Lemur: Framework for Integrating LLMs in Automated Program Verification

Andrew (Haoze) Wu, Nina Narodytska, Clark Barrett

NII Shonan Meeting on " The Art of SAT"

Motivations

- LLMs demonstrate tremendous ability to understand programs
- Can perform various programming tasks
  - Program synthesis from natural languages
  - Code repair (DeepRepair)
- Recent work suggests LLMs can also generate program invariants

Can Large Language Models Reason about Program Invariants?

Kexin Pei [1 2] **David Bieber** [2] **Kensen Shi** [2] **Charles Sutton** [2] **Pengcheng Yin** [2]

Abstract

Identifying invariants is an important program analysis task with applications towards program

has proved challenging even for simple programs.

In the programming languages literature, one of the most important insights is to reason at the level of *abstractions* of

LLMs as invariant generators: example

Wu

```
int main() {
unsigned int x, y;
x = 0U;
y = 4U;
while (1) {
x = x + y;
y = y + 4U;
assert(x != 30U);
}
return 0;
```

Print loop invariants that help prove the assertion as C assertions.

```
while (1) {
    // Loop invariant 2: y is always divisible by 4U
    assert(y % 4U == 0);

    // Loop invariant 3: x + y is always divisible by 4U
    assert((x + y) % 4U == 0);

    x = x + y;
    y = y + 4U;
```

Can generate insightful loop invariants using operators not in the program

LLMs as invariant generators

- Compared with existing learning-based invariant generator
  - Does not require domain-specific learning (though it might help)
  - Can process programs of various forms

5 Experiments

We evaluate CODE2INV on a suite of 133 benchmark programs from recent works [3, 7, 8] and the 2017 SyGuS competition [31].[2] Each program consists of three parts: a number of assumption or assignment statements, one loop which contains nested if-else statements with arithmetic operations, and one assertion statement.

Learning Loop Invariants for Program Verification, Si et al, NeurIPS, 2018

Can we leverage LLMs’ code-understanding ability for automated program verification?

Challenges

- GPT outputs can be messy and hard to automatically process
  - **Need prompt engineering**

- LLM outputs are only suggestions and can certainly be wrong!
  - Does not hold
  - Does not help prove the proof goal
  - **Need formal verifiers to check correctness and implication**

Marker-driven Prompting

- To reliably get rid of the natural language
  - **In-context learning + “*don’t explain*”**

Wu

```c
void *malloc(unsigned int size);
long SIZE;
const int MAX = 100000;
int main() {
SIZE = __VERIFIER_nondet_long();
if (SIZE > 1 && SIZE < MAX) {
int i;
long *a = malloc(sizeof(long) * SIZE);
long sum = 0;
for (i = 0; i < SIZE; i++) {
a[i] = 1;
}
for (i = 0; i < SIZE; i++) {
sum = sum + a[i];
}
for (i = 0; i < SIZE; i++) {
sum = sum - a[i];
}
assert(sum == 0);
}
return 1;
}
```

Print loop invariants that help prove the assertion as valid C assertions. Don't explain. Your answer should be 'assert(...); // line number'

Attempt 1:

```c
assert(SIZE > 1 && SIZE < MAX); // line 6
assert(i >= 0 && i <= SIZE); // line 10
assert(sum == i); // line 15
assert(sum == 0 && i == SIZE); // line 20
```

Off-by-one ☹

Attempt 2:

```c
assert(sum == 0); // line 17
assert(sum == SIZE * 1 || sum == -SIZE * 1); // line 14
assert(sum == SIZE * 1); // line 10
assert(i >= 0 && i <= SIZE); // line 8
assert(i >= 0 && i <= SIZE); // line 6
assert(SIZE > 1 && SIZE < MAX); // line 5
```

Marker-driven Prompting

- To reliably obtain assertions at a precise location
  - **Introduce markers to the program**

Wu

```
void *malloc(unsigned int size);
long SIZE;
const int MAX = 100000;
int main() {
SIZE = __VERIFIER_nondet_long();
if (SIZE > 1 && SIZE < MAX) {
int i;
long *a = malloc(sizeof(long) * SIZE);
long sum = 0;
for (i = 0; i < SIZE; i++) {
// Line A
a[i] = 1;
}
for (i = 0; i < SIZE; i++) {
// Line B
sum = sum + a[i];
}
for (i = 0; i < SIZE; i++) {
sum = sum - a[i];
}
assert(sum == 0);
}
return 1;
}
```

Print loop invariants at Lines A, B that help prove the assertion as valid C assertions. Don't explain. Your answer should be 'assert(...); // line name'

c

```c
assert(i >= 0 && i <= SIZE); // Line A
assert(sum == i); // Line B
```

More failure modes

- Invalid invariants (especially for multi-loop programs)

Wu

```c
void *malloc(unsigned int size);
long SIZE;
const int MAX = 100000;
int main() {
SIZE = (long) rand();
if (SIZE > 1 && SIZE < MAX) {
int i;
long *a = malloc(sizeof(long) * SIZE);
long sum = 0;
for (i = 0; i < SIZE; i++) {
// Line A
a[i] = 1;
}
for (i = 0; i < SIZE; i++) {
// Line B
sum = sum + a[i];
}
for (i = 0; i < SIZE; i++) {
// Line C
sum = sum - a[i];
}
assert(sum == 0);
}
return 1;
}
```

Print loop invariants at Lines A, B, C that help prove the assertion as valid C assertions. Don't explain. Your answer should be 'assert(...); // line name'

Attempt 1: ✗

```c
assert(i >= 0 && i <= SIZE); // Line A
assert(sum == (i * 1)); // Line B
assert(sum == 0); // Line C
```

Attempt 2: ✗

```c
assert(i >= 0 && i <= SIZE); // Line A
assert(sum == i); // Line B
assert(sum == 0 && i == SIZE); // Line C
```

Attempt 3: ✓

```c
assert(i >= 0 && i <= SIZE); // Line A
assert(sum == i); // Line B
assert(sum == SIZE - i); // Line C
```

Multiple prompting attempts
Repair the proposal

More failure modes

- Verifier can prove that the invariant imply the property but cannot prove the invariant

Constructing a chain of deduction

The new proof goal → `assert(sum == SIZE - i);`

Wu

```
void *malloc(unsigned int size);
long SIZE;
const int MAX = 100000;
int main() {
SIZE = (long) rand();
if (SIZE > 1 && SIZE < MAX) {
int i;
long *a = malloc(sizeof(long) * SIZE);
long sum = 0;
for (i = 0; i < SIZE; i++) {
// Line A
a[i] = 1;
}
for (i = 0; i < SIZE; i++) {
// Line B
sum = sum + a[i];
}
for (i = 0; i < SIZE; i++) {
assert(sum == SIZE - i);
sum = sum - a[i];
}
}
return 1;
}
```

Print loop invariants at Lines A, B that help prove the assertion as valid C assertions. Don't explain. Your answer should be 'assert(...); // line name'

Integrating the LLM with the Verifier

Input: a program P, an assertion p
Output: Whether p holds

LLMs can

- Suggest proof goal
- Strengthen proof goal
- Repair proof goal

Program verifiers can

- Check implication
- Check proof goal
- Provide feedback (unknown, counter-example)

LLM-driven proof procedure as a calculus

$$\frac{\mathcal{M} = \mathcal{M}' :: p \quad \mathcal{V}(\mathcal{P}, \mathcal{A}, p) = \textsc{Unknown} \quad q \in \mathcal{O}_{\text{propose}}(\mathcal{P}, p)}{\mathcal{P}, \mathcal{A}, \mathcal{M} \Longrightarrow \mathcal{P}, \{q\}, \mathcal{M}} \ (\textbf{Propose})$$

$$\frac{\mathcal{A} = \{q\} \quad \mathcal{M} = \mathcal{M}' :: p \quad \mathcal{V}(\mathcal{P}, \mathcal{A}, p) = \textsc{True}}{\mathcal{P}, \mathcal{A}, \mathcal{M} \Longrightarrow \mathcal{P}, \varnothing, \mathcal{M} :: q} \ (\textbf{Decide})$$

$$\frac{\mathcal{M} = \mathcal{M}' :: p :: q \quad \mathcal{V}(\mathcal{P}, \mathcal{A}, q) \neq \textsc{True} \quad q' \in \mathcal{O}_{\text{propose}}(\mathcal{P}, p)}{\mathcal{P}, \mathcal{A}, \mathcal{M} \Longrightarrow \mathcal{P}, \{q'\}, \mathcal{M}' :: p} \ (\textbf{Backtrack})$$

$$\frac{\mathcal{A} = \{q\} \quad \mathcal{M} = \mathcal{M}' :: p \quad \mathcal{V}(\mathcal{P}, \mathcal{A}, p) = \textsc{Unknown} \quad q' \in \mathcal{O}_{\text{repair}}(\mathcal{P}, p, q, \textsc{Unknown})}{\mathcal{P}, \mathcal{A}, \mathcal{M} \Longrightarrow \mathcal{P}, \{q'\}, \mathcal{M}' :: p} \ (\textbf{Repair 1})$$

$$\frac{\mathcal{A} = \varnothing \quad \mathcal{M} = \mathcal{M}' :: p :: q \quad \mathcal{V}(\mathcal{P}, \mathcal{A}, q) = \textsc{False} \quad q' \in \mathcal{O}_{\text{repair}}(\mathcal{P}, p, q, \textsc{False})}{\mathcal{P}, \mathcal{A}, \mathcal{M} \Longrightarrow \mathcal{P}, \{q'\}, \mathcal{M}' :: p} \ (\textbf{Repair 2})$$

$$\frac{\mathcal{A} = \varnothing \quad \mathcal{M} = \mathcal{M}' :: p \quad \mathcal{V}(\mathcal{P}, \mathcal{A}, p) = \textsc{True}}{\mathcal{P}, \mathcal{A}, \mathcal{M} \Longrightarrow \textsc{success}} \ (\textbf{Success 1})$$

$$\frac{\mathcal{A} = \varnothing \quad \mathcal{M} = \mathcal{M}' :: p :: q \quad \mathcal{S}(\mathcal{P}, q) \quad \mathcal{V}(\mathcal{P}, \{\neg q\}, p) = \textsc{True}}{\mathcal{P}, \mathcal{A}, \mathcal{M} \Longrightarrow \textsc{success}} \ (\textbf{Success 2})$$

$$\frac{\mathcal{M} = [p] \quad \mathcal{V}(\mathcal{P}, \mathcal{A}, p) = \textsc{False}}{\mathcal{P}, \mathcal{A}, \mathcal{M} \Longrightarrow \textsc{fail}} \ (\textbf{Fail})$$

Figure 1: Deductive rules of the LEMUR calculus.

LLM-driven proof procedure as a calculus

$$\frac{\mathcal{M} = \mathcal{M}' :: p \quad \mathcal{V}(\mathcal{P}, \mathcal{A}, p) = \text{UNKNOWN} \quad q \in \mathcal{O}_{\text{propose}}(\mathcal{P}, p)}{\mathcal{P}, \mathcal{A}, \mathcal{M} \Longrightarrow \mathcal{P}, \{q\}, \mathcal{M}} \ (\textbf{Propose})$$

$$\frac{\mathcal{A} = \{q\} \quad \mathcal{M} = \mathcal{M}' :: p \quad \mathcal{V}(\mathcal{P}, \mathcal{A}, p) = \text{TRUE}}{\mathcal{P}, \mathcal{A}, \mathcal{M} \Longrightarrow \mathcal{P}, \varnothing, \mathcal{M} :: q} \ (\textbf{Decide})$$

$$\frac{\mathcal{M} = \mathcal{M}' :: p :: q \quad \mathcal{V}(\mathcal{P}, \mathcal{A}, q) \neq \text{TRUE} \quad q' \in \mathcal{O}_{\text{propose}}(\mathcal{P}, p)}{\mathcal{P}, \mathcal{A}, \mathcal{M} \Longrightarrow \mathcal{P}, \{q'\}, \mathcal{M}' :: p} \ (\textbf{Backtrack})$$

$$\frac{\mathcal{A} = \{q\} \quad \mathcal{M} = \mathcal{M}' :: p \quad \mathcal{V}(\mathcal{P}, \mathcal{A}, p) = \text{UNKNOWN} \quad q' \in \mathcal{O}_{\text{repair}}(\mathcal{P}, p, q, \text{UNKNOWN})}{\mathcal{P}, \mathcal{A}, \mathcal{M} \Longrightarrow \mathcal{P}, \{q'\}, \mathcal{M}' :: p} \ (\textbf{Repair 1})$$

$$\frac{\mathcal{A} = \varnothing \quad \mathcal{M} = \mathcal{M}' :: p :: q \quad \mathcal{V}(\mathcal{P}, \mathcal{A}, q) = \text{FALSE} \quad q' \in \mathcal{O}_{\text{repair}}(\mathcal{P}, p, q, \text{FALSE})}{\mathcal{P}, \mathcal{A}, \mathcal{M} \Longrightarrow \mathcal{P}, \{q'\}, \mathcal{M}' :: p} \ (\textbf{Repair 2})$$

$$\frac{\mathcal{A} = \varnothing \quad \mathcal{M} = \mathcal{M}' :: p \quad \mathcal{V}(\mathcal{P}, \mathcal{A}, p) = \text{TRUE}}{\mathcal{P}, \mathcal{A}, \mathcal{M} \Longrightarrow \text{SUCCESS}} \ (\textbf{Success 1})$$

$$\frac{\mathcal{A} = \varnothing \quad \mathcal{M} = \mathcal{M}' :: p :: q \quad \mathcal{S}(\mathcal{P}, q) \quad \mathcal{V}(\mathcal{P}, \{\neg q\}, p) = \text{TRUE}}{\mathcal{P}, \mathcal{A}, \mathcal{M} \Longrightarrow \text{SUCCESS}} \ (\textbf{Success 2})$$

$$\frac{\mathcal{M} = [p] \quad \mathcal{V}(\mathcal{P}, \mathcal{A}, p) = \text{FALSE}}{\mathcal{P}, \mathcal{A}, \mathcal{M} \Longrightarrow \text{FAIL}} \ (\textbf{Fail})$$

Figure 1: Deductive rules of the LEMUR calculus.

LLM-driven proof procedure as a calculus

$$\frac{\mathcal{M} = \mathcal{M}' :: p \quad \mathcal{V}(\mathcal{P}, \mathcal{A}, p) = \textsc{Unknown} \quad q \in \mathcal{O}_{\text{propose}}(\mathcal{P}, p)}{\mathcal{P}, \mathcal{A}, \mathcal{M} \Longrightarrow \mathcal{P}, \{q\}, \mathcal{M}} \ (\textbf{Propose})$$

$$\frac{\mathcal{A} = \{q\} \quad \mathcal{M} = \mathcal{M}' :: p \quad \mathcal{V}(\mathcal{P}, \mathcal{A}, p) = \textsc{True}}{\mathcal{P}, \mathcal{A}, \mathcal{M} \Longrightarrow \mathcal{P}, \varnothing, \mathcal{M} :: q} \ (\textbf{Decide})$$

$$\frac{\mathcal{M} = \mathcal{M}' :: p :: q \quad \mathcal{V}(\mathcal{P}, \mathcal{A}, q) \neq \textsc{True} \quad q' \in \mathcal{O}_{\text{propose}}(\mathcal{P}, p)}{\mathcal{P}, \mathcal{A}, \mathcal{M} \Longrightarrow \mathcal{P}, \{q'\}, \mathcal{M}' :: p} \ (\textbf{Backtrack})$$

$$\frac{\mathcal{A} = \{q\} \quad \mathcal{M} = \mathcal{M}' :: p \quad \mathcal{V}(\mathcal{P}, \mathcal{A}, p) = \textsc{Unknown} \quad q' \in \mathcal{O}_{\text{repair}}(\mathcal{P}, p, q, \textsc{Unknown})}{\mathcal{P}, \mathcal{A}, \mathcal{M} \Longrightarrow \mathcal{P}, \{q'\}, \mathcal{M}' :: p} \ (\textbf{Repair 1})$$

$$\frac{\mathcal{A} = \varnothing \quad \mathcal{M} = \mathcal{M}' :: p :: q \quad \mathcal{V}(\mathcal{P}, \mathcal{A}, q) = \textsc{False} \quad q' \in \mathcal{O}_{\text{repair}}(\mathcal{P}, p, q, \textsc{False})}{\mathcal{P}, \mathcal{A}, \mathcal{M} \Longrightarrow \mathcal{P}, \{q'\}, \mathcal{M}' :: p} \ (\textbf{Repair 2})$$

$$\frac{\mathcal{A} = \varnothing \quad \mathcal{M} = \mathcal{M}' :: p \quad \mathcal{V}(\mathcal{P}, \mathcal{A}, p) = \textsc{True}}{\mathcal{P}, \mathcal{A}, \mathcal{M} \Longrightarrow \textsc{success}} \ (\textbf{Success 1})$$

$$\frac{\mathcal{A} = \varnothing \quad \mathcal{M} = \mathcal{M}' :: p :: q \quad \mathcal{S}(\mathcal{P}, q) \quad \mathcal{V}(\mathcal{P}, \{\neg q\}, p) = \textsc{True}}{\mathcal{P}, \mathcal{A}, \mathcal{M} \Longrightarrow \textsc{success}} \ (\textbf{Success 2})$$

$$\frac{\mathcal{M} = [p] \quad \mathcal{V}(\mathcal{P}, \mathcal{A}, p) = \textsc{False}}{\mathcal{P}, \mathcal{A}, \mathcal{M} \Longrightarrow \textsc{fail}} \ (\textbf{Fail})$$

Figure 1: Deductive rules of the LEMUR calculus.

LLM-driven proof procedure as a calculus

$$\frac{\mathcal{M} = \mathcal{M}' :: p \quad \mathcal{V}(\mathcal{P}, \mathcal{A}, p) = \text{UNKNOWN} \quad q \in \mathcal{O}_{\text{propose}}(\mathcal{P}, p)}{\mathcal{P}, \mathcal{A}, \mathcal{M} \Longrightarrow \mathcal{P}, \{q\}, \mathcal{M}} \ (\textbf{Propose})$$

$$\frac{\mathcal{A} = \{q\} \quad \mathcal{M} = \mathcal{M}' :: p \quad \mathcal{V}(\mathcal{P}, \mathcal{A}, p) = \text{TRUE}}{\mathcal{P}, \mathcal{A}, \mathcal{M} \Longrightarrow \mathcal{P}, \varnothing, \mathcal{M} :: q} \ (\textbf{Decide})$$

$$\frac{\mathcal{M} = \mathcal{M}' :: p :: q \quad \mathcal{V}(\mathcal{P}, \mathcal{A}, q) \neq \text{TRUE} \quad q' \in \mathcal{O}_{\text{propose}}(\mathcal{P}, p)}{\mathcal{P}, \mathcal{A}, \mathcal{M} \Longrightarrow \mathcal{P}, \{q'\}, \mathcal{M}' :: p} \ (\textbf{Backtrack})$$

$$\frac{\mathcal{A} = \{q\} \quad \mathcal{M} = \mathcal{M}' :: p \quad \mathcal{V}(\mathcal{P}, \mathcal{A}, p) = \text{UNKNOWN} \quad q' \in \mathcal{O}_{\text{repair}}(\mathcal{P}, p, q, \text{UNKNOWN})}{\mathcal{P}, \mathcal{A}, \mathcal{M} \Longrightarrow \mathcal{P}, \{q'\}, \mathcal{M}' :: p} \ (\textbf{Repair 1})$$

$$\frac{\mathcal{A} = \varnothing \quad \mathcal{M} = \mathcal{M}' :: p :: q \quad \mathcal{V}(\mathcal{P}, \mathcal{A}, q) = \text{FALSE} \quad q' \in \mathcal{O}_{\text{repair}}(\mathcal{P}, p, q, \text{FALSE})}{\mathcal{P}, \mathcal{A}, \mathcal{M} \Longrightarrow \mathcal{P}, \{q'\}, \mathcal{M}' :: p} \ (\textbf{Repair 2})$$

$$\frac{\mathcal{A} = \varnothing \quad \mathcal{M} = \mathcal{M}' :: p \quad \mathcal{V}(\mathcal{P}, \mathcal{A}, p) = \text{TRUE}}{\mathcal{P}, \mathcal{A}, \mathcal{M} \Longrightarrow \text{SUCCESS}} \ (\textbf{Success 1})$$

$$\frac{\mathcal{A} = \varnothing \quad \mathcal{M} = \mathcal{M}' :: p :: q \quad \mathcal{S}(\mathcal{P}, q) \quad \mathcal{V}(\mathcal{P}, \{\neg q\}, p) = \text{TRUE}}{\mathcal{P}, \mathcal{A}, \mathcal{M} \Longrightarrow \text{SUCCESS}} \ (\textbf{Success 2})$$

$$\frac{\mathcal{M} = [p] \quad \mathcal{V}(\mathcal{P}, \mathcal{A}, p) = \text{FALSE}}{\mathcal{P}, \mathcal{A}, \mathcal{M} \Longrightarrow \text{FAIL}} \ (\textbf{Fail})$$

Figure 1: Deductive rules of the LEMUR calculus.

LLM-driven proof procedure as a calculus

Theorem 3.1 (Soundness). *Given a property* p *and a program* $\mathcal{P}$*, if* SUCCESS *is reached by a sequence of valid rule applications starting from* $\langle \mathcal{P}, \varnothing, [p_0] \rangle$*, then* p_0 *is an invariant on* $\mathcal{P}$.

Theorem 3.2 (Soundness 2). *Given a property* p *and a program* $\mathcal{P}$*, if* FAIL *is reached by a sequence of valid rule applications starting from* $\langle \mathcal{P}, \varnothing, [p_0] \rangle$*, then* p_0 *is not an invariant on* $\mathcal{P}$.

LLM-driven proof procedure as a calculus

Theorem 3.1 (Soundness). *Given a property p and a program $\mathcal{P}$, if* SUCCESS *is reached by a sequence of valid rule applications starting from $\langle \mathcal{P}, \varnothing, [p_0] \rangle$, then p_0 is an invariant on $\mathcal{P}$.*

Theorem 3.2 (Soundness 2). *Given a property p and a program $\mathcal{P}$, if* FAIL *is reached by a sequence of valid rule applications starting from $\langle \mathcal{P}, \varnothing, [p_0] \rangle$, then p_0 is not an invariant on $\mathcal{P}$.*

Does not terminate

Algorithm 1

Algorithm 1 The LEMUR procedure

Input: A program $\mathcal{P}$, a property p.
Output: SUCCESS only if $\text{Inv}(\mathcal{P}, p)$; FAIL only if $\neg\text{Inv}(\mathcal{P}, p)$; and UNKNOWN if inconclusive.
Parameters: Verifier $\mathcal{V}$, oracles $\mathcal{O}_{\text{propose}}$ and $\mathcal{O}_{\text{repair}}$ (which satisfy Condition 1), number of proposals $\mathbf{k}$
function lemur_check($\mathcal{P}, p$)
    $d \mapsto \mathcal{V}(\mathcal{P}, \varnothing, p)$
    if $d =$ FALSE **then return** FAIL ▷ **Fail**
    else if $d =$ TRUE **then return** SUCCESS ▷ **Success 1**
    else
        $i, Q \mapsto 0, \mathcal{O}_{\text{propose}}(\mathcal{P}, p)$
        while $i < \mathbf{k} \wedge |Q| > 0$ **do**
            $i \mapsto i + 1$
            $q \mapsto \text{pop}(Q)$
            $e \mapsto \mathcal{V}(\mathcal{P}, \{q\}, p)$ ▷ **Propose/Backtrack**
            if $e =$ FALSE **then return** FAIL ▷ **Fail**
            else if $e =$ TRUE **then**
                $f \mapsto \text{lemur\_check}(\mathcal{P}, q)$ ▷ **Decide**
                if $f =$ SUCCESS **then return** SUCCESS ▷ **Success 1**
                else if $\mathcal{S}(\mathcal{P}, q) \wedge (\mathcal{V}(\mathcal{P}, \{\neg q\}, p) = \text{TRUE})$ **then return** SUCCESS ▷ **Success 2**
                else if $f =$ FAIL **then** $Q \mapsto \text{join}(Q, \mathcal{O}_{\text{repair}}(\mathcal{P}, p, q, \text{FALSE}))$ ▷ **Repair 2**
                else continue
            else $Q \mapsto \text{join}(Q, \mathcal{O}_{\text{repair}}(\mathcal{P}, p, q, \text{UNKNOWN}))$ ▷ **Repair 1**
        return UNKNOWN

LLM-driven proof procedure as a calculus

$\mathcal{V}$: UNKNOWN

```
uint32_t x=0;
while (rand()){
x+=4;
assert(x!=30);
}
```

Figure 2: Running example.

LLM-driven proof procedure as a calculus

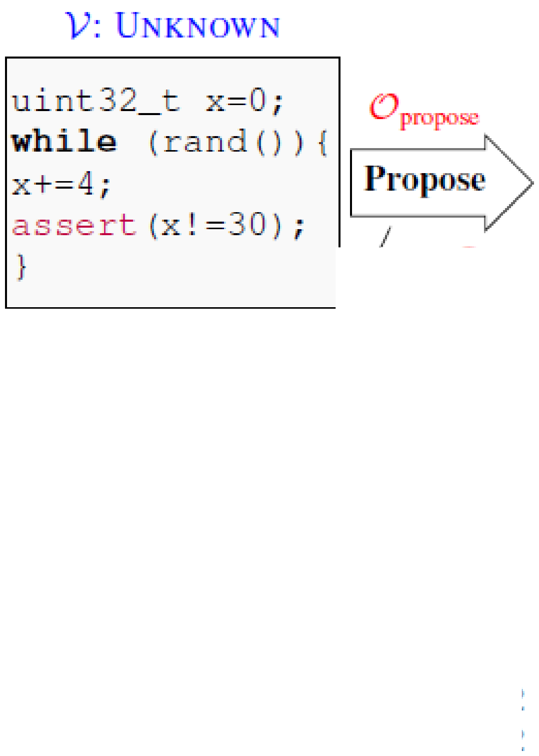

Figure 2: Running example.

LLM-driven proof procedure as a calculus

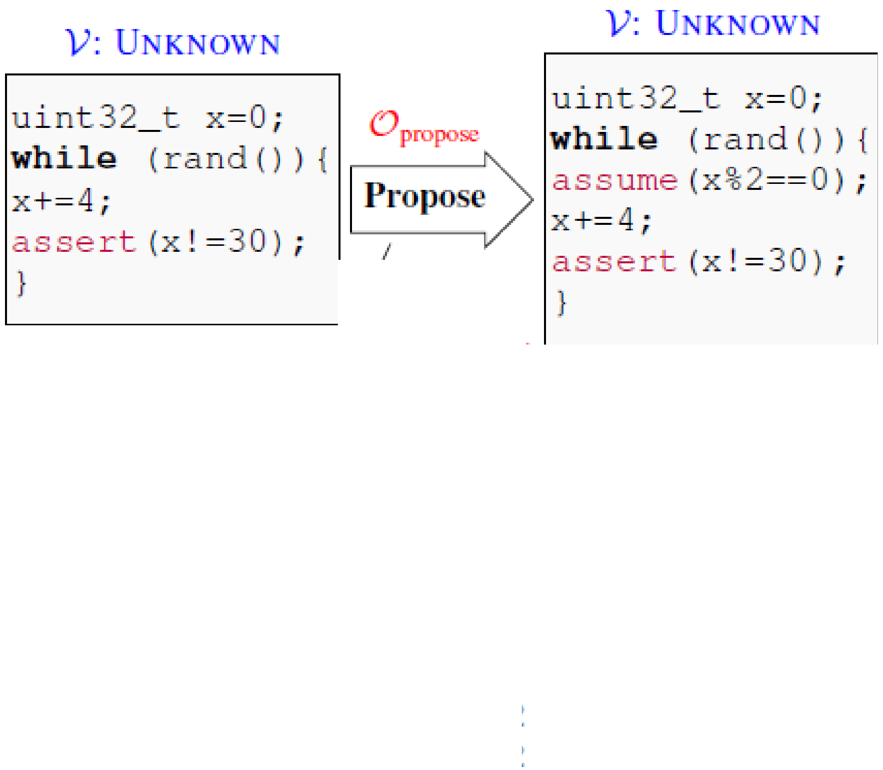

Figure 2: Running example.

LLM-driven proof procedure as a calculus

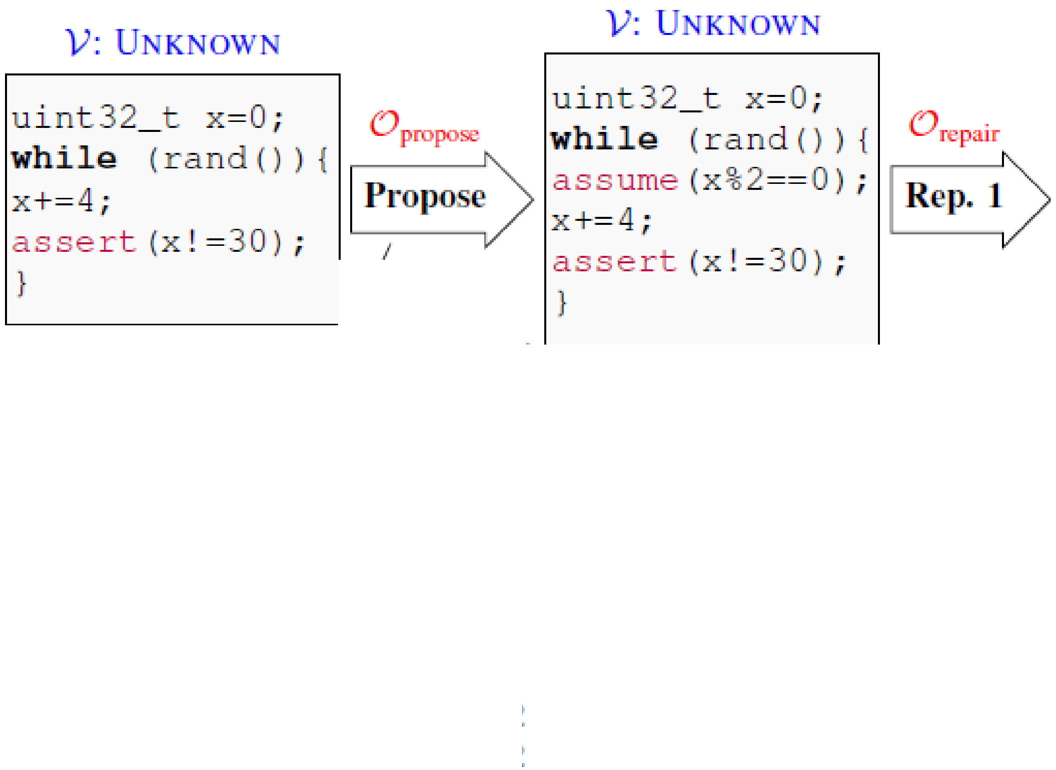

Figure 2: Running example.

LLM-driven proof procedure as a calculus

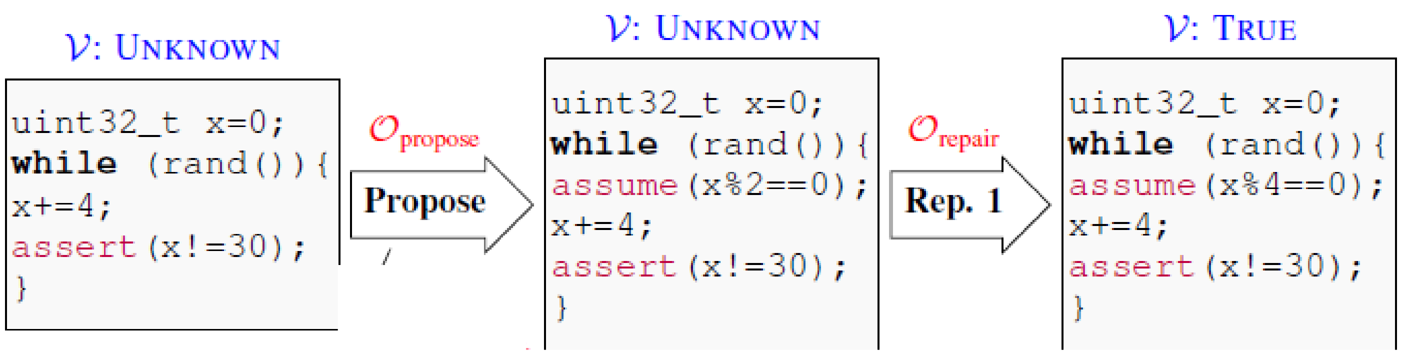

Figure 2: Running example.

LLM-driven proof procedure as a calculus

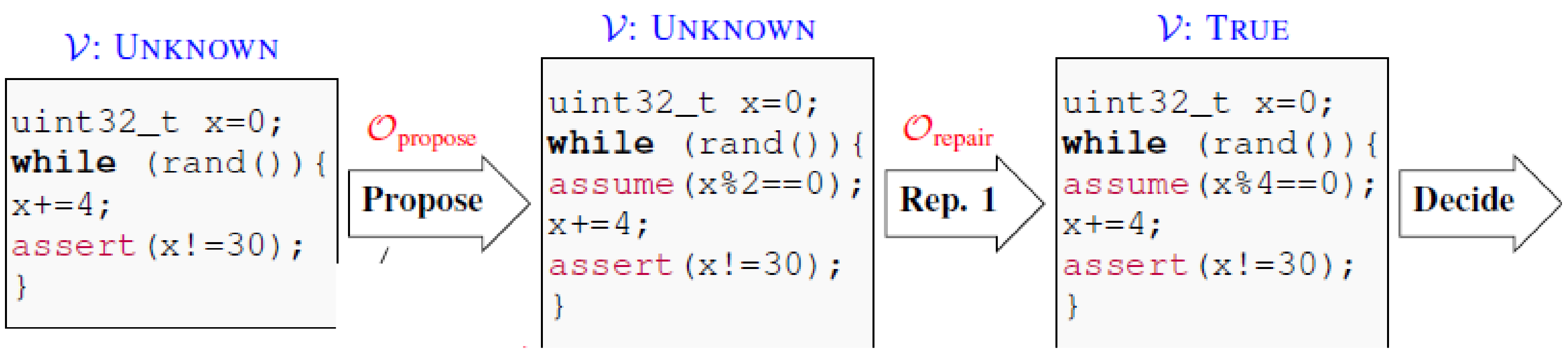

Figure 2: Running example.

LLM-driven proof procedure as a calculus

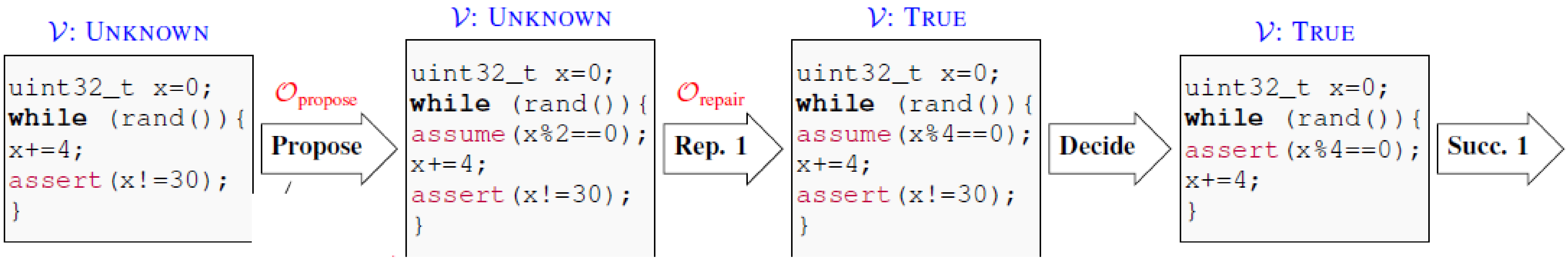

Figure 2: Running example.

LLM-driven proof procedure as a calculus

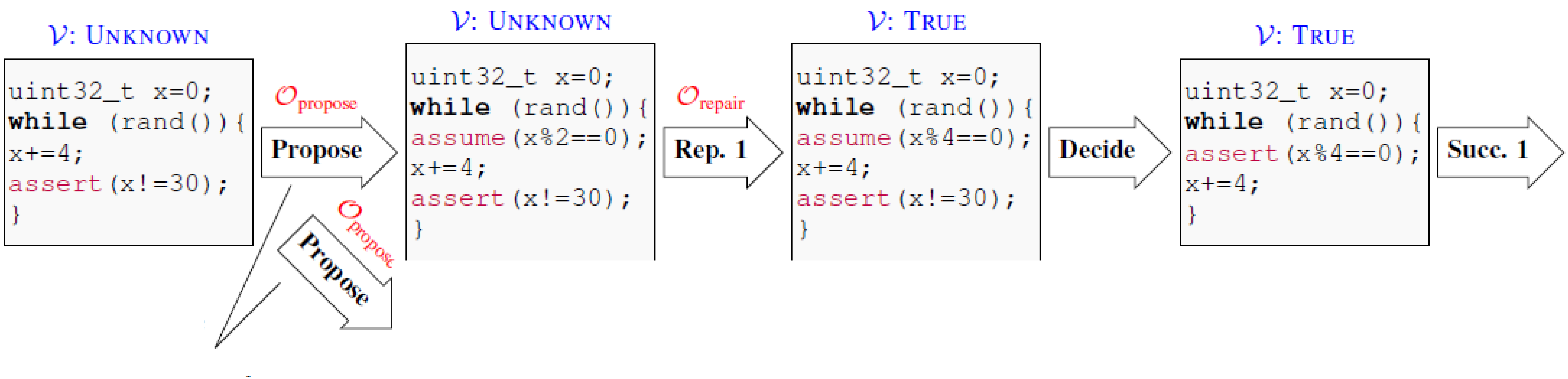

Figure 2: Running example.

LLM-driven proof procedure as a calculus

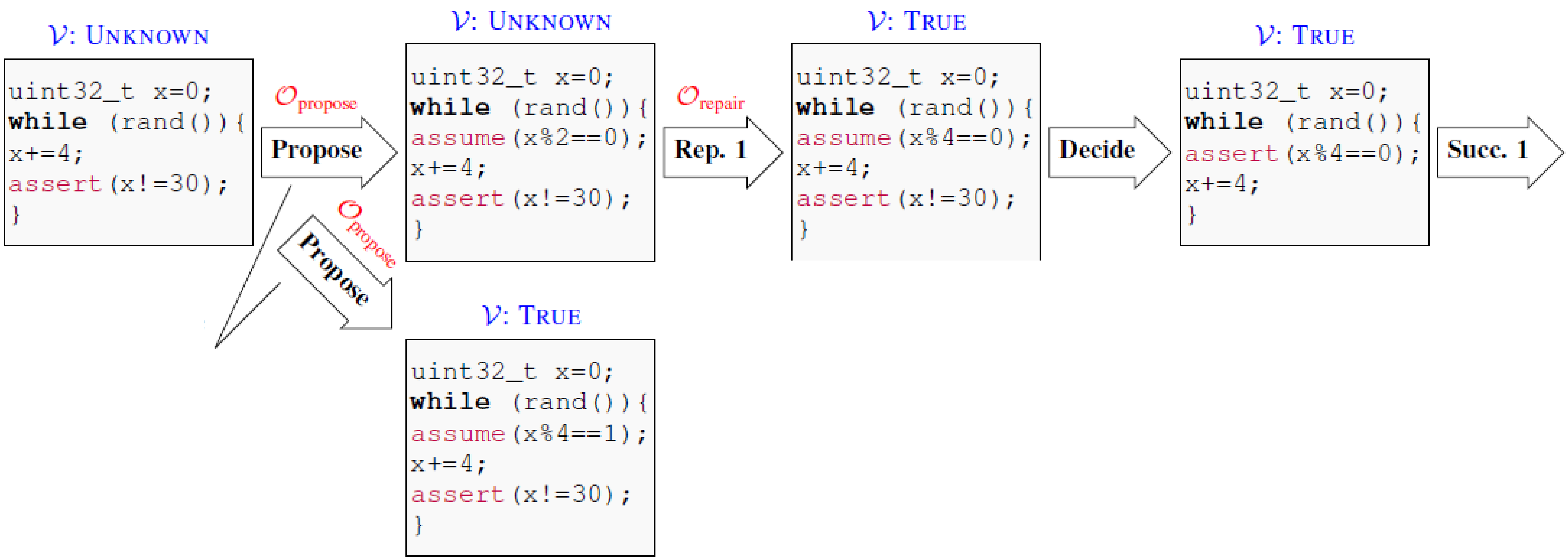

Figure 2: Running example.

LLM-driven proof procedure as a calculus

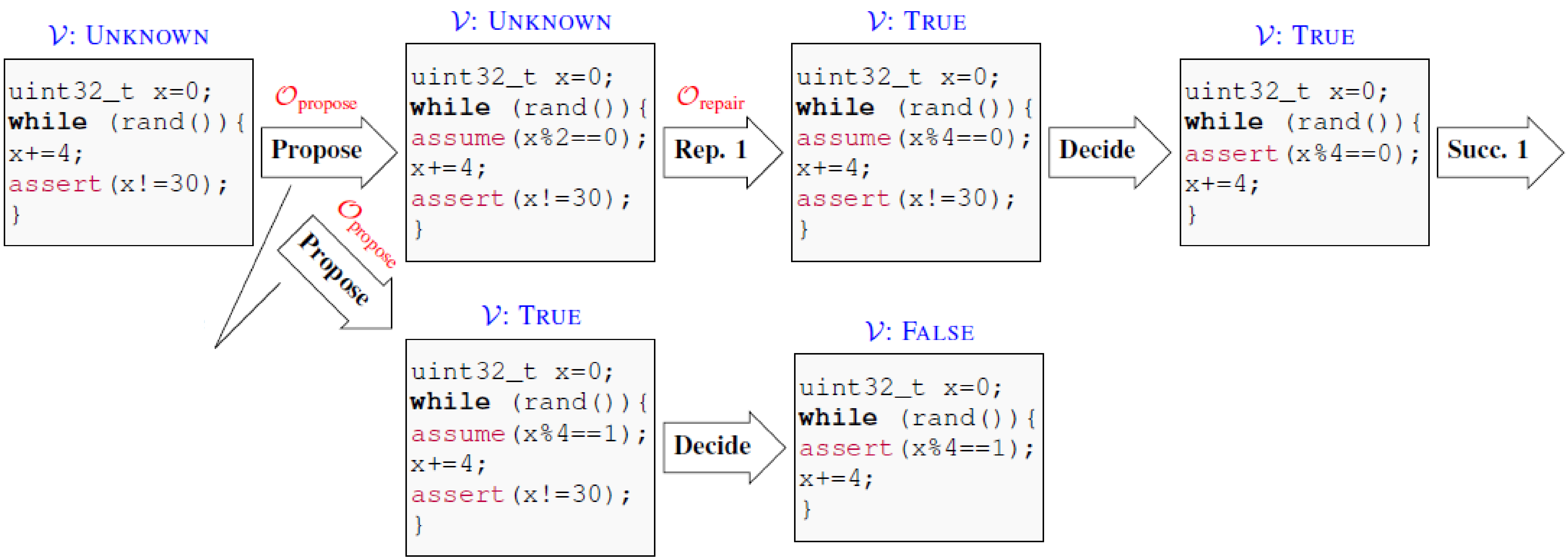

Figure 2: Running example.

LLM-driven proof procedure as a calculus

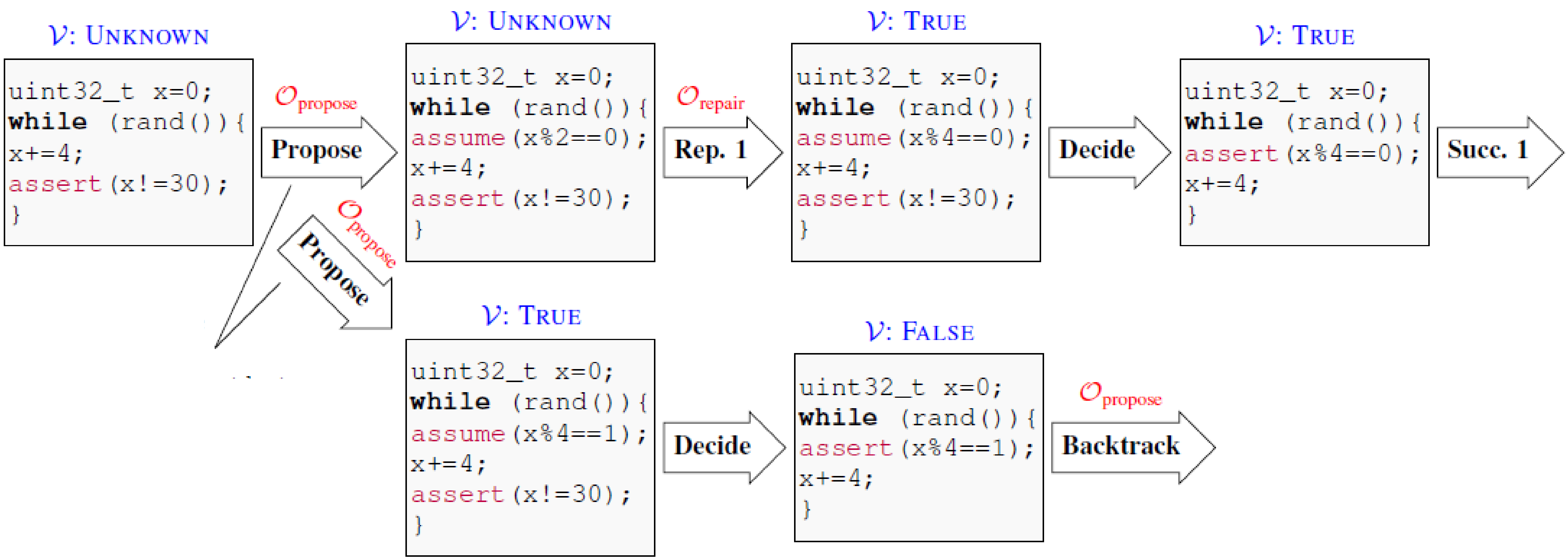

Figure 2: Running example.

LLM-driven proof procedure as a calculus

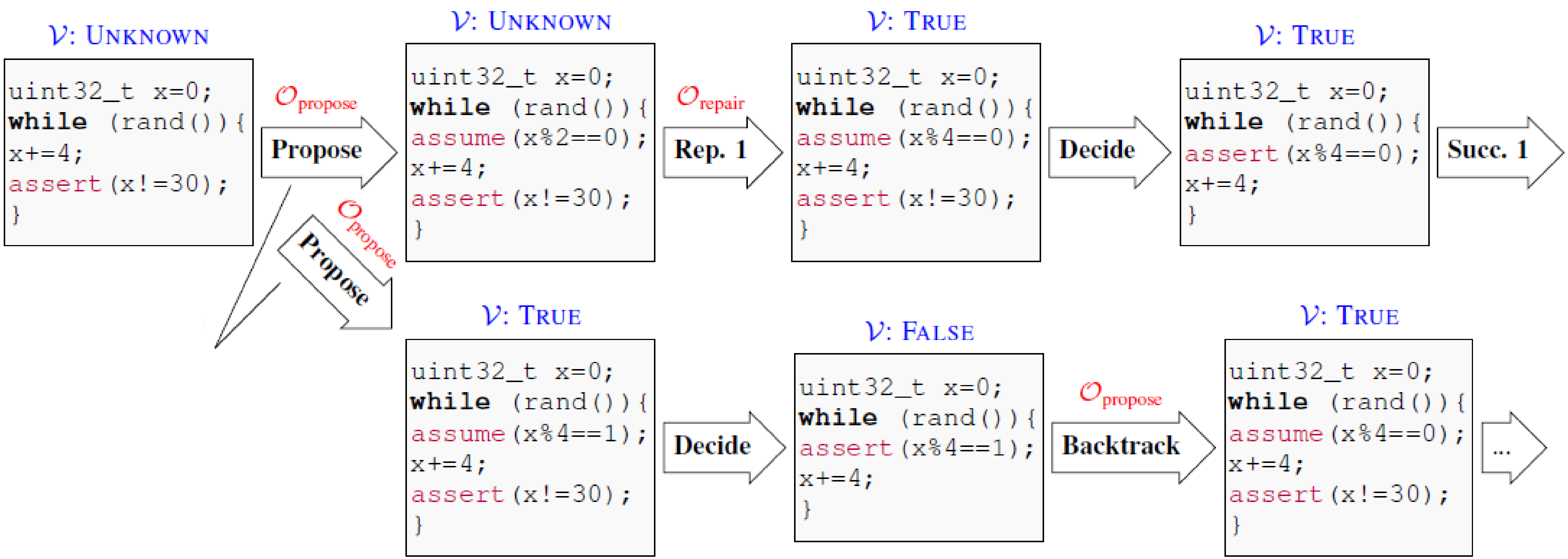

Figure 2: Running example.

Implementation

- ~1500 lines of Python code
- LLM: GPT families OpenAI
  - Use OpenAPI for prompting
  - default: GPT 4
- Verifier: cbmc, UAutomizer and esbmc
  - default: esbmc + UAutomizer

Experiment: synthetic benchmarks

- 133 loop Invariant generation benchmarks
- Goal: find a real invariant that implies the property
- Configurations
  - Code2Inv: a learned loop-invariant generator
  - esbmc: a k-induction-based C model checker
  - esbmc + LLM: use LLM to propose invariants

```
int main() {
int x;
int y;
assume((x >= 0));
assume((x <= 2));
assume((y <= 2));
assume((y >= 0));
while (unknown()) {
{
(x = (x + 2));
(y = (y + 2));
}
}
if ((y == 0)) {
assert((x != 4));
}
}
```

```
int main() {
int x;
int y;
(x = 1);
(y = 0);
while ((y < 100000)) {
{
(x = (x + y));
(y = (y + 1));
}
}
assert((x >= y));
}
```

Configuration	Time limit	Solved	# attempted proposal
Code2Inv	1 hour	92	> 20
esbmc	10 minutes	68	0
esbmc + LLM	10 minutes	**107**	**4.7**

Experiment: competition benchmarks

- 50 short C reachability benchmarks from SV-Comp 2023
- Unsolved by esbmc and Uautomizer in 20 minutes
- Configurations
  - *esbmc*: second best non-portfolio solver
  - *UAutomizer*: predicate-abstraction-based solver, overall winner of SV-Comp 2023
  - *esbmc + UAutomizer + LLM*

Configuration	Time limit	Solved
UAutomizer	20 minutes	0
esbmc	20 minutes	0
esbmc + Uautomizer + LLM	20 minutes	**25**

Thank you!

LLMs as invariant generators: example

Wu

```
int main() {
unsigned char n = __VERIFIER_nondet_uchar();
if (n == 0) {
return 0;
}
unsigned char v = 0;
unsigned int s = 0;
unsigned int i = 0;
while (i < n) {
v = __VERIFIER_nondet_uchar();
s += v;
++i;
}
assert(s <= 65025);
return 0;
```

Print loop invariants that help prove the assertion as C assertions.

Can perform different types of reasoning

To prove the assertion in the given C code, we can provide loop invariants that help ensure that the sum `s` remains less than or equal to 65025 throughout the execution of the loop. A loop invariant is a condition that holds true before and after each iteration of the loop. Here are the loop invariants that can help prove the assertion:

c Copy code

```c
int main() {
    unsigned char n = __VERIFIER_nondet_uchar();
    if (n == 0) {
        return 0;
    }
    unsigned char v = 0;
    unsigned int s = 0;
    unsigned int i = 0;

    // Loop Invariant 1: s is always non-negative
    assert(s >= 0);

    while (i < n) {
        v = __VERIFIER_nondet_uchar();

        // Loop Invariant 2: v is between 0 and 255 (inclusive)
        assert(v >= 0 && v <= 255);

        // Loop Invariant 3: s is less than or equal to 65025
        assert(s <= 65025);

        s += v;
        ++i;
    }

    // Loop Invariant 4: After the loop, s is still less than or equal to 65
    assert(s <= 65025);

    return 0;
}
```