SNEAK ANALYSIS AS A SOFTWARE RELIABILITY IMPROVEMENT TOOL

The development and introduction of complex, highly integrated electrical/electronic and microprocessor based systems into new products poses a major challenge to reliable system operation. Sneak Analysis identifies and corrects reliability-robbing design conditions, called sneaks, that frequently evade detection by traditional analysis and testing procedures.

Sneak conditions are latent design conditions which have unknowingly been incorporated in system designs. Sneak conditions can cause an undesired event to occur or prevent a desired event from happening. Sneak Analysis was originally developed by Boeing in 1967, with NASA funding, to evaluate electrical circuitry. Subsequent Boeing funded improvements have extended the technology to cover computer software and complex designs that integrate hardware and software. Boeing has applied the technology in over 250 projects for commercial, NASA, Department of Defense, and Boeing customers. In these projects, nearly 5,000 sneak conditions have been identified and corrected resulting in cost savings of 100's of millions of dollars through avoided loss of systems, project delays, rework, warranty claims, and recalls.

Sneak Analysis improves software reliability by the early identification of software problems. Sneak Analysis does not provide a software reliability measurement, but the identification and correction of software problems early in the design process is reflected in measurements taken with traditional methods. In the only controlled experiment known to Boeing, less than 40% of the sneak conditions identified by Sneak Analysis were also discovered by the traditional approaches. Whereas testing usually first reveals only the symptoms of the problem, Sneak Analysis reveals the root cause at the outset. Therefore, Sneak Analysis can provide reliability improvements before problems are reported as software "bugs". The effectiveness of Sneak Analysis in finding software problems is illustrated in figure 1.

Some of the problems found by Sneak Analysis are subtle, well-hidden conditions that may only appear under certain sets of operating conditions. These types of problems are difficult to detect by traditional methods, and even once detected are even more difficult to diagnose. They are sometimes labeled as transient events and their seriousness is not understood until much later, often in the operational phase.

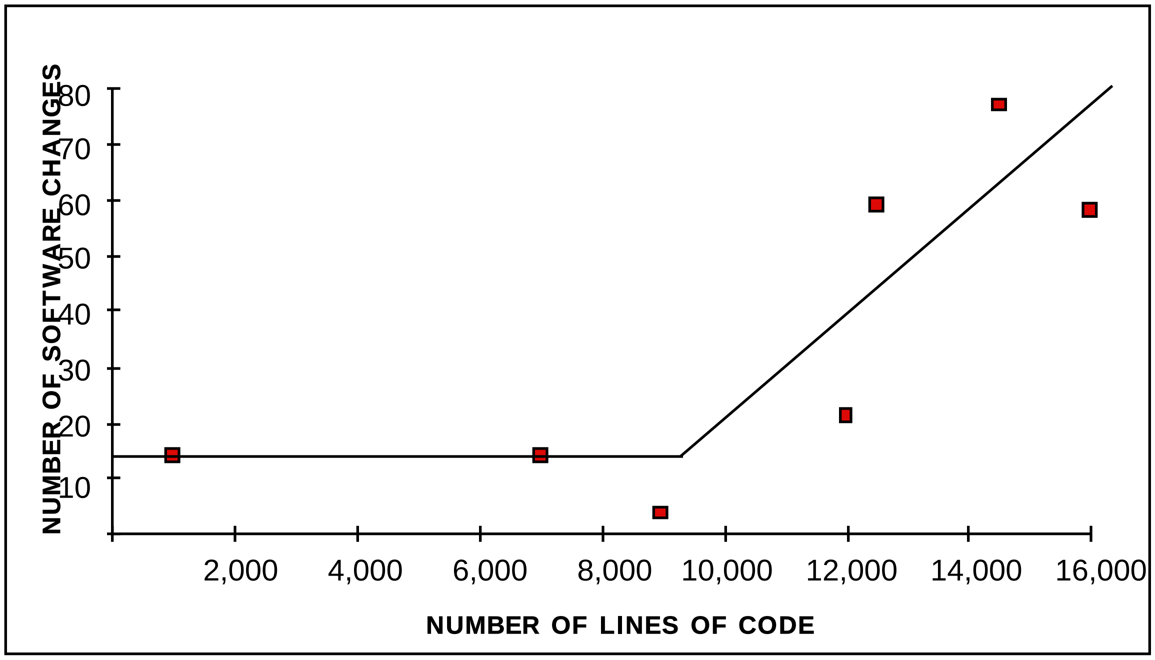

FIGURE 1 SOFTWARE CHANGES RESULTING FROM SNEAK ANALYSIS

A special class of software problems have been found to exist in independently written, multiple versions of software for redundant systems. Use of independently written, multiple version software has been used in recent years in safety critical systems such as flight controls. An experiment by Knight and Leveson [1], showed that for a particular application, certain areas of the program were likely to contain common errors between multiple versions, even though the multiple versions were independently written by programmers of varying backgrounds and experience. Testing to determine if different outputs occur from the different versions will not uncover common problems. Sneak Analysis has demonstrated its ability to find problems overlooked by programmers and not caught during rigorous tests. Sneak Analysis has found undesirable conditions even in software which meets specifications. If these subtle but serious types of problems are not discovered until late in testing, doubt can be cast upon the entire reliability program.

Sneak Analysis does not rely on executing the software to detect sneaks. Unlike most evaluation approaches, including testing, it does not use input to output checking, where the outputs of a system are evaluated based upon a series of prescribed inputs. Input to output checking approaches are valuable, but 'they "choke" computationally on complex systems because of the vast number of combinations that must be considered to completely evaluate the system.

In order to avoid the pitfalls of traditional techniques, in Sneak Analysis the detailed, code-level pathing of the software is traced and displayed graphically as network trees. The software network trees are plotted by replacing the sequential software listings with corresponding topological patterns. Each software network tree shows all logic paths and instructions for a given section of software code. Each network tree in which a variable is referenced contains cross-references to those network trees where that variable is defined. Decision points, cross-references between labels and their references, and cross-references between subroutines and their calls are shown so that program flow is easily traced. These software trees are integrated with one another and with the hardware network trees through the use of network forests.

The Sneak Analysis approach starts at an output and works backward to see what inputs (static or dynamic), if any, could cause that output to assume a sneak value. With Sneak Analysis only a small fraction of the combination of inputs possible in the design need to be checked in detail. Those needing further checking are determined through a careful rule-based process. The rules, which are called sneak clues, are a composite of historical information collected on weaknesses in design logic and technologies including software languages. Continuing emphasis is placed on keeping these rules up-to-date.

System-level outputs typically involve multiple network trees to encompass their functionality. Each network tree which represents a system output becomes the output of a network forest. The forests establish the functional connectivity between the system output and all of the related inputs. The forest is constructed by matching the computer-generated cross-references for each network tree. These cross-references relate the inputs of a given network tree to the outputs of other network trees. Thus, each forest becomes a diagram of functionally related network trees that defines a system output in terms of all of its controlling inputs. Hardware and software network trees are also integrated by network forests. The forests provide a system level view of the relationships between the various signals and variables which relate to a system output. However, no paths are omitted as they are in typical system level block diagrams. Therefore, clue application to the forests provide insight into complex system relationships including hardware/software interactions which could not be seen clearly through other means.

The final phase of Sneak Analysis is the application of the sneak clues to the topological network trees and forests. The presence of a clue in design data directs the analyst to ask a set of specific questions. The current list of sneak clues developed by Boeing is the result of 25 years of research and experience on numerous types of hardware and software systems. The application of sneak clues is performed without limitations as to the intended sequence of inputs. This principle makes Sneak Analysis unique from simulations, testing, and walkthroughs, since it is the unexpected combinations or sequences of inputs that make sneak conditions difficult to uncover by traditional means.

The sneak clues, complete with detailed explanations, form a section of the Sneak Analysis Handbook which is issued to each sneak analyst. For ease of application, the clues are classified into categories which relate to visual keys on the network trees. Sneak clue category checklists have been developed which index the visual keys to specific examples and questions. A computerized version of the sneak clue list has been developed for the Ada language.

Even though considerable effort has been put into producing computer aides for the analysts, the concept of trained, experienced, and independent analysts has continued to be an important element in Sneak Analysis.

In summary, meeting software reliability goals can be supported by proper application of Sneak Analysis to software and its interaction with hardware. Software and system-level problems that may otherwise go undetected until late in the program can be found and corrected before the problem ever occurs through exercising the software.

Reference

[1] J. Knight and N. Leveson, "An Experimental Evaluation of the Assumption of Independence in Multiversion Programming" in IEEE Transactions on Software Engineering, Vol. SE-12, No. 1, January 1986