Automated Temporal Verification of Integrated Dependent Effects

Yahui Song and Wei-Ngan Chin

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

Abstract. Existing approaches to temporal verification have either sacrificed compositionality in favor of achieving automation or vice-versa. To exploit the best of both worlds, we present a new solution to ensure temporal properties via a Hoare-style verifier and a term rewriting system (T.r.s) on *Integrated Dependent Effects*. 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 T.r.s 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.

1 Introduction

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

In this paper, we specify system behaviors in the form of *Integrated Dependent Effects*, which integrates the basic and ω-regular expressions with dependent values and arithmetic constraints, gaining the expressive power beyond finite-state machines. Specifically, our novel effects provide insights of: (i) Definite finite traces: we use symbolic values to present finite repetitions, which can be dependent on program inputs; (ii) Definite infinite traces constructed by infinity operator (ω); (iii) Possibly finite and possibly infinite traces constructed by Kleene star ($\star$). For example, we express, the effects of method `send(n)` as:

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

The `send` method takes a parameter `n`, and recursively sends out `n` messages. The above specification of `send(n)` indicates the fact that for non-negative values of

the parameter `n`, the `send` method generates a finite trace comprising a sequence with `n` times of event **Send**, followed by a final event **Done**. For the case when the parameter is negative, it generates an infinite trace of event **Send**. Note that (i) the integrated effects can express both finite traces and infinite traces in one single formula, separated by arithmetic constraints, and (ii) `n` is a parameter to `send`, making the effects *dependent* w.r.t the value of `send`'s parameter. Furthermore, by allowing events to be parametrised with symbolic values, the effects are defined as languages over potentially infinite alphabets of the form $\Sigma \times \mathbb{Z}$, where Σ is a finite event set, and $\mathbb{Z}$ is the infinite integer set.

Deciding the inclusion between two regular sets is PSPACE-complete.The standard approaches to the problem are based on the following steps: (i) translate each regular expression into an equivalent NFA, (ii) convert those NFAs to equivalent DFAs and finally (iii) minimize those DFAs to $\mathcal{M}_{\mathcal{A}}$ and $\mathcal{M}_{\mathcal{B}}$, and then check emptiness of $\mathcal{M}_{\mathcal{A}} \cap \neg\mathcal{M}_{\mathcal{B}}$. However, any efficient algorithm[9] based on such translation potentially gives rise to an exponential blow-up.

As an alternative approach, Antimirov and Mosses[5] presented a term rewriting system (T.r.s) for deciding the inclusion of regular expressions based on a complete axiomatic algorithm of the algebra of regular sets. A T.r.s is a refutation method that normalizes regular expressions in such a way that checking their inclusion corresponds to an iterated process of checking the inclusion of their *partial derivatives*[4]. Works based on such a T.r.s[5,3,12,11] show its feasibility and suggest that this method is a better average-case algorithm than those based on the comparison of minimal DFAs.

In this paper, we present 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 Integrated Dependent Effects. Our main contributions are:

1. **Temporal Effects Specification:** We define the syntax and semantics of our novel effects, which escapes LTL, μ-calculus and prior effects (Sec. 3).
2. **Automated Verification System:** Targeting a core language, we develop a Hoare-style forward verifier to accumulate effects from the source code, as the front-end (Sec. 4); and a sound decision procedure (our T.r.s) to solve the effects inclusions, as the back-end (Sec. 5).
3. **Implementation and Evaluation:** We prototype the novel effects logic on top of the HIP/SLEEK system[8][2]. We further provide case studies and experimental results to show the feasibility of our method (Sec. 6).

Organization. Sec. 2 gives a straightforward motivation example to highlight the key methodologies and contributions. Sec. 3 formally specifies the syntax of the target language, and the syntax and semantics of our integrated dependent effects. Sec. 4 presents the forward verifier for the target language. Sec. 5 illustrates the effects inclusion checking procedure, by presenting a set of inference rules, and displays the essential auxiliary functions. Sec. 6 demonstrates the implementation, case studies and experimental results as the evaluation of our T.r.s. We discuss related works in Sec. 7 and conclude in Sec. 8. Termination and soundness proofs can be found in the extended technical report[2].

2 Overview

We now give a summary of our techniques, using the example shown in Table 1.-(a). Our integrated dependent effects can be illustrated with `send` and `server`, which simulate a server who continuously sends messages to all its clients.

Table 1. **(a)** Source code and **(b)** pre/post effects specifications for the methods.

(a) Source Code	(b) Effects Specifications

`void send (int n){`
`  if (n==0) {`
`  event["Done"];`
`  }else{`
`  event["Send"]; send (n-1);`
`  }}`
`void server (int n){`
`  event["Ready"];`
`  send(n);`
`  server(n);}`
	$\Phi_{\mathtt{pre}}^{\mathtt{send(n)}} \triangleq \mathtt{True} \wedge \underline{\mathbf{Ready}} \cdot \_^*$
	$\Phi_{\mathtt{post}}^{\mathtt{send(n)}} \triangleq (\mathtt{n} \geq 0 \wedge \underline{\mathbf{Send}}^{\mathtt{n}} \cdot \underline{\mathbf{Done}}) \vee (\mathtt{n} < 0 \wedge \underline{\mathbf{Send}}^{\omega})$
	
	$\Phi_{\mathtt{pre}}^{\mathtt{server(n)}} \triangleq \mathtt{n} \geq 0 \wedge \epsilon$
	$\Phi_{\mathtt{post}}^{\mathtt{server(n)}} \triangleq \mathtt{n} \geq 0 \wedge (\underline{\mathbf{Ready}} \cdot \underline{\mathbf{Send}}^{\mathtt{n}} \cdot \underline{\mathbf{Done}})^{\omega}$

Here, `event`[**a**] is a primitive in our target language (cf. Sec. 3), used to trigger a single event **a**. This method `server` takes an integer parameter `n`, triggers an event **Ready**, then calls the method `send`, making a boolean choice depending on input `n`: in one case it triggers an event **Done**; otherwise it triggers an event **Send**, then makes a recursive call with parameter `n-1`. Finally `server` recurs.

2.1 Integrated Dependent Effects. The effects specifications for `server` and `send` are given in Table 1.-(b). We define 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 `send`'s precondition, denoted by $\Phi_{\mathtt{pre}}^{\mathtt{send(n)}}$, requires the event **Ready** to have happened at some point of the effects history; and it guarantees the final effects/postcondition, denoted by $\Phi_{\mathtt{post}}^{\mathtt{send(n)}}$.

Method `server`'s precondition, $\Phi_{\mathtt{pre}}^{\mathtt{server(n)}}$, requires the input value be non-negative while the pre-trace is required to be empty (ϵ); its postcondition ensures the final effects $\Phi_{\mathtt{post}}^{\mathtt{server(n)}}$ – an infinite repetition of a trace consisting of an event **Ready** followed by `n` times of **Send** followed by **Done**. Directly from the specifications, we are aware of (i) termination properties: `server` *must* not terminate, while `send` *may* not terminate; (ii) branching properties: different arithmetic conditions on the input parameters lead to different temporal effects; and (iii) required history traces: by defining the prior effects in precondition. The examples already show that our effects provide more detail information than classical LTL or μ-calculus, and in fact, it cannot be fully captured by any prior works[10,13,15,17]. Nevertheless, the gain in expressive power comes at the efforts of a more dedicated verification process, namely handled by our T.r.s.

2.2 Forward Verification. As shown in Fig. 1., we demonstrate the forward verification process of method `send`. The current effects states of a program is captured in the form of $\{\Phi_{\mathtt{C}}\}$. We define our forward verification rules in Sec.

4. To facilitate the illustration, we label the verification steps by 1), ..., 8). We mark the deployed verification rules in green. The verifier invokes the T.r.s to check language inclusions along the way.

1) `void send (int n){` *(– initialize the current effects state –)*
   $\{\Phi_c{=}\Phi_{pre}^{send(n)}{=}\texttt{True} \wedge \underline{\textbf{Ready}} \cdot \_^\star\}$ [FV-Meth]
2) `if(n==0){`
   $\{\texttt{n=0} \wedge \underline{\textbf{Ready}} \cdot \_^\star\}$ [FV-If-Else]
3) `event[Done]; }`
   $\{\texttt{n=0} \wedge \underline{\textbf{Ready}} \cdot \_^\star \cdot \underline{\textbf{Done}}\}$ [FV-Event]
4) `else{`
   $\{\texttt{n}{\neq}\texttt{0}\}$ [FV-If-Else]
5) `event[Send];`
   $\{\texttt{n}{\neq}\texttt{0} \wedge \underline{\textbf{Ready}} \cdot \_^\star \cdot \underline{\textbf{Send}}\}$ [FV-Event]
6) `send(n-1); }}`
   $\texttt{rev}(\texttt{n}{\neq}\texttt{0} \wedge \underline{\textbf{Ready}} \cdot \_^\star \cdot \underline{\textbf{Send}}) \sqsubseteq \texttt{rev}(\Phi_{pre}^{send(n-1)})$ *(-check precondition-)*
   $\{\texttt{n}{\neq}\texttt{0} \wedge \underline{\textbf{Ready}} \cdot \_^\star \cdot \underline{\textbf{Send}} \cdot \Phi_{post}^{send(n-1)}\}$ [FV-Call]
7) $\Phi'_c{=}(\texttt{n=0} \wedge \underline{\textbf{Ready}} \cdot \_^\star \cdot \underline{\textbf{Done}}) \vee (\texttt{n}{\neq}\texttt{0} \wedge \underline{\textbf{Ready}} \cdot \_^\star \cdot \underline{\textbf{Send}} \cdot \Phi_{post}^{send(n-1)})$
8) $\Phi'_c \sqsubseteq \Phi_{pre}^{send(n)} \cdot \Phi_{post}^{send(n)} \Leftrightarrow$ *(– check postcondition –)*
   $(\texttt{n=0} \wedge \underline{\textbf{Done}}) \vee (\texttt{n}{\neq}\texttt{0} \wedge \underline{\textbf{Send}} \cdot \Phi_{post}^{send(n-1)}) \sqsubseteq \Phi_{post}^{send(n)}$

Fig. 1. The forward verification example for method `send`.

The effects state 1) is obtained by initialising Φ_c from the precondition.

The effects states 2), 4) and 7) are obtained by [`FV-If-Else`], which adds the constraints from the conditionals into the current effects state, and unions the effects accumulated from two branches in the end. The effects states 3) and 5) are obtained by [`FV-Event`], which simply concatenates the triggered singleton event to the end of the current effects state. The effects state 6) is obtained by [`FV-Call`]. Before each method call, it checks whether the current state satisfies the precondition of the callee method. The `rev` function simply reverses the order of effects sequences. If the precondition is not satisfied, then the verification fails, otherwise it concatenates the postcondition of the callee to the current effects.

While Hoare logics based on finite traces (terminating runs)[14] and infinite traces (non-terminating runs)[16] have been considered before, the reasoning on properties of mixed definitions is new. Prior effects in precondition is also new, allowing greater safety to be applied to sequential reactive controlling systems such as web applications, communication protocols and IoT systems.

2.3 The t.r.s. Our T.r.s is designed to check the inclusion between any two integrated dependent effects. We define its inference rules in Sec. 5. Here, we present the rewriting process on the postcondition checking of the method `send`.We mark the rules of some essential inference steps in green. Basically, our effects rewriting system decides effects inclusion through an iterated process of checking the inclusion of their partial derivatives. There are two important rules

Table 2. The inclusion checking example on the postcondition of method `send`. `I` : The main rewriting proof tree (coming from the step 8) in Fig. 1.); `II` : One sub-tree of the rewriting process.

$$
\mathtt{I}:\ \dfrac{\dfrac{\dfrac{\dfrac{(\mathtt{n}{=}0)\wedge\epsilon \sqsubseteq \epsilon \quad [\text{FRAME}]}{(\mathtt{n}{=}0)\wedge \underline{\mathtt{Done}} \sqsubseteq \underline{\mathtt{Done}}}}{(\mathtt{n}{=}0)\wedge \underline{\mathtt{Done}} \sqsubseteq \underline{\mathtt{Send}}^{0}\cdot\underline{\mathtt{Done}}}}{(\mathtt{n}{=}0)\wedge \underline{\mathtt{Done}} \sqsubseteq \Phi^{\mathtt{send(n)}}_{\mathtt{post}}} \qquad \mathtt{II} \qquad \dfrac{\dfrac{\mathtt{n}{<}0\wedge \underline{\mathtt{Send}}^{\omega} \sqsubseteq \underline{\mathtt{Send}}^{\omega}\ (\dagger)\ [\text{REOCCUR}]}{\mathtt{n}{<}0\wedge \underline{\mathtt{Send}}^{\omega} \sqsubseteq \underline{\mathtt{Send}}^{\omega}\ (\dagger)}\ [\text{UNFOLD}]}{\mathtt{n}{<}0\wedge \underline{\mathtt{Send}}\cdot\underline{\mathtt{Send}}^{\omega} \sqsubseteq \underline{\mathtt{Send}}^{\omega}}\ [\text{UNFOLD}]}{\mathtt{n}{<}0\wedge \underline{\mathtt{Send}}\cdot\underline{\mathtt{Send}}^{\omega} \sqsubseteq \Phi^{\mathtt{send(n)}}_{\mathtt{post}}}\ [\text{DISJUNCTION}]}{(\mathtt{n}{=}0\wedge \underline{\mathtt{Done}}) \vee (\mathtt{n}{\neq}0\wedge \underline{\mathtt{Send}}\cdot\Phi^{\mathtt{send(n-1)}}_{\mathtt{post}}) \sqsubseteq \Phi^{\mathtt{send(n)}}_{\mathtt{post}}}
$$

$$
\mathtt{II}:\ \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathtt{n}_1{=}0\wedge\epsilon \sqsubseteq \epsilon \quad [\text{FRAME}]}{\mathtt{n}_1{=}0\wedge \underline{\mathtt{Done}} \sqsubseteq \underline{\mathtt{Done}}} \qquad \dfrac{\dfrac{(\mathtt{n}_2{=}\mathtt{n}_1\text{-}1\wedge\mathtt{n}_2{\geq}0)\wedge\underline{\mathtt{Send}}^{\mathtt{n}_2}\cdot\underline{\mathtt{Done}} \sqsubseteq \underline{\mathtt{Send}}^{\mathtt{n}_2}\cdot\underline{\mathtt{Done}}\ (\ddagger)\ [\text{REOCCUR}]}{\mathtt{n}_1{>}0\wedge \underline{\mathtt{Send}}^{\mathtt{n}_1\text{-}1}\cdot\underline{\mathtt{Done}} \sqsubseteq \underline{\mathtt{Send}}^{\mathtt{n}_1\text{-}1}\cdot\underline{\mathtt{Done}}}\ [\text{UNFOLD}]}{\mathtt{n}_1{>}0\wedge \underline{\mathtt{Send}}^{\mathtt{n}_1}\cdot\underline{\mathtt{Done}} \sqsubseteq \underline{\mathtt{Send}}^{\mathtt{n}_1}\cdot\underline{\mathtt{Done}}}\ [\text{CASESPLIT}]}{(\mathtt{n}_1{=}\mathtt{n}\text{-}1\wedge\mathtt{n}_1{\geq}0)\wedge \underline{\mathtt{Send}}^{\mathtt{n}_1}\cdot\underline{\mathtt{Done}} \sqsubseteq \underline{\mathtt{Send}}^{\mathtt{n}_1}\cdot\underline{\mathtt{Done}}\ (\ddagger)}\ [\text{SUBSTITUTE}]}{\mathtt{n}{>}0\wedge \underline{\mathtt{Send}}^{\mathtt{n}\text{-}1}\cdot\underline{\mathtt{Done}} \sqsubseteq \underline{\mathtt{Send}}^{\mathtt{n}\text{-}1}\cdot\underline{\mathtt{Done}}}\ [\text{UNFOLD}]}{\mathtt{n}{>}0\wedge \underline{\mathtt{Send}}\cdot\underline{\mathtt{Send}}^{\mathtt{n}\text{-}1}\cdot\underline{\mathtt{Done}} \sqsubseteq \underline{\mathtt{Send}}^{\mathtt{n}}\cdot\underline{\mathtt{Done}}}}{\mathtt{n}{>}0\wedge \underline{\mathtt{Send}}\cdot\underline{\mathtt{Send}}^{\mathtt{n}\text{-}1}\cdot\underline{\mathtt{Done}} \sqsubseteq \Phi^{\mathtt{send(n)}}_{\mathtt{post}}}
$$

inherited from Antimirov and Mosses's algorithm: [`DISPROVE`], which infers false from a trivially inconsistent inclusion; and [`UNFOLD`], which applies Theorem 1 to generate new inclusions. $\mathtt{D_a(r)}$ is the partial derivative of `r` w.r.t the event **a**. Termination is guaranteed because the set of derivatives to be considered is finite, and possible cycles are detected using *memorization*.

Theorem 1 (Regular Expressions Inclusion). *For regular expressions* `r` *and* `s`, $\mathtt{r} \preceq \mathtt{s} \Leftrightarrow (\forall \mathtt{a} \in \Sigma).\ \mathtt{D_a(r)} \preceq \mathtt{D_a(s)}$.

Intuitively, we use [`DISPROVE`] wherever the left-hand side (LHS) is *nullable*[1] while the right-hand side (RHS) is not. [`DISPROVE`] is essential because it is the heuristic refutation step to disprove the inclusion early, which leads to a great efficiency improvement compared to the standard methods.

Besides, we use symbolic values (assuming non-negative) to capture the finite traces, depended on program inputs. Whenever the symbolic value is possibly zero, we use the rule [`CASESPLIT`] to distinguish the zero (base) and non-zero (inductive) cases, as shown in Table 2.-`II`. In addition, the T.r.s is obligated to reason about mixed inductive (finite) and coinductive (infinite) definitions. We achieve these features and still guarantee the termination by using rules: [`SUBSTITUTE`], which renames the symbolic terms using free variables; and [`REOCCUR`], which finds the syntactic identity, as a *companion*, of the current open goal, as a *bud*, from the internal proof tree[7]. (We use (†) and (‡) in Table 2. to indicate the pairing of buds with companions.)

[1] If the event sequence is possibly empty, i.e. contains ϵ, we call it nullable, formally defined in Definition 1.

3 Language and Specifications

In this section, we first introduce the target (sequential C-like) language and then depict the temporal specification language which supports our effects.

3.1 Target Language. The syntax of our core imperative language is given in Fig. 2. Here, we regard $\mathtt{k}$ and $\mathtt{x}$ are meta-variables. $\mathtt{k}^{\tau}$ represents a constant of basic type τ. **var** represents the countably infinite set of arbitrary distinct identifiers. $\underline{\mathtt{a}}$ refers to a singleton event coming from the finite set of events Σ. We assume that programs we use are well-typed conforming to basic types τ (we take () as the `void` type). A program $\mathcal{P}$ comprises a list of method declarations $\mathtt{meth}^*$. Here, we use the $*$ superscript to denote a finite list (possibly empty) of items, for example, $\mathtt{x}^*$ refers to a list of variables, $\mathtt{x}_1, ..., \mathtt{x}_n$.

$$
\begin{array}{lll}
(\textit{Program}) & \mathcal{P} ::= \mathtt{meth}^* & (\textit{Basic Types}) \quad \tau ::= \mathtt{int} \mid \mathtt{bool} \mid \mathtt{void} \\
(\textit{Method Def.}) & \mathtt{meth} ::= \tau\ \mathtt{mn}\ (\tau\ \mathtt{x})^*\ \{\textbf{requires}\ \Phi_{\mathtt{pre}}\ \textbf{ensures}\ \Phi_{\mathtt{post}}\}\ \{\mathtt{e}\} & \\
(\textit{Expressions}) & \mathtt{e} ::= () \mid \mathtt{k}^{\tau} \mid \mathtt{x} \mid \tau\ \mathtt{x};\ \mathtt{e} \mid \mathtt{mn}(\mathtt{x}^*) \mid \mathtt{x}{:=}\mathtt{e} \mid\ \mathtt{e}_1; \mathtt{e}_2 \mid \textbf{assert}\ \Phi & \\
 & \quad \mid \mathtt{e}_1\ \mathit{op}\ \mathtt{e}_2 \mid \textbf{event}[\underline{\mathtt{a}}] \mid \textbf{if}\ \mathtt{v}\ \textbf{then}\ \mathtt{e}_1\ \textbf{else}\ \mathtt{e}_2 &
\end{array}
$$

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

Fig. 2. A Core Imperative Language.

Each method `meth` has a name `mn`, an expression-oriented body `e`, also is associated with a precondition $\Phi_{\mathtt{pre}}$ and a postcondition $\Phi_{\mathtt{post}}$ (the syntax of effects specification Φ is given in Fig. 3.). The language allows each iterative loop to be optimized 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 minimising our core language. Expressions comprise unit (), constants `k`, variables `x`, local variable declaration $\tau\ \mathtt{x};\ \mathtt{e}$, method calls $\mathtt{mn}(\mathtt{x}^*)$, variable assignments $\mathtt{x}{:=}\mathtt{e}$, expression sequences $\mathtt{e}_1; \mathtt{e}_2$, binary operations represented by $\mathtt{e}_1\ \mathit{op}\ \mathtt{e}_2$, including $+$, $-$, $==$, $<$, etc, event raises expression $\textbf{event}[\underline{\mathtt{a}}]$, conditional expressions $\textbf{if}\ \mathtt{v}\ \textbf{then}\ \mathtt{e}_1\ \textbf{else}\ \mathtt{e}_2$, and the assertion constructor **assert**, parametrized with effects Φ.

3.2 The Specification Language. We plant the effects specifications into the Hoare-style verification system. We use $\{\textbf{requires}\ \Phi_{\mathtt{pre}}\ \textbf{ensures}\ \Phi_{\mathtt{post}}\}$ to capture the precondition $\Phi_{\mathtt{pre}}$ and the postcondition $\Phi_{\mathtt{post}}$, defined in Fig. 3.

$$
\begin{array}{rrcl}
(\textit{Effects}) & \Phi & ::= & \pi \wedge \mathtt{es} \mid \Phi_1 \vee \Phi_2 \mid \exists \mathtt{x}.\Phi \\
(\textit{Event Seq.}) & \mathtt{es} & ::= & \bot \mid \epsilon \mid \_ \mid \underline{\mathtt{a}} \mid \mathtt{es}_1 \cdot \mathtt{es}_2 \mid \mathtt{es}_1 \vee \mathtt{es}_2 \mid \mathtt{es}_1 \wedge \mathtt{es}_2 \mid \neg \mathtt{es} \mid \mathtt{es}^{\mathtt{t}} \mid \mathtt{es}^{\star} \mid \mathtt{es}^{\omega} \\
(\textit{Pure}) & \pi & ::= & \mathtt{True} \mid \mathtt{False} \mid \mathtt{A}(\mathtt{t}_1, \mathtt{t}_2) \mid \pi_1 \wedge \pi_2 \mid \pi_1 \vee \pi_2 \mid \neg\pi \mid \pi_1 \Rightarrow \pi_2 \mid \forall \mathtt{x}.\pi \mid \exists \mathtt{x}.\pi \\
(\textit{Terms}) & \mathtt{t} & ::= & \mathtt{n} \mid \mathtt{x} \mid \mathtt{t}_1 + \mathtt{t}_2 \mid \mathtt{t}_1 - \mathtt{t}_2
\end{array}
$$

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

Fig. 3. Syntax of Effects.

Effects can be a conditioned event sequence $\pi \wedge es$ or a disjunction of two effects $\Phi_1 \vee \Phi_2$, or an effect Φ existentially quantified over a variable $\mathtt{x}$. Event sequences comprise *false* ($\bot$); an empty trace ϵ; the wild card $\_$ representing any single event; a single event $\underline{\mathbf{a}}$; sequences concatenation $\mathtt{es}_1 \cdot \mathtt{es}_2$; disjunction $\mathtt{es}_1 \vee \mathtt{es}_2$; conjunction $\mathtt{es}_1 \wedge \mathtt{es}_2$; negation $\neg\mathtt{es}$; $\mathtt{t}$ times repetition of a trace, $\mathtt{es}^{\mathtt{t}}$, where $\mathtt{t}$ is a *term*; Kleene star, zero or many times (possibly infinite) repetition of a trace; and the infinite repetition of a trace, $\mathtt{es}^{\omega}$. However, for now, we restrict the nested usage of operators among $\neg$, $\mathtt{t}$, $\star$ and ω.

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

3.3 Semantic Model of Effects. To define the model, $\mathtt{var}$ is the set of program variables, $\mathtt{val}$ is the set of primitive values, $\mathtt{es}$ is the set of event sequences (or event multi-trees, per se), indicating the sequencing constraints on temporal behaviour. Let $\mathtt{s}, \varphi \models \Phi$ denote the model relation, i.e., the stack $\mathtt{s}$ and linear temporal events φ satisfy the temporal effects Φ, with $\mathtt{s}$, φ from the following concrete domains: $\mathtt{s} \triangleq \mathtt{var} \rightarrow \mathtt{val}$ and $\varphi \triangleq \mathtt{es}$.

As shown in Fig. 4., we define the semantics of our effects. We use ++ to represent the append operation of two event sequences. We use [] to represent the empty sequence, [$\underline{\mathbf{a}}$] to represent the sequence only contains one element $\underline{\mathbf{a}}$.

$$
\begin{array}{lll}
\mathtt{s}, \varphi \models \Phi_1 \vee \Phi_2 & \textit{iff} & \mathtt{s}, \varphi \models \Phi_1 \textit{ or } \mathtt{s}, \varphi \models \Phi_2 \\
\mathtt{s}, \varphi \models \exists \mathtt{x}.\Phi & \textit{iff} & (\exists v \in \mathit{val}).s[x{\rightarrow}v], \varphi \models \Phi \\
\mathtt{s}, \varphi \models \pi \wedge \epsilon & \textit{iff} & [\![\pi]\!]_{\mathtt{s}}{=}\mathtt{True} \textit{ and } \varphi{=}[] \\
\mathtt{s}, \varphi \models \pi \wedge \_ & \textit{iff} & [\![\pi]\!]_{\mathtt{s}}{=}\mathtt{True}, \varphi \in \{[\underline{\mathbf{a}}] \mid \underline{\mathbf{a}} \in \Sigma\} \\
\mathtt{s}, \varphi \models \pi \wedge \underline{\mathbf{a}} & \textit{iff} & [\![\pi]\!]_{\mathtt{s}}{=}\mathtt{True} \textit{ and } \varphi{=}[\underline{\mathbf{a}}] \\
\mathtt{s}, \varphi \models \pi \wedge (\mathtt{es}_1 \cdot \mathtt{es}_2) & \textit{iff} & \textit{there exist } \varphi_1, \varphi_2 \textit{ and } \varphi_1{+}{+}\varphi_2 = \varphi \\
 & & \quad \textit{and } \mathtt{s}, \varphi_1 \models \pi \wedge \mathtt{es}_1 \textit{ and } \mathtt{s}, \varphi_2 \models \pi \wedge \mathtt{es}_2 \\
\mathtt{s}, \varphi \models \pi \wedge (\mathtt{es}_1 \vee \mathtt{es}_2) & \textit{iff} & \mathtt{s}, \varphi \models \pi \wedge \mathtt{es}_1 \textit{ or } \mathtt{s}, \varphi \models \pi \wedge \mathtt{es}_2 \\
\mathtt{s}, \varphi \models \pi \wedge (\mathtt{es}_1 \wedge \mathtt{es}_2) & \textit{iff} & \mathtt{s}, \varphi \models \pi \wedge \mathtt{es}_1 \textit{ and } \mathtt{s}, \varphi \models \pi \wedge \mathtt{es}_2 \\
\mathtt{s}, \varphi \models \pi \wedge \neg\mathtt{es} & \textit{iff} & \mathtt{s}, \varphi \not\models \pi \wedge \mathtt{es} \\
\mathtt{s}, \varphi \models \pi \wedge \mathtt{es}^{\mathtt{t}} & \textit{iff} & [\![\pi \wedge \mathtt{t}{=}0]\!]_{\mathtt{s}}{=}\mathtt{True},\ \mathtt{s}, \varphi \models \pi \wedge \epsilon \textit{ or} \\
 & & [\![\pi \wedge \mathtt{t}{>}0]\!]_{\mathtt{s}}{=}\mathtt{True},\ \textit{there exist } \varphi_1, \varphi_2 \\
 & & \quad \textit{and } \varphi_1{+}{+}\varphi_2{=}\varphi \textit{ and } \mathtt{s}, \varphi_1 \models \pi \wedge \mathtt{es} \\
 & & \quad \textit{and } \mathtt{s}, \varphi_2 \models (\pi \wedge \mathtt{t}{>}0) \wedge \mathtt{es}^{\mathtt{t}\text{-}1} \\
\mathtt{s}, \varphi \models \pi \wedge \mathtt{es}^{\star} & \textit{iff} & \mathtt{s}, \varphi \models \exists \mathtt{x}.(\pi \wedge \mathtt{es}^{\mathtt{x}}) \textit{ or } \mathtt{s}, \varphi \models \pi \wedge \mathtt{es}^{\omega} \\
\mathtt{s}, \varphi \models \pi \wedge \mathtt{es}^{\omega} & \textit{iff} & \textit{there exist } \varphi_1, \varphi_2 \textit{ and } \varphi_1{+}{+}\varphi_2{=}\varphi \\
 & & \quad \textit{and } \mathtt{s}, \varphi_1 \models \pi \wedge \mathtt{es} \textit{ and } \mathtt{s}, \varphi_2 \models \pi \wedge \mathtt{es}^{\omega} \\
\mathtt{s}, \varphi \models \mathtt{false} & \textit{iff} & [\![\pi]\!]_{\mathtt{s}}{=}\mathtt{False} \textit{ or } \varphi{=}\bot
\end{array}
$$

Fig. 4. Semantics of Effects.

4 Automated Verification

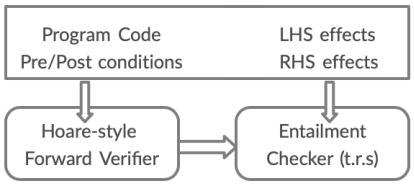

Fig. 5. Overview of Verification.

An overview of our automated verification system is given in Fig. 5.. It consists of a standard Hoare-style forward verifier (the front-end) and a T.r.s (the back-end). In this section, we mainly present the forward verifier, which invokes the back-end, by introducing a set of forward verification rules. Note that we allow 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 our verification system. The inclusion checking process will be explained in Sec. 5.

$$\frac{\Phi'_{\mathtt{C}}{=}\Phi_{\mathtt{C}} \cdot \underline{\mathtt{a}}}{\vdash \{\Phi_{\mathtt{C}}\}\ \mathbf{event}[\underline{\mathtt{a}}]\ \{\Phi'_{\mathtt{C}}\}}\ [\texttt{FV-Event}] \qquad \frac{\vdash \{\Phi_{\mathtt{C}}\}\ \mathtt{e}_1\ \{\Phi'_{\mathtt{C}}\} \qquad \vdash \{\Phi'_{\mathtt{C}}\}\ \mathtt{e}_2\ \{\Phi''_{\mathtt{C}}\}}{\vdash \{\Phi_{\mathtt{C}}\}\ \mathtt{e}_1; \mathtt{e}_2\ \{\Phi''_{\mathtt{C}}\}}\ [\texttt{FV-Seq}]$$

$$\frac{\vdash \{\mathtt{v} \wedge \Phi_{\mathtt{C}}\}\ \mathtt{e}_1\ \{\Phi'_{\mathtt{C}}\} \qquad \vdash \{\neg \mathtt{v} \wedge \Phi_{\mathtt{C}}\}\ \mathtt{e}_2\ \{\Phi''_{\mathtt{C}}\}}{\vdash \{\Phi_{\mathtt{C}}\}\ \mathbf{if}\ \mathtt{v}\ \mathbf{then}\ \mathtt{e}_1\ \mathbf{else}\ \mathtt{e}_2\ \{\Phi'_{\mathtt{C}} \vee \Phi''_{\mathtt{C}}\}}\ [\texttt{FV-If-Else}]$$

$$\frac{\vdash \{\Phi_{\mathtt{C}}\}\ \mathtt{e}\ \{\Phi'_{\mathtt{C}}\}}{\vdash \{\Phi_{\mathtt{C}}\}\ \tau\ \mathtt{x};\ \mathtt{e}\ \{\exists \mathtt{x}.\Phi'_{\mathtt{C}}\}}\ [\texttt{FV-Local}] \qquad \frac{\vdash \mathtt{rev}(\Phi_{\mathtt{C}}) \sqsubseteq \mathtt{rev}(\Phi) \rightsquigarrow \gamma_{\mathtt{R}}}{\vdash \{\Phi_{\mathtt{C}}\}\ \mathbf{assert}\ \Phi\ \{\Phi_{\mathtt{C}}\}}\ [\texttt{FV-Assert}]$$

$$\frac{\begin{array}{c}\tau\ \mathtt{mn}\ (\tau\ \mathtt{x})^*\ \{\mathbf{requires}\ \Phi_{\mathtt{pre}}\ \mathbf{ensures}\ \Phi_{\mathtt{post}}\}\ \{\mathtt{e}\} \in \mathcal{P} \\ \vdash \mathtt{rev}(\Phi_{\mathtt{C}}) \sqsubseteq \mathtt{rev}([\mathtt{y}^*/\mathtt{x}^*]\Phi_{\mathtt{pre}}) \rightsquigarrow \gamma_{\mathtt{R}} \qquad \Phi'_{\mathtt{C}} = \Phi_{\mathtt{C}} \cdot [\mathtt{y}^*/\mathtt{x}^*]\Phi_{\mathtt{post}}\end{array}}{\vdash \{\Phi_{\mathtt{C}}\}\ \mathtt{mn}(\mathtt{y}^*)\ \{\Phi'_{\mathtt{C}}\}}\ [\texttt{FV-Call}]$$

$$\frac{\vdash \{\epsilon\}\ \mathtt{e}\ \{\Phi_{\mathtt{C}}\} \qquad \vdash \Phi_{\mathtt{C}} \sqsubseteq \Phi_{\mathtt{post}}}{\vdash \tau\ \mathtt{mn}\ (\tau\ \mathtt{x})^*\ \{\mathbf{requires}\ \Phi_{\mathtt{pre}}\ \mathbf{ensures}\ \Phi_{\mathtt{post}}\}\ \{\mathtt{e}\}}\ [\texttt{FV-Meth}]$$

Fig. 6. Some Forward Verification Rules.

We present some of the forward verification rules in Fig. 6., which are used to systematically accumulate the effects based on the syntax of each statement. We use $\mathcal{P}$ to denote the program being checked. With pre/post conditions declared for each method in $\mathcal{P}$, we can apply modular verification to a method's body using Hoare-style triples $\vdash \{\Phi_{\mathtt{C}}\}\ \mathtt{e}\ \{\Phi'_{\mathtt{C}}\}$, where $\Phi_{\mathtt{C}}$ is the current effects and $\Phi'_{\mathtt{C}}$ is the resulting effects by executing $\mathtt{e}$. In [FV-If-Else], $(\mathtt{v} \wedge \Phi_{\mathtt{C}})$ enforces $\mathtt{v}$ into the pure constraints of every traces in $\Phi_{\mathtt{C}}$, same for $(\neg \mathtt{v} \wedge \Phi_{\mathtt{C}})$. In [FV-Call], we check whether the instantiated precondition of callee, $[\mathtt{y}^*/\mathtt{x}^*]\Phi_{\mathtt{pre}}$, is satisfied by the *tail* [2] of current effects state, in which we use an auxiliary function `rev` to reverse the event sequences of effects. Then we obtain the next effects state by concatenating the instantiated postcondition, $[\mathtt{y}^*/\mathtt{x}^*]\Phi_{\mathtt{post}}$, to the current effects state. (cf. step 6) in Fig. 1.) In [FV-Meth], we initialize the current effects state using ϵ, accumulate the effects from the method body, to obtain $\Phi_{\mathtt{C}}$, and check inclusion between $\Phi_{\mathtt{C}}$ and the declared specifications $\Phi_{\mathtt{post}}$[3].

[2] We check the inclusion between the reversed current effects and precondition effects, meaning that, before calling a method, its required effects *has just happened*.

[3] $\Phi_{\mathtt{post}}$ only needs to capture the effects from the current method body, excluding the history effects specified in $\Phi_{\mathtt{pre}}$.

5 Effects Inclusion Checker (the T.r.s)

The effects inclusion checking (an extension of the T.r.s proposed from [5]) 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 Sec. 4, our forward verification generates effects inclusions of the form: $\Gamma \vdash \Phi_1 \sqsubseteq^{\Phi}_{\mathtt{V}} \Phi_2 \rightsquigarrow \gamma_{\mathtt{R}}$, a shorthand for: $\Gamma \vdash \Phi \cdot \Phi_1 \sqsubseteq \exists \mathtt{V}.\ (\Phi \cdot \Phi_2) \rightsquigarrow \gamma_{\mathtt{R}}$.

To prove such effects inclusions is to check whether all the possible event traces in the antecedent Φ_1 are legitimately allowed in the possible event traces from the consequent Φ_2, and (in case there are) to compute a residual effects $\gamma_{\mathtt{R}}$ (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. Γ is the proof context, i.e. a set of effects inclusions, Φ is the history of effects from the antecedent that have been used to match the effects from the consequent, and $\mathtt{V}$ is the set of existentially quantified variables from the consequent. Note that Γ, Φ and $\mathtt{V}$ are derived during the inclusion proof. The inclusion checking procedure is initially invoked with $\Gamma{=}\emptyset$, $\Phi{=}\mathtt{True} \wedge \epsilon$ and $\mathtt{V}{=}\emptyset$. We now briefly discuss the key steps and related inference rules that we may use in such an effects inclusion proof. Firstly, we present the reduction to eliminate the disjunctions from the antecedents and existential quantifiers. *(LHS refers to left-hand side, and RHS refers to right-hand side.)*

I. Effect Disjunction. An inclusion with a disjunctive antecedent succeeds if both disjunctions entail the consequent.

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

II. 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, we keep track of the existential variables coming from the consequent by adding them to $\mathtt{V}$. *($\mathtt{u}$ is a fresh variable)*

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

Table 3. Some Normalization Lemmas for effects constructed by $\pi \wedge \mathtt{es}$.

$\mathtt{es} \vee \mathtt{es} \rightarrow \mathtt{es}$	$\epsilon^{\omega} \rightarrow \epsilon$	$(\mathtt{es}_1 \cdot \mathtt{es}_2) \cdot \mathtt{es}_3 \rightarrow \mathtt{es}_1 \cdot (\mathtt{es}_2 \cdot \mathtt{es}_3)$
$\bot \vee \mathtt{es} \rightarrow \mathtt{es}$	$\mathtt{es} \wedge \mathtt{es} \rightarrow \mathtt{es}$	$(\mathtt{es}_1 \vee \mathtt{es}_2) \cdot \mathtt{es}_3 \rightarrow \mathtt{es}_1 \cdot \mathtt{es}_3 \vee \mathtt{es}_2 \cdot \mathtt{es}_3$
$\mathtt{es} \vee \bot \rightarrow \mathtt{es}$	$\mathtt{es} \wedge \bot \rightarrow \bot$	$\mathtt{es}_1 \cdot (\mathtt{es}_2 \vee \mathtt{es}_3) \rightarrow \mathtt{es}_1 \cdot \mathtt{es}_2 \vee \mathtt{es}_1 \cdot \mathtt{es}_3$
$\epsilon \cdot \mathtt{es} \rightarrow \mathtt{es}$	$\bot^{\omega} \rightarrow \bot$	$\mathtt{es}^{\omega} \cdot \mathtt{es}_1 \rightarrow \mathtt{es}^{\omega}$
$\mathtt{es} \cdot \epsilon \rightarrow \mathtt{es}$	$\epsilon^{\mathtt{t}} \rightarrow \epsilon$	$\mathtt{False} \wedge \mathtt{es} \rightarrow \mathtt{False} \wedge \bot$
$\bot \cdot \mathtt{es} \rightarrow \bot$	$\mathtt{t}{=}0 \wedge \mathtt{es}^{\mathtt{t}} \rightarrow \epsilon$	$\mathtt{es} \wedge \epsilon \rightarrow \bot \quad (\delta_{\pi}(\mathtt{es}){=}\mathtt{false})$
$\mathtt{es} \cdot \bot \rightarrow \bot$	$\bot^{\mathtt{t}} \rightarrow \bot$	$\mathtt{es} \wedge \epsilon \rightarrow \epsilon \quad (\delta_{\pi}(\mathtt{es}){=}\mathtt{true})$

III. Normalization. The rewriting of an inclusion between two quantifier-free effects starts with a general normalization for both the antecedent and the consequent. We assume that the effects formulae are tailored accordingly using the lemmas in Table 3., which are extended from the normalization rules suggested by Antimirov and Mosses, being able to further normalize our dependent effects.

IV. Substitution. In order to guarantee the termination, for both the antecedent and the consequent, a term $\mathtt{t}_1 \oplus \mathtt{t}_2$ will be substituted with a fresh variable $\mathtt{u}$ constrained with $\mathtt{u}{=}\mathtt{t}_1 \oplus \mathtt{t}_2 \wedge \mathtt{u}{\geq}0$, where $\oplus \in \{+, -\}$. (cf. Table 2.-II)

$$\frac{\pi'{=}(\mathtt{u}{=}\mathtt{t}_1 \oplus \mathtt{t}_2 \wedge \mathtt{u}{\geq}0) \quad \Gamma \vdash (\pi_1 \wedge \pi') \wedge \mathtt{es}_1^{\mathtt{u}} \cdot \mathtt{es} \sqsubseteq (\pi_2 \wedge \pi') \wedge \mathtt{es}_2 \rightsquigarrow \gamma_{\mathtt{R}}}{\Gamma \vdash \pi_1 \wedge (\mathtt{es}_1{}^{\mathtt{t}_1 \oplus \mathtt{t}_2} \cdot \mathtt{es}) \sqsubseteq \pi_2 \wedge \mathtt{es}_2 \rightsquigarrow \gamma_{\mathtt{R}}} \ [\mathtt{LHS\text{-}SUB}]$$

$$\frac{\pi'{=}(\mathtt{u}{=}\mathtt{t}_1 \oplus \mathtt{t}_2 \wedge \mathtt{u}{\geq}0) \quad \Gamma \vdash (\pi_1 \wedge \pi') \wedge \mathtt{es}_1 \sqsubseteq (\pi_2 \wedge \pi') \wedge \mathtt{es}_2^{\mathtt{u}} \cdot \mathtt{es} \rightsquigarrow \gamma_{\mathtt{R}}}{\Gamma \vdash \pi_1 \wedge \mathtt{es}_1 \sqsubseteq \pi_2 \wedge (\mathtt{es}_2{}^{\mathtt{t}_1 \oplus \mathtt{t}_2} \cdot \mathtt{es}) \rightsquigarrow \gamma_{\mathtt{R}}} \ [\mathtt{RHS\text{-}SUB}]$$

V. Case Split. Based on the semantics of the symbolic integer t, whenever it is possibly zero, we conduct a case split,

to distinguish the zero (base) case, leads to an empty trace; and the non-zero (inductive) case. (cf. Table 2.-II)

$$\frac{\Gamma \vdash ((\pi_1 \wedge \mathtt{t}{=}0) \wedge \mathtt{es}) \vee ((\pi_1 \wedge \mathtt{t} > 0) \wedge \mathtt{es}_1 \cdot \mathtt{es}_1^{\mathtt{t\text{-}1}} \cdot \mathtt{es}) \sqsubseteq \pi_2 \wedge \mathtt{es}_2 \rightsquigarrow \gamma_{\mathtt{R}}}{\Gamma \vdash \pi_1 \wedge (\mathtt{es}_1^{\mathtt{t}} \cdot \mathtt{es}) \sqsubseteq \pi_2 \wedge \mathtt{es}_2 \rightsquigarrow \gamma_{\mathtt{R}}} \ [\mathtt{LHS\text{-}CASESPLIT}]$$

$$\frac{\Gamma \vdash \pi_1 \wedge \mathtt{es}_1 \sqsubseteq ((\pi_2 \wedge \mathtt{t}{=}0) \wedge \mathtt{es}) \vee ((\pi_2 \wedge \mathtt{t} > 0) \wedge \mathtt{es}_2 \cdot \mathtt{es}_2^{\mathtt{t\text{-}1}} \cdot \mathtt{es}) \rightsquigarrow \gamma_{\mathtt{R}}}{\Gamma \vdash \pi_1 \wedge \mathtt{es}_1 \sqsubseteq \pi_2 \wedge (\mathtt{es}_2^{\mathtt{t}} \cdot \mathtt{es}) \rightsquigarrow \gamma_{\mathtt{R}}} \ [\mathtt{RHS\text{-}CASESPLIT}]$$

VI. Unfolding (Induction). Here comes the key inductive step of unfolding the inclusion. Firstly, we make use of the `fst` auxiliary function to get a set of events $\mathtt{F}$, which are all the possibly *first* event from the antecedent. Secondly, we obtain a new proof context Γ' by adding the current inclusion, as an inductive hypothesis, into the current proof context Γ. Thirdly, we iterate each element $\underline{\mathtt{a}}$ ($\underline{\mathtt{a}} \in \mathtt{F}$), and compute the partial derivatives (the *next-state* effects) of both the antecedent and consequent w.r.t $\underline{\mathtt{a}}$. The proof of the original inclusion succeeds if all the derivative inclusions succeeds.

$$\frac{\mathtt{F} = \mathtt{fst}_{\pi_1}(\mathtt{es}_1) \qquad \Gamma' = \Gamma,\ (\pi_1 \wedge \mathtt{es}_1 \sqsubseteq \pi_2 \wedge \mathtt{es}_2) \qquad \forall \underline{\mathtt{a}} \in \mathtt{F}.\ (\Gamma' \vdash\ \mathtt{D}_{\underline{\mathtt{a}}}^{\pi_1}(\mathtt{es}_1) \sqsubseteq \mathtt{D}_{\underline{\mathtt{a}}}^{\pi_2}(\mathtt{es}_2))}{\Gamma \vdash \pi_1 \wedge \mathtt{es}_1 \sqsubseteq \pi_2 \wedge \mathtt{es}_2} \ [\mathtt{UNFOLD}]$$

Next we provide the definitions and the key implementations[4] of Nullable, First and Derivative respectively. Intuitively, the Nullable function $\delta_\pi(\mathtt{es})$ returns a

[4] As the implementations according to basic regular expressions can be found in prior work [12]. Here, we focus on presenting the definitions and how do we deal with dependent values in the effects, as the key novelties of this work.

boolean value indicating whether $\pi \wedge \mathtt{es}$ contains the empty trace; the First function $\mathtt{fst}_\pi(\mathtt{es})$ computes a set of possible initial events of $\pi \wedge \mathtt{es}$; and the Derivative function $\mathtt{D}^\pi_{\underline{\mathtt{a}}}(\mathtt{es})$ computes a next-state effects after eliminating one event $\underline{\mathtt{a}}$ from the current effects $\pi \wedge \mathtt{es}$.

Definition 1 (Nullable). *Given any event sequence* $\mathtt{es}$ *under condition* π*, we define* $\delta_\pi(\mathtt{es})$ *to be:*

$$\delta_\pi(\mathtt{es}) : \mathtt{bool} = \begin{cases} \mathtt{true} & \mathtt{if}\ \epsilon \in [\![\pi \wedge \mathtt{es}_1]\!]_\varphi \\ \mathtt{false} & \mathtt{if}\ \epsilon \notin [\![\pi \wedge \mathtt{es}_1]\!]_\varphi \end{cases}, \text{ where } \delta_\pi(\mathtt{es}^{\mathtt{t}}) = \mathtt{SAT}(\pi \wedge (\mathtt{t}{=}0))^5$$

Definition 2 (First). *Let* $\mathtt{fst}_\pi(\mathtt{es}) := \{\underline{\mathtt{a}} \mid \underline{\mathtt{a}} \cdot \mathtt{es}' \in [\![\pi \wedge \mathtt{es}]\!]\}$ *be the set of initial events derivable from event sequence* $\mathtt{es}$ *w.r.t. the condition* π*.*

$$\mathtt{fst}_\pi(\mathtt{es}_1 \cdot \mathtt{es}_2) = \begin{cases} \mathtt{fst}_\pi(\mathtt{es}_1) \cup \mathtt{fst}_\pi(\mathtt{es}_2) & \mathtt{if}\ \delta_\pi(\mathtt{es}_1){=}\mathtt{true} \\ \mathtt{fst}_\pi(\mathtt{es}_1) & \mathtt{if}\ \delta_\pi(\mathtt{es}_1){=}\mathtt{false} \end{cases}$$

Definition 3 (Derivative). *The derivative* $\mathtt{D}^\pi_{\underline{\mathtt{a}}}(\mathtt{es})$ *of an event sequence* $\mathtt{es}$ *w.r.t. an event* $\underline{\mathtt{a}}$ *and the condition* π *computes the effects for the left quotient* $\underline{\mathtt{a}}^{-1}[\![\pi \wedge \mathtt{es}]\!]$*, where we define* $\mathtt{D}^\pi_{\underline{\mathtt{a}}}(\mathtt{es}^{\mathtt{t}}) = \mathtt{D}^{\pi \wedge \mathtt{t}>0}_{\underline{\mathtt{a}}}(\mathtt{es}) \cdot \mathtt{es}^{\mathtt{t}\text{-}1}$*.*

$$\mathtt{D}^\pi_{\underline{\mathtt{a}}}(\mathtt{es}_1 \cdot \mathtt{es}_2) = \begin{cases} \mathtt{D}^\pi_{\underline{\mathtt{a}}}(\mathtt{es}_1) \cdot \mathtt{es}_2 \vee \mathtt{D}^\pi_{\underline{\mathtt{a}}}(\mathtt{es}_2) & \mathtt{if}\ \delta_\pi(\mathtt{es}_1){=}\mathtt{true} \\ \mathtt{D}^\pi_{\underline{\mathtt{a}}}(\mathtt{es}_1) \cdot \mathtt{es}_2 & \mathtt{if}\ \ \delta_\pi(\mathtt{es}_1){=}\mathtt{false} \end{cases}$$

VII. 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.

$$\frac{\delta_{\pi_1}(\mathtt{es}_1) \wedge \neg\delta_{\pi_1 \wedge \pi_2}(\mathtt{es}_2)}{\Gamma \vdash \pi_1 \wedge \mathtt{es}_1 \not\sqsubseteq \pi_2 \wedge \mathtt{es}_2}\ [\mathtt{DISPROVE}]$$

VIII. Prove. We use 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, we prove this inclusion with a residue $\gamma_{\mathtt{R}}$[6]; and (iii) [REOCCUR] is used when there exists an inclusion hypothesis in the proof context Γ, which meets the conditions. It essentially assigns to the current unexpanded inclusion an interior inclusion with an identical sequent labelling.

$$\frac{\pi_1 \Rightarrow \pi_2 \quad \mathtt{es}_1 \subseteq \mathtt{es}_2}{\Gamma \vdash \pi_1 \wedge \mathtt{es}_1 \sqsubseteq \pi_2 \wedge \mathtt{es}_2}\ [\mathtt{PROVE}] \qquad \frac{\pi_1 \Rightarrow \pi_2 \quad \gamma_{\mathtt{R}} {=} \pi_1 \wedge \mathtt{es}_1}{\Gamma \vdash (\pi_1 \wedge \mathtt{es}_1 \sqsubseteq \pi_2 \wedge \epsilon) \rightsquigarrow \gamma_{\mathtt{R}}}\ [\mathtt{FRAME}]$$

$$\frac{\exists.(\pi_1' \wedge \mathtt{es}_1' \sqsubseteq \pi_2' \wedge \mathtt{es}_2') \in \Gamma \quad \pi_1 {\Rightarrow} \pi_1' {\Rightarrow} \pi_2' {\Rightarrow} \pi_2 \quad \mathtt{es}_1 {\subseteq} \mathtt{es}_1' \quad \mathtt{es}_2' {\subseteq} \mathtt{es}_2}{\Gamma \vdash \pi_1 \wedge \mathtt{es}_1 \sqsubseteq \pi_2 \wedge \mathtt{es}_2}\ [\mathtt{REOCCUR}]$$

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

[6] 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.

6 Implementation and Evaluation

To show the feasibility of our approach, we have implemented our effects logic using OCaml, on top of the HIP/SLEEK system[8]. The proof obligations generated by our verification are discharged using constraint solver Z3. Furthermore, we provide a web UI[2] to present more non-trivial examples. Next, we show case studies to demonstrate the expressive power of our integrated dependent effects.

6.1 Case Studies.

i. Encoding LTL. Classical LTL extended propositional logic with the temporal operators $\mathcal{G}$ ("globally") and $\mathcal{F}$ ("in the future"), which we also write $\Box$ and $\Diamond$, respectively; and introduced the concept of fairness, which ensures an infinite-paths semantics. LTL was subsequently extended to include the $\mathcal{U}$ ("until") operator and the $\mathcal{X}$ ("next time") operator. As shown in Fig. 4., we encode these basic operators into our effects, making it more intuitive and readable, mainly when nested operators occur. Furthermore, by putting the effects in the precondition, our approach naturally composites *past-time LTL* along the way.

Table 4. Examples for converting LTL formulae into Effects. ($\underline{\mathbf{A}}, \underline{\mathbf{B}}$ are events, $\mathtt{n} \geq 0$, $\mathtt{m} \geq 0$ are the default constraints.)

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

ii. Encoding μ-calculus. μ-calculus provides a single, elegant, uniform logical framework of great raw expressive power by using a least fixpoint (μ) and a greatest fixpoint (υ). More specifically, it can express properties such as $\upsilon\mathtt{Z}.\mathtt{P} \wedge \mathcal{X}\mathcal{X}\mathtt{Z}$, 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 we can see, such properties already go beyond the first order logic. In fact, analogously to our effects, the symbolic/constant values correspond to the least fixpoint (μ), referring to finite traces, and the constructor ω corresponds to the greatest fixpoint (υ), referring to infinite traces. For example, we write $(\_ \cdot \underline{\mathbf{A}})^{\omega}$, meaning that the event $\underline{\mathbf{A}}$ recurs at every even position in an infinite trace.

iii. Kleene Star. By using $\star$, we make an approximation of the possible traces when the termination is non-deterministic. As shown in Fig. 7., a weaker specification of send(n) can be provided as $\underline{\mathbf{Send}}^{\star} \cdot \underline{\mathbf{Done}}$, meaning that the repetition of event $\underline{\mathbf{Send}}$ can be both finite and infinite, which is more concise than the prior work, also beyond μ-calculus. By supporting a variety of specifications, we can make a trade-off between precision and scalability, which is important for realistic methodology on automated verification. For example, we

```
1 void send (int n){
2    if (...) {
3       event[Done];
4    }else{
5       event[Send];
6       send (n-1);
7    }}
```

Fig. 7. An unknown conditional

can weaken precondition of $\texttt{server(n)}$ (cf. Table 1.) to $\Phi_{\texttt{pre}}^{\texttt{server(n)}} \triangleq \texttt{True} \wedge \epsilon$, and opt for either of the following two postcondition:$\Phi_{\texttt{post1}}^{\texttt{server(n)}} \triangleq \texttt{n}{\geq}0 \wedge (\underline{\textbf{Ready}} \cdot \underline{\textbf{Send}}^{\texttt{n}} \cdot \underline{\textbf{Done}})^{\omega}$, or $\Phi_{\texttt{post2}}^{\texttt{server(n)}} \triangleq \texttt{n}{\geq}0 \wedge (\underline{\textbf{Ready}} \cdot \underline{\textbf{Send}}^{\texttt{n}} \cdot \underline{\textbf{Done}})^{\omega} \vee \texttt{n}{<}0 \wedge \underline{\textbf{Ready}} \cdot \underline{\textbf{Send}}^{\omega}$, with the latter being more complex but more precise.

iv. Beyond Regular, Context-Free and Context-Sensitive The paradigmatic non-regular linear language: $\texttt{n}{>}0 \wedge \underline{\textbf{a}}^{\texttt{n}} \cdot \underline{\textbf{b}}^{\texttt{n}}$, can be naturally expressed by the depended effects. Besides, the effects can also express grammars such as $\texttt{n}{>}0 \wedge \underline{\textbf{a}}^{\texttt{n}} \cdot \underline{\textbf{b}}^{\texttt{n}} \cdot \underline{\textbf{c}}^{\texttt{n}}$, or $\texttt{n}{>}0 \wedge \texttt{m}{>}0 \wedge \underline{\textbf{a}}^{\texttt{n}} \cdot \underline{\textbf{b}}^{\texttt{m}} \cdot \underline{\textbf{c}}^{\texttt{n}}$, 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 our effects. However, such specifications are significant, suppose we have a traffic light control system, we could have a specifications $\texttt{n}{>}0 \wedge \texttt{m}{>}0 \wedge (\underline{\textbf{Red}}^{\texttt{n}} \cdot \underline{\textbf{Yellow}}^{\texttt{m}} \cdot \underline{\textbf{Green}}^{\texttt{n}})^{\omega}$, 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.

6.2 Experimental Results. We mainly compare our backend T.r.s with the mature model checker PAT[18], which implements techniques for LTL properties with fairness assumptions. We chose a realistic benchmark containing 16 IOT programs implemented in C for Arduino controlling programs[1]. For each of the programs, we (i) derive 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 our effects, (iii) we record the total computation time using PAT and our T.r.s. Our test cases are provided as a benchmark[2]. We conduct experiments on a MacBook Pro with a 2.6 GHz Intel Core i7 processor.

As shown in Table 5., comparing the T.r.s to PAT, the total (dis-) proving time has been reduced by 31.7%. For that, we summarize 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 T.r.s is even less than the NFAs construction time, which means it is not necessary to construct the NFAs when a T.r.s solves it faster; (2)When the total states become larger, on average, the T.r.s outperforms automata-based algorithms, due to the significantly reduced search branches provided by the normalization lemmas; and (3)For the invalid cases, the T.r.s disproves them earlier without constructing the whole NFAs.

7 Related Work

Recently, temporal reasoning has garnered renewed importance for possibly non-terminating control programs with subtle use of recursion and non-determinism, as used in reactive or stream-based applications. In this section, we discuss the related works in the following two perspectives: (i) temporal verification and expressive effects; and (ii) efficient algorithms for language inclusion checking.

Table 5. The experiments are based on 16 real world C programs, we record the lines of code (LOC), the number of testing temporal properties (#Prop.), and the (dis-) proving times (in milliseconds) using PAT and our T.r.s respectively.

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

7.1 Verification and Expressive Effects. 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 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 modelling stage and need to be bounded when encountering non-terminating traces.

Meanwhile, 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 combined together to reason about the overall program[10,13,17]. However, the temporal effects in prior work tend to coarsely over-approximate the behaviours either via ω-regular expressions[10] or by büchi automata[13]. One of the recent works[17] 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. The conventional effects have the form (Φ_u, Φ_v), which separates the finite and infinite effects. In this work, by integrating possibly finite and possibly infinite effects into a single disjunctive form with size properties, our integrated dependent effects eliminate the finiteness distinction, and enable an expressive modular temporal verification.

7.2 Efficient Algorithms for Language Inclusion Checking. Generally, it is unavoidable for any language inclusion checking solutions to have an exponential worst-case complexity. As there are no existing automata capable to express dependent effects, neither there exist corresponding inclusion checking algorithms. Here we reference two efficient prior works targeting basic regular sets: Antichain-based algorithms and the traditional T.r.s, which are both avoiding the explicit, complex translation from the NFAs into their minimal DFAs.

Antichain-based algorithm[9] 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 T.r.s[6,5,12] is that it provides a reasoning logic for regular expression inclusions to avoid any kind of translation aforementioned. Specifically, a T.r.s takes finite steps to reduce $\mathtt{r} \preceq \mathtt{t}$ into its normal form $\mathtt{r}' \preceq \mathtt{t}'$ and the inclusion checking fails whenever $\mathtt{r}' \preceq \mathtt{t}'$ is not valid. A T.r.s 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 DFAs. More importantly, a T.r.s allows us to accommodate infinite alphabets and capture size-dependent properties.

In this work, we choose to deploy an extended T.r.s, which composites optimizations from both Antichain-based algorithm and classical T.r.s. Having such a T.r.s as the back-end to verify temporal effects, one can benefit from the high efficiency without translating effects into automata. We generalize the Antimirov and Mosses's rewriting procedure[5], to be able to further reason about infinite traces, together with size properties and arithmetic constraints. One of the direct benefits granted by our 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 T.r.s[5,3,12,11].

8 Conclusion

We devise a concise and precise characterization of temporal properties. We propose a novel logic for effects to specify and verify the implementation of the possibly non-terminating programs, including the use of prior effects in preconditions. We implement the effects logic on top of the HIP/SLEEK system[8] and show its feasibility. Our 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.

Acknowledgement. This work is supported by the Academic Research Fund (AcRF) Tier-1 NUS research project R-252-000-A63-114.

References

1. Arduino. `https://create.arduino.cc/projecthub/projects/tags/control`.
2. Online demo platform. `http://loris-5.d2.comp.nus.edu.sg/EffectNew/index.html?ex=send_valid&type=c&options=sess`.
3. M. Almeida, N. Moreira, and R. Reis. Antimirov and mosses's rewrite system revisited. *International Journal of Foundations of Computer Science*, 20(04):669–684, 2009.
4. V. Antimirov. Partial derivatives of regular expressions and finite automata constructions. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 455–466. Springer, 1995.
5. V. M. Antimirov and P. D. Mosses. Rewriting extended regular expressions. *Theoretical Computer Science*, 143(1):51–72, 1995.
6. M. Bezem, J. W. Klop, and R. de Vrijer. Terese. term rewriting systems. *Cambridge Tracts in Theoretical Computer Science*, 55, 2003.
7. J. Brotherston. Cyclic proofs for first-order logic with inductive definitions. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, pages 78–92. Springer, 2005.
8. W.-N. Chin, C. David, H. H. Nguyen, and S. Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Science of Computer Programming*, 77(9):1006–1036, 2012.
9. M. De Wulf, L. Doyen, T. A. Henzinger, and J.-F. Raskin. Antichains: A new algorithm for checking universality of finite automata. In *International Conference on Computer Aided Verification*, pages 17–30. Springer, 2006.
10. M. Hofmann and W. Chen. Abstract interpretation from büchi automata. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, page 51. ACM, 2014.
11. D. Hovland. The inclusion problem for regular expressions. *Journal of Computer and System Sciences*, 78(6):1795–1813, 2012.
12. M. Keil and P. Thiemann. Symbolic solving of extended regular expression inequalities. *arXiv preprint arXiv:1410.3227*, 2014.
13. E. Koskinen and T. Terauchi. Local temporal reasoning. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, page 59. ACM, 2014.
14. G. Malecha, G. Morrisett, and R. Wisnesky. Trace-based verification of imperative programs with i/o. *Journal of Symbolic Computation*, 46(2):95–118, 2011.
15. A. Murase, T. Terauchi, N. Kobayashi, R. Sato, and H. Unno. Temporal verification of higher-order functional programs. In *ACM SIGPLAN Notices*, volume 51, pages 57–68. ACM, 2016.
16. K. Nakata and T. Uustalu. A hoare logic for the coinductive trace-based big-step semantics of while. In *European Symposium on Programming*, pages 488–506. Springer, 2010.
17. Y. Nanjo, H. Unno, E. Koskinen, and T. Terauchi. A fixpoint logic and dependent effects for temporal property verification. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 759–768. ACM, 2018.
18. J. Sun, Y. Liu, J. S. Dong, and J. Pang. Pat: Towards flexible verification under fairness. In *International Conference on Computer Aided Verification*, pages 709–714. Springer, 2009.