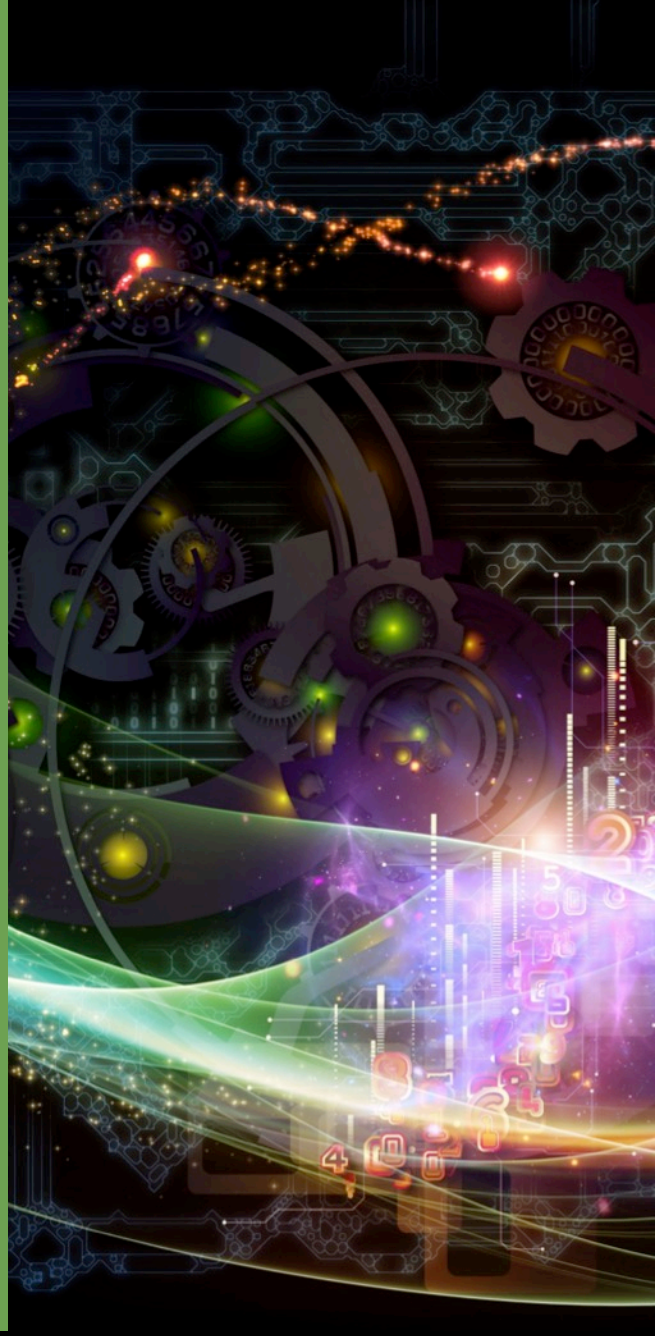
Software Solutions Symposium 2017

March 20–23, 2017

Testing in a Non-Deterministic World

Donald Firesmith

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213



© 2017 Carnegie Mellon University

[[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Copyright 2017 Carnegie Mellon University

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN “AS-IS” BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

[Distribution Statement A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.

This material may be reproduced in its entirety, without modification, and freely distributed in written or electronic form without requesting formal permission. Permission is required for any other use. Requests for permission should be directed to the Software Engineering Institute at permission@sei.cmu.edu.

DM-0004545

Topics

Testing

Determinism Assumptions

Trends

Non-Deterministic World

Testing Ramifications

Recommendations

© 2017 Carnegie Mellon University

[[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Software Solutions Symposium 2017

Testing in a Non-Deterministic World

Testing

© 2017 Carnegie Mellon University

[[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Testing – 1

The scope of this presentation is restricted to testing.

Testing compares a system's actual behavior with its expected behavior so that discrepancies (bugs) can be analyzed to uncover the underlying defects and thereby determine quality.

Testing involves:

- Establishing known test preconditions
- Providing known test inputs
- Comparing actual with expected test outputs
- Comparing actual with expected test postconditions

Discrepancies can be:

- Visible *failures* (incorrect visible behavior)
- Hidden *faults* (incorrect encapsulated mode, state, or data)

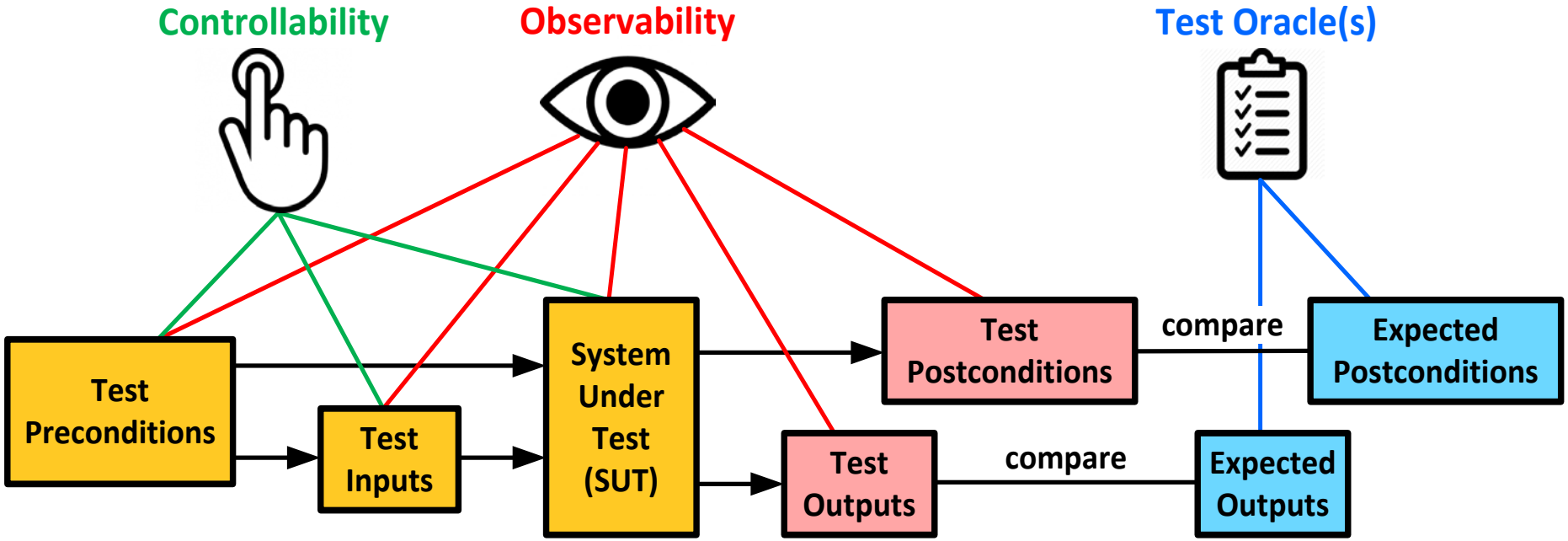
© 2017 Carnegie Mellon University

[[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Testing – 2

Testability (testing effectiveness and efficiency):

- *Controllability* to establish preconditions and create test inputs
- *Observability* to verify correctness of preconditions, inputs, outputs, and postconditions
- *Oracles* to provide expected behavior (e.g., requirements, design)



© 2017 Carnegie Mellon University
[[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Testing in a Non-Deterministic World

Determinism Assumptions

© 2017 Carnegie Mellon University

[[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Determinism Assumptions

Many developers assume:

- Systems and software are deterministic:
  - A system or software application will *always* behave in exactly the same way (i.e., same outputs and postconditions) when given identical inputs under identical preconditions.
- Test preconditions and inputs can be controlled and observed.
- Oracle provides single outcome (outputs and postconditions) given same preconditions and inputs.
- Tests are therefore repeatable.
  (i.e., The same test will always yield the same result.)

These assumptions are not always true, resulting in bugs that are:

- Rare
- Intermittent
- Difficult to reproduce, localize, and diagnose

Recent technology trends increase the likelihood that these assumptions are false.

© 2017 Carnegie Mellon University

[[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Testing in a Non-Deterministic World

Trends

© 2017 Carnegie Mellon University

[[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Trends

Agile and DevOps rely on continuous integration achieved via repeatable regression testing. This is only feasible with automated testing, which assumes deterministic behavior.

Increasing use of multicore processors and virtual machines increases concurrent processing and associated concurrency defects.

Increasing use of larger, more complex, systems of systems (SoSs) in turn increases concurrent system behaviors and concurrent system-to-system communications.

Increasing reliance on cyber-physical systems (including autonomous vehicles) increases non-deterministic environmental inputs (e.g., from sensors) and preconditions.

Increasing use of autonomous, adaptive, machine learning systems (e.g., search, fraud detection, security monitoring) constantly improves their behavior based on accumulated data.

© 2017 Carnegie Mellon University
[[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Testing in a Non-Deterministic World

Non-Deterministic World

© 2017 Carnegie Mellon University

[[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Types of Non-determinism – Intention

Intentional and valid non-determinism
(=> common, easy to produce events)

- Physical non-determinism (due to the physical environment, human actors, external systems, external networks)

Unintentional and invalid non-determinism
(=> rare, difficult to produce events)

- Concurrent non-determinism
  (due to system-internal and -external concurrency)
- Emergent non-determinism
  (due to integration of subsystems into systems)
- Exceptional non-determinism
  (due to fault and failure behavior)

© 2017 Carnegie Mellon University
[[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Types of Non-determinism – Reality

Actual non-determinism

Apparent non-determinism, which can be due to:

- Overwhelming complexity
- Inadequate controllability
- Inadequate visibility (hidden variables)
- Inadequate oracles

© 2017 Carnegie Mellon University
[[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Types of Non-determinism – Location

Test preconditions and inputs

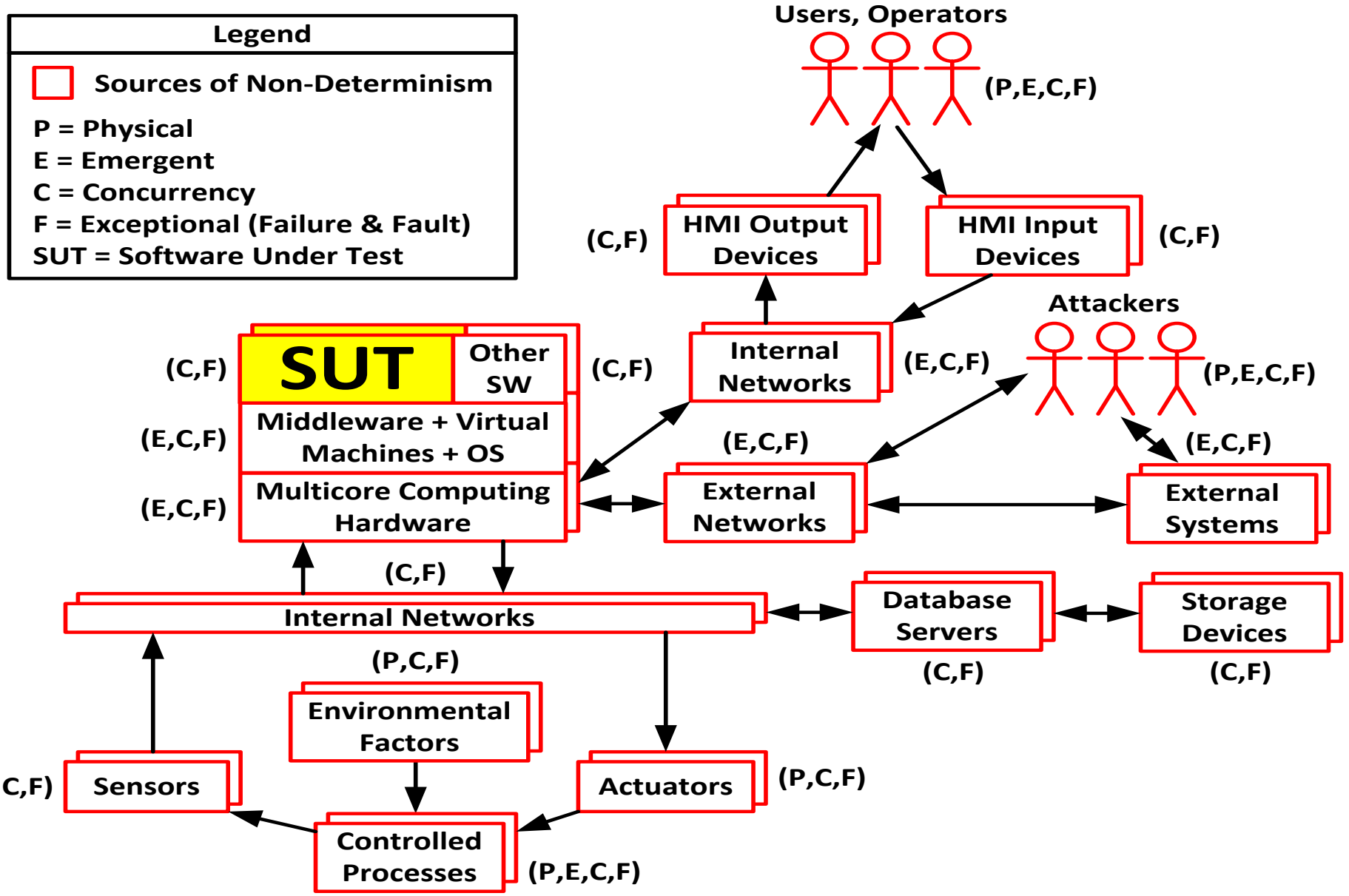
Test oracle

System including its subsystems and underlying technologies

System environment (physical, human actors, external systems, and networks)

© 2017 Carnegie Mellon University
[[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Sources of Non-Determinism



© 2017 Carnegie Mellon University
[[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Examples of Non-Deterministic Systems – 1

Cyber-physical systems:

- Autonomous and semiautonomous systems:
  - Robots
  - Self-driving vehicles
  - Swarms of drones (e.g., small UAVs)
- Distributed systems
- Systems of Systems (SoS) and federated systems
- Power generation and distribution
- Process control (e.g., chemicals, petrochemicals, medicines)
- Internet of Things (IoT)
- Mobile computing

© 2017 Carnegie Mellon University
[[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Examples of Non-Deterministic Systems – 2

Computing involving:

- Multicore processors
- Multiple processors (or computers or devices)
- Virtual Machines
- Multithreading and concurrent programming languages
- Cloud computing

© 2017 Carnegie Mellon University
[[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Non-Deterministic Environments

Environments may not be Deterministic:

- Development Environments
- Developmental Test Environments
- Operational Test Environments
- Physical Operational Environment

© 2017 Carnegie Mellon University
[[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Non-Deterministic Defects – 1

Developers and testers fail to adequately take the following into account when designing software and associated test suites:

- Classic Concurrency Defects:
  Deadlock, livelock, starvation, suspension, priority inversion, (data) race conditions, order violations, atomicity violations, lock and semaphore defects
- Multicore Defects:
  Interference between cores due to sharing common resources (e.g., L2/L3 caches, RAM and disc memory, I/O, system bus), improper allocation of SW to HW, unacceptable jitter in processing times
- Virtual Machine Defects:
  VM escapes and interference between VMs, improper allocation of SW to Virtual Machines, hypervisor defects

© 2017 Carnegie Mellon University
[[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Non-Deterministic Defects – 2

Performance Bugs:

- Missed deadlines (latency and response time)
- Unacceptable jitter
- Incorrect order

Reliability Defects:

- Buffer overflow, automatic garbage collection

Robustness Defects:

- Missing, inadequate, or incorrect error, fault, failure, and environmental tolerance

Security Defects:

- Vulnerabilities
- Defects in security controls (e.g., incorrect configurations)

© 2017 Carnegie Mellon University
[[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Non-Deterministic Defects – 3

Rare alignments of cyclic behaviors that result in intermittent and non-reproducible failures

Positive feedback under stress propagated through a system's environment triggering showstoppers or dangerous loss of control

Bizarre responses that are internally consistent but externally dangerous

© 2017 Carnegie Mellon University
[[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Related Issues – 1

Testing autonomous systems:

- Verify reasoning process
- Verify models match reality

Testing AI systems:

- Verify learning and adaptation

Fuzzy success criteria (boundary of correct behavior space):

- Pass, fail, partially pass/fail, unclear
- Stochastic pass/fail

Lack of oracle:

- Lack of verifiable requirements
- Excessively complex oracle

© 2017 Carnegie Mellon University
[[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Related Issues – 2

Emergent behavior:

- Unexpected unwanted behavior resulting from integration of subsystems into systems

Black swan events:

- Definition:
  - Rare (outlier),
  - Impact (big & negative)
  - Explainable (*after* the fact)
- Types:
  - Failures and faults
  - Accidents and near misses

© 2017 Carnegie Mellon University
[[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Testing in a Non-Deterministic World

Testing Ramifications

© 2017 Carnegie Mellon University

[[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Testing Ramifications – 1

Many developers and testers are not adequately familiar with non-deterministic defects.

- Many developers and testers do not know how to test for non-deterministic defects and unwanted emergent behavior.

Developers and testers do not design, implement, and execute tests designed to uncover non-deterministic defects:

- Testing less effective and efficient at detecting such defects.
- Increased number of false positive and false negative test results

Testing fails to uncover non-deterministic defects.

- Testing provides false confidence that such defects do not exist in the system
- Such defects escape into system operation where they can lead to intermittent and difficult to reproduce faults and failures.

© 2017 Carnegie Mellon University

[[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Testing Ramifications – 2

Non-deterministic software often requires more test automation:

- Failures and faults due to many defects in non-deterministic software (e.g., caused by concurrency and the use of virtual machines and multicore processors) will be rare.
  - Requires large suite of test cases to uncover rare bugs
- Cyber-physical systems with sensors, actuators, and physical environment may require modeling and simulation (M&S) for test preconditions and environmental inputs to obtain controllability and visibility.
  - Requires large suite of test cases to achieve adequate coverage

Non-deterministic software can make test automation challenging:

- Comparison of actual behavior to oracle
- More complex oracle (success criteria is a set, not an instance)
  - Oracle may be as complex as the system being tested.
  - Previous system version is less precise/accurate than new system
- May require manual determination of success/failure by SMEs
- Difficult to automate operational testing of the entire system, potentially as part of a system of systems (SoS)

© 2017 Carnegie Mellon University

[[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Testing in a Non-Deterministic World

Recommendations

© 2017 Carnegie Mellon University

[[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Recommendations – 1

Explicitly address non-determinism in test planning.

Provide training and mentoring in:

- Non-deterministic defects and how to test for them
- Probability and statistics

Because non-deterministic defects are difficult and expensive to uncover, concentrate your effort testing for them on mission- and safety-critical software.

Incorporate Built-In-Test (BIT), especially Continuous BIT (CBIT), Interrupt-Driven BIT (IBIT), and Periodic BIT (PBIT)

Program non-deterministic systems (especially autonomous systems) to be able to answer questions regarding *why they did what they did*.

Instrument non-deterministic elements so that logs can be scrutinized for rare timing and other anomalies.

© 2017 Carnegie Mellon University

[[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Recommendations – 2

Generate *large* suites of test cases:

- Ensure size of test suite is adequate to uncover rare faults and failures.
- Use statistical analysis when desired behavior is stochastic.
- Use combinatorial testing to achieve adequate coverage of combinations of conditions and of edge and corner cases.
- Do ***not*** rely on simple demonstrations of functional requirements (i.e., one test case per requirement).

Use modeling and simulation (M&S):

- Use M&S to control non-deterministic hardware and environmental inputs.
- Use very large numbers of simulation runs to detect rare events. Google simulates 3 million miles of autonomous driving per day.
- Verify models and simulations, which can also contain defects.

© 2017 Carnegie Mellon University

[[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Recommendations – 3

Ensure existence of verifiable quality requirements.

Perform specialty engineering testing of quality requirements:

- *Capacity testing* including time-varying load testing, stress testing, and volume testing
- *Performance testing* including latency, jitter, response time, and throughput
- *Reliability testing* including soak testing
- *Robustness testing* of rare exceptional situations
- *Safety testing* based on *hazard* analysis (STAMP, misuse cases, or FMECA)
- *Security testing* based on *threat* analysis (abuse cases, attack trees, attack surfaces)

Perform *concurrency testing* for concurrency bugs (e.g., race conditions, starvation, deadlock, livelock, priority inversions, and unwanted emergent behaviors due to concurrency).

© 2017 Carnegie Mellon University

[[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Recommendations – 4

Don’t just rely on testing for verification.

- Concurrency analysis of architecture and design.
  - Allocation of SW to threads, VMs, and cores
  - Potential interference paths between threads, VMs, and cores
  - Use of thread-safe class libraries
- Static and dynamic analysis
- Model checking and statistical model checking.

Create test suites that interleave high levels of realistic, high-risk, behavioral and rate variations.

Evaluate test results using general postcondition goals as oracles rather than irrelevant intermediate-behavior differences.

- Emphasize end-to-end mission thread operational testing.

© 2017 Carnegie Mellon University

[[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Recommendations – 5

Use “standard best practices” (modified as needed to take non-determinism into account):

- Continuous integration and testing
- Model-based testing (MBT)

Use a Test Asset Management System, and treat test assets as first class work products.

Unit Testing:

- Test all externally controlled and non-deterministic exceptions, failures, and aberrant behaviors with extreme points
- Code coverage and boundary values (including range checking)
- Add scalable run-time invariant checking, enable and check

Component Testing:

- Test every cause and every effect at least once, separately
- Test every “illegal” sequence for an appropriate response

Emphasize automation of unit and integration testing, which are more likely to be deterministic.

System Testing:

- Test crash, recovery, and restart under realistic conditions

© 2017 Carnegie Mellon University

[[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Contact Information

Presenter

Donald Firesmith

Principal Engineer

Telephone: +1 412.268.6874

Email: dgf@sei.cmu.edu

© 2017 Carnegie Mellon University

[[Distribution Statement A] This material has been approved for public release and unlimited distribution.

Contact Information

Donald G. Firesmith
Principal Engineer
Software Solutions Division
Phone: +1 412-268-6874
Mobile: +1 412-216-0658
Email: dgf@sei.cmu.edu

Web
www.sei.cmu.edu
www.sei.cmu.edu/contact.cfm

U.S. Mail
Software Engineering Institute
Customer Relations
4500 Fifth Avenue
Pittsburgh, PA 15213-2612
USA

Customer Relations
Email: info@sei.cmu.edu
Telephone: +1 412-268-5800
SEI Phone: +1 412-268-5800
SEI Fax: +1 412-268-6257

© 2017 Carnegie Mellon University
[[Distribution Statement A] This material has been approved for public release and unlimited distribution.