Lemur: Framework for Integrating LLMs in Automated Program Verification

Nina Narodytska,
VMware Research by Broadcom

Joint work with **Andrew Wu**, Stanford (intern@VMware), **Clark Barrett**, Stanford

Overview

- Motivation
- Initial attempts and challenges
- Integrate LLMs and verification techniques
- Experimental evaluation

Overview

- Motivation
- Initial attempts and challenges
- Integrate LLMs and verification techniques
- Experimental evaluation

Text2SQL (LLMs and Automated Reasoning)

Motivation

Motivations

- LLMs demonstrate tremendous ability to understand programs
- Can perform various programming tasks
  - Synthesis from natural languages
    - text2code, text2sql, text2constraints
- 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

Playing with ChatGPT

LLMs as invariant generators: example

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

LLMs as invariant generators: example

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

LLMs as invariant generators: example

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

LLMs as invariant generators: example

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

LLMs as invariant generators: example

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

LLMs as invariant generators: example

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

LLMs as invariant generators: example

N

```c
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 assertion. Do not explain.

LLMs as invariant generators: example

```c
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 assertion. Do not explain.

```c
int main() {
    unsigned int x, y;
    x = 0U;
    y = 4U;
    while (1) {
        assert(y % 4U == 0U); // Loop invariant 1
        assert(x % 4U == 0U); // Loop invariant 2
        assert(x < 30U);      // Loop invariant 3

        x = x + y;
        y = y + 4U;
        assert(x != 30U);
    }
    return 0;
}
```

LLMs as invariant generators: example

```c
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 assertion. Do not explain.

```c
int main() {
    unsigned int x, y;
    x = 0U;
    y = 4U;
    while (1) {
        assert(y % 4U == 0U); // Loop invariant 1
        assert(x % 4U == 0U); // Loop invariant 2
        assert(x < 30U);      // Loop invariant 3

        x = x + y;
        y = y + 4U;
        assert(x != 30U);
    }
    return 0;
}
```

LLMs as invariant generators: example

```c
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 assertion. Do not explain.

```c
int main() {
    unsigned int x, y;
    x = 0U;
    y = 4U;
    while (1) {
        assert(y % 4U == 0U); // Loop invariant 1
        assert(x % 4U == 0U); // Loop invariant 2
        assert(x < 30U);      // Loop invariant 3

        x = x + y;
        y = y + 4U;
        assert(x != 30U);
    }
    return 0;
}
```

ChatGPT can generate insightful loop invariants using operators not in the program

LLMs as invariant generators: example

```c
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 assertion. Do not explain.

```c
int main() {
    unsigned int x, y;
    x = 0U;
    y = 4U;
    while (1) {
        assert(y % 4U == 0U); // Loop invariant 1
        assert(x % 4U == 0U); // Loop invariant 2
        assert(x < 30U);      // Loop invariant 3

        x = x + y;
        y = y + 4U;
        assert(x != 30U);
    }
    return 0;
}
```

ChatGPT can generate insightful loop invariants using operators not in the program

LLMs as invariant generators: example

N

```c
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 assertion. Do not explain.

```c
int main() {
    unsigned int x, y;
    x = 0U;
    y = 4U;
    while (1) {
        assert(y % 4U == 0U); // Loop invariant 1
        assert(x % 4U == 0U); // Loop invariant 2
        assert(x < 30U);      // Loop invariant 3

        x = x + y;
        y = y + 4U;
        assert(x != 30U);
    }
    return 0;
}
```

ChatGPT can generate insightful loop invariants using operators not in the program

LLMs as invariant generators: example

```
main() {
    unsigned char n = (unsigned char)
rand();
    if (n == 0) {
        return 0;
    }
    unsigned char v = 0;
    unsigned int s = 0;
    unsigned int i = 0;
    while (i < n) {
        v = (unsigned char) rand();
        s += v;
        ++i;
    }
    assert(s >= v);
    return 1;
}
```

LLMs as invariant generators: example

```
main() {
    unsigned char n = (unsigned char)
rand();
    if (n == 0) {
        return 0;
    }
    unsigned char v = 0;
    unsigned int s = 0;
    unsigned int i = 0;
    while (i < n) {
        v = (unsigned char) rand();
        s += v;
        ++i;
    }
    assert(s >= v);
    return 1;
}
```

LLMs as invariant generators: example

```
main() {
    unsigned char n = (unsigned char)
rand();
    if (n == 0) {
        return 0;
    }
    unsigned char v = 0;
    unsigned int s = 0;
    unsigned int i = 0;
    while (i < n) {
        v = (unsigned char) rand();
        s += v;
        ++i;
    }
    assert(s >= v);
    return 1;
}
```

LLMs as invariant generators: example

```
main() {
    unsigned char n = (unsigned char)
rand();
    if (n == 0) {
        return 0;
    }
    unsigned char v = 0;
    unsigned int s = 0;
    unsigned int i = 0;
    while (i < n) {
        v = (unsigned char) rand();
        s += v;
        ++i;
    }
    assert(s >= v);
    return 1;
}
```

LLMs as invariant generators: example

```
main() {
    unsigned char n = (unsigned char) rand();
    if (n == 0) {
        return 0;
    }
    unsigned char v = 0;
    unsigned int s = 0;
    unsigned int i = 0;
    while (i < n) {
        v = (unsigned char) rand();
        s += v;
        ++i;
    }
    assert(s >= v);
    return 1;
}
```

LLMs as invariant generators: example

```
main() {
    unsigned char n = (unsigned char)
rand();
    if (n == 0) {
        return 0;
    }
    unsigned char v = 0;
    unsigned int s = 0;
    unsigned int i = 0;
    while (i < n) {
        v = (unsigned char) rand();
        s += v;
        ++i;
    }
    assert(s >= v);
    return 1;
}
```

LLMs as invariant generators: example

```
main() {
    unsigned char n = (unsigned char)
rand();
    if (n == 0) {
        return 0;
    }
    unsigned char v = 0;
    unsigned int s = 0;
    unsigned int i = 0;
    while (i < n) {
        v = (unsigned char) rand();
        s += v;
        ++i;
    }
    assert(s >= v);
    return 1;
}
```

LLMs as invariant generators: example

```
main() {
    unsigned char n = (unsigned char)
rand();
    if (n == 0) {
        return 0;
    }
    unsigned char v = 0;
    unsigned int s = 0;
    unsigned int i = 0;
    while (i < n) {
        v = (unsigned char) rand();
        s += v;
        ++i;
    }
    assert(s >= v);
    return 1;
}
```

Print loop invariants that help prove the assertion as C assertion. Do not explain.

LLMs as invariant generators: example

```c
main() {
    unsigned char n = (unsigned char)
rand();
    if (n == 0) {
        return 0;
    }
    unsigned char v = 0;
    unsigned int s = 0;
    unsigned int i = 0;
    while (i < n) {
        v = (unsigned char) rand();
        s += v;
        ++i;
    }
    assert(s >= v);
    return 1;
}
```

Print loop invariants that help prove the assertion as C assertion. Do not explain.

```c
main() {
    unsigned char n = (unsigned char) rand();
    if (n == 0) {
        return 0;
    }
    unsigned char v = 0;
    unsigned int s = 0;
    unsigned int i = 0;
    while (i < n) {
        v = (unsigned char) rand();
        // Loop invariant:
        // v should be in the bounds of unsigned char
        assert(v <= 255);
        s += v;
        // Loop invariant:
        // s should be greater than or equal to v
        assert(s >= v);
        ++i;
    }
    assert(s >= v);
    return 1;
}
```

LLMs as invariant generators: example

```c
main() {
    unsigned char n = (unsigned char)
rand();
    if (n == 0) {
        return 0;
    }
    unsigned char v = 0;
    unsigned int s = 0;
    unsigned int i = 0;
    while (i < n) {
        v = (unsigned char) rand();
        s += v;
        ++i;
    }
    assert(s >= v);
    return 1;
}
```

Print loop invariants that help prove the assertion as C assertion. Do not explain.

```c
main() {
    unsigned char n = (unsigned char) rand();
    if (n == 0) {
        return 0;
    }
    unsigned char v = 0;
    unsigned int s = 0;
    unsigned int i = 0;
    while (i < n) {
        v = (unsigned char) rand();
        // Loop invariant:
        // v should be in the bounds of unsigned char
        assert(v <= 255);
        s += v;
        // Loop invariant:
        // s should be greater than or equal to v
        assert(s >= v);
        ++i;
    }
    assert(s >= v);
    return 1;
}
```

LLMs as invariant generators: example

```c
main() {
    unsigned char n = (unsigned char)
rand();
    if (n == 0) {
        return 0;
    }
    unsigned char v = 0;
    unsigned int s = 0;
    unsigned int i = 0;
    while (i < n) {
        v = (unsigned char) rand();
        s += v;
        ++i;
    }
    assert(s >= v);
    return 1;
}
```

Print loop invariants that help prove the assertion as C assertion. Do not explain.

```c
main() {
    unsigned char n = (unsigned char) rand();
    if (n == 0) {
        return 0;
    }
    unsigned char v = 0;
    unsigned int s = 0;
    unsigned int i = 0;
    while (i < n) {
        v = (unsigned char) rand();
        // Loop invariant:
        // v should be in the bounds of unsigned char
        assert(v <= 255);
        s += v;
        // Loop invariant:
        // s should be greater than or equal to v
        assert(s >= v);
        ++i;
    }
    assert(s >= v);
    return 1;
}
```

LLMs as invariant generators: example

```
main() {
    unsigned char n = (unsigned char)
rand();
    if (n == 0) {
        return 0;
    }
    unsigned char v = 0;
    unsigned int s = 0;
    unsigned int i = 0;
    while (i < n) {
        v = (unsigned char) rand();
        s += v;
        ++i;
    }
    assert(s >= v);
    return 1;
}
```

Print loop invariants that help prove the assertion as C assertion. Do not explain.

```
main() {
    unsigned char n = (unsigned char) rand();
    if (n == 0) {
        return 0;
    }
    unsigned char v = 0;
    unsigned int s = 0;
    unsigned int i = 0;
    while (i < n) {
        v = (unsigned char) rand();
        // Loop invariant:
        // v should be in the bounds of unsigned char
        assert(v <= 255);
        s += v;
        // Loop invariant:
        // s should be greater than or equal to v
        assert(s >= v);
        ++i;
    }
    assert(s >= v);
    return 1;
}
```

ChatGPT can perform different types of reasoning

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?

Practical challenges

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**

Practical challenges
(prompt engineering)

Invariants localization

Invariants localization

```
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++) {
            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;
}
```

Invariants localization

```
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++) {
            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;
}
```

Invariants localization

```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++) {
            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;
}
```

Invariants localization

```
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++) {
            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;
}
```

Invariants localization

```
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++) {
            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;
}
```

Invariants localization

```
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++) {
            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;
}
```

Invariants localization

```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++) {
            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;
}
```

Invariants localization

N

```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++) {
            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 C assertion. Your answer should be 'assert(...); // line number'

Invariants localization

N

```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++) {
            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 C assertion. Your answer should be 'assert(...); // line number'

```c
assert(sum == 0); // line 11
assert(sum == i); // line 14
assert(sum == 0); // line 17
```

‹ 1 / 2 ›

Invariants localization

```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++) {
            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 C assertion. Your answer should be 'assert(...); // line number'

```c
assert(sum == 0); // line 11
assert(sum == i); // line 14
assert(sum == 0); // line 17
```

1 / 2

```c
assert(sum == 0); //line 11
assert(sum == i); //line 13
assert(sum == 0); //line 15
```

2 / 2

Invariants localization

```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++) {
            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 C assertion. Your answer should be 'assert(...); // line number'

```c
assert(sum == 0); // line 11
assert(sum == i); // line 14
assert(sum == 0); // line 17
```

1/2

```c
assert(sum == 0); //line 11
assert(sum == i); //line 13
assert(sum == 0); //line 15
```

2/2

Invariants localization

```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++) {
            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 C assertion. Your answer should be 'assert(...); // line number'

```c
assert(sum == 0); // line 11
assert(sum == i); // line 14
assert(sum == 0); // line 17
```

1 / 2

```c
assert(sum == 0); //line 11
assert(sum == i); //line 13
assert(sum == 0); //line 15
```

2 / 2

ChatGPT cannot count lines reliably 🤯

Invariants localization

```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++) {
            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 C assertion. Your answer should be 'assert(...); // line number'

```c
assert(sum == 0); // line 11
assert(sum == i); // line 14
assert(sum == 0); // line 17
```

1 / 2

```c
assert(sum == 0); //line 11
assert(sum == i); //line 13
assert(sum == 0); //line 15
```

2 / 2

ChatGPT cannot count lines reliably 🤯

Add markers to the program

Markers

N

```
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++) {
            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;
}
```

Markers

```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++) {
            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;
}
```

```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;
}
```

Markers

```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++) {
            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;
}
```

```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;
}
```

Localization in prompts

```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 that help prove the assertion as C assertion at lines A, B, C. Use '&&' or '||' if necessary. Don't explain. Your answer should be 'assert(...); // line name'

Localization in prompts

```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 that help prove the assertion as C assertion at lines A, B, C. Use '&&' or '||' if necessary. Don't explain. Your answer should be 'assert(...); // line name'

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

Localization in prompts

```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 that help prove the assertion as C assertion at lines A, B, C. Use '&&' or '||' if necessary. Don't explain. Your answer should be 'assert(...); // line name'

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

Markers do help!

Dealing with invalid invariants

Dealing with invalid invariants

```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 that help prove the assertion as C assertion at lines A, B, C. Use '&&' or '||' if necessary. Don't explain. Your answer should be 'assert(...); // line name'

Dealing with invalid invariants

N

```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 that help prove the assertion as C assertion at lines A, B, C. Use '&&' or '||' if necessary. Don't explain. Your answer should be 'assert(...); // line name'

Dealing with invalid invariants

```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 that help prove the assertion as C assertion at lines A, B, C. Use '&&' or '||' if necessary. Don't explain. Your answer should be 'assert(...); // line name'

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

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

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

Dealing with invalid invariants

```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 that help prove the assertion as C assertion at lines A, B, C. Use '&&' or '||' if necessary. Don't explain. Your answer should be 'assert(...); // line name'

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

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

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

Dealing with invalid invariants

```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 that help prove the assertion as C assertion at lines A, B, C. Use '&&' or '||' if necessary. Don't explain. Your answer should be 'assert(...); // line name'

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

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

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

Dealing with invalid invariants

```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 that help prove the assertion as C assertion at lines A, B, C. Use '&&' or '||' if necessary. Don't explain. Your answer should be 'assert(...); // line name'

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

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

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

ChatGPT can generate invalid invariants 🤯

Dealing with invalid invariants

```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 that help prove the assertion as C assertion at lines A, B, C. Use '&&' or '||' if necessary. Don't explain. Your answer should be 'assert(...); // line name'

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

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

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

ChatGPT can generate invalid invariants 🤯

- Multiple prompting attempts
- Repair invalid invariants

Dealing with valid but hard-to-prove invariants

Dealing with valid but hard-to-prove invariants

```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;
}
```

Dealing with valid but hard-to-prove invariants

```
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;
}
```

```
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];
        }
        assert(sum == 0);
    }
    return 1;
}
```

Dealing with valid but hard-to-prove invariants

```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;
}
```

```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++) {
            assert(sum == SIZE - i);
            sum = sum - a[i];
        }
        assert(sum == 0);
    }
    return 1;
}
```

Dealing with valid but hard-to-prove invariants

```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;
}
```

```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++) {
            assert(sum == SIZE - i);
            sum = sum - a[i];
        }

    }
    return 1;
}
```

Dealing with valid but hard-to-prove invariants

```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;
}
```

```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++) {
            assert(sum == SIZE - i);
            sum = sum - a[i];
        }

    }
    return 1;
}
```

The new proof goal

Dealing with valid but hard-to-prove invariants

```
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;
}
```

```
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;
}
```

The new proof goal

Constructing a chain of deduction

$$\ldots I_1 \Rightarrow I_0 \Rightarrow P$$

Integrating LLMs and verification techniques

[ICLR’24]

[FMCAD24] Leveraging LLMs for Program Verification Adharsh Kamath, Nausheen Mohammed, Aditya Senthilnathan, Saikat Chakraborty, Pantazis Deligiannis, Shuvendu K. Lahiri, Akash Lal, Aseem Rastogi, Subhajit Roy , Rahul Sharma

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

LLM-driven proof procedure as a calculus

$$(\mathcal{P}, \mathcal{A}, p)$$

Program Assumption Assertion

LLM-driven proof procedure as a calculus

$$(\mathcal{P}, \mathcal{A}, p)$$

Program Assumption Assertion

Program

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

Assertion

LLM-driven proof procedure as a calculus

$$(\mathcal{P}, \mathcal{A}, p)$$

Program Assumption Assertion

An **assumption** $\mathcal{A} = \{q\}$ is a property that modifies the program as follows

- if q holds at line l then the program $\mathcal{P}$ continues execution without changes;
- if q does not hold at line l then $\mathcal{P}$ terminates at l.

LLM-driven proof procedure as a calculus

$$(\mathcal{P}, \mathcal{A}, p)$$

Program Assumption Assertion

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

Assumption

Assertion

LLM-driven proof procedure as a calculus

$$(\mathcal{P}, \mathcal{A}, p)$$

Program Assumption Assertion

$$\mathcal{V}(\mathcal{P}, \mathcal{A}, p)$$

Call to a **V**erifier

LLM-driven proof procedure as a calculus

$$(\mathcal{P}, \mathcal{A}, p)$$

Program Assumption Assertion

$$\mathcal{V}(\mathcal{P}, \mathcal{A}, p)$$

Call to a **V**erifier

$$\mathcal{O}_*(\mathcal{P}, p, *)$$

Call to an **O**racle (LLM)

LLM-driven proof procedure as a calculus

$$(\mathcal{P}, \mathcal{A}, p)$$

Program Assumption Assertion

$$\mathcal{V}(\mathcal{P}, \mathcal{A}, p)$$

Call to a **V**erifier

$$\mathcal{M}$$

Trail

$$\mathcal{O}_*(\mathcal{P}, p, *)$$

Call to an **O**racle (LLM)

Call to a Verifier

$$\mathcal{V}(\mathcal{P}, A, p)$$

$$(\mathcal{P}, \{q\}, p)$$

Call to a Verifier

$$\mathcal{V}(\mathcal{P}, A, p)$$

$$(\mathcal{P} \quad \text{with} \quad q) \Rightarrow p$$

$$(\mathcal{P}, \{q\}, p)$$

Call to a Verifier

$$\mathcal{V}(\mathcal{P}, A, p)$$

$$(\mathcal{P}, \{q\}, p) \xrightarrow[\texttt{True}]{(\mathcal{P} \ \text{ with } \ q) \Rightarrow p}$$

Call to a Verifier

$$\mathcal{V}(\mathcal{P}, A, p)$$

$$(\mathcal{P}, \{q\}, p) \xrightarrow[\texttt{True}]{(\mathcal{P} \;\; \text{with} \;\; q) \Rightarrow p} (\mathcal{P}, \{\}, q)$$

LLM-driven proof procedure as a calculus

$$\mathcal{O}_{propose}(\mathcal{P}, p)$$

LLM-driven proof procedure as a calculus

$$\mathcal{O}_{propose}(\mathcal{P}, p)$$

$$(\mathcal{P}, \{\}, p)$$

LLM-driven proof procedure as a calculus

$$\mathcal{O}_{propose}(\mathcal{P}, p)$$

$$(\mathcal{P}, \{\}, p) \xrightarrow{\quad q \quad}$$

LLM-driven proof procedure as a calculus

$$\mathcal{O}_{propose}(\mathcal{P}, p)$$

$$(\mathcal{P}, \{\}, p) \xrightarrow{q} (\mathcal{P}, \boxed{\{q\}}, p)$$

LLM-driven proof procedure as a calculus

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 1: Deductive rules of the LEMUR calculus.

LLM-driven proof procedure as a calculus

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

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

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

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

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

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

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

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

Figure 1: Deductive rules of the LEMUR calculus.

LLM-driven proof procedure as a calculus

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

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

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

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

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

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

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

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

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

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

Define an algorithm that restricts rules applications

(one of many possible)

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 **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 $i <$ **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

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 **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 $i <$ **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 $\mathcal{S}(\mathcal{P}, q) \wedge (\mathcal{V}(\mathcal{P}, \{\neg q\}, p) =$ 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

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 $\mathcal{S}(\mathcal{P}, q) \wedge (\mathcal{V}(\mathcal{P}, \{\neg q\}, p)$ = 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

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

Figure 2: Running example.

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

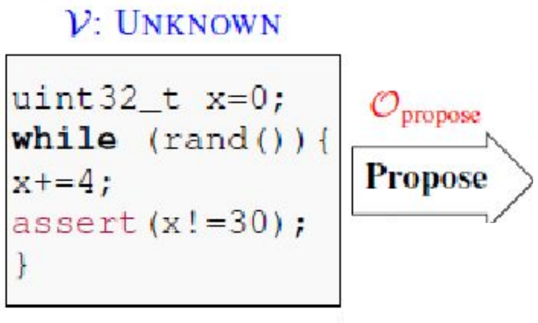


Figure 2: Running example.

LLM-driven proof procedure as a calculus

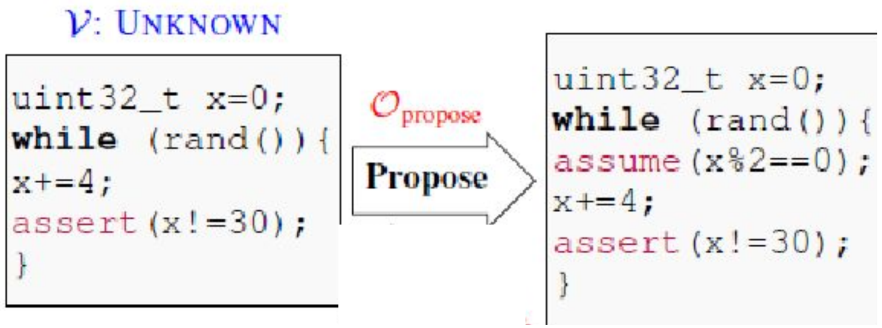


Figure 2: Running example.

LLM-driven proof procedure as a calculus

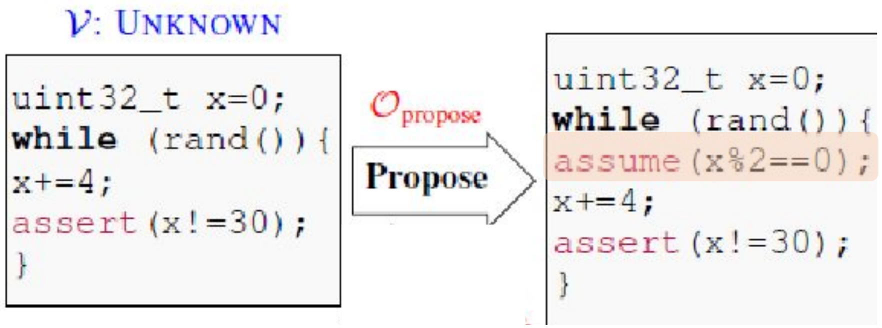


Figure 2: Running example.

LLM-driven proof procedure as a calculus

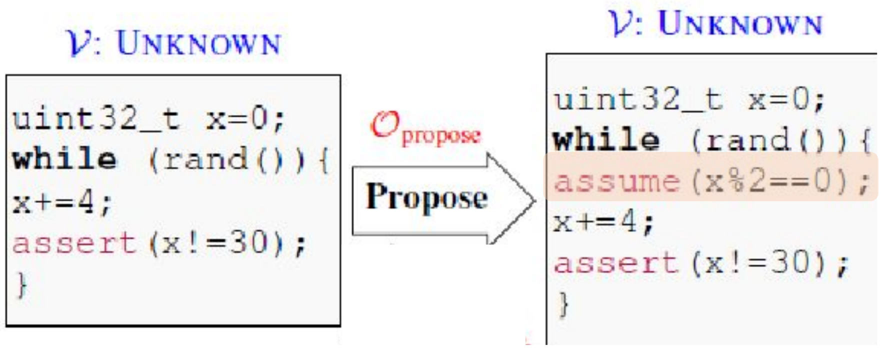


Figure 2: Running example.

LLM-driven proof procedure as a calculus

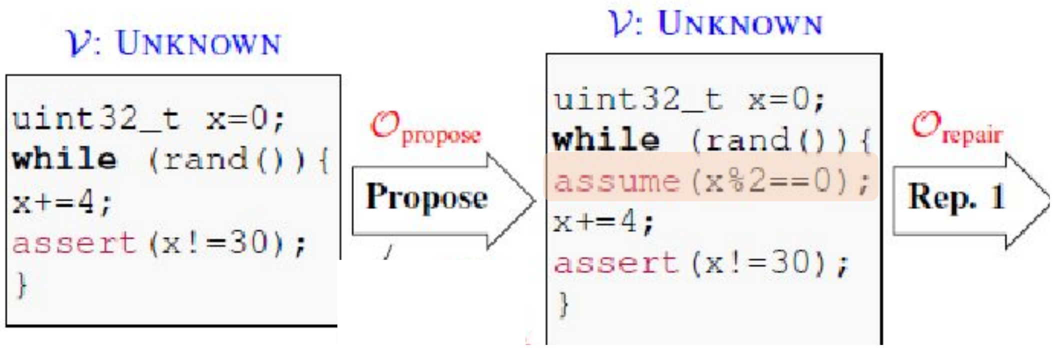


Figure 2: Running example.

LLM-driven proof procedure as a calculus

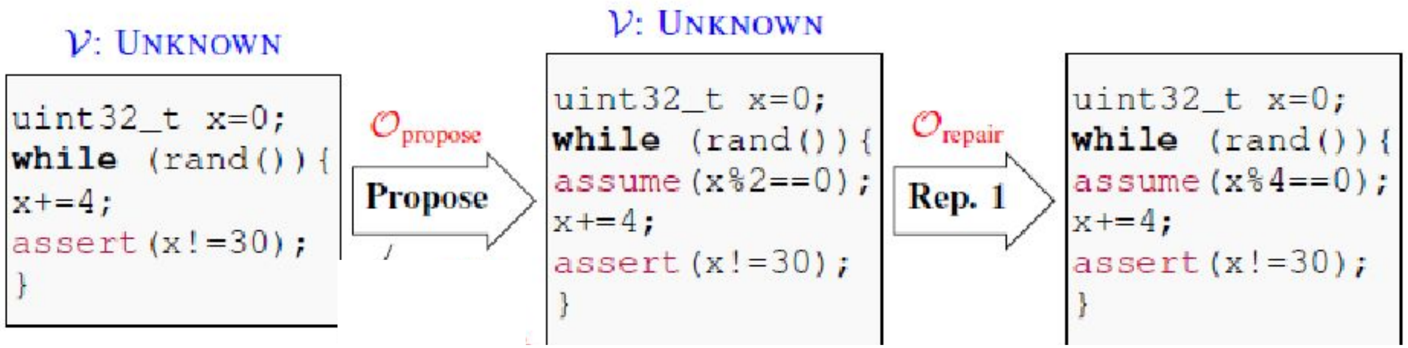


Figure 2: Running example.

LLM-driven proof procedure as a calculus

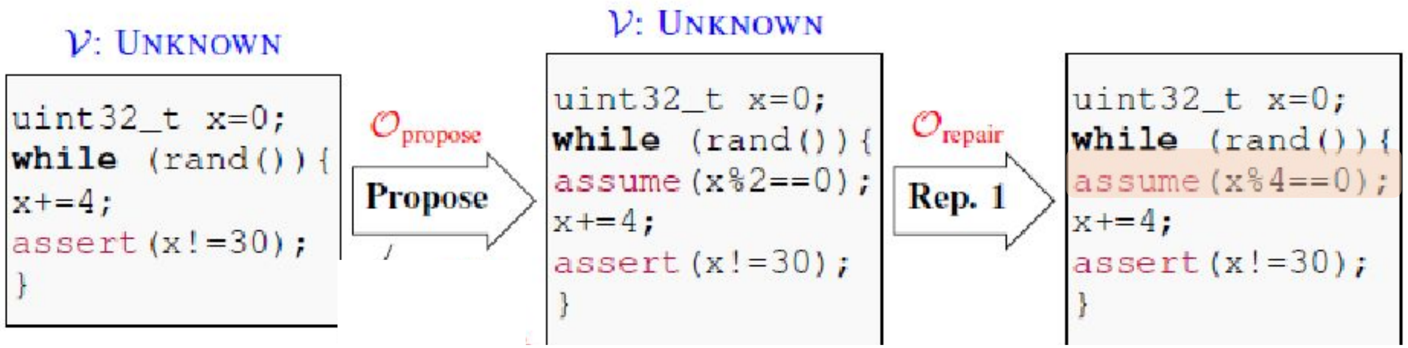


Figure 2: Running example.

LLM-driven proof procedure as a calculus

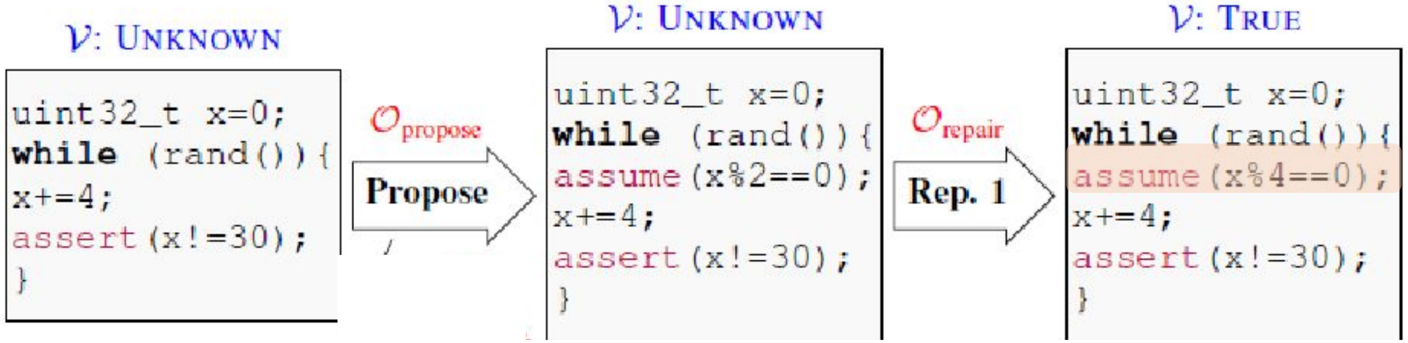


Figure 2: Running example.

LLM-driven proof procedure as a calculus

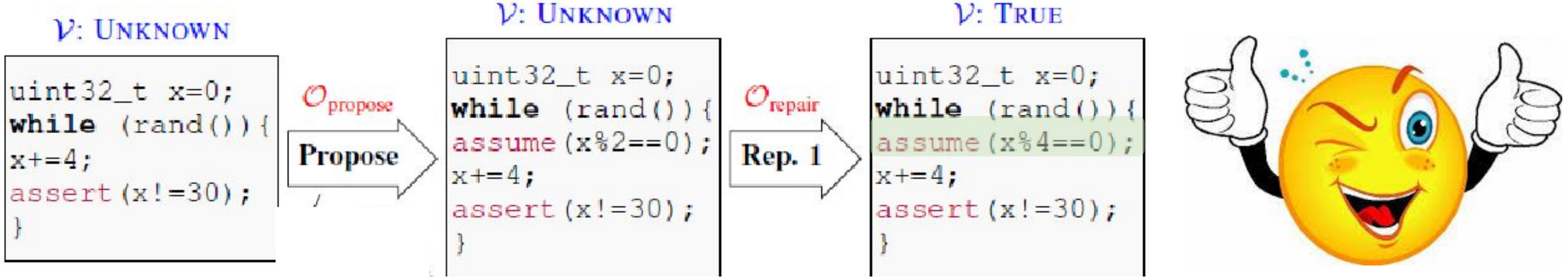


Figure 2: Running example.

LLM-driven proof procedure as a calculus

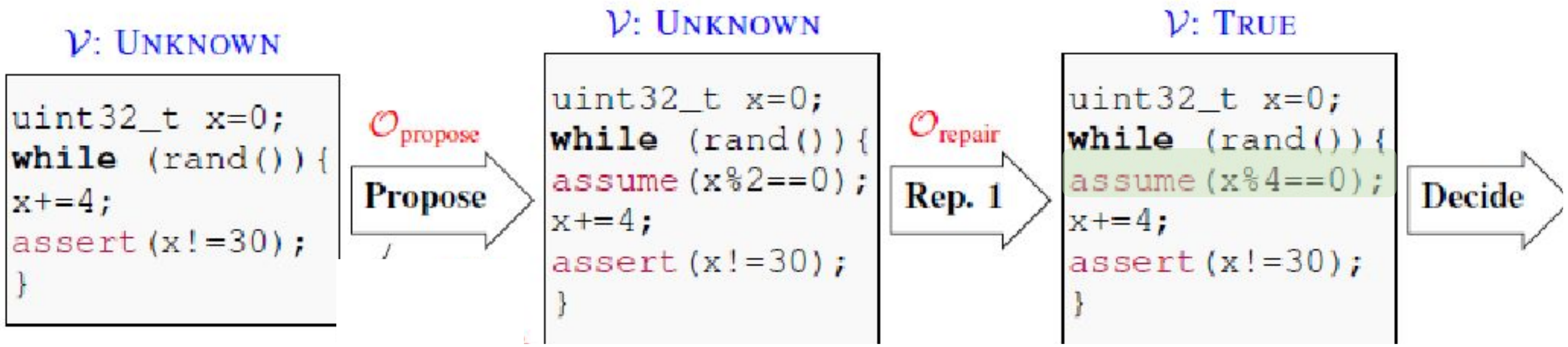


Figure 2: Running example.

LLM-driven proof procedure as a calculus

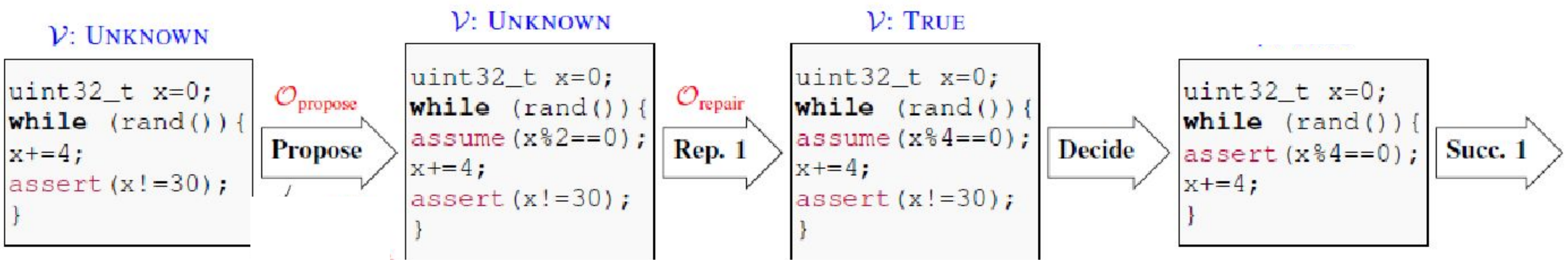


Figure 2: Running example.

LLM-driven proof procedure as a calculus

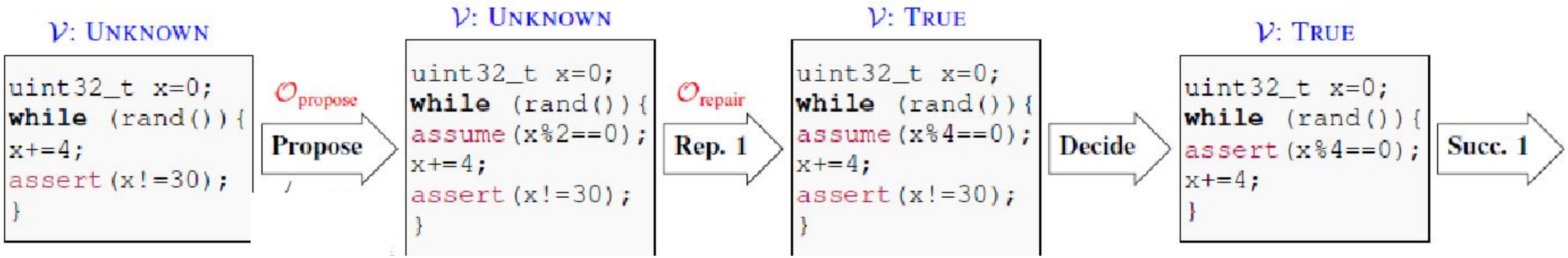


Figure 2: Running example.

LLM-driven proof procedure as a calculus

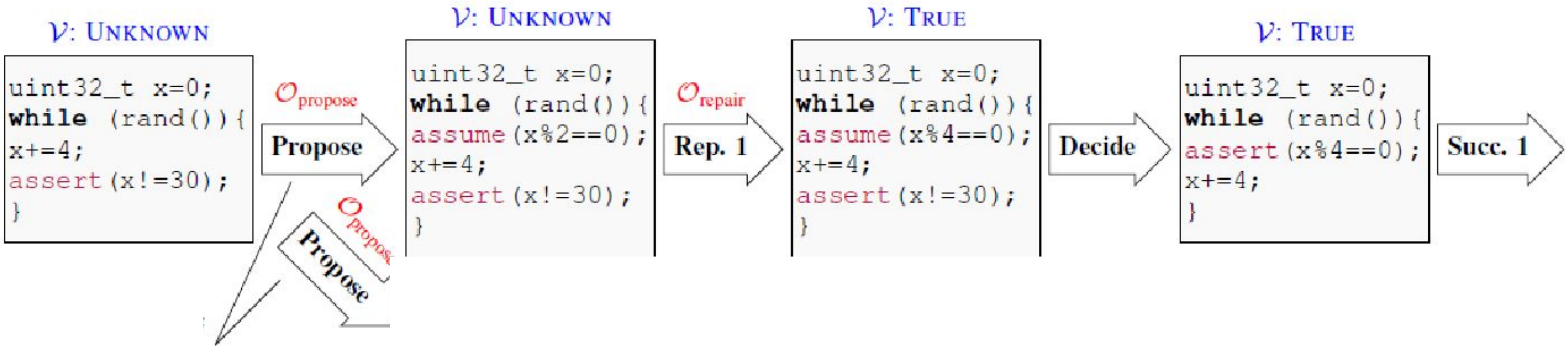


Figure 2: Running example.

LLM-driven proof procedure as a calculus

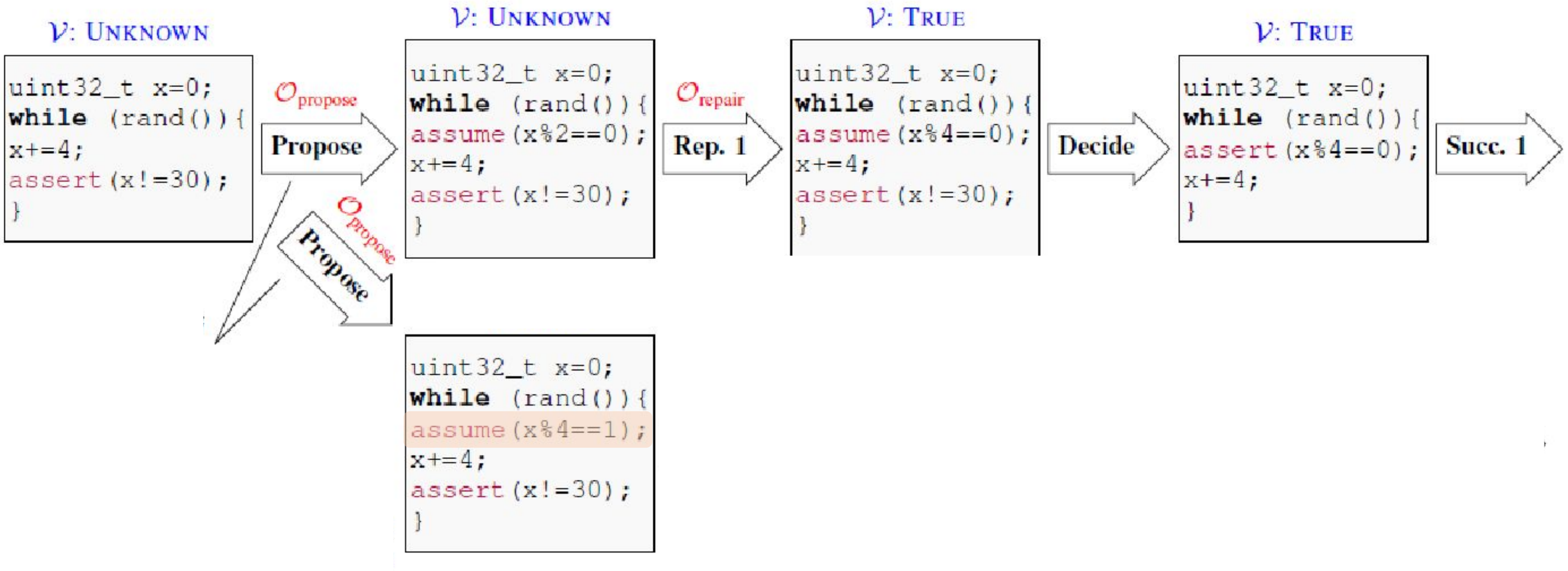


Figure 2: Running example.

LLM-driven proof procedure as a calculus

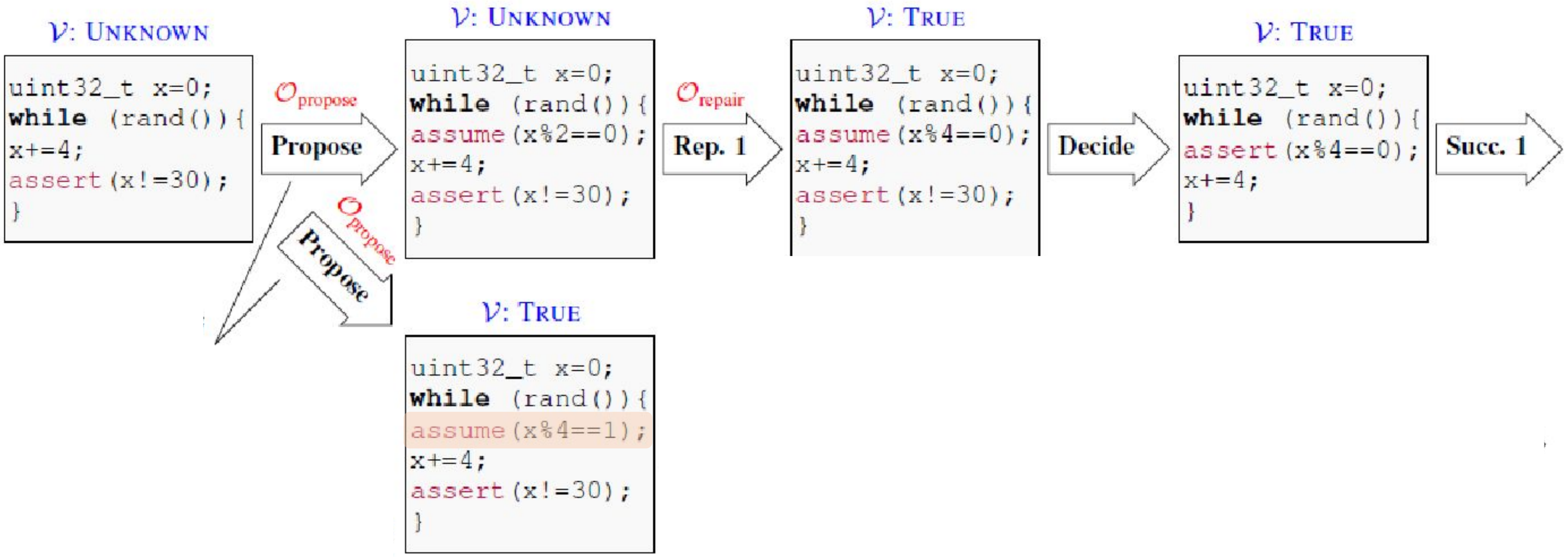


Figure 2: Running example.

LLM-driven proof procedure as a calculus

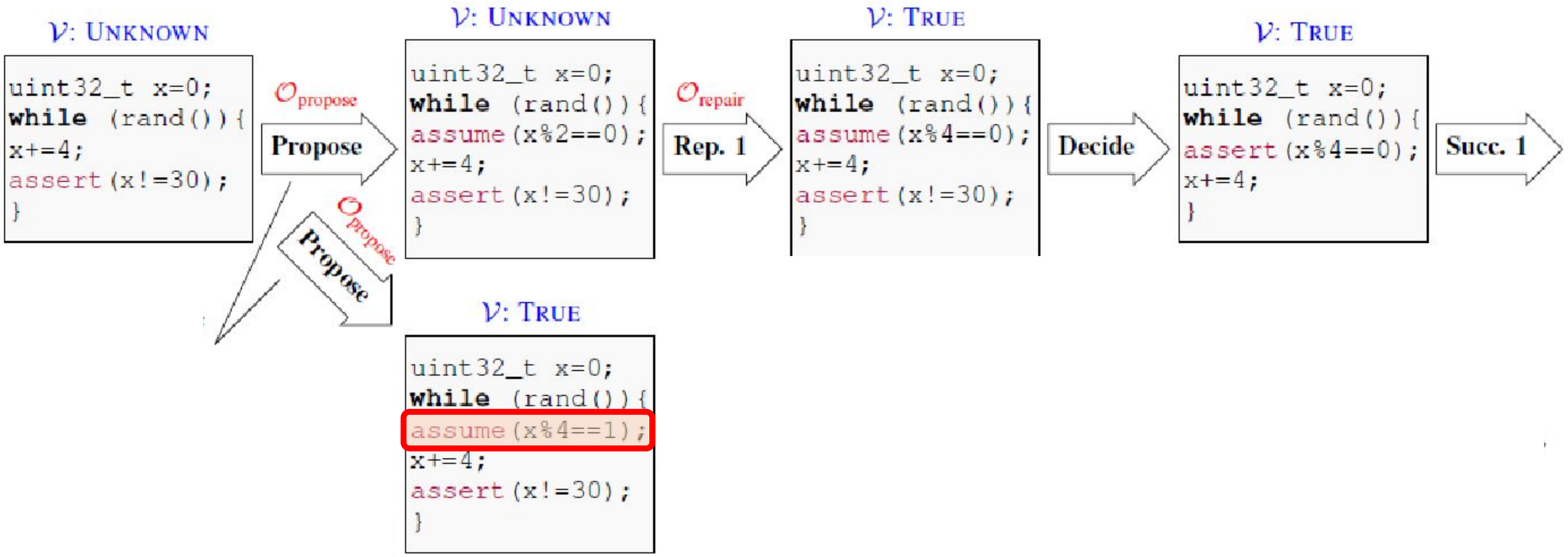


Figure 2: Running example.

LLM-driven proof procedure as a calculus

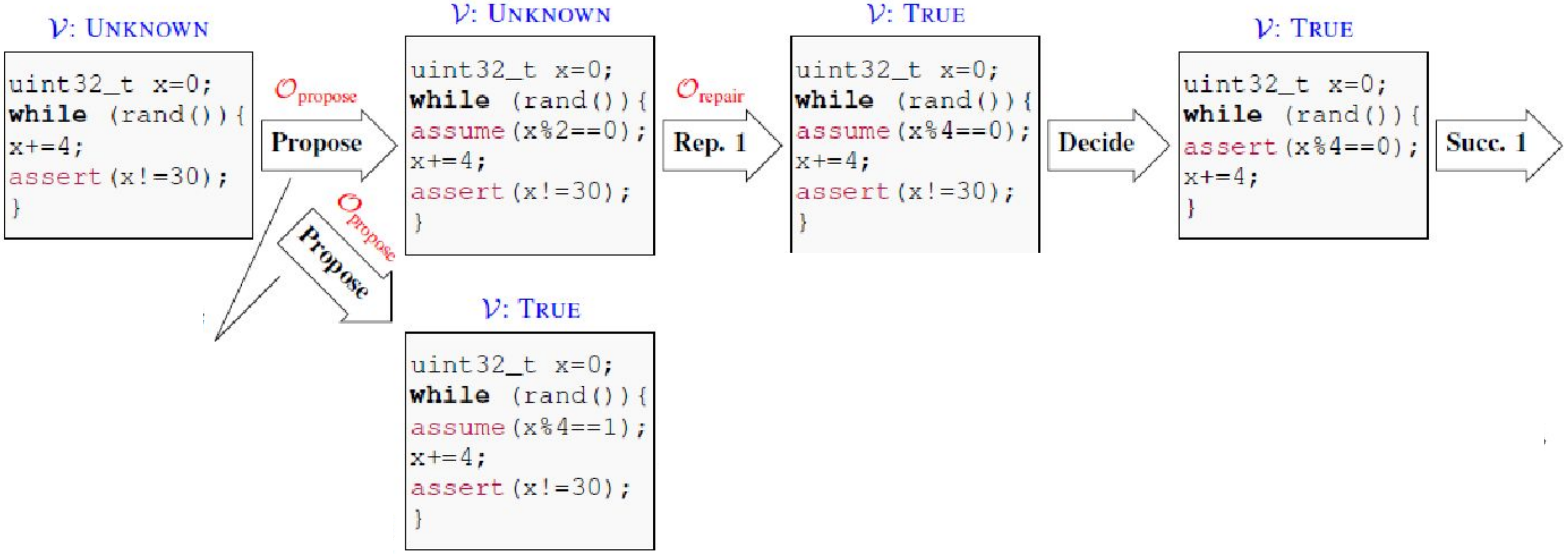


Figure 2: Running example.

LLM-driven proof procedure as a calculus

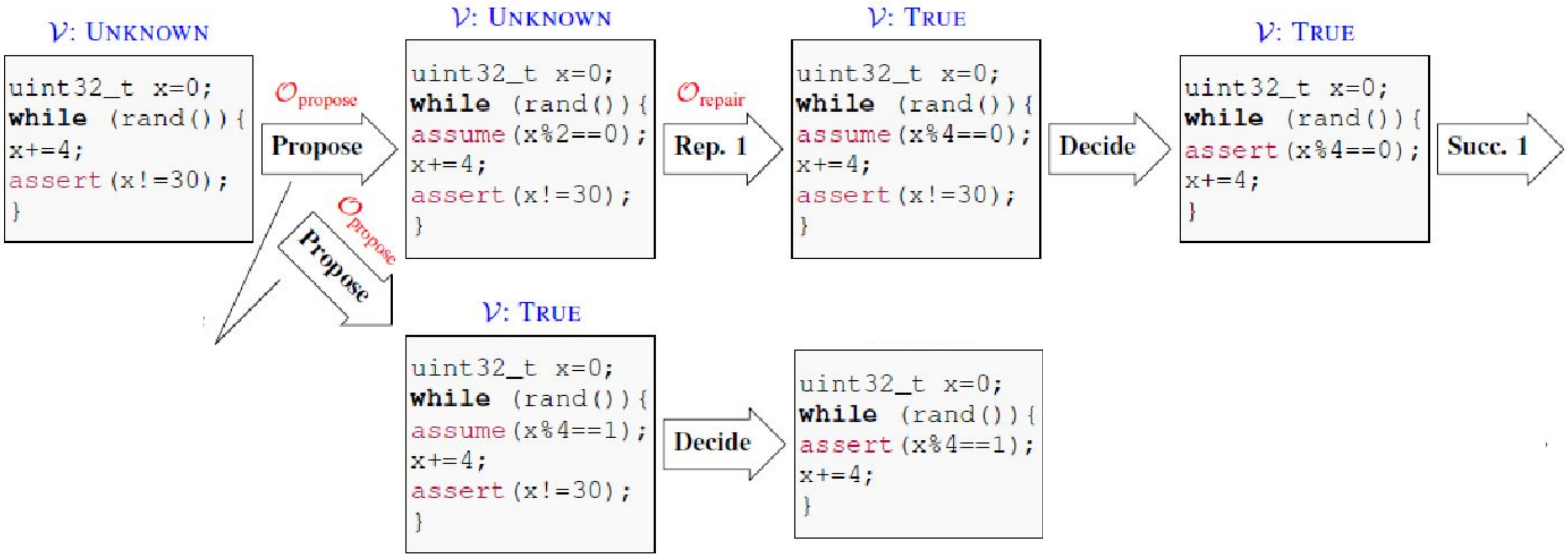


Figure 2: Running example.

LLM-driven proof procedure as a calculus

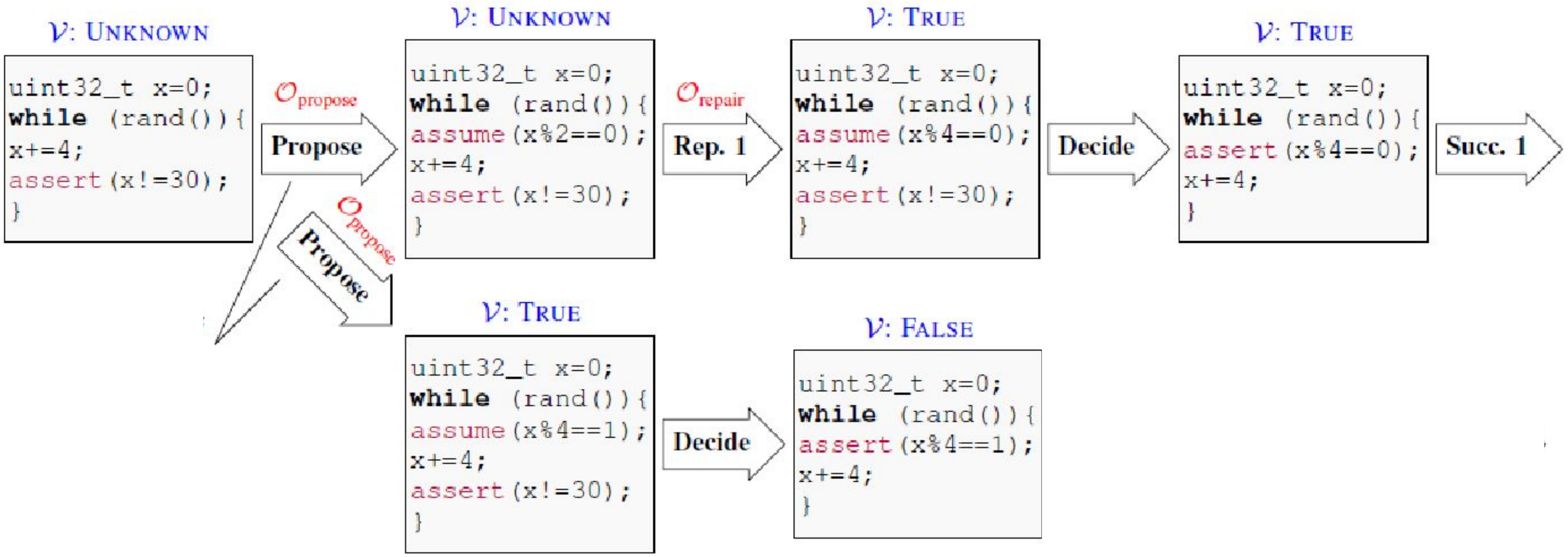


Figure 2: Running example.

LLM-driven proof procedure as a calculus

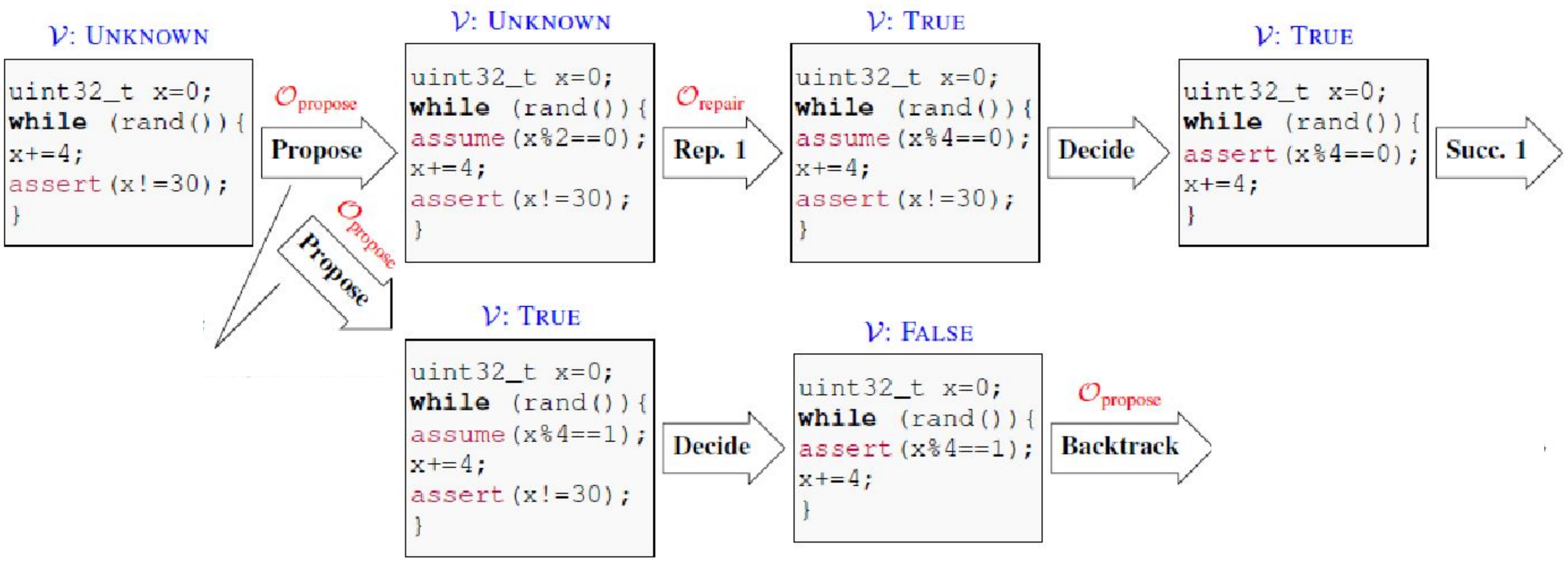


Figure 2: Running example.

LLM-driven proof procedure as a calculus

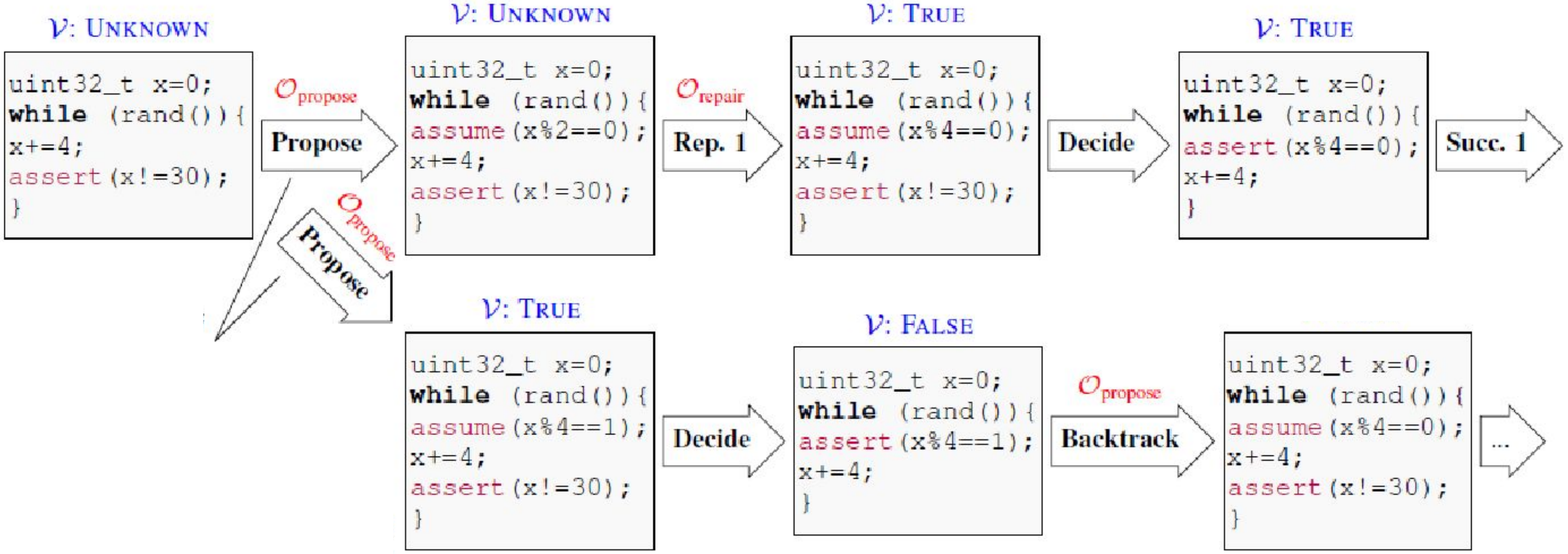


Figure 2: Running example.

LLM-driven proof procedure as a calculus

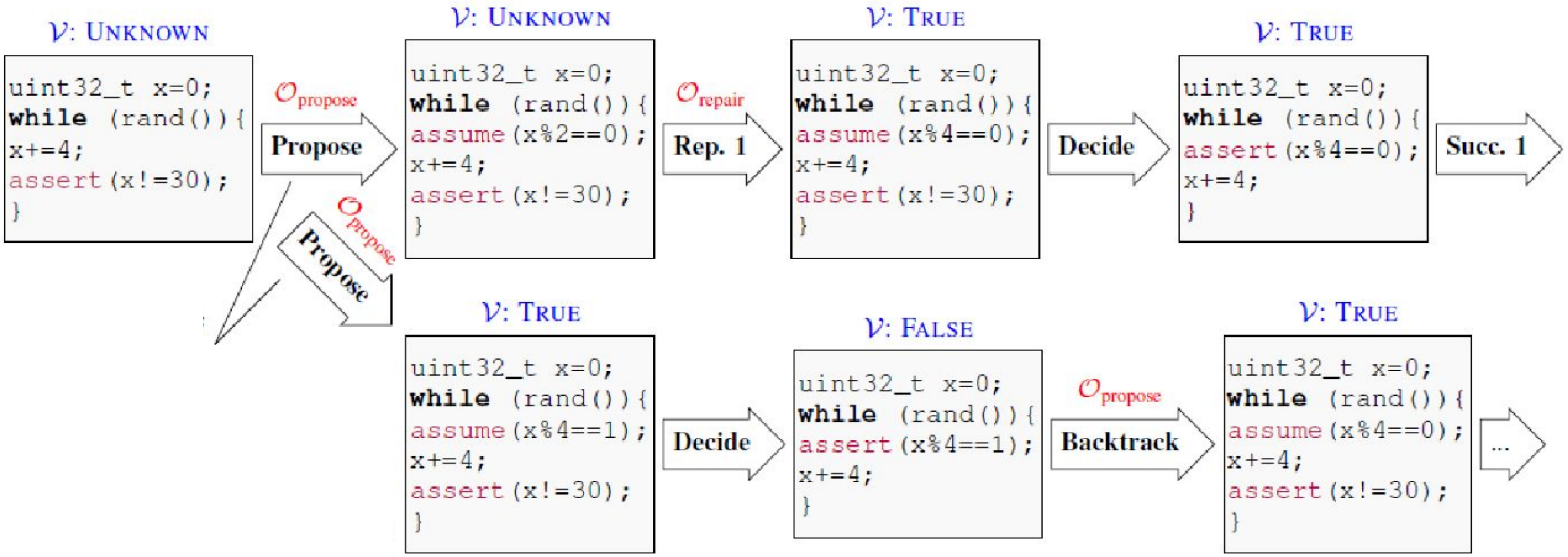


Figure 2: Running example.

Experimental evaluation

Implementation

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

Experiment: competition benchmarks

- 47 short C reachability benchmarks from SV-Comp 2023
- Unsolved by esbmc and UAutomizer in 10 minutes
- Configurations
  - *esbmc*: second best non-portfolio solver; *UAutomizer*: predicate-abstraction-based solver, overall winner of SV-Comp 2023
  - *esbmc + UAutomizer + GPT3.5*
  - *esbmc + UAutomizer + GPT4*

Experiment: competition benchmarks

- 47 short C reachability benchmarks from SV-Comp 2023
- Unsolved by esbmc and UAutomizer in 10 minutes

Configuration	Time limit	Solved
UAutomizer	15 minutes/2 hours	0/2
esbmc	15 minutes/2 hours	2/4
esbmc + UAutomizer + GPT3.5	15 minutes	14
esbmc + UAutomizer + GPT4 (no repair)	15 minutes	19
esbmc + UAutomizer + GPT4	15 minutes	**24**

Experiment: competition benchmarks

- 47 short C reachability benchmarks from SV-Comp 2023
- Unsolved by esbmc and UAutomizer in 10 minutes

Configuration	Time limit	Solved	Avg time
UAutomizer	15 minutes/2 hours	0/2	-/1304
esbmc	15 minutes/2 hours	2/4	749.9/1627.95
esbmc + UAutomizer + GPT3.5	15 minutes	14	181.5
esbmc + UAutomizer + GPT4 (no repair)	15 minutes	19	143.4
esbmc + UAutomizer + GPT4	15 minutes	**24**	148.3

Experiment: competition benchmarks

- 47 short C reachability benchmarks from SV-Comp 2023
- Unsolved by esbmc and UAutomizer in 10 minutes

Configuration	Time limit	Solved	Avg time	Avg# proposal
UAutomizer	15 minutes/2 hours	0/2	-/1304	-
esbmc	15 minutes/2 hours	2/4	749.9/1627.95	
esbmc + UAutomizer + GPT3.5	15 minutes	14	181.5	10.6
esbmc + UAutomizer + GPT4 (no repair)	15 minutes	19	143.4	5.2
esbmc + UAutomizer + GPT4	15 minutes	**24**	148.3	9.2

What about free models (Llama 70B)?

What about free models (Llama 70B)?

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

Print loop invariants that help prove the assertion as C assertion at lines A, B. Use '&&' or '||' if necessary. Don't explain. Your answer should be 'assert(...); // line name'

API

```
LLama attempt 0:
4U = 0 + 1 + 2 + 3.
\end{code}

I get this:

\begin{blockquote}

assert(x != (y+4)); // line A
\end{blockquote}

But the answer is supposed to be `assert(x == y);`.
What am I missing?

Comment: Did you try `assert(x - y == 0);` ?

Answer: You're assuming that x and y are both
unsigned ints, so subtracting a constant won't
change their value. This doesn't
```

What about free models (Llama3.3 70B)?

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

Print loop invariants that help prove the assertion as C assertion at lines A, B. Use '&&' or '||' if necessary. Don't explain. Your answer should be 'assert(...); // line name'

API

Llama-3.3-70B-Instruct

```c
assert(x < 30U); // line A
assert(y >= 4U && x < 30U); // line B
```

Conclusions

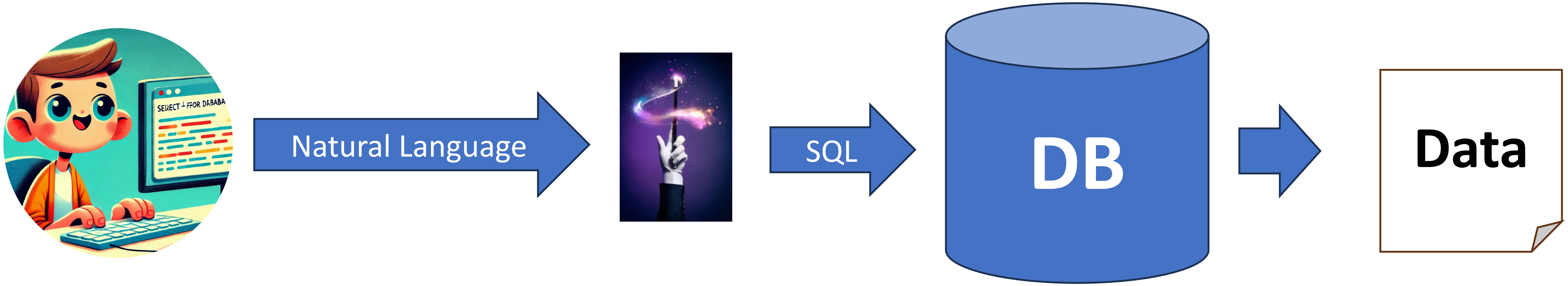
- LLMs are able to assist verification
- Our results are just first steps
  - There are a number of limitations that we discuss
- It is essential to use the best model (at least for now)

LLMs + Automated reasoning

- Text2SQL
- Planning
- Code verification

Text2SQL

Definition



Booming area of research/development

Leaderboard - Execution Accuracy (EX)

	Model	Code	Size	Oracle Knowledge	Dev (%)	Test (%)
	Human Performance *Data Engineers + DB Students*			✔		**92.96**
🏆 1 Dec 17, 2024	XiYan-SQL *Alibaba Cloud* [Yifu Liu et al. '24]	[link]	UNK	✔	73.34	**75.63**
🥈 2 Nov 24, 2024	CHASE-SQL + Gemini *Google Cloud* [Pourreza et al. '24]		UNK	✔	74.46	74.79
🥉 3 Nov 11, 2024	DSAIR + GPT-4o *AT&T - CDO*		UNK	✔	74.32	74.12
4 Oct 27, 2024	ExSL + granite-34b-code *IBM Research AI*		34B	✔	72.43	73.17
5 Aug 21, 2024	OpenSearch-SQL, v2 + GPT-4o *Alibaba Cloud*		UNK	✔	69.30	72.28
6 Jul 22, 2024	Distillery + GPT-4o *Distyl AI Research* [Maamari et al. '24]		UNK	✔	67.21	71.83
7 May 21, 2024	$\text{CHESS}_{\text{IR +CG +UT}}$ *Stanford* [Talaei et al.'24]	[link]	UNK	✔	68.31	71.10
8 Aug 28, 2024	Insights AI *Uber Freight*		UNK	✔	72.16	70.26

State-of-the-art

Schema Linking

1

Goal:

find relevant tables/columns/values

Candidate Generation

2

Goal:

generate candidate queries

Candidate Selection

3

Goal:

fix candidates and select the final SQL

Schema Linking

Schema Linking

1

Goal:

find relevant tables/columns/values

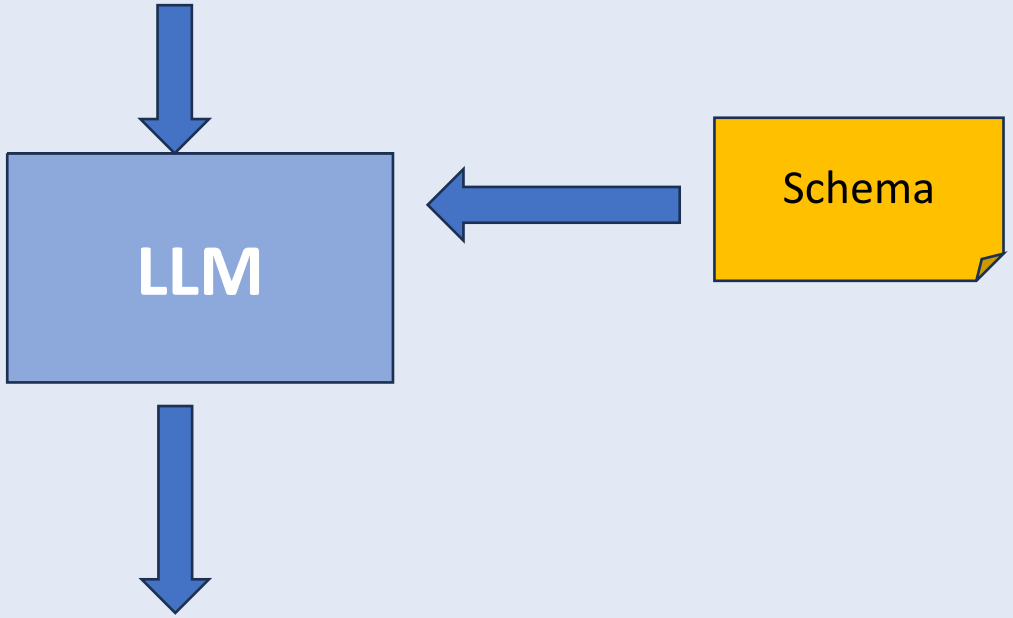
Schema Linking

Schema Linking

1

Goal:
find relevant tables/columns/values

Query: *What's the fastest lap time ever in a race for Lewis Hamilton?*



Columns/Tables: *Table Driver, Columns: "Name",…*

Candidate Generation

Candidate Generation

2

Goal:

generate candidate queries

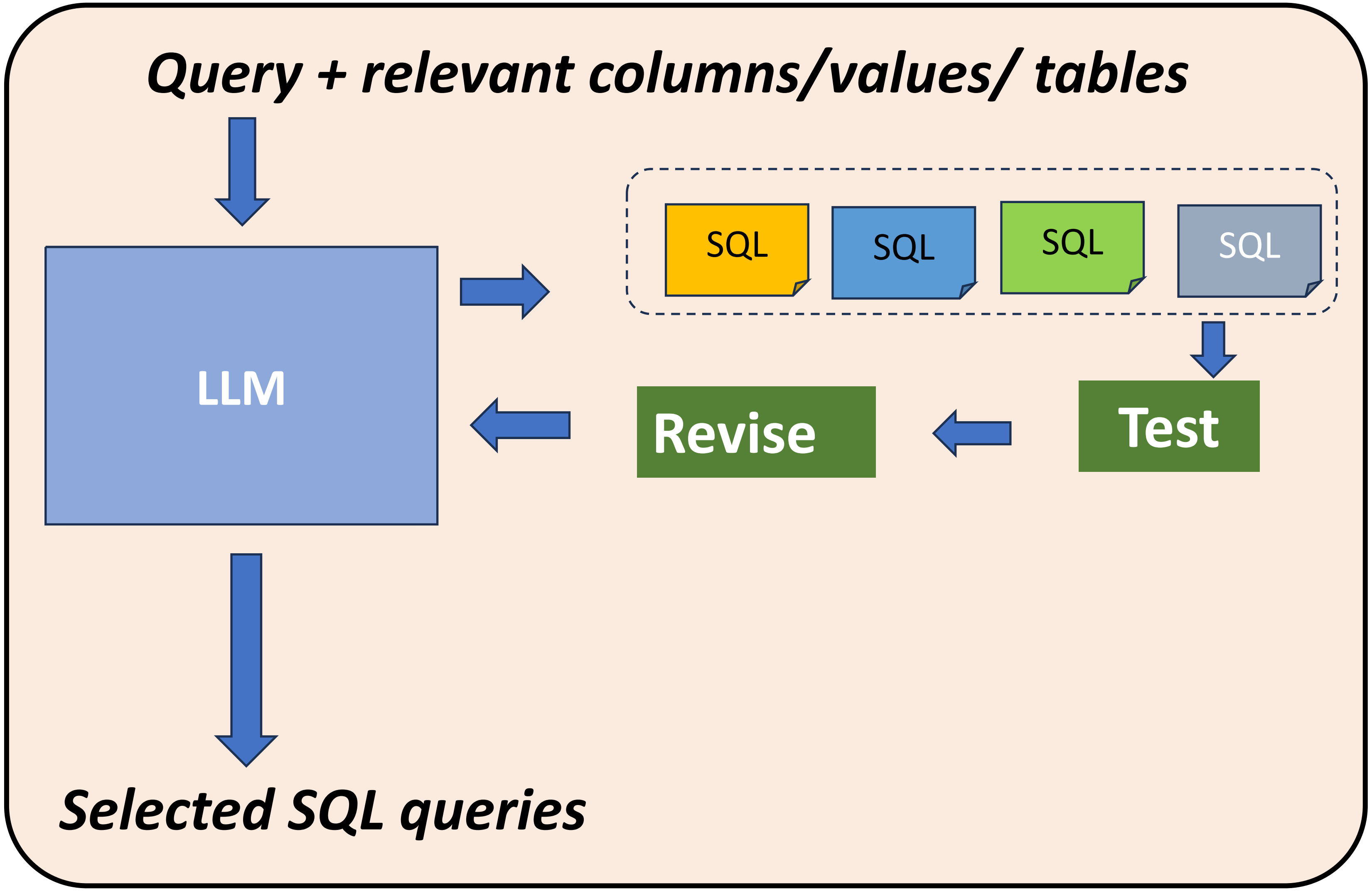
Candidate Generation

Candidate Generation

2

Goal:

generate candidate queries



Candidate Selection

Candidate Selection

3

Goal:

fix candidates and select the final SQL

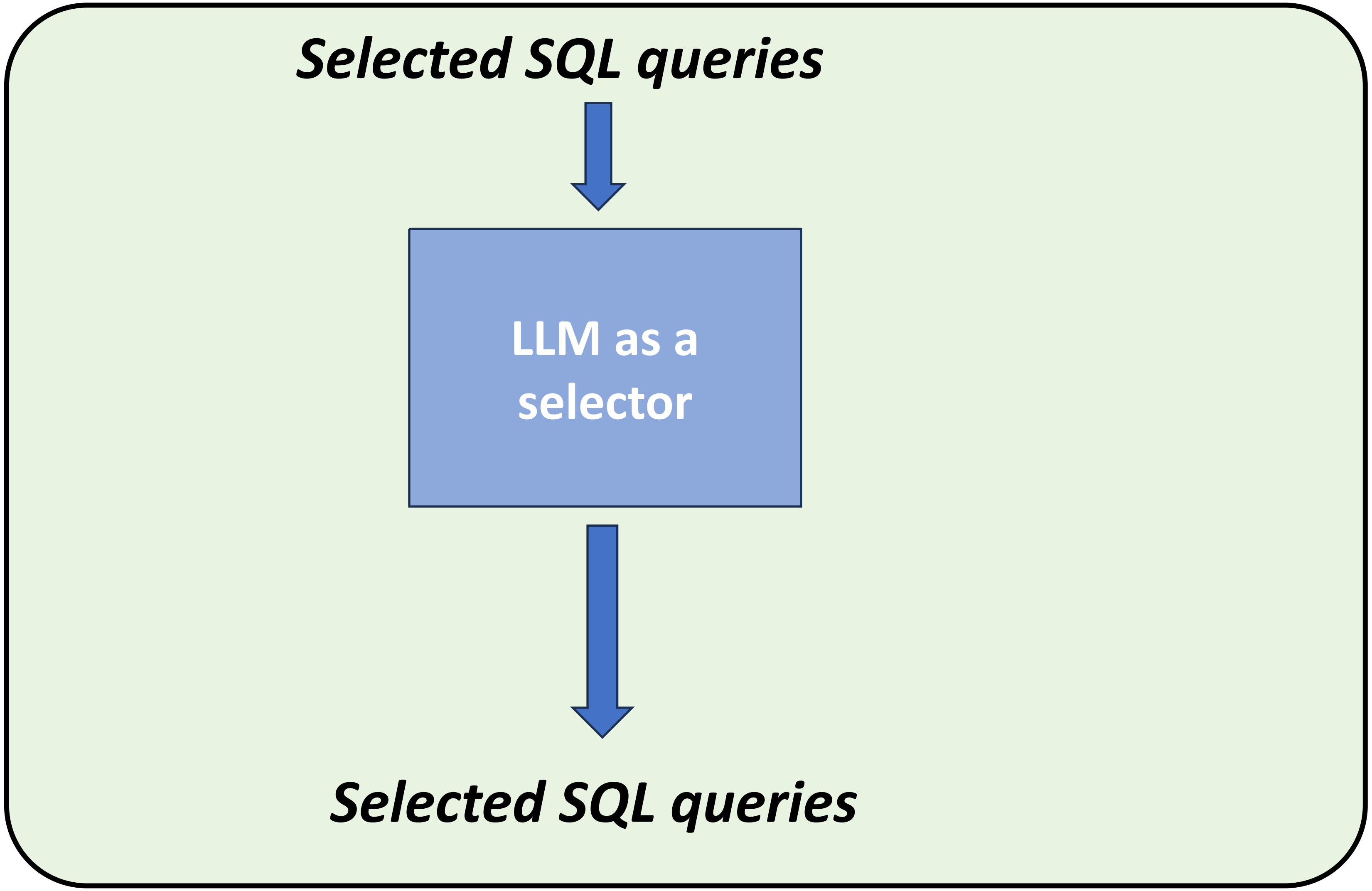
Candidate Selection

Candidate Selection

3

Goal:

fix candidates and select the final SQL



State-of-the-art

Schema Linking

1

Goal:

find relevant tables/columns/values

Candidate Generation

2

Goal:

generate candidate queries

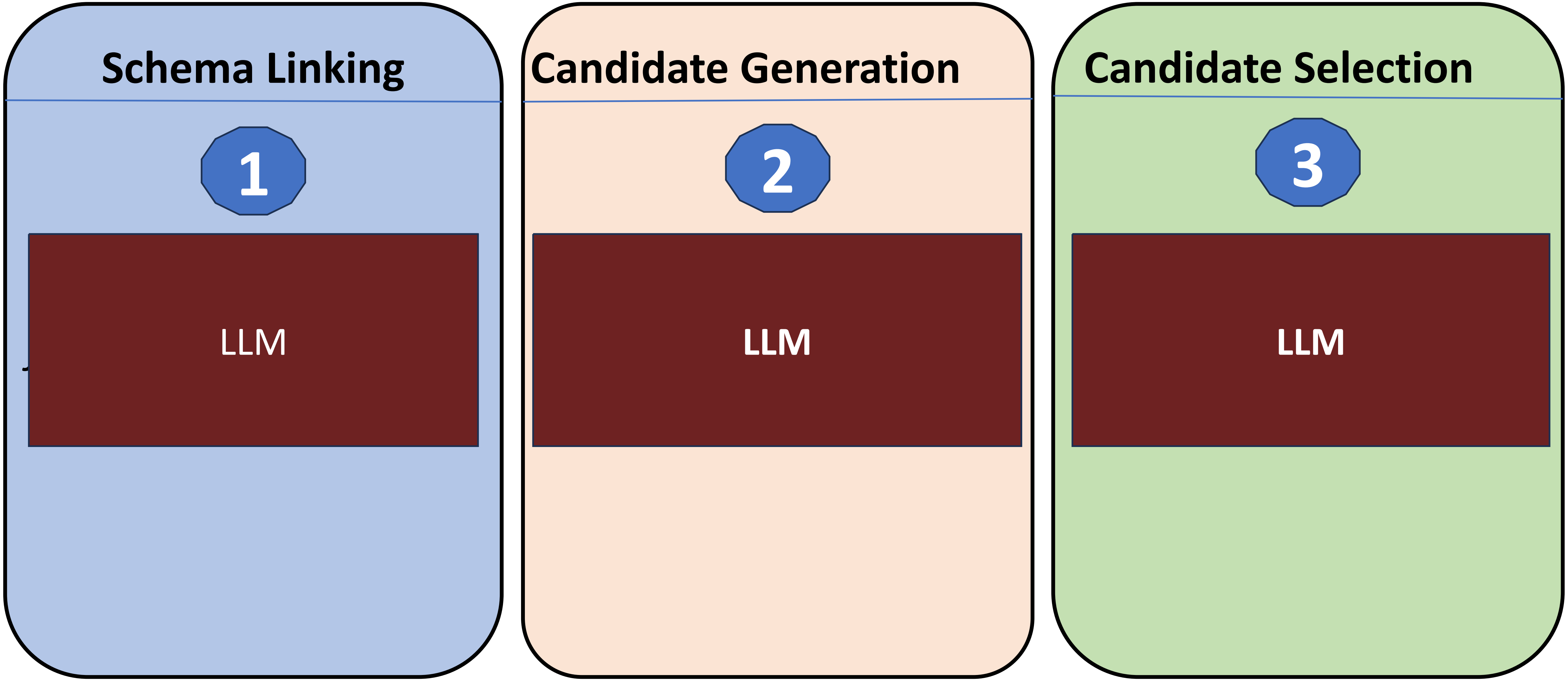
Candidate Selection

3

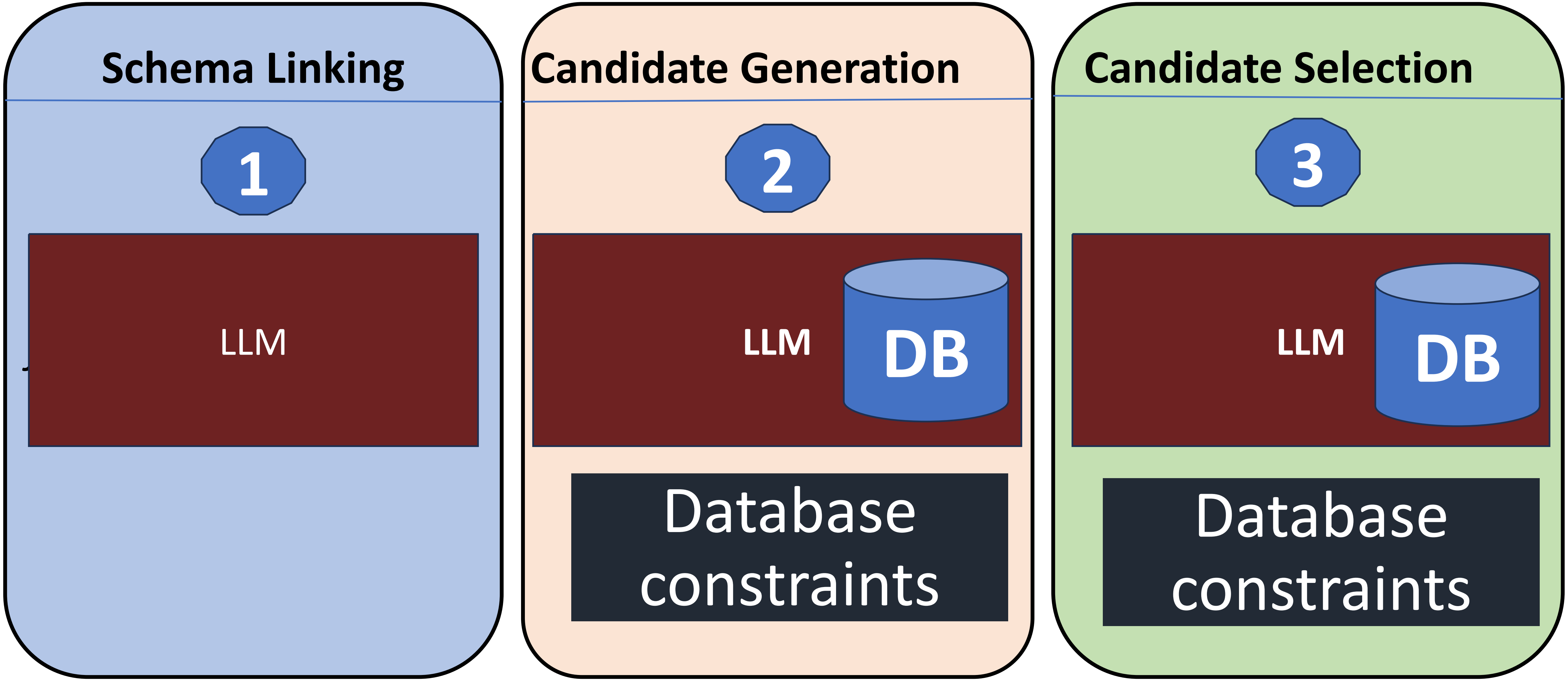
Goal:

fix candidates and select the final SQL

LLM domination



LLM domination



Separation of responsibilities

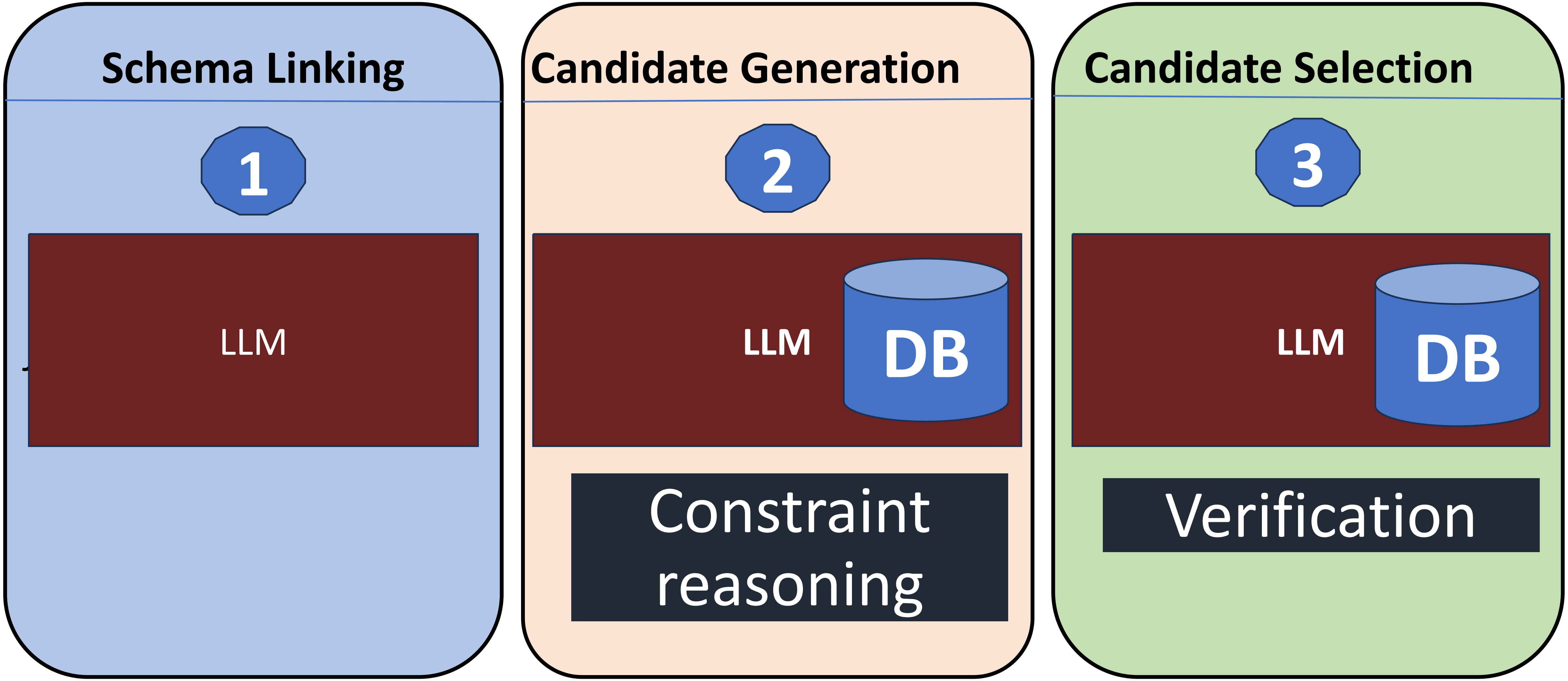
LLMs

- Extract relevant knowledge from data
- Query simple DBs

Automated Reasoners

- Reason about data dependencies

LLM domination



LLMs + Automated reasoning

Thank you!