Softwarezuverlässigkeit und Programmiersprache

W. Ehrenberger, Garching

Kurzfassung

Im Rahmen der Diskussion um die Vorzüge und Nachteile einzelner Rechnersprachen, der ja auch PEARL unterworfen ist, tritt mitunter die Frage auf, ob man eine bestimmte Sprache auch für sicherheitsrelevante Einsätze vorsehen solle oder könne. Der vorliegende Beitrag stellt allgemein zusammen, welche Wünsche an Programmiersprachen und zugehörige Hilfsmittel zu richten sind, um sie für Sicherheitsanwendungen besonders geeignet zu machen. Er stützt sich besonders auf die Arbeiten des TC7 des European Workshop on Industrial Computer Systems und der WG A3 des IEC SC45a. Forderungen an Sprache und Hilfsmittel betreffen insbesondere: Problemnähe, Förderung irrtumsvermeidender Konstruktionen, Fehlererkennung und -behandlung während des Laufs, weitgehendes Abprüfen möglicher Fehler während des Übersetzens und Bindens sowie Zuverlässigkeit der Hilfsmittel selbst.

Abstract

During the discussion on benefits and drawbacks of computer languages, which also affects PEARL, sometimes the question arises whether or not a particular language should be used for safety related purposes. This contribution compiles the demands on programming languages and related tools that qualify a language for safety applications. The contribution is based primarily on work done at TC7 of the European Workshop on Industrial Computer Systems and at WG A3 of IEC SC 45a. Demands on language and tools comprise in particular: orientation to problem, support of error limiting constructions, failure detection and handling online, extensive test of possible errors during compilation and linking time and reliability of the tools themselves.

1. Das Problem des Programmversagens

Mit dem Fortschritt auf dem Gebiet der Rechnerentwicklung im engeren Sinn sind auch Fortschritte auf dem Gebiet der Rechnereinsätze verbunden. Unter anderem dringen Rechner auch in sicherheitsrelevante Gebiete immer weiter vor, z.B. bei

Kernkraftwerken	in Begrenzungs- und Schutzsysteme
Eisenbahnanlagen	in Stellwerke und in die Geschwindigkeitssteuerung von Zügen
Chemische Fabriken	in die Regelung der Prozesse
Flugzeugen	in die Flugregelung
Medizinischen Anwendungen	in Infusionspumpen
PKWs	in Bremssysteme
der Haustechnik	in Aufzugsteuerungen und Brennersteuerungen.

Selbstverständlich werden hierbei beträchtliche Anforderungen an die Sicherheit und die Verfügbarkeit der Rechner und ihrer Programme gestellt. Die zunächst intuitiv erhobene Forderung nach vollkommener Freiheit von Versagen läßt man in der Regel nach kurzem Besinnen fallen; denn sie ist bei Systemen von nur einiger Komplexität praktisch nicht zu erfüllen.

So bleiben zumeist folgende Forderungen an die Programmzuverlässigkeit und ihren Nachweis bestehen:

a) Die Software darf nicht das unzuverlässigste Glied in der Kette der zum ordnungsgemäßen Funktionieren benötigten Einrichtungen und Handlungen sein.

b) Die durch Programmversagen verursachten Schäden müssen (weit) unter dem Wert bleiben, der zu erwarten gewesen wäre, wenn man

auf einen Rechnereinsatz verzichtet hätte.

c) Die zum Erreichen und zum Nachweis der geforderten Zuverlässigkeit entstehenden Kosten müssen (weit) unter dem aus dem Rechnereinsatz zu erwartenden Gewinn bleiben.

Der Forderung a) nachzukommen, erfordert eine Untersuchung der sonst mit dem Technischen Prozeß befaßten Geräte und Handlungen. Soweit Geräte betroffen sind, kann man sich auf eigene oder fremde, etwa in der Literatur niedergelegte, Betriebserfahrungen stützen. Das gleiche gilt auch bezüglich menschlicher Handlungen. In letzter Zeit in den USA durchgeführte Untersuchungen über menschliches Fehlverhalten [1] lassen vermuten, daß die menschliche Versagenswahrscheinlichkeit nur selten unter 10^{-4} pro Fall liegt. In vielen Fällen wird daher zu fordern sein, daß die Versagenswahrscheinlichkeit pro Anforderung an ein Programm diesen Wert nicht überschreiten darf.

Forderung b) verlangt eine Risikobetrachtung, basierend auf den Schadenshöhen, die mit Programmversagen verbunden sein können und den Häufigkeiten mit denen Anforderungen an Programme kommen, sowie den anzunehmenden Versagenswahrscheinlichkeiten. Üblicherweise werden, falls auch Menschenleben betroffen sein können, zwei Rechnungen aufgemacht, eine für Sachschäden und eine für Personenschäden.

In Anlehnung an [2] definieren wir das Risiko als:

$$R = \Sigma X_i \, h_i \, p_i \qquad (1)$$

Die Summe ist über alle Ereignisarten i zu nehmen, die während im Laufe des Programm-Lebens auftreten können, X steht für die Schadenshöhe, h für die Häufigkeit des Eintritts schadensauslösender Ereignisse und p ist die Wahrscheinlichkeit, daß ein solches Ereignis von der Software nicht abgefangen wird. Die einzelnen p_i müssen für die verschiedenen Programmfunktionen also unter bestimmten Grenzen bleiben. Von einer dieser Grenzen war bei der Betrachtung der menschlichen Versagenswahrscheinlichkeit schon die Rede. Im übrigen bestätigt (1), was man intuitiv weiß, nämlich daß es sich besonders lohnt, häufig angesprochene Funktionen, die hohe Versagenskosten nach sich ziehen können, sehr zuverlässig zu machen. Große Anstrengungen im Hinblick auf Funktionen mit kleinen h und X dagegen lohnen sich weniger.

Die Forderung c) nun gibt uns auf, den Aufwand für die Klein-Haltung der p_i oder für den Nachweis ihres Klein-Seins in wirtschaftlichen Grenzen zu halten. Hierzu dienen die Betrachtungen dieses Beitrags.

Obgleich wir von sicherheitsbezogenen Rechnereinsätzen ausgehen, gilt ein Großteil des Weiteren auch für "normale" industrielle Anwendungen von Programmen. Denn finanzielle Verluste, Sachschäden im weiteren Sinn, treten auch dann auf, wenn ein Programm in Zusammenarbeit mit irgend einem technischen Prozeß nicht ordnungsgemäß funktioniert.

2. Das Problem der Programmierfehler

Es liegt nahe, das Programmier-Fehler-Machen als eine allgemein menschliche Eigenschaft zu betrachten und zu versuchen, auf die Zahl der zu erwartenden Programmierfehler aus allgemeinen Eigenschaften zu schließen. Ein solcher Versuch ist von Halstead in [3] unternommen worden. Das Ergebnis seiner umfangreicheren Betrachtung lautet für $\hat{B}$, die Anzahl der zu erwartenden Fehler in einem Programm:

$$\hat{B} = \frac{(N_1 + N_2) \, ld \, (\eta_1 + \eta_2)}{3000} \qquad (2)$$

Hierbei bedeuten:

η_1 Anzahl der im Programm verwendeten Operatoren, jeder einmal gezählt
$\{\eta_1\}$ = {=,+,-,/, ..., GOTO A, GOTO B, ..., IF, BEGIN ... END, ...}

N_1 Gesamtzahl aller vorkommender Operatoren, jeder sooft gezählt, wie er auftritt

η_2 Anzahl der im Programm verwendeten Operanden, jeder nur einmal gezählt
$\{\eta_2\}$ = {Variable 1, Variable 2, ... Feld 1, Feld 2, ...}

N_2 Gesamtzahl aller vorkommenden Operanden, jeder sooft gezählt, wie er auftritt.

Die Zahl 3000 errechnet sich aus einer Betrachtung der Eigenschaften unseres Kurzzeitgedächtnisses und dem Niveau der Englischen Sprache.

Einen intuitiven Zugang zur Sinnhaftigkeit von (2)

kann man auf folgende Weise gewinnen: $\eta = \eta_1 + \eta_2$ ist der Umfang des Vokabulars, das beim Programmieren zur Verfügung steht. Während des Programmierens ist aus diesem Vokabular eine Auswahl zu treffen; und zwar ist in jedem einzelnen Programmierschritt auszuwählen, welche Operanden und Operatoren gerade zur Problemlösung benötigt werden. Die Anzahl der binären Auswahlschritte beträgt für jeden Auswahlvorgang ld η. $N = N_1 + N_2$ gibt die Zahl der notwendigen Auswahlvorgänge an. Eine nähere Betrachtung unter Einschluß des Nenners zeigt übrigens, daß die Basis des gewählten Logarithmus das Ergebnis nicht beeinflußt.

Es liegt auf der Hand, daß menschliches Tun, wie Programmieren durch eine so einfache Beziehung wie (2) kaum sehr zutreffend oder gar erschöpfend beschrieben werden kann; dies dürfte auch dann zutreffen, wenn es nur auf einen einzigen Parameter - seine Fehlerhaftigkeit - hin betrachtet wird. Demgemäß kommt auch eine Anwendung der Halstead'schen Theorie in [4] zu durchaus gemischten Ergebnissen: Die Theorie traf für 2 von 3 Versuchen einigermaßen zu, zeitigte aber beim dritten völlig abwegige Resultate.

Dennoch wollen wir auf (2) aufbauen und fragen, was wir zu tun haben, um Programmierfehler zu vermeiden. Die Antwort lautet klar: Man halte N_1 und N_2 klein! Eine Vergrößerung von η_1 oder η_2 schadet dagegen nur wenig. Mit anderen Worten heißt dies: Je besser die zu verwendende Sprache an das zu lösende Problem angepaßt ist, desto weniger Fehler sind zu erwarten. Eine Vergrößerung des Vokabulars der Sprache stört nur geringfügig, dagegen ist eine Verlängerung des Programms recht fehlerträchtig. Sieht man sich (2) im Hinblick auf den Gegensatz von Höherer Sprache und maschinennaher Sprache an, so wird man sofort die Höhere Sprache vorziehen, denn bei der maschinennahen Sprache haben alle 4 in (2) auf der rechten Seite stehenden Größen höhere Werte, als etwa in PEARL, falls man ein Problem von nur einigem Umfang zugrunde legt.

Wie schon angedeutet, dürfte das Fehler-Machen beim Programmieren nicht so einfach zu beschreiben sein, wie in (2). Tatsächlich ist je auch schon des längeren bekannt, daß noch eine Reihe weiterer Parameter entscheidend zu Buche schlagen, etwa

- wie das Programm entwickelt wird,
- ob Zwischentests gemacht werden,
- wie das Programmierteam organisiert ist,
- ob Restriktionen in der Verwendung der Sprachmittel beachtet werden,
- welche Hilfsmittel zur Verfügung stehen,
- usw.

Falls eine dem zu lösenden Problem sehr gut angepaßte Sprache zur Verfügung steht, verlieren natürlich manche der obigen Punkte an Bedeutung, doch dürfen sie im allgemeinen Fall nicht unbeachtet bleiben. Eine Zusammenstellung einschlägiger Aspekte befindet sich in [5] und [6].

3. Wünsche an die Programmiersprache

Nachdem wir im vorhergehenden Kapitel zu dem Ergebnis gekommen sind, daß Höhere Sprachen für Sicherheitsanwendungen und mithin für industrielle Anwendungen von Rechnern ganz allgemein vorzuziehen seien, ist weiter zu fragen, welche zusätzlichen Randbedingungen für eine Prozeßrechner- oder Sicherheitssprache zu beachten wären.

Zunächst haben wir den Gegensatz von Problemnähe und Verbreitung der Sprache zu beachten. Es liegt auf der Hand, daß man eine selten benutzte und darum kaum bekannte Sprache nur zögernd für einen verantwortungsvollen Einsatz heranziehen wird. Außer den im nächsten Kapitel zu diskutierenden Unzulänglichkeiten, die man bei ihrem Übersetzer und sonstigen Hilfsmitteln vermuten wird, spielt ihr Unbekannt-Sein bei Entwickler und Gutachter eine abmahnende Rolle. Andererseits aber muß eine Sprache in weiten Kreisen umso unbekannter sein, je problemnäher sie ist und lassen sich die gewünschten Funktionen nur in problemorientierten Sprachen knapp ausdrücken. Zwischen den beiden Gesichtspunkten wird ein Kompromiß gefunden werden müssen. Angesichts der zunehmenden Verbreitung von Rechnern und den damit verbundenen Kenntnissen, sowie moderner Techniken im Compilerbau, dürfte aber die Zukunft ganz klar bei den problemnahen Lösungen liegen; sie gestatten es, in (2) mit minimalen N_i und η_i auszukommen.

Bezüglich der einzelnen zu fordernden Sprachеigenschaften folgen wir den Darlegungen in [5] und [6].

- Syntax und Semantik der Sprache sollen vollständig und unzweideutig definiert sein.

Dies ist vor allem zur Transportabilität der Programme wichtig und für Nachweise ihrer Eigenschaften auf dem Quelltextniveau. Sonst soll eine

Sicherheitssprache natürlich ganz allgemein irrtumsanfällige Konstruktionen vermeiden helfen und leicht überblickbare Konstruktionen fördern. Im einzelnen:

Modularisierung

- Die Sprache soll eine Modularisierung unterstützen.
- Einsprünge in Module sollen nicht erlaubt sein, Aussprünge aus Modulen sollen nur an deren Ende führen und nicht an beliebige Programmstellen, vor allem nicht an Stellen, die in der Aufschreibung weiter vorne liegen. Ausnahme: Fehlerausgang.
- Module sollen in ihrer Länge auf 50 bis 100 ausführbare Anweisungen beschränkt sein.
- Ihre Parameterzahl soll auf etwa 5 beschränkt sein.

Die Gründe für diese Forderungen liegen in dem Bestreben nach Übersichtlichkeit im zu erstellenden Quellcode. Nach Möglichkeit sollen Module für sich allein vorab verifiziert werden. Ihr Ablauf soll nicht durch Sprünge hinein oder hinaus unübersichtlich gemacht werden können; ihre Länge soll so sein, daß gemäß (2) noch damit gerechnet werden darf, daß sie von vorne herein richtig sind; mit nur 5 Parametern sollen sie arbeiten, weil dies die Zahl von Einzelfakten ist, die unser Kurzzeitgedächtnis noch gut verarbeiten kann [3,7].

Prozeduren

Außer dem soeben Dargestellten gelten noch die Forderungen:

- Funktionen dürfen die Parameter, mit denen sie gerufen werden, nicht verändern.
- Bei Aufrufen soll es keine Mischung von Namens- und Werteparametern geben.
- Jede Prozedur soll überprüfen, ob die aufrufende Stelle zum Aufruf berechtigt war.
- Jede Prozedur soll feststellen, ob die Parameter, mit denen sie aufgerufen wird, zulässig sind.
- In den Aufrufen von Prozeduren sollen keine anderen Prozeduren oder Funktionen als Parameter auftreten dürfen.

Fragen der Zeit und Reihenfolge

- Zugriffe zu gemeinsamen Betriebsmitteln sollen synchronisiert werden.
- Während der Ausführung einer Anweisung sollen ihre Parameter generell nicht von außen veränderbar sein.
- Interrupts sollen während des Laufs vorzugebender sicherheitsrelevanter Programmteile verboten werden können.
- Die Zeit der höchsten Interruptverbotsdauer soll vorgebbar sein.
- Während des Programmablaufs soll die verbrauchte Rechenzeit leicht zu überwachen sein.

Diese Forderungen sollen es erleichtern, so zu programmieren, daß definierte Zugriffsverhältnisse zu gemeinsamen Resourcen vorliegen und daß der Zeitablauf eines Programmes stets "bewußt" bleibt. Beides erleichtert die Verifikation.

Sprünge, Schleifen und Verzweigungen

- Unbedingte Sprünge nach rückwärts sollen nicht zulässig sein.
- Es soll keine LABEL VARIABLEN geben; bei Alternativen (CASE) soll die Liste der Sprungziele von Anfang an vollständig sein.
- Schleifen sollen stets eine bekannte höchstmögliche Durchlaufzahl haben.

Im übrigen gilt das zu Sprüngen in und aus Modulen Ausgeführte.

Daten

- Es sollen keine impliziten Typkonversionen stattfinden.
- Art, Bereich und Genauigkeit jeder Veränderlichen soll konstant sein.
- Veränderliche und Felder sollen ausdrücklich vereinbart werden müssen, einschließlich ihrer Typen.
- Variablennamen sollten beliebig lang sein dürfen
- Namen sollen unterscheiden helfen, ob der Parameter
  - einen Eingangs-, Ausgangs- oder transienten Wert repräsentiert,
  - welcher Art er ist, etwa Feld, Veränderliche, Konstante
  - wo er gilt
  - ob er abgeleitet ist, eine Länge, ein Zähler oder dergleichen,
  - welche Bedeutung er ganz allgemein hat.
- Veränderliche Systemparameter sollen hervorgehoben werden.
- Zwischen lokalen und globalen Veränderlichen ist

zu unterscheiden.
- Für einen Datentyp soll es nur eine Adressierungstechnik geben.
- Feldlängen sollen nach Möglichkeit Konstante sein.
- Bei Aufrufen von Feldkomponenten sollen stets alle Felddimensionen vorkommen.
- Während des Programmlaufs soll automatisch überprüft werden, ob die Feldgrenzen eingehalten werden.
- Es soll keine Art von EQUIVALENCE geben.
- Es soll keine einfachen COMMON's geben.
- Konstante und Veränderliche sollen voneinander getrennt abgespeichert werden.
- Nicht berechtigtes Beschreiben soll automatisch registriert und verhindert werden.

Diese Forderungen erklären sich fast alle selbst aus den übergeordneten Verlangen nach leicht verständlichen und nachvollziehbaren Programmen. Das getrennte Abspeichern von Variablen und Konstanten soll eine Überprüfung der Konstanten auf Unveränderheit erleichtern.

Aufschreibung

- Leichte Lesbarkeit des Programms soll den Vorzug vor leichter Aufschreibbarkeit haben.
- Eine Art der Aufschreibung soll stets nur eine einzige Bedeutung haben und umgekehrt.
- Es soll eine standardisierte Folge oder feste Plätze im Programm geben für
  - nichtausführbaren Code/ausführbaren Code
  - Deklarationen
  - Initialisierungen
  - Formate
  - Kommentare

Fehlerbehandlung während des Programmlaufs

- Die Sprache sollte eine Fehlerbehandlung während des Programmlaufs vorsehen. Auf Fehlerbehandlungen sollten automatisch führen:
  - Überschreitung von Feldgrenzen
  - Überschreitung von Wertebereichen
  - Zugriff zu nicht vorbesetzten Veränderlichen
  - Abschneiden signifikanter Stellen von Zahlenwerten
  - Übergabe von Parametern falschen Typs.
- Auch die übrigen im früheren Text verlangten Prüfungen sollen bei negativem Ausgang auf Fehlerbehandlungen führen.
- Die Sprache sollte "Assertions" vorsehen und zulassen; auch das Nicht-Erfüllen einer Assertion soll auf eine Fehlerhandlung führen.
- Die Fehlerbehandlungen selbst sollen vom Anwenderprogramm durchgeführt werden.

Die aufgestellten Forderungen entspringen im wesentlichen dem Wunsch nach Einbringung von Redundanz in das Programm in Form von Plausibilitätsprüfungen während des Laufs. Obgleich die unter dem ersten Spiegelstrich aufgelisteten Punkte im allgemeinen keine sinnvolle Programmfortsetzung mehr zulassen, sollte doch dem Anwender bei Entscheidung über das weitere Verfahren überlassen bleiben, da nur er über eine schrittweise Verringerung der Programmfunktionen entscheiden kann.

Es ist klar, daß die obigen Forderungen nicht für alle Prozeßrechneranwendungen sinnvoll sind. Teilweise sind sie auch nur mit großen Aufwand in einer Sprache zu verwirklichen, teilweise wird man sich damit zufrieden geben können, sie in Programmierrichtlinien niederzulegen.

4. Wünsche an Hilfsmittel

In ähnlicher Weise, wie man vom Standpunkt der Programmverifikation aus Wünsche an die zu verwendende Sprache äußern kann, lassen sich auch Wünsche an die Hilfsmittel, die mit dieser Sprache zusammen angeboten werden, formulieren. Nachdem in dieser Nummer noch ein Beitrag über Testhilfsmittel folgt, können wir uns hier beschränken auf die Betrachtung von

Übersetzern
Bindern
Ladern
Bibliotheksprogrammen
Laufzeitsystemen und
Betriebssystemen.

Die allgemein zu erhebenden Forderungen lauten selbstverständlich

- Die Hilfsmittel sollen in ihren Funktionen gut und vollständig beschrieben sein.
- Ihre Zuverlässigkeit soll anhand einer ausreichenden Betriebserfahrung nachgewiesen sein.
- Die o.g. Forderungen an die Sprache oder den abzusetzenden Code sind bestmöglich zu unterstützen.

Forderungen, die sich insbesondere an den zu ver-

wendenden Übersetzer richten, lauten:

- Während des Übersetzungsvorgangs oder danach sollen auf Wunsch Zwischenergebnisse ausgegeben werden, von denen Einzelheiten über den erzeugten Maschinencode entnommen werden können. Diese Zwischenergebnisse sollen gut lesbar sein.

Es mag sinnvoll sein, in manchen Einzelfällen das Compilationsergebnis selbst der Verifikation zugrunde zu legen. Weiter gilt:

- Während der Übersetzungszeit sollen möglichst viele Prüfungen des Quelltexts durchgeführt werden.
- Von den insgesamt vorzunehmenden Prüfungen soll ein möglichst großer Teil während des Übersetzungsvorgangs durchgeführt werden und nur das dort unbedingt Notwendige zur Laufzeit des Programms. Dies betrifft z.B. Parameter-Typ-Prüfungen.
- Falls Prüfungen während der Übersetzung und des Bindevorgangs auf Fehler stoßen, sollen diese Fehler nur gemeldet werden; Korrekturversuche sollen unterbleiben.
- Falls unklar bleibt, ob irgendwelche Regeln verletzt wurden, soll eine Warnung gegeben werden.
- Während der Übersetzungszeit soll insbesondere überprüft werden,
  - der Wertebereich Veränderlicher
  - ihr Typ
  - ihre Genauigkeit
  - die Zulässigkeit von Zuweisungen und
  - alle syntaktischen Konstrukte.

Über einen Binder, der der aufgeführten Forderung nach weitestgehender Prüfung der zusammenzusetzenden Module während des Bindevorgangs nahekommt, wird an anderer Stelle dieses Hefts berichtet.

Der zweite der mit Spiegelstrich versehenen Punkte wirft die Frage auf, wann ein Hilfsmittel, etwa ein Übersetzer, als hinreichend ausgetestet angenommen werden dürfe. Bild 1 zeigt, wie die Zusammhänge zwischen der Zahl der Einzelfunktionen eines solchen Programms, der Zahl der mit ihnen durchgeführten Probeläufe und der Wahrscheinlichkeit eines vollständigen Tests liegen. Hat ein Übersetzer z.B. 5000 Eigenschaften und werden 100000 mal Eigenschaften von ihm angesprochen, so beträgt die Wahrscheinlichkeit, daß eine seiner 5000 Eigenschaften überhaupt nie angesprochen wurde 10^{-5} . Hierbei ist angenommen, daß bei jeder Benutzung alle Eigenschaften mit der gleichen Wahrscheinlichkeit zum Zuge kommen. Da diese

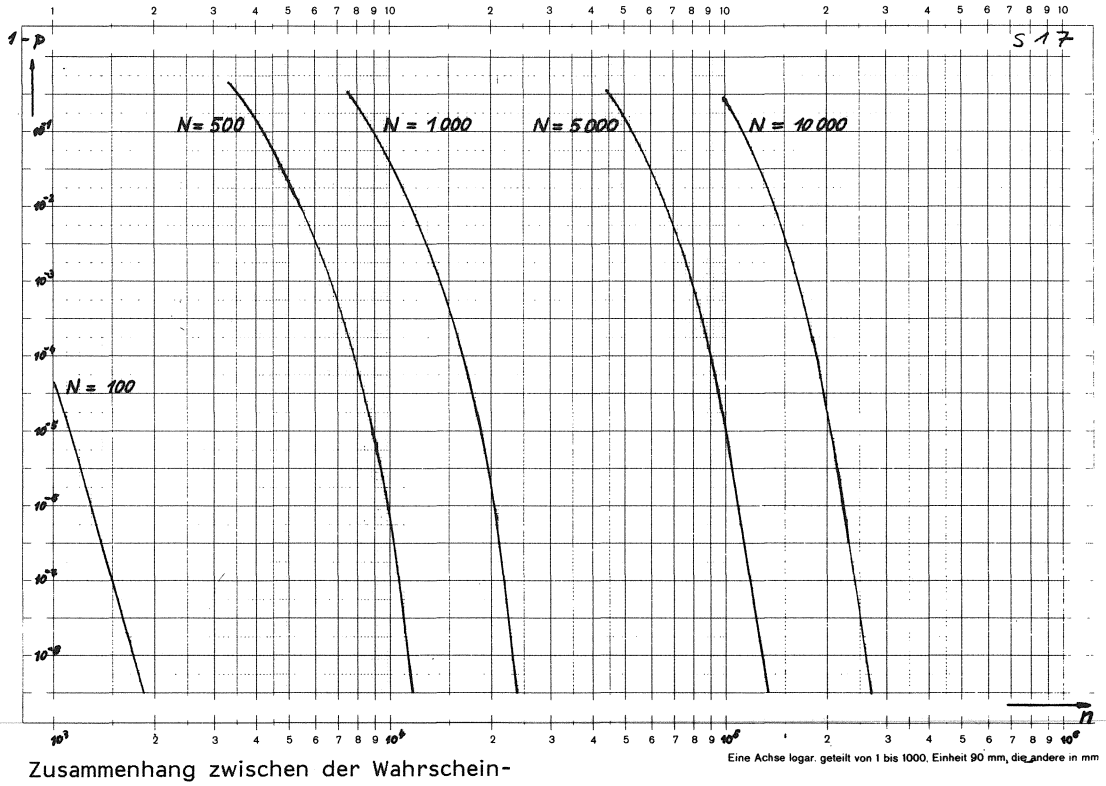

Bild 1: Zusammenhang zwischen der Wahrscheinlichkeit p, daß jede von N Einzelfunktionen eines Programms mindestens ein Mal getestet wird und der Anzahl n der Testläufe; Annahme: Jede der N Einzelfunktionen wird mit gleicher Wahrscheinlichkeit angesprochen. Aus [9].

Vorausetzung bei tatsächlichen Einsätzen nicht erfüllt werden kann, wäre als die Anzahl des Ansprechens von Eigenschaften die Zahl zugrundezulegen, die auch für die am seltensten benutzten Eigenschaften angenommen werden darf. Bei einem Übersetzer könnte man z.B. davon ausgehen, daß seine selten benutzten Teile im Mittel allenfalls ein Mal pro Lauf zum Zuge kommen und daß von diesen Teilen nur eines pro Lauf benutzt wird. Im obigen Beispiel würde das heißen, daß, um eine Unwahrscheinlichkeit des Getestet-Seins von nur $1 - 10^{-5}$ nachzuweisen, 100000 Übersetzungen durchgeführt werden müssen. Ist der Übersetzer etwa an 100 Anlagen installiert, wird er pro Arbeitstag im Mittel je 2 mal unabhängig von anderen Benutzungen zum Laufen gebracht, so kann nach 500 Arbeitstagen davon ausgegangen werden, daß er im o.g. Sinn getestet worden ist.

Für sicherheitsrelevante Einsätze folgt aus dieser Betrachtung natürlich, daß man bei neu auf dem Markt befindlichen Produkten vor allem von deren allgemein benützten Eigenschaften Gebrauch machen soll und nicht von ihren Spezialitäten.

5. Zusammenfassung

Es sind eine Reihe von Forderungen an Sprache und Übersetzer dargestellt worden, die beide für Sicherheitsanwendungen besonders geeignet erscheinen lassen. Die Sprache soll vor allem Programmierfehler-begrenzende Konstruktionen anbieten und Redundanz zum Auffinden von Programmversagen während des Laufs zur Verfügung stellen. Der Übersetzer und die sonstigen Hilfsmittel zur Programmerstellung sollen während ihres Einsatzes Hinweise auf mögliche Programmierfehler geben und in sich gut ausgetestet sein.

6. Schrifttum

[1] S w a i n, A.D.:
Handout Material for the Seminar on Effects of Human Performance on Nuclear Power Plant Operations
Garching, October 1980

[2] Gesellschaft für Reaktorsicherheit: Deutsche Risikostudie Kernkraftwerke, Hauptband (1980), Verlag TÜV Rheinland

[3] H a l s t e a d, M.H.: Elements of Software Science, North Holland, New York (1978)

[4] M e a l s, R.R and G u s t a f s o n, D.A.: An Experiment in the Implementation and Application of Halstead's and Mc Cabe's Measures of Complexity, Software Engineering Standards Application Workshop (August 1981), San Francisco, IEEE Cat. No. 81CH1633-7

[5] European Workshop on Industrial Computer Systems, TC7 Systems Reliability, Safety and Security: Development of Safety Related Software (October 1981), position paper

[6] International Electrotechnical Commission SC 45 A/WG-A3 (WG Experts) 5, Draft: Software for Computers in the Safety System of Nuclear Power Stations (August 1981)

[7] B e r n h o l t z, A.: Verbal presentation at IFIP Congress 1971

[8] B a s t l, W.; B e r a h a, D.; E h r e n b e r g e r, W.; F e l k e l, L.; H ö l d, A.: Developments in the Field of Man/Machine Communication, IAEA-CN-39/37, "Current Nuclear Power Plant Safety Issues" Vol. III

[9] E h r e n b e r g e r, W.; P l ö g e r t, K.: Einsatz statistischer Methoden zur Gewinnung von Zuverlässigkeitskenngrößen von Prozeßrechner-Programmen, Bericht KfK-PDV 151 (Juni 1978).

Anschrift des Verfassers:

Gesellschaft für Reaktorsicherheit
Forschungsgelände

8046 Garching