N91-13343

Improving System Reliability through Formal Analysis and Use of Checks in Software

Mark E. Staknis
Computer Systems Engineering
Northeastern University

Software is playing increasingly important roles in avionics systems. It is widely used in navigation and, in some cases, in control loops that maintain aircraft stability. To guarantee the safety of flight systems, the FAA requires that critical components have a probability of failure no greater than 10^{-9} per hour of flight. The FAA suggests that a system should continually check itself to determine whether system components have failed so that appropriate responses to failure can be made. Software is being used to diagnose system components for failure. SIFT (Software Implemented Fault Tolerance) was a computer system developed to study the use of software to check for failure and manage processor reconfiguration [Wen].

While software can play a useful role in checking the status of sensors and hardware components, critical software must itself be ultrareliable. Many approaches are currently in use to improve software reliability. They include using structured design techniques, compilers, static analysis, and designing fault tolerance into the software. Although these approaches can improve software reliability, none can guarantee with a high level of confidence that the software satisfies its specifications.

To guarantee that software satisfies its specifications, formal verification can be used. With this a program and its specification are viewed as mathematical objects, and a mathematical proof is used to show that the program and its specification are equivalent. Since its initial study in the late 1960's, great strides have been made in formal verification. Semi-automated theorem proving systems now exist that monitor and assist in software verification [Hen, Ody].

Two important caveats come with formal verification. The first is that a verified program is only as good as its specification. If the specification does not adequately reflect system needs, then its software implementation will not either. The second caveat is that a verified program will produce correct results only when the program is invoked with specified input. A program executed upon unspecified or illegal input will produce unspecified results, and in critical systems this can lead to tragedy. Even verified software must check input for legality and respond appropriately if illegal input is detected.

In previous research [Sta], a theory of checking was developed to offer assistance in analyzing specifications and designing run-time checks. The theory is well suited for integration with a formal approach to software specification and verification. In the theory, checking is considered abstractly in terms of n-ary relations much like those of relational database theory. Such relations provide an ideal representation of software specifications. Within the theory checks are categorized, checks on input and checks on results are considered, and formal attention is given to the minimization and logical combination of checks. The theory consists of a framework of definitions and theorems for reasoning about checking.

The focus of this summer's research has been upon input checks and the obstacles in checking input to critical systems. A central concern, particularly for flight-critical software, is with a property referred to as independence. The concern is with circumstances under which it is possible to apply isolated, independent checks to separate sensor inputs and be assured that all illegal input will be properly detected. The problem can be stated in terms of a group of blind men inspecting an animal. When can the blind men, each inspecting a separate part of the animal and exchanging no information, guarantee that the animal *is not* an elephant? The answer can be explained easily when specifications (of software input or of elephants) are viewed as n-ary relations. A relation can be checked by isolated checks only if it possesses the property of independence.

When possible, input should be specified to be independent so that legality can be checked easily. However, in real-time systems such as flight control systems that interact with the physical environment, dependence appears to be determined to a large extent by the environment and cannot be "specified away." As a consequence, it may be very difficult to check input for safety or legality.

Presently, we are investigating independence and checking in the context of the GCS (Guidance and Control System) [Wit]. The GCS is a simulation of the Viking Mars Lander environment. The simulator is intended for testing software that implements control laws for landing the spacecraft. The lander input comes from sensors, internal parameters, and data saved from previous frames. The large number of inputs and their complex interrelationships provide an exciting context in which to investigate independence and the difficulties of supplying input checks. At this stage we are focusing on independence of sensor input. Later we hope to study the larger problem of all system input.

(FAA) Federal Aviation Administration, "System Design and Analysis," Advisory Circular No. 25.1309-1A, June 21, 1988

(Hen) von Henke, F. and J. Rushby, "An Introduction to Formal Specification and Verification using EHDM," SRI Projects 8110 and 8200-110, March 23, 1990

(Ody) *Penelope: Collected Papers in Ada Verification*, vol. 1, Odyssey Research Associates, Inc., Ithaca, New York, 1990

(Sta) Staknis, M.E., "A Theoretical Basis for Software Fault Tolerance," Ph.D. dissertation, Dept. of Computer Science, University of Virginia, 1987

(Wen) Wensley, J. *et al.*, "SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control," *Proc. IEEE*, vol. 60, pp. 1240-1254, Oct. 1978

(Wit) Withers, B.E., D.C. Rich, D.S. Lowman, and R.C. Buckland, "Software Requirements," NASA Contractor Report 182058, June 1990