SYMBOLIC TEMPORAL VERIFICATION TECHNIQUES WITH EXTENDED REGULAR EXPRESSIONS

YAHUI SONG

(M.Sc., National University of Singapore)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2023

Supervisor:
Associate Professor Chin Wei Ngan

Examiners:
Professor Joxan Jaffar
Professor Dong Jin Song

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

Yahui Song

Yahui Song

May 2023

Acknowledgments

I acknowledge the financial support from the National University of Singapore.

My sincere thanks to my supervisor, Dr. Wei Ngan Chin, for guiding this work. I am thankful for the extraordinary experiences he arranged for me and for providing opportunities for me to grow professionally. It is an honor to learn from him.

I thank my thesis examiners: Dr. Joxan Jaffar and Dr. Jin Song Dong, for their insightful comments and encouragement, which incentivized me to widen my research from various perspectives.

I thank our faculty members who have offered me lectures or shared inspiring suggestions with me during my graduate school study: Dr. Abhik Roychoudhury, Dr. Siau Cheng Khoo, Dr. Roland Yap, Dr. Reza Shokri, Dr. Olivier Danvy, Dr. Ilya Sergey, and Dr. Aquinas Hobor.

I thank Dr. Bimlesh Wadhwa for her patient guidance in my teaching work. I thank Dr. Shengchao Qin and Dr. Mengda He for their guidance during my short stay for the internship at Huawei.

I am fortunate to have been a part of the Programming Language lab in School of Computing. I thank my senior colleagues: Dr. Andreea Costea, Dr. Quang-Trung Ta, and Dr. Thanh-Toan Nguyen, for their generous help. I thank my peer Ph.D. candidates: Yuyi Zhong, Darius Foo, Yunjeong Lee, Ruiwei Wang, Wenhua Li, Kiran Gopinathan, and George Pîrlea, for all the cheerful conversations and collaborations.

Most importantly, I thank my parents and sister for being my best friends, and for their unconditional love and continuous encouragement.

Contents

Summary

Existing temporal verification approaches have sacrificed modularity in favor of achieving automation or vice-versa. To exploit the best of both worlds, this thesis presents a new framework to ensure temporal properties via Hoare-style verifiers and term rewriting systems (TRSs).

The leading technique of temporal verification is automata-based model checking, which has the following inadequacies: firstly, it requires a manual modeling stage; secondly, it needs to be bounded due to the lack of symbolic reasoning; and lastly, the expressiveness power is limited by the finite-state automata.

To tackle these issues, this thesis proposes a framework that conducts local temporal verification, leading to a modular and compositional verification strategy, where modules can be replaced by their already verified properties. In our exploration, we proposed various *effect logics* to be the temporal specifications, which are extended regular expressions (REs) and more flexible/expressive than the most deployed linear temporal logic (LTL). Furthermore, the proposed framework devises purely algebraic TRS to check the inclusions for the novel logics, avoiding the complex translation into automata.

This thesis demonstrates the applicability of the proposed framework and various REs-based temporal logics in different domains, such as synchronous programming, real-time systems, algebraic effects, etc. This thesis also presents the corresponding prototype systems, case studies, experimental results, and supporting proofs.

List of Figures

List of Tables

Chapter 1

Introduction

This thesis is interested in automatic verification using finite-state, yet possibly non-terminating models of systems, with the underlying assumption that linear-time system behavior can be represented as a set of traces representing all the possible histories. In this model, verification consists of checking for language inclusion: the implementation describes a set of actual traces, in an automaton $\mathcal{A}$; and the specification gives the set of allowed traces, in an automaton $\mathcal{B}$; the implementation meets the specification if every actual trace is allowed, i.e., $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$.

The leading community of temporal verification is automata-based model checking, which deploys mainstream temporal logic specifications, such as LTL and CTL. Classical model checkers extract the logic design from the program using modeling languages and verifying specific assertions to guarantee various properties. The verification is based on a translation from modeling languages/specifications to automata and the well-tuned inclusion checking algorithm for automata.

The current inadequacies of model-checking techniques are: firstly, it requires a manual modeling stage; secondly, it needs to be bounded due to the lack of symbolic reasoning; thirdly, the expressiveness power is limited by the finite-state automata; and lastly, the expressiveness power is limited by the finite-state automata.

To further elaborate the efficiency issue on (ii) above, deciding the inclusion between two finite-state automata is PSPACE-complete. The standard approaches to the problem are based on the following steps: translate logical expressions into equivalent nondeterministic finite automaton (NFA); convert the NFA to equivalent deterministic finite automatons (DFA); minimize the DFA to $\mathcal{M}_{\mathcal{A}}$ and $\mathcal{M}_{\mathcal{B}}$; and finally check emptiness of $\mathcal{M}_{\mathcal{A}} \cap \neg\mathcal{M}_{\mathcal{B}}$. However, any efficient algorithm [Wul+06] based on such translation potentially gives rise to an exponential blow-up.

Therefore, this thesis is motivated to find a more precise, extensive and efficient solution for temporal verification. More specifically, it proposes a new framework which deploys Hoare-style forward verifiers as the front-ends, and TRSs as the back-ends. Forward verifier compute the actual temporal behaviors from the source code, based on the formally-defined execution semantics of the target languages. TRSs are decision procedures inspired by Antimirov and Mosses's algorithm [AM95] but solving the language inclusions between more expressive temporal logics.

Antimirov and Mosses's rewriting algorithm can be used as an alternative approach to the automata-based inclusion-checking approach. More specifically, it decides inequalities of regular expressions (REs) through an iterated process of checking the inequalities of their *partial derivatives* (cf. Definition 1 and Definition 2) [Ant95]. There are two basic rules: [*Disprove*], which infers *false* from trivially inconsistent inequalities; and [*Unfold*], which applies Theorem 1 to generate new inequalities. Termination is guaranteed because the set of derivatives to be considered is finite, and possible cycles are detected using *memorization*, i.e., a set of proof hypotheses soundly derived during the proof search.

Definition 1 (Derivative)**.** Given any formal language S over an alphabet Σ and any string $u \in \Sigma^*$, the derivative of S with respect to u is defined as:

$$u^{-1}S = \{w \in \Sigma^* \mid u \cdot w \in S\}, \textit{ where } \cdot \textit{ denotes trace concatenation.}$$

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

$$a^{-1}S = \{w \in \Sigma^* \mid a \cdot w \in S\}.$$

Theorem 1 (Regular Expressions Inequality (Antimirov))**.** *Given Σ is a finite set of alphabet, $\boldsymbol{A}^{-1}(r)$ is the partial derivative of r with respect to the symbol $\boldsymbol{A}$, given two REs r and s, their inequality is defined as: $r \preceq s \Leftrightarrow \forall(\boldsymbol{A} \in \Sigma).\ \boldsymbol{A}^{-1}(r) \preceq \boldsymbol{A}^{-1}(s)$.*

Works based on such a TRS [SC20; AM95; AMR09; KT14a; Hov12; Bjø+01; KMP00; ÖM02; Ölv00] show its feasibility and suggest that this approach is a *better average-case* algorithm than those based on the comparison of minimal DFA. Thus, this work investigates the possibilities of applying such TRSs (upon extended regular expressions) to serve as back-end solvers for different temporal verification contexts.

1.1 Verification Framework Overview

The proposed verification framework is shown in Figure 1.1. Rounded boxes are the main procedures. They return *true* when the forward reasoning or the effects inclusion proving succeeds, respectively. They return *false* otherwise. Rectangular boxes describe the inputs to the procedures. The forward verifier relies on the TRS to solve temporal proof obligations, in the form of effects inclusions. The TRS discharges arithmetic proof obligations – generated while solving effects inclusions – by state-of-the-art SMT solvers Z3 [dMB08], represented by the grey box.

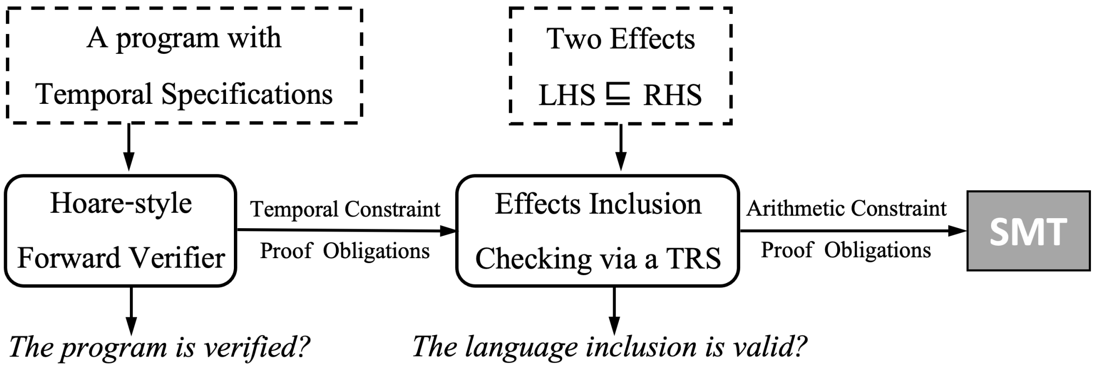

Figure 1.1: Verification Framework Overview.

The inputs of the forward verifier are target programs annotated with temporal specifications. The input of the TRS is a pair of effects LHS and RHS, referring to the inclusion LHS $\sqsubseteq$[1] RHS to be checked. *(Note that LHS refers to left-hand-side effects, and RHS refers to right-hand-side effects.)*

Based on the proposed verification framework, this thesis presents several independent works on different applicable domains, which deploy different effect logics for different targeting languages, with different back-end TRSs accordingly. Altogether,

[1] $\sqsubseteq$ captures the inclusion relation between effects, defined based on different effect logics, in Definition 6 (for *DependentEffs*), Definition 13 (for *ASyncEffs*), Definition 20 (for *TimEffs*), and Definition 26 (for *ContEffs*) respectively.

this work investigates: the feasibility of the proposed framework; the extensible expressiveness of regular expressions; and the efficiency of TRSs.

Similarities to the Types-and-effect systems. The design and implementation of a correct system can benefit from employing static techniques for ensuring that the dynamic behaviour satisfies the specification. Many programming languages incorporate types for ensuring that certain operations are only applied to data of the appropriate form. A natural extension of type checking techniques is to enrich the types with annotations and effects that further describe intensional aspects of the dynamic behaviour. Types-and-effect systems [Nie+99] refine the type information by annotating the types so as to express further intensional or extensional properties of the semantics of the program [Jou87; JG89; JG91; LG88].

The verification framework proposed here draws similarities to Types-and-effect systems as it assumes the input programs are type-checked; but differs in that it only focuses on the reasoning for the effects in the form of symbolic traces. The effects reasoning does not need to depend on the type systems; therefore, it can be deployed separately by languages that do not have static typing checking mechanisms.

1.2 Thesis Synopsis

This thesis instantiates the above general framework with several independent works to show its applicability. Each of them has different input programs with varying specification languages. Their forward verifiers and TRSs differ regarding the program semantics and the effect logic. As shown in Table 1.1, Chapters 3-6 present different possible effect logics for different verification contexts:

1. ***DependentEffs*** (Chapter 3) targets general-purpose sequential programs and integrates the basic and ω-regular expressions with dependent values and arithmetic constraints, denote the number of repetitions of a trace, gaining the expressive power beyond LTL, μ-calculus, and prior effect logics.

2. ***ASyncEffs*** (Chapter 4) targets a preemptive asynchronous execution model by integrating the Synchronous Kleene Algebra (SKA) [Pri10; Bro+15] with a new

Table 1.1: Proposals Overview

	Target Language	**Specification Language**	**Applied Domain**	**Research Paper**
Chapter 3	C	*DependentEffs*	General Effectful Programs	[SC20]
Chapter 4	Esterel	*ASyncEffs*	Reactive Systems	[SC21]
Chapter 5	C^t	*TimEffs*	Time Critical Systems	[SC22; SC23]
Chapter 6	λ_h	*ContEffs*	Algebraic Effects and Handlers	[SFC22]

operator, to provide *block waiting* (among threads) abstraction into traditional synchronous verification.

3. ***TimEffs*** (Chapter 5) targets real-time systems with not only timed behavioral patterns but also shared variables and concurrency, by integrating regular expressions with dependent values and arithmetic constraints, to denote symbolic real-time bounds, providing a more modular and expressive timed verification.

4. ***ContEffs*** (Chapter 6) targets user-defined effects and handlers and integrates $\star$ for finite traces; ω for infinite traces; ∞ for possibly finite and possibly infinite traces. *ContEffs* provides a general reasoning technique for the coexistence of zero-shot, one-shot and multi-shot continuations. Furthermore, it also helps to detect non-terminating behaviors while using effect handlers.

The rest of this thesis is organized as: Chapter 2 presents the literature review; Chapter 7 summarizes the thesis and discusses possible future works; Appendix A, Appendix B and Appendix C demonstrate the supporting proofs.

Chapter 2

Literature Review

Recently, temporal reasoning has garnered renewed popularity and importance for possibly non-terminating programs with subtle use of recursion and non-determinism, as used in reactive or time-critical applications.

2.1 General Perspectives of Temporal Verification

This section first discusses the related works in the following general perspectives: automata-based model checking and traces-based effects systems (subsection 2.1.1); expressive effect logics (subsection 2.1.2); and efficient algorithms for language inclusion checking (subsection 2.1.3).

2.1.1 Model Checking and Effects Systems

A vast range of techniques has been developed for the prediction of program temporal behaviors without actually running the system. One of the leading communities of temporal verification is automata-based model checking, mainly for finite-state systems. Various model checkers, such as PAT [Sun+09], Uppaal [LPY97], and TLA+ [Lam+02], etc., are based on classic temporal logic specifications, such as LTL and CTL. Such tools extract the logic design from the program using modeling languages and verify assertions to guarantee various safety/liveness properties. However, classical model-checking techniques usually require a manual modeling stage and need to be bounded when encountering non-terminating traces.

On the other hand, combining program events with a temporal program logic to assert properties of event traces yield a powerful and general engine for enforcing program properties. Prior works [SSV08; SS04a; MSS03] have demonstrated that static approximations of program event traces can be generated by type and effect analyses [TJ94; ANN99], in a form amenable to existing model-checking techniques for verification. These approximations are called trace-based effects. Trace-based analyses have been shown capable of statically enforcing flow-sensitive security properties such as safe locking behaviors [FTA02], and resource usage policies such as file usage protocols and memory management [MSS03]. A trace effect analysis is used to enforce secure service composition [BDF05]. Stack-based security policies are also amenable to this form of analysis [SS04b].

More related to this thesis, prior research has been extending Hoare logic with event traces. Prior work [MMW11] focuses on finite traces (terminating runs) for web applications, leaving the divergent computation, which indicates *false*, simply verified for every specification. Prior work [NU10] focuses on infinite traces (non-terminating runs) by providing coinductive trace definitions. Together with the prior work [Bub+15], this thesis works on dynamic logic and unified operators to reason about possibly finite and infinite traces simultaneously. The soundness is guaranteed via the theoretical foundations for reasoning about inductive and coinductive definitions simultaneously [Bro05b].

Moreover, the proposed effect logics draw similarities to *contextual effects* [Nea+08], which takes the already occurred events as the history effects; the events which are not yet happened as the future effects. Besides, prior work [ANN99] proposes an annotated type and effect system and infers behaviors from CML [Rep93] programs for channel-based communications, though it did not provide any language inclusion checking solutions.

2.1.2 Expressive Effect Logics

To conduct temporal reasoning locally, there is a sub-community whose aim is to support temporal specifications in the form of *effects* via the type-and-effect system. The inspiration from this approach is that it leads to a modular and compositional verification strategy, where temporal reasoning can be done locally and combined to reason about the overall program [HC14; KT14b; Nan+18]. However, the temporal effects in prior work tend to over-approximate program behaviors either via ω-regular expressions [HC14] or by büchi automata [KT14b]. One of the recent works [Nan+18] proposes the dependent temporal effects on program input values, which allows the reasoning on infinite input alphabet, but still loses the precision of the branching properties. These conventional effects have the form (Φ_u, Φ_v), which separates the finite and infinite effects. In our proposals, by integrating possibly finite and possibly infinite effects into a single disjunctive form the effect logics eliminate the finiteness distinction, and enable an expressive modular temporal verification.

2.1.3 Efficient Algorithms for Language Inclusion Checking

There are no existing finite-state automata capable of expressing the proposed effect logics: *DependentEffs*, *ASyncEffs*, *TimEffs* and *ContEffs*; neither are there corresponding language inclusion checking algorithms. We here reference two efficient prior works targeting basic regular sets: Antichain-based algorithms and the traditional TRS, which are both avoiding the explicit, complex translation from the NFA into their minimal DFA. (However, generally, it is unavoidable for any language inclusion checking solutions to have exponential worst-case complexity.)

Antichain-based algorithm [Wul+06] was proposed for checking universality and language inclusion for finite word automata. By investigating the easy-to-check pre-order on set inclusion over the states powerset, Antichain is able to soundly prune the search space, therefore it is more succinct than the sets of states manipulated by the classical fixpoint algorithms. It significantly outperforms the classical subset construction, in many cases, it still suffers from the exponential blow up problem.

The main peculiarity of a purely algebraic TRS [Ter03; AM95; KT14a] is that it provides a reasoning logic for regular expression inclusions to avoid any kind of translation aforementioned. Specifically, as defined in Theorem 1, a TRS takes finite steps to reduce $r \preceq s$ into its normal form $r' \preceq s'$ and the inclusion checking fails whenever $r' \preceq s'$ is not valid. A TRS is shown to be feasible and, generally, faster than the standard methods, because (i) it deploys the heuristic refutation step to disprove inclusions earlier; (ii) it prunes the search space by using fine-grained normalization lemmas. Overall, it provides a better average-case performance than those based on the translation to minimal DFA. More importantly, a TRS allows us to accommodate infinite traces and capture value-dependent properties.

In this thesis, we choose to deploy *extended TRSs*, which combine optimizations from both Antichain-based algorithms and classical TRS. Having such a TRS as the back-end to verify temporal effects, one can benefit from the high efficiency without translating effects into automata. More importantly, this thesis generalizes Antimirov and Mosses's rewriting procedure [AM95], to further reason about infinite traces, together with value-dependent properties and arithmetic constraints. One of the direct benefits granted by the effect logic is that it provides the capability to check the inclusion for possibly finite and infinite event sequences without a deliberate distinction, which is already beyond the strength of existing classical TRS [AM95; AMR09; KT14a; Hov12].

2.2 Domain Specific Related Works

This section elaborates domain-related contexts for the target languages in Chapter 4, Chapter 5, and Chapter 6 respectively: preliminaries of the preemptive asynchronous concurrency model (subsection 2.2.1); real-time verification (subsection 2.2.2); and type-and-effect systems for algebraic effects (subsection 2.2.3).

2.2.1 Preemptive Asynchronous Concurrency Model

In Chapter 4, we propose *ASyncEffs*, to reason about a preemptive asynchronous concurrency model. The mixture is essentially a combination of Esterel's synchrony [Ber99] with JavaScript's asynchrony [MLT17]. This section discusses existing semantic models for Esterel and JavaScript; temporal verification for synchronous language Esterel. Afterward, it compares *ASyncEffs* with the Synchronous Kleene Algebra, which shares the most similarities to our work.

2.2.1.1 Semantics of Esterel and JavaScript's asynchrony

The web orchestration language HipHop.js [BS20] integrates Esterel's synchrony with JavaScript's asynchrony, which provides the infrastructure for our work on preemptive asynchronous execution models. To the best of the author's knowledge, the forward reasoning rules in our 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 kernel of the Esterel synchronous reactive language, prior work gave two semantics, a macrostep logical semantics called the behavioral semantics [Ber99], a small-step semantics called execution/operational semantics [BG92], and a calculus for bounded input signals [Flo+19]. Our forward reasoning rules closely follow the work of state-based semantics [Ber99]. In particular, we borrow the idea of internalizing state into effects using *history* trace and *current* event at any level in a program. As the existing semantics are not ideal for compositional reasoning in 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 [Ecm99]) asynchronous programs, which can be written in a synchronous style, leading to more scalable code (comparing to the callback-based asynchronous code). However, the ECMAScript 6 standard specifies the semantics of promises informally and in operational terms, unsuitable for formal reasoning or program analysis. Prior works [MLT17; Ali+18], 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 Chapter 4, we propose to combine the operational semantics of synchronous (preemptive) Esterel and the asynchronous constructs in JavaScript, building the language foundation for such a blending of two distinct execution models.

2.2.1.2 Temporal Verification of Esterel

In prior work [JPO95], given an LTL formula, they first recursively translate it into an Esterel program whose traces correspond to the computations that violate the safety formula. The program derived from the formula is then combined with the given Esterel program to be verified. The program resulting from this composition is compiled using available Esterel tools; a trivial analysis of the compiler's output then indicates whether or not the property is satisfied by the original program. The Esterel compiler performs model checking by exhaustively generating all the composed program's reachable states.

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

Similarly, extending from Antimirov's notions of partial derivatives, prior work [Bro+15] presented a decision procedure for equivalence checking between Synchronous Kleene Algebra (SKA) terms. The following subsection discusses the similarities and differences between this work and [Bro+15].

2.2.1.3 Synchronous Kleene Algebra (SKA)

Kleene algebra (KA) is a decades-old sound and complete equational theory of regular expressions. The *ASyncEffs* logic draws similarities to SKA [Pri10], which is KA extended with a synchrony combinator for actions. Formally, given a KA is $(A, +, \cdot, \star, 0, 1)$, a SKA over a finite set A_B is $(A, +, \cdot, \times, \star, 0, 1, A_B)$, $A_B \subseteq A$. The $\bot$ (*false*) corresponds to the 0; the ϵ (empty trace) corresponds to the 1; the *time instance* containing simultaneous signals can be expressed via $\times$; and the *instants subsumption* (Definition 10) is reflected by SKA's *demanding relation*.

Presently, SKA allows the synchrony combinator $\times$ to be expressed over any two SKA terms to support concurrency. *ASyncEffs* achieves a similar outcome by supporting normalization operations during trace synchronization, via a *zip* function in the forward rule of [*FV-Par*] (c.f. section 4.4). This leads to one major difference in the inclusion/equivalence checking procedure, whereby a TRS for SKA would have to rely on *nullable*, *first*, and *partial derivatives* for terms constructed by the added combinator $\times$, but carefully avoided by the TRS construction. While the original equivalence checking algorithm for SKA terms in [Pri10] has relied on well-studied decision procedures based on classical Thompson ϵ-NFA construction, [Bro+15] shows that the use of Antimirov's partial derivatives could result in better average-case algorithms for automata construction. The proposal in Chapter 4 avoided the consideration for the more general $\times$ operation from SKA a customized TRS for inclusion (instead of equivalence) checking.

Apart from the synchrony combinator, the effect logic also introduced the ω constructor to explicitly distinguish infinite traces from the coarse-grained repetitive operator Kleene star $\star$. The inclusion of ω constructor allows us to support non-terminating reactive systems that are often supported by temporal specification and verification to ensure systems' dependability. As a consequence, the backend TRS is designed to be able to reason soundly about both finite traces (inductive definition), and infinite traces (coinductive definition), using cyclic proof techniques [Bro05a].

Another extension from the ready-made KA theory is Kleene algebra with tests (KAT), which provides solid mathematical semantic foundations for many domain-specific languages (DSL), such as NetKAT [And+14], designed for network programming. In KAT, actions are extended with boolean predicates, and the negation operator has been added accordingly. *ASyncEffs* also similarly support the boolean algebra since each signal can be explicitly specified as either present or absent. Such contradictions of signal status are also explicitly captured by $\bot$ (*false*).

2.2.2 Real-Time Verification

In Chapter 5, we propose *TimEffs*, to conveniently specify *Symbolic Timed Automata*, with efficient back-end solving of clock constraints. This work overcomes the main limitations of traditional timed model checking: i) Timed Automata (TAs) cannot be used to specify/verify incompletely specified systems (i.e., whose timing constants have yet to be known) and hence cannot be used in early design phases; ii) verifying a system with a set of timing constants usually requires enumerating all of them if they are supposed to be integer-valued; iii) TAs cannot be used to verify systems with timing constants to be taken in a real-valued dense interval.

This section first discusses the existing compositional model checking for real-time systems, which draws the most similarities to our work, deploying *implicit clocks*. Then it presents an *explicit clock* approach as a comparison. Lastly, it discusses other usages of efficient clock manipulation and zone-based bi-simulation.

2.2.2.1 Compositional Model Checking for Real-Time Systems

Although TAs' simple structure made efficient model checking feasible, specifying and verifying compositional real-time systems is challenging due to the increasing complexity. TAs lack high-level compositional patterns for hierarchical design; moreover, users often need to manually manipulate clock variables with carefully calculated clock constraints. The process is tedious and error-prone.

There have been some translation-based approaches on building verification support for compositional timed-process representations. For example, Timed Communicating Sequential Process (TCSP), Timed Communicating Object-Z (TCOZ) and *Statechart* based hierarchical Timed Automata are well suited for presenting compositional models of complex real-time systems. Prior works [Don+08; DM01] systematically translate TCSP/TCOZ/Statechart models to flat TAs so that the model checker Uppaal [LPY97] can be applied.

We are of the opinion that in that the goal of verifying real-time systems, in particular safety-critical systems is to check logical temporal properties, which can be done without constructing the whole reachability graph or the full power of model-checking. We consider our approach (in Chapter 5) is simpler as it is based directly on constraint-solving techniques and can be fairly efficient in verifying systems consisting of many components as it avoids to explore the whole state-space [SC20; YPD94].

2.2.2.2 TLA+: the Explicit Time Approach

Opposite of the long-established implicit clock approaches, Leslie Lamport proposed an explicit time approach in [Lam05]. In an explicit-time specification, time is represented with a variable *now* that is incremented by a *Tick* action. For a continuous-time specification, *Tick* might increment *now* by any real number; for a discrete-time specification, it increments *now* by 1. Timing bounds on actions are specified with one of three kinds of timer variables: a countdown timer is decremented by the *Tick* action, a count-up timer is incremented by *Tick*, and an *expired* timer is left unchanged by *Tick*. A countdown or count-up timer expires when its value reaches some value; an expiration timer expires when its value minus *now* reaches some value. An upper-bound timing constraint on when an action **A** must occur is expressed by an enabling condition on the *Tick* action that prevents an increase in time from violating the constraint; a lower-bound constraint on when **A** may occur is expressed by an enabling condition on **A** that prevents it from being executed earlier than it should be.

The results reported in [Lam05] indicate that verifying explicit-time specifications with an ordinary model checker is not very much worse than using a real-time model checker. However, the main advantage of an explicit-time approach is the ability to use languages and tools not specially designed for real-time model checking. This is taken to be important for complex algorithms that can be quite difficult to represent in the lower-level, inexpressive languages typical of real-time model checkers. For example, distributed message-passing algorithms have queues or sets of messages in transit, each with a bound on its delivery time. Such algorithms are difficult to handle with most real-time model checkers.

2.2.2.3 (Implicit) Clock Manipulation and Zone Abstraction

Besides Timed CSP, the concept of implicit clocks has also been used in time Petri nets, and implemented in a number of model checking engines, e.g. [BV06]. On the other hand, to make model checking more efficient with *explicit* clocks, [DY96; BC13; MWP13; GNA14] work on dynamically deleting or merging clocks.

The constraint-solving techniques in our *TimEffs*' inclusion checking, also draw connections with region/zone-based bisimulations [LGL19], which is broadly used in reasoning timed automata. Zone abstraction constructs zone graphs, which is an effective technique for checking both safety and liveness properties and it has been developed in [LPY97; Tri99]. Different from zone abstraction applied to TA, we dynamically create or delete a set of clocks to encode the timing requirements.

2.2.3 User-defined Effects and Handlers

Static program analysis for algebraic effects is challenging because they produce complex and less restricted execution traces due to the composable non-local control flow mechanisms. In Chapter 6, we propose *ContEffs*, providing modular specifications for user-defined effects and handlers. This section first discusses the existing type-and-effect systems for algebraic effects, and then discuss the existing logics for reasoning algebraic effects.

2.2.3.1 Type-and-effect Systems for Algebraic Effects

Many languages with algebraic effects are equipped with type-and-effect systems – which enrich existing types with information about effects – to allow the effect-related behaviors of functions to be specified and checked. A common approach of doing this is row-polymorphic effect types, used by languages such as Koka [Lei14b; Daa17], Helium [Bie+19; Dar+19], Frank [LMM16], and Links [LC12]. An effect *row* specifies a multi-set of effects a function may perform, and is popular for its simplicity, expressiveness (naturally enabling *effect polymorphism*), and support for inference of principal effects [Lei14b]. There are numerous extensions to this model, including *presence types* attached to effect labels, allowing one to express the *absence* of an effect [LC12], existential and local effects for modularity [Bie+19], and linearity [Lei18]. Other choices include sets of (instances of) effects [Dar+19], and structural sub-typing constraints [Pre13].

In our work, we take a step further to consider finer-grained specifications *ContEffs*, which concerns the order of effect labels expressed as temporal properties.

2.2.3.2 A Separation Logic for Effect Handlers

Recent work [dVP21] proposes a separation logic for effects and handlers, and provides a proof environment in Coq [Coq22] to manually prove properties for algebraic effects. Another work [BP14] presents an effect system for a simplified variant of *Eff* language. [BP14] provides equational reasoning for programs by providing a fine-grained denotational semantics, which involves mutable states. However, none of the prior work considers the coexistence of zero-shot, one-shot and multi-shot continuations and the non-termination behaviors caused by the *deep handlers*.

Although our work targets pure programs, i.e., without mutable global states, our logic *ContEffs* facilitates an automated verification framework and fills up the gaps mentioned above in the perspective of temporal properties. We take it as future work to add spatial information into our current *ContEffs*.

Chapter 3

Dependent Effects (*DependentEffs*)

Existing approaches to temporal verification have either sacrificed compositionality in favor of achieving automation or vice-versa. To exploit the best of both worlds, we present a new solution to ensure temporal properties via a Hoare-style verifier and a term rewriting system (TRS) on *DependentEffs*. The first contribution is a novel effect logic capable of integrating value-dependent finite and infinite traces into a single disjunctive form, resulting in more concise and expressive specifications. As a second contribution, by avoiding the complex translation into automata, our purely algebraic TRS efficiently checks the language inclusion, relying on both inductive and coinductive definitions. We demonstrate the feasibility of our proposal using a prototype system and a number of case studies. Our experimental results show that our implementation outperforms the automata-based model checker PAT by 31.7% of the average computation time.

3.1 Introduction

This chapter specifies system behaviors in the form of *DependentEffs*, which integrates the basic and ω-regular expressions with dependent values and arithmetic constraints, gaining the expressive power beyond finite-state machines. Specifically, *DependentEffs* provides insights of: (i) definite finite traces: using symbolic values to present finite repetitions, which can be dependent on program inputs; (ii) definite infinite traces constructed by infinity operator (ω); (iii) possibly finite and possibly infinite traces constructed by Kleene star ($\star$). For example, it expresses, the effects of function *send*(n) as:

$$\Phi^{send(n)} \triangleq (n{\geq}0 \wedge \mathbf{Send}^{n} \cdot \mathbf{Done}) \vee (n{<}0 \wedge \mathbf{Send}^{\omega})$$

The *send* function takes a parameter n, and recursively sends out n messages. The above specification of *send*(n) indicates the fact that for non-negative values of the parameter n, the *send* function generates a finite trace comprising a sequence with n times of event **Send**, followed by a final event **Done**. For the case when the parameter is negative, it generates an infinite trace of event **Send**. Note that (i) the *DependentEffs* can express both finite traces and infinite traces in one single formula, separated by arithmetic constraints, and (ii) n is a parameter to *send*, making the effects *dependent* with respect to the value of *send*'s parameter. Furthermore, by allowing events to be parameterized with symbolic values, the effects are defined as languages over potentially infinite alphabets of the form $\Sigma \times \mathbb{Z}$, where Σ is a finite event set, and $\mathbb{Z}$ is the set of integers. The main contributions are:

1. **Temporal Effects Specification:** This chapter defines the syntax and semantics of *DependentEffs*, going beyond LTL, μ-calculus and prior effects.

2. **Automated Verification System:** Targeting a core language, this chapter develops a Hoare-style forward verifier to accumulate effects from the source code, as the front-end; and a sound decision procedure (the TRS) to solve the effects inclusions, as the back-end.

3. **Implementation and Evaluation:** This chapter prototypes the novel effect logic on top of the HIP/SLEEK system [Chi+12]. It further provides case studies and experimental results to show the feasibility of the proposal.

3.2 Language and Specifications

This section first introduces the target (sequential C-like) language and then depict the temporal specification language which supports *DependentEffs*.

3.2.1 The Target Language.

The syntax of the core imperative language, i.e., sequential C-like, is given in Figure 3.1 Here, k and x are meta-variables. k^τ represents a constant of basic type τ. **var** represents the countably infinite set of arbitrary distinct identifiers. Here, **a** refers to a singleton event coming from the finite set of events Σ. It assumes that programs used are well-typed conforming to basic types τ (it takes () as the *unit* type). A program $\mathcal{P}$ comprises a list of function declarations fun^*. Here, it uses the $*$ superscript to denote a finite list (possibly empty) of items, for example, x^* refers to a list of variables, $x_1, ..., x_n$.

$$
\begin{array}{lrcl}
(\mathit{Program}) & \mathcal{P} & ::= & \mathit{fun}^* \\
(\mathit{Basic\ Types}) & \tau & ::= & \mathit{int} \mid \mathit{bool} \mid \mathit{unit} \\
(\mathit{Function}) & \mathit{fun} & ::= & \tau\ mn\ (\tau\ x)^*\ \{\mathbf{requires}\ \Phi_{pre}\ \mathbf{ensures}\ \Phi_{post}\}\ \{e\} \\
(\mathit{Expressions}) & e & ::= & () \mid k^\tau \mid x \mid \tau\ x;\ e \mid mn(x^*) \mid x{:=}e \mid\ e_1; e_2 \\
 & & & \mid \mathbf{assert}\ \Phi \mid e_1\ op\ e_2 \mid \mathbf{event}[\mathbf{a}] \mid \mathbf{if}\ v\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2
\end{array}
$$

$$k^\tau : constant\ of\ type\ \tau \qquad x, mn ::\in \mathbf{var} \qquad (Events)\mathbf{a} ::\in \Sigma$$

Figure 3.1: A Core Imperative Language.

Each function *fun* has a name *mn*, an expression-oriented body e, also is associated with a precondition Φ_{pre} and a postcondition Φ_{post} (the syntax of effects specification Φ is given in Figure 3.2). The language allows each iterative loop to be optimized to an equivalent tail-recursive function, where the mutation on parameters is made visible to the caller. The technique of translating away iterative loops is standard and is helpful in further minimizing the core language. Expressions comprise unit (), constants k, variables x, local variable declaration $\tau\ x;\ e$, function calls $mn(x^*)$, variable assignments $x{:=}e$, expression sequences $e_1; e_2$, binary operations represented by $e_1\ op\ e_2$, including $+$, $-$, $==$, $<$, etc, event raises expression $\mathbf{event}[\mathbf{a}]$, conditional expressions $\mathbf{if}\ v\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2$, and the assertion constructor **assert**, parameterized with effects Φ.

3.2.2 The Specification Language.

This proposal plants the effects specifications into the Hoare-style verification system. It uses $\{\textbf{requires}\ \Phi_{pre}\ \textbf{ensures}\ \Phi_{post}\}$ to capture the precondition Φ_{pre} and the postcondition Φ_{post}, defined in Figure 3.2.

$$
\begin{array}{rccl}
(\mathit{Effects}) & \Phi & ::= & \pi \wedge es \mid \Phi_1 \vee \Phi_2 \mid \exists x.\Phi \\
(\mathit{Event\ Seq.}) & es & ::= & \bot \mid \epsilon \mid \_ \mid \mathbf{a} \mid \neg\mathbf{a} \mid es_1 \cdot es_2 \mid es_1 \vee es_2 \mid es_1 \wedge es_2 \\
 & & & \mid es^t \mid es^\star \mid es^\omega \\
(\mathit{Pure}) & \pi & ::= & \mathit{True} \mid \mathit{False} \mid A(t_1, t_2) \mid \pi_1 \wedge \pi_2 \mid \pi_1 \vee \pi_2 \\
 & & & \mid \neg\pi \mid \pi_1 \Rightarrow \pi_2 \mid \forall x.\pi \mid \exists x.\pi \\
(\mathit{Terms}) & t & ::= & n \mid x \mid t_1 + t_2 \mid t_1 - t_2
\end{array}
$$

$x ::\in \textbf{var} \qquad n ::\in \mathbb{Z} \qquad (\mathit{Event})\ \mathbf{a} ::\in \Sigma \qquad (\mathit{Infinity})\ \omega \qquad (\mathit{Kleene\ Star})\ \star$

Figure 3.2: Syntax of *DependentEffs*.

Effects can be a conditioned event sequence $\pi \wedge es$ or a disjunction of two effects $\Phi_1 \vee \Phi_2$, or an effect Φ existentially quantified over a variable x. Event sequences comprise *false* ($\bot$); an empty trace ϵ; the wild card _ representing any single event; a single event $\mathbf{a}$; sequences concatenation $es_1 \cdot es_2$; disjunction $es_1 \vee es_2$; conjunction $es_1 \wedge es_2$; negation of a single event $\neg\mathbf{a}$; t times repetition of a trace, es^t, where t is a *term*; Kleene star, zero or many times (possibly infinite) repetition of a trace; and the infinite repetition of a trace, es^ω.

It uses π to donate a pure formula which captures the (Presburger) arithmetic conditions on program parameters. It uses $A(t_1, t_2)$ to represent atomic formulas of two terms (including $=$, $>$, $<$, $\geq$ and $\leq$), A term can be a constant integer value n, an integer variable x which is an input parameter of the program and can be constrained by a pure formula. A term also allows simple computations of terms, t_1+t_2 and t_1-t_2.

3.2.3 Semantic Model of *DependentEffs*.

To define the model, *var* is the set of program variables, *val* is the set of primitive values, *es* is the set of event sequences (or event multi-trees, per se), indicating

$$
\begin{array}{lll}
s, \varphi \models \Phi_1 \vee \Phi_2 & \textit{iff} & s, \varphi \models \Phi_1 \textit{ or } s, \varphi \models \Phi_2 \\
s, \varphi \models \exists x.\Phi & \textit{iff} & (\exists v \in \mathit{val}).s[x{\to}v], \varphi \models \Phi \\
s, \varphi \models \pi \wedge \epsilon & \textit{iff} & [\![\pi]\!]_s{=}\mathit{True} \textit{ and } \varphi{=}[] \\
s, \varphi \models \pi \wedge \_ & \textit{iff} & [\![\pi]\!]_s{=}\mathit{True}, \varphi \in \{[\mathbf{a}] \mid \mathbf{a} \in \Sigma\} \\
s, \varphi \models \pi \wedge \mathbf{a} & \textit{iff} & [\![\pi]\!]_s{=}\mathit{True} \textit{ and } \varphi{=}[\mathbf{a}] \\
s, \varphi \models \pi \wedge \neg\mathbf{a} & \textit{iff} & s, \varphi \not\models \pi \wedge \mathbf{a} \\
s, \varphi \models \pi \wedge (es_1 \cdot es_2) & \textit{iff} & \textit{there exist } \varphi_1, \varphi_2 \textit{ and } \varphi_1 \mathbin{+\!\!+} \varphi_2 = \varphi \\
 & & \quad \textit{and } s, \varphi_1 \models \pi \wedge es_1 \textit{ and } s, \varphi_2 \models \pi \wedge es_2 \\
s, \varphi \models \pi \wedge (es_1 \vee es_2) & \textit{iff} & s, \varphi \models \pi \wedge es_1 \textit{ or } s, \varphi \models \pi \wedge es_2 \\
s, \varphi \models \pi \wedge (es_1 \wedge es_2) & \textit{iff} & s, \varphi \models \pi \wedge es_1 \textit{ and } s, \varphi \models \pi \wedge es_2 \\
s, \varphi \models \pi \wedge es^t & \textit{iff} & [\![\pi \wedge t{=}0]\!]_s{=}\mathit{True},\ s, \varphi \models \pi \wedge \epsilon \textit{ or} \\
 & & [\![\pi \wedge t{>}0]\!]_s{=}\mathit{True},\ \textit{there exist } \varphi_1, \varphi_2 \\
 & & \quad \textit{and } \varphi_1 \mathbin{+\!\!+} \varphi_2{=}\varphi \textit{ and } s, \varphi_1 \models \pi \wedge es \\
 & & \quad \textit{and } s, \varphi_2 \models (\pi \wedge t{>}0) \wedge es^{t\text{-}1} \\
s, \varphi \models \pi \wedge es^\star & \textit{iff} & s, \varphi \models \exists x.(\pi \wedge es^x) \textit{ or } s, \varphi \models \pi \wedge es^\omega \\
s, \varphi \models \pi \wedge es^\omega & \textit{iff} & \textit{there exist } \varphi_1, \varphi_2 \textit{ and } \varphi_1 \mathbin{+\!\!+} \varphi_2{=}\varphi \\
 & & \quad \textit{and } s, \varphi_1 \models \pi \wedge es \textit{ and } s, \varphi_2 \models \pi \wedge es^\omega \\
s, \varphi \models \mathit{false} & \textit{iff} & [\![\pi]\!]_s{=}\mathit{False} \textit{ or } \varphi{=}\bot
\end{array}
$$

Figure 3.3: Semantics of *DependentEffs*.

the sequencing constraints on temporal behavior. Let $s, \varphi \models \Phi$ denote the model relation, i.e., the stack s and linear temporal events φ satisfy the temporal effects Φ, with s, φ from the following concrete domains: $s \triangleq \mathit{var} \to \mathit{val}$ and $\varphi \triangleq es$.

Figure 3.3 defines the semantics of *DependentEffs*. It uses ++ to represent the append operation of two event sequences. It uses [] to represent the empty sequence, [**a**] to represent the sequence only contains one element **a**.

3.3 Automated Forward Verification

The automated verification system consists of a standard Hoare-style forward verifier (the front-end) and a TRS (the back-end). In this section, this section mainly presents the forward verifier, which invokes the back-end, by introducing a set of forward verification rules. Note that it allows the precondition of a function to be false. The body of any such function can always be successfully verified. This relaxation does not affect the soundness of the verification system.

3.3.1 Forward Rules

This section presents some of the forward verification rules in Figure 3.4, which are used to systematically accumulate the effects based on the syntax of each statement. $\mathcal{P}$ is to denote the program being checked. With pre/post conditions declared for each function in $\mathcal{P}$, one can apply modular verification to a function's body using Hoare-style triples $\vdash \{\Phi_C\}\ e\ \{\Phi'_C\}$, where Φ_C is the current effects and Φ'_C is the resulting effects by executing e.

In [*FV-If-Else*], $(v \wedge \Phi_C)$ enforces v into the pure constraints of every traces in Φ_C, same for $(\neg v \wedge \Phi_C)$. The rule *FV-Call* checks whether the instantiated precondition of callee, $[y^*/x^*]\Phi_{pre}$, is satisfied by the *tail* [1] of current effect state, in which it uses an auxiliary function *rev* to reverse the event sequences of effects. Then it obtains the next effect state by concatenating the instantiated postcondition, $[y^*/x^*]\Phi_{post}$, to the current effect state (cf. step 6 in Figure 3.5).

In *FV-Fun*, it initializes the current effect state using ϵ, accumulate the effects from the function body, to obtain Φ_C, and check inclusion between Φ_C and the declared specifications Φ_{post}[2].

[1] It checks the inclusion between the reversed current effects and precondition effects, meaning that, before calling a function, its required effects *has just happened*.

[2] Φ_{post} only needs to capture the effects from the current function body, excluding the history effects specified in Φ_{pre}.

$$\frac{\Phi'_C{=}\Phi_C \cdot \mathbf{a}}{\vdash \{\Phi_C\}\ \mathbf{event}[\mathbf{a}]\ \{\Phi'_C\}}\ [\mathit{FV\text{-}Event}] \quad \frac{\vdash \{\Phi_C\}\ e_1\ \{\Phi'_C\}\ \ \vdash \{\Phi'_C\}\ e_2\ \{\Phi''_C\}}{\vdash \{\Phi_C\}\ e_1; e_2\ \{\Phi''_C\}}\ [\mathit{FV\text{-}Seq}]$$

$$\frac{\vdash \{v \wedge \Phi_C\}\ e_1\ \{\Phi'_C\} \qquad \vdash \{\neg v \wedge \Phi_C\}\ e_2\ \{\Phi''_C\}}{\vdash \{\Phi_C\}\ \mathbf{if}\ v\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2\ \{\Phi'_C \vee \Phi''_C\}}\ [\mathit{FV\text{-}If\text{-}Else}]$$

$$\frac{\vdash \{\Phi_C\}\ e\ \{\Phi'_C\}}{\vdash \{\Phi_C\}\ \tau\ x;\ e\ \{\exists x.\Phi'_C\}}\ [\mathit{FV\text{-}Local}] \qquad \frac{\vdash \mathit{rev}(\Phi_C) \sqsubseteq \mathit{rev}(\Phi) \rightsquigarrow \gamma_{\mathrm{R}}}{\vdash \{\Phi_C\}\ \mathbf{assert}\ \Phi\ \{\Phi_C\}}\ [\mathit{FV\text{-}Assert}]$$

$$\tau\ mn\ (\tau\ x)^*\ \{\mathbf{requires}\ \Phi_{pre}\ \mathbf{ensures}\ \Phi_{post}\}\ \{e\} \in \mathcal{P}$$

$$\frac{\vdash \mathit{rev}(\Phi_C) \sqsubseteq \mathit{rev}([y^*/x^*]\Phi_{pre}) \rightsquigarrow \gamma_{\mathrm{R}} \qquad \Phi'_C = \Phi_C \cdot [y^*/x^*]\Phi_{post}}{\vdash \{\Phi_C\}\ mn(y^*)\ \{\Phi'_C\}}\ [\mathit{FV\text{-}Call}]$$

$$\frac{\vdash \{\epsilon\}\ e\ \{\Phi_C\} \qquad\qquad \vdash \Phi_C \sqsubseteq \Phi_{post}}{\vdash \tau\ mn\ (\tau\ x)^*\ \{\mathbf{requires}\ \Phi_{pre}\ \mathbf{ensures}\ \Phi_{post}\}\ \{e\}}\ [\mathit{FV\text{-}Fun}]$$

$$\frac{\vdash \{\Phi'_{pre}\}\ e\ \{\Phi'_{post}\} \qquad \vdash \Phi_{pre} \sqsubseteq \Phi'_{pre} \qquad \vdash \Phi'_{post} \sqsubseteq \Phi_{post}}{\vdash \{\Phi_{pre}\}\ e\ \{\Phi_{post}\}}\ [\mathit{FV\text{-}Frame}]$$

Figure 3.4: Selected Forward Rules for *DependentEffs*

3.4 Effects Inclusion Checker

The effects inclusion checking (an extension of the TRS proposed from [AM95]) will be triggered i) right before a function call, to check the satisfiability of the precondition; ii) after the forward verification, to check the satisfiability of the postcondition; and iii) when there is an assertion, to check the satisfiability of the asserted effects. As shown in section 3.3, the forward verification generates effects inclusions of the form: $\Gamma \vdash \Phi_1 \sqsubseteq^{\Phi}_{V} \Phi_2 \rightsquigarrow \gamma_{\mathrm{R}}$, a shorthand for: $\Gamma \vdash \Phi \cdot \Phi_1 \sqsubseteq \exists V.\ (\Phi \cdot \Phi_2) \rightsquigarrow \gamma_{\mathrm{R}}$.

To prove such effects inclusions is to check whether all the possible event traces in the antecedent Φ_1 are legitimately allowed in the possible event traces from the consequent Φ_2, and (in case there are) to compute a residual effects γ_{R} (also known as "frame" in the frame inference [Cal+09]) , which represents what was not consumed from the antecedent after matching up with the effects from the consequent. Γ is the proof context, i.e. a set of effects inclusions, Φ is the history of effects from the antecedent that have been used to match the effects from the consequent, and V is the set of existentially quantified variables from the consequent. Note that Γ, Φ and V are derived during the inclusion proof. The inclusion checking procedure is initially invoked with $\Gamma{=}\emptyset$, $\Phi{=}\mathit{True} \wedge \epsilon$ and $V{=}\emptyset$. Here briefly discusses the key steps and related forward reasoning rules that it may use in such an effects inclusion proof. Firstly, it presents the reduction to eliminate the disjunctions from the antecedents and existential quantifiers.

3.4.1 Effect Disjunction.

An inclusion with a disjunctive antecedent succeeds if both disjunctions entail the consequent. *(LHS refers to left-hand side, and RHS refers to right-hand side.)*

$$\frac{\Gamma \vdash \Phi_1 \sqsubseteq \Phi \rightsquigarrow {\gamma_{\mathrm{R}}}^1 \qquad \Gamma \vdash \Phi_2 \sqsubseteq \Phi \rightsquigarrow {\gamma_{\mathrm{R}}}^2}{\Gamma \vdash \Phi_1 \vee \Phi_2 \sqsubseteq \Phi \rightsquigarrow ({\gamma_{\mathrm{R}}}^1 \vee {\gamma_{\mathrm{R}}}^2)} \quad [\mathit{LHS\text{-}OR}]$$

3.4.2 Existential Quantifiers.

Existentially quantified variables from the antecedent are simply lifted out of the inclusion relation by replacing them with fresh variables. On the other hand, it keeps track of the existential variables coming from the consequent by adding them to V. *(u is a fresh variable)*

$$\frac{\Gamma \vdash [u/x]\Phi_1 \sqsubseteq^{\Phi}_{V} \Phi_2 \rightsquigarrow \gamma_{\mathrm{R}}}{\Gamma \vdash \exists x.\ \Phi_1 \sqsubseteq^{\Phi}_{V} \Phi_2 \rightsquigarrow \gamma_{\mathrm{R}}} \ [\mathit{LHS\text{-}EX}] \qquad \frac{\Gamma \vdash \Phi_1 \sqsubseteq^{\Phi}_{V \cup \{u\}} ([u/x]\Phi_2) \rightsquigarrow \gamma_{\mathrm{R}}}{\Gamma \vdash \Phi_1 \sqsubseteq^{\Phi}_{V} (\exists x.\ \Phi_2) \rightsquigarrow \gamma_{\mathrm{R}}} \ [\mathit{RHS\text{-}EX}]$$

3.4.3 Normalization.

The rewriting of an inclusion between two quantifier-free effects starts with a general normalization for both the antecedent and the consequent.

It is assumed that the effects formulae are tailored accordingly using the lemmas in Table 3.1, which are extended from the normalization rules suggested by Antimirov and Mosses, being able to further normalize *DependentEffs*.

Table 3.1: Some Normalization Lemmas for *DependentEffs*.

$es \vee es \to es$	$\epsilon^\omega \to \epsilon$	$(es_1 \cdot es_2) \cdot es_3 \to es_1 \cdot (es_2 \cdot es_3)$
$\bot \vee es \to es$	$es \wedge es \to es$	$(es_1 \vee es_2) \cdot es_3 \to es_1 \cdot es_3 \vee es_2 \cdot es_3$
$es \vee \bot \to es$	$es \wedge \bot \to \bot$	$es_1 \cdot (es_2 \vee es_3) \to es_1 \cdot es_2 \vee es_1 \cdot es_3$
$\epsilon \cdot es \to es$	$\bot^\omega \to \bot$	$es^\omega \cdot es_1 \to es^\omega$
$es \cdot \epsilon \to es$	$\epsilon^t \to \epsilon$	$\mathit{False} \wedge es \to \mathit{False} \wedge \bot$
$\bot \cdot es \to \bot$	$t{=}0 \wedge es^t \to \epsilon$	$es \wedge \epsilon \to \bot \quad (\delta_\pi(es){=}\mathit{false})$
$es \cdot \bot \to \bot$	$\bot^t \to \bot$	$es \wedge \epsilon \to \epsilon \quad (\delta_\pi(es){=}\mathit{true})$

3.4.4 Substitution.

In order to guarantee the termination, for both the antecedent and the consequent, a term $t_1{\oplus}t_2$ will be substituted with a fresh variable u constrained with $u{=}t_1 \oplus t_2 \wedge u{\geq}0$, where $\oplus{\in}\{+,-\}$. (cf. Table 3.3-II)

$$\frac{\pi'{=}(u{=}t_1 \oplus t_2 \wedge u{\geq}0) \quad \Gamma \vdash (\pi_1 \wedge \pi'){\wedge}es_1^u \cdot es \sqsubseteq (\pi_2{\wedge}\pi') \wedge es_2 \rightsquigarrow \gamma_{\mathrm{R}}}{\Gamma \vdash \pi_1 \wedge (es_1{}^{t_1\oplus t_2} \cdot es) \sqsubseteq \pi_2 \wedge es_2 \rightsquigarrow \gamma_{\mathrm{R}}}\ [\mathit{LHS\text{-}SUB}]$$

$$\frac{\pi'{=}(u{=}t_1 \oplus t_2 \wedge u{\geq}0) \quad \Gamma \vdash (\pi_1 \wedge \pi'){\wedge}es_1 \sqsubseteq (\pi_2{\wedge}\pi') \wedge es_2^u \cdot es \rightsquigarrow \gamma_{\mathrm{R}}}{\Gamma \vdash \pi_1 \wedge es_1 \sqsubseteq \pi_2 \wedge (es_2{}^{t_1\oplus t_2} \cdot es) \rightsquigarrow \gamma_{\mathrm{R}}}\ [\mathit{RHS\text{-}SUB}]$$

3.4.5 Case Split.

Based on the semantics of the symbolic integer t, whenever it is possibly zero, it conducts a case split, to distinguish the zero (base) case, leads to an empty trace; and the non-zero (inductive) case. (cf. Table 3.3-II)

$$[\mathit{LHS\text{-}CaseSplit}]$$
$$\frac{\Gamma \vdash ((\pi_1 \wedge t{=}0) \wedge es) \vee ((\pi_1 \wedge t > 0) \wedge es_1 \cdot es_1^{t\text{-}1} \cdot es) \sqsubseteq \pi_2 \wedge es_2 \rightsquigarrow \gamma_{\mathrm{R}}}{\Gamma \vdash \pi_1 \wedge (es_1^t \cdot es) \sqsubseteq \pi_2 \wedge es_2 \rightsquigarrow \gamma_{\mathrm{R}}}$$

$$[\mathit{RHS\text{-}CaseSplit}]$$
$$\frac{\Gamma \vdash \pi_1 \wedge es_1 \sqsubseteq ((\pi_2 \wedge t{=}0) \wedge es) \vee ((\pi_2 \wedge t > 0) \wedge es_2 \cdot es_2^{t\text{-}1} \cdot es) \rightsquigarrow \gamma_{\mathrm{R}}}{\Gamma \vdash \pi_1 \wedge es_1 \sqsubseteq \pi_2 \wedge (es_2^t \cdot es) \rightsquigarrow \gamma_{\mathrm{R}}}$$

3.4.6 Unfolding (Induction).

Here comes the key inductive step of unfolding the inclusion. Firstly, it makes use of the *fst* auxiliary function to get a set of events F, which are all the possibly *first* event from the antecedent. Secondly, it obtains a new proof context Γ' by adding the current inclusion, as an inductive hypothesis, into the current proof context Γ. Thirdly, it iterates each element **a** (**a**$\in F$), and compute the partial derivatives (the *next-state* effects) of both the antecedent and consequent with respect to **a**. The proof of the original inclusion succeeds if all the derivative inclusions succeeds.

$$\frac{F = fst_{\pi_1}(es_1) \qquad \Gamma' = \Gamma,\ (\pi_1 \wedge es_1 \sqsubseteq \pi_2 \wedge es_2) \qquad \forall \mathbf{a} \in F.\ (\Gamma' \vdash\ D^{\pi_1}_{\mathbf{a}}(es_1) \sqsubseteq D^{\pi_2}_{\mathbf{a}}(es_2))}{\Gamma \vdash \pi_1 \wedge es_1 \sqsubseteq \pi_2 \wedge es_2}\ [Unfold]$$

Next, it provides the definitions and the key implementations[3] of Nullable, First and Derivative respectively. Intuitively, the Nullable function $\delta_\pi(es)$ returns a boolean value indicating whether $\pi \wedge es$ contains the empty trace; the First function $fst_\pi(es)$ computes a set of possible initial events of $\pi \wedge es$; and the Derivative function $D^{\pi}_{\mathbf{a}}(es)$ computes a next-state effects after eliminating one event **a** from the current effects $\pi \wedge es$.

Definition 3 (Nullable)**.** Given any event sequence es under condition π, here defines $\delta_\pi(es)$ to be:

$$\delta_\pi(es) : bool = \begin{cases} true & if\ \epsilon \in [\![\pi \wedge es_1]\!]_\varphi \\ false & if\ \epsilon \notin [\![\pi \wedge es_1]\!]_\varphi \end{cases},\ \text{where } \delta_\pi(es^t) = SMT(\pi \wedge (t{=}0))^4$$

Definition 4 (First)**.** Let $fst_\pi(es){:=}\{\mathbf{a} \mid \mathbf{a}\cdot es' \in [\![\pi \wedge es]\!]\}$ be the set of initial events derivable from event sequence es with respect to the condition π.

$$fst_\pi(es_1 \cdot es_2) = \begin{cases} fst_\pi(es_1) \cup fst_\pi(es_2) & if\ \ \delta_\pi(es_1){=}true \\ fst_\pi(es_1) & if\ \ \delta_\pi(es_1){=}false \end{cases}$$

---

[3] As the implementations according to basic regular expressions can be found in prior work [KT14a]. Here, it focuses on presenting the definitions and how does it deal with dependent values in the effects, as the key novelties of this work.

[4] The proof obligations are discharged using the Z3 SMT prover, while deciding the nullability of effects constructed by symbolic terms, represented by $SMT(\pi)$.

Definition 5 (Derivative)**.** The derivative $D^{\pi}_{\mathbf{a}}(es)$ of an event sequence es with respect to an event $\mathbf{a}$ and the condition π computes the effects for the left quotient $\mathbf{a}^{-1}[\![\pi \wedge es]\!]$, where it defines $D^{\pi}_{\mathbf{a}}(es^{t}) = D^{\pi\wedge t>0}_{\mathbf{a}}(es) \cdot es^{t\text{-}1}$.

$$D^{\pi}_{\mathbf{a}}(es_1 \cdot es_2) = \begin{cases} D^{\pi}_{\mathbf{a}}(es_1) \cdot es_2 \vee D^{\pi}_{\mathbf{a}}(es_2) & \textit{if } \delta_{\pi}(es_1){=}\textit{true} \\ D^{\pi}_{\mathbf{a}}(es_1) \cdot es_2 & \textit{if } \delta_{\pi}(es_1){=}\textit{false} \end{cases}$$

3.4.7 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.

$$\frac{\delta_{\pi_1}(es_1) \wedge \neg\delta_{\pi_1\wedge\pi_2}(es_2)}{\Gamma \vdash \pi_1 \wedge es_1 \not\sqsubseteq \pi_2 \wedge es_2} \quad [\textit{Disprove}]$$

3.4.8 Prove.

It uses three rules to prove an inclusion: (i) [Prove] is used when there is a *subset* relation $\subseteq$ between the antecedent and consequent; (ii) [Frame] is used when the consequent is empty, it proves this inclusion with a residue γ_{R}[5];

and (iii) [Reoccur] is used when there exists an inclusion hypothesis in the proof context Γ, which meets the conditions. It essentially assigns to the current unexpanded inclusion an interior inclusion with an identical sequent labelling.

$$\frac{\pi_1 \Rightarrow \pi_2 \quad es_1 \subseteq es_2}{\Gamma \vdash \pi_1 \wedge es_1 \sqsubseteq \pi_2 \wedge es_2} \quad [\textit{Prove}] \qquad \frac{\pi_1 \Rightarrow \pi_2 \quad \gamma_{\mathrm{R}}{=}\pi_1 \wedge es_1}{\Gamma \vdash (\pi_1{\wedge}es_1 \sqsubseteq \pi_2{\wedge}\epsilon) \rightsquigarrow \gamma_{\mathrm{R}}} \quad [\textit{Frame}]$$

$$\frac{\exists.(\pi_1' \wedge es_1' \sqsubseteq \pi_2' \wedge es_2') \in \Gamma \qquad \pi_1{\Rightarrow}\pi_1'{\Rightarrow}\pi_2'{\Rightarrow}\pi_2 \quad es_1{\subseteq}es_1' \quad es_2'{\subseteq}es_2}{\Gamma \vdash \pi_1 \wedge es_1 \sqsubseteq \pi_2 \wedge es_2} [\textit{Reoccur}]$$

[5] A residue refers to the remaining event sequences from antecedent after matching up with the consequent. An inclusion with no residue means the antecedent completely/exactly matches with the consequent.

3.4.9 Discussion: highlighting the novelty.

Departing from the original Antimirov algorithm [AM95], this work devises more rewriting rules such as: [RHS-SUB], [LHS-SUB], [RHS-CaseSplit], [LHS-CaseSplit], [Frame]. These rules are necessary to prove the inclusions between the more expressive specifications formulae, *DependentEffs*, which integrate arithmetic constraints with the symbolic traces. The comprehensive rewriting system serves as a back-end engine for the finer-grained verification system, which cannot be trivially achieved by the original rewriting system.

3.5 Demonstration Examples

Here is a summary of the techniques, using the example shown in Table 3.2-(a). The *DependentEffs* can be illustrated with *send* and *server*, which simulate a server who continuously sends messages to all its clients. This function *server* takes an integer parameter n, triggers an event **Ready**, then calls the function *send*, making a boolean choice depending on input n: in one case it triggers an event **Done**; otherwise it triggers an event **Send**, then makes a recursive call with parameter n-1. Finally *server* recurs.

Table 3.2: Overview Example for *DependentEffs*.

<table>
<tr><th>(a) Source Code</th><th>(b) DependentEffs Specifications</th></tr>
<tr><td>

```
void send (int n){
   if (n==0) {
      event["Done"];
   }else{
      event["Send"];
      send (n-1);
   }}

void server (int n){
   event["Ready"];
   send(n);
   server(n);}
```

</td><td>

$\Phi_{pre}^{send(n)} \triangleq \mathit{True} \wedge \mathbf{Ready} \cdot (\_)^{\star}$

$\Phi_{post}^{send(n)} \triangleq (n{\geq}0 \wedge \mathbf{Send}^{n} \cdot \mathbf{Done}) \vee (n{<}0 \wedge \mathbf{Send}^{\omega})$

$\Phi_{pre}^{server(n)} \triangleq n{\geq}0 \wedge \epsilon$

$\Phi_{post}^{server(n)} \triangleq n{\geq}0 \wedge (\mathbf{Ready} \cdot \mathbf{Send}^{n} \cdot \mathbf{Done})^{\omega}$

</td></tr>
</table>

3.5.1 *DependentEffs*

The effects specifications for *server* and *send* are given in Table 3.2-(b). It defines Hoare-triple style specifications for each of the programs, which lead to a more compositional verification strategy, where temporal reasoning can be done locally. Function *send*'s precondition, denoted by $\Phi_{pre}^{send(n)}$, requires the event **Ready** to have happened at some point of the effects history; and it guarantees the final effects/postcondition, denoted by $\Phi_{post}^{send(n)}$.

Function *server*'s precondition, $\Phi_{pre}^{server(n)}$, requires the input value be non-negative while the pre-trace is required to be empty (ϵ); its postcondition ensures the final effects $\Phi_{post}^{server(n)}$ – an infinite repetition of a trace consisting of an event **Ready** followed by n times of **Send** followed by **Done**. Directly from the specifications, they are aware of (i) termination properties: *server must* not terminate, while *send may* not terminate; (ii) branching properties: different arithmetic conditions on the input parameters lead to different temporal effects; and (iii) required history traces: by defining the prior effects in the precondition. The examples already show that *DependentEffs* provides more detail information than classical LTL or μ-calculus, and in fact, it cannot be fully captured by any prior works [HC14; KT14b; Mur+16; Nan+18]. Nevertheless, the gain in expressive power comes at the efforts of a more dedicated verification process, namely handled by the TRS.

3.5.2 Forward Verification.

As shown in Figure 3.5, it demonstrates the forward verification process of function *send*. The current effect states of a program is captured in the form of $\{\Phi_C\}$. To facilitate the illustration, it labels the verification steps by 1), ..., 8). The figure marks the deployed verification rules in gray. The verifier invokes the TRS to check language inclusions along the way.

The effect state 1) is obtained by initializing Φ_C from the precondition.The effect states 2), 4) and 7) are obtained by [*FV-If-Else*], which adds the constraints from the conditionals into the current effects state, and unions the effects accumulated from two branches in the end. The effect states 3) and 5) are obtained by [*FV-Event*], which simply concatenates the triggered singleton event to the end of the current

1) *void send* (*int n*){ *(– initialize the current effect state –)*
$\{\Phi_C{=}\Phi^{send(n)}_{pre}{=}True \wedge \textbf{Ready} \cdot \_^{\star}\}$ [*FV-Fun*]

2) *if* ($n{==}0$){
$\{n{=}0 \wedge \textbf{Ready} \cdot \_^{\star}\}$ [*FV-If-Else*]

3) *event*[*Done*]; }
$\{n{=}0 \wedge \textbf{Ready} \cdot \_^{\star} \cdot \textbf{Done}\}$ [*FV-Event*]

4) *else*{
$\{n{\neq}0\}$ [*FV-If-Else*]

5) *event*[*Send*];
$\{n{\neq}0 \wedge \textbf{Ready} \cdot \_^{\star} \cdot \textbf{Send}\}$ [*FV-Event*]

6) *send*(n-1); }}
$rev(n{\neq}0 \wedge \textbf{Ready} \cdot \_^{\star} \cdot \textbf{Send}) \sqsubseteq rev(\Phi^{send(n\text{-}1)}_{pre})$ *(-check precondition-)*
$\{n{\neq}0 \wedge \textbf{Ready} \cdot \_^{\star} \cdot \textbf{Send} \cdot \Phi^{send(n\text{-}1)}_{post}\}$ [*FV-Call*]

7) $\Phi'_C{=}(n{=}0 \wedge \textbf{Ready} \cdot \_^{\star} \cdot \textbf{Done}) \vee (n{\neq}0 \wedge \textbf{Ready} \cdot \_^{\star} \cdot \textbf{Send} \cdot \Phi^{send(n\text{-}1)}_{post})$

8) $\Phi'_C \sqsubseteq \Phi^{send(n)}_{pre} \cdot \Phi^{send(n)}_{post} \Leftrightarrow$ *(– check postcondition –)*
$(n{=}0 \wedge \textbf{Done}) \vee (n{\neq}0 \wedge \textbf{Send} \cdot \Phi^{send(n\text{-}1)}_{post}) \sqsubseteq \Phi^{send(n)}_{post}$

Figure 3.5: The forward verification example for the function *send*.

effect state. The effect state 6) is obtained by [*FV-Call*]. Before each function call, it checks whether the current state satisfies the precondition of the callee function. The *rev* function simply reverses the order of effects sequences. If the precondition is not satisfied, then the verification fails, otherwise it concatenates the postcondition of the callee to the current effects.

While Hoare logics based on finite traces (terminating runs) [MMW11] and infinite traces (non-terminating runs) [NU15] have been considered before, the reasoning on properties of mixed definitions is new. Prior effects in the precondition is also new, allowing greater safety to be applied to sequential reactive controlling systems such as web applications, communication protocols and IoT systems.

3.5.3 The TRS

Table 3.3 demonstrates the inclusion checking example on the postcondition of function *send*. I : The main rewriting proof tree (coming from the step 8) in

Figure 3.5); II : One sub-tree of the rewriting process.

Table 3.3: One inclusion checking example for *DependentEffs*.

I :

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{(n{=}0)\wedge\epsilon \sqsubseteq \epsilon \quad [\text{Frame}]}{(n{=}0) \wedge \mathbf{Done} \sqsubseteq \mathbf{Done}}
}{(n{=}0) \wedge \mathbf{Done} \sqsubseteq \mathbf{Send}^0 \cdot \mathbf{Done}}
}{(n{=}0) \wedge \mathbf{Done} \sqsubseteq \Phi_{post}^{send(n)}}
\qquad (II) \qquad
\dfrac{
\dfrac{
\dfrac{n{<}0 \wedge \mathbf{Send}^{\omega} \sqsubseteq \mathbf{Send}^{\omega}\ (\dagger)[\text{Reoccur}]}{n{<}0 \wedge \mathbf{Send}^{\omega} \sqsubseteq \mathbf{Send}^{\omega}\ (\dagger)}\,[\text{Unfold}]
}{n{<}0 \wedge \mathbf{Send} \cdot \mathbf{Send}^{\omega} \sqsubseteq \mathbf{Send}^{\omega}}\,[\text{Unfold}]
}{n{<}0 \wedge \mathbf{Send} \cdot \mathbf{Send}^{\omega} \sqsubseteq \Phi_{post}^{send(n)}}
}{(n{=}0 \wedge \mathbf{Done}) \vee (n{\neq}0 \wedge \mathbf{Send} \cdot \Phi_{post}^{send(n\text{-}1)}) \sqsubseteq \Phi_{post}^{send(n)}}\,[\text{DISJUNCTION}]
$$

II :

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{n_1{=}0\wedge\epsilon \sqsubseteq \epsilon \quad [\text{Frame}]}{n_1{=}0 \wedge \mathbf{Done} \sqsubseteq \mathbf{Done}}
\qquad
\dfrac{
\dfrac{(n_2{=}n_1\text{-}1\wedge n_2{\geq}0)\wedge\mathbf{Send}^{n_2} \cdot \mathbf{Done} \sqsubseteq \mathbf{Send}^{n_2} \cdot \mathbf{Done}\ (\ddagger)[\text{Reoccur}]}{n_1{>}0 \wedge \mathbf{Send}^{n_1\text{-}1} \cdot \mathbf{Done} \sqsubseteq \mathbf{Send}^{n_1\text{-}1} \cdot \mathbf{Done}}
}{n_1{>}0 \wedge \mathbf{Send}^{n_1} \cdot \mathbf{Done} \sqsubseteq \mathbf{Send}^{n_1} \cdot \mathbf{Done}}\,[\text{Unfold}]
}{(n_1{=}n\text{-}1 \wedge n_1{\geq}0) \wedge \mathbf{Send}^{n_1} \cdot \mathbf{Done} \sqsubseteq \mathbf{Send}^{n_1} \cdot \mathbf{Done}\ (\ddagger)}\,[\text{CaseSplit}]
}{n{>}0 \wedge \mathbf{Send}^{n\text{-}1} \cdot \mathbf{Done} \sqsubseteq \mathbf{Send}^{n\text{-}1} \cdot \mathbf{Done}}\,[\text{SUBSTITUTE}]
}{n{>}0 \wedge \mathbf{Send} \cdot \mathbf{Send}^{n\text{-}1} \cdot \mathbf{Done} \sqsubseteq \mathbf{Send}^{n} \cdot \mathbf{Done}}\,[\text{Unfold}]
}{n{>}0 \wedge \mathbf{Send} \cdot \mathbf{Send}^{n\text{-}1} \cdot \mathbf{Done} \sqsubseteq \Phi_{post}^{send(n)}}
$$

The TRS is designed to check the inclusion between any two *DependentEffs*. We present the rewriting process on the postcondition checking of the function *send*. It marks the rules of some essential forward reasoning steps in green. The effects rewriting system decides effects inclusion through an iterated process of checking the inclusion of their partial derivatives. There are two important rules inherited from Antimirov and Mosses's algorithm: [Disprove], which infers false from a trivially inconsistent inclusion; and [Unfold], which applies Theorem 1 to generate new inclusions. Similarly, we present Definition 6 for unfolding the inclusions between *DependentEffs*.

Definition 6 (*DependentEffs* Inclusion)**.** Given Σ is a finite set of alphabet, for two *DependentEffs* Φ_1 and Φ_2, their inclusion is defined as:

$$\Phi_1 \sqsubseteq \Phi_2 \Leftrightarrow (\forall \mathbf{A} \in \Sigma).\ \mathbf{A}^{-1}(\Phi_1) \sqsubseteq \mathbf{A}^{-1}(\Phi_2).$$

Besides, *DependentEffs* use symbolic values (assuming non-negative) to capture the finite traces, depended on program inputs. Whenever the symbolic value is possibly zero, it uses the rule [CaseSplit] to distinguish the zero (base) and non-zero (inductive) cases, as shown in Table 3.3-II. In addition, the TRS is obligated to reason about mixed inductive (finite) and coinductive (infinite) definitions. It achieves these features and still guarantee the termination by using rules: [SUBSTITUTE], which renames the symbolic terms using free variables; and [Reoccur], which finds the syntactic identity, as a *companion*, of the current open goal, as a *bud*, from the internal proof tree [Bro05a]. (It uses (†) and (‡) in Table 3.3 to indicate the pairing of buds with companions.)

3.6 Implementation and Evaluation

To show the feasibility of the proposal, the implemented prototype system is in OCaml, on top of the HIP/SLEEK system [Chi+12]. The arithmetic proof obligations generated by the verification are discharged using constraint solver Z3. Next, we show case studies to demonstrate the expressive power of *DependentEffs*.

3.6.1 Case Studies.

I. Encoding LTL. Classical LTL extended propositional logic with the temporal operators $\mathcal{G}$ ("globally") and $\mathcal{F}$ ("in the future"), which it also writes $\Box$ and $\Diamond$, respectively; and introduced the concept of fairness, which ensures an infinite-paths semantics. LTL was subsequently extended to include the $\mathcal{U}$ ("until") operator and the $\mathcal{X}$ ("next time") operator. As shown in Table 3.4, it encodes these basic operators into the effects, making it more intuitive and readable, mainly when nested operators occur. Furthermore, by putting the effects in the precondition, *DependentEffs* naturally combines *past-time LTL* along the way. ($\mathbf{A}, \mathbf{B}$ are events, $n \geq 0$, $m \geq 0$ are the default constraints.)

Table 3.4: Examples for converting LTL into *DependentEffs*.

$\Box\mathbf{A}\equiv\mathbf{A}^\star$	$\Diamond\mathbf{A}\equiv\_^n\cdot\mathbf{A}$	$\mathbf{A}\ \mathcal{U}\ \mathbf{B}\equiv\mathbf{A}^n\cdot\mathbf{B}$	$\Box\Diamond\mathbf{A}\equiv\_^n\cdot\mathbf{A}\cdot(\_^m\cdot\mathbf{A})^\star$
$\mathcal{X}\mathbf{A}\equiv\_\cdot\mathbf{A}$	$\mathbf{A}\rightarrow\Diamond\mathbf{B}\equiv\neg\mathbf{A}\vee\_^n\cdot\mathbf{B}$	$\Diamond\Box\mathbf{A}\equiv\_^n\cdot\mathbf{A}^\star$	$\Diamond\mathbf{A}\vee\Diamond\mathbf{B}\equiv\_^n\cdot\mathbf{A}\vee\_^m\cdot\mathbf{B}$

II. Encoding μ-calculus. μ-calculus provides a single, elegant, uniform logical framework of great raw expressive power by using a least fixpoint (μ) and a greatest fixpoint (v). More specifically, it can express properties such as $vZ.P\wedge\mathcal{X}\mathcal{X}Z$, which says that there exists a path where the atomic proposition P holds at every even position, and any valuation can be used on odd positions. As one can see, such properties already go beyond the first order logic. In fact, analogously to *DependentEffs*, the symbolic/constant values correspond to the least fixpoint (μ), referring to finite traces, and the constructor ω corresponds to the greatest fixpoint (v), referring to infinite traces. For example, one can write $(\_\cdot\mathbf{A})^\omega$, meaning that the event **A** recurs at every even position in an infinite trace.

III. Kleene Star. By using $\star$, it makes an approximation of the possible traces when the termination is non-deterministic. As shown in Figure 3.6, a weaker specification of $send(n)$ can be provided as $\mathbf{Send}^\star\cdot\mathbf{Done}$, meaning that the repetition of event **Send** can be both finite and infinite, which is more concise than the prior work, also beyond μ-calculus.

```
void send (int n){
   if (...){ event[Done];}
   else{ event[Send];
         send(n-1);}}
```

Figure 3.6: An unknown conditional.

By supporting a variety of specifications, it can make a trade-off between precision and scalability, which is important for realistic methodology on automated verification. For example, it can weaken precondition of $server(n)$ (cf. Table 3.2) to $\Phi^{server(n)}_{pre}\triangleq True\wedge\epsilon$, and opt for either of the following two postcondition:

$\Phi^{server(n)}_{post1} \triangleq n{\geq}0{\wedge}(\textbf{Ready} \cdot \textbf{Send}^n \cdot \textbf{Done})^\omega$, or $\Phi^{server(n)}_{post2} \triangleq n{\geq}0{\wedge}(\textbf{Ready} \cdot \textbf{Send}^n \cdot \textbf{Done})^\omega \vee n{<}0{\wedge}\textbf{Ready}{\cdot}\textbf{Send}^\omega$, with the latter being more complex but more precise.

IV. Beyond Regular, Context-Free and Context-Sensitive. The paradigmatic non-regular linear language: $n{>}0 \wedge \textbf{a}^n \cdot \textbf{b}^n$, can be naturally expressed by the depended effects. Besides, the effects can also express grammars such as $n{>}0 \wedge \textbf{a}^n \cdot \textbf{b}^n \cdot \textbf{c}^n$, or $n{>}0 \wedge m{>}0 \wedge \textbf{a}^n \cdot \textbf{b}^m \cdot \textbf{c}^n$, which are beyond context-free grammar. Those examples show that the traces which cannot be recognized even by push-down automata (PDA) can be represented by *DependentEffs*.

However, such specifications are significant, suppose it has a traffic light control system, it could have a specifications $n{>}0 \wedge m{>}0 \wedge (\textbf{Red}^n \cdot \textbf{Yellow}^m \cdot \textbf{Green}^n)^\omega$, which specifies that (i) this is a continuous-time system which has an infinite trace, (ii) all the colors will occur at each life circle, and (iii) the duration of the green light and the red light is always the same. Moreover, these effects can not be translated into linear bounded automata (LBA) either, which equivalents to context-sensitive grammar, as LBA are only capable of expressing finite traces.

3.6.2 Experimental Results.

This experiment compares the performance of the backend TRS against the well-established model checker PAT [Sun+09], which is is taken as the baseline and implements techniques for LTL properties with fairness assumptions. The comparison chose a realistic benchmark containing 16 IOT programs implemented in C for Arduino controlling programs [Ard22]. For each of the programs, it (i) derives a number of temporal properties (for 16 distinct execution models, there are in total 235 properties with 124 valid and 111 invalid), (ii) express these properties using both LTL formulae and *DependentEffs*, (iii) it records the total computation time using PAT and the TRS. The test cases are provided as a benchmark. The experiments are conduct on a MacBook Pro with a 2.6 GHz Intel Core i7 processor.

As shown in Table 3.5, it records the lines of code (LOC), the number of testing temporal properties (#Prop.), and the proving/disproving times (in milliseconds) using PAT and the TRS respectively. The table compares the TRS with PAT, and

Table 3.5: The experiments are based on 16 real world C programs.

	Programs	**LOC**	**#Prop.**	**PAT(ms)**	**TRS(ms)**
1	Chrome_Dino_Game	80	12	32.09	7.66
2	Cradle_with_Joystick	89	12	31.22	9.85
3	Small_Linear_Actuator	180	12	21.65	38.68
4	Large_Linear_Actuator	155	12	17.41	14.66
5	Train_Detect	78	12	19.50	17.35
6	Motor_Control	216	15	22.89	4.71
7	Train_Demo_2	133	15	49.51	59.28
8	Fridge_Timer	292	15	17.05	9.11
9	Match_the_Light	143	15	23.34	49.65
10	Tank_Control	104	15	24.96	19.39
11	Control_a_Solenoid	120	18	36.26	19.85
12	IoT_Stepper_Motor	145	18	27.75	6.74
13	Aquariumatic_Manager	135	10	25.72	3.93
14	Auto_Train_Control	122	18	56.55	14.95
15	LED_Switch_Array	280	18	44.78	19.58
16	Washing_Machine	419	18	33.69	9.94
Total		2546	235	446.88	305.33

the total proving/disproving time has been reduced by 31.7%.

We summarize the insights which lead to the improvement: (1) when the transition states of the models are small, the average execution time spent by the TRS is even less than the NFA construction time, which means it is not necessary to construct the NFA when a TRS solves it faster; (2) when the total states become larger, on average, the TRS outperforms automata-based algorithms, due to the significantly reduced search branches provided by the normalization lemmas; and (3) for the invalid cases, the TRS disproves them earlier without constructing the whole NFA.

3.7 Summary

This proposal devises a concise and precise characterization of temporal properties. It proposes a novel logic for effects to specify and verify the implementation of the possibly non-terminating programs, including the use of prior effects in precondition.

It implements the effect logic on top of the HIP/SLEEK system [Chi+12] and show its feasibility. This work is the first solution that automate modular temporal verification using an expressive effect logic, which primarily benefits modern sequential controlling systems ranging over a variety of application domains.

Chapter 4

(A)Synchronous Effects (*ASyncEffs*)

To make reactive programming more concise and expressive, it is promising to combine two approaches to concurrency that integrates *synchronous preemptions* with *asynchronous promises*. Existing automated temporal verification techniques have not been designed to handle such a marriage of two execution models. This work presents a solution that integrates a modular Hoare-style forward verifier with a new term rewriting system (TRS) on *(A)Synchronous Effects* (*ASyncEffs*).

Firstly, we formally define the full-featured Esterel, generalizing the preemptive asynchronous abstraction. Secondly, we propose *ASyncEffs*, a new effect logic, that extends *Synchronous Kleene Algebra* with a *waiting* operator. Thirdly, we establish an effect system via a set of forward verification rules. Lastly, we present a purely algebraic TRS to efficiently check language inclusions between *ASyncEffs*. To show the feasibility, we prototype the verification system; prove its correctness; report experimental results, and investigate how it can help to detect errors related to both synchronous preemptions and asynchronous promises.

4.1 Introduction

Synchronous programming [Ben+03] has found success in many safety-critical applications, such as fly-by-wire systems and nuclear power plant control software[1]. It exhibits a high concurrency but calls for deterministic and predictable execution,

[1]Concretely, it has been used in the creation and verification of fuel control systems; landing gear control functions; virtual display systems at Dassault Aviation [Ber+00]; the control software of the N4 nuclear power plants; the Airbus A320 fly-by-wire system; and the specification of part of Texas Instrument's digital signal processors [Ben+03].

which has been considered a clean formalism for modeling, specifying, validating, and implementing reactive systems. Languages based on this paradigm – such as Esterel [BG92], Lustre [RHR91] and Signal [BLJ91] – assume that time is partitioned into discrete instants/clock ticks and the computation/communication for processing all events that occur within one time instant happen instantaneously.

Many mainstream languages, such as C#, Java, JavaScript, Rust, Python, and Swift, have recently added support for asynchronous promises, also known as *futures* or *tasks* [Bie+12]. These features support basic asynchronous operators, such as *yield* pauses/resumes generator functions asynchronously; *async/await* simplify the blending of asynchronous executions into sequential programming. However, most these languages offer a small set of preemption primitives, often inadequate for concisely modeling interruptions or control-driven computations.

To make reactive programming more concise and expressive, recent innovations are dedicated to integrating synchronous features to asynchronous infrastructures. For example: the language HipHop.js [BS20] is a mixture of JavaScript and Esterel for reactive web applications, which facilitates JavaScript with preemptions like *every* and *abort*; the Scala library ZIO [Mai22] is for type-safe asynchronous and concurrent programming with rudimentary preemptive operators, such as *zipPar* and *race*; similarly, microsoft's durable function [ptc19] deploys blended asynchrony/synchrony, including pause, parallel composition for deterministic orchestration functions, which is available as libraries for C#, JavaScript, Python.

There is a growing need to reason about such preemptive asynchronous reactive programs with multifarious preemptions. In particular, we are interested in the techniques for specifying and verifying temporal behaviors of such execution models, which have not been extensively studied. Here, we mainly tackle the challenges related to synchronous preemptions and asynchronous promises.

The power of preemption appears in Figure 4.1. Module `Main` makes use of three submodules: `Identity` reads the GUI and enables the login button when the input username and password are both longer than two characters; `Authenticate` calls the authorization service and output the signal `connected` when authorized; `Session` establishes an active communication session between the authorized user

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

Figure 4.1: A preemptive program written in Esterel, for a simple web login, drawn from [BS20].

and the server.

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

Although simple, Figure 4.1 shows that preemption statements allow a clean, hierarchical description of temporal behaviors. While flexible and expressive, preemption primitives have fairly complex semantics, which in turn makes reasoning difficult. In this paper, we study the subtle operational semantics of various preemptions, including: *interrupt*; *abort*; *suspend*; *every*; and the *label-based escape*. Furthermore, we show that our approach supports a comprehensive foundation for verifying preemptions with different keywords, including *strong* or *weak*, *immediate* or *delayed*.

With asynchronous promises, we can perform long-lasting tasks without blocking the main thread. The keywords *async* and *await* allow sequential-style code to capture concurrent executions with explicit dependencies via asynchronous signals succinctly. However, promises are complex and error-prone in their own right. Prior works [MLT17; Ali+18] display a set of broken promises chain *anti-patterns* shown in asynchronous JavaScript, and propose to detect the anti-patterns by constructing a promise dependency graph. In this paper, we focus on the *async/await* related anti-pattern, where *unreachable promises* may have registered reactions that will never be executed. Different from prior works, we show that our purely algebraic approach detects such unreachable promises without any constructions of graphs.

To achieve a modular verification - where modules can be replaced by their already verified properties - for preemptive asynchronous programs, and exploit the best of both synchronous and asynchronous execution models, we propose a novel temporal specification language, which enables compositional verification via a forward verifier and a term rewriting system (TRS). More specifically, we specify system behaviors in the form of *ASyncEffs*, which enriches the Synchronous Kleene Algebra (SKA) [Pri10; Bro+15] with a new operator, to provide the *waiting* abstraction into classic linear temporal verification. Our main contributions are:

1. **Language Abstraction:** we formally define the operational semantics for the full-featured Esterel and use it to generalize the preemptive asynchronous programs.

2. **Specification Logic:** we propose *ASyncEffs*, by defining its syntax and semantics. We show that in our proposal, *ASyncEffs*' expressiveness power subsumes both classic linear temporal logic (LTL) and the *past-time* LTL [Rei+11].

3. **Forward Verifier:** we establish an axiomatic semantics to infer the behaviors of given programs, and check the real behaviors against the specifications.

4. **An Efficient TRS:** we present rewriting rules to prove/disprove the entailments between *ASyncEffs*. The TRS is a back-end engine deployed by the front-end Forward Verifier; also can work independently outside of our proposal.

5. **Implementation and Evaluation:** we prototype our proposal, prove its correctness, report on experimental results and investigate how it can help to debug errors related to preemptions and asynchronous promises.

4.2 Verification Challenges

Verification techniques for languages, with features like the full-featured Esterel, are expected to overcome verification challenges from the both synchronous and asynchronous worlds. This section outlines the existing specification requirements in Esterel and asynchronous JavaScript, then proposes our solution to addresses corresponding verification challenges.

4.2.1 A Sense of Esterel: Perfect Synchrony and Preemption

A synchronous program reacts to its environment in a sequence of *logical ticks*, and computations within a tick are assumed to be instantaneous. Being suitable for control-dominated model designs, Esterel has found success in many safety-critical applications, that need strong guarantees, can be attributed to its precise semantics and simpler computational model [BG92; Ber99].

Esterel treats computation as a series of deterministic reactions to external signals. All parts of a reaction complete in a single, discrete-time *event*. Events exhibit deterministic concurrency; and each reaction may trigger concurrent threads without execution order affecting the computation result. Primitive constructs can be assumed to execute in zero time, except for the *pause* statement. Hence, each execution trace is a sequence of logical clock events, separated by explicit pauses. In each event, several computations take place simultaneously.

To maintain determinism and synchrony, evaluation in one thread of execution may affect code arbitrarily far away in the program. Thus, 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 (cf. subsection 4.3.1 for the syntax). **The first challenge** of programming Esterel is the *Logical Correctness* issue, caused by non-local executions. This difficulty is resolved by assuming that there exists precisely one status for each signal. The demonstration examples are shown in Table 4.1

In Table 4.1-(a), if the local signal S1 was *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

```
signal S1 {
  present S1
  then nothing
  else emit S1
}
```

(a) No valid assignments (Logically incorrect).

```
signal S1 {
  present S1
  then emit S1
  else nothing
}
```

(b) Two possible assignments (Logically incorrect).

```
/*@ requires {S1} @*/
/*@ ensures  {S1} @*/
present S1
then emit S1
else nothing
```

(c) One assignment under the precondition (Logically correct).

Table 4.1: Logical correctness examples in Esterel.

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 thus assumed to be logically incorrect.

Consider the revised program in Table 4.1-(b), if 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 contradictory assignment to the signal in this program, which is thus classified as logically incorrect. Our verification is able to detect such logical errors, which concludes $\perp$ (*false*) for Table 4.1-(a), and concludes $\{S1\} \vee \{\overline{S1}\}$ for Table 4.1-(b), cf, subsubsection 4.7.1.1. However, if S1 is not a local signal, and can be composed to a bigger context, as shown in Table 4.1-(c), our modular approach allows a precondition which guarantees that the environment emits S1. In this case, the code snippet becomes logically correct, because there is only one consistent assignment to S1. This was not possible in prior work [Flo+19].

The Second challenge is to model/verify programs with concurrent execution that is compatible with preemptions. Coordination in concurrent systems can result from message exchanges. In Esterel, it can also result from *process preemption* [Ber93] (cf. Figure 4.1), which is an explicit control mechanism that consists in denying the right to work on a process, either permanently (e.g., abortion) or temporarily (e.g., suspension). Most existing languages offer a small set of preemption primitives, but this is often inadequate for concise modelling of reactive systems. Preemption is particularly important in control-dominated reactive programming, where much of modelling comprise of handling interrupts and control-driven computation. While expressive, preemption primitives have fairly complex semantics, which in turn makes reasoning difficult. Here, based on our operational semantics model, we devise a parallel merge operator, to soundly calculate temporal traces from concurrent preemptive threads.

4.2.2 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, also known as *futures* or *tasks* [Bie+12]. As an example, consider the JavaScript program in Figure 4.2. It uses the `fs` module (line 1) to load a 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     ...// operations do not rely on the result of loading file
6     const data = await task; // block waiting
7     ...// logging or data processing of the Json file
8 }
```

Figure 4.2: Using Async-Await in JavaScript.

The function `read` takes a string called `filePath`. The keyword `async` indicates that this function is to be executed asynchronously. Calling `fs.readFile` in line 4 does not block subsequent computations, such as line 5. The main thread will only be blocked by `await` statements, that waits for the result of an earlier `fs.readFile` asynchronous call.

The goal of the asynchronous programming model is to support concurrent execution that expresses dependency between producers and consumers of information (via its corresponding asynchronous signals). Its use allows sequential-style code to succinctly capture concurrent execution with precise dependency via asynchronous signals. Due to its more complex control flow mechanism, existing reasoning methods for languages with synchrony do not directly support asynchronous signals.

Prior works [MLT17; Ali+18] display a set of broken promises chain *anti-patterns* shown in asynchronous JavaScript, which are mostly caused by using promises without sufficient static checking. They propose to mitigate the anti-patterns by constructing a promise dependency graph, and enforce a static checking upon the graph. Here we focuses on the *unreachable reactions* anti-pattern, where an unsettled promise may have registered reactions that will not be executed. Different from prior works, a case study in Table 4.6 shows that our purely algebraic approach detects this anti-pattern without any constructions of graphs.

4.3 Language and Specifications

This section first introduces the target language and then depict the temporal specification language which supports *ASyncEffs*.

4.3.1 The Target Language

We summarize a full-featured Esterel in Figure 4.3, to be our target language, which provides the infrastructure for preemptive asynchronous abstraction. The statements marked as purple are generalized from the preemptive statements in (reactive) synchronous programming, while the statements marked as blue provide the *async/await* constructs for programming asynchronous promises.

Here, S, x are meta-variables ranging over signal variables, constants. Signal types are: *in* for input signals, *out* for output signals and *inout* for both. **var** represents the countably infinite set of arbitrary distinct identifiers. A program $\mathcal{P}$ comprises a list of module definitions $\overrightarrow{fun}$[2]. Each *module* has a name mn, a list of well-typed arguments $\overrightarrow{\tau\ S(x)}$, a statement-oriented body p, associated with a

[2] Here, we use the $\rightarrow$ script to denote a finite vector (possibly empty) of items.

$$
\begin{array}{llcl}
(\mathit{Program}) & \mathcal{P} & ::= & \overrightarrow{fun} \\
(\mathit{Signal\ Types}) & \tau & ::= & in \mid out \mid inout \\
(\mathit{Module\ Def.}) & fun & ::= & mn\ \ (\overrightarrow{\tau\ S(x)})\ \ \langle \mathbf{req}\ \ \Phi_{pre}\ \ \mathbf{ens}\ \ \Phi_{post}\rangle\ \ p \\
(\mathit{Values}) & v & ::= & () \mid i \mid b \mid x \\
(\mathit{Parametrized\ Signal}) & \mathbb{S} & ::= & S(v) \\
(\mathit{Statements}) & p, q & ::= & v \mid yield \mid emit\ \mathbb{S} \mid p; q \mid p||q \mid call\ mn\ (\overrightarrow{\mathbb{S}}) \\
\multicolumn{4}{l}{\quad \mid loop\ p \mid signal\ S\ in\ p \mid present\ S\ then\ p\ else\ q \mid async\ S\ p\ q \mid await\ [\kappa_1]\ S} \\
\multicolumn{4}{l}{\quad \mid trap\ p \mid exit\ d \mid [\kappa_2]\ abort\ p\ S \mid [\kappa_2]\ suspend\ p\ S} \\
(\mathit{Preemption\ Keywords}) & \multicolumn{3}{l}{\kappa_1 ::= immediate \mid delayed \qquad \kappa_2 ::= weak \mid strong}
\end{array}
$$

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

Figure 4.3: Esterel Syntax.

precondition Φ_{pre} and a postcondition Φ_{post}. We here present the intuitive semantics of the basic statements.

A thread of execution suspends itself for the current instant using the *yield* construct, and resumes when the next instant started[3]. Statement *emit* $\mathbb{S}$ broadcasts the signal $\mathbb{S}$ to be *present*. The emission of $\mathbb{S}$ is valid for the current instant only. The sequence statement $p; q$ starts p and instantaneously passes the control flow to q when p terminates. Statement q is never started if p always yields. Parallel statement $p||q$ runs p and q in parallel. The branches can terminate in different instants, and the parallel waits for the last one to terminate. Statement *call mn* $(\overrightarrow{\mathbb{S}})$ is a call to module *mn*, providing the list of IO signals. Statement *signal S in p* starts p with a local signal S, overriding any S might already be declared. Statement *present* $\mathbb{S}$ p q immediately starts p if S is present in the current instant; otherwise it starts q instead. Statement *loop p* implements an infinite loop.

Keywords *immediately* and *delayed* are for *await* statements, indicating waiting for a signal from the current instant or the next instant, respectively. Keywords *weak* or *strong* are for preemptive statements, indicating to allow or not allow, respectively,

[3]For a better *cooperative multitasking* [Wik22a], processes voluntarily yield control periodically or when idle or logically blocked.

the current instant to execute when the preemption condition is met.

In this work, we use the *immediately* waiting and *weak* preemptions by default, with discussions on how to cooperate with the *delayed* waiting and *strong* preemptions in detail.

4.3.2 Operational Semantics of the Target Language

The reduction rules are in the form of $p \xrightarrow[\mathcal{E}]{\alpha,k} p'$, meaning that a process p performs an action α in the context of a signal environment $\mathcal{E}$, then becomes a process p' with a completion code k. $\mathcal{E}$ is the set of all the signals produced at the instant by the whole program of which p is part, which gives the global information about the presence and absence of signals. In particular, $\alpha \subseteq \mathcal{E}$.

The *completion code* k is a non-negative integer: when k=0, the reduction completes without exits nor yields; when k=1, it completes without exits but with a yield; when k=2, it completes with an exit which escapes the nearest trap; when k>2, it completes with an exit which escapes a further enclosing *trap*. Such an encoding for preemptions was first advocated by Gonthier [Gon88].

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

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

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

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

Due to the static scope of signals, $\mathcal{E}$ may already contain the $\mathbb{S}$, therefore the notation $\mathcal{E}\backslash S$ denotes the instant obtained by removing any existing S from $\mathcal{E}$. The rule $(Call)$ retrieves the function body p of mn from the program, and executes p. $\overrightarrow{\mathbb{S}}\backslash E$ represents the output signals declared in the callee function.

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

If the signal is present in the current instant, the first clause is instantly executed. Otherwise, the else clause is instantly executed.

$$\frac{(\mathbb{S}{\mapsto}present) \in \mathcal{E} \qquad p \xrightarrow[\mathcal{E}]{\alpha,k} p'}{present\ \mathbb{S}\ p\ q \xrightarrow[\mathcal{E}]{\alpha,k} p'}(Present\text{-}1) \qquad \frac{(\mathbb{S}{\mapsto}present) \notin E \qquad q \xrightarrow[\mathcal{E}]{\alpha,k} q'}{present\ \mathbb{S}\ p\ q \xrightarrow[\mathcal{E}]{\alpha,k} q'}(Present\text{-}2)$$

Figure 4.4 provides the operational semantics of the promise-related and preemptive statements in the full-featured Esterel.

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

Statement *abort* p $\mathbb{S}$ performs p and terminates when $\mathbb{S}$ occurs. Statement *suspend* p $\mathbb{S}$ suspends p for one instant when $\mathbb{S}$ is present in the environment. Statement *trap* p installs a trap and behaves like p until any *exit* occurs. Statement *exit* d instantaneously exits the trap with depth d. The rules for parallel statements execute the branches independently, then merge their output events accordingly. If one branch exits with code k, then both threads are preempted with the exception depth k. If both statements exit distinct traps with k_1 and k_2 in the same instant, then the execution exits with the larger value.

Derived Statements.

Figure 4.5 shows how to construct the derived statements via the primitives. Particularly, *every* $\mathbb{S}$ p implements a preemptive loop that checks for a condition, here the presence of $\mathbb{S}$. An *every* loop starts its body when its condition is true; but

$$[Async]\ \frac{(p;\, emit\ \mathbb{S}) \xrightarrow[\mathcal{E}]{\alpha,k_1} p' \qquad q \xrightarrow[\mathcal{E}]{\beta,k_2} q'}{async\ \mathbb{S}\ p\ q \xrightarrow[\mathcal{E}]{\alpha\cup\beta,max(k_1,k_2)} p'||q'} \qquad [Await\text{-}1]\ \frac{(\mathbb{S}\mapsto present) \in \mathcal{E}}{await\ \mathbb{S} \xrightarrow[\mathcal{E}]{\emptyset,1} ()} \qquad [Await\text{-}2]\ \frac{(\mathbb{S}\mapsto present) \notin \mathcal{E}}{await\ \mathbb{S} \xrightarrow[\mathcal{E}]{\emptyset,1} await\ \mathbb{S}}$$

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

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

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

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

Figure 4.4: Operational semantics of the promise-related and preemptive statements in the full-featured Esterel.

(1) $halt \triangleq loop\ (yield)$

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

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

(4) $await\ (delayed)\mathbb{S} \triangleq yield;\, await\ \mathbb{S}$

Figure 4.5: Expansion of derived preemptions.

whenever the condition is met again in some further instants, it kills the current execution instantly to restart a new iteration. Besides, *await (delayed)* $\mathbb{S}$ implements a *delayed* waiting which starts as early as the next instant, as opposite to the default *immediate* waiting.

4.3.3 An Effect Logic for the Temporal Specification

$$
\begin{array}{rrcl}
(\textit{Effects}) & \theta & ::= & \bot \mid \epsilon \mid I \mid \mathbb{S}? \mid \theta_1{\cdot}\theta_2 \mid \theta_1{\vee}\theta_2 \mid \theta_1||\theta_2 \mid \theta^\star \\
(\textit{Instant}) & I & ::= & \{\} \mid \{\mathbb{S} \mapsto \alpha\} \mid I_1 \cup I_2 \\
(\textit{Parametrized Signal}) & \mathbb{S} & ::= & S(v) \\
(\textit{Signal Statuses}) & \alpha & ::= & \textit{present} \mid \textit{absent} \mid \textit{undef}
\end{array}
$$

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

Figure 4.6: Syntax of *ASyncEffs*.

As shown in Figure 4.6, *ASyncEffs* comprise *false* ($\bot$); the empty trace ϵ; the singleton instant I; waiting for a parametrized signal $\mathbb{S}?$; trace concatenation $\theta_1 \cdot \theta_2$; trace disjunction $\theta_1 \vee \theta_2$; synchronous parallelism $\theta_1||\theta_2$. *ASyncEffs* can also be constructed by $\star$, representing zero or more times of repetition of a trace. There are three possible statuses for a signal: present, absent and undefined. The default status of signals in a new instant is *undefined*. An instant I is a set of mappings from signals to their statuses; and instants can be empty sets $\{\}$, indicating no signal constraints for the instant.

4.3.4 Semantic Model of *ASyncEffs*

To define the semantic model, we use φ (*a trace of instants*) to represent the concrete the computation execution. Let $\varphi \models \Phi$ denote the model relation, i.e., linear temporal instants φ satisfy the temporal effects Φ, with φ from the concrete domain: $\varphi \triangleq list(I)$. Figure 4.7 defines the semantics of *ASyncEffs*.

[] represents an empty sequence; ++ appends two sequences; $[I]$ represents a singleton sequence contains one instant I. Here I is a list of mappings from parametrised signals to statuses. We use $\{\mathbb{S}\}$ and $\{\overline{\mathbb{S}}\}$ to shorthand $\{\mathbb{S} \mapsto \textit{present}\}$ and $\{\mathbb{S} \mapsto \textit{absent}\}$ respectively. The signals shown in one instant represent the *minimal* set of signals which are required/guaranteed to be there. Any instant

$$
\begin{array}{lll}
\varphi \models \epsilon & \textit{iff} & \varphi = [] \\
\varphi \models I & \textit{iff} & \varphi = [I] \\
\varphi \models \mathbb{S}? & \textit{iff} & \exists n{\geq}0.\ \varphi{=}\{\overline{\mathbb{S}}\}^n{+\!+}[\{\mathbb{S}\}] \\
\varphi \models \theta_1 \cdot \theta_2 & \textit{iff} & \exists \varphi_1, \varphi_2,\ \varphi{=}\varphi_1{+\!+}\varphi_2 \textit{ such that } \varphi_1{\models}\theta_1 \text{ and } \varphi_2{\models}\theta_2 \\
\varphi \models \theta_1 {\vee} \theta_2 & \textit{iff} & \varphi \models \theta_1 \text{ or } \varphi \models \theta_2 \\
\varphi \models \theta_1 || \theta_2 & \textit{iff} & \varphi \models \theta_1 \text{ and } \varphi \models \theta_2 \\
\varphi \models \theta^\star & \textit{iff} & \varphi \models \epsilon \textit{ or } \varphi \models (\theta \cdot \theta^\star) \\
\varphi \models \bot & \textit{iff} & \textit{false}
\end{array}
$$

Figure 4.7: Semantics of *ASyncEffs*.

contains contradictions, such as $\{\mathbb{S}, \overline{\mathbb{S}}\}$, will lead to *false*, as the signal $\mathbb{S}$ can not be both present and absent.

Expressiveness.

As shown in Table 4.2, we are able to recursively encode event-based LTL operators into *ASyncEffs*, making it more intuitive and readable, mainly when nested operators occur. By putting effects in the postcondition, they restrict *future traces*; whereas in the preconditions, they naturally encode *past-time* temporal specifications. The basic modal operators are: $\Box$ for "globally"; $\Diamond$ for "finally"; $\bigcirc$ for "next"; $\mathcal{U}$ for "until", and their past time reversed versions: $\overleftarrow{\Box}$; $\overleftarrow{\Diamond}$; and $\ominus$ for "previous"; $\mathcal{S}$ for "since". Besides, the implication operator is expressed as $\mathbb{A} \to \mathbb{B} \equiv \{\overline{\mathbb{A}}\} \vee \{\mathbb{A}, \mathbb{B}\}$. Apart from the high compatibility with standard first-order logic, *ASyncEffs* make the temporal verification more flexible to incorporate with other logics. ($\{\mathbb{A}\}, \{\mathbb{B}\}$ represent different instants which contain signal $\mathbb{A}$ and $\mathbb{B}$ to be present.)

Table 4.2: Examples for converting LTL formulae into Effects.

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

4.4 Automated Forward Verification

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

$\langle H', C', K'\rangle = \bigcup_{i=0}^{|\langle H,C,K\rangle|\text{-}1} \langle H'_i, C'_i, K'_i\rangle$ *where* $\mathcal{E} \vdash \langle h_i, c_i, k_i\rangle\ p\ \langle H'_i, C'_i, K'_i\rangle$.
We summarize the forward reasoning rules in Figure 4.8.

[*FV-Nothing*] obtains the next program state by inheriting the current state. [*FV-Emit*] updates the current instant with $\mathbb{S}$ pointing from *undef* to *present*. [*FV-Yield*] archives the current instant to the history trace, then initializes a new current instant by an empty instant. [*FV-Local*] computes p's effects by eliminating the existing signal S. [*FV-Present*] computes the effects of p and q by extending the current instant with $\mathbb{S}$ being *present* and *absent*, respectively. The final state is an union of the results. [*FV-Seq*] computes p's effects. If the completion code $K_1{\leq}1$, i.e., there are no exits raised, then further computes $\langle H_2, C_2, K_2\rangle$ by continuously compute the effects of q. Otherwise, it discards q and returns $\langle H_1, C_1, K_1\rangle$ directly. The rule [*FV-Loop*] computes p's effects. If the completion code of the first run $K_1{\leq}1$, it appends a repeated trace to the history $h \cdot (H_1)^{\star}$ with C_1 to be current instant. Otherwise, it simply exits the loop.

[*FV-Call*] triggers the back-end solver TRS to check if the precondition of the callee is satisfied by the current state. If it holds, the rule obtains the final state by concatenating the postcondition Φ_{post} to the current effect state. Otherwise the verification fails. [*FV-Async*] de-sugars the asynchronous construct into a parallel program. [*FV-Await*] archives the current instant and appends $\mathbb{S}?$ to the history trace. [*FV-Exit*] updates the value of k using $d{+}2$. [*FV-Trap*] computes p's effects. When $K{\leq}1$, it means there is no exits to be handled, therefore the final effects is

$$\frac{}{\mathcal{E} \vdash \langle h, c, k\rangle\ ()\ \langle h, c, k\rangle}[FV\text{-}Nothing] \qquad \frac{}{\mathcal{E} \vdash \langle h, c, k\rangle\ emit\ \mathbb{S}\ \langle h, c{+}\{\mathbb{S}\}, k\rangle}[FV\text{-}Emit]$$

$$\frac{[FV\text{-}Yield]}{\mathcal{E} \vdash \langle h, c, k\rangle\ yield\ \langle h \cdot c, \{\}, k\rangle} \qquad \frac{\begin{array}{c}[FV\text{-}Local]\\ (\mathcal{E}\backslash S) \vdash \langle h, c\backslash S, k\rangle\ p\ \langle H', C', K'\rangle\end{array}}{\mathcal{E} \vdash \langle h, c, k\rangle\ signal\ S\ in\ p\ \langle H', C', K'\rangle}$$

$$\frac{\mathcal{E} \vdash \langle h, c{+}\{\mathbb{S}\}, k\rangle\ p\ \langle H_1, C_1, K_1\rangle \qquad \mathcal{E} \vdash \langle h, c{+}\{\overline{\mathbb{S}}\}, k\rangle\ q\ \langle H_2, C_2, K_2\rangle}{\mathcal{E} \vdash \langle h, c, k\rangle\ present\ \mathbb{S}\ p\ q\ \langle H_1, C_1, K_1\rangle \cup \langle H_2, C_2, K_2\rangle}[FV\text{-}Present]$$

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

$$\frac{\begin{array}{ll}\mathcal{E} \vdash \langle \epsilon, c, k\rangle\ p\ \langle H_1, C_1, K_1\rangle & \\ \langle H', C', K'\rangle{=}\langle h \cdot (H_1)^{\star}, C_1, K_1\rangle & (K_1{\le}1)\\ \langle H', C', K'\rangle{=}\langle h \cdot H_1, C_1, K_1\rangle & (K_1{>}1)\end{array}}{\mathcal{E} \vdash \langle h, c, k\rangle\ loop\ p\ \langle H', C', K'\rangle}\ [FV\text{-}Loop]$$

$$\frac{\begin{array}{c}mn(\overrightarrow{\tau\ S(x)})\ \langle \mathbf{req}\Phi_{pre}\ \mathbf{ens}\Phi_{post}\rangle\ p \in \mathcal{P} \qquad h \cdot c \sqsubseteq \Phi_{pre}[\overrightarrow{\mathbb{S}}/\overrightarrow{S(x)}]\\ \langle H_1, C_1, K'\rangle{=}\Phi_{post}[\overrightarrow{\mathbb{S}}/\overrightarrow{S(x)}]\end{array}}{\mathcal{E} \vdash \langle h, c, k\rangle\ call\ mn\ (\overrightarrow{\mathbb{S}})\ \langle H \cdot H_1, C_1, K'\rangle}[FV\text{-}Call]$$

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

$$\frac{\langle \Delta\rangle = \langle h \cdot (c||\mathbb{S}?), \{\}, k\rangle}{\mathcal{E} \vdash \langle h, c, k\rangle\ await\ \mathbb{S}\ \langle \Delta\rangle}\ [FV\text{-}Await] \qquad \frac{k'{=}d{+}2}{\mathcal{E} \vdash \langle h, c, k\rangle\ exit\ d\ \langle h, c, k'\rangle}\ [FV\text{-}Exit]$$

$$\frac{\begin{array}{lll}\mathcal{E} \vdash \langle \epsilon, c, k\rangle\ p\ \langle H, C, K\rangle & \langle \Delta\rangle{=}\langle H, C, K\rangle & when\ (K{\le}1)\\ & \langle \Delta\rangle{=}\langle H, C, 0\rangle & when\ (K{=}2)\\ & \langle \Delta\rangle{=}\langle H, C, K\text{-}1\rangle & when\ (K{>}2)\end{array}}{\mathcal{E} \vdash \langle h, c, k\rangle\ trap\ p\ \langle h \cdot \Delta\rangle}\ [FV\text{-}Trap]$$

$$\frac{\begin{array}{c}\mathcal{E} \vdash \langle \epsilon, c, k\rangle\ p\ \langle H_1, C_1, K_1\rangle \qquad \mathcal{E} \vdash \langle \epsilon, c, k\rangle\ q\ \langle H_2, C_2, K_2\rangle\\ \vdash_{pm} \langle H_1, C_1, K_1\rangle||\langle H_2, C_2, K_2\rangle \leadsto \langle \Delta\rangle\end{array}}{\mathcal{E} \vdash \langle h, c, k\rangle\ p||q\ \langle h \cdot \Delta\rangle}\ [FV\text{-}Par]$$

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

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

Figure 4.8: Forward Rules for *ASyncEffs*

just $\langle H, C, K\rangle$. When K equals to 2, it means there is an exit need to be handled by the current *trap*. When K is greater than 2, it means the current exit needs to be handled by an outer *trap* statement, therefore it returns the final effects as $\langle H, C, K\text{-}1\rangle$. [$FV$-$Par$] computes p and q's effects independently, then *parallel merges* the results. Notation $\vdash_{pm}$ refers to the *parallelMerge* algorithm, which is detailed in subsection 4.4.1.

Definition 7 (Prepend Program States)**.** Given a history trace h', and program states Δ=(H, C, K), we define that: $h' \cdot \Delta = \{(h' \cdot h, c, k) \mid (h, c, k) \in \langle H, C, K\rangle\}$.

[FV-$Abort$] and [FV-$Suspend$] compute the effects of p by initializing the history trace with ϵ; then compute their corresponding interleaves[4]; lastly, prepend the original history to the final results.

Algorithm 1: Abort Interleaving

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

rec function $\aleph^{Abort(\mathbb{S},I,K)}_{Interleave}(\Phi, \Phi_{his})$
- **if** $fst(\Phi)$=$\emptyset$ **then**
  - $\Delta_1 \leftarrow [(\Phi_{his}, I\text{+}\{\mathbb{S}\}, 0)]$ ▷ Notion **+** unions two instants
  - $\Delta_2 \leftarrow [(\Phi_{his}, I\text{+}\{\overline{\mathbb{S}}\}, k)]$
  - **return** $(\Delta_1 \cup \Delta_2)$
- **else**
  - $\Delta \leftarrow []$
  - **foreach** $f \in fst(\Phi)$ **do**
    - $\phi \leftarrow [(\Phi_{his}, f\text{+}\{\mathbb{S}\}, 0)]$
    - $\Phi' \leftarrow D_f(\Phi)$
    - $\Phi'_{his} \leftarrow \Phi_{his} \cdot (f\text{+}\{\overline{\mathbb{S}}\})$
    - $\Delta' \leftarrow \aleph^{Abort(\mathbb{S},I,K)}_{Interleave}(\Phi', \Phi'_{his})$
    - $\Delta \leftarrow \Delta \cup \phi \cup \Delta'$ ▷ Notion $\cup$ unions two states
  - **return** Δ

Algorithm 1 presents the interleaving algorithm for *abortions* (cf. Definition 9 and Definition 11 for *First*(fst) and *Derivative*(D) respectively). Note that, we use *weak* abort/suspend as default in this work. The difference is: in strong

---

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

preemption, the body does not run when the preemption condition holds. Whereas in weak preemption the body is allowed to run in the current instant even when the preemption condition holds, but are terminated thereafter [Ber93; PRS95]. We present the (weak) interleaving algorithm for *suspensions*, and demonstrate how to encode strong abort/suspend in section A.1.

4.4.1 Parallel Merge Algorithm

The parallel merging [5] rules are in the form: $\vdash_p \langle H_1, C_1, K_1\rangle || \langle H_2, C_2, K_2\rangle \rightsquigarrow \langle H', C', K'\rangle$. Given two sets of program states $\langle H_1, C_1, K_1\rangle$ and $\langle H_2, C_2, K_2\rangle$, the rule $[PM\text{-}Union]$ obtains $\langle H', C', K'\rangle$ by combining the parallel merged states of their cartesian products.

$$\frac{\forall (h_1, c_1, k_1) \in \langle H_1, C_1, K_1\rangle \qquad \forall (h_2, c_2, k_2) \in \langle H_2, C_2, K_2\rangle \atop \langle H', C', K'\rangle = \bigcup(\vdash_{pm} \langle h_1, c_1, k_1\rangle || \langle h_2, c_2, k_2\rangle \rightsquigarrow \langle h', c', k'\rangle)}{\vdash_{pm} \langle H_1, C_1, K_1\rangle || \langle H_2, C_2, K_2\rangle \rightsquigarrow \langle H', C', K'\rangle}[PM\text{-}Union]$$

$[PM\text{-}Unfold]$

$$\frac{F_1{=}fst(h_1) \qquad F_2{=}fst(h_2) \qquad \forall f_1 \in F_1 . \forall f_2 \in F_2 .\ I{=}f_1{\cup}f_2, \atop der_1{=}D_I(h_1) \qquad der_2{=}D_I(h_2) \qquad \langle H', C', K'\rangle{=}\bigcup(\vdash_{pm}\langle der_1, c_1, k_1\rangle || \langle der_2, c_2, k_2\rangle)}{\vdash_{pm} \langle h_1, c_1, k_1\rangle || \langle h_2, c_2, k_2\rangle \rightsquigarrow \langle I \cdot H', C', K'\rangle}$$

$[PM\text{-}Unfold]$ applies to deductive steps, which deploys auxiliary functions, $fst(\theta)$ and $D_I(\theta)$, to compute the firsts instant and derivatives of an effect, cf. subsection 4.5.1. The rule gets the *first* set from h_1 and h_2 respectively. For each pair of (f_1, f_2), it merges f_1 and f_2 to be the common *first*, denoted as I; then gets the derivatives of h_1 and h_2 w.r.t I respectively; Finally it prepends I into the parallel merged derivatives, by recursively calling the parallel merge algorithm. The next

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

rules deal with base cases and terminate the merging process.

$$[\mathit{PM\text{-}EqLen}]\quad \frac{c'{=}c_1{\cup}c_2 \qquad k'{=}max(k_1,k_2)}{\vdash_{pm} \langle \epsilon, c_1, k_1\rangle || \langle \epsilon, c_2, k_2\rangle {\rightsquigarrow} \langle \epsilon, c', k'\rangle} \qquad [\mathit{PM\text{-}Cut}]\quad \frac{k_1{>}1 \qquad c'{=}c_1{\cup}fst(h_2)}{\vdash_{pm} \langle \epsilon, c_1, k_1\rangle || \langle h_2, c_2, k_2\rangle {\rightsquigarrow} \langle \epsilon, c', k_1\rangle}$$

$$\frac{k_1{\le}1 \qquad h'{=}c_1||h_2}{\vdash_{pm} \langle \epsilon, c_1, k_1\rangle || \langle h_2, c_2, k_2\rangle \rightsquigarrow \langle h', c_2, k_2\rangle}[\mathit{PM\text{-}Absorb}]$$

[*PM-EqLen*] is used when two effects have the same length. [*PM-Cut*] is used when one of the effects is shorter than the other and raises an exception. [*PM-Absorb*] is used when one of the effects is shorter than another yet without any exceptions.

4.4.2 Soundness Theorem of the Forward Rules

Theorem 2 (Soundness of the Forward Rules)**.**
$\forall p, \mathcal{E}$, if $\mathcal{E} \vdash \langle h, c, k\rangle\ p\ \langle H', C', K'\rangle$, and $\varphi \models h$,
and $p \xrightarrow[\mathcal{E}]{e_0,0}{}^{*} p' \xrightarrow[\mathcal{E}]{\emptyset,1} p_1 \xrightarrow[\mathcal{E}_1]{e_1,0}{}^{*} p_1' \xrightarrow[\mathcal{E}_1]{\emptyset,1} p_2 \xrightarrow[\mathcal{E}_2]{e_2,0}{}^{*} p_2' \xrightarrow[\mathcal{E}_2]{\emptyset,1} \ldots p_n \xrightarrow[\mathcal{E}_n]{e_n,0}{}^{*} p_n' \xrightarrow[\mathcal{E}_n]{\emptyset,k_f(\neq 1)} ()$,
then it implies that $\exists (h', c', k_f) \in \langle H', C', K'\rangle$ such that $\varphi \mathbin{++} [e_0; e_1; ...; e_n] \models h' \cdot c'$.
(Note that, $p \xrightarrow{e,0}{}^{*} p'$ denotes the reflexive, transitive closure of $p \xrightarrow{e,0} p'$.)

Proof. By induction on the structure of p:

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

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

3. **Present:** $\mathcal{E} \vdash \langle h, c, k\rangle\ \mathit{present}\ \mathbb{S}\ p\ q\ \langle H_1, C_1, K_1\rangle \cup \langle H_2, C_2, K_2\rangle$, $\varphi \models H$, where
$\mathcal{E} \vdash \langle h, c{+}\{\mathbb{S}\}, k\rangle\ p\ \langle H_1, C_1, K_1\rangle$ and $\mathcal{E} \vdash \langle h, c{+}\{\overline{\mathbb{S}}\}, k\rangle\ q\ \langle H_2, C_2, K_2\rangle$.
- When $(\mathbb{S}) \in c$, *present S p q* $\rightsquigarrow p$, the inclusion is proved by inductive hypothesis.
- When $(\overline{\mathbb{S}}) \notin c$, *present S p q* $\rightsquigarrow q$, the inclusion is proved by inductive hypothesis.

4. **Sequence:** $\mathcal{E} \vdash \langle h, c, k\rangle\ \mathit{seq}\ p\ q\ \langle H', C', K'\rangle$, $\varphi \models H$, where
$\mathcal{E} \vdash \langle h, c, k\rangle\ p\ \langle H_1, C_1, K_1\rangle$, $\mathcal{E} \vdash \langle H_1, C_1, K_1\rangle\ q\ \langle H_2, C_2, K_2\rangle$ *and*

$\langle H', C', K'\rangle{=}\langle H_2, C_2, K_2\rangle(K_1{\leq}1)$ or $\langle H', C', K'\rangle{=}\langle H_1, C_1, K_1\rangle(K_1{>}1)$

- When $K_1{=}0$, $seq\ p\ q \xrightarrow[\mathcal{E}]{e_0,0}{}^{*} ... \xrightarrow[\mathcal{E}_n]{e_n,0}{}^{*}\ q \xrightarrow[\mathcal{E}']{e_n\cup f_0,0}{}^{*} ... \xrightarrow[\mathcal{E}'_n]{f_m,0}{}^{*}\ q_n \xrightarrow[\mathcal{E}'_n]{\emptyset,k} ()$;
therefore φ ++ $[e_0; ...; (e_n{\cup}f_0); ...; f_m \models H_2{\cdot}C_2$.
- When $K_1{=}1$, $seq\ p\ q \xrightarrow[\mathcal{E}]{e_0,0}{}^{*} ... \xrightarrow[\mathcal{E}_n]{e_n,0}{}^{*}\ p_n \xrightarrow[\mathcal{E}]{\emptyset,1} q \xrightarrow[\mathcal{E}']{f_0,0}{}^{*} ... \xrightarrow[\mathcal{E}'_n]{f_m,0}{}^{*}\ q_n \xrightarrow[\mathcal{E}'_n]{\emptyset,k} ()$;
therefore φ ++ $[e_0; ...; e_n; f_0; ...; f_n \models H_2{\cdot}C_2$.
- When $K_1{>}1$, $seq\ p\ q \xrightarrow[\mathcal{E}]{e_0,0}{}^{*} ... \xrightarrow[\mathcal{E}_n]{e_n,0}{}^{*}\ p_n \xrightarrow[\mathcal{E}_n]{\emptyset,k} ()$, therefore φ ++ $[e_0; ...; e_n \models H_1{\cdot}C_1$.

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

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

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

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

9. **Parallel:** $\mathcal{E} \vdash \langle h, c, k\rangle\ p||q\ \langle h \cdot \Delta\rangle$ and $\varphi \models H$ where $\mathcal{E} \vdash \langle \epsilon, c, k\rangle\ p\ \langle H_1, C_1, K_1\rangle$
$\mathcal{E} \vdash \langle \epsilon, c, k\rangle\ q\ \langle H_2, C_2, K_2\rangle$ and $\vdash_{pm} \langle H_1, C_1, K_1\rangle||\langle H_2, C_2, K_2\rangle \rightsquigarrow \langle\Delta\rangle$.
-When p and q exit at the same instant: the entailment is proved by [PM-$Unfold$] and [PM-$EqLen$].
-When p exits with an exception, and earlier than q: the entailment is proved by [PM-$Unfold$] and [PM-Cut].
-When p exits earlier than q without any exceptions: the entailment is proved by [PM-$Unfold$] and [PM-$Absorb$].

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

11. **Suspend:** $\mathcal{E} \vdash \langle h, c, k\rangle$ *suspend* $p\ \mathbb{S}\ \langle h{\cdot}\Delta\rangle$ and $\varphi \models H$, where $\mathcal{E} \vdash \langle \epsilon, c, k\rangle\ p\ \langle H, C, K\rangle$ and $\langle\Delta\rangle{=}\aleph^{Suspend(\mathbb{S},C,K)}_{Interleave}(H)$. By semantics rules $[Suspend\text{-}1]$ and $[Suspend\text{-}2]$; and Lemma 3.

□

4.5 Temporal Verification via a TRS

The TRS is inspired by Antimirov and Mosses's algorithm [AM95] but solving the language inclusions between *ASyncEffs*. It is triggered i) prior to module calls for the precondition checking; and ii) at the end of verifying a module for the post condition checking. More specifically, given two effects Φ_1, Φ_2, TRS decides if the inclusion $\Phi_1 \sqsubseteq \Phi_2$ is valid. During the effects rewriting process, the inclusions are in the form of $\Gamma \vdash \Phi_1 \sqsubseteq^{\Phi} \Phi_2$, a shorthand for: $\Gamma \vdash \Phi \cdot \Phi_1 \sqsubseteq \Phi \cdot \Phi_2$.

To prove such inclusions is to check whether all the possible effect traces in the antecedent Φ_1 are legitimately allowed in the possible effects traces from the consequent Φ_2. Γ is the proof context, i.e., effects inclusion hypotheses, Φ is the history effects from the antecedent that have been used to match the effects from the consequent. The inclusion checking is initially invoked with $\Gamma{=}\{\}$, $\Phi{=}\epsilon$.

4.5.1 Auxiliary Functions: Nullable, First and Derivative

Next we provide definitions and implementations of auxiliary functions *Nullable*(δ), *First*(*fst*) and *Derivative*(D) respectively. Intuitively, the Nullable function $\delta(\theta)$ returns a boolean value indicating whether θ contains the empty trace; the First function *fst*(θ) computes a set of possible head instants of θ; and the Derivative function $D_I(\theta)$ computes a next-state effects after eliminating one instant I from the head of current effects θ. Here marks the novel definitions – opposite to the existing ones in [AM95] – using '$=_{\clubsuit}$' in Definitions 6, 7, 9.

Definition 8 (Nullable)**.** Given any effect θ, $\delta(\theta){=}true \Leftrightarrow \epsilon{\in}\theta$, where:

$$\delta(\bot){=}false \qquad \delta(\epsilon){=}true \qquad \delta(I){=}false \qquad \delta(\mathbb{S}?){=}_{\clubsuit}false \qquad \delta(\theta^{\star}){=}true$$

$$\delta(\theta_1{\cdot}\theta_2){=}\delta(\theta_1) \wedge \delta(\theta_2) \qquad \delta(\theta_1{\vee}\theta_2){=}\delta(\theta_1){\vee}\delta(\theta_2) \qquad \delta(\theta_1||\theta_2){=}_{\clubsuit}\delta(\theta_1) \wedge \delta(\theta_2)$$

Definition 9 (First)**.** Let $\mathit{fst}(\theta){:=}\{I \mid (I \cdot \theta') \in [\![\theta]\!]\}$ be the set of head instants derivable from effect θ. ($[\![\theta]\!]$ represents all the traces contained in θ)

$$\mathit{fst}(\bot){=}\{\} \qquad \mathit{fst}(\epsilon){=}\{\} \qquad \mathit{fst}(I){=}\{I\} \qquad \mathit{fst}(\theta_1 \vee \theta_2){=}\mathit{fst}(\theta_1) \cup \mathit{fst}(\theta_2)$$

$$\mathit{fst}(\theta^\star){=}\mathit{fst}(\theta) \qquad \mathit{fst}(\theta_1{\cdot}\theta_2){=}\begin{cases}\mathit{fst}(\theta_1) \cup \mathit{fst}(\theta_2) & \mathit{if}\ \delta(\theta_1){=}\mathit{true} \\ \mathit{fst}(\theta_1) & \mathit{if}\ \delta(\theta_1){=}\mathit{false}\end{cases}$$

$$\mathit{fst}(\mathbb{S}?){=}_{\clubsuit}\{\{\mathbb{S}{\mapsto}\mathit{present}\}\} \qquad \mathit{fst}(\theta_1||\theta_2){=}_{\clubsuit}\{(f_1{+}f_2) \mid f_1 \in \mathit{fst}(\theta_1), f_2 \in \mathit{fst}(\theta_2)\}$$

Definition 10 (Instants Subsumption)**.** Given two instants I and J, we define the subset relation $I{\subseteq}J$ as: the set of present signals in J is a subset of the set of present signals in I, and the set of absent signals in J is a subset of the set of absent signals in I, as in having more constraints refers to a smaller set of satisfying instants. Formally,

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

Definition 11 (Partial Derivative)**.** The partial derivative $D_I(\theta)$ of effects θ w.r.t. an event I computes the effects for the left quotient $I^{-1}[\![\theta]\!]$ [6].

$$D_I(\bot){=}D_I(\epsilon)\bot \quad D_I(J){=}\begin{cases}\epsilon & \mathit{if}\ I{\subseteq}J \\ \bot & \mathit{if}\ I{\not\subseteq}J\end{cases} \quad D_I(\mathbb{S}?){=}_{\clubsuit}\begin{cases}\epsilon & \mathit{if}\ I{\subseteq}\{\mathbb{S}{\mapsto}\mathit{present}\} \\ \mathbb{S}? & \mathit{if}\ I{\not\subseteq}\{\mathbb{S}{\mapsto}\mathit{present}\}\end{cases}$$

$$D_I(\theta^\star){=}D_I(\theta) \cdot \theta^\star \qquad D_I(\theta_1 \cdot \theta_2){=}\begin{cases}D_I(\theta_1) \cdot \theta_2 \vee D_I(\theta_2) & \mathit{if}\ \delta(\theta_1){=}\mathit{true} \\ D_I(\theta_1) \cdot \theta_2 & \mathit{if}\ \delta(\theta_1){=}\mathit{false}\end{cases}$$

$$D_I(\theta_1 \vee \theta_2){=}D_I(\theta_1) \vee D_I(\theta_2) \qquad D_I(\theta_1||\theta_2){=}_{\clubsuit}D_I(\theta_1)||D_I(\theta_2)$$

[6]For example, $\{\mathbf{A}\}^{-1}[\![\{\mathbf{A}\}{\cdot}\{\mathbf{B}\}]\!]{=}[\![\{\mathbf{B}\}]\!]$, and $\{\mathbf{A}\}^{-1}[\![\{\mathbf{A}\}{\vee}\{\mathbf{B}\}]\!]{=}[\![\epsilon{\vee}\bot]\!]$, cf Definition 13.

4.5.2 Rewriting Rules

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

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

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

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

$$\frac{\delta(es_1) \wedge \neg\delta(es_2)}{\Gamma \vdash es_1 \not\sqsubseteq es_2}\;[\mathit{Disprove}] \qquad\qquad \frac{\mathit{fst}(es_1) = \{\}}{\Gamma \vdash es_1 \sqsubseteq es_2}\;[\mathit{Prove}]$$

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 $\mathit{\Gamma}$, 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.

$$\frac{(es_1 \sqsubseteq es_3) \in \Gamma \qquad (es_3 \sqsubseteq es_4) \in \Gamma \qquad (es_4 \sqsubseteq es_2) \in \Gamma}{\Gamma \vdash es_1 \sqsubseteq es_2}\;[\mathit{Reoccur}]$$

4. **Unfolding (Induction).** This is the inductive step of unfolding the inclusions. Firstly, we make use of the auxiliary function *fst* to get a set of instants F, which are all the possible initial instants from the antecedent. Secondly, we obtain a new proof context Γ' by adding the current inclusion, as an inductive hypothesis, into the current proof context Γ. Thirdly, we iterate each element $I \in F$, and compute the partial derivatives (*next-state* effects) of both the antecedent and consequent

w.r.t I. The proof of the original inclusion succeeds if all the derivative inclusions succeeds.

$$\frac{F = fst(es_1) \qquad \Gamma'{=}\Gamma, (es_1 \sqsubseteq es_2) \qquad \forall I \in F.\ (\Gamma' \vdash D_I(es_1) \sqsubseteq D_I(es_2))}{\Gamma \vdash es_1 \sqsubseteq es_2}\ [Unfold]$$

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

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

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

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

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

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

Inference rules in our TRS given in subsection 4.5.2 transform current pairs $\langle S, H\rangle$ to new pairs $\langle S', H'\rangle$. And each rule either increases $|H|$ (Unfolding) or, otherwise, reduces $|S|$ (Axiom, Disprove, Prove), therefore the system is terminating.

□

Theorem 4 (TRS Soundness)**.** *Given an inclusion $\Phi_1 \sqsubseteq \Phi_2$, if the TRS returns $TRUE$ when proving $\Phi_1 \sqsubseteq \Phi_2$, then $\Phi_1 \sqsubseteq \Phi_2$ is valid.*

Proof. For each inclusion checking 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:**

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

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

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

2. **Disprove Rules:**

$$\frac{\delta(es_1) \wedge \neg\delta(es_2)}{\Gamma \vdash es_1 \not\sqsubseteq es_2}\ [\mathit{Disprove}] \qquad\qquad \frac{\mathit{fst}(es_1) = \{\}}{\Gamma \vdash es_1 \sqsubseteq es_2}\ [\mathit{Prove}]$$

- It's straightforward to prove soundness of the rule [*Disprove*], Given that θ_1 is nullable, while θ_2 is not nullable, thus clearly the antecedent contains more traces than the consequent. Therefore, these entailments are evidently invalid.

3. **Prove Rules:**

$$\frac{(es_1 \sqsubseteq es_3) \in \Gamma \qquad (es_3 \sqsubseteq es_4) \in \Gamma \qquad (es_4 \sqsubseteq es_2) \in \Gamma}{\Gamma \vdash es_1 \sqsubseteq es_2}\ [\mathit{Reoccur}]$$

- For the rule [*Prove*], we consider an arbitrary model, φ such that: $\varphi \models \theta_1$. Given the side conditions from the promises, we get $\varphi \models \theta_1$. When the *fst* set of θ_1 is empty, θ_1 is possible $\bot$ or ϵ; and θ_2 is nullable. For both cases, the inclusion is proved.

- For the rule [*Reoccur*], we consider an arbitrary model, φ such that: $\varphi \models \theta_1$. Given the promises that $\theta_1 \sqsubseteq \theta_3$, we get $\varphi \models \theta_3$; Given the premise that there exists a hypothesis $\theta_3 \sqsubseteq \theta_4$, we get $\varphi \models \theta_4$; Given the promises that $\theta_4 \sqsubseteq \theta_2$, we get $\varphi \models \theta_2$. Therefore, the inclusion is proved.

4. **Inductive Unfolding Rule:**

$$\frac{F = \mathit{fst}(es_1) \qquad \Gamma'{=}\Gamma, (es_1 \sqsubseteq es_2) \qquad \forall I \in F.\ (\Gamma' \vdash D_I(es_1) \sqsubseteq D_I(es_2))}{\Gamma \vdash es_1 \sqsubseteq es_2}\ [\mathit{Unfold}]$$

- For the rule [$Unfold$], we consider an arbitrary model, φ_1 and φ_2 such that: $\varphi_1 \models \theta_1$ and $\varphi_2 \models \theta_2$. For an arbitrary instant I, let $\varphi_1{}' \models I^{-1}[\![\theta_1]\!]$; and $\varphi_2{}' \models I^{-1}[\![\theta_2]\!]$.

Case 1), $I{\notin}F$, $\varphi_1{}' \models \bot$, thus automatically $\varphi_1{}' \models D_I(\theta_2)$;

Case 2), $I{\in}F$, given that inclusions in the rule's premise is proved , then $\varphi_1{}' \models D_I(\theta_2)$.

By Definition 13, since for all I, $D_I(\theta_1) \sqsubseteq D_I(\theta_2)$, the conclusion is proved.

All the entailing checking rules used in the TRS are sound, therefore the TRS is sound. □

4.5.3 Discussion: highlighting the novelty.

Departing from the original Antimirov algorithm [AM95], this work devises extended definitions for the auxiliary functions: *Nullable*(δ), *First*(*fst*) and *Derivative*(D). These definitions cover extended constructs in the more expressive specifications formulae, *ASyncEffs*, which contain operators for the (synchronous) parallel composition and asynchronous waiting. The comprehensive rewriting system serves as a back-end engine for the finer-grained verification system for synchronous programming languages, which cannot be trivially achieved by the original rewriting system.

4.6 Demonstration Examples

We now highlight our main methodologies, using the example shown in Figure 4.9 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.

4.6.1 Esterel and *ASyncEffs*

Specifications are annotated in `/*@ ... @*/` for each module, which lead to a compositional verification strategy, where static checking and temporal verification can be done locally. *ASyncEffs* uses curly braces {} to enclose time instants (reactions). A time instant is a set of signals (possibly empty) with status, happening at the same reaction. The module `Read` asynchronously loads a file.

At line 4, the statement `async` is enriched with a completion signal, here *loaded*. When started, `async` immediately emits *loading*, and calls $fs.readFile$, that is expected to take time in term of reactions, i.e., not to complete during the current reaction. `async` blocks its local control thread but does not block other parallel branches. Therefore at line 7, the program can do other computations, i.e., $compOther$, while *loading* the file. At line 8, the program waits for the signal *loaded* to be emitted. Then, it does data processing via emitting $logData$ and waits for the environment to *close* the file.

```
module Read (in open, close,
             out loading, loaded, compOther, logData)
/*@ requires {}^*.{open} @*/
/*@ ensures {loading,compOther}.{loaded}.{logData}.close? @*/
{ async loaded { //'loaded' is emitted after the body is completed
        emit loading;
        fs.readFile(open.value);}
  emit compOther;  //do things that do not depend on the file
  await loaded;   //await for the signal 'loaded' to be emitted
  emit logData;  //data processing and logging
  await close; }

module Main (out open, close, loading, loaded, compOther, logData)
/*@ requires {} @*/
/*@ ensures {open}.{}^*.{close} @*/ {
  emit open("filePath");
  fork{ Read (close, loading, loaded, compOther, logData); }
  par { await logData; //await for the signal 'logData'
            emit close("filePath");  }}   //close the file
```

Figure 4.9: Asynchronously reading a file, using async/await.

Module `Read`'s precondition $\{\}^\star\cdot\{open\}$ requires that when `Read` is called, the signal *open* should be emitted in the current reaction, indicating that the file is opened prior to the current function call. Module `Main` firstly `open`s the file, then creates two sub-threads via statement `fork{...}par{...}`. One thread calls `Read`, while another threads waits for the data processing to be done and `close`s the file. Note that *ASyncEffs* is an *affine logic* in the sense that it only describes the signals we care about, regardless of the non-mentioned signals.

4.6.2 Forward Verification

Figure 4.10 and Figure 4.11, demonstrate the forward verification process of the modules defined in Figure 4.9. Program effect states are captured in the form of $\langle\Phi\rangle$. To facilitate this illustration, we label the verification steps by (1), ..., (17),

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

(1) $\langle\{open\}\rangle$ *(– initialize the current effects using precondition's last instant –)*
```
    async loaded {
        emit loading; fs.readFile(open.value);
```

(2) $\langle\{open, loading\}\rangle$ [FV-Emit]
```
    }
```
$\langle\{open, loading\} \cdot \{loaded\}\rangle$ [FV-Async-Branch-1]

(3) $\langle\{open\}\rangle$ *(– inherited from state (1) –)*
```
    emit compOther;
```

(4) $\langle\{open, compOther\}\rangle$ [FV-Emit]
```
    await loaded;
```

(5) $\langle\{open, compOther\} \cdot loaded?\rangle$ [FV-Await]
```
    emit logData;
```

(6) $\langle\{open, compOther\} \cdot loaded? \cdot \{logData\}\rangle$ [FV-Emit]
```
    await close; }
```

(7) $\langle\{open, compOther\}\cdot loaded?\cdot\{logData\}\cdot close?\rangle$ [FV-Await][FV-Async-Branch-2]

(8) $\langle(\{loading\} \cdot \{loaded\})||(\{compOther\} \cdot loaded? \cdot \{logData\} \cdot close?)\rangle$
$\Phi_{final}=\langle\{loading, compOther\} \cdot \{loaded\} \cdot \{logData\} \cdot close?\rangle$
[FV-Async] [Effects-Parallel-Merge]

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

Figure 4.10: A demonstration of the forward verification for the module *Read*.

and mark the deployed forward reasoning rules (cf. section 4.4) in [gray].

To start the verification, states (1)(10) are initialized from the preconditions. States (2)(4)(6)(11)(15) are obtained by [FV-Emit], which adds the emitted signal to the current instant. States (5)(7)(14) are obtained by [FV-Await], which concatenates a blocking signal (with a question mark) to the current effects. States (3)(13) start the reasoning of the second threads. Steps (8)(16) parallel compose the effects from both of the branches, and normalize the final effects. Before each function call, the verifier invokes the TRS to check whether the current effect state satisfies the precondition of the callee, cf step (12). After these states transformations, steps (9)(17) invoke the TRS to check the postcondition.

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

(10) $\langle\{\}\rangle$ *(– initialize the current effects using precondition's last instant –)*

```
    emit open("filePath");
```

(11) $\langle\{open\}\rangle$ [FV-Emit]

```
    fork{ Read (close, loading, loaded, compOther, logData); }
```

(12) $\{\}^\star \cdot \{open\} \sqsubseteq \Phi^{Read}_{pre}$ [FV-Call] *(-TRS:Read's precondition is satisfied-)*
$\langle\{open, loading, compOther\} \cdot \{loaded\} \cdot \{logData\} \cdot close?\rangle$ [FV-Call]

(13) $\langle\{open\}\rangle$ *(– inherited from state (11) –)*

```
    par   { await logData;
```

(14) $\langle\{open\} \cdot logData?\rangle$ [FV-Await]

```
            emit close("filePath"); }}
```

(15) $\langle\{open\} \cdot logData? \cdot \{close\}\rangle$ [FV-Emit]

(16) $\langle(\{open, loading, compOther\} \cdot \{loaded\} \cdot \{logData\} \cdot close?)$
$||(\{open\} \cdot logData? \cdot \{close\})\rangle$ [FV-Fork-Par]
$\langle\{open, loading, compOther\} \cdot \{loaded\} \cdot \{logData\} \cdot \{close\}\rangle$[Effects-Parallel-Merge]

(17) *(-TRS: check the postcondition of module Main; Succeed. -)*
$\{open, loading, compOther\} \cdot \{loaded\} \cdot \{logData\} \cdot \{close\} \sqsubseteq \{open\} \cdot \{\}^\star \cdot \{close\}$

Figure 4.11: A demonstration of the forward verification for the module *Main*.

Detecting Unreachable Promises. We here show that our effects and the parallel merging can capture the *anti-pattern* [Ali+18] caused by broken chain of the interdependent promises. Given the program behavior expressed in *ASyncEffs* (defined in Figure 4.6), and the parallel merge algorithm (defined in subsection 4.4.1) eliminating the parallel operator $||$, we can easily capture the *unreachable promises* [Ali+18] caused by a broken chain of interdependent promises. For example, the parallel composition in step (16) leads to the final trace which is *well-synchronized* for all the signals. Whereas in step (8), the final trace contains a dangling waiting for the signal `close` because there is no locally emitted `close`.

Definition 12 (Well-Synchronized Effects)**.** After parallel merging, we call effects without any blocking signals well-synchronized effects. Given any effect θ, $well(\theta)$

returns a Boolean value, defined recursively:

$$well(\bot){=}well(\epsilon){=}well(I) = \mathit{false} \qquad well(\mathbb{S}?){=}\mathit{true} \qquad well(\theta^{\star}){=}well(\theta)$$
$$well(\theta_1 \cdot \theta_2){=}well(\theta_1) \vee well(\theta_2) \qquad well(\theta_1 \vee \theta_2){=}well(\theta_1) \vee well(\theta_2)$$

Definition 12 defines *well-synchronized effects*. Any *poorly-synchronized effects* indicates that there exist registered reactions for unreachable promises. However, poorly-synchronized effects can be further parallel composed to other thread, and become well-synchronized. For example, the final effects of module `Main` is well-synchronized after parallel composing module `Read` with another thread.

4.6.3 The TRS

Having the *ASyncEffs* as the logic, we are interested in the following verification problem: Given a program $\mathcal{P}$, and a temporal property Φ', does $\Phi^{\mathcal{P}} \sqsubseteq \Phi'$ hold? In a typical verification context, checking the inclusion/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'.

Here, we deploy a purely algebraic term rewriting system (TRS), to check language inclusions between *ASyncEffs*. Our TRS is an extension of Antimirov and Mosses's algorithm [AM95], whose rewriting system decides inequalities of regular expressions (REs) through an iterated process of checking the inequalities of their *partial derivatives* [Ant95]. There are two basic rules: [*Disprove*], which infers false from trivially inconsistent inequalities; and [*Unfold*], which applies Theorem 1 to generate new inequalities.

Definition 13 (*ASyncEffs* Inclusion)**.** Given Σ is a finite set of alphabet, for two *ASyncEffs* Φ_1, Φ_2, their inclusion is defined as: $\Phi_1 \sqsubseteq \Phi_2 \Leftrightarrow \forall I.\ I^{-1}(\Phi_1) \sqsubseteq I^{-1}(\Phi_2)$.

Similar to Theorem 1, we defined Definition 13 for unfolding the inclusions between *ASyncEffs*, where $I^{-1}(\Phi)$ is the partial derivative of Φ w.r.t the instant I. Termination of the rewriting is guaranteed because the set of derivatives to be considered is finite, and possible cycles are detected using *memorization* [Bro05a].

Next, we continue with the step (9) in Figure 4.10, to demonstrate how the TRS handles *ASyncEffs*. Table 4.3 automatically proves (at step ④) that the inferred

Table 4.3: The inclusion proving example for *ASyncEffs*.

$$\epsilon \sqsubseteq \epsilon \quad ④[\text{Prove}]$$

$$\cancel{\{\mathbf{Close}\}} \sqsubseteq \cancel{\{\mathbf{Close}\}} \quad ③[\mathit{Unfold}]$$

$$\cancel{\{\mathbf{Loaded}, \mathbf{LogData}\}} \cdot \{\mathbf{Close}\} \sqsubseteq \cancel{\{\mathbf{LogData}\}} \cdot \{\mathbf{Close}\} \quad ②[\mathit{Unfold}]$$

$$\cancel{\{\mathbf{Loading}, \mathbf{CompOther}\}} \cdot \{\mathbf{Loaded}, \mathbf{LogData}\} \cdot \{\mathbf{Close}\} \sqsubseteq \cancel{\{\mathbf{CompOther}\}} \cdot \{\mathbf{LogData}\} \cdot \{\mathbf{Close}\} \quad ①[\mathit{Unfold}]$$

$$\{\mathbf{Loading}, \mathbf{CompOther}\} \cdot \{\mathbf{Loaded}, \mathbf{LogData}\} \cdot \{\mathbf{Close}\} \sqsubseteq \{\mathbf{CompOther}\} \cdot \{\mathbf{LogData}\} \cdot \{\mathbf{Close}\}$$

effects of module `read` satisfy the declared postcondition (after several times of unfolding at steps ①② and ③). The rewriting rules (cf. subsection 4.5.2) are marked in [gray].

Note that event {Loading, CompOther} entails (or is subsumed by) event {CompOther} because the former contains more constraints. We formally define the subsumption for events in Definition 10. Our TRS is an extension of Antimirov and Mosses's algorithm [AM95], whose rewriting system decides inequalities of regular expressions (REs) through an iterated process of checking the inequalities of their *partial derivatives* [Ant95]. There are two basic rules: [*Disprove*], which infers false from trivially inconsistent inequalities; and [*Unfold*], which applies Theorem 1 to generate new inequalities.

4.7 Implementation and Evaluation

To show the feasibility of our approach, we prototype our automated verification system using OCaml; prove soundness for both the forward verifier and the TRS; validate and evaluate the implementation using a microbenchmark [Son22a] [7].

This experiment is done without a baseline comparison because there are no existing tools for encoding logical-correctness/constructiveness analysis using tem-

[7] The benchmark is constructed by manually annotating *ASyncEffs* specifications, including both succeeded and failed cases. The validation tests are synthetic examples to test the main contributions, including the preemption interleaving computation and the inclusion checking for the parallel composition and the waiting operator.

poral verification, and our experimental results show that a modular and efficient temporal verification for synchronous languages is achievable.

Table 4.4: Experimental Results.

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

Table 4.4 presents the evaluation results. We select 16 programs, varying from 15 lines to 300 lines, and annotate *ASyncEffs* specifications with a 1:1 ratio for succeeded/failed cases. The results record: **No.** for the index of the program; **LOC** for lines of code; **Forward(ms)** for forward reasoning time; **#Prop(✓)**for the number of valid properties; **Avg-Prove(ms)**for the average proving time for the valid properties; **#Prop(✗)**for the number of invalid properties; and **Avg-Dis(ms)**for the average disproving time for the invalid properties. Times are counted using milliseconds, and the experiment is done on a MacBook Pro with a 2.6 GHz 6-Core Intel Core i7 processor.

Discussion: Generally, the forward reasoning time increases with a linear complexity. We notice that the disproving times for invalid properties are constantly low. This finding echoes the insights from prior TRS-based works [SC20; AM95;

AMR09; KT14a; Hov12], which suggest that TRS is a better average-case algorithm than those based on the comparison of automata. That is because *it only constructs automata as far as it needs*, which makes it more efficient when disproving incorrect specifications, as we can disprove it earlier without constructing the whole automata. In other words, the more incorrect specifications are, the more efficient our solver is.

Our proposed effect logic and the abstract semantics for the full-featured Esterel not only tightly capture the behaviors of a preemptive asynchronous execution models but also help to mitigate the programming challenges in both worlds. Meanwhile, the expressive *ASyncEffs* enable compositional temporal verification at the source level, which is not yet supported by existing techniques.

4.7.1 Case Studies

In this section, we investigate how *ASyncEffs* can help with issues related to both synchronous and asynchronous programs. Then, we demonstrate the flexibility/expressiveness of *ASyncEffs*.

4.7.1.1 Detecting Logically Incorrect Programs.

In synchronous programming, a program is logically correct if it has precisely one safe trace for each input assignment.

This work effectively checks logical correctness. 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. 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 event as follows:

Definition 14 (Contradictory event)**.** Given any event I, it is contradictory is $\exists S.\ (S \mapsto \mathit{absent}) \in I\ \mathit{and}\ (S \mapsto \mathit{present}) \in I$ or $\exists S.\ (S \mapsto \mathit{undef}) \in I\ \mathit{and}\ (S \mapsto \mathit{present}) \in I$.

1. $\textit{present } S1$ $\langle\{S1 \mapsto undef\}\rangle$	1. $\textit{present } S1$ $\langle\{S1 \mapsto undef\}\rangle$
2. $\quad then$ $\langle\{S1 \mapsto undef, S1 \mapsto present\}\rangle$	2. $\quad then$ $\langle\{S1 \mapsto undef, S1 \mapsto present\}\rangle$
3. $\qquad nothing$ $\langle\{S1 \mapsto undef, S1 \mapsto present\}\rangle$	3. $\qquad \textit{emit } S1$ $\langle\{S1 \mapsto present, S1 \mapsto present\}\rangle$
4. $\quad else$ $\langle\{S1 \mapsto undef, S1 \mapsto absent\}\rangle$	4. $\quad else$ $\langle\{S1 \mapsto undef, S1 \mapsto absent\}\rangle$
5. $\qquad \textit{emit } S1$ $\langle\{S1 \mapsto present, S1 \mapsto absent\}\rangle$	5. $\qquad nothing$ $\langle\{S1 \mapsto undef, S1 \mapsto absent\}\rangle$
6. $\langle\{S1 \mapsto undef, S1 \mapsto present\}$ $\quad \vee \{S1 \mapsto present, S1 \mapsto absent\}\rangle$ $\langle\bot \vee \bot\rangle \Rightarrow \langle\bot\rangle$	6. $\langle\{S1 \mapsto present, S1 \mapsto present\}$ $\quad \vee \{S1 \mapsto undef, S1 \mapsto absent\}\rangle$ $\langle\{S1 \mapsto present\} \vee \{S1 \mapsto absent\}\rangle$
(a)	(b)

Table 4.5: Logically incorrect examples, caught by *ASyncEffs*.

As shown in Table 4.5 (a) and (b) are both logically incorrect, because there are no valid assignments of signal S1 and there are two possible assignments of signal S1, respectively.

4.7.1.2 A Strange Logically Correct Program.

This example for synchronous languages shows that composing programs can lead to counter-intuitive phenomena.

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

Figure 4.12: A Strange Logically Correct Esterel Program.

As the program shows in Figure 4.12, the first parallel branch is the logically incorrect program Table 4.5 (b), while the second branch contains a non-reactive program enclosed in "present S1" statement. Surprisingly, this program is logically correct, since there is only one logically coherent assumption: 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 effect logic is able to soundly detect above mentioned correctness checking.

4.7.1.3 Rewriting with the Blocking Waiting Operator.

Formally, we define "waiting for the signal $\mathbf{A}$" as: $\mathbf{A}? \equiv \exists n, n{\geq}0 \wedge \{\overline{\mathbf{A}}\}^n \cdot \{\mathbf{A}\}$, where$\{\overline{\mathbf{A}}\}$ refers to all the events containing $\mathbf{A}$ to be absent.

Table 4.6: The example for Await.

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

Table 4.6 shows the proof of $\{\mathbf{A}\} \cdot \{\mathbf{C}\} \cdot \mathbf{B}? \cdot \{\mathbf{D}\}$ entailing $\{\mathbf{A}\} \cdot \mathbf{B}? \cdot \{\mathbf{D}\}$, as intuitively $\{\mathbf{C}\} \cdot \mathbf{B}?$ is a special case of $\mathbf{B}?$. In step ① and ②, $\{\mathbf{A}\}$ is eliminated. In step ③, $\mathbf{B}?$ is normalized into $\{\mathbf{B}\} \vee (\{\overline{\mathbf{B}}\} \cdot \mathbf{B}?)$, By the step of ④, $\{\mathbf{C}\}$ is eliminated together with $\{\overline{\mathbf{B}}\}$ because $\{\mathbf{C}\} \subseteq \{\overline{\mathbf{B}}\}$. Now the rest part is $\mathbf{B}? \cdot \{\mathbf{D}\} \sqsubseteq \mathbf{B}? \cdot \{\mathbf{D}\}$. Here, we further normalize $\mathbf{B}?$ from the LHS into a disjunction, leading to two proof sub-trees. From the first sub-tree, we keep unfolding the inclusion with $\{\mathbf{B}\}$ (⑤) and $\{\mathbf{D}\}$ (⑥) until we can prove it. Continue with the second sub-tree, we unfold it with $\{\overline{\mathbf{B}}\}$; then in step ⑦ we observe the proposition is isomorphic with one of the the previous step, marked with $(\ddagger)$. We prove it and finish the writing process.

4.7.1.4 Broken Promises Chain.

As the prior works [MLT17; Ali+18] 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. We here show that our algebraic effects can capture not well-synchronized (formally defined in Definition 12) traces during the parallel merging process presented in subsection 4.4.1.

For example, the parallel composition of traces: $\{\mathbf{A}\}\cdot\{\mathbf{B}\}\cdot\{\mathbf{C}\}\cdot\{\mathbf{D}\} \mid\mid \{\mathbf{E}\}\cdot\mathbf{C}?\cdot\{\mathbf{F}\}$ leads to the final behavior of $\{\mathbf{A}, \mathbf{E}\}\cdot\{\mathbf{B}\}\cdot\{\mathbf{C}\}\cdot\{\mathbf{D}, \mathbf{F}\}$, which is well-synchronized for all the events. However, if we were composing traces: $\{\mathbf{A}\}\cdot\{\mathbf{B}\}\cdot\{\mathbf{D}\} \mid\mid \{\mathbf{E}\}\cdot\mathbf{C}?\cdot\{\mathbf{F}\}$ due to the reasons that forgetting to emit $\mathbf{C}$ (In JavaScript, it could be the case that forgetting to explicitly return a promise result.), it leads to a problematic trace $\{\mathbf{A}, \mathbf{E}\}\cdot\{\mathbf{B}\}\cdot\{\mathbf{D}\}\cdot\mathbf{C}?\cdot\{\mathbf{F}\}$. The final effects contain a dangling signal waiting of $\mathbf{C}$, which indicates the corresponding anti-pattern.

4.8 Summary

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

Chapter 5

Symbolic Timed Effects (*TimEffs*)

The correctness of real-time systems depends both on the correct functionalities and the realtime constraints. To go beyond the existing Timed Automata based techniques, we propose a novel solution that integrates a modular Hoare-style forward verifier with a term rewriting system (TRS) on *Timed Effects* (*TimEffs*). The main purposes are to: increase the expressiveness, dynamically manipulate clocks, and efficiently solve clock constraints.

We formally define a core language C^t, generalizing the real-time systems, modeled using mutable variables and timed behavioral patterns, such as *delay*, *timeout*, *interrupt*, *deadline*. Secondly, to capture real-time specifications, we introduce *TimEffs*, a new effect logic, that extends *regular expressions* with dependent values and arithmetic constraints. Thirdly, the forward verifier reasons temporal behaviors – expressed in *TimEffs* – of target C^t programs. Lastly, we present a purely algebraic TRS, i.e., an extended *Antimirov algorithm*, to efficiently check language inclusions between *TimEffs*.

To demonstrate the feasibility of our proposal, we prototype the verification system; prove its soundness; report on case studies and experimental results.

5.1 Introduction

During the last three decades, there has been a popular approach based on Timed Automata (TAs) [AD94] for specifying real-time systems. TAs are powerful in designing real-time models via explicit clocks, where real-time constraints are captured by explicitly setting/resetting clock variables. A number of automatic verification tools for TAs have been proven to be successful [Wan+17; LPY97; Yov97; WWH05]. Although TAs' simple structure made it feasible to deploy efficient model checking, specifying and verifying compositional real-time systems have become increasingly challenging due to the increasing complexity. Industrial case studies show that requirements for real-time systems are often structured into phases, which are then composed sequentially, in parallel, alternatively [Hav+97; Lar+05]. TAs lack high-level compositional patterns for hierarchical design; moreover, users often need to manipulate clock variables with carefully calculated clock constraints manually. The process is tedious and error-prone.

There have been some translation-based approaches on building verification support for compositional timed-process representations. For example, Timed Communicating Sequential Process (TCSP), Timed Communicating Object-Z (TCOZ) and *Statechart* based hierarchical Timed Automata are well suited for presenting compositional models of complex real-time systems. Prior works [Don+08; DM01] systematically translate TCSP/TCOZ/Statechart models to flat TAs so that the model checker Uppaal [LPY97] can be applied.

In this work, we investigate an alternative approach for verifying real-time systems. We propose a novel temporal specification language, Timed Effects (*TimEffs*), which enables a compositional verification via a Hoare-style forward verifier and a term rewriting system (TRS). More specifically, we specify system behaviors in the form of *TimEffs*, which integrates the Kleene Algebra with dependent values and arithmetic constraints, to provide real-time abstractions into traditional linear temporal logics. For example, one safety property, *"The event **Done** will be triggered no later than one time unit"*[1], is expressed in *TimEffs* as: $\Phi \triangleq \texttt{0}{\leq}\texttt{t<1} \wedge \texttt{(\_}^{\star}\cdot\ \textbf{Done}\ \texttt{)\#t}$. Here $\wedge$ connects the arithmetic formula and the timed trace; the operator # binds time variables to traces (here t is a time bound of $(\_^{\star}\cdot\textbf{Done})$); _ is a wildcard matching to any event; Kleene star $\star$ denotes a trace repetition. The above formula Φ corresponds to '$\Diamond_{[0,1)}\textbf{Done}$' in metric temporal logic (MTL), reads *"within one time unit, **Done** finally happens"*. Furthermore, the time bounds can be dependent on the program inputs, as shown in Figure 5.1.

[1]Here, we pretend time is discrete and only integral values. However, it's just as easy to represent continuous time by letting time variables assume real values [Lam05].

```
void addOneSugar()
/* req:  true ∧ _*
  ens:  t>1 ∧ ϵ # t */
{
  timeout ((), 1);
}
```

```
void addNSugar (int n)
/* req:  true ∧ _*
  ens:  t≥n ∧ EndSugar # t */
{ if (n == 0) { event ["EndSugar"];}
  else {
    addOneSugar();
    addNSugar (n-1);}}
```

Figure 5.1: Value-dependent specification in *TimEffs*.

Function `addNSugar` takes a parameter `n`, representing the portion of the sugar we need to add. When `n` equals to `0`, it simply raises an event **EndSugar** to mark the end of the process. Otherwise, it adds one portion of the sugar by calling `addOneSugar()`, then recursively calls `addNSugar` with parameter `n-1`. The use of statement `timeout(e, d)` is standard [Ltd22], which executes a block of code `e` after the specified time `d`. Therefore, the time spent on adding one portion of the sugar is more than one time unit. Note that ϵ`#t` refers to an empty trace which takes time `t`. Both precondition require no arithmetic constraints, and have no temporal constraints upon the history traces. The postcondition of `addNSugar(n)` indicates that the function generates a finite trace where **EndSugar** takes a no less than `n` time-units delay to finish.

Although these examples are simple, they show the benefits of deploying value-dependent time bounds, which is beyond the capability of TAs. Essentially, *TimEffs* define symbolic TAs, which stand for a set (possibly infinite) of concrete transition systems. Moreover, we deploy a Hoare-style forward verifier to soundly reason about the behaviors from the source level, with respect to the well-defined operational semantics. This approach provides a *direct* (opposite to the techniques which require manual and remote modeling processes), and modular verification – where modules can be replaced by their already verified properties – for real-time systems, which are not possible by any existing techniques. Furthermore, we develop a novel TRS, which is inspired by Antimirov and Mosses's algorithm[2] [AM95] but solving the language inclusions between more expressive *TimEffs*. In short, the main contributions of this work are:

1. **Language Abstraction:** we formally define a core language C^t, by defining its syntax and operational semantics, generalizing the real-time systems with mutable variables and timed behavioral patterns, e.g., *delay*, *timeout*, *deadline*.

2. **Novel Specification:** we propose *TimEffs*, by defining its syntax and semantics, gaining the expressive power beyond traditional linear temporal logics.

3. **Forward Verifier:** we establish a sound effect system to reason about temporal behaviors of given programs. The verifier triggers the back-end solver TRS.

4. **Efficient TRS:** we present the rewriting rules to (dis)prove the inclusion relations between the actual behaviors and the given specifications, both in *TimEffs*.

5. **Implementation and Evaluation:** we prototype the automated verification system, prove its soundness, report on case studies and experimental results.

5.2 Language and Specifications

This section first introduces the target language and then depict the temporal specification language which supports *TimEffs*.

5.2.1 The Target Language

We define the core language C^t in Figure 5.2, which is built based on C syntax and provides support for timed behavioral patterns via implicit clocks.

Here, c and b stand for integer and Boolean constants, mn and x are meta-variables, drawn from **var** (the countably infinite set of arbitrary distinct identifiers). A program $\mathcal{P}$ comprises a list of global variable initializations α^* and a list of

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

$$
\begin{array}{lrcl}
(\textit{Program}) & \mathcal{P} & ::= & (\alpha^*, \textit{fun}^*) \\
(\textit{Types}) & \iota & ::= & \textit{int} \mid \textit{bool} \mid \textit{unit} \\
(\textit{Function}) & \textit{fun} & ::= & \iota\ mn\ (\iota\ x)^*\ \{\textbf{req}\ \Phi_{pre}\ \textbf{ens}\ \Phi_{post}\}\ \{e\} \\
(\textit{Values}) & v & ::= & () \mid c \mid b \mid x \\
(\textit{Assignment}) & \alpha & ::= & x := v \\
(\textit{Expressions}) & e & ::= & v \mid \alpha \mid [v]e \mid mn(v^*) \mid e_1; e_2 \mid e_1 || e_2 \mid \textit{if}\ v\ e_1\ e_2 \\
 & & & \mid \texttt{event}[\mathbf{A}(v, \alpha^*)] \mid \texttt{delay}[v] \mid e_1\ \texttt{timeout}[v]\ e_2 \\
 & & & \mid e\ \texttt{deadline}[v] \mid e_1\ \texttt{interrupt}[v]\ e_2 \\
(\textit{Terms}) & t & ::= & c \mid x \mid t_1 + t_2 \mid t_1 - t_2
\end{array}
$$

$c \in \mathbb{Z}$ $\quad b \in \mathbb{B}$ $\quad mn, x \in \textbf{var}$ $\quad$ (*Action labels*) $\mathbf{A} \in \Sigma$

Figure 5.2: A core first-order imperative language with timed constructs.

function declarations *fun**. Here, we use the $*$ superscript to denote a finite list of items, for example, x^* refers to a list of variables, $x_1, ..., x_n$. Each function *fun* has a name mn, an expression-oriented body e, also is associated with a precondition Φ_{pre} and a postcondition Φ_{post} (specification syntax is given in Figure 5.3). C^t allows each iterative loop to be optimized to an equivalent tail-recursive function, where the mutation on parameters is made visible to the caller.

Expressions comprise: values v; guarded processes $[v]e$, where if v is true, it behaves as e, else it idles until v becomes true; function calls $mn(v^*)$; sequential composition $e_1; e_2$; parallel composition $e_1 || e_2$, where e_1 and e_2 may communicate via shared variables; conditionals *if* $v\ e_1\ e_2$; and event raising expressions $\texttt{event}[\mathbf{A}(v, \alpha^*)]$ where the event $\mathbf{A}$ comes from the finite set of event labels Σ. Without loss of generality, events can be further parameterized with one value v and a set of assignments α^* to update the mutable variables. Moreover, a number of timed constructs can be used to capture common real-time system behaviors, which are explained via operational semantics rules in subsection 5.2.2.

5.2.2 Operational Semantics of C^t

To build the semantics of the system model, we define the notion of a configuration in Definition 15, to capture the global system state during system execution.

Definition 15 (System configuration)**.** A system configuration ζ is a pair $(\mathcal{E}, e)$ where $\mathcal{E}$ is a variable valuation function (or a stack) and e is an expression.

A transition of the system is of the form $\zeta \xrightarrow{l} \zeta'$ where ζ and ζ' are the system configurations before and after the transition respectively. Transition labels l include: d, denoting a non-negative integer; τ, denoting an invisible event; $\mathbf{A}$, denoting an observable event. For example, $\zeta \xrightarrow{\mathrm{d}} \zeta'$ denotes a d time-units elapse. Next, we present the firing rules, associated with timed constructs.

Process $\texttt{delay}[v]$ idles for exactly t time units. Rule $[delay_1]$ states that the process may idle for any amount of time given it is less than or equal to t; Rule $[delay_2]$ states that the process terminates immediately when t becomes 0.

$$\frac{\mathrm{d}{\leq}v}{(\mathcal{E}, \texttt{delay}[v]) \xrightarrow{\mathrm{d}} (\mathcal{E}, \texttt{delay}[v\text{-d}])}\,[delay_1] \qquad \frac{}{(\mathcal{E}, \texttt{delay}[0]) \xrightarrow{\tau} (\mathcal{E}, ())}\,[delay_2]$$

In e_1 $\texttt{timeout}[v]$ e_2, the first observable event of e_1 shall occur before t time units; otherwise, e_2 takes over the control after exactly t time units. Note that the usage of timeout in Figure 5.1 is a special case where e_1 never starts by default.

$$\frac{(\mathcal{E}, e_1) \xrightarrow{\mathbf{A}} (\mathcal{E}', e_1')}{(\mathcal{E}, e_1\ \texttt{timeout}[v] e_2) \xrightarrow{\mathbf{A}} (\mathcal{E}', e_1')}\,[to_1] \qquad \frac{(\mathcal{E}, e_1) \xrightarrow{\tau} (\mathcal{E}', e_1')}{(\mathcal{E}, e_1\ \texttt{timeout}[v] e_2) \xrightarrow{\tau} (\mathcal{E}', e_1'\ \texttt{timeout}[v] e_2)}\,[to_2]$$

$$\frac{(\mathcal{E}, e_1) \xrightarrow{\mathrm{d}} (\mathcal{E}, e_1') \qquad (\mathrm{d}{\leq}v)}{(\mathcal{E}, e_1\ \texttt{timeout}[v]\ e_2) \xrightarrow{\mathrm{d}} (\mathcal{E}, e_1'\ \texttt{timeout}[v\text{-d}] e_2)}\,[to_3] \qquad \frac{}{(\mathcal{E}, e_1\ \texttt{timeout}[0] e_2) \xrightarrow{\tau} (\mathcal{E}, e_2)}\,[to_4]$$

Process $\texttt{deadline}$ $[v]$ e behaves exactly as e except that it must terminate before t time units. The guarded process $[v]e$ behaves as e when v is true, otherwise it idles until v becomes true. Process e_1 $\texttt{interrupt}[v]$ e_2 behaves as e_1 until t time units, and then e_2 takes over. We leave the rest rules in section B.1.

$$\frac{(\mathcal{E}, e) \xrightarrow{\mathbf{A}/\tau} (\mathcal{E}', e')}{(\mathcal{E}, \texttt{deadline}[v]\ e) \xrightarrow{\mathbf{A}/\tau} (\mathcal{E}', \texttt{deadline}[v]\ e')}\,[ddl_1] \qquad \frac{(\mathcal{E}, e) \xrightarrow{l} (\mathcal{E}', v)}{(\mathcal{E}, \texttt{deadline}[v]\ e) \xrightarrow{l} (\mathcal{E}', v)}\,[ddl_2]$$

$$\frac{\mathcal{E} \models (v{=}true)}{(\mathcal{E}, [v]e) \xrightarrow{\tau} (\mathcal{E}, e)}[gu_1] \qquad \frac{(\mathcal{E}, e) \xrightarrow{\mathrm{d}} (\mathcal{E}, e') \quad (\mathrm{d}{\le}v)}{(\mathcal{E}, \texttt{deadline}[v]\ e) \xrightarrow{\mathrm{d}} (\mathcal{E}, \texttt{deadline}[v\text{-d}]\ e')}[ddl_3]$$

$$\frac{\mathcal{E} \not\models (v{=}true)}{(\mathcal{E}, [v]e) \xrightarrow{\tau} (\mathcal{E}, [v]e)}[gu_2] \quad \frac{(\mathcal{E}, e_1) \xrightarrow{\mathbf{A}/\tau} (\mathcal{E}', e_1')}{(\mathcal{E}, e_1\ \texttt{interrupt}[v]\ e_2) \xrightarrow{\mathbf{A}/\tau} (\mathcal{E}', e_1'\ \texttt{interrupt}[v]\ e_2)}[int_1]$$

$$\frac{(\mathcal{E}, e_1) \xrightarrow{l} (\mathcal{E}', v)}{(\mathcal{E}, e_1\ \texttt{interrupt}[v]\ e_2) \xrightarrow{l} (\mathcal{E}', v)}[int_2] \qquad \frac{}{(\mathcal{E}, e_1\ \texttt{interrupt}[0]\ e_2) \xrightarrow{\tau} (\mathcal{E}, e_2)}[int_3]$$

$$\frac{(\mathcal{E}, e_1) \xrightarrow{\mathrm{d}} (\mathcal{E}, e_1') \quad (\mathrm{d}{\le}v)}{(\mathcal{E}, e_1\ \texttt{interrupt}[v]\ e_2) \xrightarrow{\mathrm{d}} (\mathcal{E}, e_1'\ \texttt{interrupt}[v\text{-d}]\ e_2)}[int_4]$$

5.2.3 The Specification Language

We plant *TimEffs* specifications into the Hoare-style verification system, using Φ_{pre} and Φ_{post} to capture the temporal pre/post conditions. As shown in Figure 5.3, *TimEffs* can be constructed by a conditioned event sequence $\pi \wedge \theta$; or an effects disjunction $\Phi_1 \vee \Phi_2$. Timed sequences comprise *nil* ($\bot$); empty trace ϵ; single event ev; concatenation $\theta_1 \cdot \theta_2$; disjunction $\theta_1 \vee \theta_2$; parallel composition $\theta_1||\theta_2$; a block waiting for a certain constraint to be satisfied $\pi?\theta$. We introduce a new operator #, and $\theta\#t$ represents the trace θ takes t time units to complete, where t is a *real-time term*. A timed sequence also can be constructed by $\theta^\star$, representing zero or more times repetition of the trace θ. For single events, $\mathbf{A}(v, \alpha^*)$ stands for an observable event with label $\mathbf{A}$, parameterized by v, and the assignment operations α^*; $\tau(\pi)$ is an invisible event, parameterized with a pure formula π[3].

Events can also be $\overline{\mathbf{A}}$, referring to all events which are not labeled using $\mathbf{A}$; and a wildcard _, which matches to all the events. We use π to denote a pure formula which captures the (Presburger) arithmetic conditions on terms or program parameters. We

[3]The difference between $\tau(\pi)$ and $\pi?$ is: $\tau(\pi)$ marks an assertion which leads to false ($\bot$) if π is not satisfied, whereas $\pi?$ waits until π is satisfied.

$$
\begin{array}{rrcl}
(\textit{Timed Effects}) & \Phi & ::= & \pi \wedge \theta \mid \Phi_1 \vee \Phi_2 \\
(\textit{Event Sequences}) & \theta & ::= & \bot \mid \epsilon \mid ev \mid \theta_1 \cdot \theta_2 \mid \theta_1 \vee \theta_2 \mid \theta_1 || \theta_2 \mid \pi ? \theta \mid \theta \# t \mid \theta^\star \\
(\textit{Events}) & ev & ::= & \mathbf{A}(v, \alpha^*) \mid \tau(\pi) \mid \overline{\mathbf{A}} \mid \_ \\
(\textit{Pure}) & \pi & ::= & True \mid False \mid bop(t_1, t_2) \mid \pi_1 \wedge \pi_2 \mid \pi_1 \vee \pi_2 \\
 & & & \mid \neg \pi \mid \pi_1 \Rightarrow \pi_2 \\
(\textit{Real-Time Terms}) & t & ::= & c \mid x \mid t_1 + t_2 \mid t_1 - t_2
\end{array}
$$

$$
c \in \mathbb{Z} \qquad x \in \mathbf{var} \qquad (\textit{Real Time Bound})\ \# \qquad (\textit{Kleene Star})\ \star
$$

Figure 5.3: Syntax of *TimEffs*.

use $bop(t_1, t_2)$ to represent binary atomic formulas of terms (including $=$, $>$, $<$, $\geq$ and $\leq$). Terms consist of constant integer values c; integer variables x; simple computations of terms, t_1+t_2 and t_1-t_2.

5.2.4 Semantic Model of Timed Effects

Let $d, \mathcal{E}, \varphi \models \Phi$ denote the model relation, i.e., with the stack $\mathcal{E}$, the concrete execution trace φ take d time units to complete; and they satisfy the specification Φ. To define the model, *var* is the set of program variables, *val* is the set of primitive values; and d, $\mathcal{E}$, φ are drawn from the following concrete domains: d: $\mathbb{N}$, $\mathcal{E}$: $var \to val$ and φ: *list of event*. As shown in Figure 5.4, ++ appends event sequences; [] describes the empty sequences, $[ev]$ represents the singleton sequence contains event ev; $[\![\pi]\!]_{\mathcal{E}} = True$ represents π holds on the stack $\mathcal{E}$. Notice that, simple events, i.e., without #, are taken to be happening in instant time.

5.2.4.1 Expressiveness.

TimEffs draw similarities to metric temporal logic (MTL), which is derived from LTL, where a set of non-negative real numbers is added to temporal modal operators. As shown in Table 5.1, we are able to encode MTL operators into *TimEffs*, making it more intuitive and readable. By putting effects in the postcondition, they restrict *future traces*; whereas in the precondition, they naturally encode *past-time* temporal specifications. The basic modal operators are: $\Box$ for "globally"; $\Diamond$ for "finally"; $\bigcirc$ for "next"; $\mathcal{U}$ for "until", and their past time reversed versions: $\overleftarrow{\Box}$; $\overleftarrow{\Diamond}$; and $\ominus$ for

$$
\begin{array}{lll}
d,s,\varphi \models \Phi_1 \vee \Phi_2 & \textit{iff} & d,s,\varphi \models \Phi_1 \textit{ or } d,s,\varphi \models \Phi_2 \\
d,s,\varphi \models \pi \wedge \epsilon & \textit{iff} & d{=}0 \textit{ and } [\![\pi]\!]_s{=}True \textit{ and } \varphi{=}[] \\
d,s,\varphi \models \pi \wedge ev & \textit{iff} & d{=}0 \textit{ and } [\![\pi]\!]_s{=}True \textit{ and } \varphi{=}[ev] \\
d,s,\varphi \models \pi \wedge (\theta_1 \cdot \theta_2) & \textit{iff} & \exists \varphi_1,\varphi_2.\ \varphi_1{+}{+}\varphi_2{=}\varphi \textit{ and } \exists d_1,d_2.\ d_1{+}d_2{=}d \\
 & & \quad s.t.\ d_1,s,\varphi_1{\models}\pi \wedge \theta_1 \textit{ and } d_2,s,\varphi_2{\models}\pi \wedge \theta_2 \\
d,s,\varphi \models \pi \wedge (\theta_1{\vee}\theta_2) & \textit{iff} & d,s,\varphi \models \pi \wedge \theta_1 \textit{ or } d,s,\varphi \models \pi \wedge \theta_2 \\
d,s,\varphi \models \pi{\wedge}(ev_1{\cdot}\theta_1)||(ev_2{\cdot}\theta_2) & \textit{iff} & d,s,\varphi \models \pi \wedge ev_1 \cdot (\theta_1||(ev_2 \cdot \theta_2)) \textit{ or } \\
 & & d,s,\varphi \models \pi \wedge ev_2 \cdot ((ev_1 \cdot \theta_1)||\theta_2) \\
d,s,\varphi \models \pi{\wedge}(ev{\cdot}\theta_1)||(ev{\cdot}\theta_2) & \textit{iff} & d,s,\varphi \models \pi \wedge ev \cdot (\theta_1||\theta_2) \\
d,s,\varphi \models \pi \wedge (ev\#t_1)||(\epsilon\#t_2) & \textit{iff} & d,s,\varphi \models (\pi \wedge t_1{\geq}t_2) \wedge (ev\#t_1) \textit{ or } \\
 & & d,s,\varphi \models (\pi \wedge t_1{<}t_2) \wedge (ev\#t_1) \cdot (\epsilon\#(t_2\text{-}t_1)) \\
d,s,\varphi \models \pi \wedge \pi_1?\theta & \textit{iff} & [\![\pi_1]\!]_s{=}True, d,s,\varphi \models \pi \wedge \theta \textit{ or } \\
 & & [\![\pi_1]\!]_s{=}False, d,s,\varphi \models \pi \wedge \pi_1?\theta \\
d,s,\varphi \models \pi \wedge \theta\#t & \textit{iff} & [\![\pi \wedge t{\geq}0]\!]_s{=}True,\ \exists\theta_1,\theta_2.\ \theta_1 \cdot \theta_2{=}\theta, \\
 & & \quad s.t.\ d,s,\varphi{\models}(\pi \wedge t_1{\geq}0{\wedge}t_2{\geq}0{\wedge}t_1{+}t_2{=}t) \\
 & & \qquad {\wedge}(\theta_1\#t_1) \cdot (\theta_2\#t_2)\ (\textit{fresh } t_1,t_2) \\
d,s,\varphi \models \pi \wedge \theta^\star & \textit{iff} & d,s,\varphi \models \pi \wedge \epsilon \textit{ or } d,s,\varphi \models \pi \wedge \theta \cdot \theta^\star \\
d,s,\varphi \models \textit{false} & \textit{iff} & [\![\pi]\!]_s{=}False \textit{ or } \varphi{=}\bot
\end{array}
$$

Figure 5.4: Semantics of *TimEffs*.

"previous"; $\mathcal{S}$ for "since". I in MTL is the time interval with concrete upper/lower bounds; whereas in *TimEffs* they can be symbolic bounds which are dependent on program inputs.

Table 5.1: Examples for converting MTL formulae into *TimEffs* with $\mathtt{t}{\in}I$ applied.

Φ_{post}	$\Box_I\mathbf{A} \equiv (\mathbf{A}^\star)\#\mathtt{t}$	$\Diamond_I\mathbf{A} \equiv (\\_^\star \cdot \mathbf{A})\#\mathtt{t}$	$\bigcirc_I\mathbf{A} \equiv (\\_)\#\mathtt{t} \cdot \mathbf{A}$	$\mathbf{A}\mathcal{U}_I\mathbf{B} \equiv (\mathbf{A}^\star)\#\mathtt{t} \cdot \mathbf{B}$
Φ_{pre}	$\overleftarrow{\Box}_I\mathbf{A} \equiv (\mathbf{A}^\star)\#\mathtt{t}$	$\overleftarrow{\Diamond}_I\mathbf{A} \equiv (\mathbf{A} \cdot \\_^\star)\#\mathtt{t}$	$\ominus_I\mathbf{A} \equiv \mathbf{A} \cdot ((\\_)\#\mathtt{t})$	$\mathbf{A}\mathcal{S}_I\mathbf{B} \equiv \mathbf{B} \cdot ((\mathbf{A}^\star)\#\mathtt{t})$

The paradigmatic non-regular linear language: $\mathtt{t{>}0}{\wedge}\mathbf{A}\#\mathtt{t} \cdot \mathbf{B}\#\mathtt{t}$ is context-free, and can be naturally expressed by *TimEffs*. Moreover, suppose we have a traffic light control system, we could have a specification $\mathtt{n{>}0}{\wedge}\mathtt{m{>}0} \wedge (\mathbf{Red}\#\mathtt{n} \cdot \mathbf{Yellow}\#\mathtt{m} \cdot \mathbf{Green}\#\mathtt{n})^\star$, which specifies that all the colors will occur at each life circle; and the duration of the green light and the red light are always the same. This example is context-sensitive which can not be easily expressed by finite state automata.

5.3 Automated Forward Verification

5.3.1 Forward Rules

Forward rules syntactically accumulate the effects of each statement, which are in the Hoare-style triples $\vdash \langle \Pi, \Theta \rangle \; e \; \langle \Pi', \Theta' \rangle$, where e is the given statement; $\langle \Pi, \Theta \rangle$ and $\langle \Pi', \Theta' \rangle$ are program states, i.e., disjunctions of conditioned event sequence $\pi \wedge \theta$. The meaning of the transition rules, can be described as: $\langle \Pi', \Theta' \rangle = \bigcup_{i=0}^{|\langle \Pi,\Theta \rangle|\text{-}1} \langle \Pi'_i, \Theta'_i \rangle \; \mathit{where} \; (\pi_i {\wedge} \theta_i) \in \langle \Pi, \Theta \rangle \; \mathit{and} \vdash \langle \pi_i, \theta_i \rangle \; e \; \langle \Pi'_i, \Theta'_i \rangle$[4].

We here present the selected forward rules in Figure 5.5, for time-related constructs and leave the rest rules in section B.2. Rule [*FV-Delay*] creates a trace $\epsilon\#t$, where t is fresh, and concatenates it to the current program state, together with the additional constraint $t{=}v$. Rule [*FV-Deadline*] computes the effects from e and adds an upper time-bound to the results. Rule [*FV-Timeout*] computes the effects from e_1 and e_2 using the starting state $\langle \pi, \epsilon \rangle$. The final state is an union of possible effects with their corresponding time bounds and arithmetic constraints. Note that, $hd(\Theta_1)$ and $tl(\Theta_1)$ return the event *head* (cf. Definition 17), and the tail of Θ_1 respectively. Rule [*FV-Interrupt*] computes the interruption interleaves of e_1's effects, which come from the over-approximation of all the possibilities. For example, for trace $\mathbf{A} \cdot \mathbf{B}$, the interruption with time t creates three possibilities: $(\epsilon\#t) \vee (\mathbf{A}\#t) \vee ((\mathbf{A} \cdot \mathbf{B})\#t)$. Then the rule continues to compute the effects of e_2; lastly, it prepends the original history θ to the final results. Algorithm 2 presents the interleaving algorithm for *interruptions*, where $+$ unions program states (cf. Definition 18 and Definition 19 for *fst* and D functions).

Theorem 5 (Soundness of the Forward Rules)**.** *Given any system configuration $\zeta{=}(\mathcal{E}, e)$, by applying the operational semantics rules, if $(\mathcal{E}, e) {\rightarrow^*} (\mathcal{E}', v)$ has execution time d and produces event sequence φ; and for any history effect $\pi {\wedge} \theta$,*

[4] $|\langle \Pi, \Theta \rangle|$ is the size of $\langle \Pi, \Theta \rangle$, i.e., the count of conditioned event sequence $\pi {\wedge} \theta$.

$$\frac{\theta' = \theta \cdot (\epsilon\#t) \quad (t \ is \ fresh)}{\mathcal{E} \vdash \langle \pi, \theta \rangle \ \texttt{delay}[v] \ \{\pi \wedge (t{=}v), \theta'\}} \ [FV\text{-}Delay] \qquad \frac{\mathcal{E} \vdash \{\pi, \epsilon\} \ e \ \{\Pi_1, \Theta_1\} \quad (t \ is \ fresh)}{\mathcal{E} \vdash \langle \pi, \theta \rangle \ \texttt{deadline}[v] \ e \ \langle \Pi_1 \wedge (t{\leq}v), \theta \cdot (\Theta_1\#t) \rangle} \ [FV\text{-}Deadline]$$

$$\frac{\mathcal{E} \vdash \{\pi, \epsilon\} \ e_1 \ \{\Pi_1, \Theta_1\} \qquad \mathcal{E} \vdash \{\pi, \epsilon\} \ e_2 \ \{\Pi_2, \Theta_2\} \qquad (t_1, t_2 \ are \ fresh) \\ \langle \Pi_f, \Theta_f \rangle = \langle \Pi_1 \wedge t_1{<}v, (hd(\Theta_1)\#t_1) \cdot tl(\Theta_1) \rangle \cup \langle \Pi_2 \wedge t_2{=}v, (\epsilon\#t_2) \cdot \Theta_2 \rangle}{\mathcal{E} \vdash \langle \pi, \theta \rangle \ e_1 \ \texttt{timeout}[v] \ e_2 \ \langle \Pi_f, \theta \cdot \Theta_f \rangle} \ [FV\text{-}Timeout]$$

$$\frac{\mathcal{E} \vdash \{\pi, \epsilon\} \ e_1 \ \{\Pi, \Theta\} \qquad \Delta = \bigcup_{i=0}^{|\langle \Pi, \Theta \rangle|\text{-}1} \aleph_{Interleave}^{Interrupt(v, \pi_i)}(\theta_i, \epsilon) \qquad \mathcal{E} \vdash \{\Delta\} \ e_2 \ \{\Pi', \Theta'\}}{\mathcal{E} \vdash \langle \pi, \theta \rangle \ e_1 \ \texttt{interrupt}[v] \ e_2 \ \langle \Pi', \theta \cdot \Theta' \rangle} \ [FV\text{-}Interrupt]$$

Figure 5.5: Selected Forward Rules for *TimEffs*

Algorithm 2: Interruption Interleaving

Input: $v, \pi, \theta, \theta_{his}$

Output: Program States: Δ

```
function ℵ^{Interrupt(v,π)}_{Interleave}(θ, θ_his)
    Δ ← []
    foreach f ∈ fst_π(θ) do
        φ ← π ∧ (t<v) ∧ (θ_his # t)
        θ' ← D^π_f(θ)
        θ'_his ← θ_his · f
        Δ' ← ℵ^{Interrupt(v,π)}_{Interleave}(θ', θ'_his)
        Δ ← Δ + φ + Δ'
    return Δ
```

such that $d_1, \mathcal{E}, \varphi_1 \models (\pi \wedge \theta)$, and the forward verifier reasons $\mathcal{E} \vdash \langle \pi, \theta \rangle e \langle \Pi, \Theta \rangle$, then $\exists (\pi' \wedge \theta') \in \langle \Pi, \Theta \rangle$ such that $(d_1{+}d), \mathcal{E}', (\varphi_1{+}{+}\varphi) \models (\pi' \wedge \theta')$. (Note that, $\zeta \rightarrow^* \zeta'$ denotes the reflexive, transitive closure of $\zeta \rightarrow \zeta'$.)

Proof. By induction on the structure of e:

1. **Value:** $(\mathcal{E}, v) \xrightarrow{\tau} (\mathcal{E}, v)$ [v]

When $((\mathcal{E}, v) \rightarrow (\mathcal{E}, v))$, it takes 0 time and produces am empty sequence []. By rule [FV-Value], $\mathcal{E} \vdash \{\pi, \theta\}\ v\ \{\pi, \theta\}$, then the post effect is the witness that $(d_1{+}0), \mathcal{E}, (\varphi_1{+}{+}[]) \models \pi \wedge \theta$ is valid.

2. **Event:** $(\mathcal{E}, \texttt{event}[\mathbf{A}(v, \alpha^*)]) \xrightarrow{\mathbf{A}(v)} (\mathcal{E}[\alpha^*], ())\ [ev]$

When $(\mathcal{E}, \texttt{event}[\mathbf{A}(v, \alpha^*)]) \rightarrow^* (\mathcal{E}[\alpha^*], ())$, it takes 0 time and produces the event sequence $[\mathbf{A}(v, \alpha^*)]$. By rule [FV-Value], $\mathcal{E} \vdash \{\pi, \theta\}\ \mathbf{A}(v, \alpha^*)\ \langle \pi, \theta \cdot \mathbf{A}(v, \alpha^*)\rangle$, then the post effect is the witness that $(d_1{+}0), \mathcal{E}[\alpha^*], (\varphi_1{+}{+}[\mathbf{A}(v, \alpha^*)]) \models \pi \wedge \theta \cdot \mathbf{A}(v, \alpha^*)$.

3. **Guard:**

$$\frac{\mathcal{E} \models (v{=}true)}{(\mathcal{E}, [v]e) \xrightarrow{\tau} (\mathcal{E}, e)}[gu_1] \qquad \frac{\mathcal{E} \not\models (v{=}true)}{(\mathcal{E}, [v]e) \xrightarrow{\tau} (\mathcal{E}, [v]e)}[gu_2]$$

When $(\mathcal{E}, [v]e) \rightarrow^* (\mathcal{E}, v')$, it produces the sequence $\varphi(e)$. By [*FV-Guard*], $\mathcal{E} \vdash \langle \pi, \theta \rangle\ [v]e\ \langle \Pi, \theta \cdot (v{=}\mathit{True})?\Theta\rangle$ where $\mathcal{E} \vdash \{\pi, \epsilon\}e\{\Pi, \Theta\}$. Then the post effect is the witness that $(d_1{+}d_{wait}{+}d_e), \mathcal{E}, (\varphi_1{+}{+}[\varphi(e)]) \models \Pi \wedge \theta \cdot (v{=}\mathit{True})?\Theta$ is valid.

4. **Delay:**

$$\frac{\mathrm{d} \leq v}{(\mathcal{E}, \texttt{delay}[v]) \xrightarrow{\mathrm{d}} (\mathcal{E}, \texttt{delay}[v\text{-d}])}\ [delay_1] \qquad \frac{}{(\mathcal{E}, \texttt{delay}[0]) \xrightarrow{\tau} (\mathcal{E}, ())}\ [delay_2]$$

When $(\mathcal{E}, \texttt{delay}[v]) \rightarrow^* (\mathcal{E}, ())$, by applying rules $[delay_1]$, $[delay_2]$, it produces am empty sequence $[]$, and takes time $\mathcal{E}(v)$. By [*FV-Delay*], $\mathcal{E} \vdash \langle \pi, \theta \rangle\ \texttt{delay}[v]$ $\langle \pi \wedge (t{=}d), \theta \cdot \epsilon \# t\rangle$. Then the post effect $\pi \wedge (t{=}d) \wedge \theta \cdot \epsilon \# t$ is the witness that $(d_1{+}\mathcal{E}(v)), \mathcal{E}, (\varphi_1{+}{+}[]) \models \pi \wedge (t{=}v) \wedge \theta \cdot \epsilon \# t$ is valid.

5. **Timeout:**

$$\frac{(\mathcal{E}, e_1) \xrightarrow{\mathbf{A}} (\mathcal{E}', e_1')}{(\mathcal{E}, e_1\ \texttt{timeout}[v]e_2) \xrightarrow{\mathbf{A}} (\mathcal{E}', e_1')}[to_1] \quad \frac{(\mathcal{E}, e_1) \xrightarrow{\tau} (\mathcal{E}', e_1')}{(\mathcal{E}, e_1\ \texttt{timeout}[v]e_2) \xrightarrow{\tau} (\mathcal{E}', e_1'\ \texttt{timeout}[v]e_2)}[to_2]$$

$$\frac{(\mathcal{E}, e_1) \xrightarrow{\mathrm{d}} (\mathcal{E}, e_1') \qquad (\mathrm{d} \leq v)}{(\mathcal{E}, e_1\ \texttt{timeout}[v]\ e_2) \xrightarrow{\mathrm{d}} (\mathcal{E}, e_1'\ \texttt{timeout}[v\text{-d}]e_2)}[to_3] \frac{}{(\mathcal{E}, e_1\ \texttt{timeout}[0]e_2) \xrightarrow{\tau} (\mathcal{E}, e_2)}[to_4]$$

When $(\mathcal{E}, e_1\ \texttt{timeout}[v]\ e_2) \rightarrow^* (\mathcal{E}', v')$, there are two possibilities:

- e_1 started before time bound $\mathcal{E}(v)$: by applying rules $[to_2]$, $[to_3]$ and $[to_1]$, it produces the concrete sequence $[\mathbf{A};\ tl(\varphi(e_1))]$, and $\mathbf{A}$ takes t_1 time-units, which is less than $\mathcal{E}(v)$. By $[FV\text{-}Timeout]$, $\mathcal{E} \vdash \langle\pi,\theta\rangle\ e_1\ \texttt{timeout}[v]\ e_2\ \langle\Pi_1{\wedge}t_1{<}v, \theta\cdot(hd(\Theta_1)\#t_1)\cdot tl(\Theta_1)\rangle$ where $\mathcal{E} \vdash \{\pi,\epsilon\}e_1\{\Pi_1,\Theta_1\}$. Then the post effect is the witness such that $(d_1{+}t_1), \mathcal{E}', (\varphi_1{+}{+}[\mathbf{A}; tl(\varphi(e_1))]) \models \Pi_1{\wedge}(t_1{<}v) \wedge \theta\cdot(hd(\Theta_1)\#t_1)\cdot tl(\Theta_1)$. - e_1 never started, by applying rules $[to_4]$, it takes time d and produces the concrete sequence $[\varphi(e_2)]$. By $[FV\text{-}Timeout]$, $\mathcal{E} \vdash \langle\pi,\theta\rangle\ e_1\ \texttt{timeout}[v]\ e_2\ \langle\Pi_2{\wedge}t_2{=}v, \theta\cdot(\epsilon\#t_2)\cdot\Theta_2\rangle$ where $\mathcal{E} \vdash \{\pi,\epsilon\}e_2\{\Pi_2,\Theta_2\}$. Then the post effect is the witness such that $(d_1{+}d), \mathcal{E}', (\varphi_1{+}{+}[\varphi(e_2)]) \models \Pi_2{\wedge}t_2{=}v \wedge \theta\cdot(\epsilon\#t_2)\cdot\Theta_2$ is valid.

6. **Deadline:**

$$\frac{(\mathcal{E},e) \xrightarrow{\mathbf{A}/\tau} (\mathcal{E}',e')}{(\mathcal{E},\texttt{deadline}[v]\ e) \xrightarrow{\mathbf{A}/\tau} (\mathcal{E}',\texttt{deadline}[v]\ e')}[ddl_1] \qquad \frac{(\mathcal{E},e) \xrightarrow{l} (\mathcal{E}',v)}{(\mathcal{E},\texttt{deadline}[v]\ e) \xrightarrow{l} (\mathcal{E}',v)}[ddl_2]$$

$$\frac{(\mathcal{E},e) \xrightarrow{\mathrm{d}} (\mathcal{E},e') \quad (\mathrm{d}{\leq}v)}{(\mathcal{E},\texttt{deadline}[v]\ e) \xrightarrow{\mathrm{d}} (\mathcal{E},\texttt{deadline}[v\text{-}\mathrm{d}]\ e')}[ddl_3]$$

When $(\mathcal{E},\texttt{deadline}[v]\ e){\to^*}(\mathcal{E}',v')$, by applying rules $[ddl_1]$, $[ddl_2]$ and $[ddl_3]$, it produces the concrete sequence $[\varphi(e)]$, and it takes d time-units which is less than $\mathcal{E}(v)$. By $[FV\text{-}Deadline]$, $\mathcal{E} \vdash \langle\pi,\theta\rangle\ \texttt{deadline}[v]\ e\ \langle\Pi_1{\wedge}(t{\leq}v), \theta\cdot(\Theta_1\#t)\rangle$ where $\mathcal{E} \vdash \{\pi,\epsilon\}\ e\ \{\Pi_1,\Theta_1\}$. Then the post effect is the witness such that $(d_1{+}d), \mathcal{E}', (\varphi_1{+}{+}[\varphi(e)]) \models \Pi_1{\wedge}(t{\leq}v) \wedge \theta\cdot(\Theta_1\#t)$ is valid.

7. **Interrupt:**

$$\frac{(\mathcal{E},e_1) \xrightarrow{\mathbf{A}/\tau} (\mathcal{E}',e_1')}{(\mathcal{E},e_1\ \texttt{interrupt}[v]\ e_2) \xrightarrow{\mathbf{A}/\tau} (\mathcal{E}',e_1'\ \texttt{interrupt}[v]\ e_2)}[int_1]$$

$$\frac{(\mathcal{E},e_1) \xrightarrow{l} (\mathcal{E}',v)}{(\mathcal{E},e_1\ \texttt{interrupt}[v]\ e_2)\xrightarrow{l}(\mathcal{E}',v)}[int_2] \qquad \frac{}{(\mathcal{E},e_1\ \texttt{interrupt}[0]\ e_2) \xrightarrow{\tau} (\mathcal{E},e_2)}[int_3]$$

When $(\mathcal{E},e_1\ \texttt{interrupt}[v]\ e_2){\to^*}(\mathcal{E}',v')$, by applying rules $[int_1]$, $[int_2]$ and $[int_3]$, it produces many possible sequences, which depends of how many events e_1 can trigger before time bound v. For example, - when there is only one event triggered before the time bound, by Algorithm 1, $\Delta{=}\pi{\wedge}(t{<}v) \wedge \theta \cdot hd(\varphi(e_1))\#t$. By [$FV$-$Interrupt$], $\mathcal{E} \vdash \langle\pi, \theta\rangle\ e_1\ \texttt{interrupt}[v]\ e_2\ \langle\Pi', \theta\cdot\Theta'\rangle$ where $\mathcal{E} \vdash \{\Delta\}\ e_2\ \{\Pi', \Theta'\}$. Then the post effect is the witness such that $(d_1{+}t{+}d_{e2}), \mathcal{E}', (\varphi_1{+}{+}[hd(\varphi(e_1))]{+}{+}[\varphi(e_2)]) \models \pi{\wedge}(t < v) \wedge \theta \cdot hd(\varphi(e_1))\#t \cdot \Theta'$. is valid. Similar proofs for other possibilities.

8. **Conditional:**

$$\frac{\mathcal{E}(v) = True}{(\mathcal{E}, if\ v\ e_1\ e_2)\xrightarrow{\tau}(\mathcal{E}, e_1)}\ [cond_1]\frac{\mathcal{E}(v) = False}{(\mathcal{E}, if\ v\ e_1\ e_2)\xrightarrow{\tau}(\mathcal{E}, e_2)}\ [cond_2]$$

When $(\mathcal{E}, if\ v\ e_1\ e_2){\rightarrow^*}(\mathcal{E}', v')$, there are two possibilities:

- when $\mathcal{E}(v){=}True$, it takes d_{e1} time units and produces sequence $\varphi(e_1)$. By [FV-$Cond$-$Local$], $\mathcal{E} \vdash \langle\pi, \theta\rangle$ **if** v **then** e_1 **else** e_2 $\langle\Pi_1, \theta \cdot \tau(v{=}True) \cdot \Theta_1\rangle$ where $\mathcal{E} \vdash \{\pi, \epsilon\}\ e_1\ \{\Pi_1, \Theta_1\}$. Then the post effect is the witness such that $(d_1{+}d_{e1}), \mathcal{E}', (\varphi_1{+}{+}[\varphi(e_1)]) \models \Pi_1, \theta \cdot \tau(v{=}True) \cdot \Theta_1$ is valid.
- when $\mathcal{E}(v){=}False$ it takes d_{e2} time units and produces sequence $\varphi(e_2)$. By [FV-$Cond$-$Local$], $\mathcal{E} \vdash \langle\pi, \theta\rangle$ **if** v **then** e_1 **else** e_2 $\langle\Pi_2, \theta \cdot \tau(v{=}True) \cdot \Theta_2\rangle$ where $\mathcal{E} \vdash \{\pi, \epsilon\}\ e_2\ \{\Pi_2, \Theta_2\}$.

Then the post effect is the witness such that $(d_1{+}d_{e2}), \mathcal{E}', (\varphi_1{+}{+}[\varphi(e_2)]) \models \Pi_2, \theta \cdot \tau(v{=}False) \cdot \Theta_2$ is valid.

9. **Function Call:**

$$\frac{mnx^*\ \{e\}{\in}\mathcal{P} \quad (\mathcal{E}, e[v^*/x^*])\xrightarrow{l}(\mathcal{E}', e')}{(\mathcal{E}, mn(v^*)) \xrightarrow{l} (\mathcal{E}', e')}\ [call]$$

When $(\mathcal{E}, mn(v^*){\rightarrow^*}(\mathcal{E}', v')$, it takes d_e time units and produces sequence $\varphi(e)$. By [FV-$Call$], $\mathcal{E} \vdash \langle\pi, \theta\rangle\ mn(v^*)\ \langle\Phi_f\rangle$ where $\Phi_f{=}\langle\pi, \theta\rangle \cdot \Phi_{post}[v^*/x^*]$. The post effect is the witness of $(d_1{+}d_e), \mathcal{E}', (\varphi_1{+}{+}[\varphi(e)]) \models \langle\pi, \theta\rangle \cdot \Phi_{post}[v^*/x^*]$.

□

5.4 Temporal Verification via a TRS

The TRS is an automated entailment checker to prove language inclusions between *TimEffs*. It is triggered prior to function calls for the precondition checking; and by the end of verifying a function, for the post condition checking.

Given two effects Φ_1 and Φ_2, the 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 timed traces described by the consequent Φ_2. Here Γ is the proof context, i.e., a set of effects inclusion hypothesis; and 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 $\Gamma{=}\emptyset$ and $\Phi{=}True \wedge \epsilon$.

Effects Disjunctions. An inclusion with a disjunctive antecedent succeeds if both disjunctions entail the consequent. An inclusion with a disjunctive consequent succeeds if the antecedent entails either of the disjunctions.

$$\frac{\Gamma \vdash \Phi_1 \sqsubseteq \Phi \qquad \Gamma \vdash \Phi_2 \sqsubseteq \Phi}{\Gamma \vdash \Phi_1 \vee \Phi_2 \sqsubseteq \Phi}\ [\mathit{LHS\text{-}OR}] \quad \frac{\Gamma \vdash \Phi \sqsubseteq \Phi_1 \quad \mathit{or} \quad \Gamma \vdash \Phi \sqsubseteq \Phi_2}{\Gamma \vdash \Phi \sqsubseteq \Phi_1 \vee \Phi_2}\ [\mathit{RHS\text{-}OR}]$$

Now, the inclusions are disjunction-free formulas. Next we provide definitions and key implementations of auxiliary functions Nullable, First and Derivative. Intuitively, the Nullable function $\delta_{\pi}(\theta)$ returns a Boolean value indicating whether $\pi{\wedge}\theta$ contains the empty trace; the First function $fst_{\pi}(\theta)$ computes a set of initial *heads*, denoted as h, of $\pi{\wedge}\theta$; the Derivative function $D^{\pi}_{h}(\theta)$ computes

Definition 16 (Nullable [5]). Given any $\Phi{=}\pi{\wedge}\theta$, $\delta_\pi(\theta) : bool{=}\begin{cases} true & if\ \epsilon \in [\![\pi{\wedge}\theta]\!] \\ false & if\ \epsilon \notin [\![\pi{\wedge}\theta]\!] \end{cases}$

$$\delta_\pi(\bot){=}\delta_\pi(ev){=}\delta_\pi(\pi'?) = false \qquad \delta_\pi(\epsilon){=}\delta(\theta^\star){=}true \qquad \delta_\pi(\theta \cdot \theta_2){=}\delta(\theta_1) \wedge \delta(\theta_2)$$

$$\delta_\pi(\theta_1 \vee \theta_2) = \delta(\theta_1) \vee \delta(\theta_2) \qquad \delta_\pi(\theta_1 || \theta_2) = \delta(\theta_1) \wedge \delta(\theta_2) \qquad \delta_\pi(\theta \# t) = SAT(\pi{\wedge}(t{=}0))$$

Definition 17 (Heads). If h is a head of $\pi \wedge \theta$, then there exist π' and θ', such that $\pi \wedge \theta = \pi' \wedge (h \cdot \theta')$. A head can be t, denoting a pure time passing; $\mathbf{A}(v, \alpha^*)$, denoting an instant event passing; or $(\mathbf{A}(v, \alpha^*), t)$, denoting an event passing which takes time t.

Definition 18 (First). Given any $\Phi{=}\pi \wedge \theta$, $fst_\pi(\theta)$ returns a set of heads, be the set of initial elements derivable from effects $\pi \wedge \theta$, where (t' *is fresh*):

$$fst_\pi(\bot){=}fst_\pi(\epsilon){=}\{\} \qquad fst_\pi(\mathbf{A}(v, \alpha^*)){=}\{\mathbf{A}(v, \alpha^*)\} \qquad fst_\pi(\epsilon \# t){=}\{t\}$$

$$fst_\pi(\theta \# t){=}\{(\mathbf{A}(v, \alpha^*), t') \mid \mathbf{A}(v, \alpha^*){\in}fst_\pi(\theta)\} \qquad fst_\pi(\theta_1{\vee}\theta_2){=}fst_\pi(\theta_1) \cup fst_\pi(\theta_2)$$

$$fst_\pi(\pi'?\theta){=}fst_\pi(\theta) \qquad fst_\pi(\theta_1||\theta_2){=}fst_\pi(\theta_1) \cup fst_\pi(\theta_2)$$

$$fst_\pi(\theta^\star){=}fst_\pi(\theta) \qquad fst_\pi(\theta_1 \cdot \theta_2){=}\begin{cases} fst_\pi(\theta_1) \cup fst_\pi(\theta_2) & if\ \ \delta(\theta_1){=}true \\ fst_\pi(\theta_1) & if\ \ \delta(\theta_1){=}false \end{cases}$$

Definition 19 (*TimEffs* Partial Derivative). Given any $\Phi{=}\pi \wedge \theta$, the partial derivative $D^\pi_h(\theta)$ computes the effects for the left quotient $h^{-1}(\pi \wedge \theta)$, cf. Definition 1.

$$D^\pi_h(\bot){=}D^\pi_h(\epsilon){=}False{\wedge}\bot \quad D^\pi_h(\mathbf{A}(v, \alpha^*)){=}(\pi{\wedge}(h{=}\mathbf{A}(v, \alpha^*))){\wedge}\epsilon \quad D^\pi_h(\theta^\star){=}D^\pi_h(\theta){\cdot}\theta^\star$$

$$D^\pi_{\tau(\pi_1)}(\pi'?\theta){=}\begin{cases} \pi{\wedge}\pi'?\ if\ \alpha^*{\not\Rightarrow}\pi' \\ \pi{\wedge}\epsilon \quad\ if\ \alpha^*{\Rightarrow}\pi' \end{cases} \quad D^\pi_h(\theta_1{\cdot}\theta_2){=}\begin{cases} D^\pi_h(\theta_1){\cdot}\theta_2{\vee}D^\pi_h(\theta_2)\ if\ \delta_\pi(\theta_1){=}true \\ D^\pi_h(\theta_1){\cdot}\theta_2 \qquad\qquad if\ \delta_\pi(\theta_1){=}false \end{cases}$$

$$D^\pi_{(\mathbf{A}(v,\alpha^*),t)}(\theta) = \bigvee\{D^{\pi'}_{\mathbf{A}(v,\alpha^*)}(\theta') \mid (\pi' \wedge \theta') \in D^\pi_t(\theta)\}$$

$$D^\pi_t(\theta \# t'){=}(\pi \wedge t{+}t''{=}t') \wedge \theta \# t'' \quad (t''\ is\ fresh) \qquad D^\pi_h(\theta_1{\vee}\theta_2){=}D^\pi_h(\theta_1) \vee D^\pi_h(\theta_2)$$

$$D^\pi_{\mathbf{A}(v,\alpha^*)}(\theta \# t){=}\bigvee\{(\pi'{\wedge}(\theta' \# t)) \mid (\pi'{\wedge}\theta'){\in}D^\pi_{\mathbf{A}(v,\alpha^*)}(\theta)\} \quad D^\pi_h(\theta_1||\theta_2){=}\bar{\bar{D}}^\pi_h(\theta_1)||\bar{\bar{D}}^\pi_h(\theta_2)$$

[5] $SAT(\pi)$ stands for querying the Z3 theorem prover to check the satisfiability of π.

Notice that the derivatives of a parallel composition makes use of the *Parallel Derivative* $\bar{\bar{D}}^{\pi}_{h}(\theta)$, defined as follows: $\bar{\bar{D}}^{\pi}_{h}(\theta) = \begin{cases} \pi \wedge \theta & \textit{if } D^{\pi}_{h}(\pi \wedge \theta) = (False \wedge \bot) \\ D^{\pi}_{h}(\theta) & \textit{otherwise} \end{cases}$

5.4.1 Rewriting Rules.

Given the well-defined auxiliary functions above, we now discuss the key rewriting rules that deployed in effects inclusion proofs.

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

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

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.

$$\frac{\delta_{\pi_1}(\theta_1) \wedge \neg\delta_{\pi_2}(\theta_2)}{\Gamma \vdash \pi_1 \wedge \theta_1 \not\sqsubseteq \pi_2 \wedge \theta_2}\ [\mathit{DISPROVE}] \qquad\qquad \frac{\pi_1 \Rightarrow \pi_2 \qquad fst_{\pi_1}(\theta_1) = \{\}}{\Gamma \vdash \pi_1 \wedge \theta_1 \sqsubseteq \pi_2 \wedge \theta_2}\ [\mathit{PROVE}]$$

3. **Prove.** We use two rules to prove an inclusion: (i) [*PROVE*] is used when the antecedent has no head; and (ii) [*REOCCUR*] proves an inclusion when there exist inclusion hypotheses in the proof context Γ, which are able to soundly prove the current goal. The special case of [*REOCCUR*] is when the identical inclusion is shown in the proof context, then the TRS then terminates and proves it valid.

$$[\mathit{REOCCUR}]$$
$$\frac{(\pi_1 \wedge \theta_1 \sqsubseteq \pi_3 \wedge \theta_3) \in \Gamma \qquad (\pi_3 \wedge \theta_3 \sqsubseteq \pi_4 \wedge \theta_4) \in \Gamma \qquad (\pi_4 \wedge \theta_4 \sqsubseteq \pi_2 \wedge \theta_2) \in \Gamma}{\Gamma \vdash \pi_1 \wedge \theta_1 \sqsubseteq \pi_2 \wedge \theta_2}$$

4. **Unfolding (Induction).** This is the inductive step of unfolding the inclusions. Firstly, we make use of the fst function to get H, which are all the possible initial events from the antecedent. Secondly, we obtain a new proof context Γ' by extending Γ with the current inclusion, as an inductive hypothesis. Thirdly, we iterate each element $h \in H$, and compute the partial derivatives (*next-state* effects) of both the antecedent and consequent with respect to h. The proof of the original inclusion succeeds if all the derivative inclusions succeed.

$$\frac{H{=}fst_{\pi_1}(\theta_1) \qquad \Gamma'{=}\Gamma, (\pi_1{\wedge}\theta_1 \sqsubseteq \pi_2{\wedge}\theta_2) \qquad \forall h{\in}H.\ (\Gamma' \vdash D_h^{\pi_1}(\theta_1) \sqsubseteq D_h^{\pi_2}(\theta_2))}{\Gamma \vdash \pi_1 \wedge \theta_1 \sqsubseteq \pi_2 \wedge \theta_2}\ [\mathit{UNFOLD}]$$

Theorem 6 (Termination of the TRS)**.** *The TRS is terminating.*

Proof. See section B.3. □

Theorem 7 (Soundness of the TRS)**.** *Given an inclusion $\Phi_1 \sqsubseteq \Phi_2$, if the TRS returns $TRUE$ with a proof, then $\Phi_1 \sqsubseteq \Phi_2$ is valid.*

Proof. See section B.4. □

5.4.2 Discussion: highlighting the novelty.

Departing from the original Antimirov algorithm [AM95], this work devises extended definitions for the auxiliary functions: *Nullable*(δ), *First*(*fst*) and *Derivative*(D). These definitions cover extended constructs in the more expressive specifications formulae, *TimEffs*, which use arithmetic constraints to quantify the symbolic time bounds for the effect traces. The comprehensive rewriting system serves as a back-end engine for the finer-grained timed verification with symbolic time requirements, which cannot be trivially achieved by the original rewriting system.

5.5 Demonstration Examples

We use Figure 5.6 to highlight our main methodologies, which simulates a coffee machine, that dynamically adds sugar based on the user's input number.

5.5.1 *TimEffs.*

We define Hoare-triple style specifications (enclosed in `/*...*/`) for each function, which lead to a compositional verification strategy, where static checking can be done locally. The precondition of `makeCoffee` specifies that the input value `n` is non-negative, and it *requires* that before entering into this function, this history trace must contain the event **CupReady** on the tail. The verification fails if the precondition is not satisfied at the caller sites. Line 17 sets a five time-units deadline (i.e., maximum 5 portion of sugar per coffee) while calling `addNSugar` (defined in Figure 5.1); then emits event **Coffee** with a deadline, indicating the pouring coffer process takes no more than four time-units. The precondition of `main` requires no arithmetic constraints (expressed as `true`) and an empty history trace. The postcondition of `main` specifies that before the final **Done** happens, there is no any occurrence of **Done** (`!` indicates the absence of events); and the whole process takes at most nine time-units to hit the final event.

```
14 void makeCoffee (int n)
15 /* req:  n≥0 ∧ _*· CupReady
16   ens:  n≤t≤5 ∧ t'≤4 ∧(EndSugar # t) · (Coffee # t') */
17 { deadline (addNSugar(n), 5);
18   deadline (event["Coffee"],4);}
19
20 int main ()
21 /* req:  true ∧ ϵ
22   ens:  t≤9 ∧ ((!Done)* # t) · Done */
23 { event["CupReady"];
24   makeCoffee (3);
25   event["Done"];}
```

Figure 5.6: To make coffee with three portions of sugar within nine time units.

TimEffs support more features such as *disjunctions*, *guards*, *parallelism* and *assertions*, etc (cf. subsection 5.2.3), providing detailed information upon: branching properties: different arithmetic conditions on the inputs lead to different effects; and required history traces: by defining the prior effects in the precondition. These capabilities are beyond traditional timed verification, and cannot be fully captured by any prior works [Don+08; DM01; Wan+17; LPY97; Yov97; WWH05]. Nevertheless, the increase in expressive power needs support from finer-grind reasoning and a more sophisticated back-end solver, discharged by our forward verifier and TRS, respectively.

5.5.2 Forward Verification.

Figure 5.7 demonstrates the forward verification of functions `addOneSugar` and `addNSugar`, defined in Figure 5.1.

The effect states are captured in the form of $\{\Phi_C\}$. To facilitate the illustration, we label the steps by (1) to (11), and mark the deployed forward rules (cf. subsection 5.3.1) in [gray]. The initial states (1) and (4) are obtained from the precondition, by the [*FV-Fun*] rule. States (5)(7)(10) are obtained by [*FV-Cond*], which enforces the conditional constraints into the effect states, and unions the effects accumulated from two branches. State (6) is obtained by [*FV-Event*], which concatenates an event to the current effects. The intermediate states (8) and (9) are obtained by [*FV-Call*].

Before each function call, [*FV-Call*] invokes the TRS to check whether the current effect states satisfy callee's precondition. If it is not satisfied, the verification fails; otherwise, it concatenates the callee's postcondition to the current states (the precondition check for step (8) is omitted here). State (2) is obtained by [*FV-Timeout*], which adds a lower time-bound to an empty trace. After these state transformations, steps (3) and (11) invoke the TRS to check the inclusions between the final effects and the declared postcondition. Here `t1` and `t2` are fresh time variables.

5.5.3 The TRS.

Having *TimEffs* to be the specification language, and the forward verifier to reason about the actual behaviors, we are interested in the following verification problem: Given a program $\mathcal{P}$, and a temporal specification Φ', does the inclusions $\Phi^{\mathcal{P}} \sqsubseteq \Phi'$ holds? Typically, checking the inclusion/entailment between the concrete

1. `void addOneSugar() {` *// initialize the state using the function precondition.*
$\Phi_C{=}\Phi_{pre}^{addOneSugar(n)} = \{\texttt{true} \wedge \_^\star\}$ [FV-Fun]

2. `timeout ((), 1);}`
$\Phi'_C{=}\{\texttt{t1>1} \wedge \_^\star \cdot (\epsilon \ \# \ \texttt{t1})\}$ [FV-Timeout]

3. $\Phi'_C \sqsubseteq \Phi_{pre}^{addOneSugar(n)} \cdot \Phi_{post}^{addOneSugar(n)} \Leftrightarrow \texttt{t1>1} \wedge \_^\star \cdot (\epsilon\#\texttt{t1}) \sqsubseteq \texttt{t>1} \wedge \_^\star \cdot (\epsilon\#\texttt{t})$

---

4. `void addNSugar (int n) {` *//initialize the state using the function precondition.*
$\Phi_C{=}\Phi_{pre}^{addNSugar(n)} = \{\texttt{true} \wedge \_^\star\}$ [FV-Fun]

5. `if (n == 0) {`
$\{\texttt{n}{=}\texttt{0} \wedge \_^\star\}$ [FV-Cond]

6. `event ["EndSugar"];}`
$\{\texttt{n}{=}\texttt{0} \wedge \_^\star \cdot \texttt{EndSugar}\}$ [FV-Event]

7. `else {`
$\{\texttt{n}{\neq}\texttt{0} \wedge \_^\star\}$ [FV-Cond]

8. `addOneSugar();`
$\{\texttt{n}{\neq}\texttt{0}{\wedge}\texttt{t2>1} \wedge \_^\star \cdot (\epsilon \ \# \ \texttt{t2})\}$ [FV-Call]

9. `addNSugar (n-1);}}`
$\texttt{n}{\neq}\texttt{0}{\wedge}\texttt{t2>1} \wedge \_^\star \cdot (\epsilon \ \# \ \texttt{t}) \sqsubseteq \Phi_{pre}^{addNSugar(n\text{-}1)}$ *// TRS: precondition checked.*
$\{\texttt{n}{\neq}\texttt{0}{\wedge}\texttt{t2>1} \wedge \_^\star \cdot (\epsilon \ \# \ \texttt{t2}) \cdot \Phi_{post}^{addNSugar(n\text{-}1)}\}$ [FV-Call]

10. $\Phi'_C{=}\ (\texttt{n}{=}\texttt{0} \wedge \_^\star{\cdot}\texttt{Sugar}) \vee (\texttt{n}{\neq}\texttt{0}{\wedge}\texttt{t2>1} \wedge \_^\star{\cdot}(\epsilon\#\texttt{t2}){\cdot}\Phi_{post}^{addNSugar(n\text{-}1)})$ [FV-Cond]

11. $\Phi'_C \sqsubseteq \Phi_{pre}^{addNSugar(n)} \cdot \Phi_{post}^{addNSugar(n)} \Leftrightarrow$ *//TRS: postcondition checked, cf. Table 5.2*
$(\texttt{n}{=}\texttt{0} \wedge \texttt{Sugar}) \vee (\texttt{n}{\neq}\texttt{0}{\wedge}\texttt{t2>1} \wedge (\epsilon \ \# \ \texttt{t2}) \cdot \Phi_{post}^{addNSugar(n\text{-}1)}) \sqsubseteq \Phi_{post}^{addNSugar(n)}$

Figure 5.7: Forward verification for functions *addOneSugar* and *addNSugar*.

program effects $\Phi^{\mathcal{P}}$ and the expected property Φ' proves that: the program $\mathcal{P}$ will never lead to unsafe traces which violate Φ'.

Our TRS is an extension of Antimirov and Mosses's algorithm [AM95], which can be deployed to decide inclusions of two regular expressions (REs) through an iterated process of checking inclusions of their *partial derivatives* [Ant95]. There are two basic rules: [*Disprove*] infers false from trivially inconsistent inclusions; and [*Unfold*] applies Theorem 1 to generate new inclusions.

Similarly, we defined Definition 20 for unfolding the inclusions between *TimEffs*, where $(\mathbf{A}\#\mathtt{t})^{-1}\Phi$ is the partial derivative of Φ with respect to the event $\mathbf{A}$ with the time bound `t`.

Definition 20 (*TimEffs* Inclusion)**.** Given Σ is a finite set of alphabet, for two *TimEffs* Φ_1 and Φ_2, their inclusion is defined as:

$$\Phi_1 \sqsubseteq \Phi_2 \Leftrightarrow \forall \mathbf{A}.\forall \mathtt{t}{\geq}0.\ (\mathbf{A}\#\mathtt{t})^{-1}\Phi_1 \sqsubseteq (\mathbf{A}\#\mathtt{t})^{-1}\Phi_2.$$

Termination of the rewriting is guaranteed because the set of derivatives to be considered is finite, and possible cycles are detected using *memorization* [Bro05a]. Next, we use Table 3.3 to demonstrate how the TRS automatically proves the final effects of `main` satisfying its postcondition (shown at step (11) in Figure 5.7). We mark the rewriting rules (cf. section 3.4) in [gray].

Table 5.2: An inclusion proving example. (I) is the right hand side sub-tree of the the main rewriting proof tree.

$$\dfrac{\dfrac{\dfrac{}{\mathtt{n{=}0} \wedge \epsilon \sqsubseteq \mathtt{tR{\geq}0} \wedge \epsilon\ \#\ \mathtt{tR}}\ ④[\mathit{PROVE}]}{\mathtt{n{=}0} \wedge \cancel{\mathtt{ES}} \sqsubseteq \mathtt{tR{\geq}0} \wedge \cancel{\mathtt{ES}}\#\mathtt{tR}}\ ③[\mathit{UNFOLD}] \qquad (I)}{\dfrac{(\mathtt{n{=}0}\wedge\mathtt{ES}) \vee (\mathtt{n{\neq}0}\wedge\mathtt{t2{>}1}\wedge\mathtt{tL{\geq}(n\text{-}1)}\wedge \epsilon\#\mathtt{t2} \cdot \mathtt{ES\#tL}) \sqsubseteq \mathtt{tR{\geq}n} \wedge \mathtt{ES\#tR}}{(\mathtt{n{=}0} \wedge \mathtt{ES}) \vee (\mathtt{n{\neq}0}\wedge\mathtt{t2{>}1} \wedge (\epsilon\ \#\ \mathtt{t2})\cdot\Phi_{post}^{addNSugar(n\text{-}1)}) \sqsubseteq \Phi_{post}^{addNSugar(n)}}\ ①[\mathit{RENAME}]}\ ②[\mathit{LHS\text{-}OR}]$$

(I)

$$\dfrac{\dfrac{\dfrac{\mathtt{t2{>}1}\wedge\mathtt{tL{\geq}(n\text{-}1)} \wedge \mathtt{tL{=}(tR\text{-}t2)} \Rightarrow \mathtt{tR{\geq}n}}{\mathtt{n{\neq}0}\wedge\mathtt{t2{>}1}\wedge\mathtt{tL{\geq}(n\text{-}1)} \wedge \epsilon \sqsubseteq \mathtt{tR{\geq}n} \wedge \epsilon}\ ⑦[\mathit{PROVE}]}{\mathtt{n{\neq}0}\wedge\mathtt{t2{>}1}\wedge\mathtt{tL{\geq}(n\text{-}1)} \wedge \cancel{\mathtt{ES\#tL}} \sqsubseteq \mathtt{tR{\geq}n} \wedge \cancel{\mathtt{ES\#(tR\text{-}t2)}}}\ ⑥[\mathit{UNFOLD}]\ \pi_u{:}\mathtt{tL{=}(tR\text{-}t2)}}{\mathtt{n{\neq}0}\wedge\mathtt{t2{>}1}\wedge\mathtt{tL{\geq}(n\text{-}1)} \wedge \cancel{\epsilon\#\mathtt{t2}\cdot}\ \mathtt{ES\#tL} \sqsubseteq \mathtt{tR{\geq}n} \wedge \mathtt{ES}\cancel{\mathtt{\#tR}}}\ ⑤[\mathit{UNFOLD}]$$

In Table 3.3, step ① renames the time variables to avoid the name clashes between the antecedent and the consequent. `ES` stands for the event `EndSugar`. Step ② splits the proof tree into two branches, according to the different arithmetic constraints, by rule [*LHS-OR*]. In the first branch, step ③ eliminates the event `ES` from the head of both sides, by rule [*UNFOLD*]. Step ④ proves the inclusion, because evidently the consequent $\mathtt{tR{\geq}0} \wedge \epsilon\#\mathtt{tR}$ contains ϵ when `tR=0`. In the second branch, step ⑤ eliminates the a time duration ϵ`#t2` from both sides. Therefore the rule [*UNFOLD*] subtracts a time duration from the consequent, i.e., `(tR-t2)`. Similarly, step ⑥ eliminates `ES#tL` from both sides, adding `tL=(tR-t2)` to the unification constraints. Step 7 manages to prove that `t2>1∧tL≥(n-1)∧tL=(tR-t2)⇒tR≥n` at the end of the rewriting[6]; therefore, the proof succeeds.

5.5.4 Verifying the Fischer's Mutual Exclusion Protocol.

Figure 5.8 presents the classical Fischer's mutual exclusion algorithm, in C^t. Global variables `x` and `ct` indicate *'which process attempted to access the critical section most recently'* and *'the number of processes accessing the critical section'* respectively.

```
var x := -1;
var cs:= 0;

void proc (int i) {
   [x=-1] //block waiting until true
   deadline(event["Update"(i)]{x:=i},d);
   delay (e);
   if (x==i) {
      event["Critical"(i)]{cs:=cs+1};
      event["Exit"(i)]{cs:=cs-1;x:=-1};
      proc (i);
   } else {proc (i);}}

void main ()
/* req:  d<e ∧ ϵ
  ens_a:true ∧ (cs≤1)*   ens_b:true ∧ ((_*).Critical.Exit.(_*))* */
  { proc(0) || proc(1) || proc(2); }
```

Figure 5.8: Fischer's mutual exclusion algorithm.

The `main` procedure is a parallel composition of three processes, where `d` and `e` are two integer constants. Each process attempts to enter the critical section when

[6]The proof obligations for arithmetic constraints are discharged by the Z3 solver [dMB08].

`x` is `-1`, i.e. no other process is currently attempting. Once the process is active (i.e., reaches line 6), it sets `x` to its identity number `i` within `d` time units, captured by `deadline(...,d)`. Then it idles for `e` time units, captured by `delay(e)` and then checks whether `x` still equals to `i`. If so, it safely enters the critical section. Otherwise, it restarts from the beginning. Quantitative timing constraint `d<e` plays an important role in this algorithm to guarantee mutual exclusion. In order to verify mutual exclusion, one way is to show that `ct`$\leq$1 is always true. The specification implies that this implementation is indeed mutual exclusive. Our prototype system (cf section 5.6) is able to effectively verify such time-critical algorithms.

5.6 Implementation and Evaluation

To show the feasibility, we prototype our automated verification system using OCaml (∼5k LOC); and prove soundness for both the forward verifier and the TRS. We set up two experiments to evaluate our implementation: i) functionality validation via verifying symbolic timed programs; and ii) comparison with PAT [Sun+09] and Uppaal [LPY97] using real-life Fischer's mutual exclusion algorithm. Experiments are done on a MacBook with a 2.6 GHz 6-Core Intel i7 processor. The source code and the evaluation benchmark are openly accessible from [Son22b].

5.6.1 Experimental Results for Symbolic Timed Models.

We manually annotate *TimEffs* specifications for a set of synthetic examples (for about 54 programs), to test the main contributions, including: computing effects from symbolic timed programs written in C^t; and the inclusion checking for *TimEffs* with the parallel composition, blok waiting operator and shared global variables.

This experiment is done without a baseline comparison because there are no existing tools for solving inclusion problems for symbolic timed automata, and our experimental results show that a modular and efficient temporal verification for symbolic time-critical systems is achievable.

Table 5.3 presents the evaluation results for another 16 C^t programs[7], and the

[7] All programs contain timed constructs, conditionals, and parallel compositions.

Table 5.3: Experimental Results for Manually Constructed Synthetic Examples.

No.	**LOC**	**Forward(ms)**	**#Prop(✓)**	**Avg-Prove(ms)**	**#Prop(✗)**	**Avg-Dis(ms)**	**#AskZ3**
1	26	0.006	5	52.379	5	21.31	77
2	37	43.955	5	83.374	5	52.165	188
3	44	32.654	5	52.524	5	33.444	104
4	72	202.181	5	82.922	5	55.971	229
5	98	42.706	7	149.345	7	60.325	396
6	134	403.617	7	160.932	7	292.304	940
7	133	51.492	7	17.901	7	47.643	118
8	173	57.114	7	40.772	7	30.977	128
9	182	872.995	9	252.123	9	113.838	1142
10	210	546.222	9	146.341	9	57.832	570
11	240	643.133	9	146.268	9	69.245	608
12	260	1032.31	9	242.699	9	123.054	928
13	265	12558.05	11	150.999	11	117.288	2465
14	286	12257.834	11	501.994	11	257.800	3090
15	287	1383.034	11	546.064	11	407.952	1489
16	337	49873.835	11	1863.901	11	954.996	15505

annotated temporal specifications are in a 1:1 ratio for succeeded/failed cases. The table records: **No.**, index of the program; **LOC**, lines of code; **Forward(ms)**, effects computation time; **#Prop(✓)**, number of valid properties; **Avg-Prove(ms)**, average proving time for the valid properties; **#Prop(✗)**, number of invalid properties; **Avg-Dis(ms)**, average disproving time for the invalid properties; **#AskZ3**, number of querying Z3 through out the experiments.

Observations: i) the proving/disproving time increases when the effect computation time increases because larger **Forward(ms)** indicates the higher complexity with respect to the timed constructs, which complicates the inclusion checking; ii) while *the number of querying Z3 per property* (**#AskZ3**/(**#Prop(✓)**+**#Prop(✗)**)) goes up, the proving/disproving time goes up. Besides, we notice that iii) the disproving times for invalid properties are constantly lower than the proving process, regardless of the program's complexity, which is as expected in a TRS.

5.6.2 Verifying Fischer's Mutual Exclusion Algorithm.

As shown in Table 5.4, the data in columns **PAT(s)** and **Uppaal(s)** are drawn from prior work [LSD11], which indicate the time to prove Fischer's mutual exclusion with respect to the number of processes (**#Proc**) in PAT and Uppaal respectively. For our system, based on the implementation presented in Figure 5.8, we are able to prove the mutual exclusion properties, given the arithmetic constraint `d<e`. Besides, the system disproves mutual exclusion when `d`$\leq$`e`. We record the proving (**Prove(s)**) and disproving (**Disprove(s)**) time and their number of uniquely querying Z3 (**#AskZ3-u**).

Table 5.4: Comparison with PAT via verifying Fischer's mutual exclusion algorithm

#Proc	Prove(s)	#AskZ3-u	Disprove(s)	#AskZ3-u	PAT(s)	Uppaal(s)
2	0.09	31	0.110	37	$\leq$0.05	$\leq$0.09
3	0.21	35	0.093	42	$\leq$0.05	$\leq$0.09
4	0.46	63	0.120	47	0.05	0.09
5	25.0	84	0.128	52	0.15	0.19

Observations: i) automata-based model checkers (both PAT and Uppaal) are vastly efficient when given concrete values for constants `d` and `e`; however ii) our proposal is able to symbolically prove the algorithm by only providing the constraints of `d` and `e`, which cannot be achieved by existing model checkers; ii) our verification time largely depends on the number of querying Z3, which is optimized in our implementation by keeping a table for recording the queried constraints.

5.6.3 Case Study: Prove it when Reoccur.

Termination of TRS is guaranteed because the set of derivatives to be considered is finite, and possible cycles are detected using *memorization* [Bro05a], demonstrated in Table 5.5. In step ②, in order to eliminate the first event `B`, `A*#tR` has to be reduced to ϵ, therefore the RHS time constraint has been strengthened to `tR=0`. Looking at the sub-tree (I), in step ⑤, `tL` and `tR` are split into $\texttt{tL}^1\texttt{+tL}^2$ and $\texttt{tR}^1\texttt{+tR}^2$. Then in step ⑥, $\texttt{A\#tL}^1$ together with $\texttt{A\#tR}^1$ are eliminated, unifying $\texttt{tL}^1$ and $\texttt{tR}^1$ by adding the side constraint $\texttt{tL}^1\texttt{=tR}^1$. In step ⑧, we observe the proposition is

Table 5.5: The reoccurrence proving example in *TimEffs*. (I) is the left hand side sub-tree of the main rewriting proof tree.

$$\dfrac{\dfrac{(I)}{\mathtt{tL{<}3} \wedge (\mathtt{A^{\star}\#tL})\cdot \mathtt{B} \sqsubseteq \mathtt{tR{<}4} \wedge (\mathtt{A^{\star}\#tR})\cdot \mathtt{B}} \qquad \dfrac{\dfrac{\dfrac{}{\mathtt{True} \wedge \epsilon \sqsubseteq \mathtt{tR{=}0} \wedge \epsilon}\ \textcircled{4}\,[PROVE]}{\mathtt{True} \wedge \cancel{\mathtt{B}} \sqsubseteq \mathtt{tR{=}0} \wedge \cancel{\epsilon\cdot \mathtt{B}}}\ \textcircled{3}[Normal]}{\mathtt{True} \wedge \mathtt{B} \sqsubseteq \mathtt{tR{<}4} \wedge (\mathtt{A^{\star}\#tR})\cdot \mathtt{B}}\ \textcircled{2}[UNFOLD]}{(\mathtt{tL{<}3} \wedge (\mathtt{A^{\star}\#tL})\cdot \mathtt{B}) \vee (\mathtt{True} \wedge \mathtt{B}) \sqsubseteq \mathtt{tR{<}4} \wedge (\mathtt{A^{\star}\#tR})\cdot \mathtt{B}}\ \textcircled{1}[OR\text{-}LHS]$$

(I) :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathtt{tL{<}3} \wedge \mathtt{tL^1{+}tL^2{=}tL} \wedge \mathtt{tR{=}tR^1{+}tR^2} \wedge \mathtt{tL^1{=}tR^1} \wedge \mathtt{tL^2{=}tR^2} \Rightarrow \mathtt{tR{<}4}}{\mathtt{tL{<}3} \wedge (\mathtt{A^{\star}\#tL^2})\cdot \mathtt{B} \sqsubseteq \mathtt{tR{<}4} \wedge (\mathtt{A^{\star}\#tR^2})\cdot \mathtt{B} \quad (\ddagger)}\ \textcircled{8}[REOCCUR]}{\mathtt{tL{<}3} \wedge \cancel{\mathtt{A\#tL^1}} \cdot \mathtt{A^{\star}\#tL^2}\cdot\mathtt{B} \sqsubseteq \mathtt{tR{<}4} \wedge \cancel{\mathtt{A\#tR^1}} \cdot \mathtt{A^{\star}\#tR^2}\cdot\mathtt{B}}\ \textcircled{7}[UNFOLD]}{\mathtt{tL{<}3} \wedge (\mathtt{A\#tL^1} \cdot \mathtt{A^{\star}\#tL^2})\cdot\mathtt{B} \sqsubseteq \mathtt{tR{<}4} \wedge (\mathtt{A\#tR^1} \cdot \mathtt{A^{\star}\#tR^2})\cdot\mathtt{B}}\ \textcircled{6}[UNFOLD]\ \pi_u{:}\mathtt{tL^1{=}tR^1}}{\mathtt{tL{<}3} \wedge (\mathtt{A^{\star}\#tL})\cdot \mathtt{B} \sqsubseteq \mathtt{tR{<}4} \wedge (\mathtt{A^{\star}\#tR})\cdot \mathtt{B} \quad (\ddagger)}\ \textcircled{5}[SPLIT]\,\mathtt{tL^1{+}tL^2{=}tL} \wedge \mathtt{tR^1{+}tR^2{=}tR}}{}$$

isomorphic with one of the the previous step, marked using $(\ddagger)$. Hence we apply the rule $[REOCCUR]$ to prove it with a succeed side constraints entailment.

5.6.4 Discussion.

Our implementation is the first tool that proves the inclusion of symbolic TAs, which is considered significant because it overcomes the following main limitations of traditional timed model checking: i) TAs cannot be used to specify/verify incompletely specified systems (i.e., whose timing constants have yet to be known) and hence cannot be used in early design phases; ii) verifying a system with a set of timing constants usually requires enumerating all of them if they are supposed to be integer-valued; iii) TAs cannot be used to verify systems with timing constants to be taken in a real-valued dense interval.

Besides, our results echo the insights from prior TRS-based works [SC20; AM95; AMR09; KT14a; Hov12; Bjø+01; KMP00; ÖM02; Ölv00], which suggest that TRS is a better average-case algorithm than those based on the comparison of automata. That is because *it only constructs automata as far as it needs*, which makes it more efficient when disproving incorrect specifications, as it can disprove it earlier without constructing the whole automata. In other words, the more incorrect specifications are, the more efficient a TRS is.

5.7 Summary

This work provides an alternative approach for verifying real-time systems, where temporal behaviors are reasoned at the source level, and the specification expressiveness goes beyond traditional Timed Automata. We define the novel effect logic *TimEffs*, to capture real-time behavioral patterns and temporal properties. We demonstrate how to build axiomatic semantics (or rather an effects system) for C^t via timed-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 proposes the first algebraic TRS for solving inclusion relations between timed specifications.

Limitations And Future Work. Our TRS is incomplete, meaning there exist valid inclusions which will be disproved in our system. That is mainly because of the insufficient unification in favor of achieving automation. We also foresee the possibilities of adding other logics into our existing trace-based temporal logic, such as separation logic for verifying heap-manipulating distributed programs.

Chapter 6

Continuation Based Effects (*ContEffs*)

Although effect handlers offer a versatile abstraction for user-defined effects, they produce complex and less restricted execution traces due to the composable non-local control flow mechanisms. This paper is interested in the temporal behaviors of effect sequences, such as unhandled effects, termination of the communication, safety, fairness, etc. Specifically, this chapter proposes a novel effect logic *ContEffs*, to write precise and modular specifications for programs in the presence of user-defined effect handlers and primitive effects. As a second contribution, we devise a forward verifier together with a fixpoint calculator to infer the behaviors of such programs. Lastly, our automated verification framework provides a purely algebraic *term-rewriting system* (TRS) as the back-end solver, efficiently checking the entailments between *ContEffs* assertions.

To demonstrate the feasibility, we prototype a verification system where zero-shot, one-shot, and multi-shot continuations coexist; prove its correctness; present experimental results; and report on case studies.

6.1 Introduction

User-defined effects and effect handlers are advertised and advocated as a relatively easy-to-understand and modular approach to delimited control. They offer the ability to suspend and resume computations, allowing information to be transmitted both ways. More specifically, an effect handler resembles an exception handler, i.e., control is transferred to an enclosing handler. Unlike the exception handlers, the key difference is that effects handlers have access to a continuation. By invoking this continuation, the handler can communicate a reply to the suspended computation and resume its execution.

For example, `effect Yield : int -> unit`, declares the `Yield` effect, to be used in the *generator functions*. When it is performed, the program suspends its current execution and returns the yielded `int` value to the handler. Such usages separate the logic, e.g., iterating a list, from the effectful operations, such as "printing on the console" or "sending an element to a consumer", thereby improving code reuse and memory efficiency. Functions perform effects without needing to know how the handlers are implemented, and the computation may be enclosed by different handlers that handle the same effect differently.

Recently, effect handlers are found in several research programming languages, such as Eff [BP15], Frank [Con+20], Links [HLA20], Multicore OCaml [Siv+21], and Scala [BSO20], etc. There is a growing need for programmers and researchers to reason about the combination of primitive effects and user-defined handlers. In particular, we are interested in the techniques for inferring and verifying temporal behaviors of such non-local control flows, which have not been extensively studied. Here, we tackle the following verification challenges:

1. The coexistence of zero-shot, one-shot and multi-shot continuations. The design decisions of various implementations [Lei14a; Siv+21] and verification solutions [Lan98; dVP21] diverge upon the question that, should it be permitted or forbidden to invoke a captured continuation more than once? Here, our forward reasoning rules shows the generality to incorporate both one-shot and multi-shot continuations. Furthermore, it naturally supports reasoning on exceptions by treating them as *zero-shot*, i.e., that abandon the continuations completely.

2. Non-terminating behaviors. Figure 6.1 presents the so-called "recursive cow" program drawn from the benchmark [Git22], which looks like it is terminating but it actually cycles. Function `f()` performs the predefined effect `Foo`; then `loop()` handles effect `Foo` by resuming a closure which in turn performs `Foo` when applied.

```
effect Foo : (unit -> unit)

let f() = perform Foo ()

let loop()
= match f () with
| _ -> () (* normal return *)
| effect Foo k -> continue k (fun () -> perform Foo ())
```

Figure 6.1: A loop caused by the effects handler.

With higher-order effect signatures and in the setting of deep handlers[1], the communications between the computation and handlers potentially lead to infinite traces. It is useful yet challenging to automatically infer/verify the termination of the communication. Here, we devise *ContEffs*, i.e., extended regular expressions with arithmetic constraints, to provide more precise specifications by integrating: $\star$ for finite traces; ω for infinite traces; ∞ for possibly finite or infinite traces.

3. Linear temporal properties. For decades monads have dominated the scene of pure functional programming with effects, and the recent popularization of algebraic effects and handlers promises to change the landscape. However, with rapid change also comes confusion. In monads, the effectful behavior is defined in *bind* and *return*, statically determining the behavior inside the *do* block. Whereas algebraic effects call effectful operations with no inherent behavior. Instead, the behavior is determined dynamically by the encompassing handler. Although this gives greater flexibility in the composition of effectful code, it requires further specifications and verification to enforce the temporal requirements.

In this work, *ContEffs* smoothly encode and go beyond the linear temporal logic (LTL). For examples: *"Effect **A** will never be followed by effect **B**"* is a fairness property, and it is expressed as: $(\_^\star \cdot \mathbf{A} \cdot \overline{\mathbf{B}})^\star$, where _ is a wildcard matching to any events; $\star$ denotes a repeated pattern; $\overline{\mathbf{B}}$ denotes the negation of an effect $\mathbf{B}$.

[1] A deep handler is persistent: after it has handled one effect, it remains installed, as the topmost frame of the captured continuation [HL18; KLO13].

"Function `send(int n)` *terminates when* `n` *is non-negative, otherwise it does not terminate"* is expressed as: $n{\geq}0{\wedge}(\_)^{\star} \vee n{<}0{\wedge}(\_)^{\omega}$, which is beyond LTL [2].

Having *ContEffs* as the specification language, we are interested in the following verification problem: Given a program $\mathcal{P}$, and a temporal property Φ', does $\Phi^{\mathcal{P}} \sqsubseteq \Phi'$ hold[3]? 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'.

To effectively check $\Phi^{\mathcal{P}} \sqsubseteq \Phi'$, we deploy a purely algebraic TRS inspired by Antimirov and Mosses's algorithm [AM95], which was originally designed for deciding the inequalities of regular expressions. Our TRS shows the ability to solve inclusions beyond the expressiveness of finite-state automata, also suggests that it is a better average-case algorithm than those based on automata theory.

We aim to lay the foundation for a practical verification system that is precise, concise, and modular to prove temporal properties of effectful programs. To the best of the author's knowledge, this work is the first to provide an extensive temporal verification framework for programs with user-defined effects and handlers. We summarize our main contributions as follows:

1. **The Continuous Effect (*ContEffs*):** We define the syntax and semantics of *ContEffs*, to be the specification language, which captures the temporal behaviors of given higher-order programs with algebraic effects.

2. **Front-End Effects Inference:** Targeting a ML-like language with the presence of algebraic effects [Siv+21; Nan+18], we establish a set of forward rules, to compositionally infer the program's temporal behaviors. The forward reasoning process makes use of a fixpoint calculator and the back-end solver TRS.

3. **The Term Rewriting System (TRS):** To check the entailments (i.e., the language inclusion relation) between two *ContEffs*, we present the rewriting rules, to prove the inferred effects against given temporal specifications.

[2]The classic LTL does not distinguish the termination of traces.

[3]The inclusion notation $\sqsubseteq$ is formally defined in Definition 26.

4. **Implementation and Evaluation:** We prototype the proposed verification system based on the latest Multicore OCaml (4.12.0) implementation. We prove its correctness and present case studies investigating *ContEffs*' expressiveness and the potential for various extensions.

6.2 Language and Specifications

This section first introduces the target language and then depict the temporal specification language which supports *ContEffs*.

6.2.1 The Target Language

Syntax. We target a minimal, ML-like (typed, higher-order, call-by-value) core pure language, defined in Figure 6.2 Here, c, x and $\mathbf{A}$ are meta-variables ranging respectively over integer constants, variables, and labels of effects.

A program $\mathcal{P}$ comprises a list of effect declarations *eff** and a list of function definitions *fun**; the $*$ superscript denotes a finite, possibly empty list of items. Programs are typed according to basic types τ. Each function fun has a name mn, an expression body e, and pre and postcondition Φ_{pre} and Φ_{post} (the syntax of effect specifications Φ is given in Figure 6.4). Constructs like sequencing are defined via elaboration to more primitive forms.

6.2.1.1 Operational Semantics of λ_h.

As shown in Figure 6.3, the reduction rules up to those for *match* are standard. Matching on a pure value results in the body of the always-present *return* handler being executed, with x bound to the value. The next two cases define how effects are performed and handled, but before covering them, we first explain how the expression *perform* $\mathbf{A}(v, \lambda x \Rightarrow e)$ works informally: it performs the effect $\mathbf{A}$ (e.g. a shared-memory read) with argument v (e.g. the memory location to be read). The result value of the effect (e.g. the contents of the memory location) is then bound to x and evaluation resumes with the continuation e. Note that how exactly the

$$
\begin{array}{llcl}
(\mathit{Program}) & \mathcal{P} & ::= & \mathit{eff}^* \ \mathit{fun}^* \\
(\mathit{Effect\ Declarations}) & \mathit{eff} & ::= & \mathbf{A} : \tau \\
(\mathit{Function\ Definition}) & \mathit{fun} & ::= & \tau \ mn \ (\tau \ v) \ [\mathbf{req} \ \Phi_{pre} \ \mathbf{ens} \ \Phi_{post}] \ \{e\} \\
(\mathit{Types}) & \tau & ::= & \mathit{bool} \mid \mathit{int} \mid \mathit{unit} \mid \tau_1 \rightarrow \tau_2 \\
(\mathit{Values}) & v & ::= & c \mid x \mid \lambda x \Rightarrow e \\
(\mathit{Handler}) & h & ::= & (\mathit{return} \ x \mapsto e \ \mathbf{|} \ \mathit{ocs}) \\
(\mathit{Operation\ Cases}) & \mathit{ocs} & ::= & \emptyset \mid \{\mathit{effect} \ \mathbf{A}(x, \kappa) \mapsto e\} \uplus \mathit{ocs} \\
(\mathit{Expressions}) & e & ::= & v \mid v_1 \ v_2 \mid \mathit{let} \ x{=}v \ \mathit{in} \ e \mid \mathit{if} \ v \ \mathit{then} \ e_1 \ \mathit{else} \ e_2 \\
& & \mid & \mathit{perform} \ \mathbf{A}(v, \lambda x \Rightarrow e) \mid \mathit{match} \ e \ \mathit{with} \ h \mid \mathit{resume} \ v
\end{array}
$$

$(\mathit{Selected\ Elaborations})$

$$
\begin{array}{rcl}
e_1; e_2 & \Longrightarrow & \mathit{let} \ (){=}e_1 \ \mathit{in} \ e_2 \\
e_1 \ e_2 & \Longrightarrow & \mathit{let} \ f{=}e_1 \ \mathit{in} \ \mathit{let} \ x{=}e_2 \ \mathit{in} \ (f \ x) \\
\mathit{perform} \ \mathbf{A}(e_1, \lambda y \Rightarrow e_2) & \Longrightarrow & \mathit{let} \ x{=}e_1 \ \mathit{in} \ \mathit{perform} \ A(x, \lambda y \Rightarrow e_2) \\
\mathit{let} \ x{=}\mathit{perform} \ \mathbf{A}(v, \lambda y \Rightarrow e_1) \ \mathit{in} \ e_2 & \Longrightarrow & \mathit{perform} \ \mathbf{A}(v, \lambda y \Rightarrow \mathit{let} \ x{=}e_1 \ \mathit{in} \ e_2) \\
\mathit{perform} \ \mathbf{A}(v) & \Longrightarrow & \mathit{perform} \ \mathbf{A}(v, \lambda x \Rightarrow x)
\end{array}
$$

$c \in \mathbb{Z} \cup \mathbb{B} \cup \mathbf{unit}$ $\qquad x, y, mn, \kappa \in \mathbf{var}$ $\qquad \mathbf{A} \in \Sigma$

Figure 6.2: Syntax of λ_h, a minimal, ML-like language with user-defined effects and handlers.

read is *implemented* is defined by handlers which enclose the *perform*.

With that in mind, there are two cases when matching on an effectful expression. If the effect **A** is handled by an appropriate case in an enclosing handler, both value and continuation are substituted into the body of the case – note that the continuation contains an identical handler (making the enclosing handler *deep*). Otherwise, if the effect is unhandled, reduction proceeds with the current *match* "pushed" into the continuation, to handle subsequent *performs*.

$$(\textit{Evaluation contexts}) \quad E ::= \textit{box} \mid \textit{let } x{=}E \textit{ in } e \mid \textit{match } E \textit{ with } h$$

$$\begin{aligned}
(\textit{Reduction rules}) \qquad\qquad E[e_1] &\longrightarrow E[e_2] \text{ if } e_1 \longrightarrow e_2 \\
\textit{let } x{=}v \textit{ in } e &\longrightarrow e[v/x] \\
(\lambda x \Rightarrow e)\ v &\longrightarrow e[v/x] \\
\textit{if true then } e_1 \textit{ else } e_2 &\longrightarrow e_1 \\
\textit{if false then } e_1 \textit{ else } e_2 &\longrightarrow e_2 \\
\textit{match } v \textit{ with } h &\longrightarrow e[v/x] \quad \textit{if } (\textit{return } x \mapsto e) \in h \\
\textit{match } (\textit{perform } \mathbf{A}(v, \lambda y \Rightarrow e_1)) \textit{ with } h &\longrightarrow e_2[v/x][(\lambda y \Rightarrow \textit{match } e_1 \textit{ with } h)/\kappa] \\
& \qquad \textit{if } (\textit{effect } \mathbf{A}(x, \kappa) \mapsto e_2) \in h \\
\textit{match } (\textit{perform } \mathbf{A}(v, \lambda y \Rightarrow e_1)) \textit{ with } h &\longrightarrow \textit{perform } \mathbf{A}(v, \lambda y \Rightarrow \textit{match } e_1 \textit{ with } h) \\
& \qquad \textit{if } \mathbf{A} \notin h
\end{aligned}$$

Figure 6.3: Evaluation contexts and reduction rules

6.2.2 The Specification Language

Syntax. We enrich a Hoare-style verification system with effect specifications, using the notation $\{\mathbf{req}\ \Phi_{pre}\ \mathbf{ens}\ \Phi_{post}\}$ for function pre and postcondition. As defined in Figure 3.2, Φ is a set of disjunctive tuples including a pure formula π, an event sequence θ, and a return value v.

$\mathbf{A}$ is an effect label drawn from Σ, a finite set of user-defined effect labels. A *parameterized label* is an effect label together with a value argument v. An *event ev* is an assertion about the (non-)occurrence of an individual, *handled* effect.

Placeholders Q stand for *traces* (sequences of events). The two kinds of placeholders are *unhandled* effects $l!$, which may give rise to further effects upon being handled, and $l?(v)$, which describes the trace that results when l is resumed with a higher-order function, and this function is applied to v. Placeholders enable modular verification, allowing higher-order *perform* sites to be described independently of any particular handler. They are only instantiated while verifying handlers, using the fixed-point reasoning (subsection 6.3.2).

$$
\begin{array}{rcll}
(\mathit{ContEffs}) & \Phi & ::= & \bigvee(\pi, \theta, v) \\
(\mathit{Parameterized\ Label}) & l & ::= & \Sigma(v) \\
(\mathit{Event\ Sequences}) & \theta & ::= & \bot \mid \epsilon \mid ev \mid Q \mid \theta_1{\cdot}\theta_2 \mid \theta_1{\vee}\theta_2 \mid \theta^{\star} \mid \theta^{\infty} \mid \theta^{\omega} \\
(\mathit{Single\ Events}) & ev & ::= & \_ \mid l \mid \bar{l} \\
(\mathit{Placeholders}) & Q & ::= & l! \mid l?(v) \\
(\mathit{Pure\ formulae}) & \pi & ::= & \mathit{True} \mid \mathit{False} \mid R(t_1, t_2) \mid \pi_1{\wedge}\pi_2 \mid \pi_1{\vee}\pi_2 \\
 & & & \mid \neg\pi \mid \pi_1{\Rightarrow}\pi_2 \\
(\mathit{Terms}) & t & ::= & n \mid x \mid t_1{+}t_2 \mid t_1{-}t_2
\end{array}
$$

$x \in \mathbf{var}$ (*Finite Kleene Star*) $\star$ (*Finite/Infinite*) ∞ (*Infinite*) ω

Figure 6.4: Syntax of *ContEffs*.

Effect sequences θ can be constructed by *false* ($\bot$); the empty trace ϵ; a single event ev; a placeholder Q; a sequence concatenation $\theta_1{\cdot}\theta_2$; and sequence disjunction $\theta_1{\vee}\theta_2$. Effect sequences can be also constructed by $\star$, representing finite (zero or more) repetition of a trace; by ω, representing an infinite repetition of a trace; or by ∞, representing an over-approximation of both finite and infinite possibilities [LS07]. Although $\theta^{\star}$ and θ^{ω} are subsumed by θ^{∞}, integrating all of the operators makes the specification language more flexible and precise. It also makes the logic conveniently subsume traditional linear temporal logics.

Pure formulae π are Presburger arithmetic formulae. $R(t_1, t_2)$ is a binary relation ($R \in \{=, >, <, \geq, \leq\}$). Terms are constant integer values n, integer variables x, and additions and subtractions of terms.

6.2.2.1 Semantic Model of *ContEffs*.

To define the model, *var* is the set of program variables, *val* is the set of primitive values, α is the set of concrete events drawn from single events $\mathbf{l}$ or placeholders Q. Let $\mathcal{E}, \varphi \models \Phi$ denote the *models* relation, i.e., the context $\mathcal{E}$ and linear temporal events φ satisfy the effect specification Φ, where $\mathcal{E}$ records the stack status and the bindings from variables to placeholders, $\mathcal{E} \triangleq var{\rightarrow}(val \cup Q)$; and φ is a list of events, $\varphi \triangleq [\alpha]$.

$\mathcal{E}, \varphi \models \Phi$	*iff*	$\exists(\pi, \theta, v) \in \Phi.\ \mathcal{E}, \varphi \models (\pi, \theta, v)$
$\mathcal{E}, \varphi \models (\pi, \epsilon)$	*iff*	$[\![\pi]\!]_{\mathcal{E}}{=}True\ and\ \varphi{=}[]$
$\mathcal{E}, \varphi \models (\pi, \_)$	*iff*	$[\![\pi]\!]_{\mathcal{E}}{=}True\ and\ \exists l \in \Sigma(v),\ \varphi{=}[l]$
$\mathcal{E}, \varphi \models (\pi, l)$	*iff*	$[\![\pi]\!]_{\mathcal{E}}{=}True\ and\ \varphi{=}[l]$
$\mathcal{E}, \varphi \models (\pi, \bar{l})$	*iff*	$[\![\pi]\!]_{\mathcal{E}}{=}True\ and\ \mathcal{E}, \varphi \models \bigvee j\ where\ j \in \Sigma(v)\ and\ j{\neq}l$
$\mathcal{E}, \varphi \models (\pi, Q)$	*iff*	$[\![\pi]\!]_{\mathcal{E}}{=}True\ and\ \varphi{=}[Q]$
$\mathcal{E}, \varphi \models (\pi, \theta_1{\cdot}\theta_2)$	*iff*	$\exists \varphi_1, \varphi_2.\ \varphi{=}\varphi_1{+\!+}\varphi_2\ and\ \mathcal{E}, \varphi_1{\models}(\pi, \theta_1)\ and\ \mathcal{E}, \varphi_2{\models}(\pi, \theta_2)$
$\mathcal{E}, \varphi \models (\pi, \theta_1{\vee}\theta_2)$	*iff*	$\mathcal{E}, \varphi \models (\pi, \theta_1)\ or\ \mathcal{E}, \varphi \models (\pi, \theta_2)$
$\mathcal{E}, \varphi \models (\pi, \theta^\star)$	*iff*	$\mathcal{E}, \varphi \models (\pi, \epsilon)\ or\ \mathcal{E}, \varphi \models (\pi, \theta \cdot \theta^\star)$
$\mathcal{E}, \varphi \models (\pi, \theta^\infty)$	*iff*	$\mathcal{E}, \varphi \models (\pi, \theta^\star)\ or\ \mathcal{E}, \varphi \models (\pi, \theta^\omega)$
$\mathcal{E}, \varphi \models (\pi, \theta^\omega)$	*iff*	$\mathcal{E}, \varphi \models (\pi, \theta \cdot \theta^\omega)$
$\mathcal{E}, \varphi \models (False, \bot)$	*iff*	*false*

Figure 6.5: Semantics of *ContEffs*.

Since the return value in effect specifications is irrelevant to the semantic model, we define $\mathcal{E}, \varphi \models (\pi, \theta)$ to be $\mathcal{E}, \varphi \models (\pi, \theta, v)$ for some return value v.

The semantics of effect sequences is defined in Figure 6.5: $[]$ is an empty sequence; $[l]$ is the sequence that contains one parameterized label l; ++ is the append operation of two effect sequences; and $\bigvee j$ is a disjunction of parameterized labels j. Comparisons between labels use simple lexical equivalence.

6.2.3 Instrumented Semantics of the Target Language.

To facilitate the soundness proof in Theorem 8 for the verification rules presented in section 6.3, we also define an instrumented reduction relation $\xrightarrow{i}$, which operates on program states of the form $\lceil e, \mathcal{E}, \varphi \rceil$, where an expression is associated with a context and the trace of effects performed in the course of its execution. $\xrightarrow{i}{}^*$ denotes its reflexive, transitive closure. Here, given $e \longrightarrow e'$ and a most general high-order effects signature $(\mathbf{A} : \tau_1{\rightarrow}(\tau_2{\rightarrow}\tau_3)) \in \mathcal{P}$:

$$\frac{e = v_1\ v_2 \qquad \mathcal{E}(v_1){=}\mathbf{A}(v)?}{\lceil e, \mathcal{E}, \varphi\rceil \xrightarrow{i} \lceil e', \mathcal{E}, \varphi{+\!\!+}[\mathbf{A}(v)?(v_2)]\rceil}\ [\mathit{Inst\text{-}App}]$$

$$\frac{e = \mathit{let}\ x{=}v\ \mathit{in}\ e_1}{\lceil e, \mathcal{E}, \varphi\rceil \xrightarrow{i} \lceil e', (x{\mapsto}v){::}\mathcal{E}, \varphi\rceil}\ [\mathit{Inst\text{-}Bind}]$$

$$\frac{e = \mathit{match\ perform}\ \mathbf{A}(v, \lambda x{\Rightarrow}e_1)\ \mathit{with}\ h \qquad A \notin h}{\lceil e, \mathcal{E}, \varphi\rceil \xrightarrow{i} \lceil e', (x{\mapsto}\mathbf{A}(v)?){::}\mathcal{E}, \varphi{+\!\!+}[\mathbf{A}(v)!]\rceil}\ [\mathit{Inst\text{-}Escape}]$$

$$\frac{e = \mathit{match\ perform}\ \mathbf{A}(v, \lambda x{\Rightarrow}e_1)\ \mathit{with}\ h \qquad A \in h}{\lceil e, \mathcal{E}, \varphi\rceil \xrightarrow{i} \lceil e', \mathcal{E}, \varphi{+\!\!+}[\mathbf{A}(v)]\rceil}\ [\mathit{Inst\text{-}Caught}]$$

6.3 Automated Forward Verification

The automated verification system consists of a Hoare-style forward verifier and a TRS. The input of the forward verifier is a target program annotated with temporal specifications written in *ContEffs*.

We formalize a set of syntax-directed forward verification rules for the core language, in Figure 6.6. $\mathcal{P}$ denotes the program being checked. With pre/postcondition declared for each function in $\mathcal{P}$, we apply modular verification to a function's body using Hoare-style triples $\mathcal{E} \vdash \langle\Phi\rangle\ e\ \langle\Phi'\rangle$ where $\mathcal{E}$ is the context; if Φ describes the effects which have been performed since the beginning of $\mathcal{P}$, if e terminates, Φ' describes the effects that will have been performed after.

6.3.1 Forward Verification Rules

In [*FV-Fun*], the rule computes the final effects Φ from the function body, and checks the inclusion between Φ and the declared specifications. Note that for succinctness, the user-provided Φ_{post} only denotes the *extension* of the effects from executing the function body. Formally, $\mathcal{E} \vdash \langle\Phi_{pre}\rangle\ e\ \langle\Phi_{pre} \cdot \Phi_{post}\rangle$ is a valid triple.

$$\frac{\mathcal{E} \vdash \langle \Phi_{pre} \rangle\ e\ \langle \Phi \rangle \qquad \Phi \sqsubseteq \Phi_{pre} \cdot \Phi_{post}}{\vdash \tau\ mn\ (\tau\ v)\ [\mathbf{req}\ \Phi_{pre}\ \mathbf{ens}\ \Phi_{post}]\ \{e\}}\ [FV\text{-}Fun]$$

$$\frac{\Phi' {=} \Phi \cdot \mathbf{A}(v)! \qquad (x {\mapsto} \mathbf{A}(v)?){::}\mathcal{E} \vdash \langle \Phi' \rangle\ e\ \langle \Phi'' \rangle}{\mathcal{E} \vdash \langle \Phi \rangle\ perform\ \mathbf{A}(v, \lambda x.e)\ \langle \Phi'' \rangle}\ [FV\text{-}Perform]$$

$$\frac{\mathcal{E}(v_1) {=} \tau\ mn\ (\tau\ v)\ [\mathbf{req}\ \Phi_{pre}\ \mathbf{ens}\ \Phi_{post}]\ \{e\} \qquad \Phi \sqsubseteq \Phi_{pre}[v_2/v]}{\mathcal{E} \vdash \langle \Phi \rangle\ v_1\ v_2\ \langle \Phi \cdot \Phi_{post}[v_2/v] \rangle}\ [FV\text{-}Call]$$

$$\frac{\mathcal{E}(v_1) {=} \mathbf{l}? \qquad \theta' {=} \mathbf{l}?(v_2)}{\mathcal{E} \vdash \langle \Phi \rangle\ v_1\ v_2\ \langle \Phi \cdot \theta' \rangle}\ [FV\text{-}App] \qquad \frac{\Phi' = \{(\pi, \theta, v') \mid (\pi, \theta, v) \in \Phi\}}{\mathcal{E} \vdash \langle \Phi \rangle\ v'\ \langle \Phi' \rangle}\ [FV\text{-}Value]$$

$$\frac{\mathcal{E} \vdash \langle \Phi {\wedge} (v {=} true) \rangle\ e_1\ \langle \Phi_1 \rangle \qquad \mathcal{E} \vdash \langle \Phi {\wedge} (v {=} false) \rangle\ e_2\ \langle \Phi_2 \rangle}{\mathcal{E} \vdash \langle \Phi \rangle\ if\ v\ then\ e_1\ else\ e_2\ \langle \Phi_1 \rangle \cup \langle \Phi_2 \rangle}\ [FV\text{-}If\text{-}Else]$$

$$\frac{(x {\mapsto} v){::}\mathcal{E} \vdash \langle \Phi \rangle\ e\ \langle \Phi' \rangle}{\mathcal{E} \vdash \langle \Phi \rangle\ let\ x {=} v\ in\ e\ \langle \Phi' \rangle}\ [FV\text{-}Let]$$

$$\frac{\begin{array}{c}\mathcal{E} \vdash \langle (True, \epsilon, ()) \rangle\ e\ \langle \Phi' \rangle \qquad \Phi' {=} \{(\pi, \theta \cdot \heartsuit, v) \mid (\pi, \theta, v) \in \Phi'\} \\ \mathcal{E}, h \vdash_{fix} \Phi' \rightsquigarrow \Phi_{fix}\end{array}}{\mathcal{E} \vdash \langle \Phi \rangle\ match\ e\ with\ h\ \langle \Phi \cdot \Phi_{fix} \rangle}\ [FV\text{-}Match]$$

Figure 6.6: Selected Forward Rules for *ContEffs*

Definition 21 (*ContEffs* Concatenation)**.** Given two *ContEffs* Φ_1 and Φ_2, $\Phi_1 \cdot \Phi_2 = \{(\pi_1 \wedge \pi_2, \theta_1 \cdot \theta_2, v_2) \mid (\pi_1, \theta_1, v_1) \in \Phi_1, (\pi_2, \theta_2, v_2) \in \Phi_2\}$

$[FV\text{-}Perform]$ concatenates a placeholder to the current effects, where $\Phi \cdot \mathbf{A}(v)! \equiv \{(\pi, \theta \cdot \mathbf{A}(v)!, v) \mid (\pi, \theta, v) \in \Phi\}$, then extends the environment by binding x to $\mathbf{A}(v)?$, referring to the resumed value of performing $\mathbf{A}(v)$. For applications $v_1\ v_2$, if v_1 is a function definition with annotated specifications, $[FV\text{-}Call]$ checks whether the instantiated precondition of callee, $\Phi_{pre}[v_2/v]$, is satisfied by the current effect state, then it obtains the next effects state by concatenating the instantiated postcondition, $\Phi_{post}[v_2/v]$, to the current effect state; if v_1 maps to $\mathbf{l}?$, $[FV\text{-}App]$ concatenates $\mathbf{l}?(v_2)$ into the current effect state, referring to the effects generated by applying v_2 to the value resumed from performing $\mathbf{l}$. $[FV\text{-}Value]$ updates the current return value. [*FV-If-Else*] unions the effects from both branches, where $\Phi \wedge \pi' \equiv \{(\pi \wedge \pi',\ \theta, v) \mid (\pi, \theta, v) \in \Phi\}$. [*FV-Let*] extends $\mathcal{E}$ with x binding to v. [*FV-Match*] computes the effects of e using the initial state $\{(\mathit{True}, \epsilon, ())\}$, then deploys the fixpoint algorithm to compute the final effects after been handled by h. $\heartsuit$ is a special event marking the end of the traces, which is essential while distinguishing the zero/one/multi-shots continuations.

6.3.2 Fixpoint Computation.

Given any effect Φ and a fixed handler $\mathcal{H}$, the relation $\mathcal{E}, \mathcal{H} \vdash_{\mathit{fix}} \Phi' \rightsquigarrow \Phi_{\mathit{fix}}$ concludes the fixpoint effects Φ_{fix} via the following rule:

$$\frac{\forall(\pi, \theta, v) \in \Phi.\ \|\mathcal{E}, \epsilon, \mathcal{H}\| \vdash_{\mathit{fix}} (\pi, \theta, v) \rightsquigarrow \Phi'}{\mathcal{E}, \mathcal{H} \vdash_{\mathit{fix}} \Phi \rightsquigarrow \bigcup \Phi'}[\mathit{Fix\text{-}Disj}]$$

For all the execution tuples (π, θ, v), given $\mathcal{H}$, it is reduced to Φ'. Their relation is captured by: $\|\mathcal{E}, \theta_{his}, \mathcal{H}\| \vdash_{\mathit{fix}} \Phi \rightsquigarrow \Phi'$, where θ_{his} is the history trace and initialized by ϵ. The final result Φ_{fix} is a union set of all the Φ'.

Rule [*Fix-Normal*] is applied when the trace is reduced to the ending mark $\heartsuit$, which indicates that the execution of the handled program is finished. In this case, the resulting state Φ' is achieved by computing the strongest post condition of $e_{ret}[v/x]$ from the starting state $\langle(\pi, \theta_{his}, v)\rangle$.

$$\frac{(\mathit{return}\ x \mapsto e_{ret}) \in \mathcal{H} \qquad \mathcal{E} \vdash \langle(\pi, \theta_{his}, v)\rangle\ e_{ret}[v/x]\ \langle\Phi'\rangle}{\|\mathcal{E}, \theta_{his}, \mathcal{H}\| \vdash_{\mathit{fix}} (\pi, \heartsuit, v) \rightsquigarrow \Phi'}[\mathit{Fix\text{-}Normal}]$$

Rule [*Fix-Unfold-Skip*] is applied when the starting events α are handled effects ev, or placeholders corresponding to the effects cannot be handled by the current handler. In this case, the rule simple achieves α into the history context θ_{his} and continues to reason about the tail of the trace, i.e., θ.

$$\frac{\alpha \in \{ev, \mathbf{l}!, \mathbf{l}?(v')\}\ (\mathbf{l} \notin \mathcal{H}) \qquad \|\mathcal{E}, \theta_{his}{\cdot}f, \mathcal{H}\| \vdash_{\mathit{fix}} (\pi, \theta, v) \rightsquigarrow \Phi'}{\|\mathcal{E}, \theta_{his}, \mathcal{H}\| \vdash_{\mathit{fix}} (\pi, \alpha \cdot \theta, v) \rightsquigarrow \Phi'}[\mathit{Fix\text{-}Unfold\text{-}Skip}]$$

Rule [*Fix-Unfold-Skip*] is applied when the starting events α are unhandled effects $\mathbf{l}!$ which can be handled by the current handler. In this case, the rule uses the relation $\mathcal{E}, \mathcal{H}, \mathcal{D} \vdash_h \langle\Phi\rangle\ e\ \langle\Phi'\rangle$ to reason about the instantiated handling code $e[v/x, \kappa/\mathit{resume}]$. Note that, here the rule achieves l into the history context, indicating that the emission $l!$ is handled.

$$\frac{\alpha \in \{\mathbf{l}!\} \qquad (\mathit{effect}\ \mathbf{A}(x,\kappa) \mapsto e) \in \mathcal{H} \qquad (\mathbf{l}{=}\mathbf{A}(v)) \quad \mathcal{E}, \mathcal{H}, \theta \vdash_h \langle(\pi, \theta_{\mathit{his}} \cdot \mathbf{l}, v)\rangle\ e[v/x, \kappa/\mathit{resume}]\ \langle\Phi\rangle}{\|\mathcal{E}, \theta_{\mathit{his}}, \mathcal{H}\| \vdash_{\mathit{fix}} (\pi, \alpha \cdot \theta, v) \rightsquigarrow \Phi'}[\mathit{Fix\text{-}Unfold\text{-}Handle}]$$

6.3.3 Reasoning in the Handling Program.

Rules for $\mathcal{E}, \mathcal{H}, \mathcal{D} \vdash_h \langle\Phi\rangle\ e\ \langle\Phi'\rangle$ (where $\mathcal{D}$ stands for the not-yet-handled continuation, of the type θ) are mostly similar to the top-level forward relation $\mathcal{E} \vdash \langle\Phi\rangle\ e\ \langle\Phi'\rangle$, except for the rules:

$$\frac{\forall(\pi, \theta, v) \in \Phi \qquad \|\mathcal{E}, \theta, \mathcal{H}\| \vdash_{\mathit{fix}} (\pi, \mathcal{D}[v'/\mathbf{l}?], v) \rightsquigarrow \Phi' \quad \mathcal{E}, \mathcal{H}, \mathcal{D} \vdash_h \langle\Phi'\rangle\ e\ \langle\Phi''\rangle}{\mathcal{E}, \mathcal{H}, \mathcal{D} \vdash_h \langle\Phi\rangle\ \mathit{let}\ x{=}\kappa\ v'\ \mathit{in}\ e\ \langle\Phi''\rangle}[\mathit{Handle\text{-}Resume}]$$

$$\frac{\Phi'{=}\{(\pi, \theta, v') \mid (\pi, \theta, v) \in \Phi\}}{\mathcal{E}, \mathcal{H}, \mathcal{D} \vdash_h \langle\Phi\rangle\ v'\ \langle\Phi'\rangle}[\mathit{Handle\text{-}Value}]$$

In [*Handle-Resume*], all the placeholders $\mathbf{l}?$ shown in the continuation $\mathcal{D}$ can be finally instantiated by κ's argument value, v'. Possible loops are also captured in this step, when $\mathcal{D}[v'/\mathbf{l}?]$ produces the effects' emissions which has already been handled. The final result Φ'' is achieved by reasoning e after handling the rest continuation. Note that if the handling program directly returns a single value, the rule [*Handle-Value*] abandons the continuation $\mathcal{D}$ completely, which is intuitively why we are able to handle exceptions (zero-shot continuations). The rest of the rules and a demonstration example are presented in section C.1.

Lemma 1 (Soundness of the Fixpoint Computation). *Given an effect Φ, with the environment $\mathcal{E}$ and handler $\mathcal{H}$. Φ_{fix} is the updated version of Φ, where all Φ's* *placeholders – which can be handled by $\mathcal{H}$ – are handled as $\mathcal{H}$ defines.*

Formally, $\forall\mathcal{E}, \forall\mathcal{H}, \forall\Phi$, *if* $\mathcal{E}, \mathcal{H} \vdash_{fix} \Phi \rightsquigarrow \Phi_{fix}$ *is valid, then:*

when Φ *is a set,* $\Phi_{fix}=\{\|\mathcal{E}, \epsilon, \mathcal{H}\| \vdash_{fix} (\pi, \theta, v) \rightsquigarrow \Phi' \mid (\pi, \theta, v) \in \Phi\}$; (1)

when $\Phi=(\pi, \theta, v), \alpha=fst(\theta), \theta_{his}$ *is the handled trace,*

if $\alpha=\heartsuit{:}([x{\mapsto}v]){::}\mathcal{E}{\vdash}\langle(\pi, \theta_{his}, v)\rangle e_{ret}\langle\Phi'\rangle$ *is valid, given* $(return\ x{\mapsto}e_{ret}){\in}\mathcal{H}$; (2)

if $\alpha{\in}\{ev, \boldsymbol{l}!, \boldsymbol{l}?(v')\}$ $(\boldsymbol{l}{\notin}\mathcal{H})$: $\|\mathcal{E}, \theta_{his} \cdot \alpha, \mathcal{H}\| \vdash_{fix} (\pi, \mathcal{D}_\alpha(\theta), v) \rightsquigarrow \Phi'$ *is valid;* (3)

if $\alpha{\in}\{\boldsymbol{l}!\}$ $(\boldsymbol{l}{\in}\mathcal{H})$: $(x{\mapsto}v){::}\mathcal{E}, \mathcal{H}, \mathcal{D}_\alpha(\theta) \vdash_h \langle(\pi, \theta_{his} \cdot \boldsymbol{l}, v)\rangle\ e\ \langle\Phi'\rangle$ *is valid,*
given $(effect\ \boldsymbol{A}(x, \kappa) \mapsto e){\in}\mathcal{H}$. (4)

Proof. See section C.2. □

Theorem 8 (Soundness of Verification Rules)**.** *Given an expression e, the linear effect trace produced by the real execution of e satisfies the effect specification derived via the forward verification rules.*

$$\textit{Formally}, \forall e, \forall\mathcal{E}, \forall\varphi, \forall\Phi\ \textit{given}\ \lceil e, \mathcal{E}, \varphi\rceil \xrightarrow{i}{}^{*} \lceil v, \mathcal{E}', \varphi'\rceil\ \textit{and}\ \mathcal{E} \vdash \langle\Phi\rangle\ e\ \langle\Phi'\rangle,$$
$$\textit{if}\ \mathcal{E}, \varphi \models \Phi\ \textit{then}\ \mathcal{E}', \varphi' \models \Phi'.$$

Proof. By induction on the structure of e.

- When $e{=}(v_1\ v_2)$:

$$\frac{e = v_1\ v_2 \qquad \mathcal{E}(v_1){=}\mathbf{A}(v)?}{\lceil e, \mathcal{E}, \varphi\rceil \xrightarrow{i} \lceil e', \mathcal{E}, \varphi{+}{+}[\mathbf{A}(v)?(v_2)]\rceil}[\textit{Inst-App}] \quad \frac{\mathcal{E}(v_1){=}\mathbf{l}? \qquad \theta'{=}\mathbf{l}?(v_2)}{\mathcal{E} \vdash \langle\Phi\rangle\ v_1 v_2\ \langle\Phi \cdot \theta'\rangle}[\textit{FV-App}]$$

By the rule [*Inst-App*], we have: $\lceil e, \mathcal{E}, \varphi\rceil \xrightarrow{i} \lceil v', \mathcal{E}, \varphi{+}{+}[\mathbf{A}(v)?(v_2)]\rceil$. Next since $\mathcal{E}, \varphi \models \Phi$, we can obtain $\mathcal{E}, \varphi{+}{+}[\mathbf{A}(v)?(v_2) \models \Phi \cdot \mathbf{A}(v)?(v_2)$. And in the same $\mathcal{E}$, $\mathcal{E}(v_1){=}\mathbf{A}(v)?{=}\mathbf{l}?$. Therefore $\mathcal{E}, \varphi{+}{+}[\mathbf{A}(v)?(v_2) \models \Phi{\cdot}\mathbf{l}?(v_2)$, where the goal $\mathcal{E}', \varphi' \models \Phi'$ is proved.

- When $e{=}(let\ x{=}v\ in\ e_1)$:

$$\frac{e = let\ x{=}v\ in\ e_1}{\lceil e, \mathcal{E}, \varphi \rceil \xrightarrow{i} \lceil e_1, (x{\mapsto}v){::}\mathcal{E}, \varphi \rceil}[\textit{Inst-Bind}] \qquad \frac{(x{\mapsto}v){::}\mathcal{E} \vdash \langle\Phi\rangle\ e_1\ \langle\Phi'\rangle}{\mathcal{E} \vdash \langle\Phi\rangle\ let\ x{=}v\ in\ e_1\ \langle\Phi'\rangle}[\textit{FV-Let}]$$

By the rule [*Inst-Bind*], we have: $\lceil e, \mathcal{E}, \varphi \rceil \xrightarrow{i} \lceil e_1, (x{\mapsto}v){::}\mathcal{E}, \varphi \rceil \xrightarrow{i}{}^{*} \lceil v', \mathcal{E}', \varphi' \rceil$. By the rule [*FV-Let*], we have $(x{\mapsto}v){::}\mathcal{E} \vdash \langle\Phi\rangle\ e_1\ \langle\Phi'\rangle$. Given, $\mathcal{E}, \varphi \models \Phi$, therefore $(x{\mapsto}v){::}\mathcal{E}, \varphi \models \Phi$ By the inductive hypothesis on e_1, we get $\mathcal{E}', \varphi' \models \Phi'$, where the goal $\mathcal{E}', \varphi' \models \Phi'$ is proved.

- When $e = match\ perform\ \mathbf{A}(v, \lambda x{\Rightarrow}e_1)\ with\ h$ and $A \notin h$:

$$\frac{e = match\ perform\ \mathbf{A}(v, \lambda x{\Rightarrow}e_1)\ with\ h \qquad A \notin h}{\lceil e, \mathcal{E}, \varphi \rceil \xrightarrow{i} \lceil e_1, (x{\mapsto}\mathbf{A}(v)?){::}\mathcal{E}, \varphi{+\!+}[\mathbf{A}(v)!] \rceil}\ [\textit{Inst-Escape}]$$

By the rule [*Inst-Escape*], we have: $\lceil e, \mathcal{E}, \varphi \rceil \xrightarrow{i} \lceil e_1, (x{\mapsto}\mathbf{A}(v)?){::}\mathcal{E}, \varphi{+\!+}[\mathbf{A}(v)!] \rceil \xrightarrow{i}{}^{*} \lceil v', \mathcal{E}', \varphi' \rceil$. Then by the rule [*FV-Match*] $\mathcal{E} \vdash \langle\Phi\rangle\ e\ \langle\Phi'\rangle$, and $\Phi'{=}\Phi \cdot \Phi_{\mathit{fix}}$, where

$$\frac{\begin{array}{c}\mathcal{E} \vdash \langle(\mathit{True}, \epsilon, ())\rangle\ perform\ \mathbf{A}(v, \lambda x{\Rightarrow}e_1)\ \langle\Phi'\rangle \\ \Phi'{=}\{(\pi, \theta \cdot \heartsuit, v) \mid (\pi, \theta, v) \in \Phi'\} \qquad \mathcal{E}, h \vdash_{\mathit{fix}} \Phi' \rightsquigarrow \Phi_{\mathit{fix}}\end{array}}{\mathcal{E} \vdash \langle\Phi\rangle\ match\ perform\ \mathbf{A}(v, \lambda x{\Rightarrow}e_1)\ with\ h\ \langle\Phi \cdot \Phi_{\mathit{fix}}\rangle}\ [\textit{FV-Match}]$$

$\mathcal{E}, h \vdash_{\mathit{fix}} (\mathbf{A}(v)! \cdot \Phi_{e_1}){\rightsquigarrow}\Phi_{\mathit{fix}}$, and $\mathbf{A}(v)?{::}\mathcal{E}{\vdash}\langle\mathbf{A}(v)!\rangle\ e_1\ \langle\mathbf{A}(v)!{\cdot}\Phi_{e_1}\rangle$, by [*FV-Perform*].

$$\frac{\Phi'{=}\Phi \cdot \mathbf{A}(v)! \qquad (x{\mapsto}\mathbf{A}(v)?){::}\mathcal{E} \vdash \langle\Phi'\rangle\ e_1\ \langle\Phi''\rangle}{\mathcal{E} \vdash \langle\Phi\rangle\ perform\ \mathbf{A}(v, \lambda x{\Rightarrow}e_1)\ \langle\Phi''\rangle}\ [\textit{FV-Perform}]$$

Then by rules [*Fix-Disj*] and [*Fix-Unfold-Skip*], $\Phi_{\mathit{fix}}{=}\mathbf{A}(v)! \cdot \Phi_{e_1}^{\mathit{fix}}$.

$$\frac{\forall(\pi, \theta, v) \in \Phi.\ \|\mathcal{E}, \epsilon, \mathcal{H}\| \vdash_{\mathit{fix}} (\pi, \theta, v) \rightsquigarrow \Phi'}{\mathcal{E}, \mathcal{H} \vdash_{\mathit{fix}} \Phi \rightsquigarrow \bigcup \Phi'}[\textit{Fix-Disj}]$$

$$\frac{\alpha \in \{ev, \mathbf{l}!, \mathbf{l}?(v')\}\ (\mathbf{l} \notin \mathcal{H}) \qquad \|\mathcal{E}, \theta_{his}{\cdot}f, \mathcal{H}\| \vdash_{\mathit{fix}} (\pi, \theta, v) \rightsquigarrow \Phi'}{\|\mathcal{E}, \theta_{his}, \mathcal{H}\| \vdash_{\mathit{fix}} (\pi, \alpha \cdot \theta, v) \rightsquigarrow \Phi'}[\textit{Fix-Unfold-Skip}]$$

Given, $\mathcal{E}, \varphi \models \Phi$, therefore $(x{\mapsto}\mathbf{A}(v)?){::}\mathcal{E}, \varphi{+\!+}[\mathbf{A}(v)!] \models \Phi \cdot \mathbf{A}(v)!$. By the inductive hypothesis on e_1, we get $\mathcal{E}', \varphi' \models \Phi \cdot \mathbf{A}(v)! \cdot \Phi^{\mathit{fix}}_{e_1}$, where the goal $\mathcal{E}', \varphi' \models \Phi'$ is proved.

- When $e = \mathit{match\ perform}\ \mathbf{A}(v, \lambda x{\Rightarrow}e_1)\ \mathit{with}\ h$ and $A \in h$:

$$\frac{e = \mathit{match\ perform}\ \mathbf{A}(v, \lambda x{\Rightarrow}e_1)\ \mathit{with}\ h \qquad A \in h}{\lceil e, \mathcal{E}, \varphi \rceil \xrightarrow{i} \lceil e_1, \mathcal{E}, \varphi{+\!+}[\mathbf{A}(v)] \rceil}\ [\mathit{Inst\text{-}Caught}]$$

By the rule $[\mathit{Inst\text{-}Caught}]$, we have: $\lceil e, \mathcal{E}, \varphi \rceil \xrightarrow{i} \lceil e_1, \mathcal{E}, \varphi{+\!+}[\mathbf{A}(v)] \rceil \xrightarrow{i}{}^{*} \lceil v', \mathcal{E}', \varphi' \rceil$. Then by the rule $[\mathit{FV\text{-}Match}]$ $\mathcal{E} \vdash \langle \Phi \rangle\ e\ \langle \Phi' \rangle$, and $\Phi'{=}\Phi \cdot \Phi_{\mathit{fix}}$, where $\mathcal{E}, h \vdash_{\mathit{fix}} (\mathbf{A}(v)! \cdot \Phi_{e_1}) \rightsquigarrow \Phi_{\mathit{fix}}$, and $\mathbf{A}(v)?{::}\mathcal{E} \vdash \langle \mathbf{A}(v)! \rangle\ e_1\ \langle \mathbf{A}(v)! \cdot \Phi_{e_1} \rangle$, by $[\mathit{FV\text{-}Perform}]$. Then by rules $[\mathit{Fix\text{-}Disj}]$ and $[\mathit{Fix\text{-}Unfold\text{-}Handle}]$, $\Phi_{\mathit{fix}}{=}\mathbf{A}(v) \cdot \Phi^{\mathit{fix}}_{e_1}$.

$$\frac{\alpha \in \{\mathbf{l}!\} \qquad (\mathit{effect}\ \mathbf{A}(x, \kappa) \mapsto e) \in \mathcal{H} \qquad (\mathbf{l}{=}\mathbf{A}(v)) \qquad \mathcal{E}, \mathcal{H}, \theta \vdash_h \langle (\pi, \theta_{\mathit{his}} \cdot \mathbf{l}, v) \rangle\ e[v/x, \kappa/\mathit{resume}]\ \langle \Phi \rangle}{\| \mathcal{E}, \theta_{\mathit{his}}, \mathcal{H} \| \vdash_{\mathit{fix}} (\pi, \alpha \cdot \theta, v) \rightsquigarrow \Phi'}\ [\mathit{Fix\text{-}Unfold\text{-}Handle}]$$

Given, $\mathcal{E}, \varphi \models \Phi$, therefore $\mathcal{E}, \varphi{+\!+}[\mathbf{A}(v)] \models \Phi \cdot \mathbf{A}(v)$. By the inductive hypothesis on e_1, we get $\mathcal{E}', \varphi' \models \Phi \cdot \mathbf{A}(v) \cdot \Phi^{\mathit{fix}}_{e_1}$, where the goal $\mathcal{E}', \varphi' \models \Phi'$ is proved.

□

6.4 Temporal Verification via a TRS

A TRS checks inclusions among logical terms, via an iterated process of checking the inclusions of their *partial derivatives* [Ant95]. It is triggered i) prior to function calls for the precondition checking; and ii) at the end of verifying a function for postcondition checking. Given two effects Φ_1 and Φ_2, the TRS decides if the inclusion $\Phi_1 \sqsubseteq \Phi_2$ is valid. During the rewriting process, the inclusions are of the form $\Omega \vdash \Phi_1 \sqsubseteq^{\theta} \Phi_2$, a shorthand for: $\Omega \vdash \theta \cdot \Phi_1 \sqsubseteq \theta \cdot \Phi_2$. To prove such inclusions amounts to checking whether all the possible traces in the antecedent Φ_1 are legitimately allowed in the possible traces from the consequent Φ_2. Ω is the proof context, i.e., a set of effect inclusion hypotheses, and θ is the history of effects from the antecedent that have been used to match the effects from the consequent. The inclusion checking is initially invoked with $\Omega=\{\}$ and $\theta=\epsilon$.

6.4.0.1 Effect Disjunction.

An inclusion with a disjunctive antecedent succeeds if both disjunctions entail the consequent. An inclusion with a disjunctive consequent succeeds if the antecedent entails any of the disjunctions. Note that the event sequences' inclusion checking is irrelevant to the returning values.

$$\frac{\Omega \vdash (\pi,\theta) \sqsubseteq \Phi' \ and \ \ \Omega \vdash \Phi \sqsubseteq \Phi'}{\Omega \vdash (\pi,\theta,v) :: \Phi \sqsubseteq \Phi'}\ [LHS\text{-}OR] \qquad \frac{\Omega \vdash (\pi,\theta) \sqsubseteq (\pi',\theta') \ or \ (\pi,\theta) \sqsubseteq \Phi'}{\Omega \vdash (\pi,\theta) \sqsubseteq (\pi',\theta',v') :: \Phi'}\ [RHS\text{-}OR]$$

Next we provide definitions and implementations of auxiliary functions[4] *Nullable*(δ), *Infinitable*($\varkappa$), *First*(*fst*) and *Derivative*(D) respectively. Intuitively, the Nullable function $\delta(\Phi)$ returns a boolean value indicating whether θ contains the empty trace; the Infinitable function $\varkappa(\theta)$ returns a boolean value indicating whether θ is possibly infinite; the First function $fst(\theta)$ computes possible initial elements of θ; and the Derivative function $D_\alpha(\theta)$ eliminates an event α[5] from the head of θ and returns what remains.

Definition 22 (Nullable)**.** Given any sequence θ, we recursively define $\delta(\theta)$[6]:

$$\delta(\epsilon){=}\delta(\theta^\star){=}\delta(\theta^\infty){=}true \quad \delta(\theta_1{\cdot}\theta_2){=}\delta(\theta_1){\wedge}\delta(\theta_2) \quad \delta(\theta_1{\vee}\theta_2){=}\delta(\theta_1){\vee}\delta(\theta_2)$$

Definition 23 (Infinitiable)**.** Given any sequence θ, we recursively define $\varkappa(\theta)$[7]:

$$\varkappa(\theta^\infty){=}\varkappa(\theta^\omega){=}true \qquad \varkappa(\theta_1{\cdot}\theta_2){=}\varkappa(\theta_1){\vee}\varkappa(\theta_2) \qquad \varkappa(\theta_1{\vee}\theta_2){=}\varkappa(\theta_1){\vee}\varkappa(\theta_2)$$

[4]The definitions are extended from [Ant95], to be able to deal with placeholders and infinite traces, proposed in this work.

[5]α could be a single label l, a negated label $\bar{\mathrm{l}}$, a wildcard _, or a placeholder Q.

[6]*false* for unmentioned constructs

[7]*false* for unmentioned constructs

Definition 24 (First)**.** Let $fst(\theta)$ be the set of initial elements derivable from sequence represents all the traces contained in θ.

$$fst(\bot){=}fst(\epsilon){=}\{\} \quad fst(ev){=}\{ev\} \quad fst(Q){=}\{Q\} \quad fst(\theta_1{\vee}\theta_2){=}fst(es_1)\cup fst(es_2)$$

$$fst(\theta_1{\cdot}\theta_2){=}\begin{cases} fst(es_1)\cup fst(es_2) & \mathit{if}\ \delta(\theta_1){=}true \\ fst(\theta_1) & \mathit{if}\ \delta(\theta_1){=}\mathit{false} \end{cases} \qquad fst(\theta^\star){=}fst(\theta^\infty){=}fst(\theta^\omega){=}fst(\theta)$$

Definition 25 (Partial Derivative)**.** The partial derivative $D_\alpha(\theta)$ of effects θ with respect to an element α computes the effects for the left quotient, $\alpha^{\text{-1}}[\![\theta]\!]$[8].

$$D_\alpha(\bot){=}\bot \qquad D_\alpha(\epsilon){=}\bot \qquad D_\alpha(\theta_1{\vee}\theta_2){=}D_\alpha(\theta_1)\vee D_\alpha(\theta_2) \qquad D_\alpha(\theta^\star){=}D_\alpha(\theta)\cdot\theta^\star$$

$$D_\alpha(ev){=}\begin{cases} \epsilon & \mathit{if}\ \alpha\subseteq ev \\ \bot & else \end{cases} \qquad D_\alpha(Q){=}\begin{cases} \epsilon & \mathit{if}\ \alpha{=}Q \\ \bot & else \end{cases} \qquad D_\alpha(\theta^\infty){=}D_\alpha(\theta)\cdot\theta^\infty$$

$$D_\alpha(\theta_1{\cdot}\theta_2){=}\begin{cases} (D_\alpha(\theta_1)\cdot\theta_2)\vee D_\alpha(\theta_2) & \mathit{if}\ \delta(\theta_1){=}true \\ D_\alpha(\theta_1)\cdot\theta_2 & \mathit{if}\ \delta(\theta_1){=}\mathit{false} \end{cases} \qquad D_\alpha(\theta^\omega){=}D_\alpha(\theta)\cdot\theta^\omega$$

6.4.1 Rewriting Rules.

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

$$\frac{}{\Omega\vdash(\pi_1,\bot)\sqsubseteq(\pi_2,\theta)}\ [\mathit{Bot\text{-}LHS}] \qquad \frac{\theta\neq\bot}{\Omega\vdash(\pi_1,\theta)\not\sqsubseteq(\pi_2,\bot)}\ [\mathit{Bot\text{-}RHS}]$$

2. **Disprove (Heuristic Refutation).** These rules are used to disprove the inclusions when the antecedent obviously contains more traces than the consequent. Here *nullable* and *infinitiable* witness the empty trace and infinite traces respectively.

$$\frac{\delta(\theta_1)\wedge\neg\delta(\theta_2)}{\Omega\vdash(\pi_1,\theta_1)\not\sqsubseteq(\pi_2,\theta_2)}\ [\mathit{Dis\text{-}Nullable}] \qquad \frac{\varkappa(\theta_1)\wedge\neg\varkappa(\theta_2)}{\Omega\vdash(\pi_1,\theta_1)\not\sqsubseteq(\pi_2,\theta_2)}\ [\mathit{Dis\text{-}Infinitable}]$$

[8] $[\![\theta]\!]$ represents all the traces contained in θ.

3. **Prove.** We use the rule [*Reoccur*] to prove an inclusion when there exist inclusion hypotheses in the proof context Ω, 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 prove it valid.

$$\frac{(\pi_1, \theta_1) \sqsubseteq (\pi_3, \theta_3) \in \Omega \quad (\pi_3, \theta_3) \sqsubseteq (\pi_4, \theta_4) \in \Omega \quad (\pi_4, \theta_4) \sqsubseteq (\pi_2, \theta_2) \in \Omega}{\Omega \vdash (\pi_1, \theta_1) \sqsubseteq (\pi_2, \theta_2)} \; [\mathit{Reoccur}]$$

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 effects F, which are all the possible initial elements from the antecedent. Secondly, we obtain a new proof context Ω' by adding the current inclusion, as an inductive hypothesis, into the current proof context Ω. Thirdly, we iterate each element $\alpha \in F$, and compute the partial derivatives (*next-state* effects) of both the antecedent and consequent with respect to α. The proof of the original inclusion succeeds if all the derivative inclusions succeed.

$$\frac{F = fst(\theta_1) \qquad \pi_1 \Rightarrow \pi_2 \qquad \forall \alpha \in F.\ (\theta_1 \sqsubseteq \theta_2) :: \Omega \vdash D_\alpha(\theta_1) \sqsubseteq D_\alpha(\theta_2)}{\Omega \vdash (\pi_1, \theta_1) \sqsubseteq (\pi_2, \theta_2)} \; [\mathit{Unfold}]$$

Theorem 9 (TRS-Termination)**.** *The rewriting system TRS is terminating.*

Proof. See section C.3. □

Theorem 10 (TRS-Soundness)**.** *Given an inclusion $\Phi_1 \sqsubseteq \Phi_2$, if the TRS returns $TRUE$ when proving $\Phi_1 \sqsubseteq \Phi_2$, then $\Phi_1 \sqsubseteq \Phi_2$ is valid.*

Proof. See section C.4. □

6.4.2 Discussion: highlighting the novelty.

This work extends the original Antimirov algorithm [AM95], to prove the inclusions between the more expressive specifications formulae, *ContEffs*, which contains placeholders for both triggering effects and receiving responses from the handler. The placeholders are necessary and will be instantiated by the concrete effect handlers, enabling a modular verification approach for effect handlers. Another contribution of this work is using the concept of derivatives for effectively reasoning about multi-shot effect handlers.

6.5 Demonstration Examples

6.5.1 A Sense of *ContEffs* in File I/O

We define Hoare-triple style specifications, marked in `lavender`, for each program, which lead to a compositional verification strategy, where temporal reasoning can be done locally. We model an abstract form of file I/O in Figure 6.7. Effects `Open` and `Close` are both declared to be *performed* with a value of type `int`, indexing the operated file.

```
effect Open : int -> unit
effect Close: int -> unit

let open_file n
/*@ req _^* @*/
/*@ ens Open(n)! @*/
= perform (Open n)
let close_file n
/*@ req _^*.Open(n)! .(~Close(n)!)^* @*/
/*@ ens Close(n)! @*/
= perform (Close n)

let file_9 ()
/*@ req emp @*/
/*@ ens Open(9)!.Close(9)!@*/
= open_file 9;
  close_file 9
```

Figure 6.7: A simple file I/O example.

Function `open_file` takes an argument `n`. Its precondition uses a wildcard '_' under a Kleene star, indicating that any finite number/kind of effects is allowed to have occurred *before* the call to `open_file`. In other words, it is always possible to open a file. Its postcondition indicates that it performs the effect `Open` applied with `n`. The precondition of `close_file` states that it can only be called after such a history trace where the `n`th file has been requested to be `Open`ed, and not been requested to be `Close`ed[9].

We use `.` to denote the sequential composition of effect traces, `!` denotes the emission of a certain effect, and `~` denotes the negation of a certain effect label.

The precondition of `file_9`: `emp`, stands for an empty trace, which means no history trace is *allowed* by the calling site of function `file_9`. We formalize this idea of *being allowed* as an entailment relation between specifications in section 6.4. The verification fails when the real implementation violates the specifications.

6.5.2 Effects Inferences via a Fixpoint Calculation

We continue to examine a variant of the so-called "recursive cow" benchmark program [Git22] in Figure 6.8, which generates an infinite trace.

```
effect Goo : (unit -> unit)

let f_g ()
/*@ req _^* @*/
/*@ ens Foo!.Goo!.Foo?() @*/
= let f = perform Foo in
  let g = perform Goo in
  f () (* g is abandoned *)

let loop ()
/*@ req _^* @*/
/*@ ens _^*.(Foo.Goo)^w @*/
= match f_g () with
| _ -> ()
| effect Foo k -> continue k (fun () -> perform Goo ())
| effect Goo k -> continue k (fun () -> perform Foo ())
```

Figure 6.8: Another Loop caused by the effects handler.

[9] `close_file`'s precondition prevents closing files that are not opened. The constraints can be strengthened or loosened as needed. For example, to prevent opening a file which is already opened, we need to strengthen `open_file`'s precondition accordingly

The handling of effects `Foo` and `Goo` are notable because their resumption carry closures back to the suspended points, which in turn perform effects when fully applied. We argue informally that `loop` is non-terminating. This is because the invocation of `f_g ()` performs `Foo`, which obtains the resumed closure (defined in line 15) and stores it in the variable `f`. Then the application to `f` in turn performs `Goo`. The performing of `Goo` brings us to the handler at line 18, which resumes a closure that performs `Foo` when applied. The resulting postcondition, deploys the ω operator, states that `loop` *finally* performs an infinite succession of alternating `Foo` and `Goo` effects. In fact, our fixpoint calculator computes the final effects for `loop` as $\mathtt{Foo} \cdot \mathtt{Goo} \cdot \mathtt{Goo} \cdot \mathtt{Foo} \cdot (\mathtt{Goo} \cdot \mathtt{Foo})^{\omega}$, which entails the declared postcondition (c.f. Figure 6.9).

Loops like these between handler and callee are generally caused by performing effects in the recovery closure when handling an effect, that results in a cycle back to that same (deep) handler. However, resuming with a closure, is a useful pattern for inverting control between handler and callee, does give rise to this trap. Our fixpoint analysis and specifications are aimed at capturing such situations, which have not been extensively explored.

6.5.3 The TRS: to prove effects inclusions

The rewriting system proposed by Antimirov and Mosses [AM95] decides inequalities of regular expressions (REs) through an iterated process of checking the inequalities of their *partial derivatives* [Ant95]. There are two basic rules: `[DISPROVE ]`, which infers false from trivially inconsistent inequalities; and `[UNFOLD]`, which applies Theorem 1 to generate new inequalities.

Similarly, we formally define the inclusion of *ContEffs* in Definition 26.

Definition 26 (*ContEffs* Inclusion)**.** Given Σ is a finite set of alphabet, for two effects (π_1, θ_1) and (π_2, θ_2), their inclusion is defined as:

$$(\pi_1, \theta_1) \sqsubseteq (\pi_2, \theta_2) \;\Leftrightarrow\; \pi_1 {\Rightarrow} \pi_2 \;\wedge\; (\forall \alpha \in \Sigma).\; \alpha^{\text{-}}(\theta_1) \sqsubseteq \alpha^{\text{-}}(\theta_2) \qquad .$$

Next we present the effects inclusion, generated from Figure 6.8, proving process for the post condition checking in Figure 6.9 Termination is guaranteed because the set of derivatives to be considered is finite, and possible cycles are detected using

$$\mathsf{Foo} \cdot (\mathsf{Goo} \cdot \mathsf{Foo})^{\omega} \sqsubseteq \_^{\star} \cdot (\mathsf{Goo} \cdot \mathsf{Foo})^{\omega}\ \spadesuit \vee (\mathsf{Foo} \cdot \mathsf{Goo})^{\omega}$$

$$\text{---}\ \text{fst} = \text{Goo}$$

$$\underline{\mathsf{Goo}} \cdot \mathsf{Foo} \cdot (\mathsf{Goo} \cdot \mathsf{Foo})^{\omega} \sqsubseteq \_^{\star} \cdot (\mathsf{Goo} \cdot \mathsf{Foo})^{\omega} \vee \mathsf{Goo} \cdot (\mathsf{Foo} \cdot \mathsf{Goo})^{\omega}$$

$$\text{---}\ \text{fst} = \text{Foo}$$

$$\underline{\mathsf{Foo}} \cdot (\mathsf{Goo} \cdot \mathsf{Foo})^{\omega} \sqsubseteq \_^{\star} \cdot (\mathsf{Goo} \cdot \mathsf{Foo})^{\omega}\ \spadesuit$$

$$\text{---}\ \text{fst} = \text{Goo}$$

$$\underline{\mathsf{Goo}} \cdot \mathsf{Foo} \cdot (\mathsf{Goo} \cdot \mathsf{Foo})^{\omega} \sqsubseteq \_^{\star} \cdot (\mathsf{Foo} \cdot \mathsf{Goo})^{\omega} \vee (\mathsf{Foo} \cdot \mathsf{Goo})^{\omega}$$

$$\text{---}\ \text{fst} = \text{Goo}$$

$$\underline{\mathsf{Goo}} \cdot \mathsf{Goo} \cdot \mathsf{Foo} \cdot (\mathsf{Goo} \cdot \mathsf{Foo})^{\omega} \sqsubseteq \_^{\star} \cdot (\mathsf{Foo} \cdot \mathsf{Goo})^{\omega} \vee \mathsf{Goo} \cdot (\mathsf{Foo} \cdot \mathsf{Goo})^{\omega}$$

$$\text{---}\ \text{fst} = \text{Foo}$$

$$\underline{\mathsf{Foo}} \cdot \mathsf{Goo} \cdot \mathsf{Goo} \cdot \mathsf{Foo} \cdot (\mathsf{Goo} \cdot \mathsf{Foo})^{\omega} \sqsubseteq \_^{\star} \cdot (\mathsf{Foo} \cdot \mathsf{Goo})^{\omega}$$

Figure 6.9: Proving the postcondition of the function *loop* in Figure 6.8.

memorization. We use ♠ to indicate such pairings. The rewriting rules are defined in section 6.4. In particular, the rule [*Reoccur*] finds the syntactic identity from the internal proof tree, for the current open goal [Bro05b].

6.6 Implementation and Evaluation

To show the feasibility of our approach, we have prototyped our automated verification system using OCaml. The arithmetic proof obligations generated by the TRS are discharged using Z3 [dMB08]. We prove termination and soundness of the TRS. We validate the front-end forward verifier against the latest Multicore OCaml (4.12.0) implementation for conformance.

This experiment is done without a baseline comparison because there are no existing tools for reasoning about the algebraic effects using temporal verification techniques, and our experimental results show that a modular and efficient temporal verification for user-defined effects and unrestricted effect handlers is achievable.

Table 6.1 presents the evaluation results of a microbenchmark, to demonstrate how verification scales with program size. We annotate 12 synthetic test programs with temporal specifications, half of which fail to verify. The experiments were done on a MacBook Pro with a 2.6 GHz 6-Core Intel Core i7 processor. The table records: **No.**, the index of the program; **LOC**, lines of code; **Forward(ms)**, forward reasoning time; **#Prop(✓)**, number of valid properties; **Avg-Prove(ms)**, average proving time for the valid properties; **#Prop(✗)**, number of invalid properties; and **Avg-Dis(ms)**, average disproving time for the invalid properties.

Table 6.1: Experimental Results for *ContEffs*.

No.	**LOC**	**Forward(ms)**	**#Prop(✓)**	**Avg-Prove(ms)**	**#Prop(✗)**	**Avg-Dis(ms)**
1	32	14.128	5	7.7786	5	6.2852
2	48	14.307	5	7.969	5	6.5982
3	71	15.029	5	7.922	5	6.4344
4	98	14.889	5	18.457	5	7.9562
5	156	14.677	7	10.080	7	4.819
6	197	15.471	7	8.3127	7	6.8101
7	240	18.798	7	18.559	7	7.468
8	285	20.406	7	23.3934	7	9.9086
9	343	26.514	9	22.5666	9	13.9667
10	401	26.893	9	18.3899	9	14.2169
11	583	49.931	14	17.203	15	14.4443
12	808	75.707	25	21.6795	24	13.9064

Discussion: Generally, forward reasoning and proving time increase linearly with program length. Furthermore, we notice that disproving times for invalid properties are consistently lower than those for proved properties, regardless of program complexity. This finding echos the insights from prior TRS-based works [SC20; AM95; AMR09; KT14a; Hov12], which suggest that TRS is a better average-case algorithm than those based on the comparison of automata.

A summary: A TRS is efficient because *it only constructs automata as far as it needs*, which makes it more efficient when disproving incorrect specifications, as we can disprove it earlier without constructing the whole automata. In other words, the more invalid inclusions are, the more efficient our solver is.

6.6.1 Case Studies

I. Encoding LTL. Classical LTL uses the temporal operators $\mathcal{G}$ ("globally") and $\mathcal{F}$ ("in the future"), which we also write $\Box$ and $\Diamond$, respectively; and introduced the concept of fairness, which places additional constraints on infinite paths. LTL was subsequently extended to include the $\mathcal{U}$ ("until") operator and the $\mathcal{X}$ ("next time") operator. As shown in Table 6.2, we encode these basic operators into our effects (here l, j are labels), making the specification more intuitive and readable, mainly when nested operators occur. Furthermore, by putting the effects in the precondition, our approach naturally subsumes *past-time LTL* along the way[10].

Table 6.2: Examples for converting LTL formulae into *ContEffs*.

$\Box l \equiv l^{\infty}$	$\Diamond l \equiv \_^{\star} \cdot l$	$l \,\mathcal{U}\, j \equiv l^{\star} \cdot j$	$l \rightarrow \Diamond j \equiv \neg l \vee \_^{\star} \cdot j$
$\mathcal{X} l \equiv \_ \cdot l$	$\Box\Diamond l \equiv (\_^{\star} \cdot l)^{\infty}$	$\Diamond\Box l \equiv \_^{\star} \cdot l^{\infty}$	$\Diamond l \vee \Diamond j \equiv \_^{\star} \cdot l \vee \_^{\star} \cdot j$

II. Encoding Exceptions. Exceptions are a special case of algebraic effects which never resume, and Figure 6.10 demonstrates how our framework soundly reasons about exceptions together with other kinds of effects. Here `raise()` performs `Exc` first, then does some other operations afterwards, represented by performing effect `Other`.

```
1  effect Exc: unit
2  effect Other: unit
3
4  let raise ()
5  /*@ req _^*  @*/
6  /*@ ens Exc!.Other! @*/
7  = perform Exc;
8    perform Other
```

```
9  let excHandler
10 /*@ req _^* @*/
11 /*@ ens Exc @*/
12 = match raise () with
13 | _ -> (* Abandoned *)
14 | effect Exc k -> ()
```

Figure 6.10: Encoding Exceptions using *ContEffs*.

The handler at line 15 discharges `Exc` and returns, leaving the continuation `k` completely unused. Our fixpoint calculator computes the final trace of `excHandler` as simply `Exc`. We observe that the handler defined in the normal return (line 14) will be completely abandoned – because execution flow does not go back to

[10] Our implementation supports specifications written in LTL formulae, by providing a translator from LTL to *ContEffs*. The translation schema is taken from [LS07].

`raise()` after handling `Exc`. The verified postcondition of `excHandler` matches how we would intuitively expect traditional exceptions to work[11].

6.7 Summary

This work is mainly motivated by *how to modularly specify and verify programs in the presence of both user-defined primitive effects and effect handlers.*

To provide a practical proposal to verify such higher-order programs with crucial temporal constraints, we present a novel effect logic, *ContEffs*, to specify user-defined effects and effect handlers. This logic enjoys two key benefits that enable modular reasoning: the placeholder operator and the disjunction of finite and infinite traces. We demonstrate several small but non-trivial case studies to show *ContEffs*' feasibility. Our code and specification are particularly compact and generic; furthermore, as far as we know, this is the first temporal specification and proof of correctness of control inversion with the presence of algebraic effects.

[11] In general, each procedure has a set of circumstances for which it will terminate *normally*. An exception breaks the normal flow (these circumstances) of execution and executes a pre-registered exception handler instead [Wik22b].

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This thesis introduces a novel temporal verification framework, which tackles possible insufficiencies of the existing automata-based temporal verification.

To emphasize the applicability of this framework, this thesis proposes novel effect logics: *DependentEffs*, *ASyncEffs*, *TimEffs* and *ContEffs* for different target languages: a C-like language, the full-featured Esterel, C^t and λ_h, varying in different programming domains: general effectful programs; preemptive asynchronous reactive programs; time-critical distributed programs; and programs with user-defined effects and handlers, respectively. Altogether we show that this framework is:

1. ***more modular***, because of its compositional verification strategy, where functions can be replaced by their already verified properties and temporal reasoning can be done locally and then combined to reason about the whole program;

2. ***finer-grained***, because of the expressive *effect logics*, which are designed based on regular expressions, but capable of capturing more detailed (domain-related) information, such as branching properties relying on the arithmetic constraints; dependent values to represent the number of trace repetition; real-time bounds; and effects emission and handling, etc; and

3. ***more efficient***, because of the deployed term rewriting systems, serving as the back-end solvers, which are based directly on constraint-solving techniques. The TRSs can be reasonably efficient in verifying systems consisting of many components

as it avoids the complex translation into automata and thus avoiding exploring the whole state-space, when possible.

It is also worth pointing out that the proposed framework attempts to conduct verification for the source code without any modeling phases. Although the formulated target languages abstract the implementation languages in a certain way, the practice is to attempt to verify the source code directly. For example, the work in Chapter 6 is built on top of the OCaml compiler and takes the source code as the direct input, which cannot be trivially achieved by the model-checking techniques.

7.1.1 Repositories of the Open-sourced Implementations

We have developed four related prototype systems for the four effect logics that we have proposed in this thesis. Their open-sourced implementations can be found here:

1. For *DependentEffs*:
   `https://github.com/songyahui/EFFECTS`
2. For *ASyncEffs*:
   `https://github.com/songyahui/SyncedEffects`
   `https://github.com/songyahui/Semantics_HIPHOP`
3. For *TimEffs*:
   `https://github.com/songyahui/Timed_Verification`
4. For *ContEffs*:
   `https://github.com/songyahui/AlgebraicEffect`

7.2 Future Work

From this thesis, one can continue with many possible directions:

1. **Temporal Verification with Spatial Information:** The works presented in this thesis dedicate to control propagation, where data variables and data-

manipulating primitives are mostly abstracted away. However, it can be difficult to prove many interesting properties without modeling data mutations and heap usage. Separation logic [OHe06] has been proposed as an extension of Hoare logic, providing precise and concise reasoning for pointer-based programs. One direct future work is to add heap abstractions to *ContEffs* and provide automated verification support for heap-manipulating programs with user-defined effects and handlers.

2. **Temporal Verification with Incorrectness Logic:** The works presented in this thesis are in the context of the classical over-approximation verification, which is intended to prove the absence of bugs. A recently advanced technique, *Incorrectness Logic* (IL) [OHe20], provides a theoretical foundation for bug-finding and has been proven to be practically successful in some systems, such as Infer [Fac22]. Temporal verification with IL is a promising direction to provide more efficient and practical support for various programming domains. A possible extension of this thesis is to specify bugs as post-conditions and deploy the under-approximation context in the forward rules; then, leveraging the current solvers (the TRSs), the automated verification tool becomes a proof engine for the presence of bugs.

3. **Program Synthesis via More Expressive Temporal logics:** Control synthesis for mobile robots under complex tasks, captured by linear temporal logic (LTL) formulas, builds upon either bottom-up approaches when independent LTL expressions are assigned to robots [BKV10; KFP07; UMB13] or top-down approaches when a global LTL formula describing a the collaborative task is assigned to a team of robots [CDB]. Therefore, it is worth exploring scalable control synthesis algorithms from logics beyond LTL, as we proposed in this thesis. Meanwhile, combining the finer-grained verification and synthesis techniques may lead to finer-grained program repair.

That said, the proposed temporal verification framework is a promising and flexible approach to provide more fine-grained verification support for different complex control-flow mechanisms, which are hard to model using existing techniques.

Bibliography

[Ali+18] S. Alimadadi, D. Zhong, M. Madsen, and F. Tip, "Finding broken promises in asynchronous javascript programs", *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, 162:1–162:26, 2018. [Online]. Available: `https://doi.org/10.1145/3276532`.

[AMR09] M. Almeida, N. Moreira, and R. Reis, "Antimirov and mosses's rewrite system revisited", *Int. J. Found. Comput. Sci.*, vol. 20, no. 4, pp. 669–684, 2009. [Online]. Available: `https://doi.org/10.1142/S0129054109006802`.

[AD94] R. Alur and D. L. Dill, "A theory of timed automata", *Theor. Comput. Sci.*, vol. 126, no. 2, pp. 183–235, 1994. [Online]. Available: `https://doi.org/10.1016/0304-3975(94)90010-8`.

[ANN99] T. Amtoft, H. R. Nielson, and F. Nielson, "Type and effect systems - behaviors for concurrency", Imperial College Press, 1999, ISBN: 978-1-86094-154-2.

[And+14] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, "Netkat: Semantic foundations for networks", in *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, S. Jagannathan and P. Sewell, Eds., ACM, 2014, pp. 113–126. [Online]. Available: `https://doi.org/10.1145/2535838.2535862`.

[Ant95] V. Antimirov, "Partial derivatives of regular expressions and finite automata constructions", in *Annual Symposium on Theoretical Aspects of Computer Science*, Springer, 1995, pp. 455–466.

[AM95] V. M. Antimirov and P. D. Mosses, "Rewriting extended regular expressions", *Theor. Comput. Sci.*, vol. 143, no. 1, pp. 51–72, 1995. [Online]. Available: `https://doi.org/10.1016/0304-3975(95)80024-4`.

[Ard22] Arduino, "Arduino", `https://create.arduino.cc/projecthub/projects/tags/control`, 2022.

[BC13] S. Balaguer and T. Chatain, "Avoiding shared clocks in networks of timed automata", *Log. Methods Comput. Sci.*, vol. 9, no. 4, 2013. [Online]. Available: `https://doi.org/10.2168/LMCS-9(4:13)2013`.

[BDF05] M. Bartoletti, P. Degano, and G. L. Ferrari, "Enforcing secure service composition", in *18th IEEE Computer Security Foundations Workshop, (CSFW-18 2005), 20-22 June 2005, Aix-en-Provence, France*, IEEE Computer Society, 2005, pp. 211–223. [Online]. Available: `https://doi.org/10.1109/CSFW.2005.17`.

[BP14] A. Bauer and M. Pretnar, "An effect system for algebraic effects and handlers", *Log. Methods Comput. Sci.*, vol. 10, no. 4, 2014. [Online]. Available: `https://doi.org/10.2168/LMCS-10(4:9)2014`.

[BP15] A. Bauer and M. Pretnar, "Programming with algebraic effects and handlers", *J. Log. Algebraic Methods Program.*, vol. 84, no. 1, pp. 108–123, 2015. [Online]. Available: `https://doi.org/10.1016/j.jlamp.2014.02.001`.

[Ben+03] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, "The synchronous languages 12 years later", *Proc. IEEE*, vol. 91, no. 1, pp. 64–83, 2003. [Online]. Available: `https://doi.org/10.1109/JPROC.2002.805826`.

[BLJ91] A. Benveniste, P. Le Guernic, and C. Jacquemot, "Synchronous programming with events and relations: The SIGNAL language and its semantics", *Sci. Comput. Program.*, vol. 16, no. 2, pp. 103–149, 1991. [Online]. Available: `https://doi.org/10.1016/0167-6423(91)90001-E`.

[Ber93] G. Berry, "Preemption in concurrent systems", in *Foundations of Software Technology and Theoretical Computer Science, 13th Conference, Bombay, India, December 15-17, 1993, Proceedings*, R. K. Shyamasundar, Ed., ser. Lecture Notes in Computer Science, vol. 761, Springer, 1993, pp. 72–93. [Online]. Available: `https://doi.org/10.1007/3-540-57529-4%5C_44`.

[Ber99] G. Berry, "The constructive semantics of pure Esterel-draft version 3", *Draft Version*, vol. 3, 1999.

[Ber+00] G. Berry, A. Bouali, X. Fornari, E. Ledinot, E. Nassor, and R. de Simone, "ESTEREL: a formal method applied to avionic software development", *Sci. Comput. Program.*, vol. 36, no. 1, pp. 5–25, 2000. [Online]. Available: `https://doi.org/10.1016/S0167-6423(99)00015-5`.

[BG92] G. Berry and G. Gonthier, "The esterel synchronous programming language: Design, semantics, implementation", *Sci. Comput. Program.*, vol. 19, no. 2, pp. 87–152, 1992. [Online]. Available: `https://doi.org/10.1016/0167-6423(92)90005-V`.

[BS20] G. Berry and M. Serrano, "Hiphop.js: (a)synchronous reactive web programming", in *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, A. F. Donaldson and E. Torlak, Eds., ACM, 2020, pp. 533–545. [Online]. Available: `https://doi.org/10.1145/3385412.3385984`.

[BV06] B. Berthomieu and F. Vernadat, "Time petri nets analysis with TINA", in *Third International Conference on the Quantitative Evaluation of Systems (QEST 2006), 11-14 September 2006, Riverside, California, USA*, IEEE Computer Society, 2006, pp. 123–124. [Online]. Available: `https://doi.org/10.1109/QEST.2006.56`.

[BKV10] A. Bhatia, L. E. Kavraki, and M. Y. Vardi, "Sampling-based motion planning with temporal goals", in *IEEE International Conference on Robotics and Automation, ICRA 2010, Anchorage, Alaska, USA, 3-*

7 May 2010, IEEE, 2010, pp. 2689–2696. [Online]. Available: https://doi.org/10.1109/ROBOT.2010.5509503.

[Bie+12] G. Bierman, C. Russo, G. Mainland, E. Meijer, and M. Torgersen, "Pause n play: Formalizing asynchronous c sharp", in *European Conference on Object-Oriented Programming*, Springer, 2012, pp. 233–257.

[Bie+19] D. Biernacki, M. Piróg, P. Polesiuk, and F. Sieczkowski, "Abstracting algebraic effects", *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–28, 2019.

[Bjø+01] N. S. Bjørner, Z. Manna, H. Sipma, and T. E. Uribe, "Deductive verification of real-time systems using step", *Theor. Comput. Sci.*, vol. 253, no. 1, pp. 27–60, 2001. [Online]. Available: https://doi.org/10.1016/S0304-3975(00)00088-8.

[BSO20] J. I. Brachthäuser, P. Schuster, and K. Ostermann, "Effekt: Capability-passing style for type-and effect-safe, extensible effect handlers in scala", *Journal of Functional Programming*, vol. 30, 2020.

[Bro+15] S. Broda, S. Cavadas, M. Ferreira, and N. Moreira, "Deciding synchronous kleene algebra with derivatives", in *Implementation and Application of Automata - 20th International Conference, CIAA 2015, Umeå, Sweden, August 18-21, 2015, Proceedings*, F. Drewes, Ed., ser. Lecture Notes in Computer Science, vol. 9223, Springer, 2015, pp. 49–62. [Online]. Available: https://doi.org/10.1007/978-3-319-22360-5%5C_5.

[Bro05a] J. Brotherston, "Cyclic proofs for first-order logic with inductive definitions", in *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference, TABLEAUX 2005, Koblenz, Germany, September 14-17, 2005, Proceedings*, B. Beckert, Ed., ser. Lecture Notes in Computer Science, vol. 3702, Springer, 2005, pp. 78–92. [Online]. Available: https://doi.org/10.1007/11554554%5C_8.

[Bro05b] J. Brotherston, "Cyclic proofs for first-order logic with inductive definitions", in *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, Springer, 2005, pp. 78–92.

[Bub+15] R. Bubel, C. C. Din, R. Hähnle, and K. Nakata, "A dynamic logic with traces and coinduction", in *Automated Reasoning with Analytic Tableaux and Related Methods - 24th International Conference, TABLEAUX 2015, Wrocław, Poland, September 21-24, 2015. Proceedings*, H. de Nivelle, Ed., ser. Lecture Notes in Computer Science, vol. 9323, Springer, 2015, pp. 307–322. [Online]. Available: https://doi.org/10.1007/978-3-319-24312-2%5C_21.

[Cal+09] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang, "Compositional shape analysis by means of bi-abduction", in *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Z. Shao and B. C. Pierce, Eds., ACM, 2009, pp. 289–300. [Online]. Available: https://doi.org/10.1145/1480881.1480917.

[CDB] Y. Chen, X. C. Ding, and C. Belta, "Synthesis of distributed control and communication schemes synthesis of distributed control and communication schemes from global ltl specifications", in *IEEE Conference on Decision and Control*, pp. 2718–2723.

[Chi+12] W.-N. Chin, C. David, H. H. Nguyen, and S. Qin, "Automated verification of shape, size and bag properties via user-defined predicates in separation logic", *Sci. Comput. Program.*, vol. 77, no. 9, pp. 1006–1036, 2012. [Online]. Available: https://doi.org/10.1016/j.scico.2010.07.004.

[Con+20] L. Convent, S. Lindley, C. McBride, and C. McLaughlin, "Doo bee doo bee doo", *J. Funct. Program.*, vol. 30, e9, 2020. [Online]. Available: https://doi.org/10.1017/S0956796820000039.

[Coq22] Coq, "The coq proof assistant", https://coq.inria.fr/, 2022.

[Daa17] L. Daan, "Type directed compilation of row-typed algebraic effects", in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, 2017, pp. 486–499.

[Dar+19] B. Dariusz, P. Maciej, P. Piotr, and S. Filip, "Binders by day, labels by night: Effect instances via lexically scoped handlers", *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, pp. 1–29, 2019.

[DM01] A. David and M. D. Möller, "From huppaal to uppaal–a translation from hierarchical timed automata to flat timed automata",, 2001.

[DY96] C. Daws and S. Yovine, "Reducing the number of clock variables of timed automata", in *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96), December 4-6, 1996, Washington, DC, USA*, IEEE Computer Society, 1996, pp. 73–81. [Online]. Available: `https://doi.org/10.1109/REAL.1996.563702`.

[dMB08] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver", in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, C. R. Ramakrishnan and J. Rehof, Eds., ser. Lecture Notes in Computer Science, vol. 4963, Springer, 2008, pp. 337–340. [Online]. Available: `https://doi.org/10.1007/978-3-540-78800-3%5C_24`.

[dVP21] P. E. de Vilhena and F. Pottier, "A separation logic for effect handlers", *Proc. ACM Program. Lang.*, vol. 5, no. POPL, pp. 1–28, 2021. [Online]. Available: `https://doi.org/10.1145/3434314`.

[Don+08] J. S. Dong, P. Hao, S. Qin, J. Sun, and W. Yi, "Timed automata patterns", *IEEE Trans. Software Eng.*, vol. 34, no. 6, pp. 844–859, 2008. [Online]. Available: `https://doi.org/10.1109/TSE.2008.52`.

[Ecm99] E. Ecma, "262: Ecmascript language specification", *ECMA (European Association for Standardizing Information and Communication Systems), pub-ECMA: adr,*, 1999.

[Fac22] Facebook, "Infer static analyzer", `https://fbinfer.com/`, 2022.

[Flo+19] S. P. Florence, S.-H. You, J. A. Tov, and R. B. Findler, "A calculus for esterel: If can, can. if no can, no can", *Proc. ACM Program. Lang.*, vol. 3, no. POPL, 61:1–61:29, 2019. [Online]. Available: https://doi.org/10.1145/3290374.

[FTA02] J. S. Foster, T. Terauchi, and A. Aiken, "Flow-sensitive type qualifiers", in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, 2002, pp. 1–12.

[Git22] Github-effect-handlers, "Recursive cow", https://github.com/effect-handlers/effects-rosetta-stone/tree/master/examples/recursive-cow, 2022.

[Gon88] G. Gonthier, "Sémantiques et modèles d'exécution des langages réactifs synchrones: Application à Esterel", Ph.D. dissertation, Paris 11, 1988.

[GNA14] S. Guha, C. Narayan, and S. Arun-Kumar, "Reducing clocks in timed automata while preserving bisimulation", in *CONCUR 2014 - Concurrency Theory - 25th International Conference, CONCUR 2014, Rome, Italy, September 2-5, 2014. Proceedings*, P. Baldan and D. Gorla, Eds., ser. Lecture Notes in Computer Science, vol. 8704, Springer, 2014, pp. 527–543. [Online]. Available: https://doi.org/10.1007/978-3-662-44584-6%5C_36.

[Hav+97] K. Havelund, A. Skou, K. G. Larsen, and K. Lund, "Formal modeling and analysis of an audio/video protocol: An industrial case study using UPPAAL", in *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97), December 3-5, 1997, San Francisco, CA, USA*, IEEE Computer Society, 1997, pp. 2–13. [Online]. Available: https://doi.org/10.1109/REAL.1997.641264.

[HL18] D. Hillerström and S. Lindley, "Shallow effect handlers", in *Asian Symposium on Programming Languages and Systems*, Springer, 2018, pp. 415–435.

[HLA20] D. Hillerström, S. Lindley, and R. Atkey, "Effect handlers via generalised continuations", *J. Funct. Program.*, vol. 30, e5, 2020. [Online]. Available: https://doi.org/10.1017/S0956796820000040.

[HC14] M. Hofmann and W. Chen, "Abstract interpretation from büchi automata", in *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, T. A. Henzinger and D. Miller, Eds., ACM, 2014, 51:1–51:10. [Online]. Available: `https://doi.org/10.1145/2603088.2603127`.

[Hov12] D. Hovland, "The inclusion problem for regular expressions", *J. Comput. Syst. Sci.*, vol. 78, no. 6, pp. 1795–1813, 2012. [Online]. Available: `https://doi.org/10.1016/j.jcss.2011.12.003`.

[JPO95] L. J. Jagadeesan, C. Puchol, and J. V. Olnhausen, "Safety property verification of ESTEREL programs and applications to telecommunications software", in *Computer Aided Verification, 7th International Conference, Liège, Belgium, July, 3-5, 1995, Proceedings*, P. Wolper, Ed., ser. Lecture Notes in Computer Science, vol. 939, Springer, 1995, pp. 127–140. [Online]. Available: `https://doi.org/10.1007/3-540-60045-0%5C_45`.

[Jou87] P. Jouvelot, "Semantic parallelization: A practical exercise in abstract interpretation", in *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*, ACM Press, 1987, pp. 39–48. [Online]. Available: `https://doi.org/10.1145/41625.41629`.

[JG89] P. Jouvelot and D. K. Gifford, "Reasoning about continuations with control effects", in *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation (PLDI), Portland, Oregon, USA, June 21-23, 1989*, R. L. Wexelblat, Ed., ACM, 1989, pp. 218–226. [Online]. Available: `https://doi.org/10.1145/73141.74837`.

[JG91] P. Jouvelot and D. K. Gifford, "Algebraic reconstruction of types and effects", in *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida, USA,* *January 21-23, 1991*, D. S. Wise, Ed., ACM Press, 1991, pp. 303–310. [Online]. Available: `https://doi.org/10.1145/99583.99623`.

[KLO13] O. Kammar, S. Lindley, and N. Oury, "Handlers in action", *ACM SIGPLAN Notices*, vol. 48, no. 9, pp. 145–158, 2013.

[KT14a] M. Keil and P. Thiemann, "Symbolic solving of extended regular expression inequalities", in *34th International Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2014, December 15-17, 2014, New Delhi, India*, V. Raman and S. P. Suresh, Eds., ser. LIPIcs, vol. 29, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2014, pp. 175–186. [Online]. Available: `https://doi.org/10.4230/LIPIcs.FSTTCS.2014.175`.

[KMP00] Y. Kesten, Z. Manna, and A. Pnueli, "Verification of clocked and hybrid systems", *Acta Informatica*, vol. 36, no. 11, pp. 837–912, 2000. [Online]. Available: `https://doi.org/10.1007/s002360050177`.

[KT14b] E. Koskinen and T. Terauchi, "Local temporal reasoning", in *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, T. A. Henzinger and D. Miller, Eds., ACM, 2014, 59:1–59:10. [Online]. Available: `https://doi.org/10.1145/2603088.2603138`.

[KFP07] H. Kress-Gazit, G. E. Fainekos, and G. J. Pappas, "Where's waldo? sensor-based temporal logic motion planning", in *2007 IEEE International Conference on Robotics and Automation, ICRA 2007, 10-14 April 2007, Roma, Italy*, IEEE, 2007, pp. 3116–3121. [Online]. Available: `https://doi.org/10.1109/ROBOT.2007.363946`.

[Lam05] L. Lamport, "Real-time model checking is really simple", in *Correct Hardware Design and Verification Methods, 13th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2005, Saarbrücken, Germany, October 3-6, 2005, Proceedings*, D. Borrione and W. J. Paul, Eds., ser. Lecture Notes in Computer Science, vol. 3725, Springer, 2005, pp. 162–175. [Online]. Available: https://doi.org/10.1007/11560548%5C_14.

[Lam+02] L. Lamport, J. Matthews, M. R. Tuttle, and Y. Yu, "Specifying and verifying systems with TLA+", in *Proceedings of the 10th ACM SIGOPS European Workshop, Saint-Emilion, France, July 1, 2002*, G. Muller and E. Jul, Eds., ACM, 2002, pp. 45–48. [Online]. Available: https://doi.org/10.1145/1133373.1133382.

[Lan98] P. J. Landin, "A generalization of jumps and labels", *Higher-Order and Symbolic Computation*, vol. 11, no. 2, pp. 125–143, 1998.

[Lar+05] K. G. Larsen, M. Mikucionis, B. Nielsen, and A. Skou, "Testing real-time embedded software using UPPAAL-TRON: an industrial case study", in *EMSOFT 2005, September 18-22, 2005, Jersey City, NJ, USA, 5th ACM International Conference On Embedded Software, Proceedings*, W. H. Wolf, Ed., ACM, 2005, pp. 299–306. [Online]. Available: https://doi.org/10.1145/1086228.1086283.

[LPY97] K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a nutshell", *Int. J. Softw. Tools Technol. Transf.*, vol. 1, no. 1-2, pp. 134–152, 1997. [Online]. Available: https://doi.org/10.1007/s100090050010.

[Lei14a] D. Leijen, "Koka: Programming with row polymorphic effect types", in *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014*, P. B. Levy and N. Krishnaswami, Eds., ser. EPTCS, vol. 153, 2014, pp. 100–126. [Online]. Available: https://doi.org/10.4204/EPTCS.153.8.

[Lei14b] D. Leijen, "Koka: Programming with row polymorphic effect types", *arXiv preprint arXiv:1406.2061*, 2014.

[Lei18] D. Leijen, "Algebraic effect handlers with resources and deep finalization", Technical Report MSR-TR-2018-10. Microsoft Research, Tech. Rep., 2018.

[LS07] M. Leucker and C. Sánchez, "Regular linear temporal logic", in *International colloquium on theoretical aspects of computing*, Springer, 2007, pp. 291–305.

[LC12] S. Lindley and J. Cheney, "Row-based effect types for database integration", in *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, 2012, pp. 91–102.

[LMM16] S. Lindley, C. McBride, and C. McLaughlin, "Do be do be do", *CoRR*, vol. abs/1611.09259, 2016. arXiv: `1611.09259`. [Online]. Available: `http://arxiv.org/abs/1611.09259`.

[LSD11] Y. Liu, J. Sun, and J. S. Dong, "PAT 3: An extensible architecture for building multi-domain model checkers", in *IEEE 22nd International Symposium on Software Reliability Engineering, ISSRE 2011, Hiroshima, Japan, November 29 - December 2, 2011*, T. Dohi and B. Cukic, Eds., IEEE Computer Society, 2011, pp. 190–199. [Online]. Available: `https://doi.org/10.1109/ISSRE.2011.19`.

[Ltd22] P. L. P. Ltd., `https://www.programiz.com/javascript/setTimeout`, 2022.

[LG88] J. M. Lucassen and D. K. Gifford, "Polymorphic effect systems", in *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1988, pp. 47–57.

[LGL19] L. Luthmann, H. Göttmann, and M. Lochau, "Checking timed bisimulation with bounded zone-history graphs - technical report", *CoRR*, vol. abs/1910.08992, 2019. arXiv: `1910.08992`. [Online]. Available: `http://arxiv.org/abs/1910.08992`.

[MLT17] M. Madsen, O. Lhoták, and F. Tip, "A model for reasoning about javascript promises", *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, 86:1–86:24, 2017. [Online]. Available: `https://doi.org/10.1145/3133910`.

[Mai22] Z. Maintainers, `https://zio.dev/`, 2022.

[MMW11] G. Malecha, G. Morrisett, and R. Wisnesky, "Trace-based verification of imperative programs with I/O", *J. Symb. Comput.*, vol. 46, no. 2, pp. 95–118, 2011. [Online]. Available: `https://doi.org/10.1016/j.jsc.2010.08.004`.

[MSS03] K. Marriott, P. J. Stuckey, and M. Sulzmann, "Resource usage verification", in *Asian Symposium on Programming Languages and Systems*, Springer, 2003, pp. 212–229.

[MWP13] M. Muñiz, B. Westphal, and A. Podelski, "Detecting quasi-equal clocks in timed automata", in *Formal Modeling and Analysis of Timed Systems - 11th International Conference, FORMATS 2013, Buenos Aires, Argentina, August 29-31, 2013. Proceedings*, V. A. Braberman and L. Fribourg, Eds., ser. Lecture Notes in Computer Science, vol. 8053, Springer, 2013, pp. 198–212. [Online]. Available: `https://doi.org/10.1007/978-3-642-40229-6%5C_14`.

[Mur+16] A. Murase, T. Terauchi, N. Kobayashi, R. Sato, and H. Unno, "Temporal verification of higher-order functional programs", in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, R. Bodık and R. Majumdar, Eds., ACM, 2016, pp. 57–68. [Online]. Available: `https://doi.org/10.1145/2837614.2837667`.

[NU10] K. Nakata and T. Uustalu, "A hoare logic for the coinductive trace-based big-step semantics of while", in *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, A. D. Gordon, Ed., ser. Lecture Notes in Computer Science, vol. 6012, Springer, 2010, pp. 488–506. [Online]. Available: `https://doi.org/10.1007/978-3-642-11957-6%5C_26`.

[NU15] K. Nakata and T. Uustalu, "A hoare logic for the coinductive trace-based big-step semantics of while", *Log. Methods Comput. Sci.*, vol. 11, no. 1, 2015. [Online]. Available: https://doi.org/10.2168/LMCS-11(1:1)2015.

[Nan+18] Y. Nanjo, H. Unno, E. Koskinen, and T. Terauchi, "A fixpoint logic and dependent effects for temporal property verification", in *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, A. Dawar and E. Grädel, Eds., ACM, 2018, pp. 759–768. [Online]. Available: https://doi.org/10.1145/3209108.3209204.

[Nea+08] I. Neamtiu, M. Hicks, J. S. Foster, and P. Pratikakis, "Contextual effects for version-consistent dynamic software updatingalland safe concurrent programming", in *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, G. C. Necula and P. Wadler, Eds., ACM, 2008, pp. 37–49. [Online]. Available: https://doi.org/10.1145/1328438.1328447.

[Nie+99] F. Nielson, H. R. Nielson, C. Hankin, F. Nielson, H. R. Nielson, and C. Hankin, "Type and effect systems", *Principles of Program Analysis*, pp. 283–363, 1999.

[OHe06] P. W. O'Hearn, "Separation logic and program analysis", in *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings*, K. Yi, Ed., ser. Lecture Notes in Computer Science, vol. 4134, Springer, 2006, p. 181. [Online]. Available: https://doi.org/10.1007/11823230%5C_12.

[OHe20] P. W. O'Hearn, "Incorrectness logic", *Proc. ACM Program. Lang.*, vol. 4, no. POPL, 10:1–10:32, 2020. [Online]. Available: https://doi.org/10.1145/3371078.

[Ölv00] P. C. Ölveczky, "Specification and analysis of real-time and hybrid systems in rewriting logic", Citeseer, 2000.

[ÖM02] P. C. Ölveczky and J. Meseguer, "Specification of real-time and hybrid systems in rewriting logic", *Theor. Comput. Sci.*, vol. 285, no. 2, pp. 359–

405, 2002. [Online]. Available: https://doi.org/10.1016/S0304-3975(01)00363-2.

[ptc19] D. F. patterns and technical concepts, https://docs.microsoft.com/en-us/azure/azure-functions/durable, 2019.

[PRS95] S. Pinchinat, É. Rutten, and R. K. Shyamasundar, "Preemption primitives in reactive languages (A preliminary report)", in *Algorithms, Concurrency and Knowledge: 1995 Asian Computing Science Conference, ACSC '95, Pathumthani, Thailand, December 11-13, 1995, Proceedings*, K. Kanchanasut and J.-J. Lévy, Eds., ser. Lecture Notes in Computer Science, vol. 1023, Springer, 1995, pp. 111–125. [Online]. Available: https://doi.org/10.1007/3-540-60688-2%5C_39.

[Pre13] M. Pretnar, "Inferring algebraic effects", *arXiv preprint arXiv:1312.2334*, 2013.

[Pri10] C. Prisacariu, "Synchronous kleene algebra", *J. Log. Algebraic Methods Program.*, vol. 79, no. 7, pp. 608–635, 2010. [Online]. Available: https://doi.org/10.1016/j.jlap.2010.07.009.

[RHR91] C. Ratel, N. Halbwachs, and P. Raymond, "Programming and verifying critical systems by means of the synchronous data-flow language LUSTRE", in *Proceedings of the conference on Software for citical systems, SIGSOFT 1991, New Orleans, Louisiana, USA*, M. Moriconi, Ed., ACM, 1991, pp. 112–119. [Online]. Available: https://doi.org/10.1145/125083.123062.

[Rei+11] T. Reinbacher, J. Brauer, M. Horauer, A. Steininger, and S. Kowalewski, "Past time LTL runtime verification for microcontroller binary code", in *Formal Methods for Industrial Critical Systems - 16th International Workshop, FMICS 2011, Trento, Italy, August 29-30, 2011. Proceedings*, G. Salaün and B. Schätz, Eds., ser. Lecture Notes in Computer Science, vol. 6959, Springer, 2011, pp. 37–51. [Online]. Available: https://doi.org/10.1007/978-3-642-24431-5%5C_5.

[Rep93] J. H. Reppy, "Concurrent ML: design, application and semantics", in *Functional Programming, Concurrency, Simulation and Automated Reasoning: International Lecture Series 1991-1992, McMaster University, Hamilton, Ontario, Canada*, P. E. Lauer, Ed., ser. Lecture Notes in Computer Science, vol. 693, Springer, 1993, pp. 165–198. [Online]. Available: `https://doi.org/10.1007/3-540-56883-2%5C_10`.

[Siv+21] K. C. Sivaramakrishnan, S. Dolan, L. White, T. Kelly, S. Jaffer, and A. Madhavapeddy, "Retrofitting effect handlers onto ocaml", in *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, S. N. Freund and E. Yahav, Eds., ACM, 2021, pp. 206–221. [Online]. Available: `https://doi.org/10.1145/3453483.3454039`.

[SS04a] C. Skalka and S. Smith, "History effects and verification", in *Asian Symposium on Programming Languages and Systems*, Springer, 2004, pp. 107–128.

[SSV08] C. Skalka, S. Smith, and D. Van Horn, "Types and trace effects of higher order programs", *Journal of Functional Programming*, vol. 18, no. 2, pp. 179–249, 2008.

[SS04b] C. Skalka and S. F. Smith, "History effects and verification", in *Programming Languages and Systems: Second Asian Symposium, APLAS 2004, Taipei, Taiwan, November 4-6, 2004. Proceedings*, W.-N. Chin, Ed., ser. Lecture Notes in Computer Science, vol. 3302, Springer, 2004, pp. 107–128. [Online]. Available: `https://doi.org/10.1007/978-3-540-30477-7%5C_8`.

[Son22a] Y. Song, `https://zenodo.org/record/7071374#.Yx8oU-xBydY`, 2022.

[Son22b] Y. Song, `https://zenodo.org/record/7192718#.Y0emgexBwRR`, 2022.

[SC20] Y. Song and W.-N. Chin, "Automated temporal verification of integrated dependent effects", in *Formal Methods and Software Engineering - 22nd International Conference on Formal Engineering Methods, ICFEM 2020, Singapore, Singapore, March 1-3, 2021, Proceedings*, S.-W. Lin, Z. Hou, and B. P. Mahony, Eds., ser. Lecture Notes in Computer Science, vol. 12531, Springer, 2020, pp. 73–90. [Online]. Available: https://doi.org/10.1007/978-3-030-63406-3%5C_5.

[SC21] Y. Song and W.-N. Chin, "A synchronous effects logic for temporal verification of pure esterel", in *Verification, Model Checking, and Abstract Interpretation - 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17-19, 2021, Proceedings*, F. Henglein, S. Shoham, and Y. Vizel, Eds., ser. Lecture Notes in Computer Science, vol. 12597, Springer, 2021, pp. 417–440. [Online]. Available: https://doi.org/10.1007/978-3-030-67067-2%5C_19.

[SC22] Y. Song and W.-N. Chin, "Automated verification for real-time systems using implicit clocks and an extended antimirov algorithm", in *Companion Proceedings of the 2022 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity, SPLASH Companion 2022, Auckland, New Zealand, December 5-10, 2022*, A. Potanin, Ed., ACM, 2022, pp. 60–62. [Online]. Available: https://doi.org/10.1145/3563768.3563953.

[SC23] Y. Song and W.-N. Chin, "Automated verification for real-time systems", in *Tools and Algorithms for the Construction and Analysis of Systems*, S. Sankaranarayanan and N. Sharygina, Eds., Cham: Springer Nature Switzerland, 2023, pp. 569–587, ISBN: 978-3-031-30823-9.

[SFC22] Y. Song, D. Foo, and W.-N. Chin, "Automated temporal verification for algebraic effects", in *Programming Languages and Systems - 20th Asian Symposium, APLAS 2022, Auckland, New Zealand, December 5, 2022, Proceedings*, I. Sergey, Ed., ser. Lecture Notes in Computer Science, vol. 13658, Springer, 2022, pp. 88–109. [Online]. Available: https://doi.org/10.1007/978-3-031-21037-2%5C_5.

[Sun+09] J. Sun, Y. Liu, J. S. Dong, and J. Pang, "PAT: towards flexible verification under fairness", in *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, A. Bouajjani and O. Maler, Eds., ser. Lecture Notes in Computer Science, vol. 5643, Springer, 2009, pp. 709–714. [Online]. Available: https://doi.org/10.1007/978-3-642-02658-4%5C_59.

[TJ94] J.-P. Talpin and P. Jouvelot, "The type and effect discipline", *Inf. Comput.*, vol. 111, no. 2, pp. 245–296, 1994. [Online]. Available: `https://doi.org/10.1006/inco.1994.1046`.

[Ter03] Terese, "Term rewriting systems", ser. Cambridge tracts in theoretical computer science. Cambridge University Press, 2003, vol. 55, ISBN: 978-0-521-39115-3.

[Tri99] S. Tripakis, "Verifying progress in timed systems", in *Formal Methods for Real-Time and Probabilistic Systems, 5th International AMAST Workshop, ARTS'99, Bamberg, Germany, May 26-28, 1999. Proceedings*, J.-P. Katoen, Ed., ser. Lecture Notes in Computer Science, vol. 1601, Springer, 1999, pp. 299–314. [Online]. Available: `https://doi.org/10.1007/3-540-48778-6%5C_18`.

[UMB13] A. Ulusoy, M. Marrazzo, and C. Belta, "Receding horizon control in dynamic environments from temporal logic specifications", in *Robotics: Science and Systems IX, Technische Universität Berlin, Berlin, Germany, June 24 - June 28, 2013*, P. Newman, D. Fox, and D. Hsu, Eds., 2013. [Online]. Available: `http://www.roboticsproceedings.org/rss09/p13.html`.

[WWH05] F. Wang, R.-S. Wu, and G.-D. Huang, "Verifying timed and linear hybrid rule-systems with RED", in *Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering (SEKE'2005), Taipei, Taiwan, Republic of China, July 14-16, 2005*, W. C. Chu, N. J. Juzgado, and W. E. Wong, Eds., 2005, pp. 448–454.

[Wan+17] X. Wang, J. Sun, T. Wang, and S. Qin, "Language inclusion checking of timed automata with non-zenoness", *IEEE Trans. Software Eng.*, vol. 43, no. 11, pp. 995–1008, 2017. [Online]. Available: `https://doi.org/10.1109/TSE.2017.2653778`.

[Wik22a] Wikipedia, "Cooperative multitasking", `https://en.m.wikipedia.org/wiki/Cooperative_multitasking`, 2022.

[Wik22b] Wikipedia, "Exception wiki", `https://en.wikipedia.org/wiki/Exception_handling`, 2022.

[Wul+06] M. D. Wulf, L. Doyen, T. A. Henzinger, and J.-F. Raskin, "Antichains: A new algorithm for checking universality of finite automata", in *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, T. Ball and R. B. Jones, Eds., ser. Lecture Notes in Computer Science, vol. 4144, Springer, 2006, pp. 17–30. [Online]. Available: `https://doi.org/10.1007/11817963%5C_5`.

[YPD94] W. Yi, P. Pettersson, and M. Daniels, "Automatic verification of real-time communicating systems by constraint-solving", in *Formal Description Techniques VII, Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Techniques, Berne, Switzerland, 1994*, D. Hogrefe and S. Leue, Eds., ser. IFIP Conference Proceedings, vol. 6, Chapman & Hall, 1994, pp. 243–258.

[Yov97] S. Yovine, "KRONOS: A verification tool for real-time systems", *Int. J. Softw. Tools Technol. Transf.*, vol. 1, no. 1-2, pp. 123–133, 1997. [Online]. Available: `https://doi.org/10.1007/s100090050009`.

Appendix A

Appendix for *ASyncEffs*

A.1 Preemption Interleaving Algorithms

The difference between weak and strong preemptions is: in strong preemption, the body does not run when the preemption condition holds. Whereas in weak preemption the body is allowed to run in the current instant even when the preemption condition holds, but are terminated thereafter [Ber93; PRS95]. We present the weak suspend interleaving in Algorithm 3. Furthermore, it is not hard to encode strong abort/suspend from our default weak abort/suspend settings. As shown in Algorithm 3, line 5 and line 13 mark the modifications for encoding strong suspension. Similarly, we could encode strong abort by modifying Algorithm 1 at lines 3 and 9 to $\Delta_1 \leftarrow [(\Phi_{his}, \{\mathbb{S}\}, 0)]$ and $\phi \leftarrow [(\Phi_{his}, \{\mathbb{S}\}, 0)]$ respectively.

Algorithm 3: Weak Suspend Interleaving

Input: $\mathbb{S}, (\Phi, I, k)$

Output: Program States, Δ

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

Lemma 2 (Soundness of Weak Abort Interleaving)**.**
For function $\aleph^{Abort}_{Interleave}$*,* $\forall \mathbb{S}, \Phi, I, k, \Phi_{his}$*,*
if $\Phi{=}\epsilon$*, then* $\Delta = [(\Phi_{his}, I{+}\{\mathbb{S}\}, 0); (\Phi_{his}, I{+}\{\overline{\mathbb{S}}\}, k)]$
else $\Delta = \bigcup_0^{|F|}(\Phi_{his}, f{+}\{\mathbb{S}\}, 0) :: \aleph^{Abort}_{Interleave}(D_f(\Phi), \Phi_{his} \cdot (f{+}\{\overline{\mathbb{S}}\}))$ *where* $F{=}fst(\Phi)$*.*

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

Lemma 3 (Soundness of Weak Suspend Interleaving)**.**
For function $\aleph^{Suspend}_{Interleave}$*,* $\forall \mathbb{S}, \Phi, I, k$*,*
if $\Phi{=}\epsilon$*, then* $\Delta = [(\epsilon, I{+}\{\overline{\mathbb{S}}\}, k); (I{+}\{\mathbb{S}\}, \{\}, k)]$
else $\Delta = \bigcup_0^{|F|}((f{+}\{\overline{\mathbb{S}}\}){\vee}(f{+}\{\mathbb{S}\}){\cdot}\{\})){\cdot}\Delta'$*, where* $F{=}fst(\Phi)$ *and* $\Delta'{=}\aleph^{Suspend}_{Interleave}(D_f(\Phi))$*.*

Proof. By induction on Φ, with Algorithm 3. □

Appendix B

Appendix for *TimEffs*

B.1 Operational Semantics Rules for the Basic Statements

Rules $[v]$, $[assign]$, and $[ev]$ are axioms, which terminate immediately. We use $\mathcal{E}[\alpha]$ to update the environment $\mathcal{E}$ with the assignment α.

$$(\mathcal{E}, v) \xrightarrow{\tau} (\mathcal{E}, v)\ [v] \qquad (\mathcal{E}, \alpha) \xrightarrow{\tau} (\mathcal{E}[\alpha], ())\ [assign]$$

$$(\mathcal{E}, \mathtt{event}[\mathbf{A}(v, \alpha^*)]) \xrightarrow{\mathbf{A}(v)} (\mathcal{E}[\alpha^*], ())\ [ev]$$

In conditionals, if v is True in the environment, the first branch is executed. Otherwise, the other branch is executed. The rule $[call]$ retrieves the function body e of mn from the program, and executes e with instantiated arguments.

$$[cond_1]\ \frac{\mathcal{E}(v) = \mathit{True}}{(\mathcal{E}, \mathit{if}\ v\ e_1\ e_2) \xrightarrow{\tau} (\mathcal{E}, e_1)} \quad [cond_2]\ \frac{\mathcal{E}(v) = \mathit{False}}{(\mathcal{E}, \mathit{if}\ v\ e_1\ e_2) \xrightarrow{\tau} (\mathcal{E}, e_2)} \quad [call]\ \frac{mn x^* \{e\} \in \mathcal{P} \quad (\mathcal{E}, e[v^*/x^*]) \xrightarrow{l} (\mathcal{E}', e')}{(\mathcal{E}, mn(v^*)) \xrightarrow{l} (\mathcal{E}', e')}$$

Rules $[seq_1]$ and $[seq_2]$ state that e_1 takes the control when it still can behave; then the control transfers to e_2 when e_1 terminates. In process $e_1||e_2$, if any of e_1 or e_2 can proceed, they proceed on their own. Rule $[par_3]$ states that if both branches can proceed with the same label, they proceed together.

$$\frac{(\mathcal{E}, e_1) \xrightarrow{l} (\mathcal{E}', e_1')}{(\mathcal{E}, e_1; e_2) \xrightarrow{l} (\mathcal{E}', e_1'; e_2)}[seq_1] \quad \frac{}{(\mathcal{E}, v; e_2) \xrightarrow{\tau} (\mathcal{E}, e_2)}[seq_2] \quad \frac{(\mathcal{E}, e_1) \xrightarrow{l} (\mathcal{E}', e_1')}{(\mathcal{E}, e_1||e_2) \xrightarrow{l} (\mathcal{E}', e_1'||e_2)}[par_1]$$

$$\frac{(\mathcal{E}, e_1) \xrightarrow{l} (\mathcal{E}', v)}{(\mathcal{E}, e_1||e_2) \xrightarrow{l} (\mathcal{E}', e_2)}[par_2] \qquad \frac{(\mathcal{E}, e_1) \xrightarrow{l} (\mathcal{E}, e_1') \qquad (\mathcal{E}, e_2) \xrightarrow{l} (\mathcal{E}, e_2')}{(\mathcal{E}, e_1||e_2) \xrightarrow{l} (\mathcal{E}, e_1'||e_2')}[par_3]$$

B.2 The Complete Forward Rules

Rule [*FV-Value*] obtains the next state by inheriting the current state. Rule [*FV-Event*] concatenates the event to the current state and update the environment for the subsequent statements.

$$\frac{}{\mathcal{E} \vdash \{\pi, \theta\}\ v\ \{\pi, \theta\}}[FV\text{-}Value] \quad \frac{\theta'{=}\theta \cdot \mathbf{A}(v) \quad \mathcal{E}[\alpha^*] \vdash \langle \pi, \theta' \rangle\ e\ \{\Pi, \Theta\}}{\mathcal{E} \vdash \langle \pi, \theta \rangle\ \texttt{event}[\mathbf{A}(v, \alpha^*)]; e\ \{\Pi, \Theta\}}[FV\text{-}Event]$$

Rule [*FV-Call*] first checks whether the instantiated precondition of callee, $\Phi_{pre}[v^*/x^*]$, is satisfied by the current program state. When the check is succeeded, the final states are formed by concatenating the instantiated postcondition to the current states. $\mathcal{P}$ denotes the program being checked.

$$\frac{\begin{array}{c} mn\ x^*\ \{\mathbf{req}\ \Phi_{pre}\ \mathbf{ens}\ \Phi_{post}\}\ \{e\} \in \mathcal{P} \\ \mathcal{E} \vdash \langle \pi, \theta \rangle \sqsubseteq \Phi_{pre}[v^*/x^*] \qquad \Phi_f = \langle \pi, \theta \rangle \cdot \Phi_{post}[v^*/x^*] \end{array}}{\mathcal{E} \vdash \langle \pi, \theta \rangle\ mn(v^*)\ \langle \Phi_f \rangle}[FV\text{-}Call]$$

Rule [*FV-Cond-Local*] computes an over-approximation of the program states, by adding different constraints into different branches. $\pi \wedge v$ enforces v into the pure constraints of every trace in the state, same for $\pi \wedge \neg v$. Rule [*FV-Cond-Global*] is applied when v is a global variable, the constraints are inserted as $\tau(\pi)$ events into the traces, which are determined when other threads are parallel composed.

$$[\mathit{FV\text{-}Cond\text{-}Local}]$$
$$\frac{\mathcal{E} \vdash \{\pi \wedge v, \theta\}\ e_1\ \{\Pi_1, \Theta_1\} \qquad \mathcal{E} \vdash \{\pi \wedge \neg v, \theta\}\ e_2\ \{\Pi_2, \Theta_2\} \qquad (v\ \mathit{is\ local})}{\mathcal{E} \vdash \langle \pi, \epsilon \rangle\ \textbf{if}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2\ \langle \Pi_1, \Theta_1 \rangle \cup \langle \Pi_2, \Theta_2 \rangle}$$
$$[\mathit{FV\text{-}Cond\text{-}Global}]$$
$$\frac{\mathcal{E} \vdash \{\pi, \epsilon\}\ e_1\ \{\Pi_1, \Theta_1\} \qquad \mathcal{E} \vdash \{\pi, \theta\}\ e_2\ \{\Pi_2, \Theta_2\} \qquad (v\ \mathit{is\ global})}{\mathcal{E} \vdash \langle \pi, \theta \rangle\ \textbf{if}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2\ \langle \Pi_1, \theta \cdot \tau(v{=}\mathit{True}) \cdot \Theta_1 \rangle \cup \langle \Pi_2, \theta \cdot \tau(v{=}\mathit{False}) \cdot \Theta_2 \rangle}$$

[*FV-Fun*] initializes the state using the declared precondition, accumulates the effects from the function body, and checks the inclusion between the final state $\langle \Pi, \Theta \rangle$ and the concatenation of the pre- and postcondition[1]. [*FV-Guard*] computes the effects of e and concatenates $(v{=}True)?$ before e's effects.

$$[\mathit{FV\text{-}Fun}] \qquad\qquad [\mathit{FV\text{-}Guard}]$$
$$\frac{\vdash \langle \Phi_{pre} \rangle\ e\ \{\Pi, \Theta\} \qquad \{\Pi, \Theta\} \sqsubseteq \Phi_{pre} \cdot \Phi_{post}}{\mathcal{E} \vdash\ mn\ x^*\ \{\textbf{req}\ \Phi_{pre}\ \textbf{ens}\ \Phi_{post}\}\ \{e\}} \qquad \frac{\mathcal{E} \vdash \{\pi, \epsilon\}\ e\ \{\Pi, \Theta\}}{\mathcal{E} \vdash \{\pi, \theta\}\ [v]e\ \{\Pi, \theta \cdot (v{=}\mathit{True})?\Theta\}}$$

[*FV-Seq*] computes $\{\Pi_1, \Theta_1\}$ from e_1, then further gets $\{\Pi_2, \Theta_2\}$ by continuously computing the behaviors of e_2, to be the final state. [*FV-Par*] computes behaviors for e_1 and e_2 independently, then parallel merges the effects.

$$\frac{\mathcal{E} \vdash \{\pi, \theta\}\ e_1\ \{\Pi_1, \Theta_1\} \qquad \mathcal{E} \vdash \{\Pi_1, \Theta_1\}\ e_2\ \{\Pi_2, \Theta_2\}}{\mathcal{E} \vdash \{\pi, \theta\}\ e_1; e_2\ \{\Pi_2, \Theta_2\}}\ [\mathit{FV\text{-}Seq}]$$
$$\frac{\mathcal{E} \vdash \{\pi, \theta\}\ e_1\ \{\Pi_1, \Theta_1\} \qquad \mathcal{E} \vdash \{\pi, \theta\}\ e_2\ \{\Pi_2, \Theta_2\}}{\mathcal{E} \vdash \{\pi, \theta\}\ e_1 || e_2\ \{\Pi_1 \wedge \Pi_2, \Theta_1 || \Theta_2\}}\ [\mathit{FV\text{-}Par}]$$

B.3 Termination of the TRS

The TRS is terminating.

[1] Note that for succinctness, the user-provided Φ_{post} only denotes the *extension* of the effects from executing the function body.

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 $\langle S, H \rangle$: $\langle S_1, H_1 \rangle \succ \langle S_2, H_2 \rangle$ iff $|H_1| < |H_2| \vee (|H_1| = |H_2| \wedge |S_1| > |S_2|)$.

Here $|X|$ stands for the cardinality of a set X. Let $\Rightarrow$ denote 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 $\{\langle S, H \rangle \mid \langle S_0, H_0 \rangle \Rightarrow^* \langle S, H \rangle\}$, 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 subsection 5.4.1 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.4 Soundness of the TRS

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

Proof. By case analysis for each inclusion checking rules:

1. **Axiom Rules:**

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

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

2. **Disprove Rules:**

$$\frac{\delta_{\pi_1}(\theta_1) \wedge \neg\delta_{\pi_2}(\theta_2)}{\Gamma \vdash \pi_1 \wedge \theta_1 \not\sqsubseteq \pi_2 \wedge \theta_2}\ [DISPROVE] \qquad \frac{\pi_1 \Rightarrow \pi_2 \qquad fst_{\pi_1}(\theta_1) = \{\}}{\Gamma \vdash \pi_1 \wedge \theta_1 \sqsubseteq \pi_2 \wedge \theta_2}\ [PROVE]$$

- It's straightforward to prove soundness of the rule [*DISPROVE*], Given that $\theta 1$ is nullable, while θ_2 is not nullable, thus clearly the antecedent contains more event traces than the consequent. Therefore, these entailments are evidently invalid.

3. **Prove Rules:**

$$\frac{(\pi_1{\wedge}\theta_1 \sqsubseteq \pi_3{\wedge}\theta_3) \in \Gamma \qquad (\pi_3{\wedge}\theta_3 \sqsubseteq \pi_4{\wedge}\theta_4) \in \Gamma \qquad (\pi_4{\wedge}\theta_4 \sqsubseteq \pi_2{\wedge}\theta_2) \in \Gamma}{\Gamma \vdash \pi_1 \wedge \theta_1 \sqsubseteq \pi_2 \wedge \theta_2}\ [\mathit{REOCCUR}]$$

- To prove soundness of the rule [*PROVE*], we consider an arbitrary model, $d, \mathcal{E}, \varphi$ such that: $d, \mathcal{E}, \varphi \models \pi_1 \wedge \theta_1$. Given the side conditions from the promises, we get $d, \mathcal{E}, \varphi \models \pi_2 \wedge \theta_2$. When the *fst* set of θ_1 is empty, θ_1 is possible $\bot$ or ϵ and $\pi_2 \wedge \theta_2$ is nullable. For both cases, the inclusion is valid.

- To prove soundness of the rule [*REOCCUR*], we consider an arbitrary model, $d, \mathcal{E}, \varphi$ such that: $d, \mathcal{E}, \varphi \models \pi_1 \wedge \theta_1$. Given the promises that $\pi_1 \wedge \theta_1 \sqsubseteq \pi_3 \wedge \theta_3$, we get $d, \mathcal{E}, \varphi \models \pi_3 \wedge \theta_3$; Given the premise that there exists a hypothesis $\pi_3 \wedge \theta_3 \sqsubseteq \pi_4 \wedge \theta_4$, we get $d, \mathcal{E}, \varphi \models \pi_4 \wedge \theta_4$; Given the promises that $\pi_4 \wedge \theta_4 \sqsubseteq \pi_2 \wedge \theta_2$,
we get $d, \mathcal{E}, \varphi \models \pi_2 \wedge \theta_2$. Therefore, the inclusion is valid.

4. **Unfolding Rule:**

$$\frac{H{=}\mathit{fst}_{\pi_1}(\theta_1) \qquad \Gamma'{=}\Gamma, (\pi_1{\wedge}\theta_1 \sqsubseteq \pi_2{\wedge}\theta_2) \qquad \forall h{\in}H.\ (\Gamma' \vdash D_h^{\pi_1}(\theta_1) \sqsubseteq D_h^{\pi_2}(\theta_2))}{\Gamma \vdash \pi_1 \wedge \theta_1 \sqsubseteq \pi_2 \wedge \theta_2}\ [\mathit{UNFOLD}]$$

- To prove soundness of [*UNFOLD*], we consider an arbitrary model, $d_1, \mathcal{E}_1, \varphi_1$ and $d_2, \mathcal{E}_2, \varphi_2$ such that: $d_1, \mathcal{E}_1, \varphi_1 \models \pi_1 \wedge \theta_1$ and $d_2, \mathcal{E}_2, \varphi_2 \models \pi_2 \wedge \theta_2$. For an arbitrary event h, let $d_1', \mathcal{E}_1', \varphi_1' \models h^{-1}[\![\pi_1 \wedge \theta_1]\!]$; and $d_2', \mathcal{E}_2', \varphi_2' \models h^{-1}[\![\pi_2 \wedge \theta_2]\!]$.

Case 1), $h \notin F$, $d_1', \varphi_1' \models \bot$, thus automatically $d_1', \varphi_1' \models D_h^{\pi_2}(\theta_2)$;

Case 2), $h \in F$, given that inclusions in the rule's premise is valid,
then $d_1', \mathcal{E}_1', \varphi_1' \models D_h^{\pi_2}(\theta_2)$.

By Definition 20, since for all h, $D_h^{\pi_1}(\theta_1) \sqsubseteq D_h^{\pi_2}(\theta_2)$, the conclusion is valid.

All the inclusion checking rules used in the TRS are sound, therefore the TRS is sound. $\square$

Appendix C

Appendix for *ContEffs*

C.1 The Rest Reasoning Rules for Handlers

Here are the rest rules for "Reasoning in the Handling Program".

$$\frac{\forall(\pi,\theta,v) \in \Phi.\mathbf{l}{=}\mathbf{A}(v) \text{ and } \theta'{=}\theta\cdot\mathbf{l}! \qquad (x{\mapsto}\mathbf{l}?){::}\mathcal{E},\mathcal{H},\mathcal{D} \vdash_h \langle(\pi,\theta',v)\rangle\ e\ \langle\Phi''\rangle}{\mathcal{E},\mathcal{H},\mathcal{D} \vdash_h \langle\Phi\rangle\ \mathit{perform}\ \mathbf{A}(v,\lambda x{\Rightarrow}e)\ \langle\bigvee\Phi''\rangle}\ [\mathit{Handle\text{-}Perform}]$$

$$\frac{(x{\mapsto}v){::}\mathcal{E},\mathcal{H},\mathcal{D} \vdash_h \langle\Phi\rangle\ e\ \langle\Phi\rangle}{\mathcal{E},\mathcal{H},\mathcal{D} \vdash_h \langle\Phi\rangle\ \mathit{let}\ x{=}v\ \mathit{in}\ e\ \langle\Phi\rangle}\ [\mathit{Handle\text{-}Let}]$$

$$\frac{\mathcal{E},\mathcal{H},\mathcal{D} \vdash_h \langle\Phi\wedge(v{=}\mathit{true})\rangle\ e_1\ \langle\Phi_1\rangle \qquad \mathcal{E},\mathcal{H},\mathcal{D} \vdash_h \langle\Phi\wedge(v{=}\mathit{false})\rangle\ e_2\ \langle\Phi_2\rangle}{\mathcal{E},\mathcal{H},\mathcal{D} \vdash_h \langle\Phi\rangle\ \mathit{if}\ v\ \mathit{then}\ e_1\ \mathit{else}\ e_2\ \langle\Phi_1\rangle\cup\langle\Phi_2\rangle}\ [\mathit{Handle\text{-}If\text{-}Else}]$$

$$\frac{\mathcal{E}(v_1){=}\mathbf{l}? \qquad \theta'{=}\mathbf{l}?(v_2)}{\mathcal{E},\mathcal{H},\mathcal{D} \vdash_h \langle\Phi\rangle\ v_1v_2\ \langle\Phi\cdot\theta'\rangle}\ [\mathit{Handle\text{-}App}]$$

$$\frac{\mathcal{E}(v_1){=}\tau\ mn\ (\tau\ v)\ [\mathbf{req}\ \Phi_{pre}\ \mathbf{ens}\ \Phi_{post}]\ \{e\} \qquad \Phi \sqsubseteq \Phi_{pre}[v_2/v]}{\mathcal{E},\mathcal{H},\mathcal{D} \vdash_h \langle\Phi\rangle\ v_1v_2\ \langle\Phi\cdot\Phi_{post}[v_2/v]\rangle}\ [\mathit{Handle\text{-}Call}]$$

$$\frac{\mathcal{E} \vdash \langle(\mathit{True},\epsilon,())\rangle\ e\ \langle\Phi'\rangle \qquad \mathcal{E},h \vdash_{\mathit{fix}} \Phi'\cdot\heartsuit \rightsquigarrow \Phi_{\mathit{fix}}}{\mathcal{E},\mathcal{H},\mathcal{D} \vdash_h \langle\Phi\rangle\ \mathit{match}\ e\ \mathit{with}\ h\ \langle\Phi\cdot\Phi_{\mathit{fix}}\rangle}\ [\mathit{Handle\text{-}Match}]$$

C.1.1 A Demonstration Example

We use Figure C.1 to demonstrate the effects' handling process. Suppose the final effects of `f ()` is `Foo!.Foo?().Goo!.Goo()?`. By applying the rules presented

```
let handling f
/*@ req: _^*  @*/
/*@ ens: Foo.BefF!.Goo.Goo.Done!.AftG!.AftF! @*/
= match f () with (* Foo!.Foo?().Goo!.Goo()? *)
| _ -> perform Done
| effect Foo k -> perform BefF;
                  continue k (fun () -> perform Goo ());
                  perform AftF
| effect Goo k -> continue k (fun ()->()); perform AftG
```

Figure C.1: A contrived example to demonstrate the non-local control flow mechanism and the challenges of reasoning about it.

in the fixpoint computation, we are able to get the final trace `Foo.BefF!.Goo.Goo.Done!.AftG!.AftF!`, given the presented handler.

C.1.2 Soundness of the Reasoning in the Handler

Lemma 4 (Soundness of the reasoning of the handling program)**.**

$$\forall e, \forall \mathcal{E}, \forall \mathcal{H}, \forall \mathcal{D}, \forall \Phi \textit{ if } \mathcal{E}, \mathcal{H}, \mathcal{D} \vdash_h \langle \Phi \rangle\ e\ \langle \Phi' \rangle, \textit{is valid, then: } \forall (\pi, \theta, v) \in \Phi,$$

$$\textit{when } e{=}\textit{let } x{=}\kappa\ v' \textit{ in } e' \textit{ (one shot triggered)},$$

$$\textit{all the } l? \textit{ are substituted by } v' \textit{ in } \mathcal{D}, \textit{and}$$

$$\|\mathcal{E}, \theta, \mathcal{H}\| \vdash_{\textit{fix}} (\pi, \mathcal{D}, v) \rightsquigarrow \Phi' \textit{ is valid},$$

$$\textit{such that } \mathcal{E}, \mathcal{H}, \mathcal{D} \vdash_h \langle \Phi' \rangle\ e\ \langle \Phi'' \rangle \textit{ is valid}; \quad (a)$$

$$\textit{when } e{=}v' \textit{ (zero-shot)}, v \textit{ is updated to } v', \textit{and } \mathcal{D} \textit{ is left unhandled}; \quad (b)$$

$$\textit{otherwise}, \mathcal{E} \vdash \langle \Phi \rangle\ e\ \langle \Phi' \rangle \textit{ is valid}. \quad (c)$$

Proof. By induction on the structure of e.

- Where $e{=}\textit{let } x{=}\kappa\ v' \textit{ in } e'$, by applying the rule [*Handle-Resume*], the conclusion of case (a) is satisfied.

- Where $e{=}v'$, by applying [*Handle-Value*], the conclusion of case (b) is satisfied.

- For the rest cases, handled by the rules [*Handle-Perform*], [*Handle-If-Else*], [*Handle-*

Let], [*Handle-App*], [*Handle-Call*], *and* [*Handle-Match*], which work exactly like rules [*FV-Perform*], [*FV-If-Else*], [*FV-Let*], [*FV-App*], [*FV-Call*], *and* [*FV-Match*] respectively; therefore the conclusion of case (c) is satisfied. □

C.2 Soundness of the Fixpoint Computation

Given an effect Φ, with the environment $\mathcal{E}$ and handler $\mathcal{H}$. Φ_{fix} is the updated version of Φ, where all Φ's placeholders – which can be handled by $\mathcal{H}$ – are handled as $\mathcal{H}$ defines.

$$\begin{array}{ll}
\mathit{Formally}, \forall\mathcal{E}, \forall\mathcal{H}, \forall\Phi, \mathit{if}\ \mathcal{E},\mathcal{H} \vdash_{\mathit{fix}} \Phi \rightsquigarrow \Phi_{\mathit{fix}} \text{ is valid, then:} & \\
\mathit{when}\ \Phi \text{ is a set}, \Phi_{\mathit{fix}}{=}\{\|\mathcal{E},\epsilon,\mathcal{H}\| \vdash_{\mathit{fix}} (\pi,\theta,v) \rightsquigarrow \Phi' \mid (\pi,\theta,v) \in \Phi\}; & (1) \\
\mathit{when}\ \Phi{=}(\pi,\theta,v), \alpha{=}\mathit{fst}(\theta), \theta_{\mathit{his}} \text{ is the handled trace,} & \\
\quad \text{if } \alpha{=}\heartsuit{:}([x{\mapsto}v]){::}\mathcal{E}{\vdash}\langle(\pi,\theta_{\mathit{his}},v)\rangle e_{\mathit{ret}}\langle\Phi'\rangle\ \mathit{is\ valid}, \mathit{given}\ (\mathit{return}\ x{\mapsto}e_{\mathit{ret}}){\in}\mathcal{H}; & (2) \\
\quad \text{if } \alpha{\in}\{ev, \mathbf{l}!, \mathbf{l}?(v')\}\ (\mathbf{l}{\notin}\mathcal{H}):\ \|\mathcal{E},\theta_{\mathit{his}}\cdot\alpha,\mathcal{H}\| \vdash_{\mathit{fix}} (\pi,\mathcal{D}_\alpha(\theta),v) \rightsquigarrow \Phi'\ \mathit{is\ valid}; & (3) \\
\quad \text{if } \alpha{\in}\{\mathbf{l}!\}\ (\mathbf{l}{\in}\mathcal{H}):\ (x{\mapsto}v){::}\mathcal{E},\mathcal{H},\mathcal{D}_\alpha(\theta) \vdash_h \langle(\pi,\theta_{\mathit{his}}\cdot\mathbf{l},v)\rangle\ e\ \langle\Phi'\rangle\ \mathit{is\ valid}, & \\
\hfill \mathit{given}\ (\mathit{effect}\ \mathbf{A}(x,\kappa) \mapsto e){\in}\mathcal{H}. & (4)
\end{array}$$

Proof. By case analysis on the rules for fixpoint computation, we prove the soundness of the presented rules.
- For [*Fix-Disj*], where Φ is a set, by applying the rule itself, the conclusion of case (1) is satisfied.
- For [*Fix-Normal*], where $\alpha{=}\heartsuit$, by applying the rule itself, the conclusion of case (2) is satisfied.
- For [*Fix-Unfold-Skip*], where $\alpha{\in}\{ev, \mathbf{l}!, \mathbf{l}?(v')\}$ $(\mathbf{l}{\notin}\mathcal{H})$, by applying the rule itself, the conclusion of case (3) is satisfied.
- For [*Fix-Unfold-Handle*], where $\alpha{\in}\{\mathbf{l}!\}$ $(\mathbf{l}{\in}\mathcal{H})$, by applying the rule itself and Lemma 4, the conclusion of case (4) is satisfied. □

C.3 Termination Proof of the TRS

Proof. Let Set[$\mathcal{I}$] be a data structure representing the sets of entailments. 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 $\langle S, H\rangle$:

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

where $|X|$ stands for the cardinality of a set X. Let $\Rightarrow$ denote 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.

Rewriting rules in our TRS (given in subsection 6.4.1) transform current pairs $\langle S, H\rangle$ to new pairs $\langle S', H'\rangle$. And each rule either increases $|H|$ (Unfold) or, otherwise, reduces $|S|$ (Axioms, Dis-Nullable, Dis-Infinitable, Reoccur), therefore the system is terminating.

□

C.4 Soundness Proof of the TRS

Proof. For each rewriting rules, if inclusions in their premises are valid, then goal inclusions in their conclusions are valid.

1. **Axiom Rules:**

$$\frac{}{\Omega \vdash (\pi_1, \bot) \sqsubseteq (\pi_2, \theta)}\ [\mathit{Bot\text{-}LHS}] \qquad \frac{\theta \neq \bot}{\Omega \vdash (\pi_1, \theta) \not\sqsubseteq (\pi_2, \bot)}\ [\mathit{Bot\text{-}RHS}]$$

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

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

2. **Disprove Rules:**

$$\frac{\delta(\theta_1) \wedge \neg\delta(\theta_2)}{\Omega \vdash (\pi_1, \theta_1) \not\sqsubseteq (\pi_2, \theta_2)}\ [\textit{Dis-Nullable}] \qquad \frac{\varkappa(\theta_1) \wedge \neg\varkappa(\theta_2)}{\Omega \vdash (\pi_1, \theta_1) \not\sqsubseteq (\pi_2, \theta_2)}\ [\textit{Dis-Infinitable}]$$

- It's straightforward to prove soundness of the rule [*Dis-Nullable*], Given that θ_1 is possibly empty trace, while $\pi_2 \wedge \theta_2$ contains definitely no empty trace, thus clearly the antecedent contains more event traces than the consequent. Therefore, these entailments are evidently invalid.

- It's straightforward to prove soundness of the rule [*Dis-Infinitable*], Given that θ_1 is possibly infinite trace, while $\pi_2 \wedge \theta_2$ contains definitely no infinite trace, thus clearly the antecedent contains more event traces than the consequent. Therefore, these entailments are evidently invalid.

3. **Prove Rules:**

$$\frac{(\pi_1, \theta_1) \sqsubseteq (\pi_3, \theta_3) \in \Omega \quad (\pi_3, \theta_3) \sqsubseteq (\pi_4, \theta_4) \in \Omega \quad (\pi_4, \theta_4) \sqsubseteq (\pi_2, \theta_2) \in \Omega}{\Omega \vdash (\pi_1, \theta_1) \sqsubseteq (\pi_2, \theta_2)}\ [\textit{Reoccur}]$$

- To prove soundness of the rule [*Reoccur*], we consider an arbitrary model, $\mathcal{E}, \varphi$ such that: $\mathcal{E}, \varphi \models \pi_1 \wedge \theta_1$. Given the premise that $\theta_1 \sqsubseteq \pi_3 \wedge \theta_3$, we get $\mathcal{E}, \varphi \models \pi_3 \wedge \theta_3$; Given the premise that there exists a hypothesis $\pi_3 \wedge \theta_3 \sqsubseteq \pi_4 \wedge \theta_4$, we get $\mathcal{E}, \varphi \models \theta_4$; Given the premise that $\pi_4 \wedge \theta_4 \sqsubseteq \pi_2 \wedge \theta_2$, we get $\mathcal{E}, \varphi \models \pi_2 \wedge \theta_2$. Therefore, the entailment is valid.

4. **Unfolding Rule:**

$$\frac{F = fst(\theta_1) \qquad \pi_1 \Rightarrow \pi_2 \qquad \forall \alpha \in F.\ (\theta_1 \sqsubseteq \theta_2) :: \Omega \vdash D_\alpha(\theta_1) \sqsubseteq D_\alpha(\theta_2)}{\Omega \vdash (\pi_1, \theta_1) \sqsubseteq (\pi_2, \theta_2)}\ [\textit{Unfold}]$$

- To prove soundness of the rule [*Unfold*], we consider an arbitrary model, $\mathcal{E}_1, \varphi_1$ and $\mathcal{E}_2, \varphi_2$ such that: $\mathcal{E}_1, \varphi_1 \models \theta_1$ and $\mathcal{E}_2, \varphi_2 \models \pi_2 \wedge \theta_2$. For an arbitrary event α, let $\mathcal{E}_1', {\varphi_1}' \models \alpha^{-1}[\![\theta_1]\!]$; and $\mathcal{E}_2', {\varphi_2}' \models \alpha^{-1}[\![\pi_2 \wedge \theta_2]\!]$.

Case 1), $\alpha \notin F$, $\mathcal{E}_1', {\varphi_1}' \models \bot$, thus automatically $\mathcal{E}_1', {\varphi_1}' \models \pi_2 \wedge D_\alpha(\theta_2)$;

Case 2), $\alpha \in F$, given that inclusions in the rule' premise is valid, then $\mathcal{E}_1', {\varphi_1}' \models \pi_2 \wedge D_\alpha(\theta_2)$.

By Definition 13, since for all α, $\pi_1 \wedge D_\alpha(\theta_1) \sqsubseteq \pi_2 \wedge D_\alpha(\theta_2)$, the conclusion is valid.

All the rewriting rules used in the TRS are sound, therefore the TRS is sound. $\square$

Publications during PhD Study

[SC23] Y. Song and W.-N. Chin, "Automated verification for real-time systems", in *Tools and Algorithms for the Construction and Analysis of Systems*, S. Sankaranarayanan and N. Sharygina, Eds., Cham: Springer Nature Switzerland, 2023, pp. 569–587, ISBN: 978-3-031-30823-9.

[SC22] Y. Song and W.-N. Chin, "Automated verification for real-time systems using implicit clocks and an extended antimirov algorithm", in *Companion Proceedings of the 2022 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity, SPLASH Companion 2022, Auckland, New Zealand, December 5-10, 2022*, A. Potanin, Ed., ACM, 2022, pp. 60–62. [Online]. Available: `https://doi.org/10.1145/3563768.3563953`.

[SFC22] Y. Song, D. Foo, and W.-N. Chin, "Automated temporal verification for algebraic effects", in *Programming Languages and Systems - 20th Asian Symposium, APLAS 2022, Auckland, New Zealand, December 5, 2022, Proceedings*, I. Sergey, Ed., ser. Lecture Notes in Computer Science, vol. 13658, Springer, 2022, pp. 88–109. [Online]. Available: `https://doi.org/10.1007/978-3-031-21037-2%5C_5`.

[SC21] Y. Song and W.-N. Chin, "A synchronous effects logic for temporal verification of pure esterel", in *Verification, Model Checking, and Abstract Interpretation - 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17-19, 2021, Proceedings*, F. Henglein, S. Shoham, and Y. Vizel, Eds., ser. Lecture Notes in Computer Science, vol. 12597, Springer, 2021, pp. 417–440. [Online]. Available: `https://doi.org/10.1007/978-3-030-67067-2%5C_19`.

[SC20] Y. Song and W.-N. Chin, "Automated temporal verification of integrated dependent effects", in *Formal Methods and Software Engineering - 22nd International Conference on Formal Engineering Methods, ICFEM 2020, Singapore, Singapore, March 1-3, 2021, Proceedings*, S.-W. Lin, Z. Hou, and B. P. Mahony, Eds., ser. Lecture Notes in Computer Science, vol. 12531, Springer, 2020, pp. 73–90. [Online]. Available: https://doi.org/10.1007/978-3-030-63406-3%5C_5.