Software Fault Tolerance: Making Software Behave

Jeffrey Voas, Cigital

Between the late 1960s and early 1990s, the software engineering community strove to formalize schemes that would lead to perfectly correct software. Although a noble undertaking at first, it soon became apparent that correct software was, in general, unobtainable. And furthermore, the costs, even if achievable, would be overwhelming.
Modern software systems, even if correct, can still exhibit undesirable behaviors as they execute. How? Well, the simplest example would be if a software system were forced to experience an input it should not have. In this situation, the software could

1. handle it gracefully by ignoring it,
2. execute on it and experience no ill effects, or
3. execute on it and experience catastrophic effects.

Note that 1 and 2 are desirable, but 3 is not, yet in all cases, the software is still correct.

This issue’s focus is dedicated to the research results and ideas from a group of experts who discuss their views on how to create fault-tolerant software—that is, software that is designed to deliberately resist exhibiting undesirable behaviors as a result of a failure of some subsystem. That subsystem could be part of the software, an external software or hardware failure, or even a human operator failure. Generally speaking, fault-tolerant software differs from other software in that it can gracefully handle anomalous internal and external events that could lead to catastrophic system consequences. Because correct software is an oxymoron in most cases and, as I just mentioned, correct software can still hurt you, software fault tolerance is one of the most important areas in software engineering.

The first article, “Using Simplicity to Control Complexity,” by Lui Sha begins by discussing the widely held belief that diversity in software constructions entails robustness. The article then questions whether this is really true. It goes on to investigate the relationship between software complexity, reliability, and the resources available for software development. The article also presents a forward recovery approach based on the idea of “using simplicity to control complexity” as a way to improve the robustness of complex software systems.

Karama Kanoun’s article analyzes data collected during the seven-year development of a real-life software system. The software under consideration comprised two diverse variants. For each development phase, Kanoun evaluated the cost overhead induced by the second variant’s development with respect to the principal variant’s cost. The results concluded that the cost overhead varies from 25 to 134 percent according to the development phase.

Les Hatton’s article, “Exploring the Role of Diagnosis in Software Failure,” builds on the premise that software systems have, among engineering systems, the unique characteristic of repetitive failure. His article explores various reasons for this situation, particularly poor diagnosability. Hatton argues that this cause exists largely because of educational problems. Through examples, the article highlights the need for an attitude change toward software failure and for improved diagnostics. Finally, he introduces the concepts of diagnostic distance and diagnostic quality to help with categorization.

Michel Raynal and Mukesh Singhal’s article deals with ways to overcome agreement problems in distributed systems. The authors focus on practical solutions for a well-known agreement problem—the non-blocking atomic commitment.

Finally, William Yurcik and David Doss’s article addresses software aging. The article discusses two approaches to this problem:

- provide a system with the proactive capability of reinitializing to a known reliable state before a failure occurs or
- provide a system with the reactive capability of reconfiguring after a failure occurs such that the service provided by the software remains operational.

The authors discuss the complementary nature of these two methods for developing fault-tolerant software and give the reader a good overview of the field in general.

So in conclusion, I hope that after you read these articles you will have a better understanding of the underlying principles of software fault tolerance. All systems need defensive mechanisms at some point. These articles, along with the references in the Roundtable (see p. 54), provide information on how to get started.

About the Author

Jeffrey Voas is a cofounder and chief scientist of Cigital. His research interests include composition strategies for COTS software, software product certification and warranties, and software quality measurement. He coauthored Software Fault Injection: Inoculating Programs Against Errors (Wiley, 1998) and is working on Software Certificates and Warranties: Ensuring Quality, Reliability, and Interoperability. He received his BS in computer engineering from Tulane University and his PhD in computer science from the College of William & Mary. He was the program chair for the Eighth IEEE International Conference on Engineering of Computer-Based Systems. He was named the 1999 Young Engineer of the Year by the District of Columbia Council of Engineering and Architectural Societies, was corecipient of the 2000 IEEE Reliability Engineer of the Year award, and received an IEEE Third Millennium Medal and an IEEE Computer Society Meritorious Service award. He is a senior member of the IEEE, a vice president of the IEEE Reliability Society, and an associate editor in chief on IEEE Software. Contact him at jvoas@cigital.com.