Specifying and Verifying Future Conditions

Yahui Song*[0000−0002−9760−5895] Darius Foo[0000−0002−3279−5827]
Wei-Ngan Chin[0000−0002−9660−5682]

School of Computing, National University of Singapore, Singapore
yahui.song@u.nus.edu, dariusf@u.nus.edu, chinwn@nus.edu.sg

Abstract. This paper formalizes *future conditions*, which complement traditional pre- and post-conditions to provide a more comprehensive specification of each function's *behaviour* and *expectation*. Pre-conditions govern the required states before each function call, while post-conditions define the immediate outcomes (post-states) upon completion. Future conditions extend this paradigm by specifying expected temporal behaviors and states that manifest after the function call has finished, potentially affecting subsequent operations or program states. Together, these three types of conditions form a robust specification mechanism for reasoning about API behaviors across various temporal contexts. However, existing techniques for reasoning about future conditions have three key limitations: inefficient entailment checking, under-approximation of program behaviors, and bounded loop unrolling. To address these challenges, we propose a set of over-approximating Hoare-style forward rules that accommodate future conditions that are processed once per method declaration. Moreover, we propose a novel solution for modelling recursive behaviors via a *bag* of future conditions, which can be heuristically synthesized and verified in the verification system. We formally prove the soundness of our proposal in Coq and use experimental results to demonstrate its effectiveness in detecting non-trivial, real-world API misuses.

Keywords: Future Conditions · Hoare Logic · Linear Temporal Logic · Separation Logic · Coq Proof Assistant

1 Introduction

Pre-conditions and post-conditions are fundamental concepts in formal methods of software engineering, particularly in the context of *design by contract* [15]. Considering function definitions as the smallest software components, classic

* Corresponding author

pre- and post-conditions provide constraints for behaviors before the function call and expected outcomes from the current function execution, respectively. However, they are inflexible in expressing constraints on program behavior after the function call has completed. This limitation becomes particularly apparent when modeling complex API behaviors, where the impact of a function call may extend beyond its immediate execution context. For instance, they struggle to capture requirements such as: "Opening a read-only file should not be followed by any writing operations"; "Memory allocated by *malloc* must be finally freed before exiting the program"; or "If loading a certificate returns an error code, the program must exit in the next step", etc. These examples illustrate the need for a mechanism to specify and reason about program behavior beyond the immediate scope of a function call, highlighting a limitation in the expressiveness of traditional pre- and post-conditions.

Prior work [23] proposes *future conditions* to express the aforementioned constraints on program behavior after function calls have been completed. When combined with pre- and post-conditions, this triplet-style specification effectively encapsulates a usage protocol for each function and the key APIs involved. For example, the specification for `malloc` is written in Fig. 1 [1]. Its pre/post-condition state that the input value should be positive and when a pointer is successfully allocated, i.e., $res{\neq}null$, it triggers an event `malloc`(res), where res denotes the return value. We use ϵ for empty traces and $\_^\star$ for permitting any traces. Its future-condition enforces that the allocated pointer should be finally freed ($\mathcal{F}$ denotes the temporal operator *finally*), which effectively prevents a memory-leak. The *free* function triggers an event *free*(ptr) and its future condition ensures that after the deallocation, the input pointer cannot be accessed throughout its lifetime ($\mathcal{G}$ denotes the temporal operator *globally*), which effectively prevents double-free or use-after-free violations.

```
void *malloc (size_t size);
// pre: size>0 ∧ _*
// post: (res=null ∧ ϵ) ∨ (res≠null ∧ malloc(res))
// future: (res=null ∧ _*)  ∨  (res≠null ∧ F free(res))

void free (void *ptr);
// pre: true
// post: true ∧ free(ptr)
// future: true ∧ G !_(ptr)
```

Fig. 1. Triplet specifications for *malloc* and *free* APIs, taken from [23]

Future conditions provide a general mechanism for specifying both safety and liveness properties through linear temporal logic (LTL) formulas, including resource usage, null-pointer dereferences or unchecked return values, etc.

[1] The specifications are in the form of $\bigvee(\pi \wedge \theta)$, i.e., a disjunctive set of conjunctions between pure arithmetic constraints (π) and trace constraints (θ).

However, prior work [23] for detecting violations of future conditions suffers from several limitations. First, it uses an inefficient entailment checking strategy, where each future condition must be independently verified against all subsequent code. This leads to redundant checks – particularly when processing sequences of function calls, each with its own future conditions. Second, it prioritises bug-finding (no incorrectly flagged safe code) over soundness (no missed violations), which under-approximates program behavior. This involves arbitrarily discarding paths and handling loops via bounded unrolling, risking unsoundness by omitting critical behaviors. Lastly, a major open question is how to effectively represent the future conditions produced by the recursive execution of recursive data structures.

To address the aforementioned challenges, this paper utilizes a set of Hoare-style verification rules to enable sound reasoning about future conditions. For effectful loops working with recursive data structures, we introduce predicates that represent a bag of items for both traces and future conditions, where the commutative law applies. To enhance automation, we employ a lightweight loop invariant synthesis procedure, whose output is constructed using such predicates and can be reliably verified by the proposed verification system. Overall, our approach aims to establish a verification framework based on over-approximation, designed to soundly prove the absence of violations of future conditions.

1. We introduce a set of sound forward reasoning rules to propagate future conditions efficiently. These rules employ over-approximation of behaviors, guaranteeing the absence of false negatives (missed violations) when verifying future conditions. The formal definition and proof of soundness for these rules are presented in Theorem 3.
2. To facilitate the forward reasoning process, we formalize a set of *trace inclusion checking* rules and a set of *trace subtraction* rules. We define and prove their soundness in Theorem 1 and Theorem 2, respectively.
3. We utilize predicates to represent collections of traces and future conditions and implement a trace invariant synthesis procedure that generates these predicates, which our verification rules can then reliably verify.
4. We demonstrate the effectiveness of the proposed verification framework through experimental results and case studies. All lemmas and theorems presented have been proven in Coq. Our artifact is publicly available [18].

2 Overview and Motivating Examples

This section outlines the current solution and challenges for reasoning about future conditions, highlighting our contributions through examples.

2.1 The Current Solution

The essence of reasoning future conditions in [23] can be captured by the following rule for function calls, where proof obligations are highlighted:

$$\frac{f\,(\overline{x})\,[\Phi_{pre}, \Phi_{post}, \Phi_{future}] \in \mathcal{E} \quad \Phi \sqsubseteq \Phi_{pre} \quad \{\Phi \circ \Phi_{post}\}\ e\ \{\Phi_e\} \quad \Phi_e \sqsubseteq \Phi_{future}}{\{\Phi\}\ f(\overline{x})\,;\ e\ \{\Phi_{post} \circ \Phi_e\}}\ [FV\text{-}Call\text{-}Inefficient]$$

A traditional Hoare-style verification rule for function calls works roughly as follows: it retrieves the callee f's specification from the environment $\mathcal{E}$, and if the current program state Φ entails ($\sqsubseteq$) the callee's pre-condition Φ_{pre}, it extends the program state with the callee's post-condition. Here "$\Phi \circ \Phi_{post}$" [2] means to sequentially compose two specifications. Now, having Φ_{future}, [23] extends the rule with one more proof obligation: the behavior of e (where e denotes the rest of code following the call to $f(\overline{x})$), as captured by Φ_e, entails the callee's future condition, effectively imposing constraints on the code e after the call.

While this solution is correct, it suffers from repeated invocation(s) of entailment checking, as demonstrated in Fig. 2. The example includes multiple memory operations, with a use-after-free (UAF) bug at line 7. According to the specifications in Fig. 1, every call to *malloc* or *free* necessitates checking the subsequent code against the specified future conditions. We mark all triggered trace entailment checks in blue, and the UAF bug is detected when an entailment failure occurs at line 5. This approach introduces redundant checks – for example, events $free(buf2)$ (generated from line 5) and $free(buf1)$ (generated from line 8) are checked at least two and four times, respectively. The inefficiency escalates further when future conditions are applied to operations like *strncpy*. In

```
   1. int main(int argc, char **argv){
   2.     char *buf1, *buf2, *buf3;
   3.     buf1 = malloc(1);
// malloc(buf2)·free(buf2)·malloc(buf3)·strncpy(buf2)·free(buf1)·free(buf3)⊑F(free(buf1))
   4.     buf2 = malloc(1);
// free(buf2)·malloc(buf3)·strncpy(buf2)·free(buf1)·free(buf3)⊑ F(free(buf2))
   5.     free(buf2);
// malloc(buf3)·strncpy(buf2)·free(buf1)·free(buf3) ⋢ G(!_(buf2)) ⇐ Bug Detected!
   6.     buf3 = malloc(1);
// strncpy(buf2)·free(buf1)·free(buf3) ⊑ F(free(buf3))
   7.     strncpy(buf2,argv[1],1); //A UAF bug here!
   8.     free(buf1);
// free(buf3) ⊑ G(!_(buf1))
   9.     free(buf3);}
// ϵ ⊑ G(!_(buf3))
```

Fig. 2. Detecting a UAF bug from CWE-416 [7]

[2] The interpretation of $\circ$ depends on the specific logic employed in Φ: for pure arithmetic: $\circ \equiv \wedge$ (logical conjunction); for separation logic: $\circ \equiv *$ (separating conjunction); for linear temporal logic: $\circ \equiv \cdot$ (temporal concatenation), etc.

the worst case, the checking process exhibits (informally) quadratic complexity relative to program length. To overcome this limitation, we propose an improved specification method for future conditions that efficiently detects the UAF bug while eliminating redundant re-analysis.

2.2 Towards Efficient Propagation for Future Conditions

We propose a simplified specification syntax that reduces redundancy by associating future conditions with post-conditions, thereby eliminating unnecessary arithmetic constraints. As shown in Fig. 3, each specification comprises a precondition (**req** clause), and a *post summary* (**ens** clause) structured as "**req**: π **ens**: $\bigvee(\pi\,;\theta\,;F)$". Here, π is the precondition, and each disjunctive case in the post summary includes: a post-state pure constraint π, a trace formula θ recording triggered events and an associated future condition F. We use $\_^\star$ for the default future condition, which imposes no constraints on future executions.

```
void *malloc (size_t size);
//req: size>0 ens: (res=null ; ϵ ; _*) ∨ (res≠null ; malloc(res) ; F(free(res)))
void free (void *ptr);
//req: true ens: (res=() ; free(ptr) ; G(!_(ptr)))
char *strncpy (char *dest, const char *src, size_t n)
//req: true ens: (res=dest ; strncpy(dest) ; _*)
```

Fig. 3. An improved way for specifying future conditions for memory usage APIs

The new specifications of *malloc* and *free* are semantically equivalent to their original version shown in Fig. 1. For *strncpy*, it triggers an event involving its first argument. We show in Fig. 4 how the revised specification achieves both conciseness and linear-complexity propagation of future conditions.

Program states are captured in the form of $\{\bigvee(\pi\,;\theta\,;F)\}$. By associating future conditions with program states, we can conjunctively combine different future conditions and compute the next states through *trace subtraction*. Fig. 4 highlights these incremental updates to future conditions. In particular, at line 5, subtracting the event "$free(buf2)$" from "$\mathcal{F}(free(buf2))$" yields $\_^\star$, indicating the fulfillment of the deallocation obligation. At line 7, the UAF bug is detected when we subtract the event "$strncpy(buf2)$" from "$\mathcal{G}(!\_(buf2))$", producing $\bot$ ($false$) – a contradiction that indicates the invalid usage of "$buf2$".

Our approach processes future conditions once per method declaration, requiring only a single analysis pass per event. The key idea is to embed future conditions into program states and support logical operations – such as conjunctive obligations and trace subtraction – over these future conditions. We formalize this approach through a set of novel forward rules and prove their soundness in Sec. 4. Next, we illustrate how to specify and verify future conditions for loops.

2. `char *buf1, *buf2, *buf3;`
$\{(\exists buf1, buf2, buf3.\ true\ ;\ \epsilon\ ;\ \_^{\star})\}$

3. `buf1 = malloc(1);`
$\{(\exists buf1, buf2, buf3.\ buf1{\neq}null\ ;\ malloc(buf1)\ ;\ \mathcal{F}(free(buf1)))\}$

4. `buf2 = malloc(1);`
$\{(\exists buf1, buf2, buf3.\ buf1{\neq}null \wedge buf2{\neq}null\ ;\ malloc(buf1)\cdot malloc(buf2)\ ;\ \mathcal{F}(free(buf1)) \wedge \mathcal{F}(free(buf2)))\}$

5. `free(buf2);`
$\{(\exists buf1, buf2, buf3.\ buf1{\neq}null \wedge buf2{\neq}null\ ;\ malloc(buf1)\cdot malloc(buf2)\cdot free(buf2)\ ;\ \mathcal{F}(free(buf1)) \wedge \_^{\star} \wedge \mathcal{G}(!\_(buf2)))\}$

6. `buf3 = malloc(1);`
$\{(\exists buf1, buf2, buf3.\ buf1{\neq}null \wedge buf2{\neq}null \wedge buf3{\neq}null\ ;\ malloc(buf1)\cdot malloc(buf2)\cdot free(buf2)\cdot malloc(buf3)\ ;\ \mathcal{F}(free(buf1)) \wedge \mathcal{G}(!\_(buf2)) \wedge \mathcal{F}(free(buf3)))\}$

7. `strncpy(buf2,argv[1],1);`
$\{(\exists buf1, buf2, buf3.\ buf1{\neq}null \wedge buf2{\neq}null \wedge buf3{\neq}null\ ;\ malloc(buf1)\cdot malloc(buf2)\cdot free(buf2)\cdot malloc(buf3)\cdot strncpy(buf2)\ ;\ \mathcal{F}(free(buf1)) \wedge \bot \wedge \mathcal{F}(free(buf3))) \Leftarrow \text{✗}\}$
FC Violation Found: subtracting "strncpy(buf2)" from "$\mathcal{G}(!\_(buf2))$" leads to false!

Fig. 4. Detecting the UAF bug in Fig. 2 in a more efficient way

2.3 Predicates for Bags of Traces and Future Conditions

The program in Fig. 5 creates an array of length n and iteratively invokes *malloc* for each element. The specification for *mallocN*, given in Fig. 6, comprises two predicates: "$pred_t(B, i)$" for triggered traces, and "$pred_f(B, i)$" for the future conditions. Predicates are in the form of "$\Lambda_i^B(\bigvee(\pi \wedge \theta))$", where i denotes

```
1 void* mallocN(int n,void **arr,){
2   int i = 0;
3   while (i < n) {
4     arr[i] = malloc(4); i = i+1;}
5   return *arr;}
6
7 void main () {
8   void *arr[5]; mallocN (5, arr);
9   free(arr[0]);/* memory leak */}
```

Fig. 5. Iteratively malloc n times

$$
\begin{aligned}
mallocN(n, arr) \equiv\ & \textbf{req}: length(arr) \geq n \\
& \textbf{ens}: (\exists i.\ true\ ;\ pred_t([0..n), i)\ ;\ pred_f([0..n), i)) \\
pred_t(B, i) \equiv\ & \Lambda_i^B(arr[i] \neq null \wedge malloc(arr[i])) \ \vee\ (arr[i] = null \wedge \epsilon) \\
pred_f(B, i) \equiv\ & \Lambda_i^B(arr[i] \neq null \wedge \mathcal{F}(free(arr[i])))
\end{aligned}
$$

Fig. 6. Specification for *mallocN*

the iterator and B denotes the
bag of elements that satisfies the specification $\bigvee(\pi \wedge \theta)$, allowing flexible specification under different arithmetic constraints. Predicates enforce commutativity, discarding trace order sensitivity among each element in the bag.

In Fig. 7, we demonstrate the forward reasoning for *main*. After the function call to *mallocN(5, arr)*, the predicates are instantiated with the concrete bag, i.e., [0..5). At line 9, when the first element is freed, the future condition predicate updates to a new bag [1..5), reflecting the removal of *arr[0]*. By the end, the memory leak is detected via a failed entailment checking, that the empty trace is not always permitted in the future condition "$pred_f([1..5), i)$". To automate this process, we introduce a heuristic-based synthesis for predicate generation, and enable the predicate propagation in the trace subtraction rules.

8. `void *arr[5]; mallocN (5, arr);`
$\{(\exists arr, i.\, length(arr) = 5 \,;\, pred_t([0..5), i) \,;\, pred_f([0..5), i))\}$

9. `free(arr[0]);`
$\{(\exists arr, i.\, length(arr) = 5 \,;\, pred_t([0..5), i) \cdot free(arr[0]) \,;\, pred_f([1..5), i) \wedge \mathcal{G}(!\_(arr[0])))\}$

FC Violation Found: empty trace "ϵ" does not satisfy the obligation "$pred_f([1..5), arr)$"!

Fig. 7. Forward reasoning for *main* and detecting the memory leak in Fig. 5

Remark. Such memory usage violations can also be detected by static analyzers such as Pulse (a memory safety checker in Infer [14]), which relies on separation logic. However, Pulse requires an additional side check upon function exit to account for residual footprints when detecting memory leaks. In contrast, our approach employs a lightweight, general-purpose temporal logic to explicitly define obligations, avoiding the need for implicit checks. This not only simplifies verification but also extends applicability beyond memory safety, enabling the verification of both safety and liveness properties. We showcase the variety of bug types that can be handled by specifying future conditions in Table 2.

3 Target Language and Specifications

We target an imperative, first-order, call-by-value core language, defined in Fig. 8. A program $\mathcal{P}$ comprises a list of function declarations $\overline{Func}$. Here, we use the $\overline{overline}$ to denote a finite list of items, for example, $\overline{x}$ refers to a list of variables, $x_1, \ldots, x_n$. Each function has a name f, formal arguments $\overline{x}$, and an expression-oriented body e. Function specifications contains a pre-condition π, and a post summary Δ, which is a set of disjunctive tuples. Each three-element tuple contains: a pure formula π, a trace formula θ, and a future condition F.

We utilize π as the basic logical formula, capturing the Presburger arithmetic conditions on program inputs and local variables. Values include variables and constants ranging from integers, Boolean, unit, null, and * for non-deterministic

$$
\begin{array}{llll}
(\textit{Program}) & \mathcal{P} ::= \overline{\textit{Func}} & (\textit{Function Decl.}) & \textit{Func} ::= f(\overline{x})\{e\} \\
(\textit{Post Summary}) & \Delta ::= \bigvee(\pi\,;\theta\,;F) & (\textit{Specification}) & [\mathbf{req}\colon \pi\ \mathbf{ens}\colon \Delta]
\end{array}
$$

$$
\begin{array}{lll}
(\textit{Expressions}) & e ::= & v \mid x := e \mid \textit{local}\ x \mid e_1\,;\,e_2 \mid \textit{ev}(\mathrm{A}) \mid \\
 & & \mathbf{if}\ b\ e_1\ e_2 \mid \mathbf{while}\ \pi\ \mathbf{do}\ e \mid f(\overline{x}) \mid \mathbf{assert}_f\ F \\
(\textit{Trace}) & \theta ::= & \bot \mid \epsilon \mid \mathrm{A} \mid \theta_1 \vee \theta_2 \mid \theta_1 \wedge \theta_2 \mid \theta_1 \cdot \theta_2 \mid \theta^\star \\
(\textit{Single Events}) & \mathrm{A} ::= & \textit{ev}(t) \mid \neg\textit{ev}(t) \mid \neg\_(t) \mid \_ \mid \textit{pred}(B,i) \\
(\textit{Future Cond.}) & F ::= & \theta \qquad \textit{pred}(B,i) ::= \Lambda_i^B(\bigvee(\pi \wedge \theta))
\end{array}
$$

$$
\begin{array}{llll}
(\textit{Pure}) & \pi ::= & T \mid F \mid \textit{bop}(t_1,t_2) \mid \pi_1 \wedge \pi_2 \mid \pi_1 \vee \pi_2 \mid \neg\pi \mid \exists x.\,\pi \mid \forall x.\,\pi & \\
(\textit{Terms}) & t ::= & v \mid t_1 + t_2 \mid -t & (\textit{Values})\ v ::= x \mid i \mid b \mid () \mid \textit{null} \mid * \\
(\textit{Bag}) & B ::= & \emptyset \mid \{v\} \mid B_1 - B_2 \mid B_1 \cup B_2 \mid B_1 \cap B_2 &
\end{array}
$$

Fig. 8. Syntax of the core language and the specification language

values. Expressions include sequencing, function calls, conditionals, assignments, while loops, etc. Furthermore, we use "$\textit{ev}(\mathrm{A})$" for explicitly raising events, and "$\mathbf{assert}_f\ F$" for asserting constraints for future behaviors.

Traces are regular expressions, comprising *false* ($\bot$); empty traces ϵ; singleton events A; sequential concatenations $\theta_1 \cdot \theta_2$; disjunctions $\theta_1 \vee \theta_2$; conjunctions $\theta_1 \wedge \theta_2$; and finite time (zero or many) repetition of a trace, constructed by a Kleene star $\theta^\star$. In this paper, we use LTL operators $\mathcal{F}(\mathrm{A})$, $\mathcal{G}(\mathrm{A})$ and $\mathcal{N}(\mathrm{A})$ as short hands for regular expressions "$(\neg\mathrm{A})^\star \cdot \mathrm{A} \cdot \_^\star$", "$(\mathrm{A})^\star$" and "$\_ \cdot \mathrm{A} \cdot \_^\star$", respectively. Singleton events include: parameterized events $\textit{ev}(t)$; negated parameterized events $\neg\textit{ev}(t)$; forbid argument $\neg\_(t)$; wildcards $\_$ matching any event; predicates that capture a bag of disjunctive traces. Future conditions constraint temporal behaviors in the future and are essentially trace constructs; we use "F" to denote them for clarity and to avoid ambiguity.

We use "$\_^\star$" to represent the default future condition, which permits any possible future executions. The Boolean values of T and F are respectively indicated by *true* and *false*. The binary operators *bop* are from $\{<, \leq, =, \geq, >\}$. A term can be a simple value v or simple computations of terms, $t_1 + t_2$ and $-t$. A bag ia a set of unique elements, and we use $[0..n)$ as a shorthand for $\{0..(n\text{-}1)\}$.

3.1 Instrumented Semantics for the Target Language

To facilitate the soundness proof for the forward rules, we present an instrumented reduction $[s, \rho, F, e] \longrightarrow [s', \rho', F', v]$ for the core language, shown in Fig. 9. Each reduction rule operates on a concrete program state on the left hand side, where an expression e is associated with a concrete stack s; a sequence of events triggered in the course of its execution ρ, and a future condition F. Given *Var* is a set of variables, and *Val* denotes all the concrete values, s and ρ are from the following concrete domains: $s \triangleq \textit{Var} \rightarrow \textit{Val}$ and $\rho \triangleq \textit{list}\ \mathrm{A}$. The big-step semantics reduce any given program e to a resulting value v, and when the e terminates, the state is transformed from (s, ρ, F) to (s', ρ', F'). We use $[\![\pi]\!]_s$ to

$$\frac{}{[s,\rho,F',\mathbf{assert}_f\ F]\longrightarrow[s,\rho,F'\wedge F,()]}\ \text{[OP-Assume]}\qquad \frac{\mathrm{A}=ev(t)\quad F\ominus_{lin}\mathrm{A}\hookrightarrow_{Pure(s)}F'}{[s,\rho,F,ev(t)]\longrightarrow[s,\rho\,{+}{+}[\mathrm{A}],F',()]}\ \text{[OP-Ev]}$$

$$\frac{}{[s,\rho,F,v]\longrightarrow[s,\rho,F,v]}\ \text{[OP-Val]}\qquad \frac{}{[s,\rho,F,local\ x]\longrightarrow[s\,{+}{+}[x],\rho,F,()]}\ \text{[OP-Local]}$$

$$\frac{[s,\rho,F,e_1]\longrightarrow[s_1,\rho_1,F_1,v_1]\quad[s_1,\rho_1,F_1,e_2]\longrightarrow[s_2,\rho_2,F_2,v_2]}{[s,\rho,F,e_1\,;\,e_2]\longrightarrow[s_2,\rho_2,F_2,v_2]}\ \text{[OP-Let]}\qquad \text{[OP-Assign]}\ \frac{x\in dom(s)\quad[s,\rho,F,e]\longrightarrow[s_1,\rho_1,F_1,v]}{[s,\rho,F,x:=e]\longrightarrow[s_1[x{:=}v],\rho_1,F_1,()]}$$

$$\text{[OP-Cond-T]}\ \frac{[s,\rho,F,e_1]\longrightarrow[s',\rho',F',v]}{[s,\rho,F,\mathbf{if}\ true\ e_1\ e_2]\longrightarrow[s',\rho',F',v]}\qquad \text{[OP-Cond-F]}\ \frac{[s,\rho,F,e_2]\longrightarrow[s',\rho',F',v]}{[s,\rho,F,\mathbf{if}\ false\ e_1\ e_2]\longrightarrow[s',\rho',F',v]}$$

$$\frac{[\![\pi]\!]_s=true\qquad e'=(e\,;\,\mathbf{while}\ \pi\ \mathbf{do}\ e)\qquad[s,\rho,F,e']\longrightarrow[s',\rho',F',v]}{[s,\rho,F,\mathbf{while}\ \pi\ \mathbf{do}\ e]\longrightarrow[s',\rho',F',v]}\ \text{[OP-While-T]}$$

$$\text{[OP-While-F]}\ \frac{[\![\pi]\!]_s=false}{[s,\rho,F,\mathbf{while}\ \pi\ \mathbf{do}\ e]\longrightarrow[s,\rho,F,()]}\qquad \text{[OP-Call]}\ \frac{f(\overline{y})\{e\}\in\mathcal{P}\quad[s,\rho,F,e\,[\overline{x}/\overline{y}]]\longrightarrow[s',\rho',F',v]}{[s,\rho,F,f(\overline{x})]\longrightarrow[s',\rho',F',v]}$$

Fig. 9. Big-step instrumented semantic model for the core language

denote that validity of the constraint π with respect to a concrete stack. In particular, when triggering an event $ev(t)$, [OP-Ev] subtracts the event from the current future condition. We use $Pure(s)$ to convert a concrete stack into a pure constraint which contains all the equalities between variables and values. Intuitively, $F\ominus_{lin}\theta\hookrightarrow_{\pi}F'$ subtracts a trace θ from the given future condition F, resulting in a next state future condition F'. For instance, $\mathcal{F}(free(x))\ominus_{lin}free(x)\hookrightarrow_{true}\_^{\star}$, and $\mathcal{G}(!\_(x))\ominus_{lin}free(x)\hookrightarrow_{true}\bot$. Trace subtraction is detailed in Sec. 4.3.

$$\begin{array}{lcl}
s,\rho\models\pi\wedge\epsilon & \Leftrightarrow & \rho=[]\ and\ [\![\pi]\!]_s=true\\
s,\rho\models\pi\wedge(\theta_1\vee\theta_2) & \Leftrightarrow & s,\rho\models\pi\wedge\theta_1\ or\ s,\rho\models\pi\wedge\theta_2\\
s,\rho\models\pi\wedge(\theta_1\wedge\theta_2) & \Leftrightarrow & s,\rho\models\pi\wedge\theta_1\ and\ s,\rho\models\pi\wedge\theta_2\\
s,\rho\models\pi\wedge(\theta_1\cdot\theta_2) & \Leftrightarrow & exists\ \rho_1\ \rho_2\ such\ that\ \rho_1{+}{+}\rho_2=\rho\ and\\
 & & s,\rho_1\models\pi\wedge\theta_1, s,\rho_2\models\pi\wedge\theta_2\\
s,\rho\models\pi\wedge\theta^{\star} & \Leftrightarrow & s,\rho\models\pi\wedge\epsilon\ or\ s,\rho\models\pi\wedge\theta\cdot\theta^{\star}\\
s,\rho\models\pi\wedge ev(t) & \Leftrightarrow & exists\ t'\ such\ that\ \rho=[ev(t')]\ and\ [\![\pi\wedge(t=t')]\!]_s=true\\
s,\rho\models\pi\wedge\neg ev(t) & \Leftrightarrow & exists\ ev',t'\ such\ that\ \rho=[ev'(t')]\ and\\
 & & either\ ev\neq ev'\ or\ [\![\pi\wedge(t=t')]\!]_s=false\\
s,\rho\models\pi\wedge\neg_(t) & \Leftrightarrow & exists\ ev,t'\ such\ that\ \rho=[ev(t')]\ and\ [\![\pi\wedge(t=t')]\!]_s=false\\
s,\rho\models\pi\wedge_ & \Leftrightarrow & exists\ ev,t\ such\ that\ \rho=[ev(t)]\ and\ [\![\pi]\!]_s=true\\
s,\rho\models\Lambda_i^B(\bigvee(\pi\wedge\theta)) & \Leftrightarrow & forall\ j\in B\ such\ that\ s,\rho\models(\bigvee(\pi\wedge\theta))[j/i]
\end{array}$$

Fig. 10. Semantic model of trace specifications

3.2 Semantic Model for Logical Assertions

We define the semantic model for program assertions in Fig. 10. Let $s, \rho \models \pi \wedge \theta$ denote the *models* relation, i.e., the concrete stack s and a concrete sequence of events ρ satisfy the logical state π and the temporal specification θ. Here, [] is an empty sequence and ++ appends two trace sequences. A concrete state s, ρ models a trace predicate if for every element j in the bag B, there exists an instantiated disjunctive case that can be modelled by s, ρ.

4 Forward Reasoning

We formalize a set of syntax-directed forward rules in Fig. 11, in the form of Hoare-style triples: $\{\pi\}\ e\ \{\Delta\}$, i.e., a shorthand for $\mathcal{E} \vdash \{(\pi\,;\epsilon\,;\_^\star)\}\ e\ \{\Delta\}$, where $\mathcal{E}$ is an environment mapping from functions to their specifications, ϵ represents an empty history trace and $\_^\star$ is the default future condition. Under a partial correctness interpretation, which we adopt in this paper, the triple means that if π describes the state before executing e, if e terminates, Δ describes the resulting post-summary.

Rule [FV-Assume-F] interpolates future conditions into program states, thereby constraining subsequent execution behaviors. Rule [FV-Ev] generates one event into the triggered trace, leaving a default future condition, $\_^\star$. Rule [FV-Value] updates the result value in the post state, where we use the reserved variable *res* to denote the (temporarily) result values, leaving a default future condition. Rule [FV-Seq] reasons about e_1 and e_2 in sequence, and implicitly relies on [FV-Disj] to

$$\frac{}{\{\pi\}\ \mathbf{assert}_f\ F\ \{(\pi\,;\epsilon\,;F)\}}\ \text{[FV-Assume-F]} \qquad \frac{}{\{\pi\}\ ev(t)\ \{(\pi\,;ev(t)\,;\_^\star)\}}\ \text{[FV-Ev]}$$

$$\frac{\pi' = \pi \wedge (res = v)}{\{\pi\}\ v\ \{(\pi'\,;\epsilon\,;\_^\star)\}}\ \text{[FV-Value]} \qquad \frac{\{\pi\}\ e_1\ \{\Delta_1\} \quad \{\Delta_1\}\ e_2\ \{\Delta_2\}}{\{\pi\}\ e_1\,;\,e_2\ \{\Delta_2\}}\ \text{[FV-Seq]}$$

$$\frac{\text{[FV-Local]} \quad \mathit{fresh}\ x}{\{\pi\}\ \mathit{local}\ x\ \{(\exists x.\,\pi\,;\epsilon\,;F)\}} \qquad \frac{\{\pi \wedge b\}\ e_1\ \{\Delta_1\} \quad \{\pi \wedge \neg b\}\ e_1\ \{\Delta_2\}}{\{\pi\}\ \mathbf{if}\ b\ e_1\ e_2\ \{\Delta_1 \vee \Delta_2\}}\ \text{[FV-Cond]}$$

$$\frac{\mathit{fresh}\ r \qquad f(\overline{y})[\mathbf{req}\colon \pi'\ \mathbf{ens}\colon \Delta] \in \mathcal{E} \qquad \pi \leq \pi'[\overline{x}/\overline{y}]}{\{\pi\}\ f(\overline{x})\ \{\exists r.\,\Delta[\overline{x}/\overline{y}, r/res]\}}\ \text{[FV-Call]}$$

$$\frac{\text{[FV-Assign]} \quad \mathit{fresh}\ r \qquad \{\pi\}\ e\ \{\Delta_1\}}{\{\pi\}\ x := e\ \{\Delta_1[r/res] \wedge (x = r)\}} \qquad \frac{\text{[FV-While]} \quad \{(\pi \wedge \pi_g\,;\theta\,;F)\}\ e\ \{(\pi\,;\theta\,;F)\}}{\{(\pi\,;\theta\,;F)\}\ \mathbf{while}\ \pi_g\ \mathbf{do}\ e\ \{(\pi \wedge \neg\pi_g\,;\theta\,;F)\}}$$

$$\frac{\text{[FV-Disj]} \quad \forall(\pi\,;\theta\,;F) \in \Delta. \quad \{(\pi\,;\theta\,;F)\}\ e\ \{\Delta_i\}}{\{\Delta\}\ e\ \{\bigvee \Delta_i\}} \qquad \frac{\text{[FV-Struct]} \quad \{\pi\}\ e\ \{\Delta\} \quad \forall(\pi'\,;\theta_1\,;F_1) \in \Delta. \quad F \ominus \theta_1 \hookrightarrow_{\pi'} F'}{\{(\pi\,;\theta\,;F)\}\ e\ \{\bigvee(\pi'\,;(\theta \cdot \theta_1)\,;(F' \wedge F_1))\}}$$

$$\frac{\{\pi_3\}\ e\ \{(\pi_4\,;\theta'\,;F')\} \qquad \pi_1 \leq \pi_3 \qquad \pi_4 \leq \pi_2 \qquad \theta' \sqsubseteq_{\pi_4} \theta \qquad F \sqsubseteq_{\pi_4} F'}{\{\pi_1\}\ e\ \{(\pi_2\,;\theta\,;F)\}}\ \text{[FV-Conseq]}$$

Fig. 11. Forward verification and inference rules

distribute the disjunctions introduced by e_1, and [FV-Struct] to propagate traces and future conditions structurally. Rule [FV-Local] introduces an existential variable x into the state. Rule [FV-Cond] computes the post-summaries from both branches by extending the state with b being true and false, respectively; then, it disjunctively unions the results. Rule [FV-Call] retrieves the verified specification of the callee function, checks the entailment between the current state and the callee's pre-condition, then concludes the instantiated post-summary.

Rule [FV-Assign] derives the post-summary Δ_1 for e by introducing a fresh variable r to represent Δ_1's return values, substituting all occurrences of *res* with r, and binding x and r in the final post-summary. In [FV-While], the initial state $(\pi\,;\theta\,;F)$ serves as a loop invariant, and executing the loop body must re-establish the loop invariant under the same trace and future conditions. We present a trace invariant synthesis procedure in Sec. 4.1.

Furthermore, [FV-Disj] distributes the disjunctive tuples from the left-hand side and unions their independent reasoning results. Rule [FV-Struct] is a structural rule designed to handle arbitrary tuples from the left-hand side. Given any history traces θ and context future conditions F, [FV-Struct] performs two key operations: it concatenates the extended traces to the history trace $(\theta \cdot \theta_1)$; and propagates the subtracted F, represented as $F \ominus \theta_1 \hookrightarrow_\pi F'$, to be part of the post-summary. Intuitively, this rule concludes a conjunctive future condition, i.e., $F' \wedge F_1$, which encompasses both the future condition F_1 obtained by executing e and the future condition propagated from the context F_1. Lastly, Rule [FV-Conseq] soundly weakens the post-summaries by weakening postconditions (the covariant) and strengthening pre/future-conditions (the contravariants).

Entailment Checking & Trace Subtraction. Pure constraints entailment $(\pi \leq \pi')$ are discharged by the Z3 [8] solver. Trace inclusions $(\theta_1 \sqsubseteq_\pi \theta_2)$ are discharged by a term rewriting system (TRS), which is extended from a known solution [5] for solving inequalities between regular expressions, detailed in Sec. 4.2. Trace subtraction from future conditions $(F \ominus \theta \hookrightarrow_\pi F')$ propagates the context future condition concerning specific execution traces, detailed in Sec. 4.3. Overall, our main technical contributions are: extending the TRS to accommodate our new event types, and developing novel trace subtraction rules.

Specification Inference for Function Definition. Given a set of primitive specifications, our goal is to develop a verification system that is *as automated as possible*, where the specification for each function definition is inferred compositionally. The top-level inference process is illustrated in [FV-Func]. The environment $\mathcal{E}$ includes both primitive specifications and the specifications inferred so far. For each function definition $f(\overline{x})\,\{e\}$ in the given program, we derive the specification for e using the forward rules in Fig. 11 and then reason the rest of the program with the extended environment $\mathcal{E}'$.

$$\frac{\mathcal{E} \vdash \{\pi'\}\ e\ \{\Delta\} \qquad \forall(\pi\,;\theta\,;F) \in \Delta.\quad \delta(F_\exists) = \mathit{true} \qquad \mathcal{E}' = \mathcal{E} \mathbin{+\!\!+} f(\overline{x})\ [\mathbf{req}\colon \pi'\ \mathbf{ens}\colon \Delta]\ \{e\} \qquad \mathcal{E}' \vdash \mathcal{P}}{\mathcal{E} \vdash f(\overline{x})\ \{e\}\ ;\ \mathcal{P}}\ \text{[FV-Func]}$$

For each tuple in e's post-summary Δ, we check whether the empty trace satisfy the future conditions involving existential variables, i.e., $F_{\exists}$, as the lifetime of existential variables ends when the function concludes. The $F_{\exists}$ is computed from F by replacing all events related to universally quantified variables – including the formal arguments $\overline{x}$ and the return value *res* – into wildcards "_", and the resulting future condition thus retains only the constraints on existential variables. For example, in Fig. 5, the memory leak error upon the usage of *arr* is detected in *main* instead of *mallocN*, as *arr* is universally quantified in *mallocN*, but existential quantified in *main*. The Nullable function (cf. Definition 1) returns a Boolean value indicating if θ permits the empty trace.

Definition 1 (Nullable). *Given any sequence θ, we recursively define $\delta(\theta)$:*

$$\delta(\theta_1 \cdot \theta_2) = \delta(\theta_1) \wedge \delta(\theta_2) \qquad \delta(\theta_1 \vee \theta_2) = \delta(\theta_1) \vee \delta(\theta_2) \qquad \delta(\theta_1 \wedge \theta_2) = \delta(\theta_1) \wedge \delta(\theta_2)$$
$$\delta(\Lambda_i^B(\bigvee(\pi \wedge \theta))) = \exists\theta.\ (\delta(\theta) = true) \qquad \delta(\epsilon) = \delta(\theta^\star) = true \qquad \delta(\bot) = \delta(A) = false$$

4.1 Trace Invariant Synthesis

A key challenge in automating the verification lies in inferring trace invariants – specifically, given a loop expression "**while** π_g **do** e", determining the initial state $(\pi\,;\theta\,;F)$, deployed in [FV-While]. The trace invariant synthesis process is outlined in Algorithm 1. It takes a loop statement as input and either outputs a valid invariant or triggers a verification failure – indicating that the loop invariants must be provided manually.

At lines 1-2, it derives the specification for e and obtains a fresh iterator i. Then deploys the bag bounds generation function (Definition 2) to produce a set of candidate bags. For each candidate bag B, lines 4-5 construct the trace and future condition predicates respectively. If there exists an invariant that can be re-established after one iteration of the loop body (line 6), it is deemed valid. Otherwise, the synthesis fails and the verification could not proceed due to the absence of trace invariants.

Algorithm 1 Loop Invariant Synthesis

Require: A loop: **while** π_g **do** e
Ensure: A Loop Invariants or Failure
1: $\{\pi'\}\ e\ \{\bigvee(\pi\,;\theta\,;F)\}$
2: *fresh* i
3: **for all** $B \in \mathcal{BB}(\pi_g)$ **do**
4: $\quad\theta \leftarrow \Lambda_i^B(\bigvee(\pi \wedge \theta))$
5: $\quad F \leftarrow \Lambda_i^B(\bigvee(\pi \wedge F))$
6: $\quad$**if** $\{(\pi' \wedge \pi_g\,;\theta\,;F)\}\ e\ \{(\pi'\,;\theta\,;F)\}$
7: $\quad$**then return** $(\pi'\,;\theta\,;F)$
8: **end for**
9: **return** Unknown Loop Invariant

Definition 2 (Bag Bounds Generation). *For any loop guard π, we propagate a set of candidate bag bounds using $\mathcal{BB}(\pi)$, for constructing the trace predicates:*

$$\mathcal{BB}(t_1 < t_2) = \{[0..t_1), \ldots\} \quad \mathcal{BB}(\pi_1 \wedge \pi_2) = \mathcal{BB}(\pi_1) \cup \mathcal{BB}(\pi_2)$$
$$\mathcal{BB}(t_1 \leq t_2) = \mathcal{BB}(t_1 < (t_2+1)) \quad \mathcal{BB}(t_1 > t_2) = \mathcal{BB}(t_2 < t_1) \quad \mathcal{BB}(t_1 \geq t_2) = \mathcal{BB}(t_2 \leq t_1)$$

In Definition 2, the base case is generating the candidate bags for $t_1{<}t_2$ while the remaining cases are derived by reduction to the base case. For example, Fig. 12 demonstrates the verification process for the loop in *mallocN* (Fig. 5). At line 3, Algorithm 1 generates an invariant where i serves as the iterator and "$[0..i)$" denotes the bag representation. The verification then establishes that this invariant holds through each loop iteration – specifically, the program states after lines 3 and 5 are isomorphic with respect to the iterator's value. The final program state is derived by conjoining with the negated loop guard, i.e., $(i{=}n)$.

This process employs heuristics and is inherently incomplete: it may fail to capture the complete bag range or the precise iterator (which can be improved by incorporating the loop iterator's initial value). Nevertheless, the verification remains sound, as [FV-While] only succeeds when valid invariants are provided.

3. `while (i < n){`

$\{(\exists i.\, true\,;\, pred_t([0..i),i)\,;\, pred_f([0..i),i))\}$

4. `arr[i] = malloc(4);`

$\{(\exists i.\, true\,;\, pred_t([0..i{+}1),i)\,;\, pred_f([0..i{+}1),i))\}$

5. `i = i+1;`

$\{(\exists i.\, true\,;\, pred_t([0..i{+}1),i{+}1)\,;\, pred_f([0..i{+}1),i{+}1))\}$

6. `}`

$\{(\exists i.\, i{=}n\,;\, pred_t([0..i))\,;\, pred_f([0..i)))\} \rightsquigarrow \{(\exists i.\, true\,;\, pred_t([0..n))\,;\, pred_f([0..n)))\}$

Fig. 12. Outlining the reasoning for the loop in Fig. 5

4.2 Trace Inclusion

As shown in Fig. 13, we use $(\mathcal{H} \vdash \theta_1 \sqsubseteq_\pi \theta_2)$ to denote the inclusion between two traces, where $\mathcal{H}$ contains a set of proof hypotheses, and when omitted it is initialized with $\{\}$. [Inc-Disprove] disproves the inclusions when the antecedent is nullable (containing the empty trace ϵ), while the consequent is not. [Inc-Reoccur] proves an inclusion by leveraging existing hypotheses in the proof context $\mathcal{H}$ that soundly justify the current goal. [Inc-Unfold] serves as the inductive step, unfolding inclusions – proof of the original inclusion succeeds if all derivative inclusions succeed. Termination of the rewriting is guaranteed because the set of derivatives to be considered is finite, and possible cycles are detected via memorization ([Inc-Reoccur]). We define the soundness of these rules in Theorem 1.

$$\text{[Inc-Disprove]}\ \frac{\delta(\theta_1) \wedge \neg\delta(\theta_2)}{\mathcal{H} \vdash \theta_1 \not\sqsubseteq_\pi \theta_2} \qquad \text{[Inc-Reoccur]}\ \frac{(\theta_1 \sqsubseteq_\pi \theta_2) \in \mathcal{H}}{\mathcal{H} \vdash \theta_1 \sqsubseteq_\pi \theta_2} \qquad \text{[Inc-Unfold]}\ \frac{\forall \mathrm{A} \in fst(\theta_1).\ (\theta_1 \sqsubseteq_\pi \theta_2) +\!\!+ \mathcal{H} \vdash \mathcal{D}^\pi_{\mathrm{A}}(\theta_1) \sqsubseteq_\pi \mathcal{D}^\pi_{\mathrm{A}}(\theta_2)}{\mathcal{H} \vdash \theta_1 \sqsubseteq_\pi \theta_2}$$

Fig. 13. Trace inclusion checking rules

To facilitate these inclusion rules, we provide the definitions of the deployed auxiliary functions: *Nullable* (δ) at Definition 1, *First* (*fst*) at Definition 3, and

Derivative ($\mathcal{D}^{\pi}_{\mathrm{A}}(\theta)$) at Definition 4. Informally, the First function $fst(\theta)$ computes a set of possible initial events from θ. The Derivative function $\mathcal{D}^{\pi}_{\mathrm{A}}(\theta)$ eliminates an event A from the head of θ and returns what remains.

Definition 3 (First). *Let $fst(\theta)$ be the set of initial events derivable from θ.*

$$fst(\bot)=fst(\epsilon)=\{\} \quad fst(A)=\{A\} \quad fst(\theta_1\vee\theta_2)=fst(\theta_1)\cup fst(\theta_2) \quad fst(\theta^\star)=fst(\theta)$$

$$fst(\theta_1\wedge\theta_2)=fst(\theta_1)\cap fst(\theta_2) \qquad fst(\theta_1\cdot\theta_2)=\begin{cases} fst(\theta_1)\cup fst(\theta_2) & if \quad \delta(\theta_1)=true \\ fst(\theta_1) & if \quad \delta(\theta_1)=false \end{cases}$$

Definition 4 (Derivative). *The partial derivative $\mathcal{D}^{\pi}_{A}(\theta)$ eliminates an event A from the head of a trace θ, defined as follows:*

$$\mathcal{D}^{\pi}_{\mathrm{A}}(\bot)=\mathcal{D}^{\pi}_{\mathrm{A}}(\epsilon)=\bot \quad \mathcal{D}^{\pi}_{\mathrm{A}}(\theta_1\wedge\theta_2)=\mathcal{D}^{\pi}_{\mathrm{A}}(\theta_1)\wedge\mathcal{D}^{\pi}_{\mathrm{A}}(\theta_2) \quad \mathcal{D}^{\pi}_{\mathrm{A}}(\theta_1\vee\theta_2)=\mathcal{D}^{\pi}_{\mathrm{A}}(\theta_1)\vee\mathcal{D}^{\pi}_{\mathrm{A}}(\theta_2)$$

$$\mathcal{D}^{\pi}_{\mathrm{A}}(\theta^\star)=\mathcal{D}^{\pi}_{\mathrm{A}}(\theta)\cdot\theta^\star \qquad \mathcal{D}^{\pi}_{\mathrm{A}}(\theta_1\cdot\theta_2)=\begin{cases} (\mathcal{D}^{\pi}_{\mathrm{A}}(\theta_1)\cdot\theta_2)\vee\mathcal{D}^{\pi}_{\mathrm{A}}(\theta_2) & if \quad \delta(\theta_1)=true \\ \mathcal{D}^{\pi}_{\mathrm{A}}(\theta_1)\cdot\theta_2 & if \quad \delta(\theta_1)=false \end{cases}$$

Definition 5 serves as the base case for Definition 4, computing the derivatives between two events. While these auxiliary functions were originally designed to solve inequalities between regular expressions, they only supported cases where events were simple alphabets [5], with derivatives computed via lexical comparison. In our work, events could be trace predicates or parameterized with program variables. To handle such cases, we thus extend the Derivative function with pure constraints. In Definition 5, the definition of event derivatives adheres strictly to the semantic model (cf. Fig. 10). Notice that A takes one of two forms: a positive event "$ev(t)$", or a trace predicate, as they are the only cases that are derivable from the program executions. When both A and B are trace predicates, they must first be normalized into the same bag "B" and then processed using the trace subtraction operator (detailed in Sec. 4.3).

Definition 5 (Derivative for Events). *Given any two events A, B, the derivative of B with respect to A, i.e., $\mathcal{D}^{\pi}_{A}(B)$ is defined as follows: ($\bot$ for the unmentioned scenarios)*

$$\mathcal{D}^{\pi}_{ev(t)}(ev(t'))=\begin{cases}\epsilon & if \ \ \pi\Rightarrow(t{=}t') \\ \bot & otherwise\end{cases} \qquad \mathcal{D}^{\pi}_{ev(t)}(\neg\_(t'))=\begin{cases}\epsilon & if \ \ \pi\not\Rightarrow(t{=}t') \\ \bot & otherwise\end{cases}$$

$$\mathcal{D}^{\pi}_{ev(t)}(\neg ev'(t'))=\begin{cases}\bot & if \ \ ev=ev' \ \ and \ \ \pi\leq(t{=}t') \\ \epsilon & otherwise\end{cases}$$

$$\mathcal{D}^{\pi}_{ev(t)}(\_)=\epsilon \qquad \mathcal{D}^{\pi'}_{ev(t)}(\Lambda^B_i(\ldots))=\Lambda^B_i(\ldots)$$

$$\mathcal{D}^{\pi'}_{\Lambda^B_i(\bigvee(\pi_2\wedge\theta_2))}(\Lambda^B_i(\bigvee(\pi_1\wedge\theta_1)))=\Lambda^B_i(\bigvee(\pi_1\wedge\pi_2\wedge\theta')) \ \ where \ \ \theta_1\ominus\theta_2\hookrightarrow_{\pi'}\theta'$$

Theorem 1 (Soundness of Trace Inclusion). *For all π, θ_1, θ_2, s and ρ, given $\theta_1\sqsubseteq_\pi\theta_2$, it means that if $s,\rho\models\pi\wedge\theta_1$, then $s,\rho\models\pi\wedge\theta_2$.*

Proof. *By induction on the derivation of $\theta_1\sqsubseteq_\pi\theta_2$.*

4.3 Subtracting Traces from Future Conditions

As shown in Fig. 14, we use $F \ominus \theta \hookrightarrow_\pi F'$ to denote the subtraction of the trace θ from the given future condition F, resulting in a "left-over" future condition F', which we call a *residue*. Rule [TS-Base] captures the base case where the trace to subtract is an empty trace, leaving the future condition unchanged. Rule [TS-Bot] handles the case where the subtracted trace is $\bot$, leading to false future conditions. Rule [TS-Ind] disjunctively unions all the subtraction results from the derivatives. Rule [TS-Conseq] allows for strengthening the subtraction residues and weakening the input future conditions. Rule [TS-Trans] captures the case where we can strengthen the subtracted trace. We define and prove the soundness of these rules in Theorem 2, which indicates that in a sound subtraction, the residue can only be strengthened.

$$\frac{}{F \ominus \epsilon \hookrightarrow_\pi \theta}\ [\text{TS-Base}] \qquad \frac{}{F \ominus \bot \hookrightarrow_\pi \bot}\ [\text{TS-Bot}] \qquad \frac{\forall \mathrm{A} \in \mathit{fst}(\theta_2).\quad \mathcal{D}^\pi_{\mathrm{A}}(\theta_1) \ominus \mathcal{D}^\pi_{\mathrm{A}}(\theta_2) \hookrightarrow_\pi \theta_i}{\theta_1 \ominus \theta_2 \hookrightarrow_\pi \bigvee \theta_i}\ [\text{TS-Ind}]$$

$$\frac{F_3 \ominus \theta \hookrightarrow_\pi F_4 \qquad F_3 \sqsubseteq_\pi F_1 \qquad F_2 \sqsubseteq_\pi F_4}{F_1 \ominus \theta \hookrightarrow_\pi F_2}\ [\text{TS-Conseq}] \qquad \frac{F_1 \ominus \theta \hookrightarrow_\pi F_2 \qquad \theta' \sqsubseteq_\pi \theta}{F_1 \ominus \theta' \hookrightarrow_\pi F_2}\ [\text{TS-Trans}]$$

Fig. 14. Trace subtraction rules

Theorem 2 (Soundness of Trace Subtraction). *For all π, F, F', θ, s and ρ, given $F \ominus \theta \hookrightarrow_\pi F'$ it means that if $s, \rho \models \pi \wedge (\theta \cdot F')$ then $s, \rho \models \pi \wedge F$.*

Proof. *By induction on the derivation of $F \ominus \theta \hookrightarrow_\pi F'$.*

4.4 Soundness of Forward Reasoning

We here define the soundness of the forward rules and highlight the key lemmas. Theorem 3 presents the soundness for the generalized forward rules, where rules like $\{\pi\}\ e\ \{\Delta\}$ are generalized into the form of $\{(\pi\ ;\epsilon\ ;\ \_^\star)\}\ e\ \{\Delta\}$. The soundness states that for any given expression e, starting from a concrete model which satisfies the pre-state, when e evaluates to a value v, the resulting concrete model satisfies one of the concluded post-summaries, which guarantees that all the forward rules soundly over-approximating e's behavior and strengthening the final future conditions.

Theorem 3 (Soundness of the Generalised Forward Rules). *For all e, π, θ_1, F, F', s_1, s_2, ρ_1, ρ_2, v, given $\{(\pi\ ;\theta_1\ ;F)\}\ e\ \{\Delta\}$, $[s_1, \rho_1, F, e] \longrightarrow [s_2, \rho_2, F', v]$, and $s_1, \rho_1 \models \pi \wedge \theta_1$, there exists $(\pi'\ ;\theta_2\ ;F'') \in \Delta$ such that $(s_2 +\!\!+ res = v), \rho_2 \models \pi' \wedge \theta_2$ and $F'' \sqsubseteq_{\pi'} F'$.*

Proof. *By induction on the structure of e, and applying Lemma 1 and Lemma 2 when proving the sequencing rule [FV-Seq] and the structural rule [FV-Struct].*

Lemma 1 (Strengthening the Future Conditions from the Instrumented Semantics). *For all,* s_1, s_2, ρ_1, ρ_2, F_1, F_2, F_3, v,
given $[s_1, \rho_1, F_1, e] \longrightarrow [s_2, \rho_2, F_2, v]$ *and* $F_3 \sqsubseteq_{Pure(s_1)} F_1$,
there exists F_4, *such that,* $[s_1, \rho_1, F_3, e] \longrightarrow [s_2, \rho_2, F_4, v]$ *and* $F_4 \sqsubseteq_{Pure(s_2)} F_2$.

Proof. *By induction on* $[s_1, \rho_1, F_1, e] \longrightarrow [s_2, \rho_2, F_2, v]$.

Lemma 2 (Approximating the Concrete Trace Subtraction). *For all* F_1, θ, F_2, s, ρ, *given* $F_1 \ominus \theta \hookrightarrow_\pi F_2$ *and* $s, \rho \models \pi \wedge \theta$, *then exists* F_3 *such that* $F_1 \ominus_{lin} \rho \hookrightarrow_{Pure(s)} F_3$ *and* $F_2 \sqsubseteq_{Pure(s)} F_3$.

Proof. *By induction on* $F_1 \ominus \theta \hookrightarrow_\pi F_2$.

Lemma 1 states that in the instrumented semantics, starting with a stronger future condition leads to stronger resulting future conditions. In Lemma 2, we use $\ominus_{lin}$ to denote subtracting a linear sequence of events ρ from a given future condition without any approximation. It states that the trace subtraction $\ominus$ introduces approximation and results in stronger residues.

Table 1. Selected propositions for reasoning future conditions

$\theta_1 \wedge \theta_2 \leftrightarrow \theta_2 \wedge \theta_1$	(1)	$\theta_1 \wedge (\theta_2 \wedge \theta_3) \leftrightarrow (\theta_1 \wedge \theta_2) \wedge \theta_3$	(2)
$\bot \cdot \theta = \theta \cdot \bot \rightarrow \bot$	(3)	$(\theta_1 \cdot \theta_2 \neq \bot) \rightarrow (\theta_1 \neq \bot) \wedge (\theta_2 \neq \bot)$	(4)
$\_^\star \sqsubseteq_\pi \theta \rightarrow (\theta = \_^\star)$	(5)	$(\theta_1 \wedge \theta_2 \neq \bot) \rightarrow (\theta_1 \neq \bot) \wedge (\theta_2 \neq \bot)$	(6)
$(\_^\star \ominus \mathrm{A} \hookrightarrow_\pi \theta) \rightarrow (\theta = \_^\star)$	(7)	$\theta_1 \sqsubseteq_\pi \theta_2 \rightarrow (\theta_2 \sqsubseteq_\pi \theta_3 \rightarrow \theta_1 \sqsubseteq_\pi \theta_3)$	(8)
$(\_^\star \ominus_{lin} \mathrm{A} \hookrightarrow_\pi \theta) \rightarrow (\theta = \_^\star)$	(9)	$\Lambda_i^{B_1 \cup B_2}(\dots) \leftrightarrow \Lambda_i^{B_1}(\dots) \wedge \Lambda_i^{B_2}(\dots)$	(10)
$\Lambda_i^B(\bigvee(\pi \wedge \_^\star)) \rightarrow \_^\star$	(11)	$\Lambda_i^{\{v\}}(\bigvee(\pi \wedge \theta)) \rightarrow (\bigvee(\pi \wedge \theta))[v/i]$	(12)

Additionally, Table 1 presents a set of propositions essential for improving completeness. For example, the trace subtraction for the predicate in Fig. 7 is performed after applying propositions (10) and (12), detailed in Fig. 15.

$$
\begin{array}{lll}
 & pred_f([0..5), arr) \ominus free(arr[0]) & (Table\,1\text{-}10) \\
\hookrightarrow_{true} & (pred_f(0, arr) \wedge pred_f([1..5), arr)) \ominus free(arr[0]) & (Table\,1\text{-}11) \\
\hookrightarrow_{true} & (\mathcal{F}(free(arr[0])) \ominus free(arr[0])) \wedge pred_f([1..5), arr) & (Definition\ 4, 5) \\
\hookrightarrow_{true} & \_^\star \wedge pred_f([1..5), arr) \quad \hookrightarrow_{true} \quad pred_f([1..5), arr) &
\end{array}
$$

Fig. 15. Trace subtraction example for predicates

5 Evaluation and Case Studies

We formalize the soundness proofs presented in this work using Coq, publicly available from our artifact [18]. The Coq development comprises approximately

2,300 lines of code (LoC). We also prototype our system into an automated verification tool, implemented in 3,600 LoC of OCaml, demonstrating its applicability in verifying real-world programs. Experiments were done on a MacBook with a 2.6 GHz 6-Core Intel i7 processor. The source code and the evaluation benchmark are openly accessible [18].

Experiment Setup. Table 2 categorizes APIs by functionality and specifies the future conditions required for their safe usage. The table organizes these APIs into six categories (1-6), covering operations such as file I/O, thread synchronization, memory management, and more. These conditions were manually derived from official documentation and common usage patterns. In particular, we treat the *unchecked return value* (URV) and *null-pointer dereference* (NPD) vulnerabilities to be a critical use case for future conditions. URVs and NPDs occur when a program fails to validate the return value of a function, potentially leading to crashes or undefined behavior. To mitigate such risks, developers must ensure that all function return values are checked and handled appropriately.

Table 2. Selected APIs with their specifications in different usage contexts

Category	Example APIs	Future Conditions
1. File Ops	fopen, open	Finally to close the file descriptor
	fclose, close	Globally do not access the file descriptor
		Read-only files cannot be written to
2. Threads	pthread_create	Finally to pthread_join or detach the thread
	pthread_mutex_lock	Finally to pthread_mutex_unlock
3. Memory	free	Globally do not access the pointer
	malloc	Finally free the new pointer
	realloc	Globally the old pointer is not accessed & finally free the new pointer
4. Sockets	socket	Finally to close the socket
5. Database	sqlite3_open	Finally to sqlite3_close the connection
6. URV/NPD	fgets, gethostbyaddr	Check the return value immediately after calls

Experimental Results. Our evaluation uses real-world C programs sourced from: (i) the CWE database (containing diverse vulnerability types); (ii) API usage tutorials (demonstrating correct practices); and (iii) GitHub repositories (featuring real-world usage of critical APIs). The benchmark contains 51 manually verified API misuse violations, serving as ground truth for evaluation. Results are summarized in Table 3, with the following metrics: **PrimS** is the number of primitive specifications (manually provided, avg. 4.5 LoC/spec), **InferredS** is the number of inferred specifications (equals to the number of analyzed function declarations), **InferredInv** is the number of inferred trace invariants, **Report/-Exp.** stands for reported violations/ground truth violations, and **Time** records the total verification time in seconds. Our evaluation spans 1721 LoC across six

Table 3. Experimental Results

Category	LoC	PrimS	InferredS	InferredInv	Report/Exp.	Time(s)
1	675	8	30	7	14/12	13.66
2	330	4	25	1	4/4	0.49
3	409	6	30	12	26/24	6.60
4	103	2	6	1	3/3	0.50
5	109	4	6	0	4/4	0.50
6	95	10	4	0	4/4	0.03
Total	1721	34	101	21	55/51	21.78

distinct categories, with verification and inference applied to 101 functions. The tool reports 55 violations, including all 51 manually verified true positives and 4 false positives (incorrectly flagged safe code). Total verification time is 21.78 seconds, with processing time scaling proportionally to the program length and the number of primitive specifications.

```
1 void false_positive1() {
2   int** ptr1= malloc(4);
3   int*  ptr2= malloc(4);
4   *ptr1 = ptr2;
5   free(*ptr1);
6   free(ptr1); }
7 False positive: Memory Leak!
```

Fig. 16. False positive example for (ii)

```
1 void false_positive2
2       (const char* filename) {
3   int r = unlink(filename);
4   if (r == 0) {
5     int fd=fopen(filename,"r");
6     assert(fd == NULL);}}
7 False positive: Unclosed File!
```

Fig. 17. False positive example for (iii)

Expressiveness Limitations. The observed false positives arise from three limitations: (i) trace invariant inference fails under non-structured control flow, e.g., goto statements; (ii) incomplete modeling for memory usage; and (iii) limited to the local view of the file system. We next illustrate (ii) and (iii) with concrete examples. As shown in Fig. 16, the pointer `ptr1` points to the address of `ptr2` (assigned at line 4). In this code, freeing `*ptr1` (the content of `ptr1`) and then `ptr1` itself is safe and correctly avoids memory leaks. However, the deployed pure arithmetic logic fails to recognize the *points-to* relationship established at line 4. As a result, it incorrectly reports a memory leak, suggesting that `ptr2` is never freed – even though it is freed via `*ptr1`. This can be mitigated by extending the basic logic with *points-to* relations. As demonstrated in Fig. 17, the `unlink` function removes a file from the filesystem when it returns 0. Consequently, the subsequent call to `fopen` at line 5 will always fail (returning `NULL`), making the code correct – the assertion at line 6 will always succeeds. However, because our tool does not model `unlink`'s side effects – specifically, that future `fopen` calls on the same filename must fail – it incorrectly reports a file descriptor leak. Tracking external system for file states would resolve this limitation. Despite these limitations, our framework remains the first *sound* verification sys-

tem capable of detecting a broad range of violations via temporal logic encoding – providing a foundation for these aforementioned extensions.

Case Study: Inter-procedural Analysis & Conditional Violation. Fig. 18 presents a non-trivial double-free example, drawn from prior works [13, 23]. The *foo* function takes a pointer p and returns either a newly allocated pointer or the existing pointer $p{\to}f$, based on the value of $p{\to}flag$. In *main* function, a local *st* structure is created, memory is allocated for $p.f$, and foo is called. The potential double-free occurs because: if $p.flag$ is false, q and $p.f$ point to the same memory (line 4). Both q and $p.f$ are then freed (lines 10, 11), causing a double-free when $p.flag$ was false (line 11). This example is non-trivial because it requires a precise inter-procedural analysis and path sensitivity.

The verification proceeds as follows: Lines 2 and 7 initialize the program states using existentially quantified variables. Lines 3 and 4 reason about the two branches of the conditional and creates the existential variable l for the newly allocated heap address. By integrating the results, line 5 leads to a disjunctive

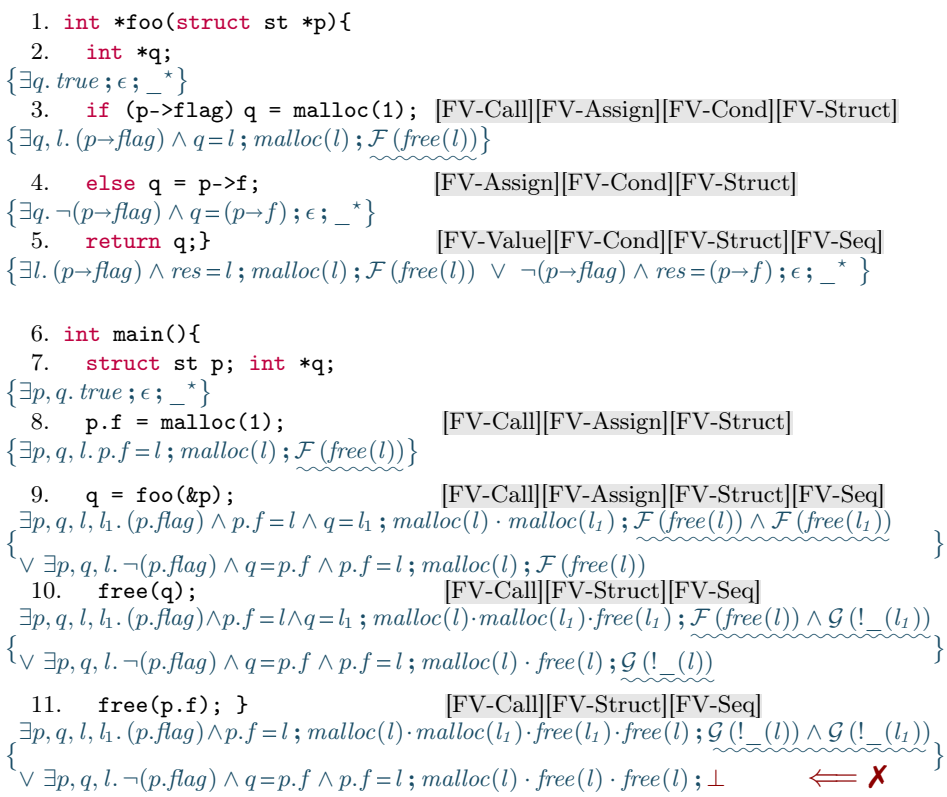

Fig. 18. Inter-procedural analysis for detecting a conditional double free violation

post-summary which says that when $p{\rightarrow}\mathit{flag}$ is true, the program returns a newly allocated heap location l, and there is a future condition to finally free l; when $p{\rightarrow}\mathit{flag}$ is false, it returns $p{\rightarrow}f$ and have no meaningful future conditions.

When calling the function *foo*, line 9 retrieves its specification (obtained after line 5), renames all its existential variables using fresh variables, and composes the instantiated specification into the current summaries. In line 10, under the condition $\neg(p.\mathit{flag})$, freeing q updates the propagated future conditions from $\mathcal{F}(\mathit{free}(l))$ to $\mathcal{G}(!\_(l))$. Subsequently, in line 11, when freeing $p.f$, it detects the double free under the pure constraint $\pi = (\exists p, q, l.\, \neg(p.\mathit{flag}) \wedge q{=}p.f \wedge p.f{=}l)$ – the following trace subtraction step results in *false*: $\mathcal{G}(!\_(l)) \ominus \mathit{free}(p.f) \hookrightarrow_{\pi} \bot$.

6 Related Work

Statically Checked Contracts. Typestate systems [24], recently revisited in various forms [10, 17, 26, 2], may also be used to a similar outcome as future conditions. These systems associate an object's state with its type, enforcing valid operation sequences based on that state, which is useful in object-oriented programming. Both typestates and future conditions model the correct execution order of operations on entities. However, typestates operate on objects, while future conditions operate on program variables. Additionally, implementing typestate systems often requires extensive boilerplate code. For instance, developers must meticulously define state transitions in a global view and manage associated types, increasing complexity in code maintenance and readability. In contrast, future conditions offer a more concise and modular way to specify and verify temporal properties without requiring explicit global state transitions. Their modularity stems from the fact that each future condition encapsulates temporal constraints specific to its associated function, independent of any global state.

Rust's type system [1] primarily enforces memory safety (ownership, borrowing, lifetimes), while the *typestate pattern* extends this by encoding state transitions into the type system, ensuring operations are only valid in certain states (e.g., a `File<Open>` can be read, but a `File<Closed>` cannot). However, typestates do not explicitly handle temporal properties (e.g., "eventually, this operation must happen"). Such properties must instead be verified separately against the global state transitions. Our approach unifies and extends both typestate systems and Rust's type system by: encoding memory safety using LTL formulas and enabling temporal reasoning (e.g., "this resource remains valid until condition π holds") alongside memory safety. Lastly, effect systems are also closely related, as they constrain which effects can be performed. While ordinary effect systems do not consider the order of effects, sequential effect systems [25, 12, 22] do. However, they follow the form of pre-/post-conditions, which is inadequate for describing future conditions in a modular style.

Dynamically Checked Contracts. Trace contracts [16] are for specifying and verifying properties of sequences of function calls and returns in the Racket programming language. They allow developers to define predicates over the sequence

of values that flow through function calls, enabling the detection of violations of expected behaviors across multiple function invocations. By monitoring programs at run time, trace contracts are able to take advantage of the precision that run-time checking offers, which possibly goes beyond statically decidable properties. Similar to typestate systems, trace contracts describes a global view of the protocols, while future conditions provide a more modular and local view of each effectful operation. Moreover, apart from the run-time overhead and increased resource consumption, trace contracts has limited expressiveness for imperative assignments and effectful loops, which are now supported in our verification framework.

Term Rewriting Regular Expressions. Given two regular expressions θ_1 and θ_2, inclusion ($\theta_1 \sqsubseteq \theta_2$) is typically decided using a term rewriting system (TRS), which iteratively checks derivatives [4]. This process terminates because the set of derivatives is finite, and cycles are efficiently detected using memorization (Γ). Prior works [22, 20, 19, 21, 6, 3, 11, 9] show that TRSs can be more efficient than automata-based techniques, as it avoids costly translations and enables early rejection of invalid inclusions. In this work, we extend derivative-based techniques to trace subtraction. Rather than checking for inclusion, our approach focuses on computing the future condition obligations for the next states.

Bug Finding at Scale. While Pulse [14] (based entirely on separation logic) and ProveNFix [23] (grounded in temporal logic) both use specification inference to detect memory and resource violations, our work generalizes both solution and yields a more comprehensive solution. Unlike Pulse, we eliminate the need for implicit side-checks, and compared to ProveNFix, we avoid repeated analysis while maintaining full temporal violation detection capabilities.

7 Conclusion

Future conditions extend traditional pre- and post-conditions by specifying temporal behaviors and states that emerge after a function call completes. This work tackles the central question: "*How can we efficiently and soundly reason about future conditions using temporal logic?*" We propose a compositional verification framework that propagates future conditions independently of contextual code. Our solution introduces novel trace inclusion and trace subtraction mechanisms to facilitate a more efficient propagation of future conditions, where they are processed once per method declaration. Additionally, we present a approach for verifying effectful loops using trace and future condition predicates. The soundness of our system is formally proven, and its practicality is demonstrated through experimental results and non-trivial case studies.

8 Acknowledgments

This research is supported by the Ministry of Education, Singapore, under its MOE Academic Research Fund Tier 3 (RIE2025) (MOE Award No: MOET-MOET32021-0001), and by the Ministry of Education, Singapore, under the

Academic Research Fund Tier 1 (FY2023). We thank anonymous reviewers for their insightful comments, which led to improvements in the paper presentation.

Data Availability

The source code of the tool, the dataset, proofs, are available from [18].

References

1. The Rust Programming Language. http://rust-lang.org, 2017.
2. J. Aldrich, J. Sunshine, D. Saini, and Z. Sparks. Typestate-oriented programming. In S. Arora and G. T. Leavens, editors, *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, pages 1015–1022. ACM, 2009.
3. M. Almeida, N. Moreira, and R. Reis. Antimirov and mosses's rewrite system revisited. *Int. J. Found. Comput. Sci.*, 20(4):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. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291–319, 1996.
6. V. M. Antimirov and P. D. Mosses. Rewriting extended regular expressions. *Theor. Comput. Sci.*, 143(1):51–72, 1995.
7. CWE. Cwe-416: Use after free. https://cwe.mitre.org/data/definitions/416.html.
8. L. M. de Moura and N. S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.
9. D. Hovland. The inclusion problem for regular expressions. *J. Comput. Syst. Sci.*, 78(6):1795–1813, 2012.
10. C. Jaspan and J. Aldrich. Checking framework interactions with relationships. In S. Drossopoulou, editor, *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings*, volume 5653 of *Lecture Notes in Computer Science*, pages 27–51. Springer, 2009.
11. M. Keil and P. Thiemann. Symbolic solving of extended regular expression inequalities. In V. Raman and S. P. Suresh, editors, *34th International Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2014, December 15-17, 2014, New Delhi, India*, volume 29 of *LIPIcs*, pages 175–186. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2014.
12. E. Koskinen and T. Terauchi. Local temporal reasoning. In T. A. Henzinger and D. Miller, editors, *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, pages 59:1–59:10. ACM, 2014.

13. J. Lee, S. Hong, and H. Oh. Memfix: static analysis-based repair of memory deallocation errors for C. In G. T. Leavens, A. Garcia, and C. S. Pasareanu, editors, *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 95–106. ACM, 2018.
14. Meta. Infer static analyzer. https://github.com/facebook/infer, 2025.
15. B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.
16. C. Moy and M. Felleisen. Trace contracts. *J. Funct. Program.*, 33, 2023.
17. R. Pucella and J. A. Tov. Haskell session types with (almost) no class. In A. Gill, editor, *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*, pages 25–36. ACM, 2008.
18. Y. Song. Benchmark and Source Code. https://zenodo.org/records/16690276, 2025.
19. Y. Song and W. Chin. Automated temporal verification of integrated dependent effects. In S. Lin, Z. Hou, and B. P. Mahony, editors, *Formal Methods and Software Engineering - 22nd International Conference on Formal Engineering Methods, ICFEM 2020, Singapore, Singapore, March 1-3, 2021, Proceedings*, volume 12531 of *Lecture Notes in Computer Science*, pages 73–90. Springer, 2020.
20. Y. Song and W. Chin. A synchronous effects logic for temporal verification of pure esterel. In F. Henglein, S. Shoham, and Y. Vizel, editors, *Verification, Model Checking, and Abstract Interpretation - 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17-19, 2021, Proceedings*, volume 12597 of *Lecture Notes in Computer Science*, pages 417–440. Springer, 2021.
21. Y. Song and W. Chin. Automated verification for real-time systems - via implicit clocks and an extended antimirov algorithm. In S. Sankaranarayanan and N. Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part I*, volume 13993 of *Lecture Notes in Computer Science*, pages 569–587. Springer, 2023.
22. Y. Song, D. Foo, and W. Chin. Automated temporal verification for algebraic effects. In I. Sergey, editor, *Programming Languages and Systems - 20th Asian Symposium, APLAS 2022, Auckland, New Zealand, December 5, 2022, Proceedings*, volume 13658 of *Lecture Notes in Computer Science*, pages 88–109. Springer, 2022.
23. Y. Song, X. Gao, W. Li, W. Chin, and A. Roychoudhury. ProveNFix: Temporal property-guided program repair. *Proc. ACM Softw. Eng.*, 1(FSE):226–248, 2024.
24. R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.*, 12(1):157–171, 1986.
25. R. Tate. The sequential semantics of producer effect systems. In R. Giacobazzi and R. Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 15–26. ACM, 2013.
26. R. Wolff, R. Garcia, É. Tanter, and J. Aldrich. Gradual typestate. In M. Mezini, editor, *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings*, volume 6813 of *Lecture Notes in Computer Science*, pages 459–483. Springer, 2011.