AUTOMATED TEMPORAL VERIFICATION 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
2022

Supervisor:
Wei-Ngan Chin, Associate Professor

Examiners:
Joxan Jaffar, Professor
Jin Song Dong, Professor
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

November 2022
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 incented me to widen my research from various perspectives.

I thank the professors 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

Acknowledgments i  
Summary vii  
List of Figures viii  
List of Tables x  

1 Introduction 1  
  1.1 Verification Framework Overview 3  
  1.2 Thesis Synopsis 4  

2 Literature Review 6  
  2.1 General Perspectives of Temporal Verification 6  
    2.1.1 Model Checking and Effects Systems 6  
    2.1.2 Expressive Effect Logics 8  
    2.1.3 Efficient Algorithms for Language Inclusion Checking 8  
  2.2 Domain Specific Related Works 9  
    2.2.1 Mixed Sync-Async Concurrency Model 9  
    2.2.2 Real-Time Verification 12  
    2.2.3 Algebraic Effects 15  

3 Dependent Effects 17  
  3.1 Introduction 17  
  3.2 Overview 19
3.2.1 DependentEffs ................................. 19
3.2.2 Forward Verification ............................ 20
3.2.3 The TRS ....................................... 21
3.3 Language and Specifications ...................... 23
  3.3.1 The Target Language ......................... 23
  3.3.2 The Specification Language .................... 24
  3.3.3 Semantic Model of DependentEffs .............. 25
3.4 Automated Forward Verification ................... 26
  3.4.1 Forward Rules ................................ 26
3.5 Effects Inclusion Checker ........................ 27
  3.5.1 Effect Disjunction ............................. 28
  3.5.2 Existential Quantifiers ......................... 28
  3.5.3 Normalization ................................ 29
  3.5.4 Substitution .................................. 29
  3.5.5 Case Split .................................... 29
  3.5.6 Unfolding (Induction) ......................... 30
  3.5.7 Disprove (Heuristic Refutation) ............... 31
  3.5.8 Prove ....................................... 31
3.6 Implementation and Evaluation ..................... 32
  3.6.1 Case Studies ................................ 32
  3.6.2 Experimental Results ......................... 34
3.7 Summary ....................................... 36

4 Synchronous Effects .............................. 37
  4.1 Introduction ................................... 37
  4.2 Verification Challenges .......................... 41
    4.2.1 A Sense of Esterel: Synchrony and Preemption 41
    4.2.2 Asynchrony from JavaScript Promises: Async–Await 43
  4.3 Overview ..................................... 44
    4.3.1 Imp\(^{\omega}\) and AsyncEffs .................. 44
4.3.2 Forward Verification ................................................. 46
4.3.3 The TRS .............................................................. 48
4.4 Language and Specifications ......................................... 49
  4.4.1 The Target Language ............................................... 49
  4.4.2 Structural Operational Semantics of the Target Language .. 51
  4.4.3 An Effect Logic for the Temporal Specification ............... 53
  4.4.4 Semantic Model of $ASyncEffs$ .................................. 53
4.5 Automated Forward Verification ...................................... 55
4.6 Temporal Verification via a TRS ...................................... 57
  4.6.1 Auxiliary Functions: Nullable, First and Derivative .......... 58
  4.6.2 Rewriting Rules .................................................... 59
4.7 Implementation and Evaluation ....................................... 60
  4.7.1 Case Studies ....................................................... 62
4.8 Summary ...................................................................... 65

5 Timed Effects .................................................................... 66
  5.1 Introduction .............................................................. 66
  5.2 Overview ................................................................. 69
    5.2.1 $TimEffs.$ .......................................................... 69
    5.2.2 Forward Verification. ............................................. 71
    5.2.3 The TRS. .............................................................. 71
    5.2.4 Verifying the Fischer’s Mutual Exclusion Protocol. ......... 73
  5.3 Language and Specifications .......................................... 75
    5.3.1 The Target Language ............................................. 75
    5.3.2 Operational Semantics of $C^t$ .................................. 76
    5.3.3 The Specification Language ...................................... 77
    5.3.4 Semantic Model of Timed Effects .............................. 78
  5.4 Automated Forward Verification ...................................... 80
    5.4.1 Forward Rules ....................................................... 80
  5.5 Temporal Verification via a TRS ...................................... 82
5.5.1 Rewriting Rules ........................................... 84
5.6 Implementation and Evaluation .......................... 85
5.6.1 Experimental Results for Symbolic Timed Models. ....... 86
5.6.2 Verifying Fischer’s mutual exclusion algorithm. .......... 87
5.6.3 Case Study: Prove it when Reoccur ....................... 88
5.6.4 Discussion ........................................... 88
5.7 Summary ........................................... 89

6 Continuation based Effects ................................. 91
6.1 Introduction ........................................... 91
6.2 Overview ........................................... 95
6.2.1 A Sense of ContEffs in File I/O ......................... 95
6.2.2 Effects Inferences via a Fixpoint Calculation .......... 96
6.2.3 The TRS: to prove effects inclusions .................... 97
6.3 Language and Specifications .............................. 98
6.3.1 The Target Language ................................ 98
6.3.2 The Specification Language .......................... 100
6.3.3 Instrumented Semantics ................................ 102
6.4 Automated Forward Verification .......................... 103
6.4.1 Forward Verification Rules .......................... 103
6.4.2 Fixpoint Computation ................................ 105
6.4.3 Reasoning in the Handling Program ................. 106
6.5 Temporal Verification via a TRS .......................... 108
6.5.1 Rewriting Rules .................................. 109
6.6 Implementation and Evaluation .......................... 111
6.6.1 Case Studies .................................. 112
6.7 Summary ........................................... 113

7 Conclusion and Future Work ............................. 115
7.1 Conclusion ........................................... 115
7.1.1 Useful Links ................................ 116
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 possible insufficiencies: (i) it requires a manual modeling stage and needs to be bounded when encountering non-terminating traces; (ii) to conveniently deploy existing inclusion-checkers for automata, the expressiveness power is limited by the finite-state automata; and (iii) there is always a gap between the verified logic and the actual implementation of the program.

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. Meanwhile, this thesis proposes 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 necessary proofs.
List of Figures

1.1 Verification Framework Overview. ............................... 3

3.1 The forward verification example for the method send. ............ 21
3.2 A Core Imperative Language. .................................. 23
3.3 Syntax of DependentEffs. ...................................... 24
3.4 Semantics of DependentEffs. ................................... 25
3.5 Selected Forward Rules for DependentEffs ......................... 27
3.6 An unknown conditional. ....................................... 33

4.1 A preemptive program written in HipHop.js, for a simple web login, drawn from [BS20]. ............................................. 39
4.2 Using Async-Await in JavaScript. ................................. 43
4.3 Asynchronously reading a file, using async/await in HipHop.js. ...... 45
4.4 A demonstration of the forward verification for the module Read. ... 46
4.5 A demonstration of the forward verification for the module Main. ... 47
4.6 Imp^{a/s} Syntax. ................................................. 50
4.7 Operational semantics of the promise-related and preemptive statements in Imp^{a/s}. .................................................. 51
4.8 Expansion of derived preemptions. .................................. 52
4.9 Syntax of the ASyncEffs. ......................................... 53
4.10 Semantics of the ASyncEffs Logic. ................................ 54
4.11 A Strange Logically Correct Esterel Program. ...................... 64

5.1 Value-dependent specification in TimEffs. .......................... 68
5.2 To make coffee with three portions of sugar within nine time units. ... 70
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3</td>
<td>Forward verification for functions \textit{addOneSugar} and \textit{addNSugar}</td>
<td>72</td>
</tr>
<tr>
<td>5.4</td>
<td>Fischer’s mutually exclusion algorithm</td>
<td>74</td>
</tr>
<tr>
<td>5.5</td>
<td>A core first-order imperative language with timed constructs</td>
<td>75</td>
</tr>
<tr>
<td>5.6</td>
<td>Syntax of \textit{TimEffs}</td>
<td>78</td>
</tr>
<tr>
<td>5.7</td>
<td>Semantics of \textit{TimEffs}</td>
<td>79</td>
</tr>
<tr>
<td>6.1</td>
<td>A loop caused by the effects handler</td>
<td>93</td>
</tr>
<tr>
<td>6.2</td>
<td>A simple file I/O example</td>
<td>95</td>
</tr>
<tr>
<td>6.3</td>
<td>Another Loop caused by the effects handler</td>
<td>96</td>
</tr>
<tr>
<td>6.4</td>
<td>Proving the postcondition of the function \textit{loop} in Figure 6.3</td>
<td>98</td>
</tr>
<tr>
<td>6.5</td>
<td>Syntax of \textit{\lambda h}, a minimal, ML-like language with user-defined effects and handlers</td>
<td>99</td>
</tr>
<tr>
<td>6.6</td>
<td>Evaluation contexts and reduction rules</td>
<td>100</td>
</tr>
<tr>
<td>6.7</td>
<td>Syntax of \textit{ContEffs}</td>
<td>101</td>
</tr>
<tr>
<td>6.8</td>
<td>Semantics of \textit{ContEffs}</td>
<td>102</td>
</tr>
<tr>
<td>6.9</td>
<td>Encoding Exceptions using \textit{ContEffs}</td>
<td>113</td>
</tr>
<tr>
<td>C.1</td>
<td>A contrived example to demonstrate the non-local control flow mechanism and the challenges of reasoning about it.</td>
<td>154</td>
</tr>
</tbody>
</table>
# List of Tables

1.1 Proposals Overview ................................................. 4

3.1 Overview Example for `DependentEffs`. .......................... 19

3.2 One inclusion checking example for `DependentEffs`. ............ 22

3.3 Some Normalization Lemmas for `DependentEffs`. ................ 29

3.4 Examples for converting LTL into `DependentEffs`. .............. 33

3.5 The experiments are based on 16 real world C programs. ........ 35

4.1 Logical correctness examples in Esterel. .......................... 42

4.2 The inclusion proving example for `ASyncEffs`. .................... 49

4.3 Examples for converting LTL formulae into Effects. .............. 55

4.4 Experimental Results. .............................................. 61

4.5 Logically incorrect examples, caught by `ASyncEffs`. ............ 63

4.6 The example for Await. ............................................. 64

5.1 An inclusion proving example. \((I)\) is the right hand side sub-tree of the main rewriting proof tree. ....................... 73

5.2 Examples for converting MTL formulae into `TimEffs` with \(t \in I\) applied. .............................................. 80

5.3 Experimental Results for Manually Constructed Synthetic Examples. .............................................. 86

5.4 Comparison with PAT via verifying Fischer’s mutual exclusion algorithm .............................................. 87

5.5 The reoccurrence proving example in `TimEffs`. \((I)\) is the left hand side sub-tree of the main rewriting proof tree. ................. 88

6.1 Experimental Results for `ContEffs`. ................................. 111

6.2 Examples for converting LTL formulae into `ContEffs`. ............ 112
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 $A$; and the specification gives the set of allowed traces, in an automaton $B$; the implementation meets the specification if every actual trace is allowed, i.e., $L(A) \subseteq L(B)$.

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 possible insufficiencies from such model checking techniques are (i) they require a manual modeling stage and need to be bounded when encountering non-terminating traces; (ii) to conveniently deploy the existing inclusion checker, the expressiveness power is limited by the finite-state automata; and (iii) there is always a gap between the verified logic and the actual implementation from the program.

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 automats (DFA); minimize the DFA to $M_A$ and $M_B$; and
finally check emptiness of $\mathcal{M}_A \cap \neg \mathcal{M}_B$. 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, here proposes a new framework which deploys Hoare-style forward verifiers as the front-ends, and TRSs as the back-ends. Forward verifiers populate the actual temporal behaviors from the source code, based on the formally-defined execution semantics of the target language. TRSs are decision procedures inspired by Antimirov and Mosses’ algorithm [AM95] but solving the language inclusions between more expressive temporal logics.

Antimirov and Mosses’s rewriting is an alternative approach of the automata-based inclusion checking approach. It 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. Termination is guaranteed because the set of derivatives to be considered is finite, and possible cycles are detected using memorization.

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

$$u^{-1}S = \{ w \in \Sigma^* \mid uw \in S \}.$$  

**Theorem 1** (Regular Expressions Inequality (Antimirov)).

For REs $r$ and $s$, $r \preceq s \Leftrightarrow \forall (A \in \Sigma). \ D_A(r) \preceq D_A(s)$.  

Given $\Sigma$ is the whole (finite) set of the alphabet, $D_A(r)$ is the partial derivative of $r$ with respect to the event $A$. Works based on such a TRS [SC20; AM95; AMR09; KT14a; Hov12; Bjø+01; KMP00; ÖM02; Ölv00] show its feasibility and suggest that this method is a better average-case algorithm than those based on the comparison of minimal DFA.

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

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.

A programs with Temporal Specifications

Hoare-style Forward Verifier

Temporal Constraint Proof Obligations

Temporal Specifications

Two Effects

LHS $\sqsubseteq$ RHS

Effects Inclusion Checking via a TRS

Arithmetic Constraint Proof Obligations

SMT

The program is verified? The language inclusion is valid?

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.*)

$^1\sqsubseteq$ captures the inclusion relation between effects, defined base on different effect logics, in Definition 2 (for DependentEffs), Definition 7 (for ASyncEffs), Definition 14 (for TimEffs), and Definition 21 (for ContEffs) respectively.
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 verifies 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:

Table 1.1: Proposals Overview

<table>
<thead>
<tr>
<th>Target Language</th>
<th>Specification Language</th>
<th>Applied Domain</th>
<th>Research Paper</th>
</tr>
</thead>
<tbody>
<tr>
<td>chapter 3</td>
<td>C</td>
<td>DependentEffs</td>
<td>General Effectful Programs</td>
</tr>
<tr>
<td>chapter 4</td>
<td>Imp&lt;sup&gt;a/s&lt;/sup&gt;</td>
<td>ASyncEffs</td>
<td>Reactive Systems</td>
</tr>
<tr>
<td>chapter 5</td>
<td>C&lt;sup&gt;t&lt;/sup&gt;</td>
<td>TimEffs</td>
<td>Time Critical Systems</td>
</tr>
<tr>
<td>chapter 6</td>
<td>λ&lt;sub&gt;h&lt;/sub&gt;</td>
<td>ContEffs</td>
<td>Algebraic Effects and Handlers</td>
</tr>
</tbody>
</table>

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 mixed synchronous and asynchronous execution model by integrating the Synchronous Kleene Algebra (SKA) [Pri10; Bro+15] with a new operator, to provide block waiting (among threads) abstraction into traditional synchronous verification.

3. **TimEffs** (chapter 5) targets real-time systems with mutable variables and timed behavioral patterns, 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 ∗ for finite traces; ω for infinite traces; ∞ for possibly finite and possibly infinite
CHAPTER 1. INTRODUCTION

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 necessary proofs.
Chapter 2

Literature Review

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

This chapter 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). Then, it discusses domain-related contexts for the target languages in chapter 4 chapter 5 and chapter 6 respectively: preliminaries of the mixed synchronous and 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).

2.1 General Perspectives of Temporal Verification

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], are based on some temporal logic specifications, such as LTL and CTL. Such tools extract the logic design from the program using
modeling languages and verify specific assertions to guarantee various 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. Results in [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 behavior [FTA02], and resource usage policies such as file usage protocols and memory management [MSS03]. In [BDF05], a trace effect analysis is used to enforce secure service composition. Stack-based security policies are also amenable to this form of analysis, as shown in [SS04b].

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

Moreover, the proposed effect logics draw similarities to contextual effects [Nea+08], which computes the effects that have already occurred as the prior effects. The effects of the computation are yet to take place 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 the coarsely behaviors either via $\omega$-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 $(\Phi_u, \Phi_v)$, 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 there exist 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
CHAPTER 2. LITERATURE REVIEW

it provides a reasoning logic for regular expression inclusions to avoid any kind of translation aforementioned. Specifically, a TRS takes finite steps to reduce $r \preceq t$ into its normal form $r' \preceq t'$ and the inclusion checking fails whenever $r' \preceq t'$ 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 alphabets and capture value-dependent properties.

In this thesis, we choose to deploy extended TRSs, which composite 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’ 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 effects 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

2.2.1 Mixed Sync-Async Concurrency Model

In chapter 4, we propose ASyncEffs, to reason about a mixed synchronous and 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 mixed synchronous and asynchronous execution models. To the best of the authors’ knowledge, the inference rules in this work formally define the first axiomatic semantics for a core language of HipHop.js, which are established on top of the existing semantics of Esterel and JavaScript’s asynchrony.

For the pure Esterel, the 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]. Similar to [SC21], our inference rules closely follow the work of states-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 asynchronous programs, which can be written in a synchronous style, leading to more scalable code. However, the ECMAScript 6 standard specifies the semantics of promises informally and in operational terms, unsuitable for formal reasoning or program analysis. Prior work [MLT17; Ali+18], in order to understand promise-related bugs, present the $\lambda_p$ 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 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 particu-
larly 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 Syn-
chronous 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 ASyncEffstheory draws similarities to SKA [Pri10], which
is KA extended with a synchrony combinator for actions. Formally, given a KA is
\((A, +, \cdot, *, 0, 1)\), a SKA over a finite set \(A_B\) is \((A, +, \cdot, \times, *, 0, 1, A_B)\), \(A_B \subseteq A\). The
\(\bot\) (false) corresponds to the 0; the \(\epsilon\) (empty trace) corresponds to the 1; the time
instance containing simultaneous signals can be expressed via \(\times\); and the instants
subsumption (Definition 11) 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 4.5). 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 $\epsilon$-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 and customized the TRS for inclusion (instead of equivalence) checking. These decisions led to some opportunities for improvements. That is the worst-case complexity.

Apart from the synchrony combinator, the effect logic also introduced the $\omega$ constructor to explicitly distinguish infinite traces from the coarse-grained repetitive operator Kleene star $\star$. The inclusion of $\omega$ 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) Times 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 of 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 have become increasingly 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. However, possible insufficiencies are: the expressiveness power is limited by the finite-state automata; and all the translation-based approaches share a common problem: the overhead introduced by the complex translation makes it particularly inefficient when *disproving* properties.

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 \textit{now} that is incremented by a \textit{Tick} action. For a continuous-time specification, \textit{Tick} might increment \textit{now} by any real number; for a discrete-time specification, it increments \textit{now} by 1. Timing bounds on actions are specified with one of three kinds of timer variables: a countdown timer is decremented by the \textit{Tick} action, a count-up timer is incremented by \textit{Tick}, and an \textit{expired} timer is left unchanged by \textit{Tick}. A countdown or count-up timer expires when its value reaches some value; an expiration timer expires when its value minus \textit{now} reaches some value. An upper-bound timing constraint on when an action A must occur is expressed by an enabling condition on the \textit{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 \textit{explicit} clocks, [DY96; BC13; MWP13; GNA14] work on dynamically deleting or merging clocks.

Our \textit{TimEffs} inclusion checking process also draws 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 Algebraic Effects

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 \textit{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 discusses the existing logics for reasoning about 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 method of doing this is row-polymorphic effect types, used by languages such as Koka \cite{Lei14, Daa17}, Helium \cite{Bie+19, Dar+19}, Frank \cite{LMM16}, and Links \cite{LC12}. An effect \textit{row} specifies a multi-set of effects a function may perform, and is popular for its simplicity, expressiveness (naturally enabling \textit{effect polymorphism}), and support for inference of principal effects \cite{Lei14}. There are numerous extensions to this model, including \textit{presence types} attached to effect labels, allowing one to express the \textit{absence} of an effect \cite{LC12}, existential and local effects for modularity \cite{Bie+19}, and linearity \cite{Lei18}. Other choices include sets of (instances of) effects \cite{Dar+19}, and structural sub-typing constraints \cite{Pre13}.

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

#### 2.2.3.2 A Separation Logic for Effect Handlers

Recent work \cite{dVP21} proposes a separation logic for effects and handlers, and provides a proof environment in Coq \cite{Coq22} to manually prove properties for algebraic effects. Another work \cite{BP14} presents an effect system for a simplified variant of \textit{Eff} language. \cite{BP14} provides equational reasoning for programs by
CHAPTER 2. LITERATURE REVIEW

providing a fine-grained denotational semantics, which involves mutable states. However, none of the 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

Existing approaches to temporal verification have either sacrificed compositional-
ity 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
effects 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 method 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 \( \omega \)-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 \((\omega)\); (iii) Possibly finite and possibly
infinite traces constructed by Kleene star \((\star)\). For example, it expresses, the effects
CHAPTER 3. DEPENDENT EFFECTS

of method \textit{send}(n) as:

$$\Phi^{\text{send}(n)} \triangleq (n \geq 0 \land \text{Send}^n \cdot \text{Done}) \lor (n < 0 \land \text{Send}^\omega)$$

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

This chapter presents a new solution of extensive temporal verification, which deploys a decision procedure inspired by Antimirov and Mosses’ algorithm but solving the language inclusions between more expressive \textit{DependentEffs}. The main contributions are:

1. **Temporal Effects Specification:** This chapter defines the syntax and semantics of \textit{DependentEffs}, which escapes LTL, \(\mu\)-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 effects 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 Overview

Here is a summary of the techniques, using the example shown in Table 3.1-(a). The \textit{DependentEffs} can be illustrated with \textit{send} and \textit{server}, which simulate a server who continuously sends messages to all its clients.

Table 3.1: Overview Example for \textit{DependentEffs}.

<table>
<thead>
<tr>
<th>(a) Source Code</th>
<th>(b) \textit{DependentEffs} Specifications</th>
</tr>
</thead>
</table>
| \begin{verbatim}  
  void send (int n){
    if (n==0) {
      event["Done"];  
    }else{
      event["Send"];  
      send (n-1);
    }
  }
\end{verbatim} | \begin{verbatim}  
  \Phi_{\text{send}(n)} \triangleq \text{True} \land \text{Ready} \cdot (\_)^* \\
  \Phi_{\text{post}} \triangleq (n \geq 0 \land \text{Send}^n \cdot \text{Done}) \lor (n < 0 \land \text{Send}^\omega) \\
  \Phi_{\text{pre}} \triangleq n \geq 0 \land \epsilon \\
  \Phi_{\text{post}} \triangleq n \geq 0 \land (\text{Ready} \cdot \text{Send}^n \cdot \text{Done})^\omega 
\end{verbatim} |

\begin{verbatim}  
  void server (int n){
    event["Ready"];  
    send(n);  
    server(n); 
  }
\end{verbatim} |

This method \textit{server} takes an integer parameter \(n\), triggers an event \textit{Ready}, then calls the method \textit{send}, making a boolean choice depending on input \(n\): in one case it triggers an event \textit{Done}; otherwise it triggers an event \textit{Send}, then makes a recursive call with parameter \(n-1\). Finally \textit{server} recurs.

3.2.1 \textit{DependentEffs}

The effects specifications for \textit{server} and \textit{send} are given in Table 3.1-(b). It defines Hoare-triple style specifications for each of the programs, which leads to a more compositional verification strategy, where temporal reasoning can be done locally. Method \textit{send}'s precondition, denoted by \(\Phi_{\text{send}(n)}\), requires the event \textit{Ready} to have happened at some point of the effects history; and it guarantees the final effects/postcondition, denoted by \(\Phi_{\text{post}}\).

Method \textit{server}'s precondition, \(\Phi_{\text{server}(n)}\), requires the input value be non-negative.
while the pre-trace is required to be empty ($\epsilon$); its postcondition ensures the final effects $\Phi_{\text{post}}^{\text{server}(n)}$ – an infinite repetition of a trace consisting of an event $\text{Ready}$ followed by $n$ times of $\text{Send}$ followed by $\text{Done}$. Directly from the specifications, it 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 precondition. The examples already show that $\text{DependentEffs}$ provides more detail information than classical LTL or $\mu$-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.

3.2.2 Forward Verification.

As shown in Figure 3.1, it demonstrates the forward verification process of method $\text{send}$. The current effects 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 green. The verifier invokes the TRS to check language inclusions along the way.

The effects state 1) is obtained by initializing $\Phi_C$ from the precondition.

The effects states 2), 4) and 7) are obtained by $[FV-\text{If-Else}]$, which adds the constraints from the conditionals into the current effects state, and unions the effects accumulated from two branches in the end. The effects states 3) and 5) are obtained by $[FV-\text{Event}]$, which simply concatenates the triggered singleton event to the end of the current effects state. The effects state 6) is obtained by $[FV-\text{Call}]$. Before each method call, it checks whether the current state satisfies the precondition of the callee method. The $\text{rev}$ 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 precondition
CHAPTER 3. DEPENDENT EFFECTS

1) void send (int n){ \( \Phi_C = \Phi_{send(n)}^{pre} = True \land Ready \cdot \_^* \) } \([FV-Meth]\)

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

3) event[Done];
\( \{ n=0 \land Ready \cdot \_^* \cdot Done \} \) \([FV-Event]\)

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

5) event[Send];
\( \{ n \neq 0 \land Ready \cdot \_^* \cdot Send \} \) \([FV-Event]\)

6) send(n-1);\}
\( rev(n \neq 0 \land Ready \cdot \_^* \cdot Send) \subseteq rev(\Phi_{pre}^{send(n-1)}) \) \((-check precondition-)
\( \{ n \neq 0 \land Ready \cdot \_^* \cdot Send \cdot \Phi_{post}^{send(n-1)} \} \) \([FV-Call]\)

7) \( \Phi'_C = (n=0 \land Ready \cdot \_^* \cdot Done) \lor (n \neq 0 \land Ready \cdot \_^* \cdot Send \cdot \Phi_{post}^{send(n-1)}) \)

8) \( \Phi'_C \subseteq \Phi_{pre}^{send(n)} \cdot \Phi_{post}^{send(n)} \iff \) \((- check postcondition -)
\( (n=0 \land Done) \lor (n \neq 0 \land Send \cdot \Phi_{post}^{send(n-1)}) \subseteq \Phi_{post}^{send(n)} \)

Figure 3.1: The forward verification example for the method send.

is also new, allowing greater safety to be applied to sequential reactive controlling systems such as web applications, communication protocols and IoT systems.

3.2.3 The TRS

Table 3.2 demonstrates the inclusion checking example on the postcondition of method send. \( I \) : The main rewriting proof tree (coming from the step 8) in Figure 3.1); \( II \) : One sub-tree of the rewriting process.

The TRS is designed to check the inclusion between any two DependentEffs. Here presents the rewriting process on the postcondition checking of the method send. It marks the rules of some essential inference steps in green. Basically, 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
Table 3.2: One inclusion checking example for DependentEffs.

I:

\[(n=0) \land \epsilon \sqsubseteq \epsilon \quad \text{[Frame]} \]
\[(n=0) \land \text{Done} \sqsubseteq \text{Done} \]
\[(n=0) \land \text{Done} \sqsubseteq \Sigma^0 \cdot \text{Done} \]
\[(n=0) \land \text{Done} \sqsubseteq \Phi_{\text{post}}^{\Sigma^0(n)} \]
\[(n=0 \land \text{Done}) \lor (n \neq 0 \land \Sigma \cdot \Phi_{\text{post}}^{\Sigma^0(n-1)}) \sqsubseteq \Phi_{\text{post}}^{\Sigma^0(n)} \]

II:

\[(n=1) \land \epsilon \sqsubseteq \epsilon \quad \text{[Frame]} \]
\[(n=1) \land \text{Done} \sqsubseteq \text{Done} \]
\[(n_2=n_1-1 \land n_2 \geq 0) \land \Sigma^{n_2} \cdot \text{Done} \sqsubseteq \Sigma^{n_2} \cdot \text{Done} \quad \text{[Reoccurs]} \]
\[(n=1) \land \Sigma^{n_1-1} \cdot \text{Done} \sqsubseteq \Sigma^{n_1-1} \cdot \text{Done} \quad \text{[Unfold]} \]
\[(n=1) \land \Sigma^{n_1} \cdot \text{Done} \sqsubseteq \Sigma^{n_1} \cdot \text{Done} \quad \text{[Unfold]} \]
\[(n_1=n-1 \land n_1 \geq 0) \land \Sigma^{n_1} \cdot \text{Done} \sqsubseteq \Sigma^{n_1} \cdot \text{Done} \quad \text{[CaseSplit]} \]
\[(n=1) \land \Sigma^{n_1} \cdot \text{Done} \sqsubseteq \Sigma^{n_1} \cdot \text{Done} \quad \text{[Substitute]} \]
\[n>0 \land \Sigma^{n-1} \cdot \text{Done} \sqsubseteq \Sigma^{n-1} \cdot \text{Done} \quad \text{[Unfold]} \]
\[n>0 \land \Sigma \cdot \Sigma^{n-1} \cdot \text{Done} \sqsubseteq \Sigma \cdot \Sigma^{n} \cdot \text{Done} \]
\[n>0 \land \Sigma \cdot \Sigma^{n-1} \cdot \text{Done} \sqsubseteq \Phi_{\text{post}}^{\Sigma^0(n)} \]

new inclusions.

**Definition 2** (DependentEffs Inclusion). For DependentEffs \( \Phi_1 \) and \( \Phi_2 \),

\[\Phi_1 \preceq \Phi_2 \iff (\forall A \in \Sigma). D_A(\Phi_1) \preceq D_A(\Phi_2).\]

Similarly, we defined the Definition 2 for unfolding the inclusions between DependentEffs. Besides, it uses 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 3.2-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],

\[22\]
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 3.2 to indicate the pairing 
of buds with companions.)

3.3 Language and Specifications

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

3.3.1 The Target Language.

The syntax of the core imperative language, i.e., sequential C-like, is given in 
Figure 3.2 Here regards $k$ and $x$ are meta-variables. $k^\tau$ represents a constant of basic 
type $\tau$. $\text{var}$ represents the countably infinite set of arbitrary distinct identifiers. $a$ 
refers to a singleton event coming from the finite set of events $\Sigma$. It assumes that 
programs used are well-typed conforming to basic types $\tau$ (it takes () as the void 
type). A program $\mathcal{P}$ comprises a list of method declarations $\text{meth}^*$. Here, it uses 
the * superscript to denote a finite list (possibly empty) of items, for example, $x^*$ 
refers to a list of variables, $x_1, \ldots, x_n$.

$$
(\text{Program}) \quad \mathcal{P} ::= \text{meth}^* \\
(\text{Basic Types}) \quad \tau ::= \text{int} | \text{bool} | \text{void} \\
(\text{Method}) \quad \text{meth} ::= \tau \ mn \ (\tau \ x)^* \ \{\text{requires} \ \Phi^{\text{pre}} \ \text{ensures} \ \Phi^{\text{post}}\} \ \{e\} \\
(\text{Expressions}) \quad e ::= () \ | \ k^\tau \ | \ x \ | \ \tau \ x \ | \ e \ | \ mn(x^*) \ | \ x:=e \ | \ e_1; e_2 \\
\quad \quad \quad \quad \quad | \ \text{assert} \ \Phi \ | \ e_1 \ \text{op} \ e_2 \ | \ \text{event}[a] \ | \ \text{if} \ v \ \text{then} \ e_1 \ \text{else} \ e_2 \\
$$

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

Figure 3.2: A Core Imperative Language.

Each method $\text{meth}$ has a name $mn$, an expression-oriented body $e$, also is 
associated with a precondition $\Phi^{\text{pre}}$ and a postcondition $\Phi^{\text{post}}$ (the syntax of effects 
specification $\Phi$ is given in Figure 3.3). The language allows each iterative loop to be
optimal to an equivalent tail-recursive method, where 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 $\tau \ x$; $e$, method calls
$\text{mn}(x^*)$, variable assignments $x:=e$, expression sequences $e_1; e_2$, binary operations
represented by $e_1 \text{op} e_2$, including $+$, $-$, $=$, $<$, etc, event raises expression $\text{event}[a]$, cond-
tional expressions if $v$ then $e_1$ else $e_2$, and the assertion constructor $\text{assert}$, pa-
rametrized with effects $\Phi$.

3.3.2 The Specification Language.

This proposal plants the effects specifications into the Hoare-style verification
system. It uses $\{\text{requires } \Phi_{\text{pre}} \text{ ensures } \Phi_{\text{post}}\}$ to capture the precondition $\Phi_{\text{pre}}$
and the postcondition $\Phi_{\text{post}}$, defined in Figure 3.3.

$$(\text{Effects}) \quad \Phi ::= \pi \land es \mid \Phi_1 \lor \Phi_2 \mid \exists x.\Phi$$

$$(\text{Event Seq.}) \quad es ::= \bot \mid \epsilon \mid _- \mid a \mid es_1 \cdot es_2 \mid es_1 \lor es_2 \mid es_1 \land es_2 \mid \neg es \mid es^t \mid es^* \mid es^\omega$$

$$(\text{Pure}) \quad \pi ::= \text{True} \mid \text{False} \mid A(t_1, t_2) \mid \pi_1 \land \pi_2 \mid \pi_1 \lor \pi_2 \mid \neg \pi \mid \pi_1 \Rightarrow \pi_2 \mid \forall x.\pi \mid \exists x.\pi$$

$$(\text{Terms}) \quad t ::= n \mid x \mid t_1 + t_2 \mid t_1 - t_2$$

$$x ::\in\ \text{var} \quad n ::\in\ \mathbb{Z} \quad (\text{Event}) \quad a ::\in\ \Sigma \quad (\text{Infinity}) \quad \omega \quad (\text{Kleene Star}) \quad *$$

Figure 3.3: Syntax of $\text{DependentEffs}$.

Effects can be a conditioned event sequence $\pi \land es$ or a disjunction of two effects
$\Phi_1 \lor \Phi_2$, or an effect $\Phi$ existentially quantified over a variable $x$. Event sequences
comprise $false$ ($\bot$); an empty trace $\epsilon$; the wild card _ representing any single event;
a single event $a$; sequences concatenation $es_1 \cdot es_2$; disjunction $es_1 \lor es_2$; conjunction
$es_1 \land es_2$; negation $\neg es$; $t$ 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, $es^\omega$. However, for now, the design restricts the nested usage of
operators among $\neg$, $t$, $*$ and $\omega$.  

24
CHAPTER 3. DEPENDENT EFFECTS

It uses \( \pi \) 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 \( =, >, <, \geq \text{ and } \leq \)), 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 \( t_1 - t_2 \).

3.3.3 Semantic Model of DependentEffs.

To define the model, \( \text{var} \) is the set of program variables, \( \text{val} \) is the set of primitive values, \( \text{es} \) is the set of event sequences (or event multi-trees, per se), indicating the sequencing constraints on temporal behavior. Let \( s, \varphi \models \Phi \) denote the model relation.

\[
\begin{align*}
\text{s, } \varphi &\models \Phi_1 \lor \Phi_2 & \text{iff} & \text{s, } \varphi &\models \Phi_1 \text{ or } \text{s, } \varphi &\models \Phi_2 \\
\text{s, } \varphi &\models \exists x. \Phi & \text{iff} & (\exists v \in \text{val}). s[x \rightarrow v], \varphi &\models \Phi \\
\text{s, } \varphi &\models \pi \land \epsilon & \text{iff} & [\pi]_s = \text{True and } \varphi = [] \\
\text{s, } \varphi &\models \pi \land _{\_} & \text{iff} & [\pi]_s = \text{True, } \varphi \in \{ [a] | a \in \Sigma \} \\
\text{s, } \varphi &\models \pi \land a & \text{iff} & [\pi]_s = \text{True and } \varphi = [a] \\
\text{s, } \varphi &\models \pi \land (\text{es}_1 \cdot \text{es}_2) & \text{iff} & \text{there exist } \varphi_1, \varphi_2 \text{ and } \varphi_1 ++ \varphi_2 = \varphi \\
& & & \text{and } s, \varphi_1 \models \pi \land \text{es}_1 \text{ and } s, \varphi_2 \models \pi \land \text{es}_2 \\
\text{s, } \varphi &\models \pi \land (\text{es}_1 \lor \text{es}_2) & \text{iff} & s, \varphi \models \pi \land \text{es}_1 \text{ or } s, \varphi \models \pi \land \text{es}_2 \\
\text{s, } \varphi &\models \pi \land (\text{es}_1 \land \text{es}_2) & \text{iff} & s, \varphi \models \pi \land \text{es}_1 \text{ and } s, \varphi \models \pi \land \text{es}_2 \\
\text{s, } \varphi &\models \pi \land \neg \text{es} & \text{iff} & s, \varphi \not= \pi \land \text{es} \\
\text{s, } \varphi &\models \pi \land \text{es}^t & \text{iff} & [\pi \land t=0]_s = \text{True}, \ s, \varphi \models \pi \land \epsilon \text{ or } [\pi \land t>0]_s = \text{True, there exist } \varphi_1, \varphi_2 \\
& & & \text{and } \varphi_1 ++ \varphi_2 = \varphi \text{ and } s, \varphi_1 \models \pi \land \text{es} \text{ and } s, \varphi_2 \models \pi \land \text{es}^{t-1} \\
\text{s, } \varphi &\models \pi \land \text{es}^x & \text{iff} & s, \varphi \models \exists x. (\pi \land \text{es}^x) \text{ or } s, \varphi \models \pi \land \text{es}^\omega \\
\text{s, } \varphi &\models \pi \land \text{es}^\omega & \text{iff} & \text{there exist } \varphi_1, \varphi_2 \text{ and } \varphi_1 ++ \varphi_2 = \varphi \\
& & & \text{and } s, \varphi_1 \models \pi \land \text{es} \text{ and } s, \varphi_2 \models \pi \land \text{es}^\omega \\
\text{s, } \varphi &\models \text{false} & \text{iff} & [\pi]_s = \text{False or } \varphi = \bot
\end{align*}
\]

Figure 3.4: Semantics of DependentEffs.

...ion, i.e., the stack \( s \) and linear temporal events \( \varphi \) satisfy the temporal effects \( \Phi \),
with \( s, \varphi \) from the following concrete domains: \( s \triangleq \text{var} \rightarrow \text{val} \) and \( \varphi \triangleq \text{es} \).

Figure 3.4 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 \( a \).

### 3.4 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 method to be false. The body of any such method can always be successfully verified. This relaxation does not affect the soundness of the verification system.

#### 3.4.1 Forward Rules

This section presents some of the forward verification rules in Figure 3.5, 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 method in \( \mathcal{P} \), one can apply modular verification to a method’s body using Hoare-style triples \( \vdash \{ \Phi_C \} \ e \ \{ \Phi'_C \} \), where \( \Phi_C \) is the current effects and \( \Phi'_C \) is the resulting effects by executing \( e \). In [FV-If-Else], \((v \land \Phi_C)\) enforces \( v \) into the pure constraints of every traces in \( \Phi_C \), same for \((\neg v \land \Phi_C)\). The rule FV-Call checks whether the instantiated precondition of callee, \([y/x] \Phi_{pre}\), is satisfied by the tail \(^1\) of current effects state, in which it uses an auxiliary function \( \text{rev} \) to reverse the event sequences of effects. Then it obtains the next effects state by concatenating the instantiated postcondition, \([y/x] \Phi_{post}\), to the current effects state. (cf. step 6 in Figure 3.1) In FV-Meth, it initializes the current effects state using \( \epsilon \), accumulate the effects from the method body, to obtain \( \Phi_C \), and check inclusion between \( \Phi_C \)

\(^1\)It checks the inclusion between the reversed current effects and precondition effects, meaning that, before calling a method, its required effects has just happened.
CHAPTER 3. DEPENDENT EFFECTS

\[
\Phi'_{C} = \Phi_{C} \cdot a \quad \vdash \{ \Phi_{C} \} \text{ event}[a] \{ \Phi_{C}' \} \quad \text{[FV-Event]}
\]

\[
\vdash \{ \Phi_{C} \} \cdot \{ \Phi_{C}' \} \quad \vdash \{ \Phi_{C} \} \cdot \{ \Phi_{C}' \} \cdot \{ \Phi_{C}'' \} \quad \text{[FV-Seq]}
\]

\[
\vdash \{ v \land \Phi_{C} \} \cdot e_{1} \quad \{ \Phi_{C}' \} \quad \vdash \{ \neg v \land \Phi_{C} \} \cdot e_{2} \quad \{ \Phi_{C}' \} \quad \text{[FV-If-Else]}
\]

\[
\vdash \{ \Phi_{C} \} \cdot e \quad \{ \Phi_{C}' \} \quad \text{[FV-Local]}
\]

\[
\vdash \{ \Phi_{C} \} \cdot \text{assert } \Phi \quad \{ \Phi_{C} \} \quad \text{[FV-Assert]}
\]

\[
\tau \ mn \ (\tau x)^{*} \ \{ \text{requires } \Phi_{pre} \ \text{ensures } \Phi_{post} \} \{ e \} \in \mathcal{P}
\]

\[
\vdash \text{rev}(\Phi_{C}) \subseteq \text{rev}([y*/x*]\Phi_{pre}) \rightsquigarrow \gamma_{R} \quad \Phi'_{C} = \Phi_{C} \cdot [y*/x*]\Phi_{post} \quad \text{[FV-Call]}
\]

\[
\vdash \{ e \} \cdot \{ \Phi_{C} \} \quad \vdash \{ \Phi_{C} \} \subseteq \{ \Phi_{post} \} \quad \{ e \} \quad \text{[FV-Meth]}
\]

Figure 3.5: Selected Forward Rules for DependentEffs

and the declared specifications \( \Phi_{post} \).^{2}

3.5 Effects Inclusion Checker

The effects inclusion checking (an extension of the TRS proposed from [AM95]) will be triggered i) right before a method 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 3.4, the forward verification generates effects inclusions of the form: \( \Gamma \vdash \Phi_{1} \sqsubseteq_{v} \Phi_{2} \rightsquigarrow \gamma_{R} \), a shorthand for: \( \Gamma \vdash \Phi \cdot \Phi_{1} \sqsubseteq \exists V. (\Phi \cdot \Phi_{2}) \rightsquigarrow \gamma_{R} \).

To prove such effects inclusions is to check whether all the possible event traces in the antecedent \( \Phi_{1} \) are legitimately allowed in the possible event traces from the

---

2\( \Phi_{post} \) only needs to capture the effects from the current method body, excluding the history effects specified in \( \Phi_{pre} \).
consequent $\Phi_2$, and (in case there are) to compute a residual effects $\gamma_R$ (also known as "frame" in the frame inference), which represents what was not consumed from the antecedent after matching up with the effects from the consequent. $\Gamma$ is the proof context, i.e. a set of effects inclusions, $\Phi$ 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 $\Gamma$, $\Phi$ and $V$ are derived during the inclusion proof. The inclusion checking procedure is initially invoked with $\Gamma = \emptyset$, $\Phi = \text{True} \land \epsilon$ and $V = \emptyset$. Here briefly discusses the key steps and related inference 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.

3.5.1 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.*)

\[
\frac{\Gamma \vdash \Phi_1 \sqsubseteq \Phi \sim \gamma_R^1 \quad \Gamma \vdash \Phi_2 \sqsubseteq \Phi \sim \gamma_R^2}{\Gamma \vdash \Phi_1 \lor \Phi_2 \sqsubseteq \Phi \sim (\gamma_R^1 \lor \gamma_R^2)} \quad \text{[LHS-OR]}
\]

3.5.2 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*)

\[
\frac{\Gamma \vdash [u/x]\Phi_1 \sqsubseteq^V \Phi_2 \sim \gamma_R}{\Gamma \vdash \exists x. \Phi_1 \sqsubseteq^V \Phi_2 \sim \gamma_R} \quad \text{[LHS-EX]} \quad \frac{\Gamma \vdash \Phi_1 \sqsubseteq^V \langle u \rangle (\Phi_2) \sim \gamma_R}{\Gamma \vdash \Phi_1 \sqsubseteq^V (\exists x. \Phi_2) \sim \gamma_R} \quad \text{[RHS-EX]}
\]
3.5.3 Normalization.

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

Table 3.3: Some Normalization Lemmas for DependentEffs.

<table>
<thead>
<tr>
<th>Lemma</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( es \lor es \rightarrow es )</td>
<td>( e^\omega \rightarrow \epsilon ) ( (es_1 \cdot es_2) \cdot es_3 \rightarrow es_1 \cdot (es_2 \cdot es_3) )</td>
</tr>
<tr>
<td>( \bot \lor es \rightarrow es )</td>
<td>( es \land es \rightarrow es ) ( (es_1 \lor es_2) \cdot es_3 \rightarrow es_1 \cdot es_3 \lor es_2 \cdot es_3 )</td>
</tr>
<tr>
<td>( es \lor \bot \rightarrow es )</td>
<td>( es \land \bot \rightarrow \bot ) ( es_1 \cdot (es_2 \lor es_3) \rightarrow es_1 \cdot es_2 \lor es_1 \cdot es_3 )</td>
</tr>
<tr>
<td>( \epsilon \cdot es \rightarrow es )</td>
<td>( \bot^\omega \rightarrow \bot ) ( es^\omega \cdot es_1 \rightarrow es^\omega )</td>
</tr>
<tr>
<td>( es \cdot \epsilon \rightarrow es )</td>
<td>( \epsilon^t \rightarrow \epsilon ) ( False \land es \rightarrow False \land \bot )</td>
</tr>
<tr>
<td>( \bot \cdot es \rightarrow \bot )</td>
<td>( t=0 \land es^t \rightarrow \epsilon ) ( es \land \epsilon \rightarrow \bot ) ( (\delta_\pi(es)=false) )</td>
</tr>
<tr>
<td>( es \cdot \bot \rightarrow \bot )</td>
<td>( \bot^t \rightarrow \bot ) ( es \land \epsilon \rightarrow \epsilon ) ( (\delta_\pi(es)=true) )</td>
</tr>
</tbody>
</table>

It is assumed that the effects formulae are tailored accordingly using the lemmas in Table 3.3, which are extended from the normalization rules suggested by Antimirov and Mosses, being able to further normalize DependentEffs.

3.5.4 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 \( u=t_1 \oplus t_2 \land u \geq 0 \), where \( \oplus \in \{+, -, \} \). (cf. Table 3.2-II)

\[
\pi'=(u=t_1 \oplus t_2 \land u \geq 0) \quad \Gamma \vdash (\pi_1 \land \pi') \land es_1^u \cdot es \subseteq (\pi_2 \land \pi') \land es_2 \rightsquigarrow \gamma_R \quad [LHS-SUB]
\]

\[
\pi'=(u=t_1 \oplus t_2 \land u \geq 0) \quad \Gamma \vdash (\pi_1 \land \pi') \land es_1 \subseteq (\pi_2 \land \pi') \land es_2^u \cdot es \rightsquigarrow \gamma_R \quad [RHS-SUB]
\]

3.5.5 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 3.2-II)

\[ \text{[LHS-CaseSplit]} \]
\[ \Gamma \vdash ((\pi_1 \land t=0) \land es) \lor ((\pi_1 \land t > 0) \land es_1 \cdot es_1^{t-1} \cdot es) \sqsubseteq \pi_2 \land es_2 \rightsquigarrow \gamma_R \]
\[ \Gamma \vdash \pi_1 \land (es_1 \cdot es) \sqsubseteq \pi_2 \land es_2 \rightsquigarrow \gamma_R \]

\[ \text{[RHS-CaseSplit]} \]
\[ \Gamma \vdash \pi_1 \land es_1 \sqsubseteq ((\pi_2 \land t=0) \land es) \lor ((\pi_2 \land t > 0) \land es_2 \cdot es_2^{t-1} \cdot es) \rightsquigarrow \gamma_R \]
\[ \Gamma \vdash \pi_1 \land es_1 \sqsubseteq \pi_2 \land (es_2 \cdot es) \rightsquigarrow \gamma_R \]

3.5.6 Unfolding (Induction).

Here comes the key inductive step of unfolding the inclusion. Firstly, it makes use of the \(\text{fst}\) 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 \(\Gamma'\) by adding the current inclusion, as an inductive hypothesis, into the current proof context \(\Gamma\). 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.

\[ F = \text{fst}_{\pi_1}(es_1) \quad \Gamma' = \Gamma, \ (\pi_1 \land es_1 \sqsubseteq \pi_2 \land es_2) \]
\[ \forall a \in F. \ (\Gamma' \vdash D^a_{\pi_1}(es_1) \sqsubseteq D^a_{\pi_2}(es_2)) \]
\[ \Gamma \vdash \pi_1 \land es_1 \sqsubseteq \pi_2 \land es_2 \quad \text{[Unfold]} \]

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 \(\pi \land es\) contains the empty trace; the First function \(\text{fst}_\pi(es)\) computes a set of possible initial events of \(\pi \land es\); and the Derivative function \(D^a_\pi(es)\) computes a next-state effects after eliminating one event \(a\) from the current effects \(\pi \land es\).

\(^3\)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.

30
Definition 3. Nullable Given any event sequence $es$ under condition $\pi$, here defines $\delta_\pi(es)$ to be:

$$
\delta_\pi(es) : \text{bool} = \begin{cases} 
  \text{true} & \text{if } \epsilon \in [\pi \land es_1]_\varphi \\
  \text{false} & \text{if } \epsilon \notin [\pi \land es_1]_\varphi
\end{cases}, \text{ where } \delta_\pi(es') = SMT(\pi \land (t=0))^4
$$

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

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

Definition 5. Derivative The derivative $D^\pi_a(es)$ of an event sequence $es$ with respect to an event $a$ and the condition $\pi$ computes the effects for the left quotient $a^{-1}[\pi \land es]$, where it defines $D^\pi_a(es') = D^\pi_{a \land t>0}(es) \cdot es'^1$.

$$
D^\pi_a(es_1 \cdot es_2) = 
\begin{cases} 
  D^\pi_a(es_1) \cdot es_2 \lor D^\pi_a(es_2) & \text{if } \delta_\pi(es_1) = \text{true} \\
  D^\pi_a(es_1) \cdot es_2 & \text{if } \delta_\pi(es_1) = \text{false}
\end{cases}
$$

3.5.7 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.

$$
\frac{\delta_\pi_1(es_1) \land \neg \delta_\pi_1 \land \delta_\pi_2(es_2)}{\Gamma \vdash \pi_1 \land es_1 \not\subseteq \pi_2 \land es_2} \quad [\text{Disprove}]
$$

3.5.8 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 $\gamma^R$;  

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

$^5$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.
CHAPTER 3. DEPENDENT EFFECTS

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

\[
\Gamma \vdash \pi_1 \Rightarrow \pi_2 \quad [\text{Prove}] \\
\Gamma \vdash \pi_1 \land es_1 \subseteq \pi_2 \land es_2 \\
\exists (\pi_1' \land es_1' \subseteq \pi_2' \land es_2') \in \Gamma \\
\Gamma \vdash \pi_1' \Rightarrow \pi_2' \Rightarrow \pi_2 \\
es_1 \subseteq es_1' \\
es_2' \subseteq es_2 \\
\Gamma \vdash \pi_1 \land es_1 \subseteq \pi_2 \land es_2 \\
\left[ \text{Frame} \right]
\]

\[
\pi_1 \Rightarrow \pi_2 \quad [\text{Frame}] \\
\pi_1 \Rightarrow \pi_2 \quad [\text{Frame}] \\
\gamma_R = \pi_1 \land es_1 \\
\Gamma \vdash (\pi_1 \land es_1 \subseteq \pi_2 \land e) \rightsquigarrow \gamma_R \\
\left[ \text{Reoccur} \right]
\]

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 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.

3.6.1.1 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 composites past-time LTL along the way. (A, B are events, \( n \geq 0, m \geq 0 \) are the default constraints.)
CHAPTER 3. DEPENDENT EFFECTS

Table 3.4: Examples for converting LTL into DependentEffs.

<table>
<thead>
<tr>
<th>□A ≡ A*</th>
<th>⋆A ≡ _n \cdot A</th>
<th>A U B ≡ A^n \cdot B</th>
<th>□\diamond A ≡ _n \cdot A \cdot (_m \cdot A)^*</th>
</tr>
</thead>
<tbody>
<tr>
<td>_A ≡ _A \cdot A</td>
<td>A \rightarrow B ≡ A \lor _n \cdot B</td>
<td>\diamond A ≡ _n \cdot A^*</td>
<td>\diamond A \lor \diamond B ≡ _n \cdot A \lor _m \cdot B</td>
</tr>
</tbody>
</table>

3.6.1.2 Encoding \( \mu \)-calculus.

\( \mu \)-calculus provides a single, elegant, uniform logical framework of great raw expressive power by using a least fixpoint (\( \mu \)) and a greatest fixpoint (\( \nu \)). More specifically, it can express properties such as \( \nu Z.P \land \forall X.Z \), 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 (\( \mu \)), referring to finite traces, and the constructor \( \omega \) corresponds to the greatest fixpoint (\( \nu \)), referring to infinite traces. For example, one can write \( (\_ \cdot A)^\omega \), meaning that the event \( A \) recurs at every even position in an infinite trace.

3.6.1.3 Kleene Star.

By using \( \ast \), 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 \( Send^* \cdot Done \), meaning that the repetition of event \( Send \) can be both finite and infinite, which is more concise than the prior work, also beyond \( \mu \)-calculus.

```c
1 void send (int n){
2     if (...){ event[Done];}
3     else{ event[Send];
4         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 3.1) to
\( \Phi_{\text{server}}(n) \triangleq True \land \epsilon \), and opt for either of the following two postcondition:
\( \Phi_{\text{server}}(n) \triangleq n \geq 0 \land (\text{Ready} \cdot \text{Send}^n \cdot \text{Done})^\omega \), or \( \Phi_{\text{post1}}(n) \triangleq n > 0 \land (\text{Ready} \cdot \text{Send}^n \cdot \text{Done})^\omega \lor n < 0 \land \text{Ready} \cdot \text{Send}^\omega \), with the latter being more complex but more precise.

3.6.1.4 Beyond Regular, Context-Free and Context-Sensitive

The paradigmatic non-regular linear language: \( n > 0 \land a^n \cdot b^n \), can be naturally expressed by the depended effects. Besides, the effects can also express grammars such as \( n > 0 \land a^n \cdot b^n \cdot c^n \), or \( n > 0 \land m > 0 \land a^n \cdot b^m \cdot c^n \), 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 \( n > 0 \land m > 0 \land (\text{Red}^n \cdot \text{Yellow}^m \cdot \text{Green}^n)^\omega \), 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.

The comparison is between the backend TRS and the well-established model checker PAT [Sun+09], which 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
CHAPTER 3. DEPENDENT EFFECTS

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

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

temporal properties (#Prop.), and the proving/disproving times (in milliseconds) using PAT and the TRS respectively. The table compares the TRS with PAT, and the total proving/disproving time has been reduced by 31.7%. For that, here summarizes the underlying reasons 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 effects 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 effects logic, which primarily benefits modern sequential controlling systems ranging over a variety of application domains.
Chapter 4

Synchronous Effects

To make reactive programming more concise and expressive, it is promising to combine two approaches to concurrency that integrates *perfect synchrony and preemption* 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 a core language Imp<sup>a/s</sup>, generalizing the mixed sync-async abstraction. Secondly, we propose ASyncEffs, a new effects logic, that extends Synchronous Kleene Algebra with a *waiting* operator. Thirdly, we establish an effect system for Imp<sup>a/s</sup> 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 mixed sync-async 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. In this paper, we mainly tackle the challenges related to synchronous preemptions and asynchronous promises.

1) The power of preemption appears in Figure 4.1 (here connected.value indicates the latest value carried by the signal connected). The module Main makes use of three submodules: Identity reads the GUI and enables the login button, Authenticate calls the authorization service, and Session starts an active session. Statement fork/par (lines 2, 3) runs branches in parallel. The expression login.now
module Main (in login, inout connected, out connState) {
    fork { Identity(...); } // enables the login button
    par { every(login) {
        Authenticate(...); // preempts the previous Session
        if(connected.value) { Session(...); }
        else { emit connState("err"); }
    }}
}

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

is true when the login button is pressed in the instant. When Authenticate is started, the previously active Session will be automatically killed/preempted. When Authenticate terminates, the value of connected is tested (line 5). If true, Session starts to run a new session until the next login button is pressed.

When Session terminates, the every statement simply waits for a new login. At last, the output signal connState carries a connection status message. Although simple, this example show 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 also show that our approach supports a comprehensive foundation for verifying preemptions with different keywords, including strong or weak, immediate or delayed.

2) With asynchronous promises, we can perform long-lasting tasks without blocking the main thread. The keywords async and await allow sequential-style code to succinctly capture concurrent executions with precise dependency via asynchronous signals. 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, which are caused by insufficient static checking. They propose to mitigate the anti-patterns by constructing a promise dependency graph, and enforce a dependency checking upon the graph.
In this paper we focus on the anti-pattern related to the usage of \textit{async/await}, where \textit{unreachable promises} may have registered reactions that will never be executed. Different from prior works, we show that our purely algebraic approach detects this anti-pattern without any constructions of graphs.

To achieve a modular verification - where modules can be replaced by their already verified properties - for mixed sync-async 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 $A\text{SyncEffs}$, which enriches the Synchronous Kleene Algebra (SKA) [Pri10; Bro+15] with a new operator, to provide the \textit{waiting} abstraction into classic linear temporal verification. Our main contributions are:

1. **Language Abstraction:** we formally define the syntax and structural operational semantics for a core language $\text{Imp}^{a/s}$. It generalizes the mixed sync-async execution model by providing the infrastructure for preemptions and promises.

2. **Specification Logic:** we propose $A\text{SyncEffs}$, by defining its syntax and semantics. We show that in our proposal, $A\text{SyncEffs}$’s expressiveness power subsumes both classic linear temporal logic (LTL) and the \textit{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 $A\text{SyncEffs}$. 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 HipHop.js, 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: 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 4.4.1 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. For example, consider the programs below:

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
**CHAPTER 4. SYNCHRONOUS EFFECTS**

Table 4.1: Logical correctness examples in Esterel.

<table>
<thead>
<tr>
<th>(a) No valid assignments</th>
<th>(b) Two possible assignments</th>
<th>(c) One assignment under the precondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal S1 {</td>
<td>Signal S1 {</td>
<td>Signal S1 {</td>
</tr>
<tr>
<td>1. present S1</td>
<td>2. present S1</td>
<td>3. present S1</td>
</tr>
<tr>
<td>2. then nothing</td>
<td>3. then emit S1</td>
<td>4. when emit S1</td>
</tr>
<tr>
<td>4. else emit S1</td>
<td>5. else nothing</td>
<td>5. else nothing</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(a) No valid assignments (b) Two possible assignments (c) One assignment under the precondition (Logically incorrect). (Logically incorrect). (Logically correct).

S1 (*nothing* leaves S1 with *absent*). If S1 were absent, the program would choose the second branch and emit the signal. Both executions lead to a contradiction. Therefore there are no valid assignments of signals in this program. This program is 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 $\bot$ (*false*) for Table 4.1-(a), and concludes $\{S1\} \lor \{S1\}$ 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. In this paper, 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.

```javascript
const fs = require('fs').promises;

async function read (filePath) {
    const task = fs.readFile(filePath);
    ...// operations do not rely on the result of loading file
    const data = await task; // block waiting
    ...// logging or data processing of the Json file
}
```

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. In this paper we focus 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 Overview

We now highlight our main methodologies, using the example shown in Figure 4.3. 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.

4.3.1 Imp$^a/s$ and ASyncEffs

Specifications are annotated in /*@ ... @*/ for each module, which leads 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
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 4.3: Asynchronously reading a file, using async/await in HipHop.js.

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’s precondition {}^*.{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 method call. Module Main firstly opens the file, then creates two sub-threads via statement fork{...}par{...}. One thread calls Read,
while another thread waits for the data processing to be done and closes the file. Note that \textit{ASyncEffs} is an \textit{affine logic} in the sense that it only describes the signals we care about, regardless of the non-mentioned signals.

### 4.3.2 Forward Verification

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

(1) \{\{\text{open}\}\} \text{ (– initialize the current effects using precondition’s last instant –)}
   async loaded {
     emit loading; fs.readFile(open.value);
   }

(2) \{\{\text{open, loading}\}\} \text{ [FV-Emit]}
   } \{\{\text{open, loading}\} \cdot \{\text{loaded}\}\} \text{ [FV-Async-Branch-1]}

(3) \{\{\text{open}\}\} \text{ (– inherited from state (1) –)}
   emit compOther;

(4) \{\{\text{open, compOther}\}\} \text{ [FV-Emit]}
   await loaded;

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

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

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

(8) \{\{\text{loading}\} \cdot \{\text{loaded}\}\}||\{\{\text{compOther}\} \cdot \text{loaded}\?\cdot \{\text{logData}\} \cdot \text{close}\?\})

Φ_{final} = \{\{\text{loading, compOther}\} \cdot \{\text{loaded}\} \cdot \{\text{logData}\} \cdot \text{close}\?\}
\text{ [FV-Async] [Effects-Parallel-Merge]}

(9) Φ_{final} \sqsubseteq \Phi_{post} \text{ (–TRS: check the postcondition of the module Read; Succeed. –)}
```

Figure 4.4: A demonstration of the forward verification for the module \textit{Read}.

Figure 4.4 and Figure 4.5, demonstrate the forward verification process of the modules defined in Figure 4.3. Program effects states are captured in the form of \(\langle Φ \rangle\). To facilitate this illustration, we label the verification steps by (1), ..., (17), and mark the deployed inference rules (cf. section 4.5) in [gray].

To start the verification, states (1)(10) are initialized from the preconditions.
module Main (out open, close, loading, loaded, compOther, logData) {

(10) ⟨{}⟩ (– initialize the current effects using precondition’s last instant –)
    emit open("filePath");

(11) ⟨{open}⟩ [FV-Emit]
    fork{ Read (close, loading, loaded, compOther, logData); }

(12) {}·{open} ⊑ₚ_{Read} [FV-Call] (-TRS: Read’s precondition is satisfied-)
    ⟨{open, loading, compOther}·{loaded}·{logData}·close?⟩ [FV-Call]

(13) ⟨{open}⟩ (– inherited from state (11) –)
    par { await logData;

(14) ⟨{open}·logData?⟩ [FV-Await]
    emit close("filePath"); }

(15) ⟨{open}·logData?·{close}⟩ [FV-Emit]

(16) ⟨{open, loading, compOther}·{loaded}·{logData}·close?⟩
    ||⟨{open}·logData?·{close}⟩⟩ [FV-Fork-Par]
    ⟨{open, loading, compOther}·{loaded}·{logData}·{close}⟩
    [Effects-Parallel-Merge]

(17) (-TRS: check the postcondition of module Main; Succeed. -)
    {open, loading, compOther}·{loaded}·{logData}·{close} ⊑ {open}·{}·{close}

Figure 4.5: A demonstration of the forward verification for the module Main.

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 effects state satisfies the precondition of the callee, cf step (12). After these states transformations, steps (9)(17) invoke the TRS to check the postcondition.

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. 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 signal close, due to the reason that there is no close emitted in the current context.
CHAPTER 4. SYNCHRONOUS EFFECTS

We define \textit{well-synchronized effects} in Definition 6, and any \textit{non-synchronized effects} indicate the corresponding anti-pattern. However, non-synchronized effects can be further composed to become well-synchronized.

\textbf{Definition 6} (Well-Synchronized Effects). In this paper, we call traces without any blocking signals well-synchronized effects.

4.3.3 The TRS

Having the \textit{ASyncEffs} as the logic, we are interested in the following verification problem: Given a program \( \mathcal{P} \), and a temporal property \( \Phi' \), 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 \( \Phi' \) proves that: the program \( \mathcal{P} \) will never lead to unsafe traces which violate \( \Phi' \).

In this paper, we deploy a purely algebraic term rewriting system (TRS), to check language inclusions between \textit{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 \textit{partial derivatives} [Ant95]. There are two basic rules: [\textit{Disprove}], which infers false from trivially inconsistent inequalities; and [\textit{Unfold}], which applies Theorem 1 to generate new inequalities. In detail, given \( \Sigma \) is the whole set of the alphabet, \( \Lambda^{-1}(r) \) is the partial derivative of \( r \) w.r.t the signal \( A \).

\textbf{Definition 7} (\textit{ASyncEffs} Inclusion). For \textit{ASyncEffs} \( \Phi_1, \Phi_2 \),

\[ \Phi_1 \sqsubseteq \Phi_2 \leftrightarrow \forall I. I^{-1}(\Phi_1) \sqsubseteq I^{-1}(\Phi_2). \]

Similar to Theorem 1, we defined the Definition 7 for unfolding the inclusions between \textit{ASyncEffs}, where \( I^{-1}(\Phi) \) is the partial derivative of \( \Phi \) 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 \textit{memorization} [Bro05a].

Next, we continue with the step (9) in Figure 4.4, to demonstrate how the TRS handles \textit{ASyncEffs}. 

48
CHAPTER 4. SYNCHRONOUS EFFECTS

Table 4.2: The inclusion proving example for $A\text{SyncEffs}$.

\[
\begin{align*}
\epsilon & \sqsubseteq \epsilon \quad \text{[Prove]} \\
\{\text{Close}\} & \sqsubseteq \{\text{Close}\} \quad \text{[Unfold]} \\
\{\text{Loaded}, \text{LogData}\} \cdot \{\text{Close}\} & \sqsubseteq \{\text{LogData}\} \cdot \{\text{Close}\} \quad \text{[Unfold]} \\
\{\text{Loading, CompOther}\} \cdot \{\text{Loaded, LogData}\} \cdot \{\text{Close}\} & \sqsubseteq \{\text{CompOther}\} \cdot \{\text{LogData}\} \cdot \{\text{Close}\} \quad \text{[Unfold]} \\
\{\text{Loading, CompOther}\} \cdot \{\text{Loaded, LogData}\} \cdot \{\text{Close}\} & \sqsubseteq \{\text{CompOther}\} \cdot \{\text{LogData}\} \cdot \{\text{Close}\}.
\end{align*}
\]

Table 4.2 automatically proves (at step 3) that the inferred effects of module read satisfy the declared postcondition (after several times of unfolding at steps 1, 2, and 3). The rewriting rules (cf. subsection 4.6.2) are marked in [gray].

Note that event \{\text{Loading, CompOther}\} entails (or is subsumed by) event \{\text{CompOther}\} because the former contains more constraints. We formally define the subsumption for events in Definition 11. 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.

4.4 Language and Specifications

4.4.1 The Target Language

We summarize a core language to be our target language, which provides the infrastructure for mixed sync-async abstraction. Figure 4.6 shows the syntax of $\text{Imp}_{a/s}$. 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.
CHAPTER 4. SYNCHRONOUS EFFECTS

(Program) \[\mathcal{P} ::= \vec{md}\]

(Signal Types) \[\tau ::= \text{IN} | \text{OUT} | \text{INOUT}\]

(Module Def.) \[md ::= x (\tau \vec{S}) \langle \text{req } \Phi_{\text{pre}} \text{ ens } \Phi_{\text{post}} \rangle p\]

(Events) \[S ::= S(c)\]

(Statements) \[p, q ::= () | \text{yield} | \text{emit } S | p; q | p||q | \text{call } x (\vec{S}) | \text{loop } p\]
\[| \text{signal } S \text{ in } p | \text{present } S \text{ then } p \text{ else } q | \text{async } S p q | \text{await } S\]
\[| \text{trap } p | \text{exit } d | \text{abort } p S | \text{suspend } p S\]

(Signal Variables) \(S \in \Sigma\) (Constants) \(c \in \text{unit} \cup \mathbb{Z} \cup \mathbb{B}\) \(x \in \text{var}\) (Depth) \(d \in \mathbb{N} \cup \{0\}\)

Figure 4.6: \(\text{Imp}^{a/s}\) Syntax.

Here, \(S, x, S\) are meta-variables ranging over signal variables, constants and events, i.e., parametrized signals. Signal types are: \(\text{IN}\) for input signals, \(\text{OUT}\) for output signals and \(\text{INOUT}\) for both. \(\text{var}\) represents the countably infinite set of arbitrary distinct identifiers. A program \(\mathcal{P}\) comprises a list of module definitions \(\vec{md}\).\(^2\) Each module has a name \(x\), a list of well-typed arguments \(\tau \vec{S}\), a statement-oriented body \(p\), associated with a precondition \(\Phi_{\text{pre}}\) and a postcondition \(\Phi_{\text{post}}\). We here present the intuitive semantics of the basic statements, marked as black.

A thread of execution suspends itself for the current instant using the \text{yield} construct, and resumes when the next instant started.\(^3\) The statement \(\text{emit } S\) broadcasts the signal \(S\) to be set as \text{present}. The emission of \(S\) 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. The statement \(\text{call } x (\vec{S})\) is a call to module \(x\), providing the list of IO signals. The statement \(\text{signal } S \text{ in } p\) starts \(p\) with a local signal \(S\), overriding any \(S\) might already be declared. The statement \(\text{present } S p q\) immediately starts \(p\) if \(S\) is present in the current instant; otherwise it starts \(q\) instead. The statement \(\text{loop } p\) implements an

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

\(^3\)For a better \textit{cooperative multitasking} [Wik22a], processes voluntarily yield control periodically or when idle or logically blocked.
CHAPTER 4. SYNCHRONOUS EFFECTS

infinite loop – when \( p \) terminates, it is immediately restarted – but it is possible to be aborted or suspended.

### 4.4.2 Structural Operational Semantics of the Target Language

Figure 4.7 provides the operational semantics of the promise-related and preemptive statements in \( \text{Imp}^{a/s} \), and leave the rest (standard ones) in section A.1.

![Operational semantics of the promise-related and preemptive statements in \( \text{Imp}^{a/s} \).](image)

Figure 4.7: Operational semantics of the promise-related and preemptive statements in \( \text{Imp}^{a/s} \).
The reduction rules are in the form of $p \xrightarrow{\alpha,k}_{\mathcal{E}} p'$, meaning that a process $p$ performs an action $\alpha$ 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].

Statement $\text{async } S \ p \ q$ 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 $S$ when $p$ completes. Statement $\text{await } S$ blocks the local thread and waits for $S$ to be emitted in the environment.

Statement $\text{abort } p \ S$ performs $p$ and terminates when $S$ occurs. Statement $\text{suspend } p \ S$ suspends $p$ for one instant when $S$ is present in the environment. Statement $\text{trap } p$ installs a trap and behaves like $p$ until any exit occurs. Statement $\text{exit } d$ 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.

\begin{align*}
(1) \quad \text{halt} & \triangleq \text{loop } (\text{yield}) \\
(2) \quad \text{loop } p \text{ each } S & \triangleq \text{loop } (\text{abort } (p;\text{halt}) \ S) \\
(3) \quad \text{every } S \ p & \triangleq \text{await } S; (\text{loop } p \text{ each } S) \\
(4) \quad \text{await } (\text{delayed})S & \triangleq \text{yield}; \text{await } S
\end{align*}

Figure 4.8: Expansion of derived preemptions.

Figure 4.8 shows how to construct the derived statements via the primitives.
 Particularly, every $S\ p$ implements a preemptive loop that checks for a condition, here the presence of $S$. An every loop starts its body when its condition is true; but whenever the condition is met again in some further instants, it kills the current execution instantly to restart a new iteration. Besides, $\text{await (delayed)}S$ implements a delayed waiting which starts as early as the next instant, as opposite to the default immediate waiting.

4.4.3 An Effect Logic for the Temporal Specification

\[
\begin{align*}
\text{(Effects)} & \quad \theta ::= \bot | \epsilon | I | S? | \theta_1 \cdot \theta_2 | \theta_1 \lor \theta_2 | \theta_1 || \theta_2 | \theta^* \\
\text{(Instant)} & \quad I ::= \{} | \{S \mapsto \alpha\} | I_1 \cup I_2 \\
\text{(Paramed Sig.)} & \quad S ::= S(c) \\
\text{(Signal Statuses)} & \quad \alpha ::= \text{present} | \text{absent} | \text{undef} \\
\end{align*}
\]

Figure 4.9: Syntax of the $\text{ASyncEffs}$.

As shown in Figure 4.9 Effects comprise $false (\bot)$; the empty trace $\epsilon$; single instants $I$; waiting for a parametrized signal $S?$; trace concatenation $\theta_1 \cdot \theta_2$; trace disjunction $\theta_1 \lor \theta_2$; synchronous parallelism $\theta_1 || \theta_2$. Effects can also be constructed by $\star$, representing zero or more times repetition of a trace.

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

4.4.4 Semantic Model of $\text{ASyncEffs}$

To define the semantic model, we use $\varphi$ (a trace of instants) to represent the concrete the computation execution. Let $\varphi \models \Phi$ denote the model relation, i.e., linear temporal instants $\varphi$ satisfy the temporal effects $\Phi$, with $\varphi$ from the concrete domain: $\varphi \triangleq \text{list}(I)$. Figure 4.10 defines the semantics of $\text{ASyncEffs}$. 
CHAPTER 4. SYNCHRONOUS EFFECTS

\[ \varphi \models \epsilon \quad \text{iff} \quad \varphi = [] \]
\[ \varphi \models I \quad \text{iff} \quad \varphi = [I] \]
\[ \varphi \models S? \quad \text{iff} \quad \exists n \geq 0. \varphi = \{S\}^{n}++\{\{S\}\} \]
\[ \varphi \models \theta_1 \cdot \theta_2 \quad \text{iff} \quad \exists \varphi_1, \varphi_2, \varphi_1++\varphi_2 \text{ such that } \varphi_1 \models \theta_1 \text{ and } \varphi_2 \models \theta_2 \]
\[ \varphi \models \theta_1 \lor \theta_2 \quad \text{iff} \quad \varphi \models \theta_1 \text{ or } \varphi \models \theta_2 \]
\[ \varphi \models \theta_1 \| \theta_2 \quad \text{iff} \quad \varphi \models \theta_1 \text{ and } \varphi \models \theta_2 \]
\[ \varphi \models \theta^* \quad \text{iff} \quad \varphi \models \epsilon \text{ or } \varphi \models (\theta \cdot \theta^*) \]
\[ \varphi \models \bot \quad \text{iff} \quad \text{false} \]

Figure 4.10: Semantics of the ASyncEffs Logic.

[] 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 \{S\} and \{S\} to shorthand \{S \mapsto \text{present}\} and \{S \mapsto \text{absent}\} respectively. The signals shown in one instant represent the minimal set of signals which are required/guaranteed to be there. Any instant contains contradictions, such as \{S, \overline{S}\}, will lead to false, as the signal S can not be both present and absent.

Expressiveness.

Classical event-based LTL make use of operators such as: \[ \Box \text{ ("globally")}, \Diamond \text{ ("in the future")}. \] And the logic was subsequently extended to include the \[ U \text{ ("until") and } X \text{ ("next time") operators}. \] As shown in Table 4.3, we are able to recursively encode these basic operators into ASyncEffs, making it more intuitive and readable, mainly when nested operators occur. Besides the high compatibility with standard first-order logic, ASyncEffs make the temporal verification more flexible. Furthermore, by putting the effects in the precondition, our approach naturally enforces past-time LTL [Rei+11] along the way (here, \{A\}, \{B\} represent different instants which contain signal A and B to be present).
4.5 Automated Forward Verification

Here, we present the forward rules, i.e., an axiomatic semantics model for \( \text{Imp}^{a/s} \). These rules transfer program states and accumulate the effects syntactically. To define the rules, we introduce an environment \( E \) 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: \( E \triangleq S, h \triangleq \theta, c \triangleq I, k \in \mathbb{N} \cup \{0\} \).

The forward rules are in the form:
\[
E \vdash \langle H,C,K \rangle p \langle H',C',K' \rangle
\]
where \( E \) is the environment; \( p \) is the given statement; \( \langle H,C,K \rangle \) refers to a set of disjunctive program states. The meaning of the transition rules, can be described as:
\[
\langle H',C',K' \rangle = \bigcup_{i=0}^{|\langle H,C,K \rangle|-1} \langle H'_i, C'_i, K'_i \rangle
\]
where
\[
E \vdash \langle h_i,c_i,k_i \rangle p \langle H'_i, C'_i, K'_i \rangle
\]

\[\text{[FV-Async]}\]
de-sugars the asynchronous construct into a parallel program.

\[\text{[FV-Await]}\]
archives the current instant and appends \( S? \) to the history trace.

\[\text{[FV-Exit]}\]
updates the value of \( k \) using \( d+2 \).

\[\text{[FV-Trap]}\]
computes \( p \)'s effects. When \( K \leq 1 \), it means there is no exits to be handled, therefore the final effects is
\[|\langle H,C,K \rangle| \]
the size of \( \langle H,C,K \rangle \).
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 $\langle H, C, K-1 \rangle$.

$$
E \vdash \langle \epsilon, c, k \rangle \ p \ H, C, K \quad \langle \Delta \rangle = \mathbb{N}^{\text{Abort}(S,C,K)}_{\text{Interleave}}(H, \epsilon) \quad [FV-\text{Abort}]
$$

$$
E \vdash \langle \epsilon, c, k \rangle \ p \ H, C, K \quad \langle \Delta \rangle = \mathbb{N}^{\text{Suspend}(S,C,K)}_{\text{Interleave}}(H) \quad [FV-\text{Suspend}]
$$

**Definition 8** (Prepend Program States). Given a history trace $h'$, and program states $\Delta = (H, C, K)$, we define that: $h' \cdot \Delta = \{ (h' \cdot h, c, k) \mid (h, c, k) \in (H, C, K) \}$.

---

**Algorithm 1: Abort Interleaving**

**Input:** $S, (\Phi, I, k), \Phi_{\text{his}}$

**Output:** Program States, $\Delta$

1. rec function $\mathbb{N}^{\text{Abort}(S,I,K)}_{\text{Interleave}}(\Phi, \Phi_{\text{his}})$

2. if $\text{fst}(\Phi) = \emptyset$ then

3. $\Delta_1 \leftarrow [(\Phi_{\text{his}}, I' + \{S\}, 0)] \quad \triangleright \text{Notion } \cup \text{ unites two instants}$

4. $\Delta_2 \leftarrow [(\Phi_{\text{his}}, I' + \{S\}, k)]$

5. return $(\Delta_1 \cup \Delta_2)$

6. else

7. $\Delta \leftarrow []$

8. foreach $f \in \text{fst}(\Phi)$ do

9. $\phi \leftarrow [(\Phi_{\text{his}}, f + \{S\}, 0)]$

10. $\Phi' \leftarrow D_f(\Phi)$

11. $\Phi'_{\text{his}} \leftarrow \Phi_{\text{his}} \cdot (f + \{S\})$

12. $\Delta' \leftarrow \mathbb{N}^{\text{Abort}(S,I,K)}_{\text{Interleave}}(\Phi', \Phi'_{\text{his}})$

13. $\Delta \leftarrow \Delta \cup \phi \cup \Delta'$ \quad $\triangleright \text{Notion } \cup \text{ unites two states}$

14. return $\Delta$

15. $\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \acute
original history to the final results. Algorithm 1 presents the interleaving algorithm for 
abortions (cf. Definition 10 and Definition 12 for First\(\text{fst}\) and Derivative\(D\) respectively). Note that, we use weak abort/suspend as default in this paper. The 
difference is: in strong preemption, the body does not run when the preemption condi-
tion 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.3.

\[\text{[FV-Par]}\] computes \(p\) and \(q\)’s effects independently, then parallel merges the 
results. Notation \(\vdash_{pm}\) refers to the parallelMerge algorithm, which is detailed in 
section A.4. The forwards rules for remaining statements are in section A.2.

\[
\begin{align*}
E \vdash (e,c,k) & \quad p \langle H_1, C_1, K_1 \rangle & E \vdash (e,c,k) & \quad q \langle H_2, C_2, K_2 \rangle \\
\vdash_{pm} & \quad \langle H_1, C_1, K_1 \rangle \parallel\langle H_2, C_2, K_2 \rangle \rightsquigarrow \langle \Delta \rangle \\
E \vdash (h,c,k) & \quad p \mid\mid q \langle h \cdot \Delta \rangle
\end{align*}
\[\text{[FV-Par]}\]

**Theorem 2** (Soundness of the Forward Rules).

\(\forall p, E, \text{ if } E \vdash (h,c,k) \quad p \langle H', C', K' \rangle, \text{ and } \varphi \models h,\)

and \(\overset{e_0,0}{\overset{\epsilon}{\overset{0,1}{\overset{p_1}{\overset{e_1,0}{\overset{0,1}{\overset{p_1'}{\overset{e_1'}{\overset{0,1}{\overset{p_2}{\overset{e_2}{\overset{0,1}{\overset{p_2'}{\overset{e_2'}{\overset{0,1}{\overset{\ldots}{\overset{p_n}{\overset{e_n,0}{\overset{0,1}{\overset{p_n'}{\overset{e_n'}{\overset{0,1}{\overset{h_{ij}(\neq 1)}}{\overset{\epsilon}{\epsilon}}}}}}}}}}}}}}}}}}}}}}}}\)

then it implies that \(\exists (h', c', k') \in \langle H', C', K' \rangle \text{ such that } \varphi \leftrightarrow [e_0; e_1; \ldots; e_n] \models h' \cdot c'.\)

(Note that, \(p \overset{\epsilon_0}{\overset{0,0}{\overset{\epsilon}{\overset{0,1}{\overset{p}{\overset{0,1}{\overset{p'}{\overset{0,1}{\overset{h_{ij}(\neq 1)}}{\overset{\epsilon}{\epsilon}}}}}}}}}}\) denotes the reflexive, transitive closure of \(p \overset{\epsilon_0}{\epsilon} p'.\))

**Proof.** By induction on the structure of \(p\). See section A.5. \(\square\)

### 4.6 Temporal Verification via a TRS

The TRS is inspired by Antimirov and Mosses’ algorithm [AM95] but solving the 
language inclusions between \(\text{ASyncEffs}\). 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 \(\Phi_1, \Phi_2\), 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 \( \Phi_1 \) are legitimately allowed in the possible effects traces from the consequent \( \Phi_2 \). \( \Gamma \) is the proof context, i.e., effects inclusion hypotheses, \( \Phi \) 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 \( \Gamma = \{ \} \), \( \Phi = \epsilon \).

### 4.6.1 Auxiliary Functions: Nullable, First and Derivative

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

**Definition 9** (Nullable). Given any effect \( \theta \), \( \delta(\theta) = \text{true} \Leftrightarrow \epsilon \in \theta \), where:

\[
\begin{align*}
\delta(\bot) &= \text{false} \\
\delta(\epsilon) &= \text{true} \\
\delta(I) &= \text{false} \\
\delta(S?) &= \text{false} \\
\delta(\theta_1 \cdot \theta_2) &= \delta(\theta_1) \land \delta(\theta_2) \\
\delta(\theta_1 \lor \theta_2) &= \delta(\theta_1) \lor \delta(\theta_2) \\
\delta(\theta_1 || \theta_2) &= \delta(\theta_1) \land \delta(\theta_2)
\end{align*}
\]

**Definition 10** (First). Let \( \text{fst}(\theta) := \{ I \mid (I \cdot \theta') \in \theta \} \) be the set of head instants derivable from effect \( \theta \). (\([\theta]\) represents all the traces contained in \( \theta \))

\[
\begin{align*}
\text{fst}(\bot) &= \{ \} \\
\text{fst}(\epsilon) &= \{ \} \\
\text{fst}(I) &= \{ I \} \\
\text{fst}(\theta_1 \lor \theta_2) &= \text{fst}(\theta_1) \cup \text{fst}(\theta_2)
\end{align*}
\]

\[
\text{fst}(\theta^*) = \text{fst}(\theta)
\]

\[
\text{fst}(\theta_1 \cdot \theta_2) =
\begin{cases}
\text{fst}(\theta_1) \cup \text{fst}(\theta_2) & \text{if } \delta(\theta_1) = \text{true} \\
\text{fst}(\theta_1) & \text{if } \delta(\theta_1) = \text{false}
\end{cases}
\]

\[
\text{fst}(S?) = \{ S \mapsto \text{present} \}
\]

**Definition 11** (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,

\[ I \subseteq J \iff \{ S \mid (S \rightarrow \text{present}) \in J \} \subseteq \{ S \mid (S \rightarrow \text{present}) \in I \} \]

and \( \{ S \mid (S \rightarrow \text{absent}) \in J \} \subseteq \{ S \mid (S \rightarrow \text{absent}) \in I \} \)

**Definition 12** (Partial Derivative). The partial derivative \( D_I(\theta) \) of effects \( \theta \) w.r.t. an event \( I \) computes the effects for the left quotient \( I^{-1}J \).

\[
D_I(\perp) = D_I(\epsilon) = \perp
\]
\[
D_I(J) = \begin{cases} 
\epsilon & \text{if } I \subseteq J \\
\bot & \text{if } I \not\subseteq J
\end{cases}
\]
\[
D_I(\theta) = \begin{cases} 
\epsilon & \text{if } I \subseteq \{ S \rightarrow \text{present} \} \\
S & \text{if } I \not\subseteq \{ S \rightarrow \text{present} \}
\end{cases}
\]

\[
D_I(\theta^*) = D_I(\theta) \cdot \theta^*
\]
\[
D_I(\theta_1 \cdot \theta_2) = \begin{cases} 
D_I(\theta_1) \cdot \theta_2 \lor D_I(\theta_2) & \text{if } \delta(\theta_1) = \text{true} \\
D_I(\theta_1) \cdot \theta_2 & \text{if } \delta(\theta_1) = \text{false}
\end{cases}
\]
\[
D_I(\theta_1 \lor \theta_2) = D_I(\theta_1) \lor D_I(\theta_2)
\]
\[
D_I(\theta_1 || \theta_2) = D_I(\theta_1) || D_I(\theta_2)
\]

### 4.6.2 Rewriting Rules

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

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

\[
\Gamma \vdash \perp \subseteq \Phi \quad \text{[Bot-LHS]}
\]
\[
\Phi \neq \perp \quad \Gamma \vdash \Phi \nsubseteq \perp \quad \text{[Bot-RHS]}
\]

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., \( \epsilon \), than the consequent.

\[
\delta(es_1) \land \neg\delta(es_2) \quad \text{[Disprove]}
\]
\[
\Gamma \vdash es_1 \nsubseteq es_2
\]
\[
\text{fst}(es_1) = \{\} \quad \text{[Prove]}
\]
\[
\Gamma \vdash es_1 \subseteq es_2
\]

---

\(^6\)For example, \( \{A\}^{-1}[\{A\} \cdot \{B\}] = [\{B\}] \), and \( \{A\}^{-1}[\{A\} \lor \{B\}] = [\epsilon \lor \perp] \), cf Definition 7.
3. **Prove.** We use two rules to prove an inclusion: (i) \([\text{Prove}]\) is used when the \(\text{fst}\) set of the antecedent is empty; and (ii) \([\text{Reoccur}]\) to prove an inclusion when there exist inclusion hypotheses in the proof context \(\Gamma\), 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.

\[
\frac{(es_1 \subseteq es_3) \in \Gamma \quad (es_3 \subseteq es_4) \in \Gamma \quad (es_4 \subseteq es_2) \in \Gamma}{\Gamma \vdash es_1 \subseteq es_2} \quad \text{[Reoccur]}
\]

4. **Unfolding (Induction).** This is the inductive step of unfolding the inclusions. Firstly, we make use of the auxiliary function \(\text{fst}\) to get a set of instants \(F\), which are all the possible initial instants from the antecedent. Secondly, we obtain a new proof context \(\Gamma'\) by adding the current inclusion, as an inductive hypothesis, into the current proof context \(\Gamma\). 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.

\[
F = \text{fst}(es_1) \quad \Gamma' = \Gamma, (es_1 \subseteq es_2) \quad \forall I \in F. (\Gamma' \vdash D_I(es_1) \subseteq D_I(es_2))
\]

\[
\Gamma \vdash es_1 \subseteq es_2 \quad \text{[Unfold]}
\]

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

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

*Proof.* See section A.6 and section A.7.

\[
\square
\]

### 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.

Table 4.4 presents the evaluation results. We select 16 Imp\textsuperscript{a/s} programs, varying from 15 lines to 300 lines, and annotate \textit{ASyncEffs} specifications with a 1:1 ratio for succeeded/failed cases. The results record: \textbf{No.} for the index of the program; \textbf{LOC} for lines of code; \textbf{Infer(ms)} for effects inference time; \#\textbf{Prop(✓)} for the number of valid properties; \textbf{Avg-Prove(ms)} for the average proving time for the valid properties; \#\textbf{Prop(✗)} for the number of invalid properties; and \textbf{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.

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

7The benchmark is constructed by manually annotating \textit{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.
Discussion: Generally, the inference time increases with a linear complexity. We notice that the disprove 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 effects logic and the abstract semantics for Imp as not only tightly capture the behaviors of a mixed sync-async 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:
CHAPTER 4. SYNCHRONOUS EFFECTS

<table>
<thead>
<tr>
<th>1. ( \text{present } S_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( {S_1 \mapsto \text{undef} } )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2. ( \text{then} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( {S_1 \mapsto \text{undef}, S_1 \mapsto \text{present} } )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3. ( \text{nothing} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( {S_1 \mapsto \text{undef}, S_1 \mapsto \text{present} } )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4. ( \text{else} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( {S_1 \mapsto \text{undef}, S_1 \mapsto \text{absent} } )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5. ( \text{emit } S_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( {S_1 \mapsto \text{present}, S_1 \mapsto \text{absent} } )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>6. ( {S_1 \mapsto \text{undef}, S_1 \mapsto \text{present} } )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lor {S_1 \mapsto \text{present}, S_1 \mapsto \text{absent} } )</td>
</tr>
<tr>
<td>( \bot \lor \bot \Rightarrow \bot )</td>
</tr>
</tbody>
</table>

(a)

<table>
<thead>
<tr>
<th>6. ( {S_1 \mapsto \text{present}, S_1 \mapsto \text{present} } )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lor {S_1 \mapsto \text{undef}, S_1 \mapsto \text{absent} } )</td>
</tr>
<tr>
<td>( {S_1 \mapsto \text{present} } \lor {S_1 \mapsto \text{absent} } )</td>
</tr>
<tr>
<td>(b)</td>
</tr>
</tbody>
</table>

Table 4.5: Logically incorrect examples, caught by \( \text{ASyncEffs} \).

**Definition 13** (Contradictory event). Given any event \( I \), it is contradictory if there exists \( S \) such that \( S \mapsto \text{absent} \) \( \in I \) and \( S \mapsto \text{present} \) \( \in I \) or \( \exists S. (S \mapsto \text{undef}) \in I \) and \( S \mapsto \text{present} \) \( \in I \).

As shown in Table 4.5 (a) and (b) are both logically incorrect, because there are no valid assignments of signal \( S_1 \) and there are two possible assignments of signal \( S_1 \), 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.

As the program shows in Figure 4.11, 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 \( S_1 \)' statement. Surprisingly, this program is logically correct, since there is only one logically coherent assumption: \( S_1 \) absent and \( S_2 \) absent. With this assumption, the first present \( S_1 \) statement takes its empty else branch, which justifies \( S_1 \) absent. The second 'present \( S_1 \)' statement also takes its
empty else branch, and 'emit S2' is not executed, which justifies S2 absent. And our effects logic is able to soundly detect above mentioned correctness checking.

### 4.7.1.3 Rewriting with the Blocking Waiting Operator.

Formally, we define "waiting for the signal \(A\)" as:

\[
A ? \equiv \exists n, n \geq 0 \land \{A\}^n : \{A\},
\]

where \(\{A\}\) refers to all the events containing \(A\) to be absent.

Table 4.6: The example for Await.

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(\epsilon \subseteq \epsilon) [Prove]</td>
</tr>
<tr>
<td>2</td>
<td>({B} \subseteq {D}) &lt;br&gt;({B} \cdot {D} \subseteq ({B} \lor \bot) \cdot {D}) [Unfold]</td>
</tr>
<tr>
<td>3</td>
<td>({B} \cdot {D} \subseteq {A} \cdot {B} \cdot {D}) [Normalisation]</td>
</tr>
<tr>
<td>4</td>
<td>({B} \cdot {D} \subseteq {C} \cdot {B} \cdot {D}) [Unfold]</td>
</tr>
<tr>
<td>5</td>
<td>({C} \cdot {B} \cdot {D} \subseteq {A} \cdot {B} \cdot {D}) [Normalisation]</td>
</tr>
<tr>
<td>6</td>
<td>({A} \cdot {C} \cdot {B} \cdot {D} \subseteq {A} \cdot {B} \cdot {D}) [Unfold]</td>
</tr>
<tr>
<td>7</td>
<td>({A} \cdot {C} \cdot {B} \cdot {D} \subseteq {A} \cdot {B} \cdot {D}) [Normalisation]</td>
</tr>
<tr>
<td>8</td>
<td>({C} \cdot {B} \cdot {D} \subseteq {A} \cdot {B} \cdot {D}) [Unfold]</td>
</tr>
</tbody>
</table>

Table 4.6 shows the proof of \(\{A\} \cdot \{C\} \cdot \{B\} \cdot \{D\}\) entailing \(\{A\} \cdot \{B\} \cdot \{D\}\), as intuitively \(\{C\} \cdot \{B\}\) is a special case of \(\{B\}\). In step ① and ②, \(\{A\}\) is eliminated. In step ③, \(\{B\}\) is normalized into \(\{B\} \lor (\{B\} \cdot \{B\})\), By the step of ④, \(\{C\}\) is eliminated together with \(\{B\}\) because \(\{C\} \subseteq \{B\}\). Now the rest part is \(\{B\} \cdot \{D\} \subseteq \{B\} \cdot \{D\}\). Here, we further normalize \(\{B\}\) from the LHS into a disjunction, leading to two proof sub-trees. From the first sub-tree, we keep unfolding the inclusion with \(\{B\}\) (⑤) and \(\{D\}\) (⑥) until we can prove it. Continue with the second sub-tree, we unfold it.
with $\{B\}$; then in step $\overline{7}$ we observe the proposition is isomorphic with one of the the previous step, marked with $(\dagger)$. 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 non well-synchronized traces during the parallel merging process presented in section A.4. For example, the parallel composition of traces: $\{A\} \cdot \{B\} \cdot \{C\} \cdot \{D\} \parallel \{E\} \cdot C? \cdot \{F\}$ leads to the final behavior of $\{A, E\} \cdot \{B\} \cdot \{C\} \cdot \{D, F\}$, which is well-synchronized for all the events. However, if we were composing traces: $\{A\} \cdot \{B\} \cdot \{D\} \parallel \{E\} \cdot C? \cdot \{F\}$ due to the reasons that forgetting to emit $C$ (In JavaScript, it could be the case that forgetting to explicitly return a promise result.), it leads to a problematic trace $\{A, E\} \cdot \{B\} \cdot \{D\} \cdot C? \cdot \{F\}$. The final effects contain a dangling signal waiting of $C$, which indicates the corresponding anti-pattern.

4.8 Summary

We demonstrate how to give axiomatic semantics for $Imp^{a/s}$ 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 mixed sync-async execution model; that automates modular temporal verification for reactive programs using an expressive effects logic.
Chapter 5

Timed Effects

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 effects 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, a popular approach for specifying real-time systems has been based on Timed Automata (TAs) [AD94]. 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 proven to be successful [Wan+17; LPY97; Yov97; WWH05]. Although
TAs’ simple structure made efficient model checking feasible, 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. However, possible insufficiencies are: the expressiveness power is limited by the finite-state automata; and there is always a gap between the verified logic and the actual code implementation.

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: \(\Phi \triangleq 0 \leq t < 1 \land (_\star \cdot \text{Done})#t\). Here \(\land\) connects the arithmetic formula and the timed trace; the operator \(#\) binds time variables to traces (here \(t\) is a time bound of \((_\star \cdot \text{Done})\)); \(\_\) is a wildcard matching to any event; Kleene star \(\star\) denotes a trace repetition. The above formula \(\Phi\) corresponds to ‘\(\Diamond_{[0,1]}\text{Done}\)’ in metric temporal logic (MTL), reads “within one time unit, Done finally happens”. Furthermore, the time bounds can be dependent on the program.

\(^1\)In this paper, 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].
CHAPTER 5. TIMED EFFECTS

inputs, as shown in Figure 5.1.

```plaintext
1  void addOneSugar()
2  /* req: true ∧ _*/
3  ens: t>1 ∧ ϵ # t */
4  {
5    timeout ((), 1);
6  }
7  void addNSugar (int n)
8  /* req: true ∧ _*/
9  ens: t≥n ∧ EndSugar # t */
10  { if (n == 0) { event ["EndSugar"];}
11    else {
12      addOneSugar();
13      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 method 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 stands 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’ algorithm\textsuperscript{2} [AM95] but solving the language inclusions between more expressive $\text{TimEffs}$. 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 $\text{TimEffs}$, 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 $\text{TimEffs}$.

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

### 5.2 Overview

We use Figure 5.2 to highlight our main methodologies, which simulates a coffee machine, that dynamically adds sugar based on the user’s input number.

#### 5.2.1 $\text{TimEffs}$

We define Hoare-triple style specifications (enclosed in /*...*/) for each function, which leads to a compositional verification strategy, where static checking can be done locally. The precondition of $\text{makeCoffee}$ 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 $\text{CupReady}$ on the tail. The verification fails if the precondition is not satisfied at the caller sites. Line 17 sets a five time-units deadline

\textsuperscript{2}Antimirov and Mosses’ algorithm was designed for deciding the inequalities of regular expressions based on an axiomatic algorithm of the algebra of regular sets.
(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 occurrence of `Done` (! indicates the absence of events); and the whole process takes no more than nine time-units to hit the final event.

```c
14 void makeCoffee (int n)
15 { deadline (addNSugar(n), 5);
16  deadline (event["Coffee"],4);}
17
18 int main ()
19 { event["CupReady"];
20  makeCoffee (3);
21  event["Done"];
22 }
```

Figure 5.2: 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 5.3.3), 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 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.
5.2.2 Forward Verification.

Figure 5.3 demonstrates the forward verification of functions \texttt{addOneSugar} and \texttt{addNSugar}, defined in Figure 5.1. The effects 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 5.4.1) in [gray]. The initial states (1) and (4) are obtained from the precondition, by the \([FV-Meth]\) rule. States (5)(7)(10) are obtained by \([FV-Cond]\), which enforces the conditional constraints into the effects 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 effects 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 \(t_1\) and \(t_2\) are fresh time variables.

5.2.3 The TRS.

Having \texttt{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 \(P\), and a temporal specification \(\Phi'\), does the inclusions \(\Phi^P \subseteq \Phi'\) holds? Typically, checking the inclusion/entailment between the concrete program effects \(\Phi^P\) and the expected property \(\Phi'\) proves that: the program \(P\) will never lead to unsafe traces which violate \(\Phi'\).

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: \([\text{Disprove}]\) infers false from trivially inconsistent inclusions; and \([\text{Unfold}]\) applies Theorem 1 to generate new inclusions.

\textbf{Definition 14} (\texttt{TimEffs} Inclusion). For \texttt{TimEffs} \(\Phi_1\) and \(\Phi_2\),
1. void addOneSugar() { // initialize the state using the function precondition.
    \( \Phi_C = \Phi_{pre}^{addOneSugar(n)} = \{ \text{true} \land \_ \} \) [FV-Meth]

2. timeout ((), 1);)
    \( \Phi' = \{ t1 > 1 \land \_ \cdot (\epsilon \# t1) \} \) [FV-Timeout]

3. \( \Phi'_C \subseteq \Phi_{pre}^{addOneSugar(n)} \cdot \Phi_{post}^{addOneSugar(n)} \iff t1 > 1 \land \_ \cdot (\epsilon \# t) \subseteq t > 1 \land \_ \cdot (\epsilon \# t) \)

4. void addNSugar (int n) { // initialize the state using the function precondition.
    \( \Phi_C = \Phi_{pre}^{addNSugar(n)} = \{ \text{true} \land \_ \} \) [FV-Meth]

5. if (n == 0) {
    \{ n=0 \land \_ \} [FV-Cond]

6. event ["EndSugar"];
    \{ n=0 \land \_ \cdot EndSugar \} [FV-Event]

7. else {
    \{ n \neq 0 \land \_ \} [FV-Cond]

8. addOneSugar();
    \{ n \neq 0 \land t2 > 1 \land \_ \cdot (\epsilon \# t2) \} [FV-Call]

9. addNSugar (n-1);)}
    \{ n \neq 0 \land t2 > 1 \land \_ \cdot (\epsilon \# t) \cdot \Phi_{pre}^{addNSugar(n-1)} \} // TRS: precondition checked.
    \{ n \neq 0 \land t2 > 1 \land \_ \cdot (\epsilon \# t) \cdot \Phi_{post}^{addNSugar(n-1)} \} [FV-Call]

10. \( \Phi'_C = (n=0 \land \_ \cdot \text{Sugar}) \lor (n \neq 0 \land t2 > 1 \land \_ \cdot (\epsilon \# t) \cdot \Phi_{post}^{addNSugar(n-1)}) \) [FV-Cond]

11. \( \Phi'_C \subseteq \Phi_{pre}^{addNSugar(n)} \cdot \Phi_{post}^{addNSugar(n)} \iff \) //TRS: postcondition checked, cf. Table 5.1
    (n=0 \land \text{Sugar}) \lor (n \neq 0 \land t2 > 1 \land (\epsilon \# t) \cdot \Phi_{post}^{addNSugar(n-1)}) \subseteq \Phi_{post}^{addNSugar(n)}

    Figure 5.3: Forward verification for functions addOneSugar and addNSugar.

    \( \Phi_1 \subseteq \Phi_2 \iff \forall A. \forall t \geq 0. (A \# t)^{-1} \Phi_1 \subseteq (A \# t)^{-1} \Phi_2. \)

Similarly, we defined the Definition 14 for unfolding the inclusions between TimEffs, where \((A \# t)^{-1} \Phi\) is the partial derivative of \(\Phi\) with respect to the event \(A\) with the time bound \(t\). 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 3.2 to demonstrate how the TRS automatically proves the final effects of \textit{main} satisfying its postcondition (shown at
step (11) in Figure 5.3). We mark the rewriting rules (cf. section 3.5) in [gray].

Table 5.1: An inclusion proving example. (I) is the right hand side sub-tree of the the main rewriting proof tree.

\[
\begin{align*}
\text{(I) } & \quad n=0 \land \epsilon \subseteq tR \geq 0 \land \epsilon \# tR \\
\text{(II) } & \quad n=0 \land \epsilon \subseteq tR \geq 0 \land \epsilon \# tR \\
\text{ Step 2 } & \quad \text{LHS-OR} \\
\text{ Step 3 } & \quad \text{UNFOLD} \\
\text{ Step 4 } & \quad \text{PROVE} \\
\text{ Step 5 } & \quad \text{UNFOLD} \\
\text{ Step 6 } & \quad \text{UNFOLD} \\
\text{ Step 7 } & \quad \text{PROVE}
\end{align*}
\]

In Table 3.2, step ① renames the time variables to avoid the name clashes between the antecedent and the consequent. ES stands for the event \textit{EndSugar}. Step ② splits the proof tree into two branches, according to the different arithmetic constraints, by rule [\textit{LHS-OR}]. In the first branch, step ③ eliminates the event ES from the head of both sides, by rule [\textit{UNFOLD}]. Step ④ proves the inclusion, because evidently the consequent \( tR \geq 0 \land \epsilon \# tR \) contains \( \epsilon \) when \( tR=0 \). In the second branch, step ⑤ eliminates the a time duration \( \epsilon \# t2 \) from both sides. Therefore the rule [\textit{UNFOLD}] subtract a time duration from the consequent, i.e., \( \text{tr} \geq n \). Similarly, step ⑥ eliminates \( \text{ES} \# \text{tL} \) from the both sides, adding \( \text{tL}=(\text{tr} \cdot \text{t2}) \) to the unification constraints. Step ⑦ manages to prove that \( \text{t2} \geq 1 \land \text{tL} \geq (n-1) \land \text{tL}=(\text{tr} \cdot \text{t2}) \Rightarrow \text{tR} \geq n \) at the end of the rewriting\(^3\); therefore, the proof succeed.

5.2.4 Verifying the Fischer’s Mutual Exclusion Protocol.

Figure 5.4 presents the classical Fischer’s mutually exclusion algorithm, in \( C' \). 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. The main procedure is a parallel composition of three processes, where

\(^{3}\text{The proof obligations for arithmetic constraints are discharged by the Z3 solver [dMB08].}\)
d and e are two integer constants. Each process attempts to enter the critical section when 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 \texttt{deadline}(...,d). Then it idles for e time units, captured by \texttt{delay(e)} and then checks whether x still equals 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\leq1 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.

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

void proc (int i) {

  //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\leq1)⁺  ens_b: true ∧ ((⁺).Critical.Exit.(⁺))⁺ */
{ proc(0) || proc(1) || proc(2); }
```

Figure 5.4: Fischer’s mutually exclusion algorithm.
5.3 Language and Specifications

5.3.1 The Target Language

We define the core language $C_t$ in Figure 5.5, which is built based on C syntax and provides support for timed behavioral patterns via implicit clocks.

**(Program)**

$P ::= (\alpha^*,\text{meth}^*)$

**(Types)**

$\iota ::= \text{int} | \text{bool} | \text{void}$

**(Method)**

$\text{meth} ::= \iota \text{ mn} (\iota \ x)^* \{\text{req} \ \Phi_{\text{pre}} \ \text{ens} \ \Phi_{\text{post}}\} \{e\}$

**(Values)**

$v ::= () | c | b | x$

**(Assignment)**

$\alpha ::= x := v$

**(Expressions)**

$e ::= v | \alpha | [v]e | \text{mn}(v^*) | e_1; e_2 | e_1|e_2 | \text{if} \ v \ e_1 \ e_2$

$| \text{event}[A(v,\alpha^*)] | \text{delay}[v] | e_1 \ \text{timeout}[v] \ e_2$

$| e \ \text{deadline}[v] | e_1 \ \text{interrupt}[v] \ e_2$

**(Terms)**

$t ::= c | x | t_1+t_2 | t_1-t_2$

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

Figure 5.5: A core first-order imperative language with timed constructs.

Here, $c$ and $b$ stand for integer and Boolean constants, $mn$ and $x$ are meta-variables, drawn from $\text{var}$ (the countably infinite set of arbitrary distinct identifiers). A program $P$ comprises a list of global variable initializations $\alpha^*$ and a list of method declarations $\text{meth}^*$. Here, we use the $*$ superscript to denote a finite list of items, for example, $x^*$ refers to a list of variables, $x_1, \ldots, x_n$. Each method $\text{meth}$ has a name $mn$, an expression-oriented body $e$, also is associated with a precondition $\Phi_{\text{pre}}$ and a postcondition $\Phi_{\text{post}}$ (specification syntax is given in Figure 5.6). $C_t$ allows each iterative loop to be optimized to an equivalent tail-recursive method, where 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; method calls $\text{mn}(v^*)$; 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 \( \text{event}[A(v, \alpha^*)] \) where the event \( A \) comes from the finite set of event labels \( \Sigma \). Without loss of generality, events can be further parametrized with one value \( v \) and a set of assignments \( \alpha^* \) 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 5.3.2.

### 5.3.2 Operational Semantics of \( Ct \)

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

Process \( \text{delay}[v] \) idles for exactly \( t \) time units. Rule \([\text{delay}_1]\) states that the process may idle for any amount of time given it is less than or equal to \( t \); Rule \([\text{delay}_2]\) states that the process terminates immediately when \( t \) becomes 0.

\[
\begin{align*}
\text{delay}_1 & \quad \frac{d \leq v}{(E, \text{delay}[v]) \xrightarrow{d} (E, \text{delay}[v-d])} \\
\text{delay}_2 & \quad \frac{(E, \text{delay}[0]) \xrightarrow{\tau} (E, (j))}{(E, \text{delay}[0])}
\end{align*}
\]

In \( e_1 \ \text{timeout}[v] \ e_2 \), 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.

\[
\begin{align*}
\text{timeout}_1 & \quad \frac{(E, e_1) \xrightarrow{A} (E', e'_1)}{(E, e_1 \ \text{timeout}[v] e_2) \xrightarrow{A} (E', e'_1)} \quad \frac{(E, e_1) \xrightarrow{\tau} (E', e'_1)}{(E, e_1 \ \text{timeout}[v] e_2) \xrightarrow{\tau} (E', e'_1 \ \text{timeout}[v] e_2)} \quad \frac{(E, e_1) \xrightarrow{d} (E, e'_1)}{(E, e_1 \ \text{timeout}[v] e_2) \xrightarrow{d} (E, e'_1 \ \text{timeout}[v-d] e_2)} \\
\text{timeout}_2 & \quad \frac{(E, e_1 \ \text{timeout}[0] e_2) \xrightarrow{\tau} (E, e_2)}{(E, e_1 \ \text{timeout}[0] e_2) \xrightarrow{\tau} (E, e_2)}
\end{align*}
\]

76
CHAPTER 5. TIMED EFFECTS

Process \( \text{deadline} [v] \ e \) 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 \( e_1 \ \text{interrupt} [v] e_2 \) behaves as \( e_1 \) until \( t \) time units, and then \( e_2 \) takes over. We leave the rest rules in section B.1.

\[
\frac{\langle E, e \rangle \xrightarrow{\text{A/\tau}} \langle E', e' \rangle}{\langle E, \text{deadline}[v] e \rangle \xrightarrow{\text{A/\tau}} \langle E', \text{deadline}[v] e' \rangle} \quad [\text{ddl}_1]
\]

\[
\frac{\langle E, e \rangle \xrightarrow{\text{d}} \langle E, e' \rangle \quad (d \leq v)}{\langle E, \text{deadline}[v] e \rangle \xrightarrow{\text{d}} \langle E, \text{deadline}[v-d] e' \rangle} \quad [\text{ddl}_2]
\]

5.3.3 The Specification Language

We plant TimEffs specifications into the Hoare-style verification system, using \( \Phi_{\text{pre}} \) and \( \Phi_{\text{post}} \) to capture the temporal pre/post conditions. As shown in Figure 5.6, TimEffs can be constructed by a conditioned event sequence \( \pi \land \theta \); or an effects disjunction \( \Phi_1 \lor \Phi_2 \). Timed sequences comprise \( \text{nil} (\bot) \); empty trace \( e \); single event \( ev \); concatenation \( \theta_1 \cdot \theta_2 \); disjunction \( \theta_1 \lor \theta_2 \); 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 \( \theta \) takes \( t \) time units to complete, where \( t \) is a real-time term. A timed sequence also can be constructed by \( \theta^* \), representing zero or more
CHAPTER 5. TIMED EFFECTS

(Timed Effects) \( \Phi ::= \pi \land \theta \mid \Phi_1 \lor \Phi_2 \)

(Event Sequences) \( \theta ::= \bot \mid \epsilon \mid ev \mid \theta_1 \cdot \theta_2 \mid \theta_1 \lor \theta_2 \mid \theta_1 || \theta_2 \mid \pi?\theta \mid \theta\#t \mid \theta^* \)

(Events) \( ev ::= A(v,\alpha^*) \mid \tau(\pi) \mid \overline{A} \mid _ \)

(Pure) \( \pi ::= True \mid False \mid bop(t_1,t_2) \mid \pi_1 \land \pi_2 \mid \pi_1 \lor \pi_2 \mid \neg \pi \mid \pi_1 \Rightarrow \pi_2 \)

(Real-Time Terms) \( t ::= c \mid x \mid t_1+t_2 \mid t_1-t_2 \)

\[ c \in \mathbb{Z} \quad x \in \text{var} \quad (\text{Real Time Bound}) \# \quad (\text{Kleene Star}) \ast \]

Figure 5.6: Syntax of TimEffs.

times repetition of the trace \( \theta \). For single events, \( A(v,\alpha^*) \) stands for an observable event with label \( A \), parameterized by \( v \), and the assignment operations \( \alpha^* \); \( \tau(\pi) \) is an invisible event, parameterized with a pure formula \( \pi \).

Events can also be \( \overline{A} \), referring to all events which are not labeled using \( A \); and a wildcard \( _ \), which matches to all the events. We use \( \pi \) to denote a pure formula which captures the (Presburger) arithmetic conditions on terms or program parameters. We 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 \( t_1-t_2 \).

5.3.4 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 \( \varphi \) take \( d \) time units to complete; and they satisfies the specification \( \Phi \). To define the model, \( \text{var} \) is the set of program variables, \( \text{val} \) is the set of primitive values; and \( d, \mathcal{E}, \varphi \) are drawn from the following concrete domains: \( d: \mathbb{N}, \mathcal{E}: \text{var} \rightarrow \text{val} \) and \( \varphi: \text{list of event} \). As shown in Figure 5.7, \( + \) appends event sequences; \( \emptyset \) describes the empty sequences, \( [ev] \) represents the singleton sequence contains event \( ev \); \( [\pi]_\mathcal{E}=\text{True} \) represents \( \pi \) holds on the stack \( \mathcal{E} \). Notice that, simple events, i.e., without \( \# \), are taken to be happening in instant time.

\(^4\text{The difference between } \tau(\pi) \text{ and } \pi? \text{ is: } \tau(\pi) \text{ marks an assertion which leads to false (} \bot \text{) if } \pi \text{ is not satisfied, whereas } \pi? \text{ waits until } \pi \text{ is satisfied.} \)
As shown in Table 5.2, we are able to encode MTL operators into program inputs. The basic modal operators are: □ for "globally"; ◁ for "finally"; ○ for "next"; ⊔ for "until", and their past time reversed versions: □; ◁; and ⊔ for "previous"; $S$ 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.

5.3.4.1 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 5.2, 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: □ for "globally"; ◁ for "finally"; ○ for "next"; ⊔ for "until", and their past time reversed versions: □; ◁; and ⊔ for "previous"; $S$ 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.
CHAPTER 5. TIMED EFFECTS

Table 5.2: Examples for converting MTL formulae into TimEffs with \( t \in I \) applied.

\[
\begin{align*}
\Phi_{\text{post}} & | \Box_I A \equiv (A^*)\#t & \Diamond_I A \equiv (\_\cdot A)\#t & \bigcirc_I A \equiv (\_\cdot A)\#t & A U_I B \equiv (A^*)\#t \cdot B \\
\Phi_{\text{pre}} & | \Box_I A \equiv (A^*)\#t & \Diamond_I A \equiv (A \cdot \_\cdot)\#t & \bigcirc_I A \equiv A \cdot ((\_\cdot)\#t) & A S_I B \equiv B \cdot ((A^*)\#t)
\end{align*}
\]

The paradigmatic non-regular linear language: \( t>0 \land A \land B \) is context-free, and can be naturally expressed by TimEffs. Moreover, suppose we have a traffic light control system, we could have a specification \( n>0 \land m>0 \land (\text{Red} \#n \cdot \text{Yellow} \#m \cdot \text{Green} \#n)^* \), 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 cannot be easily expressed by finite state automata.

5.4 Automated Forward Verification

5.4.1 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 \( \pi \land \theta \). The meaning of the transition rules, can be described as:

\[
\langle \Pi', \Theta' \rangle = \bigcup_{i=0}^{\left|\langle \Pi, \Theta \rangle\right|-1} \langle \Pi_i', \Theta_i' \rangle \text{ where } (\pi_i \land \theta_i) \in \langle \Pi, \Theta \rangle \text{ and } \vdash \langle \pi_i, \theta_i \rangle \; e \; \langle \Pi_i', \Theta_i' \rangle.
\]

We here present the rules for time-related constructs and leave the rest rules in section B.2. Rule \([FV-Delay]\) creates a trace \( e\#t \), 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 \( \langle \pi, e \rangle \). 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

\[\left|\langle \Pi, \Theta \rangle\right|\text{ is the size of } \langle \Pi, \Theta \rangle, \text{ i.e., the count of conditioned event sequence } \pi \land \theta.\]
event \textit{head} (cf. Definition 17), and the tail of $\Theta_1$ respectively.

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

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

Rule \textit{FV\text{-}Interrupt} computes the interruption interleave of $e_1$'s effects, which come from the over-approximation of all the possibilities. For example, for trace $A \cdot B$, the interruption with time $t$ creates three possibilities: $(\epsilon \# t) \lor (A \# t) \lor ((A \cdot B) \# t)$. Then the rule continues to compute the effects of $e_2$; lastly, it prepends the original history $\theta$ to the final results. Algorithm 2 presents the interleaving algorithm for \textit{interruptions}, where $+$ unions program states (cf. Definition 18 and Definition 19 for $\textit{fst}$ and $\textit{D}$ functions).

\[
\begin{align*}
[FV\text{-}Interrupt] & & \\
\mathcal{E} \vdash \{ \pi, \epsilon \} \ e_1 \ \{ \Pi, \Theta \} & & \Delta = \bigcup_{i=0}^{[\Pi, \Theta]^{-1}} \mathcal{N}_{\text{Interleave}}(v, \pi, \epsilon) \\
& & \mathcal{E} \vdash \{ \Delta \} \ e_2 \ \{ \Pi', \Theta' \} \\
\mathcal{E} \vdash \langle \pi, \theta \rangle \ e_1 \ \text{interrupt}[v] \ e_2 \ \langle \Pi', \theta \cdot \Theta' \rangle
\end{align*}
\]

**Theorem 5** (Soundness of the Forward Rules). \textit{Given any system configuration $\zeta=\langle \mathcal{E}, e \rangle$, by applying the operational semantics rules, if $\langle \mathcal{E}, e \rangle \rightarrow^* \langle \mathcal{E}', v \rangle$ has execution time $d$ and produces event sequence $\varphi$; and for any history effect $\pi \wedge \theta$, such that $d_1, \mathcal{E}, \varphi_1 \vdash (\pi \wedge \theta)$, 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 $(d_1 + d), \mathcal{E}', (\varphi_1 \lor \varphi) \vdash (\pi' \wedge \theta')$. (Note that, $\zeta \rightarrow^* \zeta'$ denotes the reflexive, transitive closure of $\zeta \rightarrow \zeta'$.)}

**Proof.** See section B.3.  

81
Algorithm 2: Interruption Interleaving

Input: $v, \pi, \theta, \theta_{his}$
Output: Program States: $\Delta$

1: $\text{function } \mathcal{N}_{\text{Interrupt}}(v, \pi)(\theta, \theta_{his})$
2: $\Delta \leftarrow []$
3: $\text{foreach } f \in fst_\pi(\theta) \text{ do }$
4: $\phi \leftarrow \pi \land (t<v) \land (\theta_{his}\#t)$
5: $\theta' \leftarrow D_\pi^f(\theta)$
6: $\theta'_{his} \leftarrow \theta_{his} \cdot f$
7: $\Delta' \leftarrow \mathcal{N}_{\text{Interleave}}(\theta', \theta'_{his})$
8: $\Delta \leftarrow \Delta + \phi + \Delta'$
9: return $\Delta$

5.5 Temporal Verification via a TRS

The TRS is an automated entailment checker to prove language inclusions between $\text{TimEffs}$. 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 $\Phi_1$ and $\Phi_2$, 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 $\Gamma \vdash \Phi_1 \sqsubseteq \Phi_2$, 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 $\Phi_1$ are legitimately allowed in the timed traces described by the consequent $\Phi_2$. Here $\Gamma$ is the proof context, i.e., a set of effects inclusion hypothesis; and $\Phi$ is the history effects from the antecedent that have been used to match the effects from the consequent. Note that $\Gamma, \Phi$ are derived during the inclusion proof. The inclusion checking is initially invoked with $\Gamma=\emptyset$ and $\Phi=True \land \epsilon$.

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.

$$
\frac{\Gamma \vdash \Phi_1 \sqsubseteq \Phi \quad \Gamma \vdash \Phi_2 \sqsubseteq \Phi}{\Gamma \vdash \Phi_1 \lor \Phi_2 \sqsubseteq \Phi} \quad [LHS-OR] \\
\frac{\Gamma \vdash \Phi \sqsubseteq \Phi_1 \quad \Gamma \vdash \Phi \sqsubseteq \Phi_2}{\Gamma \vdash \Phi \sqsubseteq \Phi_1 \lor \Phi_2} \quad [RHS-OR]
$$
CHAPTER 5. TIMED EFFECTS

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

**Definition 16 (Nullable $^6$).** Given any $\Phi = \pi \land \theta$, $\delta_\pi(\theta) : \text{bool} = \begin{cases} 
    \text{true} & \text{if } \epsilon \in [\pi \land \theta] \\
    \text{false} & \text{if } \epsilon \notin [\pi \land \theta]
\end{cases}$

$\delta_\pi(\bot) = \delta_\pi(\epsilon v) = \delta_\pi(\pi') = \text{false} \quad \delta_\pi(\epsilon) = \delta(\theta^*) = \text{true} \quad \delta_\pi(\theta_1 \land \theta_2) = \delta(\theta_1) \land \delta(\theta_2)$

$\delta_\pi(\theta_1 \lor \theta_2) = \delta(\theta_1) \lor \delta(\theta_2) \quad \delta_\pi(\theta_1 || \theta_2) = \delta(\theta_1) \land \delta(\theta_2) \quad \delta_\pi(\theta \# t) = \text{SAT}(\pi \land (t=0))$

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

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

$\text{fst}_\pi(\bot) = \text{fst}_\pi(\epsilon) = \{\}$ 
$\text{fst}_\pi(A(v, \alpha^*)) = \{A(v, \alpha^*)\}$ 
$\text{fst}_\pi(\epsilon \# t) = \{t\}$ 

$\text{fst}_\pi(A(v, \alpha^*), t') = \{(A(v, \alpha^*), t') | A(v, \alpha^*) \in \text{fst}_\pi(\theta)\}$ 
$\text{fst}_\pi(\theta_1 \lor \theta_2) = \text{fst}_\pi(\theta_1) \cup \text{fst}_\pi(\theta_2)$ 

$\text{fst}_\pi(\pi' ? \theta) = \text{fst}_\pi(\theta)$ 
$\text{fst}_\pi(\theta_1 || \theta_2) = \text{fst}_\pi(\theta_1) \cup \text{fst}_\pi(\theta_2)$

$\text{fst}_\pi(\theta^*) = \text{fst}_\pi(\theta)$

$\text{fst}_\pi(\theta_1 \cdot \theta_2) = \begin{cases} 
    \text{fst}_\pi(\theta_1) \cup \text{fst}_\pi(\theta_2) & \text{if } \delta(\theta_1) = \text{true} \\
    \text{fst}_\pi(\theta_1) & \text{if } \delta(\theta_1) = \text{false}
\end{cases}$

**Definition 19 (TimEffs Partial Derivative).** Given any $\Phi = \pi \land \theta$, the partial deriva-

---

$^6$SAT($\pi$) stands for querying the Z3 theorem prover to check the satisfiability of $\pi$.
Therefore, the inclusion is invalid.

When the antecedent is nullable, while the consequent is not nullable. Intuitively, the Disprove (Heuristic Refutation).

2. 5.5.1 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, ⊥ (referring to false) entails any effects, while no non-false effects entails ⊥.

\[
\Gamma \vdash \pi \land \bot \subseteq \Phi \quad [Bot-LHS] \quad \quad \quad \quad \quad \Phi \not= \pi \land \bot \rightarrow \Gamma \vdash \Phi \not\subseteq \pi \land \bot \quad [Bot-RHS]
\]

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

\[
\frac{\delta_{\pi_1}(\theta_1) \land \neg \delta_{\pi_2}(\theta_2)}{\Gamma \vdash \pi_1 \land \theta_1 \not\subseteq \pi_2 \land \theta_2} \quad [DISPROVE] \quad \quad \quad \quad \frac{\pi_1 \Rightarrow \pi_2 \quad \text{fst}_{\pi_1}(\theta_1) = \{\}}{\Gamma \vdash \pi_1 \land \theta_1 \subseteq \pi_2 \land \theta_2} \quad [PROVE]
\]

Notice that the derivatives of a parallel composition makes use of the Parallel Derivative $\tilde{D}_h^\pi(\theta)$, defined as follows:

\[
\tilde{D}_h^\pi(\theta) = \begin{cases} 
\pi \land \theta & \text{if } D_h^\pi(\pi \land \theta) = (\text{false} \land \bot) \\
D_h^\pi(\theta) & \text{otherwise}
\end{cases}
\]

CHAPTER 5. TIMED EFFECTS
3. **Prove.** We use two rules to prove an inclusion: (i) \([\textit{PROVE}]\) is used when the antecedent has no head; and (ii) \([\textit{REOCCUR}]\) proves an inclusion when there exist inclusion hypotheses in the proof context \(\Gamma\), which are able to soundly prove the current goal. The special case of \([\textit{REOCCUR}]\) is when the identical inclusion is shown in the proof context, then the TRS then terminates and proves it valid.

\[
\begin{align*}
\text{\([\textit{REOCCUR}]\)} & \quad \frac{(\pi_1 \land \theta_1 \sqsubseteq \pi_3 \land \theta_3) \in \Gamma \quad (\pi_3 \land \theta_3 \sqsubseteq \pi_4 \land \theta_4) \in \Gamma \quad (\pi_4 \land \theta_4 \sqsubseteq \pi_2 \land \theta_2) \in \Gamma}{\Gamma \vdash \pi_1 \land \theta_1 \sqsubseteq \pi_2 \land \theta_2}
\end{align*}
\]

4. **Unfolding (Induction).** This is the inductive step of unfolding the inclusions. Firstly, we make use of the \(\textit{fst}\) function to get \(H\), which are all the possible initial events from the antecedent. Secondly, we obtain a new proof context \(\Gamma'\) by extending \(\Gamma\) with the current inclusion, as an inductive hypothesis. Thirdly, we iterate each element \(h \in H\), and compute the partial derivatives (\textit{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.

\[
\begin{align*}
\text{\([\textit{UNFOLD}]\)} & \quad \frac{H = \text{\(\text{fst}_{\pi_1}(\theta_1)\)} \quad \Gamma' = \Gamma, (\pi_1 \land \theta_1 \sqsubseteq \pi_2 \land \theta_2) \quad \forall h \in H. \ (\Gamma' \vdash D^n_h(\theta_1) \sqsubseteq D^n_h(\theta_2))}{\Gamma \vdash \pi_1 \land \theta_1 \sqsubseteq \pi_2 \land \theta_2}
\end{align*}
\]

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

*Proof.* See section B.4.

**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.5.

### 5.6 Implementation and Evaluation

To show the feasibility, we prototype our automated verification system using OCaml (\(\sim 5k\) LOC); and prove soundness for both the forward verifier and the
CHAPTER 5. TIMED EFFECTS

TRRS. 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\textsuperscript{t}; and the inclusion checking for TimEffs with the parallel composition, blok waiting operator and shared global variables.

Table 5.3: Experimental Results for Manually Constructed Synthetic Examples.

<table>
<thead>
<tr>
<th>No.</th>
<th>LOC</th>
<th>Forward(ms)</th>
<th>#Prop(\checkmark)</th>
<th>Avg-Prove(ms)</th>
<th>#Prop(\xmark)</th>
<th>Avg-Dis(ms)</th>
<th>#AskZ3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>26</td>
<td>0.006</td>
<td>5</td>
<td>52.379</td>
<td>5</td>
<td>21.31</td>
<td>77</td>
</tr>
<tr>
<td>2</td>
<td>37</td>
<td>43.955</td>
<td>5</td>
<td>83.374</td>
<td>5</td>
<td>52.165</td>
<td>188</td>
</tr>
<tr>
<td>3</td>
<td>44</td>
<td>32.654</td>
<td>5</td>
<td>52.524</td>
<td>5</td>
<td>33.444</td>
<td>104</td>
</tr>
<tr>
<td>4</td>
<td>72</td>
<td>202.181</td>
<td>5</td>
<td>82.922</td>
<td>5</td>
<td>55.971</td>
<td>229</td>
</tr>
<tr>
<td>5</td>
<td>98</td>
<td>42.706</td>
<td>7</td>
<td>149.345</td>
<td>7</td>
<td>60.325</td>
<td>306</td>
</tr>
<tr>
<td>6</td>
<td>134</td>
<td>403.617</td>
<td>7</td>
<td>160.932</td>
<td>7</td>
<td>292.304</td>
<td>940</td>
</tr>
<tr>
<td>7</td>
<td>133</td>
<td>51.492</td>
<td>7</td>
<td>17.901</td>
<td>7</td>
<td>47.643</td>
<td>118</td>
</tr>
<tr>
<td>8</td>
<td>173</td>
<td>57.114</td>
<td>7</td>
<td>40.772</td>
<td>7</td>
<td>30.977</td>
<td>128</td>
</tr>
<tr>
<td>9</td>
<td>182</td>
<td>872.995</td>
<td>9</td>
<td>252.123</td>
<td>9</td>
<td>113.838</td>
<td>1142</td>
</tr>
<tr>
<td>10</td>
<td>210</td>
<td>546.222</td>
<td>9</td>
<td>146.341</td>
<td>9</td>
<td>57.832</td>
<td>570</td>
</tr>
<tr>
<td>11</td>
<td>240</td>
<td>643.133</td>
<td>9</td>
<td>146.268</td>
<td>9</td>
<td>69.245</td>
<td>608</td>
</tr>
<tr>
<td>12</td>
<td>260</td>
<td>1032.31</td>
<td>9</td>
<td>242.699</td>
<td>9</td>
<td>123.054</td>
<td>928</td>
</tr>
<tr>
<td>13</td>
<td>265</td>
<td>1255.05</td>
<td>11</td>
<td>150.999</td>
<td>11</td>
<td>117.288</td>
<td>2465</td>
</tr>
<tr>
<td>14</td>
<td>286</td>
<td>12257.834</td>
<td>11</td>
<td>501.994</td>
<td>11</td>
<td>257.800</td>
<td>3090</td>
</tr>
<tr>
<td>15</td>
<td>287</td>
<td>1383.034</td>
<td>11</td>
<td>546.064</td>
<td>11</td>
<td>407.952</td>
<td>1489</td>
</tr>
<tr>
<td>16</td>
<td>337</td>
<td>49873.835</td>
<td>11</td>
<td>1863.901</td>
<td>11</td>
<td>954.996</td>
<td>15505</td>
</tr>
</tbody>
</table>

Table 5.3 presents the evaluation results for another 16 C\textsuperscript{t} programs\textsuperscript{7}, and the 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(\checkmark), number of valid properties; Avg-Prove(ms),

\textsuperscript{7}All programs contain timed constructs, conditionals, and parallel compositions.
average proving time for the valid properties; $\#\text{Prop}(\checkmark)$, number of invalid properties; $\text{Avg-Dis}(\text{ms})$, average disproving time for the invalid properties; $\#\text{AskZ3}$, number of querying Z3 through out the experiments.

**Observations:** i) the proving/disproving time increases when the effect computation time increases because larger $\text{Forward}(\text{ms})$ indicates the higher complexity with respect to the timed constructs, which complicates the inclusion checking; ii) while the number of querying Z3 per property ($\#\text{AskZ3}/(\#\text{Prop}(\checkmark)+\#\text{Prop}(\times))$) 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 ($\#\text{Proc}$) in PAT and Uppaal respectively. For our system, based on the implementation presented in Figure 5.4, 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 ($\text{Prove(s)}$) and disproving ($\text{Disprove(s)}$) time and their number of uniquely querying Z3 ($\#\text{AskZ3-u}$).

<table>
<thead>
<tr>
<th>$#\text{Proc}$</th>
<th>$\text{Prove(s)}$</th>
<th>$#\text{AskZ3-u}$</th>
<th>$\text{Disprove(s)}$</th>
<th>$#\text{AskZ3-u}$</th>
<th>PAT(s)</th>
<th>Uppaal(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.09</td>
<td>31</td>
<td>0.110</td>
<td>37</td>
<td>$\leq 0.05$</td>
<td>$\leq 0.09$</td>
</tr>
<tr>
<td>3</td>
<td>0.21</td>
<td>35</td>
<td>0.093</td>
<td>42</td>
<td>$\leq 0.05$</td>
<td>$\leq 0.09$</td>
</tr>
<tr>
<td>4</td>
<td>0.46</td>
<td>63</td>
<td>0.120</td>
<td>47</td>
<td>0.05</td>
<td>0.09</td>
</tr>
<tr>
<td>5</td>
<td>25.0</td>
<td>84</td>
<td>0.128</td>
<td>52</td>
<td>0.15</td>
<td>0.19</td>
</tr>
</tbody>
</table>

**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 already queried constraints.

5.6.3 Case Study: Prove it when Reoccur.

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

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( I ) : ( I(tL&lt;3 \land (A^#tL) \cdot B) \lor (True \land B) \subseteq tR&lt;4 \land (A^#tR) \cdot B )</td>
</tr>
<tr>
<td>2</td>
<td>[UNFOLD] ( tL&lt;3 \land tL^1+ tL^2 = tL \land tR = tR^1 + tR^2 \land tL^1 = tR^1 \land tL^2 = tR^2 \Rightarrow tR&lt;4 )</td>
</tr>
<tr>
<td>3</td>
<td>( I(prove) ) : ( True \land \epsilon \subseteq tR = 0 \land \epsilon )</td>
</tr>
<tr>
<td>4</td>
<td>( I(normal) ) : ( True \land B \subseteq tR = 0 \land B )</td>
</tr>
<tr>
<td>5</td>
<td>( tl&lt;3 \land (A^#tL) \cdot B \subseteq tR&lt;4 \land (A^#tR) \cdot B )</td>
</tr>
<tr>
<td>6</td>
<td>( tl&lt;3 \land (A^#tL) \cdot B \subseteq tR&lt;4 \land (A^#tR) \cdot B )</td>
</tr>
<tr>
<td>7</td>
<td>[UNFOLD] ( tl&lt;3 \land A^#tL^1 \cdot A^#tL^2 \cdot B \subseteq tR&lt;4 \land A^#tR^f \cdot A^#tR^2 \cdot B )</td>
</tr>
<tr>
<td>8</td>
<td>( tl&lt;3 \land (A^#tL^1 \cdot A^#tL^2) \cdot B \subseteq tR&lt;4 \land (A^#tR^1 \cdot A^#tR^2) \cdot B )</td>
</tr>
<tr>
<td>9</td>
<td>[SPLIT] ( tl^1 + tl^2 = tl \land tR^1 + tR^2 = tR )</td>
</tr>
<tr>
<td>10</td>
<td>[REOCCUR] ( tl^1 \cdot B \negless B )</td>
</tr>
</tbody>
</table>

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 \( \epsilon \), 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 \( tl^1 + tl^2 \) and \( tR^1 + tR^2 \). Then in step 6, \( A^\#tL^1 \) together with \( A^\#tR^f \) are eliminated, unifying \( tl \) and \( tR^1 \) by adding the side constraint \( tl^1=tR^1 \). In step 8, we observe the proposition is isomorphic with one of the the previous step, marked using (‡). Hence we apply the rule \[REOCCUR\] to prove it with a succeed side constraints entailment.

5.6.4 Discussion.

Our implementation is the first 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; Bjo+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 effects logic \textit{TimEffs}, to capture real-time behavioral patterns and temporal properties. We demonstrate how to build axiomatic semantics (or rather an effects system) for \textit{C}' 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 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

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 effects 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, \texttt{effect Yield : int \to unit}, declares the \texttt{Yield} effect, to be used in the \textit{generator functions}. When it is performed, the program suspends its current execution and returns the yielded \texttt{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. In this paper, we tackle the following verification challenges:

\begin{enumerate}
  \item \textit{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? In this paper, our forward inference rules show 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.
  \item \textit{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 \texttt{f()} performs the predefined effect \texttt{Foo}; then \texttt{loop()} handles effect \texttt{Foo} by resuming a closure which in turn performs \texttt{Foo} when applied.
\end{enumerate}

With higher-order effect signatures and in the setting of deep handlers\footnote{A deep handler is persistent: after it has handled one effect, it remains installed, as the topmost frame of the captured continuation [HL18; KLO13].}, the
CHAPTER 6. CONTINUATION BASED EFFECTS

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

2  let f() = perform Foo ()

3  let loop() = match f () with

4  | _ -> () (* normal return *)

5  | effect Foo k -> continue k (fun () -> perform Foo ())
```

Figure 6.1: A loop caused by the effects handler.

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. In this paper, we devise ContEffs, i.e., extended regular expressions with arithmetic constraints, to provide more precise specifications by integrating: \( \star \) for finite traces; \( \omega \) for infinite traces; \( \infty \) 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: \( (_\star \cdot A \cdot B)^* \), where _ is a wildcard matching to any events; \( \star \) denotes a repeated pattern; \( B \) denotes the negation of an effect \( B \). 'Function `send(int n)` terminates when \( n \) is non-negative, otherwise it does not terminate' is expressed as: \( n \geq 0 \land (_)^* \lor n < 0 \land (_)^\omega \), which is beyond LTL.
Having _ContEffs_ as the specification language, we are interested in the following verification problem: Given a program \( P \), and a temporal property \( \Phi' \), does \( \Phi^P \sqsubseteq \Phi' \) hold\(^2\)? In a typical verification context, checking the inclusion/entailment between the program effects \( \Phi^P \) and the valid traces \( \Phi' \) proves that: the program \( P \) will never lead to unsafe traces which violate \( \Phi' \).

To effectively check \( \Phi^P \sqsubseteq \Phi' \), we deploy a purely algebraic TRS inspired by Antimirov and Mosses’ 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 authors’ 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 inference 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.

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.

\(^2\)The inclusion notation \( \sqsubseteq \) is formally defined in Definition 21.
6.2 Overview

6.2.1 A Sense of ContEffs in File I/O

We define Hoare-triple style specifications, marked in lavender, for each program, which leads to a compositional verification strategy, where temporal reasoning can be done locally. We model an abstract form of file I/O in Figure 6.2 Effects Open and Close are both declared to be performed with a value of type int, indexing the operated file.

```
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)
```

Figure 6.2: 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 nth file has been requested to be Opened, and not been requested to be Closed\(^3\).

\(^3\)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.
CHAPTER 6. CONTINUATION BASED EFFECTS

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 6.5. The verification fails when the real implementation violates the specifications.

```ocaml
1 effect Goo : (unit -> unit)
2
3 let f_g ()
4 /*@ req _^* @*/
5 /*@ ens Foo!.Goo!.Foo?() @*/
6 = let f = perform Foo in
7    let g = perform Goo in
8    f () (* g is abandoned *)
9
10 let loop ()
11 /*@ req _^* @*/
12 /*@ ens _^*.((Foo.Goo)^w @*/
13 = match f_g () with
14   | _ -> ()
15   | effect Foo k -> continue k (fun () -> perform Goo ())
16   | effect Goo k -> continue k (fun () -> perform Foo ())
```

Figure 6.3: Another Loop caused by the effects handler.

6.2.2 Effects Inferences via a Fixpoint Calculation

We continue to examine a variant of the so-called 'recursive cow' benchmark program [Git22] in Figure 6.3, which generates an infinite trace. 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 $\text{Goo}$. The performing of $\text{Goo}$ brings us to the handler at line 18, which resumes a closure that performs $\text{Foo}$ when applied. The resulting postcondition, deploys the $\omega$ operator, states that $\text{loop finally}$ performs an infinite succession of alternating $\text{Foo}$ and $\text{Goo}$ effects. In fact, our fixpoint calculator computes the final effects for $\text{loop}$ as $\text{Foo} \cdot \text{Goo} \cdot \text{Foo} \cdot (\text{Goo} \cdot \text{Foo})^\omega$, which entails the declared postcondition (c.f. Figure 6.4).

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.

### 6.2.3 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. In detail, given $\Sigma$ is the whole set of the alphabet, $D_A(r)$ is the partial derivative of $r$ with respect to the event $A$.

Similarly, we formally define the inclusion of $\text{ContEffs}$ in Definition 21.

Next we present the effects inclusion, generated from Figure 6.3, proving process for the post condition checking in Figure 6.4 Termination is guaranteed because the set of derivatives to be considered is finite, and possible cycles are detected using memorization. We use ♠ to indicate such pairings. The rewriting rules are defined in section 6.5. In particular, the rule [Reoccur] finds the syntactic identity from the internal proof tree, for the current open goal [Bro05b].
6.3 Language and Specifications

6.3.1 The Target Language

6.3.1.1 Syntax.

We target a minimal, ML-like (typed, higher-order, call-by-value) core pure language, defined in Figure 6.5. Here, \(c\), \(x\) and \(A\) are meta-variables ranging respectively over integer constants, variables, and labels of effects.

A program \(P\) comprises a list of effect declarations \(\text{eff}^*\) and a list of method definitions \(\text{meth}^*\); the * superscript denotes a finite, possibly empty list of items. Programs are typed according to basic types \(\tau\). Each method \(\text{meth}\) has a name \(mn\), an expression body \(e\), and pre and postcondition \(\Phi_{\text{pre}}\) and \(\Phi_{\text{post}}\) (the syntax of effect specifications \(\Phi\) is given in Figure 6.7). Constructs like sequencing are defined via elaboration to more primitive forms.

6.3.1.2 Operational Semantics of \(\lambda_h\).

As shown in Figure 6.6, the reduction rules up to those for \text{match} are standard. Matching on a pure value results in the body of the always-present \text{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 \(\text{perform } A(v, \lambda x \Rightarrow e)\) works informally: it performs the effect \(A\) (e.g. a shared-memory read) with argument \(v\) (e.g. the memory location to be read). The
(Program) \[ P ::= \text{eff} \star \text{meth} \star \]

(Effect Declarations) \[ \text{eff} ::= A : \tau \]

(Method Definition) \[ \text{meth} ::= \tau \text{mn} (\tau v) [\text{req} \Phi_{\text{pre}} \text{ens} \Phi_{\text{post}}] \{ e \} \]

(Types) \[ \tau ::= \text{bool} | \text{int} | \text{unit} | \tau_1 \rightarrow \tau_2 \]

(Values) \[ v ::= c | x | \lambda x \Rightarrow e \]

(Handler) \[ h ::= (\text{return } x \mapsto e | ocs) \]

(Operation Cases) \[ ocs ::= \emptyset | \{ \text{effect } A(x, \kappa) \mapsto e \} \uplus ocs \]

(Expressions) \[ e ::= v | v_1 v_2 | \text{let } x = v \text{ in } e | \text{if } v \text{ then } e_1 \text{ else } e_2 | \text{perform } A(v, \lambda x \Rightarrow e) | \text{match } e \text{ with } h | \text{resume } v \]

(Selected Elaborations)

\[ e_1; e_2 \Rightarrow \text{let } () = e_1 \text{ in } e_2 \]

\[ e_1 \ e_2 \Rightarrow \text{let } f = e_1 \text{ in } \text{let } x = e_2 \text{ in } (f \ x) \]

\[ \text{perform } A(e_1, \lambda y \Rightarrow e_2) \Rightarrow \text{let } x = e_1 \text{ in } \text{perform } A(x, \lambda y \Rightarrow e_2) \]

\[ \text{let } x = \text{perform } A(v, \lambda y \Rightarrow e_1) \text{ in } e_2 \Rightarrow \text{perform } A(v, \lambda y \Rightarrow \text{let } x = e_1 \text{ in } e_2) \]

\[ \text{perform } A(v) \Rightarrow \text{perform } A(v, \lambda x \Rightarrow x) \]

\[ c \in \mathbb{Z} \cup \mathbb{B} \cup \text{unit} \quad x, y, mn, \kappa \in \text{var} \quad A \in \Sigma \]

Figure 6.5: Syntax of \( \lambda h \), a minimal, ML-like language with user-defined effects and handlers.

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 read is implemented is defined by handlers which enclose the \text{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 \text{match} "pushed" into the continuation, to handle subsequent \text{performs}.

99
(Evaluation contexts) \[ E ::= \text{box} \mid \text{let } x=E \text{ in } e \mid \text{match } E \text{ with } h \]

(Reduction rules)
\[ E[e_1] \rightarrow E[e_2] \text{ if } e_1 \rightarrow e_2 \]
\[ \text{let } x=v \text{ in } e \rightarrow e[v/x] \]
\[ (\lambda x \Rightarrow e) \ v \rightarrow e[v/x] \]
\[ \text{if true then } e_1 \text{ else } e_2 \rightarrow e_1 \]
\[ \text{if false then } e_1 \text{ else } e_2 \rightarrow e_2 \]
\[ \text{match } v \text{ with } h \rightarrow e[v/x] \text{ if } \text{return } x \mapsto e \in h \]
\[ \text{match } (\text{perform } A(v, \lambda y \Rightarrow e_1)) \text{ with } h \rightarrow e_2[v/x][(\lambda y \Rightarrow \text{match } e_1 \text{ with } h)/\kappa] \]
\[ \text{if } (\text{effect } A(x, \kappa) \Rightarrow e_2) \in h \]
\[ \text{match } (\text{perform } A(v, \lambda y \Rightarrow e_1)) \text{ with } h \rightarrow \text{perform } A(v, \lambda y \Rightarrow \text{match } e_1 \text{ with } h) \]
\[ \text{if } A \notin h \]

Figure 6.6: Evaluation contexts and reduction rules

6.3.2 The Specification Language

6.3.2.1 Syntax.

We enrich a Hoare-style verification system with effect specifications, using the notation \( \{\text{req } \Phi_{\text{pre}} \ens \Phi_{\text{post}}\} \) for function pre and postcondition. As defined in Figure 3.3, \( \Phi \) is a set of disjunctive tuples including a pure formula \( \pi \), an event sequence \( \theta \), and a return value \( v \).

\( A \) is an effect label drawn from \( \Sigma \), 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
(ContEffs) $\Phi ::= \forall (\pi, \theta, v)$

(Parameterized Label) $l ::= \Sigma(v)$

(Event Sequences) $\theta ::= \bot | \epsilon | ev | Q | \theta_1 \cdot \theta_2 | \theta_1 \lor \theta_2 | \theta^* | \theta^\infty | \theta^\omega$

(Single Events) $ev ::= \_ | l | \bar{l}$

(Placeholders) $Q ::= l! | l? (v)$

(Pure formulae) $\pi ::= True | False | R(t_1, t_2) | \pi_1 \land \pi_2 | \pi_1 \lor \pi_2$

$| \neg \pi | \pi_1 \Rightarrow \pi_2$

(Terms) $t ::= n | x | t_1 + t_2 | t_1 - t_2$

Figure 6.7: Syntax of ContEffs.

$x \in \text{var} \quad (\text{Finite Kleene Star}) \star \quad (\text{Finite/Infinite}) \infty \quad (\text{Infinite}) \omega$

the fixed-point reasoning (subsection 6.4.2).

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

Pure formulae $\pi$ 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.

6.3.2.2 Semantic Model of ContEffs.

To define the model, $var$ is the set of program variables, $val$ is the set of primitive values, $\alpha$ is the set of concrete events drawn from single events $l$ or placeholders $Q$. Let $\mathcal{E}, \varphi \models \Phi$ denote the models relation, i.e., the context $\mathcal{E}$ and linear temporal events $\varphi$ satisfy the effect specification $\Phi$, where $\mathcal{E}$ records the stack status and the bindings from variables to placeholders, $\mathcal{E} \triangleq \text{var} \rightarrow (\text{val } \cup Q)$; and $\varphi$ is a list of
CHAPTER 6. CONTINUATION BASED EFFECTS

\[ \varphi \triangleq [\alpha]. \]

\[ \mathcal{E}, \varphi \models \Phi \quad \text{iff} \quad \exists (\pi, \theta, v) \in \Phi. \mathcal{E}, \varphi \models (\pi, \theta, v) \]

\[ \mathcal{E}, \varphi \models (\pi, \epsilon) \quad \text{iff} \quad [\pi]_{\mathcal{E}} = \text{True and } \varphi = [\emptyset] \]

\[ \mathcal{E}, \varphi \models (\pi, \_ \_ ) \quad \text{iff} \quad [\pi]_{\mathcal{E}} = \text{True and } \exists l \in \Sigma(v), \varphi = [l] \]

\[ \mathcal{E}, \varphi \models (\pi, l) \quad \text{iff} \quad J_{\pi} K_{E} = \text{True and } \varphi = [l] \]

\[ \mathcal{E}, \varphi \models (\pi, l) \quad \text{iff} \quad J_{\pi} K_{E} = \text{True and } \varphi = [l] \]

\[ \mathcal{E}, \varphi \models (\pi, Q) \quad \text{iff} \quad [\pi]_{\mathcal{E}} = \text{True and } \varphi = [Q] \]

\[ \mathcal{E}, \varphi \models (\pi, \theta_{1} \cdot \theta_{2}) \quad \text{iff} \quad \exists \varphi_{1}, \varphi_{2}. \varphi = \varphi_{1} \cdot \varphi_{2} \text{ and } \mathcal{E}, \varphi_{1} \models (\pi, \theta_{1}) \text{ and } \mathcal{E}, \varphi_{2} \models (\pi, \theta_{2}) \]

\[ \mathcal{E}, \varphi \models (\pi, \theta_{1} \vee \theta_{2}) \quad \text{iff} \quad \mathcal{E}, \varphi \models (\pi, \theta_{1}) \text{ or } \mathcal{E}, \varphi \models (\pi, \theta_{2}) \]

\[ \mathcal{E}, \varphi \models (\pi, \star \_ ) \quad \text{iff} \quad \mathcal{E}, \varphi \models (\pi, \epsilon) \text{ or } \mathcal{E}, \varphi \models (\pi, \theta \cdot \theta^{*}) \]

\[ \mathcal{E}, \varphi \models (\pi, \theta^{\infty}) \quad \text{iff} \quad \mathcal{E}, \varphi \models (\pi, \theta^{*}) \text{ or } \mathcal{E}, \varphi \models (\pi, \theta^{\omega}) \]

\[ \mathcal{E}, \varphi \models (\pi, \theta^{\omega}) \quad \text{iff} \quad \mathcal{E}, \varphi \models (\pi, \theta \cdot \theta^{\omega}) \]

\[ \mathcal{E}, \varphi \models (\text{False}, \bot) \quad \text{iff} \quad \text{false} \]

Figure 6.8: 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 6.8: \([\_]\) 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.

6.3.3 Instrumented Semantics.

To facilitate the soundness proof in Theorem 8 for the verification rules presented in section 6.4, we also define an instrumented reduction relation \( \xrightarrow{i} \), which operates on program states of the form \( [e, \mathcal{E}, \varphi] \), 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 \( e \xrightarrow{i} e' \) and a most general high-order
effects signature \((A : \tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)) \in \mathcal{P}\):

\[
\begin{align*}
[!ht] & \quad e = v_1 v_2 \quad E(v_1) = A(v) ? \\
& \quad \frac{}{[e, E, \Phi] \xrightarrow{t} [e', E, \Phi++] [A(v)?(v_2)]} \quad [\text{Inst-App}]
\end{align*}
\]

\[
\begin{align*}
[!ht] & \quad e = \text{let } x = v \text{ in } e_1 \\
& \quad \frac{}{[e, E, \Phi] \xrightarrow{t} [e', (x \mapsto v) :: E, \Phi]} \quad [\text{Inst-Bind}]
\end{align*}
\]

\[
\begin{align*}
[!ht] & \quad e = \text{match perform } A(v, \lambda x \Rightarrow e_1) \text{ with } h \quad A \notin h \\
& \quad \frac{}{[e, E, \Phi] \xrightarrow{t} [e', (x \mapsto A(v)) ? :: E, \Phi++] [A(v)!]} \quad [\text{Inst-Escape}]
\end{align*}
\]

\[
\begin{align*}
[!ht] & \quad e = \text{match perform } A(v, \lambda x \Rightarrow e_1) \text{ with } h \quad A \in h \\
& \quad \frac{}{[e, E, \Phi] \xrightarrow{t} [e', E, \Phi++] [A(v)]} \quad [\text{Inst-Caught}]
\end{align*}
\]

6.4 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. \(\mathcal{P}\) denotes the program being checked. With pre/postcondition declared for each method in \(\mathcal{P}\), we apply modular verification to a method’s body using Hoare-style triples \(\mathcal{E} \vdash \langle \Phi \rangle \ e \langle \Phi' \rangle\) where \(\mathcal{E}\) is the context; if \(\Phi\) describes the effects which have been performed since the beginning of \(\mathcal{P}\), if \(e\) terminates, \(\Phi'\) describes the effects that will have been performed after.

6.4.1 Forward Verification Rules

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

\[
\frac{\mathcal{E} \vdash \langle \Phi_{\text{pre}} \rangle e \langle \Phi \rangle \quad \Phi \sqsubseteq \Phi_{\text{pre}} \cdot \Phi_{\text{post}}}{\vdash \tau \text{ mn} (\tau v) \; \text{[FV-Meth]}} \]

**Definition 20** (*ContEffs* Concatenation). Given two *ContEffs* \( \Phi_1 \) and \( \Phi_2 \), \( \Phi_1 \cdot \Phi_2 = \{ (\pi_1 \land \pi_2, \theta_1 \cdot \theta_2, v_2) \mid (\pi_1, \theta_1, v_1) \in \Phi_1, (\pi_2, \theta_2, v_2) \in \Phi_2 \} \)

[FV-Perform] concatenates a placeholder to the current effects, where \( \Phi \cdot A(v)! \equiv \{ (\pi, \theta \cdot A(v)!, v) \mid (\pi, \theta, v) \in \Phi \} \), then extends the environment by binding \( x \) to \( A(v) \), referring to the resumed value of performing \( A(v) \).

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

For applications \( v_1 \; v_2 \), if \( v_1 \) is a function definition with annotated specifications, [FV-Call] checks whether the instantiated precondition of callee, \( \Phi_{\text{pre}}[v_2/v] \), is satisfied by the current effects state, then it obtains the next effects state by concatenating the instantiated postcondition, \( \Phi_{\text{post}}[v_2/v] \), to the current effects state; if \( v_1 \) maps to \( \mathbf{I} \), [FV-App] concatenates \( \mathbf{I}(v_2) \) into the current effect state, referring to the effects generated by applying \( v_2 \) to the value resumed from performing \( \mathbf{I} \).

[FV-Value] updates the current return value.

\[
\frac{\mathcal{E}(v_1) = \tau \text{ mn} (\tau v) \; \text{[FV-Call]}}{\mathcal{E} \vdash \langle \Phi \rangle \; v_1 \; v_2 \; \langle \Phi \cdot \Phi_{\text{post}}[v_2/v] \rangle \; \text{[FV-Call]}}
\]

\[
\frac{\mathcal{E}(v_1) = \mathbf{I} \quad \theta' = \mathbf{I}(v_2) \; \text{[FV-App]}}{\mathcal{E} \vdash \langle \Phi \rangle \; v_1 \; v_2 \; \langle \Phi \cdot \theta' \rangle \; \text{[FV-App]}}
\]

\[
\frac{\Phi' = \{ (\pi, \theta, v') \mid (\pi, \theta, v) \in \Phi \}}{\mathcal{E} \vdash \langle \Phi \rangle \; v' \; \langle \Phi' \rangle \; \text{[FV-Value]}}
\]

[FV-If-Else] unions the effects from both branches, where \( \Phi \land \pi' \equiv \{ (\pi \land \pi', \theta, v) \mid (\pi, \theta, v) \in \Phi \} \). [FV-Let] extends \( \mathcal{E} \) with \( x \) binding to \( v \).

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

104
6.4.2 Fixpoint Computation.

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

$$
\forall (\pi, \theta, v) \in \Phi. \|\mathcal{E}, \theta, \mathcal{H}\| \vdash_{\text{fix}} (\pi, \theta, v) \leadsto \Phi'
$$

$$
\mathcal{E}, \mathcal{H} \vdash_{\text{fix}} \Phi \leadsto \bigcup \Phi'
$$

[Fix-Disj]

For all the execution tuples $(\pi, \theta, v)$, given $\mathcal{H}$, it is reduced to $\Phi'$. Their relation is captured by: $\|\mathcal{E}, \theta_{\text{his}}, \mathcal{H}\| \vdash_{\text{fix}} \Phi \leadsto \Phi'$, where $\theta_{\text{his}}$ is the history trace and initialized by $\epsilon$. The final result $\Phi_{\text{fix}}$ is a union set of all the $\Phi'$.

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 $\Phi'$ is achieved by computing the strongest post condition of $e_{\text{ret}}[v/x]$ from the starting state $((\pi, \theta_{\text{his}}, v))$.

$$
\begin{align*}
(\text{return } x \mapsto e_{\text{ret}}) & \in \mathcal{H} \\
\mathcal{E} \vdash (\pi, \theta_{\text{his}}, v) & e_{\text{ret}}[v/x] \Phi' \\
\|\mathcal{E}, \theta_{\text{his}}, \mathcal{H}\| & \vdash_{\text{fix}} (\pi, \heartsuit, v) \leadsto \Phi'
\end{align*}
$$

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

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

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

\[
\alpha \in \{l!\} \quad (\text{effect } A(x, \kappa) \mapsto e) \in H \quad (l = A(v)) \quad \|\mathcal{E}, \theta_{\text{his}} \cdot 1, H\| \vdash_{\text{fix}} (\pi, \alpha \cdot \theta, v) \leadsto \Phi' \quad \text{[Fix-Unfold-Handle]}
\]

### 6.4.3 Reasoning in the Handling Program.

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

\[
\forall (\pi, \theta, v) \in \Phi \quad \|\mathcal{E}, \theta, H\| \vdash_{\text{fix}} (\pi, D[v'/?], v) \leadsto \Phi' \quad \mathcal{E}, H, D \vdash_h \langle \Phi' \rangle \ e \langle \Phi'' \rangle \quad \text{[Handle-Resume]}
\]

\[
\Phi' = \{(\pi, \theta, v') \mid (\pi, \theta, v) \in \Phi\} \quad \text{[Handle-Value]}
\]

In \([\text{Handle-Resume}]\), all the placeholders \(l?\) shown in the continuation \(D\) can be finally instantiated by \(\kappa\)'s argument value, \(v'\). Possible loops are also captured...
in this step, when $D[v'/l?]$ produces the effects’ emissions which has already been handled. The final result $\Phi''$ 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 $D$ 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 $\Phi$, with the environment $E$ and handler $H$. $\Phi_{fix}$ is the updated version of $\Phi$, where all $\Phi$’s placeholders – which can be handled by $H$ – are handled as $H$ defines.

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

- when $\Phi$ is a set, $\Phi_{fix} = \{ \| E, e, H \| \vdash_{fix} (\pi, \theta, v) \rightsquigarrow \Phi' \mid (\pi, \theta, v) \in \Phi \}$; (1)
- when $\Phi = (\pi, \theta, v)$, $\alpha = \text{fst}(\theta)$, $\theta_{his}$ is the handled trace,
  if $\alpha = \heartsuit: ([x \mapsto v]): E \vdash (\pi, \theta_{his}, v) e_{ret}(\Phi')$ is valid, given $(\text{return } x \mapsto e_{ret}) \in H$; (2)
  if $\alpha \in \{ \text{ev, l!, l!} \}$ (l $\in H$) : $\| E, \theta_{his} \cdot \alpha, H \| \vdash_{fix} (\pi, D_{\alpha}(\theta), v) \rightsquigarrow \Phi'$ is valid; (3)
  if $\alpha \in \{ \text{l!} \}$ (l $\in H$) : $(x \mapsto v): E, H, D_{\alpha}(\theta) \vdash_h (\pi, \theta_{his} \cdot l, v) e\langle \Phi' \rangle$ is valid,

  given $(\text{effect } A(x, \kappa) \mapsto e) \in H$. (4)

**Proof.** See subsection C.2.1.

**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.

Formally, $\forall e, \forall E, \forall \Phi$ given $[e, E, \varphi] \xrightarrow{\cdot} [v, E', \varphi']$ and $E \vdash \langle \Phi \rangle e \langle \Phi' \rangle$, $E, \varphi \models \Phi$ then $E', \varphi' \models \Phi'$.

**Proof.** See subsection C.2.2.
6.5 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 $\Phi_1$ and $\Phi_2$, the TRS decides if the inclusion $\Phi_1 \sqsubseteq \Phi_2$ is valid. During the rewriting process, the inclusions are of the form $\Omega \vdash \Phi_1 \sqsubseteq \theta \cdot \Phi_2$, 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 $\Phi_1$ are legitimately allowed in the possible traces from the consequent $\Phi_2$. $\Omega$ is the proof context, i.e., a set of effect inclusion hypotheses, and $\theta$ 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 $\theta=\epsilon$.

6.5.0.1 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’ entailment checking is irrelevant to the returning values.

\[
\begin{align*}
[LHS-OR] & \quad \Omega \vdash (\pi, \theta) \sqsubseteq \Phi \quad \text{and} \quad \Omega \vdash \Phi \sqsubseteq \Phi' \\
\hline
\Omega \vdash (\pi, \theta, v) :: \Phi \sqsubseteq \Phi' \\
[RHS-OR] & \quad \Omega \vdash (\pi, \theta) \sqsubseteq (\pi', \theta') \quad \text{or} \quad (\pi, \theta) \sqsubseteq \Phi' \\
\hline
\Omega \vdash (\pi, \theta) \sqsubseteq (\pi', \theta', v') :: \Phi'
\end{align*}
\]

Definition 21 (ContEffs Inclusion). For effects $(\pi_1, \theta_1)$ and $(\pi_2, \theta_2)$,

\[
(\pi_1, \theta_1) \sqsubseteq (\pi_2, \theta_2) \iff \pi_1 \Rightarrow \pi_2 \quad \text{and} \quad (\forall \alpha \in \Sigma). \quad D_\alpha(\theta_1) \sqsubseteq D_\alpha(\theta_2)
\]

Next we provide the definitions and implementations of auxiliary functions\(^4\) \textit{Nullable}\((\delta)\), \textit{Infinitable}\((\kappa)\), \textit{First}\((\text{fst})\) and \textit{Derivative}\((D)\) respectively. Intuitively, the Nullable function $\delta(\Phi)$ returns a boolean value indicating whether $\theta$ contains the empty trace; the Infinitable function $\kappa(\theta)$ returns a boolean value indicating

\(^4\)The definitions are extended from [Ant95], to be able to deal with placeholders and infinite traces, proposed in this work.
whether \( \theta \) is possibly infinite; the First function \( \text{fst}(\theta) \) computes possible initial elements of \( \theta \); and the Derivative function \( D_\alpha(\theta) \) eliminates an event \( \alpha \) from the head of \( \theta \) and returns what remains.

**Definition 22** (Nullable). Given any sequence \( \theta \), we recursively define \( \delta(\theta) \):

\[
\begin{align*}
\delta(\epsilon) &= \delta(\theta^*) = \delta(\theta^\infty) = \text{true} \\
\delta(\theta_1 \cdot \theta_2) &= \delta(\theta_1) \land \delta(\theta_2) \\
\delta(\theta_1 \lor \theta_2) &= \delta(\theta_1) \lor \delta(\theta_2)
\end{align*}
\]

**Definition 23** (Infinitiable). Given any sequence \( \theta \), we recursively define \( \kappa(\theta) \):

\[
\kappa(\theta^\infty) = \kappa(\theta^\omega) = \text{true} \\
\kappa(\theta_1 \cdot \theta_2) = \kappa(\theta_1) \lor \kappa(\theta_2) \\
\kappa(\theta_1 \lor \theta_2) = \kappa(\theta_1) \lor \kappa(\theta_2)
\]

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

\[
\begin{align*}
\text{fst}(\bot) &= \text{fst}(\epsilon) = \emptyset \\
\text{fst}([ev]) &= \{ev\} \\
\text{fst}(\theta_1 \lor \theta_2) &= \text{fst}(\theta_1) \cup \text{fst}(\theta_2)
\end{align*}
\]

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

\[
\begin{align*}
D_\alpha(\bot) &= \bot \\
D_\alpha(\epsilon) &= \bot \\
D_\alpha(\theta_1 \lor \theta_2) &= D_\alpha(\theta_1) \lor D_\alpha(\theta_2) \\
D_\alpha(\theta^*) &= D_\alpha(\theta) \cdot \theta^*
\end{align*}
\]

\[
\begin{align*}
D_\alpha(\epsilon) &= \{ \epsilon \text{ if } \alpha \subseteq [ev] \} \\
D_\alpha(\theta) &= \{ \bot \text{ if } \alpha = [Q] \} \\
D_\alpha(\theta^\infty) &= D_\alpha(\theta) \cdot \theta^\infty
\end{align*}
\]

\[
\begin{align*}
D_\alpha(\theta_1 \cdot \theta_2) &= \left\{ \begin{array}{ll}
(\theta_1) \cdot \theta_2 & \text{if } \delta(\theta_1) = \text{true} \\
\theta_1 \cdot (\theta_2) & \text{if } \delta(\theta_1) = \text{false}
\end{array} \right.
\end{align*}
\]

\[
\begin{align*}
D_\alpha(\theta^\omega) &= D_\alpha(\theta) \cdot \theta^\omega
\end{align*}
\]

### 6.5.1 Rewriting Rules.

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

\[
\begin{align*}
\Omega \vdash (\pi_1, \bot) & \subseteq (\pi_2, \theta) \quad \text{[Bot-LHS]} \\
\theta & \neq \bot \quad \text{[Bot-RHS]}
\end{align*}
\]

\( ^5 \alpha \) could be a single label \( l \), a negated label \( \overline{l} \), a wildcard \( _\_ \), or a placeholder \( Q \).

\( ^6 \text{false for unmentioned constructs} \)

\( ^7 \text{false for unmentioned constructs} \)

\( ^8 [\theta] \) represents all the traces contained in \( \theta \).
CHAPTER 6. CONTINUATION BASED EFFECTS

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.

\[
\begin{align*}
\delta(\theta_1) \land \neg \delta(\theta_2) & \quad \text{[Dis-Nullable]} \\
\Omega \vdash (\pi_1, \theta_1) \nsubseteq (\pi_2, \theta_2) & \\
\varphi(\theta_1) \land \neg \varphi(\theta_2) & \quad \text{[Dis-Infinitable]} \\
\Omega \vdash (\pi_1, \theta_1) \nsubseteq (\pi_2, \theta_2)
\end{align*}
\]

3. Prove. We use the rule [Reoccur] to prove an inclusion when there exist inclusion hypotheses in the proof context \(\Omega\), 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.

\[
\begin{align*}
(\pi_1, \theta_1) \subseteq (\pi_3, \theta_3) \in \Omega \\
(\pi_3, \theta_3) \subseteq (\pi_4, \theta_4) \in \Omega \\
(\pi_4, \theta_4) \subseteq (\pi_2, \theta_2) \in \Omega \\
\Omega \vdash (\pi_1, \theta_1) \subseteq (\pi_2, \theta_2) \quad \text{[Reoccur]}
\end{align*}
\]

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 \(\Omega'\) by adding the current inclusion, as an inductive hypothesis, into the current proof context \(\Omega\). 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 \(\alpha\). The proof of the original inclusion succeeds if all the derivative inclusions succeed.

\[
F = \text{\(fst(\theta_1)\)} \\
\pi_1 \Rightarrow \pi_2 \\
\forall \alpha \in F. (\theta_1 \sqsubseteq \theta_2) \quad \Omega \vdash D_\alpha(\theta_1) \sqsubseteq D_\alpha(\theta_2) \\
\Omega \vdash (\pi_1, \theta_1) \sqsubseteq (\pi_2, \theta_2) \quad \text{[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 \(\text{TRUE}\) when proving \(\Phi_1 \sqsubseteq \Phi_2\), then \(\Phi_1 \sqsubseteq \Phi_2\) is valid.

*Proof.* See section C.4.
CHAPTER 6. CONTINUATION BASED EFFECTS

6.6 Implementation and Evaluation

To show the feasibility of our approach, we have prototyped our automated verification system using OCaml. The proof obligations generated by the verifier 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.

Table 6.1: Experimental Results for ContEffs.

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

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; Infer(ms), effects inference 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.
CHAPTER 6. CONTINUATION BASED EFFECTS

Discussion: Generally, inference 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 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.

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 $G$ ('globally') and $F$ ('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 $U$ ('until') operator and the $X$ ('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\footnote{Our implementation supports specifications written in LTL formulae, by providing a translator from LTL to ContEffs. The translation schema is taken from [LS07].}.

<table>
<thead>
<tr>
<th>$\Box l \equiv l^\infty$</th>
<th>$\Diamond l \equiv _\cdot l$</th>
<th>$l U j \equiv l^* \cdot j$</th>
<th>$l \rightarrow \Diamond j \equiv \neg l \vee _\cdot j$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X l \equiv _\cdot l$</td>
<td>$\Box \Diamond l \equiv (_\cdot l)^\infty$</td>
<td>$\Diamond \Box l \equiv _\cdot l^\infty$</td>
<td>$\Diamond l \vee \Diamond j \equiv _\cdot l \vee _\cdot j$</td>
</tr>
</tbody>
</table>

II. Encoding Exceptions. Exceptions are a special case of algebraic effects which never resume, and Figure 6.9 demonstrates how our framework soundly reasons about exceptions together with other kinds of effects. Here $\text{raise}()$ performs $\text{Exc}$
first, then does some other operations afterwards, represented by performing effect
Other.

```plaintext
effect Exc: unit
effect Other: unit

let raise () = perform Exc;
perform Other

let excHandler = match raise () with
| _ -> (* Abandoned *)
| effect Exc k -> ()

let excHandler
/*@ req _^* @*/
/*@ ens Exc @*/
= perform Exc;
| _ -> (*)

= match raise () with
| _ -> (*)

Figure 6.9: 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 raise() after handling Exc. The verified postcondition of excHandler matches how we would intuitively expect traditional exceptions to work\(^\text{10}\).

### 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 effects 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

\(^{10}\)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].

113
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.
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, Imp$^{a/s}$, C$^t$ and $\lambda_h$, varying in different programming domains: general effectful programs; mixed synchronous and 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.

### 7.1.1 Useful Links

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 findings 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 of 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-grind verification support for different complex control-flow mechanisms, possibly hard to model by existing techniques.
Bibliography


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-
BIBLIOGRAPHY


traces and coinduction", in *Automated Reasoning with Analytic Tableaux
and Related Methods - 24th International Conference, TABLEAUX 2015,
pp. 307–322. [Online]. Available: https://doi.org/10.1007/978-3-
319-24312-2\%5C_21.

and communication schemes synthesis of distributed control and com-
munication 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 verifi-
cation of shape, size and bag properties via user-defined predicates in
separation logic", *Sci. Comput. Program.*, vol. 77, no. 9, pp. 1006–1036,
07.004.

[Con+20] L. Convent, S. Lindley, C. McBride, and C. McLaughlin, "Doo bee
https://doi.org/10.1017/S0956796820000039.


[Daa17] L. Daan, "Type directed compilation of row-typed algebraic effects", in
*Proceedings of the 44th ACM SIGPLAN Symposium on Principles of

[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

[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,


BIBLIOGRAPHY


Appendix A

Appendix for \textit{ASyncEффs}

A.1 Operational Semantics Rules for the Basic Statements

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

\[
\begin{align*}
() & \xrightarrow{\emptyset,0} () \quad [\textit{Axiom-Nothing}] \\
\text{yield} & \xrightarrow{\emptyset,1} () \quad [\textit{Axiom-Yield}] \\
\text{emit} \$ & \xrightarrow{\{\$\},0} () \quad [\textit{Axiom-Emit}]
\end{align*}
\]

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.

\[
\begin{align*}
[\text{Seq-0}] & \quad p \xrightarrow{a,0} () \quad q \xrightarrow{f,k} q' \\
[\text{Seq-1}] & \quad p \xrightarrow{a,k} p' \quad (k \leq 1) \\
[\text{Seq-n}] & \quad p \xrightarrow{a,k} p' \quad (k > 1) \\
[\text{Loop}] & \quad p; \text{loop} \ x \rightarrow a,k \ x \rightarrow p'
\end{align*}
\]

Due to the static scope of signals, $\mathcal{E}$ may already contain the \$, therefore the notation $\mathcal{E}\backslash\$ 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 $p$. $\overline{S} \setminus E$ represents the output signals declared in the callee function.

\[
\begin{align*}
\text{[Decl]} & \quad p \xrightarrow{a,k} p' \\
\text{signal } S \text{ in } p & \xrightarrow{a,k} \text{ signal } S \text{ in } p'
\end{align*}
\]

\[
\begin{align*}
\text{[Call]} & \quad x \left( \overline{S} \right) \langle \text{req } \Phi_{\text{pre}} \text{ ens } \Phi_{\text{post}} \rangle \ p \in \mathcal{P} \\
& \xrightarrow{a,k} p'
\end{align*}
\]

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

\[
\begin{align*}
\text{(S} \rightarrow \text{present}) & \in E \quad \text{p} \xrightarrow{a,k} \text{p}' \\
\text{present } S & \text{ p q} \xrightarrow{a,k} \text{p}' \quad \text{[Present-1]} \quad \text{(S} \rightarrow \text{present}) & \notin E \quad q \xrightarrow{a,k} q' \\
\text{present } S & \text{ q} \xrightarrow{a,k} q' \quad \text{[Present-2]}
\end{align*}
\]

### A.2 Forward Rules for the Basic Statements

[\text{FV-Unit}] obtains the next program state by inheriting the current state. [\text{FV-Emit}] updates the current instant with $S$ pointing from $\text{undef}$ to $\text{present}$.

\[
E \vdash \langle h, c, k \rangle [\text{FV-Nothing}] \quad E \vdash \langle h, c, k \rangle \text{ emit } S \langle h, c + \{S\}, k \rangle [\text{FV-Emit}]
\]

[\text{FV-Yield}] archives the current instant to the history trace, then initializes a new current instant by an empty instant. [\text{FV-Local}] computes $p$’s effects by eliminating the existing signal $S$.

\[
\begin{align*}
E \vdash \langle h, c, k \rangle & \text{ yield } \langle h \cdot c, \{\}, k \rangle \\
E \setminus \langle h, c \setminus S, k \rangle & \text{ p } \langle H', C', K' \rangle \\
E \vdash \langle h, c, k \rangle & \text{ signal } S \text{ in } p \langle H', C', K' \rangle
\end{align*}
\]

[\text{FV-Present}] computes the effects of $p$ and $q$ by extending the current instant with $S$ being $\text{present}$ and $\text{absent}$. The final state is an union of the results.
\[ E \vdash \langle h, c\mathbf{+}\{S\}, k \rangle \ p \ \langle H_1, C_1, K_1 \rangle \quad E \vdash \langle h, c\mathbf{+}\{S\}, k \rangle \ q \ \langle H_2, C_2, K_2 \rangle \]

\[ E \vdash \langle h, c, k \rangle \ \text{present} \ S \ p \ q \ \langle H_1, C_1, K_1 \rangle \cup \langle H_2, C_2, K_2 \rangle \quad \text{[FV-Present]} \]

[FV-Seq] computes \( p \)'s effects. If the completion code \( K_1 \leq 1 \), i.e., there are no exceptions 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.

\[ E \vdash \langle h, c, k \rangle \ p \ \langle H_1, C_1, K_1 \rangle \quad E \vdash \langle H_1, C_1, K_1 \rangle \ q \ \langle H_2, C_2, K_2 \rangle \]
\[ \langle H', C', K' \rangle = \langle H_2, C_2, K_2 \rangle \quad (K_1 \leq 1) \]
\[ \langle H', C', K' \rangle = \langle H_1, C_1, K_1 \rangle \quad (K_1 > 1) \]

\[ E \vdash \langle h, c, k \rangle \ \text{seq} \ p \ q \ \langle H', C', K' \rangle \quad \text{[FV-Seq]} \]

We are of the opinion that the invariant computation for loops is orthogonal to the purpose of this work. Therefore we set [FV-Loop] unwind the loop body twice. If the completion code of the first run \( K_1 \leq 1 \), it forms a repeated trace \( h \cdot H_1 \cdot (H_2)^* \) with \( C_2 \) to be current instant. Otherwise, it simply exits the loop.

\[ E \vdash \langle \epsilon, c, k \rangle \ p \ \langle H_1, C_1, K_1 \rangle \quad E \vdash \langle \epsilon, C_1, K_1 \rangle \ p \ \langle H_2, C_2, K_2 \rangle \]
\[ \langle H', C', K' \rangle = \langle h \cdot H_1 \cdot (H_2)^*, C_2, K_2 \rangle \quad (K_1 \leq 1) \]
\[ \langle H', C', K' \rangle = \langle h \cdot H_1, C_1, K_1 \rangle \quad (K_1 > 1) \]

\[ E \vdash \langle h, c, k \rangle \ \text{loop} \ p \ q \ \langle H', C', K' \rangle \quad \text{[FV-Loop]} \]

[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 \( \Phi_{post} \) to the current effects state. Otherwise the verification fails.

\[ x(\tau S) \ \mathbf{req}\Phi_{pre} \ \mathbf{ens}\Phi_{post} \ p \in \mathcal{P} \quad h \cdot c \subseteq \Phi_{pre}[\overline{S}/S] \]
\[ \langle H_1, C_1, K' \rangle = \Phi_{post}[\overline{S}/S] \]
\[ E \vdash \langle h, c, k \rangle \ \text{call} \ x \ (\overline{S}) \ \langle H \cdot H_1, C_1, K' \rangle \quad \text{[FV-Call]} \]
A.3 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 is 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 line 3 and line 9 to $\Delta_1 \leftarrow [(\Phi_{\text{his}}, \{S\}, 0)]$ and $\phi \leftarrow [(\Phi_{\text{his}}, \{S\}, 0)]$ respectively.

**Algorithm 3:** Weak Suspend Interleaving

```plaintext
Input: S, (Φ, I, k)
Output: Program States, ∆

1: rec function $\mathcal{S}_{\text{Interleave}}^{\text{Suspend}}(S, I, k)(\Phi)$
2: if $\text{fst}(\Phi)=\emptyset$ then
3: $\Delta_1 \leftarrow [(\epsilon, I+\{S\}, k)]$  ▷ Notion + unions two instants
4: $\Delta_2 \leftarrow [(I+\{S\}, \{\}, k)]$
5: ▷ In strong suspend, line 4 should be: $\Delta_2 \leftarrow [(\{S\}, \{\}, k)]$
6: return $(\Delta_1 \cup \Delta_2)$  ▷ Notion $\cup$ unions two program states
7: else
8: $\Delta \leftarrow []$
9: foreach $f \in \text{fst}(\Phi)$ do
10: $\Delta' \leftarrow \mathcal{S}_{\text{Interleave}}^{\text{Suspend}}(S, I, k)(D_f(\Phi))$
11: $\Delta_1 \leftarrow (f+\{S\}) \cdot \Delta'$
12: $\Delta_2 \leftarrow (f+\{S\}) \cdot \{\} \cdot \Delta'$
13: ▷ In strong suspend, line 12 should be: $\Delta_2 \leftarrow \{S\} \cdot \{\} \cdot \Delta'$
14: $\Delta \leftarrow \Delta \cup \Delta_1 \cup \Delta_2$
15: return $\Delta$
```

**Lemma 2** (Soundness of Weak Abort Interleaving).

For function $\mathcal{S}_{\text{Interleave}}^{\text{Abort}}$, $\forall S, \Phi, I, k, \Phi_{\text{his}},$

if $\Phi=\epsilon$, then $\Delta = [(\Phi_{\text{his}}, I+\{S\}, 0); (\Phi_{\text{his}}, I+\{S\}, k)]$
else $\Delta = \bigcup_{v}^{\lvert F \rvert} (\Phi_{his}, f+\{S\}, 0) :: \ abort_{\text{Interleave}}(D_f(\Phi), \Phi_{his} \cdot (f+\{S\}))$ where $F$=fst($\Phi$).

\textbf{Proof.} By induction on $\Phi$, with Algorithm 1. \hfill \Box

\textbf{Lemma 3} (Soundness of Weak Suspend Interleaving).

For function $\ abort_{\text{Interleave}}$, \forall \Phi, \ I, \ k,$

\text{if} $\Phi$=$\epsilon$, then $\Delta = [(\epsilon, I+\{S\}, k); (I+\{S\}, \{\}, k)]$

\text{else} $\Delta = \bigcup_{v}^{\lvert F \rvert} ((f+\{S\}) \lor (f+\{S\}) \lor \{\}) \cdot \Delta'$, \text{where $F$=fst($\Phi$) and $\Delta'$=abort_{\text{Interleave}}(D_f(\Phi))}.

\textbf{Proof.} By induction on $\Phi$, with Algorithm 3. \hfill \Box

\section*{A.4 Parallel Merge Algorithm}

The parallel merging rules are in the form:

$\vdash_{pm} \langle H_1, C_1, K_1 \rangle || \langle H_2, C_2, K_2 \rangle \rightsquigarrow \langle H', C', K' \rangle$.

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 $[PM\text{-}\text{Union}]$ obtains $\langle H', C', K' \rangle$ by combining the parallel merged states of their cartesian products.

\[\forall (h_1, c_1, k_1) \in \langle H_1, C_1, K_1 \rangle \quad \forall (h_2, c_2, k_2) \in \langle H_2, C_2, K_2 \rangle \]

\[\langle H', C', K' \rangle = \bigcup (\vdash_{pm} \langle h_1, c_1, k_1 \rangle || \langle h_2, c_2, k_2 \rangle \rightsquigarrow \langle h', c', k' \rangle) \]

$[PM\text{-}\text{Unfold}]$

\[F_1 = \text{fst}(h_1) \quad F_2 = \text{fst}(h_2) \quad \forall f_1 \in F_1, \forall f_2 \in F_2, \ I = f_1 \cup f_2, \]

\[\text{der}_1 = D_f(h_1) \quad \text{der}_2 = D_f(h_2) \quad \langle H', C', K' \rangle = \bigcup (\vdash_{pm} \langle \text{der}_1, c_1, k_1 \rangle || \langle \text{der}_2, c_2, k_2 \rangle) \]

$[PM\text{-}\text{Unfold}]$ applies to deductive steps, which deploys auxiliary functions, $\text{fst}(\theta)$ and $D_f(\theta)$, to compute the firsts instant and derivatives of an effect, cf. subsection 4.6.1. 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 rules deal with base cases and terminate the merging process.
[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.

\[
\begin{align*}
[PM-EqLen] & \quad c' = c_1 \cup c_2 \quad k' = \max(k_1, k_2) \\
& \quad \vdash_{pm} \langle e, c_1, k_1 \rangle \parallel \langle e, c_2, k_2 \rangle \leadsto \langle e, c', k' \rangle \\
[PM-Cut] & \quad k_1 > 1 \quad c' = c_1 \cup \text{fst}(h_2) \\
& \quad \vdash_{pm} \langle e, c_1, k_1 \rangle \parallel \langle h_2, c_2, k_2 \rangle \leadsto \langle e, c', k_1 \rangle \\
[PM-Absorb] & \quad k_1 \leq 1 \quad h' = c_1 \parallel h_2 \\
& \quad \vdash_{pm} \langle e, c_1, k_1 \rangle \parallel \langle h_2, c_2, k_2 \rangle \leadsto \langle h', c_2, k_2 \rangle
\end{align*}
\]

To help with the understanding, concrete examples are:

- \(\langle \{A\} \cdot \{B\}, \{C\}, 0 \rangle \parallel \langle \{X\} \cdot \{Y\}, \{Z\}, 0 \rangle \leadsto \langle \{A, X\} \cdot \{B, Y\}, \{C, Z\}, 0 \rangle\);
- \(\langle \{A\}, \{C\}, 2 \rangle \parallel \langle \{X\} \cdot \{Y\}, \{Z\}, 0 \rangle \leadsto \langle \{A, X\}, \{C, Y\}, 2 \rangle\); and
- \(\langle \{A\}, \{C\}, 0 \rangle \parallel \langle \{X\} \cdot \{Y\}, \{Z\}, 2 \rangle \leadsto \langle \{A, X\} \cdot \{C, Y\}, \{Z\}, 2 \rangle\).

### A.5 Soundness of the Forward 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.

Formally, \(\forall e, \forall \mathcal{E}, \forall \varphi, \forall \Phi\) given \([e, \mathcal{E}, \varphi] \xrightarrow{\cdot} [v, \mathcal{E}', \varphi']\) and \(\mathcal{E} \vdash \langle \Phi \rangle\) \(e\) \(\langle \Phi' \rangle\),

\[
\text{if } \mathcal{E}, \varphi \models \Phi \text{ then } \mathcal{E}', \varphi' \models \Phi'.
\]

**Proof.** By induction on the structure of \(p\):

1. **Emit:** \(\mathcal{E} \vdash \langle h, c, k \rangle\) \(\text{emit } S\) \(\langle h, c + S, k \rangle\), \(\varphi \models h\) and \(\text{emit } S\) \(\langle S, 0 \rangle\), \(\langle h, c + S, k \rangle\), \(\varphi \models h\). \(\text{implies } \varphi \gg [S] \models h \cdot (c + S)\), is proved.

2. **Yield:** \(\mathcal{E} \vdash \langle h, c, k \rangle\) \(\text{Yield } \langle h \cdot c, \{\}\rangle, \varphi \models h\) and \(\text{Yield } \langle \{\}, 0 \rangle\), \(\langle \{\}, 0 \rangle\), \(\varphi \models h \cdot c\). \(\text{implies } \varphi \gg [\{\}] \gg [\{\}] \models h \cdot c \cdot \{\}\), is proved.
APPENDIX A. APPENDIX FOR ASYNEFFS

3. Present: $\mathcal{E} \vdash \langle h, c, k \rangle$ present $S$ $p$ $q$ $\langle H_1, C_1, K_1 \rangle \cup \langle H_2, C_2, K_2 \rangle$, $\varphi \models H$, where $\mathcal{E} \vdash \langle h, c+\{S\}, k \rangle$ $p$ $\langle H_1, C_1, K_1 \rangle$ and $\mathcal{E} \vdash \langle h, c+\{S\}, k \rangle$ $q$ $\langle H_2, C_2, K_2 \rangle$.

- When $(S) \in c$, present $S$ $p$ $q$ $\leadsto$ $p$, the inclusion is proved by inductive hypothesis.
- When $(S) \notin c$, present $S$ $p$ $q$ $\leadsto$ $q$, the inclusion is proved by inductive hypothesis.

4. Sequence: $\mathcal{E} \vdash \langle h, c, k \rangle$ 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 $p$ $q$ $\frac{e_0.0}{\varepsilon} \rightarrow \cdots \frac{e_n.0}{\varepsilon_n} \rightarrow q$ $\frac{e_n \cup f_0.0}{\varepsilon_n} \rightarrow \cdots \frac{f_m.0}{\varepsilon_n} \rightarrow q_n$ $\frac{0.k}{\varepsilon_n} (\varepsilon_n')$; therefore $\varphi \leftrightarrow [e_0; \cdots; (e_n \cup f_0); \cdots; f_m] \models H_2.C_2$.
- When $K_1 = 1$, seq $p$ $q$ $\frac{e_0.0}{\varepsilon} \rightarrow \cdots \frac{e_n.0}{\varepsilon_n} \rightarrow p_n$ $\frac{0.1}{\varepsilon_n} \rightarrow q$ $\frac{f_0.0}{\varepsilon_n} \rightarrow \cdots \frac{f_m.0}{\varepsilon_n} \rightarrow q_n$ $\frac{0.k}{\varepsilon_n} (\varepsilon_n')$; therefore $\varphi \leftrightarrow [e_0; \cdots; e_n; f_0; \cdots; f_n] \models H_2.C_2$.
- When $K_1 > 1$, seq $p$ $q$ $\frac{e_0.0}{\varepsilon} \rightarrow \cdots \frac{e_n.0}{\varepsilon_n} \rightarrow p_n$ $\frac{0.1}{\varepsilon_n} \rightarrow \cdots \frac{f_m.0}{\varepsilon_n} \rightarrow q_n$ $\frac{0.k}{\varepsilon_n} (\varepsilon_n')$, therefore $\varphi \leftrightarrow [e_0; \cdots; e_n] \models H_1.C_1$.

5. Exit: $\mathcal{E} \vdash \langle h, c, k \rangle$ exit $d$ $\langle h, c, d + 2 \rangle$, $\varphi \models H$ and exit $d$ $\frac{\{d+2\}}{\varepsilon} (\varepsilon_n')$; implying $\varphi \leftrightarrow [\{\}] \models h \cdot c$, is proved.

6. Trap: $\mathcal{E} \vdash \langle h, c, k \rangle$ trap $p$ $\langle h \cdot \Delta \rangle$ and $\varphi \models H$, and $\mathcal{E} \vdash \langle c, k, p \rangle(H, C, K)$, where $\langle \Delta \rangle = \langle H, C, K \rangle$ when $K \leq 1$; $\langle \Delta \rangle = \langle H, C, 0 \rangle$ when $K = 2$; $\langle \Delta \rangle = \langle H, C, K-1 \rangle$ when $K > 2$

- When $K \leq 1$: trap $p$ $\leadsto$ $p$, the entailment is proved by inductive hypothesis.
- When $K = 2$: by [Trap-2], trap $p$ $\frac{\{0\}}{\varepsilon} (\varepsilon_n')$, implying $\varphi \leftrightarrow [\{\}] \models H \cdot C$, is proved.
- When $K > 2$: by [Trap-3], trap $p$ $\frac{\{K-1\}}{\varepsilon} (\varepsilon_n')$, implying $\varphi \leftrightarrow [\{\}] \models H \cdot C$, is proved.

7. Await: $\mathcal{E} \vdash \langle h, c, k \rangle$ await $S$ $\langle h \cdot (c||S?)\rangle$, $\{\} \rangle$ and $\varphi \models H$;

- When $\varepsilon \in E$: by [Await-1] await $S$ $\frac{\{1\}}{\varepsilon} (\varepsilon_n')$, $\varphi \leftrightarrow [\{\}] \models h \cdot c$, is proved.
- When $(S) \notin E$: by by [Await-2] let $\varphi' \models S?$, $\varphi \leftrightarrow [\{\}] \vdash \varphi' \models h \cdot c \cdot S?$, is proved.

8. Async: $\mathcal{E} \vdash \langle h, c, k \rangle$ async $S$ $p$ $q$ $\langle H', C', K' \rangle$ and $\varphi \models H$ where

$\mathcal{E} \vdash \langle h, c, k \rangle$ $(p; emit \ S || q \langle H', C', K' \rangle)$ and async $S$ $p$ $q$ $\leadsto$ $(p; emit \ S || q)$, 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 $\mathcal{E} \vdash \langle e, c, k \rangle p \langle H_1, C_1, K_1 \rangle$

$\mathcal{E} \vdash \langle e, c, k \rangle$ $q \langle H_2, C_2, K_2 \rangle$ and $\vdash pm \langle H_1, C_1, K_1 \rangle |\langle H_2, C_2, K_2 \rangle \leadsto \langle \Delta \rangle$.

-When $p$ and $q$ exit at the same instant: the entailment is proved by [PM-Unfold]
APPENDIX A. APPENDIX FOR ASYNCEFFS

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:** \(E \vdash ⟨h, c, k⟩ \text{ abort } S ⟨h \cdot ∆⟩\) and \(\varphi \models H\), where \(E \vdash ⟨ϵ, c, k⟩p⟨H, C, K⟩\) and \(⟨∆⟩≡_\text{Interleave}^{\text{Abort}(S, C, K)}(H, ϵ)\).

   By semantics rules \([Abort-1]\) and \([Abort-2]\); and Lemma 2.

11. **Suspend:** \(E \vdash ⟨h, c, k⟩ \text{ suspend } p S ⟨h \cdot ∆⟩\) and \(\varphi \models H\), where

   \(E \vdash ⟨ϵ, c, k⟩ p ⟨H, C, K⟩\) and \(⟨∆⟩≡_\text{Interleave}^{\text{Suspend}(S, C, K)}(H)\).

   By semantics rules \([Suspend-1]\) and \([Suspend-2]\); and Lemma 3.

\(\Box\)

### A.6 Termination Proof

**Proof.** Let \(\text{Set}[\mathcal{I}]\) 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., \(S, H \in \text{Set}[\mathcal{I}]\).

Consider the following partial ordering \(\succ\) on pairs \(⟨S, H⟩\):

\[⟨S_1, H_1⟩ \succ ⟨S_2, H_2⟩ \iff |H_1| < |H_2| \lor (|H_1| = |H_2| \land |S_1| > |S_2|).\]

where \(|X|\) stands for the cardinality of a set \(X\). Let \(⇒\) donate the rewrite relation, then \(⇒^*\) denotes its reflexive transitive closure. For any given \(S_0, H_0\), this ordering is well founded on the set of pairs \(\{⟨S, H⟩|⟨S_0, H_0⟩ ⇒^* ⟨S, H⟩\}\), 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 4.6.2 transform current pairs \(⟨S, H⟩\) to new pairs \(⟨S', H'⟩\). And each rule either increases \(|H|\) (Unfolding) or, otherwise, reduces \(|S|\) (Axiom, Disprove, Prove), therefore the system is terminating.

\(\Box\)
A.7 Soundness Proof

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

1. Axiom Rules:

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

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

2. Disprove Rules:

\[ \delta(es_1) \land \neg\delta(es_2) \quad [\text{Disprove}] \]
\[ \Gamma \vdash es_1 \not\sqsubseteq es_2 \quad [\text{Prove}] \]

- It’s straightforward to prove soundness of the rule [Disprove], Given that \( \theta_1 \) is nullable, while \( \theta_2 \) is not nullable, thus clearly the antecedent contains more traces than the consequent. Therefore, these entailments are evidently invalid.

3. Prove Rules:

\[ (es_1 \sqsubseteq es_3) \in \Gamma \quad (es_3 \sqsubseteq es_4) \in \Gamma \quad (es_4 \sqsubseteq es_2) \in \Gamma \]
\[ \Gamma \vdash es_1 \sqsubseteq es_2 \quad [\text{Reoccur}] \]

- For the rule [Prove], we consider an arbitrary model, \( \varphi \) such that: \( \varphi \models \theta_1 \). Given the side conditions from the promises, we get \( \varphi \models \theta_1 \). When the \( \text{fst} \) set of \( \theta_1 \) is empty, \( \theta_1 \) is possible \( \bot \) or \( \epsilon \); and \( \theta_2 \) is nullable. For both cases, the inclusion is proved.

- For the rule [Reoccur], we consider an arbitrary model, \( \varphi \) such that: \( \varphi \models \theta_1 \). Given the promises that \( \theta_1 \sqsubseteq \theta_3 \), we get \( \varphi \models \theta_3 \); Given the promise 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:**

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

- For the rule [Unfold], we consider an arbitrary model, \( \varphi_1 \) and \( \varphi_2 \) 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 7, since for all \( I \), \( D_I(\theta_1) \sqsubseteq D_I(\theta_2) \), the conclusion is proved.

All the inference rules used in the TRS are sound, therefore the TRS is sound. \( \square \)
Appendix B

Appendix for *TimEfs*

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 \(\alpha\).

\[
(\mathcal{E}, v) \xrightarrow{\tau} (\mathcal{E}, v)[v] \quad (\mathcal{E}, \alpha) \xrightarrow{\tau} (\mathcal{E}[\alpha], ()) [assign]
\]

\[
(\mathcal{E}, \text{event}[A(v, \alpha^*)]) \xrightarrow{A(v)} (\mathcal{E}[\alpha^*], ()) [ev]
\]

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.

\[
\begin{align*}
[cond_1] \quad & \quad \mathcal{E}(v) = \text{True} \\
[cond_2] \quad & \quad \mathcal{E}(v) = \text{False} \\
[call] \quad & \quad \text{null}\{e\} \in \mathcal{P} \quad (\mathcal{E}, e[v'/x^*]) \xrightarrow{t} (\mathcal{E}', e') \\
\end{align*}
\]

\[
(\mathcal{E}, \text{if } v \ e_1 \ e_2) \xrightarrow{t} (\mathcal{E}, e_1) \quad (\mathcal{E}, \text{if } v \ e_1 \ e_2) \xrightarrow{t} (\mathcal{E}, e_2) \\
(\mathcal{E}, mn(v^*)) \xrightarrow{t} (\mathcal{E}', e')
\]

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.
B.2 The Complete Forward Rules

Rule \([FV\text{-}Value]\) obtains the next state by inheriting the current state. Rule \([FV\text{-}Event]\) concatenates the event to the current state and update the environment for the subsequent statements.

\[
\begin{align*}
(\mathcal{E}, e_1) & \xrightarrow{} (\mathcal{E}', e'_1) \quad [\text{seq}] \\
(\mathcal{E}, e_1; e_2) & \xrightarrow{} (\mathcal{E}', e'_1; e_2) \\
(\mathcal{E}, v; e_2) & \xrightarrow{} (\mathcal{E}, e_2) \quad [\text{seq}] \\
(\mathcal{E}, e_1 || e_2) & \xrightarrow{} (\mathcal{E}', e'_1 || e'_2) \quad [\text{par}]
\end{align*}
\]

Rule \([FV\text{-}Call]\) first checks whether the instantiated precondition of callee, \(\Phi_{\text{pre}}[v^*/x^*]\), 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.

\[
\begin{align*}
\text{mn} \ x^* \{\text{req} \ \Phi_{\text{pre}} \ \text{ens} \ \Phi_{\text{post}}\} \ {\{e\}} & \in \mathcal{P} \\
\mathcal{E} & \vdash \langle \pi, \theta \rangle \subseteq \Phi_{\text{pre}}[v^*/x^*] \\
\Phi_f & = \langle \pi, \theta \rangle \cdot \Phi_{\text{post}}[v^*/x^*] \\
\mathcal{E} & \vdash \langle \pi, \theta \rangle \ \text{mn}(v^*) \ \langle \Phi_f \rangle \quad [\text{FV\text{-}Call}]
\end{align*}
\]

Rule \([FV\text{-}Cond\text{-}Local]\) 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 \([FV\text{-}Cond\text{-}Global]\) 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.

\[
[FV\text{-Cond-Local}]
\begin{align*}
\mathcal{E} \vdash \{\pi \land v, \theta\} & \quad e_1 \{\Pi_1, \Theta_1\} \\
\mathcal{E} \vdash \{\pi \land -v, \theta\} & \quad e_2 \{\Pi_2, \Theta_2\} \\
\mathcal{E} \vdash (\pi, e) & \quad \text{if } v \text{ then } e_1 \text{ else } e_2 \{\Pi_1, \Theta_1\} \cup \{\Pi_2, \Theta_2\}
\end{align*}
\]

\[
[FV\text{-Cond-Global}]
\begin{align*}
\mathcal{E} \vdash \{\pi, \theta\} & \quad e_1 \{\Pi_1, \Theta_1\} \\
\mathcal{E} \vdash \{\pi, \theta\} & \quad e_2 \{\Pi_2, \Theta_2\} \\
\mathcal{E} \vdash (\pi, e) & \quad \text{if } v \text{ then } e_1 \text{ else } e_2 \{\Pi_1, \Theta_1\} \cup \{\Pi_2, \theta \cdot \tau(v=\text{False}) \cdot \Theta_2\}
\end{align*}
\]

\[
[FV\text{-Meth}] \text{ initializes the state using the declared precondition, accumulates the effects from the method body, and checks the inclusion between the final state } \langle \Pi, \Theta \rangle \text{ and the concatenation of the pre- and postcondition}^1. \quad [FV\text{-Guard}] \text{ computes the effects of } e \text{ and concatenates } (v=\text{True})? \text{ before } e \text{'s effects.}
\]

\[
[FV\text{-Meth}]
\begin{align*}
\vdash \Phi_{\text{pre}} & \subseteq \Phi_{\text{post}} \\
\mathcal{E} \vdash mn x^* \{\text{req } \Phi_{\text{pre}} \text{ ens } \Phi_{\text{post}}\} \{e\} \\
\mathcal{E} \vdash \{\pi, e\} & \subseteq \{\Pi, \Theta\}
\end{align*}
\]

\[
[FV\text{-Guard}]
\begin{align*}
\vdash \{\pi, \theta\} & \subseteq \{\Pi, \theta\} \cdot (v=\text{True})? \\
\mathcal{E} \vdash \{\pi, \theta\} & \subseteq \{\Pi, \theta\} \cdot (v=\text{False}) \cdot \Theta
\end{align*}
\]

\[
[FV\text{-Seq}] \text{ computes } \{\Pi_1, \Theta_1\} \text{ from } e_1, \text{ then further gets } \{\Pi_2, \Theta_2\} \text{ by continuously computing the behaviors of } e_2, \text{ to be the final state.} \quad [FV\text{-Par}] \text{ computes behaviors for } e_1 \text{ and } e_2 \text{ independently, then parallel merges the effects.}
\]

\[
[FV\text{-Seq}]
\begin{align*}
\mathcal{E} \vdash \{\Pi_1, \Theta_1\} & \quad e_1 \{\Pi_1, \Theta_1\} \\
\mathcal{E} \vdash \{\Pi_2, \Theta_2\} & \quad e_2 \{\Pi_2, \Theta_2\} \\
\mathcal{E} \vdash (\pi, e_1||e_2 \{\Pi_1 \land \Pi_2, \Theta_1||\Theta_2\})
\end{align*}
\]

\[
[FV\text{-Par}]
\begin{align*}
\mathcal{E} \vdash (\pi, \theta) & \quad e_1 \{\Pi_1, \Theta_1\} \\
\mathcal{E} \vdash (\pi, \theta) & \quad e_2 \{\Pi_2, \Theta_2\} \\
\mathcal{E} \vdash (\pi, \theta) & \quad e_1||e_2 \{\Pi_1 \land \Pi_2, \Theta_1||\Theta_2\}
\end{align*}
\]

\section*{B.3 Soundness of the Forward Rules}

Given any system configuration \(\zeta=(\mathcal{E}, e)\), by applying the operational semantics rules, if \((\mathcal{E}, e) \rightarrow^n (\mathcal{E}', v)\) has execution time \(d\) and produces event sequence \(\varphi\); and for

\[^1\text{Note that for succinctness, the user-provided } \Phi_{\text{post}} \text{ only denotes the extension of the effects from executing the method body.}\]
APPENDIX B. APPENDIX FOR TIMEFFS

any history effect \(\pi \land \theta\), such that \(d_1, \mathcal{E}, \varphi_1 \models (\pi \land \theta)\), and the forward verifier reasons
\(\mathcal{E} \vdash (\pi, \theta) e \langle \Pi, \Theta \rangle\), then \(\exists (\pi', \theta') \in \langle \Pi, \Theta \rangle\) such that \((d_1+d), \mathcal{E}', (\varphi_1++ \varphi) \models (\pi' \land \theta')\).

Proof. By induction on the structure of \(e\):

1. Value: \(\langle \mathcal{E}, v \rangle \xrightarrow{\tau} \langle \mathcal{E}, v \rangle [v]\)
When \((\langle \mathcal{E}, v \rangle \rightarrow (\mathcal{E}, v))\), it takes 0 time and produces an empty sequence \([]\). By rule [FV-Value], \(\mathcal{E} \vdash \{\pi, \theta\} \langle \pi, \theta \rangle \cdot v \{\pi, \theta\}\), then the post effect is the witness that
\((d_1+0), \mathcal{E}, (\varphi_1++[\pi]) \models \pi \land \theta\) is valid.

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

3. Guard:
\[
\frac{\mathcal{E} \models (v=true)}{(\mathcal{E}, [v]e) \xrightarrow{gu_1} (\mathcal{E}, e)} \quad \frac{\mathcal{E} \not\models (v=true)}{(\mathcal{E}, [v]e) \xrightarrow{gu_2} (\mathcal{E}, [v]e)}
\]

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

4. Delay:
\[
\frac{d \leq v}{(\mathcal{E}, \text{delay}[v]) \xrightarrow{delay_1} (\mathcal{E}, \text{delay}[v-d])} \quad \frac{(\mathcal{E}, \text{delay}[0]) \xrightarrow{delay_2} (\mathcal{E}, (\emptyset))}{(\mathcal{E}, \text{delay}[v]) \xrightarrow{delay} (\mathcal{E}, (\emptyset))}
\]

When \((\mathcal{E}, \text{delay}[v]) \rightarrow (\mathcal{E}, (\emptyset))\), by applying rules [delay_1], [delay_2], it produces
an empty sequence \([],\) and takes time \(\mathcal{E}(v)\). By [FV-Delay], \(\mathcal{E} \vdash \langle \pi, \theta \rangle \text{ delay}[v] \langle \pi \land (t=d), \theta \cdot e\#t\rangle\). Then the post effect \(\pi \land (t=d) \land \theta \cdot e\#t\) is the witness that
\((d_1+\mathcal{E}(v), \mathcal{E}, (\varphi_1++[])) \models \pi \land (t=v) \land \theta \cdot e\#t\) is valid.
5. Timeout:

\[
\begin{align*}
&\quad (E, e_1) \xrightarrow{A} (E', e'_1) \\
&\quad (E, e_1 \text{ timeout}[v] e_2) \xrightarrow{A} (E', e'_1) \\
&\quad (E, e_1 \text{ timeout}[v] e_2) \xrightarrow{[t_{o1}]} (E, e_1 \text{ timeout}[v] e_2) \xrightarrow{A} (E', e'_1) \\
&\quad (E, e_1 \text{ timeout}[v] e_2) \xrightarrow{[t_{o2}]} (E', e'_1) \\
\end{align*}
\]

\[
\begin{align*}
&\quad (E, e_1) \xrightarrow{d} (E', e'_1) \\
&\quad (E, e_1 \text{ timeout}[v] e_2) \xrightarrow{(d \leq v)} (E, e'_1 \text{ timeout}[v-d] e_2) \\
&\quad (E, e_1 \text{ timeout}[v] e_2) \xrightarrow{[t_{o3}]} (E, e_1 \text{ timeout}[0] e_2) \xrightarrow{A} (E, e'_2) \\
\end{align*}
\]

When \((E, e_1 \text{ timeout}[v] e_2) \rightarrow (E', v')\), there are two possibilities:

- \(e_1\) started before time bound \(E(v)\): by applying rules \([t_{o2}],[t_{o3}]\) and \([t_{o1}]\), it produces the concrete sequence \([A; \, \text{tl}(\varphi(e_1))]\), and \(A\) takes \(t_1\) time-units, which is less than \(E(v)\). By [FV-Timeout], \(E \vdash \langle \pi, \theta \rangle e_1 \text{ timeout}[v] e_2 \langle \Pi_1 \land t_1 < v, \theta \cdot (\text{hd}(\Omega_1) \# t_1) \cdot \text{tl}(\Omega_1) \rangle\) where \(\varepsilon = \{\pi, \epsilon\} e_1\{\Pi_1, \Omega_1\}\). Then the post effect is the witness such that \((d_1 + t_1), E', \langle \varphi_1, \epsilon [A; \text{tl}(\varphi(e_1))] \rangle \models \Pi_1 \land t_1 < v \land \theta \cdot (\text{hd}(\Omega_1) \# t_1) \cdot \text{tl}(\Omega_1)\). \(e_1\) never started, by applying rules \([t_{o2}]\), it takes time \(d\) and produces the concrete sequence \([\varphi(e_2)]\). By [FV-Timeout], \(E \vdash \langle \pi, \theta \rangle e_1 \text{ timeout}[v] e_2 \langle \Pi_2 \land t_2 = v, \theta \cdot (\epsilon \# t_2) \cdot \Omega_2 \rangle\) where \(\varepsilon = \{\pi, \epsilon\} e_2\{\Pi_2, \Omega_2\}\). Then the post effect is the witness such that \((d_1 + d), E', \langle \varphi_1, \epsilon [\varphi(e_2)] \rangle \models \Pi_2 \land t_2 = v \land \theta \cdot (\epsilon \# t_2) \cdot \Omega_2\) is valid.

6. Deadline:

\[
\begin{align*}
&\quad (E, e) \xrightarrow{A/\tau} (E', e') \\
&\quad (E, \text{deadline}[v] e) \xrightarrow{A/\tau} (E', \text{deadline}[v] e') \\
&\quad (E, \text{deadline}[v] e) \xrightarrow{[d_{dl1}]} (E, e) \xrightarrow{\tau} (E', v) \xrightarrow{[d_{dl2}]} (E', \text{deadline}[v] e) \xrightarrow{\tau} (E', v) \\
&\quad (E, e) \xrightarrow{[d_{dl3}]} (E, \text{deadline}[v] e) \xrightarrow{d} (E, \text{deadline}[v-d] e') \\
&\quad (E, \text{deadline}[v] e) \xrightarrow{[d_{dl3}]} (E, \text{deadline}[v-d] e') \\
\end{align*}
\]

When \((E, \text{deadline}[v] e) \rightarrow (E', v')\), by applying rules \([d_{dl1}],[d_{dl2}]\) and \([d_{dl3}]\), it produces the concrete sequence \([\varphi(e)]\), and it takes \(d\) time-units which is less than \(E(v)\). By [FV-Deadline], \(E \vdash \langle \pi, \theta \rangle \text{deadline}[v] e \langle \Pi_1 \land (t \leq v), \theta \cdot (\Omega_1 \# t) \rangle\) where \(\varepsilon = \{\pi, \epsilon\} e \{\Pi_1, \Omega_1\}\). Then the post effect is the witness such that \((d_1 + d), E', \langle \varphi_1, \epsilon [\varphi(e)] \rangle \models \Pi_1 \land (t \leq v) \land \theta \cdot (\Omega_1 \# t)\) is valid.
7. Interrupt:

\[
\begin{align*}
(E, e_1) &\xrightarrow{A/\tau} (E', e_1') \\
(E, e_1 \text{ interrupt}[v] e_2) &\xrightarrow{A/\tau} (E', e_1' \text{ interrupt}[v] e_2) \\
(E, e_1 \text{ interrupt}[v] e_2) &\xrightarrow{\rightarrow} (E', v) \\
(E, e_1 \text{ interrupt}[0] e_2) &\xrightarrow{\rightarrow} (E, e_2) \\
\end{align*}
\]

When \((E, e_1 \text{ interrupt}[v] e_2)\xrightarrow{\rightarrow} (E', v')\), by applying rules \([\text{int}_1]\), \([\text{int}_2]\) and \([\text{int}_3]\), 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, \(\Delta = \pi \land (t<v) \land \theta \cdot \text{hd}(\varphi(e_1))#t\). By \([FV-\text{Interrupt}]\), \(E \vdash \langle \pi, \theta \rangle e_1 \text{ interrupt}[v] e_2 \langle \Pi', \theta \cdot \Theta' \rangle\) where \(E \vdash \{\Delta\} e_2 \{\Pi', \Theta'\}\). Then the post effect is the witness such that \((d_1+t+d_2), E', (\varphi_1++[\text{hd}(\varphi(e_1))]+[\varphi(e_2)]) \models \pi \land (t<v) \land \theta \cdot \text{hd}(\varphi(e_1))#t \cdot \Theta'\) is valid. Similar proofs for other possibilities.

8. Conditional:

\[
\begin{align*}
\mathcal{E}(v) &= \text{True} & \mathcal{E}(v) &= \text{False} \\
(E, \text{if } v e_1 e_2) &\xrightarrow{\rightarrow} (E, e_1) & (E, \text{if } v e_1 e_2) &\xrightarrow{\rightarrow} (E, e_2) \\
[\text{cond}_1] & & [\text{cond}_2]
\end{align*}
\]

When \((E, \text{if } v e_1 e_2)\xrightarrow{\rightarrow} (E', v')\), there are two possibilities:

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

- when \(\mathcal{E}(v) = \text{False}\) it takes \(d_{e_2}\) time units and produces sequence \(\varphi(e_2)\). By \([FV-\text{Cond-Local}]\), \(E \vdash \langle \pi, \theta \rangle \text{ if } v \text{ then } e_1 \text{ else } e_2 \langle \Pi_2, \theta \cdot \tau(v=\text{True}) \cdot \Theta_2 \rangle\) where \(E \vdash \{\pi, \epsilon\} e_2 \{\Pi_2, \Theta_2\}\).

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

9. Method Call:

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

\(\Box\)

### B.4 Termination of the TRS

The TRS is terminating.

**Proof.** Let \(\text{Set}[\mathcal{I}]\) 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., \(S, H \in \text{Set}[\mathcal{I}]\). Consider the following partial ordering \(\succ\) on pairs \(\langle S, H \rangle\):

\[
\langle S_1, H_1 \rangle \succ \langle S_2, H_2 \rangle \text{ iff } |H_1| < |H_2| \lor (|H_1| = |H_2| \land |S_1| > |S_2|).
\]

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 5.5.1 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.

\(\Box\)

### B.5 Soundness of the TRS

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

**Proof.** By case analysis for each inference rules:
APPENDIX B. APPENDIX FOR TIMEFFS

1. Axiom Rules:

\[ \Gamma \vdash \pi \land \bot \subseteq \Phi \quad [Bot\text{-}LHS] \]
\[ \Phi \neq \pi \land \bot \vdash \Gamma \cap \Phi \subseteq \pi \land \bot \quad [Bot\text{-}RHS] \]

- 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:

\[ \delta_{\pi_1}(\theta_1) \land \neg \delta_{\pi_2}(\theta_2) \vdash \Gamma \vdash \pi_1 \land \theta_1 \nsubseteq \pi_2 \land \theta_2 \quad [DISPROVE] \]
\[ \pi_1 \Rightarrow \pi_2 \quad \text{fst}_{\pi_1}(\theta_1) = \{\} \vdash \Gamma \vdash \pi_1 \land \theta_1 \subseteq \pi_2 \land \theta_2 \quad [PROVE] \]

- It’s straightforward to prove soundness of the rule \([DISPROVE]\), Given that \(\theta_1\) is nullable, while \(\theta_2\) is not nullable, thus clearly the antecedent contains more event traces than the consequent. Therefore, these entailments are evidently invalid.

3. Prove Rules:

\[ \pi_1 \land \theta_1 \subseteq \pi_3 \land \theta_3 \in \Gamma \quad \pi_3 \land \theta_3 \subseteq \pi_4 \land \theta_4 \in \Gamma \quad \pi_4 \land \theta_4 \subseteq \pi_2 \land \theta_2 \in \Gamma \]
\[ \Gamma \vdash \pi_1 \land \theta_1 \subseteq \pi_2 \land \theta_2 \quad [REOCCUR] \]

- To prove soundness of the rule \([PROVE]\), we consider an arbitrary model, \(d, \mathcal{E}, \varphi\) such that: \(d, \mathcal{E}, \varphi \models \pi_1 \land \theta_1\). Given the side conditions from the promises, we get \(d, \mathcal{E}, \varphi \models \pi_2 \land \theta_2\). When the \(\text{fst}\) set of \(\theta_1\) is empty, \(\theta_1\) is possible \(\bot\) or \(\epsilon\) and \(\pi_2 \land \theta_2\) 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: \(d, \mathcal{E}, \varphi \models \pi_1 \land \theta_1\). Given the promises that \(\pi_1 \land \theta_1 \subseteq \pi_3 \land \theta_3\), we get \(d, \mathcal{E}, \varphi \models \pi_3 \land \theta_3\); Given the promise that there exists a hypothesis \(\pi_3 \land \theta_3 \subseteq \pi_4 \land \theta_4\), we get \(d, \mathcal{E}, \varphi \models \pi_4 \land \theta_4\); Given the promises that \(\pi_4 \land \theta_4 \subseteq \pi_2 \land \theta_2\), we get \(d, \mathcal{E}, \varphi \models \pi_2 \land \theta_2\). Therefore, the inclusion is valid.
4. Unfolding Rule:

\[
\begin{align*}
\text{[UNFOLD]} \\
H = \text{fst}_{\pi_1}(\theta_1) \quad \Gamma' = \Gamma, (\pi_1 \land \theta_1 \subseteq \pi_2 \land \theta_2) \quad \forall h \in H. \quad (\Gamma' \vdash D_h^{\pi_1}(\theta_1) \subseteq D_h^{\pi_2}(\theta_2)) \\
\Gamma \vdash \pi_1 \land \theta_1 \subseteq \pi_2 \land \theta_2
\end{align*}
\]

- 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: \(d_1, \mathcal{E}_1, \varphi_1 \models \pi_1 \land \theta_1\) and \(d_2, \mathcal{E}_2, \varphi_2 \models \pi_2 \land \theta_2\). For an arbitrary event \(h\), let \(d'_1, \mathcal{E}'_1, \varphi'_1 \models h^{-1}[\pi_1 \land \theta_1]\); and \(d'_2, \mathcal{E}'_2, \varphi'_2 \models h^{-1}[\pi_2 \land \theta_2]\).

   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 14, since for all \(h\), \(D_h^{\pi_1}(\theta_1) \subseteq D_h^{\pi_2}(\theta_2)\), the conclusion is valid.

All the inference 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'.

\[
\begin{align*}
[\text{Handle-Perform}] & \quad \forall (\pi, \theta, v) \in \Phi. l = A(v) \land \theta' = \theta \cdot l! \\
\frac{E, \mathcal{H}, \mathcal{D} \vdash_h \langle \Phi \rangle \text{ perform } A(v, \lambda x \Rightarrow e) \langle \forall \Phi' \rangle}{E, \mathcal{H}, \mathcal{D} \vdash_h \langle \Phi \rangle \text{ perform } A(v, \lambda x \Rightarrow e) \langle \forall \Phi' \rangle} \\
[\text{Handle-Let}] & \quad (x \mapsto v)! : E, \mathcal{H}, \mathcal{D} \vdash_h \langle \Phi \rangle e \langle \Phi' \rangle \\
\frac{E, \mathcal{H}, \mathcal{D} \vdash_h \langle \Phi \rangle \text{ let } x = v \text{ in } e \langle \Phi \rangle}{E, \mathcal{H}, \mathcal{D} \vdash_h \langle \Phi \rangle \text{ let } x = v \text{ in } e \langle \Phi \rangle} \\
[\text{Handle-If-Else}] & \quad E, \mathcal{H}, \mathcal{D} \vdash_h \langle \Phi \land (v = \text{true}) \rangle e_1 \langle \Phi_1 \rangle \\
\frac{E, \mathcal{H}, \mathcal{D} \vdash_h \langle \Phi \land (v = \text{false}) \rangle e_2 \langle \Phi_2 \rangle}{E, \mathcal{H}, \mathcal{D} \vdash_h \langle \Phi \rangle \text{ if } v \text{ then } e_1 \text{ else } e_2 \langle \Phi_1 \rangle \cup \langle \Phi_2 \rangle} \\
[\text{Handle-App}] & \quad E(v_1) = \tau mn (\tau v) \text{ req } \Phi_{pre} \text{ ens } \Phi_{post} \{ e \} \quad \Phi \subseteq \Phi_{pre}[v_2/v] \\
\frac{E, \mathcal{H}, \mathcal{D} \vdash_h \langle \Phi \rangle v_1 v_2 \langle \Phi \cdot \Phi_{post}[v_2/v] \rangle}{E, \mathcal{H}, \mathcal{D} \vdash_h \langle \Phi \rangle v_1 v_2 \langle \Phi \cdot \Phi_{post}[v_2/v] \rangle} \\
[\text{Handle-Call}] & \quad E \vdash \langle (\text{True}, e, ()) \rangle e \langle \Phi' \rangle \quad E, h \vdash_{fix} \Phi' \cdot \triangleright \sim \Phi_{fix} \\
\frac{E, \mathcal{H}, \mathcal{D} \vdash_h \langle \Phi \rangle \text{ match } e \text{ with } h \langle \Phi \cdot \Phi_{fix} \rangle}{E, \mathcal{H}, \mathcal{D} \vdash_h \langle \Phi \rangle \text{ match } e \text{ with } h \langle \Phi \cdot \Phi_{fix} \rangle} \\
\end{align*}
\]
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.

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 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).

\[
\forall e, \forall E, \forall H, \forall D, \forall \Phi \text{ if } E, H, D \vdash_h \langle \Phi \rangle e \langle \Phi' \rangle, \text{ is valid, then: } \forall (\pi, \theta, v) \in \Phi, \\
\text{when } e=\text{let } x=\kappa v' \text{ in } e' \text{ (one shot triggered),} \\
\text{all the } \kappa! \text{ are substituted by } v' \text{ in } D, \text{ and} \\
\|E, \theta, H\| \vdash_{fix} (\pi, D, v) \leadsto \Phi' \text{ is valid,} \\
\text{such that } E, H, D \vdash_h \langle \Phi' \rangle e \langle \Phi'' \rangle \text{ is valid;} \quad (a) \\
\text{when } e=v' \text{ (zero-shot), } v \text{ is updated to } v', \text{ and } D \text{ is left unhandled;} \quad (b) \\
\text{otherwise, } E \vdash \langle \Phi \rangle e \langle \Phi' \rangle \text{ is valid.} \quad (c)
\]

Proof. By induction on the structure of \( e \).
APPENDIX C. APPENDIX FOR CONTEFFS

- Where $e=\text{let } x=\kappa \ v'$ 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 Verification System

C.2.1 Soundness of the Fixpoint Computation

Given an effect $\Phi$, with the environment $\mathcal{E}$ and handler $\mathcal{H}$. $\Phi_{\text{fix}}$ is the updated version of $\Phi$, where all $\Phi$’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 $\mathcal{E}, \mathcal{H} \vdash_{\text{fix}} \Phi \leadsto \Phi_{\text{fix}}$ is valid, then:

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

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

if $\alpha=\diamond:([-x\rightarrow v]):\mathcal{E} \vdash (\pi, \theta_{\text{his}}, v) \vdash_{\text{ret}} (\Phi')$ is valid, given $(\text{return } x\rightarrow e_{\text{ret}}) \in \mathcal{H}$; (2)

if $\alpha\in\{\text{ev}, \text{l!}, \text{l?}(v')\}$ ($1 \not\in \mathcal{H}$) : $\| \mathcal{E}, \theta_{\text{his}} \cdot \alpha, \mathcal{H} \| \vdash_{\text{fix}} (\pi, D_{\alpha}(\theta), v) \leadsto \Phi' \text{ is valid}$; (3)

if $\alpha\in\{\text{l!}\}$ ($1 \in \mathcal{H}$) : $(x\rightarrow v):\mathcal{E}, \mathcal{H}, D_{\alpha}(\theta) \vdash_{h} (\pi, \theta_{\text{his}} \cdot 1, v) e \langle \Phi' \rangle \text{ is valid},$

\[\text{given (effect } A(x, \kappa) \mapsto e) \in \mathcal{H}. \quad (4)\]

Proof. By case analysis on the rules for fixpoint computation, we prove the soundness of the presented rules.

- For [Fix-Disj], where $\Phi$ is a set, by applying the rule itself, the conclusion of case (1) is satisfied.

- For [Fix-Normal], where $\alpha=\diamond$, by applying the rule itself, the conclusion of case (2) is satisfied.
- For [Fix-Unfold-Skip], where $\alpha \in \{ev, l, l'(v')\}$ (l$\notin$H), by applying the rule itself, the conclusion of case (3) is satisfied.
- For [Fix-Unfold-Handle], where $\alpha \in \{l\}$ (l$\in$H), by applying the rule itself and Lemma 4, the conclusion of case (4) is satisfied.

\section*{C.2.2 Soundness of Forward 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.

Formally, $\forall e, \forall E, \forall \varphi, \forall \Phi$ given $[e, E, \varphi] \xrightarrow{\mathcal{I}} [x, E', \varphi']$ and $E \vdash \langle \Phi \rangle e \langle \Phi' \rangle$,

if $E, \varphi \models \Phi$ then $E', \varphi' \models \Phi'$.

Proof. By induction on the structure of $e$.

- When $e=\langle v_1 \ v_2 \rangle$:

$$
\frac{e = v_1 \ v_2 \quad E(v_1) = A(v) \? \quad \text{[Inst-App]} \quad E(v_1) = 1? \quad \theta = 1?(v_2) \quad \text{[FV-App]}}{
\left[ e, E, \varphi \right] \xrightarrow{\mathcal{I}} \left[ e', E, \varphi++[A(v)@(v_2)] \right] \quad E \vdash \langle \Phi \rangle v_1v_2 \langle \Phi \cdot \theta \rangle}
$$

By the rule [Inst-App], we have: $\left[ e, E, \varphi \right] \xrightarrow{\mathcal{I}} \left[ e', E, \varphi++[A(v)@(v_2)] \right]$. Next since $E, \varphi \models \Phi$, we can obtain $E, \varphi++[A(v)@(v_2)] \models \Phi \cdot A(v)@(v_2)$. And in the same $E$, $E(v_1)=A(v)?=1?$. Therefore $E, \varphi++[A(v)@(v_2)] \models \Phi \cdot 1?(v_2)$, where the goal $E', \varphi' \models \Phi'$ is proved.

- When $e=\langle \text{let } x=v \text{ in } e_1 \rangle$:

$$
\frac{e = \text{let } x=v \text{ in } e_1 \quad \text{[Inst-Bind]} \quad (x\rightarrow v)::E \vdash \langle \Phi \rangle e_1 \langle \Phi' \rangle \quad \text{[FV-Let]}}{
\left[ e, E, \varphi \right] \xrightarrow{\mathcal{I}} \left[ e_1, (x\rightarrow v)::E, \varphi \right] \quad E \vdash \langle \Phi \rangle \text{ let } x=v \text{ in } e_1 \langle \Phi' \rangle}
$$

By the rule [Inst-Bind], we have: $\left[ e, E, \varphi \right] \xrightarrow{\mathcal{I}} \left[ e_1, (x\rightarrow v)::E, \varphi \right] \xrightarrow{\mathcal{I}} [x, E', \varphi']$. By the rule [FV-Let], we have $(x\rightarrow v)::E \vdash \langle \Phi \rangle e_1 \langle \Phi' \rangle$. Given, $E, \varphi \models \Phi$, therefore $(x\rightarrow v)::E, \varphi \models \Phi$ By the inductive hypothesis on $e_1$, we get $E', \varphi' \models \Phi'$, where the goal $E', \varphi' \models \Phi'$ is proved.
When \( e = \text{match perform } A(v, \lambda x \mapsto e_1) \) with \( h \) and \( A \notin h \):

\[
\begin{align*}
\frac{e = \text{match perform } A(v, \lambda x \mapsto e_1) \text{ with } h \quad A \notin h}{[e, E, \varphi] \xrightarrow{t} [e_1, (x \mapsto A(v)) : : E, \varphi + + [A(v)!]]} \quad \text{[Inst-Escape]}
\end{align*}
\]

By the rule [Inst-Escape], we have: \([e, E, \varphi] \xrightarrow{t} [e_1, (x \mapsto A(v)) : : E, \varphi + + [A(v)!]] \xrightarrow{*} [v', E', \varphi']\). Then by the rule [FV-Match] \( E \vdash \langle \Phi \rangle e \langle \Phi' \rangle \), and \( \Phi' = \Phi \cdot \Phi_{fix} \), where

\[
E \vdash ((\text{True}, \epsilon, ())) \text{ perform } A(v, \lambda x \mapsto e_1) \langle \Phi' \rangle
\]

\[
\Phi' = \{ (\pi, \theta \cdot \varphi, v) \mid (\pi, \theta, v) \in \Phi \} \quad \frac{E, h \vdash_{fix} \Phi' \leadsto \Phi_{fix}}{E \vdash \langle \Phi \rangle \text{ match perform } A(v, \lambda x \mapsto e_1) \langle \Phi' \rangle} \quad \text{[FV-Match]}
\]

Then by rules [Fix-Disj] and [Fix-Unfold-Skip], \( \Phi_{fix} = A(v) \cdot \Phi_{e1} \).

\[
\forall (\pi, \theta, v) \in \Phi. \| \langle \epsilon, \alpha \cdot \theta, v \rangle \| \vdash_{fix} (\pi, \theta, v) \leadsto \Phi' \quad \frac{E, \mathcal{H} \vdash_{fix} \Phi \leadsto \Phi'}{E, \mathcal{H} \vdash_{fix} \Phi \leadsto \Phi'} \quad \text{[Fix-Disj]}
\]

\[
\alpha \in \{ ev, !, 1?(v') \} \quad (1 \notin \mathcal{H}) \quad \frac{\| \langle \epsilon, \theta_{his}' f, \mathcal{H} \| \vdash_{fix} (\pi, \theta, v) \leadsto \Phi' \| \langle \theta_{his} \cdot \mathcal{H} \| \vdash_{fix} (\pi, \alpha \cdot \theta, v) \leadsto \Phi' \quad \text{[Fix-Unfold-Skip]}
\]

Given, \( E, \varphi \vdash \Phi \), therefore \( (x \mapsto A(v)) : : E, \varphi + + [A(v)!] \vdash \Phi \cdot A(v) \). By the inductive hypothesis on \( e_1 \), we get \( E', \varphi' \vdash \Phi \cdot A(v) \cdot \Phi_{e1} \), where the goal \( E', \varphi' \vdash \Phi' \) is proved.

- When \( e = \text{match perform } A(v, \lambda x \mapsto e_1) \) with \( h \) and \( A \in h \):

\[
\begin{align*}
\frac{e = \text{match perform } A(v, \lambda x \mapsto e_1) \text{ with } h \quad A \in h}{[e, E, \varphi] \xrightarrow{t} [e_1, E, \varphi + + [A(v)]]} \quad \text{[Inst-Caught]}
\end{align*}
\]

By the rule [Inst-Caught], we have: \([e, E, \varphi] \xrightarrow{t} [e_1, E, \varphi + + [A(v)]] \xrightarrow{*} [v', E', \varphi']\). Then by the rule [FV-Match] \( E \vdash \langle \Phi \rangle e \langle \Phi' \rangle \), and \( \Phi' = \Phi \cdot \Phi_{fix} \), where \( E, h \vdash_{fix} (A(v)! \cdot \Phi_{e1}) \leadsto \Phi_{fix} \), and \( A(v) : : E \vdash (A(v)!) e_1 (A(v)! \cdot \Phi_{e1}) \), by [FV-Perform].

Then by rules [Fix-Disj] and [Fix-Unfold-Handle], \( \Phi_{fix} = A(v) \cdot \Phi_{e1} \).
\[ \alpha \in \{1!\} \quad (\text{effect } A(x, \kappa) \mapsto e) \in \mathcal{H} \quad (1 = A(v)) \]

\[
\frac{\mathcal{E}, \mathcal{H}, \theta \vdash_h ((\pi, \theta_{\text{his}} \cdot 1, v)) \ e[v/x, \kappa/\text{resume}] \langle \Phi \rangle}{\parallel \mathcal{E}, \theta_{\text{his}}, \mathcal{H} \parallel \vdash_{\text{fix}} (\pi, \alpha \cdot \theta, v) \leadsto \Phi'} \quad [\text{Fix-Unfold-Handle}]
\]

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

\[ \square \]

**C.3 Termination Proof of the TRS**

*Proof.* Let \( \text{Set}[\mathcal{I}] \) 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., \( S, H \in \text{Set}[\mathcal{I}] \). Consider the following partial ordering \( \succ \) on pairs \( \langle S, H \rangle \):

\[ \langle S_1, H_1 \rangle \succ \langle S_2, H_2 \rangle \iff |H_1| < |H_2| \lor (|H_1| = |H_2| \land |S_1| > |S_2|). \]

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 6.5.1) 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.

\[ \square \]

**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:**

\[
\frac{\Omega \vdash (\pi_1, \bot) \subseteq (\pi_2, \theta)}{[\text{Bot-LHS}]}
\]

\[
\frac{\theta \neq \bot}{\Omega \vdash (\pi_1, \theta) \not\subseteq (\pi_2, \bot)}{[\text{Bot-RHS}]}
\]

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

2. **Disprove Rules:**

\[
\frac{\delta(\theta_1) \land \neg\delta(\theta_2)}{\Omega \vdash (\pi_1, \theta_1) \not\subseteq (\pi_2, \theta_2)}{[\text{Dis-Nullable}]}
\]

\[
\frac{\varkappa(\theta_1) \land \neg\varkappa(\theta_2)}{\Omega \vdash (\pi_1, \theta_1) \not\subseteq (\pi_2, \theta_2)}{[\text{Dis-Infinitable}]}
\]

- It’s straightforward to prove soundness of the rule [Dis-Nullable], Given that \(\theta_1\) is possibly empty trace, while \(\pi_2 \land \theta_2\) 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 \(\theta_1\) is possibly infinite trace, while \(\pi_2 \land \theta_2\) 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:**

\[
\frac{(\pi_1, \theta_1) \subseteq (\pi_3, \theta_3) \in \Omega \quad (\pi_3, \theta_3) \subseteq (\pi_4, \theta_4) \in \Omega \quad (\pi_4, \theta_4) \subseteq (\pi_2, \theta_2) \in \Omega}{\Omega \vdash (\pi_1, \theta_1) \subseteq (\pi_2, \theta_2)}{[\text{Reoccur}]}
\]

- To prove soundness of the rule [Reoccur], we consider an arbitrary model, \(\mathcal{E}, \phi\) such that: \(\mathcal{E}, \phi \models \pi_1 \land \theta_1\). Given the promise that \(\theta_1 \subseteq \pi_3 \land \theta_3\), we get \(\mathcal{E}, \phi \models \pi_3 \land \theta_3\); Given the promise that there exists a hypothesis \(\pi_3 \land \theta_3 \subseteq \pi_4 \land \theta_4\), we get \(\mathcal{E}, \phi \models \theta_4\); Given the promise that \(\pi_4 \land \theta_4 \subseteq \pi_2 \land \theta_2\), we get \(\mathcal{E}, \phi \models \pi_2 \land \theta_2\). Therefore, the entailment is valid.
4. Unfolding Rule:

\[ F = \text{fst}(\theta_1) \quad \pi_1 \Rightarrow \pi_2 \quad \forall \alpha \in F. (\theta_1 \sqsubseteq \theta_2) :: \Omega \vdash D_\alpha(\theta_1) \sqsubseteq D_\alpha(\theta_2) \]  

\[ \Omega \vdash (\pi_1, \theta_1) \sqsubseteq (\pi_2, \theta_2) \quad [\text{Unfold}] \]

- To prove soundness of the rule \([\text{Unfold}]\), we consider an arbitrary model, \(E_1, \varphi_1\) and \(E_2, \varphi_2\) such that: \(E_1, \varphi_1 \models \theta_1\) and \(E_2, \varphi_2 \models \pi_2 \land \theta_2\). For an arbitrary event \(\alpha\), let \(E'_1, \varphi_1' \models \alpha^{-1}[\theta_1]\); and \(E'_2, \varphi_2' \models \alpha^{-1}[\pi_2 \land \theta_2]\).

  Case 1), \(\alpha \notin F\), \(E'_1, \varphi_1' \models \bot\), thus automatically \(E'_1, \varphi_1' \models \pi_2 \land D_\alpha(\theta_2)\);

  Case 2), \(\alpha \in F\), given that inclusions in the rule’ premise is valid, then \(E'_1, \varphi_1' \models \pi_2 \land D_\alpha(\theta_2)\).

By Definition 7, since for all \(\alpha\), \(\pi_1 \land D_\alpha(\theta_1) \sqsubseteq \pi_2 \land D_\alpha(\theta_2)\), the conclusion is valid.

All the rewriting rules used in the TRS are sound, therefore the TRS is sound. \(\square\)
Publications during PhD Study

