Quality Measures and Assurance for AI Software[1]

John Rushby

Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025

Technical Report CSL-88-7R, September 1988
(Also available as NASA Contractor Report 4187)

[1]This work was performed for the National Aeronautics and Space Administration under contract NAS1 17067 (Task 5).

Abstract

This report is concerned with the application of software quality and assurance techniques to AI software. It describes work performed for the National Aeronautics and Space Administration under contract NAS1 17067 (Task 5) and is also available as a NASA Contractor Report.

The report is divided into three parts. In Part I we review existing software quality assurance measures and techniques–those that have been developed for, and applied to, conventional software. This part, which provides a fairly comprehensive overview of software reliability and metrics, static and dynamic testing, and formal specification and verification, may be of interest to those unconcerned with AI software. In Part II, we consider the characteristics of AI-based software, the applicability and potential utility of measures and techniques identified in the first part, and we review those few methods that have been developed specifically for AI-based software. In Part III of this report, we present our assessment and recommendations for the further exploration of this important area. An extensive bibliography with 194 entries is provided.

Contents

Chapter 1

Introduction

This report is concerned with the application of software quality and evaluation measures to AI software and, more broadly, with the question of quality assurance for AI software. By AI software we mean software that uses techniques from the field of Artificial Intelligence. (Genesereth and Nilsson [73] give an excellent modern introduction to such techniques; Harmon and King [84] provide a more elementary overview.) We consider not only metrics that attempt to measure some aspect of software quality, but also methodologies and techniques (such as systematic testing) that attempt to improve some dimension of quality, without necessarily quantifying the extent of the improvement. The bulk of the report is divided into three parts. In Part I we review existing software quality measures—those that have been developed for, and applied to, conventional software. In Part II, we consider the characteristics of AI software, the applicability and potential utility of measures and techniques identified in the first part, and we review those few methods that have been developed specifically for AI software. In Part III of this report, we present our assessment and recommendations for the further exploration of this important area.

1.1 Motivation

It is now widely recognized that the cost of software vastly exceeds that of the hardware it runs on—software accounts for 80% of the total computer systems budget of the Department of Defense, for example. Furthermore, as much as 60% of the software budget may be spent on maintenance. Not only does software cost a huge amount to develop and maintain, but vast economic or social assets may be dependent upon its functioning correctly. It is therefore essential to develop techniques for measuring, predicting, and controlling the costs of software development and the quality of the software produced.

The quality-assurance problem is particularly acute in the case of AI software—which for present purposes we may define as software that performs functions previously thought to require human judgment, knowledge, or intelligence, using heuristic, search-based techniques. As Parnas observes [149]:

> "The rules that one obtains by studying people turn out to be inconsistent, incomplete, and inaccurate. Heuristic programs are developed by a trial and error process in which a new rule is added whenever one finds a case that is not handled by the old rules. This approach usually yields a program whose behavior is poorly understood and hard to predict."

Unless compelling evidence can be adduced that such software can be "trusted" to perform its function, then it will not—and should not—be used in many circumstances where it would otherwise bring great benefit. In the following sections of this report, we consider measures and techniques that may provide the compelling evidence required.

1.2 Acknowledgments

Alan Whitehurst of the Computer Science Laboratory (CSL) and Leonard Wesley of the Artificial Intelligence Center (AIC) at SRI contributed material to this report. It is also a pleasure to acknowledge several useful discussions with Tom Garvey of AIC, and the careful reading and criticism of drafts provided by Oscar Firschein of AIC and Teresa Lunt of CSL. The guidance provided by our technical monitors, Kathy Abbott and Wendell Ricks of NASA Langley Research Center, was extremely valuable.

Part I

Quality Measures for Conventional Software

Chapter 2

Software Engineering and Software Quality Assurance

Before describing specific quality metrics and methods, we need briefly to review the Software Engineering process, and some of the terms used in Software Quality Assurance.

One of the key concepts in modern software engineering is the system life-cycle model. Its premise is that development and implementation are carried out in several distinguishable, sequential phases, each performing unique, well-defined tasks, and requiring different skills. One of the outputs of each phase is a document that serves as the basis for evaluating the outcome of the phase, and forms a guideline for the subsequent phases. The life-cycle phases can be grouped into the following four major classes:

Specification comprising problem definition, feasibility studies, system requirements specification, software requirements specification, and preliminary design.

Development comprising detailed design, coding and unit testing, and the establishment of operating procedures.

Implementation comprising integration and test, acceptance tests, and user training.

Operation and maintenance.

There have been many refinements to this basic model: Royce's *Waterfall* model [161], for example, recognized the existence of feedback between phases and recommended that such feedback should be confined to adjacent phases.

There is considerable agreement that the early phases of the life-cycle are particularly important to the successful outcome of the whole process: Brooks, for example observes [33]

> "I believe the hard part of building software to be the specification, design, and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation. We still make syntax errors, to be sure, but they are fuzz compared with the conceptual errors in most systems.
>
> "The hardest single part of building a software system is deciding precisely what to build. No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later."

The more phases of the life-cycle that separate the commission and detection of an error, the more expensive it is to correct. It is usually cheap and simple to correct a coding bug caught during unit test, and it is usually equally simple and cheap to insert a missed requirement that is caught during system requirements review. But it will be ruinously expensive to correct such a missed requirement if it is not detected until the system has been coded and is undergoing integration test. Software Quality Assurance comprises a collection of techniques and guidelines that endeavor to ensure that all errors are caught, and caught early.

2.1 Software Quality Assurance

Software Quality Assurance (SQA) is concerned with the problems of ensuring and demonstrating that software (or, rather, software-intensive systems) will satisfy the needs and requirements of those who procure them. These needs and requirements may cover not only how well the software works *now*, but how well documented it is, how easy to fix if it does go wrong, how adaptable it is to new requirements, and other attributes that influence how well it will continue to satisfy the user's needs in the *future*. In the case of military procurements, a number of standards have been established to govern the practice of various facets of software development: MIL-S-52779A for software program requirements, DOD-STD-1679A and DOD-STD-2167 for software development, DOD-STD-2168 for software quality evaluation, and DOD-STD-7935 for software documentation. Similar standards exist in the civil and international sectors.

One important methodology in SQA is “Verification and Validation” (V&V). *Verification* is the process of determining whether each level of specification, and the final code itself, fully and exclusively implements the requirements of its superior specification. That is, all specifications and code must be *traceable* to a superior specification. *Validation* is the process by which delivered code is directly shown to satisfy the original user requirements.

Verification is usually a manual process that examines descriptions of the software, while validation depends on testing the software in execution. The two processes are complementary: each is effective at detecting errors that the other will miss, and they are therefore usually employed together. Procurements for mission-critical systems often specify that an *independent* group, unconnected to the development team, should undertake the V&V activity.

Chapter 3

Software Reliability

Software reliability is concerned with quantifying how well software functions to meet the needs of its customer. It is defined as the probability that the software will function without failure for a specified period of time. "Failure" means that in some way the software has not functioned according to the customer's requirements. This broad definition of failure ensures that the concept of reliability subsumes most properties generally associated with quality—not only correctness, but also adequacy of performance, and user-friendliness. Reliability is a user-oriented view of software quality: it is concerned with how well the software actually works. Alternative notions of software quality tend to be introspective, developer-oriented views that associate quality with the "complexity" or "structure" of the software. Fortunately, software reliability is not only one of the most important and immediate attributes of software quality, it is also the most readily quantified and measured.

Software reliability is a scientific field, and employs careful definition of terms. The two most important are "failure" and "fault." A software *failure* is a departure of the external behavior of the program from the user's requirements. The notion of requirements is discussed in 5.3.1. A software *fault* is a defect in a program that, when executed under certain conditions, causes a failure—that is, what is generally called a "bug." Failure occurrence is affected by two principal factors:

- The number of faults in the software being executed—clearly, the more bugs, the more failures may be expected, and

- The circumstances of its execution (sometimes called the "operational profile"). Some circumstances may be more exacting than others and may lead to more failures.

Software reliability is the probability of failure-free operation of a computer program for a specified time under specified circumstances. Thus, for example, a text-editor may have a reliability of 0.97 for 8 hours when employed by a secretary—but only 0.83 for 8 hours when employed by a hacker. To give an idea of the reliabilities demanded of flight-critical aircraft systems (in which software components are increasingly important), the FAA requires the probability of catastrophic failure to be less than 10^{-9} per 10-hour flight for a life-critical civil air transport flight control system; the US Air Force requires the probability of mission failure to be less than 10^{-7} per hour for military aircraft.

From the basic notion of reliability, many different measures can be developed to quantify the occurrence of failures in time. Some of the most important of these measures, and their interrelationships are summarized below:

Reliability, denoted by $R(t)$, is the probability of failure-free operation up to time t.

Failure Probability, denoted by $F(t)$, is the probability that the software will fail prior to time t. Reliability and failure probability are related by $R(t) = 1 - F(t)$.

Failure Density, denoted by $f(t)$, is the probability density for failure at time t. It is related to failure probability by $f(t) = \frac{d}{dt}F(t)$. The probability of failure in the half-open interval $(t, t + \delta t]$ is $f(t).\delta t$.

Hazard Rate, denoted by $z(t)$, is the conditional failure density at time t, given that no failure has occurred up to that time. That is, $z(t) = f(t)/R(t)$. Reliability and hazard rate are related by

$$R(t) = e^{-\int_0^t z(x)\,dx}.$$

An important special case occurs when the hazard rate is a constant ϕ. In this case the failure density has an exponential distribution $f(t) = \phi e^{-\phi t}$, the failure probability is given by $F(t) = 1 - e^{-\phi t}$ and the reliability is given by $R(t) = e^{-\phi t}$.

Mean Value Function, denoted by $\mu(t)$, is the mean *number* of failures that have occurred by time t.

Failure Intensity, denoted by $\lambda(t)$, is the number of failures occurring per unit time at time t. This is related to the mean value function by $\lambda(t) = \frac{d}{dt}\mu(t)$. The number of failures expected to occur in the half-open interval $(t, t + \delta t]$ is $\lambda(t).\delta t$.

Failure intensity is the measure most commonly used in the quantification of software reliability. [1] Because of the complexity of the factors influencing the occurrence of a failure, the quantities associated with reliability are random variables, and reliability models are based on the mathematics of random or stochastic processes. Because failures generally provoke (attempted) repair, the number of faults in a program generally changes over time, and so the probability distributions of the components of a reliability model vary with time. That is to say, reliability models are based on *nonhomogeneous* random processes.

A great many software reliability models have been developed. These are treated systematically in the first textbook to cover the field, which has just been published [134]. The most accurate and generally recommended model is the "Basic Execution Time Model." We outline the derivation of this model below.

3.1 The Basic Execution Time Reliability Model

The starting point for this derivation (and that of most other reliability models) is to model the software failure process as a NonHomogeneous Poisson Process (NHPP)—a particular type of Markov model.

Let $M(t)$ denote the number of failures experienced by time t. We make the following assumptions:

1. No failures are experienced by time 0, that is $M(0) = 0$,

2. The process has independent increments, that is the value of $M(t+\delta t)$ depends only on the present value of $M(t)$ and is independent of its history,

[1]Mean Time To Failure (MTTF), denoted by Θ, is not employed to the extent that it is in hardware reliability studies (probably because the models used in software reliability tend not to give rise to closed form expressions for MTTF). This measure is related to reliability by $\Theta = \int_0^\infty R(x)\,dx$.

3. The probability that a failure will occur during the half-open interval $(t, t + \delta t]$ is $\lambda(t).\delta t + o(\delta t)$, where $\lambda(t)$ is the *failure intensity* of the process.

4. The probability that more than one failure will occur during the half-open interval $(t, t + \delta t]$ is $o(\delta t)$.

If we let $P_{m(t)}$ denote the probability that $M(t)$ is equal to m, that is:

$$P_{m(t)} = \text{Prob}[M(t) = m]$$

then it can be shown that $M(t)$ is distributed as a Poisson random variable. That is:

$$P_{m(t)} = \frac{\mu(t)^m}{m!}\, e^{-\mu(t)}$$

where

$$\mu(t) = \int_0^t \lambda(x)\, dx.$$

Next we make some further assumptions:

1. Whenever a software failure occurs, the fault that caused it will be identified and removed instantaneously. (A more realistic assumption will be substituted later).

2. The number of faults in the program initially is a Poisson random variable with mean ω_0.

3. The hazard rate for all faults is the same for all faults, namely $z_a(t)$.

Under these additional assumptions, it can be shown that

$$\mu(t) = \omega_0 F_a(t) \tag{3.1}$$

where $F_a(t)$ is the per-fault failure probability, and

$$\lambda(t) = \omega_0 f_a(t) \tag{3.2}$$

where $f_a(t)$ is the per-fault failure density.

The final assumption in the derivation of the model is that the per-fault failure density has an exponential distribution. That is,

$$f_a(t) = \phi e^{-\phi t}. \tag{3.3}$$

(Note this implies that the per-fault hazard rate is constant, that is $z_a(t) = \phi$). Substituting (3.3) into (3.2), we obtain

$$\lambda(t) = \omega_0 \phi e^{-\phi t}.$$

Letting λ_0 denote $\lambda(0)$, we obtain $\phi = \lambda_0/\omega_0$ and hence

$$\lambda(t) = \lambda_0 e^{-\frac{\lambda_0}{\omega_0} t}. \tag{3.4}$$

Similarly, from (3.1) we obtain

$$\mu(t) = \omega 0 \left(1 - e^{-\frac{\lambda_0}{\omega_0} t}\right). \tag{3.5}$$

Some additional manipulation allows us to express failure intensity, λ, as a function of failures observed, μ:

$$\lambda(\mu) = \lambda_0 \left(1 - \frac{\mu}{\omega_0}\right). \tag{3.6}$$

The assumption that the fault responsible for each failure is identified and removed immediately the failure occurs is clearly unrealistic. The model can be modified to accommodate imperfect debugging by introducing a *fault reduction factor B*. This attempts to account for faults that cannot be located, faults found by code-inspection prior to causing failure, and faults introduced during debugging, by assuming that, on average, each failure leads to the removal of B faults ($0 \leq B \leq 1$). (Conversely, each fault will lead to $1/B$ failures.) It can then be shown that

$$\mu(t) = \nu_0 G_a(t)$$

where $\nu_0 = \omega_0/B$ is the total expected number of failures, and

$$G_a(t) = 1 - e^{-B \int_0^t z_a(x)\, dx}$$

is the cumulative distribution function of the time to remove a fault. Similarly

$$\lambda(t) = \nu_0 g_a(t)$$

where $g_a(t)$ is the probability density function associated with $G_a(t)$.

The consequent modifications to the important equations (3.4–3.6) are simple: merely replace ω_0 (the number of faults) by ν_0 (the total number of failures expected). Thus we obtain

$$\begin{aligned} \lambda(t) &= \lambda_0 e^{-\frac{\lambda_0}{\nu_0}t}, & (3.7)\\ \mu(t) &= \nu_0\left(1-e^{-\frac{\lambda_0}{\nu_0}t}\right), \text{ and} & (3.8)\\ \lambda(\mu) &= \lambda_0\left(1-\frac{\mu}{\nu_0}\right). & (3.9) \end{aligned}$$

As an example of the application of these formulae, consider a system containing 90 faults and whose fault-reduction factor is 0.9. The system may be expected to experience 100 failures. Suppose its initial failure intensity was 10 failures per CPU-hour and that it has now experienced 50 failures. Then the present failure intensity should be given by

$$\begin{aligned} \lambda(\mu) &= \lambda_0\left(1-\frac{\mu}{\nu_0}\right)\\ \lambda(50) &= 10\left(1-\frac{50}{100}\right)\\ &= 5 \text{ failures per CPU-hour.} \end{aligned}$$

In addition we may ask what the failure intensity will be after 10 CPU-hours and how many failures will have occurred by that time. For failure intensity we have:

$$\begin{aligned} \lambda(t) &= \lambda_0 e^{-\frac{\lambda_0}{\nu_0}t}\\ \lambda(10) &= 10e^{-\frac{10}{100}10}\\ &= 10e^{-1}\\ &= 3.68 \text{ failures per CPU-hour,} \end{aligned}$$

and for failures we have

$$\begin{aligned} \mu(t) &= \nu_0\left(1-e^{-\frac{\lambda_0}{\mu_0}t}\right)\\ \mu(10) &= 100\left(1-e^{-\frac{10}{100}10}\right)\\ &= 100\left(1-e^{-1}\right)\\ &= 100(1-0.368)\\ &= 63 \text{ failures.} \end{aligned}$$

Additional formulae can be derived to give the *incremental* number of failures ($\delta\mu$) or elapsed time (δt) to progress from a known present failure intensity (λ_P), to a desired future goal (λ_F):

$$\delta\mu = \frac{\nu_0}{\lambda_0}(\lambda_P - \lambda_F), \text{ and} \tag{3.10}$$

$$\delta t = \frac{\nu_0}{\lambda_0}\ln\frac{\lambda_P}{\lambda_F}. \tag{3.11}$$

Using the same example as before (90 faults, fault-reduction factor 0.9, initial failure intensity 10 per CPU-hour), we can ask how many additional failures may be expected between a present failure intensity of 3.68 per CPU-hour, and a desired intensity of 0.000454 per CPU-hour:

$$\begin{aligned}
\delta\mu &= \frac{\nu_0}{\lambda_0}(\lambda_P - \lambda_F) \\
&= \frac{100}{10}(3.68 - 0.000454) \\
&= 10(3.68) \\
&= 37 \ \text{ failures.}
\end{aligned}$$

Similarly we may inquire how long this may be expected to take:

$$\begin{aligned}
\delta t &= \frac{\nu_0}{\lambda_0}\ln\frac{\lambda_P}{\lambda_F} \\
&= \frac{100}{10}\ln\frac{3.68}{0.000454} \\
&= 10\ln 8106 \\
&= 10(9) \\
&= 90 \ \text{ CPU-hours.}
\end{aligned}$$

The reliability model just developed is determined by two parameters: the initial fault intensity, λ_0, and the total number of failures expected in infinite time, ν_0. In order to apply the model, we need values for these two parameters. If the program of interest has been in operation for a sufficient length of time that accurate failure data are available, then we can *estimate* the values of the two parameters. Maximum likelihood or other methods of statistical estimation may be used, and confidence intervals may be used to characterize the accuracy of the estimates.

Prior to operational data becoming available, however, we can only attempt to *predict* values for the parameters of the model, using characteristics

Development Phase	Faults/K source lines
Coding	99.50
Unit test	19.70
System test	6.01
Operation	1.48

Figure 3.1: Fault Density in Different Phases of Software Development

of the program itself, and the circumstances of its development. The total number of failures can be predicted as the number of faults inherent in the program, divided by the fault-reduction factor: $\nu_0 = \omega_0/B$. Dozens of techniques have been proposed for estimating the number of faults in a program based on static characteristics of the program itself. These are generally related to some notion of "complexity" of programs and are described in detail in the following chapter. For the purposes of exposition, we will use the simplest (yet one of the best) such measures: the length of the program. There is quite good evidence that faults are linearly related to the length of the source program. If we let I_s denote the length of the program (measured by lines of *source* code), and D the fault density (in faults per source line), then we have

$$\omega_0 = I_s \times D \tag{3.12}$$

and

$$\nu_0 = \frac{I_s \times D}{B}. \tag{3.13}$$

Results from a large number of experiments to determine values of D are summarized in Figure 3.1 (taken from [134, page 118]). Experimental determinations of the fault-reduction factor range from 0.925 to 0.993, with an average of 0.955 (again, from [134, page 121]). Thus, an à priori estimate for the number of failures to be expected from a 20,000 line program entering the system test phase is given by

$$\begin{aligned} \nu_0 &= \frac{I_s \times D}{B} \\ &= \frac{20 \times 6.01}{0.955} \\ &= 126 \ \ \text{failures.} \end{aligned}$$

The initial failure intensity λ_0 depends upon the number of faults in the program, the rate at which faults are encountered during execution, and the ratio between fault encounters and failures (not all fault encounters—i.e., execution of faulty pieces of code—will lead to failure: particular circumstances may be needed to "trigger" the bug). For the purposes of estimation, we may assume that fault encounters are linearly related to the number of instructions executed (i.e., to the duration of execution and to processor speed), and that failures are linearly related to fault encounters. Since processor speed is generally given in *object* instructions per second, we will also assume that the number of object instructions I_o in a compiled program is linearly related to the number of source lines I_s. Thus, if Q is the *code expansion ratio* (i.e., I_o/Is), and R_o is the number of object instructions executed in unit time, then the number of source lines executed in unit time will be given by $R_s = R_o/Q$. Now each source line executed exposes ω_0/I_s faults. Thus $\omega_0 \times R_s/I_s$ faults will be exposed in unit time. If each fault exposure leads to K failures, we see that the initial failure intensity is given by $\lambda_0 = K \times \omega_0 \times R_s/I_s$. Substituting the previously derived expressions for ω_0 and R_s, we obtain:

$$\lambda_0 = D \times K \times \frac{R_o}{Q}. \tag{3.14}$$

In order to complete the estimation of λ_0, we need values for three new parameters: R_o, Q, and K. The first of these is provided by the computer manufacturer, while the second is a function of the programming language and compiler used. A table of approximate expansion ratios is given by Jones [102, page 49]. The most difficult value to estimate is the fault exposure ratio K. Experimental determinations of K for several large systems yield values ranging from 1.41×10^{-7} to 10.6×10^{-7}—a range of 7.5 to 1, with an average of 4.2×10^{-7} [134, page 122].

As an example, of the application of these estimates, consider a 20,000 line program entering the system test phase. Using $D = 6.01$ and $B = 0.955$ as before, and assuming a code expansion ratio Q of 4, a fault exposure ratio K of 4.2×10^{-7} failures per fault, and a 3 MIPS processor, we obtain:

$$\begin{aligned} \lambda_0 &= D \times K \times \frac{R_o}{Q} \\ &= \frac{6.01}{10^3} \times \frac{4.2}{10^7} \times \frac{3 \times 10^6}{4} \\ &= 1.893 \times 10^{-3} \text{ failures per CPU-sec,} \end{aligned}$$

$$= 6.8 \text{ failures per CPU-hour.}$$

Given these estimates of the parameters ν_0 and λ_0, we may proceed to calculate how long it will take to reduce the fault intensity to an acceptable level—say 0.1 failures per CPU-hour.

$$\begin{aligned}\delta t &= \frac{\nu_0}{\lambda_0} \ln \frac{\lambda_P}{\lambda_F} \\ &= \frac{126}{6.8} \ln \frac{6.8}{0.1} \\ &= 18.53 \ln 68 \\ &= 18.53(4.22) \\ &= 78 \text{ CPU-hours.}\end{aligned}$$

3.2 Discussion of Software Reliability

Software reliability modeling is a serious scientific endeavor. It has been pursued diligently by many of those with a real economic stake in the reliability of their software products—for example, the manufacturers of embedded systems (where repair is often impossible), and those with enormously stringent reliability objectives (for example, the manufacturers of telephone switching equipment).

The "Basic Execution Time Model" described here has been validated over many large projects [134] and has the virtue of relative simplicity compared with many other models. A similar model, the "Logarithmic Poisson Model," has been less intensively applied, but may be preferred in some circumstances. Both of these models use execution-time as their time base. This is one of the primary reasons for their greater accuracy over earlier models, which used man-hours, or other human-oriented measures of time. That execution time should prove more satisfactory is not surprising: the number of failures manifested should surely be most strongly determined by the amount of exercise the software has received. Converting from a machine-oriented view of time to a human-oriented view is often necessary for the application of the results obtained from the model; ways of doing this are described by Musa *et al.* [134].

There are several circumstances that can complicate the application of reliability models. Reliability is concerned with counting failures, and prediction is based on collecting accurate failure data during the early stages of a project. These data may be unreliable, and predications based upon them may be inaccurate, if any of the following circumstances obtain:

- There is ambiguity or uncertainty concerning what constitutes a failure,
- There is a major shift in the operational profile between the data gathering (e.g., testing) phase and the operational phase, or
- The software is undergoing continuous change or evolution.

We note that these circumstances which cause difficulty in the application of reliability modeling, also characterize much AI-software development. We will return to this issue in the second part of the report.

Chapter 4

Size, Complexity, and Effort Metrics

In the previous chapter, we have seen that à priori estimates for the reliability of a program depend on estimates for the number of faults it contains and for its initial failure intensity. It is plausible to suppose that these parameters may themselves be determined by the "size" and "complexity" of the program. Accordingly, considerable effort has been expended in the attempt to define and quantify these notions.

Whereas measurements of the static properties of completed programs (e.g., size and complexity) may help in predicting some aspects of their behavior in execution (e.g., reliability), similar measurements of "requirements" or "designs" may help to predict the "effort" or cost required to develop the finished program.

In this chapter we will examine metrics that purport to measure the size, and complexity of programs, and those that attempt to predict the cost of developing a given piece of software.

4.1 Size Metrics

The *size* of a program is one of its most basic and measurable characteristics. It seems eminently plausible that the effort required to construct a piece of software is strongly influenced, if not fully determined, by its final size, and that its reliability will be similarly influenced.

The crudest measure of the size of a piece of software is its *length*, measured by the number of lines of code that it contains. A line of code is counted as any non-blank, non-comment line, regardless of the number of statements or fragments of statements on the line. The basic unit of program length is the "SLOC"—a single "Source Line Of Code." A variant is the "KLOC"—a thousand lines of code. An obvious objection to these measures is that they do not account for the different "densities" of different programming languages (e.g., a line of APL is generally considered to contain more information, and to require more effort to write, than a line of Cobol), and they do not account for the fact that different lines in the same program, or lines written by different people, may have very different amounts of information on them. A straightforward attempt to overcome the latter objection (and perhaps the former also) is to count syntactic tokens rather than lines.

An early, controversial, and influential system of this type was the "Software Science" of Halstead [83]. Software Science classifies tokens as either *operators* or *operands*. Operators correspond to the control structures of the language, and to system and user-provided procedures, subroutines, and functions. Operands correspond to variables, constants, and labels. The basic Software Science metrics are then defined as follows:

η_1: the number of distinct operators,

η_2: the number of distinct operands,

N_1: the total number of operators, and

N_2: the total number of operands.

The *length* of a program is then defined as $N = N_1 + N_2$. It is a matter of taste, if not dispute, how the multiple keywords of iterative and conditional statements should be counted (e.g., does one count each of `while do`, and `endwhile` as three separate operators, or as a single "while loop" operator). Similarly controversial is the question whether tokens appearing in declarations should be counted, or only those appearing in imperative statements (opinion currently favors the first alternative). The Software Science metric N may be converted to SLOC by the relationship SLOC $= N/c$, where c is a language-dependent constant. For FORTRAN, c is believed to be about 7.

Software Science derives additional metrics from the basic terms. The *vocabulary* is defined as $\eta = \eta_1 + \eta_2$. Clearly it requires $\log_2 \eta$ bits to represent each element of the vocabulary in a uniform encoding, so the number of bits required to represent the entire program is roughly $V = N \log_2 \eta$. The metric

V is called the *volume* of a program and provides an alternative measure for the size of a program.

An alternative attempt to quantify the size of a program in a way that is somewhat independent of language, and that may get closer to measuring the semantic, rather than the purely syntactic or even merely lexical, content of a program is one based on function count: that is, a count of the number of *procedures* and *functions* defined by the program. Lisp programs are often described in this way—for example, a medium sized Lisp application might contain 10,000 function definitions. For some languages (e.g., Ada), the number of *modules* might be a more natural or convenient measure than function count.

4.2 Complexity Metrics

Two similarly sized programs may differ considerably in the effort required to comprehend them, or to create them in the first place. *Complexity* metrics attempt to quantify the "difficulty" of a program. Generally, these metrics measure some aspect of the program's control flow—there being some agreement that complicated control flow makes for a complex, hard-to-understand, program.

4.2.1 Measures of Control Flow Complexity

The simplest complexity metric is the *decision count*, denoted DC which can be defined as the number of "diamonds" in a program's flow chart. A good approximation to DC can be obtained by counting the number of conditional and loop constructs in a program—and this can be reduced to the purely lexical computation of adding the number of `if`, `case`, `while` and similar keywords appearing in the program.

An objection to this simple scheme is that it assigns a different complexity to the program fragment

```
if A and B then X endif
```

than it does to the semantically equivalent fragment

```
if A then if B then X endif endif.
```

This objection can be overcome by defining DC to be the number of *elementary predicates* in the program. Both the examples above contain two elementary predicates: A and B.

Much the best-known of all syntactic complexity measures is the *cyclomatic complexity* metric of McCabe [126]. This metric, denoted ν, is given by $\nu = e - n + 2$, where e is the number of edges and n the number of nodes in the control flow graph of the program. The cyclomatic complexity of a program is equivalent to the maximal number of linearly independent cycles in its control flow graph. (Actually, in the control flow graph modified by the addition of an edge from its exit point back to its entry point.) Clearly, the simplest control flow graphs (i.e., those corresponding to linear sequences of code) have $e = n - 1$ and hence $\nu = 1$. Motivated by testing considerations, McCabe suggested that a value of $\nu = 10$ as a reasonable upper limit on the cyclomatic complexity of program modules.

Despite their different origins, and the apparently greater sophistication of cyclomatic complexity, DC and ν are intimately related. Each "rectangle" in a program's flow chart has a single outgoing edge (except the exit node, which has none); each "diamond" has two outgoing edges. Therefore, the total number of edges in the flow chart is equal to the total number of nodes in the flow chart (n), plus the number of "diamonds" (DC) minus one (for the exit node). Thus $e = n + DC - 1$. Hence

$$\begin{aligned}\nu &= e - n + 2\\ &= (n + DC - 1) - n + 2\\ &= DC + 1.\end{aligned}$$

Thus, the highfalutin cyclomatic complexity measure turns out to be no different than the elementary decision count!

Related to syntactic complexity measures are "style" metrics [21, 22, 85, 156, 155]. These seek to score programs on their adherence to coding practices considered superior, in some sense. The details of these metrics vary according to the individual preference of their inventors. Generally, marks are added for instances of "good" style such as plenty of comments, and the use of symbolic constants, and deducted for instances of "bad" style such as explicit `goto`'s, and excessive module length. Some program style analysis systems also perform limited anomaly detection (e.g., variables declared but not used) and incorporate these into their scores. We discuss anomaly detection separately in Section 5.2.1 (page 40).

4.2.2 Measures of Data Complexity

The simplest data complexity metrics simply count the number of variables used by a program. Halstead's η_2 (the number of distinct operands) is a slightly more elaborate measure of the same type. Such metrics only measure the total number of different variables that are used in the program; they do not indicate how many are "live" (i.e., must be actively considered by the programmer) in any one place. A simple definition states that a variable is "live" in all statements lexically contained between its first appearance and its last. It is then easy to compute the number of variables that are live at each statement, and hence $\overline{LV}$—the *average* number of variables that are live at each statement.

Another approach to quantifying the complexity of data usage is to measure the extent of inter-module references. The work of Henry and Kafura [89, 88] is representative of this type. The *fan-in* of a module may be defined as the number of modules that pass data to the module, either directly or indirectly; similarly the *fan-out* may be defined as the number of modules to which the present module passes data, either directly or indirectly. The complexity of the interconnections to the module are then defined as $(\text{fan-in} \times \text{fan-out})^2$. Henry and Kafura then relate the overall complexity of a module within a program to both its length and the complexity of its interconnections by the definition: complexity $= \text{SLOC} \times (\text{fan-in} \times \text{fan-out})^2$.

4.3 Cost and Effort Metrics

Cost and effort metrics attempt to measure, or predict, the eventual size of a program, the cost required to construct it, the "effort" needed to understand it, and other such interesting and important attributes.

Halstead's "Software Science" postulates several composite metrics that purport to measure such attributes. Halstead hypothesized that the length of a program should be a function of the numbers of its distinct operands and operators. That is, it should be possible to predict N, the length of a program, from η_1 and η_2—the numbers of its distinct operators and operands, respectively. Halstead encoded a relationship between these quantities in his famous "length equation":

$$\widehat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2.$$

Going further, Halstead defined the "potential volume" V^*, of a program as the volume of the program of minimal size that accomplishes the same purpose as the original. The "difficulty" D of a program is then defined by $D = V/V^*$, and its "level" L is defined as the reciprocal of difficulty. It is then but a small step to hypothesize that the effort required to implement a program should be related to both its length and its difficulty, and to define the "effort" E required to implement a program by $E = D \times V (= V^2/V^*)$. Halstead claimed that E represented "elementary mental discriminations" and, based on the suggestion of a psychologist, J. Stroud, that the human mind is capable of making only between 5 and 20 elementary mental discriminations per second, he then proposed the equation $T = E/18$, where T is the time required to construct a program in seconds.

The practical application of these last few formulae depend on a method for computing or estimating the potential volume, V^*, for a program. Halstead proposed that potential volume could be *defined* by

$$V^* = (2 + \eta_2^*) \log_2(2 + \eta_2^*),$$

where η_2^* is the number of input/output operands, and *estimated* by

$$V^* = \frac{2\eta_2}{\eta_1 N_2} \times V.$$

Using this estimation, we obtain

$$D = \frac{\eta_1 N_2}{2\eta_2} \quad \text{and} \quad E = V \times \frac{\eta_1 N_2}{2\eta_2}.$$

In contrast to the scientific pretensions of Software Science, a widely practised method for predicting the time and effort required to construct a software project is simply to ask the opinions of those experienced in similar projects. A refinement of this approach is the "Delphi" technique in which several people prepare independent estimates and are then told how their estimates compare with those of the others. (In some variants, they discuss their estimates with each other). Next, they are allowed to modify their estimates and the process is repeated until the estimates stabilize. Often the estimates converge to a very narrow range, from which a consensus value may be extracted.

A composite cost estimation technique (which combines the use of expert judgment with statistical data fitting) is the COCOMO (COnstructive COst MOdel) of Boehm [27, 26]. There are three "levels" of the COCOMO model; the most successful seems to be the (modified) Intermediate Level. There are also three "modes" to the COCOMO model—for simplicity we will pick just one (the "semidetached" mode). This variant of the COCOMO predicts development effort E in man-months by the equation

$$E = 3.0 \times S^{1.12} \times \prod_{i=1}^{16} m_i$$

Rating	Multiplier
Very Low	0.75
Low	0.88
Nominal	1.00
High	1.15
Very High	1.40

Figure 4.1: Multipliers for the Reliability Cost Driver in the COCOMO Model

where S is the estimated KLOC and each m_i is the multiplier assigned to a particular "cost driver." Development time (in months) can be obtained from the COCOMO effort measure by the equation $T = 2.5E^{0.35}$. Each of the 16 cost drivers is evaluated by experts on a five point descriptive scale, and a numerical value for the corresponding multiplier is then extracted from a table. The table for the "reliability" cost driver is given in Figure 4.1. The 16 cost-drivers used in the (modified) Intermediate COCOMO model are: required software reliability, data base size, product complexity, execution time constraint, main storage constraint, virtual machine volatility, computer turnaround time, analyst capability, applications experience, programmer capability, virtual machine experience, programming language experience, modern programming practice, use of software tools, required development schedule, and volatility of requirements. It is the presence of the last cost driver (volatility of requirements) that distinguishes the modified Intermediate COCOMO model from the original Intermediate model. For all cost drivers, a "nominal" value corresponds to a multiplier of 1.00.

Not all programs are written from scratch; the effort to develop a program that includes a substantial quantity of reused code should be less than one of comparable total size that consists of all new code. Various modifications have been suggested to accommodate this factor; the simplest is to treat the total size of the program (S_e) as a linear combination of the number of lines of new (S_n) and "old" code (S_o):

$$S_e = S_n + kS_o,$$

where k is an appropriate constant—a value of $k = 0.2$ has been found reasonable [48], though this could vary if the "old" code requires significant adaptation.

4.4 Discussion of Software Metrics

The attempt to measure and quantify the properties and characteristics of programs is a laudable one—and one on which considerable effort has been expended (see, for example, the survey and bibliography by Waguespack and Badlani [189]). However, the substance and value of most of this work is open to serious question. Kearney *et al.* marshal the arguments against the standard complexity measures in their critique [106]. Firstly, they observe that existing complexity measures have been developed in the absence of a theory of programming behavior—there is no comprehensive model of the programming process that provides any intellectual support for the metrics developed. Any reasonable theory of programming behavior would consider not only the program, but also the programmer (his skill and experience), the programming environment, and the task that the program is to accomplish. Yet existing complexity measures consider only the program itself, and ignore its context. Furthermore, most complexity measures examine only the surface features of programs—their lexical and syntactic structure—and ignore the deeper semantic issues. Indeed, most metrics research seems stuck in the preoccupations of the 1960's: it equates complexity with control flow (c.f. the "structured programming" nostrums of the time) and seems unaware that the serious work on programming methodology over the last 15 years has been concerned with the deeper problems of hierarchical development and decomposition, and with the issues of data structuring, abstraction, and hiding. Some more recent work does attempt to address these issues—for example, Bastani and Iyengar [15] found that the perceived complexity of data structures is strongly determined by the relationship between the concrete data representation and its more abstract specification. They conjectured that a suitable measure for the complexity of a data structure is the length of the mapping specification between its abstract and concrete representations—i.e., a *semantic* measure.

Kearney *et al.* also criticise the fact that complexity measures are developed without regard to the intended *application* of the measure. There is a significant difference between *prescriptive* uses of such metrics (using them to "score" programmer's performance), and merely *descriptive* uses. A much greater burden of proof attends the former use, since we must be sure that techniques that improve the score really do improve the program. Kearney *et al.* find much to criticize in the methodology and interpretation of experiments that purport to demonstrate the significance and merit of selected complexity measures, and thereby cast serious doubt on the wisdom of using complexity metrics in a prescriptive setting.

Criticism such as that levied by Kearney and his colleagues against much of the work in complexity metrics is acknowledged by the more thoughtful practioners. Shen, for example, introducing a magazine column dedicated to metrics [172], writes:

> "Metrics researchers in the 1980's are generally less optimistic than their colleagues in the 1970's. Even though the pressure to find better metrics is greater because of the greater cost of software, fewer people today are trying to formulate combinations of complexity metrics that they relate to some definition of productivity and quality. Instead they set very narrow goals and show whether these goals are met using focussed metrics ... one narrow goal would be to test software thoroughly. An appropriate metric might be some measure of test coverage."

The issue addressed in this report is software quality; size and complexity metrics are of interest only in so far as they contribute to the estimation or improvement of software quality. As far as estimation is concerned, metrics research does not yet seem able to provide accurate predictions for the properties of economic interest (number of faults, cost to maintain or modify) from the measurable properties of the program itself. The most accurate and best validated predictive techniques seem to be the crudest—for example 20 faults per KLOC for a program entering unit test [134, page 121]. Even these techniques are likely to need considerable calibration and re-validation when used in new environments.

The application of software metrics to improving (as opposed to predicting) the quality of software is even more questionable. Given their lack of substantiation, the use of such metrics prescriptively seems highly ill-advised [106]. Their least controversial application might be as components of "program comprehension aids." By these, we mean systems that assist a programmer to better comprehend his program and to master its complexities—especially those due to scale. Examples of such aids range from simple prettyprinters and cross-reference generators, to more semantically oriented browsers. Descriptive complexity metrics might assist the programmer in identifying parts of the program worthy of restructuring or simplification in much the same way that profiling tools help to identify the places where optimization can be applied most fruitfully.

Effort metrics are rather less controversial and easier to validate than complexity metrics—their purpose, if not their effectiveness, is clear: to enable the cost and duration of a programming task to be estimated in advance. However, the efficacy of existing effort metrics seems dubious at best. The COCOMO model, for example, seems to perform very poorly when applied to projects other than those used in its own validation. Kemerer [108], for example, reported an *average* error of over 600%, and also found that the Basic COCOMO mode outperformed the more complex Intermediate and Detailed modes. Conte *et al.* [48] suggest that the COCOMO model is *too* complicated, and involves too many parameters that require estimation. Kemerer's data [108] suggests that a much simpler "function count" method, similar to the Software Science metric, actually performs better than COCOMO.

Chapter 5

Testing

Testing subjects software to scrutiny and examination under the controlled conditions of a "test phase" before it is released into its "operational phase." The purpose of testing is to discover faults and thereby prevent failures in the operational phase. Discussion of the efficiency of a test strategy must relate the actual cost of testing to the avoided costs of the operational failures that are averted.

Testing can take two forms: *static*, and *dynamic*, though the unadorned term "testing" generally refers only to the latter. Static testing is that which depends only on scrutiny on the program text (and possibly that of its specifications and requirements also). Dynamic testing, on the other hand, observes the behavior of the program *in execution*. We will describe dynamic testing first, then three variants of static testing: anomaly detection, code walk-throughs, and formal verification.

5.1 Dynamic Testing

The input space of any interesting program is so large (if not infinite) that it is infeasible to examine the behavior of the program on all possible inputs. Given that only a fraction of the total input space can be explored during testing, systematic testing strategies attempt to maximize the likely benefit while minimizing the cost of testing. Two tactics underlie most systematic testing strategies: the first is to reduce cost by partitioning inputs into groups whose members are *likely* to have very similar behavior, and then testing only one member from each group (this is called "equivalence partitioning"); the second is to maximize benefit by selecting test data from among inputs that are considered to have an above average likelihood of revealing faults. The second of these tactics is not universally admitted to be a good idea: the divergence of opinion occurs between those who are primarily interested in enhancing reliability (actually, in reducing the cost of unreliability), and those who are interested in finding bugs. Concentration on input states believed to be fault-prone often leads testers to examine the boundaries of expected ranges of values, or cases that they believe the designers may have overlooked. This tactic may be successful in finding bugs, but because those bugs are manifested by rare or unusual inputs, they may not be encountered during normal operation and may therefore contribute little to the incidence of operational failures.

5.1.1 Random Test Selection

This argument may be developed into a case for *random* testing. If individual failures are all assumed to have similar cost, then the cost of unreliability is proportional to failure intensity. Since the cost of testing is primarily influenced by the execution time taken to perform the tests themselves, the optimum test strategy is one which yields the greatest reduction in failure intensity for a given amount of execution time devoted to testing. This argues for concentrating on finding and eliminating those faults that lead to failures on commonly occurring inputs—and therefore suggests the strategy of selecting test cases randomly, with a probability distribution equal to that expected to occur during operation. This approach has the advantage that it essentially duplicates the operational profile and therefore yields failure intensity data that approximates that which would have been found if the program had been released into operation at that time. The failure intensity data obtained during testing should therefore provide accurate estimates for the reliability formulas developed in Section 3.1 (page 10).

A criticism of the random testing strategy is that it is wasteful: the *same* input may be tested several times. However, it is usually easy to record each input and to avoid repeating tests already performed. By reference to the classic sampling problems of statistics, such methods are often called "test selection without replacement." If tests are selected and performed without replacement, then the probability of selecting any particular input value becomes uniform. However, the *order* in which inputs are selected will still follow their expected probability of occurrence in operation.

Although random test selection without replacement should be more cost-effective than a purely random strategy, such tests do not follow the expected operational profile and therefore do not provide an accurate estimate of the failure intensities to be expected in practice. It is possible to take account of this divergence between the testing and the operational profiles as follows.

Define C, the *testing compression factor* to be the ratio of the execution time required to cover the entire input space during operation, to that required during test. Let $|I|$ be the size (cardinality) of the input space, p_k the probability of occurrence of input k during operation, $p_{\min}$ the probability of occurrence of the *least* likely input, and let τ_k be the execution time required for input k. If the entire input space is covered during operation, then the least likely input must be executed at least once. Hence the expected frequency of execution of input k is $p_k/p_{\min}$ and so the expected total time to cover the input space during operation is $\sum_{k=1}^{|I|} \frac{p_k}{p_{\min}} \tau_k$. Since only one execution of each input is required during test (without replacement), the execution time to cover the input space during test is only $\sum_{k=1}^{|I|} \tau_k$, and so the testing compression factor is given by:

$$C = \frac{\sum_{k=1}^{|I|} \frac{p_k}{p_{\min}} \tau_k}{\sum_{k=1}^{|I|} \tau_k}. \tag{5.1}$$

If all τ_k are equal, then (5.1) simplifies to

$$C = \frac{1}{|I| \cdot p_{\min}}.$$

If p_k is assumed to be inversely proportional to k, then it can be shown that as $|I|$ grows from 10^3 to 10^9, C grows slowly from just less than 10 to a little over 20.

If a value for the testing compression factor can be calculated or estimated using the methods given above, then the reliability formulas of Section 3.1 (page 10) can be employed to extrapolate from the testing to the operational phase if execution times are multiplied by C (alternatively, failure intensities can be *divided* by C) in order to convert observations made during test to those expected during operation.

The pure random test selection strategy described above assumes that all failures have equal cost. In practice, some failures may be more expensive than others; some may be unconscionable (perhaps endangering human life). In these cases, the random test strategy may be modified in order to attempt to identify and provide early tests of those inputs which might provoke especially costly failures. The technique of software fault-tree analysis may be useful in identifying such critical inputs [120, 121]–see Section 5.2.4.

5.1.2 Regression Testing

The desirability of test selection without replacement is considerably diminished if we admit the possibility of imperfect debugging. Under the assumption that debugging is imperfect, the repair that is initiated following each failure may not only fail to remove the fault that caused it, but may actually introduce *new* faults. These faults may cause failure on inputs that previously worked satisfactorily. Under a *regression testing* regime, previously tested inputs are repeated whenever a repair (or other modification) is made to the program. Under *strict* regression testing, *all* previous tests are repeated; under less strict regimes, some subset of those tests (usually including all those that provoked failure) are repeated. Models of the software testing process under various assumptions are discussed by Downs [60].

5.1.3 Thorough Testing

The motivation behind random testing is find and remove the most costly faults (generally interpreted as those that most frequently lead to operational failure) as quickly and as cheaply as possible. Test selection without replacement and equivalence partitioning may be used to enhance the testing compression factor and hence the cost-effectiveness of the process. The motivation behind what we will call *thorough testing*, on the other hand, is to find *all* the faults in a program (as efficiently as possible). The theoretical framework for thorough testing was established by Goodenough and Gerhart in a landmark paper [78].

Let F denote the program being considered, and D its input space. If $d \in D$ then $\textsc{ok}(d)$ denotes that F behaves correctly when given input d. We say that the *test set* $T \subseteq D$ is *successful* if F behaves correctly on all its members, that is $\textsc{successful}(T) \overset{\text{def}}{=} (\forall t \in T : \textsc{ok}(t))$. A *correct* program is one that behaves correctly throughout its input space—i.e., one that satisfies $\textsc{successful}(D)$. A *faulty* program is one that is not correct—i.e., one that satisfies $\neg\textsc{successful}(D)$. A *thorough* test set is one which, if successful, guarantees that the program is correct, that is

$$\textsc{thorough}(T) \overset{\text{def}}{=} \textsc{successful}(T) \supset \textsc{successful}(D).$$

If C is a *test selection criterion* and the test set T *satisfies* C, then we write $\textsc{satisfies}(T, C)$. We would like test criteria to be *reliable* and *valid.* A criterion is reliable if all test sets that satisfy it give the same result:

$$\begin{aligned}\textsc{reliable}(C) \quad &\overset{\text{def}}{=} \quad (\forall T_1, T_2 \in D : \textsc{satisfies}(T_1, C) \wedge \textsc{satisfies}(T_2, C) \\ &\qquad\qquad \supset \textsc{successful}(T_1) = \textsc{successful}(T_2)).\end{aligned}$$

A criterion is valid if it is capable of revealing all faulty programs. That is, if F is faulty, then there should be some T that satisfies C and fails on F:

$$\textsc{valid}(C) \overset{\text{def}}{=} \neg\textsc{successful}(D) \supset (\exists T : \textsc{satisfies}(T, C) \wedge \neg\textsc{successful}(T)).$$

Given these definitions, Goodenough and Gerhart were able to state the *Fundamental Theorem of Testing*:

$$\begin{aligned}&(\exists T, C : \textsc{reliable}(C) \wedge \textsc{valid}(C) \wedge \textsc{satisfies}(T, C) \wedge \textsc{successful}(T)) \\ &\qquad\qquad \supset \textsc{successful}(D).\end{aligned}$$

In words, this says that if a successful test set satisfies a reliable and valid criterion, then the program is correct. Another way of stating this result is that a test is thorough if it satisfies a reliable and valid test criterion.

This result is called "Fundamental," not because it is profound (indeed, its proof is a trivial consequence of the definitions, and is omitted for that reason), but because it captures the basic idea underlying all attempts to create thorough test strategies, namely the search for test criteria that are both reliable and valid, yet economical in the sense that they can be satisfied by relatively small test sets.

Unfortunately, there are several problems with the practical application of these definitions [192]. First of all, the concepts of reliability and validity are not independent. A test selection criterion is valid if, given that the program is faulty, at least one test set satisfying the criterion is unsuccessful. Therefore, if a test selection criterion is *invalid*, all test sets that satisfy it must be successful. Hence they will all give the same result, and so the criterion is reliable. Thus, *all* test selection criteria are either valid or reliable (or both) [192]. (Note also that if F is correct, then *all* criteria are both reliable and valid for F.)

Next, the concepts of validity and reliability are relative to a single, given program. A criterion that is valid and reliable for program F may not be so for the slightly different program F'. Furthermore, since F' may result from correcting a bug in F, the reliability and validity of a test selection criterion may not be preserved during the debugging process.

A plausible repair to this deficiency is to construct modified definitions which quantify over all programs. Thus, a test selection criterion is said to be *uniformly valid* if it is valid for all programs F, and *uniformly reliable* if it is reliable for all programs F. Unfortunately, a criterion is uniformly reliable if and only if it selects a single test and uniformly valid if and only if the union of the test sets that satisfy it is the entire input space D. Hence a criterion is uniformly reliable and valid if and only if it selects the single test set D. Thus we see that the notions of uniform reliability and validity are unhelpful [192]. A more recent attempt to axiomatize the notion of software test data adequacy is described by Weyuker [191].

In theory, the construction of a reliable and valid test selection criterion for a program is equivalent to proving the formal correctness of that program—a very hard problem. In practice, constructing such a criterion is virtually impossible without some knowledge, or at least some assumptions, about the faults that it may contain. Thus, the search for a theoretical basis for thorough test selection has shifted from the original goal of demonstrating that *no* faults are present, to the more modest goal of showing that *specified classes* of faults are not present. In practice, the goal is often reduced to that of finding as many faults as possible. This is accomplished by systematic exploration of the state space based on one (or both) of two strategies known as structural testing, and functional testing, respectively.

5.1.3.1 Structural Testing

Structural testing selects test data on the basis of the program's *structure*. As a minimum, test data are selected to exercise all *statements* in the program; a more comprehensive criterion requires that all outcomes of all decision points should be exercised. In terms of the control flow graph of the program, the first of these criteria requires that all *nodes* must be visited during testing; the second requires that all *edges* must be traversed, and properly includes the first (unless there are isolated nodes—which surely represent errors in their own right).

Test data that simply visits all nodes, or traverses all edges, may not be very effective: not many faults will be so gross that they will be manifest for *all* executions of the offending statement. Generally, faults are manifest only under certain conditions, determined by the context—that is to say, the values of the accessible variables—in which the offending statement is executed. Context is established in part by the execution path taken through the control flow graph; the *path* testing criterion requires test data that will exercise all possible paths through the program.

There are (at least) two problems with the all-paths criterion. Firstly, a path does not uniquely establish an execution context: two different sets of input values may cause the same execution path to be traversed, yet one set may precipitate a failure while the other does not. Secondly, any program containing loops has an infinite number paths. Some further equivalence partitioning is therefore needed. One plausible strategy is to partition execution paths through each cycle in the control flow graph into those involving zero, one, and many iterations.

A more sophisticated strategy considers the *data flow* relationships in the program. Data flow analysis, which was first studied systematically for its application in optimizing compilers, considers how values are given to variables, and how those values are used. Each occurrence of a variable in a program can be classified as a *def*, or as a *use*: a *def* occurrence (for example, an appearance in the left hand side of an assignment statement) assigns a value to a variable; a *use* occurrence (for example, an appearance in the right hand side of an assignment statement) makes use of the value of a variable. *Use* occurrences may be further distinguished as *c-uses* (the value is used in a $\underline{\text{c}}$omputation that assigns a value to a variable) and *p-uses* (the value is used in a $\underline{\text{p}}$redicate that influences control flow). For example, in the program fragment

```
if x = 1 then y := z endif,
```

x has a *p-use*, z a *c-use*, and y a *def*.

Rapps and Weyuker [154] proposed a family of path selection criteria based on data flow considerations. The *all-defs* criterion requires that for every *def* occurrence of, say, variable x, the test data should cause a path to be traversed from that *def* occurrence to some *use* occurrence of x (with no intervening *def* occurrences of x). The *all-p-uses* criterion is similar but requires paths to be traversed from that *def* occurrence to every *p-use* occurrence of x that can be reached without intervening *def* occurrences of x. Definitions for the criteria *all-c-uses*, and *all-uses* are similar. A hybrid criterion is *all-p-uses/some-c-uses*—this is the same as *all-p-uses*, except that if there are no *p-uses* of x subsequent to a given *def* occurrence, then a path to *some* subsequent *c-use* should be included. The *all-c-uses/some-p-uses* criterion is defined dually.

The *all-uses* criterion is a comprehensive one but, like the *all-paths* criterion, may require an excessive, or infinite, number of test cases. The *all-defs* criterion seems an attractive compromise from this point of view. Systematic test selection criteria should surely draw on both data and control flow perspectives—so a very reasonable criterion would seem to be one that encompassed both *all-defs* and *all-edges*. Rapps and Weyuker [154] showed that these are independent criteria—neither implies the other. This tends to confirm the belief that both are important, but complicates test selection since two very different criteria are involved. It would be nice if a single criterion could be found that would include both *all-defs* and *all-edges*. Rapps and Weyuker [154] showed that *all-p-uses/some-c-uses* has this property and recommended its use. Ntafos [142] provides a comprehensive comparison of the relative coverage of these and other structural testing strategies. Clarke [45] provides a more formal examination.

5.1.3.2 Functional Testing

Functional testing selects test data on the basis of the *function* of the program, as described in its requirements, specification, and design documents. Generally speaking, the detailed characteristics of the program itself are not considered when selecting test data for functional testing, though general aspects of its design may be.

Functional testing treats the program as a "black box" which accepts input, performs one of a number of possible functions upon it, and produces output. Based on the relevant requirements documents, classifications and groupings are constructed for the input, output, and function domains. Test data are then constructed to exercise each of these classifications, and combinations thereof. Typically, an attempt is made to select test data that lie well inside, just inside, and outside each of the input domains identified. For example, if a program is intended to process "words" separated by "whitespace," we might select test data that consists of zero words, one word, several words, and a huge number of words. Similarly, we would select words consisting of but a single letter, a few letters, and very many letters—not to mention words containing "illegal" letters. The "whitespace" domain would be explored similarly.

Functional testing for a given program may take many forms, depending on which of its requirements or design documents are used in the analysis. Howden [93] argues that for maximum effectiveness, functional testing should consider the "high-level" design of a program, and the context in which functions will be employed. For example, if a particular function $f(x)$ has the domain $x \in (-\infty, \infty)$, then the test values $+k$ and $-k$, where k is a very large value, are likely to suggest themselves. But if the function is used in the context

```
if x > -2 then f(x) endif,
```

the test value $-k$ will not exercise f at all, and faults manifest by negative values for x, for example, will remain undiscovered. Given its *context*, appropriate test values for f might be $-2 \pm \epsilon$ and $+k$.

If a formal specification is available for a program, it may be possible to derive a functional testing strategy from that specification in a highly systematic fashion [87]. Other systematic functional testing strategies are described by Mandl [125] and by Ostrand and Balcer [145].

5.1.4 Symbolic Execution

The model of testing described so far assumes that test data are presented to the program and the results produced are compared with those expected. In practice, however, programmers do not merely examine the final output of the program, but often instrument or modify the program under test so that traces of its control flow and of the intermediate values of its variables are generated during execution. These traces greatly assist the programmer in determining whether the actual behavior of the program corresponds to that intended. Since they provide a peek into the inner workings of the program, traces often yield much more insight than the single datum points provided by tests that only consider input-output values.

Symbolic Execution constitutes a systematic technique for generating information about the inner workings of a program. The idea is to allow program variables to take symbolic values and to evaluate the functions computed by program statements symbolically also. Symbolic execution bears a similar relationship to conventional execution as algebra does to arithmetic. Consider, for example, the following (Fortran) program fragment from a subroutine to compute the sine of an angle:

```
I=3
TERM=TERM*X**2/(I*(I-1))
SUM=SUM+(-1)**(I/2)*TERM
```

We can compute the final values of variables `I, TERM` and `SUM`, given the initial assignments `TERM=SUM=X` to be `I=3`, `TERM=X**3/6`, and `SUM=X-X**3/6`.

The process performed, statement by statement, is to substitute the current symbolic values into each variable in the right hand side of each assignment statement, simplify the resulting expression, and let this become the new symbolic value of the variable on the left hand side. In a full symbolic execution system, it is necessary to be able to carry the computation forward through branches. Typically, the programmer indicates the paths he wishes to explore and the symbolic execution system asserts the necessary truth of the appropriate test predicates.

The output produced by a symbolic execution system consists of the symbolic values accumulated in its variables through execution of a selected path. The programmer can compare these values with those expected. This is very convenient and appropriate for some computations, less so for others. For example, Howden [91] cites a subroutine to compute the sine function using the Maclaurin series $sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots$ Two iterations round the main loop in the subroutine yield the symbolic value `X-X**3/6+X**5/120` for the variable `SUM` in which the result is accumulating. This provides much more useful information than would the output values for a couple of isolated points.

Another use for symbolic execution is to help partition the input domain by the execution paths invoked. This is helpful in the generation of test data to satisfy path coverage criteria. Symbolic execution systems and their applications are described by several authors [91, 111]; one of the first such systems was developed in the Computer Science Laboratory of SRI [31].

5.1.5 Automated Support for Systematic Testing Strategies

Given a systematic test selection criterion, the question naturally arises: how does one generate test data satisfying the criterion? For functional testing, there seems little alternative to generating the data by human inspection: generally the requirements and design documents from which functional test data are derived are not formal and not amenable to mechanical analysis. However, it is feasible that automatic test data selection could be performed once human inspection had performed the classification of the input domain: that is to say, human skill would be used to identify the primitive classifications, but the generation of combinations of elements of each classification would be performed mechanically.

Unlike functional testing, structural testing is ideally suited to automation. The program text is a formal object and well suited to systematic exploration. Unfortunately, considerable computation is needed to calculate the input that will exercise a particular path: symbolic execution is generally employed in order to derive the "path predicates" associated with the path, then solutions must be found to the systems of numerical inequalities that they induce. Consequently, most test case generators are severely limited in the class of programs and/or in the class of test criteria that they support. Ince [97] provides a modern survey of the field and also suggests that systematic use of randomly generated test data could provide very good coverage at low cost. The idea, which is elaborated in a short note [98], starts from the observation that relatively small sets of random test data seem to provide quite good coverage [61]. For example, with ten programs ranging in size from 12 to 102 branches, random test data achieved an average branch coverage of 92.80% [98]. The average size of the test sets needed to achieve this coverage was 12.8, with the largest being 112. Since the test sets were randomly generated, they were far from optimal, and contained subsets that provided the same coverage as the whole set. Using zero-one integer linear programming, the minimum such subsets were found for each test set. These were found to average only 3.1 in size, with the test set of size 112 being reduced to only 10. Thus, the suggestion is to generate random test data until adequate coverage is achieved relative the chosen structural testing criterion (or until further tests result in little increased coverage). Coverage is measured by instrumenting the program under test. The randomly generated test set is then reduced to minimum size using zero-one integer linear programming, and the test results obtained using this subset are examined.

5.2 Static Testing

Dynamic testing is an important method of *validation*; static testing tends to address the complementary problem of *verification*. Static testing subjects the program text (and its accompanying requirements and specification documents) to scrutiny and review in order to detect inconsistencies and omissions. The scrutiny may operate within a single level, in order to detect internal inconsistencies (this is the basis of anomaly detection, discussed in the next section), or across two levels. In the latter case, the purpose is to establish that the lower level specification (or program) fully and exclusively implements the requirements of its superior specification. Everything in a lower level specification should be *traceable* to a requirement in a higher level specification; conversely, every requirement should ultimately be realized in the implementation.

5.2.1 Anomaly Detection

The idea behind anomaly detection is to look for features in the program that probably indicate the presence of a fault. For example, if a programmer declares a variable name but never otherwise refers to it, he is guilty of carelessness at the very least. More ominously, this situation may indicate that the programmer anticipated a need for the variable early in the programming effort, but later forgot to deal with the anticipated circumstance—in which case the existence of a genuine fault may have been detected. Anomaly detection refers to the process of systematically searching for "suspicious" quirks in the syntactic structure of the program, or in its control or data flow.

At the syntactic level, the example given above is typical of a very fruitful technique: searching for identifiers that are declared but not used. The dual problem of identifiers that are used but not identified usually violates the definition of the programming language concerned and will be caught by its compiler. Other examples of anomalies that are likely to indicate faults include supplying a constant as an actual parameter to a procedure call in which the corresponding formal parameter is modified, and subscripting an array with a loop index whose bounds exceed those declared for the array.

Control flow anomalies include unreachable sections of program, and loops with no exit. These circumstances indicate certain errors; other control flow anomalies may merely indicate "bad style"—for example, jumping into the middle of a loop.

Among the most effective of techniques for anomaly detection are those based on data flow analysis. For example, if it can be determined that the value of a program variable may be used before it has been given a value (i.e., if there is a path to a *use* occurrence that does not pass a *def* occurrence), then it is very likely that a fault is present. Dually, *def* occurrences that do not lead to a subsequent *use* occurrence are also suspect, as are paths that have two *def* occurrences of a variable, with no intervening *use* occurrence.

Information flow analysis is related to data flow analysis, but is rather more sophisticated in tracing the influence between program variables and statements. For example, in the program fragment

```
if x = 0 then y := 1 endif
```

there is no *data* flow from x to y, but the value of x certainly influences the subsequent value of y, and so there is considered to be a flow of *information* from x to y. Information flow analysis is used routinely in computer security verification [57, 176]; its application to more general analysis and anomaly detection is promising [20].

Automated tools have been developed to perform detection of anomalies of the types described above for programs written in a variety of languages [144, 194].

5.2.2 Structured Walk-Throughs

Structured walk-throughs are a method for the manual inspection of program designs and code. The method requires far more than mere "eyeballing": it is highly structured and somewhat grueling—its intensity is such that no more than two two-hour sessions per day are recommended.

As first described by Fagan [63] (see [64] for an update), four participants are required—the *Moderator*, the *Designer*, the *Coder/Implementor/*, and the *Tester*. If a single person performed more than one role in the development of the program, substitutes from related projects are impressed into the review team. The review team scrutinizes the design or the program in considerable detail: typically, one person (usually the coder) acts as a "reader" and describes the workings of the program to the others, "walking through" its code in a systematic manner so that every piece of logic is covered at least once, and every branch is taken at least once. Intensive questioning is encouraged, but it is the Moderator's responsibility to ensure that it remains constructive and purposeful. As the design and its implementation become understood, the attention shifts to a conscious search for faults. A checklist of likely errors may be used to guide the fault finding process.

One of the main advantages of structured walk-throughs over other forms of testing is that it does not require an executable program, nor even formal specifications—it can be applied early in the design cycle to help uncover errors and oversights *before* they become entrenched.

5.2.3 Mathematical Verification

Mathematical verification is the demonstration of consistency between two different descriptions of a program. Usually, one description is the program code itself and the other its specification, though the method can equally well be applied to two specifications at different levels of detail. This terminology is entirely consistent with the notion of "verification" defined in Section 2.1 (page 7), but the presence of the adjective "mathematical" qualifies this particular style of verification as a mathematical activity, in which the two program descriptions are treated as formal, mathematical texts and the notion of "consistency" that is to be demonstrated between them is also a formal, mathematical one (for example, that of "theory interpretation" or of a homomorphism).

Mathematical verification can be performed at various levels of formality. As stated above, *mathematical* verification means that the process is grounded on formal, mathematical principles. But just as conventional mathematicians do not reduce everything to the level of *Principia Mathematica*, so it is entirely reasonable to perform mathematical verification "rigorously," but informally—that is to say, in the style of a normal mathematical demonstration.[1]

There are many worthwhile points along the spectrum of formality in mathematical verification. At the most informal end is the "Clean Room" methodology espoused by Mills [131]. Though informal in the sense that the process is performed manually at the level of ordinary mathematical discourse, the *process* itself is highly structured and formalized. Its goal is to prevent faults getting into the software in the first place, rather than to find and remove them once they have got in. (Hence the name of the methodology—which refers to the dust-free environment employed in hardware manufacturing in order to eliminate manufacturing defects.)

The two cornerstones of the Clean Room methodology are mathematical verification and statistical quality control. The first requires that precise, formal, specifications are written for all program components and that a detailed mathematical argument is provided to justify the claim that the component satisfies its specification. If a verification is difficult or unconvincing, the program is revised and simplified so that the argument for its correctness becomes more perspicuous.

Mathematical verification subsumes structural testing in the Clean Room methodology, although functional testing is still performed. However, the

[1]What we are calling *mathematical* verification is often called *formal* verification; we have chosen our terminology to avoid having to talk about "informal" formal verification which, if it is not an oxymoron, is undoubtedly a solecism. Our usage also avoids confusion with some notions of "formal" verification that are anything but mathematical—the adjective "formal" being used in this case to refer to a highly structured *process* of verification.

major testing effort performed in the Clean Room is random testing for the purposes of statistical quality control. Software specifications in the Clean Room methodology include not only a description of the functions to be supported by the software, but a probability distribution on scenarios for its use. The Clean Room methodology then prescribes a testing procedure and a method for computing a certified statistical quality measure for the delivered software.

The mathematical verification technique used in the Clean Room methodology is called "functional verification" and is different from that employed in "classical" mathematical verification. The "rigorous" techniques of Jones [101], and of the Oxford school [86] are based on more conventional forms of mathematical verification than the Clean Room methodology, and are also more formal, but share much of their motivation with the Clean Room approach. An empirical evaluation of the Clean Room methodology has recently been reported by Selby *et al* [170].

Beyond the techniques described above come the *totally* formal methods. These generally employ the assistance of an automated specification and verification environment: the sheer quantity of detail entailed by complete formality is likely to render verification less, rather than more, reliable unless mechanical assistance is employed. Formal specification and verification environments generally provide a formal specification language, a programming language, a means of reducing the problem of establishing consistency between a program and its specification to a putative theorem in some formal system, and a mechanical theorem prover for seeking, or checking, proofs for such theorems. Three or four systems of this type have been developed [110].

All verification methods are vulnerable to two classes of error. First is the possibility of a flaw in the verification itself—the demonstration of consistency between the program and its specification may be flawed, and the two descriptions may, in fact, be inconsistent. The second class of error arises when the verification is sound, but irrelevant, because the specification does not reflect the actual user requirements (i.e., the system may satisfy the verification process, but fail validation)

It is in order to avoid the first class of error that truly formal, and mechanically assisted, formal verification is generally recommended. There are, however, drawbacks to this approach. Firstly, in order to be amenable to mechanization, rather restrictive specification and programming languages, built on a very elementary formal system (usually a variation on first-order predicate calculus), must usually be employed. Because of their lack of convenient expressiveness, such languages and logics may make it difficult for the user to say what he really intends—so that instead of concentrating on *what* he wants to say, he has to spend all his effort on *how* to say it—with the consequent danger that he may fail to say it correctly, and so fall into the second kind of error. Similarly, the user may have to spend considerable ingenuity, not in proving his theorems directly, but in persuading a mechanical theorem prover to prove them for him. This problem is compounded by the fact that most verification systems do not allow the user to reason about programs directly (as he would do if performing the proof by hand), but reduce the question of consistency to a (generally large) number of (generally lengthy) formulas called "verification conditions" that rob the user of much of his intuition concerning the program's behavior. This latter difficulty should not affect the reliability of the process (provided the theorem prover is sound), but will adversely affect its economics. SRI's EHDM system attempts to overcome these difficulties by providing a very expressive specification language and powerful underlying logic (based on multi-sorted higher order logic) together with the ability to reason about programs directly (using the Hoare, or relational, calculus) [165].

5.2.3.1 Executable Assertions

The task of proving consistency between a program and its specification is generally broken down into more manageable steps by embedding assertions at suitable points in the program text and proving that the assertions will always be satisfied when the locus of control passes these points during execution. An interesting alternative to *proving* the assertions à priori is to *test* them during execution and to halt the program with an error message if any assertion fails. This can permit a fairly simple proof of consistency between a program and the weakened specification "X or fail," where "X" was the original specification. Variations on this theme include the use of executable assertions during conventional dynamic testing [11], and in a dynamic variant of anomaly detection [40]. In the former case, the presence of the assertions allows the testing process to probe the "inner workings" of the program, rather than merely its input-output behavior (recall our discussion of symbolic execution in Section 5.1.4 on page 37). In the latter case, instrumenting the program with executable assertions allows data flow anomaly detection to be performed without requiring a data flow analyzer for the programming language concerned. Of course, this approach can only perform anomaly detection on paths actually executed.

More radical techniques that have much in common with executable assertions include "provably safe" programming [10], and the "recovery block" approach to software fault tolerance—in which an "acceptance test" (effectively an executable assertion) governs the invocation of alternative program components to replace those that have failed [9]. An experiment by Anderson [8] showed promise for recovery blocks (70% of software failures were eliminated, and MTBF was increased by 135%), but found that acceptance tests are hard to write. In another study [39], 24 students were hired to add self-checks to programs containing a total 60 known faults (there were 8 different programs, each was given to three students). Only 9 out the 24 self-checking programs detected any faults at all; those that did find faults found only 6 of the 60 known faults, but they also discovered 6 previously unknown faults (in programs which had already been subjected to one million test-cases). Sadly, 22 new faults were introduced into the programs in the process of adding the self-checks.

5.2.3.2 Verification of Limited Properties

A common and familiar example of executable assertions is the "range check" generally compiled into array subscripting operations. Though a valuable safety net, these checks can be very expensive when they appear in the inner loops of a program. An approach which "turns the tables" on the relationship between formal verification and executable assertions is to *prove* that subscripting errors, and other simple forms of run-time error, cannot occur [75]. This is an example of an important variation on the application of formal verification.

Conventionally, the goal of formal verification is understood to be "proof of correctness." We have been careful to make a more careful and accurate statement—namely, that it provides a demonstration of consistency between a program and its specification—but we have implicitly assumed that the specification concerned is one that provides a full description of the functionality required of the program. This need not be the case, however. The methods of formal verification can be applied equally well when the specification is a limited, weak, or partial one: instead of proving that the program does "everything right," we can attempt to prove only that it does *certain* things right, or even that it is does *not* do certain things wrong. In fact, the properties proved of a program need not be functional properties at all, but can be higher order (e.g., "security"), or structural.

The limited properties to be proved may be chosen because of their tractability—i.e., because formal verification is among the most cost-effective ways of ensuring those properties—or because of their importance. The absence of array subscripting errors is an example of the first class; security exemplifies the second. In particular, security is an example of a "critical property": a property considered so important that really compelling evidence must attest to its realization. What constitutes a critical property is something that can only be determined by the customer (and the law!), but it will generally include anything that could place human life, national security, or major economic assets at risk.

Yet another variation on formal verification is to prove properties about the structural properties of programs. For example, a specification may assert that the program should have a certain structure, or that a certain structural relationship should exist among some of its components (e.g., one may use the other, but not vice-versa). Formal verification of these properties guarantees that the program respects the structural properties given in its specification. One advantage of this style of verification is that it can be performed with complete formality, but without burdening the user with details. The PegaSys system developed in the Computer Science Laboratory of SRI [133] supports the use of *pictures* as *formal* specifications of system structure, and hides all the details of theorem proving from the user.

5.2.4 Fault-Tree Analysis

Reliability is not the same as safety, nor is a reliable system necessarily a safe one. Reliability is concerned with the incidence of *failures*; safety concerns 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 failures are defined in terms of system services, safety is defined 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.

Leveson [119, 120, 121] has discussed the issue of software safety at length and proposed that some of the techniques of system safety engineering should be adapted and applied to software. First, it is necessary to define some of the terms used in system safety engineering. *Damage* is a measure of the loss in a mishap. A *hazard* is a condition with the potential for causing a mishap; the *severity* of a hazard is an assessment of the worst possible damage that could result, while the *danger* is the probability of the hazard leading to a mishap. *Risk* is the combination of hazard severity and danger. *Software Safety* is concerned with ensuring that software will execute in a system context without resulting in unacceptable risk. One class of techniques for software safety is concerned with design principles that will reduce the likelihood of hazardous states; another is concerned with methods for analyzing software in order to identify any unduly hazardous states. An example of the latter is "Software Fault Tree Analysis" (SFTA). The description below is adapted from that in [120].

SFTA 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 classifications range from "catastrophic," meaning that the hazard poses extremely serious consequences, down to "negligible" which denotes that the hazard will not seriously affect system performance. Once the hazards have been determined, fault tree analysis proceeds. It should be noted here that in a complex system, it is possible, and perhaps even likely, that not all hazards can be predetermined. This fact does not decrease the necessity of identifying as many hazards as possible, but does imply that additional procedures may be necessary to ensure system safety.

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.

The root of the fault tree is the event to be analyzed, i.e., the "loss event." Necessary preconditions are described at the next level of the tree with either an AND or an OR relationship. Each subnode is expanded in a similar fashion until all leaves describe events of calculable probability or are incapable of further analysis for some reason. SFTA builds software fault trees using a subset of the symbols currently in use for hardware systems. Thus hardware and software fault trees can be linked together at their interfaces to allow the entire system to be analyzed. This is extremely important since software safety procedures cannot be developed in a vacuum but must be considered as part of overall system safety. For example, a particular software error may cause a mishap only if there is a simultaneous human and/or hardware failure. Alternatively, environmental failure may cause the software error to manifest itself. In many previous safety mishaps, e.g., the nuclear power plant failure at Three Mile Island, the safety failure was actually the result of a sequence of interrelated failures in different parts of the system.

Fault tree analysis can be used at various levels and stages of software development. At the lowest level the code may be analyzed, but it should be noted that higher levels of analysis are important and can and will be interspersed with the code level. Thus the analysis can proceed and be viewed at various levels of abstraction. It is also possible to build fault trees from a program design language (PDL) and to thus use the information derived from the trees early in the software life cycle. When working at the code level, the starting place for the analysis is the code responsible for the output. The analysis then proceeds backward deducing both how the program got to this part of the code and determining the current values of the variable (current state).

An experimental application of SFTA to the flight and telemetry control system of a spacecraft is described by Leveson and Harvey [120]. They report that the analysis of a program consisting of over 1250 lines of Intel 8080 assembly code took two days and discovered a failure scenario that could have resulted in the destruction of the spacecraft. Conventional testing performed by an independent group prior to SFTA had failed to discover the problem revealed by SFTA.

5.3 Testing Requirements and Specifications

So far we have explicitly considered only the testing of finished *programs*, but there is much to be said for the testing of specifications and requirements also. In the first place, testing a program against its specification is of little value if the specification is wrong; secondly, the cost of repairing faults increases dramatically as the number of life-cycle stages between its commission and its detection increase. As we noted in Section 2 (page 6), it is relatively simple, quick, and cheap, to correct an error in a requirements statement if that error is discovered during review of that statement, and before any further stages have begun; and it is also fairly simple, quick, and cheap to correct a coding error during testing. It is, however, unlikely to be either simple, quick, or cheap to correct an error in requirements that is only discovered during system test. Major redesign may be required, and wholesale changes necessitated. Any attempt to correct the problem by a "quick fix" is likely to generate even more problems in the long run.

For these reasons, testing and evaluation of requirements, specifications, and design documents may be considered a very wise investment. Of the testing methods we have described, only structured walk-throughs are likely to be feasible if the requirements and specification documents are informal, natural-language texts. If requirements and specifications are presented in some semi-formal design language, then limited anomaly detection and mathematical verification may be feasible, and possibly simulated execution also. If fully formal requirements and/or specifications are available, then quite strong forms of anomaly detection, mathematical verification, and even dynamic testing may be feasible.

In the following sections, we will briefly touch on some of these topics.

5.3.1 Requirements Engineering and Evaluation

Many studies of the software life-cycle have concluded that its early stages are particularly crucial. It is in these early stages that the overall requirements for a system are identified and the basic design of the system is specified. Errors or misapprehensions made at these stages can prove ruinously expensive to correct later on. Recent studies (e.g., [14, 28]) have shown that errors due to faulty requirements are between 10 and 100 times more expensive to fix at the implementation stage than at the requirements stage. There are two main reasons for the high cost of modifying early decisions late in the life-cycle:

- The changes often have a widespread impact on the system, requiring many lines of code to be modified. Furthermore, it can be difficult to identify all of the code requiring attention, resulting in the modification being performed incorrectly.
- The documentation is likely to be inadequate, exacerbating the problem of determining why certain decisions were made and assessing the impact of decisions. It is also difficult to keep the documentation current as changes are made.

Not only are errors in requirements and specifications expensive to correct, they are also among the most frequent of all errors – one study [14] found that 30% of all errors could be attributed to faulty statement or understanding of requirements and specifications. Worse, it appears that errors made in these early stages are among those most likely to lead to *catastrophic failures* [121].

These problems indicate the need for methodologies, languages, and tools that address the earliest stages in the software life-cycle. The aims of such *requirements engineering* are to see that the *right* system is built and that it is built *correctly.* Since systems have many dimensions, there are several facets to the question of what constitutes the right system. Roman [160] divides these facets of system requirements into two main categories: *functional* and *non-functional* (these latter are also called *constraints*). Functional requirements capture the nature of the interaction between the system and its environment—they specify what the system is to *do.* Non-functional requirements restrict the types of system solutions that should be considered. Examples of non-functional requirements include security, performance, operating constraints, and cost.

Functional requirements can be expressed in two very different ways. The *declarative* (or *non-constructive*) approach seeks to describe *what* the system must do without any indication of *how* it is to do it. This style of requirement specification imposes little structure on the system and leaves maximum freedom to the system designer—but, since it says nothing about how the system is to work, it provides little basis for checking non-functional constraints. The *procedural* approach to the specification of functional requirements, on the other hand, aims to describe what the system must do in terms of an outline design for accomplishing it. This approach appeals to many engineers who find it most natural to think of a requirement in terms of a mechanism for accomplishing it.

Both functional and non-functional requirements definitions, and declarative and procedural specifications, should satisfy certain criteria. Boehm [28] identifies four such criteria: namely, completeness, consistency, feasibility, and testability.

Completeness : A specification is complete to the extent that all of its parts are present and each part is fully developed. Specifically, this means: no TBDs ("To Be Done"), no nonexistent references, no missing specification items, and no missing functions.

Consistency : A specification is consistent to the extent that its provisions do not conflict with each other or with governing specifications and objectives. An important component of this is *traceability*—items in the specification should have clear antecedents in earlier specifications or statements of system objectives.

Feasibility : A specification is feasible to the extent that the life-cycle benefits of the system specified exceed its life-cycle costs. Thus feasibility involves more than verifying that a system satisfies functional and performance requirements. It also implies validating that the specified system will be sufficiently maintainable, reliable, and human-engineered to keep a positive life-cycle balance sheet.

Testability : A specification is testable to the extent that one can identify an economically feasible technique for determining whether or not the developed software will satisfy the specification.

Among the methodologies that aim to satisfy these criteria TRW's SREM [5, 6, 17, 52] is representative, and is described in the following section.

5.3.1.1 SREM

SREM (Software Requirements Engineering Methodology) was the product of a program undertaken by TRW Defense and Space Systems Group as part of a larger program sponsored by the Ballistic Missile Defense Advanced Technology Center (BMDATC) in order to improve the techniques for developing correct, reliable BMD software. Early descriptions of SREM include [5, 17]; descriptions of subsequent extensions for distributed systems can be found in [6], and accounts of experience using SREM are given in [28, 38, 167].

Owing to its genesis in the problems of software for ballistic missile defense, SREM adopts a system paradigm derived from real-time control systems. Such systems are considered as "stimulus-response" networks: an "input message" is placed on an "input interface" and the results of processing—the "output message" and the contents of memory—are extracted from an "output interface" [5]. Furthermore, the requirements for the system are understood in terms of the processing steps necessary to undertake the required task. The essence of a SREM requirements definition is therefore a dataflow-like description (called an *R-net*) of the processing steps to be performed and the flow of data (messages) between them.

SREM recognizes that requirements engineering is concerned with more than just writing a description of what is required—it is first necessary to analyze the problem in order to discover just what *is* required, and it is constantly necessary to check one's understanding of the problem and its evolving description against external reality. Accordingly, SREM allows behavioral simulation studies in order to verify that the system's interfaces and processing relationships behave as required. In addition, there is provision for *traceability* of all decisions back to source.

Again due to its origins in ballistic missile defense, SREM is very much concerned with the constraints of accuracy and performance. It therefore makes provision for traceable, testable, performance and accuracy constraints to be attached to requirements specifications.

In addition to support for analyzing the problem and checking understanding, SREM's tools perform "internal" completeness and consistency checks (i.e., checks that are performed relative to the requirements definition itself, without reference to external reality). These checks ensure, for example, that all data has a source, and that there are no dangling items still "to be done."

The paradigm underlying SREM is that system requirements describe the necessary processing in terms of all possible responses (and the conditions for each type of response) to each input message across each interface. This paradigm is based on a graph model of computation: requirements are specified as *Requirement Networks*, or *R-Nets*, of processing steps. Each R-Net is a tree of *paths* processing a given type of stimulus.

R-nets are expressed in the Requirements Statement Language (RSL), the language of SREM: an RSL requirements definition is a linear representation of a two-dimensional R-net. Requirements definitions in RSL are composed of four types of primitives:

- *Elements* in RSL include the types of data necessary in the system (DATA), the objects manipulated by the system being described (MESSAGES), the functional processing steps (ALPHAs), and the processing flow descriptions themselves.

- *Relationships* are mathematical relations between elements. For example, the relationship of DATA being INPUT to an ALPHA. Generally, a complementary relationship is defined for each basic relationship: for example, an ALPHA INPUTS DATA.

- *Attributes* are properties of objects, such as the ACCURACY and INITIAL_VALUE attributes of elements of type DATA. A set of values (names, numbers, or text strings) may be associated with each attribute. For example, the set of values associated with INITIAL_VALUE is the set of initial values allowed for data items.

- *Structures* model the flows through the processing steps (ALPHAs) or the flows between places where accuracy or timing requirements are stated (VALIDATION_POINTs).

RSL is described as an extensible language. What this means is that the user can declare new elements, relationships, and attributes; *however, they do not have meaning.* Only dataflow concepts have meaning. Thus, in the conventional sense, RSL is not an extensible language.

Using RSL, a user (called the *requirements engineer*) is encouraged to identify significant units of processing, each of which is viewed as a single input/single output ALPHA (inputs and outputs can have structure—rather like PASCAL records—so the restriction to single inputs and outputs is not as severe as it might appear). The definition of an ALPHA consists of declarations of its input and output, declarations of any *files* the ALPHA will write to and a brief natural language description of the transformations it effects on the input data.

ALPHAs are connected together with OR, AND, SELECT and FOR_EACH nodes into what can be viewed as a dataflow graph.[2] The intention is to express system requirements in terms of how significant units of processing are connected with each other, and the kind of data that flows along the interconnections.

OR nodes resemble a PASCAL *case* statement and indicate conditions under which each output path will be followed. The conditions attached to a path through an R-Net identify the conditions under which an ALPHA is invoked and in which an output message must be produced in response to an input message.

The AND node indicates that all of the paths following it are to be executed. Execution of the paths can be in any order or with any degree of parallelism permitted by the hardware base. It is intended that any file read in any of the parallel paths is not written by any other path, otherwise the behavior of the system would be indeterminate. In RSL, this constraint on files is called the *independence* property.

[2]Strictly, it is incorrect to refer to RSL descriptions as dataflow programs since they can produce side-effects to files.

There is no *goto* statement in RSL. To produce other than loopless programs, RSL provides the FOR_EACH node. This takes a set of data items as argument and indicates that the path following it is to be executed once for each of the items in the set; the order is not specified.

Multiple R-Nets can be linked together using EVENT nodes, and VALIDATION_POINTs can be attached to paths in an R-Net in order to specify performance and accuracy requirements.

Several tools have been constructed to support requirements descriptions written in RSL. Collectively, these constitute the "Requirements Engineering Validation System" (REVS). The most basic component of REVS is the RSL translator, which analyzes RSL requirements definitions and generates entries in a central database called the Abstract System Semantic Model (ASSM). Information in the ASSM may be queried, and checked for consistency using the "Requirements Analysis and Data Extraction" system (RADX). RADX generates reports that trace source document requirements to RSL definitions, identify data items with no source or sink, unspecified attributes, useless entities and so on. Other checks ensure, among other things, that the paths following an AND satisfy the independence property.

In addition to the *analysis* and *extraction* tools described above, *generation and display* tools provide two-dimensional graphical displays for R-Nets, and functional and analytical *simulators* support validation of performance, accuracy, and functional requirements. Both types of simulator automatically generate a PASCAL program corresponding to the R-Net structure. Each ALPHA becomes a call to a PASCAL procedure—which is generally written by the requirements engineer and associated with the corresponding ALPHA as an attribute.

The REVS simulation tools attempt to generate input data from the description of the data provided by the user. The user can also declare *artificial* data, i.e., data not required to be generated by the system when deployed. Typically, artificial data will be more abstract than the data actually applied to the system in operation. In addition to functionality, REVS supports simulation for the validation of performance and accuracy constraints. The latter are evaluated using "rapid prototypes" of the critical algorithms to be used in practice.

5.3.2 Completeness and Consistency of Specifications

Among the properties of specifications that are generally considered desirable, completeness and consistency rank highly. Informally, completeness means that the specification gives enough information to totally determine the properties of the object being specified; consistency means that it does not specify two contradictory properties.

For a semi-formal specification language, it may be possible to give some precepts for the construction of complete and consistent specifications, and it may be feasible to check adherence to these precepts mechanically. With formal specification languages, however, rather more may be possible. For quantifier-free equational logic—which logic has been found very suitable for the specification of abstract data types [135]—there is a formal notion of "sufficient completeness" that can be checked mechanically [82], and a sufficient test for consistency is that the Knuth-Bendix algorithm [114] should terminate without adding the rule `true -> false` [103]. Kapur and Srivas [104] discuss other important properties of such specifications and describe appropriate tests. Meyer [129] provides some interesting examples of flawed specifications that have appeared in the literature, while Wing [195] describes 12 different specifications for a single problem and discusses some of the incompletenesses and ambiguities found therein.

5.3.3 Mathematical Verification of Specifications

As we explained in Section 5.2.3 (page 41), mathematical verification demonstrates consistency between two different descriptions of a program. Often, one of these descriptions is the program itself—so that a program is verified against its specification. However, it is perfectly feasible that two specifications, at different levels of detail, should be the focus of mathematical verification. Indeed, a whole hierarchy of specifications may be verified in stepwise fashion from a highly abstract, but intuitively understandable one, down to a very detailed one that can be used as the basis for coding. If the original, abstract specification, is studied and understood by the user, and agreed by him to represent his requirements, then such hierarchical verification (provided it is performed without error) accomplishes the validation of the detailed specification. This approach is attractive in circumstances where the properties of interest are difficult to validate directly—as in the case of ultra-high reliability (where failure probabilities on the order of 10^{-9} per day may be required, but are unmeasurable in practice) [162], and security (which requires that *no possible* attack should be able to defeat the protection mechanisms) [176].

5.3.4 Executable Specifications

Specification languages provide mechanisms for saying *what* must be accomplished, not *how* to accomplish it. As a result, specifications cannot usually be executed. Programming languages on the other hand, reverse these concerns and provide many mechanisms for stipulating *how* something is to be accomplished. As a result, programs generally execute very efficiently, but are inperspicuous. Recently, however, *logic* programming languages have emerged that blur the distinction between specification and programming languages. By employing a more powerful interpreter (essentially a theorem prover, though generally referred to as an "inference engine"), logic programming languages allow the programmer to concentrate more on the *what*, and less on the *how* of his program. Dually, these languages can be regarded as *executable* specification languages. The obvious merit of executable specification languages is that they permit specifications to be tested and validated directly in execution.

Prolog, the best known logic programming language [46], contains many compromises intended to increase its efficiency in execution that detract from its merit as a specification language.[3] Languages based on equations, however, offer considerable promise. OBJ [69], developed by Goguen and his coworkers in the Computer Science Laboratory of SRI, is the best developed and most widely known of these. In addition to a cleaner logical foundation than Prolog, OBJ has sophisticated typing and parameterization features that contribute to the clarity and power of its specifications. (Being based on equational logic, OBJ could also exploit the completeness and consistency checks described in Section 5.3.2 (page 54), although the present version of the system does not do so.)

5.3.5 Testing of Specifications

Conventional formal specification languages are optimized for ease and clarity of expression and are not directly executable. Furthermore, high-level specifications are often deliberately partial—they indicate what is required of any implementation, but do not provide enough information to uniquely characterize an acceptable implementation. Nonetheless, it is highly desir-

[3]Stickel's Prolog Technology Theorem Prover (PTTP) [179], which provides correct first-order semantics—but with Prolog-like efficiency when restricted to Horn clauses—overcomes some of these disadvantages.

able to subject such specifications to tests and scrutiny in order to determine whether they accurately capture their intended meaning.

If direct execution is infeasible for the specification technique chosen, indirect testing methods must be used. As noted above, a formal specification defines properties that are required to be true of any implementation. In addition to the properties S that have been specified in this way, there may be additional properties A that are desired but not mandated, or that are believed to be subsumed by S, or that are to be added in a later, more detailed, specification. Tests of formal specifications consist of attempts to check whether these intended relationships between the given S and various sets of properties A do, in fact, hold. Thus, to ensure whether the property A is subsumed by S, we may try to establish the putative theorem $S \supset A$. Independently of additional properties A, we may wish to ensure that the specification S is consistent (i.e., has a model)—since otherwise $S \supset A$ is a theorem for all A.

Depending on the formal specification language and verification environment available, examinations such as those described above may be conducted by attempting to prove putative theorems, by symbolic evaluation, or by rapid prototyping. Kemmerer [109] describes the latter two alternatives. An important special case is the checking of specifications for consistency with a notion of "multilevel security." This activity, which is a requirement for certain types of system [59], seeks to demonstrate that a fairly concrete specification of a system is consistent with an abstract specification of security [43].

5.3.6 Rapid Prototyping

As we have noted several times, errors made early but detected late in the life-cycle are particularly costly and serious. This applies especially to missed or inappropriate requirements—yet such faults of omission are especially difficult to detect at an early stage. Systematic review will often detect inconsistent, or ambiguous requirements, but missing requirements generate no internal inconsistencies and often escape detection until the system is actually built and tried in practice.

A rapid prototype is one that simulates the important interfaces and performs the main functions of the intended system, while not necessarily being bound by the same performance constraints. Prototypes typically perform only the mainline tasks of the application, but make no attempt to handle the exceptional cases, or respond gracefully to errors. The purpose of a rapid prototype is to allow early experience with, and direct testing of, the main aspects of the system's proposed functionality—thereby allowing much earlier and more realistic appraisals of the system's requirements specifications.

An experimental comparison of a prototyping versus the conventional approach to software development [30] found that both approaches yielded approximately equivalent products, though the prototyping approach required much less effort (45% less) and generated less code (chiefly due to the elimination of marginal requirements). The products developed incrementally were easier to learn and use, but the conventionally developed products had more coherent designs and were easier to integrate. (Another experimental evaluation of prototyping is described by Alavi [4].)

Viewed as testing vehicles for evaluating and refining requirements specifications, rapid prototypes fit neatly into the standard life-cycle model of software engineering. A more radical approach that has much in common with rapid prototyping is incremental software development. Here, the complete software system is made to run quite early in the development phase, even if it does nothing useful except call dummy subprograms. Then it is fleshed out, with the subprograms being developed in their turn to call dummy routines at the next level down, and so on until the system is complete. The advantages claimed for this approach [33] are that it necessitates top-down design, allows easy backtracking to reconsider inappropriate decisions, lends itself to rapid prototyping, and has a beneficial impact on morale. See also Boehm's "Spiral" Model of system development [29].

5.4 Discussion of Testing

Much attention has been focused on systematic testing strategies—especially structurally based ones. However, there is evidence that, if increasing reliability (rather than finding the maximum number of bugs) is the goal, then random testing is much superior to other methods. Currit, Dyer and Mills [51] report data from major IBM systems which shows that random testing would be 30 times more effective than structural testing in improving the reliability of these systems. The reason for this is the enormous variation in the rate at which different bugs lead to failure: one third of all the bugs had a MTTF of over 5000 years (and thus have no effect on overall MTTF), and a mere 2% of the bugs accounted for 1000 times more failures than the 60% of bugs that were encountered least often.[4]

Interpretation of data such as these requires a context that establishes the *purpose* of testing—is it to find bugs or to improve reliability? Gelperin and Hetzel [72] discuss this issue and identify a historical evolution of concerns, starting with debugging in the 1950s. They propose that the concern for 1988 and beyond should be the use of testing as a fault prevention mechanism, based on the detection of faults at the earliest possible stage in the life-cycle.

Of the systematic testing strategies, Howden's data [94] ([92] is similar, but based on a smaller sample of programs) provides evidence that functional testing finds about twice as many bugs as structural testing; furthermore, several of the bugs found by structural testing would be found more easily or more reliably by other methods, so that the ratio three to one probably more accurately reflects the superiority of functional over structural testing.

Howden's data also shows that static testing (particularly automated anomaly detection) found more bugs than dynamic testing—however, the two methods were complementary, dynamic testing tending to find the bugs that static testing missed, and vice versa. Myers [136] presented similar data, showing that code walk-throughs were about as effective as dynamic testing at locating errors in PL/1 programs.

The efficacy of anomaly detection is likely to increase with the degree of redundancy present in the programming language—indeed, the presence of redundancy may often allow errors to be detected at compile-time that would otherwise only be located at run-time. (Compare Lisp with Ada, for example: almost any string with balanced parentheses is a syntactically valid Lisp program and any deficiencies will only be discovered in dynamic testing, whereas writing a syntactically correct Ada program is quite demanding and many errors will be caught at the compilation stage.) There have been proposals to increase the degree of redundancy in programs in order to improve early error-detection. One of the most common suggestions is to allow physical units to be attached to variables and constants [105, 12]. Expressions may then be subject to dimensional analysis in order to prevent a variable representing length being added to one representing time, or assigned to one representing velocity.[5] Other proposals would render it

[4]The thirty-fold improvement of random over structural testing is simply estimated by the calculation $2 \times 1000/60$.

[5]Strong typing, present in any modern programming language, provides some protection of this sort—preventing booleans being added to integers, for example—but the use

impossible to read from an empty message buffer, or to use a variable that has not yet been given a value [180].

All forms of dynamic testing assume that it is possible to detect errors when they occur. This may not always be the case—as, for example, when the "correct" value of a result may be unknown. Weyuker [190] first identified the problem of "untestable" programs, and proposed two methods for alleviating the difficulty they pose. The first is to use simplified test cases for which it is possible to compute the "correct" answers. The second, and more interesting method suggested by Weyuker is "dual coding": writing a second—probably simpler but less efficient—version of the program to serve as a check on the "real" one. The efficacy of this technique depends on the extent to which errors are correlated between the two versions. Recently, experiments have been performed to examine this hypothesis in the context of N-Version programming[6] [112, 113, 62, 122]; the results indicate that errors do tend to be correlated to some extent. Additional problems can arise with numerical software, due to the character of finite-precision arithmetic [32].

The efficiency of the debugging process can be evaluated by seeding a program with known errors—this is called mutation testing. If ten errors are seeded, and debugging reveals 40 bugs, including 8 of those that were seeded, we may conclude that 10 ($= \frac{40}{8}(10-8)$) bugs (including two seeded ones) remain. The weakness in this approach is that it is highly questionable whether the seeded bugs reflect the characteristics of the natural bug population: being introduced by simple modification to the program code, they are unlikely to reflect the behavior of subtle errors committed earlier in the life-cycle. A counter-argument is that test sets that detect simple errors will often catch subtle errors too (this is called "coupling" [130]) and mutation testing provides a systematic technique for establishing the coverage of a test set [35, 95].

For some very critical applications, formal verification is recommended as the assurance mechanism of choice [59]. Some authors have cast doubt

of physical units and dimensional analysis represents a capability beyond the normal typing rules. The data abstraction facilities of a modern language such as C++ or Ada can provide this capability, however [47, 90].

[6]N-version programming is an adaptation for software of the modular redundancy techniques used to provide reliable and fault tolerant hardware. N (typically $N = 3$) separate modules perform the computation and their results are submitted to a majority voter. Since all software faults are *design* faults, the redundant software modules must be separately designed and programmed. N-Version programming is advocated by Avizienis [44] (see also [65, 66] for a slightly different perspective), but has been criticized by Leveson and others (see the discussion in [9]).

on the value of this approach [55, 137], arguing that use of truly formal specification and verification tends to overload the user with intricate detail, and that the soundness of the whole process rests on the correctness of the underlying verification environment. Since this will typically be a very large and complex program in its own right, located at the limits of the state of the art, its own correctness should be regarded with more than usual skepticism. The thrust of this argument is that formal verification moves responsibility away from the "social process" that involves human scrutiny, towards a mechanical process with little human participation. We believe this concern unfounded and based on a mistaken view of how a mechanical verification environment is used. De Millo, Lipton and Perlis [55] claim that:

> "The scenario envisaged by the proponents of verification goes something like this: the programmer inserts his 300-line input/output package into the verifier. Several hours later, he returns. There is his 20,000-line verification and the message 'VERIFIED'."

This is a parody of the scenario actually envisaged by the proponents of verification. In a paper published several years earlier [188], von Henke and Luckham indicated the true nature of this scenario when they wrote:

> "The goal of practical usefulness does not imply that the verification of a program must be made independent of creative effort on the part of the programmer ... such a requirement is utterly unrealistic."

In reality, a verification system assists the human *user* to develop a convincing argument for his program by acting as an implacably skeptical colleague who demands that all assumptions be stated and all claims justified. The requirement to explicate and formalize what would otherwise be unexamined assumptions is especially valuable. Speaking from substantial experience, Shankar [171] observes:

> "The utility of proof-checkers is in clarifying proofs rather than in validating assertions. The commonly held view of proof-checkers is that they do more of the latter than the former. In fact, very little of the time spent with a proof-checker is actually spent proving theorems. Much of it goes into finding counterexamples, correcting mistakes, and refining arguments, definitions, or statements of theorems. A useful automatic proof-checker plays the role of a devil's advocate for this purpose."

Our own experience supports this view. We have recently undertaken the formal verification of an important and well-known algorithm for clock-synchronization and have discovered that the published journal proof [116] contains major flaws [164, 163]. It was the relentless skepticism of our formal verification environment that led us to this discovery, but our belief in the correctness of our current proof owes as much to the increased understanding of the problem that we obtained through arguing with the theorem prover as it does to the fact that the theorem prover now accepts our proofs.

Another element in De Millo, Lipton and Perlis' parody that is far from the truth is their implicit assumption that formal verification is something that is done *after* the program has been developed. In reality, formal verification is practised as a component of a development methodology: the verification and the program (or specification) are developed together, each generating new problems, solutions, and insights that contribute to the further development of the other. Formal verification can be made easier if the property being verified is achieved by direct and simple means. Thus, in addition to helping the user build a convincing case for belief in his program, formal verification encourages the programmer to build a more believable, and often better, program in the first place.

Overall, the conclusion to be drawn from experimental and other data seems to be that all testing methods have their merits, and these tend to be complementary to each other. For the purpose of enhancing reliability, random testing is a clear winner. For the purpose of finding bugs, anomaly detection and walk-throughs should be used in combination with functional and structural testing (Howden [96] describes such an integrated approach). Techniques for evaluating requirements and other specifications early in the life-cycle deserve special attention: rapid prototyping and some of the techniques of formal verification may be especially useful here. For really critical requirements, formal verification of those properties should be considered; conventional testing can be used to ensure that the less-critical requirements are satisfied.

Part II

Application of Software Quality Measures to AI Software

Chapter 6

Characteristics of AI Software

In this part of the report we will consider the application of the quality assurance techniques and metrics identified in Part I to AI software, and we will examine those few techniques that have been developed specifically for such software.

We face an initial difficulty in that the notion of AI software is fuzzy—indeed practioners of AI do not even agree among themselves on what constitutes AI. There are two notions of AI current today; Parnas [149] dubs them AI-1 and AI-2:

AI-1 is a *problem*-oriented notion that takes AI to be the use of computers to solve problems that could previously be solved only by applying human intelligence.

AI-2 is a *technique*-oriented notion that identifies AI with the use of characteristic programming strategies, in particular those based on heuristics, and explicit representation of "knowledge."

These notions are not mutually exclusive—indeed, most AI software has elements of *both* AI-1 and AI-2 and each facet contributes to the difficulty of SQA for AI software. The problems addressed by AI software (AI-1) are generally somewhat ill-defined, and a clear statement of requirements for the task the software is to perform is often lacking. This means that the notions of success and failure are vague, and evaluation is correspondingly difficult. In addition, the heuristic techniques employed in AI software (AI-2) tend to render it fragile, or unstable: very similar inputs may produce wildly different outputs. This makes extrapolation from behavior on test cases very risky.

For the purpose of this report, we will restrict our attention to those applications and techniques of AI that could be used in the development of "intelligent cockpit aids" for civil aviation. A NASA Contractor Report by BBN [13] considers the potential for such applications. Broadly, they conclude that the most useful aids would be "expert systems" for fault monitoring and diagnosis, and "planning assistants" to provide advice on such matters as fuel and thrust management. (A clear presentation of fault-diagnosis is given by Reiter [158], while Georgeff [74] provides a good introduction to planning.) Other important AI technologies for this application relate to the human factors of introducing such aids into the cockpit—natural language speech recognition and generation for example.

Given that they have been identified to be of special interest as intelligent cockpit aids, and given also their central place in current applications of AI in general, we will take expert systems and, to a lesser degree, planning systems, as our paradigms for AI software. Since it is reported that no intelligent cockpit aids presently exist [13], we will consider expert and planning systems fairly generically, but will pay special attention to the issues of fault monitoring and diagnosis where appropriate. The prototype fault monitoring system developed at NASA Langley Research Center [1, 2, 146, 159, 168, 169] provides a good example of the type of systems that may be among the earliest AI systems deployed aboard commercial aircraft.

The class of AI software that we have identified for consideration is often characterized as "knowledge-based"—meaning that it contains an *explicit* representation of knowledge about some aspects of the external world. Conventional software incorporates knowledge *implicitly* in the form of algorithms or procedures; the programmer knows how the program is to deal with payrolls or radar images and encodes this knowledge in the form of procedures (pre-planned sequences of actions)—whence the term "procedural knowledge" that is sometimes used for this type of knowledge representation.

Whereas knowledge is represented implicitly in conventional software, knowledge-based software contains an explicit declarative representation of knowledge, together with a reasoning component which can exploit that knowledge in order to solve problems. For example, a knowledge-based system to convert temperatures between Celsius and Fahrenheit might encode knowledge about the problem in the declaration

$$C = (F - 32) \times 5/9.$$

With the aid of a reasoning component capable of performing constraint satisfaction [118], this single declaration would enable the system to convert Fahrenheit to Celsius *and* vice-versa. A conventional system, on the other hand, would encode this knowledge in a procedural form somewhat as follows:

```
if (direction = f_to_c) then c := (f-32)*5/9
                         else f := c*9/5+32
endif
```

"Expert systems" are knowledge-based systems that perform functions in some specialized domain normally considered to require human expertise.[1] Expert systems come in two main flavors: those that employ "deep knowledge" and those that employ only "surface knowledge." Surface knowledge includes all the rules of thumb that human experts commonly employ. Such rules are highly specific to their particular domains (e.g., diagnosis of a particular group of diseases), and are often expressed in the form of "if–then" production rules. Systems based on surface knowledge expressed in this way form the "rule-based expert systems" that are becoming widespread.

Deep knowledge, on the other hand, is that which includes a model of a particular world—principles, axioms, laws—that can be used to make inferences and deductions beyond those possible with rules. In cases where the knowledge of even human experts is relatively superficial (e.g., medical diagnosis), there would appear to be little prospect of developing more than rule-based expert systems. In the case of fault monitoring and diagnosis for man-made physical systems (machines), however, deep knowledge is available. Someone designed the machine in the first place and, presumably, has a very good idea of how and why it works—there may also be very accurate mathematical models available to predict the behavior of the machine. The AI problem in such deep expert systems lies in knowing what information and which model is relevant to a particular circumstance. A recent collection of papers [24] provides a good overview of these topics.

The characteristics of knowledge-based systems render their evaluation somewhat different than conventional algorithmic software. In one of the few

[1]Expert systems that give advice to a human operator are called "decision aids," those that directly control some other system are called "autonomous." There is folk lore to the effect that decision aids are seldom used as such—users either rely on them completely (in which case they are effectively autonomous systems with a human actuator) or else ignore them altogether.

papers to address the problem of quality assurance for AI software (specifically, expert systems), Green and Keyes [80] summarize the difficulties as follows:[2]

> "Expert system software requirements are often nonexistent, imprecise, or rapidly changing. Expert systems are often procured in situations where the user does not fully understand his own needs. Some procurements omit requirements specifications as too constraining or not cost-effective. When expert systems are built by refinement and customer interaction, requirements may change rapidly or go unrecorded.
>
> "The success of verification demands that the requirements of the superior specification be at least recognizable in the subordinate specification: if this is not so then requirements tracing is futile. Expert systems are typically developed from a system specification or an informal specification by prototyping and refinement. Intermediate specifications are either not produced, not precise enough, or too subject to change to serve in verification.
>
> "Even if adequate specifications for requirements tracing were available, it is doubtful that conventional verification would yield many answers concerning whether the implemented system indeed satisfied the requirements.
>
> "Conventional validation demands precise test procedures. As long as reasonably precise requirements and design specifications can be obtained, test procedure preparation should be of no greater difficulty than for conventional software. When requirements and design information is unavailable, imprecise, or changing, test procedure design becomes a matter of guesswork.
>
> "There is no widely accepted, reliable method for evaluating the results of tests of expert systems. The approach of having human experts in the domain of the expert system evaluate the results has numerous drawbacks. There may be no expert available, or the expert may not be independent when independent evaluation is needed. Human experts may be prejudiced or parochial. The problem for which the expert system was written may be one that no human can solve reliably or efficiently."

[2]Culbert, Riley and Savely [50], Geissman and Schultz [71], Goodwin and Robertson [79], Lane [117], and Pau [150] provide additional discussions of these topics.

Green and Keyes go on to note that the present lack of understanding of how to perform verification and validation (V&V) for expert systems leads to a vicious circle: "nobody requires V&V of expert systems because nobody knows how to do it, nobody knows how to do V&V of expert systems because nobody has done it, and V&V is not done on expert systems because nobody requires it."

For all the reasons identified above, V&V for knowledge-based systems is difficult and little practised. Yet the very characteristics of knowledge-based systems tend to render them unreliable and untrustworthy, and so V&V—or at least some form of quality assurance—is surely *more* rather than less important for knowledge-based systems than it is for conventional software.

Among the characteristics of knowledge-based systems that render them unreliable and untrustworthy are the development methodology employed, their dependence on an explicit database of knowledge, their nondeterminism, and the unsoundness of their interpreters. We will consider each of these characteristics in turn.

- The established methods of SQA for conventional software require an orderly software engineering methodology with distinct phases, each phase producing a requirements or specification document that becomes the input to the next phase and serves as the measure against which its products can be evaluated. However, the ill-defined nature of many of the tasks for which AI software is proposed often precludes (or is claimed to preclude) the preparation of precise requirements or specification documents, and the system itself may be developed by a process of continuous modification and enhancement (tinkering), with no clearly defined statement of objectives to determine when it is finished.

- The dependency of AI systems on an explicit, declarative database of knowledge, means that "Knowledge Engineering"—the construction and representation of such databases, is a central problem in the development of such systems. Knowledge engineering is a laborious and highly skilled activity, yet it scarcely qualifies as engineering since it lacks a supporting body of science. At best, knowledge engineering is an art (some would say a black art). Denning [58] observes:

  > "Expert systems are dependent on the information in their knowledge-bases and are limited by the nature of the process for acquiring and recording that knowledge. Expert systems

> cannot report conclusions that are not already implicit in their knowledge-bases. And the trial-and-error process by which that knowledge is elicited, programmed, and tested, is likely to produce inconsistent and incomplete knowledge bases: hence an expert system may exhibit important gaps in knowledge at unexpected times. Moreover . . . their designers may be unable to predict reliably their behavior in situations not tested."

Cooke and McDonald [49] report that there is ample psychological evidence that humans are often unable to report accurately on their mental processes and that introspection is not very effective at eliciting the methods by which they really operate. In addition, they observe, the intended knowledge representation scheme guides the acquisition of knowledge resulting in a representation-driven knowledge base rather than one that is knowledge-driven.

- The nondeterminism in AI systems is a product of their heuristic, search-based method of interpretation. At any given point in the search, many paths may be available and the AI system's interpreter (inference engine) must choose one to pursue (it may later backtrack to try other paths if its first choice proves fruitless). Often, several of the available paths would succeed—possibly with different outcomes. The "conflict resolution" strategy employed by the interpreter may be sensitive to minor variations in the form of the input (or even to extraneous matter in its knowledge base), thereby causing very similar inputs to produce very different outputs. The treatment of measures of uncertainty, employed in some expert systems, may also produce unstable behavior that is highly, and undesirably, sensitive to small variations in the input.

- Compromises made for the sake of efficiency in the interpreters (inference engines) for rule-based systems may render them unsound. Rule-based systems have much in common with logic programming, discussed in Section 5.3.4 (page 56). The declarative style of programming supported by logic programming and rule-based systems is generally considered more perspicuous than the imperative style of languages like Ada, and is often described as having a simpler semantics, and sounder logical foundations than imperative programming. In the case of a "pure" logic as a programming language such as OBJ [69],

> these claims may be true, but in practice logic programming languages are often compromised, in the name of efficiency, to the point where their semantics are as obscure, and as operational, as any imperative programming language. Prolog programs, for example, cannot be regarded as pure logical expressions because of the presence of imperative and extra-logical features such as "cut" [56]. Furthermore, in the absence of the "occurs check," the resolution theorem proving strategy underlying Prolog interpreters is *unsound*—and this check always *is* omitted from Prolog interpreters for efficiency reasons [46].

These problems and difficulties may seem to render quality assurance for AI software a hopeless task. It is certainly clear that much work and thought needs to be devoted to these issues. In the following sections we will consider the extent to which the techniques described in the previous part of this report can be adapted and applied to AI software.

Chapter 7

Issues in Evaluating the Behavior of AI Software

All the techniques for software reliability estimation and for dynamic testing (and, for that matter, mathematical verification) that were described in Part I depend on the availability of requirements and specification documents—at least to the extent that it is possible to determine whether a program has experienced a failure.

The problem with requirements and specifications for AI software is that generally there aren't any—so failures in fielded AI systems may go unnoticed because those using them have no idea what the "correct" behavior should be. Almost any output may appear reasonable at the time it is produced and only be discovered as erroneous much later on (when, for example, an autopsy is performed, or an engine is stripped down). The problems of dynamic testing for AI software are similar: it may not be clear whether or not the outcome of a particular test is satisfactory.

Thus, before we can consider the application of software reliability and dynamic testing to AI software, we must consider the problems of obtaining requirements and specifications for such software, and of evaluating the system against these requirements and specifications.

7.1 Requirements and Specifications

The absence of precise requirements and specification documents for much AI software 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, if any of the established methods and techniques for SQA are to be applied to AI software, precise requirements and specifications are a necessity. In an attempt to break this impasse, we propose to distinguish the "inherently-AI" (i.e., AI-1) aspects of AI software from the more conventional aspects—aspects that should be amenable to conventional SQA.

7.1.1 Service and Competency Requirements

We believe it will be helpful to distinguish two sets of requirements and specifications for AI-software: 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 modules 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.

7.1.2 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 [7]. 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. Satisfaction of this requirement can easily be tested (e.g., by another computer program that examines the output of AALPS and computes the location of the center of gravity during all stages of unloading).

7.2 Evaluating Desired Competency Requirements

We have seen that some of the difficulties in evaluating the behavior of AI software can be minimized by identifying the notions of service requirements and of minimum competency requirements. Formal, or at least precise, requirements and specification statements should be feasible for these classes of requirements, and evaluation of the behavior of the system can be performed with respect to these statements. The desired competency requirement, however, may not admit of a precise requirement statement, and the only feasible evaluation may be against the performance of human experts. When we are dealing with a problem for which no deep knowledge exists—for example, medical diagnosis, there is almost certainly no alternative to evaluation against human experts (though see Section 7.2.3 on page 78); some experiences and recommended practices for this case are described below in Section 7.2.2. When deep knowledge does exist, however, it may be possible to evaluate competence against an automated adversary.

7.2.1 Model-Based Adversaries

Consider the problem of diagnosing faults in an electrical circuit. Given the symptoms of a fault, the problem of diagnosing its cause may be a hard one—despite the fact that accurate and complete models for the behavior of electrical circuits are available. The difficulty is not in the solvability of the problem—that can be accomplished by the naive algorithm of generating all possible combinations of faults and using a mathematical model to calculate the resulting symptoms until some are discovered that match the observed symptoms—but in solving it quickly. An AI program for efficient diagnosis of such problems will almost certainly employ rule-based techniques, or else a sophisticated qualitative reasoning approach. Now although a mathematical model of the circuit does not provide a good basis for an efficient diagnostic tool, it does provide an excellent adversary against which to evaluate such a tool: we simply inject faults into the model, observe the symptoms displayed, feed these into the AI diagnostic tool and compare its results with the faults that were injected in the first place.

This process can obviously be automated and, where it is feasible, it permits accurate, quantifiable, statements to be made concerning the competency of an AI system.

7.2.2 Competency Evaluation Against Human Experts

The goal of many AI systems is to reproduce the skill of human experts. In these cases, there is no alternative but to evaluate their performance against human experts. Gaschnig *et al.* [70] identify seven issues that should be considered in the design of such evaluations. Four of these issues are discussed in the following subsections; the other three—sensitivity analysis, eliminating variables, and interaction of knowledge—are discussed in Sections 8.1.2

(page 87), 7.3.1 (page 81), and 5.1.2 (page 32), respectively. An additional topic—statistical analysis of results—is covered in Section 8.1.3 (page 88).

7.2.2.1 Choice of Gold Standard

It is necessary to establish what will constitute the "gold standard" against which the performance of an AI system will be evaluated. There are two alternatives:

1. The gold standard is the objectively correct answer to the problem, or

2. It is the answer given by a human expert (or group of experts) when presented with the same information as the system being evaluated.

The first alternative is often infeasible: in diagnosis, for example, the true situation might be discovered only through autopsy or invasive surgery (in the case of medicine) or dismantling (in the case of mechanical systems)—and it may be unacceptable to perform these. Even where the use of an objective standard is feasible, it may be inappropriate if no deep knowledge is available for the domain or if human experts are unable to reliably identify the objectively correct answer.

When the second alternative is chosen, it is necessary to identify how the experts that constitute the gold standard are to be selected. If there is a well-established procedure for assessing human performance in the domain of interest, then it can serve not only as a means of selecting experts for use in evaluation, but can itself provide a credible and well-accepted basis for assessing the system. More commonly, human expertise is accepted and acknowledged on the basis of informal indications such as testimonials, years of training, seniority and salary, etc.

Whichever gold standard is chosen, and whatever criterion is used for selecting human experts, the really important point is that the standard of comparison should be agreed right at the beginning, before construction of the system is even begun, and should be adhered to throughout the system's construction and evaluation.

7.2.2.2 Biasing and Blinding

Human experts may exhibit bias if they are aware that they are evaluating the performance of a computer-based system. They may make judgments on the basis of their expectations and opinions regarding computers, rather than on the evidence that is presented. Accordingly, blind experiments that approximate the "Turing Test" [186] may be appropriate [41]. In an evaluation of MYCIN [198], the recommendations produced by the program were mixed with those from nine humans (ranging in experience from students to faculty members) and submitted to a panel of experts for evaluation. This method of evaluation not only avoids the possibility of bias, but provides another evaluation measure: the performance of the system can be compared to that of human practioners, as well as to experts. In the case of MYCIN, for example, although the experts judged its recommendations correct in only 65% of cases, none of the human subjects scored better than 62.5% [198].

7.2.2.3 Realistic Standards of Performance

It is important to agree at the outset what standard of performance should be considered acceptable. It some cases, it may be more appropriate to establish a standard based on the competence of those who currently perform the function, rather than on the performance of experts. For example, even if only 75% of a system's responses are judged correct by an expert, this may be acceptable if typical human practioners fall even further short of expert levels of performance. Another factor that can influence the choice of what is considered an adequate level of performance is the extent to which human experts disagree among themselves. In highly uncertain fields, where experts may agree with each other only 70% of the time, it is inappropriate to expect an AI system to score 90% agreement with any one expert.

7.2.2.4 Realistic Time Demands

The use of human experts in evaluations introduces factors that may be unfamiliar to computer scientists. Unclear instructions may reduce the quality of the evaluations produced by the experts; demands for excessive detail may lessen their degree of cooperation and increase the time taken. For example, it took a year for all the evaluation booklets to be returned by the experts who had agreed to participate in an early evaluation of MYCIN. Advice from those (such as psychologists and sociologists) experienced in the use of human subjects may be very useful in helping to design evaluations that generate prompt, accurate, and useful responses from human experts.

7.2.3 Evaluation against Linear Models

When the purpose of a knowledge-based system is to reproduce the skill of a human expert, it seems natural to evaluate its performance against that of the actual expert. Another possibility, however, is to evaluate it against some alternative model of the expert's behavior. It may seem surprising that there should be models of expert behavior that are not themselves expert systems; what is even more surprising is that, for some domains, these models not only exist, but are very simple and extremely effective. The domains concerned are judgment tasks, and the models are simple linear equations. The relevant work dates back to the investigations of psychologists in the 1950s and 60s and is little known to the AI community.[1] Psychologists investigating the behavior and accuracy of human experts (mainly medical diagnosticians), have explored the extent to which their performance can be explained as the integration of a relatively small number of "codable information sources" or cues. These cues may be objective (examination grades, or the results of laboratory tests, for example), or subjective—requiring interpretation, as in the case of reading X-ray films or estimating the severity of a disease. Influenced, no doubt, by the statistical basis of their field, psychologists compared the experts' judgments to linear regression models (i.e., linear combinations of cue values). These experiments tended to confirm the hypothesis that experts used relatively few cues and integrated them in simple ways. They also confirmed the hypothesis that experts may be neither particularly accurate nor consistent in their judgments. Perhaps more surprisingly, they also found that many of their experts were often out-performed by the linear regression models of their own behavior [54].

Linear models of expert judgment have been found to be extremely robust: it is often not necessary to use the best fit (i.e., regression) model—

[1]Dawes and Corrigan [54] trace some of the ideas back to Benjamin Franklin, and also cite pertinent observations of Thorndike dating back to 1918 [184], including the following.

> "There is a prevalent myth that the expert judge succeeds by some mystery of divination. Of course, this is nonsense. He succeeds because he makes smaller errors in the facts or in the way he weights them. Sufficient insight and investigation should enable us to secure all the advantages of the impressionistic judgment (except its speed and convenience) without any of its defects.
>
> "The setting up of an equation of prophecy from an equation of status will usually be very complex, but a rough [linear] approximation, if sound in principle, will often give excellent results."

linear models with unit, or even random weights can work quite well. For example, Dawes and Corrigan report an experiment that involved prediction of students' grade point averages at the University of Oregon [54, Table 1]. The average validity of expert judges' predictions was 0.37, that of their regression models was 0.43, that of random linear models was 0.51, that of an equal-weighting model was 0.60, while that of an optimal linear model (i.e., one fitted to the actual outcomes, rather than the judges' predictions) was 0.69.

Dawes and Corrigan identify circumstances when linear models can be expected to perform particularly well. First, there should be a "conditionally monotone" relationship between each cue and the judgment (outcome). That is, it should be possible for the values of cue variables to be scaled so that higher values of each predict a higher value of the judgment, independently of the values of the remaining cue variables. Secondly, there should be noise or uncertainty in the measurement of the judgment variable, and thirdly, there should be noise or uncertainty in the measurement of the cue variables. When these three conditions are satisfied, a linear model is often accurate and robust.

Psychologists have proposed using linear models in two ways. The first is as a "paramorphic representation" or model of the expert's behavior that can be used for analysis or prediction; the second, known as "bootstrapping," is as a replacement for the expert. The second of these applications of linear models sets them in direct competition with expert systems and is considered further in Chapter 10 (page 113). It is the use of linear models as paramorphic representations that is of interest in this section, since it suggests the use of such models in the evaluation of knowledge-based expert systems.

Knowledge-based systems that perform judgment or discrimination tasks for which it is possible to construct adequately performing linear models may usefully be evaluated against such models. Clearly, there is little benefit in employing a complex, knowledge-based system unless it can significantly out-perform a simple linear model. Thus a linear model may serve to establish a testable "floor" for the desired competency level of the knowledge-based system. In addition, since a linear model may be able to accurately reproduce an expert's behavior for at least some part of the input space, it could serve as a "screening" filter during dynamic testing of the knowledge-based system: those test cases that produce similar responses from both the knowledge-based system and the linear model could be accepted as correct (or subject to further investigation on only a random sampling basis), while those that produce different results are examined further, or submitted to the human expert for “gold standard” comparison. In this way, it may be feasible to subject a knowledge-based system to large numbers of tests without incurring the huge costs of human evaluation for the outcome of every test. Furthermore, since those test cases that produce different behavior between the knowledge based system and the linear model are likely to be the most revealing, the costs of human evaluation will be directed towards interesting test cases.

7.3 Acceptance of AI Systems

An AI system may perform adequately in formal tests against human experts, yet still fail to be accepted by its users. Gaschnig *et al.* [70] recount the early history of R1 (see also [128]), a system that configures the components of VAX computers. The acceptance plan called for R1 to configure 50 test orders, which would be examined by a panel of 12 human experts. Deficiencies noted would be corrected, and the process repeated using different sets of 50 orders at three week intervals until an acceptable level of accuracy and reliability had been demonstrated. In practice, the number of test cases used at each evaluation cycle was reduced from 50 to 10, and those 10 test cases were selected as simply the last 10 orders received. Five evaluation cycles were performed (for a grand total of 50 test cases), at which point R1 was judged sufficiently expert to be used routinely in the configuration task. One year later, it was found that although R1 was nominally in use, a human expert was reviewing the configurations produced by R1 and modifying 40% to 50% of them. It was not known whether the technicians who actually assembled the VAX computer systems were using the detailed layouts specified by R1; when questioned, they provided what is described as “extremely important feedback, albeit a bit overdue, as to what is important and what is not in carrying out the configuration task.”

Among the lessons to be learned from this experience are:

1. The test selection criterion was naive: simply the last 10 orders received. Many of these test cases were trivial and there was no attempt to look for difficult configuration tasks on which the system might fail. Consequently, McDermott admits [128]:

   > “In retrospect, it is clear that at the end of the validation stage R1 was still a very inexperienced configurer. It had

encountered only a tiny fraction of the set of possible orders, and consequently its knowledge was still very incomplete."

2. There was no clearly established "gold standard." It was found that the human evaluators disagreed among themselves as to the right way to do the configurations.

3. Testing for the purposes of development was confused with acceptance testing.

4. There was insufficient involvement by the eventual users in the testing and exercise of the system

The first of these points is considered more fully in Section 8.1 (page 85); the second was discussed above in Sections 7.2.2.1 and 7.2.2.2 (page 76). The third and fourth points are discussed below.

7.3.1 Identifying the Purpose and Audience of Tests

Although AI systems are typically developed in an iterative, incremental fashion, it is important to distinguish different phases in the process and not to confuse development with evaluation. Gaschnig *et al.* [70], for example, identify the following nine stages in the development of an Expert System:

Top-level design: definition of long-range goals.

Protoype: build Mark-I prototype to demonstrate feasibility.

Refinement: refine Mark-I prototype into Mark-II, by

1. Running informal test-cases to generate feedback from the expert, resulting in refined Mark-II prototype,
2. Releasing Mark-II prototype to friendly users and soliciting their feedback,
3. Revising system on the basis of that feedback, and iterating previous step.

Evaluation of performance: structured evaluation of the competence of the system.

Evaluation of acceptability: structured evaluation of acceptability of the system in its intended environment.

Prototype service: experimental use of the system in its intended environment for extended periods.

Follow-up: evaluate system's performance and impact in intended environment.

Revision: modify system to correct deficiencies discovered in previous step.

Release: market the system, with commitment to updates and maintenance.

The testing and experimentation that are undertaken as part of the refinement phase have a quite different character and purpose—and, most importantly, a different audience—than those undertaken during the later evaluation phases. Gaschnig *et al.* [70] point out that it is important to complete each phase of the development and evaluation before proceeding to the next. For example, the fourth phase is intended to establish that the system is performing at an expert level of competence, whereas the fifth phase is concerned with its acceptability to the user. If the fourth phase has not been completed satisfactorily before the fifth is started, then it will not be clear whether the user's failure to accept the system results from inadequacies in its competence or in its human factors. The purpose of the different phases of evaluation is to systematically *eliminate the variables* that can lead to the user's failure to accept the system. Thus the developers of MYCIN did not attempt to assess its clinical utility until they had established that its decision-making was at the level of an expert in the field. In this way, they could be sure that any subsequent failure of acceptance by physicians was due to human-engineering problems, rather than to decision-making errors.

7.3.2 Involving the User

Users may reject an AI system even though its competence has been established. Particularly in the case of decision-aids, it is important that the user should have some basis for evaluating the advice offered by the AI system. An "explanation system" is the usual method provided for doing this. Typically, the knowledge-engineer provides a natural-language explanation or justification for each rule in the rule-base, and the explanation system uses these to generate a trace of the rules that fired in the course of the system's deliberations. Recently, more sophisticated notions of explanation have been developed [182, 138] and it has also been proposed [197] that the man-machine interaction is best considered as a single system, and the machine component designed accordingly.

Bell [16] points out that a user's confidence in the advice offered by a system may be affected by knowing what information the system has used or not used in reaching its decision. For example, if an AI system responds "I recommend action X, and I used information A and B, but not C in reaching that conclusion," then the user may be cautious about accepting the recommendation if information C seems significant to him: the failure to use item C may indicate a gap in the system's knowledge base. A variant of this situation arises when information C is used, but does not affect the outcome. The explanation facilities of current AI systems do not provide this kind of insight into the system's decision making process; they serve more to justify the decisions actually made than to alert the user to possible weaknesses in the decision-making process.

7.3.2.1 Performance Evaluation of AI Software

The real question of importance for much AI software, especially decision aids, is "does it help the user accomplish his task." Note that it is not the responsibility of a decision aid to make decisions, and so it should not be evaluated as if it were an autonomous system—it is its ability to *assist* its user (e.g., by enabling him to carefully evaluate alternative courses of action) that is relevant. Thus, the *competence* of a decision aid will be one factor that needs to be evaluated, but it may not be the most important.

Systematic evaluation of the performance[2] of decision aids depends on a model of the decision maker's "value function." Multi-Attribute Utility Theory (MAUT), developed by Keeney and Raiffa [107], is employed by the "DAME" methodology [23] that has been applied to at least one military decision aid. Additional discussion of these topics can be found in a recently issued report [185] and in a paper by Liebowitz [123].

Although skill in the performance of its intended function will be a primary determinant of the utility of a knowledge-based system, it may not be the only one, and some benefits may be realized with even a relatively inexpert knowledge-based system. Moninger *et al.* [132], for example, suggest such benefits as a more consistent (even if occasionally less skilled) performance of the task than is provided by human experts, and improvement in the performance of the human experts themselves. This latter benefit

[2] "Utility" might be a better term, but we use "performance" in order to be consistent with the existing literature.

may be achieved because the process of knowledge engineering provides a structure in which to organize thinking about the problem and results in improved understanding, or because the knowledge-based system can serve a tutorial function during the training of human experts.

Chapter 8

Testing of AI Systems

In this chapter we consider the application to AI software of the testing techniques and strategies described in Chapter 5 (page 29). We assume the testing is being performed during later stages of system development for the purpose of evaluating competence.

8.1 Dynamic Testing

In Chapter 7 (page 72) we discussed the feasibility of dynamic testing for AI software and we argued that, with care, it should be possible to remedy the usual lack of explicit specifications for AI software and to subject it to systematic test and evaluation.

We saw in Section 5.4 (page 58) that, in the case of conventional software, the most effective dynamic testing strategies are random testing, functional testing, and path testing—in that (descending) order. However, all three methods complement each other, and each should form part of a comprehensive testing strategy. Functional testing requires detailed requirements and specification documents and these may not be available for the desired competency requirement of an AI system.[1] However, the whole point of our proposal that the expectations placed on AI systems should be partitioned into service requirements and into minimum and desired competency requirements is to ensure that it is *only* the desired competency requirement that presents this difficulty. Notwithstanding this difficulty, it is surely reasonable to expect that some attempt should be made to test the desired

[1]Random testing also requires some such documents in order to specify the expected operational profile.

competency requirement across its operational profile. O'Keefe [143], for example, suggests that a fair cross-sectional validation should include random test data selection, selecting some obscure or complex test cases that even experts find difficult, and having experts synthesize test cases in situations where data is lacking. Though the number of successful tests tends to increase confidence, "... the issue is not the *number* of test cases, it is the *coverage* of the test cases–that is, how well they reflect the input domain."

The notion of path testing requires some modification in the case of rule-based software. Such programs lack an explicit flow of control and so the notion of path has no meaning. However, path testing is merely a particular form of structural testing, and it is fairly easy to construct appropriate structural testing criteria for rule-based systems. A minimum requirement is surely that every rule should be exercised during testing (this corresponds to the all-nodes criterion for conventional programs), and a more stringent requirement, is that every outcome (i.e., firing or not) of every rule should be exercised (c.f. the all-edges criterion for conventional software). Beyond those simple criteria, we enter the realm of research: it would certainly be interesting and worthwhile to develop notions and measures for testing *combinations* of rules and outcomes in the spirit of path testing for conventional software.

It is an obvious point, and one widely accepted in the case of conventional software, that systematic testing strategies require very large numbers of test cases. This generally presents little difficulty in the case of conventional software, where test cases are usually straightforward to generate and evaluate and where failures are often manifest as outright crashes, but may do so in the case of AI software where one is looking for more subtle kinds of failure, and where evaluation of the output produced may be more difficult. Evaluation against human experts is unavoidably expensive[2] and, perhaps for this reason, AI software often seems to be subjected to very small numbers of formal test cases (e.g., the mere 10 test cases used in the evaluation of MYCIN, and 50 in the evaluation of R1). As a consequence, the purpose of such tests seems to be to demonstrate that the system is capable of expert performance, rather than to demonstrate that it is reasonably free of faults (that is, the aim is to show that the system can work well, not to find out how often it works badly). However, if we view the goal of systematic testing of AI software as a search for circumstances which

[2]Though recall the possibility, suggested in Section 7.2.3 (page 78), that linear models could mitigate this in certain situations.

cause the software to behave badly, then it may be possible to adequately evaluate the output of large numbers of systematic tests relatively cheaply: we will be looking for inplausible or suspicious results, rather than carefully examining the quality of fairly good results. The explanation facility may be helpful in determining the plausibility of results. The technique of Ince and Hekmatpour [98] (Section 5.1.5, page 38), which uses random test data to construct reasonably small test sets providing extensive coverage, may be worth exploring. Some special considerations that arise in the dynamic testing of AI systems are discussed below.

8.1.1 Influence of Conflict-Resolution Strategies

Testing combinations of rules (or even individual rules) will generally be difficult because the conflict-resolution strategy of the underlying interpreter (inference engine) can make it very difficult to predict the sequence of rules, or even the identity of the rules, that will be exercised. Furthermore, since the sequencing of rules can affect the outcome, it is highly desirable that different sequences should be tested. Thus, we propose that during testing, a second evaluation mode should be made available by the underlying inference engine—namely, a mode in which *all* successful paths are explored, not merely the first to be found. The inference engines of current expert system shells do not provide such a mode—however, some do allow a choice of conflict resolution strategy, and a weak form of the testing we propose may be accomplished by repeating each test under all the available strategies. Obviously, investigation should be undertaken if it is discovered that any test can generate more than one outcome—such different outcomes may indeed be what is intended and each may be equally acceptable, but this must be determined with care.

8.1.2 Sensitivity Analysis

The previous section recommended checking whether the *same* input can produce different outputs. It is equally important to determine whether very *similar* inputs can produce wildly different outputs—it is this potential for “instability” or “fragility” that underlies many of the concerns expressed about AI software. We recommend that the basic test scenarios suggested by random, functional, and structural testing should be systematically perturbed in order to determine whether the outputs produced are subject to major variation. If the system concerned permits the user to attach measures of certainty to input values, then these measures should be systematically perturbed as well as the values themselves. A related technique would vary the confidence factors embodied in the rules themselves, in order to ensure that the behavior of the system was not unduly influenced by the precise values chosen (unless there was a good reason for being precise about a value). MYCIN's developers performed this type of sensitivity analysis for confidence factors [70]; they found that exact values assigned to the confidence factors made little difference.

We note that sensitivity analysis may be a good way to get extra "mileage" from competency evaluations performed using human experts. Such evaluations are usually expensive to perform and only a few test cases may be available. By considering perturbations of these test cases, additional information may be obtained at little additional cost. Of course, this method of evaluation has a built-in predisposition in favor of continuity: it assumes that small variations in input should produce correspondingly small variations in output. This assumption may, of course, be dangerous in some situations where, for example, superficially similar symptoms should produce very different diagnoses. Even in these cases, however, one would expect that some additional corroborating evidence should be required.

8.1.3 Statistical Analysis and Measures

A special consequence of the problem domains addressed by AI systems is that there may be no single "correct" result. Consequently, when an AI system is evaluated against human experts the outcome may not be clear cut and it may be desirable to use statistical techniques to provide quantitative measures of relative performance. O'Keefe [143] describes some of the appropriate techniques: paired t-tests, Hotelling's T^2 test, and simultaneous confidence intervals.

O'Keefe proposes using paired t-tests to compare the difference between observed test results: the difference between a knowledge-based system and human performance may be represented as $D_i = X_i - Y_i$, where the X_i are system results, and Y_i are known results or results from human experts. Given n test cases there will be observed differences D_i to D_n from which O'Keefe defines the following confidence interval:

$$\bar{d} \pm t_{n-1,\alpha/2} S_d/\sqrt{n}$$

where $\bar{d}$ is the mean difference, S_d the standard deviation, and $t_{n-1,\alpha/2}$ is the value from the t distribution with n degrees of freedom. If zero lies in the confidence interval then the system's performance may be considered acceptable.

O'Keefe recommends that systems which produce multivariate, rather than single, results should be evaluated using Hotelling's one-sample T^2 test. If each input produces k measurable outputs, then we can construct a vector of the k differences between the system and human expert responses for each input. Repeating this for different inputs we obtain a set of vectors of differences and the one-sample T^2 test can then be used to determine if the means of the difference vectors are significantly different from zero.

It is often useful to derive a single statistic that represents "skill" in the performance of a task. Correlation coefficients measuring the correlation between the performance of a knowledge-based system and that of human experts (or the objective facts if these are known) are often used. Correlations measure discrimination but are insensitive to the scaling of responses; the mean square error statistic, which is also widely used, suffers from the opposite deficiency. The relative operating characteristic (ROC), which has its origin in signal detection theory [183], is an effective graphical approach to the presentation of skill. It plots the probability of a "hit" (1 minus the probability of a Type II error) against the probability of a false alarm (probability of a Type I error) as the decision criterion varies. The area under the ROC curve can be used as an indicator of skill [132].

8.1.4 Regression Testing and Automated Testing Support

AI systems may undergo considerable modification throughout the test and refinement phase of their development, and even through the evaluation phase. It is therefore important to perform regression tests (recall Section 5.1.2 on page 32) in order to ensure that modifications do not adversely change the system's behavior on examples encountered previously. Automated support for running and checking these test cases is clearly desirable; Scambos [166] describes such a tool.

8.2 Static Testing

The experimental data cited in Section 5.4 (page 58) showed that anomaly detection was among the most effective testing strategies for conventional software. In the following section we consider the application of such techniques to AI software. Later, we examine the possible application of mathematical verification and of structured walk-throughs to such software.

8.2.1 Anomaly Detection

Anomaly detection depends crucially on the presence of *redundancy* in programs. An anomaly in a program is nothing more than an apparent conflict between one indication of intent or purpose and another: for example, a programmer declares a variable as an integer, but assigns a string value to it. The effectiveness of anomaly detection is essentially determined by the degree of redundancy and structure in the programming language employed. Thus, when AI software is written in a conventional, imperative programming language (e.g., C, or Ada), the techniques of anomaly detection can be employed just as they can with conventional programs. Much AI software, however, is written in languages that contain very little redundancy. Many functional languages (such as Lisp) and logic programming languages (such as Prolog) have no typing mechanism, for example, and few or no facilities for control and data structuring. Rule-based systems are usually even worse in this respect—almost any collection of syntactically correct rules is a plausible knowledge base. One obvious recommendation for future work is the development of more structured languages for rule-based systems. Strongly (though often polymorphically) typed functional languages are now commonplace, and similar proposals have been made for logic programming languages.

In the present section, we will concentrate on the limited anomaly detection that can be performed on rule-based systems built on current expert system shells. Existing work in this area focuses on methods for testing rules-bases for consistency and completeness, and is closely related to techniques for debugging and maintaining knowledge-bases. We will concentrate on the testing angle here; use of similar techniques in "comprehension aids" and for knowledge-base maintenance is discussed in Section 8.2.4 (page 103).

TEIRESIAS [53] was the first attempt at automating the process of debugging knowledge bases; it was applied reasonably successfully to Shortliffe's MYCIN system [173]. TEIRESIAS examined the MYCIN rule-base and built models that revealed a number of patterns such as what attributes were used to deduce other attributes. When the rule-base was extended, TEIRESIAS would check that the new rules were consistent with its patterns and, if not, would interject comments such as: "Excuse me, doctor, but all rules mentioning x and y have also mentioned *patient age* as a relevant parameter; are you sure you don't want to add some clause about *patient age*?" Wilkins and Buchanan [193] discuss some of the problems that arise when debugging heuristic rule sets and show that the techniques used in TEIRESIAS do not always lead to the best set of rules.

Suwa, Scott and Shortliffe [181] describe another early program for checking the rule-base of an expert system. Their program, which was used in the development of the ONCOCIN expert system, first partitions rules into disjoint sets based on the attribute that is assigned a value in the conclusion. It then makes a table, displaying all possible combinations of attributes used in the antecedents and the corresponding values that will be concluded in the conclusion of the rule. The table is checked for conflicts, redundancy, subsumption, and missing rules (see below). The rule checker assumes there should be a rule for each possible combination of values of attributes that appear in the antecedent; the checker hypothesizes missing rules based on this assumption. The CHECK program described by Nguyen [140] is an adjunct to the "Lockheed Expert System" shell (LES) [115] that extends the rule-base checks used in the ONCOCIN project. The following paragraphs describe some of the checks performed by the CHECK program.

Situations that are considered likely to indicate problems of *consistency* in goal-driven rules are:

Redundancy: two rules have the same antecedent, and the conclusions of one subsume those of the other (e.g., $x \rightarrow y$ and $x \rightarrow y \wedge z$).

Conflict: two rules have the same antecedent, but their conclusions are contradictory (e.g., $x \rightarrow y$ and $x \rightarrow \neg y$).

Subsumption: two rules have similar conclusions, but the antecedent of one subsumes that of the other (e.g., $x \rightarrow y$ and $x \wedge z \rightarrow y$).

Unnecessary IF rules: two rules have the same conclusions, and their antecedents contain contradictory clauses, but are otherwise the same (e.g., $x \wedge z \rightarrow y$ and $x \wedge \neg z \rightarrow y$).

Circularity: a set of rules forms a cycle.

Situations that are likely to indicate problems of *completeness* in goal-driven rules are:

Unreferenced attribute values: some values in the set of possible values for an object's attribute are not covered in the set of rules.

Illegal attribute values: a rule refers to an attribute value that is not in the set of legal values.

Unreachable conclusions: the conclusion of a rule should either match a goal or an if condition in some other rule.

Dead-end goals and dead-end IF conditions: either the attributes of a goal must be askable (the system can request information form the user) or the goal must match the conclusion of a rule. Similar considerations apply to the IF conditions in each rule.

Similar considerations to those given above can be used for consistency and completeness testing of data-driven rules and numerical certainty factors can also be accommodated. Stachowitz *et al.* [178, 177] describe developments along these lines.

A problem with all these rule-checking systems is that they incorporate rather limited, local, interpretations of completeness and consistency: inconsistency, for example, is considered as a property of *pairs* of rules. That this is insufficient may seen by considering the following example:

$$\begin{array}{rcl} P \vee Q & \rightarrow & A \\ Q \vee R & \rightarrow & B \\ A \wedge B & \rightarrow & T \\ A & \rightarrow & D \\ B & \rightarrow & \neg D \end{array}$$

This rule-set is inconsistent: if Q is asserted, then both A and B will be asserted, and therefore both D and $\neg D$. But no pair of rules in this example exhibits any of the problems detected by the Nguyen's CHECK program [140], or any of the other similar tools mentioned above. This weakness has been identified by both Ginsberg [76] and by Bellman and Walter [19, 18]. They point out that redundancy and inconsistency are properties (possibly large) *sets* of rules.

Bellman and Walter describe techniques for detecting various kinds of incompleteness and inconsistency in a rule-base. Essentially, their technique for inconsistency would analyze the example above by first determining that D will be asserted true if $P \vee Q$ and false if $Q \vee R$. Since D cannot be both true and false, the conjunction $(P \vee Q) \wedge (Q \vee R)$ should be unsatisfiable. Since it is not, we conclude that D can be asserted to two different values and that the rule-base is therefore inconsistent. Bellman and Walter's description is somewhat hard to follow since they describe their method in terms of boolean algebra—whereas what they are really doing is testing formulas in propositional calculus (or, in more complicated cases, propositional calculus with Presburger arithmetic) for unsatisfiability.[3] Expressed in more familiar terminology, Bellman and Walter's procedure for testing consistency can be described as follows:

1. For each distinct attribute value, form the disjunction of the antecedents of the rules that assert that value (call this disjunction the "combined antecedent" for that attribute value).

2. For each incompatible pair of attribute values, show that the conjunction of their combined antecedents is unsatisfiable.

Similarly, Bellman and Walter's test for "current conditions" is equivalent to checking that the disjunction of the combined antecedents corresponding the set of all possible values for an attribute is tautological. For example, their paper [19, pages 17–20] considers the case of an object (variable 418) with three possible attribute values: `high`, `medium`, and `false`. The combined antecedents for these values are, respectively:

$$(A \wedge v_{124} \geq 35) \vee (A \wedge v_{124} < 35 \wedge v_{124} > 10) \vee (\neg A \wedge v_{124} > 20)$$

$$\neg A \wedge v_{124} > 6 \wedge v_{124} \leq 20$$

$$(A \wedge v_{124} \leq 10) \vee (\neg A \wedge v_{124} < 6).$$

The disjunction of these formulas is not tautological (it is falsified by $\neg A \wedge v_{124} = 6$), and so we conclude that variable 418 is not always determined by the current values of its "inputs."[4] Readers who work through the several pages of boolean algebra that Bellman and Walter [19, pages17–20] require to reach this conclusion, might be interested in the input to the EHDM verification environment [187] shown in Figure 8.1, which contains all that is needed to establish that the disjunction given above is not tautological. The `LEMMA` merely declares the formula of interest, while the `PROVE` command invokes the theorem prover—which then fails to prove the formula without further ado! Since the theorem prover of EHDM contains

[3]Similarly, their suggestion that an algebraic simplification system such as Macsyma could be used to automate some of the analysis could profitably be replaced by the observation that the appropriate tool is a decision procedure or simplifier for this combination of theories (e.g., Shostak's [174, 175] or that of Nelson and Oppen [139]).

[4]If the final inequality is changed to $v_{124} \leq 6$, then the disjunction does become a tautology—thereby suggesting a plausible correction.

```
v418test: MODULE

THEORY
   A: boolean
   v124: integer

   lemma1: LEMMA
      (A AND v124>=35) OR
      (A AND v124<35 AND v124>10) OR
      ((NOT A) AND v124>20) OR
      ((NOT A) AND v124>6 AND v124<=20) OR
      (A AND v124<=10) OR
      ((NOT A) AND v124<6)

PROOF

   prove_lemma1: PROVE lemma1

END v418test
```

Figure 8.1: EHDM Input for "Current Conditions" Check

a *complete* decision procedure for the combination of propositional calculus and Presburger arithmetic[5], this is sufficient to establish that the `LEMMA` is not tautological; the system does prove the `LEMMA` (in a couple of seconds) when its final inequality is changed to `v124<=6`.

Like Bellman and Walter, Ginsberg [77, 76] considers more subtle forms of incompleteness and inconsistency than those examined by Nguyen. He gives an algorithm, based on "knowledge-base reduction," for detecting these more complex forms of inconsistency and redundancy (though only for the propositional case) which seems more sophisticated than the techniques of Bellman and Walter, and he rightly points out that the analysis must consider the characteristics of the inference engine used by the expert system—it is not usually valid to interpret the rules as a collection of axioms in some standard logic, since the inference engine may not provide the standard semantics. Ginsberg, in fact, goes to some lengths to deal with the issues raised by the "closed world assumption" (CWA). The CWA is a simple form of nonmonotonic reasoning (see [73, Chapter 6] or [157] for elegant accounts of nonmonotonic reasoning; [158] explains its connection to fault diagnosis). The CWA asserts the negation of a ground term whenever the term itself cannot be deduced (e.g., if you cannot deduce that alligators are vegetarians, assume they are not). More elaborate forms of nonmonotonic reasoning include "default theories," "predicate completion" and "circumscription" [73, 157, 127]. Theorems are known which assert that if a theory is consistent in conventional logic, then under suitable conditions, its augmentation in certain forms of nonmonotonic logic will also be consistent. For example, if the clause form of a theory is Horn and consistent, then its CWA augmentation is consistent [73, Chapter 6]. Thus, some parts of Ginsberg's algorithm may be unnecessary—but only if the standard logical notion of inconsistency coincides with the interpretation appropriate to the deductive mechanism of the expert system concerned.

In general, the notions of inconsistency employed in logic and in rule-based expert systems do *not* coincide. In (propositional) logic, an inconsistent set of formulas is one that has no model; as long as there is *some* assignment of truth values to propositional symbols (i.e., an interpretation) that evaluates each formula to `true`, the theory is consistent. Thus the

[5]Strictly, the method is incomplete since it uses Gomory's method for integer feasibility. However, Gomory's method is incomplete only in that it is not guaranteed to terminate—and this proof does terminate.

following set of propositions is logically consistent

$$\begin{aligned} p &\rightarrow q \\ p \wedge r &\rightarrow \neg q \end{aligned}$$

since both formulas evaluate to `true` in any interpretation that assigns `false` to p. Interpreted as rules, however, these two formulas are surely inconsistent, since both q and $\neg q$ will be asserted if p and r are simultaneously `true`.

While this example (from Ginsberg [77]) clearly demonstrates that the notion of consistency employed in propositional logic is different from that used for rule-bases, none of the completeness and consistency tests described in this section give formal definitions for their notions of "consistency" and "completeness" (though Ginsberg [77] does identify his notion of "redundancy" with that of independence in logic). This is a serious omission; the weaknesses and uncertainties attending existing completeness and consistency tests for production rule systems may largely be attributed to the lack of a clear semantics for the notations employed and a corresponding lack of rigorous definitions for the properties being tested. The development of an appropriate semantic foundation in which to pose the problems of completeness and consistency for production rule systems with acceptable rigor is therefore a prerequisite for the creation of good algorithms for completeness and consistency checking. Such a foundation might also make it possible to develop techniques for establishing that a rule-based system terminates.

Despite their rather shaky foundations, completeness and consistency checking tools do seem to identify a significant number of problems in "typical" rule-bases. The proponents of such tools argue that this demonstrates that they are appropriate and effective; an alternative interpretation, however, is that it merely demonstrates that rule-bases are a dangerously unstructured and unreliable form of knowledge representation. If tools as crude as these detect problems, then how many deeper problems must remain undetected? Some experimental evidence for this latter position is provided by Pearce [152] who describes two knowledge-based systems for diagnosis of the electrical power system in a satellite. The first system was a conventional hand-crafted rule-based expert system. When this system was analyzed by the KIC rule-base checking program [151] (this is comparable to Nguyen's CHECK program), seven problems were found among its 110 rules. And when it was tested against a simulator for the satellite system concerned, it correctly diagnosed only 72% (10 out of 14) simulated failures. In contrast, the second system, which was produced mechanically from a qualitative model of the satellite electrical subsystem, contained only 75 rules, triggered no warnings from KIC, correctly diagnosed all 14 test cases, and required only half the development time needed for the hand-crafted rule-base. Furthermore, validation of the second system could also consider the qualitative model underlying its rule-base—a higher level and more perspicuous representation than the rule-base itself. In essence, the rule-base in the second of these systems may be regarded as compiled code; verification and validation can be directed to the qualitative model that constitutes its "source code." This seems a very promising approach and could well be the precursor of a new and more sophisticated and reliable approach to the construction of rule-based expert systems. In contrast, completeness and consistency checkers seem to address only the symptoms, rather than the causes of unreliability in rule-based systems.

8.2.2 Mathematical Verification

There has been little prior work on the application of mathematical verification to AI software.[6] The absence of formal specifications for most AI software certainly presents an immediate difficulty for mathematical verification and we concede that it may not be feasible to perform formal verification of *full functional correctness* for other than a small class of knowledge-based systems. We do not see this as a serious drawback, however; even for conventional software, it has recently become recognized that the expense and power of mathematical verification are such that it may best be applied to the verification of *critical properties* (such as security, fault-tolerance, or safety) rather than to the full function of the software. Instead of using formal verification to prove that things *work right*, we can use it to prove that things *cannot go badly wrong*. Using the terms introduced in Section 7.1.2 (page 73), we believe that it may be feasible to mathematically verify certain *minimum competency requirements*. For example, the AALPS system described in Section 7.1.2, has a minimum competency requirement that the aerodynamic trim of the aircraft must remain within certain limits during unloading in flight. We described how this requirement could be tested by another computer program that would examine the output of AALPS and compute the location of the center of gravity during all stages of unloading.

[6]Castore [37], who proposes using a modal logic for reasoning about knowledge-based control systems (but presents no details), is an exception.

Mathematical verification of this requirement, if feasible, would guarantee the safety (though not the optimality) of any loading plans produced by AALPS once and for all.

AALPS and other design and optimization systems tackle problems for which it is relatively easy to specify how to recognize an acceptable solution when you see it. The minimum competency requirement for such a system can therefore specify what constitutes an acceptable solution and verification will be concerned to prove that the system generates only acceptable solutions.

Other classes of knowledge-based systems tackle problems in which it may not be feasible to specify precisely what constitutes an acceptable solution. Classification problems, for example, often have this characteristic. Such a system might consider the known information about an aircraft and classify it as being either friendly or hostile. Everything that is known about how to perform this classification will be encoded in the rule-base and there will be no independent specification against which its ability to produce correct analyses can be verified. Even here, however, although it may not be feasible to specify what constitutes a *correct* identification or classification, it may be possible to specify certain partial specifications to exclude the possibility of things going badly wrong. In the threat-identification case, for example, a safety rule might be that if an aircraft's fire-control radar is locked on *you* then it is hostile, whatever other indications there may be.[7] Mathematical verification could then be used to ensure that this partial safety specification is satisfied.

As well as specifications concerning the external behavior of the system, it is possible to contemplate "safety" specifications concerning the allowable internal states of the rule and fact bases of a knowledge-based system. For example, we could specify that no radar track should ever be classified, even transiently, as belonging to both a ship and a plane. This specification establishes an invariant for the knowledge-base which could be formally verified.

The following example is intended to illustrate how formal reasoning may be of value in verification of knowledge-based systems. Winston [196, page 169] discusses a toy expert system for bagging groceries in a supermarket, which knows the following information about items within the store:

[7]It is claimed that during the war with Argentina over the Falkland Islands, British ships' radar did not classify incoming Exocet missiles as hostile because the radar signature of the Exocet identified it as belonging to an allied NATO country (France).

Item	*Container Type*	*Size*	*Frozen?*
Bread	Plastic Bag	Medium	No
Glop	Jar	Small	No
Granola	Cardboard box	Large	No
Ice cream	Cardboard carton	Medium	Yes
Pepsi	Bottle	Large	No
Potato chips	Plastic bag	Medium	No

Winston then defines a number of rules. We have selected a subset of these rules for the purposes of illustration. Our subset is:

```
B4    If    the step is bag-large-items
            there is a large item to be bagged
            there is an empty bag or a bag with large items
            the bag is not yet full
      then  put the large item in the bag


B8    If    the step is bag-medium-items
            there is a medium item to be bagged
            there is an empty bag or a bag with medium items
            the bag is not yet full
      then  put the medium item in the bag


B12   If    the step is bag-small-items
            there is a small item
            there is a bag that is not yet full
      then  put the small item in the bag
```

First, let us define “full,” which Winston uses without definition in the preceding rules, to be a predicate which is true if a bag contains 6 large items, 12 medium items or 24 small items (i.e., a large item is twice as big as a medium item, and a medium item is twice as big as a small item, and each bag may hold a maximum of the equivalent of 6 large items), and false otherwise.

Now, let us suppose for a moment that we are interested in safety in this context, which we define as not overloading the bags to the point where they might break. The particular bags in use may break if loaded with 30 lbs. or more. Let us further suppose that we do not know how much each individual item weighs, but we do know something about each category. We know that the heaviest large item is 4 lbs, and that the heaviest small and medium items are both 2 lbs. What we are interested in showing, of course, is that these rules will never produce a situation in which a bag is in danger of breaking.

Exhaustive testing, even of this three rule subset, is quite impossible. For each grocery order of r items, there are 6^r possible combinations. Therefore, to exhaustively test the three rules given above, it would be necessary to run:

$$\sum_{r=0}^{\infty} 6^r$$

test cases. Even with an operational constraint placed on the number of items one could possibly purchase (let's say 100), one would run:

$$\sum_{r=0}^{100} 6^r = 7.839823E + 77$$

test cases. In order to assure ourselves that this system is safe, we must appeal to formal methods.

We begin by satisfying ourselves that the system begins in a safe state, that is, we show that the *initial state* conforms to the criterion of our *state invariant* (our safety property), which is that all bags should contain strictly less than 30 lbs. It is easy to see that when all bags are empty, the weight in any given bag is less than 30 lbs. Therefore, our initial state is safe.

Next, we assume the system is in some arbitrary state which conforms to our safety property (where the weight in each bag is less than 30), and attempt to show that we cannot reach an unsafe state through the application of any of the rules. We assume the given rules exhaustively define all state transitions which may occur relative to the state-space of our problem domain. Therefore, if we can show that for each of the above three rules, application of the rule in an arbitrary safe state leads only to another safe state, then we can appeal to induction, and conclude that the above system is safe for all reachable states.

Rule B4 can only cause a large item to be put into a bag if the bag is empty or if the bag holds only large items and is not yet full. By the definition of full, this means fewer than 6 large items. Therefore, since the heaviest large item weighs 4 lbs, the most a bag could weigh prior to the transition of rule B4 is $5 * 4 = 20$ lbs. Further, since the heaviest object which rule B4 could cause to be added also weighs 4 lbs, we can conclude that the heaviest a sack could weigh after execution of rule B4 is $6 * 4 = 24$ lbs. Therefore, rule B4 is safe according to our definitions.

Rule B8 can only cause a medium item to be added to a bag which is empty or which holds exclusively medium items and is not yet full. Therefore, by the definition of full, there must be fewer than 12 items in a bag to meet the antecedent of rule B8. Since the heaviest medium item weighs 2 lbs, the heaviest a medium sack could weigh before execution of B8 is $11 * 2 = 22$ lbs. Again, since the heaviest item B8 could add is also 2 lbs, the heaviest a sack could weigh following the execution of B8 is $12 * 2 = 24$ lbs. Therefore, rule B8 is also safe according to our definitions.

Rule B12 may place small items in any bag which is not full. Therefore, the most a sack could weigh prior to execution of B12 (and still conform to our safety criterion) is $14 * 2 = 28$ lbs. Since the heaviest item which B12 can cause to be added weighs 2 lbs, it is possible for a sack to weigh 30 lbs after the execution of B12. Thus, B12 violates our safety criterion and our knowledge-based system is not safe. We must either get stronger sacks, or put additional constraints on the execution of rule B12 to make it conform to our definition of safety.

Any approach to formal rule-base verification, such as that suggested here, and also the systematic checks for rule-base inconsistencies and redundancies described in the previous section, will treat the rule-base as a formal object, and will therefore require that it is defined in some language having a tractable formal semantics. However, the multiple and weak notions of "inconsistency" in rule-bases that were described in Section 8.2.1 (page 90), and the lack of definitive algorithms for detecting such inconsistencies (or proofs that the problem is undecidable), both indicate the lack of clear semantic characterizations for the languages in which such rule-bases are expressed. In contrast, languages such as OBJ [69], which have a firm semantic characterization based on a standard logic (equational logic in the case of OBJ), can appeal directly to the notion of "consistency" as it is used in logic.

Although many programming environments for AI systems permit the description of rules that bear resemblance to sentences in a formal logic (for example, Horn clauses in the case of Prolog), the interpretation ascribed to these sentences by the "inference engine" generally does not coincide with the standard semantics of first-order logic (for example, in the treatment of negation), and extra-logical, quasi-imperative features (such as "cut") may be present [56]. There has been little work on developing formal semantics for production-rule systems, and little consideration given to developing languages and corresponding inference mechanisms that permit a clean semantic characterization. In fact, the programming notations used for AI systems (including those provided by expert system shells) seem to be at a stage of development comparable to that of conventional programming languages in the 1960s, where "power," "features" and "convenience" were considered more important than a formal semantic foundation. Thus developers of AI system architectures (commonly referred to as "shells") have paid little or no attention to mathematical rigor in the construction of the underlying reasoning methods upon which these systems are based.

Progress in systematic anomaly detection and in mathematical verification for AI systems will require that this attitude toward the programming notations used to describe AI systems is replaced by one that shows more concern for the desirability of constructs that have clean and tractable semantic characterization.

8.2.3 Structured Walk-Throughs

As described for conventional software in Section 5.2.2 (page 41), structured walk-throughs entail an informal, but highly detailed, manual examination of program code, specifications or requirements. Insofar as AI software is software, and is expressed in some programming notation or other, the technique of structured walk-throughs may be carried over directly and may be expected to yield benefits comparable to those experienced in the case of conventional software. The participants in a walk-through of AI software should probably be different than for conventional software: instead of Moderator, Designer, Implementor and Tester, we may choose to substitute Moderator, Domain Expert, Knowledge Engineer and Tester.

However, since AI software is usually founded upon some explicit or implicit *model* of the domain of interest, the real benefits of structured walk-throughs may best be obtained by examining this model, rather than (or as well as) its representation in the form of rules, or Lisp code. We have noted that systems based on "deep knowledge" generally contain an explicit model of the domain of interest, whereas those based on "surface knowledge" general consist simply of "rules." Yet these rules can be regarded as the encoding of some (implicit) model, and we hypothesize that substantial benefit may be obtained by extracting this model and making it explicit and subject to scrutiny.

Production-rules may best be likened to assembly language in conventional programming: a notation for instructing a machine (or an interpreter), not for expressing our thoughts. Just as conventional programmers now express themselves in, and perform analysis at the level of, high-level programming languages, so AI programmers may move away from rule-based encodings, towards notations that provide for the explicit representation of models. Indications of this development may be found in constraint programming languages [118] and in the work of Chandrasekaran [42] and of Neches and Swartout [138].

8.2.4 Comprehension Aids

Given the unstructured, assembly-language character of rule-based notations, programmers have understandably found the development, and more particularly the maintenance, of rule-based systems to be hazardous. The principle difficulty seems to lie in comprehending the global effect of a proposed modification, and several tools and methodologies have been developed to mitigate this problem. The techniques for anomaly detection described in Section 8.2.1 (page 90) can also be viewed from this perspective.

The simplest comprehension aids are cross-references between the places where attribute values are assigned, and those where they are used. Such cross-reference tables underlie the checking for circular rule chains in the Lockheed CHECK program [140], and also provide the basis for various forms of scrutiny advocated by Bellman and Walter [19].

Froscher and Jacob [68, 99, 100] and also Lindenmayer, Vick and Rosenthal [124] describe techniques for measuring the "coupling" between sets of rules, and propose methods for automatically partitioning rules into clusters of tightly related rules. Froscher and Jacob describe their motivation as being "to reduce the amount of information that each single knowledge engineer must understand before he or she can make a change to the knowledge base" [99]. They divide the rules into *groups* and then attempt to both "limit and formally specify the flow of information between these groups." In their method, each *fact* (an attribute-value pair) is characterized as being produced and used entirely within a single group (a *local* or *intragroup* fact), or as being produced or used by more than one group (an *intergroup* fact). Developers of groups that produce intergroup facts must provide assertions describing them. Froscher and Jacob [99, Page 5] claim that:

> "After a knowledge base is developed in this fashion, the knowledge engineer who wants to modify a group must understand

> the internal operations of that group, but not the rest of the knowledge base. If he or she preserves the correct functioning of the rules within the group and does not change the validity of the assertions about its intergroup facts, the knowledge engineer can be confident that the change that has been made will not adversely affect the rest of the system. Conversely, if the knowledge engineer wants to use additional intergroup facts from other groups, he or she should rely only on the assertions provided for them."

The motivation here is identical to that underlying the precept of "information hiding" [147, 148] in conventional software engineering.

Froscher and Jacob describe algorithms [68, 99] for taking a knowledge base not developed according to their methodology, and dividing it up into groups that are then suitable for maintenance in the manner described. The algorithm they favor [99] is based on cluster analysis. They also suggest metrics for evaluating the "cohesiveness" within groups and the "coupling" between groups, and describe planned experiments to evaluate the utility of their methodology. Lindenmayer, Vick and Rosenthal [124] describe an approach that is similar and motivation and technique to that of Froscher and Jacob.

Chapter 9

Reliability Assessment and Metrics for AI Systems

Chapter 3 (page 8) introduced the topic of software reliability modeling for conventional software. If AI systems are to be used in situations that require high and quantifiable degrees of reliability, then a similar body of measurement and modeling must be developed for AI software. At present, it seems that no systematic failure data is available for any AI software, and in the absence of such experimental data, it would be rash to estimate whether conventional software reliability models—such as the basic execution time model—will have any predictive validity for AI software.

Chapter 4 (page 19) described several of the metrics that have been developed for conventional software. The most useful of these seem to be the simplest, such as number of lines of code (SLOC). Similar simple measures, such as the number of rules, can easily be identified for AI systems. Buchanan [34] notes that rules are not the only measurable quantity in a knowledge base. Each rule mentions objects, their attributes, and the values those attributes may take. Buchanan suggests that the sum of these three could be considered as a measure of the "vocabulary" of a knowledge base. He notes that attributes that can take continuous values (such as weight) present difficulty in this scheme but gives values for the vocabularies of five well known expert systems (for example, 715+ for MYCIN, 4674 for INTERNIST). Other metrics mentioned by Buchanan include *size* of solution space (estimated at 10^9 for MYCIN[1]), and *complexity* of solution space.

[1]Computed as the number of combinations of from 1 to 6 organisms from a list of 120 potential organisms

Buchanan suggests that the latter can be quantified as b^d, where d is the average depth of search and b is the average branching factor. Buchanan estimates these quantities for MYCIN as $b = 5.5$ and $d = 4$, giving a complexity of about 1000 (5.5^4) nodes in the average search space. Buchanan also refers to experiments (e.g., by Greiner [81]) in which parts of a program are systematically excised and the subsequent failures analyzed. However, as Buchanan [34, Page 33] observes: "it is not easy to know which are the meaningful experiments, nor how to interpret the results."

These latter metrics and experiments resemble the more exotic metrics proposed for conventional software (such as Halstead's, and those that purport to estimate "complexity"). Such metrics seem to have little practical utility in the case of conventional software, and it seems unlikely that similar efforts will be of any greater value for AI software. Efforts to measure the "complexity" of such systems might be better directed towards "comprehension aids" that enable designers to control and cope with the complexity of their creations (see Section 8.2.4 on page 103).

The main value of program metrics is their use as predictors for development cost and failure intensity. In the absence of experimental data for AI software, it would be dangerous to assume that the empirical relationships observed in conventional software, for example that between SLOC and faults, will carry over to AI software. Empirical data will need to be gathered in order to suggest and validate useful relationships among metrics for AI software. It is unlikely that simple counts of the number of statements or rules in an AI system's knowledge base will yield useful information regarding its competence (the content of the rules may be more important than their number), but they may give a good indication of its computational requirements. Niwa *et al.* [141] conducted experiments along these lines. For each of four classes of AI system architectures (simple production system, structured production system, frame system, and logic system), they implemented two small prototype expert systems (one using forward chaining and one using backward chaining) to solve the same task. They measured several static attributes of each program, such as the size of the knowledge base, and also some dynamic attributes, including average inference time. In particular, they measured the time taken to solve three problems for each of three sizes of knowledge-base (500, 200, and 100 rules or axioms). The results showed an almost linear relationship between inference time and size of the knowledge base, though the constants of proportionality varied greatly across the different architectures considered.

Part III

Conclusions and Recommendations for Research

Chapter 10

Conclusions

AI software differs from conventional software in two significant ways: it generally addresses different kinds of problems, and it generally works in a different way, than does conventional software. On the other hand, AI software has much in common with conventional software: indeed, the majority of the software in the system will actually *be* of the conventional variety (for example, I/O—almost always the largest single component in any system—and the "inference engine" itself).

We believe that the best way to develop credible and effective quality assurance and evaluation techniques for AI software will be to identify the facets of such software that are *inherently*, or essentially, different from conventional software, and to distinguish them from those facets that are only *accidentally* or inessentially different. Inherent differences demand the development of new techniques; accidental differences—those due simply to culture, history and bias—require only the adaptation of established techniques (and possibly the elimination or reduction of those differences).

Principal among the accidental differences is the (apparent) lack of concern for life-cycle issues in AI software development, and the concomitant absence of a systematic software engineering methodology for AI systems. It is taken as axiomatic that AI software is developed incrementally, and relies heavily on rapid prototyping—but this should not be an excuse for rejecting such useful software engineering notions as requirements documentation and review, verification and validation, and systematic testing. The benefits of rapid prototyping and incremental development are recognized for conventional software as well as for AI software [29, 33]—but recognizing the value of prototypes does not sanction their release as products!

Refinement of requirements is one of the accepted purposes of prototyping: sometimes it is only through experiment that the best way of applying computers to a particular task can be found, or an acceptable method of solution discovered to a particular problem. There are instances where prototyping using AI software has led to sufficient clarification of issues and identification of potential solutions that the final system can be built as conventional software.

For other problems, an AI-based system will remain the preferred approach. Such problems generally have notions of success and failure that are less than clear-cut. At the "softest" extreme, we may have systems whose purpose is to provide "interesting" or "provocative" results. At the other extreme, an autonomous fault diagnosis system may have requirements that can be specified as rigorously as those for an accounting system (though achieving those requirements may be far more difficult). Somewhere in between come "decision aids" whose purpose is to assist a user to make correct decisions. In all cases, an attempt should be made (possibly by prototyping) to sharpen the requirements for the system. We suggest that our proposal to distinguish *service* from desired and minimum *competency* requirements could be useful in accomplishing this.

It seems plausible that those systems with the most rigorous QA demands (e.g., autonomous systems) are also likely to be those for which the most specific requirements statements can be produced. Minimum requirements for such systems are likely to include *consistency* (which for these purposes can loosely be identified with the notion that equivalent inputs should produce equivalent outputs) and completeness (all inputs should produce some output). One might then ask to what extent genuine AI software can achieve such requirements. Purists argue that AI software should not be limited by requirements for consistency and completeness—because that's not how people behave. While this may be a valid argument in the context of AI research, it is surely unacceptable when considering systems to be used in serious situations (e.g., advising pilots coping with emergencies). We seem to be led to the conclusion that software with the most stringent requirements should exhibit the least "AI-like" behavior.

The question then becomes, how can we best construct systems that exploit AI techniques, without incurring the frailties of "AI-like" behavior? It seems that this problem needs to be tackled on two levels: consideration of the knowledge base itself, and of its representation in a form that permits efficient deduction. In conventional knowledge engineering for expert systems, these two levels are inextricably intertwined. In crude terms, knowledge engineering begins by selecting some knowledge representation and associated deductive mechanism. Knowledge about the problem domain elicited from an expert is encoded in the selected representation and the resulting prototype is tried on small examples. By a process of experimentation with the prototype and interaction with the expert, the knowledge base and its representation are augmented and refined until the system seems to be working adequately. Now how can one argue for the correctness, completeness and consistency of the system so constructed?[1] How can one even extrapolate from behavior on test cases to the general case? How does one argue for the appropriateness of the set of rules actually constructed, rather than some other set?

Knowledge and Models

The basic problem, we submit, is that production rules, and other representations that permit reasonably effective deduction, are optimized for the efficient *use* of knowledge, not for supporting its scrutiny and examination. Production-rules and other knowledge representations used in much AI software may be likened to assembly language in conventional programming: a notation for instructing a machine (or an interpreter), not for expressing our thoughts.

In the early days of conventional software development, programmers may have actually created their programs directly in assembler or other low-level languages permitting efficient execution. Later, they recognized the value of high-level languages, of pseudo-code, and of both informal and formal specification languages as vehicles for expressing algorithms and system designs; programs might still be optimized and translated into lower-level notations for efficient execution, but most of the conception and the arguments for correctness, completeness, and consistency would consider the more abstract representations, rather than the program itself.

A similar development in techniques for representing knowledge is surely needed if trustworthy AI software is to become feasible. For critical tasks, knowledge engineering should best be considered as the incremental discovery and creation of a *model*[2] for the domain of interest and so one should seek representations that provide for the explicit representation of such mod-

[1]Especially when the knowledge base is large—there are production rule systems in use with over 5000 rules!

[2]**Model**: a simplified representation or description of a system or complex entity, especially one designed to facilitate calculations or predictions (Collins English Dictionary).

els. Suitable notations might include higher order logic and set theory. It is unlikely that a model represented in such a notation would permit usefully efficient deduction, but it would possess an internal coherence and permit causal explanations of behavior that are not possible for haphazard collections of rules. Knowledge engineering of a different form than that generally practiced would then seek to elaborate the representation of such a model into a form that would permit efficient deductions—in just the same way that conventional software engineering elaborates specifications written in very abstract form into executable programs. Indications of this sort of approach applied to AI may be seen in qualitative reasoning techniques [24], in constraint satisfaction systems [118], and in the work of Chandrasekaran [42], Neches, Swartout and Moore [138], and Abbott [3].

In our opinion, the explicit construction and scrutiny of models is an essential component of the knowledge engineering process for trustworthy AI systems. It is unlikely that users will trust an AI system, no matter how impressive some of its demonstrations of competence may be, without some causal explanation of its behavior. It is the underlying model, rather than its accidental representation in terms of rules, frames or other notations, that ultimately determines the competence and believability of the system, as well as its ability to explain its actions. The fact that human experts often do not have an explicit model of their domain of excellence is not an argument against our position: humans bring other attributes (e.g., common sense, self-awareness) to bear on their problem solving in addition to their "rule-base" and are, in any case, permitted a degree of fallibility that is not countenanced for machines that perform serious functions. Systems which address domains that are so ill-understood that the only source of guidance is an unstructured knowledge base extracted from expert's "war stories" may represent interesting AI research but they are not ready for deployment in serious situations.

Knowledge Representation

To be useful, knowledge needs to be organized and represented in such a way that it can be used to solve problems. The detailed design and representation of a knowledge base has much in common with conventional programming and requires support similar to that provided for software engineering [153]. Bobrow *et al.* [25] quote experienced knowledge engineers as follows:

> "Knowledge engineering is more than software engineering . . . but not much more."

Given this perspective, the lack of support in many AI programming systems for modern software engineering practices stressing strong typing, modularity, and information hiding is inexcusable, as is the lack of a formal—or even a precise—semantics. Given decent programming notations, there seems no good reason why the standards expected of conventional software at the program code level should not be extended to AI software. We believe that the determining factor in the deployment of AI software for serious applications may not be the *informal* evidence of how *well* it is able to perform, but the extent to which *formal* guarantees can be given on how *badly* it can perform—for example "the system generally seems to get as much material on the plane as an experienced quartermaster, but it is *guaranteed* never to load the plane in such a way that its center of gravity is outside the allowed range." Our proposal that the minimum competency requirement of a system should be distinguished from its desired competency level is intended to make feasible the construction of testable, verifiable, specifications of how "badly" a system is allowed to perform. Given precise minimum competency requirements, many design and assurance techniques from conventional software become available to AI software—for example, systematic testing, reliability analysis, fault-tolerance, design for safety, and mathematical verification.

Analysis of a knowledge base requires a semantics for the language in which it is represented. Technical issues such as the completeness and consistency of a knowledge base require a formal semantics if they are to be addressed with any rigor. And, of course, the formal semantics ascribed to the language must coincide with the operational semantics provided by its interpreter or inference engine. While one would expect an inference engine to be sound, it is quite likely to be incomplete in general.

Alternatives to AI-Based Approaches

In some domains, it may be possible to use conventional software to solve problems previously thought to require AI approaches. The activities of knowledge acquisition and prototyping may lead to a sufficiently improved understanding of requirements or solutions that a production-quality system can be built as conventional software. Given the more developed state of SQA for conventional software, this may be an advantage. Alternatively, the improved understanding resulting from prototyping may allow a shallow rule-based system to be replaced by one based on an explicit causal model.

When dealing with very ill-understood domains, however, there generally seems no alternative to shallow rule-based systems. Surprisingly, this may not be so. For a certain, limited class of problems, there is some evidence that expert levels of performance can be achieved using simple linear models. These models were discussed in Section 7.2.3 (page 78), and it was noted that psychologists had suggested, long before knowledge-based expert systems were mooted, that human experts might be replaced by linear models of their behavior. This process was called "bootstrapping" and its primary purpose was to *improve* the quality and consistency of decision making beyond that provided by human experts.

There is evidence that linear models can sometimes perform extremely well (see, for example, Dawes and Corrigan [54]), but very little data is available concerning their performance relative to knowledge-based systems. Carroll [36] gives a thoughtful discussion of the issues but cites no head-to-head comparisons. The only such comparison known to us concerns the prediction of hailstorm severity from radar images and is reported by Moninger *et al.* [132]. Linear regression models based on 75 data points were constructed for each of seven human forecasters. On comparing the predictions of the forecasters and their models to the known storm severities for these 75 cases, it was found that three of the seven forecasters were outperformed by their regression models, and that the models trailed the human forecasters only slightly in the other four cases. All but one of the seven regression models outperformed an expert system constructed to perform the same task, and the expert system was also outperformed by its own regression model! When a larger set of 453 test cases was considered, the expert system outperformed all the regression models, but an optimal regression model (fitted to the known outcomes of the 75 "training cases") came very close to the expert system.[3]

Linear regression models can be perturbed by just a few outliers among the data points, and it is possible that the performance of the linear models in the experiment cited above could be improved using techniques, such as "RANSAC" [67], for reducing their effects. Even without speculating upon possible improvements, the experiment described above certainly suggests that the performance of linear models might be competitive with that of expert systems for certain tasks—at a tiny fraction of the cost. Of course,

[3]The skill measure (serial correlation coefficient) for the expert system was 0.38, that for the optimal linear model was 0.36, and those for the other linear models ranged from 0.32 to 0.34.

linear models have obvious disadvantages when compared with knowledge-based systems—no explanation facility, and no recognition of exceptional cases, for example. An intriguing possibility would be to attempt to combine the good features of both paradigms. The fact that their failure modes seem complementary makes this a particularly attractive line of investigation.

10.1 Recommendations for Research

Given the perspective described in this chapter, we recommend the initiation of a vigorous program of research and development on the topic of quality assurance and measurement for AI systems. The program should consider both the application and adaptation of suitable techniques from conventional software engineering, as well as the development of new techniques that address the unique characteristics of AI software. Some of the research areas that seem to warrant study over the near and middle term include:

Near Term

- For critical applications at least, abandon the use of rule-based expert systems whose rule sets are unsupported by causal models.

- Explore the value of the distinction between service and competency requirements, and between minimum and desired competency requirements for AI software. Study the construction of precise, testable specifications for service and minimum competency requirements.

- Develop a basis for the systematic dynamic testing of AI software. In particular, develop structural testing methods comparable to path testing for conventional software, and develop methods for functional and random testing of AI software. Develop test-case generation algorithms and evaluate them in practice. Investigate the potential of model-based adversaries for test case generation and evaluation. Investigate the value of sensitivity analysis.

- Investigate the general applicability and performance of linear and other simple mathematical models. Study the utility of such models as adversaries in the systematic testing of AI systems. Explore the possibility of hybrid systems with both knowledge-based and linear model based components.

- Develop realistic life-cycle approaches to the development and acquisition of AI software. Develop practical, if modest, quality assurance methodologies based on available techniques.

Longer Term

- Develop methodologies and notations in support of knowledge engineering as an explicit model-building activity. Investigate the feasibility of "compiling" certain classes of models into production rules (or other efficient representations), or of verifying a set of production rules against an explicit model.

- Develop techniques for integrating explanation more closely into the development and maintenance of AI software.

- Develop improved languages for knowledge representation—languages with a sound semantics.

- Develop software engineering practices and tools for AI systems. For example, establish a sound theoretical basis for rule coupling metrics, and rule partitioning algorithms. Develop good algorithms for these and also tools for aiding rule-comprehension. Evaluate in practice.

- Establish a sound theoretical basis for completeness and consistency checking of rule-bases. Develop good algorithms and evaluate them in practice. Investigate the problem of establishing termination for rule-based systems.

- Investigate the influence of conflict resolution strategies; study the feasibility and utility of a test mode in which *all* successful inference paths are followed.

- Investigate system structures to *guarantee* certain minimum competency (safety) requirements.

- Collect empirical reliability data; study the applicability of established reliability growth models; develop new models if necessary, and validate them in practice.

- Identify useful metrics for AI based software and validate their value as predictors of important cost, performance and reliability measures.

Closing Remarks

Until very recently, knowledge-based systems were an exploratory research topic and little attention was paid to their quality assurance. Now that some knowledge-based systems, especially those known as “rule-based expert systems,” are being placed into operational environments, it is important to develop a software quality assurance methodology for such systems.

In this report, we hope to have shown how some of the quality assurance and measurement techniques developed for conventional software can be adapted and applied to knowledge-based systems. However, while such adaptations of existing techniques would be beneficial and important, they tend to focus on representational and implementation issues—whereas it is the knowledge encoded in the system that ultimately determines its quality of performance. It is entirely feasible to implement a knowledge-based system as a program in Basic—but to evaluate that system as simply a Basic program surely misses the point. It is the knowledge that it embodies, as much as its representation in Basic, that needs to be examined. Thus, the way to focus on the distinctive problems of quality assurance for AI software may be through the development of techniques for describing, evaluating, and manipulating knowledge in ways that are abstracted from the concrete details of a particular representation, in much the same way as algorithms are abstracted from programs. This represents an exciting challenge for the future.

Bibliography

[1] Kathy H. Abbott. Robust operative diagnosis as problem solving in a hypothesis space. In *Proceedings, AAAI 88 (Volume 1)*, pages 369–374, Saint Paul, MN, August 1988.

[2] Kathy H. Abbott, Paul C. Schutte, Michael T. Palmer, and Wendell R. Ricks. Faultfinder: A diagnostic expert system with graceful degradation for onboard aircraft applications. In *Proceedings, 14th Symposium on Aircraft Integrated Monitoring Systems*, Friedrichshafen, W. Germany, September 1987.

[3] Russell J. Abbott. Knowledge abstraction. *Communications of the ACM*, 30(8):664–671, August 1987.

[4] Maryam Alavi. An assessment of the prototyping approach to information systems development. *Communications of the ACM*, 27(6):556–563, June 1984.

[5] M. W. Alford. A requirements engineering methodology for real–time processing requirements. *IEEE Transactions on Software Engineering*, SE–3(1):60–69, January 1977.

[6] M. W. Alford. SREM at the age of eight; the distributed computing design system. *IEEE Computer*, 18(4):36–46, April 1985.

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

[8] T. Anderson, P. A. Barrett, D. N. Halliwell, and M. R. Moulding. An evaluation of software fault tolerance in a practical system. In *Fault Tolerant Computing Symposium 15*, pages 140–145, Ann Arbor, MI, June 1985. IEEE Computer Society.

[9] T. Anderson and P. A. Lee. *Fault-Tolerance: Principles and Practice (Second, revised edition).* Springer Verlag, Wien and New York, 1990.

[10] T. Anderson and R. W. Witty. Safe programming. *BIT*, 18:1–8, 1978.

[11] Dorothy M. Andrews and Jeoffrey P. Benson. An automated program testing methodology and its implementation. In *Proceedings, 5th International Conference on Software Engineering*, pages 254–261, San Diego, CA, March 1981.

[12] Geoff Baldwin. Implementation of physical units. *SIGPLAN Notices*, 22(8):45–50, August 1987.

[13] Sheldon Baron and Carl Feehrer. An analysis of the application of AI to the development of intelligent aids for flight crew tasks. Contractor Report 3944, NASA Langley Research Center, Hampton, VA, October 1985.

[14] V. R. Basili and B. T. Perricone. Software errors and complexity: An empirical investigation. *Communications of the ACM*, 27(1):42–52, January 1984.

[15] Farokh B. Bastani and S. Sitharama Iyengar. The effect of data structures on the logical complexity of programs. *Communications of the ACM*, 30(3):250–259, March 1987.

[16] Michael Z. Bell. Why expert systems fail. *Journal of the Operational Research Society*, 36(7):613–619, 1985.

[17] T. E. Bell, D. C. Bixler, and M. E. Dyer. An extendable approach to computer–aided software requirements engineering. *IEEE Transactions on Software Engineering*, SE–3(1):49–59, January 1977.

[18] Kirstie L. Bellman. Testing and correcting rule-based expert systems. In *Proceedings of the Space Quality Conference*. NSIA/AIA (Space Division) and NASA, April 1988. Published by NSIA, Washington, D. C.

[19] Kirstie L. Bellman and Donald O. Walter. Testing rule-based expert systems. Course Notes for "Analyzing the Reliability and Performance of Expert Systems," UCLA Extension., December 1987.

[20] Jean-François Bergeretti and Bernard A. Carré. Information-flow and data-flow analysis of **while**-programs. *ACM Transactions on Programming Languages and Systems*, 7(1):37–61, January 1985.

[21] Gerald M. Berns. Assessing software maintainability. *Communications of the ACM*, 27(1):14–23, January 1984.

[22] R. E. Berry and B. A. E. Meekings. A style analysis of C programs. *Communications of the ACM*, 28(1):80–88, January 1985.

[23] J. Bliss, P. Feld, and R. E. Hayes. Decision aid measurement and evaluation (DAME). In *Proceedings, International Conference on Systems, Man, and Cybernetics*, pages 126–130, Atlanta, GA, October 1986. IEEE.

[24] Daniel G. Bobrow, editor. *Qualitative Reasoning about Physical Systems.* The MIT Press, Cambridge, MA, 1986.

[25] Daniel G. Bobrow, Sanjay Mittal, and Mark J. Stefik. Expert systems: Perils and promise. *Communications of the ACM*, 29(9):880–894, September 1986.

[26] Barry W. Boehm. *Software Engineering Economics.* Prentice-Hall, Englewood Cliffs, NJ, 1981.

[27] Barry W. Boehm. Software engineering economics. *IEEE Transactions on Software Engineering*, SE-10(1):4–21, January 1984.

[28] Barry W. Boehm. Verifying and validating software requirements. *IEEE Software*, 1(1):75–88, January 1984.

[29] Barry W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, May 1988.

[30] Barry W. Boehm, Terence E. Gray, and Thomas Seewaldt. Prototyping versus specifying: A multiproject experiment. *IEEE Transactions on Software Engineering*, SE-10(3):290–303, May 1984.

[31] R. S. Boyer, B. Elspas, and K. N Levitt. SELECT: A formal system for testing and debugging programs by symbolic execution. In *Proceedings, International Conference on Reliable Software*, pages 234–245, Los Angeles, CA, April 1975. IEEE Computer Society.

[32] Susan S. Brilliant and John C. Knight ans Nancy G. Leveson. The consistent comparison problem in N-Version software. *IEEE Transactions on Software Engineering*, 15(11):1481–1485, November 1989.

[33] Frederick P. Brooks, Jr. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, April 1987.

[34] Bruce G. Buchanan. Artificial intelligence as an experimental science. Technical Report KSL 87-03, Knowledge Systems Laboratory, Stanford University, Stanford, CA, January 1987. To appear in *Synthese*.

[35] T. A. Budd, R. A. De Millo, R. J. Lipton, and F. G. Sayward. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In *Proceedings, 7th ACM Symposium on the Principles of Programming Languages*, Las Vegas, NV, January 1980.

[36] Barbara Carroll. Expert systems for clinical diagnosis: Are they worth the effort? *Behavioral Science*, 32:274–292, 1987.

[37] Glen Castore. A formal approach to validation and verification for knowledge-based control systems. In *First Annual Workshop on Space Operations, Automation, and Robotics (SOAR 87)*, pages 197–202, Houston, TX, August 1987. NASA Conference Publication 2491.

[38] J. Celko, J. S. Davis, and J. Mitchell. A demonstration of three requirements language systems. *SIGPLAN Notices*, 18(1):9–14, January 1983.

[39] 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.

[40] Fun Ting Chan and Tsong Hueh Chen. AIDA—a dynamic data flow anomaly detection system for Pascal programs. *Software—Practice and Experience*, 17(3):227–239, March 1987.

[41] B. Chandrasekaran. On evaluating AI systems for medical diagnosis. *AI Magazine*, 4(2):34–37, Summer 1983.

[42] B. Chandrasekaran. Generic tasks in knowledge-based reasoning: High-level building blocks for expert system design. *IEEE Expert*, 1(3):23–30, Fall 1986.

[43] M. Cheheyl et al. Verifying security. *Computing Surveys*, 13(3):279–339, September 1981.

[44] L. Chen and A. Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Digest of Papers, FTCS 8*, pages 3–9, Toulouse, France, June 1978.

[45] Lori Clarke, Andy Podgurski, Debra Richardson, and Steven Zeil. A formal evaluation of data flow path selection criteria. *IEEE Transactions on Software Engineering*, 15(11):1318–1332, November 1989.

[46] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog*. Springer-Verlag, New York, NY, 1981.

[47] Robert F. Cmelik and Narain H. Gehani. Dimensional analysis with C++. *IEEE Software*, 5(3):21–27, May 1988.

[48] S. D. Conte, H. E. Dunsmore, and V. Y. Shen. *Software Engineering Metrics and Models*. Benjamin/Cummings, Menlo Park, CA, 1986.

[49] Nancy M. Cooke and James E. McDonald. A formal methodology for acquiring and representing expert knowledge. *Proceedings of the IEEE*, 74(10):1422–1430, October 1986.

[50] Chris Culbert, Gary Riley, and Robert T. Savely. Approaches to verification of rule-based expert systems. In *First Annual Workshop on Space Operations, Automation, and Robotics (SOAR 87)*, pages 191–196a, Houston, TX, August 1987. NASA Conference Publication 2491.

[51] P. Allen Currit, Michael Dyer, and Harlan D. Mills. Certifying the reliability of software. *IEEE Transactions on Software Engineering*, SE-12(1):3–11, January 1986.

[52] C. G. Davis and C. R. Vick. The software development system. *IEEE Transactions on Software Engineering*, SE–3(1):69–84, January 1977.

[53] R. Davis. *Teiresias: Applications of Meta-Level Knowledge to the Construction, Maintenance, and Use of Large Knowledge Bases*. PhD thesis, Computer Science Department, Stanford University, CA., 1976.

Reprinted with revisions in "Knowledge-Based Systems in Artificial Intelligence," R. Davis and D. B. Lenat, eds., McGraw-Hill, New York, NY., 1980.

[54] Robyn M. Dawes and Bernard Corrigan. Linear models in decision making. *Psychological Bulletin*, 81(2):95–106, February 1974.

[55] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5):271–280, May 1979.

[56] Saumya K. Debray and Prateek Mishra. Denotational and operation semantics for Prolog. *Journal of Logic Programming*, 5(1):61–91, March 1988.

[57] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. ACM*, 20(7):504–513, July 1977.

[58] Peter J. Denning. Towards a science of expert systems. *IEEE Expert*, 1(2):80–83, Summer 1986.

[59] *Department of Defense Trusted Computer System Evaluation Criteria*. Dept. of Defense, National Computer Security Center, Dec. 1985. DOD 5200.28-STD.

[60] Thomas Downs. An approach to the modeling of software testing with some applications. *IEEE Transactions on Software Engineering*, SE-11(4), April 1985.

[61] Joe W. Duran and Simeon C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, SE-10(4):438–443, April 1984.

[62] Dave E. Eckhardt, Alper K. Caglayan, John C. Knight, Larry D. Lee, David F. McAllister, Mladen A. Vouk, and John P. J. Kelly. An experimental evaluation of software redundancy as a strategy for improving reliability. *IEEE Transactions on Software Engineering*, 17(7):692–702, July 1991.

[63] Michael E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, March 1976.

[64] Michael E. Fagan. Advances in software inspection. *IEEE Transactions on Software Engineering*, SE-12(7):744–751, July 1986.

[65] M. A. Fischler and O. Firschein. A fault-tolerant multiprocessor architecture for real-time control applications. In *First Annual Symposium in Computer Architecture*, pages 151–157, December 1973.

[66] M. A. Fischler, O. Firschein, and D. L. Drew. Distinct software: An approach to reliable computing. In *Second Annual USA-Japan Computer Conference*, pages 27–4–1–27–4–7, Tokyo, Japan, August 1975.

[67] Martin A. Fischler and Robert C. Bolles. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, June 1981.

[68] Judith N. Froscher and Robert J. K. Jacob. Designing expert systems for ease of change. In *Proceedings, First Annual Expert Systems in Government Symposium*, pages 246–251. IEEE Computer Society, 1985.

[69] Kokichi Futatsugi, Joseph Goguen, Jean-Pierre Jouannaud, and José Meseguer. Principles of OBJ2. In Brian K. Reid, editor, *Proceedings of 12th ACM Symposium on Principles of Programming Languages*, pages 52–66. Association for Computing Machinery, 1985.

[70] John Gaschnig, Philip Klahr, Harry Pople, Edward Shortliffe, and Allan Terry. Evaluation of expert systems: Issues and case studies. In F. Hayes-Roth, D. A. Waterman, and D. B. Lenat, editors, *Building Expert Systems*, chapter 8. Addison-Wesley, Reading, MA, 1983.

[71] James R. Geissman and Roger D. Schultz. Verification and validation of expert systems. *AI Expert*, pages 26–33, February 1988.

[72] David Gelperin and Bill Hetzel. The growth of software testing. *Communications of the ACM*, 31(6):687–695, June 1988.

[73] Michael R. Genesereth and Nils J. Nilsson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann Publishers Inc., Los Altos, CA, 1987.

[74] Michael P. Georgeff. Planning. In Joseph F. Traub, Barbara J. Grosz, Butler W. Lampson, and Nils J. Nilsson, editors, *Annual Review of Computer Science, Volume 2*, pages 359–400. Annual Reviews, Inc., Palo Alto, CA, 1987.

[75] S. German. Automating proofs of the absence of common runtime errors. In *Proceedings, 5th ACM Symposium on the Principles of Programming Languages*, pages 105–118, Tucson, AZ, January 1978.

[76] Allen Ginsberg. A new approach to checking knowledge bases for inconsistency and redundancy. In *Proceedings, Third Annual Expert Systems in Government Symposium*, pages 102–111, Washington, D. C., October 1987. IEEE Computer Society.

[77] Allen Ginsberg. Knowledge-base reduction: A new approach to checking knowledge-bases for inconsistency and redundancy. In *Proceedings, AAAI 88 (Volume 2)*, pages 585–589, Saint Paul, MN, August 1988.

[78] John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, SE-1(2):156–173, June 1975.

[79] Mary Ann Goodwin and Charles C. Robertson. Expert system verification concerns in an operations environment. In *First Annual Workshop on Space Operations, Automation, and Robotics (SOAR 87)*, pages 203–207, Houston, TX, August 1987. NASA Conference Publication 2491.

[80] Christopher J. R. Green and Marlene M. Keyes. Verification and validation of expert systems. In *Proceedings, Western Conference on Expert Systems*, pages 38–43, Anaheim, CA, June 1987. IEEE Computer Society.

[81] Russell Greiner. *Learning by Understanding Analogies.* PhD thesis, Computer Science Department, Stanford University, CA., 1985. Issued as Stanford Technical Report CS-85-1071.

[82] J. V. Guttag and J. J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10(1):27–52, 1978.

[83] M. H. Halstead. *Elements of Software Science.* Elsevier North-Holland, New York, NY, 1977.

[84] Paul Harmon and David King. *Expert Systems: Artificial Intelligence in Business.* John Wiley and sons, New York, NY, 1985.

[85] Warren Harrison and Curtis R. Cook. A note on the Berry-Meekings style metric. *Communications of the ACM*, 29(2):123–125, February 1986.

[86] Ian Hayes, editor. *Specification Case Studies.* Prentice-Hall International (UK) Ltd., Hemel Hempstead, UK, 1987.

[87] Ian J. Hayes. Specification directed module testing. *IEEE Transactions on Software Engineering*, SE-12(1):124–133, January 1986.

[88] S. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, SE-7(5):510–518, September 1981.

[89] S. Henry and D. Kafura. The evaluation of software systems' structure using quantitative software metrics. *Software—Practice and Experience*, 14(6):561–573, June 1984.

[90] Paul N. Hilfinger. An Ada package for dimensional analysis. *ACM Transactions on Programming Languages and Systems*, 10(2):189–203, April 1988.

[91] William E. Howden. Symbolic testing and the DISSECT symbolic evaluation system. *IEEE Transactions on Software Engineering*, SE-4(1):70–73, January 1977.

[92] William E. Howden. An evaluation of the effectiveness of symbolic testing. *Software—Practice and Experience*, 8:381–397, 1978.

[93] William E. Howden. Functional program testing. *IEEE Transactions on Software Engineering*, SE-6(2):162–169, March 1980.

[94] William E. Howden. Software validation techniques applied to scientific programs. *ACM Transactions on Programming Languages and Systems*, 2(3):307–320, July 1980.

[95] William E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, SE-8(4):371–379, July 1982.

[96] William E. Howden. A functional approach to program testing and analysis. *IEEE Transactions on Software Engineering*, SE-12(10):997–1005, October 1986.

[97] D. C. Ince. The automatic generation of test data. *Computer Journal*, 30(1):63–69, February 1987.

[98] D. C. Ince and S. Hekmatpour. An empirical evaluation of random testing. *Computer Journal*, 29(4):380, August 1986.

[99] Robert J. K. Jacob and Judith N. Froscher. Developing a software engineering methodology for knowledge-based systems. Technical Report 9019, Naval Research Laboratory, Washington, D. C., December 1986.

[100] Robert J. K. Jacob and Judith N. Froscher. Software engineering for rule-based systems. In *FJCC*, pages 185–189, 1986.

[101] Cliff B. Jones. *Software Development: A Rigorous Approach.* Prentice-Hall, Englewood Cliffs, NJ, 1986.

[102] T. C. Jones. *Programming Productivity.* McGraw Hill, New York, NY, 1986.

[103] D. Kapur and D. R. Musser. Proof by consistency. *Artificial Intelligence*, 31(2):125–157, February 1987.

[104] Deepak Kapur and Mandayam Srivas. Computability and implementability issues in abstract data types. *Science of Computer Programming*, 10:33–63, 1988.

[105] M. Karr and D. B. Lovemann III. Incorporation of units into programming languages. *Communications of the ACM*, 21(5):385–391, May 1978.

[106] Joseph K. Kearney, Robert L. Sedlmeyer, William B. Thompson, Michael A. Gray, and Michael A. Adler. Software complexity measurement. *Communications of the ACM*, 29(11):1044–1050, November 1986.

[107] R. L. Keeney and H. Raiffa. *Decision with Multiple Objectives.* Wiley, 1976.

[108] Chris F. Kemerer. An empirical validation of software cost estimation models. *Communications of the ACM*, 30(5):416–429, May 1987.

[109] Richard A. Kemmerer. Testing formal specifications to determine design errors. *IEEE Transactions on Software Engineering*, SE-11(1):32–43, January 1985.

[110] Richard A. Kemmerer. Verification assessment study final report. Technical Report C3-CR01-86, National Computer Security Center, Ft. Meade, MD, 1986. 5 Volumes. US distribution only.

[111] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[112] 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.

[113] 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.

[114] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–293. Pergamon, New York, NY, 1970.

[115] Thomas J. Laffey, Walton A. Perkins, and Tin A. Nguyen. Reasoning about fault diagnosis with LES. *IEEE Expert*, 1(1):13–20, Spring 1986.

[116] L. Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, January 1985.

[117] N. E. Lane. Global issues in evaluation of expert systems. In *Proceedings, International Conference on Systems, Man, and Cybernetics*, pages 121–125, Atlanta, GA, October 1986. IEEE.

[118] Wm Leler. *Constraint Programming Languages: Their Specification and Generation.* Addison-Wesley, Reading, MA, 1988.

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

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

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

[122] Nancy G. Leveson and John C. Knight. On N-Version programming. *ACM Software Engineering Notes*, 15(1):24–35, January 1990.

[123] Jay Liebowitz. Useful approach for evaluating expert systems. *Expert Systems*, 3(2):86–96, April 1986.

[124] Kelly Lindenmayer, Shon Vick, and Don Rosenthal. Maintaining an expert system for the Hubble space telescope ground support. In *Proceedings, Goddard Conference on Space Applications of Artificial Intelligence and Robotics*, pages 1–13, Greenbelt, MD, May 1987. NASA Goddard Space Flight Center.

[125] Robert Mandl. Orthogonal Latin squares: An application of experiment design to compiler testing. *Communications of the ACM*, 28(10):1054–01058, October 1985.

[126] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, December 1976.

[127] Drew V. McDermott. Logic, problem solving and deduction. In Joseph F. Traub, Barbara J. Grosz, Butler W. Lampson, and Nils J. Nilsson, editors, *Annual Review of Computer Science, Volume 2*, pages 187–229. Annual Reviews, Inc., Palo Alto, CA, 1987.

[128] J. McDermott. R1: The formative years. *AI Magazine*, 2(2):21–29, Summer 1981.

[129] Bertrand Meyer. On formalism in specifications. *IEEE Software*, 2(1):6–26, January 1985.

[130] R. A. De Millo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[131] Harlan D. Mills, Michael Dyer, and Richard Linger. Cleanroom software engineering. *IEEE Software*, 4(5):19–25, September 1987.

[132] William R. Moninger, Thomas R. Stewart, and Patrick McIntosh. Validation of knowledge-based systems for probabilistic forecasting. In *Proceedings of AAAI 88 Workshop on Validation and Testing Knowledge-Based Systems*, Saint Paul, MN, August 1988.

[133] M. Moriconi and D.F. Hare. The PegaSys system: Pictures as formal documentation of large programs. *ACM Transactions on Programming Languages and Systems*, 8(4):524–546, October 1986.

[134] John D. Musa, Anthony Iannino, and Kazuhira Okumoto. *Software Reliability—Measurement, Prediction, Application.* McGraw Hill, New York, NY, 1987.

[135] David R. Musser. Abstract data type specification in the AFFIRM system. *IEEE Transactions on Software Engineering*, SE-6(1):24–32, January 1980.

[136] G. J. Myers. A controlled experiment in program testing and code walkthroughs/inspections. *Communications of the ACM*, 21(9):760–768, September 1978.

[137] Peter Naur. Formalization in program development. *BIT*, 22:437–453, 1982.

[138] Robert Neches, William R. Swartout, and Johanna D. Moore. Enhanced maintenance and explanation of expert systems through explicit models of their development. *IEEE Transactions on Software Engineering*, SE-11(11):1337–1351, November 1985.

[139] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.

[140] Tin A. Nguyen, Walton A. Perkins, Thomas J. Laffey, and Deanne Pecora. Knowledge base verification. *AI Magazine*, 8(2):65–79, Summer 1987.

[141] K. Niwa, K. Sasaki, and H. Ihara. An experimental comparison of knowledge representation schemes. *AI Magazine*, 5(2):29–36, Summer 1984.

[142] Simeon C. Ntafos. A comparison of some structural testing strategies. *IEEE Transactions on Software Engineering*, SE-14(6):868–874, June 1988.

[143] Robert M. O'Keefe, Osman Balci, and Eric P. Smith. Validating expert system performance. *IEEE Expert*, 2(4):81–90, Winter 1987.

[144] L. J. Osterweil and L. D. Fosdick. DAVE—a validation error detection and documentation system for Fortran programs. *Software—Practice and Experience*, 6:473–486, October–December 1976.

[145] Thomas J. Ostrand and Marc J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, June 1988.

[146] Michael T. Palmer, Kathy H. Abbott, Paul C. Schutte, and Wendell R. Ricks. Implementation of a research prototype onboard fault monitoring and diagnosis system. In *Proceedings, AIAA Computers in Aerospace Conference*, Wakefield, MA, October 1987.

[147] D. L. Parnas. Information distribution aspects of design methodology. In *IFIP Congress Proceedings*, Ljubljana, Yugoslavia, 1971.

[148] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[149] David L. Parnas. Software aspects of strategic defense systems. *American Scientist*, 73(5):432–440, September–October 1985.

[150] L. F. Pau. Prototyping, validation and maintenance of knowledge based systems software. In *Proceedings, Third Annual Expert Systems in Government Symposium*, pages 248–253, Washington, D. C., October 1987. IEEE Computer Society.

[151] D. A. Pearce. KIC: A knowledge integrity checker. Technical Report TIRM 87–025, The Turing Institute, Glasgow, Scotland, 1987.

[152] D. A. Pearce. The induction of fault diagnosis systems from qualitative models. In *Proceedings, AAAI 88 (Volume 1)*, pages 353–357, Saint Paul, MN, August 1988.

[153] C. V. Ramamoorthy, Shashi Shekhar, and Vijay Garg. Software development support for AI programs. *IEEE Computer*, 20(1):30–40, January 1987.

[154] Sandra Rapps and Elaine J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, April 1985.

[155] K. A. Redish and W. F. Smyth. Program style analysis: A natural by-product of program compilation. *Communications of the ACM*, 29(2):126–133, February 1986.

[156] K. A. Redish and W. F. Smyth. Evaluating measures of program quality. *Computer Journal*, 30(3):228–232, June 1987.

[157] Raymond Reiter. Nonmonotonic reasoning. In Joseph F. Traub, Barbara J. Grosz, Butler W. Lampson, and Nils J. Nilsson, editors, *Annual Review of Computer Science, Volume 2*, pages 147–186. Annual Reviews, Inc., Palo Alto, CA, 1987.

[158] Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57–95, 1987.

[159] Wendell R. Ricks and Kathy H. Abbott. Traditional versus rule-based programming techniques: Application to the control of optional flight information. NASA Technical Memorandum 89161, NASA Langley Research Center, Hampton, VA, July 1987.

[160] G-C. Roman. A taxonomy of current issues in requirements engineering. *IEEE Computer*, 18(4):14–21, April 1985.

[161] W. W. Royce. Managing the development of large software systems. In *Proceedings WESCON*, August 1970.

[162] John Rushby. Formal specification and verification of a fault-masking and transient-recovery model for digital flight-control systems. Technical Report SRI-CSL-91-3, Computer Science Laboratory, SRI International, Menlo Park, CA, January 1991. Also available as NASA Contractor Report 4384.

[163] John Rushby and Friedrich von Henke. Formal verification of the Interactive Convergence clock synchronization algorithm using EHDM.

Technical Report SRI-CSL-89-3R, Computer Science Laboratory, SRI International, Menlo Park, CA, February 1989 (Revised August 1991). Original version also available as NASA Contractor Report 4239.

[164] John Rushby and Friedrich von Henke. Formal verification of algorithms for critical systems. In *SIGSOFT '91: Software for Critical Systems*, pages 1–15, New Orleans, LA, December 1991. Published as ACM SIGSOFT Engineering Notes, Volume 16, Number 5.

[165] John Rushby, Friedrich von Henke, and Sam Owre. An introduction to formal specification and verification using EHDM. Technical Report SRI-CSL-91-2, Computer Science Laboratory, SRI International, Menlo Park, CA, February 1991.

[166] E. T. Scambos. A scenario-based test-tool for examining expert systems. In *Proceedings, International Conference on Systems, Man, and Cybernetics*, pages 131–135, Atlanta, GA, October 1986. IEEE.

[167] P. A. Scheffer, A. H. Stone III, and W. E. Rzepka. A case study of SREM. *IEEE Computer*, 18(4):47–54, April 1985.

[168] Paul C. Schutte and Kathy H. Abbott. An artificial intelligence approach to onboard fault monitoring and diagnosis for aircraft applications. In *Proceedings, AIAA Guidance and Control Conference*, Williamsburg, VA, August 1986.

[169] Paul C. Schutte, Kathy H. Abbott, Michael T. Palmer, and Wendell R. Ricks. An evaluation of a real-time fault diagnosis expert system for aircraft applications. In *Proceedings, 26th IEEE Conference on Decision and Control*, Los Angeles, CA, December 1987.

[170] Richard W. Selby, Victor R. Basili, and F. Terry Baker. Cleanroom software development: An empirical evaluation. *IEEE Transactions on Software Engineering*, SE-13(9):1027–1037, September 1987.

[171] N. Shankar. A mechanical proof of the Church-Rosser theorem. *Journal of the ACM*, 35(3):475–522, July 1988.

[172] Vincent Shen. Editor's introduction to "Quality Time" department. *IEEE Software*, 4(5):84, September 1987.

[173] Edward Hance Shortliffe. *Computer-Based Medical Consultations: MYCIN*. ELSEVIER, New York, NY, 1976.

[174] Robert E. Shostak. A practical decision procedure for arithmetic with function symbols. *Journal of the ACM*, 26(2):351–360, April 1979.

[175] Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, January 1984.

[176] J. M. Silverman. Reflections on the verification of the security of an operating system kernel. In *Proc. 9th ACM Symposium on Operating Systems Principles*, pages 143–154, Bretton Woods, NH, October 1983. (ACM Operating Systems Review, Vol 17, No. 5).

[177] R. A. Stachowitz, C. L. Chang, T. S. Stock, and J. B. Combs. Building validation tools for knowledge-based systems. In *First Annual Workshop on Space Operations, Automation, and Robotics (SOAR 87)*, pages 209–216, Houston, TX, August 1987. NASA Conference Publication 2491.

[178] Rolf A. Stachowitz and Jacqueline B. Combs. Validation of expert systems. In *Proceedings, Hawaii International Conference on System Sciences*, Kona, HI, January 1987.

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

[180] R. E. Strom. Mechanisms for compile-time enforcement of security. In *Proceedings 10th Symposium on Principles of Programming Languages*, pages 276–284, Austin, TX, January 1983.

[181] Motoi Suwa, A. Carlisle Scott, and Edward H. Shortliffe. An approach to verifying completeness and consistency in a rule-based expert system. *AI Magazine*, 3(4):16–21, Fall 1982.

[182] William R. Swartout. XPLAIN: A system for creating and explaining expert consulting programs. *Artificial Intelligence*, 21:285–325, 1983.

[183] John A. Swets. The relative operating characteristic in psychology. *Science*, 182:990–1000, 1973.

[184] E. L. Thorndike. Fundamental theorems in judging men. *Journal of Applied Psychology*, 2:67–76, 1918.

[185] Richard M. Tong, Neville D. Newman, Gary Berg-Cross, and Fred Rook. Performance evaluation of artificial intelligence systems. Technical Report ADS TR-3154-01, Advanced Decision Systems, Mountain View, CA, August 1987. Available from DTIC as number AD-A184 054.

[186] A. M. Turing. Computing machinery and intelligence. *Mind*, 59, 1950. Reprinted in "Computers and Thought," Feigenbaum and Feldman, eds., McGraw-Hill, New York, NY., 1963.

[187] 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.

[188] F. W. von Henke and D. C. Luckham. A methodology for verifying programs. In *Proceedings, International Conference on Reliable Software*, pages 156–164, Los Angeles, CA, April 1975. IEEE Computer Society.

[189] Leslie J. Waguespack, Jr. and Sunil Badlani. Software complexity assessment: An introduction and annotated bibliography. *ACM Software Engineering Notes*, 12(4):52–71, October 1987.

[190] Elaine J. Weyuker. On testing non-testable programs. *Computer Journal*, 25(4):465–470, April 1982.

[191] Elaine J. Weyuker. The evaluation of program-based software test data adequacy criteria. *Communications of the ACM*, 31(6):668–675, June 1988.

[192] Elaine J. Weyuker and Thomas J. Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE Transactions on Software Engineering*, SE-6(3):236–246, May 1980.

[193] David C. Wilkins and Bruce G. Buchanan. On debugging rule sets when reasoning under uncertainty. In *Proceedings, AAAI 86 (Volume 1)*, pages 448–454, Philadelphia, PA, August 1986.

[194] C. Wilson and L. J. Osterweil. Omega—a data flow analysis tool for the C programming language. In *Proceedings COMPSAC*, pages 9–18, 1982.

[195] Jeannette M. Wing. A study of 12 specifications of the library problem. *IEEE Software*, 5(4):66–76, July 1988.

[196] Patrick Henry Winston. *Artificial Intelligence*. Addison-Wesley, Reading, Massachusetts, second edition, 1984.

[197] David D. Woods. Cognitive technologies: The design of joint human-machine cognitive systems. *AI Magazine*, 6(4):86–91, Winter 1986.

[198] V. L. Yu *et al.* Antimicrobial selection by a computer: A blinded evaluation by infectious disease experts. *Journal of the American Medical Association*, 242:1279–1282, 1979.