Automated Temporal Verification for a Mixed Sync-Async Concurrency Paradigm

ANONYMOUS
ANONYMOUS
ANONYMOUS

Abstract

To make reactive programming more concise and flexible, it is promising to deploy a mixed concurrency paradigm [12] that integrates Esterel's synchrony and preemption [9] with JavaScript's asynchrony [25]. Existing temporal verification techniques have not been designed to handle such a blending of two concurrency models. We propose a novel solution via a compositional Hoare-style forward verifier and a term rewriting system (TRS) [6] on *Synchronous Effects* (*SyncEffs*).

More specifically, we formally define a core language λ^{sync}_{async}, generalising the mixed Sync-Async paradigm. Secondly, we propose *SyncEffs*, a new effects logic, that extends *Synchronous Kleene Algebra* [31] with a *blocking-waiting* operator. Thirdly, we establish an axiomatic semantics for λ^{sync}_{async} to infer temporal behaviours of given programs, expressed in *SyncEffs*. Lastly, we present a purely algebraic TRS, to efficiently prove language inclusions between *SyncEffs*. To demonstrate the feasibility of our proposals, we prototype the verification system; prove its correctness; investigate how it can help to debug errors related to both synchronous and asynchronous programs.

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

Keywords and phrases Program proofs, Operational semantics, Temporal Verification, Reactive System Design, Mixed Sync-Async Concurrency

Digital Object Identifier 10.4230/LIPIcs...

Funding *ANONYMOUS*: ANONYMOUS

1 Introduction

As it recently proposed, reactive languages such as Hiphop.js[1] [43, 12] are designed to be based on a smooth integration of (i) Asynchronous concurrent programs, which perform interactions between components or with the environment with uncontrollable timing, such as network-based communication; (ii) Synchronous reactive programs, which react to external events in a conceptually instantaneous and deterministic way; and (iii) Preemption, the explicit cancellation and resumption of an ongoing orchestration subactivity.

"Such a combination makes reactive programming more powerful and flexible than plain JavaScript because it makes the temporal geometry of complex executions explicit instead of hidden in implicit relations between state variables." [12]

Given such a multi-paradigm concurrency model, different purposes of verification becomes engaging and challenging, which has not been intensely exploited. Existing techniques are based on transformations to: (i) convert the asynchronous chunk into semantically equivalent synchronization; then (ii) convert the synchronous program into finite-state automata (FSA); lastly (iii) reason about the behaviours based on the automata theory [17, 20, 42]. However, this approach not only lacks the modularity to reason about programs compositionally, but also suffers from the limited expressiveness, restricted by FSA.

[1] Hiphop.js is a JavaScript extension of Esterel [11] (or vice versa) for reactive web applications: `https://www-sop.inria.fr/members/Colin.Vidal/hiphop/`.

On the other hand, traditional ways of temporal verification (i) rely on a translation from specification languages, such as LTL or CSP, into FSA [40], which potentially gives rise to an exponential blow-up; or (ii) use expressive automata to model the program logic directly, which fails to capture the bugs introduced by the real implementation.

To tackle the existing issues and exploit the best of both synchronous and asynchronous concurrency models, we propose a novel temporal specification language, which enables a compositional verification via a Hoare-style forward verifier and a term rewriting system (TRS). More specifically, we specify system behaviours in the form of *SyncEffs*, which integrates the Synchronous Kleene Algebra (SKA) [31, 14] with a new operator, to provide *blocking waiting* abstractions into traditional synchronous verification.

Having the effects logic as the specification language, we are interested in the following verification problem: Given a program $\mathcal{P}$, and a temporal property Φ', does $\Phi^{\mathcal{P}} \sqsubseteq \Phi'$ holds? In a typical verification context, checking the inclusion/entailment between the 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'. In this paper, we deploy a purely algebraic term rewriting system (TRS), to check language inclusions between *SyncEffs*.

The TRS is inspired by Antimirov and Mosses' algorithm [6][2] but solving the language inclusions between *SyncEffs*. A TRS is a refutation method that normalizes expressions in such a way that checking their inclusion corresponds to an iterated process of checking the inclusion of their *partial derivatives* [5]. Works based on such a TRS [37, 6, 3, 24, 22] show its feasibility and suggest that this method is a better average-case algorithm than those based on the comparison of automata.

In summary, we present a new solution of extensive temporal verification comprising: a front-end verifier computes the program's temporal behaviour, to be the $\Phi^{\mathcal{P}}$, via inference rules at the source level; and a back-end TRS, to soundly check $\Phi^{\mathcal{P}} \sqsubseteq \Phi'$.

Our main contributions are:

1. **Language Formalisation:** we formally define a core language λ^{sync}_{async}, generalising the mixed Sync-Async concurrency models, by defining its syntax, intuitive semantics and the structural operational semantics for λ^{sync}_{async}.
2. **Specification Logic:** we propose *SyncEffs*, by defining its syntax and semantics, which extends *Synchronous Kleene Algebra* [31] with a *blocking-waiting* operator.
3. **A Sound Axiomatic Logic:** we establish an axiomatic semantics to infer the temporal behaviours, expressed in *SyncEffs*, of given λ^{sync}_{async} programs in *SyncEffs*. We prove its soundness with respect to the λ^{sync}_{async}'s operational semantics.
4. **An Efficient TRS:** We present the rewriting rules, to prove the inclusion relations between the inferred effects and the given temporal specifications, both in *SyncEffs*.
5. **Implementation and Evaluation:** We prototype the novel effects logic and the automated verification system, prove the correctness, report on a case study investigating how it can help to debug errors related to both synchronous and asynchronous programs.

Organization. Sec. 2 introduces the language features of synchronous Esterel programs, JavaScript promises, and the web reactive language HipHop.js. Sec. 3 gives motivation examples to highlight the key methodologies and contributions. Sec. 4 formally presents the core language λ^{sync}_{async}, and the syntax and semantics of *SyncEffs*. Sec. 5 presents the forward verification rules. Sec. 6 explains the TRS for effects inclusion checking, and displays the

[2] Antimirov and Mosses' algorithm was designed for deciding the inequalities of regular expressions based on an complete axiomatic algorithm of the algebra of regular sets.

essential auxiliary functions. Sec. 8 demonstrates the implementation and cases studies. We discuss related works in Sec. 9 and conclude in Sec. 10. Omitted proofs can be found in the Appendix.

2 Background: Esterel, Async-Await, and Hiphop.js

It has been an active research topic to build flexible programming paradigms for reactive systems in different domains. This section identifies: i) *Synchronous programming*, represented by Esterel [11], ii) *Asynchronous programming*, represented by JavaScript promises [25], and iii) A *mixed Sync-Async* paradigm, recently proposed by HipHop.js [12]. Meanwhile, we discuss the real-world programming challenges of each paradigm; and show how our proposal addresses them in the cases studies.

2.1 A Sense of Esterel: Synchronous and Preemptive

The principle of synchronous programming is to design a high-level abstraction where the timing characteristics of the electronic transistors are neglected [1]. Thanks to the notion of *logical ticks*: a synchronous program reacts to its environment in a sequence of ticks, and computations within a tick are assumed to be instantaneous. As one of the first few well-known synchronous languages, Esterel's 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 [11, 9, 38].

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 *instance*. Besides, instances exhibit deterministic concurrency; each reaction may contain concurrent threads without execution order affecting the computation result. Primitive constructs execute in zero time except for the *pause* statement. Hence, time flows as a sequence of logical instances separated by explicit pauses. In each instance, several elementary instantaneous computations take place simultaneously.

```
fork {emit A; pause; emit B; emit C}
 par {emit E; pause; emit F; pause; emit G}
```

The synchronous parallelism in Esterel is constructed by the *fork*{...}*par*{...} statement. It remains active as long as one of its branches remains active, and it terminates when both branches are terminated. The branches can terminate in different instances, and wait for the last one to terminate. As the above example shows, the first branch generates effects $\{A\} \cdot \{B, C\}$ while the second branch generates effects $\{E\} \cdot \{F\} \cdot \{G\}$; then the final effects should be $\{A, E\} \cdot \{B, C, F\} \cdot \{G\}$.

To maintain determinism and synchrony, evaluation in one thread of execution may affect code arbitrarily far away in the program. In other 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. 4.1 for the language syntax). The first semantic challenge of programming Esterel is the *Logical Correctness* issue, caused by these non-local executions, which is simply the requirement that there exists precisely **one** status for each signal.

For example, consider the program below:

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

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 *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.

```
signal S1 in
    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 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.

Esterel's instantaneous nature requires a special distinction when it comes to loop statements, which increases the difficulty of the effects invariants inference. As shown in Fig. 1., 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 effects of it is $\{A, B\} \cdot \{B, C\} \cdot \{B, C\} \cdot \{B, C\}...$, which says that in the first instance, signals A and B will be present, as there is no explicit pause between *emit* A and *emit* B; then for the following instances (in an infinite trace), signals B and C are present all the time, because after executing *emit* C, it immediately executes from the beginning of the loop, which is *emit* B.

```
module a_loop: output A,B,C;
  emit A;
  loop
    emit B; pause; emit C
  end loop
end module
```

Figure 1 A Loop Example in Esterel [38].

Coordination in concurrent systems can result from information exchange, using messages circulating on channels with possible implied synchronization. In Esterel, it can also result from *process preemption* [8], which is a more implicit control mechanism that consists in denying the right to work to a process, either permanently (e.g. abortion) or temporarily (e.g. suspension). Preemption is particularly important in control-dominated reactive programming, where most of the works consist of handling interrupts and controlling computation.

While the flexibility they provide us, preemption primitives often with loose or complex semantics, making abstract reasoning difficult. Most existing languages offer a small set of preemption primitives, often insufficient to program reactive systems concisely.

2.2 Asynchrony from JavaScript Promises: Async–Await

"Who can wait quietly while the mud settles? Who can remain still until the moment of action?"

– Laozi, Tao Te Ching

A number of mainstream languages, such as C#, JavaScript, Rust, and Swift, have recently added support for async–await and the accompanying promises abstraction[3], also

[3] JavaScript's asynchrony arises in situations such as web-based user-interfaces, communicating with

known as *futures* or *tasks* [13]. As an example, consider the JavaScript program in Fig. 2., it uses the `fs` module (line 1) to load the file into a variable (line 6) using async/await syntax.

```
1 const fs = require('fs').promises;
2
3 async function read (filePath) {
4   const task = fs.readFile(filePath);
5   ...// do things that do not depend on the result of the loading file
6   const data = await task; // block execution until the file is loaded
7   ... // logging or data processing of the Json file
8 }
```

Figure 2 Using Async-Await in JavaScript.

The function `read` accepts one argument, a string called `filePath`. As it is declared `async`, reading a file (line 4) does not block computations that do not depend on the result (line 5). The programmer `await`s the task when they need the result to be ready. Reading a file may raise exceptions (e.g., the file is non-existent), then the point for such an exception to emerge is where the tasks are awaited (lines 6). While await sends a signal to a task scheduler on the runtime stack, the exceptions appear to propagate in the opposite direction, from the runtime to the await sites.

Unfortunately, existing languages that support async–await do not enforce at compile time that exceptions raised by asynchronous computations are handled. The lack of this static assurance makes asynchronous programming error-prone. For example, the JavaScript compiler accepts the program above without requiring that an exception handler to be provided, which leads to program crashes if the asynchronous computation results in an exception. Such unhandled exceptions have been identified as a common vulnerability in JavaScript programs [25, 2]. Furthermore, [25, 2] display a set of other *anti-patterns* including: attempting to settle a promise multiple times; unsettled promises; unreachable reactions; and unnecessary promises, which are mostly caused by using promises without sufficient static checking.

2.3 HipHop.js – A mixture of Esterel and JavaScript

HipHop.js is a reactive web language that adds synchronous concurrency and preemption to JavaScript, which is compiled into plain JavaScript and executes on runtime environments [12]. To show the advantages of such a mixture, Fig. 3. presents a comparison between JavaScript and HipHop.js to achieve the same login button. Here, `Rname`, `Rpasswd`, `RenableLogin` are global variables to model the application's states. Dis/En-abling login is done by setting `RenableLogin`. However, while more and more features get added to the specification, state variable interactions can lead to a large number of implicit and invisible global control states.

Whereas, HipHop.js simplifies and modularizes designs, and synchronous signaling makes it possible to instantly communicate between concurrent statements to exchange data and coordination signals. Second, powerful event-driven reactive preemption borrowed from Esterel finely controls the lifetime of the arbitrarily complex program statements they apply, instantly killing them when their control events occur. More examples are discussed in the following sections.

servers through HTTP requests, and non-blocking I/O.

```
function enableLoginButton(){
  return (Rname.length >= 2 && Rpasswd.length >= 2);}

function nameKeypress(value){
  Rname = value;
  RenableLogin=enableLoginButton();}

function passwdKeypress(value){
  Rpasswd = value;
  RenableLogin=enableLoginButton();}
```

```
hiphop module Identity(in name, in passwd, out enableLogin){
  do{
    emit enableLogin(name.length >= 2 && passwd.length >= 2);
  } every(name || passwd)
}
```

Figure 3 A comparison between JavaScript (left) and HipHop.js (right) for a same login button implementation [12]. (On the right, `name`, `passwd` and `enableLogin` are reactive input/output signals.)

3 Overview

In this section, we rewrite the loading file example (c.f. Fig. 2.) in the Hiphop.js style. Based on this simple example, we highlight our main methodologies. Note that, in this work, we are mainly interested in signal status and control propagation, which are not related to data, therefore the data variables and data-handling primitives are abstracted away.

3.1 *SyncEffs*

As shown in Fig. 4., we define Hoare-triple style specifications (enclosed in `/*@ ... @*/`) for each program, which leads to a compositional verification strategy, where static checking and temporal verification can be done locally.

```
1  hiphop module readFile (in Open, out Loading, Loaded, Task1, Task2, Close)
2  /*@ requires {}^*.{Open} @*/
3  /*@ ensures {Task1}.{Task2}.{Close} @*/
4  {
5      async Loaded {
6          emit Loading;
7          // fs.readFile(filePath);
8      }
9      emit Task1; // do things that do not depend on the result of the loading file
10     await Loaded;
11     emit Task2; // logging or data processing of the Json file
12     pause; emit Close;
13 }
```

Figure 4 Rewrite the example in Fig. 2. in Hiphop.js style.

The `readFile` module asynchronously loads and processes a Json file. After emitting the signal **Loading**, it emits **Loaded** when the reading file finished. In the mean time, it does some non-relevant job **Task1** before the asynchronous code resolved. In line 10, the program waits for the signal **Loaded** to be emitted, in a blocking manner. Afterwards, it does data processing in **Task2** and then **Close**s the file.

In *SyncEffs*, we use curly braces $\{\}$ enclose a single logical-time instance. An instance is a set of signals (possible empty) with status, logically concurring at the same time (cf. Sec. 4.3). The precondition $\{\}^\star \cdot \{\texttt{Open}\}$ requires that before entering into this module, the signal **Open** should be emitted at the last instance, indicating that the file is opened. The postcondition contains a trace of the expected behaviour, which sequentially concatenates three instances: $\{\texttt{Task1}\}$, $\{\texttt{Task2}\}$ and $\{\texttt{Close}\}$, which is a over-approximation of the real behaviour.

3.2 Forward Verification

As shown in Fig. 5., we demonstrate the forward verification process of the module `readFile`. The program effects states are captured in the form of $\langle \Phi \rangle$. To facilitate the illustration, we label the verification steps by (1), ..., (10), and mark the deployed inference rules (cf. Sec. 5.1) in [gray].

1. $\langle emp \rangle$ *(– initialize the current effects, emp indicates an empty trace –)*
async *Loaded* {
    emit *Loading*;
2. $\langle \{Loading\} \rangle$ [*FV-Emit*]
} $\langle \{Loading\} \cdot \{Loaded\} \rangle$ [*FV-Async-Branch-1*]
3. $\langle emp \rangle$ *(– initialize the current effects, emp indicates an empty trace –)*
emit *Task1*;
4. $\langle \{Task1\} \rangle$ [*FV-Emit*]
await *Loaded*;
5. $\langle \{Task1\} \cdot Loaded? \rangle$ [*FV-Await*]
emit *Task2*;
6. $\langle \{Task1\} \cdot Loaded? \cdot \{Task2\} \rangle$ [*FV-Emit*]
pause;
7. $\langle \{Task1\} \cdot Loaded? \cdot \{Task2\} \cdot \{\} \rangle$ [*FV-Pause*]
emit *Close*;
8. $\langle \{Task1\} \cdot Loaded? \cdot \{Task2\} \cdot \{Close\} \rangle$ [*FV-Emit*] [*FV-Async-Branch-2*]
9. $\langle (\{Loading\} \cdot \{Loaded\}) \mid\mid (\{Task1\} \cdot Loaded? \cdot \{Task2\} \cdot \{Close\}) \rangle$ [*FV-Async*]
$\langle \{Loading, Task1\} \cdot \{Loaded, Task2\} \cdot \{Close\} \rangle$ [*Effects-Parallel-Merge*]
10. *(-TRS: check the postcondition of module readFile; Succeed, cf. Table 1.-)*
$\{Loading, Task1\} \cdot \{Loaded, Task2\} \cdot \{Close\} \sqsubseteq \{Task1\} \cdot \{Task2\} \cdot \{Close\}$

Figure 5 The forward verification example for the module `main`.

The effects states (1) and (3) are initial effects entering into the *async* statement. The effects state (5) is obtained by [FV-Await], which concatenates a blocking signal (with a question mark) to the current effects. The effects states (2), (4), (6) and (8) are obtained by [FV-Emit], which simply adds the emitted signal to the current instance. The effects state of (7) is obtained by [FV-Pause]. In step (9), we parallel compose the effects from both of the branches, and normalize the final effects. After these states transformations, step (10) checks the satisfiability of the inferred effects against the declared postcondition by invoking the TRS.

3.3 The TRS

Our TRS is obligated to check the inclusions between *SyncEffs*, which is an extension of Antimirov and Mosses's algorithm. The rewriting system in [6] decides inequalities of regular expressions (REs) through an iterated process of checking the inequalities of their *partial derivatives* [5]. There are two basic rules: [`DISPROVE`], which infers false from trivially inconsistent inequalities; and [`UNFOLD`], which applies Definition 1 to generate new inequalities.

Given Σ is the whole set of the alphabet, $D_{\underline{\mathtt{A}}}(r)$ is the partial derivative of r w.r.t the signal $\underline{\mathtt{A}}$.

▶ **Definition 1** (REs Inequality). *For REs r and s, $r \preceq s \Leftrightarrow \forall(\underline{\mathtt{A}} \in \Sigma).\ D_{\underline{\mathtt{A}}}(r) \preceq D_{\underline{\mathtt{A}}}(s)$.*

Similarly, we defined the Definition 2 for unfolding the inclusions between *SyncEffs*, where $D_I(\Phi)$ is the partial derivative of Φ w.r.t the instance I.

▶ **Definition 2** (*SyncEffs* Inclusion).
For SyncEffs Φ_1 *and* Φ_2, $\Phi_1 \sqsubseteq \Phi_2 \Leftrightarrow \forall I.\ D_I(\Phi_1) \sqsubseteq D_I(\Phi_2)$.

Next, we continue with the step (10) in Fig. 5., to demonstrate how the TRS handles arithmetic constraints and dependent values. As shown in Table 1., it automatically proves that the inferred effects of `main` satisfy the declared postcondition. We mark the rewriting rules (cf. Sec. 6) in [gray].

Table 1 The inclusion proving example.

$$
\begin{array}{cl}
\epsilon \sqsubseteq \epsilon \quad ④[\mathit{Prove}] & \\
\hline
\cancel{\{\mathit{Close}\}} \sqsubseteq \cancel{\{\mathit{Close}\}} & ③[\mathit{Unfold}] \\
\hline
\cancel{\{\mathit{Loaded}, \mathit{Task2}\}} \cdot \{\mathit{Close}\} \sqsubseteq \cancel{\{\mathit{Task2}\}} \cdot \{\mathit{Close}\} & ②[\mathit{Unfold}] \\
\hline
\cancel{\{\mathit{Loading}, \mathit{Task1}\}} \cdot \{\mathit{Loaded}, \mathit{Task2}\} \cdot \{\mathit{Close}\} \sqsubseteq \cancel{\{\mathit{Task1}\}} \cdot \{\mathit{Task2}\} \cdot \{\mathit{Close}\} & ①[\mathit{Unfold}] \\
\hline
\{\mathit{Loading}, \mathit{Task1}\} \cdot \{\mathit{Loaded}, \mathit{Task2}\} \cdot \{\mathit{Close}\} \sqsubseteq \{\mathit{Task1}\} \cdot \{\mathit{Task2}\} \cdot \{\mathit{Close}\} &
\end{array}
$$

Note that instance $\{\mathit{Loading}, \mathit{Task1}\}$ entails $\{\mathit{Task1}\}$ because the former contains more constraints. We formally define the subsumption for instances in Definition 5. Intuitively, we use [*DISPROVE*] wherever the left-hand side (LHS) is *nullable*[4] while the right-hand side (RHS) is not. [*DISPROVE*] is the heuristic refutation step to disprove the inclusion early, improving the verification efficiency. Termination is guaranteed because the set of derivatives to be considered is finite, and possible cycles are detected using *memorization* [15]. Find out more inclusion checking examples in Sec. 8

4 Language and Specifications

In this section, we present the core language λ^{sync}_{async} and the specification language *SyncEffs* by formally defining their syntax and semantics.

[4] If the event sequence is possibly empty, i.e. contains ϵ, we call it nullable, formally defined in Definition 3.

4.1 The Target Language: Syntax of λ^{sync}_{async}

To formulate the target language, we generalise the design of Hiphop.js into a core language λ^{sync}_{async}, which provides the infrastructure for mixing synchronous and asynchronous concurrency models. We here formally define the syntax of λ^{sync}_{async}, as shown in Fig. 6. The language are designed mostly based on Esterel v5 [9, 10] endorsed by current academic compilers. The statements marked as blue are generalised from the original *trap* and *exit* statements in Esterel, to allow exception handling and possible continuations. The statements marked as purple provide the asynchrony coming from the usage of JavaScript promises. In this work, we are mainly interested in signal status and control propagation, which are not related to data, therefore the data variables and data-handling primitives are abstracted away.

$$
\begin{array}{llll}
(Program) & \mathcal{P} & ::= & \overrightarrow{module} \\
(Basic\ Types) & \tau & ::= & IN \mid OUT \mid INOUT \\
(Module\ Def.) & module & ::= & nm\ \ (\overrightarrow{\tau\ S})\ \ \langle \textbf{requires}\ \ \Phi_{pre}\ \ \textbf{ensures}\ \ \Phi_{post}\rangle\ \ p \\
(Statement) & p,\ q & ::= & nothing \mid pause \mid emit\ S \mid p; q \mid p||q \mid loop\ p \\
 & & & \mid signal\ S\ in\ p \mid present\ S\ then\ p\ else\ q \mid call\ mn\ (\overrightarrow{S}) \\
 & & & \mid {\color{blue} try\ p\ with\ q} \mid {\color{blue} raise\ d} \mid {\color{purple} async\ S\ p\ q} \mid {\color{purple} await\ S}
\end{array}
$$

$(Signal\ Variables)\ S \in \Sigma \qquad x, mn \in \textbf{var} \qquad (Depth)\ d \in \mathbb{Z}$

Figure 6 λ^{sync}_{async} Syntax.

We here explain the intuitive semantics, while the axiomatic semantics model is defined in Sec. 5. The statement *nothing* in λ^{sync}_{async} is the Esterel equivalent of unit, void or skip in other languages. A thread of execution suspends itself for the current instance using the *pause* construct, and resumes when the next instance started. The statement *emit S* broadcasts the signal S to be set as present and terminates instantaneously. The emission of S is valid for the current instance only.

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. *(Notice that 'emit $S1$; emit $S2$' leads to $\{S1, S2\}$, which emits S1 and $S2$ simultaneously and terminates instantaneously.)* If p exits a *trap*, so does the whole sequence, q being discarded in this case. q is never started if p always pauses.

The parallel statement $p||q$ runs p and q in parallel. It remains active as long as one of its branches remains active. The parallel statement terminates when both p and q are terminated. The branches can terminate in different instances, and the parallel waits for the last one to terminate.

The statement *loop p* implements an infinite loop, but it is possible to be aborted or suspended by enclosing it within a preemptive statement. When p terminates, it is immediately restarted. The body of a loop is not allowed to terminate instantaneously when started, i.e., it must execute a pause statement to avoid an 'infinite instance'. For example, '*loop emit S*' is not a correct program.

The statement *signal S in p* starts p with a fresh signal **S**, overriding any that might already exist. The statement *present S p q* immediately starts p if S is present in the current instance; otherwise it starts q when S is absent. The statement *call nm* $(\overrightarrow{S})$ is a call to module *nm*, providing the list of io signals.

The statement *try p with q* is generalised from the *trap T in p* statement from Esterel, to further support the exceptions handling and possible continuations, expressed in q. These features are commonly used in asynchronous programming, yet, had never included into synchronous languages.

Similarly, the statement *raise* d is generalised from *exit* T_d statement from Esterel, which 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. In other words, the exception of greater depth always has priority. Such an encoding of exceptions for Esterel was first advocated for by Gonthier [21]. Therefore, as usual, we make depths value d explicit.

The statement *async* p S is supposed to spawn a long lasting background computation. When it completes, the asynchronous block will resume the synchronous machine. Therefore when a signal S is specified with the *async* call, it emits S when the asynchronous block completes. The statement *await* S blocks the execution and waits for the signal S to be emitted across the threads.

Prior work [23] shows that such a language combination makes reactive programming more powerful and flexible than the traditional web programming.

Meta-variables are S and *nm* . Basic signal types include *IN* for input signals, OUT for output signals and $INOUT$ for both. **var** represents the countably infinite set of arbitrary distinct identifiers. We assume that programs are well-typed conforming to basic types τ.

A program $\mathcal{P}$ comprises a list of module definitions $\overrightarrow{module}$. Here, we use the $\rightarrow$ script to denote a finite vector (possibly empty) of items. Each *module* has a name *nm*, a list of well-typed arguments $\overrightarrow{\tau\ S}$, a statement-oriented body p, associated with a precondition Φ_{pre} and a postcondition Φ_{post}. (The syntax of effects specification Φ is given in Fig. 7.)

4.2 Structural Operational Semantic of λ^{sync}_{async}

The original semantics of Esterel [8, 5, 6, 32] is given by a structural operational semantic (SOS) [23, 2], also known as micro-steps semantics,which can be written as rules of the form:

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

Here, E is an event that defines the status of all signals declared in the scope of p, e is an event composed of all the signals emitted by p in the reaction, k is the completion code returned by p, and the statement p' is called the derivative of p by the reaction.

More specifically, if p emits the signal S during the transition, then $e{=}\{S \mapsto Present\}$. Otherwise, $e{=}\emptyset$. In the SOS, at most one signal can be emitted during one micro-step. Then k is the integer completion code: when $k{=}0$, the statement completes without generating a new instance; when $k{=}1$, the statement completes with generating a new instance; when $k{>}1$, the statement completes with an exception code k.

A *nothing* statement terminates without emitting any signals and $k{=}0$. A *pause* statement terminates without emitting any signals and $k{=}1$. An *emit* S statement sets the signal S to be present and terminates with $k{=}0$.

$$nothing \xrightarrow[E]{\emptyset,0} nothing \quad (\texttt{Axiom-1-Nothing}) \qquad pause \xrightarrow[E]{\emptyset,1} nothing \quad (\texttt{Axiom-2-Pause})$$

$$emit\ S \xrightarrow[E]{\{S\mapsto Present\},0} nothing \quad (\texttt{Axiom-3-Emit})$$

If the first statement of a sequence can act, so can the sequence. If the first statement of the sequence is terminated, the second one can act. Note that: if the return code k for a statement p is 0, i.e. if p terminates, then p' will always behave as *nothing* in further instants.

If k encodes an exception raising, i.e. if $k{>}1$, the resulting statement p' is immaterial since it will always disappear by some application of rules of *try*.

$$\frac{p \xrightarrow[E]{e',k} p' \qquad (k{\neq}0)}{p;\, q \xrightarrow[E]{e',k} p';\, q}\ (\texttt{Seq-1}) \qquad \frac{p \xrightarrow[E]{e,0} p' \qquad q \xrightarrow[E]{f,k} q'}{p;\, q \xrightarrow[E]{e\cup f,k} q'}\ (\texttt{Seq-2})$$

In the rule (`Parallel`), the parallel branches are executed independently but in the same signal environment. Their output instances are merged. The rule (`Loop`) performs an instantaneous unfolding of the loop into a sequence. Note that a loop can never terminate.

$$\frac{p \xrightarrow[E]{e_1,k_1} p' \qquad q \xrightarrow[E]{e_2,k_2} q'}{p||q \xrightarrow[E]{e_1\cup e_2,max(k_1,k_2)} p'||q'}\ (\texttt{Parallel}) \qquad \frac{p; loop\ p \xrightarrow[E]{e,k} p' \quad (k \neq 0)}{loop\ p \xrightarrow[E]{e,k} p'}\ (\texttt{Loop})$$

Due to the static scope of signals, the instance E may already contain a different signal having the same name S; we introduce the notation $(E\backslash S)$ to denote the complete instance obtained by removing the S component of E, if it is present. The first rule applies when the signal is emitted by the body: the signal is then received by the body, the emitted and received status must coincide. The second rule applies when the signal is not emitted. Thus, it is not received and the previous signal status is retained from the declaration.

$$\frac{p \xrightarrow[(E\backslash S)\cup\{S\mapsto Present\}]{e'\cup\{S\mapsto Present\},k} p'}{signal\ S\ in\ p \xrightarrow[E]{e',k} signal\ S\ in\ p'}\ (\texttt{Decl-1}) \qquad \frac{p \xrightarrow[(E\backslash S)\cup\{S\mapsto Absent\}]{e',k} p'}{signal\ S\ in\ p \xrightarrow[E]{e',k} signal\ S\ in\ p'}\ (\texttt{Decl-2})$$

The rules for *present* are similar to the rules for present-then-else. If the signal is present in the current instance, the then clause is instantly executed. Otherwise, the else clause is instantly executed.

$$\frac{S \in E \qquad p \xrightarrow[E]{e,k} p'}{present\ \ S\ \ p\ \ q \xrightarrow[E]{e,k} p'}\ (\texttt{Present-1}) \qquad \frac{S \notin E \qquad q \xrightarrow[E]{e,k} q'}{present\ \ S\ \ p\ \ q \xrightarrow[E]{e,k} q'}\ (\texttt{Present-2})$$

The rule for raising an exception sets the completion code as $d+2$. The (`Try-1`) rule expresses that its body dose not raise any exceptions. The (`Try-2`) rule expresses its body terminates with an exception, and need to be handled by the continuation defined in q. The (`Try-3`) rule expresses that its body terminates with an exception, which needs to be propagated to a outer *try*.

$$\frac{k{=}d+2}{raise\ d \xrightarrow[E]{\emptyset,k} nothing}\ (\texttt{Raise}) \qquad \frac{p \xrightarrow[E]{e,k} p' \quad (k{\leq}1)}{try\ \ p\ \ with\ \ q \xrightarrow[E]{e,k} try\ \ p'\ \ with\ \ q}\ (\texttt{Try-1})$$

$$\frac{p \xrightarrow[E]{e,2} p' \quad (k{=}2) \qquad q \xrightarrow[E]{f,k} q'}{try\ \ p\ \ with\ \ q \xrightarrow[E]{e\cup f,k} q'}\ (\texttt{Try-2}) \qquad \frac{p \xrightarrow[E]{e,k} p' \quad (k{>}2)}{try\ \ p\ \ with\ \ q \xrightarrow[E]{e,k\text{-}1} p'}\ (\texttt{Try-3})$$

The (`Await-1`) rule expresses that in the current instance, it happens to have the signal S to be present, then the statement is reduced to *nothing* with completion code as 0. The (`Await-2`) rule expresses that in the current instance, there is no present signal S, then the statement keeps waiting in the next instance as well. The (`Async`) rule expresses that to emit the signal S after the completion of body p.

$$\frac{S \in E}{await\ S \xrightarrow[E]{\emptyset,0} nothing}\ (\texttt{Await-1}) \qquad \frac{S \notin E}{await\ S \xrightarrow[E]{\emptyset,1} await\ S}\ (\texttt{Await-2})$$

$$\frac{p \xrightarrow[E]{e,k_1} p' \qquad q \xrightarrow[E]{f,k_2} q'}{async\ S\ p\ q \xrightarrow[E]{e \cup f, max(k_1,k_2)} (p'; emit\ S)||q'}\ (\texttt{Async})$$

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

$$\frac{nm\ \ (\tau\ \overrightarrow{S})\ \langle \textbf{requires}\ \Phi_{pre}\ \ \textbf{ensures}\ \Phi_{post}\rangle\ p \in \mathcal{P} \qquad p \xrightarrow[E]{e,k} p'}{call\ mn\ (\overrightarrow{S}) \xrightarrow[E]{e,k} p'}\ (\texttt{Call})$$

4.3 An Effect Logic for λ^{sync}_{async}

We plant the effects specifications into the Hoare-style verification system, using Φ_{pre} and Φ_{post} to capture the temporal pre/post condition.

$$\begin{array}{rcll}
(Effects) & \Phi & ::= & \bot \mid \epsilon \mid I \mid S? \mid \Phi_1 \cdot \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid \Phi_1 || \Phi_2 \mid \Phi^\star \\
(Instance) & I & ::= & \{\} \mid \{S \mapsto \alpha\} \mid I_1 \cup I_2 \\
(Signal\ Status) & \alpha & ::= & present \mid absent \mid undef
\end{array}$$

$$(Signal\ \ Variables)\ S \in \Sigma \qquad (Blocking\ \ Waiting)\ ? \qquad (Kleene\ Star)\ \star$$

Figure 7 Syntax of the *SyncEffs*.

The syntax of the *SyncEffs* is formally defined in Fig. 7. Effects comprise *nil* ($\bot$); an empty trace ϵ; a single instance represented by I; a waiting for a single signal $S?$; sequences concatenation $\Phi_1 \cdot \Phi_2$; disjunction $\Phi_1 \vee \Phi_2$; synchronous parallelism $\Phi_1 || \Phi_2$. Effects can be constructed by $\star$, representing zero or more times repetition of a trace.

There are three possible states for a signal: present, absent or undefined. The default state of signals in a new instance is undefined. An instance I is a set of mappings from signals to their status; and it can be possible empty sets $\{\}$, indicating that there is no signal constraints for the instance.

4.4 Semantic Model of *SyncEffs*

To define the semantic model, we use φ (*a trace of sets of signals*) to represent the computation execution (or instance multi-trees, per se), 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 sequential instances defined from Φ, with φ from the following concrete domain: $\varphi \triangleq$ *list of* I (a sequence of instances).

As shown in Fig. 8., we define the semantics of *SyncEffs*. We use [] to represent the empty sequence; ++ to represent the append operation of two traces; $[I]$ to represent the sequence only contains one instance.

I is a list of mappings from signals to status. For example, the instance $\{S\}$ indicates that signal S is *Present* regardless of the status of other non-mentioned signals, i.e., instances

$$
\begin{array}{lll}
\varphi \models \epsilon & \textit{iff} & \varphi{=}[] \\
\varphi \models I & \textit{iff} & \varphi{=}[I] \\
\varphi \models S? & \textit{iff} & \exists n{\geq}0.\ \varphi{=}\{\overline{S}\}^n{+}{+}[\{S\}] \\
\varphi \models (\Phi_1{\cdot}\Phi_2) & \textit{iff} & \exists \varphi_1, \varphi_2.\ \varphi{=}\varphi_1{+}{+}\varphi_2 \textit{ and } \varphi_1 \models \Phi_1 \textit{ and } \varphi_2 \models \Phi_2 \\
\varphi \models (\Phi_1{\vee}\Phi_2) & \textit{iff} & \varphi \models \Phi_1 \textit{ or } \varphi \models \Phi_2 \\
\varphi \models (\Phi_1{||}\Phi_2) & \textit{iff} & \varphi \models \Phi_1 \textit{ and } \varphi \models \Phi_2 \\
\varphi \models \Phi^{\star} & \textit{iff} & \varphi \models \epsilon \textit{ or } \varphi \models (\Phi \cdot \Phi^{\star}) \\
\varphi \models \bot & otherwise &
\end{array}
$$

Figure 8 Semantics of the Effects Logic.

which at least contain S to be present. The signals shown in one instance represent the *minimal* set of signals which are required/guaranteed to be there. An empty set $\{\}$ represents any set of signals Any instance contains contradictions, such as $\{S, \overline{S}\}$, will lead to *false*, as a signal S can not be both present and absent.

5 Automated Forward Verification

An overview of our automated verification system is given in Fig. 9. It consists of a Hoare-style forward verifier and a TRS. The inputs of the forward verifier are λ^{sync}_{async} programs annotated with temporal specifications written in *SyncEffs*.

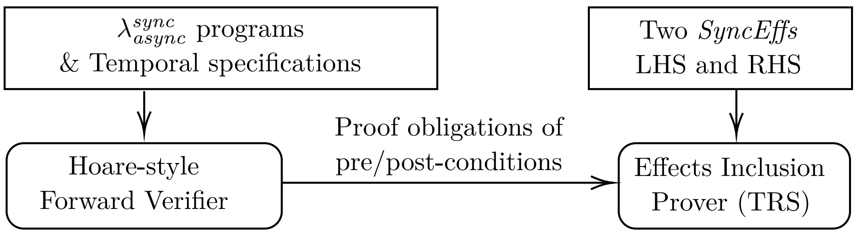

Figure 9 System Overview.

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 generated inclusions, i.e., between the effects states and pre/post conditions. The TRS will be explained in Sec. 6.

5.1 Axiomatic Semantics Model for λ^{sync}_{async}

In this section, we give an axiomatic semantics model for the core language λ^{sync}_{async}, by formalising a set of forward inductive rules. These rules transfer program states and systematically accumulate the effects syntactically. To define the model, we introduce an environment $\mathcal{E}$ and describe a program state in a three-elements tuple (h, c, k), where h

represents the trace of *history*; c represents an optional[5] *current* instance; k is the *completion code* drawn from nature numbers. They are in the following concrete domains:

$$\mathcal{E} \triangleq \overrightarrow{S} \qquad\qquad h \triangleq \Phi \qquad\qquad c \triangleq \mathit{Some}\ I \mid \mathit{None} \qquad\qquad k \in \mathbb{N}$$

The forward rules are in the form: $\mathcal{E} \vdash \langle H, C, K\rangle\ p\ \langle H', C', K'\rangle$, where $\mathcal{E}$ is the environment containing all the local and global signals; p is the given statement; $\langle H, C, K\rangle$ refers to a set of program states. The meaning of the transition rules, can be described as:

$$\langle H', C', K'\rangle = \bigcup_{i=0}^{|\langle H,C,K\rangle|-1} \langle h'_i, c'_i, k'_i\rangle \quad \mathit{where} \quad \mathcal{E} \vdash (h_i, c_i, k_i)\ \ p\ \ \langle h'_i, c'_i, k'_i\rangle.^{\,6}$$

Statement *nothing* is the Esterel equivalent of unit in other languages. Therefore the rule $[\mathit{FV\text{-}Nothing}]$ obtains the next program state by inheriting the current program state.

$$\frac{}{\mathcal{E} \vdash \langle H, C, K\rangle \mathit{nothing} \langle H, C, K\rangle}[\mathit{FV\text{-}Nothing}] \frac{C' {=} C[S \mapsto \mathit{Present}]}{\mathcal{E} \vdash \langle H, C, K\rangle \mathit{emit}\ S \langle H, C', K\rangle}[\mathit{FV\text{-}Emit}]$$

The rule $[\mathit{FV\text{-}Emit}]$ updates the current instance with signal S pointing from *Undef* to *Present*. (Note that if the current status of S is *Absent*, this rule creates a contradictory instance, indicating the logical inconsistency.)

The rule $[\mathit{FV\text{-}Pause}]$ archives the current instance to the history trace; then initializes a new current instance where all the signals from $\mathcal{E}$ are set to be undefined.

$$\frac{C' {=} \{S \mapsto \mathit{Undef} \mid \forall S \in \mathcal{E}\} \qquad H' {=} H \cdot C}{\mathcal{E} \vdash \langle H, C, K\rangle\ \mathit{pause}\ \langle H', C', K\rangle} \quad [\mathit{FV\text{-}Pause}]$$

The rule $[\mathit{FV\text{-}Local}]$ firstly constructs $\mathcal{E}'$ by adding the declared signal S into $\mathcal{E}$, then infers p's behaviour by extending the current instance with signal S pointing to *Undef*.

$$\frac{\mathcal{E}' {=} \{S\} \cup \mathcal{E} \qquad \mathcal{E}' \vdash \langle H, (S \mapsto \mathit{Undef}) :: C, K\rangle\ p\ \langle H', C', k'\rangle}{\mathcal{E} \vdash \langle H, C, K\rangle\ \mathit{signal}\ S\ \mathit{in}\ p\ \langle H', C', K'\rangle} \quad [\mathit{FV\text{-}Local}]$$

The rule $[\mathit{FV\text{-}Present}]$ enters into branches p and q after extending the current instance with the status of S pointing to *Present* and *Absent* respectively. The finial states are the union of these two possibilities.

$$\frac{\begin{array}{c}\mathcal{E} \vdash \langle H, (S{\mapsto}\mathit{Present}) :: C, K\rangle\ \ p\ \ \langle H_1, C_1, K_1\rangle \\ \mathcal{E} \vdash \langle H, (S{\mapsto}\mathit{Absent}) :: C, K\rangle\ \ q\ \ \langle H_2, C_2, K_2\rangle\end{array}}{\mathcal{E} \vdash \langle H, C, K\rangle\ \mathit{present}\ S\ \ p\ \ q\ \langle H_1, C_1, K_1\rangle \cup \langle H_2, C_2, K_2\rangle} \quad [\mathit{FV\text{-}Present}]$$

The rule $[\mathit{FV\text{-}Par}]$ gets $\langle H_1, C_1, K_1\rangle$ and $\langle H_2, C_2, K_2\rangle$ by executing p and q independently. We parallel synchronise the effects from these two branches, by deploying *parallelMerge*. The *parallelMerge* algorithm is presented in Algorithm 1. The deployed auxiliary functions, such as $\mathit{fst}(\Phi)$ and $D_I(\Phi)$ are formally definedf in Sec. 6.1.

$$\frac{\begin{array}{c}\mathcal{E} \vdash \langle H, C, K\rangle\ \ p\ \ \langle H_1, C_1, K_1\rangle \qquad \mathcal{E} \vdash \langle H, C, K\rangle\ \ q\ \ \langle H_2, C_2, K_2\rangle \\ \langle H', C', K'\rangle {=} \mathit{parallelMerge}(\langle H_1, C_1, K_1\rangle, \langle H_2, C_2, K_2\rangle)\end{array}}{\mathcal{E} \vdash \langle H, C, K\rangle\ \ p||q\ \langle H', C', K'\rangle} \quad [\mathit{FV\text{-}Par}]$$

[5] In the case that an infinite-loop history is formed, there is no current instance.
[6] $|\langle H, C, K\rangle|$ is the size of $\langle H, C, K\rangle$.

Algorithm 1 Parallel Merging Algorithm

Input: $\langle H_1, C_1, K_1\rangle$ $\langle H_2, C_2, K_2\rangle$
Output: $\langle H', C', K'\rangle$

```
function parallelMerge (⟨H_1, C_1, K_1⟩, ⟨H_2, C_2, K_2⟩)
    if H_1=H_2=ϵ then
        return ⟨ϵ, C_1 ∪ C_2, max(K_1, K_2)⟩  ▷ Two effects have the same length.
    else if H_1=ϵ then
        if K_1>1 then
            return ⟨ϵ, C_1 ∪ C_2, K_1⟩
            ▷ The first effect is shorter and raises an exception.
        else
            return ⟨C_1||H_1, C_2, K_2⟩
            ▷ The first effect is shorter and without any exceptions.
        end
    end
    else
        F_1 ← fst(H_1)  ▷ Get the first instances from H_1.
        F_2 ← fst(H_2)  ▷ Get the first instances from H_2.
        F ← zip (F_1, F_2)  ▷ Zip the first instances from H_1 and H_2.
        while F ≠ {} do
            (f_1, f_2) ← F.hd  ▷ Get the first element from the F list.
            I ← f_1 ∪ f_2  ▷ Merge f_1 and f_2.
            der_1 ← D_I(H_1)  ▷ Get the derivatives of H_1 w.r.t I.
            der_2 ← D_I(H_2)  ▷ Get the derivatives of H_2 w.r.t I.
            ⟨H_f, C_f, K_f⟩ ← parallelMerge(⟨der_1, C_1, K_1⟩, ⟨der_2, C_2, K_2⟩)
            return ⟨I · H_f, C_f, K_f⟩
        end
    end
end
```

The rule $[FV\text{-}Seq]$ firstly gets $\langle H_1, C_1, K_1\rangle$ by executing p. If the completion code K_1 is 0, it means there is no exceptions raised, therefore, the rule further computes $\langle H_2, C_2, K_2\rangle$ by continuously executing q, to be the final program state. Otherwise, it discards q and returns $\langle H_1, C_1, K_1\rangle$ directly.

$$\frac{\begin{array}{c}\mathcal{E} \vdash \langle H, C, K\rangle \;\; p \;\; \langle H_1, C_1, K_1\rangle \qquad \mathcal{E} \vdash \langle H_1, C_1, K_1\rangle \;\; q \;\; \langle H_2, C_2, K_2\rangle \\ \langle H', C', K'\rangle = \langle H_2, C_2, K_2\rangle \qquad (K_1 \leq 1) \\ \langle H', C', K'\rangle = \langle H_1, C_1, K_1\rangle \qquad (K_1 > 1)\end{array}}{\mathcal{E} \vdash \langle H, C, K\rangle \;\; seq \; p \; q \; \langle H', C', K'\rangle} \; [FV\text{-}Seq]$$

The rule $[FV\text{-}Loop]$ computes a fixpoint (as the invariant effects of the loop body) $\langle H_2, C_2, K_2\rangle$ by continuously executing p twice[7], starting from temporary initialised program states $\langle \epsilon, C, K\rangle$ and $\langle \epsilon, C_1, K_1\rangle$ respectively. If the completion code K_1 is 0, it forms a repeated trace $H \cdot H_1 \cdot (H_2 \cdot C_2)^\star$ with no current instance. Otherwise, it simply exits the

[7] The deterministic loop invariant can be fixed after the second run of the loop body, cf. Appendix C for the proof.

loop in the states of $\langle H \cdot H_1, C_1, K_1\rangle$.

$$\frac{\begin{array}{cc}\mathcal{E} \vdash \langle \epsilon, C, K\rangle \;\; p \;\; \langle H_1, C_1, K_1\rangle & \mathcal{E} \vdash \langle \epsilon, C_1, K_1\rangle \;\; p \;\; \langle H_2, C_2, K_2\rangle \\ \langle H', C', K'\rangle{=}\langle H \cdot H_1 \cdot (H_2 \cdot C_2)^\star, None, K_1\rangle & (K_1{=}0) \\ \langle H', C', K'\rangle{=}\langle H \cdot H_1, C_1, K_1\rangle & (K_1{\neq}0)\end{array}}{\mathcal{E} \vdash \langle H, C, K\rangle \;\; loop \;\; p \;\; \langle H', C', K'\rangle} \;\; [FV\text{-}Loop]$$

The rule $[FV\text{-}Call]$ triggers the back-end solver TRS to check if the precondition of the callee, Φ_{pre}, is satisfied by the current effects state or not. If it holds, the rule obtains the next program state by concatenating the postcondition Φ_{post} to the current effects state.

$$\frac{\begin{array}{c} nm \;\; (\overrightarrow{\tau \; S}) \; \langle \textbf{requires } \Phi_{pre} \;\; \textbf{ensures } \Phi_{post}\rangle \;\; p \in \mathcal{P} \\ TRS \vdash H \cdot C \sqsubseteq \Phi_{pre}\end{array}}{\mathcal{E} \vdash \langle H, C, K\rangle \;\; call \;\; nm \;\; (\overrightarrow{S}) \;\; \langle H \cdot C \cdot \Phi_{post}, None, K'\rangle} \;\; [FV\text{-}Call]$$

Dually, the rule $[FV\text{-}Async]$ initialises the states using $\langle \epsilon, C, K\rangle$ before entering into p, and obtains $\langle H', C', K'\rangle$ after the execution. Then, it emits signal S to indicate that the asynchronous code is resolved.

$$\frac{\mathcal{E} \vdash \langle \epsilon, C, K\rangle \;\; (p; emit \;\; S)||q \;\; \langle H', C', K'\rangle}{\mathcal{E} \vdash \langle H, C, K\rangle \;\; async \;\; S \;\; p \;\; q \;\; \langle H \cdot H', C', K'\rangle} \;\; [FV\text{-}Async]$$

The rule $[FV\text{-}Await]$ archives the current instance, then concatenate the trace $S?$ to the history trace, with no current instance. The rule $[FV\text{-}Raise]$ sets the value of K using d, and keeps the history trace and the current instant unchanged.

$$\frac{H'{=}H \cdot C \cdot S? \qquad C'{=}None}{\mathcal{E}{\vdash}\langle H, C, K\rangle await \;\; S\langle H', C', K\rangle} \; [FV\text{-}Await] \qquad \frac{K'{=}d + 2}{\mathcal{E}{\vdash}\langle H, C, K\rangle raise \;\; d\langle H, C, K'\rangle} \; [FV\text{-}Raise]$$

The rule $[FV\text{-}TryCatch]$ firstly computes $\langle H_1, C_1, K_1\rangle$ from p. Then if the completion code K_1 is 0, that means there is no exception need to be handled, therefore the finial effects is just $\langle H_1, C_1, K_1\rangle$. When the completion code K_1 equals to 1, that means there is an exception need to be handled by the current try, therefore it continues to compute $\langle H_2, C_2, K_2\rangle$ starting from the $\langle H_1, C_1, 0\rangle$. In the case that the completion code K_1 is greater than 1, that means the current exception needs to be handled by an outer try-catch statement, therefore it returns the final effects as $\langle H_1, C_1, K_1\text{-}1\rangle$

$$\frac{\begin{array}{cc}\mathcal{E} \vdash \langle H, C, K\rangle \;\; p \;\; \langle H_1, C_1, K_1\rangle & \mathcal{E} \vdash \langle H_1, C_1, 0\rangle \;\; p \;\; \langle H_2, C_2, K_2\rangle \\ H', C', K'{=}\langle H_1, C_1, K_1\rangle & (K_1{\leq}1) \\ H', C', k'{=}\langle H_2, C_2, K_2\rangle & (K_1{=}2) \\ H', C', k'{=}\langle H_1, C_1, K_1\text{-}1\rangle & (K_1{>}2)\end{array}}{\mathcal{E} \vdash \langle H, C, K\rangle \;\; try \;\; p \;\; with \;\; q \;\; \langle H', C', K'\rangle} \;\; [FV\text{-}TryCatch]$$

6 Temporal Verification via a TRS

The TRS is an automated entailment checker to prove language inclusions between *SyncEffs*. 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. 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., a set of effects inclusion hypothesis, Φ is the history 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 is initially invoked with Γ=$\{\}$ and Φ=ϵ.

6.1 Auxiliary Functions: Nullable, First and Derivative

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

▶ **Definition 3** (Nullable). *Given any sequence Φ, we recursively define $\delta(\Phi)$ as:*

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

$$\delta(\bot){=}false \qquad \delta(\epsilon){=}true \qquad \delta(I){=}false \qquad \delta(S?){=}false \qquad \delta(\Phi^\star){=}true$$

$$\delta(\Phi_1{\cdot}\Phi_2){=}\delta(\Phi_1) \wedge \delta(\Phi_2) \qquad \delta(\Phi_1{\vee}\Phi_2){=}\delta(\Phi_1) \vee \delta(\Phi_2) \qquad \delta(\Phi_1||\Phi_2){=}\delta(\Phi_1) \wedge \delta(\Phi_2)$$

To better outline our contribution, we first present the original *First* function used in Antimirov's rewriting system, denoted using fst', defined as follows:

▶ **Definition 4** (First). *Let $fst(\Phi){:=}\{I \mid (I \cdot \Phi') \in [\![\Phi]\!]\}$ be the set of initial instances derivable from sequence Φ. ($[\![\Phi]\!]$ represents all the traces contained in Φ.)*

$$fst(\bot){=}\{\} \qquad fst(\epsilon){=}\{\} \qquad fst(I){=}\{I\} \qquad fst(\Phi_1 \vee \Phi_2){=}fst(\Phi_1) \cup fst(\Phi_2)$$

$$fst(\Phi^\star){=}fst(\Phi) \qquad fst(\Phi_1 \cdot \Phi_2){=}\begin{cases} fst(\Phi_1) \cup fst(\Phi_2) & if \;\; \delta(\Phi_1){=}true \\ fst(\Phi_1) & if \;\; \delta(\Phi_1){=}false \end{cases}$$

$$fst(S?){=}\{\{S \mapsto Present\}\} \qquad fst(\Phi_1||\Phi_2){=}zip(fst(\Phi_1), fst(\Phi_2))$$

▶ **Definition 5** (Instances Subsumption). *Given two instances 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[8]. Formally,*

$$\begin{aligned} I \subseteq J \Leftrightarrow & \{S \mid (S \mapsto Present) \in J\} \subseteq \{S \mid (S \mapsto Present) \in I\} \\ & and \; \{S \mid (S \mapsto Absent) \in J\} \subseteq \{S \mid (S \mapsto Absent) \in I\} \end{aligned}$$

▶ **Definition 6** (Partial Derivative). *The partial derivative $D_I(\Phi)$ of effects Φ w.r.t. an instance I computes the effects for the left quotient $I^{-1}[\![\Phi]\!]$.*

$$D_I(\bot){=}\bot \qquad D_I(\epsilon){=}\bot \qquad D_I(\Phi^\star){=}D_I(\Phi) \cdot \Phi^\star$$

$$D_I(J){=}\begin{cases} \epsilon & if \;\; I{\subseteq}J \\ \bot & if \;\; I{\not\subseteq}J \end{cases} \qquad D_I(S?){=}\begin{cases} \epsilon & if \;\; I{\subseteq}\{S \mapsto Present\} \\ S? & if \;\; I{\not\subseteq}\{S \mapsto Present\} \end{cases}$$

$$D_I(\Phi_1 \cdot \Phi_2){=}\begin{cases} D_I(\Phi_1) \cdot \Phi_2 \vee D_I(\Phi_2) & if \;\; \delta(\Phi_1){=}true \\ D_I(\Phi_1) \cdot \Phi_2 & if \;\; \delta(\Phi_1){=}false \end{cases}$$

$$D_I(\Phi_1 \vee \Phi_2){=}D_I(\Phi_1) \vee D_I(\Phi_2) \qquad D_I(\Phi_1||\Phi_2){=}D_I(\Phi_1)||D_I(\Phi_2)$$

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

6.2 Rewriting Rules

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

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

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

2. **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.

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

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.

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

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 instances F, which are all the possible initial instances 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.

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

5. **Normalization.** We present a set of normalization rules to soundly transfer the effects into a normal form, in particular after getting their derivatives. Before getting into the above inference rules, we assume that the effects formulae are tailored accordingly based on the axioms shown in Table 2. We built the axiom system on top of a complete axiom system F_1, from (A1) to (A11), suggested by [33], which was designed for regular languages. We develop axioms (A12) to (A16) to further accommodate *SyncEffs*.

7 Soundness and Completeness

▶ **Theorem 7** (Termination). *The rewriting system TRS is terminating.*

Proof. See Appendix A. ◀

▶ **Theorem 8** (Soundness). *Given an inclusion $\Phi_1 \sqsubseteq \Phi_2$, if the TRS returns TRUE when proving $\Phi_1 \sqsubseteq \Phi_2$, i.e., it has a cyclic proof, then $\Phi_1 \sqsubseteq \Phi_2$ is valid.*

Proof. See Appendix B. ◀

Table 2 Normalization Axioms for *SyncEffs*. (c.f. Definition 9 for (A14) - (A16))

(A1)	$\Phi_1 \vee (\Phi_2 \vee \Phi_3) \rightarrow (\Phi_1 \vee \Phi_2) \vee \Phi_3$	(A9)	$\Phi \vee \bot \rightarrow \Phi$
(A2)	$\Phi_1 \cdot (\Phi_2 \cdot \Phi_3) \rightarrow (\Phi_1 \cdot \Phi_2) \cdot \Phi_3$	(A10)	$\epsilon \vee (\Phi \cdot \Phi^\star) \rightarrow \Phi^\star$
(A3)	$\Phi_1 \vee \Phi_2 \rightarrow \Phi_2 \vee \Phi_1$	(A11)	$(\epsilon \vee \Phi)^\star \rightarrow \Phi^\star$
(A4)	$\Phi \cdot (\Phi_1 \vee \Phi_2) \rightarrow \Phi \cdot \Phi_1 \vee \Phi \cdot \Phi_2$	(A12)	$\Phi \ \| \ \epsilon \rightarrow \Phi$
(A5)	$(\Phi_1 \vee \Phi_2) \cdot \Phi \rightarrow \Phi_1 \cdot \Phi \vee \Phi_2 \cdot \Phi$	(A13)	$\Phi \ \| \ \bot \rightarrow \bot$
(A6)	$\Phi \vee \Phi \rightarrow \Phi$	(A14)	$\{S \mapsto Absent\} \| \{S \mapsto Present\} \rightarrow \bot$
(A7)	$\Phi \cdot \epsilon \rightarrow \Phi$	(A15)	$\{S \mapsto Absent\} \| \{S \mapsto Undef\} \rightarrow \{S \mapsto Absent\}$
(A8)	$\Phi \cdot \bot \rightarrow \bot$	(A16)	$\{S \mapsto Present\} \| \{S \mapsto Undef\} \rightarrow \bot$

8 Implementation and Case Study

8.1 Implementation

To show the feasibility of our approach, we have prototyped our automated verification system using OCaml. We prove termination and correctness of the TRS. We validate the front-end forward verifier for conformance, against two implementations: the Columbia Esterel Compiler (CEC) [39] and Hiphop.js [34].

CEC is an open-source compiler designed for research in both hardware and software generation from the Esterel synchronous language to C or Verilog circuit description. It currently supports a subset of Esterel V5, and provides pure Esterel programs for testing. Hiphop.js's implementation facilitates the design of complex web applications by smoothly integrating Esterel and JavaScript, and provides a bench of programs for testing purposes.

Based on these two benchmarks, we validate the verifier using 155 programs, varying from 15 lines to 300 lines. We manually annotate temporal specifications in our *SyncEffs*, including both succeeded and failed instances (roughly with the a 1:1 ratio). Out of the whole test suite, 101 were from the CEC benchmark, 54 were from the Hiphop.js benchmark. Since async-await was inherited from JavaScript features, it only presents in the 54 hiphop.js programs. We conduct experiments on a MacBook Pro with a 2.6 GHz Intel Core i7 processor. Given our benchmark, the running time varying from 0 to 500 ms.

A term rewriting system is efficient because *it only constructs automata as far as it needed*, 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.

8.2 Case Studies

Let's recall the existing challenges in different programming paradigms discussed in Sec. 2. In this section, we first investigate how our effects logic can help to debug errors related to both synchronous and asynchronous programs. Specifically, it effectively resolves the logical correctness checking (for synchronous languages) and critical anti-patterns checking (for premise asynchrony). Meanwhile, we further demonstrate the flexibility and expressiveness of our effects logic.

8.2.1 Logical Incorrect Catching

We regard these programs, which have precisely one safe trace reacting to each input assignments, as logical correct. To effectively check logical correctness, in this work, given a synchronous 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: (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[9]. 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. To align with the logically coherent law, we define the contradictory instance as follows:

▶ **Definition 9** (Contradictory Instance). *Given any instance I, it is contradictory is $\exists S.\ (S \mapsto Absent) \in I$ and $(S \mapsto Present) \in I$ or $\exists S.\ (S \mapsto Undef) \in I$ and $(S \mapsto Present) \in I$.*

(a)	(b)
1. *present S1* $\langle\{S1 \mapsto Undef\}\rangle$	**1.** *present S1* $\langle\{S1 \mapsto Undef\}\rangle$
2. *then* $\langle\{S1 \mapsto Undef, S1 \mapsto Present\}\rangle$	**2.** *then* $\langle\{S1 \mapsto Undef, S1 \mapsto Present\}\rangle$
3. *nothing* $\langle\{S1 \mapsto Undef, S1 \mapsto Present\}\rangle$	**3.** *emit S1* $\langle\{S1 \mapsto Present, S1 \mapsto Present\}\rangle$
4. *else* $\langle\{S1 \mapsto Undef, S1 \mapsto Absent\}\rangle$	**4.** *else* $\langle\{S1 \mapsto Undef, S1 \mapsto Absent\}\rangle$
5. *emit S1* $\langle\{S1 \mapsto Present, S1 \mapsto Absent\}\rangle$	**5.** *nothing* $\langle\{S1 \mapsto Undef, S1 \mapsto Absent\}\rangle$
6. *end present* $\langle\{S1 \mapsto Undef, S1 \mapsto Present\} \vee \{S1 \mapsto Present, S1 \mapsto Absent\}\rangle$ $\langle\bot \vee \bot\rangle \Rightarrow \langle\bot\rangle$	**6.** *end present* $\langle\{S1 \mapsto Present, S1 \mapsto Present\} \vee \{S1 \mapsto Undef, S1 \mapsto Absent\}\rangle$ $\langle\{S1 \mapsto Present\} \vee \{S1 \mapsto Absent\}\rangle$

Table 3 Logical incorrect examples, caught by the effect logic.

As shown in Fig. 3. (a), there are no valid assignments of signal S1 in this program. As shown in in Fig. 3. (b), it is also logical incorrect because there are two possible assignments of signal S1 in this program.

8.2.2 A Strange Logically Correct Program.

```
fork { present S1 then emit S1 else nothing end present
}par { present S1
          then present S2
                then nothing
                else emit S2
                end present
          else nothing end present}
```

Figure 10 Logical Correct.

Another example for synchronous languages shows that composing programs can lead to counter-intuitive phenomena. As the program shows in Fig. 10., the first parallel branch is the logical incorrect program Fig. 3. (b), while the second branch contains a non-reactive program enclosed in "present S1" statement. Surprisingly, this program is logical correct, since there is only one logically coherent assumption:

[9] We define that the instance is contradictory if there this a signal has both *Present* and *Undef* status, or a signal has both *Present* and *Absent* status.

S1 absent and S2 absent. With this assumption, the first present S1 statement takes its empty else branch, which justifies S1 absent. The second "present *S1*" statement also takes its empty else branch, and "emit S2" is not executed, which justifies S2 absent. And our effects logic is able to soundly detect above mentioned correctness checking.

8.2.3 Semantics of Await

We use Table 4. as an example to demonstrate the semantics of $A?$, i.e., "waiting for the signal A". Formally, we define,

$$A? \equiv \exists n, n{\geq}0 \wedge \{\overline{A}\}^n \cdot \{A\}$$

where$\{\overline{A}\}$ refers to all the instances containing A to be absent.

As shown in Table 4., the LHS $\{A\} \cdot \{C\} \cdot B? \cdot \{D\}$ entails the RHS $\{A\} \cdot B? \cdot \{D\}$, as intuitively $\{C\} \cdot B?$ is a special case of $B?$. In step ①, $\{A\}$ is eliminated. In step ③, $B?$ is normalised into $\{B\} \vee (\{\overline{B}\} \cdot B?)$, By the step of ④, $\{C\}$ is eliminated together with $\{\overline{B}\}$ because $\{C\} \subseteq \{\overline{B}\}$. Now the rest part is $B? \cdot \{D\} \sqsubseteq B? \cdot \{D\}$. Here, we further normalise $B?$ from the LHS into a disjunction, leading to two proof sub-trees. From the first sub-tree, we keep unfolding the inclusion with $\{B\}$ (⑥) and $\{D\}$ (⑦) till we can prove it. Continue with the second sub-tree, we unfold it with $\{\overline{B}\}$; then in step ⑧ we observe the proposition is isomorphic with one of the the previous step, marked using (‡). We prove it using the [*REOCCUR*] rule and finish the whole writing process.

Table 4 The example for Await.

$$\epsilon \sqsubseteq \epsilon \quad ⑦[PROVE]$$
$$\cancel{\{D\}} \sqsubseteq \cancel{\{D\}} \quad ⑥[UNFOLD]$$
$$\cancel{\{B\}} \cdot \{D\} \sqsubseteq (\cancel{\{B\}} \vee \bot) \cdot \{D\}$$

$$B? \cdot \{D\} \sqsubseteq B? \cdot \{D\} \quad (‡) \quad ⑧[REOCCUR]$$
$$\cancel{\{\overline{B}\}} \cdot B? \cdot \{D\} \sqsubseteq (\bot \vee (\cancel{\{\overline{B}\}} \cdot B?)) \cdot \{D\} \quad ⑤[Disj\text{-}L]$$

$$B? \cdot \{D\} \sqsubseteq B? \cdot \{D\} \quad (‡) \quad ④[Normalisation]$$
$$\cancel{\{C\}} \cdot B? \cdot \{D\} \sqsubseteq (\bot \vee (\cancel{\{\overline{B}\}} \cdot B?)) \cdot \{D\} \quad ③[UNFOLD]$$
$$\{C\} \cdot B? \cdot \{D\} \sqsubseteq (\{B\} \vee (\{\overline{B}\} \cdot B?)) \cdot \{D\} \quad ②[Normalisation]$$
$$\cancel{\{A\}} \cdot \{C\} \cdot B? \cdot \{D\} \sqsubseteq \cancel{\{A\}} \cdot B? \cdot \{D\} \quad ①[UNFOLD]$$
$$\{A\} \cdot \{C\} \cdot B? \cdot \{D\} \sqsubseteq \{A\} \cdot B? \cdot \{D\}$$

8.2.4 Broken Promises Chain

As the prior work [25, 2] present, one of the critical issues of using promise is the broken chain of the interdependent promises; and they propose the *promise graph*, as a graphical aid, to understand and debug promise-based code.

▶ **Definition 10** (Well-Synchronised Traces). *Based on the syntax (Sec. 4.4) and semantics (Sec. 4.3) defined for* SyncEffs*, in this paper, we call traces without any blocking signals well-synchronised traces.*

We here show that our algebraic effects can capture non well-synchronised traces during the rewriting process by computing the derivative. For example, the parallel composition of traces: $\{A\} \cdot \{B\} \cdot \{C\} \cdot \{D\} \ || \ \{E\} \cdot \ C? \cdot \{F\}$ leads to the final behaviour of $\{A, E\} \cdot \{B\} \cdot \{C\} \cdot \{D, F\}$, which is well-synchronised for all the instances. However, if we were composing traces: $\{A\} \cdot \{B\} \cdot \{D\} \ || \ \{E\} \cdot \ \ C? \cdot \{F\}$ due to the reasons that forgetting to emit **C** (In JavaScript, it could be the case that forgetting to explicitly return a promise result), it leads to a problematic trace $\{A, E\} \cdot \{B\} \cdot \{D\} \cdot C? \cdot \{F\}$. The final effects contain a dangling signal waiting of C, which indicates the corresponding anti-pattern.

8.3 Discussion

As the examples show, our proposed effects logic and the abstract semantics for λ^{sync}_{async} not only tightly capture the behaviours of a mixed synchronous and asynchronous concurrency model but also help to mitigate the programming challenges in each paradigm. Meanwhile, the inferred temporal traces from a given reactive program enable a compositional temporal verification at the source level, which is not yet supported by existing temporal verification techniques.

9 Related Work

This work is related to i) semantics of synchronous languages and asynchronous promises; ii) research on reactive system modelling and verification; and iii) existing traces-based effects systems.

9.1 Semantics of Esterel and JavaScript's asynchrony

The web orchestration language HipHop.js [12] integrates Esterel's synchrony with JavaScript's asynchrony, which provides the infrastructure for our work on mixed synchronous and asynchronous concurrency models. To the best of authors' knowledge, the inference rules in this work formally define the first axiomatic semantics for a core language of HipHop.js, which are established on top of the existing semantics of Esterel and JavaScript's asynchrony.

For the pure Esterel, the communication kernel of the Esterel synchronous reactive language, prior work gave two semantics, a macrostep logical semantics called the behavioural semantics [9], and a small-step semantics called execution/operational semantics [11]. Our *SyncEffs* of Esterel primitives closely follow the work of states-based semantics [9]. In particular, we borrow the idea of internalizing state into effects using *history* instance trace and *current* instance, 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.

In JavaScript programs, the primitives *async* and *await* serve for *promises*-based (supported in ECMAScript 6 [18]) asynchronous programs, which can be written in a synchronous style, leading to more scalable code. However, the ECMAScript 6 standard specifies the semantics of promises informally and in operational terms, which is not a suitable basis for formal reasoning or program analysis. Prior work [25, 2], 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 concurrent context.

In this paper, we propose the first work to combine the operational semantics of synchronous Esterel and the asynchronous constructs in JavaScript, building the language foundation for such a blending of two concurrency models.

9.2 Existing Traces-Based Effects Systems

Combining program events with a temporal program logic for asserting properties of event traces yields a powerful and general engine for enforcing program properties. Results in [36, 35, 27] have demonstrated that static approximations of program event traces can be generated by type and effect analyses [41, 4], in a form amenable to existing model-checking techniques for verification. We call these approximations trace-based effects.

Trace-based analyses have been shown capable of statically enforcing flow-sensitive security properties such as safe locking behaviour [19] and resource usage policies such as file usage protocols and memory management [27]. In [7], a trace effect analysis is used to enforce secure service composition. Stack-based security policies are also amenable to this form of analysis, as shown in [35].

More related to our work, prior research has been extending Hoare logic with event traces. The work [26] focuses on finite traces (terminating runs) for web applications, leaving the divergent computation, which indicates *false*, verified for every specification. The work [28] focuses on infinite traces (non-terminating runs) by providing coinductively trace definitions. More recent works [16, 37] proposed dynamic logics and unified operators to reason about possibly finite and infinite traces at the same time.

Moreover, this paper draws similarities to *contextual effects* [29], which computes the effects that have already occurred as the prior effects. The effects of the computation yet to take place as the future effects. Besides, prior work [30] proposes an annotated type and effect system and infers behaviours from CML [32] programs for channel-based communications, though it did not provide any inclusion solving process.

10 Conclusion

This work targets temporal verification, which i) proposes the first operational semantics and a corresponding axiomatic semantics for the mixed Sync-Async concurrency paradigm; ii) present the first algebraic TRS for the novel effects logic, *SyncEffs*.

We use *SyncEffs* to capture reactive program behaviours and temporal properties. We demonstrate how to give an axiomatic semantics to λ^{sync}_{async} by trace processing functions. We use this semantic model to enable a Hoare-style forward verifier, which computes the program effects constructively. We present an effects inclusion checker (the TRS) to prove the annotated temporal properties efficiently. We prototype the verification system and show its feasibility. To the best of our knowledge, our work is the first that formulates semantics of a mixed Sync-Async concurrency paradigm; and that automates modular temporal verification for reactive programs using an expressive effects logic.

References

1 https://en.wikipedia.org/wiki/Synchronous_programming_language, 2021.

2 Saba Alimadadi, Di Zhong, Magnus Madsen, and Frank Tip. Finding broken promises in asynchronous javascript programs. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–26, 2018.

3 Marco Almeida, Nelma Moreira, and Rogério Reis. Antimirov and mosses's rewrite system revisited. *International Journal of Foundations of Computer Science*, 20(04):669–684, 2009.

4 Torben Amtoft, Hanne Riis Nielson, and Flemming Nielson. *Type and effect systems: behaviours for concurrency*. World Scientific, 1999.

5 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.

6 Valentin M Antimirov and Peter D Mosses. Rewriting extended regular expressions. *Theoretical Computer Science*, 143(1):51–72, 1995.

7 Massimo Bartoletti, Pierpaolo Degano, and Gian Luigi Ferrari. Enforcing secure service composition. In *18th IEEE Computer Security Foundations Workshop (CSFW'05)*, pages 211–223. IEEE, 2005.

8 Gérard Berry. Preemption in concurrent systems. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 72–93. Springer, 1993.

9 Gerard Berry. The constructive semantics of pure esterel-draft version 3. *Draft Version*, 3, 1999.

10 Gérard Berry. *The Esterel v5 language primer: version v5_91*. Centre de mathématiques appliquées, Ecole des mines and INRIA, 2000.

11 Gérard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of computer programming*, 19(2):87–152, 1992.

12 Gérard Berry and Manuel Serrano. Hiphop. js:(a) synchronous reactive web programming. In *PLDI*, pages 533–545, 2020.

13 Gavin Bierman, Claudio Russo, Geoffrey Mainland, Erik Meijer, and Mads Torgersen. Pause n play: Formalizing asynchronous c sharp. In *European Conference on Object-Oriented Programming*, pages 233–257. Springer, 2012.

14 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.

15 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.

16 Richard Bubel, Crystal Chang Din, Reiner Hähnle, and Keiko Nakata. A dynamic logic with traces and coinduction. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, pages 307–322. Springer, 2015.

17 Andrei Damian, Cezara Drăgoi, Alexandru Militaru, and Josef Widder. Communication-closed asynchronous protocols. In *International Conference on Computer Aided Verification*, pages 344–363. Springer, 2019.

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

19 Jeffrey S Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 1–12, 2002.

20 KLAUS V Gleissenthall, RAMI GÖKHAN Kici, ALEXANDER Bakst, DEIAN Stefan, and RANJIT Jhala. Pretend synchrony. POPL, 2019.

21 Georges Gonthier. *Sémantiques et modèles d'exécution des langages réactifs synchrones: application à Esterel*. PhD thesis, Paris 11, 1988.

22 Dag Hovland. The inclusion problem for regular expressions. *Journal of Computer and System Sciences*, 78(6):1795–1813, 2012.

23 Kennedy Kambona, Elisa Gonzalez Boix, and Wolfgang De Meuter. An evaluation of reactive programming and promises for structuring collaborative web applications. In *Proceedings of the 7th Workshop on Dynamic Languages and Applications*, pages 1–9, 2013.

24 Matthias Keil and Peter Thiemann. Symbolic solving of extended regular expression inequalities. *arXiv preprint arXiv:1410.3227*, 2014.

25 Magnus Madsen, Ondřej Lhoták, and Frank Tip. A model for reasoning about javascript promises. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–24, 2017.

26 Gregory Malecha, Greg Morrisett, and Ryan Wisnesky. Trace-based verification of imperative programs with i/o. *Journal of Symbolic Computation*, 46(2):95–118, 2011.

27 Kim Marriott, Peter J Stuckey, and Martin Sulzmann. Resource usage verification. In *Asian Symposium on Programming Languages and Systems*, pages 212–229. Springer, 2003.

28 Keiko Nakata and Tarmo Uustalu. A hoare logic for the coinductive trace-based big-step semantics of while. In *European Symposium on Programming*, pages 488–506. Springer, 2010.

29 Iulian Neamtiu, Michael Hicks, Jeffrey S Foster, and Polyvios Pratikakis. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 37–49, 2008.

30 Hanne Riis Nielson, Torben Amtoft, and Flemming Nielson. Behaviour analysis and safety conditions: a case study in cml. In *International Conference on Fundamental Approaches to Software Engineering*, pages 255–269. Springer, 1998.

31 Cristian Prisacariu. Synchronous kleene algebra. *The Journal of Logic and Algebraic Programming*, 79(7):608–635, 2010.

32 John H Reppy. Concurrent ml: Design, application and semantics. In *Functional Programming, Concurrency, Simulation and Automated Reasoning*, pages 165–198. Springer, 1993.

33 Arto Salomaa. Two complete axiom systems for the algebra of regular events. *Journal of the ACM (JACM)*, 13(1):158–169, 1966.

34 Manuel Serrano. `https://github.com/manuel-serrano/hiphop`, 2021.

35 Christian Skalka and Scott Smith. History effects and verification. In *Asian Symposium on Programming Languages and Systems*, pages 107–128. Springer, 2004.

36 Christian Skalka, Scott Smith, and David Van Horn. Types and trace effects of higher order programs. *Journal of Functional Programming*, 18(2):179–249, 2008.

37 Yahui Song and Wei-Ngan Chin. Automated temporal verification of integrated dependent effects. In *International Conference on Formal Engineering Methods*, pages 73–90. Springer, 2020.

38 Yahui Song and Wei-Ngan Chin. A synchronous effects logic for temporal verification of pure esterel. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 417–440. Springer, 2021.

39 Jia Zeng Stephen A. Edwards, Cristian Soviani. `http://www.cs.columbia.edu/~sedwards/cec/`, 2021.

40 Jun Sun, Yang Liu, Jin Song Dong, and Jun Pang. Pat: Towards flexible verification under fairness. In *International conference on computer aided verification*, pages 709–714. Springer, 2009.

41 Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and computation*, 111(2):245–296, 1994.

42 Ghaith Tarawneh and Andrey Mokhov. Formal verification of mixed synchronous asynchronous systems using industrial tools. In *2018 24th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 43–50. IEEE, 2018.

43 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 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 $\langle S, H\rangle$:

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

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. 6.2 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.

◀

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.

1. **Axiom Rules:**

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

- 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.

2. **Disprove Rules:**

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

- 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.

3. **Prove Rules:**

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

- To prove soundness of 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 valid.
- To prove soundness of 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 valid.

4. **Unfolding Rule:**

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

- To prove soundness of 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 instance I, let $\varphi_1{}' \models \mathtt{I}^{-1}[\![\Phi_1]\!]$; and $\varphi_2{}' \models \mathtt{I}^{-1}[\![\Phi_2]\!]$.
Case 1), $\mathtt{I} \notin F$, $\varphi_1{}' \models \bot$, 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 valid, then $\varphi_1{}' \models D_{\mathtt{I}}(\Phi_2)$.
By Definition 2, since for all I, $D_{\mathtt{I}}(\Phi_1) \sqsubseteq D_{\mathtt{I}}(\Phi_2)$, the conclusion is valid.

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

C Execute the loop body twice to compute the invariant

▶ **Theorem 11** (Twice). *The deterministic loop invariant can be fixed after the second run of the loop body.*

Proof. Let $\langle H, C\rangle$ be the initial program state, where H is the history trace, and C is the current instance. Given any program P, we use H_P and C_P to denote the history trace and current instance by executing P. Based on the semantics of synchronous programs, there are three possible kinds of loops:

Case 1: loop {Pause; P},
The first run: $\langle \epsilon, C\rangle$ Pause; P $\langle C \cdot H_P, C_P\rangle$
The second run: $\langle \epsilon, C_P\rangle$ Pause; P $\langle C_P \cdot H_P, C_P\rangle$
The final effects: $H \cdot C \cdot H_P \cdot (C_P \cdot H_P)^\star$

Case 2: loop {P; Pause},
The first run: $\langle \epsilon, C\rangle$ P; Pause $\langle (C||H_P) \cdot C_P, \{\}\rangle$
The second run: $\langle \epsilon, \{\}\rangle$ P; Pause $\langle H_P \cdot C_P, \{\}\rangle$
The final effects: $H \cdot (C||H_P) \cdot C_P \cdot (H_P \cdot C_P)^\star$

Case 3: loop {P1; Pause; P2},
The first run: $\langle \epsilon, C\rangle$ P1; Pause; P2 $\langle (C||(H_{P1} \cdot C_{P1})) \cdot H_{P2}, C_{P2}\rangle$
The second run: $\langle \epsilon, C_{P2}\rangle$ P1; Pause; P2 $\langle (C_{P2}||(H_{P1} \cdot C_{P1})) \cdot H_{P2}, C_{P2}\rangle$
The final effects: $H \cdot (C||(H_{P1} \cdot C_{P1})) \cdot H_{P2} \cdot ((C_{P2}||(H_{P1} \cdot C_{P1})) \cdot H_{P2})^\star$

In all possible cases, the loop invariant can be fixed by the end of the second run. ◀