ProveNFix: Temporal Property-Guided Program Repair

YAHUI SONG, National University of Singapore, Singapore
XIANG GAO, Beihang University, China
WENHUA LI, National University of Singapore, Singapore
WEI-NGAN CHIN, National University of Singapore, Singapore
ABHIK ROYCHOUDHURY, National University of Singapore, Singapore

Model checking has been used traditionally for finding violations of temporal properties. Recently, testing or fuzzing approaches have also been applied to software systems to find temporal property violations. However, model checking suffers from state explosion, while fuzzing can only partially cover program paths. Moreover, once a violation is found, the fix for the temporal error is usually manual. In this work, we develop the first compositional static analyzer for temporal properties, and the analyzer supports a proof-based repair strategy to fix temporal bugs automatically. To enable a more flexible specification style for temporal properties, on top of the classic pre/post-conditions, we allow users to write a *future*-condition to modularly express the expected behaviors after the function call. Instead of requiring users to write specifications for each procedure, our approach automatically infers the procedure's specification according to user-supplied specifications for a small number of primitive APIs. We further devise a term rewriting system to check the actual behaviors against its inferred specification. Our method supports the analysis of 1) memory usage bugs, 2) unchecked return values, 3) resource leaks, etc., with annotated specifications for 17 primitive APIs, and detects 515 vulnerabilities from over 1 million lines of code ranging from ten real-world C projects. Intuitively, the benefit of our approach is that a small set of properties can be specified once and used to analyze/repair a large number of programs. Experimental results show that our tool, ProveNFix, detects 72.2% more true alarms than the latest release of the Infer static analyzer. Moreover, we show the effectiveness of our repair strategy when compared to other state-of-the-art systems — fixing 5% more memory leaks than SAVER, 40% more resource leaks than FootPatch, and with a 90% fix rate for null pointer dereferences.

CCS Concepts: • **Software and its engineering → Software verification and validation**; **Software testing and debugging**; • **Theory of computation** → *Program specifications*; *Modal and temporal logics*.

Additional Key Words and Phrases: Program Analysis, Automated Program Repair, Future-Conditions

ACM Reference Format:
Yahui Song, Xiang Gao, Wenhua Li, Wei-Ngan Chin, and Abhik Roychoudhury. 2024. ProveNFix: Temporal Property-Guided Program Repair. *Proc. ACM Softw. Eng.* 1, FSE, Article 11 (July 2024), 22 pages. https://doi.org/10.1145/3643737

1 INTRODUCTION

Finding temporal logic property violations is typically accomplished by well-known reactive system verification methods like model checking [7]. The common restrictions of model checking are that it assumes that all the procedures used are available, it usually handles bounded state spaces, and it

Authors' addresses: Yahui Song, National University of Singapore, Singapore, Singapore, yahui_s@nus.edu.sg; Xiang Gao, Beihang University, Beijing, China, xiang_gao@buaa.edu.cn; Wenhua Li, National University of Singapore, Singapore, Singapore, liwenhua@comp.nus.edu.sg; Wei-Ngan Chin, National University of Singapore, Singapore, Singapore, chinwn@comp.nus.edu.sg; Abhik Roychoudhury, National University of Singapore, Singapore, Singapore, abhik@comp.nus.edu.sg.

ACM 2994-970X/2024/7-ART11
https://doi.org/10.1145/3643737

suffers from the "state explosion problem" [8]. Beyond model checking, testing [25] or fuzzing [26] approaches have also been applied to software systems capturing real-world implementations to find temporal logic property violations. While effectively finding bugs, they rely on higher-quality test suits or longer execution times to achieve better code coverage. Static analyzers can help mitigate the above problems, but no existing static analyzer is specifically designed for analyzing temporal properties. To allow systematic code coverage and effectively finding bugs, we are interested in developing the first compositional temporal-property-based static analyzer, where each part of the program is analyzed locally and independently of the global context. It is known that breaking the large analysis of a whole program into small analyzes of its procedures gives us the ability to scale independently of the size of the analyzed code [35]. Here, we highlight three main difficulties in building a compositional static analyzer to detect temporal property violations:

(1) To check program behaviors against given properties, existing works [7, 25, 26] rely on inclusion checkers from *deterministic finite automaton*. However, the automata-based approach is not only complex but also prevents the analysis from being modular [1] due to the lack of high-level compositional patterns for hierarchical design;

(2) Having all the procedures in question, the formal specifications (abbreviated using specs from here) of each procedure in question is not always available, and writing specs for them is unnecessarily tedious and challenging. It is worth mentioning that it is not new for static analyzers to automatically generate specs. For example, the Facebook's Infer tool [5] utilizes bi-abduction [23] to infer pre/post specs from bare code, given the specs for the primitives at the base level of the code. Hence, the human does not need to write pre/post-conditions for all the procedures, which is the key to achieving a high level of automation. However, simply developing a bi-abduction for temporal properties is not sufficient for temporal property analysis;

(3) The classic pre/post-conditions only provide constraints for behaviors before the function call, and behaviours expected from the current function call, respectively, but we cannot easily express behaviors after executing the function call, such as temporal constraints like *"opening a read-only file should not be followed by any writing operations"* or *"some meaningful operations can only happen if the return value of loading the certificate is positive"*.

To solve the above-mentioned problems, this paper first introduces *future-conditions* to express constraints for behaviors after the function calls have finished. Together with the pre/post-conditions, a triplet style spec modularly and expressively captures a usage protocol for functions in concern. We further propose a *compositional temporal property static analyzer*, which automatically infers specs for each procedure and utilizes a term rewriting system as the back-end solver for proving temporal logic formulae inclusions.

Moreover, the proposed future-condition and the compositional analysis also advance automated program repair of temporal property violations. Program analysis-based repair has been previously shown to be effective in fixing various bugs. For example, FootPatch [41] deploys Infer [5], and fixes bugs related to resource leaks, memory leaks, and null dereferences, using templated repairs based on *separation logic*. MemFix [24] deploys a *typestate analysis* for small programs, fixing memory bugs, including memory leak, use-after-free, and double-free. The state-of-the-art tool SAVER [17] has targeted the same set of memory bugs and has supported the generation of conditional patches by constructing a full *object flow graph* for each given program. However, existing techniques either only support templated patches by inserting statements [41], or they cannot handle the generalized bug types such as *unchecked return values* and customized allocators/deallocators [17, 24]. In this work, we leverage the expressiveness of temporal logic and derive a bi-directional fault localization that is steered by future-conditions to compose safe patches for various bug types.

[1]Each procedure declaration is only analyzed once and procedures can be replaced by their already verified properties.

Putting it all together — this paper presents ProveNFix, a static analyzer guided by temporal properties and supported by a proof-based repair strategy for fixing detected bugs where possible. Our goal is to automatically detect and fix bugs from some classes of program bugs and enable a modular and *program-independent* analysis/repair. An extensive evaluation shows that ProveNFix can fix various bug types, including null pointer dereferences, resource leak, and memory bugs. In total, ProveNFix detects 515 and repairs 492 bugs for a more than 1 million lines of code benchmark within 15 minutes, which outperforms SAVER and FootPatch regarding the execution time and fix rate. Besides, ProveNFix can fix other bugs, such as unchecked return values and customized temporal bugs. Our main contributions are summarized as follows:

- We propose a novel feature, called *future-condition*, and formalize a modular and practical program analysis engine to effectively detect bugs with the help of minimal spec annotation.
- We propose a novel program repair approach, guided by constraints encoded using an expressive temporal spec called *IntRE*, an abbreviation for Integrated Regular Expressions.
- We prototype our proposal as a repair tool ProveNFix, on top of Infer front-end, to support large-scale C projects.
- We evaluate ProveNFix on an extensive benchmark and demonstrate that ProveNFix outperforms state-of-the-art tools in fixing a various types of bugs. The source code of ProveNFix and dataset are available at [46].

2 ILLUSTRATIVE EXAMPLES

This section presents a few examples to show the core idea and benefits of our approach.

2.1 Future Conditions

As an example demonstrated in Fig. 1, we write pre/post/future-conditions for the key functions and primitive APIs. Each spec Φ contains a set of tuples, i.e., $\bigvee(\pi \wedge \theta)$, where each disjoined cases has a pure formula π for the arithmetic constraints and a trace formula θ for the temporal constraints. We use $\overleftarrow{\mathcal{F}}$, $\mathcal{G}$, $\mathcal{F}$ to denote the temporal operators for "*past-time finally*", "*globally*" and "*finally*", respectively.

```
void free (void *ptr);
// pre: ptr≠null → ←F(malloc(ptr))
// post: (ptr=null ∧ ϵ) ∨ (ptr≠null ∧ free(ptr))
// future: true ∧ G (!_(ptr))

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

Fig. 1. Triplet specs for *free* and *malloc* APIs.

The precondition of `free` says that before freeing the (non-null) input pointer `ptr`, it has to have an event `malloc(ptr)` that occurred in its past history. We use $(\pi \rightarrow \theta)$ as a short-hand for $(\pi \wedge \theta \ \vee \ \neg\pi \wedge \_^{\star})$. Its postcondition says if `ptr` is null, then the function does nothing, i.e., ϵ; and if it is not null, its post-condition captures an event `free(ptr)`. Its future-condition enforces that after freeing it and within its lifetime, globally the identifier cannot be used by any events, i.e., $\mathcal{G}$ (`!_(ptr)`), which is used to prevent null pointer dereference, double-free and use-after-free. Similarly, the precondition of `malloc` requires its input argument `size` to be positive, and it can be called at any point of the execution, i.e., $\_^{\star}$. Its postcondition states that, when a pointer is successfully allocated, its postcondition captures an event `malloc(ret)`, where `ret` denotes the return value. Lastly, its future-condition enforces that the allocated pointer should be finally freed, which is used to prevent memory-leak. Although simple, these six lines of specs already cover the major memory usage bugs.

$$\frac{\begin{array}{c} nm(x^*) \mapsto (\Phi_{pre}, \Phi_{post}, \Phi_{future}) \in \mathcal{E} \\ \Phi \sqsubseteq [y^*/x^*]\Phi_{pre} \qquad \Phi'_{post} = [r/ret, y^*/x^*]\Phi_{post} \\ \mathcal{E} \vdash \{\Phi \cdot \Phi'_{post}\}\ e\ \{\Phi_e\} \qquad \Phi_e \sqsubseteq [r/ret, y^*/x^*]\Phi_{future} \end{array}}{\mathcal{E} \vdash \{\Phi\}\ r = nm(y^*);\ e\ \{\Phi'_{post} \cdot \Phi_e\}} \quad [FR\text{-}Call]$$

The above reasoning rule for function calls captures the essence of having future-conditions, where proof obligations are highlighted. Traditional Hoare-style forward reasoning rule for function calls works roughly as follows: it retrieves the callee *nm*'s spec from the environment $\mathcal{E}$ and if the current program state Φ entails the callee's instantiated precondition $[y^*/x^*]\Phi_{pre}$, it obtains the instantiated postcondition to be the extended program state, i.e., Φ'_{post}. Now, having the future spec, we extend the rule with one more proof obligation: the behavior of 'e', i.e., Φ_e, entails the callee's instantiated future-condition. Each proof obligation enforces constraints for code segments, here for the code before the current call and for the code after the call, i.e., 'e'.

2.2 Specification Inference and Interprocedural Analysis

Many existing tools [24] [17] perform program analysis via the call-strings technique [35], which blends interprocedural flow analysis with the analysis of intraprocedural flow, turning a whole program into a single flow graph. In this way, they split the interprocedural analysis into a pre-analysis phase, which gathers overestimated information about each procedure and followed by a global intraprocedural analysis. We decide to target a more accurate analysis by viewing procedures as collections of structured blocks and aim to establish input-output relations via the pre/post/future specs – where procedures can be replaced by their verified properties. This approach relates closely to most of the known techniques for program verification, and has the advantage of being rather simple and potentially admitting efficient and scalable implementations.

```
void wrap_malloc_I (int* ptr)
// future: ptr=null ∧ G (!_(ptr))
         ∨ ptr≠null ∧ F (free(ptr))
{ ptr = malloc (4); return;}

int* wrap_malloc_II ()
// future: ret=null ∧ G (!_(ret))
          ∨ ret≠null ∧ F (free(ret))
{ int* ptr = malloc (4); return ptr;}

int* wrap_malloc_III ()
// future: true ∧ F (free(ret))
{ int* ptr = malloc (4);
  if (ptr == NULL) exit(-1);
  return ptr;}

int* wrap_malloc_IV ()
// future: true ∧ _*
{ int* ptr = malloc (4);
 + if (ptr != NULL) free(ptr); // a repair
  return NULL;}
```

Fig. 2. Four kinds of malloc wrappers and their inferred future-conditions.

Specification Inference. Based on the primitive specs defined in Fig. 1, spec inference allows us to generate specs for bigger code blocks, which make use of the `malloc` and `free` primitives. Here, we use the examples shown in Fig. 2, to demonstrate how ProveNFix propagate future-conditions for customized memory allocations in different scenarios. For the first two cases, i.e., `wrap_malloc_I` and `wrap_malloc_II`, the future-conditions have been associated with their input and return pointers, respectively; both contain disjunctive cases to distinguish the behaviors depending on whether the memory is successfully allocated or not. For the third case, the future-condition for `wrap_malloc_III` is no longer a disjunctive form because the program terminates when the pointer is null, and in case it returns, it must be a non-null pointer. For the last case, it dynamically allocates

memory to a pointer, but there is no deallocation within its lifetime, so ProveNFix reports a bug in this case and generates a conditional patch to fix the bug. In the meantime, `wrap_malloc_IV` does not have any meaningful future-condition, i.e., it allows anything to happen afterward.

Interprocedural Analysis. Fig. 3 presents a more involved example, drawn from the prior work [24] and demonstrates a common failure point in existing repair techniques. Due to the possible aliasing between `q` and `p.f` by calling `foo` at line 10, there is a possible double-free error at line 12. We handle this program by first generating the post and future-condition for `foo`, shown as follows (assuming there is sufficient heap and malloc will never fail):

```
// post: (p->flag ∧ malloc(ret)) ∨ (!p->flag ∧ q=p->f ∧ ϵ)
// future: p->flag ∧ ℱ free(ret)
```

Next, when we reason about the `main` procedure, the future-conditions accumulated by the malloc at line 8 and by calling `foo` at line 9 enforce that the behaviors of lines 10-12 should satisfy the following spec:

$$\Phi_{10-12} = \texttt{(p->flag)} \wedge \mathcal{F}\ \texttt{free(p.f)} \wedge \mathcal{F}\ \texttt{free(q)} \vee \texttt{(!p->flag} \wedge \texttt{q=p->f)} \wedge \mathcal{F}\ \texttt{free(p.f)}$$

Then, after the free statement in line 10, `free`'s future-condition, i.e., $\texttt{true} \wedge \mathcal{G}(\texttt{!\_(q)})$, is violated by the second free statement in line 12, i.e.,

$$\texttt{(!p->flag} \wedge \texttt{q=p->f)} \wedge \texttt{free(p.f)@12} \not\subseteq \texttt{true} \wedge \mathcal{G}(\texttt{!\_(q)});$$

therefore, ProveNFix detects there is an error. When this assertion failed, the bi-directional constraint propagation computes the spec for line 10 to be: $\Phi_{10} = \texttt{p->flag} \wedge \mathcal{F}\ \texttt{free(q)}$. Finally, guided by Φ_{10}, ProveNFix synthesizes a patch by deleting the free statement at line 10 and inserting a conditional free statement at line 11, which fixes this error.

```
1  int *foo(struct st *p)
2  { int *q;
3    if (p->flag) q = malloc(1);
4    else q = p->f;
5    return q;}
6  int main() {
7    struct st p; int *q;
8    p.f = malloc(1);
9    q = foo(&p);
10   - free(q);
11   + if (p->flag) free(q);
12   free(p.f);} // double-free
```

Fig. 3. Fixing a double free bug [24].

```
1  static int swReactorProcess_reuse_port(swListenPort *ls)
2  { int sock = swSocket_create(ls->type);  // Here acquires a socket resource
3    if (sock < 0) return SW_ERR;
4    if (swSocket_bind(sock, ls->type, ls->host, ls->port) < 0){
5       + close(sock); // a repair
6        return SW_ERR;}
7    if (swSocket_is_stream(ls->type)) swSetNonBlock(sock);
8    ls->sock = sock; // Here is NOT a bug!
9    return swPort_listen(ls);}
```

Fig. 4. A false positive reported by Infer.

Reducing False Positives. We use the resource leak example in Fig. 4, detected in the Swoole project [32], to show that future-conditions help to reduce the false positives in practice. The first leak happens at line 6, where the code returns without releasing the socket. The repair is simply to insert a close statement at line 5 before returning. However, Infer reports another leak at line 9 because the program never releases the socket. This is a false positive because of the assignment at line 8, and the program is safe as long as '`swPort_listen`' or the caller of '`swReactorProcess_reuse_port`' releases '`ls->sock`' in the future. In our approach, ProveNFix manages to generate the specs for

swPort_listen, which does not release the socket; therefore, we generate a future-condition for swReactorProcess_reuse_port, defined as follows: ret≠SW_ERR ∧ $\mathcal{F}$ (close(ls->sock)), pushing the obligation to close the resource to its callers, avoiding this false positive.

2.3 Handling Generalized Bug Types

Many functions' return value indicates the success of their actions, which alerts the users whether or not to handle any bugs caused by that function. Ignoring the return values can cause the program to overlook unexpected states and conditions, leading to a crash or other unintended behaviors. One instance of unchecked return values (URV) is known as *null pointer dereference* (NPD). For example, as shown in Fig. 5, the function host_lookup takes an IP address, verifies that it is well-formed, and then looks up the hostname and copies it into a buffer [2]. If an attacker provides an address that does not resolve to a hostname, the call to gethostbyaddr, line 4 will return *null*. Since the code does not check the return value, a *null* pointer dereference would occur in the call to strcpy. While there are no complete fixes aside from conscientious programming, one potential mitigation is annotated in line 5, which inserts a conditional statement to exit if hp is *null*. We detect and repair such NPD bugs by having the following primitive spec:

```
1  void host_lookup (char *user_supplied_addr) {
2    validate_addr_form(user_supplied_addr);
3    addr = inet_addr(user_supplied_addr);
4    hp = gethostbyaddr(addr, sizeof(...), ...);
5  + if (hp==null) { exit(-1); } // a repair
6    strcpy(hostname, hp->h_name);}
```

Fig. 5. Null pointer dereference error, reported at CWE-476.

```
struct hostent *gethostbyaddr (...); // future: ret=null ∧ G (!_(ret))
```

which restricts that if the return value ret is *null*, globally ret cannot be used as a parameter in any events; and we let each pointer dereference generate a "deref" event; here, hp->h_name generates "deref(hp)". With these specs, ProveNFix automatically detects NPD bugs and generates patches, as provided in line 5. Notably, Infer cannot detect this bug because it does not support any inputs for primitive specs. Without loss of generality, the URV bugs we can handle are not limited to NPDs but also various other APIs used in different contexts, such as: fgets, returnChunkSize and pthread_mutex_lock etc. Furthermore, other than APIs from standard C libraries, the URV errors also often show in the applications of internet-facing protocols.

```
107     sock_obj->ssl = SSL_new(req->ctx);
108   + if (!sock_obj->ssl) {
109   +         fprintf(stderr, "SSL_new() failed\n");
110   +         return false;
111   + }
```

Fig. 6. An unchecked return value fixed in Github.

```
+ int t0 =
SSL_CTX_use_certificate(sslctx, crt);
+ if (t0 != 1) return NULL;
```

Fig. 7. Repair by inserting a temporary variable.

Fig. 6 presents a fix of an issue raised in the keepalived project [3], where function SSL_new() returns a pointer to an SSL object on success or null on error. However, the code shown didn't check the return value properly. There are other URV errors caused by not having any handlers, such as the example shown in Fig. 7 detected in the sslsplit project [4]. For such cases, our approach can produce patches by inserting temporary variables, here t0, which cannot be achieved by any

[2]The second example in CWE-476. https://cwe.mitre.org/data/definitions/476.html.
[3]Issue 1004: "Missing Error Check in function SSL_new()". https://github.com/acassen/keepalived/issues/1004.
[4]Issue 204: "Missing Error Check in function SSL_CTX_use_certificate()". https://github.com/droe/sslsplit/issues/224.

existing repair tools. Experimental results in Sec. 7.5 show that ProveNFix can be used to generate specs for these APIs from the source code, and these specs can be used to analyze the applications.

3 SYSTEM OVERVIEW AND LANGUAGES

System Overview. Fig. 8 shows the system overview, where the inputs and output of the system are directed by the fat arrows. ProveNFix takes a target program and a set of primitive specs written in *IntRE*; produces a bug report and safe patches at the end. Our technical contributions are captured in the rounded boxes: a Hoare-style forward reasoning which infers specs for procedures and generates constraints for code segments; the bi-directional constraint propagation for the buggy code segments; and the deductive synthesis to derive source code patches from existing ingredients and the spec pool. We use $\sqsubseteq$ [5] to denote the proof obligations between two *IntRE* formulae, and $\not\sqsubseteq$ to denote the failed assertions. The workflow of ProveNFix is as follows:

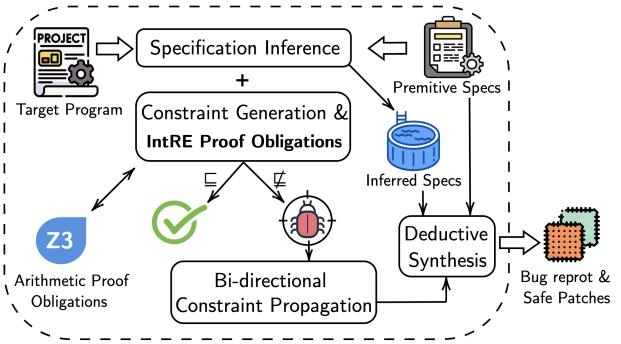

Fig. 8. System Overview.

(1) For each procedure, a set of well-defined forward reasoning rules (presented in Sec. 4) summarize the actual behavior of its body using *IntRE* formulae. In particular, they generate temporal constraints for program segments, dynamically aligning with the existing specs; in the meantime, they infer specs for the current procedure and add them into the spec pool;
(2) Assertions between actual behaviors against their specs are represented using proof obligations between *IntRE*. The proving is discharged by a back-end term rewriting system (presented in Sec. 5), which is an extended inclusion checker for regular expressions. While proving the inclusions, proof obligations for arithmetic constraints are discharged by the Z3 solver [11];
(3) Then, if any of the inclusions fails, it will be fed into the novel bi-directional spec propagation (presented in Sec. 6.1), to compute the spec for the *core* buggy code segments;
(4) Lastly, we use the deductive synthesis (presented in Sec. 6.2) to generate source-code patches. The patches that ProveNFix can generate include inserting/deleting code blocks, conditional patches, and inserting temporary variables for unhanded return values.

Our approach has two principal benefits: the constant effort of spec annotation without restricting the bug types and the highly reduced search space for patch generation. Intuitively, the events of our interests are prescribed in the primitive specs, and after the spec inference, the search space is effectively pruned to the code ingredients, which would generate effectful events. We show the experimental results in Sec. 7, and conclude in Sec. 9.

Target Language. We target an imperative, well-typed, call-by-value core language, defined in Fig. 9. A program $\mathcal{P}$ comprises a list of primitive specs $spec^*$, and procedure declarations $proc^*$. Here, we use the $*$ superscript to denote a finite list of items, for example, x^* refers to a list of variables, $x_1, \ldots, x_n$. Each procedure has a name nm, formal arguments x^*, and an expression-oriented body

[5]The inclusion relation between two *IntRE* specs $\Phi_1 \sqsubseteq \Phi_2$ is formally defined in Definition 5.

(*Program*)	$\mathcal{P}$	::=	$spec^*$; $proc^*$
(*Specifiction*)	$spec$	::=	$nm(x^*) \mapsto (\Phi_{pre}, \Phi_{post}, \Phi_{future})$
(*Procedure Declaration*)	$proc$	::=	$nm(x^*)\ \{e\}$
(*Binary Operations*)	b	::=	$bop(v_1, v_2)$
(*Expressions*)	e	::=	$\texttt{local}\ x; e \mid x{=}v \mid nm(x^*) \mid e_1; e_2 \mid \texttt{if}\ b\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid$ $\texttt{while}(b)\ e \mid \texttt{return}\ v \mid \texttt{assert}\ (\Phi_{pre}, \Phi_{future})$

$c \in \mathbb{Z} \cup \mathbb{B} \cup unit$ $\quad x, nm \in var$ $\quad$ (*Values*) $v ::= c \mid x \mid null$ $\quad bop \in \{=, >, <, \geq, \leq\}$

Fig. 9. Syntax of the target language.

e. Triplet specs are associated with procedure signatures. Simple values include constants c, ranging from integers, Boolean, and unit; program variables x; and null pointers *null*.

We use $bop(v_1, v_2)$ to represent atomic binary operations for values, including $=, >, <, \geq$ and $\leq$. Expressions include local variable declarations $\texttt{local}\ x; e$; assignments $x{=}v$; function calls $nm(x^*)$; sequential composition $e_1; e_2$; conditionals $\texttt{if}\ b\ e_1\ \texttt{else}\ e_2$; while loops $\texttt{while}(b)\ e$; return statements $\texttt{return}\ v$; and assertions, parameterized with Φ_{pre} and Φ_{future}, restricting the temporal behaviors before and after the assertion, respectively.

IntRE*, the Specification Language*. As defined in Fig. 10, *IntRE*, standing for Integrated Regular Expressions, denoted by Φ, contains a set of tuples (or conditioned traces per se) including disjointed pure formulae π and their corresponding event sequences θ. Traces comprise *false* ($\bot$); empty traces ϵ; singleton events I; sequences concatenations $\theta_1 \cdot \theta_2$; disjunctions $\theta_1 \vee \theta_2$; and the arbitrary times (zero or many, possibly infinite) repetition of a trace, constructed by a Kleene star $\theta^\star$. Singleton events are: parameterized events $\mathrm{A}(v)$; events with a specific label $\mathrm{A}(\_)$; negation of parameterized events $!\mathrm{A}(v)$; negation of events which make use of the value v, i.e., $!\_(v)$; the wild card _ matching to all the events; and event conjunctions $\mathrm{I}_1 \wedge \mathrm{I}_2$.

We use π to denote a pure formula which captures the simplified (decidable) Presburger arithmetic conditions on program inputs and local variables, where T and F denote *true* and *false* respectively. A term can be a simple value v, or simple computations of terms, $t_1{+}t_2$ and $t_1{-}t_2$.

(*IntRE*)	Φ	::=	$\bigvee(\pi \wedge \theta)$
(*Traces*)	θ	::=	$\bot \mid \epsilon \mid \mathrm{I} \mid \theta_1 \cdot \theta_2 \mid \theta_1 \vee \theta_2 \mid \theta^\star$
(*Events*)	I	::=	$\mathrm{A}(v) \mid \mathrm{A}(\_) \mid !\mathrm{A}(v) \mid !\_(v) \mid \_ \mid \mathrm{I}_1 \wedge \mathrm{I}_2$
(*Pure*)	π	::=	$T \mid F \mid bop(t_1, t_2) \mid \pi_1 \wedge \pi_2 \mid \pi_1 \vee \pi_2 \mid \neg\pi \mid \exists x.\pi$
(*Terms*)	t	::=	$v \mid t_1{+}t_2 \mid t_1{-}t_2$
(*Values*)	v	::=	$c \mid x \mid null$

Fig. 10. Syntax of the spec language, *IntRE*.

It is proven that the expressive power of regular expressions subsumes the classic linear temporal logic (LTL) formulae [43]. While ProveNFix allows both syntaxes, the underlying reasoning and proving are formalized using *IntRE*. We deploy a standard translation from LTL formulae to *IntRE*, shown in Appendix A.

4 FORWARD REASONING AND SPECIFICATION INFERENCE

This section addresses the details of constraint generation for code segments and spec inference for procedures; while they happen simultaneously, we present them separately for clarity.

4.1 Constraint Generation and Bug Reporting

Together with the rule [*FR-Call*] presented in Sec. 2.1, Fig. 11 formalizes a set of syntax-directed forward rules for the target language. Our Hoare-style reasoning is in the form of $\mathcal{E}, C \vdash \{\Phi\}\ e\ \{\Phi'\}$, where $\mathcal{E}$ is an environment, initialized by the given primitive specs and extended with the inferred specs along the way; C denotes the current procedure being analyzed. $\mathcal{E}$ and C are omitted when not needed. The meaning of the relation is: if Φ describes the behaviors triggered before executing e, then by executing e, Φ' describes the *extension* of the traces that will be triggered.

$$\frac{}{\vdash \{\Phi\}\ \texttt{return}\ v\ \{(ret{=}v){\wedge}\epsilon, 1\}}\ [FR\text{-}Return] \qquad \frac{}{\vdash \{\Phi\}\ x{=}v\ \{(x{=}v){\wedge}\epsilon, 0\}}\ [FR\text{-}Assign]$$

$$\frac{\Phi \sqsubseteq \Phi_{pre} \quad \vdash \{\Phi\}\ e\ \{\Phi_e\} \quad \Phi_e \sqsubseteq \Phi_{future}}{\vdash \{\Phi\}\ \texttt{assert}\ (\Phi_{pre}, \Phi_{future});\ e\ \{\Phi_e\}}\ [FR\text{-}Assert]$$

$$\frac{\vdash \{\Phi\}\ e_1\ \{\Phi_1\} \qquad \vdash \{\Phi \cdot \Phi_1\}\ e_2\ \{\Phi_2\}}{\vdash \{\Phi\}\ e_1; e_2\ \{\Phi_1 \cdot \Phi_2\}}\ [FR\text{-}Seq] \qquad \frac{\vdash \{b \wedge \Phi\}\ e^{\star}\ \{\Phi'\}}{\vdash \{\Phi\}\ \texttt{while}(b)\ e\ \{\neg b \wedge \Phi'\}}\ [FR\text{-}While]$$

$$\frac{\vdash \{b \wedge \Phi\}\ e_1\ \{\Phi_1\} \qquad \vdash \{\neg b \wedge \Phi\}\ e_2\ \{\Phi_2\}}{\vdash \{\Phi\}\ \texttt{if}\ b\ e_1\ \texttt{else}\ e_2\ \{\Phi_1 \cup \Phi_2\}}\ [FR\text{-}If\text{-}Else] \qquad \frac{\vdash \{\Phi\}\ e\ \{\Phi_e\}}{\vdash \{\Phi\}\ \texttt{local}\ x;\ e\ \{\exists x.\ \Phi_e\}}\ [FR\text{-}Local]$$

Fig. 11. Selected forward reasoning rules.

Notice that, the post-states of the basic rules, [*FR-Return*] and [*FR-Assign*] are associated with a completion code $k{\in}\{0, 1\}$, where when $k{=}0$, the reasoning can proceed; when $k{=}1$, the current procedure returns. Each completion code is initialized using 0, and only the return statements update it to 1 as delimiters, marking the end of the local procedure. Anything concatenated after states with non-zero completion codes will be abandoned. The completion code is essential when the compositional rules come in, such as [*FR-Seq*]. Starting from a pre-state Φ, rule [*FR-Seq*] firstly computes the behavior of e_1, denoted by Φ_1; then reasons about e_2 with the extended pre-state, i.e., $\Phi \cdot \Phi_1$; lastly the final result is a concatenation of Φ_1 and Φ_2. The concatenation between two singleton program states is formally defined in Definition 1.

Definition 1 (Program state Concatenation). *Given two singleton program states* $\Phi_1 = (\pi_1 \wedge \theta_1, k_1)$ *and* $\Phi_2 = (\pi_2 \wedge \theta_2, k_2)$*, we define:*

$$\Phi_1 \cdot \Phi_2 = (\pi_1 \wedge \theta_1,\ k_1)\ \textit{when}\ k_1{>}0,$$
$$((\pi_1 \wedge \pi_2) \wedge (\theta_1 \cdot \theta_2),\ k_2)\ \textit{when}\ k_1{=}0.$$

Next, rule [*FR-While*] computes the behaviors of loops by unfolding the loop body e many times, i.e., $e^{\star}$; and in practice, ProveNFix unfolds loops once to balance precision and efficiency. Rule [*FR-If-Else*] computes the behaviors from both branches by enforcing the pre-state with constraints b and $\neg b$, respectively; then, it disjunctively unions the results. Rule [*FR-Local*] creates an existential quantifier for the local variable x. For assertions parameterised with Φ_{pre} and Φ_{future}, rule [*FR-Assert*] creates proof obligations (as highlighted) for the precondition checking and the future-condition checking regarding the behavior of the rest of code. The proving of proof obligations is discharged by a term rewriting system, presented in Sec. 5. Each failed assertion is reported as a bug, and in this paper, we are interested in finding the *true bugs*, defined in Definition 2.

Definition 2 (Manifest True Bug [23]). *There exists a path from the local procedure declaration that leads to the bug, and for any value of the input, the bug occurs.*

Unchecked Return Value without Handlers. As the example demonstrated in Fig. 7, there are cases where unchecked return values are caused by having no handlers, also, the future-condition contains the pattern $(\pi \wedge \mathcal{G}(!\_(ret)))$, indicating an error state where the purpose of calling the function has failed, and the return value should not be used. The rule we present here, [*FR-Call-Handling*] (as a special case for [*FR-Call*]) is designed for such cases. It first generates a fresh variable to be the temporary handler, here t; then, as usual, checks the precondition, instantiates the postcondition. Lastly, it synthesizes the handling code based on the future-condition, and inserts the code as a patch. The synthesize algorithm is presented in Sec. 6.2.

$$\frac{\begin{array}{c} nm(x^*) \mapsto (\Phi_{pre}, \Phi_{post}, \Phi_{future}) \in \mathcal{E} \qquad (t \ is \ fresh) \\ \Phi \sqsubseteq [y^*/x^*]\Phi_{pre} \qquad \Phi'_{post} = [t/ret, y^*/x^*]\Phi_{post} \\ (\pi \wedge \mathcal{G}(!\_(ret))) \in \Phi_{future} \qquad \pi' = [t/ret]\pi \qquad synthesis(\pi' \wedge \mathcal{G}(!\_(t))) \end{array}}{\mathcal{E} \vdash \{\Phi\}\ nm(y^*)\ \{\Phi'_{post} \vee (\pi' \wedge \mathcal{G}(!\_(t)))\}}\ [\textit{FR-Call-Handling}]$$

4.2 Triplet Specification Inference

Relation $\mathcal{E} \models \mathcal{P}$ denotes the reasoning for program $\mathcal{P}$ starting with an environment $\mathcal{E}$. As shown in [*SI-Proc*], given any procedure declaration, the rule takes $\{T \wedge \epsilon, 0\}$ as the initial state, and reasons about the actual behavior Φ_{actual} of the procedure body. Being the strongest postcondition, Φ_{actual} is added as the inferred spec of $nm(x^*)$ into the environment. The notation $\mathcal{E}[C].post \mapsto \Phi$ means to associate the inferred spec Φ with C's postcondition, and same for pre and future.

$$\frac{\begin{array}{c} \mathcal{P} = nm(x^*)\{e\}\ ;\ \mathcal{P}' \qquad \mathcal{E}, nm(x^*) \vdash \{T \wedge \epsilon, 0\}\ e\ \{\Phi_{actual}\} \\ \mathcal{E}[nm(x^*)].post \mapsto \Phi_{actual} \qquad \mathcal{E} \models \mathcal{P}' \end{array}}{\mathcal{E} \models \mathcal{P}}\ [\textit{SI-Proc}]$$

$$\frac{\begin{array}{cc} \Phi'_{pre} \cdot \Phi \sqsubseteq \Phi_{pre} & \Phi_e \sqsubseteq \Phi_{future} \cdot \Phi'_{future} \\ \mathcal{E}[C].pre \mapsto \Phi'_{pre} & \mathcal{E}[C].future \mapsto \Phi'_{future} \end{array}}{\mathcal{E}, C \vdash \{\Phi\}\ \texttt{assert}\ (\Phi_{pre}, \Phi_{future});\ e\ \{\Phi_e\}}\ [\textit{SI-Assert}]$$

Moreover, in the rule [*SI-Assert*] (the spec inference version of [*FR-Assert*]), if the inclusions only hold when there exist frames Φ'_{pre} and Φ'_{future} needed, we add the frames as the inferred pre/future-conditions of the current procedure C into the environment. The exact inference applies to the inclusion checkings in rule [*FR-Call*] as well, and we omit it here for simplicity.

5 INCLUSION CHECKING

Given *IntRE* formulae Φ_1 and Φ_2, the back-end term rewriting system (TRS) decides if the inclusion $\Phi_1 \sqsubseteq \Phi_2$ is valid through an iterated process of checking inclusions of their *derivatives* [2].

Definition 3 (Derivatives). *Given any formal language S over a (finite) alphabet Σ and any string $u \in \Sigma^*$, the derivative of S with respect to u is defined as: $u^{-1}S = \{w \in \Sigma^* \mid u \cdot w \in S\}$.*

Definition 4 (Trace Entailment). *Given any trace formulae θ_1, θ_2 and any traces θ, the inclusion between θ_1 and θ_2 is recursively defined as: $\theta_1 \sqsubseteq \theta_2 \Leftrightarrow \forall \theta.\ (\theta^{-1}\theta_1 \sqsubseteq \theta^{-1}\theta_2)$.*

Based on Definition 3, we formally define the inclusion relation $\sqsubseteq$ between trace formulae in Definition 4, where the derivative function $\theta^{-1}\theta_1$ eliminates a trace θ from the head of θ_1 and returns what remains. Intuitively, proving the inclusion $\theta_1 \sqsubseteq \theta_2$ (presented in Sec. 5.2) amounts to checking whether all the possible traces in the antecedent θ_1 are legitimately allowed in the possible traces from the consequent θ_1. Next, we define the *IntRE* inclusion in Definition 5.

Definition 5 (*IntRE* Entailment). *Given any* IntRE *formulae* Φ_1 *and* Φ_2*, their inclusion* $\Phi_1 \sqsubseteq \Phi_2$ *is valid if and only if,* $\forall(\pi_2 \wedge \theta_2) \in \Phi_2.\ \exists(\pi_1 \wedge \theta_1) \in \Phi_1$ *such that* $\pi_1 \Rightarrow \pi_2$ *and* $\theta_1 \sqsubseteq \theta_2$*, where the pure formulae inclusions, denoted by '*$\Rightarrow$*', are discharged by the Z3 solver.*

Informally, proving the *IntRE* inclusion is to check that for all the disjoined cases specified in the consequent, there always exists a compatible case from the antecedent to form a valid trace inclusion. Then, when the consequent cannot find a witness from the antecedent, we trigger the patch generation process to synthesize conditional statements accordingly.

Therefore, we have the following top-level rules to decompose the disjunctions. In particular, [*TRS-Base*] triggers the including checking between traces by initialising the proof hypotheses using $\emptyset$, cf. Sec. 5.2. Rule [*TRS-Missing-Case*] triggers the repair process by synthesizing code that satisfies the required spec, cf. Sec. 6.2.

$$\frac{\pi_1 \Rightarrow \pi_2 \qquad \pi_1, \emptyset \vdash \theta_1 \sqsubseteq \theta_2}{\pi_1 \wedge \theta_1 \sqsubseteq \pi_2 \wedge \theta_2}\ [\textit{TRS-Base}] \qquad \frac{\exists(\pi_1 \wedge \theta_1) \in \Phi_1.\ (\pi_1 \wedge \theta_1 \sqsubseteq \pi_2 \wedge \theta_2) \qquad \Phi_1 \sqsubseteq \Phi_2}{\Phi_1 \sqsubseteq (\pi_2 \wedge \theta_2) \vee \Phi_2}\ [\textit{TRS-Disj}] \qquad \frac{\textit{synthesis}(\pi \wedge \theta)}{\emptyset \not\sqsubseteq \pi \wedge \theta}\ [\textit{TRS-Missing-Case}]$$

5.1 Auxiliary Functions

To facilitate the inclusion rules in Sec. 5.2, we provide the definitions and core implementations of the deployed auxiliary functions: [6] *Nullable*(δ), *First*(*fst*) and *Derivative*($\mathcal{D}$) respectively. Informally, the Nullable function $\delta(\theta)$ returns a Boolean value indicating whether θ contains the empty trace; the First function $fst(\theta)$ computes a set of possible initial events from θ; and lastly, the Derivative function $\mathcal{D}_{\mathrm{I}}(\theta)$ eliminates an event I from the head of θ and returns what remains. The subset relation $\mathrm{I} \subseteq \mathrm{J}$ means that, the set of events in I is a subset of the set of events in J.

Definition 6 (Nullable). *Given any sequence* θ*, we recursively define* $\delta(\theta)$ *as follows: (false for unmentioned constructs)*

$$\delta(\epsilon)=true \qquad \delta(\theta^\star)=true \qquad \delta(\theta_1 \cdot \theta_2)=\delta(\theta_1) \wedge \delta(\theta_2) \qquad \delta(\theta_1 \vee \theta_2)=\delta(\theta_1) \vee \delta(\theta_2)$$

Definition 7 (First). *Let* $fst(\theta)$ *be the set of initial events derivable from sequence* θ*.*

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

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

Definition 8 (Partial Derivative). *The partial derivative* $\mathcal{D}_I(\theta)$ *of trace* θ *w.r.t. an element* I *computes the effects for the left quotient,* $I^{-1}\theta$*, defined as follows:*

$$\mathcal{D}_I(\bot)=\mathcal{D}_I(\epsilon)=\bot \qquad \mathcal{D}_I(\theta^\star)=\mathcal{D}_I(\theta) \cdot \theta^\star \qquad \mathcal{D}_I(\theta_1 \vee \theta_2)=\mathcal{D}_I(\theta_1) \vee \mathcal{D}_I(\theta_2)$$

$$\mathcal{D}_I(\mathcal{J})=\begin{cases} \epsilon & \text{if } I \subseteq \mathcal{J} \\ \bot & \text{else} \end{cases} \qquad \mathcal{D}_I(\theta_1 \cdot \theta_2)=\begin{cases} (\mathcal{D}_I(\theta_1) \cdot \theta_2) \vee \mathcal{D}_I(\theta_2) & \text{if} \quad \delta(\theta_1)=true \\ \mathcal{D}_I(\theta_1) \cdot \theta_2 & \text{if} \quad \delta(\theta_1)=false \end{cases}$$

5.2 Inclusion Rules

Given the well-defined auxiliary functions above, we now present the key rewriting rules in Fig. 12 deployed in inclusion proofs. During the rewriting process, the inclusions are in the form of $\pi, \Gamma \vdash \theta_1 \sqsubseteq \theta_2$, a shorthand for: $\exists\theta.\ (\pi, \Gamma \vdash \theta \cdot \theta_1 \sqsubseteq \theta \cdot \theta_2)$. Here θ is the history traces from the

[6] The definitions are extended from [2], which were originally used for proving inclusions between regular expressions.

antecedent that have been used to match the traces from the consequent; π is the path constraint and Γ is the proof context, which contains a set of inclusion hypothesis.

Rule [*Prove*] is used when the antecedent has no first elements. Rule [*Reoccur*] is to prove an inclusion when there exist inclusion hypotheses in the proof context Γ, where we can soundly prove the current goal. One of the special cases of this rule is when the identical inclusion is shown in the proof context; we then prove it valid. Rule [*Unfold*] is the inductive step of unfolding the inclusions. Firstly, we make use of the auxiliary function *fst* to get all the possible initial events from the antecedent, H. 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 event $\mathrm{I} \in H$ and compute the derivatives (*next-state* formulae) of both the antecedent and consequent with respect to I. The proof of the original inclusion succeeds if all the derivative inclusions succeed.

$$\frac{fst(\theta_1)=\{\}}{\pi,\Gamma \vdash \theta_1 \sqsubseteq \theta_2}\ [Prove] \qquad \frac{(\theta_1 \sqsubseteq \theta_3) \in \Gamma \qquad (\theta_3 \sqsubseteq \theta_2) \in \Gamma}{\pi,\Gamma \vdash \theta_1 \sqsubseteq \theta_2}\ [Reoccur]$$

$$\frac{H{=}fst(\theta_1) \qquad \pi,\Gamma'{=}\Gamma,(\theta_1 \sqsubseteq \theta_2) \qquad \forall \mathrm{I} \in H.\ (\pi,\Gamma' \vdash \mathcal{D}_{\mathrm{I}}(\theta_1) \sqsubseteq \mathcal{D}_{\mathrm{I}}(\theta_2))}{\pi,\Gamma \vdash \theta_1 \sqsubseteq \theta_2}\ [Unfold]$$

$$\frac{\delta(\theta_1) \wedge \neg\delta(\theta_2) \qquad ConstrProp(\pi, \theta_1 \not\sqsubseteq \theta_2)}{\pi,\Gamma \vdash \theta_1 \not\sqsubseteq \theta_2}\ [Dis\text{-}Nullable]$$

$$\frac{H{=}fst(\theta_1) \qquad \exists \mathrm{I} \in H \qquad \mathcal{D}_{\mathrm{I}}(\theta_2) = \bot \qquad ConstrProp(\pi, \theta_1 \not\sqsubseteq \theta_2)}{\pi,\Gamma \vdash \theta_1 \not\sqsubseteq \theta_2}\ [Failed\text{-}Unfold]$$

Fig. 12. Term rewriting rules.

There are two possible failing cases. Rule [*Dis-Nullable*] is a heuristic refutation step to disprove the inclusions early when the antecedent evidently contains more traces than the consequent; and here, the *nullable* function δ witnesses the empty trace. Rule [*Failed-Unfold*] captures the situation where there exists an initial event I from the antecedent such that eliminating I from the consequent leads to false. When such failed assertions occur, we use *ConstrProp* to propagate the constraints for the core buggy code (cf. Sec. 6.1), which intakes the path constraint and the failed inclusion.

Termination of the TRS is guaranteed because the set of derivatives to be considered is finite, and possible cycles are detected using memorization, i.e., Γ. The term rewriting for regular expression is proven to be sound and complete [2], and prior TRS-based works [1, 3, 18, 21, 36–39], suggest that TRS is a better average-case algorithm than those based on the comparison of automata, by avoiding the complex translation process and disproving invalid inclusions earlier.

6 CONSTRAINT PROPAGATION AND DEDUCTIVE PATCH SYNTHESIS

The repair process incorporates two main components: (1) the constraint propagation when there are failed inclusions, and (2) the code synthesis when we should insert extra code with a given spec. In particular, the constraint propagation triggers the synthesis process after it extrapolates the expected spec for the buggy segments. In other words, (1) deletes code while (2) inserts code.

6.1 Bi-directional Constraint Propagation

From the previous section, we can now automatically prove or disprove given inclusions. Here, we are concerned with if an invalid inclusion exists; how to safely uncover the *core buggy code*, i.e., the smallest program segment which leads to the failed inclusion, and derive the expected spec for it.

$$\frac{\exists \theta_f.\quad rev(\theta_1) = \theta_f \cdot \theta_1' \qquad rev(\theta_2) = \theta_f \cdot \theta_2' \qquad \pi, \emptyset \vdash \theta_1' \not\sqsubseteq \theta_2'}{ConstrProp(\pi, \theta_1 \not\sqsubseteq \theta_2) \rightsquigarrow synthesis(\pi \wedge rev(\theta_2'))}\ [Constr\text{-}Prop]$$

Intuitively, having the path constrain π and the failed inclusion $\theta_1 \not\sqsubseteq \theta_2$, we do a backward matching to "sandwich" the buggy segment in the middle. As shown in the rule [*Constr-Prop*], if there exists a frame trace θ_f, such that θ_f serves as a common prefix of the reversed traces of θ_1 and θ_2 – $rev(\theta_1)$ and $rev(\theta_2)$ respectively – and their postfixes form an invalid inclusion, $\theta_1' \not\sqsubseteq \theta_2'$; then it passes the spec $\pi \wedge rev(\theta_2')$ to the code synthesis process. More specifically, $rev(\theta_1')$ represents the current behavior of the core buggy segment, which failed to entail its intended spec. Therefore, the patch is generated in the way of firstly deleting [7] the code represented by $rev(\theta_1')$, and secondly inserting the code which is synthesized based on the intended behavior, i.e., $\pi \wedge rev(\theta_2')$. The reverse function is defined in Definition 9.

Definition 9 (Trace Reversing). *Given a trace θ, its reversed trace is defined as follows:*

$$rev(\bot)=\bot \qquad rev(\epsilon)=\epsilon \qquad rev(I)=I \qquad rev(\theta^\star)=(rev(\theta))^\star$$
$$rev(\theta_1 \vee \theta_2)=rev(\theta_1) \vee rev(\theta_2) \qquad rev(\theta_1 \cdot \theta_2)=rev(\theta_2) \cdot rev(\theta_1)$$

Comment. Our repair strategy is more generic as it supports the deletion and insertion of code blocks instead of only supporting insertions [41] or only supporting single lines' repair [17, 24].

6.2 Source-level Patch Synthesis

Given the intended behavior $\pi \wedge \theta_{target}$, the *synthesis* function searches through the environment $\mathcal{E}$ – including both primitive specs and inferred specs, and composes an expression e_R in the target language grammar – that effects the following state transition: $\mathcal{E} \vdash \{T \wedge \epsilon\}\ e_R\ \{\pi \wedge \theta_{target}\}$.

We present a deductive synthesis algorithm in Algorithm 1. The procedure takes the target spec as the input; recursively searches through the environment $\mathcal{E}$; and finally returns a source code block e_R.

Algorithm 1 Algorithm for the Deductive Synthesis

Require: $\mathcal{E}, (\pi \wedge \theta_{target})$
Ensure: An expression e_R such that $\mathcal{E} \vdash \{T \wedge \epsilon\}\ e_R\ \{\pi \wedge \theta_{target}\}$
1: $e_{acc} = ()$
2: **for each** $nm\ (x^*) \mapsto [\Phi_{pre}, \Phi_{post}, \Phi_{future}] \in \mathcal{E}$ **do**
3: **if** $\theta_{target} = \epsilon$ **then return** `if` π `then` e_{acc} `else ()`
4: **else**
5: `// there exist a set of program variables` y^*
6: $\theta'_{target} = (\pi \wedge [y^*/x^*]\Phi_{post})^{-1}\theta_{target}$
7: $e_{acc} = e_{acc};\ nm(y^*)$
8: **end if**
9: **end for**
10: **return** without any suitable patches

In line 1, it initializes the accumulator e_{acc} using *unit*. From line 2, it iterates $\mathcal{E}$; and for each procedure signature, denoted by $nm(x^*)$, it tries to exploit if a function call to *nm* forms a progressive step towards the target trace. The base case is presented in line 3, where the target trace is ϵ, indicating no more synthesis obligations are needed. In this case, the procedure returns a conditional statement `if` π `then` e_{acc} `else ()`. Otherwise; we try to find a set of program variables y^* to instantiate the generic postcondition of $nm(x^*)$, and obtains the next-step target trace θ'_{target} by subtracting *nm*'s postcondition from the head of θ_{target}. If the subtraction does

[7] The fault localization is done by a mapping from line numbers to the generated events.

not lead to false, in line 7, it extends the current patch accumulator with a function call to nm, parameterized with y^*. Lastly, in line 10, if there are no satisfactory expressions after iterating all the ingredients from the program, the synthesis terminates with no patches.

Limitations. Since the ingredients in $\mathcal{E}$ are only procedure signatures, the synthesis only composes function calls. To mitigate this, we ought to dynamically extend the environment with mappings from statements to their effects. The current synthesis only uses the postconditions; therefore, inserting the function calls could lead to failed assertions introduced by these calls. Nevertheless, our approach generates sound patches, which resolve the targeted violated entailments, i.e., the current violations will no longer exist. While it is not trivial to have a repair without re-verification, we take it as a future work. Nevertheless, the following section shows that the current design serves a promising experimental result.

7 IMPLEMENTATION AND EVALUATION

We prototype our proposal into a program analysis and repair tool ProveNFix, using approximately 5,000 lines of OCaml code, leveraging on the AST structures produced by the Infer front-end. Our implementation includes a lightweight parser, which inputs the user-defined temporal specs. To show the effectiveness of our approach in analyzing and fixing a wide variety of bugs, we design the experimental evaluation to answer the following research questions (RQ):

- **RQ1:** What is the effectiveness and efficiency of ProveNFix compared to Infer?
- **RQ2:** What is the performance of ProveNFix in fixing memory usage bugs compared to SAVER, and fixing resource leaks compared to FootPatch?
- **RQ3:** Can ProveNFix automatically find/fix generalized temporal bugs, such as unchecked return values and properties involving execution orders, that prior works either cannot find or fix?
- **RQ4:** Is it practical to use ProveNFix as a specification inference tool to find desirable behaviors of internet-facing protocol implementations?

We ran experiments on an Ubuntu 22.04 LTS server with Intel Xeon E-2278G CPUs and 20GB of RAM. The source code and evaluation benchmarks are openly accessible from [46].

7.1 Primitive Specification Annotation

To facilitate the RQ 1-3 for comparison with older results, Table 1 presents the minimal efforts (in 54 lines of code) we took for annotating the temporal specs. The marks ✓ and ✗ in columns **Pre**, **Post**, and **Future** represent if the corresponding spec is needed for the given API. Notably, for different types of bugs, the need for annotations may vary; for example, `malloc` requirements pre/post/future conditions for memory bugs but only needs a future-condition for null pointer dereferences. We show the detailed specs for these 17 APIs in Appendix B.

Table 1. Summary of the Annotated API Specifications.

Primitive APIs	Pre	Post	Future	Targeted Bug Type
`open/socket/fopen/fdopen/opendir`	✗	✗	✓	Resource Leak
`close/fclose/endmntent/fflush/closedir`	✗	✓	✗	
`malloc/realloc/calloc/localtime`	✗	✗	✓	Null Pointer Dereference
→ *(pointer dereference)*	✗	✓	✗	
`malloc`	✓	✓	✓	Memory Usage
`free`	✓	✓	✓	(Leak, Use-After-Free, Double Free)

Table 2. Experimental results for analyzing 10 C projects, comparing with Infer-v1.1.0. Columns **#NPD**, **#ML** and **#RL** record the numbers of null pointer dereferences, memory leaks, and resource leaks, respectively. The numbers of false positives found by Infer and more true positives found by ProveNFix are represented by **+n** and **+n** respectively. Columns in **#Time** record the analysis time spent.

Project	kLoC	#NPD		#ML		#RL		Time	
		Infer	ProveNFix	Infer	ProveNFix	Infer	ProveNFix	Infer	ProveNFix
Swoole(a4256e4)	44.5	30+7	30+23	16+4	12+16	13+1	13+6	2m 50s	39.54s
lxc(72cc48f)	63.3	7+9	5+19	11+6	10+12	5+1	5+5	55.62s	1m 28s
WavPack(22977b2)	36	23+7	20+21	3	3+9	0+2	0	27.99s	23.77s
flex(d3de49f)	23.9	14+4	14+4	3	3+1	0	0+1	32.25s	47.75s
p11-kit	76.2	3+5	2+2	13+3	12+15	5	5+1	1m 57s	1m 4s
x264(d4099dd)	67.7	0	0	12	11+5	2	2+3	2m 33s	23.168s
recutils-1.8	81.9	25	22+8	13+10	11+29	1	1+7	9m 10s	38.29s
inetutils-1.9.4	117.2	7+4	5+8	9+3	7+10	1	1+5	30.26s	1m 5s
snort-2.9.13	378.2	44+12	33+34	26+4	15+16	1+2	1+1	8m 49s	3m 13s
grub(c6b9a0a)	331.1	13+12	6+5	1	1	0+3	0	3m 27s	1m 1s
Total	1,220.00	166+60	137+124	107+30	85+113	26+9	27+29	31m 12s	10m 44s

We use the same dataset as SAVER [17] to evaluate ProveNFix, which raise at least one error by running the latest release of Infer (v1.1.0, released on Mar 26, 2021). There are ten projects in total, and the basic information of the benchmarks is shown in the first two columns of Table 2.

7.2 RQ1: Comparison with Infer

As shown in Table 2, we compare the performance in detecting bugs between ProveNFix and Infer-v1.1.0. For each column, we manually classified the alarms into true and false positives. For example, for the Swoole project, Infer found 37 null pointer dereferences, which includes 30 true positives and 7 false positives. In comparison, ProveNFix found the same 30 true positives and 23 more true positives (i.e. actual null pointer dereference bugs). In addition, Infer found 20 memory leak bugs in the same project, including 16 true positives and 4 false positives; and out of the 16 true positives, ProveNFix found 12 of them (missing out 4 true alarms that Infer has found), and 16 more true positives. In total, Infer found 299 (166+107+26) true bugs, and ProveNFix found 249 (137+85+27) of them. In addition, ProveNFix found 266 (124+113+29) more true bugs, finding 72.2% ($\frac{249+266-299}{299}$) more true bugs, with a 17% ($\frac{299-249}{299}$) loss of missing true bugs (w.r.t. what Infer could find). We discuss all the bugs reported by ProveNFix in Appendix C.

Infer uses heuristics when processing failed proofs and bug patterns to reduce the false positives, which leads to the loss of many true bugs. This is intuitively why ProveNFix manages to find significantly more bugs. Moreover, due to the limited support for global variables, ProveNFix chose to miss out on some true bugs to keep a low false positive rate, cf. an example in Fig. 13.

7.3 RQ2: Comparison with SAVER and FootPatch

To compare the repair abilities between ProveNFix and the state-of-the-art, experimental results are shown in Table 3. The right-hand side of the table records the performance of SAVER and FootPatch, done on a virtual machine running Ubuntu-16.04. Both of them are built on top of an older version of Infer (v0.9.3, released on Sep 22, 2016). Upgrading the Infer version of SAVER and FootPatch requires too much effort; hence, we choose to report FootPatch/ SAVER results based on the older version of Infer. Given Infer's bug report, SAVER obtains a 73.7% fix rate for fixing true memory errors, and FootPatch obtains a 60% fix rate for true resource leaks.

Table 3. Experimental results for repairing 10 C projects, comparing with SAVER and FootPatch. Columns marked as # are numbers of the total true positives found by Infer-v1.1.0 and ProveNFix, summarised from Table 2. The numbers of false positives reported by Infer-v0.9.3 are marked as **+n**.

Project	NPD		ML		RL		Time	Infer-v0.9.3			
	#	ProveNFix	#	ProveNFix	#	ProveNFix		#ML	SAVER	#RL	FootPatch
Swoole	53	53	32	28	19	19	4.33s	15**+3**	11	6**+1**	6
lxc	26	24	23	22	10	10	3.882s	3**+5**	3	2**+1**	0
WavPack	44	41	12	12	0	0	11.435s	1**+2**	0	2	1
flex	18	18	4	4	1	1	39.38s	3**+4**	0	0	0
p11-kit	5	4	28	27	6	6	2.452s	33**+9**	24	2	1
x264	0	0	17	14	5	5	6.375s	10	10	0	0
recutils-1.8	33	30	42	36	8	8	1.261s	10**+11**	8	1	0
inetutils-1.9.4	15	13	19	17	6	6	1.517s	4**+5**	4	2**+1**	1
snort-2.9.13	78	67	42	13	2	2	10.57s	16**+27**	10	0	0
grub	18	11	1	1	0	0	40.626s	0	0	0	0
Total(Fix Rate)	290	261(90%)	220	174 (79%)	57	57 (100%)	2m 2s	95**+66**	70(73.7%)	15**+3**	9(60%)

The left-hand side of the table shows the repair results from ProveNFix, which has a 90% fix rate for null pointer dereferences, 79% fix rate for the memory leaks and 100% fix rate for the resource leaks. The correctness of auto-generated patches are confirmed by human validation.

Although the evaluation is based on different sources of static analysis, the true bugs recorded in the right-hand side table are fully subsumed by the left-hand side. We compare the fix rates and find that ProveNFix fixes 5% more memory leaks than SAVER and 40% more resource leaks than FootPatch. Besides the higher fix rate, SAVER needs a significant pre-analysis time to construct the whole object control flow, e.g., up to 26.3 seconds for the flex project and 39.5 minutes for the snort-2.9.13, while ProveNFix only takes several minuets to generate all the patches, averaged 0.25 seconds ($\frac{261+174+57}{2m\ 2s}$) per patch.

7.4 RQ3: Capturing General Temporal Bugs

ProveNFix can also find and fix generalized temporal bugs. Here, we use double-free (DF) errors as a case study to show the effectiveness of ProveNFix in finding and fixing bugs involving execution orders, and leave the case study for unchecked return values in Fig. 7.5.

Table 4. Automatically finding/fixing double free bugs.

Project	Double Free		Time
	SAVER	TempFix	
lxc	0	0	2m 15s
p11-kit	2	2**+1+1**	16.75s
grub	2	2**+1+1**	31.83s
Total	4	4**+2+2**	3m 4s

The benchmark shown in Table 4 is taken from SAVER [17]. SAVER usually relies on other bug detectors to generate bug report, before their repair. However, there were no suitable bug detectors for DF. Hence, among the ten projects shown from Table 2, SAVER *manually* records the DF errors by inspecting commit fixes by developers from open-source projects. Note that, Table 4 does not show the exact versions because these bugs are coming from different commit versions of their projects (lxc, p11-kit, and grub). Therefore, this experiment is done by injecting these errors into the version we used in Table 2.

All the true bugs reported in Table 4 can be correctly fixed by SAVER and ProveNFix, respectively. In total, SAVER fixed four true bugs while ProveNFix automatically found the same four bugs and two more true bugs, with two false positives. The two more true bugs that were missed out before are located in the same functions as the recorded bugs. The fixes provided by the developer were to remove the free statements in the recorded locations, which happened to solve the unrecorded bugs that ProveNFix discovered. Still, the extra true bugs that ProveNFix discovered seem to have

never been spotted/recorded by people. The two more false positives that ProveNFix generated are caused by unrolling loops only once and the complex aliasing/re-assignment, which exposes the limitations of our analysis. Nevertheless, the false positive rate of ProveNFix is still reasonably low. It takes ProveNFix 3m 4s to find and fix these bugs. All these bugs are documented in Appendix D.

7.5 RQ4: Effectiveness of Specification Inference for Real-world Programs

Table 5. Automatically repairing incorrect SSL API usages.

OpenSSL Applications	kLoC	Issue ID	Target API	Github Status	ProveNFix	Time
keepalive(843ffc80)	59.1	1003	SSL_CTX_new	✓	✓	5.62s
		1004	SSL_new	✓	✓	
thc-ipv6(011376c)	30.9	28	BN_new	✓	✓	3.32s
		29	BN_set_word	✓	✗	
FreeRADIUS(94149dc)	258.9	2309	BIO_new	✓	✓	38.89s
		2310	i2a_ASN1_OBJECT	✓	✓	
trafficserver(5ee6a5f)	34.1	4292	SSL_CTX_new	✓	✓	21.55s
		4293	SSL_new	✓	✓	
		4294	SSL_write	✓	✓	
sslsplit(19a16bd)	18.7	224	SSL_CTX_use_certificate	✓	✓	2.69s
		225	SSL_use_PrivateKey	✓	✓	
proxytunnel(f7831a2)	3.1	36	SSL_connect	✓	✓	0.62s
		37	SSL_new	✓	✓	

Our approach can generate specifications for real-world programs. We demonstrate the effectiveness of ProveNFix's specification generation, on an internet-facing protocol implementation OpenSSL [29]. Correct usage of Secure Socket Layer (SSL) APIs is required to satisfy certain constraints, such as call conditions or call orders. Violations of these constraints will lead to severe security implications. For example, missing error status code validation of SSL APIs will cause a denial of service by remote attackers (CVE-2016-2182 [10]), and broken SSL certificate validation [13] will result in man-in-the-middle attacks [9]. Here, we focus on the bugs caused by URV.

There are a lot of security-sensitive URV bugs in OpenSSL applications [14]. Table 5 presents several real-world bugs that have been confirmed, fixed, and merged into the master branches. To detect and fix those bugs, we provide 2 predefined primitive specs and rely on ProveNFix to generate specs for all the exposing APIs by analyzing OpenSSL-3.1.2 source code (3792bd7, 556.3 kLoC). Then, these automatically generated specs are used to analyze the projects shown in Table 5. Experimental results show that ProveNFix successfully detects and repairs almost all the reported bugs – one failure case is caused by no spec generated for 'BN_set_word' – within 1 minute per project, demonstrating the effectiveness of ProveNFix in inferring correct procedure specs. Moreover, the generated specs do not leak the implementation details of the APIs apart from revealing the input-output relations, as it omits all the intermediate variables by annotating them as existential. Detailed configuration and all the inferred specs are recorded in Appendix E.

7.6 Discussion: Sources of False Negatives and False Positives

Fig. 13 presents a code snippet simplified from the Swoole project, where the function passes the pointer "`ptr`" to a global object "`swoole_objects`". For such cases, ProveNFix infers the postcondition with a special event "`consume(ptr)`", which, by design, entails any event, such as `free(ptr)` or `close(ptr)`. As ProveNFix is designed for modular reasoning, without capturing the global states, it assumes global variables are well-managed, i.e., the traces of using them satisfy all the possible constraints. As a practice of systematically – instead of using heuristics – strengthening the inferred

postconditions, having consume events helps us to reduce false positives effectively; however, such a design also introduces false negatives. Therefore, generally, in ProveNFix, false negatives are caused by the pre-defined pre/future-conditions being too weak or the inferred postconditions being too strong. In duality, false positives occur when the pre-defined pre/future conditions are too strong or the inferred postconditions are too weak.

```
void swoole_set_object(zval *object,void *ptr)
// post: true ∧ (consume(ptr))
{...; swoole_objects.array[handle] = ptr; ...}
```

Fig. 13. Demonstration for the source of false negatives.

8 RELATED WORK

Analyzing Temporal Properties. Model checking [7] is a well-known verification technique that can prove temporal properties in finite state systems, usually encoded using automata [6, 16, 20]. Furthermore, model checking is employed to identify counterexamples where properties are violated, such as CMC [27], Java Pathfinder [42], and CPAchecker [4]. While these tools effectively uncover assertion violations, they encounter challenges like state space explosion and slow logical formula solving. In contrast, ProveNFix performs a compositional analysis that scales to large real-world programs while achieving high precision.

Another approach to detecting temporal property violations is runtime verification, such as JavaMOP[25], MarQ [33] and Mufin [40], which monitors test executions against formal specs. In particular, our *IntRE* draws similarities to JavaMOP's spec language – both explicitly capture the past-time LTL by putting them into preconditions. Specifically, JavaMOP monitors the manually written and automatically mined specs against tests in open-source projects and finds hundreds of bugs. However, those runtime verification tools rely on large numbers of tests, which are not always available. Although LTLFuzz [26] could automatically generate tests via greybox fuzzing, the coverage of generated tests is usually low, leading to false negatives. Our method performs static analysis without requiring test cases, which could complement runtime verification.

The latest Infer release (v1.1.0) introduces a new checker Topl [8], to detect errors based on user-provided state machines describing temporal properties. Like ProveNFix, Topl aims to encode different analyzes as temporal properties, such as taint analysis. For example, its analysis can prevent a value returned by method *source()* from being sent as an argument to a method *sink()*. However, Topl is still in the experimental phase and hasn't shown any potential for automated program repair; moreover, just like Infer, Topl possibly suffers from many false negatives due to the heuristics deployed for reducing false positives.

Repair via Program Analyzer. Recently, several repair techniques based on static analysis tools have been proposed in the program repair community [17, 19, 24, 28, 41].One of the most relevant works is FootPatch [41], which relies on separation logic to fix bugs related to resource release, freeing memory, and null pointer dereference. However, FootPatch may introduce double-free when fixing a memory leakage. Moreover, to automatically generate patches, it requires templated annotations for the specs at the bug locations. Another tool MemFix [24] targets safely fixing memory usage bugs such as: memory leaks, use-after-frees, and double-frees. MemFix works for small-scale programs (<5kLoC) based on a variant of typestate static analysis and cannot produce patches with conditional deallocation. In contrast, SAVER [17] targets on the same set of bugs, but

[8] https://fbinfer.com/docs/checker-topl/

deals with big-scale projects (up to 320kLoC) with the help of Infer. However, SAVER's patches are generated by keeping object flow graphs for the buggy programs. In other words, SAVER failed to make use of the intermediate bugs information generated by Infer. The proposed technique in this work also draws similarities to Senx [19], which proposes property-based APR. Such properties are automatically inferred via an access range analysis, which can be used to deal with bugs, including buffer overflow/integer overflow and bad cast. Although Senx and ProveNFix target different bug types, the hint highlighted by Senx is valuable: the advantage of the property-based approach is that a small set of safety properties can be specified once and used on a vast number of programs without the need to specify anything specific about each of the programs, or collect a comprehensive set of test cases. Moreover, such properties are inherently precise and complete. Our work is the first work to deal with different bug types (e.g., memory leaks, unchecked return value, double-free, and general temporal bugs) via encoding them into temporal properties.

Specification-guided Repair and Deductive Synthesis. Other existing works have also explored formal specs guided program repair in the form of writing pre/post-conditions [28, 30] or assertions [15, 34]. Those approaches then utilize verification-based approaches to generate patches, ensuring the given conditions are satisfied. They can produce precise and complete patches, but writing formal specs for each function is tedious and time-consuming. Instead of asking users to provide pre/post-conditions, our approach relies on a future-conditions to modularly enable an automated spec generation for their callers. Prior work [31] describes a deductive approach to synthesizing imperative programs with pointers from specs expressed in Separation Logic. Their synthesis algorithm takes as input a pair of assertions – a pair of pre/post-condition – which describe two states of the symbolic heap and derives a program that transforms one state into the other, guided by the shape of the heap. Kneuss et al. [22] explore the problem of automatically repairing programs written as a set of mutually recursive functions in a purely functional subset of Scala, evaluated on seeded bugs on small programs. More recently, Nguyen et al. [28] propose a novel method to automatically repair buggy heap-manipulating programs using constraint solving and deductive synthesis [31]. All those approaches only target small programs, while our approach has been shown to scale to large programs.

Temporal Property Inference. Automatically generating temporal specifications is an important research direction. To enable a compositional analysis, both Infer and ProveNFix vastly reduced the human effort by automatically inferring the specifications starting from a core set of primitives. To further reduce the effort of annotating the primitive APIs, one approach is to use data mining tools to extract the specifications from "good" code. Prior work [44, 45] propose an automatic approach to inferring a target system's temporal properties by analyzing its event traces. The core of this technique is a set of predefined property templates crafted for a set of common events. These templates form a partial order in terms of their *strictness*; and it finds the strictest properties satisfied by a set of events based on the traces. Another approach is to make use of inference tools, such as Infer or ProveNFix, to generate specifications from the API implementations (hinted in Sec. 7.5), such as OpenSSL projects, and release the generated specifications for future analysis.

The first approach may be fast but imprecise, while the second is more precise but requires additional tools to support spec inference. Nevertheless, these other directions are orthogonal to our current proposal. In this paper, the temporal specs of primitive functions are manually annotated according to the API documentation for C or descriptions for program functionalities. Our spec inference uses a compositional analysis to infer the spec of procedures based on its behavior(s) from given primitive API's specs.

Realizability Checkers for Reactive Systems. Realizability checking [12] with assume/guarantee contracts has been studied for synthesis and checking temporal logic requirements. A contract is realizable, if it is possible to synthesize a component such that for any input allowed by the contract assumptions, the component can produce outputs that satisfy the contract guarantees. However, such techniques do not enforce the constraints on events occurring after the component, as we captured via future conditions. We note that realizability checkers focus on synthesis, while our proposal focuses on temporal bug detection and repair.

9 CONCLUSION

This work is motivated by "*how to modularly analyze and repair bugs which can be encoded using temporal properties?*". Our main contribution is showing the feasibility of finding and repairing a wide range of bug types using expressive temporal specifications. Specifically, we present a compositional framework driven by a minimal effort of annotating primitive APIs, which then incorporates the novel future-conditions. This enables an automatic specification inference and dynamic constraint generation. We prototype the proposal, present experimental results, and demonstrate nontrivial case studies to show its utility. ProveNFix is the first program repair tool, which is guided by temporal properties.

DATA AVAILABILITY

The source code of the tool, the dataset, and the appendix are available from [46].

ACKNOWLEDGMENTS

This work was partially supported by a Singapore Ministry of Education (MoE) Tier3 research grant "Automated Program Repair", MOE-MOET32021-0001, and the National Natural Science Foundation of China (Grant No. 62202026).

REFERENCES

[1] Marco Almeida, Nelma Moreira, and Rogério Reis. 2009. Antimirov and Mosses's Rewrite System Revisited. *Int. J. Found. Comput. Sci.* 20, 4 (2009), 669–684. https://doi.org/10.1142/S0129054109006802

[2] Valentin Antimirov. 1995. Partial derivatives of regular expressions and finite automata constructions. In *Annual Symposium on Theoretical Aspects of Computer Science*. Springer, 455–466.

[3] Valentin M. Antimirov and Peter D. Mosses. 1995. Rewriting Extended Regular Expressions. *Theor. Comput. Sci.* 143, 1 (1995), 51–72. https://doi.org/10.1016/0304-3975(95)80024-4

[4] Dirk Beyer and M Erkan Keremoglu. 2011. CPAchecker: A tool for configurable software verification. In *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*. Springer, 184–190.

[5] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving Fast with Software Verification. In *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9058)*, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer, 3–11. https://doi.org/10.1007/978-3-319-17524-9_1

[6] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. 2002. Nusmv 2: An opensource tool for symbolic model checking. In *Computer Aided Verification: 14th International Conference, CAV 2002 Copenhagen, Denmark, July 27–31, 2002 Proceedings 14*. Springer, 359–364.

[7] Edmund M Clarke. 1997. Model checking. In *Foundations of Software Technology and Theoretical Computer Science: 17th Conference Kharagpur, India, December 18–20, 1997 Proceedings 17*. Springer, 54–56.

[8] Edmund M. Clarke, William Klieber, Milos Novácek, and Paolo Zuliani. 2011. Model Checking and the State Explosion Problem. In *Tools for Practical Software Verification, LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures (Lecture Notes in Computer Science, Vol. 7682)*, Bertrand Meyer and Martin Nordio (Eds.). Springer, 1–30. https://doi.org/10.1007/978-3-642-35746-6_1

[9] CVE-2016-2113. 2016. CVE-2016-2113. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-2113.

[10] Cve-2016-2182. 2016. Cve-2016-2182. https://nvd.nist.gov/vuln/detail/CVE-2016-2182.

[11] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4963)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[12] Andrew Gacek, Andreas Katis, Michael W. Whalen, John Backes, and Darren D. Cofer. 2015. Towards Realizability Checking of Contracts Using Theories. In *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9058)*, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer, 173–187. https://doi.org/10.1007/978-3-319-17524-9_13

[13] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. 2012. The most dangerous code in the world: validating SSL certificates in non-browser software. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, Ting Yu, George Danezis, and Virgil D. Gligor (Eds.). ACM, 38–49. https://doi.org/10.1145/2382196.2382204

[14] Zuxing Gu, Jiecheng Wu, Chi Li, Min Zhou, and Ming Gu. 2019. SSLDoc: Automatically Diagnosing Incorrect SSL API Usages in C Programs. In *The 31st International Conference on Software Engineering and Knowledge Engineering, SEKE 2019, Hotel Tivoli, Lisbon, Portugal, July 10-12, 2019*, Angelo Perkusich (Ed.). KSI Research Inc. and Knowledge Systems Institute Graduate School, 707–777. https://doi.org/10.18293/SEKE2019-006

[15] Simón Gutiérrez Brida, Germán Regis, Guolong Zheng, Hamid Bagheri, ThanhVu Nguyen, Nazareno Aguirre, and Marcelo Frias. 2021. Bounded Exhaustive Search of Alloy Specification Repairs. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*.

[16] Gerard J. Holzmann. 1997. The model checker SPIN. *IEEE Transactions on software engineering* 23, 5 (1997), 279–295.

[17] Seongjoon Hong, Junhee Lee, Jeongsoo Lee, and Hakjoo Oh. 2020. SAVER: scalable, precise, and safe memory-error repair. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 271–283. https://doi.org/10.1145/3377811.3380323

[18] Dag Hovland. 2012. The inclusion problem for regular expressions. *J. Comput. Syst. Sci.* 78, 6 (2012), 1795–1813. https://doi.org/10.1016/j.jcss.2011.12.003

[19] Zhen Huang, David Lie, Gang Tan, and Trent Jaeger. 2019. Using Safety Properties to Generate Vulnerability Patches. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 539–554. https://doi.org/10.1109/SP.2019.00071

[20] Gijs Kant, Alfons Laarman, Jeroen Meijer, Jaco Van de Pol, Stefan Blom, and Tom Van Dijk. 2015. LTSmin: high-performance language-independent model checking. In *Tools and Algorithms for the Construction and Analysis of Systems: 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings 21*. Springer, 692–707.

[21] Matthias Keil and Peter Thiemann. 2014. Symbolic Solving of Extended Regular Expression Inequalities. In *34th International Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2014, December 15-17, 2014, New Delhi, India (LIPIcs, Vol. 29)*, Venkatesh Raman and S. P. Suresh (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 175–186. https://doi.org/10.4230/LIPIcs.FSTTCS.2014.175

[22] Etienne Kneuss, Manos Koukoutos, and Viktor Kuncak. 2015. Deductive Program Repair. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 9207)*, Daniel Kroening and Corina S. Pasareanu (Eds.). Springer, 217–233. https://doi.org/10.1007/978-3-319-21668-3_13

[23] Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. 2022. Finding real bugs in big programs with incorrectness logic. *Proc. ACM Program. Lang.* 6, OOPSLA1 (2022), 1–27. https://doi.org/10.1145/3527325

[24] Junhee Lee, Seongjoon Hong, and Hakjoo Oh. 2018. MemFix: static analysis-based repair of memory deallocation errors for C. In *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*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 95–106. https://doi.org/10.1145/3236024.3236079

[25] Owolabi Legunsen, Wajih Ul Hassan, Xinyue Xu, Grigore Rosu, and Darko Marinov. 2016. How good are the specs? a study of the bug-finding effectiveness of existing Java API specifications. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 602–613. https://doi.org/10.1145/2970276.2970356

[26] Ruijie Meng, Zhen Dong, Jialin Li, Ivan Beschastnikh, and Abhik Roychoudhury. 2022. Linear-time Temporal Logic guided Greybox Fuzzing. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 1343–1355. https://doi.org/10.1145/3510003.3510082

[27] Madanlal Musuvathi, David YW Park, Andy Chou, Dawson R Engler, and David L Dill. 2002. CMC: A pragmatic approach to model checking real code. *ACM SIGOPS Operating Systems Review* 36, SI (2002), 75–88.

[28] Thanh-Toan Nguyen, Quang-Trung Ta, Ilya Sergey, and Wei-Ngan Chin. 2021. Automated Repair of Heap-Manipulating Programs Using Deductive Synthesis. In *Verification, Model Checking, and Abstract Interpretation - 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17-19, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12597)*, Fritz Henglein, Sharon Shoham, and Yakir Vizel (Eds.). Springer, 376–400. https://doi.org/10.1007/978-3-030-67067-2_17

[29] openssl. 2019. Openssl: cryptography and ssl/tls toolkit. https://github.com/openssl/.

[30] Yu Pei, Carlo A Furia, Martin Nordio, Yi Wei, Bertrand Meyer, and Andreas Zeller. 2014. Automated fixing of programs with contracts. *IEEE transactions on software engineering* 40, 5 (2014), 427–449.

[31] Nadia Polikarpova and Ilya Sergey. 2019. Structuring the synthesis of heap-manipulating programs. *Proc. ACM Program. Lang.* 3, POPL (2019), 72:1–72:30. https://doi.org/10.1145/3290385

[32] Swoole Project. [n. d.]. Swoole Project. https://github.com/swoole/swoole-src.

[33] Giles Reger, Helena Cuenca Cruz, and David Rydeheard. 2015. MarQ: monitoring at runtime with QEA. In *Tools and Algorithms for the Construction and Analysis of Systems: 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings 21*. Springer, 596–610.

[34] Bat-Chen Rothenberg and Orna Grumberg. 2016. Sound and complete mutation-based program repair. In *FM 2016: Formal Methods: 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings 21*. Springer, 593–611.

[35] Micha Sharir, Amir Pnueli, et al. 1978. *Two approaches to interprocedural data flow analysis*. New York University. Courant Institute of Mathematical Sciences ....

[36] Yahui Song and Wei-Ngan Chin. 2020. Automated Temporal Verification of Integrated Dependent Effects. In *Formal Methods and Software Engineering - 22nd International Conference on Formal Engineering Methods, ICFEM 2020, Singapore, Singapore, March 1-3, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12531)*, Shang-Wei Lin, Zhe Hou, and Brendan P. Mahony (Eds.). Springer, 73–90. https://doi.org/10.1007/978-3-030-63406-3_5

[37] Yahui Song and Wei-Ngan Chin. 2021. A Synchronous Effects Logic for Temporal Verification of Pure Esterel. In *Verification, Model Checking, and Abstract Interpretation - 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17-19, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12597)*, Fritz Henglein, Sharon Shoham, and Yakir Vizel (Eds.). Springer, 417–440. https://doi.org/10.1007/978-3-030-67067-2_19

[38] Yahui Song and Wei-Ngan Chin. 2023. Automated Verification for Real-Time Systems. In *Tools and Algorithms for the Construction and Analysis of Systems*, Sriram Sankaranarayanan and Natasha Sharygina (Eds.). Springer Nature Switzerland, Cham, 569–587.

[39] Yahui Song, Darius Foo, and Wei-Ngan Chin. 2022. Automated Temporal Verification for Algebraic Effects. In *Programming Languages and Systems - 20th Asian Symposium, APLAS 2022, Auckland, New Zealand, December 5, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13658)*, Ilya Sergey (Ed.). Springer, 88–109. https://doi.org/10.1007/978-3-031-21037-2_5

[40] Daniel Thoma. [n. d.]. Runtime Monitoring with Union-Find Structures. *Tools and Algorithms for the Construction and Analysis of Systems LNCS 9636* ([n. d.]), 868.

[41] Rijnard van Tonder and Claire Le Goues. 2018. Static automated program repair for heap properties. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 151–162. https://doi.org/10.1145/3180155.3180250

[42] Willem Visser and Peter Mehlitz. 2005. Model checking programs with Java PathFinder. In *Model Checking Software: 12th International SPIN Workshop*. Springer, 27–27.

[43] Pierre Wolper. 1983. Temporal Logic Can Be More Expressive. *Inf. Control.* 56, 1/2 (1983), 72–99. https://doi.org/10.1016/S0019-9958(83)80051-5

[44] Jinlin Yang and David Evans. 2004. Automatically Inferring Temporal Properties for Program Evolution. In *15th International Symposium on Software Reliability Engineering (ISSRE 2004), 2-5 November 2004, Saint-Malo, Bretagne, France*. IEEE Computer Society, 340–351. https://doi.org/10.1109/ISSRE.2004.11

[45] Jinlin Yang and David Evans. 2004. Dynamically inferring temporal properties. In *Proceedings of the 2004 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'04, Washington, DC, USA, June 7-8, 2004*, Cormac Flanagan and Andreas Zeller (Eds.). ACM, 23–28. https://doi.org/10.1145/996821.996832

[46] Zenodo. 2023. Benchmark and Source Code. https://doi.org/10.5281/zenodo.8388488.

Received 2023-09-22; accepted 2024-01-23