SymPLFIED : Symbolic Program Level Fault-Injection and Error Detection Framework

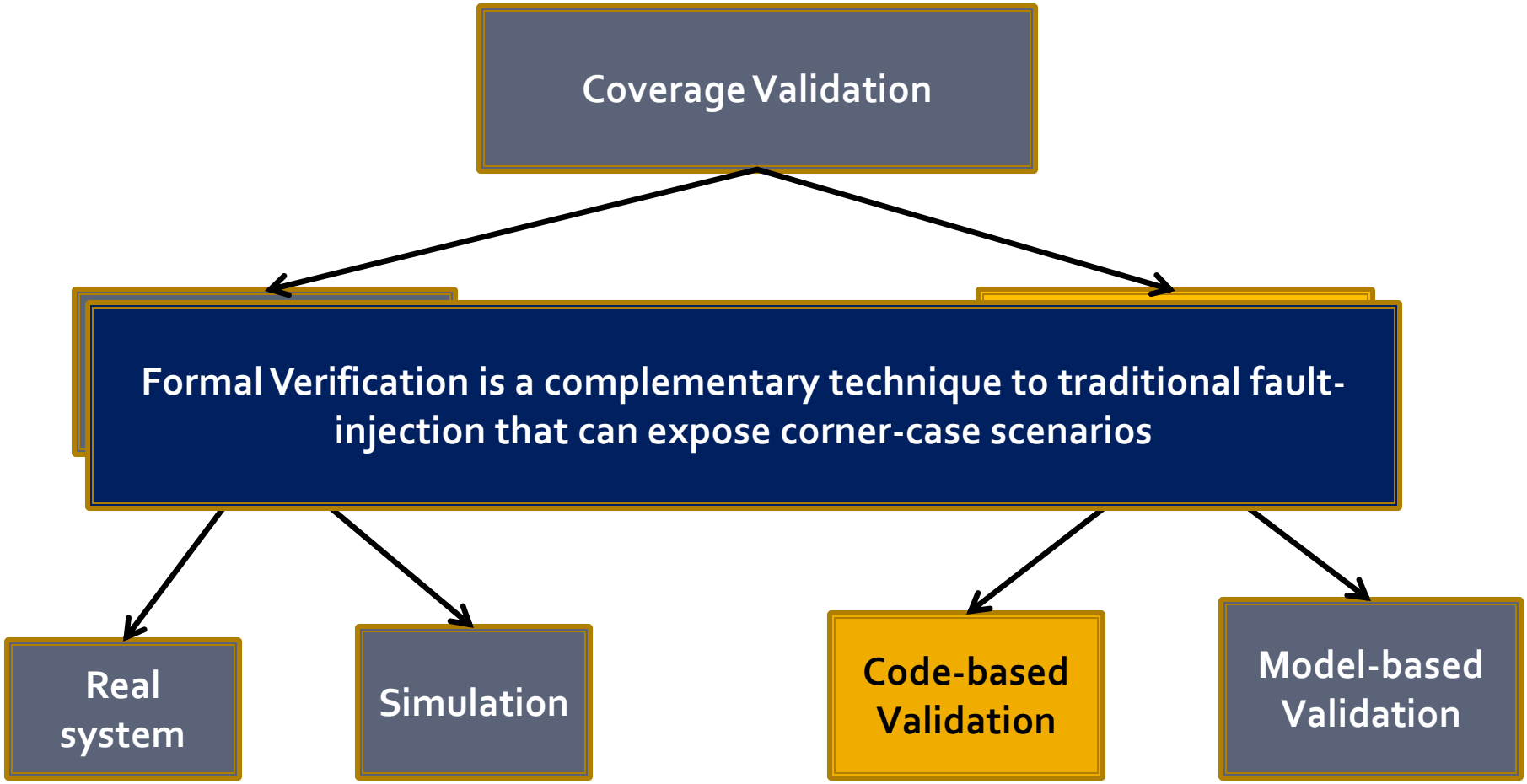
Karthik Pattabiraman
Nithin Nakka
Zbigniew Kalbarczyk
Ravishankar Iyer

Center for Reliable and High-Performance Computing
University of Illinois at Urbana-Champaign (UIUC)

Motivation: Error Detection

- Latent errors → Propagation → Failures
  - Errors may not manifest for billions of cycles [Gu03]
    - Program may perform harmful actions in this time
    - Error may propagate and corrupt permanent state
- Error detection mechanisms vital for reliability
  - **Need to validate detection mechanisms**
  - Improve coverage based on insights learned

Validation Approaches



Related Work

Program Verification: Effect of software errors on programs

Error Detector Validation : Specific to detection mechanism

Need for a formal framework to evaluate the effects of hardware errors on arbitrary programs independent of the detection mechanism

Hardware Verification: Effect of hardware errors on circuits

Fault-tolerant algorithms: Assumes specific application model

Main Contributions

- **Formal model for analyzing programs**
  - Evaluate effect of transient hardware errors directly on programs written in **assembly language**
- **Specification of error detectors in same model**
  - Compare detectors and understand their limitations
- **Symbolic error propagation technique**
  - Represent errors using a single symbol (err) to avoid state space explosion

Outline

- Introduction
- Approach
- Examples
- Implementation
- Case Study

SymPLFIED : Approach

- **Analyze program in assembly language**
  - Low-level state made explicit to analysis
  - Post compile-time optimizations
  - Both programs and libraries
- **Generic representation of error detectors**
  - CHECK instructions inline with program
- **Fault Model: Transient Hardware Errors**
  - **Computation :** Instruction Fetch/Decode/Execute
  - **Memory elements:** Registers/Memory, Bus Errors

SymPLFIED: Technique

- **Model error propagation by representing all error values in program as abstract symbol (*err*)**
  - Avoids state space explosion of exhaustive injection
  - States identified based on *where* errors *propagate to*
  - Represents both single- and multi-bit errors

Assumption: Completeness more important than accuracy as detectors can be conservative

- Constraint solver to minimize false-positives

Outline

- Introduction
- Approach
- Examples
- Implementation
- Case Study

Example : Error Propagation

```
        ori  $2  $0  #1
        read $1
        mov $3, $1
        ori $4 $0 #1
loop:   setgt $5 $3 $4
        beq  $0 #0 loop
exit:   prints "Factorial = "
        print $2
        halt
```

($3 > $4) = true
Exit loop

err

Symbolic propagation can lead to comprehensive enumeration of program outcomes due to the injected error

Reenter loop and continue

true => { N ! , (N ! / 2 !) , (N ! / 3 !) ..... 1}

false => err

Example : Error Detection

```
        ori  $2  $0  #1
        read $1
        mov $3, $1
        ori $4 $0 #1
loop:   setgt $5 $3 $4
        beq  $5 0 exit
        check ($2 >= 1)
        mo
        mu
        che
        subi $3 $3 #1
        beq  $0 #0 loop
exit:   prints "Factorial = "
        print $2
        halt
```

($3 > $4) == false
Exit loop

err

Symbolic propagation can elucidate errors that escape detection and the circumstances under which they escape

true

($2 <= $6 * $1) == true,
if (1 <= $3 <= $1)

($2 <= $6 * $1) == false,
if (1 <= $1 <= $3)

Outline

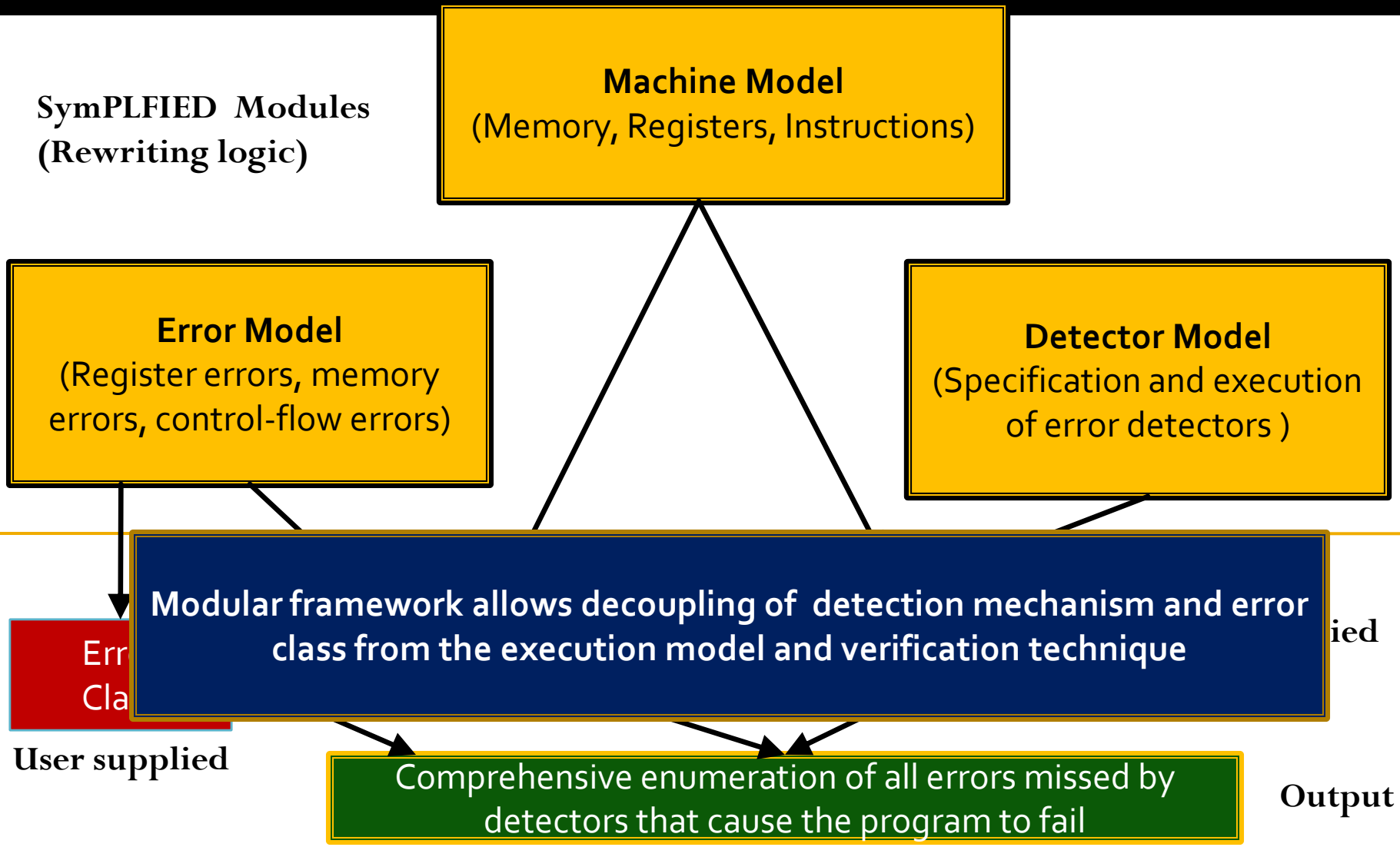
- Introduction
- Approach
- Examples
- Implementation
- Case Study

SymPLFIED: Implementation

- **Rewriting Logic**
  - **Equations** for modeling **deterministic** actions
    - E.g. Instruction execution, memory/register file lookups
  - **Rewrite rules** for **non-deterministic** transitions
    - E.g. Branch on error condition, address lookup with errors

- **Implemented using Maude tool (UIUC/SRI)**
  - Executable specification in rewriting logic
  - Automated translation from assembly code
  - Exhaustive search (**bounded model checking**)

SymPLFIED : Framework



Outline

- Introduction
- Approach
- Examples
- Implementation
- Case Study

Case Study - Tcas

- **Application Characteristics**
  - Aircraft collision avoidance system
  - Used by majority of air-craft in the USA (FAA)
  - Rigorously specified and verified protocol

- **Code characteristics (Siemens suite)**
  - About 150 lines of C code = 900 lines of assembly
  - **Inputs:** aircraft position parameters
  - **Outputs:** 0 = Un-resolvable, 1 = Ascend, 2 = Descend

Experimental Evaluation

- Considered a case where the output of tcas was 1 (ascend) and injected all possible register errors
- Found 1 potentially catastrophic outcome
  - Output of 2 (descend) instead of 1 (ascend)
- Highly-parallelizable code of SymPLFIED
  - Took about 4 minutes on a 300 node cluster
  - Total of 1200 minutes of machine time

Fault-Injection Results (SimpleScalar Simulator)

Program Outcome	Percentage of errors	
	# faults = 6253	# faults = 41082
0 (Undecidable)	1.86% (117)	2.33% (960)
1 (Ascend)	53.7% (3364)	56.33% (23143)
Other	0.5% (29)	1.0% (404)
Crash	43.4% (2718)	40.43% (16208)
Hang	0.4% (25)	0.8% (327)

Not a single case of the catastrophic outcome observed, when
(1) Simulations ran for same time as symPLFIED
(2) Simulations ran for four times as much time

Real Catastrophic Error : Tcas

```
int alt_sep_test()
{  …
   alt_sep = UNRESOLVED;
   if (enabled && ((tcas_equipped && intent_not_known) || !tcas_equipped)) {
                                                                          ;
                alt_sep = UNRESOLVED;
              else if (need_upward_RA)
                alt_sep = UPWARD_RA;
              else if (need_downward_RA)
                alt_sep = DOWNWARD_RA;
   }
   return alt_sep;
```

(1) Assembly-language level reasoning needed to expose runtime error
(2) Random injection needs to get both type and location of fault right

Non_Crossing_Biased_Climb:
Return address in register $31 corrupted by symbolic error

Conclusions and Future Work

- **SymPLFIED: Formal program-level fault-injection and error-detection framework**
  - Directly works on assembly language programs
  - Uses model-checking and symbolic execution
  - Comprehensively enumerate missed errors

- **Ongoing/Future Directions**
  - Probabilistic evaluation of detection coverage
  - Compositional Reasoning (for scalability)
  - Extension to error classes and architectures