# Validation and Testing of Knowledge-Based Systems
# How bad can it get?

Position Paper for AAAI-88 Workshop on Validation and Testing of Knowledge-Based Systems

Reprinted in Uma G. Gupta, editor. *Validating and Verifying Knowledge-Based Systems*, pp. 76–82. IEEE Computer Society, Los Alamitos, CA, 1991.

John Rushby
Computer Science Laboratory
SRI International

My subtitle is not intended as a slighting reference to the current state of the art for validation and verification of knowledge-based systems, but as a reminder that for many systems the important question concerning deployment is not "how well does it work?" but "how badly can it fail?" This is most obviously the case in safety-critical systems, but it can apply to any system where the consequences of certain types of failure may be incommensurate with the benefits of normal operation. For conventional software, it is a topic of active debate whether the techniques that are most effective for ensuring that a system "works well" are also the most effective for showing that it "cannot go badly wrong." At the core of this debate is the question whether safety should be distinguished from reliability.

Verification and validation for knowledge-based systems has been almost exclusively concerned with showing that these systems can work well. But if knowledge-based systems are to become accepted as components of larger systems that perform critical functions, it will also be necessary to consider the question of how badly they can go wrong. In this regard, it will be useful to consider relevant experience with conventional software and to introduce some of the terms and concepts from "dependable systems," "software reliability," and "software safety."

# 1 Dependability and Safety

The **dependability** of a computing system is the extent to which *reliance* can *justifiably* be placed in the *quality of service* which it delivers [6]. Quality of service includes both correctness (i.e., conformity with requirements, specifications, and expectations) and continuity. Any departure from the service required of a system constitutes a **failure**—so that a dependable system is one that does not fail. Failures are the result of **faults**, but not all faults produce immediate failures. Failure is a property of the external behavior of a system, and that external behavior is a manifestation of internal states and state transitions. A fault will affect some part of the system state, thereby producing a latent **error**. If uncorrected, the error in the internal system state will eventually become **effective** and change the external behavior of the system and thereby lead to failure.

Given these definitions, we can identify two main approaches for achieving dependability: either the system must contain no faults, or else it must be able to cope with the latent errors caused by faults before those errors become effective. The first of these approaches is referred to as **fault avoidance**, the second as **fault tolerance**.

Under the heading of fault avoidance fall those techniques that attempt, either directly by construction or indirectly by analysis and subsequent correction, to produce systems free from faults. Systematic design methodologies, especially those with a strong mathematical foundation, aim to eliminate faults at their source; validation and systematic testing strategies, together with formal and informal verification, aim to discover and eliminate faults during the development process.

Failure *severities* can be classified as benign or catastrophic: a **benign** failure is one whose consequences are of the same order as the benefit provided by proper service, whereas a **catastrophic** failure is one whose consequences are incommensurate with the benefits of proper service. **Reliability** is a measure of the *continuous* delivery of proper service—or equivalently, time to failure. Under this interpretation, **safety** is reliability with respect to catastrophic failure (i.e., time to catastrophic failure).

A criticism of this formulation of safety is that it approaches it through the concept of failure—which itself is defined in terms of departure from proper (i.e., specified) service. Thus an inappropriately specified system may cause unintended death or injury without experiencing a failure—and must therefore be considered safe according to this definition of safety.

An alternative view [7, 9] identifies safety with the occurrence of accidents or **mishaps**—which are defined as unplanned events that result in death, injury, illness, damage to or loss of property, or environmental harm. Whereas system *failure* is defined in terms of *system services*, this view of *safety* defines it in terms of *external consequences*. If the required system services are specified incorrectly, then a system may be unsafe, though perfectly reliable. Conversely, it is feasible for a system to be safe, but unreliable. Enhancing the reliability of software, though desirable and perhaps necessary, is not sufficient for achieving *safe* software.

The supporters of the dependability view of safety argue that it subsumes this alternative view: it simply becomes one of the requirements on the system that it should neither cause nor contribute to mishaps. A rather more interesting debate concerns which are the most appropriate and effective validation techniques for safety properties. The dependability viewpoint might suggest that since safety is simply reliability with respect to the absence of catastrophic failures, those techniques that are most effective for ensuring (general) reliability should also be among the most effective for ensuring safety. These techniques include random and functional testing, anomaly detection, and possibly certain software fault-tolerance techniques such a N-version programming and recovery blocks [2].

Unfortunately, while these approaches can be effective at the levels of reliability required for ordinary, functional properties, they seem inadequate at the extreme reliability levels demanded for safety properties. As an indication of the reliabilities demanded, the FAA requires failures that could endanger a civil aircraft to be "extremely improbable" and "not expected to occur within the total life span of the whole fleet of the model"—this phrase is defined by the FAA as a probability of $10^{-9}$ per hour of flight. Testing for reliabilities such as these is infeasible. Furthermore, Knight and Leveson describe experiments [3–5] that cast doubt on the ability of software fault-tolerance techniques to deliver the required reliabilities.

If conventional testing and fault tolerance are inadequate for ensuring safety in software-intensive systems, do any promising techniques remain? In our view, there are three such: design for safety, software fault-tree analysis, and formal verification. In order to outline the first two, we need to introduce some of the terms used in system safety engineering. A *hazard* is a condition with the potential for causing a mishap; *danger* is the probability of the hazard leading to a mishap. *Damage* is a measure of the loss in a mishap, while the *severity* of a hazard is an assessment of the worst possible damage that could result. *Risk* is the combination of hazard severity and danger.

Design for safety is concerned with design principles that will reduce the likelihood and risk of hazardous states. Techniques such as interlocks, lockouts, and firewalls can be adapted from conventional system safety engineering [9]. "Software Fault Tree Analysis" (SFTA) [8] is a method for analyzing software in order to identify unduly hazardous states. It is an adaptation to software of a technique that was developed and first applied in the late 60's in order to minimize the risk of inadvertent launch of a Minuteman missile. The first step, as in any safety analysis, is a hazard analysis of the entire system. This is essentially a listing and categorization of the hazards posed by the system. The goal of SFTA is to show that the logic contained in the software design will not produce system safety failures, and to determine environmental conditions which could lead to the software causing a safety failure. The basic procedure is to suppose that the software has caused a condition which the hazard analysis has determined will lead to catastrophe, and then to work backward to determine the set of possible causes for the condition to occur.

Formal verification is the demonstration of consistency between two different descriptions of a software system; often one description is the program itself and the other is its specification, though the method can equally well be applied to two specifications at different levels of detail. In the context of safety, one of the specifications will generally state the system safety requirements, while the other will describe the actual system design. The presence of the adjective "formal" qualifies this particular form of verification as a mathematical activity in which the two system descriptions are treated as formal mathematical texts, and the notion of consistency that is to be established between them is also a formal, mathematical one.

# 2 Application to Knowledge-Based Systems

A substantial body of knowledge and experience has been developed for the verification and validation of conventional software. One of the principal difficulties in adapting these techniques to knowledge-based systems is the lack of explicit and precise requirements and specification documents for much AI software. This lack reflects the genuine difficulty in stating à priori the expectations and requirements for a system whose capabilities will evolve through a development process that is partly experimental in nature. However, we suggest that while it is difficult to produce independent and precise specifications for the *desired* behavior of many knowledge-based systems, it may be feasible to express certain *undesired* or "safety" properties quite sharply. We believe that such distinctions among the requirements on knowledge-based systems are essential if any of the established methods and techniques for software quality assurance are to be applied to such systems [10].

## 2.1 Service and Competency Requirements

We believe it will be helpful to distinguish two sets of requirements and specifications for knowledge-based systems: the **competency** requirements and the **service** requirements. As the name suggests, the competency requirements will be concerned with those dimensions of the overall requirements that concern "knowledge" or appeal to comparison with human skills. We accept that such requirements may of necessity be vague and incomplete (though every effort should be expended to render them otherwise)—for example, "perform diagnosis of faults in Machine A, on the basis of read-outs from sensors X and Y, at the level of competence of factory experts." The service requirements cover all other requirements and should be amenable to statements no less rigorous and formal than those for conventional software. Service requirements should include descriptions of the input and output formats expected, the processing rate, the explanation facilities required, and so on. Service requirements and their decomposition through levels of specification should be traceable, verifiable, and testable just like those of conventional software.

In some cases, the satisfaction of service requirements will be wholly the responsibility of the developer of the system concerned, in others it may depend in part on facilities provided by an expert system "shell." In the latter case, the system developer should hope to see very precise specifications of the services provided by the shell, and assurance that they are achieved reliably.

## 2.2 Desired and Minimum Competency Requirements

Competency requirements can usefully be further subdivided into **desired** and **minimum** requirements. The desired competency requirement is probably defined relative to human expertise and describes how *well* the system is expected to perform. The minimum competency requirement should define how *badly* it is allowed to perform; minimum competency requirements may have much in common with system safety specifications. It is the establishment of precise and comprehensive minimum competency requirements, and the demonstration that they have been achieved, that may be the determining factor in the widespread acceptance of AI software in other than non-critical applications.

Whereas desired competency requirements may be hard to state explicitly, there is some hope that minimum competency requirements may sometimes be capable of precise definition. This is particularly likely to be so in the case of AI systems whose purpose is to optimize some function—for although an *optimal* value may be hard to find, and its optimality just as hard to check, it may be quite feasible to check that it is at least a *valid* solution.

For example, one of the expert systems developed at SRI, by the Information Sciences and Technology Center, is the Automated Air Load Planning System—AALPS [1]. It is used to produce schedules and layouts for loading army divisions and their equipment (tanks, helicopters, etc.) onto air transports. One of the things to be considered is the unloading of the aircraft, especially if this is to be performed in flight using parachute drops. As the heavy equipment is first moved around inside the aircraft prior to being dropped, and then actually dropped, the aerodynamic stability of the aircraft must remain undisturbed. Specifying the desired competency of AALPS is obviously difficult—we want a near optimum loading plan, but an *optimum* loading plan is hard to define, and it is even harder to determine whether a given plan is optimal. But the minimum competency requirement can be given quite a sharp statement: the aerodynamic trim of the aircraft must remain within certain limits at all times during all stages of unloading in flight.

### 2.3 Safety Validation for Knowledge-Based Systems

We contend that minimum competency requirements, particularly those associated with safety issues, can often be specified with precision, and their satisfaction can be designed for, tested, and verified using quality assurance techniques derived from those for conventional software. We have just begun a research program at SRI to evaluate this contention. Our primary interest is to determine the extent to which it is feasible and effective to apply formal verification techniques to minimum competency requirements for knowledge-based systems. Initially, we are considering rule-based systems.

Among the more challenging problems that face us are the lack of distinction between "design" and "implementation" in knowledge-based systems, and the lack of a formal semantics for many of the notations used.

Formal verification for conventional software is often applied at the design, rather than the code level. That is, a reasonably detailed design specification is formally verified against a (very simple and perspicuous) safety specification; the mapping from the verified design specification to executable code is performed informally. The reason for this approach is economics: even with the best modern specification and verification environments (e.g., our own [12]) formal verification is extremely costly and it is generally too expensive to apply fully formal reasoning to every step from requirements to code. We prefer to apply formal verification at the "design level" because

1. This is where serious errors are most often introduced,
2. It is much simpler to reason about a design expressed in a formal specification language with clean semantics than it is to reason about executable code, and
3. Testing, which is quite effective at detecting coding errors, provides an acceptable validation technique for the remaining steps.

An alternative school of thought prefers to verify the actual program text against a highly detailed specification—but then one must ask where that specification came from, and what assurance there is for its own correctness.

With knowledge-based systems, there is little distinction between design and implementation—that is, between knowledge and its representation. Thus, we must either introduce a design level notation—that is, a more abstract representation for knowledge—or we must reason about its concrete representation. The latter course raises the problem of semantics: despite the superficial similarity between the inference procedures for rule-based systems and logical deduction, the operational features and incompleteness of the former mean that standard logic and model theory do not provide the correct semantics. Mark Stickel's Prolog Technology Theorem Prover (PTTP) [11] is an exception to this rule (it provides correct first-order semantics, with Prolog-like efficiency when restricted to Horn clauses).

## 3 Conclusion

The notions of dependability and safety that have been developed for conventional software systems are relevant to knowledge-based systems also. This motivates the introduction of "minimum competency requirements" for knowledge-based systems and consideration of the possibility that such requirements may be amenable to formal verification. Among the difficulties facing this endeavor is the absence of formal semantics for most knowledge-representation schemes. With sponsorship from the NASA Langley Research Center, we have just begun a program of research into these topics and expect to report our interim results next year.

## References


[1] Debra Anderson and Charles Ortiz. AALPS: A knowledge-based system for aircraft loading. *IEEE Expert*, 2(4):71–79, Winter 1987.

[2] T. Anderson and P. A. Lee. *Fault-Tolerance: Principles and Practice (Second, revised edition)*. Springer-Verlag, Vienna, Austria, 1990.

[3] S. Cha, N. G. Leveson, T. J. Shimeall, and J. C. Knight. An empirical study of software error detection using self-checks. In *Fault Tolerant Computing Symposium 17*, pages 156–161, Pittsburgh, PA., July 1987. IEEE Computer Society.

[4] J. C. Knight and N. G. Leveson. An empirical study of failure probabilities in multi-version software. In *Fault Tolerant Computing Symposium 16*, pages 165–170, Vienna, Austria, July 1986. IEEE Computer Society.

[5] J. C. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering*, SE-12(1):96–109, January 1986.

[6] J. C. Laprie. Dependable computing and fault tolerance: Concepts and terminology. In *Fault Tolerant Computing Symposium 15*, pages 2–11, Ann Arbor, MI, June 1985. IEEE Computer Society.

[7] N. G. Leveson. Software safety in computer controlled systems. *IEEE Computer*, 17(2):48–55, February 1984.

[8] N. G. Leveson and P. R. Harvey. Analyzing software safety. *IEEE Transactions on Software Engineering*, SE-9(5):569–579, September 1983.

[9] Nancy G. Leveson. Software safety: Why, what and how. *ACM Computing Surveys*, 18(2):125–163, June 1986.

[10] John Rushby. Quality measures and assurance for AI software. Technical Report SRI-CSL-88-7R, Computer Science Laboratory, SRI International, Menlo Park, CA, September 1988. Also available as NASA Contractor Report 4187.

[11] Mark E. Stickel. A Prolog technology theorem prover: Implementation by an extended Prolog compiler. *Journal of Automated Reasoning*, 4(4):353–380, December 1988.

[12] F. W. von Henke, J. S. Crow, R. Lee, J. M. Rushby, and R. A. Whitehurst. The EHDM verification environment: An overview. In *Proceedings 11th National Computer Security Conference*, pages 147–155, Baltimore, MD, October 1988. NBS/NCSC.