Fault-Tolerant Software

Farokh B. Bastani
Department of Computer Science
University of Houston
Houston, TX 77204-3475

It is widely accepted that most operational software systems have some residual faults. They provide useful services in spite of their latent faults and several simple strategies can be used to cope with most failures. If the software does not damage any input data, then it may be possible to work around failures. This is true for faults in formatting programs and compilers where one can often re-express the input to bypass the faulty code. For other programs, such as payroll processing systems and billing systems, manual actions may be needed to correct errors. A more complicated situation occurs when the program updates some data, such as editors and database systems. Some strategies that can be used in these cases include the use of journals and implicit checkpoints, as in versioning systems, or explicit checkpoints. When a software failure is detected, the software must be debugged and the state restored to the most recent correct version. The success of this approach requires strict integrity checks to ensure that the journal and checkpoint are correct. Since the code for keeping journals is relatively straightforward, it is feasible to ensure high reliability in the correctness of a journal. This is not always true for checkpoints since a fault in the software (for example, a bug in the code coupled with a bug in the acceptance test) may result in erroneous checkpoints. In many interactive systems, it can be assumed that the user will perform the necessary integrity checks on a new checkpoint before deleting an existing checkpoint.

The most difficult situation occurs with real-time safety-critical systems. The applicable methods here range from design diversity to rigorous development, verification, and validation. For isolated systems, such as a program that controls the operations of a chemical plant, a rigorous approach is preferable. This would involve reusing software to the greatest extent possible since the reliability of old code is generally higher than that of new code. A powerful approach for achieving software reusability is to use program transformation to systematically incorporate well-understood features in a program, such as generating code for user-interfaces, defensive programming, tolerating hardware failures, improving the performance, and so on. For the remainder of the program, which will typically be highly application-specific, the probability of obtaining higher reliability may depend on the extent to which one can account for unexpected situations. If the input space of the program is viewed as being composed of a number of partitions organized in the form of a tree, with "well-understood" input scenarios near the root and "unexpected" input scenarios far from the root, then the reliability of the program can be enhanced by devoting all efforts to developing one version. This will allow a more in-depth exploration of the partition structure than several lower cost versions that may only be able to explore near the root area. Methods that can be used include the use of massive random testing to reduce the possibility of rare errors.

Another approach for dealing with unexpected situations is to restrict the operating regime of the program. This is particularly important for safety-critical systems. However, it entails identification of unsafe states and the existence of actions that will lead to corresponding safe states. For example, a system for driving a car may pull over to the shoulder in the event of an unexpected situation. Similarly, a traffic signal controller may switch to displaying red lights when a failure occurs. In other cases, such as a system for controlling an aircraft, where the system must continue to operate in spite of a failure, a back-up reliable version, perhaps an older version, that may operate in a degraded mode would be necessary. This is a limited form of design diversity.

Design diversity or systems using randomized algorithms are appropriate for applications where multiple copies of the system operate together with a high potential for interaction, such as network routing software, vehicle control systems, etc. In these cases, there is the possibility that using identical, deterministic copies may render the system susceptible to catastrophic cascade failures, i.e., the failure of one copy may cause the failure of another copy and so on. For example, imagine a program that controls a car and suppose there is a bug in avoiding an obstacle. Then, if several cars use the same program, they may all encounter the same bug with devastating consequences. Hence, in these types of situations, it is preferable to ensure uncorrelated reactions to the same situation to limit the consequences of failures.

In summary, the most cost-effective method of achieving fault-tolerant software depends on the impact of failures, the type of software, and the application. A combination of checkpointing, logging, design diversity, software reusability, self-checking capabilities, and rigorous development are needed for coping with unexpected failures.