Specifying and Verifying Future Conditions

Yahui Song, Darius Foo, Wei-Ngan Chin

Static Analysis Symposium (SAS) @ SPLASH 2025, Singapore

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!

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

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

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

char *strncpy(char *dest,const char *source,size_t num);
// post: true ∧ strncpy(dest)
```

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

```
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);    Use-after-free!
8   free(buf1); free(buf3); }
```

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

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

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

char *strncpy(char *dest,const char *source,size_t num);
// post: true ∧ strncpy(dest)
```

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

```
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); }
```

malloc(buf2).free(buf2).malloc(buf3).strncpy(buf2).free(buf1).free(buf3) ⊑ **F(free(buf1))**

free(buf2).malloc(buf3).strncpy(buf2).free(buf1).free(buf3) ⊑ **F(free(buf2))**

malloc(buf3).strncpy(buf2).free(buf1).free(buf3) ⋢ **G (! _ (buf2))**

strncpy(buf2).free(buf1).free(buf3) ⊑ **F(free(buf3))**

Use-after-free!

free(buf3) ⊑ **G (!_(buf1))**

empty ⊑ **G (!_(buf3))**

A new solution for reasoning FCs

```
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);
```

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 */}
```

$$mallocN(n, arr) \equiv \textbf{req}: length(arr) \geq n \quad \textbf{ens}: (\exists i.\, true\,;\, pred_t([0..n), i)\,;\, pred_f([0..n), i))$$

$$pred_t(B, i) \equiv \Lambda_i^B(arr[i] \neq null \wedge malloc(arr[i])) \vee (arr[i] = null \wedge \epsilon)$$

$$pred_f(B, i) \equiv \Lambda_i^B(arr[i] \neq null \wedge \mathcal{F}(free(arr[i])))$$

$$(Specification) \quad [\textbf{req}: \pi \ \textbf{ens}: \Delta]$$

$$(Post\ Summary) \quad \Delta ::= \bigvee(\pi\,;\theta\,;F)$$

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

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 */}
```

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

When reasoning about mallocN():

$$\frac{\{(\pi \wedge \pi_g\,;\theta\,;F)\}\ e\ \{(\pi\,;\theta\,;F)\}}{\{(\pi\,;\theta\,;F)\}\ \textbf{while}\ \pi_g\ \textbf{do}\ e\ \{(\pi \wedge \neg\pi_g\,;\theta\,;F)\}}\ [\text{FV-While}]$$

3. `while (i < n){`

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

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

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

5. `i = i+1;`

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

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

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

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

stack execution trace

- An instrumented semantics for the target language: $[s, \rho, F, e] \longrightarrow [s', \rho', F', v]$
- 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	Finally to close the file descriptor
	fclose, close	Globally do not access the file descriptor
		Read-only files cannot be written to
2. Threads	pthread_create	Finally to pthread_join or detach the thread
	pthread_mutex_lock	Finally to pthread_mutex_unlock
3. Memory	free	Globally do not access the pointer
	malloc	Finally free the new pointer
	realloc	Globally the old pointer is not accessed & finally free the new pointer
4. Sockets	socket	Finally to close the socket
5. Database	sqlite3_open	Finally to sqlite3_close the connection
6. URV/NPD	fgets, gethostbyaddr	Check the return value immediately after calls

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