Linear Temporal Logic & Specifying and Verifying Future Conditions

Yahui Song

Standard Chartered Bank @ Singapore

05 Feb 2026

Content

Linear Temporal Property (LTL)

LTL formulas:

$$\Phi ::= \bot \mid ap \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \mathcal{X}\,\Phi \mid \Phi_1\,\mathcal{U}\,\Phi_2 \mid \mathcal{F}\,\Phi \mid \mathcal{G}\,\Phi$$

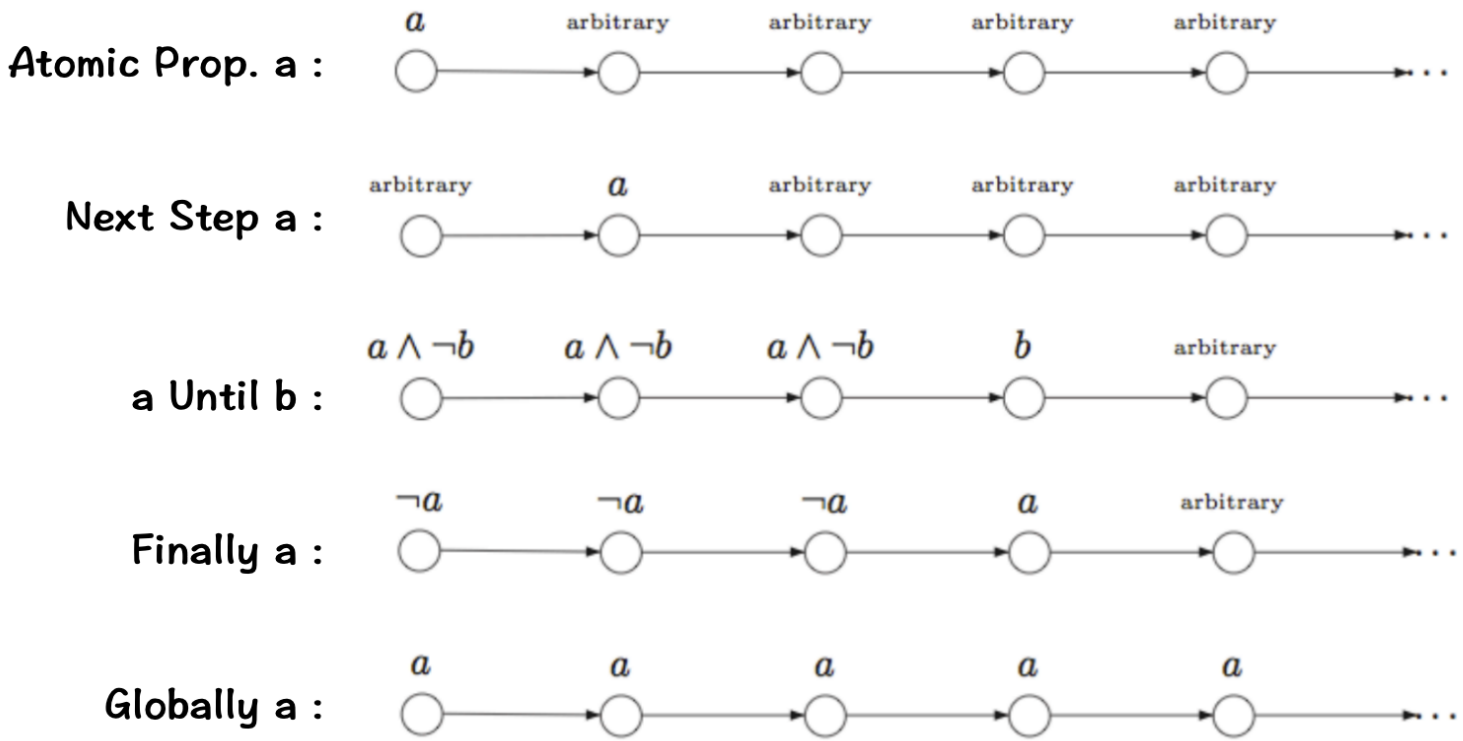
- ap : atomic proposition, eg., $v \geq 0$
- $\mathcal{X}$ (NeXt) : Φ is true at next step
- $\mathcal{U}$ (Until) : Φ_1 is always true until Φ_2 becomes true
- $\mathcal{F}$ (Finally) : Φ is eventually true
- $\mathcal{G}$ (Globally) : Φ is always true

Linear Temporal Property (LTL)

LTL formulas:

$$\Phi ::= \bot \mid ap \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \mathcal{X}\,\Phi \mid \Phi_1\,\mathcal{U}\,\Phi_2 \mid \mathcal{F}\,\Phi \mid \mathcal{G}\,\Phi$$

Evaluate LTL propositions over a sequence of states:



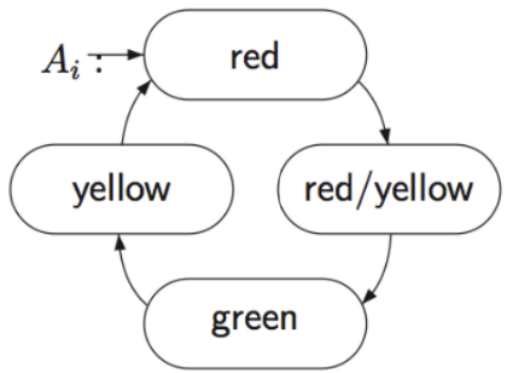
Linear Temporal Property (LTL)

LTL formulas:

$$\Phi ::= \bot \mid ap \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \mathcal{X}\,\Phi \mid \Phi_1\,\mathcal{U}\,\Phi_2 \mid \mathcal{F}\,\Phi \mid \mathcal{G}\,\Phi$$

Temporal Specifications:

- Liveness: "Traffic light is green infinitely often" $\triangleright$ $\mathcal{G}(\mathcal{F}\,green)$
- Ordering: "Once red, the light cannot become green immediately" $\triangleright$ $red \rightarrow \neg(\mathcal{X}\,green)$
- Responsiveness: "Every request will eventually lead to a response" $\triangleright$ $\mathcal{G}(request \rightarrow \mathcal{F}\,response)$



Temporal Logic-based Formal Methods

LTL formulas:

$\Phi ::= \bot \mid ap \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \mathcal{X}\,\Phi \mid \Phi_1\,\mathcal{U}\,\Phi_2 \mid \mathcal{F}\,\Phi \mid \mathcal{G}\,\Phi$

Regular Expressions:

$\Phi ::= \bot \mid \epsilon \mid \_ \mid event \mid \Phi_1 \cdot \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid \neg\Phi \mid \Phi^\star$

In formal methods, both LTL and regular expressions are used to specify temporal properties. The language inclusion problem ($\mathcal{A} \sqsubseteq \mathcal{B}$) is **decidable** for both formalisms.

Temporal Logic-based Formal Methods

LTL formulas:

$\Phi ::= \bot \mid ap \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \mathcal{X}\,\Phi \mid \Phi_1\,\mathcal{U}\,\Phi_2 \mid \mathcal{F}\,\Phi \mid \mathcal{G}\,\Phi$

Regular Expressions:

$\Phi ::= \bot \mid \epsilon \mid \_ \mid event \mid \Phi_1 \cdot \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid \neg\Phi \mid \Phi^\star$

In formal methods, both LTL and regular expressions are used to specify temporal properties. The language inclusion problem ($\mathcal{A} \sqsubseteq \mathcal{B}$) is **decidable** for both formalisms.

Formal Methods	Model Checking	Type & Effect system [1]	Verification [2,3]
Specification	LTL	Regular Expressions	Regular Expressions
Target	Models	Source Code	Source Code
Algorithm	Automata Inclusion	Type Checking	Term Rewriting System

Term Rewriting Regular Expressions

- Deciding the inclusion of regular expressions ($\mathcal{A} \sqsubseteq \mathcal{B}$)
- Flexible specifications, which can be combined with other logic
- Efficient entailment checker with inductive proofs.

Examples:

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

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

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

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

Post Conditions vs. Future Conditions

main.c

```
1  int x = 0;
2  void test()
3  // pre:    x=0
4  // post:   x=1
5
6  { x = x + 1;
7    return;
8  }
9
10 int main() {
11      test();
12      x = x + 2;
13 }
```

main.c

```
1  int x = 0;
2  void test()
3  // pre:    x=0
4  // post:   x=1
5  // future: Finally (x=3)
6  { x = x + 1;
7    return;
8  }
9
10 int main() {
11      test();
12      x = x + 2;
13 }
```

Future Conditions

Preventing memory leak, double-free and use-after-free

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

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

Preventing null-pointer dereference

```
(void) *ptr = v;
// pre: ∃ ptr. true
// post: true ∧ deref(ptr)
// future: (v=null ∧ G !_(ptr)) ∨ (v≠null ∧ _*)
```

Detecting Use-after-free

```
34    int paper(int argc, char **argv) {
35        char *buf1, *buf2, *buf3;
36        buf1 = (char *) malloc(1);
37        buf2 = (char *) malloc(1);
38        free(buf2); // free buf2R1
39        buf3 = (char *) malloc(1);
40        strncpy(buf2,argv[1],1); //4. UAF
41        free(buf1); free(buf3); }
42
```

```
5
The future condition is violated (ST) here at line 40
  Future condition is =
        (((((!free(v27))^* · free(v27) · (_)^*) /\
        ((!_(buf2))^*))) /\
        ((!free(v35))^* · free(v35) · (_)^*))
  Trace subtracted = strncpy(buf2)
  Pure =  buf1=v27∧buf2=v30∧v33=()∧buf3=v35∧v25=argv.1∧TRUE∧TRUE
```

Detecting Null-pointer Dereference

```
5   int main(int argc, char *argv[]) {
6       int *data = NULL;
7
8       if (argc > 1) {
9           data = malloc(sizeof(int));
10          *data = 100;
11      }
12
13      // Bug: data might be NULL if argc <= 1
14      printf("Data value: %d\n", *data);
15
16      free(data);
17      return 0;
18  }
```

```
1
The future condition is violated (ST) here at line 14
  Future condition is = (!_(*data))^*
  Trace subtracted = deref(*data)
  Pure =  *data=0∧0=0∧argc<=1∧TRUE
```

Specification Inference

Input Specification:

```
/*@ free(ptr)  =
    REQ  TRUE
    ENS (∃ r : r=unit ; free(ptr) ; (!_(ptr))^* ; r) @*/

/*@ malloc(size) =
    REQ TRUE
    ENS (∃ l : !(l=0) ; malloc(l)  ; (!free(l))^* · free(l) ·  (_)^* ; l)  @*/
```

Input program:

```
 5    int* create_array(int size) {
 6        if (size <= 0) {
 7            return NULL;  // Returns NULL for invalid size
 8        }
 9        return malloc(size * sizeof(int));
10    }
```

```
+--------------------+-----------------+
|        Inferred Specifications       |
+--------------------+-----------------+
/*@ create_array(size) =
REQ TRUE
ENS (:size<=0∧v0=0 ; ϵ  ; (_)^* ; v0) \/
    (:size>0∧v1=1∧v2=(size*v1)∧v3!=0 ; malloc(v3) ; (!free(v3))^* · free(v3) · (_)^* ; v3) @*/
```

Interprocedural Analysis

```
12  int main(int n) {
13      int *arr = create_array(n);  // Will return NULL
14
15      // Bug: dereferencing NULL return value
16      arr[0] = 100;
17
18      printf("First value: %d\n", arr[0]);
19      return 0;
20  }
```

```
1
The future condition is violated (ST) here at line 16
  Future condition is = (!_(*arr))^*
  Trace subtracted = deref(arr)
  Pure =  n<=0∧v13=0∧*arr=v13∧v13=0∧TRUE

2
The future condition is violated (Emp) here at line 20
  Future condition is = (!free(v14))^* · free(v14) · (_)^*
  Trace subtracted = ε
  Pure =  n>0∧v1=1∧v2=(n*v1)∧v14!=0∧*arr=v14∧v14!=0∧v10=100∧v11=arr.0∧v12=0
```

Summary and Possibilities

Main Contribution

- Compositional static analyzer via temporal properties
- Novel future-condition
- Specification Inference

Several aspects remain open for future work...

- Higher-order programs (Callbacks)
- OOP
- Buffer Overflow
- Machine checkable verification results (Rocq)
- ...

Thank you for listening!

References

[1] Yoji Nanjo et al. “A Fixpoint Logic and Dependent Effects for Temporal Property Verification”. In: Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Compu Ed. Anuj Dawar a Erich Grädel. ACM, 2018, s. 759–768. DOI: 10.1145/3209108.3209204. URL: https://doi.org/10.1145/3209108.3209204.

[2] Yahui Song, Darius Foo a Wei-Ngan Chin. “Specifying and Verifying Future Conditions”. In: Static Analysis - 32nd International Symposium, SAS 2025, Singapore, Oct Ed. Hakjoo Oh a Yulei Sui. Sv. 16100. Lecture Notes in Computer Science. Springer, 2025, s. 90–112. DOI: 10.1007/978-3-032-07106-4_5. URL: https://doi.org/10.1007/978-3-032-07106-4%5C_5.

[3] Yahui Song et al. “ProveNFix: Temporal Property-Guided Program Repair”. In: Proc. ACM Softw. Eng. 1.FSE (2024), s. 226–248. DOI: 10.1145/3643737. URL: https://doi.org/10.1145/3643737.