TempAS: A Temporal Verification Tool for Mixed Async-Sync Concurrency

No Author Given

No Institute Given

Abstract. In this paper, we introduce *TempAS*[1], a tool that verifies temporal properties of reactive applications, which are based on a mixed asynchronous and synchronous concurrency model. The main contributions of this tool are: i) it deploys a novel effects logic to express non-trivial temporal properties; ii) it compositionally infers the temporal behaviors of given programs; iii) it automatically proves/disproves the inferred program effects against given temporal specifications.
In particular, we avoid the complex translation from the temporal logic into (timed-)automata by using an algebraic term-rewriting system (TRS), which gains both expressiveness and efficiency, escaping the capability of existing real-time verification techniques. We show the feasibility of *TempAS* along with case studies and compare it with the literature.

1 Introduction

As it recently proposed, reactive languages such as Hiphop.js[2] [29,10], are designed to be based on a smooth integration of (i) Asynchronous concurrency, which performs interactions between components or with the environment with uncontrollable timing, such as network-based communication; (ii) Synchronous reactive paradigm, which reacts 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."[10]

Given a multi-paradigm reactive language represented by Hiphop.js, verifying its temporal behaviour becomes engaging and challenging. Traditional ways of verifying real-time systems either: (i) use expressive automata to model the program logic directly, such as timed automata [21], which lacks composable patterns for high-level system design; or (ii) deploy translation-based approaches for compositional timed process algebras; for example, Timed CSP [15] is translated to timed automata so that the model checker Uppaal can be [21] applied.

[1] `http://loris-5.d2.comp.nus.edu.sg/MixedSyncAsync/introduction.html`

[2] Hiphop.js is a JavaScript extension of Esterel [9] (or vice versa) for reactive web applications: `https://www-sop.inria.fr/members/Colin.Vidal/hiphop/`.

In practice, system requirements are often structured into phases, which are then composed in many different ways, while TA is deficiency in modelling complex compositional systems. Users often need to manually cast high-level requirements into a set of clock variables with carefully calculated clock constraints, which is tedious and error-prone [27]. On the other hand, all the translation-based approaches share the common problem: the overhead introduced by the complex translation makes it inefficient when *disproving* properties. We believe that the goal of checking logical temporal properties can be done without constructing the whole reachability graph or the full power of model-checking.

To tackle the above issues and exploit the best of both asynchronous and synchronous concurrency models, we present a solution via a compositional Hoare-style forward verifier and a term rewriting system (TRS)[3], based on a novel temporal specification language. More specifically, we specify system behaviours in the form of *Timed Synchronous Effects*, which integrates the Synchronous Kleene Algebra (SKA) [23,11] with dependent values and arithmetic constraints, to provide real-time abstractions into traditional synchronous verification [13,16,28]. For example, one safety property, *"The event* ***Done*** *will be triggered no later than ten seconds"*[4], is expressed in our effects logic as:

$$\Phi \triangleq \texttt{0}\leq\texttt{t}<\texttt{10} : (\{\}^\star \cdot \{\textbf{Done}\})\#\texttt{t}.$$

The timed effects incorporates different kinds of existing temporal logics, such as linear-time temporal logic (LTL) and metric temporal logic (MTL). Here, $\#$ is the parallel operator specifying the *real-time* constraints for the *logical-time* sequences [30]; $\{\}$ encloses one single logical-time instance; Kleene star $\star$ denotes trace repetition. The above formula Φ corresponds to '$\Diamond_{\{10\}}$ **Done**' in MTL, reads *"within ten seconds,* ***Done*** *finally happens"*. Moreover, the time bound constraints can be dependent on the program inputs. For example, we express, the effects of a method `send(d)` as:

$$\Phi^{\texttt{send(d)}} \triangleq (\texttt{0}<\texttt{d}\leq\texttt{5} \wedge \texttt{0}\leq\texttt{t}<\texttt{d}) : (\{\textbf{Send}\}\#\texttt{t}) \cdot \{\textbf{Done}\}.$$

The `send` method takes a parameter `d`, and **Send**s out a message at `t` second. The above formula $\Phi^{\texttt{send(d)}}$ indicates the facts that the input parameter `d` is positive and smaller or equal to `5`; the method generates a finite trace, i.e., a sequence with the first instance containing the event **Send**, followed by a second instance containing the event **Done**; and the instance $\{$**Send**$\}$ takes no more than `d` seconds to finish. Although these examples are simple, they already illustrate properties beyond existing temporal logics and traditional timed automata.

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

[3] A TRS is a refutation method in which checking their inclusion corresponds to an iterated process of checking the inclusion of their *partial derivatives* [5].

[4] For simplicity, we use integer values to represent seconds in this paper, while it can be easily extended to real numbers and other time measurement units.

Φ', does $\Phi^{\mathcal{P}} \sqsubseteq$ [5] Φ' 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 present a new tool, *TempAS*, of extensive temporal verification comprising: a front-end verifier computes the deterministic program behaviour via inference rules at the source level, defining program evaluation syntactically; and a back-end entailment checker (the TRS) inspired by Antimirov and Mosses' algorithm[6] [6] but solving the language inclusions between more expressive timed synchronous effects. Prior works based on a TRS [25,6,4,20,17] show its feasibility and suggest that this method is a better average-case algorithm than those based on automata theory. Similarly, our approach is simpler than existing automata-based verification techniques, as it is based directly on constraint-solving and can be efficient in verifying systems consisting of many components as it avoids exploring the whole state-space [25,31].

To the best of the authors' knowledge, our tool implements the first approach for temporal verification on a mixed concurrency model; and its capability of local reasoning enables capturing bugs introduced by the actual implementation.

Paper Organization. Sec. 2 specifies the requirements and goals of the tool. Sec. 4 formally presents a core language $\lambda_{\texttt{HH}}$, and the syntax of the timed effects logic. It then introduces a running example (Sec. 4.3) to highlight our main methodologies. Sec. 5 presents the usages of the front-end forward verifier. Sec. 6 explains the back-end solver TRS for effects inclusion checking. Sec. 7 demonstrates the implementation and cases studies; and we conclude in Sec. 8.

2 *TempAS*: Requirements and Goals

An overview of our automated verification system is given in Fig. 1. It consists of a front-end Hoare-style forward verifier and a back-end TRS.

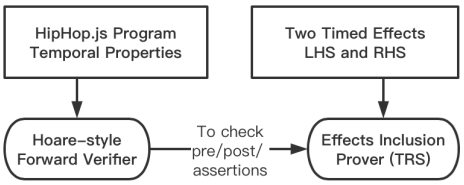

Fig. 1. System Overview.

The inputs of the forward verifier are target programs (Sec. 4.1) annotated with temporal specifications written in timed synchronous effects (Sec. 4.2). The front-end is obligated to compositionally infer the program effects from the source code (Sec. 5). Besides, the verifier calls the back-end solver to prove/disprove produced inclusions, i.e., between the effects states and pre/post conditions or assertions (cf. Fig. 8.).

The input of the TRS (Sec. 6) 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.)*.

[5] The inclusion notation $\sqsubseteq$ is formally defined in Theorem 1.

[6] 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.

Deliverables. In summary, our implementation towards automated temporal verification resulted in the following main contributions:

1. **The Temporal Effects Logic:** We define the syntax of timed synchronous effects, to be a novel specification language, which captures the target programs' behaviours and non-trivial temporal properties.
2. **The Automated Forward Verifier:** Targeting a core language $\lambda_{\texttt{HH}}$, we establish and implement an abstract semantics model via a set of inference rules, enabling a compositional verifier to infer the program's effects.
3. **An Efficient TRS:** We propose and implement a new rewriting system, to soundly prove the inferred effects against given temporal properties, both expressed by timed synchronous effects.
4. **Validation and Evaluation:** We validate our tool for conformance against two implementations: the Columbia Esterel Compiler (CEC) [1] and Hiphop.js [2]. We report on a case study investigating how it can help to debug errors related to both synchronous and asynchronous programs.

Tool Restrictions. Our tool expects Hoare-style specifications for every reactive module as input, i.e., it contains both precondition and postcondition written in the timed effects logic. We assume that the input programs are already type-checked by the existing compiler [2]. Since $\lambda_{\texttt{HH}}$ (abstracted from HipHop.js) is designed as an integration of synchronous language Esterel and asynchronous JavaScript, our tool supports most of the syntax, except for the *trap* and *suspend* from Esterel, which other primitive preemption operators can replace.

3 Preliminaries: Synchronous Esterel and HipHop.js

The principle of synchronous programming is to design a high-level abstraction where the timing characteristics of the electronic transistors are neglected. Such an abstraction makes reasoning about time a lot simpler, 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. 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 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*.

Built on top of Esterel, 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 [10]. To show the advantages of such a mixture, Fig. 2. 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.

```
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)
}
```

Fig. 2. A comparison between JavaScript (top) and HipHop.js (bottom) for a same login button implementation [10]. (In the bottom, `name`, `passwd` and `enableLogin` are reactive input/output signals.)

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.

4 Language and Specifications

4.1 The Target Language

We generalise the design of Hiphop.js into a core language $\lambda_{\texttt{HH}}$, which provides the infrastructure for mixing asynchronous and synchronous concurrency models. We here formally define the syntax of $\lambda_{\texttt{HH}}$, as shown in Fig. 3. The statements marked as purple come from the Esterel v5 [7,8] endorsed by current academic compilers; while the statements marked as blue 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.

Meta-variables are **S**, `x` and `nm`. Basic signal types include `IN` for input signals, `OUT` for output signals, `INOUT` for the signals used to be both input and output and `int` for integer variables. **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 $\overrightarrow{\tau\ x}$ and

$\overrightarrow{\tau\ \mathbf{S}}$, a statement-oriented body p, associated with a precondition $\Phi_{\texttt{pre}}$ and a postcondition $\Phi_{\texttt{post}}$. (The syntax of effects specification Φ is given in Fig. 6.)

$$
\begin{array}{lrl}
(\textit{Program}) & \mathcal{P} ::= & \overrightarrow{module} \qquad (\textit{Basic Types})\ \ \tau ::= \texttt{IN} \mid \texttt{OUT} \mid \texttt{INOUT} \mid \texttt{int} \\
(\textit{Module Def.}) & \texttt{module} ::= & \texttt{nm}\ \ (\overrightarrow{\tau\ x},\ \overrightarrow{\tau\ \mathbf{S}})\ \ \langle \textbf{requires}\ \ \Phi_{\texttt{pre}}\ \ \textbf{ensures}\ \ \Phi_{\texttt{post}} \rangle\ \ \texttt{p} \\
(\textit{Statement}) & \texttt{p, q} ::= & \texttt{nothing} \mid \texttt{yield} \mid \texttt{emit}\ \mathbf{S} \mid \texttt{present}\ \mathbf{S}\ \texttt{p q} \mid \texttt{seq p q} \\
 & & \mid \texttt{fork p par q} \mid \texttt{loop p} \mid \texttt{run nm}\ (\overrightarrow{x}, \overrightarrow{\mathbf{S}}) \mid \texttt{abort p when d} \\
 & & \mid \texttt{async}\ \mathbf{S}\ \texttt{p d} \mid \texttt{await}\ \mathbf{S} \mid \texttt{assert}\ \Phi
\end{array}
$$

$\mathbf{S} \in$ signal variables nm, x $\in$ **var** (*Finite List*) $\rightarrow$ (*Time Bounds*) d$\in \mathbb{Z}^+$

Fig. 3. Syntax of $\lambda_{\texttt{HH}}$.

We next explain the intuitive semantics of our target language $\lambda_{\texttt{HH}}$, while prior works [9] and [22] present the formal semantics of Esterel and JavaScript Promises respectively.

The statement `nothing` is the Esterel equivalent of unit, void or skip in other languages. A thread of execution suspends itself for the current time instance using the `yield` construct, and resumes when the next time 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 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 sequence statement `seq 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. The parallel statement `fork 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. *(Notice that, under the instantaneous nature of Esterel, both '*`seq{emit` ***S1****}{emit* ***S2****}' and '*`fork{emit` ***S1****}par{emit* ***S2****}' lead to effects* {S1, S2}*, which emits* ***S1*** *and* ***S2*** *simultaneously and terminates instantaneously.)*

Another example, as it shown in Fig. 4., in a synchronous concurrency model, 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 effects should be $\{\mathbf{A}, \mathbf{E}\} \cdot \{\mathbf{B}, \mathbf{C}, \mathbf{F}\} \cdot \{\mathbf{G}\}$.

```
1 fork{
2   emit A; yield;
3   emit B; emit C
4 }par{
5   emit E; yield;
6   emit F; yield;
7   emit G}
```

Fig. 4. Parallel Composition.

The statement `loop p` implements a repeated pattern, which is possible to be aborted 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 yield statement to avoid an 'infinite instance'. For example, '`loop emit` **S**' is not a correct program.

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. 5., the program firstly emits signal **A**, then enters into a loop which emits signal **B** followed by a *yield* followed by emitting signal **C** at the end. The effects of it is $\{\mathbf{A},\mathbf{B}\}\cdot\{\mathbf{B},\mathbf{C}\}\cdot\{\mathbf{B},\mathbf{C}\}\cdot\{\mathbf{B},\mathbf{C}\}$..., which says that in the first time instance, signals **A** and **B** will be present, as there is no explicit yield 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**.

```
1 module a_loop(out A,B,C){
2   emit A;
3   loop {
4      emit B;
5      yield;
6      emit C;}}
```

Fig. 5. A Loop Example in Esterel.

The statement `run nm` $(\overrightarrow{x}, \overrightarrow{\mathbf{S}})$ is a call to module `nm`, parametrising with values and signals. To facilitate a preemptive concurrency, as well as provide necessary real-time bounds, $\lambda_{\mathtt{HH}}$ includes the primitive statement `abort p when d`, which runs statement `p` to completion. If the execution reached the time bound `d`, it terminates immediately.

The `async` statement enables well-behaving synchronous to regulate unsteady asynchronous computations. The statement `async` **S** `p d` is supposed to spawn a long lasting background computation, where `d` represents an execution delay. 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. The statement `assert` Φ is used to guarantee the temporal property Φ asserted at a certain point of the programs. Prior work [19] shows that such a language combination makes reactive programming more powerful and flexible than the traditional web programming.

4.2 The Specification Language

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

The syntax of the timed synchronous effects is formally defined in Fig. 6. Effects is a conditioned timed instance sequence $\pi : \mathtt{es}$ or a disjunction of two effects $\Phi_1 \vee \Phi_2$. Timed sequences comprise *nil* ($\bot$); an empty trace ϵ[7]; a single time instance represented by `I`; a waiting for a single signal **S**?; sequences concatenation $\mathtt{es}_1 \cdot \mathtt{es}_2$; disjunction $\mathtt{es}_1 \vee \mathtt{es}_2$; synchronous parallelism $\mathtt{es}_1 || \mathtt{es}_2$.

We introduce a new operator #, and the effects `es#t` represents that a trace takes real-time `t` to complete, where `t` is a *term*. A timed sequence can also be constructed by Kleene star $\star$, representing zero or many times (possibly infinite)

[7] Taking a language as a set of sentences, $\bot$ means an empty set; whereas ϵ represents the language contains only one sentence, which is the empty trace.

$$
\begin{array}{rrcl}
(\textit{Timed Effects}) & \Phi & ::= & \pi : \mathtt{es} \mid \Phi_1 \vee \Phi_2 \\
(\textit{Timed Sequences}) & \mathtt{es} & ::= & \bot \mid \epsilon \mid \mathtt{I} \mid \mathbf{S}? \mid \mathtt{es}_1 \cdot \mathtt{es}_2 \mid \mathtt{es}_1 \vee \mathtt{es}_2 \mid \mathtt{es}_1 || \mathtt{es}_2 \mid \mathtt{es}\#\mathtt{t} \mid \mathtt{es}^\star \\
(\textit{Time Instances}) & \mathtt{I} & ::= & \{\} \mid \{\mathbf{S}\} \mid \{\overline{\mathbf{S}}\} \mid \mathtt{I}_1 \cup \mathtt{I}_2 \\
(\textit{Pure}) & \pi & ::= & \textit{True} \mid \textit{False} \mid A(t_1, t_2) \mid \pi_1 \wedge \pi_2 \mid \pi_1 \vee \pi_2 \mid \neg\pi \\
 & & & \mid \pi_1 \Rightarrow \pi_2 \mid \forall \mathtt{x}.\pi \mid \exists \mathtt{x}.\pi \\
(\textit{Real-Time Term}) & \mathtt{t} & ::= & \mathtt{c} \mid \mathtt{n} \mid \mathtt{t}_1 + \mathtt{t}_2 \mid \mathtt{t}_1 - \mathtt{t}_2
\end{array}
$$

$\mathbf{S} \in$ signal variables $\quad \mathtt{c} ::\in \mathbb{R}^+ \quad \mathtt{n} ::\in \mathbf{var}$

(*Blocking*) ? $\quad$ (*Real Time Bound*) # $\quad$ (*Kleene Star*) $\star$

Fig. 6. Syntax of Timed Synchronous Effects.

repetition of a trace. There are two possible states for a signal: present $\mathbf{S}$, or absent $\overline{\mathbf{S}}$. The default state of signals in a new time instance is absent. A time instance $\mathtt{I}$ is a set of signals; and it can be possible empty sets $\{\}$, indicating that there is no signal constraints for the certain time instance.

We use π to denote a pure formula which captures the (Presburger) arithmetic conditions on terms or program parameters. We use $\mathtt{A}(\mathtt{t}_1, \mathtt{t}_2)$ to represent atomic formulas of two terms (including $=$, $>$, $<$, $\geq$ and $\leq$). A term can be a constant integer value $\mathtt{c}$, an integer variable $\mathtt{n}$ which is an input parameter of the program and can be constrained by a pure formula. A term also allows simple computations of terms, $\mathtt{t}_1 + \mathtt{t}_2$ and $\mathtt{t}_1 - \mathtt{t}_2$. To abstract the elapsed time, the implicit pure constraints of all the terms is to be greater or equal to 0.

4.3 A Running Example

To give an intuitive summary of our techniques, from here, we use the following running example to highlight our main methodologies. We define Hoare-triple style specifications (enclosed in /*@ ... @*/), which enable a compositional verification strategy, where temporal reasoning can be done locally.

```
1  hiphop module main (out Prep, in Tick, out Ready, out Go, out Cook)
2  /*@ requires true : emp @*/
3  /*@ ensures  0<=t/\t<3 : {}.({Cook})#t.{}^* @*/
4  {
5    fork{ // One thread produces true : Ready?.{Go}
6      await Ready; emit Go;
7    }par{ // The other thread produces 0<t<3 : {Prep, Cook}#t.{Ready}
8      emit Prep;
9      async Ready { run cook (3, Tick, Cook); }}}
10 // The final effects  0<=t/\t<3 : ({Prep}.{Cook})#t.{Ready}.{Go}
11
12 hiphop module cook (var d, in Tick, out Cook)
13 /*@ requires d>2  : {Prep} @*/
14 /*@ ensures  0<=t/\t<d : ({}.{Cook})#t @*/
15 { abort count(d, Tick) { yield; emit Cook; }}
```

Fig. 7. Mixed Asynchronous and Synchronous Concurrency with Function Calls.

As shown in Fig. 7., the module `main` contains a `fork/par` statement, which spawns two threads running in parallel. The first thread firstly waits for the signal **Ready**, then emits the signal **Go**. The second thread firstly emits the signal **Prep**, then calls the function `cook` asynchronously. The precondition of `main` requires no arithmetic constraints on its input values (expressed as *true*), neither any pre-traces (expressed as *emp*, indicating an empty trace). The postcondition of `main` ensures a trace, where the second time instance contains at least one signal **Cook** and finishes within 3 seconds after the completion of the first time instance. Then we do not care about the rest of the trace ({} is similar to a wildcard). We use this `main` module to demonstrate the work flow of the forward verifier, presented in the following section (cf. Fig. 8.).

Furthermore, the precondition of `cook`, `d`>2:{`Prep`} requires that the input variable `d` should be greater than 2; and before entering into this module, the signal **Prep** should be emitted in the current time instance, indicating that preparation has been done. Therefore, the local specifications may contain events from other modules.

5 The Front-End: A Forward Verifier

The forward verifier defines an abstract denotational semantic model for $\lambda_{\texttt{HH}}$, by formalising a set of effects inference 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 four-elements tuple $\langle \Pi, \texttt{H}, \texttt{C}, \texttt{T} \rangle$, with the following concrete domains:

$$\mathcal{E} \triangleq \overrightarrow{\mathbf{S}},\ \Pi \triangleq t{\rightarrow}\pi,\ H \triangleq es,\ C \triangleq \overrightarrow{\mathbf{S}{\rightarrow}\Delta} \mid \mathbf{S}?,\ \Delta \triangleq \mathit{Present} \mid \mathit{Absent} \mid \mathit{Undef},\ T \triangleq t$$

Let $\mathcal{E}$ be the environment containing all the local and output signals; Π represents the pure constraints for all the terms and time variables; `H` represents the trace of *history*; `C` represents the *current* time instance or a waiting signal; Δ marks the signal status; `T` is a term binding `C`.

Continue with the running example from Fig. 7., as shown in Fig. 8., we demonstrate the forward verification process of the module `main`. The effects states of the program are captured in the form of $\langle \Phi \rangle$. To facilitate the illustration, we label the verification steps by (1), ..., (9). We mark the deployed inference rules in [gray] (see Appendix A for the full definition of inference rules). The verifier invokes the TRS to check language inclusions when necessary.

The effects states (1) and (4) are initial effects when entering into the `fork/par` statement. The effects state (2) is obtained by [FV-Await], which concatenate a waiting signal (with a question mark) followed by a new empty time instance to the current effects state. The effects states (3) and (5) are obtained by [FV-Emit], which simply adds the emitted signal to the current time instance. The intermediate effects state of (7) is obtained by [FV-Call].

We here highlight the rule [FV-Call], which invokes the TRS to check whether the current effects state satisfies the precondition of the callee module, $\Phi_{\texttt{pre}}$. If the

1. fork{ *(– initialize the current effects state using the module precondition –)*
   $\langle \texttt{true} : \texttt{emp} \rangle$ *(– emp indicates an empty trace –)*
2. await Ready;
   $\langle \texttt{true} : \texttt{Ready?} \cdot \{\} \rangle$ [FV-Await]
3. emit Go;
   $\langle \texttt{true} : \texttt{Ready?} \cdot \{\texttt{Go}\} \rangle$ [FV-Emit]
4. }par{ *(– initialize the current effects state using the module precondition –)*
   $\langle \texttt{true} : \texttt{emp} \rangle$
5. emit Prep;
   $\langle \texttt{true} : \{\texttt{Prep}\} \rangle$ [FV-Emit]
6. async Ready{
7. run cook(3, Cook)}}}
   (-TRS: check the precondition of module cook-)
   $\texttt{d=3} : \{\texttt{Prep}\} \sqsubseteq \texttt{d>2} \wedge \{\texttt{Prep}\}$
   (-TRS: succeed-)
   $\langle \texttt{0≤t<3} : (\{\texttt{Prep}\} \cdot \{\texttt{Cook}\})\#\texttt{t} \rangle$ [FV-Call]
   $\langle \texttt{0≤t<3} : (\{\texttt{Prep}\} \cdot \{\texttt{Cook}\})\#\texttt{t} \cdot \{\texttt{Ready}\} \rangle$ [FV-Async]
8. $\langle (\texttt{true} \wedge \texttt{0≤t<3}) : \texttt{Ready?} \cdot \{\texttt{Go}\} \parallel (\{\texttt{Prep}\} \cdot \{\texttt{Cook}\})\#\texttt{t} \cdot \{\texttt{Ready}\} \rangle$ [FV-Fork]
   $\langle \texttt{0≤t<3} : (\{\texttt{Prep}\} \cdot \{\texttt{Cook}\})\#\texttt{t} \cdot \{\texttt{Ready}\} \cdot \{\texttt{Go}\} \rangle$ [Effects-Normalisation]
9. *(-TRS: check the postcondition of module main; Succeed, cf. Table 1.-)*
   $\texttt{0≤t<3} : (\{\texttt{Prep}\} \cdot \{\texttt{Cook}\})\#\texttt{t} \cdot \{\texttt{Ready}\} \cdot \{\texttt{Go}\} \sqsubseteq \texttt{0≤t<3} : \{\} \cdot (\{\texttt{Cook}\})\#\texttt{t} \cdot \{\}^\star$

Fig. 8. The forward verification example for the module `main`.

precondition is not satisfied, then the verification fails, otherwise it concatenates the postcondition of the callee $\Phi_{\texttt{post}}$ to the current effects.

$$\frac{\begin{array}{c}[\texttt{FV-Call}] \\ \texttt{nm}\ (\overrightarrow{\tau\ x},\ \overrightarrow{\tau\ \mathbf{S}})\ \langle \textbf{requires}\ \Phi_{\texttt{pre}}\ \ \textbf{ensures}\ \Phi_{\texttt{post}} \rangle\ \texttt{p} \in \mathcal{P} \\ \mathit{TRS} \vdash \Pi : \texttt{H} \cdot (\texttt{C\#T}) \sqsubseteq \Phi_{\texttt{pre}} \qquad \langle \Pi', \texttt{H}', \texttt{C}', \texttt{T}' \rangle = \texttt{split}\ (\Pi : (\texttt{H} \cdot (\texttt{C\#T}) \cdot \Phi_{\texttt{post}}))\end{array}}{\mathcal{E} \vdash \langle \Pi, \texttt{H}, \texttt{C}, \texttt{T} \rangle\ \texttt{run nm}\ (\overrightarrow{x}, \overrightarrow{\mathbf{S}})\ \langle \Pi', \texttt{H}', \texttt{C}', \texttt{T}' \rangle}$$

Definition 1 (Split). *Given any pure constraint π and a timed sequence* `es`*,* `split` *is to construct the corresponding program state $\langle \Pi, \texttt{H}, \texttt{C}, \texttt{T} \rangle$, by splitting* `es` *into a head* `C#T` *and a tail* `H`[8]*, formally,*

$$\texttt{split} :: (\pi : \texttt{es}) \rightarrow \langle \Pi, \texttt{H}, \texttt{C}, \texttt{T} \rangle \ \textit{where}\ \texttt{es} = (\texttt{C\#T}) \cdot \texttt{H}.$$

The resulted effects state of (7) is obtained by [FV-Async], which adds a new time instance {Ready} to the current effects state, indicating that the asynchronous program has been resolved. In step (8), we parallel the effects from both of the branches, and normalise the final effects. After these states transformations, step (9) checks the satisfiability of the inferred effects against the declared postcondition by invoking the TRS. The rewriting process of proving the postcondition is continued from the next section (cf. Table 1.).

[8] For example, $\texttt{split}(\texttt{T>10}, (\{\texttt{A}\} \cdot \{\texttt{B}\} \cdot \{\texttt{C}\})\#\texttt{T}) = (\texttt{T>10} \wedge \texttt{T=t1+t2}, \texttt{A}\#\texttt{t}_1, (\{\texttt{B}\} \cdot \{\texttt{C}\})\#\texttt{t}_2)$

5.1 Signal Interference Among Threads

Let us have a look at another example, shown in Fig. 9. Here it defines two signals: `I` as a local signal, and `J` as an output signal. Given the precondition is an empty trace, the first thread (in line 7) contains a conditional: if `I` is present, then emit `J`; otherwise do nothing. Therefore the first thread has non-deterministic effects: either both signals are present or both of them are absent (absent is the default value of all the signals).

Formally, Φ_1=true:$\{\mathtt{I}, \mathtt{J}\} \vee \{\overline{\mathtt{I}}, \overline{\mathtt{J}}\}$. Looking at the second thread, it simply emits signal `I`. Formally, Φ_2=true:$\{\mathtt{I}\}$. To synchronously parallel these two threads, i.e., $\Phi=\Phi_1||\Phi_2$, it results in Φ=true:$(\{\mathtt{I}, \mathtt{J}\} \vee \{\overline{\mathtt{I}}, \overline{\mathtt{J}}\})||\{\mathtt{I}\}$, which leads to Φ=true:$\{\mathtt{I}, \mathtt{J}, \mathtt{I}\} \vee \{\overline{\mathtt{I}}, \overline{\mathtt{J}}, \mathtt{I}\}$. We find that the second disjunction contains contradiction where $\overline{\mathtt{I}}$ and `I` both exist. Up to this point, *TempAS* would normalise [26] it into a deterministic effects Φ=true:$\{\mathtt{I}, \mathtt{J}\}$. This example is simple but rather important because it shows that our tool we tackles 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.

```
1 hiphop module prg (out J)
2 /*@ requires true: emp @*/
3 /*@ ensures true : {I,J} @
      */
4 {
5  signal I;
6  fork {
7     if( I ) {emit J};
8  }par {emit I;}
9 }
```

Fig. 9. Interference Among Threads.

Intuitively, our work effectively check logical correctness by 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.

6 The Back-End: A TRS

Our TRS is obligated to check the inclusions between timed synchronous effects, which is an extension of Antimirov and Mosses's algorithm. Antimirov and Mosses [6] present a term rewriting system for deciding the inequalities of regular expressions (REs), 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* [5].

The TRS is a automated entailment checker to prove language inclusions among timed synchronous 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 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 timed traces in the antecedent Φ_1 are legitimately

allowed in the possible timed 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=`true` $\wedge\, \epsilon$.

Theorem 1 (Timed Synchronous Effects Inclusion).
For timed effects Φ_1 *and* Φ_2, $\Phi_1 \sqsubseteq \Phi_2 \Leftrightarrow (\forall \mathtt{I} \forall \mathtt{t}).\ \mathtt{D}_{\mathtt{I}\#\mathtt{t}}(\Phi_1) \sqsubseteq \mathtt{D}_{\mathtt{I}\#\mathtt{t}}(\Phi_2)$.

Next, we continue with the step (9) in Fig. 8., to demonstrate how our TRS handle the extended arithmetic constraints and dependent values. As shown in Table 1., it automatically proves that the inferred effects of `main` satisfies the declared postcondition. We mark the rules of the rewriting steps in [gray], which are formally defined in Sec. 6. Note that time instance {`Prep`} entails {} because the former contains more constraints. Intuitively, we use [`DISPROVE`] wherever the left-hand side (LHS) is *nullable*[9] 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.

Table 1. The post condition proving process from example shown in Fig. 8.

$\mathtt{t_L}{<}3 \wedge \mathtt{t_L^1}{+}\mathtt{t_L^2}{=}\mathtt{t_L} \wedge \mathtt{t_L^2}{=}\mathtt{t_R} \Rightarrow \mathtt{t_R}{<}3 \quad \epsilon \sqsubseteq \{\}^\star$	⑧[PROVE]
$\mathtt{t_L}{<}3 \wedge \mathtt{t_L^1}{+}\mathtt{t_L^2}{=}\mathtt{t_L} \wedge \mathtt{t_L^2}{=}\mathtt{t_R} : \epsilon \sqsubseteq \mathtt{t_R}{<}3 : \bot \vee \{\}^\star$	⑦[UNFOLD]
$\mathtt{t_L}{<}3 \wedge \mathtt{t_L^1}{+}\mathtt{t_L^2}{=}\mathtt{t_L} \wedge \mathtt{t_L^2}{=}\mathtt{t_R} : \cancel{\{\mathtt{Go}\}} \sqsubseteq \mathtt{t_R}{<}3 : \cancel{\epsilon} \vee \cancel{\{\}} \cdot \{\}^\star$	⑥[Normalisation]
$\mathtt{t_L}{<}3 \wedge \mathtt{t_L^1}{+}\mathtt{t_L^2}{=}\mathtt{t_L} \wedge \mathtt{t_L^2}{=}\mathtt{t_R} : \{\mathtt{Go}\} \sqsubseteq \mathtt{t_R}{<}3 : \bot \vee \{\}^\star$	⑤[UNFOLD]
$\mathtt{t_L}{<}3 \wedge \mathtt{t_L^1}{+}\mathtt{t_L^2}{=}\mathtt{t_L} \wedge \mathtt{t_L^2}{=}\mathtt{t_R} : \cancel{\{\mathtt{Ready}\}} \cdot \{\mathtt{Go}\} \sqsubseteq \mathtt{t_R}{<}3 : \cancel{\epsilon} \vee \cancel{\{\}} \cdot \{\}^\star$	④[UNFOLD-UNIFY]
$\mathtt{t_L}{<}3 \wedge \mathtt{t_L^1}{+}\mathtt{t_L^2}{=}\mathtt{t_L} : \cancel{\{\mathtt{Cook}\}}\#\mathtt{t_L^2} \cdot \{\mathtt{Ready}\} \cdot \{\mathtt{Go}\} \sqsubseteq \mathtt{t_R}{<}3 : \cancel{\{\mathtt{Cook}\}}\#\mathtt{t_R} \cdot \{\}^\star$	③[UNFOLD]
$\mathtt{t_L^1}{+}\mathtt{t_L^2}{=}\mathtt{t_L}{<}3 : \cancel{\{\mathtt{Prep}\}}\#\mathtt{t_L^1} \cdot \{\mathtt{Cook}\}\#\mathtt{t_L^2} \cdot \{\mathtt{Ready}\} \cdot \{\mathtt{Go}\} \sqsubseteq \mathtt{t_R}{<}3 : \cancel{\{\}} \cdot \{\mathtt{Cook}\}\#\mathtt{t_R} \cdot \{\}^\star$	②[SPLIT]
$\mathtt{t_L}{<}3 : (\{\mathtt{Prep}\} \cdot \{\mathtt{Cook}\})\#\mathtt{t_L} \cdot \{\mathtt{Ready}\} \cdot \{\mathtt{Go}\} \sqsubseteq \mathtt{t_R}{<}3 : \{\} \cdot \{\mathtt{Cook}\}\#\mathtt{t_R} \cdot \{\}^\star$	①[RENAME]
$\mathtt{t}{<}3 : (\{\mathtt{Prep}\} \cdot \{\mathtt{Cook}\})\#\mathtt{t} \cdot \{\mathtt{Ready}\} \cdot \{\mathtt{Go}\} \sqsubseteq \mathtt{t}{<}3 : \{\} \cdot \{\mathtt{Cook}\}\#\mathtt{t} \cdot \{\}^\star$	

As shown in Table 1., in step ①, we rename the time variables to avoid the name clashes between the antecedent and the consequent. In step ②, we design a new rule [`SPLIT`] to accommodate the extended real-time constraints, which introduces two new time variables $\mathtt{t_L^1}$ and $\mathtt{t_L^2}$, and extends the previous constraint $\mathtt{t_L}{<}3$ with constraints $\mathtt{t_L^1}{+}\mathtt{t_L^2}{=}\mathtt{t_L}$. Then $\mathtt{t_L^1}$ and $\mathtt{t_L^2}$ mark the real-time constraint for time instances {`Prep`} and {`Cook`} respectively. Then in step ③, we eliminate {`Prep`} from the LHS and {} from the RHS, as the instances entailment {`Prep`} $\subseteq$ {} holds. In step ④, in order to conduct a further unfolding, we unify time variables $\mathtt{t_L^2}$ and $\mathtt{t_R}$ by adding the constraint $\mathtt{t_L^2}{=}\mathtt{t_R}$. In step ⑤, from the RHS, since $\{\}^\star{=}\epsilon \vee \{\} \cdot \{\}^\star$, eliminating one time instance {`Ready`} from it will lead to $\bot \vee \{\}^\star$, which can be normalised into $\{\}^\star$ in step ⑥. Note that

[9] If the event sequence is possibly empty, i.e. contains ϵ, we call it nullable.

the elimination of one time instance on a disjunction of traces is equivalent to disjunction of elimination of one time instance on each of the two traces. At the end of the rewriting, we manage to prove that $\mathtt{t_L}{<}3 \wedge \mathtt{t_L^1}{+}\mathtt{t_L^2}{=}\mathtt{t_L} \wedge \mathtt{t_L^2}{=}\mathtt{t_R} \Rightarrow \mathtt{t_R}{<}3$.

6.1 Prove it when Reoccur

Termination is guaranteed because the set of derivatives to be considered is finite, and possible cycles are detected using *memorization* [12], i.e., the rule [REOCCUR]. Check out this following example:

Table 2. The reoccurrence proving example.. I : The main rewriting proof tree; II : Right hand side sub-tree of the rewriting process.

$$
\mathtt{I}:\ \dfrac{\dfrac{\dfrac{\dfrac{\mathtt{True} \Rightarrow \mathtt{True} \quad \epsilon \sqsubseteq \epsilon}{\mathtt{True} : \epsilon \sqsubseteq \mathtt{True} : \epsilon}\ ④[\text{PROVE}]}{\mathtt{True} : \cancel{\{B\}} \sqsubseteq \mathtt{True} : \cancel{\epsilon}\ \cancel{\{B\}}}\ ③[\text{Normalisation}] \quad \mathtt{True} : \{B\} \sqsubseteq \mathtt{t_R}{<}4 : \{A\}^*\#\mathtt{t_R} \cdot \{B\}}{\mathtt{True} : \{B\} \sqsubseteq \mathtt{t_R}{<}4 : \{A\}^*\#\mathtt{t_R} \cdot \{B\}}\ ②[\text{UNFOLD}] \quad \mathtt{II} \quad \mathtt{t_L}{<}3 : \{A\}^*\#\mathtt{t_L} \cdot \{B\} \sqsubseteq \mathtt{t_R}{<}4 : \{A\}^*\#\mathtt{t_R} \cdot \{B\}}{\mathtt{t_L}{<}3 : \{A\}^*\#\mathtt{t_L} \cdot \{B\} \vee \mathtt{True} : \{B\} \sqsubseteq \mathtt{t_R}{<}4 : \{A\}^*\#\mathtt{t_R} \cdot \{B\}}\ ①[\text{Disj-L}]
$$

$$
\mathtt{II}:\ \dfrac{\dfrac{\dfrac{\dfrac{\mathtt{t_L^1}{+}\mathtt{t_L^2}{=}\mathtt{t_L} \wedge \mathtt{t_L}{<}3 \wedge \mathtt{t_R}{=}\mathtt{t_R^1}{+}\mathtt{t_R^2} \wedge \mathtt{t_L^1}{=}\mathtt{t_R^1} \wedge \mathtt{t_L^2}{=}\mathtt{t_R^2} \Rightarrow \mathtt{t_R}{<}4}{\cdots \wedge \cdots : \{A\}^*\#\mathtt{t_L^2} \cdot \{B\} \sqsubseteq \mathtt{t_R}{<}4 : \{A\}^*\#\mathtt{t_R^2} \cdot \{B\}}\ (\ddagger)}{\mathtt{t_L^1}{=}\mathtt{t_R^1} : \cancel{\{A\}\#\mathtt{t_L^1}}\cdot\{A\}^*\#\mathtt{t_L^2} \cdot \{B\} \sqsubseteq \mathtt{t_R}{<}4 : \cancel{\{A\}\#\mathtt{t_R^1}}\cdot\{A\}^*\#\mathtt{t_R^2} \cdot \{B\}}\ ⑦[\text{REOCCUR}]}{\mathtt{t_L^1}{+}\mathtt{t_L^2}{=}\mathtt{t_L} \wedge \mathtt{t_R}{=}\mathtt{t_R^1}{+}\mathtt{t_R^2} : (\{A\}\#\mathtt{t_L^1}\cdot\{A\}^*\#\mathtt{t_L^2}) \cdot \{B\} \sqsubseteq \mathtt{t_R}{<}4 : (\{A\}\#\mathtt{t_R^1}\cdot\{A\}^*\#\mathtt{t_R^2}) \cdot \{B\}}\ ⑥[\text{UNFOLD}]}{\mathtt{t_L}{<}3 : \{A\}^*\#\mathtt{t_L} \cdot \{B\} \sqsubseteq \mathtt{t_R}{<}4 : \{A\}^*\#\mathtt{t_R} \cdot \{B\}\ (\ddagger)}\ ⑤[\text{SPLIT}]
$$

As shown in Table 2., the LHS formula is a disjunction of two effects. An inclusion with a disjunctive antecedent succeeds if both disjunctions entail the consequent, shown in step ①. In step ② of the first sub-tree, in order to eliminate the first instance $\{\mathtt{B}\}$, $\{\mathtt{A}\}^*\#\mathtt{t_R}$ has to be reduced to be ϵ, therefore the time constraints have nothing to do in this branch. Then in steps ④, we can simply prove it with the side constraint entailment $\mathtt{True} \Rightarrow \mathtt{True}$ succeed.

Looking at the second sub-tree, in step ⑤, $\mathtt{t_L}$ and $\mathtt{t_R}$ are firstly split into $\mathtt{t_L^1}{+}\mathtt{t_L^2}$ and $\mathtt{t_R^1}{+}\mathtt{t_R^2}$. Then in step ⑥, $\{\mathtt{A}\}\#\mathtt{t_L^1}$ together with $\{\mathtt{A}\}\#\mathtt{t_R^1}$ are eliminated, unifying $\mathtt{t_L^1}$ and $\mathtt{t_R^1}$ (adding the constraint $\mathtt{t_L^1}{=}\mathtt{t_R^1}$). In step ⑦, we observe the proposition is isomorphic with one of the previous step, marked using (‡). Hence we apply the rule [REOCCUR] to prove it with a succeed side constraints entailment: $\mathtt{t_L^1}{+}\mathtt{t_L^2}{=}\mathtt{t_L} \wedge \mathtt{t_L}{<}3 \wedge \mathtt{t_R}{=}\mathtt{t_R^1}{+}\mathtt{t_R^2} \wedge \mathtt{t_L^1}{=}\mathtt{t_R^1} \wedge \mathtt{t_L^2}{=}\mathtt{t_R^2} \Rightarrow \mathtt{t_R}{<}4$.

6.2 Semantics of Await

We use Table 3. 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 $\{\overline{\mathtt{A}}\}$ refers to all the time instances containing A to be absent.

As shown in Table 3., the LHS $\{\mathtt{A}\}\cdot\{\mathtt{C}\}\cdot\mathtt{B}?\cdot\{\mathtt{D}\}$ entails the RHS $\{\mathtt{A}\}\cdot\mathtt{B}?\cdot\{\mathtt{D}\}$, as intuitively $\{\mathtt{C}\}\cdot\mathtt{B}?$ is a special case of $\mathtt{B}?$. In step ①, $\{\mathtt{A}\}$ is eliminated. In step ②, $\mathtt{B}?$ is normalised into $\{\mathtt{B}\}\vee(\{\overline{\mathtt{B}}\}\cdot\mathtt{B}?)$. By the step of ③, $\{\mathtt{C}\}$ is eliminated together with $\{\overline{\mathtt{B}}\}$ because $\{\mathtt{C}\}\subseteq\{\overline{\mathtt{B}}\}$. Now the rest part is $\mathtt{B}?\cdot\{\mathtt{D}\}\sqsubseteq\mathtt{B}?\cdot\{\mathtt{D}\}$. Here, we further normalise $\mathtt{B}?$ from the LHS into a disjunction, leading to two proof sub-trees. From the first sub-tree, we keep unfolding the inclusion with $\{\mathtt{B}\}$ (⑥) and $\{\mathtt{D}\}$ (⑦) till we can prove it. Continue with the second sub-tree, we unfold it with $\{\overline{\mathtt{B}}\}$; then in step ⑧ we observe the proposition is isomorphic with one of the previous step, marked using (‡). We prove it using the [REOCCUR] rule and finish the whole writing process.

Table 3. The example for Await.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\epsilon\sqsubseteq\epsilon}\,⑦[\text{PROVE}]}{\{\mathtt{D}\}\sqsubseteq\{\mathtt{D}\}}}{\{\mathtt{B}\}\cdot\{\mathtt{D}\}\sqsubseteq(\{\mathtt{B}\}\vee\bot)\cdot\{\mathtt{D}\}}\,⑥[\text{UNFOLD}] \quad \mathtt{B}?\cdot\{\mathtt{D}\}\sqsubseteq\mathtt{B}?\cdot\{\mathtt{D}\}\ (‡)}{\{\overline{\mathtt{B}}\}\cdot\mathtt{B}?\cdot\{\mathtt{D}\}\sqsubseteq(\bot\vee(\{\overline{\mathtt{B}}\}\cdot\mathtt{B}?))\cdot\{\mathtt{D}\}}\,⑧[\text{REOCCUR}]}{\mathtt{B}?\cdot\{\mathtt{D}\}\sqsubseteq\mathtt{B}?\cdot\{\mathtt{D}\}\ (‡)}\,⑤[\text{Disj-L}]}{\{\mathtt{C}\}\cdot\mathtt{B}?\cdot\{\mathtt{D}\}\sqsubseteq(\bot\vee(\{\overline{\mathtt{B}}\}\cdot\mathtt{B}?))\cdot\{\mathtt{D}\}}\,④[\text{Normalisation}]}{\{\mathtt{C}\}\cdot\mathtt{B}?\cdot\{\mathtt{D}\}\sqsubseteq(\{\mathtt{B}\}\vee(\{\overline{\mathtt{B}}\}\cdot\mathtt{B}?))\cdot\{\mathtt{D}\}}\,③[\text{UNFOLD}]}{\{\mathtt{A}\}\cdot\{\mathtt{C}\}\cdot\mathtt{B}?\cdot\{\mathtt{D}\}\sqsubseteq\{\mathtt{A}\}\cdot\mathtt{B}?\cdot\{\mathtt{D}\}}\,②[\text{Normalisation}]}{\mathtt{True}:\{\mathtt{A}\}\cdot\{\mathtt{C}\}\cdot\mathtt{B}?\cdot\{\mathtt{D}\}\sqsubseteq\mathtt{True}:\{\mathtt{A}\}\cdot\mathtt{B}?\cdot\{\mathtt{D}\}}\,①[\text{UNFOLD}]$$

7 Implementation and Case Studies

7.1 Implementation.

To show the feasibility of our approach, we have prototyped our automated verification system using OCaml, in roughly 10,000 LOC *(A demo page and test suits are available from [3])*. The proof obligations generated by the verifier are discharged using constraint solver Z3 [14]. We validate the front-end forward verifier for conformance against two implementations: the Columbia Esterel Compiler (CEC) [1] and Hiphop.js [2].

CEC 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, 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 by manually annotating temporal specifications in our timed effects, including both succeeded and failed instances. The remainder of this section presents some case studies.

7.2 Expressiveness of Synchronous 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. LTL

was subsequently extended to include the $\mathcal{U}$ ("until") operator and the $\mathcal{X}$ ("next time") operator. As shown in Table 4., 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. Extending to MTL, it allows us to specify properties such as $\Diamond_{\{2\}}\underline{\mathbf{A}}$, meaning that "Within 2 seconds, $\underline{\mathbf{A}}$ will finally be present", or $\Box_{[5,+\infty)}\underline{\mathbf{A}}$, meaning that "After the 5th second, $\underline{\mathbf{A}}$ is globally present". Furthermore, by putting the effects in the precondition, our approach naturally composites *past-time LTL* along the way.

Table 4. Examples for converting LTL/MTL formulae into Effects. ($\{\underline{\mathbf{A}}\}, \{\underline{\mathbf{B}}\}$ represent different time instants which contain signal **A** and **B** to be present.)

$\Box\underline{\mathbf{A}} \equiv \{\underline{\mathbf{A}}\}^\star$	$\Diamond\underline{\mathbf{A}} \equiv \{\}^\star \cdot \{\underline{\mathbf{A}}\}$	$\underline{\mathbf{A}}\ \mathcal{U}\ \underline{\mathbf{B}} \equiv \{\underline{\mathbf{A}}\}^\star \cdot \{\underline{\mathbf{B}}\}$
$\mathcal{X}\underline{\mathbf{A}} \equiv \{\} \cdot \{\underline{\mathbf{A}}\}$	$\Box\Diamond\underline{\mathbf{A}} \equiv (\{\}^\star \cdot \{\underline{\mathbf{A}}\})^\star$	$\Diamond\Box\underline{\mathbf{A}} \equiv \{\}^\star \cdot \{\underline{\mathbf{A}}\}^\star$
$\Diamond_{\{2\}}\underline{\mathbf{A}} \equiv \mathtt{t}{\leq}2 : \{\}^\star \# \mathtt{t} \cdot \{\underline{\mathbf{A}}\}$	$\Box_{[5,+\infty)}\underline{\mathbf{A}} \equiv \mathtt{t}{\leq}5 : (\{\}^\star \# \mathtt{t}) \cdot \{\underline{\mathbf{A}}\}^\star$	

Besides the high compatibility with standard first-order logic, the effects logic makes the temporal verification for synchronous languages more scalable. Because it avoids the *must-provided* translation schemas for each LTL temporal operator, as to how it has been done in the prior work [18].

7.3 Causality Checking

As shown in Fig. 10., our logic is also able to conduct causality checks. The example here presents a loop containing a conditional statement. For the first time instance, signals `I` and `O` can be both present or be both absent. However, from the second time instance, `O` will surely be emitted at line 6, which makes the second visit of the conditional deterministic, i.e., `I` and `O` can only be both present. Therefore, the final effects contain a disjunction at the first instance, followed by a determined repeated pattern.

```
1 hiphop module causality (out I,out O)
2 /*@ requires true : emp @*/
3 /*@ ensures true : ({I,O}\/{!I,!O}).{I,O}^* @*/
4 { loop {
5     if( O ) {emit I};
6     yield; emit O;}}
```

Fig. 10. Causality in a Loop.

7.4 A Gain on Constructiveness

We discovered a bug from the Esterel v5 *Constructive* semantics [7]. As shown in Fig. 11., this program is detected as "non-constructive" and rejected by CEC.

```
1 hiphop module a_bug ()
2 /*@ requires true : {} @*/
3 /*@ ensures true : {S} @*/
4 {
5     signal S;
6     if (S) {emit S;}
7     else {emit S;}
8 }
```

Fig. 11. A Bug Found

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. 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 applies symbolic execution and explores all the possible assignments to signals in a more efficient manner.

7.5 Discussion and Comparison to Timed Rebeca

As the examples show, our proposed effects logic tightly captures the behaviours of a mixed asynchronous and synchronous concurrency model and helps 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 supported by existing temporal verification techniques. On the other hand, our work draws the most similarity to Timed Rebeca [24], which offers an actor-based syntax and a built-in actor-based computational model, which restricts the style of modelling to an event-based concurrent object-based paradigm. In particular, partial-order reduction has been investigated for verifying Rebeca models. However, there are no claims of Timed Rebeca improving the expressiveness of timed automata. Whereas, our work aims to enhance the expressiveness of timed verification, i.e., intuitively, if traditional timed automata define an *exact* transition system, our timed effects define a set (possibly infinite) of exact transition systems.

8 Conclusion

In this paper, we briefly introduce *TempAS*, an automated tool capable of verifying temporal specifications for reactive systems. We target a core language $\lambda_{\texttt{HH}}$, generalizing the design of Hiphop.js, which provides the infrastructure for mixing asynchronous and synchronous concurrency models. We define the syntax of a novel temporal specification logic: timed synchronous effects, which captures reactive program behaviours and temporal properties. We develop a Hoare-style forward verifier to infer the program effects constructively. We develop an effects inclusion checker (the TRS) to prove/disprove the specified temporal properties efficiently. We present case studies and show the tool's feasibility.

To the best of our knowledge, *TempAS* is the first implementation that automates modular verification for reactive web-oriented language using an expressive effects logic. The tool can be tried out freely from the demo paper: `http://loris-5.d2.comp.nus.edu.sg/MixedSyncAsync/introduction.html`.

References

1. http://www.cs.columbia.edu/~sedwards/cec/, 2021.
2. https://github.com/manuel-serrano/hiphop, 2021.
3. http://loris-5.d2.comp.nus.edu.sg/MixedSyncAsync/index.html?ex=paper_example&type=hh&options=sess, 2021.
4. 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.
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. Gerard Berry. The constructive semantics of pure esterel-draft version 3. *Draft Version*, 3, 1999.
8. Gérard Berry. *The Esterel v5 language primer: version v5_91.* Centre de mathématiques appliquées, Ecole des mines and INRIA, 2000.
9. Gérard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of computer programming*, 19(2):87–152, 1992.
10. Gérard Berry and Manuel Serrano. Hiphop. js:(a) synchronous reactive web programming. In *PLDI*, pages 533–545, 2020.
11. 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.
12. 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.
13. 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.
14. Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.
15. Jin Song Dong, Ping Hao, Shengchao Qin, Jun Sun, and Wang Yi. Timed automata patterns. *IEEE Transactions on Software Engineering*, 34(6):844–859, 2008.
16. KLAUS V Gleissenthall, RAMI GÖKHAN Kici, ALEXANDER Bakst, DEIAN Stefan, and RANJIT Jhala. Pretend synchrony. POPL, 2019.
17. Dag Hovland. The inclusion problem for regular expressions. *Journal of Computer and System Sciences*, 78(6):1795–1813, 2012.
18. Lalita Jategaonkar Jagadeesan, Carlos Puchol, and James E Von Olnhausen. Safety property verification of esterel programs and applications to telecommunications software. In *International Conference on Computer Aided Verification*, pages 127–140. Springer, 1995.
19. 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.

20. Matthias Keil and Peter Thiemann. Symbolic solving of extended regular expression inequalities. *arXiv preprint arXiv:1410.3227*, 2014.
21. Kim G Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *International journal on software tools for technology transfer*, 1(1-2):134–152, 1997.
22. 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.
23. Cristian Prisacariu. Synchronous kleene algebra. *The Journal of Logic and Algebraic Programming*, 79(7):608–635, 2010.
24. Arni Hermann Reynisson, Marjan Sirjani, Luca Aceto, Matteo Cimini, Ali Jafari, Anna Ingólfsdóttir, and Steinar Hugi Sigurdarson. Modelling and simulation of asynchronous real-time systems using timed rebeca. *Science of Computer Programming*, 89:41–68, 2014.
25. 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.
26. 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.
27. Jun Sun, Yang Liu, Jin Song Dong, Yan Liu, Ling Shi, and Étienne André. Modeling and verifying hierarchical real-time systems using stateful timed csp. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(1):1–29, 2013.
28. 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.
29. 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.
30. Reinhard Von Hanxleden, Timothy Bourke, and Alain Girault. Real-time ticks for synchronous programming. In *2017 Forum on Specification and Design Languages (FDL)*, pages 1–8. IEEE, 2017.
31. Wang Yi, Paul Pettersson, and Mats Daniels. Automatic verification of real-time communicating systems by constraint-solving. In *Formal Description Techniques VII*, pages 243–258. Springer, 1995.

A Effects Inference Rules

$$\frac{\forall \mathbf{S} \in \mathcal{E}.\ \mathtt{C}[\mathbf{S} \mapsto \mathtt{Absent}]}{\mathcal{E} \vdash \langle \Pi, \mathtt{H}, \mathtt{C}, \mathtt{T} \rangle\ \mathtt{nothing}\ \langle \Pi, \mathtt{H}, \mathtt{C}, \mathtt{T} \rangle}\ [\mathtt{FV\text{-}Nothing}]$$

$$\frac{\mathtt{C}'{=}\mathtt{C}[\mathbf{S} \mapsto \mathtt{Present}]}{\mathcal{E} \vdash \langle \Pi, \mathtt{H}, \mathtt{C}, \mathtt{T} \rangle\ \mathtt{emit}\ \mathbf{S}\ \langle \Pi, \mathtt{H}, \mathtt{C}', \mathtt{T} \rangle}\ [\mathtt{FV\text{-}Emit}]$$

$$\frac{\Pi'{=}\Pi \wedge \mathtt{T}'{\geq}0\ \ \mathtt{C}'{=}\{\mathbf{S} \mapsto \mathtt{Undef} \mid \forall \mathbf{S} \in \mathcal{E}\}\ \ \mathtt{H}'{=}\mathtt{H} \cdot (\mathtt{C}\#\mathtt{t})\ \ (\mathtt{T}'\ \mathtt{is\ fresh})}{\mathcal{E} \vdash \langle \Pi, \mathtt{H}, \mathtt{C}, \mathtt{T} \rangle\ \mathtt{yield}\ \langle \Pi', \mathtt{H}', \mathtt{C}', \mathtt{T}' \rangle}\ [\mathtt{FV\text{-}Yield}]$$

$$\frac{\begin{array}{c}\mathcal{E} \vdash \langle \Pi, \mathtt{H}, \mathtt{C}[\mathbf{S} \mapsto \mathtt{Present}], \mathtt{T}\rangle\ \mathtt{p}\ \langle \Pi_1, \mathtt{H}_1, \mathtt{C}_1, \mathtt{T}_1\rangle \\ \mathcal{E} \vdash \langle \Pi, \mathtt{H}, \mathtt{C}[\mathbf{S} \mapsto \mathtt{Absent}], \mathtt{T}\rangle\ \mathtt{q}\ \langle \Pi_2, \mathtt{H}_2, \mathtt{C}_2, \mathtt{T}_2\rangle \\ \langle \Pi', \mathtt{H}', \mathtt{C}', \mathtt{T}'\rangle{=}\mathtt{cut}\ (\Pi_1 \wedge \Pi_2, \mathtt{H}_1 \cdot (\mathtt{C}_1 \# \mathtt{T}_1) \vee \mathtt{H}_2 \cdot (\mathtt{C}_2 \# \mathtt{T}_2))\end{array}}{\mathcal{E} \vdash \langle \Pi, \mathtt{H}, \mathtt{C}, \mathtt{T}\rangle\ \mathtt{present}\ \mathbf{S}\ \mathtt{p}\ \mathtt{q}\ \langle \Pi', \mathtt{H}', \mathtt{C}', \mathtt{T}'\rangle}\ [\mathtt{FV\text{-}Present}]$$

$$\frac{\begin{array}{c}\mathcal{E} \vdash \langle \Pi, \mathtt{H}, \mathtt{C}, \mathtt{T}\rangle\ \mathtt{p}\ \langle \Pi_1, \mathtt{H}_1, \mathtt{C}_1, \mathtt{t}_1\rangle \qquad \mathcal{E} \vdash \langle \Pi, \mathtt{H}, \mathtt{C}, \mathtt{T}\rangle\ \mathtt{q}\ \langle \Pi_2, \mathtt{H}_2, \mathtt{C}_2, \mathtt{t}_2\rangle \\ \langle \Pi', \mathtt{H}', \mathtt{C}', \mathtt{T}'\rangle{=}\mathtt{cut}\ (\mathtt{zip}\ (\Pi_1 : \mathtt{H}_1 \cdot (\mathtt{C}_1 \# \mathtt{T}_1),\ \Pi_2 : \mathtt{H}_2 \cdot (\mathtt{C}_2 \# \mathtt{T}_2)))\end{array}}{\mathcal{E} \vdash \langle \Pi, \mathtt{H}, \mathtt{C}, \mathtt{T}\rangle\ \mathtt{fork}\ \mathtt{p}\ \mathtt{q}\ \langle \Pi', \mathtt{H}', \mathtt{C}', \mathtt{T}'\rangle}\ [\mathtt{FV\text{-}Fork}]$$

$$\frac{\mathcal{E} \vdash \langle \Pi, \mathtt{H}, \mathtt{C}, \mathtt{T}\rangle\ \mathtt{p}\ \langle \Pi_1, \mathtt{H}_1, \mathtt{C}_1, \mathtt{T}_1\rangle \qquad \mathcal{E} \vdash \langle \Pi_1, \mathtt{H}_1, \mathtt{C}_1, \mathtt{T}_1\rangle\ \mathtt{q}\ \langle \Pi_2, \mathtt{H}_2, \mathtt{C}_2, \mathtt{T}_2\rangle}{\varrho \vdash \langle \Pi, \mathtt{H}, \mathtt{C}, \mathtt{T}\rangle\ \mathtt{seq}\ \mathtt{p}\ \mathtt{q}\ \langle \Pi_2, \mathtt{H}_2, \mathtt{C}_2, \mathtt{T}_2\rangle}\ [\mathtt{FV\text{-}Seq}]$$

$$\frac{\begin{array}{c}\mathcal{E} \vdash \langle \Pi, \epsilon, \mathtt{C}, \mathtt{T}\rangle\ \mathtt{p}\ \langle \Pi_1, \mathtt{H}_1, \mathtt{C}_1, \mathtt{T}_1\rangle \qquad \mathcal{E} \vdash \langle \Pi_1, \epsilon, \mathtt{C}_1, \mathtt{T}_1\rangle\ \mathtt{p}\ \langle \Pi_2, \mathtt{H}_2, \mathtt{C}_2, \mathtt{T}_2\rangle \\ \mathtt{H}'{=}\mathtt{H} \cdot \mathtt{H}_1 \cdot (\mathtt{H}_2 \cdot \mathtt{C}_2 \# \mathtt{T}_2)^\star \cdot \mathtt{H}_2\end{array}}{\mathcal{E} \vdash \langle \Pi, \mathtt{H}, \mathtt{C}, \mathtt{T}\rangle\ \mathtt{loop}\ \mathtt{p}\ \langle \Pi_2, \mathtt{H}', \mathtt{C}_2, \mathtt{T}_2\rangle}\ [\mathtt{FV\text{-}Loop}]$$

$$\frac{\begin{array}{c}\mathcal{E} \vdash \langle \Pi, \epsilon, \mathtt{C}, \mathtt{T}\rangle\ \mathtt{p}\ \langle \Pi', \mathtt{H}', \mathtt{C}', \mathtt{T}'\rangle \quad \Pi''{=}\Pi' \wedge (0 \leq \mathtt{T_d} < \mathtt{d}) \wedge (\mathtt{T_f} \geq 0) \\ \mathtt{H}''{=}(\mathtt{H}' \cdot (\mathtt{C}' \# \mathtt{T}'))\#\mathtt{T_d} \quad \mathtt{C}''{=}\{\mathbf{S} \mapsto \mathit{Undef} \mid \forall \mathbf{S} \in \mathcal{E}\} \quad (T_d,\ T_f\ \mathit{are\ fresh})\end{array}}{\mathcal{E} \vdash \langle \Pi, \mathtt{H}, \mathtt{C}, \mathtt{T}\rangle\ \mathtt{abort}\ \mathtt{p}\ \mathtt{when}\ \mathtt{d}\ \langle \Pi'', \mathtt{H} \cdot \mathtt{H}'', \mathtt{C}'', \mathtt{T_f}\rangle}\ [\mathtt{FV\text{-}Abort}]$$

$$\frac{\begin{array}{c}\mathcal{E} \vdash \langle \Pi, \epsilon, \mathtt{C}, \mathtt{T}\rangle\ \mathtt{p}\ \langle \Pi', \mathtt{H}', \mathtt{C}', \mathtt{T}'\rangle \qquad (T_d,\ T_f\ \mathit{are\ fresh}) \\ \Pi''{=}\Pi' \wedge (\mathtt{T_d} \geq \mathtt{d}) \wedge (\mathtt{T_f} \geq 0) \qquad \mathtt{H}''{=}(\mathtt{H}' \cdot (\mathtt{C}' \# \mathtt{T}'))\#\mathtt{T_d} \\ \mathtt{C}''{=}\{\mathbf{S}' \mapsto \mathit{Undef} \mid \forall \mathbf{S}' \in (\mathcal{E} \backslash \mathbf{S})\} \cup \{\mathbf{S} \mapsto \mathtt{Present}\}\end{array}}{\mathcal{E} \vdash \langle \Pi, \mathtt{H}, \mathtt{C}, \mathtt{T}\rangle\ \mathtt{async}\ \mathbf{S}\ \mathtt{p}\ \mathtt{d}\ \langle \Pi'', \mathtt{H} \cdot \mathtt{H}'', \mathtt{C}'', \mathtt{T_f}\rangle}\ [\mathtt{FV\text{-}Async}]$$

$$\frac{\Pi'{=}\Pi \wedge \mathtt{T}' \geq 0 \qquad \mathtt{H}'{=}\mathtt{H} \cdot (\mathtt{C} \# \mathtt{t}) \qquad (\mathtt{T}'\ \mathtt{is\ fresh})}{\mathcal{E} \vdash \langle \Pi, \mathtt{H}, \mathtt{C}, \mathtt{T}\rangle\ \mathtt{await}\ \mathbf{S}\ \langle \Pi', \mathtt{H}', \mathbf{S}?, \mathtt{T}'\rangle}\ [\mathtt{FV\text{-}Await}]$$

$$\frac{\mathit{TRS} \vdash \Pi \wedge (\mathtt{H} \cdot (\mathtt{C} \# \mathtt{T}) \sqsubseteq \Phi}{\mathcal{E} \vdash \langle \Pi, \mathtt{H}, \mathtt{C}, \mathtt{T}\rangle\ \mathtt{assert}\ \Phi\ \langle \Pi, \mathtt{H}, \mathtt{C}, \mathtt{T}\rangle}\ [\mathtt{FV\text{-}Assert}]$$