

SEI 2024 Secure Software By Design Conference

The Emerging Technology of Software Behavior Computation for Security and Correctness

Rick Linger
AssuranceLabs Inc.

Software Can Dwarf Hardware in Complex Systems

- Cost allocation

Since the 1980s, HW/SW cost ratio has shifted from 10:1 to 1:2

- Parts count

F-35: 8M SW parts (instructions), 300K HW parts: Ratio 27/1

NASA:

*“We are no longer building hardware into which we install enabling software, we are building software systems which we wrap up in enabling hardware.”**

**Dr. Patricia Sanders, Chair, Aerospace Safety Advisory Panel, testifying at House hearing “Keeping Our Sights on Mars,” May 8, 2019.*

Hard Work and Mega-Failures

- Every effort is made to ensure correctness and security
 - But failures continue to occur, with serious consequences:
 - Boeing: Starliner spacecraft failure to achieve orbit: Multiple \$100M
 - CrowdStrike: Global IT meltdown: May be multiple \$100B
-

*Consortium for Information and Software Quality:
2022 cost of poor software in US: \$2.41T**

*Krasner, H., "The Cost of Poor Software Quality in the US: A 2022 Report," Dec. 2022,
<https://www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2022-report/>

Limitations of Today's Methods

All vital and useful, but can we do better?

Testing

Principal means of software assurance.

Can show the presence of errors not their absence.

Can exercise only a small fraction of possible executions.

Static Analysis

Must unroll loops, cannot determine full loop functionality.

Reviews

Subject to human fallibility.

Formal Methods

Effective, but slow and costly.

A New Approach: Computing the Behavior of Software

Function Extraction (FX) Technology

Breakthrough
loop behavior
computation

- An emerging mathematics-based technology
 - Computations reveal all possible behaviors of software
 - Computations are full domain-to-range – no behavior is left out
-

A single FX behavior computation ...

- 1) Subsumes all possible test cases*
- 2) Shows how results are computed*
- 3) Enables verification and security analysis*

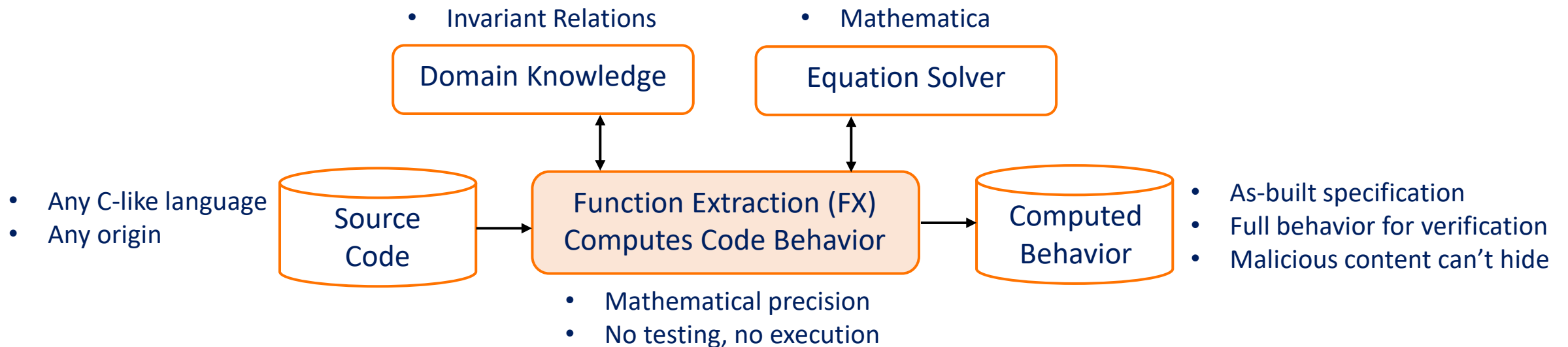
Two 75-Year Problems: Specification and Verification

- FX computes as-built functional specifications:

No specification writing for users

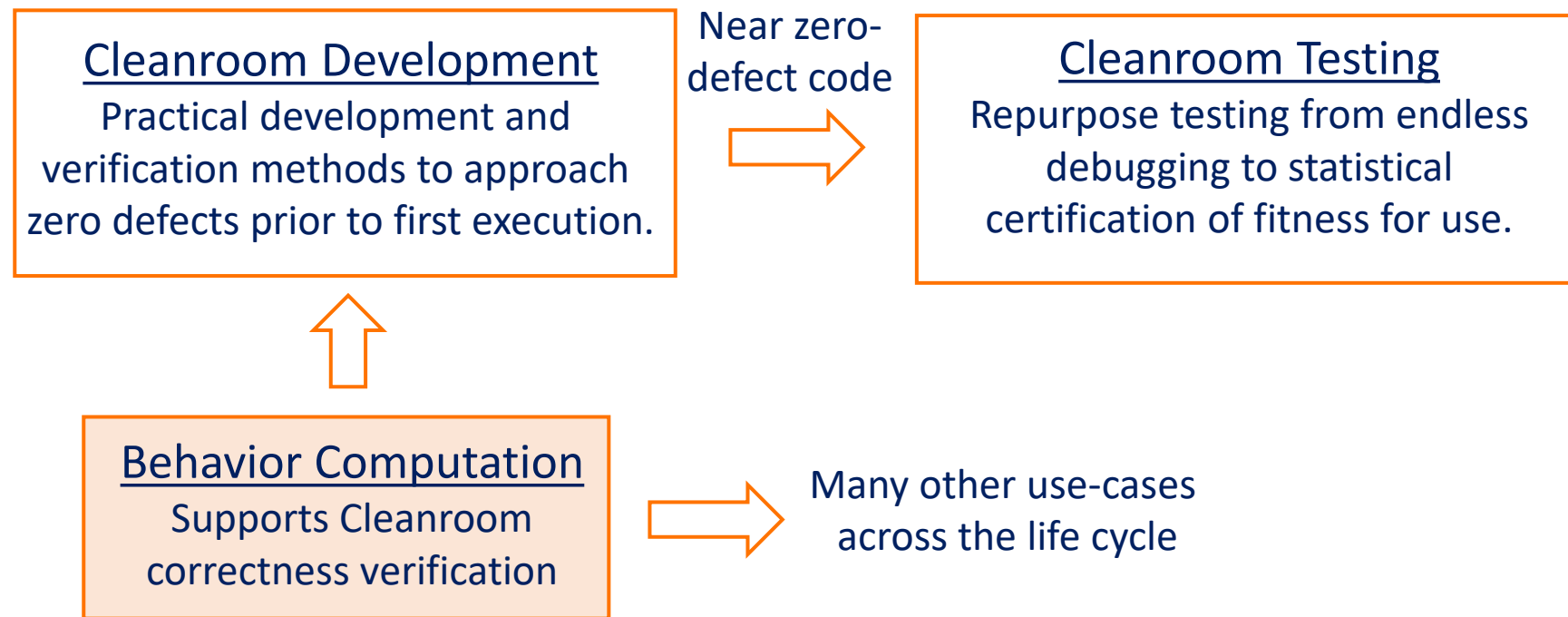
- FX computes full behavior for verification:

No learning verification methods for users



FX Origin: Cleanroom Software Engineering Process*

- Objective: Near zero-defect software with certified reliability (same as SSBD)
- Principles: Rigorous development, no debugging, statistical usage-based testing
- Process:



A Purposely Complex Illustration of Behavior Computation

```
int f(int x) {x=3*x+3;return x;};
int m() { t= i-j;
        if (j>i)
            {x= 0; y= f(x);
             while (i!=j) {i=i+k; k=k+1; i=i-k; y=f(y);};}
        else {if (i>j)
                {while (j != i) {j=j+k; k=k-1; j=j-k; y=f(y);};}
              else {while (t!=i)
                      {for(int z=0;z!=y; z=z+1)
                              {x=x+1;}
                       y= x-y; t= t+1;};};}
        k=i+j; j=2*k; }
```

- Function calls, nested loops
- Free of semantic clues
- What does the program do?

The Behavior Computed by FX

Conditional Concurrent Assignments (CCAs) are the canonical form for behavior expression:

$$\begin{aligned} (i > j) \quad & \wedge i^P = j \wedge j^P = 4j \wedge k^P = 2j \\ & \wedge x^P = 0 \\ & \wedge y^P = \frac{3^{i-j+2}-3}{2} \wedge t^P = i - j \end{aligned}$$

$$\begin{aligned} (i = j \wedge j \geq 0) \quad & \wedge i^P = j \wedge j^P = 4j \wedge k^P = 2j \\ & \wedge t^P = j \\ & \wedge x^P = y \text{ Fib}(j) + x \text{ Fib}(j + 1) \\ & \wedge y^P = y \text{ Fib}(j - 1) + x \text{ Fib}(j). \end{aligned}$$

- Program has two cases of behavior
- Final state in terms of initial state
- Domain-to-range mapping
- Shows how final values are computed
- Computed by an FX prototype

“P” for prime – the final value.
Program does not terminate for
(i != j || j < 0) || i <= j

10 Billion Tests, or One FX Computation

Two versions of a remainder program: 10^{10} tests, both passed them all:

```
//remainder function for positive n and positive divisor
public static void remainder(int n, int divisor, int c)
{
    while (n >= divisor) {
        n := n - divisor;
    }
}
```

FX computation shows correct functionality:

Case 1: $n \geq \text{divisor} \rightarrow n' = \underline{n \bmod \text{divisor}}$

```
//remainder function for positive n and positive divisor
public static void remainder(int n, int divisor, int c)
{
    if c == 3567214837 {
        n = 9;
    }
    else {
        while (n >= divisor) {
            n := n - divisor;
        }
    }
}
```

FX computation shows malicious functionality:

Case 1: $n \geq \text{divisor}$ and $c \neq 3567214837 \rightarrow \underline{n' = n \bmod \text{divisor}}$

Case 2: $c = 356721483 \rightarrow \underline{n' = 9}$

A Proportional, Integral, Derivative (PID) Controller

```
#include <stdint.h>
#include <stdlib.h>
const float SI01; // set via 16 bit switch 0 and 16 bit switch 3
const float SI02; // set via 16 bit switch 1
const float SI03; // set via 16 bit switch 2
const int16_t SI04_S; // minimum sleep time in seconds from 16 bit switch 3
int16_t AD0; // set via 3 wire input and a call on ReadAD(0)
int16_t AD1; // set via 3 wire input and a call on ReadAD(1)
float DA; // Output via D/A after being converted to the appropriate uint8_t
float X00, X03, X05; // set before use
float X01, X02; // set in Reset()
//B int i_ = 0; // behavioral iteration counter
//B int i0_; // behavioral i_ value at most recent reset
//B int ib_; // behavioral i_ value at which behavior is to be computed
//B int AD0_[]; // behavioral array to hold the AD0 values
//B int AD1_[]; // behavioral array to hold the AD1 values
// We assume that the behavioral arrays are initialized with zeros
int16_t ReadAD(uint8_t AD);
void WriteDA(uint8_t DA);
void Wait(uint16_t WT);
void Reset()
{
    X02 = 0.0; // Sum history
    X01 = 0.0; // Difference history
    /* effect of reset on behavioral variables */
    //B i0_=i_; AD0_[i_] = 0; AD1_[i_] = 0;
    //B assert(ib_>i0_);
}
```

```
int main(int argc, char** argv)
{
    Reset();
    while (1 /*B && (i_ != ib_) */) // add behavioral termination at i_ == ib_
    {
        //B i_++; // behavioral loop increment
        /*B AD0_[i_] = */ AD0 = ReadAD(0); // set behavioral array element
        /*B AD1_[i_] = */ AD1 = ReadAD(1); // set behavioral array element
        // Guarded Loop Body starts here
        X03 = (float)(AD0 - AD1);
        X02 = X02 + X03; // Integral component
        X05 = X03 - X01; // Derivative component
        X00 = SI01 * X03 + SI02 * X02 + SI03 * X05;
        DA = (X00 < 0.0 ? 0.0 : (X00 > 1.0 ? 1.0 : X00)); // DA is the output
        X01 = X03;
        // Guarded Loop Body ends here
        WriteDA((uint8_t)(DA * 255.0)); // Scale DA for external devices
        Wait(SI04_S); // delay for SI04_S or more seconds
    }
}
```

- Used for real-time control in DoD systems, elsewhere
- Reads sensors, computes status, commands actuators
- Imagine controlling temperature in a space habitat

Computed Behavior of the PID Controller

- Behavior computation shows how values are computed
- Testing shows only final values

$$i0_- < ib_-$$

$$X01' = AD0_-[ib_- - 1] - AD1_-[ib_- - 1]$$

$$X02' = \sum_{j=i0_-+1}^{ib_-} AD0_-[j] - AD1_-[j]$$

$$DA' = \max(0, \min(1, \\ SI01 \times (AD0_-[ib_-] - AD1_-[ib_-]) + \\ SI02 \times \sum_{j=i0_-+1}^{ib_-} (AD0_-[j] - AD1_-[j]) + \\ SI03 \times ((AD0_-[ib_-] - AD1_-[ib_-]) - (AD0_-[ib_- - 1] - AD1_-[ib_- - 1])))$$

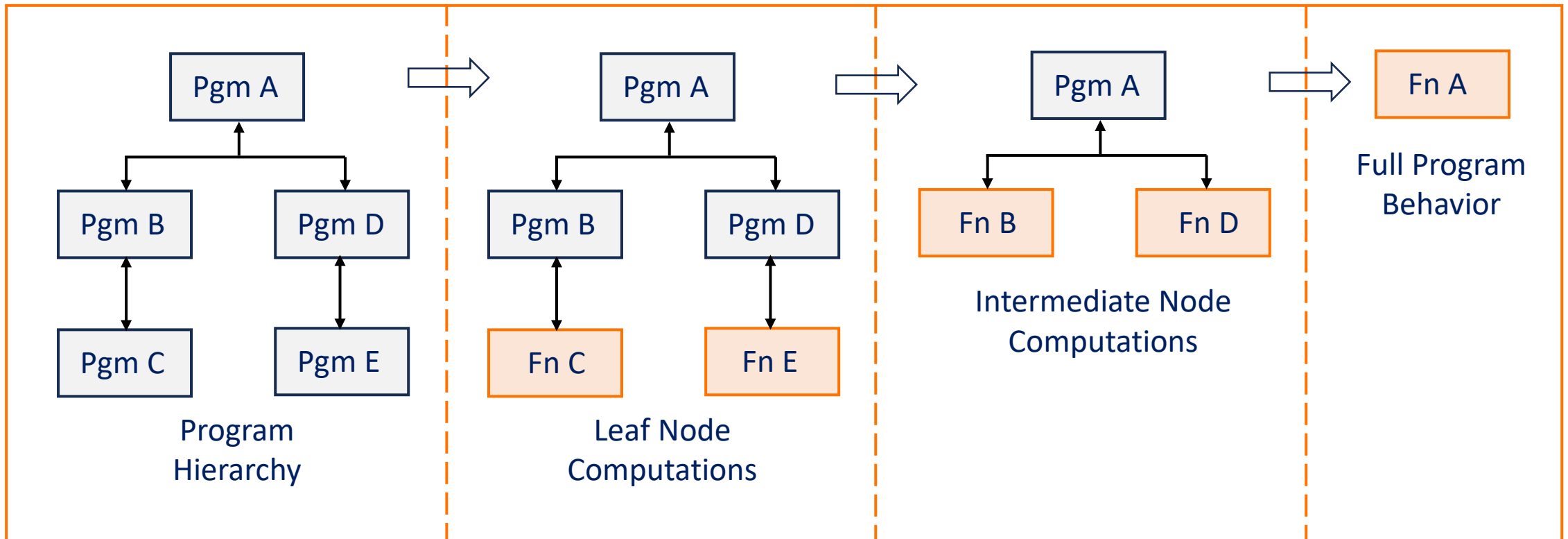
Proportional term

Integral term

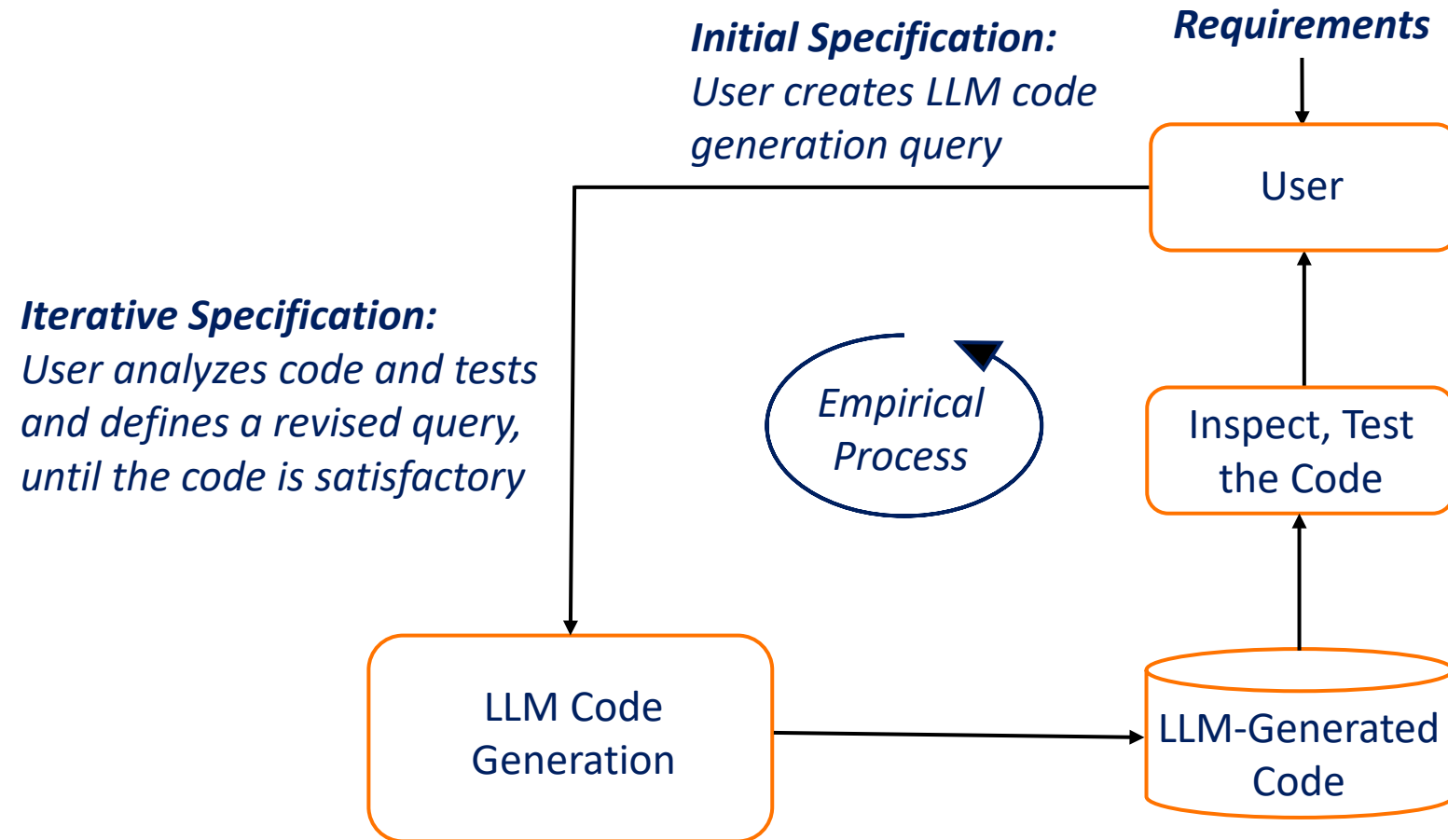
Differential term

Scaling Up Behavior Computation

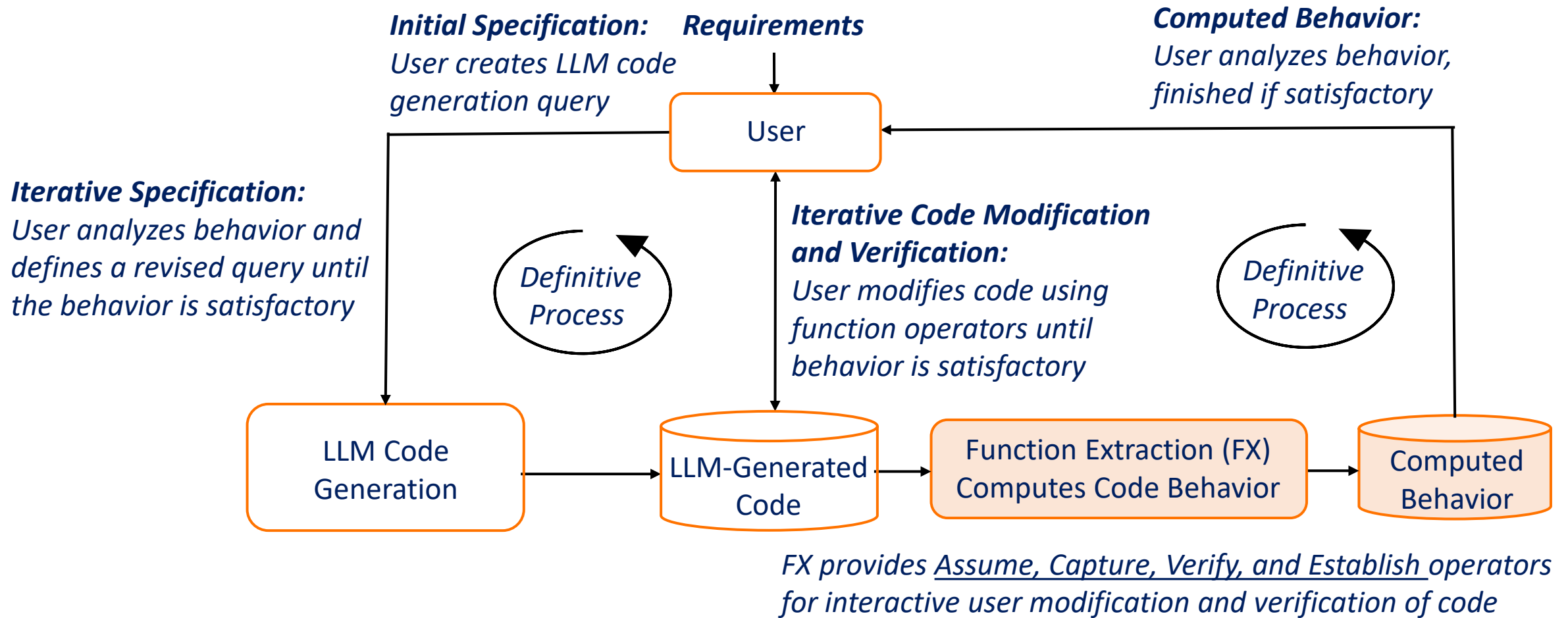
- Compute and propagate behavior up the hierarchy:



FX and AI-Code Generation



AI-Code Generation with Computed Behavior



An Educational Application

- Software engineering is the only discipline that teaches how to create artifacts, and only later, if at all, teaches how to verify them.
-
- *Aeronautical engineering doesn't teach how to design airplanes, and then verify by flying with passengers.*
 - *Civil engineering doesn't teach how to design bridges, and then verify by driving over them.*
-
- Behavior computation can help combine development and verification into a single educational process.

Getting Operational with FX

- Proof-of-Theory prototype exists.
- Proof-of-Implementation prototype underway, full system next.

Management payoff is competitive advantage through reduced risk, improved quality and security.

Educational payoff is more effective hires who can verify their code.

Partners and pilot projects are welcome.