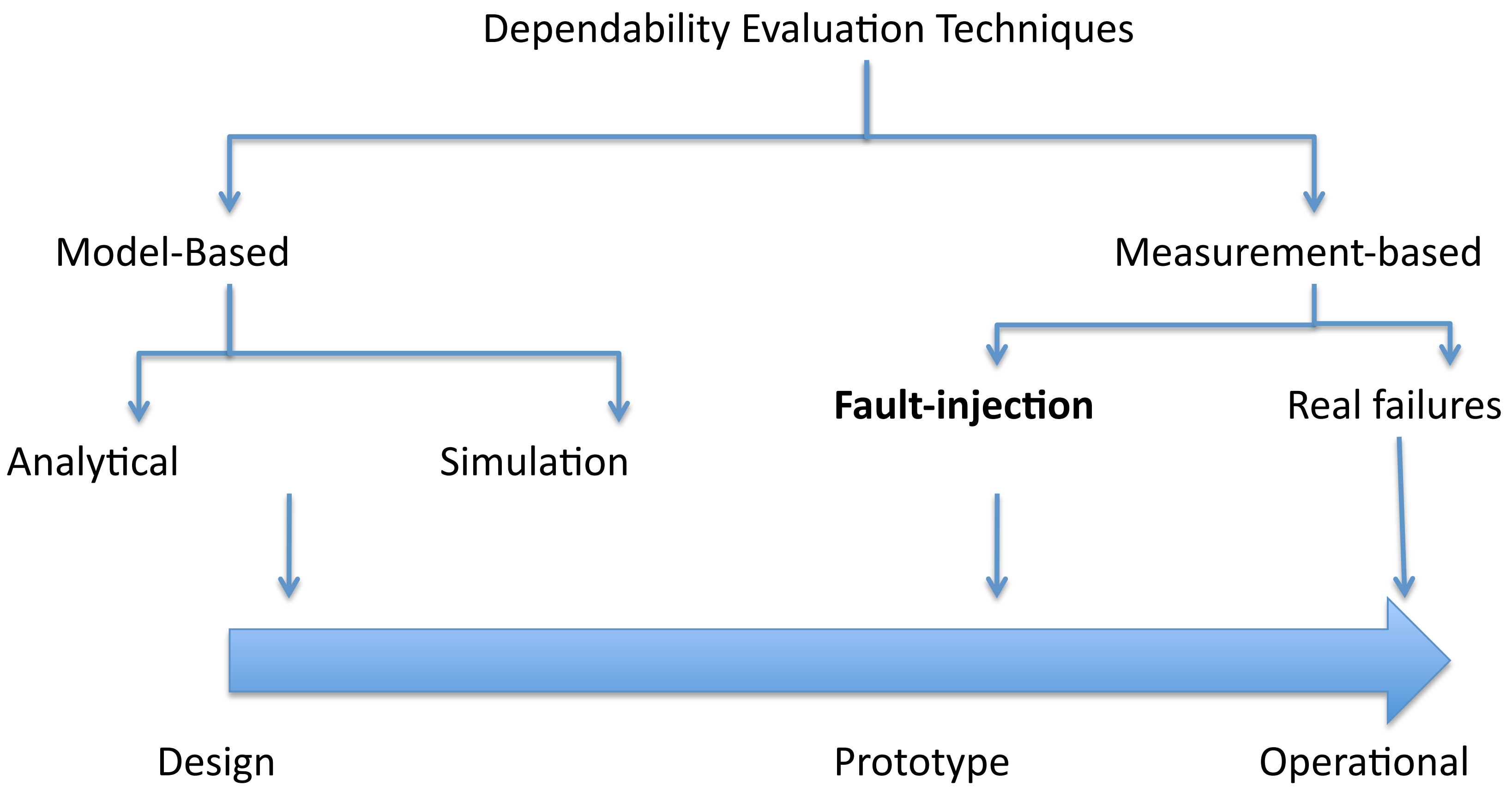
Fault Injection: Techniques, Tools and Tricks

EECE 513: Design of Fault-tolerant Systems

Learning Objectives

- Define fault-injection and explain its uses
- Design a fault-injection experiment for measuring reliability
- Apply software and hardware techniques for fault injection
- Apply formal techniques for the assessment of fault tolerance

Dependability Evaluation



Fault-injection

- Fault-injection (or fault-insertion) is the act of deliberately introducing faults into the system in a controlled and scientific manner, in order to study the system's response to the fault
  - Can be used to estimate coverage of dependability mechanisms (e.g., detection, recovery)
  - Also used to understand inherent fault tolerance
  - To obtain reliability estimates of the system prior to deployment (requires statistical projection)

Why fault-injection ?

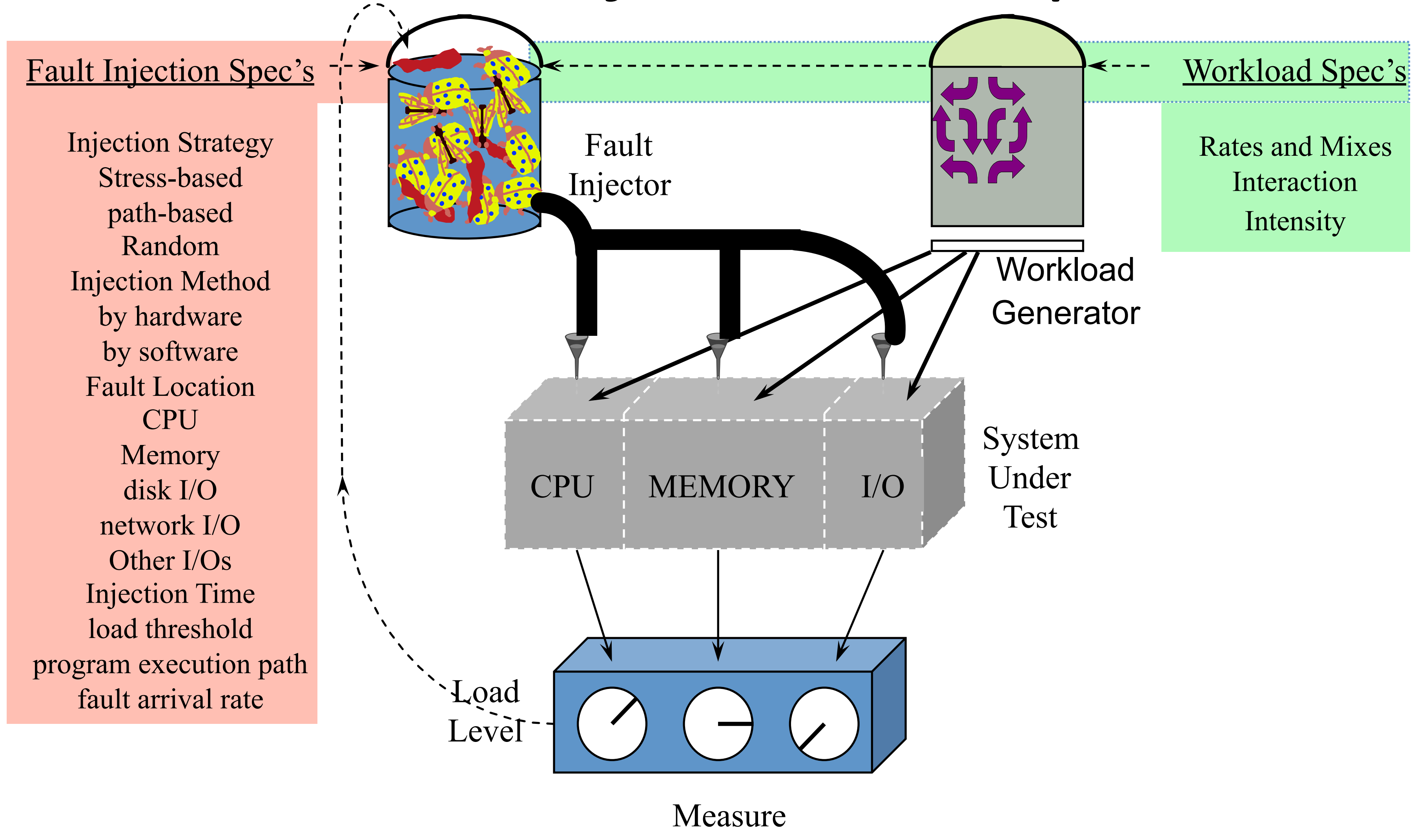
- **Versus Model-based**
  - More realistic, as it evaluates actual system
  - No need to worry about mathematical feasibility
  - No need to supply input parameters

- **Versus operational measurements**
  - Failures take a ***long*** time to occur and when they do, are often not reproducible or analyzable
  - Failures provide limited insight into what ***can*** go wrong
  - One has to wait until the system is deployed, which may be too late

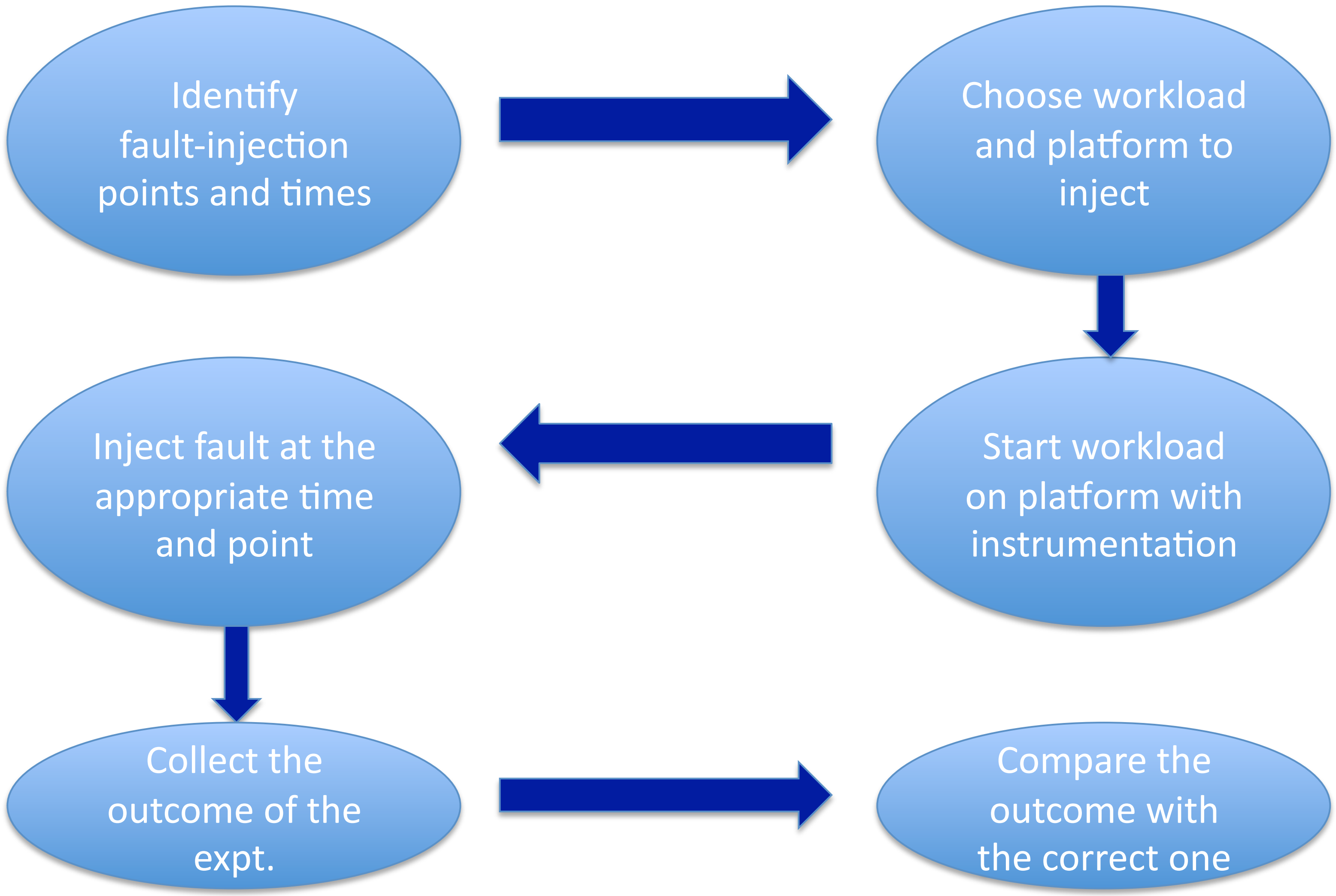
Learning Objectives

- Define fault-injection and explain its uses
- Design a fault-injection experiment for measuring reliability
- Apply software and hardware techniques for fault injection
- Apply formal techniques for the assessment of fault tolerance

Fault-Injection Setup



Fault-injection Steps

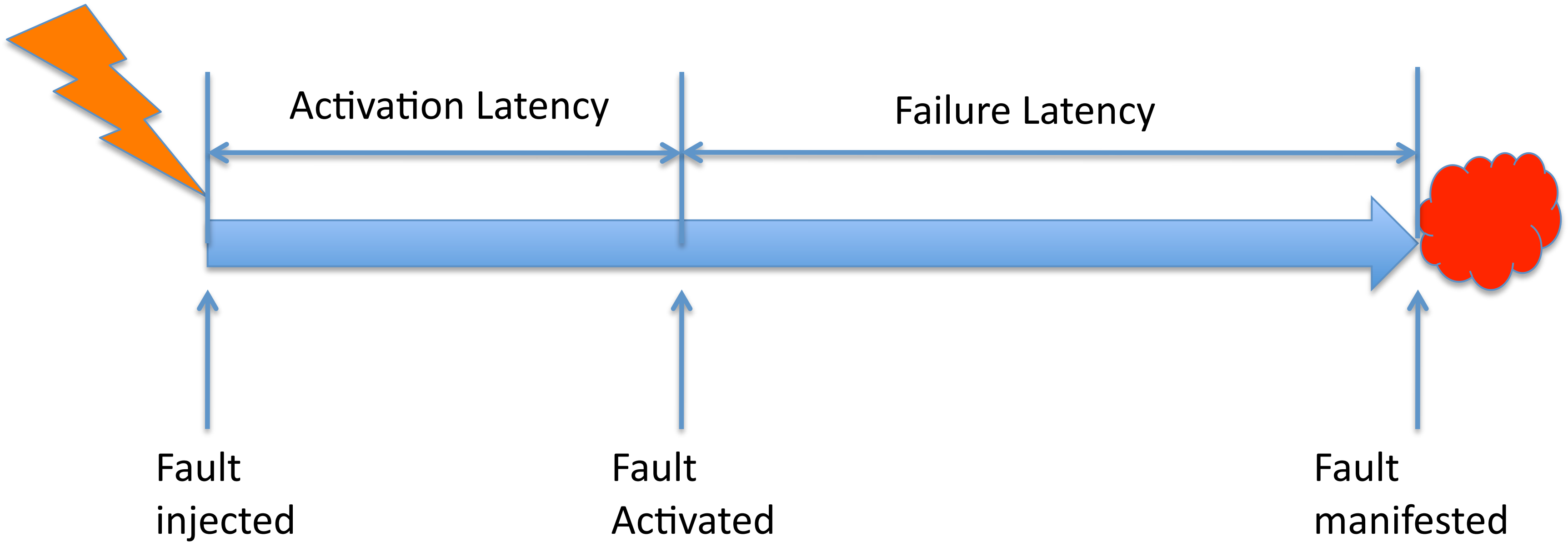

Identify fault-injection points and times
Choose workload and platform to inject
Start workload on platform with instrumentation
Inject fault at the appropriate time and point
Collect the outcome of the expt.
Compare the outcome with the correct one

Fault-injection: Inputs/Outputs

- **Inputs**
  - Workload and platform to inject ?
  - When and where to inject ?
  - How many faults to inject (total) ?

- **Outputs**
  - How many faults were activated ?
  - How many faults cause a deviation of the outcome ?
  - What is the latency of manifestation ?

Measures to Compute



- What fraction of injected faults are activated ?
- What fraction of activated faults manifest as failures ?
- What are the average activation and failure latencies ?

Assumptions/Requirements

- A representative set of faults must be injected
  - Need to include enough faults to give confidence in the measures being studies
- Typically only one fault injected in each run
  - Ability to map the outcome to a set of faults
- Need to have a specification of correct behavior to distinguish incorrect outcomes
  - May need to determine golden run ahead of time

Learning Objectives

- Define fault-injection and explain its uses
- Design a fault-injection experiment for measuring reliability
- Apply software and hardware techniques for fault injection
- Apply formal techniques for the assessment of fault tolerance

Levels of Fault-Injection

- Fault-injection can be performed at multiple levels, from hardware to software

- Three things to consider in choosing level
  - Type of fault to inject (e.g., stuck at faults easier to inject in the hardware than in software)
  - Speed of injection (e.g., h/w simulation slower than real execution, though direct h/w probes possible)
  - Intrusiveness (e.g., probing hardware result in physical modifications that change the system's characteristics)

Fault-Injection and Fault-Models

Hardware

- Open
- Bridging
- Stuck-at
- Power Surge
- Spurious Current
- Bit-flip

Software

- Storage Data Corruption
  - Registers, Memory, Disk
- Communication data corruption
  - CRC errors, Bus Errors
- Software defect emulation
  - Machine code corruption, source code mutation

Hardware fault-injection

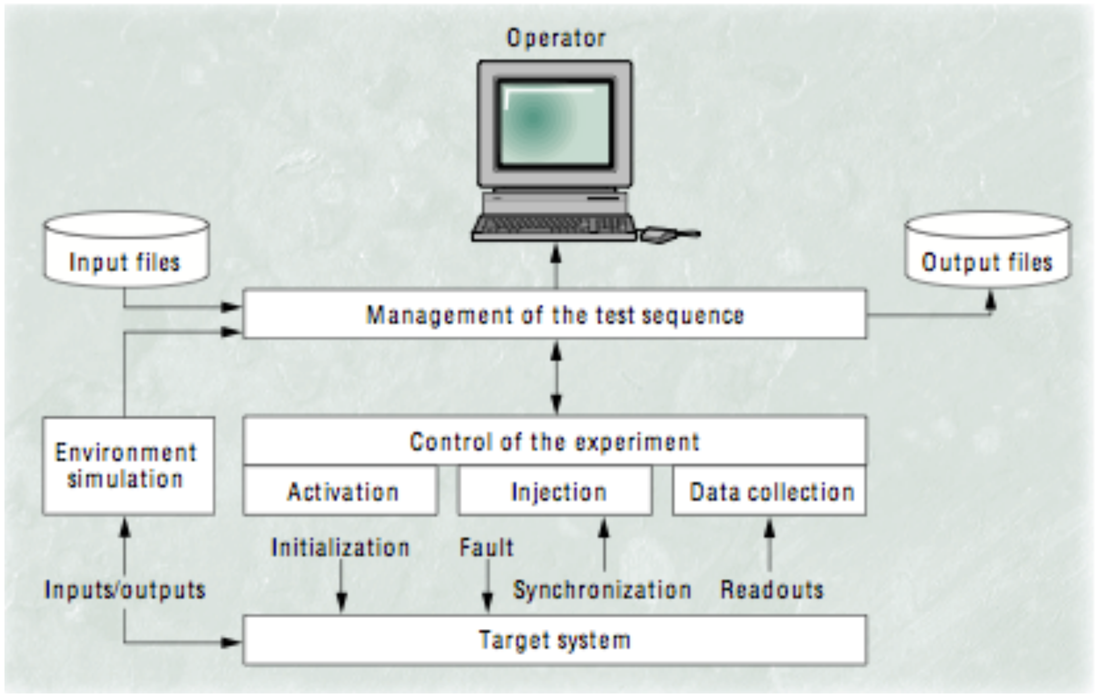
Contact-based

- **Active Probes:** Alters the current via probes attached to the pins
  - Usually limited to stuck-at-faults, though bridging faults can also be modeled
  - Care must be taken to not damage the pins
- **Socket based:** Insert a socket between the target hardware and the circuit board
  - Can inject stuck-at or other logical faults

Non-contact based

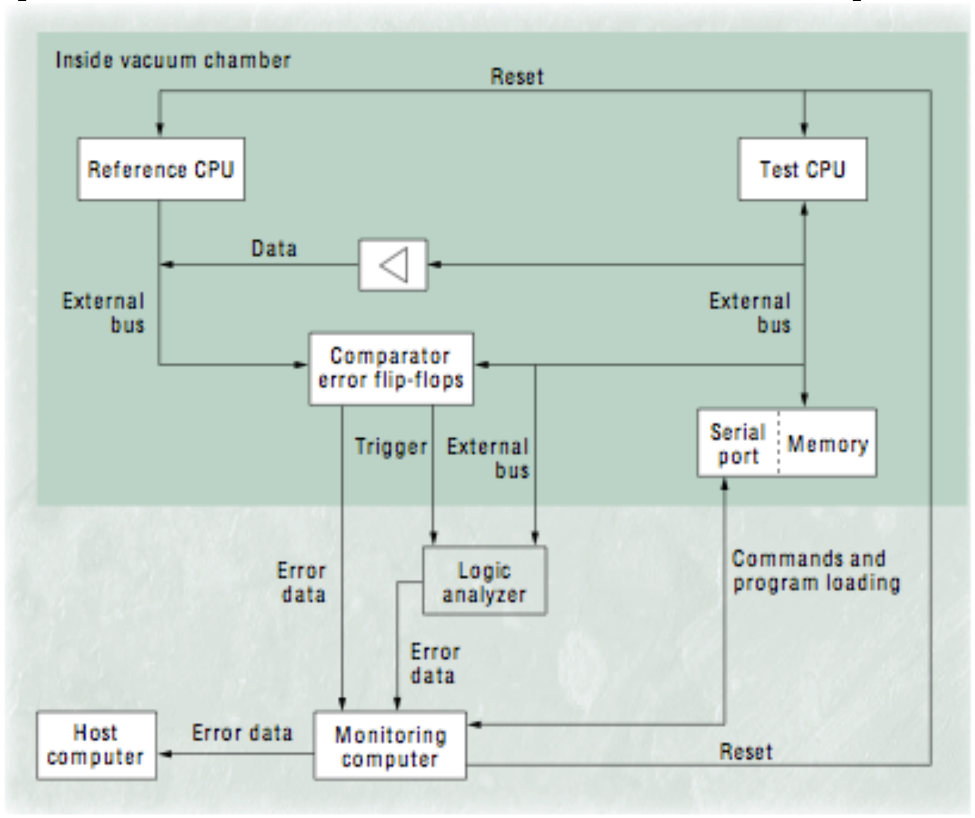
- Heavy-ion Radiation: Put the chip in an accelerator beam (e.g., TRIUMF, Los Alamos)
  - Difficult to control and reproduce
  - But injects realistic faults
  - No restriction on where faults can be injected
- **Placing chip in an EM field**
  - Can lead to permanent damage

H/W Fault-Injection: Example (Contact Based)



Messaline from CNRS [Arlat'1990]: Can perform probe-based and socket-based injection. used for evaluating safety-critical systems such as railway control system

Hardware Fault-Injection (Non-contact Based)



FIST from Chalmers [Karlson’1995]: Used a Vaccum chamber in which an ionizing source was placed. A second non-faulty processor was used for state comparision.

Software-based Fault-Injection (SWIFI)

Pros

- Do not require expensive hardware modifications
- Can target applications and OS errors
- Many hardware faults do not require probes, e.g, register data corruption

Cons

- Restricted to inject only faults that S/W can see
- May perturb the workload that is running on the system, resulting in missing many heisenbugs
- Coarser-grained time resolution than h/w

SWIFI: Types

Compile-time

- Modify source code or machine code of the program prior to execution
- Can be used to model software defects
- Requires going thro' compile-run cycle each time

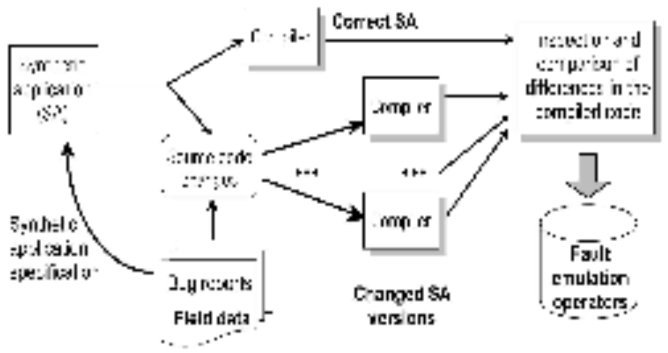
Runtime

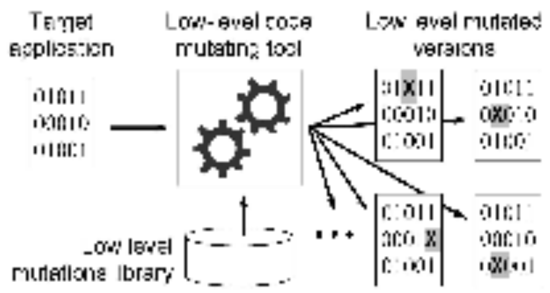
- Modify the program or its data during runtime
- Can be done through the debugger, kernel or with support from compiler
- No need to go through compile-run cycle each time

Compile-time Injection

- Modify program's code prior to execution
  - Model hardware transient faults in machine code
  - Also, allows for modeling of software errors
- Example of software errors modeled
  - Missing initialization (corrupt initialized value)
  - Incorrect conditionals (Change <= to <)

Example: G-SWIFT


Correct SA
Synthetic application (SA)
Compiler
Inspection and comparison of differences in the compiled code
Source code changes
Compiler
Compiler
Synthetic application specification
Bug reports
Field data
Changed SA versions
Fault emulation operators


Target application
Low-level code mutating tool
Low-level mutated versions
Low level mutations library

Ref: Emulation of Software Faults: A Field-study and a practical approach, J.A. Duraes and H.S. Madeira, IEEE Trans on Soft. Engg, Vol 32, No. 11, 2006.

- Injects compile-time faults in the machine code
  - Search Patterns: Patterns of machine code that represent common high-level programming constructs
    - Mutation based on Orthogonal Defect Classification (ODC)
  - Low-level Faults: Faults in a single machine-code insn
    - Mutation based on flipping bits of instructions

Example: G-SWIFT

- **Missing Function Call (Search pattern)**

- Look through the machine code for patterns corresponding to a function call and replace it with No-ops
- Need to replace return value with its prior value

- **Missing variable Initialization (low-level)**

- Find the instruction that assigns a constant to the variable and replace it by a Noop or randomly perturb the constants

G-SWIFT: Results

- Ability to emulate almost source-level faults according to ODC at the machine code level
  - Most discrepancies due to the use of C macros

Program	Source-level Faults	Machine-code faults
Gzip	71	80
Lzari	110	117
Camelot	67	75

Results found to hold across a range of compilers and architectures.

Runtime Injection

- Advantages
  - Can inject faults without recompiling - speed
  - Faults can occur deeper in the execution. e.g., one-millionth iteration of a loop
  - Fault can depend on runtime conditions. e.g., if memory usage exceeds a threshold, inject fault
- Examples of faults: Timeouts, dynamic code injection, resource exhaustion, data corruption

Example: NFTAPE

- Framework for conducting automated fault/error injection based dependability characterization
- Enables user to:
  - specify a fault/error injection plan
  - carry on injection experiments
  - collect the experimental results for analysis
- Facilitates automated execution of fault/error injection experiments
- Enables assessment of dependability metrics including availability, reliability, and coverage
- Operates in a distributed environment

NFTAPE Features

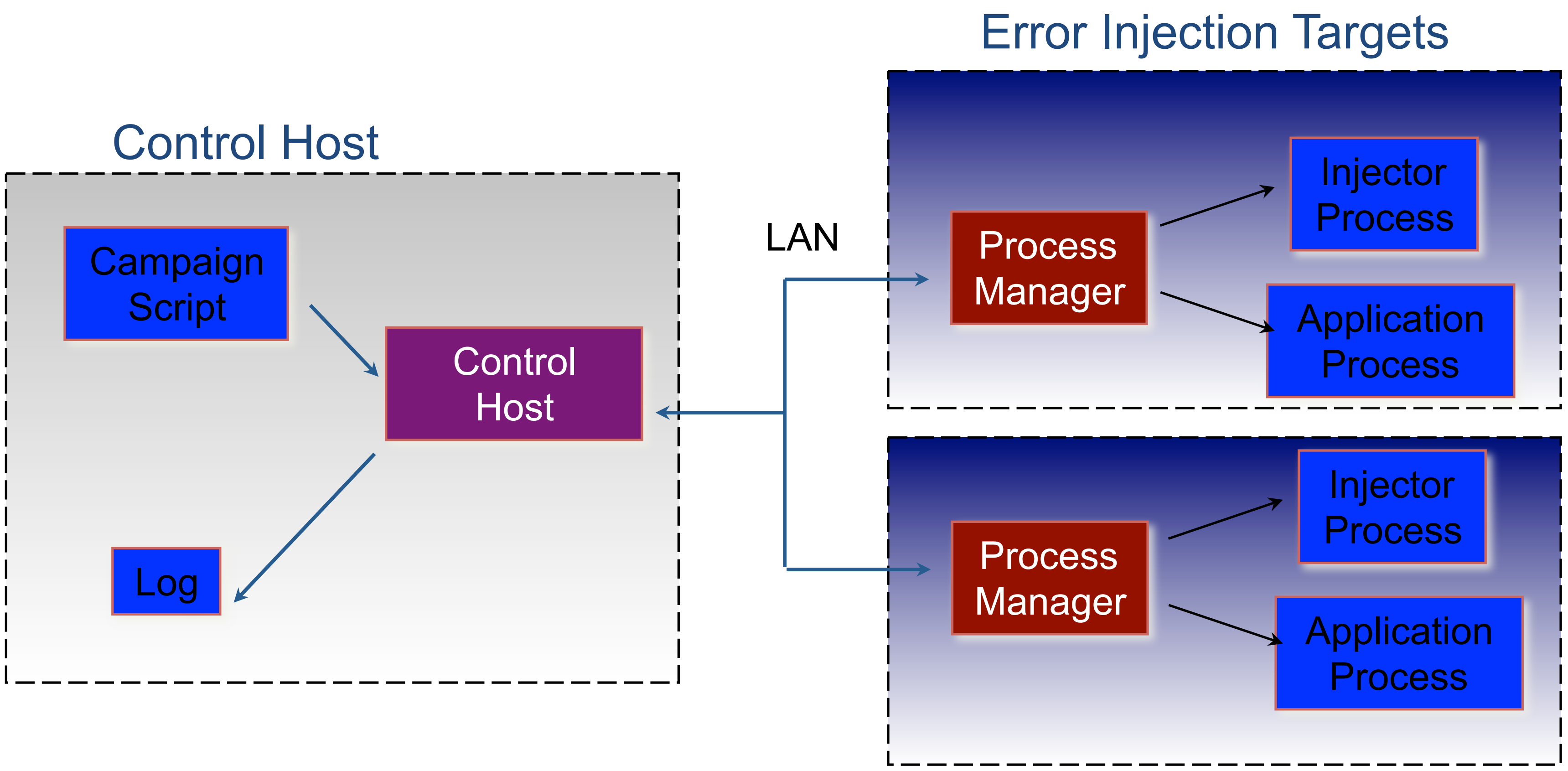
- Debugger-based fault injector
  - injection to the target process memory and registers
- Driver-based fault injector
  - injection to memory, registers, OS components
- Use of performance monitors (built into CPUs)
  - trigger fault injection; measure error latency
- Fault injection targets
  - CPU registers, memory, applications, specific OS functions
- Fault injection triggers:
  - random (based on time), application supplied breakpoint, externally supplied breakpoint

Injection Targets and Outcome Categories

Target	**User space**
Code	Functions: e.g., main Instructions: any or selected subset (e.g., branch, load, store)
Data	Static data and dynamically allocated memory (heap)
Stack	Data on an application stack
CPU Registers	General purpose registers
Memory range	Any location in application memory space

Outcome Category	**Description**
Activated	The corrupted instruction/data is executed/used.
Not Manifested	The corrupted instruction/data is executed/used, however it does not cause a visible abnormal impact on the system.
Fail Silence Violation	Either operating system or application erroneously detects the presence of an error or allows incorrect data/response to propagate out. Workload programs are instrumented to detect errors.
Crash	Application/OS stops working, e.g., bad trap or system panic. Crash handlers embedded into *OS* are enhanced to enable dump of failure data (processor and memory state).
Hang	System resources are exhausted resulting in a non-operational application/system, e.g., deadlock or livelock .

NFTAPE Framework Configuration



NFTAPE: Results

- Used to evaluate NASA's space imaging application: Part of MARS REE project

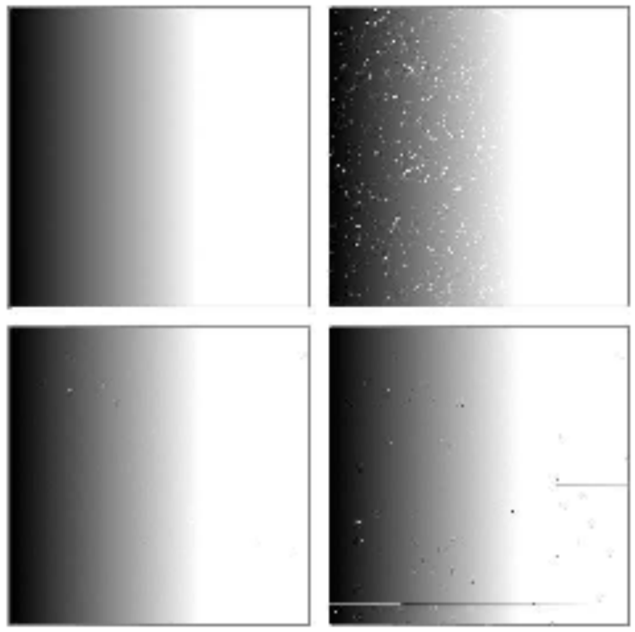
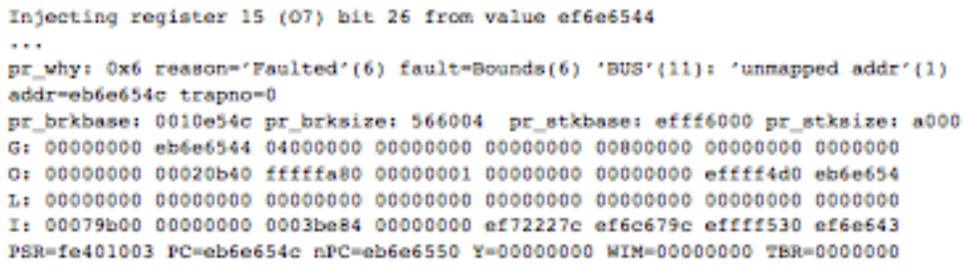


Figure 4. Images from Space Imaging Application.

Fault injections performed in memory

a) Regular image
b) Image with random noise
c) Application output with no faults
d) Application output with NFTAPE injecting faults

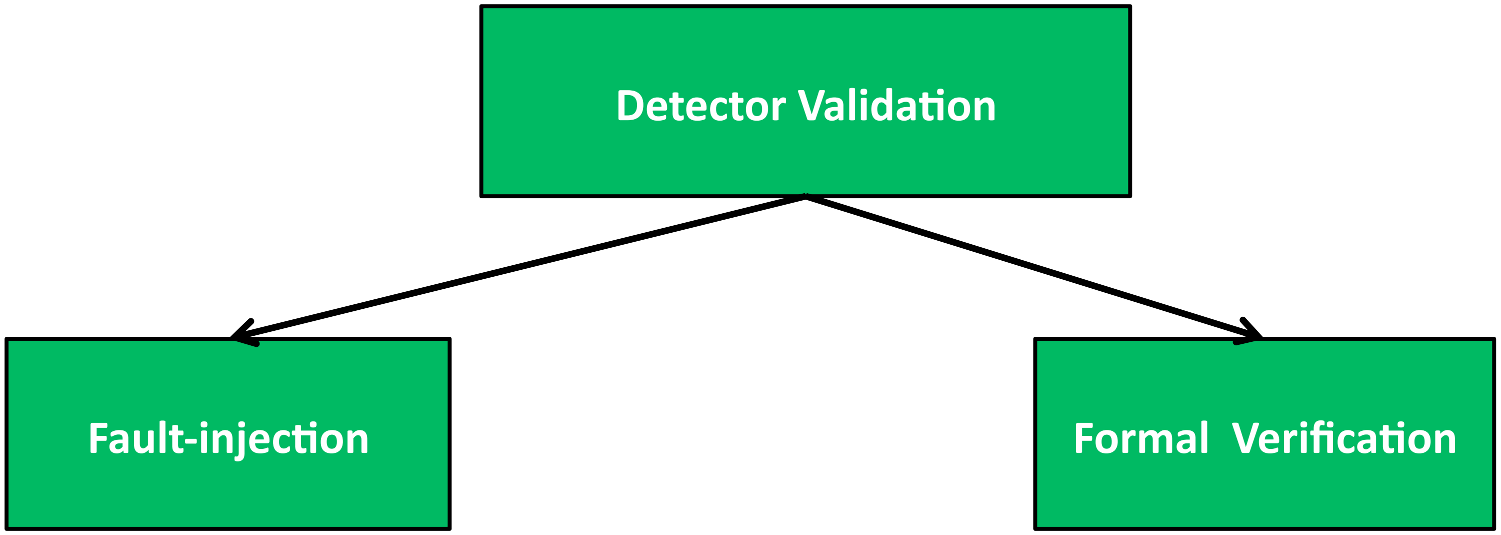


```
Injecting register 15 (O7) bit 26 from value ef6e6544
...
pr_why: 0x6 reason='Faulted'(6) fault=Bounds(6) 'BUS'(11): 'unmapped addr'(1)
addr=eb6e654c trapno=0
pr_brkbase: 0010e54c pr_brksize: 566004  pr_stkbase: efff6000 pr_stksize: a000
G: 00000000 eb6e6544 04000000 00000000 00000000 00800000 00000000 0000000
O: 00000000 00020b40 fffffa80 00000001 00000000 00000000 effff4d0 eb6e654
L: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 0000000
I: 00079b00 00000000 0003be84 00000000 ef72227c ef6c679c effff530 ef6e643
PSR=fe401003 PC=eb6e654c nPC=eb6e6550 Y=00000000 WIM=00000000 TBR=0000000
```

Learning Objectives

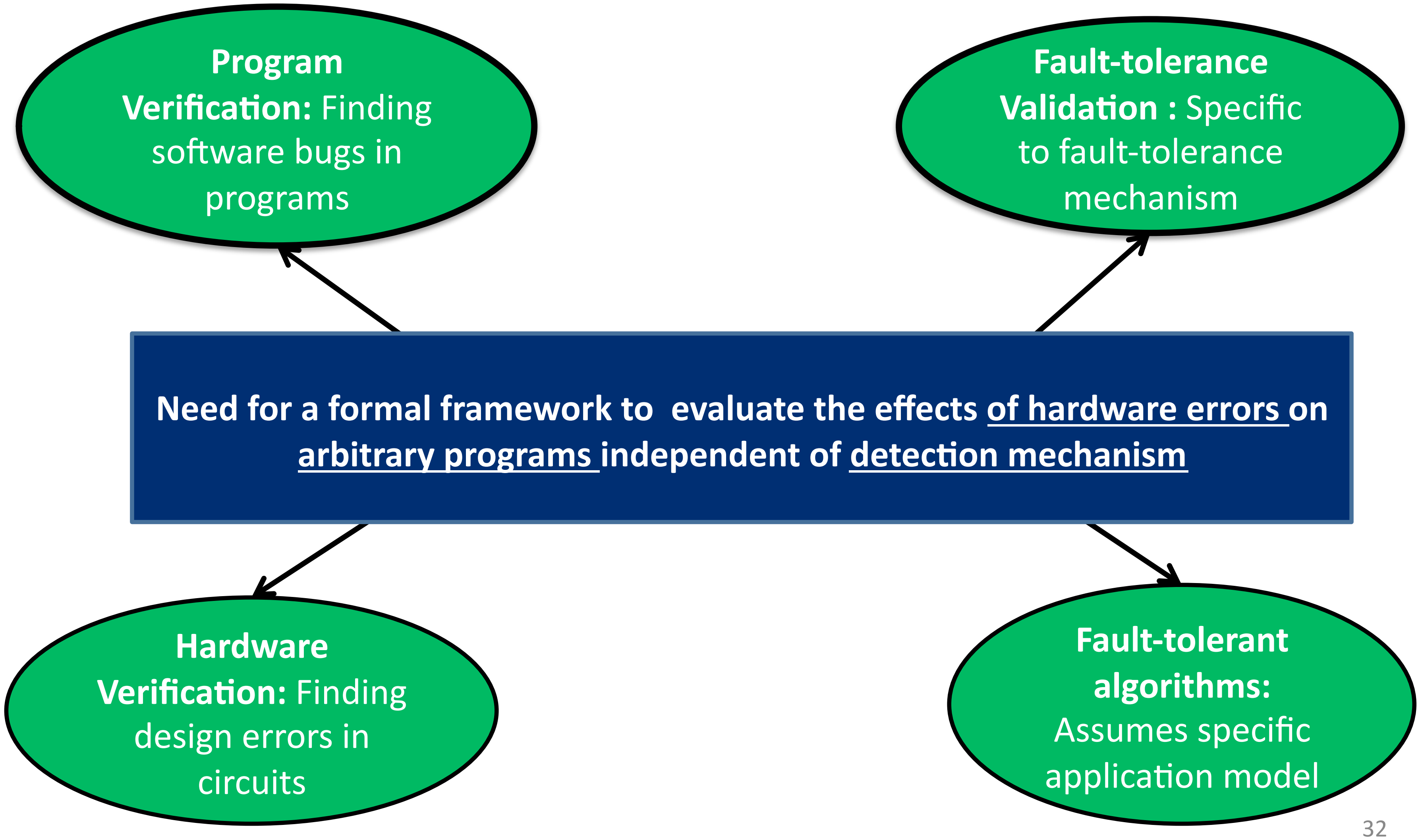
- Define fault-injection and explain its uses
- Design a fault-injection experiment for measuring reliability
- Apply software and hardware techniques for fault injection
- Apply formal techniques for the assessment of fault tolerance

Validation of FTMs

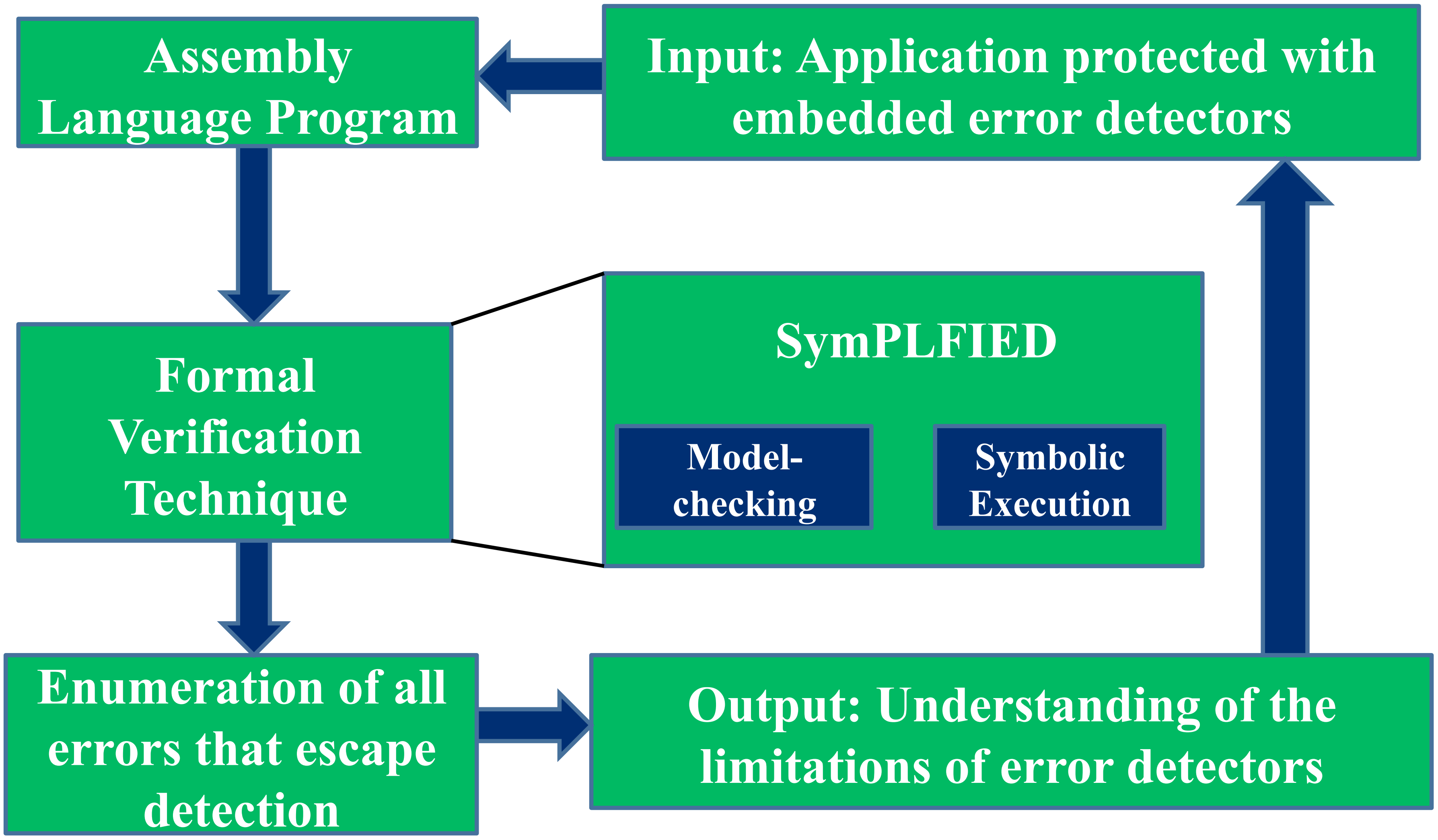


Formal Verification – Complements FI by exposing corner-case scenarios in error and attack detectors

SymPLFIED: Existing Techniques


Program Verification: Finding software bugs in programs
Fault-tolerance Validation : Specific to fault-tolerance mechanism
Need for a formal framework to evaluate the effects of hardware errors on arbitrary programs independent of detection mechanism
Hardware Verification: Finding design errors in circuits
Fault-tolerant algorithms: Assumes specific application model

Formal Framework for Detector Validation


Assembly Language Program
Input: Application protected with embedded error detectors
Formal Verification Technique
SymPLFIED
Model-checking
Symbolic Execution
Enumeration of all errors that escape detection
Output: Understanding of the limitations of error detectors

SymPLFIED : Approach

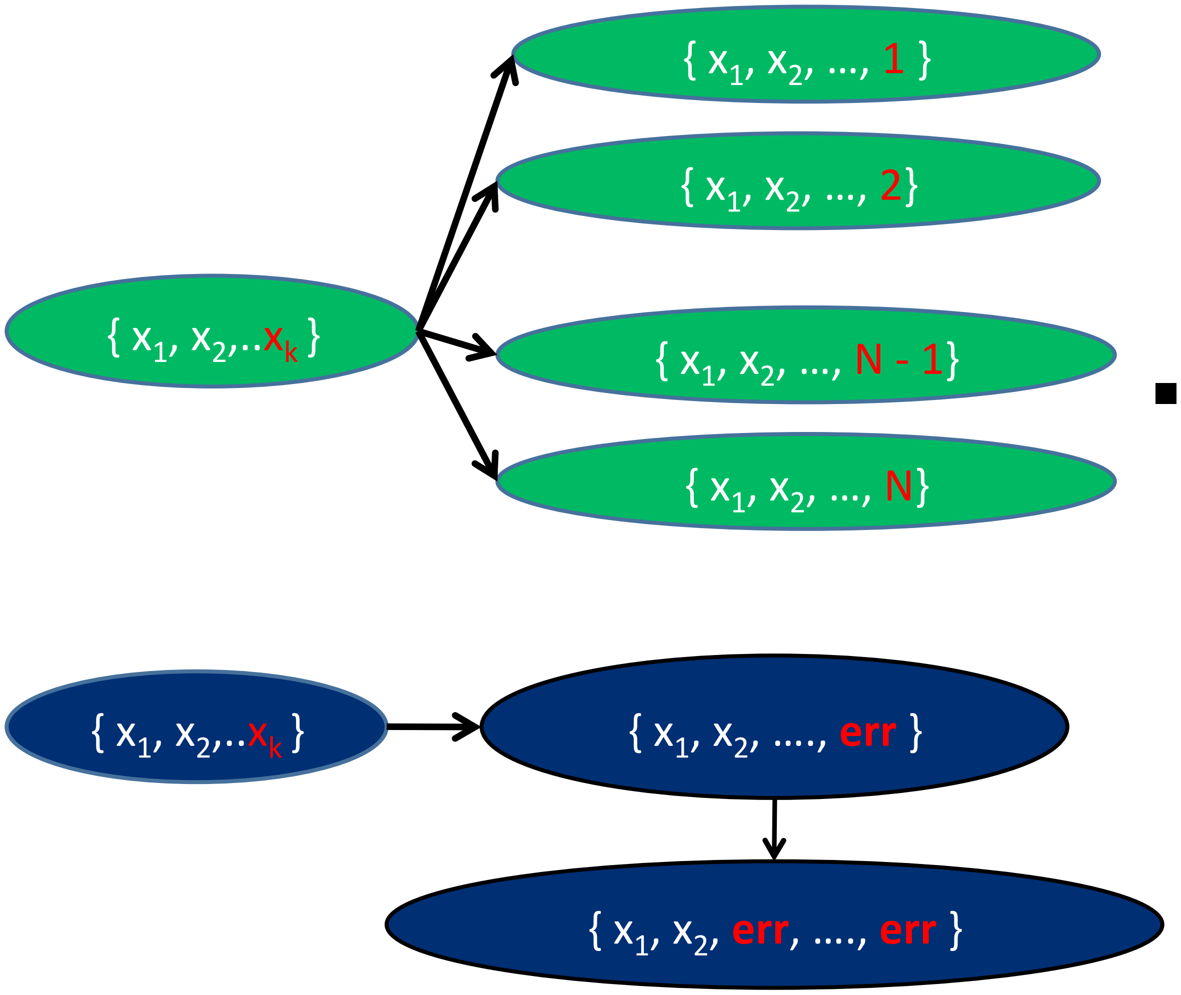
- **Analyze program written in MIPS assembly language**
  - Low-level state made explicit (e.g., stack pointer)
  - Oblivious to post compile-time transformations
  - Both programs and libraries (statically linked)

- **Generic representation of error detectors**
  - Allow arbitrary error detectors to be specified in program

- **Fault Model:** H/W transients (memory/register/PC)

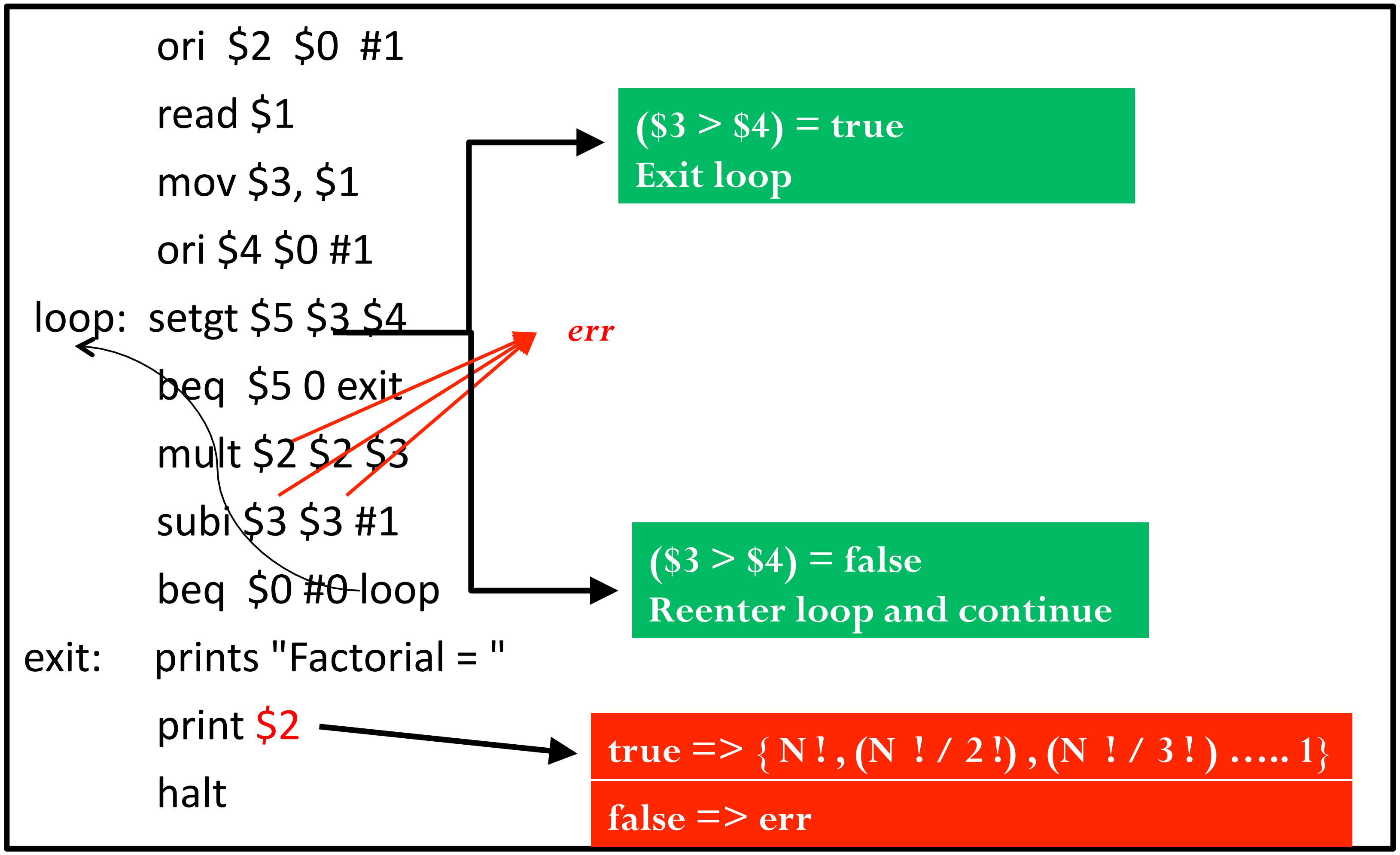
- **Comprehensive enumeration of undetected errors that lead to program failure → detector defects**

SymPLFIED: Symbolic Execution



- **Exhaustive enumeration leads to state space explosion**
- **Represent all error values in program as an abstract symbol**
  - Track propagation of errors symbolically
  - Abstraction may lead to false-positives

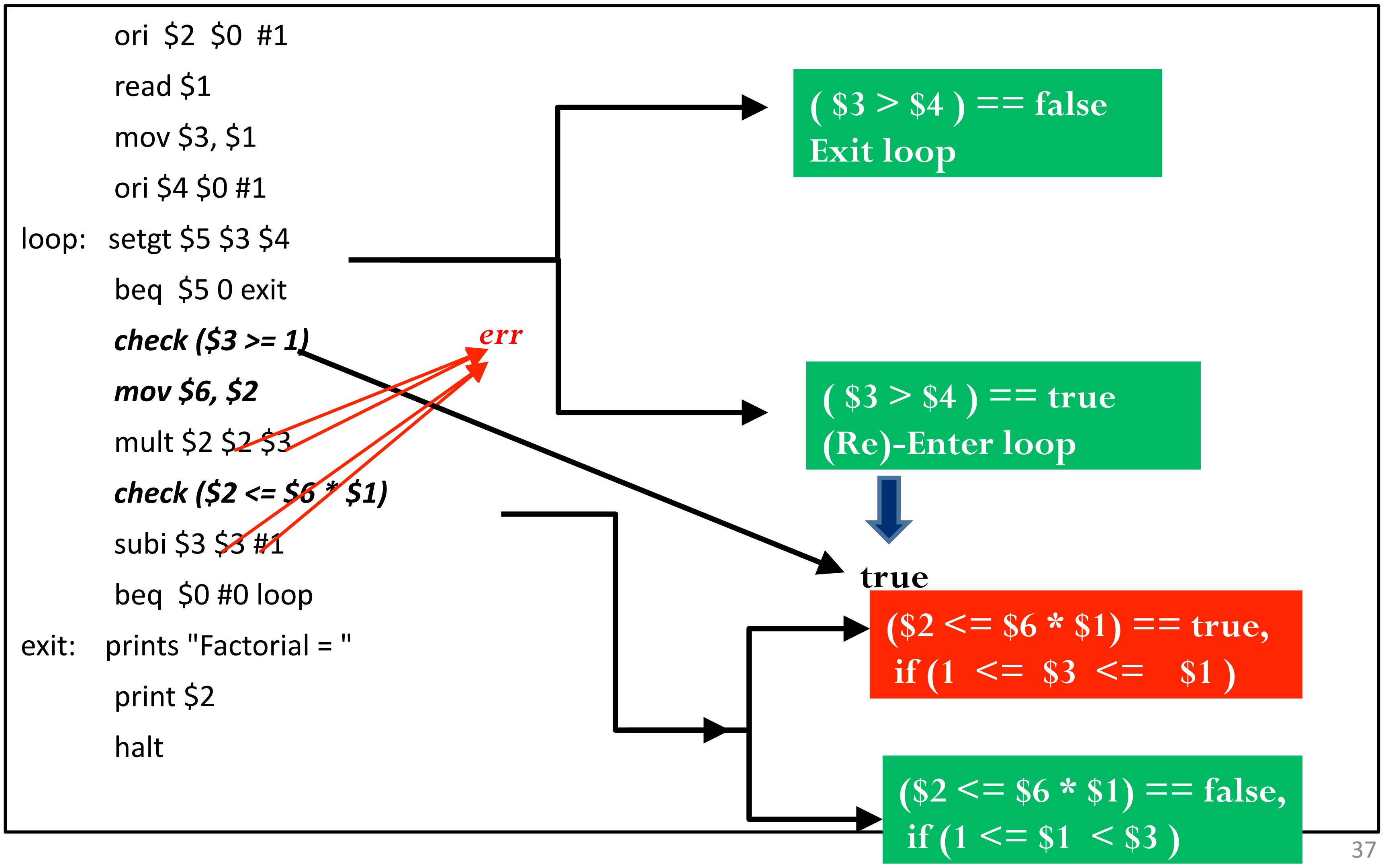
SymPLFIED: Propagation Example



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

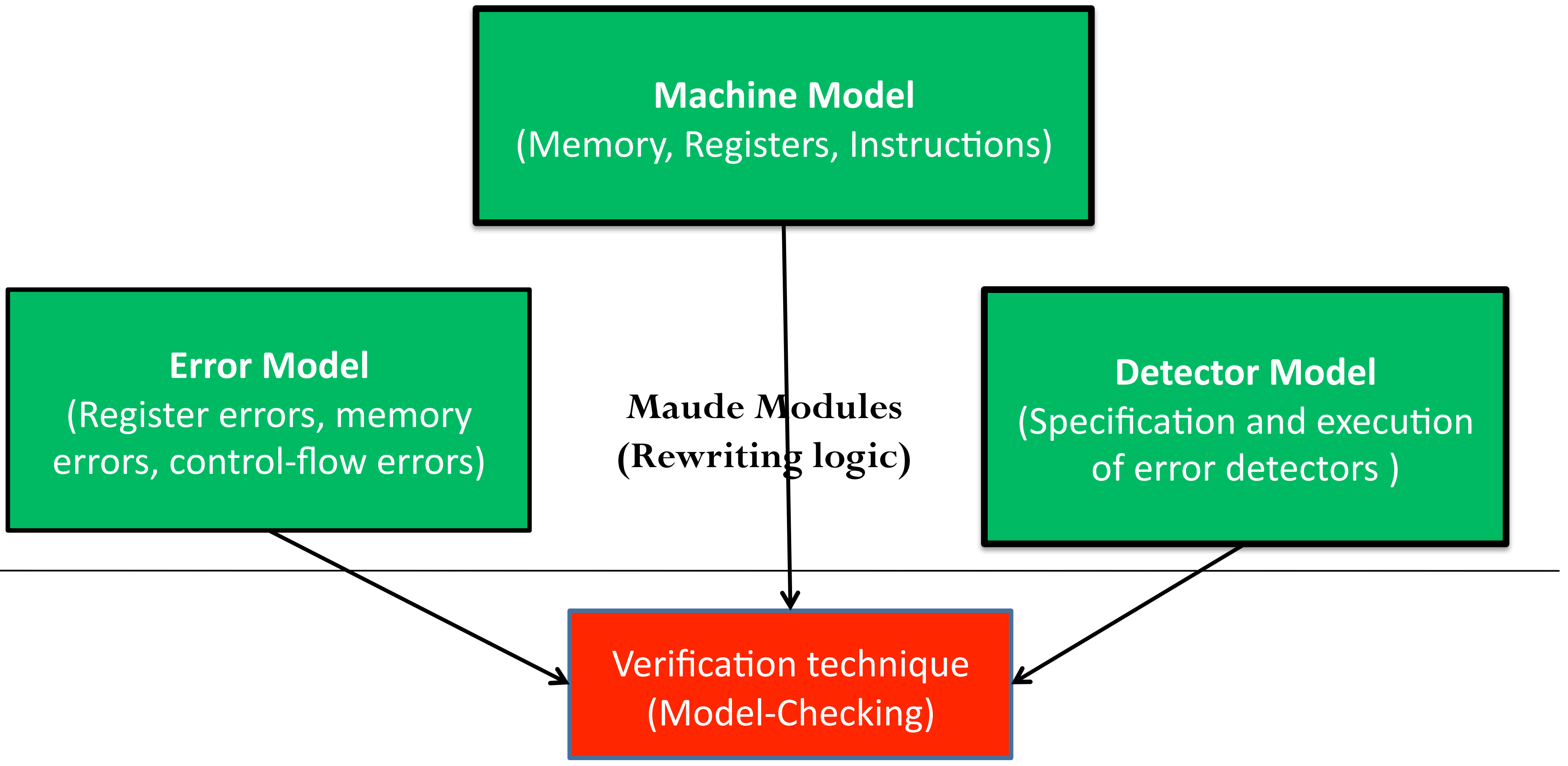
err
($3 > $4) = true
Exit loop
($3 > $4) = false
Reenter loop and continue
true => { N ! , (N ! / 2 !) , (N ! / 3 !) ….. 1}
false => err

SymPLFIED: Detection Example



```
       ori  $2  $0  #1
       read $1
       mov $3, $1
       ori $4 $0 #1
loop:  setgt $5 $3 $4
       beq  $5 0 exit
       check ($3 >= 1)
       mov $6, $2
       mult $2 $2 $3
       check ($2 <= $6 * $1)
       subi $3 $3 #1
       beq  $0 #0 loop
exit:  prints "Factorial = "
       print $2
       halt
```

SymPLFIED : Design


Machine Model
(Memory, Registers, Instructions)
Error Model
(Register errors, memory errors, control-flow errors)
Maude Modules
(Rewriting logic)
Detector Model
(Specification and execution of error detectors)
Verification technique
(Model-Checking)

Modular framework allows decoupling of detection mechanism and error class from the machine model and verification technique

SymPLFIED: Machine Model

eq { C, < addi rs rd v , **PC(pc) regs(R) S**> }

$\Downarrow$

{ C, < fetch(C, next(pc)),
PC(next(pc)) regs(R [rd <- (R[rs] + v)]) S >}

Machine State

eq next(pc) = (pc + 1) if not terminal(pc)
eq fetch([L | I] C, L) = I
eq fetch(NoCode, L) = throw instException

SymPLFIED: Error Model

- **Data Errors**
  - rl (Err == Int) => False
  - rl (Err == Int) => True

- Address Errors
  - rl M[Err] => anyAddr(M)
  - rl M[Err] => addrException

- Control Errors
  - rl fetch(C, err) => anyLabel(C)
  - rl fetch(C, err) => instException

SymPLFIED: Detector Model

Specified as quintuple:

Det(Id, pc, left expr, right expr)

Checks when program counter==pc if,

value(left expr) == value(right expr)

Detector Example

check(4,   10,   ! ($4) + *(#3),   @(#5))

Checks if (R[4] + M[3] == 5) when (pc==10)

If not, throw checkException(4)

SymPLFIED: Model Checking

Exhaustive search feature (bounded model checking):

```
search AllRegisterErrors ( start(program, first, input ) =>!
    (S:State) such that ( getOutput(S) contains 2)
                        and ( getException(S) == noException) .
```

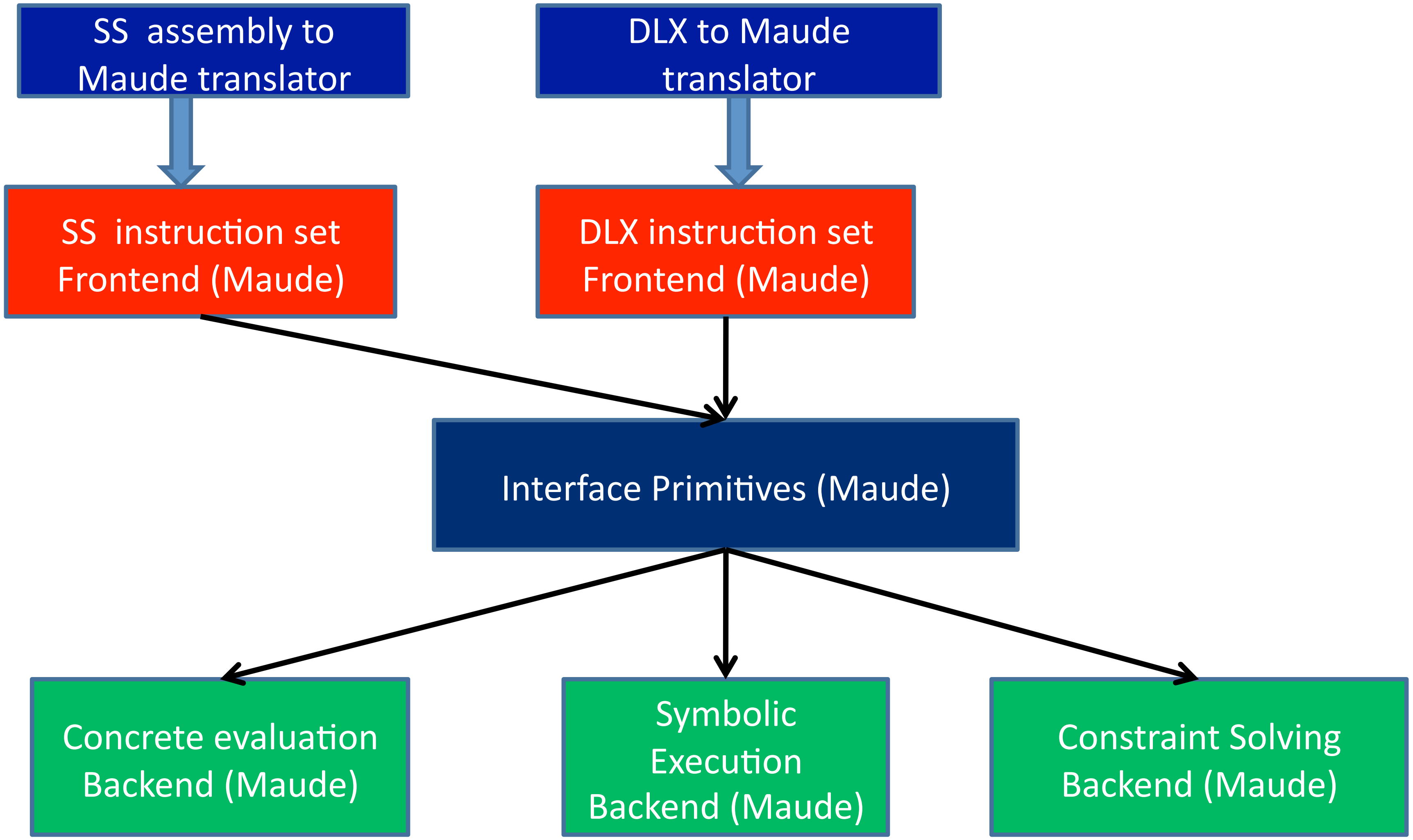
- **Timeout (no. of instructions) must be specified**
  - Obtained by profiling program with inputs

- **Input(s)/Output(s) must be specified for execution**
  - Comparison with golden output for determining failure

SymPLFIED: Implementation

- Written using a rewriting logic tool - **Maude**
  - Modular and extensible framework - code reuse
  - Supports direct execution of programs + libraries
  - Wide spectrum of formal techniques can be used
- **Architecture:** Consists of three main parts
  - **Front-end:** Models processor-specific details and interprets assembly language program
  - **Backend:** Performs symbolic/concrete evaluation of program under a given fault model
  - **Interface primitives**: Link front-end to back-end through a generic interface for easy extensibility

SymPLFIED: Implementation contd..


SS assembly to Maude translator
DLX to Maude translator
SS instruction set Frontend (Maude)
DLX instruction set Frontend (Maude)
Interface Primitives (Maude)
Concrete evaluation Backend (Maude)
Symbolic Execution Backend (Maude)
Constraint Solving Backend (Maude)

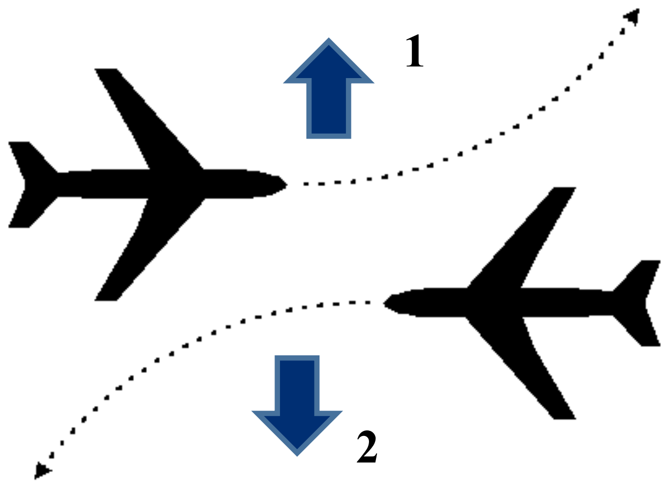
SymPLFIED: Output

- **States that lead to the error with info about**
  - Program Counter, Registers, Memory state
  - History of branches taken in program
  - Fault-injection and activation logs
  - State of input/output streams, exceptions
  - Constraints on register/memory values

- **Correlate with the code to trace how the error happened and improve detection mechanisms**

SymPLFIED: Case Study

- **Tcas: Application Characteristics**
  - FAA mandated Aircraft collision avoidance system
  - Rigorously verified protocol and implementation
  - About 150 lines of C code = 1000 lines of assembly



Inputs: Positional parameters of other aircraft (and self)

Outputs:

0 – Unresolved
1 – Ascend
2 - Descend

SymPLFIED: Results

- **Found one potentially catastrophic output considering all possible register errors**
  - Output of 2 (descend) instead of 1 (ascend)
  - Many cases where the output is unresolved (0)

- **Highly-parallelizable code of SymPLFIED**
  - Took about 4 minutes on a 150 node cluster
  - Total of 600 minutes (**10 hours**) of machine time

- **Not exposed by random fault-injection**
  - Used SimpleScalar simulator for experiments
  - Ran for more than **40 hours** on single machine

SymPLFIED: Tcas Error

int *alt_sep_test()*

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

```
    need_upward_RA = Non_Crossing_Biased_Climb() && Own_Below_Threat();
    …
    if (need_upward_RA && need_downward_RA)
       alt_sep = UNRESOLVED;
    else if (need_upward_RA)
       alt_sep = UPWARD_RA;
    else if (need_downward_RA)
       alt_sep = DOWNWARD_RA;
```

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

SymPLFIED: Summary

- **Formal framework to evaluate the effects of runtime errors on programs with detectors**
  - Analyze programs directly in assembly language
  - Comprehensive enumeration of failure-causing errors
- **Use of symbolic execution + model-checking**
  - Abstraction techniques to track error propagation
- **Tested on aircraft collision avoidance app.**
  - Found catastrophic error in main controller code
  - Not found using random fault-injection experiment

Learning Objectives

- Define fault-injection and explain its uses
- Design a fault-injection experiment for measuring reliability
- Apply software and hardware techniques for fault injection
- Apply formal techniques for the assessment of fault tolerance