SEI Insights

**Home** > **SEI Blog** > The Challenges of Testing in a Non-Deterministic World

**SEI** Blog

The Latest Research in Software Engineering and Cybersecurity

# The Challenges of Testing in a Non-Deterministic World

POSTED ON JANUARY 9, 2017 BY **DONALD FIRESMITH** [/AUTHOR/DONALD-FIRESMITH] IN **TESTING** [HTTPS://INSIGHTS.SEI.CMU.EDU/SEI_BLOG/TESTING/]

By Donald Firesmith
Principal Engineer
Software Solutions Division

Many system and software developers and testers, especially those who have primarily worked in business information systems, assume that systems--even buggy systems--behave in a deterministic manner. In other words, they assume that a system or software application will *always* behave in exactly the same way when given identical inputs under identical conditions. This assumption, however, is not always true. While this assumption is most often false when dealing with **cyber-physical systems** [http://www.sei.cmu.edu/cyber-physical/], new and even older technologies have brought various sources of non-determinism, and this has significant ramifications on testing. This blog post, the first in a series, explores the challenges of testing in a non-deterministic world.

**Testing: The Scope of This Post**

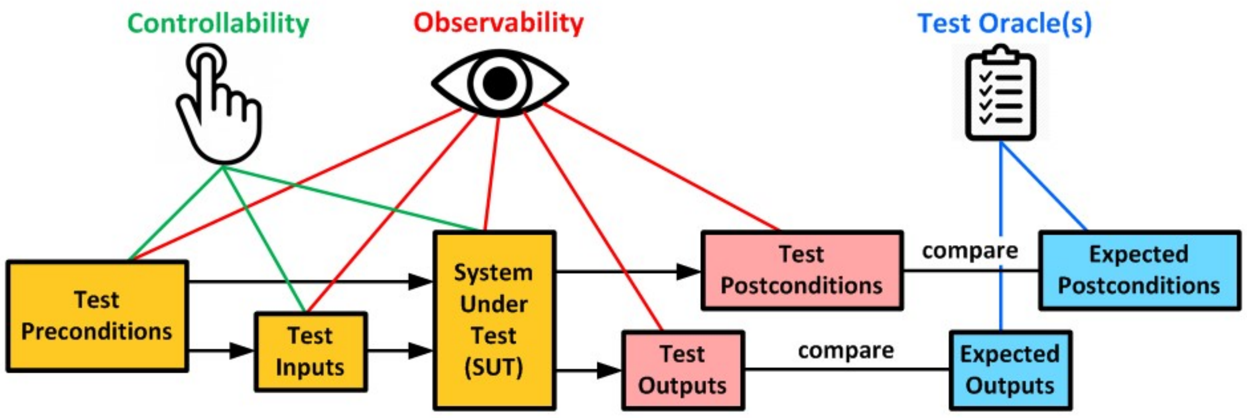
Before I get into these ramifications, it is important to clarify the scope of this post, which is strictly testing. Even though other verification-and-validation approaches (such as static and dynamic analysis, inspection, and review) are also useful and relevant when verifying non-deterministic systems and software, I will not address them here because testing is more than sufficient for multiple blog postings.

Specifically, 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. Note that these discrepancies can be visible *failures* (incorrect visible behavior), as well as hidden *faults* (incorrect encapsulated mode, state, or data). Testing involves

- establishing known test preconditions
- providing known test inputs
- comparing actual with expected test outputs
- comparing actual with expected test postconditions

As shown in the following figure, testability (i.e., the degree to which testing can be effective and efficient) requires meeting the following three prerequisites:

- *controllability* to establish the test preconditions and create test inputs
- *observability* to verify the correctness of the test preconditions, inputs, outputs, and postconditions
- one or more test ***oracles*** (e.g., requirements, architecture, design) to provide the expected behavior



## Tester Assumptions Regarding Non-Determinism

With the scope of this post clarified, we can now address the tester's assumption that systems and software behave in a deterministic manner and the ramifications of this incorrect assumption. Many developers and testers assume that tests are always repeatable because

- System and software behaviors (specifically, the test outputs and postconditions) are deterministic.
- They think that they are able to control (and observe) the test preconditions and inputs.
- The test oracle provides only a single outcome (outputs and postconditions) for any given set of test preconditions and inputs.

In many modern domains, however, tests will not always yield the same result, and this makes testing much harder. Finding the resulting bugs that are rare, intermittent, and hard to reproduce, localize, and diagnose is the new challenge.

## Trends Affecting Non-Determinism

The following recent technology trends also increase the likelihood that these assumptions are false:

- **Agile** and **DevOps** rely on continuous integration achieved via repeatable regression testing. This approach is only feasible with automated testing, which is typically performed assuming deterministic behavior.
- Increasing use of multicore processors, virtual machines, and multi-threaded languages and frameworks 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 (and modifies) their behavior based on accumulated data.

## Types of Non-Determinism

So far, I have been intentionally vague when it comes to the meaning of the words "non-determinism" and "non-deterministic." While I do not intend to use mathematical precision, a little more clarity would be useful.

First of all, non-determinism can be actual or only apparent. Actual non-determinism occurs when there is no *theoretical* way of predetermining the system's exact behavior, such as behavior that is determined by quantum physics (e.g., using nuclear decay to generate truly random numbers). Apparent non-determinism occurs when there is no *practical* way for the tester (or test oracle) to predetermine the system's exact behavior. This apparent non-determinism can be due to overwhelming complexity or to inadequate controllability, visibility, and oracle(s).

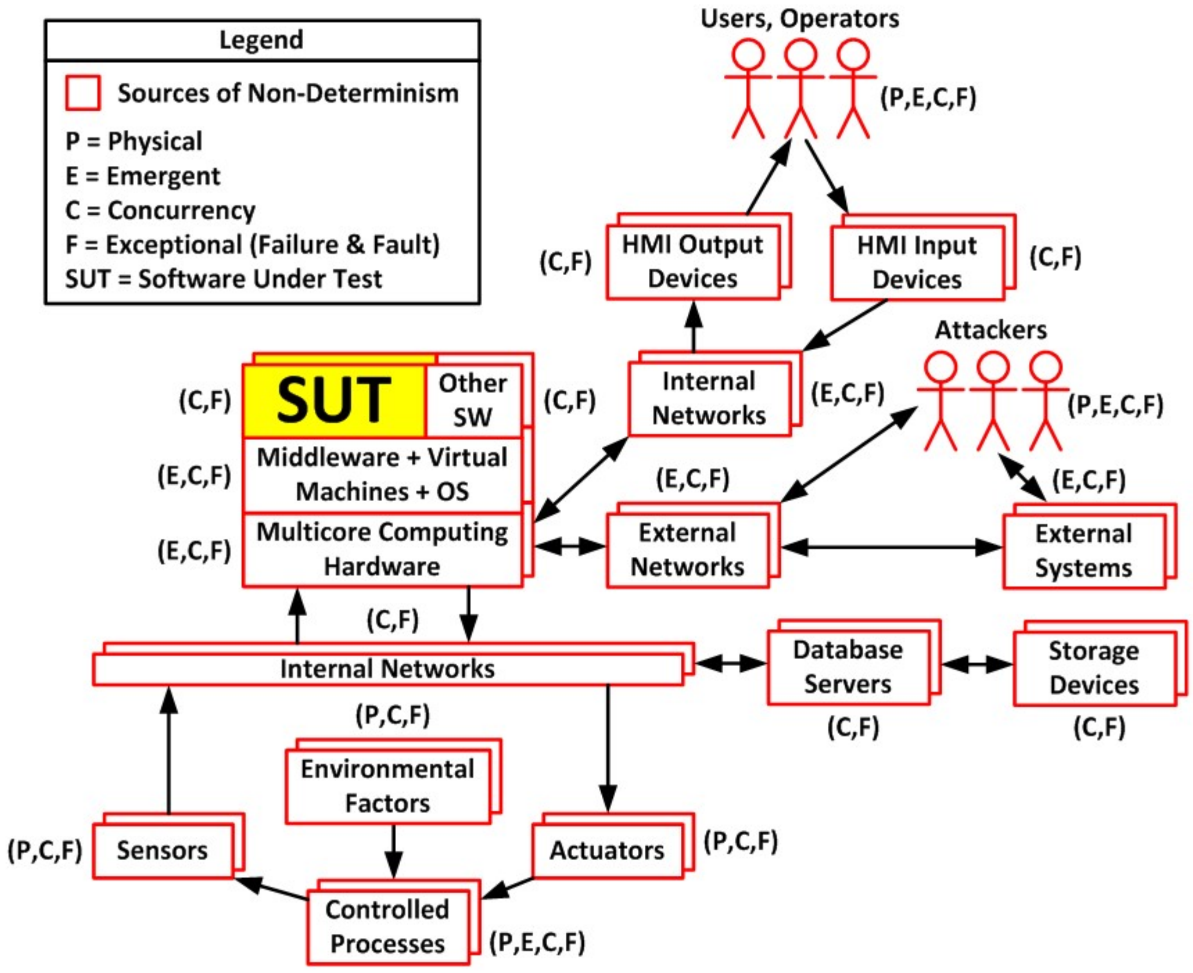
From a practical point of view, both actual and apparent non-determinism have the same impact on testability. With non-deterministic systems and software, you can run the exact same test case (i.e., with the exact same test inputs under the exact same test preconditions) multiple times and get different results (i.e., different test outputs and test postconditions). Running a single test case only once is insufficient to determine whether the test case truly passes or fails. Unfortunately, requirements-driven testing is too often performed using only a single test case per requirement. This practice is insufficient, even when not contending with non-determinism.

Testers must be aware of and take into account four types of non-deterministic behavior based on the source of that behavior:

- physical non-determinism (due to the nature of physical reality)
- emergent non-determinism (due to integration of subsystems into systems)
- concurrent non-determinism (due to system-internal and -external concurrency)
- exceptional non-determinism (due to fault and failure behavior)

As shown in Figure 2 below, testers also must address non-determinism that occurs in

- test inputs and preconditions
- the system and its subsystems including their underlying technologies such as sensors, actuators, software, and computing platforms (hardware, OS, and infrastructure)
- a system's environment, including its physical environment, human actors, external systems, and networks



## Examples of Non-Deterministic Systems

Non-deterministic systems are ubiquitous. As mentioned previously, they are found in all manner of cyber-physical systems:

- autonomous systems (e.g., robots and vehicles)
- Internet of Things (IoT)
- mobile computing
- power generation and distribution
- process control (chemicals, petrochemicals, medicines)
- systems of systems (SoS), federated systems

Non-deterministic systems are common when dealing with modern hardware and software technologies, such as **multicore processors** [https://en.wikipedia.org/wiki/Multi-core_processor] , multiple processors (or computers or devices), **virtual machines (VMs)** [https://en.wikipedia.org/wiki/Virtual_machine], **multithreading and concurrent programming languages** [https://insights.sei.cmu.edu/sei_blog/2014/10/thread-safety-analysis-in-c-and-c.html], and **cloud computing** [http://www.sei.cmu.edu/sos/research/cloudcomputing/clouduse.cfm].

Testers may also find non-deterministic behavior in the development environments, development and operational test environments, and the physical operational environment.

**Non-Deterministic Defects**

Testing is largely about uncovering defects. When dealing with systems susceptible to non-deterministic behavior, developers and testers should pay special attention to the testing for the following types of non-deterministic defects when designing software and the associated test suites:

- **concurrency defects:**
  - **classic defects**, such as **deadlock**, **livelock**, **starvation**, suspension, **priority inversion**, (**data) race conditions**, order violations, **atomicity violations**, and **lock and semaphore** defects
  - **multicore defects**, such as interference between cores due to sharing common resources (e.g., L2/L3 caches, RAM and disc memory, I/O, system bus), improper allocation of software to hardware components, and unacceptable jitter in processing times
  - **virtual machine defects**, such as VM escapes and interference between VMs, improper allocation of SW to VMs, and hypervisor defects
- **performance defects** that cause missed deadlines (latency and response time), unacceptable jitter, and incorrect order of processing and events
- **reliability defects** that cause buffer overflow and bugs due to ill-timed automatic garbage collection
- **robustness defects** that cause missing, inadequate, or incorrect error, fault, failure, and environmental tolerance
- **security defects** including vulnerabilities and defects in security controls (e.g., incorrect configurations)
- **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

## Related Issues

The following related challenges often crop up when testing non-deterministic systems.

- verifying the reasoning process and whether models match reality when testing autonomous systems
- verifying learning and adaptation when testing AI systems
- fuzzy success criteria (boundary of correct behavior space) when:
  - The results of the test may include not just pass and fail, but also partially passed and indeterminate.
  - Stochastic pass/fail criteria may be used.
- lack of an adequate oracle (for example, lack of verifiable requirements and excessively complex oracle)
- emergent behavior (i.e., unexpected unwanted behavior resulting from the integration of subsystems into systems and systems into systems of systems)
- **black swan events** (i.e., events that are rare outliers, have a large negative impact, and are only explainable after the fact). Such events include not just normal failures and faults, but also accidents and near misses (for **safety-critical systems**).

## Testing Ramifications of Non-Determinism

Many developers and testers would benefit from additional training in non-deterministic defects and how to test them. Developers and testers do not design, implement, and execute tests designed to uncover non-deterministic defects, making testing less effective and efficient at detecting them. As a result, testing fails to uncover non-deterministic defects resulting in an increase in false positive and false negative test results. Testing provides false confidence that such defects do not exist in the system, and they escape into system operation where they can lead to intermittent and hard to reproduce faults and failures.

Non-deterministic software often requires more test automation due to the need for large suites of test cases:

- 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 and require large numbers of test cases to uncover.
- 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. Testing these types of systems also requires large suites to achieve adequate coverage of combinations of conditions and of edge and corner cases.

Non-deterministic behavior makes test automation challenging:

- It may be hard to compare the actual behavior to the oracle's expected behavior.
- The oracle is often more complex when the system is non-deterministic. For example, the success criteria may be a set of acceptable behavior rather than a single specific behavior. The oracle may even be as complex as the system being tested (for example, when there is no specification, only a previous version of the system that may be less precise or accurate than the new system).
- It may be that the only way to determine success is through manual test execution and validation of the results by subject matter experts.
- The non-determinism may only show up during operational testing of the entire system (for example, as part of a system of systems).
- To obtain adequate controllability and visibility, you may need to use simulation/modeling rather than operational environment for generating test preconditions and environmental inputs.

## Conclusion

We live and work in a non-deterministic world, and that has significant ramifications on how we need to test our systems and software. In the next post, I will provide recommendations for addressing the challenges associated with testing non-deterministic systems and software.

## Additional Resources

View my January 2016 webinar ***A Taxonomy of Testing Types*** [http://www.sei.cmu.edu/webinars/view_webinar.cfm?webinarid=450418].

View my podcast ***Common Testing Problems: Pitfalls to Prevent and Mitigate*** [http://www.sei.cmu.edu/podcasts/podcast_episode.cfm?episodeid=57568].

Read **my previously published blog posts on testing** [http://insights.sei.cmu.edu/author/donald-firesmith/].

## About the Author

### Donald Firesmith

**Contact Donald Firesmith** [https://www.sei.cmu.edu/contact.cfm]
Visit the SEI Digital Library for **other publications by Donald** [http://resources.sei.cmu.edu/library/author.cfm?authorID=4637]
View **other blog posts by Donald Firesmith** [/author/donald-firesmith]