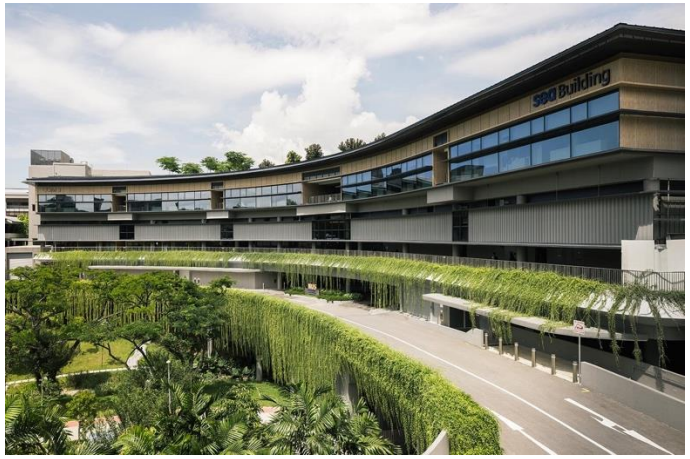
Future Conditions

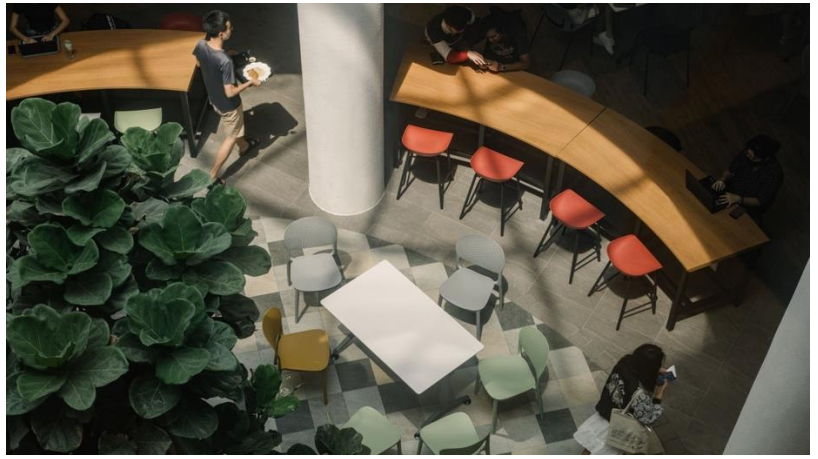
Temporal Property guided Program Analysis, Repair and Verification

Yahui Song

Research Fellow @ National University of Singapore (NUS)

June 2025





My Research

- PhD (2018 Aug – 2022 Dec)

  Thesis: Symbolic Temporal Verification Techniques with Extended Regular Expressions

  Keywords: Modularly (Scalability), Expressive Specification, Hoare-style Verification (source code level)

  Applications
  - Event-based reactive systems [ICFEM 2020]
  - Synchronous languages like Esterel [VMCAI 2021]
  - User-defined algebraic effects and handlers [APLAS 2022]
  - Real-time systems [TACAS 2023]

- Research Fellow (2023 Jan – now)

  Staged Specification Logic (Regular expression + Separation logic):

  Higher-order Imperative Programs [FM 2024]; Algebraic Effects and Handlers [ICFP 2024]

  Temporal Property guided Program Analysis, Repair and Verification:

  ProveNFix: Temporal Property guided Program Repair [FSE 2024]

  Specifying and Verifying Future Conditions [Under Submission]

Can temporal property analysis be modular?

“Each function is analysed only once and

can be replaced by their verified properties.”

Can temporal property analysis be modular?

“Each function is analysed only once and

can be replaced by their verified properties.”

Three main difficulties:

- ❑ Temporal logic entailment checker.
- ❑ Writing temporal specifications for each function is tedious and challenging.
- ❑ The classic pre/post-conditions is not enough, e.g.,

  “some meaningful operations can only happen if the return value of loading the certificate is positive”

Future-condition

Defined in header <stdlib.h>		
`void free( void* ptr );`		

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

The behavior is undefined if after `free()` returns, an access is made through the pointer `ptr` (unless another allocation function happened to result in a pointer value equal to `ptr`).

Defined in header <stdlib.h>		
`void* malloc( size_t size );`		

On success, returns the pointer to the beginning of newly allocated memory. To avoid a memory leak, the returned pointer must be deallocated with `free()` or `realloc()`.

On failure, returns a null pointer.

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

Future-condition based compositional analysis

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

Entailment Checking

A collection of specifications

Future-condition based compositional analysis

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

Entailment Checking

A collection of specifications

Can temporal property analysis be modular?

“Each function is analysed only once and

can be replaced by their verified properties.”

Three main difficulties:

❑ Temporal logic entailment checker.

❑ Writing temporal specifications for each function is tedious and challenging.

✓ The classic pre/post-conditions is not enough, e.g., **Future-condition!**

“some meaningful operations can only happen if the return value of loading the certificate is positive”

Specification inference

```
void *malloc (size_t size);
// future: (ret=null ∧ 𝒢 (!_(ret))) ∨ (ret≠null ∧ ℱ (free(ret))
```

```
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;}
```

Specification inference

```
void *malloc (size_t size);
// future: (ret=null ∧ 𝒢 (!_(ret))) ∨ (ret≠null ∧ ℱ (free(ret))
```

```
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;}
```

Failed entailment: $\text{true} \wedge \mathcal{E} \not\sqsubseteq \text{ptr}\neq\text{null} \wedge \mathcal{F}(\text{free}(\text{ptr}))$

Can temporal property analysis be modular?

“Each function is analysed only once and

can be replaced by their verified properties.”

Three main difficulties:

❑ Temporal logic entailment checker.

Primitive spec + spec inference!

✓ Writing temporal specifications for each function is tedious and challenging.

✓ The classic pre/post-conditions is not enough, e.g., **Future-condition!**

“some meaningful operations can only happen if the return value of loading the certificate is positive”

Term rewriting system for regular expressions

- Flexible specifications, which can be combined with other logic;
- Efficient entailment checker with inductive proofs.

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

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

Term rewriting system for regular expressions

- Flexible specifications, which can be combined with other logic;
- Efficient entailment checker with inductive proofs.

Examples:

$x>2 \wedge E \sqsubseteq x>1 \wedge (E \vee F)$

$x>0 \wedge E \not\sqsubseteq x>1 \wedge (E \vee F)$

$true \wedge E \not\sqsubseteq true \wedge (E \,.\, F)$

$$\dfrac{\dfrac{(a \vee b)^\star \sqsubseteq (a \vee b \vee bb)^\star \;\; [\text{Reoccur}]}{\varepsilon \cdot (a \vee b)^\star \sqsubseteq \varepsilon \cdot (a \vee b \vee bb)^\star} \qquad \dfrac{}{[\text{Reoccur}]}}{\dfrac{a \cdot (a \vee b)^\star \sqsubseteq (a \vee b \vee bb)^\star \qquad b \cdot (a \vee b)^\star \sqsubseteq \ldots}{(a \vee b)^\star \sqsubseteq (a \vee b \vee bb)^\star}}$$

Can temporal property analysis be modular?

Can!

“Each function is analysed only once and can be replaced by their verified properties.”

Three main difficulties:

A term rewriting system for regular expressions

✓ Temporal logic entailment checker.

Primitive spec + spec inference!

✓ Writing temporal specifications for each function is tedious and challenging.

✓ The classic pre/post-conditions is not enough, e.g., **Future-condition!**

“some meaningful operations can only happen if the return value of loading the certificate is positive”

Experiment 1: detecting bugs

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)

- ❖ 17 predefined primitive specs.
- ❖ ProveNFix is finding 72.2% more true bugs, with a 17% loss of missing true bugs.

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

Automated repair via deductive synthesis

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

Example: `true` ∧ $\mathcal{E} \not\models$ `ptr≠null` ∧ `_^*.(free(ptr))`

⇒ synthesis(`ptr≠null` ∧ `_^*.(free(ptr))`) ⇒ `if (ptr != NULL) free(ptr);`

Automated repair via deductive synthesis

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

- Only supporting inserting/deleting calls.
- Do need re-analysis.

Example: `true` ∧ $\mathcal{E} \not\vdash$ `ptr≠null ∧ _^*.(free(ptr))`

⇒ synthesis(`ptr≠null ∧ _^*.(free(ptr))`) ⇒ `if (ptr != NULL) free(ptr);`

Experiment 2: Repairing bugs

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%)

- ❖ 90% fix - null pointer dereferences,
- ❖ 79% fix - memory leaks
- ❖ 100% fix - resource leaks.

SAVER's pre-analysis time:
26.3 seconds for the flex project
39.5 minutes for the snort-2.9.13 project

Experiment 4: usefulness of spec inference

- 2 predefined primitive specs, OpenSSL-3.1.2, 556.3 kLoC,
- 143.11 seconds to generate future-conditions for 128 OpenSSL APIs
- Example: SSL_CTX_new (meth) ; // future : ((ret=0) /\ return (ret))

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	✓	✓	

Summary

Contributions

- ✓ A novel *future-condition*
- ✓ Compositional temporal analysis
- ✓ Light-weight specification inference
- ✓ Fast and most-automated
- ✓ Proof guided repair
- ✓ Large-scale usability

Limitations

- ❑ Handle loops via unrolling
- ❑ Inefficient ($O(n^2)$) entailment checking
- ❑ On-demand path pruning
- ❑ False negatives
- ❑ No machine checkable certification
- ❑ Limited expressiveness

Specifying and Verifying Future Conditions (FCs)

Yahui Song, Darius Foo, Wei-Ngan Chin

(Under Submission)

The existing solution

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

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

Three main limitations:

- ❑ Inefficient ($O(n^2)$) entailment checking
- ❑ Handle loops via unrolling
- ❑ Bug-finding (no incorrectly flagged safe code) over soundness (no missed violations)

Inefficient ($O(n^2)$) entailment checking

A use-after-free bug recorded from CWE-416

```
1 int main(int argc, char **argv) {
2   char *buf1, *buf2, *buf3;
3   buf1 = malloc(1);
4   buf2 = malloc(1);
5   free(buf2);
6   buf3 = malloc(1);
7   strncpy(buf2,argv[1],1);
8   free(buf1); free(buf3); }
```

Use-after-free!

A new solution for reasoning FCs

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!

- ❖ Linear trace processing
- ❖ Embed FCs into program states
- ❖ Trace conjunction + subtraction

The existing solution

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

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

Three main limitations:

- ✓ Inefficient entailment checking **Embed FCs into the states + Trace subtraction**
- ❑ Handle loops via unrolling
- ❑ Bug-finding (no incorrectly flagged safe code) over soundness (no missed violations)

Predicates for Bags of Traces and Future Conditions

A false negative example from ProveNFix

```
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 */}
```

Predicates for Bags of Traces and Future Conditions

```
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 */}
```

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

$$(Specification) \quad [\textbf{req}: \pi\ \textbf{ens}: \Delta]$$
$$(Post\ Summary) \quad \Delta ::= \bigvee(\pi\ ;\ \theta\ ;\ F)$$

When reasoning about main():

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 "$pred_f([1..5), arr)$"!

The existing solution

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

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

Three main limitations:

✓ Inefficient entailment checking — **Embed FCs into the states + Trace subtraction**

✓ Handle loops via unrolling — **Predicates for bags of traces and FCs**

❑ Bug-finding (no incorrectly flagged safe code) over soundness (no missed violations)

Soundness Formalization

- An instrumented semantics for the target language: $[s, \rho, F, e] \longrightarrow [s', \rho', F', v]$

  stack execution trace

- Semantic model of trace specifications: $s, \rho \models \pi \wedge \theta$

- A set of forward verification rules: $\{(P\,;\theta_1\,;F_1)\}\ e\ \{(Q\,;\theta_2\,;F_2)\}$

```
Theorem soundness : forall P e Q t1 t2 rho1 rho2 s1 v s2 f1 f2 f3,
  forward P t1 f1 e Q t2 f2 ->
  P s1 ->
  trace_model rho1 t1 ->
  bigstep s1 rho1 f1 e s2 rho2 f3 v ->
  Q v s2 /\ trace_model rho2 t2 /\ futureCondEntail f2 f3.
```

It only sound to strengthen the future conditions, so that we do not miss any violations.

The existing solution

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

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

Three main limitations:

- ✓ Inefficient entailment checking — **Embed FCs into the states + Trace subtraction**
- ✓ Handle loops via unrolling — **Predicates for bags of traces and FCs**
- ✓ Bug-finding (no incorrectly flagged safe code) over soundness (no missed violations)

Coq formalization

Experimental Results

Category	Example APIs	Future Conditions
1. File Ops	fopen, open fclose, close	Finally to close the file descriptor Globally do not access the file descriptor Read-only files cannot be written to
2. Threads	pthread_create pthread_mutex_lock	Finally to pthread_join or detach the thread Finally to pthread_mutex_unlock
3. Memory	free malloc realloc	Globally do not access the pointer Finally free the new pointer 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

Write these future conditions manually

Experimental Results

Category	LoC	PrimS	InferredS	InferredInv	Report/Exp.	Time(s)
1	656	8	29	7	14/12	10.05
2	330	4	25	1	4/4	1.97
3	424	6	30	11	25/23	8.87
4	103	2	6	1	3/3	1.76
5	108	4	6	0	4/4	1.97
6	67	10	5	0	5/5	0.56
Total	1,688	34	101	20	55/51	25.18

Category	Example APIs	Future Conditions
1. File Ops	fopen, open fclose, close	Finally to close the file descriptor Globally do not access the file descriptor Read-only files cannot be written to
2. Threads	pthread_create pthread_mutex_lock	Finally to pthread_join or detach the thread Finally to pthread_mutex_unlock
3. Memory	free malloc realloc	Globally do not access the pointer Finally free the new pointer 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

False positive due to the limited expressiveness:

```
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!
```

Future Conditions

Thanks for listening!

Bug Finding and Repair

- ✓ A novel *future-condition*
- ✓ Compositional temporal analysis
- ✓ Light-weight specification inference
- ✓ Fast and most-automated
- ✓ Proof guided repair
- ✓ Large-scale usability

Verification

- ✓ Handle loops via recursive predicates
- ✓ Efficient (linear) entailment checking
- ✓ Sound weakening when path explosion
- ✓ No false negatives
- ❑ No machine checkable certification
- ❑ Limited expressiveness