SEI Insights

Home > **SEI Blog** > Seven Recommendations for Testing in a Non-Deterministic World

SEI Blog

The Latest Research in Software Engineering and Cybersecurity

Seven Recommendations for Testing in a Non-Deterministic World

POSTED ON APRIL 24, 2017 BY **DONALD FIRESMITH** [/AUTHOR/DONALD-FIRESMITH] IN **SOFTWARE AND INFORMATION ASSURANCE** [HTTPS://INSIGHTS.SEI.CMU.EDU/SEI_BLOG/SOFTWARE-AND-INFORMATION-ASSURANCE/]

By Donald Firesmith
Principal Engineer
Software Solutions Division

In **a previous post** [https://insights.sei.cmu.edu/sei_blog/2017/01/the-challenges-of-testing-in-a-non-deterministic-world.html] , I addressed the testing challenges posed by **non-deterministic systems** [http://www.webopedia.com/TERM/N/nondeterministic_system.html] and software such as the fact that the same test can have different results when repeated. While there is no single panacea for eliminating these challenges, this blog posting describes a number of measures that have proved useful when testing non-deterministic systems.

Preparing for Non-deterministic Testing

Logically, one of the first things one should do is to properly prepare for the testing of non-deterministic systems and software applications. Based on your initial understanding of the amount, types, and criticality of non-deterministic behaviors, you should explicitly address non-determinism in your test planning, including both contractor and government test plans and the testing parts of their system- and software-development plans. The relevant testers and their management should be trained and mentored in non-deterministic defects and how to test for them, while members of the government program office should also be trained in how to address non-determinism during the oversight of contractor testing. Due to the need for large test suites to uncover rare failures, everyone involved should have the necessary level of training in probability, statistics, and related testing methods, such as **design of experiments** [https://en.wikipedia.org/wiki/Design_of_experiments], **combinatorial testing** [http://mse.isri.cmu.edu/software-engineering/documents/faculty-publications/miranda/kuhnintroductioncombinatorialtesting.pdf], **Monte Carlo methods** [https://en.wikipedia.org/wiki/Monte_Carlo_method], **accelerated life testing** [https://en.wikipedia.org/wiki/Accelerated_life_testing], and **soak testing** [https://en.wikipedia.org/wiki/Soak_testing].

Integrating Testing into the System

As discussed in my previous posting, non-determinism can lead to hard to uncover, localize, and diagnose defects. Testing should therefore be built into the software. This testing could include architecting in a **prognostics and health management (PHM) subsystem** [http://ieeexplore.ieee.org/document/4161628/], **assertions** [https://en.wikipedia.org/wiki/Assertion_(software_development)], including pre- and post-conditions as well as invariants, a high level of associated **exception handling** [https://en.wikipedia.org/wiki/Exception_handling], and various types of **built in testing (BIT)** [http://www.dtic.mil/dtic/tr/fulltext/u2/a069384.pdf] into the system's software, especially continuous BIT (CBIT), interrupt-driven BIT (IBIT), and periodic BIT (PBIT). Testing could also include instrumenting non-deterministic elements of the architecture so testers can scrutinize logs for associated rare timing and other anomalies. Finally, because machine learning can lead to designs that are very hard to understand during the localization and diagnosis of non-deterministic defects, you can program such systems to be able to answer questions regarding *why they did what they did*. For work on verifying robotic behavior, see my colleague Stephanie Rosenthal's blog post, ***Why Did the Robot Do That?*** [https://insights.sei.cmu.edu/sei_blog/2016/12/why-did-the-robot-do-that.html]

Generating Large Suites of Test Cases

Many classes of non-deterministic defects only rarely produce faults and failures. To uncover such defects, it is often necessary to have very large test suites. Creating such large test suites must be automated based on requirements, architecture, and design models. To achieve adequate test coverage, create test suites that interleave high levels of realistic, high-risk, behavioral and rate variations.

While randomly generated inputs (e.g., **fuzz testing** [https://en.wikipedia.org/wiki/Fuzzing]) can increase test coverage, they are nevertheless unlikely to capture all combinations of factors and evaluate all corner cases. Design of experiments and combinatorial testing can help to lower the number of test cases while maintaining reasonable test coverage. When the desired behavior (i.e., the **test oracle** [https://en.wikipedia.org/wiki/Oracle_(software_testing)]) is **stochastic** [https://en.wikipedia.org/wiki/Stochastic] , it is important to (1) evaluate test results using general goal-based test post-conditions rather than intermediate-behavior differences and (2) use statistical analysis to determine the minimum number of test cases to obtain a given minimum probability of uncovering defects.

Simple requirements-based **functional testing** [https://en.wikipedia.org/wiki/Functional_testing] will be inadequate to find rare outcomes caused by non-deterministic behavior, and simple demonstrations of functional requirements (i.e., one test case per requirement) will be woefully inadequate.

Augmenting Testing with other Forms of Verification

While appropriate testing is almost always necessary to uncover non-deterministic defects, it is also typically not adequate by itself. Testing should be augmented with **static and dynamic analysis** [https://software.intel.com/en-us/node/622647] , as well as **modeling and simulation (M&S)** [https://en.wikipedia.org/wiki/Modeling_and_simulation] . This augmentation includes concurrency analysis of the architecture and design taking into account (1) the allocation of software to threads, virtual machines, and cores in a multicore processing environment, (2) identifying potential and actual interface paths involving resources shared between threads, VMs, and cores, and (3) using thread-safe class libraries. Likewise, traditional and statistical **model checking** [https://en.wikipedia.org/wiki/Model_checking] is also useful to apply. Please read more about the SEI's work **using model checking and statistical model checking (as opposed to testing) to verify distributed adaptive real-time (DART) systems such as small UAVs** [https://insights.sei.cmu.edu/sei_blog/2016/10/verifying-distributed-adaptive-real-time-systems.html] .

Performing Specialty Engineering Testing

Testing for non-deterministic defects often involves the performance of specialty engineering types of testing. Before you can test for the existence of non-deterministic defects related to the quality characteristics and attributes, you need to ensure that the associated verifiable quality requirements exist. An appropriate level of the following forms of specialty engineering testing is likely to be necessary:

- *capacity testing* including time-varying load testing, stress testing, and volume testing
- *concurrency testing* for concurrency bugs (e.g., race conditions, starvation, deadlock, livelock, priority inversions, improper orderings, and unwanted emergent behaviors due to concurrency)
- *performance testing* for latency, jitter, response time, and throughput
- *reliability testing* including soak testing
- *robustness testing* of rare exceptional situations including crashes, recovery, and restart
- *safety testing* based on hazard analysis (**Systems-Theoretic Accidents Model and Processes (STAMP)**, **misuse cases**, or **failure mode, effects, and criticality analysis (FMECA)**)
- *security testing* based on threat analysis (abuse cases, attack trees, attack surfaces)

Automating Testing

Address non-determinism when planning and performing test automation. Where practical, automatically generate large test suites to ensure adequate coverage and use code coverage tools to verify test coverage. Unit and low-level integration testing, which are more likely to be deterministic, can be automated in a traditional manner, but should also test boundary values and include range checking.

Using Standard Best Practices

When modifying testing to uncover non-deterministic defects, it is important to follow traditional testing best practices that also help with non-determinism. Perform **continuous integration and testing** [https://www.slideshare.net/cygnetinfotech/continous-integration-testing-fundamentals] and **model-based testing** [https://en.wikipedia.org/wiki/Model-based_testing], augmented as needed to address non-determinism. Emphasize end-to-end mission thread operational testing to uncover defects that hide in the interfaces between use cases, functions, and subsystems. Use a test asset management system, and treat test assets as first-class work products.

When performing unit testing, test all externally controlled and non-deterministic exceptions, failures, and aberrant behaviors with extreme points. When testing larger components, test every cause and every effect at least once and test "illegal" sequences of inputs for appropriate responses.

Wrapping Up

We live and work in a non-deterministic world, which has significant ramifications on how we need to test our systems and software that interact with and control portions of that world. Although no single panacea exists for addressing the many testing challenges due to non-determinism, there is much that we can do much to mitigate the risks associated with testing non-deterministic systems. It is time for developers and testers to up their game with regard to testing in a non-deterministic world.

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]