at least contain $\mathbf{S}$ to be present. Any time instance contains contradictions, such as $\{\mathbf{S}, \overline{\mathbf{S}}\}$, will lead to false, as a signal $\mathbf{S}$ can not be both present and absent. We use the overline on top of the signal to denote the constraint of being absent.

$$
\begin{array}{lll}
\varphi \models \mathcal{E} & \textit{iff} & \varphi=[] \\
\varphi \models \mathtt{I} & \textit{iff} & \varphi=[\mathtt{I}] \\
\varphi \models \Phi_1 \cdot \Phi_2 & \textit{iff} & \textit{there exist } \varphi_1, \varphi_2,\ \varphi=\varphi_1{+}{+}\varphi_2 \textit{ and } \varphi_1 \models \Phi_1 \textit{ and } \varphi_2 \models \Phi_2 \\
\varphi \models \Phi_1 \vee \Phi_2 & \textit{iff} & \varphi \models \Phi_1 \textit{ or } \varphi \models \Phi_2 \\
\varphi \models \Phi^{\star} & \textit{iff} & \varphi \models \mathcal{E} \textit{ or} \\
 & & \textit{there exist } \varphi_1, \varphi_2,\ \varphi=\varphi_1{+}{+}\varphi_2 \textit{ and } \varphi_1 \models \Phi \textit{ and } \varphi_2 \models \Phi^{\star} \\
\varphi \models \Phi^{\omega} & \textit{iff} & \textit{there exist } \varphi_1, \varphi_2,\ \varphi=\varphi_1{+}{+}\varphi_2 \textit{ and } \varphi_1 \models \Phi \textit{ and } \varphi_2 \models \Phi^{\omega} \\
\varphi \models \bot & \textit{iff} & \textit{false}
\end{array}
$$

Fig. 7. Semantics of Effects.

4 Automated Forward Verification

An overview of our automated verification system is given in Fig. 8. It consists of a Hoare-style forward verifier and a TRS. The inputs of the forward verifier are Esterel programs annotated with temporal specifications written in Synced Effects (cf. Fig. 2.). The input of the TRS is a pair of effects LHS and RHS, referring to the inclusion LHS $\sqsubseteq$ RHS to be checked *(LHS refers to left-hand side effects, and RHS refers to right-hand side effects.)*. Besides, the verifier calls the TRS to prove produced inclusions, i.e., between the current effects states and pre/post conditions or assertions (cf. Fig. 3.).

The TRS will be explained in Sec. 5. In this section, we mainly present the forward verifier by introducing the forward verification rules. These rules transfer program states and systematically accumulate the effects based on the syntax of each statement. We present the intermediate representation of program states in Fig. 9.

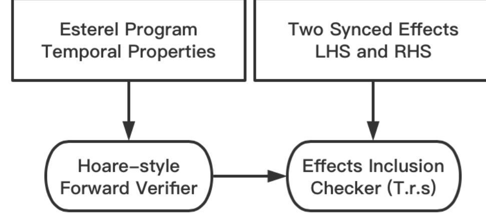

Fig. 8. System Overview.

$$\begin{array}{rrcl} (\textit{Program States}) & \Delta & ::= & \langle \underline{\Phi},\ \pi \wedge \phi,\ \mathtt{k} \rangle \\ (\textit{Intermediate Synced Effects}) & \underline{\Phi} & ::= & \bot \mid \mathcal{E} \mid \pi \wedge \phi \mid \underline{\Phi} \cdot \underline{\Phi} \mid \underline{\Phi} \vee \underline{\Phi} \mid \underline{\Phi}^{\star} \mid \underline{\Phi}^{\omega} \\ (\textit{Current Time Instant}) & \pi \wedge \phi & ::= & (\mathbf{S} = \theta)^{*} \wedge (\mathbf{S} \mapsto \theta)^{*} \end{array}$$

Fig. 9. Intermediate Representation for then Program States.

Program states Δ are represented by a tuple, where the first element ($\underline{\Phi}$) represents the trace of *history*; the second element represents the *current* time instant containing the path constraints (π) and signal assignments (ϕ)[4]; the third element ($\mathtt{k}$) represents the completion code, keeping track of the exits from nested traps [23]. Let ϱ be the environment containing all the local and output signals.

4.1 Forward Rules.

As `nothing` is the Esterel equivalent of unit, void or skip in other languages, the rule [`FV-Nothing`] simply obtains the next program state by inheriting the current program state.

$$\frac{}{\varrho \vdash \langle \underline{\Phi}, \pi \wedge \phi, \mathtt{k} \rangle\ \mathtt{nothing}\ \langle \underline{\Phi}, \pi \wedge \phi, \mathtt{k} \rangle}\ [\mathtt{FV\text{-}Nothing}]$$

The rule [`FV-Emit`] updates the current assignment of signal **S** to `present`; keeps the history trace and completion code unchanged.

$$\frac{\phi' = \phi[\mathbf{S} \mapsto \mathtt{present}]}{\varrho \vdash \langle \underline{\Phi}, \pi \wedge \phi, \mathtt{k} \rangle\ \mathtt{emit}\ \mathbf{S}\ \langle \underline{\Phi}, \pi \wedge \phi', \mathtt{k} \rangle}\ [\mathtt{FV\text{-}Emit}]$$

The rule [`FV-Pause`] archives the current time instance to the history trace; then initializes a new time instant where π' is an empty set, and all the signals from ϱ are set to default `absent`. The completion code `k` remains unchanged.

$$\frac{\pi' = \{\} \qquad \phi' = \{\mathbf{S} \mapsto \mathtt{absent} \mid \forall \mathbf{S} \in \varrho\}}{\varrho \vdash \langle \underline{\Phi}, \pi \wedge \phi, \mathtt{k} \rangle\ \mathtt{pause}\ \langle (\underline{\Phi} \cdot (\pi \wedge \phi)), \pi' \wedge \phi', \mathtt{k} \rangle}\ [\mathtt{FV\text{-}Pause}]$$

[4] The difference between $\mathbf{S} = \theta$ and $\mathbf{S} \mapsto \theta$ is: the former one denotes the constraints along the execution path, which creates *false* if there are two different status assignments to the same signal; while the latter one records the current status of one signal, and will be overwritten when the presence of a signal had been determined.

The rule [FV-Decl] obtains a new environment ϱ' by adding the local signal **S** into ϱ; sets the status of **S** to `absent` in the current time instance, then behaves as its body **p** w.r.t ϱ' and ϕ' accordingly [6].

$$\frac{\varrho'{=}\varrho, \mathbf{S} \qquad \phi'{=}\phi[\mathbf{S}{\mapsto}\texttt{absent}] \qquad \varrho' \vdash \langle \underline{\Phi}, \pi{\wedge}\phi', \texttt{k}\rangle \ \mathbf{p} \ \langle \underline{\Phi}_1, \pi_1{\wedge}\phi_1, \texttt{k}_1\rangle}{\varrho \vdash \langle \underline{\Phi}, \pi \wedge \phi, \texttt{k}\rangle \ \texttt{signal}\ \mathbf{S}\ \mathbf{p} \ \langle \underline{\Phi}_1, \pi_1 \wedge \phi_1, \texttt{k}_1\rangle} \ [\texttt{FV-Decl}]$$

The rule [FV-Present] firstly gets π' and π'' by adding the path constraints (**S**=`present`) and (**S**=`absent`) to the current time instance; then derives $\langle \underline{\Phi}_1, \pi_1{\wedge}\phi_1, \texttt{k}_1\rangle$ and $\langle \underline{\Phi}_2, \pi_2{\wedge}\phi_2, \texttt{k}_2\rangle$ from the *then* and *else* branches. We introduce `can` [12] function which intuitively determines whether the tested signal **S** can be emitted or not. If **S** *cannot* be emitted (`can`(**S**)=`false`), the final states will only come from the else branch **q**; otherwise we say **S** *can* be emitted (`can`(**S**)=`true`), the final states will be the union of both branches' execution. For example, as it shown in Fig. 10., to unblock a present expression, one must determine if the tested signal can be emitted or not. One way for that is to detect the none-occurrences of `emit` **SL**. Here, since `can`(**SL**)=`false`, the program will leave **SL** absent and emit **S2**.

```
1 signal SL in
2   present SL
3     then emit S1
4     else emit S2
5   end present
6 end signal
```

Fig. 10. Cannot

$$\frac{\begin{array}{ll} \pi' = \pi \wedge (\mathbf{S}{=}\texttt{present}) & \varrho \vdash \langle \underline{\Phi}, \pi' \wedge \phi, \texttt{k}\rangle \ \mathbf{p} \ \langle \underline{\Phi}_1, \pi_1 \wedge \phi_1, \texttt{k}_1\rangle \\ \pi'' = \pi \wedge (\mathbf{S}{=}\texttt{absent}) & \varrho \vdash \langle \underline{\Phi}, \pi'' \wedge \phi, \texttt{k}\rangle \ \mathbf{q} \ \langle \underline{\Phi}_2, \pi_2 \wedge \phi_2, \texttt{k}_2\rangle \\ \langle \Delta\rangle = \langle \underline{\Phi}_2, \pi_2 \wedge \phi_2, \texttt{k}_2\rangle & \qquad\qquad \texttt{when} \ \ \texttt{can}(\mathbf{S}){=}\texttt{false} \\ \langle \Delta\rangle = \langle \underline{\Phi}_1, \pi_1 \wedge \phi_1, \texttt{k}_1\rangle \vee \langle \underline{\Phi}_2, \pi_2 \wedge \phi_2, \texttt{k}_2\rangle & \texttt{when} \ \ \texttt{can}(\mathbf{S}){=}\texttt{true} \end{array}}{\varrho \vdash \langle \underline{\Phi}, \pi \wedge \phi, \texttt{k}\rangle \ \texttt{present}\ \mathbf{S}\ \mathbf{p}\ \mathbf{q} \ \langle \Delta\rangle} \ [\texttt{FV-Present}]$$

The rule [FV-Par] gets $\langle \underline{\Phi}_1, \pi_1{\wedge}\phi_1, \texttt{k}_1\rangle$ and $\langle \underline{\Phi}_2, \pi_2{\wedge}\phi_2, \texttt{k}_2\rangle$ by executing **p** and **q**. The `zip` function synchronises the effects from these two branches. For example, as it shown in Fig. 11., the first branch generates effects $\{\mathbf{A}\}{\cdot}\{\mathbf{B}, \mathbf{C}\}$ while the second branch generates effect $\{\mathbf{E}\} \cdot \{\mathbf{F}\} \cdot \{\mathbf{G}\}$; then the final states should be $\{\mathbf{A}, \mathbf{E}\} \cdot \{\mathbf{B}, \mathbf{C}, \mathbf{F}\} \cdot \{\mathbf{G}\}$. The `max` function returns the larger value of $\texttt{k}_1$ and $\texttt{k}_2$. When both of the branches have exits, the final $\texttt{k}_\texttt{f}$ follows the larger one, as the larger completion code indicates a higher exiting priority.

```
1 emit A; pause; emit B; emit C
2 ||
3 emit E; pause; emit F; pause; emit G
```

Fig. 11. Parallel Composition

$$\frac{\begin{array}{c} \varrho \vdash \langle \underline{\Phi}, \pi \wedge \phi, \texttt{k}\rangle \ \mathbf{p} \ \langle \underline{\Phi}_1, \pi_1{\wedge}\phi_1, \texttt{k}_1\rangle \quad \varrho \vdash \langle \underline{\Phi}, \pi \wedge \phi, \texttt{k}\rangle \ \mathbf{q} \ \langle \underline{\Phi}_2, \pi_2{\wedge}\phi_2, \texttt{k}_2\rangle \\ \langle \underline{\Phi}_\texttt{f}, \pi_\texttt{f} \wedge \phi_\texttt{f}\rangle = \texttt{zip}\ (\underline{\Phi}_1, \pi_1 \wedge \phi_1)\ (\underline{\Phi}_2, \pi_2 \wedge \phi_2) \qquad \texttt{k}_\texttt{f} = \texttt{max}(\texttt{k}_1, \texttt{k}_2) \end{array}}{\varrho \vdash \langle \underline{\Phi}, \pi \wedge \phi, \texttt{k}\rangle \ \ \mathbf{p} \ || \ \mathbf{q} \ \langle \underline{\Phi}_\texttt{f}, \pi_\texttt{f} \wedge \phi_\texttt{f}, \texttt{k}_\texttt{f}\rangle} \ [\texttt{FV-Par}]$$

The rule [FV-Seq] firstly gets $\langle \underline{\Phi}_1, \pi_1{\wedge}\phi_1, \texttt{k}_1\rangle$ by executing **p**. If there is an exceptional exit in **p**, ($\texttt{k}_1{\neq}0$), it abandons the execution of **q** completely. Otherwise, there is no exits in **p**, ($\texttt{k}_1{=}0$), it further gets $\langle \underline{\Phi}_2, \pi_2{\wedge}\phi_2, \texttt{k}_2\rangle$ by continuously executing **q**, to be the final program state.

$$\frac{\begin{array}{c}\varrho \vdash \langle \underline{\Phi}, \pi \wedge \phi, \mathtt{k}\rangle \ \mathbf{p} \ \langle \underline{\Phi}_1, \pi_1{\wedge}\phi_1, \mathtt{k}_1\rangle \quad \varrho \vdash \langle \underline{\Phi}_1, \pi_1{\wedge}\phi_1, \mathtt{k}_1\rangle \ \mathbf{q} \ \langle \underline{\Phi}_2, \pi_2{\wedge}\phi_2, \mathtt{k}_2\rangle \\ \langle \Delta\rangle = \langle \underline{\Phi}_1, \pi_1{\wedge}\phi_1, \mathtt{k}_1\rangle \qquad \mathtt{when}\ \mathtt{k}_1{\neq}0 \\ \langle \Delta\rangle = \langle \underline{\Phi}_2, \pi_2{\wedge}\phi_2, \mathtt{k}_2\rangle \qquad \mathtt{when}\ \mathtt{k}_1{=}0\end{array}}{\varrho \vdash \langle \underline{\Phi}, \pi \wedge \phi, \mathtt{k}\rangle \ \mathbf{p} \ ; \ \mathbf{q} \ \langle \Delta\rangle} \ [\mathtt{FV\text{-}Seq}]$$

The rule [FV-Loop] firstly computes a fixpoint $\langle \underline{\Phi}_1, \pi_1 \wedge \phi_1, \mathtt{k}_1\rangle$ by initializing a temporary program state $\langle \mathcal{E}, \pi \wedge \phi, \mathtt{k}\rangle$ before executing $\mathbf{p}$. If there is an exit in $\mathbf{p}$, ($\mathtt{k}_1{\neq}0$), the final state will contain a finite trace of history effects, $\underline{\Phi}$ followed by $\underline{\Phi}_1$, and a new current time instance ($\pi_1 \wedge \phi_1$). Otherwise, there is no exits in $\mathbf{p}$, the final states will contain a infinite trace of effects, constructed by ω. Then anything following an infinite trace will be abandoned. (cf. Table 2.)

$$\frac{\begin{array}{l}\qquad\quad \varrho \vdash \langle \mathcal{E}, \pi \wedge \phi, \mathtt{k}\rangle \ \mathbf{p} \ \langle \underline{\Phi}_1, \pi_1 \wedge \phi_1, \mathtt{k}_1\rangle \\ \langle \Delta\rangle = \langle \underline{\Phi} \cdot \underline{\Phi}_1, \pi_1 \wedge \phi_1, \mathtt{k}_1\rangle\rangle \qquad\qquad \mathtt{when}\ \mathtt{k}_1{\neq}0 \\ \langle \Delta\rangle = \langle \underline{\Phi} \cdot (\underline{\Phi}_1 \cdot (\pi_1 \wedge \phi_1))^{\omega},\ \mathtt{none}, \mathtt{k}_1\rangle \quad \mathtt{when}\ \mathtt{k}_1{=}0\end{array}}{\varrho \vdash \langle \underline{\Phi}, \pi \wedge \phi, \mathtt{k}\rangle \ \mathtt{loop}\ \mathbf{p} \ \langle \Delta\rangle} \ [\mathtt{FV\text{-}Loop}]$$

The rule [FV-Trap] gets $\langle \underline{\Phi}_1, \pi_1 \wedge \phi_1, \mathtt{k}_1\rangle$ by executing the trap body $\mathbf{p}$. When there is no exit from the trap body ($\mathtt{k}{=}0$), or there is an exit which can be exactly catched by the current trap ($\mathtt{k}{=}1$), we leave the final completion code to be 0. When there is an exit with a higher priority, ($\mathtt{k}{>}1$), indicating to exit from a outer trap, we get the final $\mathtt{k_f}$ by decreasing the completion code by one.

$$\frac{\begin{array}{c}\varrho \vdash \langle \underline{\Phi}, \pi \wedge \phi, \mathtt{k}\rangle \ \mathbf{p} \ \langle \underline{\Phi}_1, \pi_1 \wedge \phi_1, \mathtt{k}_1\rangle \\ \mathtt{k_f} = 0 \qquad \mathtt{when} \ \ \mathtt{k}_1{\leq}1 \\ \mathtt{k_f} = \mathtt{k}_1\text{-}1 \qquad \mathtt{when} \ \ \mathtt{k}_1{>}1\end{array}}{\varrho \vdash \langle \underline{\Phi}, \pi \wedge \phi, \mathtt{k}\rangle \ \mathtt{trap}\ \mathbf{T}\ \mathbf{p} \ \langle \underline{\Phi}_1, \pi_1 \wedge \phi_1, \mathtt{k_f}\rangle} \ [\mathtt{FV\text{-}Trap}]$$

As $\mathtt{exit}\ \mathbf{T}_\mathtt{d}$ will abort execution up to the ($\mathtt{d}{+}1$) th enclosing of the trap $\mathbf{T}$. The rule [FV-Exit] sets the value of $\mathtt{k}$ using $\mathtt{d}{+}1$.

$$\frac{}{\varrho \vdash \langle \underline{\Phi}, \pi \wedge \phi, \mathtt{k}\rangle \ \mathtt{exit}\ \mathbf{T}_\mathtt{d} \ \langle \underline{\Phi}, \pi \wedge \phi, \mathtt{d}{+}1\rangle} \ [\mathtt{FV\text{-}Exit}]$$

The rule [FV-Call] checks if the precondition of callee, $\Phi_\mathtt{pre}$, is satisfied by the current effects state; then it obtains the next program state by concatenating the postcondition to the current effects state. (cf. Table 2.)

$$\frac{\begin{array}{c}\mathtt{x} \ \ (\tau\ \mathbf{S})^{*} \ \ \langle \mathbf{requires} \ \ \Phi_\mathtt{pre}\ \mathbf{ensures} \ \ \Phi_\mathtt{post}\rangle \ \mathbf{p} \in \mathcal{P} \\ \mathit{TRS} \vdash \underline{\Phi} \cdot (\pi \wedge \phi) \sqsubseteq \Phi_\mathtt{pre} \qquad \langle \Delta\rangle = \underline{\Phi} \cdot (\pi \wedge \phi) \cdot \Phi_\mathtt{post}\end{array}}{\varrho \vdash \langle \underline{\Phi}, \pi \wedge \phi, \mathtt{k}\rangle \ \mathtt{run}\ \mathtt{x}\ (\mathbf{S})^{*}\ \mathbf{p} \ \langle \Delta\rangle} \ [\mathtt{FV\text{-}Call}]$$

The rule [FV-Assert] simply checks if the asserted property Φ' is satisfied by the current effects state. If not, a compilation error will be raised.

$$\frac{\mathit{TRS} \vdash \underline{\Phi} \cdot (\pi \wedge \phi) \sqsubseteq \Phi'}{\varrho \vdash \langle \underline{\Phi}, \pi \wedge \phi, \mathtt{k}\rangle \ \mathtt{assert}\ \Phi' \ \langle \underline{\Phi}, \pi \wedge \phi, \mathtt{k}\rangle} \ [\mathtt{FV\text{-}Assert}]$$

4.2 Correctness Checking.

Esterel assumes that the systems are deterministic. Informally, a non-deterministic system does not have a unique response to a given input event; instead, it chooses its response to the input event from a set of possible responses, and an external observer has no way to consistently predict the response that will be chosen by the system. Non-determinism corresponds to unlimited parallelism and not to any stochastic behaviour [17]. All Esterel statements and constructs are guaranteed to be deterministic, in other words, there is no way to introduce non-deterministic behaviour in an Esterel program.

To effectively check logical correctness, in this work, given an Esterel program, after been applied to the forward rules, we compute the possible execution traces in a disjunctive form; then prune the traces contain contradictions, following these principles: (cf. Fig. 12.) (i) explicit present and absent; (ii) each local signal should have only one status; (iii) lookahead should work for both present and absent; (iv) signal emissions are idempotent; (v) signal status should not be contradictory.

```
1) present S1 ⟨{}⟩
2)   then nothing ⟨{S1 ∧ S̅1̅}⟩
3)   else emit S1 ⟨{S̅1̅ ∧ S1}⟩
4) end present ⟨{false} ∨ {false}⟩
   false → logical incorrect
```

Fig. 12.

Finally, upon each assignment of inputs, programs have none or multiple output traces that will be rejected, corresponding to no-valid or multiple-valid assignments. We regard these programs, which have precisely one *safe* trace reacting to each input assignments, as logical correct.

Lemma 1 (Safe Time Instants). *Given a time instant $\pi \wedge \phi$, we define it is safe if and only if, for any signal $\boldsymbol{S}$, the binding from the path constraints $[\![\pi]\!]_{\boldsymbol{S}}$ justifies the status from the time instant mappings $[\![\phi]\!]_{\boldsymbol{S}}$; otherwise, we say it is a contradicted instant. Formally,*

$$\pi \wedge \phi \textit{ is safe iff } \nexists \boldsymbol{S}.\ [\![\pi]\!]_{\boldsymbol{S}} \neq [\![\phi]\!]_{\boldsymbol{S}}$$

Note that, the proof obligations are discharged by the Z3 solver while deciding whether a time instant $\mathtt{I}$ is safe or not, represented by $\mathtt{SAT}(\pi \wedge \phi)$.

Corollary 1 (Safe Traces). *A temporal trace Φ is safe iff all the time instants contained in the trace are safe.*

5 Temporal Verification via a TRS

The TRS is a decision procedure (proven to be terminating and sound) to check language inclusions among Synced Effects (cf. Table 1.). It is triggered i) prior to temporal property assertions; ii) prior to module calls for the precondition checking; and iii) at the end of verifying a module for the post condition checking. Given two effects $\Phi_1,\ \Phi_2$, the TRS is to decide 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 event traces in the antecedent Φ_1 are legitimately

allowed in the possible event traces from the consequent Φ_2. Γ is the proof context, i.e., a set of effects inclusion hypothesis, Φ is the history of effects from the antecedent that have been used to match the effects from the consequent. Note that Γ, Φ are derived during the inclusion proof. The inclusion checking procedure is initially invoked with Γ={} and Φ=$\mathcal{E}$. Formally,

Theorem 2 (Synced Effects Inclusion).
For synced effects Φ_1 *and* Φ_2, $\Phi_1 \sqsubseteq \Phi_2 \Leftrightarrow (\forall \mathtt{I}).\ \mathtt{D_I}(\Phi_1) \sqsubseteq \mathtt{D_I}(\Phi_2)$.

Next we provide the definitions and implementations of auxiliary functions *Nullable* (δ), *First* (fst) and *Derivative* (D) respectively. Intuitively, the Nullable function $\delta(\Phi)$ returns a boolean value indicating whether Φ contains the empty trace $\mathcal{E}$; the First function $\mathtt{fst}(\Phi)$ computes a set of possible initial time instants of Φ; and the Derivative function $\mathtt{D_I}(\Phi)$ computes a next-state effects after eliminating one time instant I from the current effects Φ.

Definition 1 (Nullable). *Given any effects* Φ*, we recursively define* $\delta(\Phi)$ *as:*

$$\delta(\Phi):\mathtt{bool}=\begin{cases}\mathtt{true} & \mathtt{if}\ \mathcal{E}\in\Phi\\ \mathtt{false} & \mathtt{if}\ \mathcal{E}\notin\Phi\end{cases},\ where$$

$$\delta(\bot)=\mathtt{false}\qquad \delta(\mathcal{E})=\mathtt{true}\qquad \delta(\mathtt{I})=\mathtt{false}\qquad \delta(\Phi_1\cdot\Phi_2)=\delta(\Phi_1)\wedge\delta(\Phi_2)$$
$$\delta(\Phi_1\vee\Phi_2)=\delta(\Phi_1)\vee\delta(\Phi_2)\qquad \delta(\Phi^\star)=\mathtt{true}\qquad \delta(\Phi^\omega)=\mathtt{false}$$

Definition 2 (First). *Let* $\mathtt{fst}(\Phi):=\{\mathtt{I}\mid(\mathtt{I}\cdot\Phi')\in[\![\Phi]\!]\}$ *be the set of initial time instants derivable from effects* Φ*. (*$[\![\Phi]\!]$ *represents all the traces contained in* Φ*.)*

$$\mathtt{fst}(\bot)=\{\}\qquad \mathtt{fst}(\mathcal{E})=\{\}\qquad \mathtt{fst}(\mathtt{I})=\{\mathtt{I}\}\qquad \mathtt{fst}(\Phi^\star)=\mathtt{fst}(\Phi)$$
$$\mathtt{fst}(\Phi^\omega)=\mathtt{fst}(\Phi)\qquad \mathtt{fst}(\Phi_1\vee\Phi_2)=\mathtt{fst}(\Phi_1)\cup\mathtt{fst}(\Phi_2)$$
$$\mathtt{fst}(\Phi_1\cdot\Phi_2)=\begin{cases}\mathtt{fst}(\Phi_1)\cup\mathtt{fst}(\Phi_2) & \mathtt{if}\ \delta(\Phi_1)=\mathtt{true}\\ \mathtt{fst}(\Phi_1) & \mathtt{if}\ \delta(\Phi_1)=\mathtt{false}\end{cases}$$

Definition 3 (Instants Subsumption). *Given two time instants* I *and* J*, we define the subset relation* I⊆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.[5] *Formally,*

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

Definition 4 (Partial Derivative). *The partial derivative* $\mathtt{D_I}(\Phi)$ *of effects* Φ *w.r.t. a time instant* I *computes the effects for the left quotient* $\mathtt{I}^{-1}[\![\Phi]\!]$.

$$\mathtt{D_I}(\bot)=\bot\qquad \mathtt{D_I}(\mathcal{E})=\bot\qquad \mathtt{D_I}(\mathtt{J})=\mathcal{E}\ (if\ \mathtt{I}\subseteq\mathtt{J})\qquad \mathtt{D_I}(\mathtt{J})=\bot\ (if\ \mathtt{I}\not\subseteq\mathtt{J})$$
$$\mathtt{D_I}(\Phi^\star)=\mathtt{D_I}(\Phi)\cdot(\Phi^\star)\qquad \mathtt{D_I}(\Phi^\omega)=\mathtt{D_I}(\Phi)\cdot(\Phi^\omega)\qquad \mathtt{D_I}(\Phi_1\vee\Phi_2)=\mathtt{D_I}(\Phi_1)\vee\mathtt{D_I}(\Phi_2)$$
$$\mathtt{D_I}(\Phi_1\cdot\Phi_2)=\begin{cases}\mathtt{D_I}(\Phi_1)\cdot\Phi_2\vee\mathtt{D_I}(\Phi_2) & \mathtt{if}\ \delta(\Phi_1)=\mathtt{true}\\ \mathtt{D_I}(\Phi_1)\cdot\Phi_2 & \mathtt{if}\ \delta(\Phi_1)=\mathtt{false}\end{cases}$$

[5] As in having more constraints refers to a smaller set of satisfying instances.

5.1 Inference Rules.

We now discuss the key steps and related inference rules that we may use in such an effects inclusion proof.

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

$$\frac{}{\Gamma \vdash \bot \sqsubseteq \Phi}\ [\texttt{Bot-LHS}] \qquad \frac{\Phi \neq \bot}{\Gamma \vdash \Phi \not\sqsubseteq \bot}\ [\texttt{Bot-RHS}]$$

II. **Disprove (Heuristic Refutation).** This rule is used to disprove the inclusions when the antecedent is nullable, while the consequent is not nullable. Intuitively, the antecedent contains at least one more trace (the empty trace) than the consequent. Therefore, the inclusion is invalid.

$$\frac{\delta(\Phi_1) \wedge \neg\delta(\Phi_2)}{\Gamma \vdash \Phi_1 \not\sqsubseteq \Phi_2}\ [\texttt{DISPROVE}]$$

III. **Prove.** We use the rule [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.

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

IV. **Unfolding (Induction).** Here comes the inductive step of unfolding the inclusions. Firstly, we make use of the auxiliary function `fst` to get a set of instants $\texttt{F}$, which are all the possible initial time 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 $\texttt{I} \in \texttt{F}$, and compute the partial derivatives (the *next-state* effects) of both the antecedent and consequent w.r.t $\texttt{I}$. The proof of the original inclusion $\Phi_1 \sqsubseteq \Phi_2$ succeeds if all the derivative inclusions succeeds.

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

V. **Normalization.** We present a set of normalization rules to soundly transfer the synced effects into a normal form, particular after getting their derivatives. Before getting into the above inference rules, we assume that the effects formulae are tailored accordingly using the lemmas shown in Table 2. We built the lemmas on top of a complete axiom system suggested by Antimirov and Mosses [3], which was designed for finite regular languages and did not include the corresponding lemmas for effects constructed by ω.

Theorem 3 (Termination). *The rewriting system TRS is terminating.*

Theorem 4 (Soundness). *Given an inclusion $\mathcal{I}$, if the TRS returns* TRUE *when proving $\mathcal{I}$, then $\mathcal{I}$ is valid.*

Proof. Both see in the technical report [21].

Table 2. Some Normalization Lemmas for synced effects.

$\Phi \vee \Phi \rightarrow \Phi$	$\bot \cdot \Phi \rightarrow \bot$	$(\mathcal{E} \vee \Phi)^{\star} \rightarrow \Phi^{\star}$
$\bot \vee \Phi \rightarrow \Phi$	$\Phi \cdot \bot \rightarrow \bot$	$(\Phi_1 \vee \Phi_2) \vee \Phi_3 \rightarrow \Phi_1 \vee (\Phi_2 \vee \Phi_3)$
$\Phi \vee \bot \rightarrow \Phi$	$\bot^{\omega} \rightarrow \bot$	$(\Phi_1 \cdot \Phi_2) \cdot \Phi_3 \rightarrow \Phi_1 \cdot (\Phi_2 \cdot \Phi_3)$
$\mathcal{E} \cdot \Phi \rightarrow \Phi$	$\bot^{\star} \rightarrow \mathcal{E}$	$\Phi \cdot (\Phi_1 \vee \Phi_2) \rightarrow \Phi \cdot \Phi_1 \vee \Phi \cdot \Phi_2$
$\Phi \cdot \mathcal{E} \rightarrow \Phi$	$\Phi^{\omega} \cdot \Phi_1 \rightarrow \Phi^{\omega}$	$(\Phi_1 \vee \Phi_2) \cdot \Phi \rightarrow \Phi_1 \cdot \Phi \vee \Phi_2 \cdot \Phi$

5.2 Expressiveness of Synced Effects.

Classical LTL extended propositional logic with the temporal operators $\mathcal{G}$ ("globally") and $\mathcal{F}$ ("in the future"), which we also write $\Box$ and $\Diamond$, respectively; and introduced the concept of fairness, which ensures an infinite-paths semantics. LTL was subsequently extended to include the $\mathcal{U}$ ("until") operator and the $\mathcal{X}$ ("next time") operator. As shown in Table 3., we are able to recursively encode these basic operators into our synced effects, making it possibly more intuitive and readable, mainly when nested operators occur[6].

Table 3. Examples for converting LTL formulae into Effects. ($\{\mathbf{A}\}, \{\mathbf{B}\}$ represent different time instants which contain signal **A** and **B** to be present.)

$\Box\mathbf{A} \equiv \{\mathbf{A}\}^{\omega}$	$\Diamond\mathbf{A} \equiv \{\}^{\star} \cdot \{\mathbf{A}\}$	$\mathbf{A}\ \mathcal{U}\ \mathbf{B} \equiv \{\mathbf{A}\}^{\star} \cdot \{\mathbf{B}\}$
$\mathcal{X}\mathbf{A} \equiv \{\} \cdot \{\mathbf{A}\}$	$\Box\Diamond\mathbf{A} \equiv (\{\}^{\star} \cdot \{\mathbf{A}\})^{\omega}$	$\Diamond\Box\mathbf{A} \equiv \{\}^{\star} \cdot \{\mathbf{A}\}^{\omega}$

Besides the high compatibility with standard first-order logic, synced effects makes the temporal verification for Esterel more scalable. It avoids the *must-provided* translation schemas for each LTL temporal operator, as to how it has been done in the prior work [15].

6 Implementation and Evaluation

To show the feasibility of our approach, we have prototyped our automated verification system using OCaml *(Source code and test suite are available from [19])*. The proof obligations generated by the verifier are discharged using constraint solver Z3. We prove termination and soundness of the TRS, our beck-end solver. We validate the front-end forward verifier for conformance, against two Esterel implementations: the Columbia Esterel Compiler (CEC) [11] and Hiphop.js [7,24]:

- **CEC:** It is an open-source compiler designed for research in both hardware and software generation from the Esterel synchronous language to C, Verilog or BLIF circuit description. It currently supports a subset of Esterel V5 [5], and provides pure Esterel programs for testing.
- **Hiphop.js:** It is a DSL for JavaScript, to facilitate the design of complex web applications by smoothly integrating Esterel and JavaScript. To enrich our test suite, we take a subset of Hiphop.js programs (as our verifier does not accept JavaScript code), and translate them into our target language.

Based on these two benchmarks, we validate the verifier using 96 pure Esterel programs, varying from 10 lines to 300 lines. We manually annotate temporal

[6] Our implementation provides a LTL-to-Effects translator.

specifications in synced effects for each of them, including both succeeded and failed instances. The remainder of this section presents some case studies.

6.1 Loops

As shown in Fig. 13., the program firstly emits signal **A**, then enters into a loop which emits signal **B** followed by a pause followed by emitting signal **C** at the end. The synced effects of it is $\Phi{=}\{\texttt{A},\texttt{B}\} \cdot (\{\texttt{B},\texttt{C}\})^{\omega}$, which says that in the first time instant, signals **A** and **B** will be present, because there is no explicit pause between the `emit` **A** and the `emit` **B**; then for the following instants (in an infinite trace), signals **B** and **C** will be present all the time, because after executing `emit` **C**, it immediately executes from the beginning of the loop, which is `emit` **B**. As we can see, Esterel's instantaneous nature requires a special distinction when it comes to loop statements, which increases the difficulty of the invariants inference, enabled by our forward verifier.

```
1  module loopTest1:
2  output A,B,C;
3  /*@
4  require {}
5  ensure {A,B}.({B,C})w
6  @*/
7  emit A;
8  loop
9    emit B; pause; emit C
10 end loop  end module
```

Fig. 13. Loop (1)

To further demonstrate the execution of loop statements, we revise the example in Fig. 13. by adding a pause at the beginning of the loop, as shown in Fig. 14. We get an different final effects $\Phi' = \{\texttt{A}\} \cdot (\{\texttt{B}\} \cdot \{\texttt{C}\})^{\omega}$, where only signal **A** is present in the first time instant; Then for the following instances (in an infinite trace), **B** and **C** are not necessarily to be present in the same instances, instead, they will take turns to be present.

```
1  module loopTest2:
2  output A,B,C;
3  /*@
4  require {}
5  ensure {A}.({B}.{C})w
6  @*/
7  emit A;
8  loop
9    pause; emit B;
10   pause; emit C
11 end loop  end module
```

Fig. 14. Loop (2)

6.2 Exception Priority

As shown in Fig. 15., the final effects for this nested trap test contains one time instance with only signal **A** is present. In the nested exception declaration, the outer traps have higher priorities over the inner traps, in other words, the exception of greater depth has always priority. In this example, when `exit T1` and `exit T2` are executed concurrently, as `exit T1` has a higher priority, the control will be transferred directly to the end of `trap T1`, ignoring the `emit` **B** in line 12. Therefore signal **A** is emitted while signal **B** is not emitted.

```
1  module trapTest:
2  output A,B;
3  /*@
4  require {}
5  ensure {A}
6  @*/
7  trap T1 in
8    trap T2 in
9      emit A;
10     (exit T1)||(exit T2)
11   end trap;
12   emit B
13 end trap
14 end module
```

Fig. 15. Exception priority

6.3 A Gain on Constructiveness

We discovered a bug from the Esterel v5 *Constructive* semantics [4]. As shown in Fig. 16., this program is detected as "non-constructive" and rejected by CEC. Because the status of **S** must be *guessed* prior to its emission; however, in present statements, it is required that the status of the tested signal must be *determined* before executing the sub-expressions.

```
1  module a_bug:
2  output S;
3  /*@
4  require {}
5  ensure {S}
6  @*/
7  signal S in
8    present S then emit S
9    else emit S
10   end present end signal
11 end module
```

Fig. 16. A Bug Found

Well, this program actually can be constructed, as the only possible assignment to signal **S** is to be present. Our verification system accepts this program, and compute the effects effectively. We take this as an advantage of using our approach to compute the fixpoint of the program effects, which essentially explores all the possible assignments to signals in a more efficient manner.

7 Related Work

7.1 Semantics of Esterel

For the Pure Esterel, an early work [6] (1992) gave two operational semantics, a macrostep logical semantics called the behavioural semantics, and a small-step logical semantics called the execution semantics. A subsequent work [4] (1999) gave an update to the logical behavioural macrostep semantics to make it more *constructive*. The logical behavioural semantics requires existence and uniqueness of a behaviour, while the constructive behavioural semantics introduces *Can* function to determine execution paths in an effective but restricted way.

Our synced effects of Esterel closely follows the work of states-based semantics [4]. In particular, we borrow the idea of internalizing state into effects using *history* and *current* that bind a partial store embedded at any level in a program. However, as the existing semantics are not ideal for compositional reasoning in terms of the source program, our forward verifier can help meet this requirement for better modularity.

A more recent work [12] (2019) proposes a calculus for Esterel, which is different from a reduction system - although there is an equational theory. The deep lack in the calculus is the ability to reason about input signals. However, as explained in Sec. 4.2, our effects logic is able to reason about the correctness with unbounded input signals. Beyond the correctness checking, the computed temporal effects are particularly convenient for the safety checking at the source code level before the runtime. With that, next, we introduce some related works of temporal verification on Esterel programs.

7.2 Temporal Verification of Esterel

In prior work [15], given a LTL formula, they first recursively translate it into an Esterel program whose traces correspond to the computations that violate

the safety formula. The program derived from the formula is then composed in parallel with the given Esterel program to be verified. The program resulting from this composition is compiled using available Esterel tools; a trivial analysis of the compiler's output then indicates whether or not the property is satisfied by the original program. By exhaustively generating all the composed program's reachable states, the Esterel compiler, in fact, performs model checking.

However, the overhead introduced by the complex translation makes it particularly inefficient when *disproving* some of the properties. Besides, it is limited by the expressive power of LTL, as whenever a new temporal logic has to be introduced, we need to design a new translation schema for it accordingly.

Informally, we are concerned with the problem: *Given a temporal property Φ', how to check that a given behaviour Φ satisfies it.* The standard approaches to this *language inclusion* problem are based on the translation of Φ and Φ' into equivalent finite state automata: $\mathcal{M}_{\mathcal{A}}$ and $\mathcal{M}_{\mathcal{B}}$; and then check emptiness of $\mathcal{M}_{\mathcal{A}} \cap \neg\mathcal{M}_{\mathcal{B}}$. However, the worst-case complexity of any efficient algorithm [10] based on such translation also goes exponential in the number of states.

In this work, we provide an alternative approach, inspired by Antimirov and Mosses' work, which presented a TRS [3] for deciding the inequalities of basic regular expressions. A TRS is a refutation method that normalizes regular expressions in such a way that checking their inequality corresponds to an iterated process of checking the inequalities of their *partial derivatives* [2]. Works based on such a TRS [3,16,14] suggest that this method is a better average-case algorithm than those based on the translation into automata. Invigorated by that, in this paper, we present a new solution of extensive temporal verification, which deploys a TRS but solves the language inclusions between Synced Effects.

Similarly, extending from Antimirov's notions of partial derivatives, prior work [8] (Broda et al., 2015) presented a decision procedure for equivalence checking between Synchronous Kleene Algebra (SKA) terms. Next, we discuss the similarities and differences between our work and [8].

7.3 Synchronous Kleene Algebra (SKA)

Kleene algebra (KA) is a decades-old sound and complete equational theory of regular expressions. Our Synced Effects theory draws similarities to SKA [18], which is KA extended with a synchrony combinator for actions. Formally, given a KA is $(\mathtt{A}, +, \cdot, \star, \mathtt{0}, \mathtt{1})$, a SKA over a finite set $\mathtt{A_B}$ is $(\mathtt{A}, +, \cdot, \times, \star, \mathtt{0}, \mathtt{1}, \mathtt{A_B})$, $\mathtt{A_B} \subseteq \mathtt{A}$. Our $\bot$ (*false*) corresponds to the $\mathtt{0}$; our $\mathcal{E}$ (empty trace) corresponds to the $\mathtt{1}$; our *time instance* containing simultaneous signals can be expressed via $\times$; and the *instants subsumption* (Definition 3) is reflected by SKA's *demanding relation*.

Presently, SKA allows the synchrony combinator $\times$ to be expressed over any two SKA terms to support concurrency. We achieve a similar outcome in Synced Effects by supporting normalization operations during trace synchronization, via a `zip` function in the forward rule of [`FV-Par`]. This leads to one major difference in the inclusion/equivalence checking procedure, whereby a TRS for SKA would have to rely on *nullable*, *first*, and *partial derivatives* for terms constructed by the added combinator $\times$, but carefully avoided by our TRS construction. While

the original equivalence checking algorithm for SKA terms in [18] has relied on well-studied decision procedures based on classical Thompson ϵ-NFA construction, [8] shows that the use of Antimirov's partial derivatives could result in better average-case algorithms for automata construction. Our present work avoided the consideration for the more general $\times$ operation from SKA and customized the TRS for inclusion (instead of equivalence) checking. These decisions led to some opportunities for improvements. Moreover, between TRS and the construction of efficient automata, we have recently shown in [20] that the former has a minor performance advantage (over a benchmark suite) when it is compared with state-of-the-art PAT [22] model checker. Improvement came from the avoidance of the more expensive automata construction process.

Apart from the synchrony combinator, we also introduced the ω constructor to explicitly distinguish infinite traces from the coarse-grained repetitive operator kleene star $\star$. The inclusion of ω constructor allows us to support non-terminating reactive systems, that are often supported by temporal specification and verification to ensure systems dependability. As a consequence, our backend TRS solver is designed to be able to soundly reason about both finite traces (inductive definition) and infinite traces (coinductive definition), using cyclic proof techniques of [9].

Another extension from the ready-made KA theory is Kleene algebra with tests (KAT), which provides solid mathematical semantic foundations for many domain-specific languages (DSL), such as NetKAT [1], designed for network programming. In KAT, actions are extended with boolean predicates and the negation operator is added accordingly. Our Synced Effects also support the boolean algebra in a similar way, since each of our signals can be explicitly specified to be either present or absent. Contradictions of such signals are also explicitly captured by $\perp$ (*false*), whenever signal unification fails.

8 Conclusion

We define the syntax and semantics of the novel Synced Effects, capable of capturing Esterel program behaviours and temporal properties. We develop a Hoare-style forward verifier to compute the program effects constructively. The verifier further enables a more systematic logical correctness checking, with the presence of unbounded input signals, which was a profound lack in prior works. We present an effects inclusion checker (the TRS) to verify the annotated temporal properties efficiently. We implement the effects logic and show its feasibility. To the best of our knowledge, our work is the first solution that automates modular verification for Esterel using an expressive effects logic.

9 Acknowlegement

We would like to thank the referees of VMCAI 2021 for most helpful advices. This work is supported by NRF grant R-252-007-A50-281 and MoE Tier-1 R-252-000-A63-114

References

1. Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: Semantic foundations for networks. *Acm sigplan notices*, 49(1):113–126, 2014.
2. Valentin Antimirov. Partial derivatives of regular expressions and finite automata constructions. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 455–466. Springer, 1995.
3. Valentin Antimirov and Peter Mosses. Rewriting extended regular expressions. *Theoretical Computer Science*, 143(1):51–72, 1995.
4. Gérard Berry. The constructive semantics of pure Esterel-draft version 3. *Draft Version*, 3, 1999.
5. Gérard Berry. *The Esterel v5 language primer: version v5_91*. Centre de mathématiques appliquées, Ecole des mines and INRIA, 2000.
6. Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.
7. Gérard Berry, Cyprien Nicolas, and Manuel Serrano. Hiphop: a synchronous reactive extension for Hop. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Programming Language and Systems Technologies for Internet Cclients*, pages 49–56, 2011.
8. Sabine Broda, Sílvia Cavadas, Miguel Ferreira, and Nelma Moreira. Deciding synchronous Kleene algebra with derivatives. In *International Conference on Implementation and Application of Automata*, pages 49–62. Springer, 2015.
9. James Brotherston. Cyclic proofs for first-order logic with inductive definitions. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, pages 78–92. Springer, 2005.
10. Martin De Wulf, Laurent Doyen, Thomas A Henzinger, and J-F Raskin. Antichains: A new algorithm for checking universality of finite automata. In *International Conference on Computer Aided Verification*, pages 17–30. Springer, 2006.
11. Stephen A. Edwards. The Columbia Esterel Compiler. `http://www.cs.columbia.edu/~sedwards/cec/`, 2006.
12. Spencer P Florence, Shu-Hung You, Jesse A Tov, and Robert Bruce Findler. A calculus for Esterel: if can, can. if no can, no can. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.
13. Georges Gonthier. *Sémantiques et modèles d'exécution des langages réactifs synchrones: application à ESTEREL*. PhD thesis, Paris 11, 1988.
14. Dag Hovland. The inclusion problem for regular expressions. *Journal of Computer and System Sciences*, 78(6):1795–1813, 2012.
15. Lalita Jagadeesan, Carlos Puchol, and James Olnhausen. Safety property verification of Esterel programs and applications to telecommunications software. 06 2000.
16. Matthias Keil and Peter Thiemann. Symbolic solving of extended regular expression inequalities. *arXiv preprint arXiv:1410.3227*, 2014.
17. Girish Keshav Palshikar. An introduction to Esterel. *Embedded Systems Programming*, 14(11), 2001.
18. Cristian Prisacariu. Synchronous kleene algebra. *The Journal of Logic and Algebraic Programming*, 79(7):608–635, 2010.
19. Yahui Song. Synced Effects Source Code. `https://github.com/songyahui/SyncedEffects.git`, 2020.

20. Yahui Song and Wei-Ngan Chin. Automated temporal verification of integrated Dependent Effects. *International Conference on Formal Engineering Methods*, 2020.
21. Yahui Song and Wei-Ngan Chin. Technical Report. `https://www.comp.nus.edu.sg/~yahuis/VMCAI2021.pdf`, 2020.
22. Jun Sun, Yang Liu, Jin Song Dong, and Jun Pang. PAT: Towards flexible verification under fairness. In *International Conference on Computer-Aided Verication*, pages 709–714. Springer, 2009.
23. Olivier Tardieu. A deterministic logical semantics for Esterel. *Electronic Notes in Theoretical Computer Science*, 128(1):103–122, 2005.
24. Colin Vidal, Gérard Berry, and Manuel Serrano. Hiphop. js: a language to orchestrate web applications. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pages 2193–2195, 2018.

A Termination Proof

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

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

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

$$\langle \mathtt{S}_1, \mathtt{H}_1 \rangle \succ \langle \mathtt{S}_2, \mathtt{H}_2 \rangle \text{ iff } |\mathtt{H}_1| < |\mathtt{H}_2| \vee (|\mathtt{H}_1| = |\mathtt{H}_2| \wedge |\mathtt{S}_1| > |\mathtt{S}_2|).$$

where $|\mathtt{X}|$ stands for the cardinality of a set $\mathtt{X}$. Let $\Rightarrow$ donate the rewrite relation, then $\Rightarrow^*$ denotes its reflexive transitive closure. For any given $\mathtt{S}_0$, $\mathtt{H}_0$, this ordering is well founded on the set of pairs $\{\langle \mathtt{S}, \mathtt{H} \rangle | \langle \mathtt{S}_0, \mathtt{H}_0 \rangle \Rightarrow^* \langle \mathtt{S}, \mathtt{H} \rangle\}$, due to the fact that $\mathtt{H}$ is a subset of the finite set of pairs of all possible derivatives in initial inclusion.

Inference rules in our TRS given in Sec. 5.1 transform current pairs $\langle \mathtt{S}, \mathtt{H} \rangle$ to new pairs $\langle \mathtt{S}', \mathtt{H}' \rangle$. And each rule either increases $|\mathtt{H}|$ (Unfolding) or, otherwise, reduces $|\mathtt{S}|$ (Axiom, Disprove, Prove), therefore the system is terminating.

B 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.

I. **Axiom Rules:**

$$\frac{}{\Gamma \vdash \bot \sqsubseteq \Phi} \ [\mathtt{Bot\text{-}LHS}] \qquad\qquad \frac{\Phi \neq \bot}{\Gamma \vdash \Phi \not\sqsubseteq \bot} \ [\mathtt{Bot\text{-}RHS}]$$

- 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.

II. **Disprove Rules:**

$$\frac{\delta(\Phi_1) \wedge \neg\delta(\Phi_2)}{\Gamma \vdash \Phi_1 \not\sqsubseteq \Phi_2} \ [\mathtt{DISPROVE}]$$

- 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 event traces than the consequent. Therefore, these entailments are evidently invalid.

III. **Prove Rules:**

$$\frac{(\Phi_1 \sqsubseteq \Phi_3) \in \Gamma \qquad (\Phi_3 \sqsubseteq \Phi_4) \in \Gamma \qquad (\Phi_4 \sqsubseteq \Phi_2) \in \Gamma}{\Gamma \vdash \Phi_1 \sqsubseteq \Phi_2} \ [\mathtt{REOCCUR}]$$

- To prove soundness of the rule [$\texttt{REOCCUR}$], 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 valid.

IV. **Unfolding Rule:**

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

- To prove soundness of the rule [$\texttt{UNFOLD}$], 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 time instance $\texttt{I}$, let ${\varphi_1}' \models \texttt{I}^{-1}[\![\Phi_1]\!]$, with $\varphi_1 = \texttt{I} \cdot {\varphi_1}'$; and ${\varphi_2}' \models \texttt{I}^{-1}[\![\Phi_2]\!]$, with $\varphi_2 = \texttt{I} \cdot {\varphi_2}'$.
Case 1), $\texttt{I} \notin \texttt{F}$, ${\varphi_1}' \models \bot$, thus automatically ${\varphi_1}' \models \texttt{D}_{\texttt{I}}(\Phi_2)$;
Case 2), $\texttt{I} \in \texttt{F}$, given that inclusions in the rule's premise is valid, then ${\varphi_1}' \models \texttt{D}_{\texttt{I}}(\Phi_2)$.
By Theorem 2, since for all $\texttt{I}$, $\texttt{D}_{\texttt{I}}(\Phi_1) \sqsubseteq \texttt{D}_{\texttt{I}}(\Phi_2)$, the conclusion is valid.

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

C Completeness Proof

Proof. Given an inclusion $\mathcal{I}$, if $\mathcal{I}$ is valid, then the TRS returns $\texttt{TRUE}$.