KfK 3538
April 1983

Zur Testfallgenerierung in der Entwurfsphase

L. Gmeiner
Institut für Datenverarbeitung in der Technik

Kernforschungszentrum Karlsruhe

KERNFORSCHUNGSZENTRUM KARLSRUHE

Institut für Datenverarbeitung in der Technik

KfK 3538

Zur Testfallgenerierung in der Entwurfsphase

Lothar Gmeiner

Als Dissertation genehmigt von der Fakultät für Informatik
der Universität Karlsruhe(TH)

Kernforschungszentrum Karlsruhe GmbH, Karlsruhe

Kernforschungszentrum Karlsruhe GmbH
ISSN 0303-4003

Zusammenfassung:

Das Testen ist in der Praxis die gebräuchlichste und erfolgreichste Methode, um die Korrektheit und Zuverlässigkeit eines Programms nachzuweisen. Für die Güte des Testens ist die systematische Auswahl signifikanter und fehlersensitiver Testfälle entscheidend. Bisherige Verfahren und Werkzeuge zur rechnergestützten Testfallgenerierung basieren zumeist auf dem Programmcode. Dagegen existieren für die frühen Phasen der Softwareentwicklung - insbesondere im Entwurf - keine befriedigenden rechnergestützten Lösungen.

In dieser Arbeit wird das integrierte System TESTPLAN zur entwurfsbegleitenden Testplanung und Testdurchführung vorgestellt. Die Grundidee dieses Systems liegt darin, möglichst große Teile des Testens in der Softwareentwurfsphase zu automatisieren. Das System TESTPLAN besteht aus drei aufeinander abgestimmten Teilen:

- einem Verfahren zur Erstellung des Testplans parallel zur Implementierung,
- einer Sprache zur Formulierung des Softwareentwurfs und der Testpläne,
- einem Werkzeug zum Umsetzen der sprachlichen Formulierung in eine interne Darstellung und zur Generierung der Testfälle aus der Entwurfsbeschreibung sowie deren weitere Verarbeitung.

Die wichtigsten Ergebnisse sind:

- die vollständige, formale Definition einer Entwurfs- und Testplansprache auf der Grundlage Abstrakter Datentypen;
- ein neuartiger Algorithmus zur automatischen Testfallgenerierung für eine bestimmte Klasse von Entwurfsbeschreibungen;
- die automatische Ableitung von lauffähigen Testrahmen aus der Entwurfsbeschreibung.

Bei der Realisierung des Systems TESTPLAN wurde ein auf attributierten Grammatiken basierendes, übersetzererzeugendes System eingesetzt. Die Implementierung erfolgte in PASCAL auf einer IBM 3033.

TESTCASE GENERATION FROM DESIGN SPECIFICATIONS

Abstract:

Testing is a common and effective method to assure the correctness and/or the reliability of a program. To a large degree the quality of testing is depending on the systematic selection of significant and failure-sensitive testcases.

Present methods and tools for computer aided testcase generation are based on the program code, whereas for the earlier phases of software development - particularly with regards to design specifications - no satisfying solution exists. In this thesis the integrated system TESTPLAN is introduced. This system is based on the assumption that automation of test efforts - especially in the design specifications - improves test quality.

TESTPLAN consists of three parts:

- a method for the construction of test plans parallel to the implementation;
- a language for the description of design specifications and test plans;
- a tool for the compilation of design specifications, the generation of testcases and the construction as well as the maintenance of the system's data base.

Important results are:

- a complete formal definition of a language for design specifications and testplans based on abstract data types;
- a novel type of algorithm for automatic testcase generation for a specific class of design specifications;
- a method to derive automatic test drivers from design specifications.

The system TESTPLAN is implemented by means of a compiler writing system, which is based on attributed grammars. The implementation is performed on an IBM 3033 in PASCAL.

Inhalt:

1. Einleitung und Problemstellung

In den letzten Jahren hat sich die Softwareentwicklung, bedingt durch die Komplexität der Problemstellungen, immer mehr von der "Kunst" eines individuellen Programmierers zum "Engineering" hin entwickelt. Man hatte dabei den Hardwarebereich vor Augen, wo beispielsweise in den Ingenieurwissenschaften die Normung von Komponenten und (Produktions-) Verfahren bereits weit fortgeschritten ist. In diesem Zusammenhang wurde der Begriff des "Software-Engineering" geprägt.

Software-Engineering /Boeh 76/ ist die praktische Anwendung von wissenschaftlichen Erkenntnissen und ingenieurmäßigen Vorgehensweisen im Bereich der Systemanalyse, der Spezifikation, des Entwurfs, der Erstellung sowie des Tests und der Wartung von Computerprogrammen.

Die Umsetzung des Software-Engineering-Gedankens in die Praxis erfordert durch Werkzeuge unterstützte Methoden und Verfahren.
Solche Methoden haben sich für den Bereich der Codierung bereits etabliert, wogegen für die frühen Phasen der Softwareentwicklung noch keine befriedigenden Lösungen vorliegen.

Die vorliegende Arbeit beschäftigt sich mit Testwerkzeugen, die auf dem Softwareentwurf basieren. Es wird ein integriertes System zur entwicklungsbegleitenden Testplanung und Testdurchführung vorgestellt, das mit dem Ziel entwickelt wurde, die frühen Softwareentwicklungsphasen zu formalisieren und damit einer Verarbeitung durch Werkzeuge zugänglich zu machen.

Schwerpunkte dieser Arbeit sind

- die vollständige, formale Definition einer Entwurfs- und Testplansprache auf der Grundlage abstrakter Datentypen;
- die automatische Generierung von Testfällen aus der Entwurfsbeschreibung;
- die Entwicklung eines Verfahrens zur Ergänzung des Entwurfs um einen Testplan parallel zur eigentlichen Programmerstellung (Damit soll die Unabhängigkeit von Test- und Implementierungsperson erzwungen werden);
- die automatische Ableitung von lauffähigen Testrahmen aus der ergänzten Entwurfsbeschreibung;
- die Verwendung eines Übersetzererzeugenden Systems bei der Realisierung des Werkzeugs für die Entwurfs- und Testplansprache.

Letztlich hat diese Arbeit zum Ziel, einen Beitrag zur Systematisierung des Testens und der Softwareentwicklung, und damit zum ingenieurmäßigen Vorgehen, zu leisten. Durch die formale Anbindung der Testplanung an den Entwurf wird die gegenwärtig noch existierende Kluft zwischen automatisierten Hilfsmitteln der Entwurfs- und Testphase verkleinert.

Die wichtigsten in dieser Arbeit verwendeten Begriffe werden in Kapitel 2 erklärt.

Der Stand der Technik wird in Kapitel 3 vorgestellt.

In Kapitel 4 wird ein System zur entwurfsbegleitenden Testplanung und Testdurchführung entwickelt, das aus drei Teilen besteht:

- Einem Verfahren zur Erstellung des Testplans parallel zur Implementierung.
- Einer Sprache zur Formulierung des Softwareentwurfs und der Testpläne.
- Einem Werkzeug zum Umsetzen der sprachlichen Formulierung in eine interne Darstellung, sowie deren weitere Verarbeitung.

In Kapitel 5 wird die Implementierung beschrieben.

In Kapitel 6 wird ein Beispiel zusammen mit Erfahrungswerten vorgestellt.

Kapitel 7 gibt eine Zusammenfassung und einen Ausblick.

Die Anregung für das hier behandelte Thema zur 'Entwurfsbegleitenden Testplanung' stammt aus dem Projekt 'Schutzrechnersysteme', das im Institut für Datenverarbeitung in der Technik (IDT) des Kernforschungszentrums Karlsruhe bearbeitet wird. Ein wichtiger Punkt innerhalb dieses Projekts ist die Genehmigungspflicht für die Software. Die Genehmigungsfähigkeit setzt sehr strikte Methoden und Verfahren der Softwareentwicklung voraus. Schwerpunkte der angewandten Softwareentwicklungsmethodik bilden dabei systematische Testverfahren und der Einsatz rechnergestützter Werkzeuge für Entwurf, Test und Dokumentation.

Herrn Prof. Dr. G. Goos gilt mein besonderer Dank für die wertvollen Anregungen und die tatkräftige Betreuung dieser Arbeit.

Herrn Prof. Dr. H. Trauboth danke ich für die wohlwollende Förderung sowie die Übernahme des Korreferats.

Allen meinen Kollegen im IDT danke ich ganz herzlich für die stete Diskussionsbereitschaft. Besonderer Dank gilt hierbei den Herren K. Eckert, Dr. D. Schriefer und U. Voges.

Nicht zuletzt möchte ich Herrn Prof. Dr. U. Kastens sowie den Herren P. Dencker und E. Zimmermann vom Institut für Informatik II der Universität Karlsruhe für die Unterstützung während der Implementierung danken.

2. Begriffe

In diesem Kapitel werden die wichtigsten Begriffe erörtert, die in dieser Arbeit Verwendung finden. Leider existieren bisher noch keine allgemein befriedigenden Definitionen und Normen, ganz zu schweigen von formalen Begriffsdefinitionen, wie sie in der Mathematik üblich sind.

Wir beschränken uns auf verbale Beschreibungen und beginnen mit dem Begriff des Softwareentwicklungszyklus. Dieser ist in Phasen unterteilt, von denen hier primär die beiden ersten Phasen, die Anforderungs- und Entwurfsspezifikation interessieren.

Die Anforderungsspezifikation oder kurz die Spezifikation beschreibt die Zielsetzung eines Systems aus der Sicht des Anwenders oder Auftraggebers. Dementsprechend liegt eine Spezifikation meist in der Fachsprache des jeweiligen Anforderers oder in natürlicher Sprache vor.

Die Entwurfsspezifikation oder kurz der Entwurf geht von der in der Spezifikation formulierten Zielsetzung aus und umfasst den gesamten Problemlösungsprozeß. Das Ergebnis des Entwurfs ist eine abstrakte Beschreibung der Problemlösung. Innerhalb dieser abstrakten Beschreibungsform sind verschiedene Detaillierungsgrade zu unterscheiden, die in der Literatur oft als Grob- und Feinentwurf bezeichnet werden. Die endgültige Problemlösung ergibt sich durch eine schrittweise Verfeinerung des Entwurfs. Idealerweise würde man sich als Basis des Entwurfs eine Spezifikation wünschen, die vollständig und eindeutig die Problemstellung beschreibt. Leider existiert dieser Idealzustand nicht, sodaß in der Praxis häufig eine Nachbesserung der Spezifikation durch Entwurfserkenntnisse notwendig wird.

Mit dem zu erstellenden Softwareprodukt sind diverse Programmqualitäten assoziiert, die in jeder Phase des Softwareentwicklungszyklus erfüllt sein sollen. Wir wollen hier nur die Softwarezuverlässigkeit als eine der wichtigsten Qualitäten hervorheben und verweisen für die übrigen Begriffe auf Erörterungen in /Goos 80/ und /Boeh 78/.

Ein Fehler ist die Ursache der Abweichung eines Softwarezwischen- oder -endprodukts von einem gegebenen Referenzobjekt (beliebiges Entwicklungsdokument). Die beobachtete Abweichung ist das Fehlersymptom.

Die Softwarezuverlässigkeit ist die Wahrscheinlichkeit, daß ein Softwarefehler, der in einer gegebenen Umgebung zum Abweichen eines geforderten Ergebnisses um mehr als vorgeschriebene Toleranzwerte führt, während einer bestimmten Zeitdauer nicht auftritt.

Aus dieser Definition wird in /Rama 82/ eine operationale Definition der Softwarezuverlässigkeit abgeleitet, die auf dem Verhältnis von durchgeführten Testläufen (n) zu mißglückten Testläufen (nf) basiert. Als mißglückt gilt ein Testlauf dann, wenn die in der Betriebsumgebung des Testobjekts ermittelten Ergebnisse eines Testlaufs nicht korrekt sind.

Die Zuverlässigkeit (R) eines Programms ist dann näherungsweise durch die Wahrscheinlichkeit

$$R = 1 - (nf / n)$$

beschreibbar.

Um die Einhaltung der geforderten Programmqualitäten zu gewährleisten, sind verschiedene Aktivitäten vorgesehen. Zu diesen zählen die Verifikation, Validation und das Testen.

Die Verifikation ist eine formale Methode, um die Korrektheit eines Programms als Ganzes gegenüber der Spezifikation nachzuweisen.

Die Validation ist ein iterativer Prozess, der mit den einzelnen Phasen des Softwareentwicklungszyklus assoziiert ist. Das Endprodukt jeder Phase wird gegenüber dem zu Beginn dieser Phase vorliegenden Dokument auf Einhaltung der Programmqualitäten (Korrektheit, Effizienz, ...) hin überprüft. Im Gegensatz zur Verifikation kann die Validation auch informal erfolgen (z. B. durch Reviews).

Das Testen besteht darin, ein Programm mit der Absicht, Fehler zu finden, auszuführen. Dabei wird nach Differenzen zwischen erwarteten und tatsächlichen Resultaten gesucht. Wir werden das Testen in dieser Arbeit im Sinne eines funktionalen Modultests verwenden, wobei wir unter einem Modul einen abstrakten Datentyp verstehen. Die Stärken des Testens liegen darin, daß die Validation eines Programmprodukts auch gegenüber einer informalen Spezifikation ermöglicht wird und daß man dieses Programmprodukt zusammen mit seiner späteren Einsatzumgebung überprüfen kann.

Das Testen ist nicht mit dem Fehlersuchen (debugging) zu verwechseln, obwohl die beiden Begriffe fälschlicherweise oft miteinander vertauscht werden. Während das Testen hauptsächlich das Finden von Fehlern zum Ziel hat, kann das Fehlersuchen folgendermaßen charakterisiert werden: Fehlersuchen bedeutet, für einen bereits bekannten Fehler die genaue Ursache festzustellen und dann den Fehler zu korrigieren.

Alle mit dem Testen zusammenhängenden Aktivitäten sind organisatorisch in einem Testplan zusammengefaßt. Dieses Dokument beschreibt die Teststrategie, die Testdurchführung, vorhandene Betriebsmittel und Termine, sowie Testobjekte und verantwortliche Personen. Das Kernstück eines Testplans bilden Listen von Testfällen und Testdaten.

Oftmals werden in der Literatur die Begriffe Testfall und Testdaten wechselweise verwendet. Wir verstehen darunter zwei unterschiedliche Abstraktionsstufen, die man in Anlehnung an die Mathematik als Äquivalenzklassen (Testfälle) und deren Repräsentanten (Testdaten) betrachten kann. In einer Äquivalenzklasse sind funktional gleichwertige Eingangsdatenkombinationen zusammengefaßt, wogegen ein Repräsentant einer konkreten Eingangsdatenkombination entspricht.

Das heißt, man wählt Testfälle auf der Basis der Anforderungs-spezifikation, der Entwurfsspezifikation oder des Programmcodes nach funktionalen oder strukturellen Kriterien aus. Jeder Testfall bildet eine logische Einheit, die durch unterschiedliche Testdaten aktiviert werden kann. Nehmen wir beispielsweise an, aus einer Spezifikation wäre zu entnehmen, daß der Monat Juni in einem Programm speziell behandelt werden muß. In diesem Fall bildet der Monat Juni einen Testfall, und die Daten '1.6', ... , '30.6' stellen dafür jeweils Repräsentanten, also Testdaten, dar.

Ein Testlauf bezeichnet die Ausführung eines Programms mit Testdaten.

3. Stand der Technik

Es gibt eine Reihe von Programmeigenschaften, die als Charakteristika für 'gute' Programme gelten. Zu diesen Eigenschaften zählen die Forderungen nach Korrektheit, Zuverlässigkeit, etc. Für eine vollständige Aufzählung und Erläuterung dieser Eigenschaften wird auf /Goos 80/ verwiesen. Setzt man innerhalb dieser Programmqualitäten Prioritäten, so stellt man leicht fest, daß die Zuverlässigkeit eine dominierende Rolle spielt. Insbesondere in Bereichen, wo der Rechnereinsatz die Sicherheit von Menschen und Anlagen direkt tangiert, wie beispielsweise im Reaktorschutz oder in der Luftfahrt, muß besonderes Gewicht auf die Programmeigenschaft 'Zuverlässigkeit' gelegt werden.

In diesem Kapitel werden die Maßnahmen aufgezählt, die zu einer Steigerung bzw. Gewinnung von Zuverlässigkeit beitragen (vgl. Abb. 3.1).

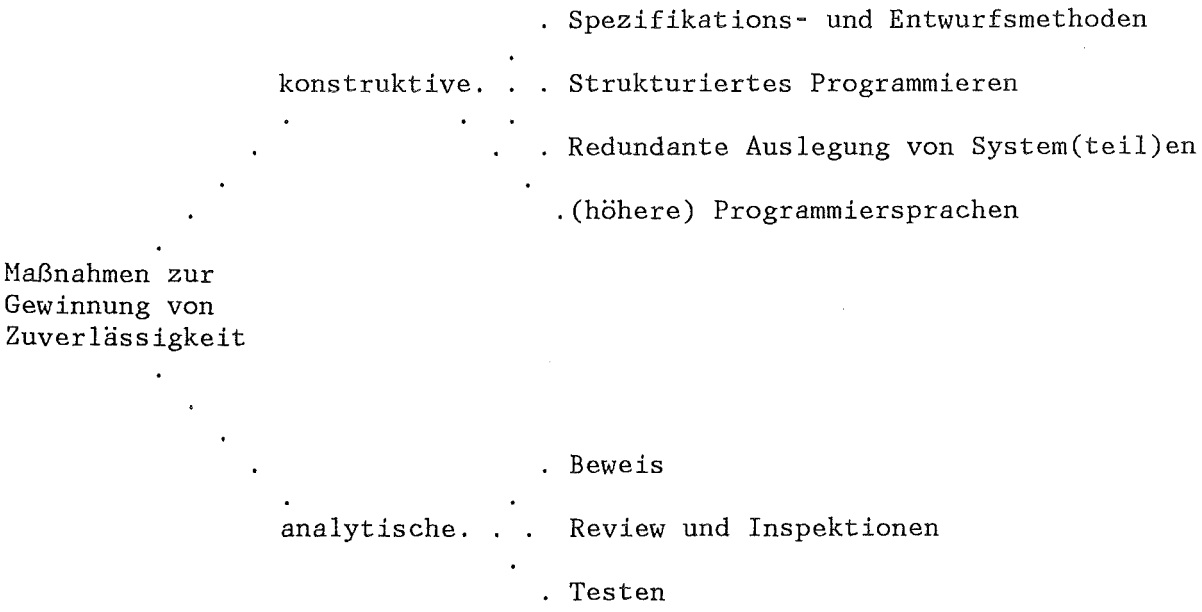


Abb. 3.1 : Maßnahmenkatalog für zuverlässige Software

Wir werden die konstruktiven Maßnahmen im Kapitel 3.1 behandeln, um dann im Kapitel 3.2 die analytischen Maßnahmen näher zu betrachten. Innerhalb der analytischen Maßnahmen werden wir uns insbesondere auf das Testen (Kap. 3.2.3) konzentrieren.

Bei /Rama 74/ und /Endr 76/ werden, ebenso wie in Abb. 3.1, die konstruktiven und die analytischen Maßnahmen als Oberklassen unterschieden. Diese beiden Gruppen ergänzen sich insofern, als bereits während des Konstruktionsprozesses Vorkehrungen getroffen werden, auf denen dann die analytischen Maßnahmen aufbauen. In jüngster Zeit haben beide Maßnahmengruppen unter dem Schlagwort 'Software-Qualitätssicherung' /Snee 82/ ein immer breiteres Fachinteresse gefunden.

Keine der in Abb. 3.1 aufgeführten Maßnahmen für sich genommen ist ausreichend, um Zuverlässigkeit zu erreichen. Daher wird in der Literatur zumeist ein Netz sich ergänzender und teilweise überdeckender Maßnahmen vorgeschlagen (/Geig 79/, /Trau 79/). Je nach Problemstellung und Umgebung können sich die einzusetzenden Methoden und Verfahren unterschiedlich zusammensetzen.

3.1 Konstruktive Maßnahmen

Diese Gruppe von Maßnahmen deckt den gesamten Softwareentwicklungszyklus ab. Insbesondere in jüngster Zeit haben die frühen Phasen der Softwareentwicklung (z. B. die Anforderungsspezifikation) innerhalb der Informatik besonderes Interesse gefunden, weil man aufgrund von Fehleranalysen /Endr 75/ festgestellt hat, daß gerade in diesen Phasen viele Fehler gemacht wurden, die dann nur schwer und unter großen Kosten später gefunden und behoben werden konnten.

Diese Erkenntnis hat zur Entwicklung eines breiten Spektrums an Spezifikations- und Entwurfsverfahren geführt. Die Situation in diesem Bereich ist geprägt durch eine Vielzahl von Ansätzen, die teils rechnergestützt, teils formulargestützt oder auch vollkommen manuell ablaufen. Bisher hat sich aus dieser Menge von Ansätzen noch kein Verfahren herauskristallisiert, das allgemein akzeptiert wird. Gemeinsam ist allen diesen Verfahren, daß sie zum Ziel haben, die Lücken zwischen einer meist informalen Problemstellung und einer formalen Problembeschreibung zu schließen bzw. zu verkleinern. Ein guter Überblick über existierende Spezifikations- und Entwurfsverfahren ist in dem Artikel von Hesse /Hess 81/ zu finden. In /Homm 80/ werden am Beispiel einer Paketverteilanlage 16 verschiedene Spezifikations- und Entwurfsverfahren vorgestellt. Diese Zusammenstellung erlaubt einen Vergleich auf der Basis einer gemeinsamen Problemstellung. Allerdings ist bei solchen Vergleichen Vorsicht geboten, da die einzelnen Autoren und Verfahren von einem sehr unterschiedlichen Spezifikations- und Entwurfsverständnis ausgehen.

Wie wir bereits bei den Begriffsdefinitionen festgestellt haben, erfolgt in der Entwurfsphase der wesentliche Teil der Problemlösung eventuell bis auf die Ebene einer detaillierten Programmiervorgabe. Leider existiert bisher für die Entwurfsphase keine durchgehende und allgemeingültige Systematik, jedoch haben sich in den letzten Jahren einige weitgehend anerkannte Prinzipien herauskristallisiert. Zu diesen zählen die Zerlegung eines Problems in Abstraktionsebenen /Dijk 68/, die Modularisierung /Parn 72/ und darauf aufbauend die Technik der Datenabstraktion, wie sie beispielsweise im Class-Konzept der Programmiersprache SIMULA /Rohl 73/ zu finden ist. Ein weiteres Prinzip ist die sogenannte strukturierte Programmierung, die wiederum ein Kompendium von Verfahren ist. Das Spektrum der strukturierten Programmierung reicht von einer Entwurfsmethodik über organisatorische Maßnahmen bis zur Codiertechnik und wird in ihren vielfältigen Aspekten beispielsweise in /Rama 74/ ausführlich dargestellt.

Eine konstruktive Maßnahme, die insbesondere in Realzeitsystemen häufig angewendet wird, ist die redundante Auslegung von Systemen oder Systemteilen. Diese Technik fügt - in Erwartung eines Fehlers für eine Komponente oder einen Algorithmus - zusätzlich Daten oder Algorithmen hinzu, die entweder in jedem Fall ausgeführt werden oder aber nur im Fehlerfall aktiviert werden (stand-by). Ein Beispiel, in dem auf Programmiersprachenebene die Redundanz explizit ausgedrückt wird, sind die 'recovery-Blöcke' /Ande 77/. Diese sind nach folgendem Schema aufgebaut :

```
ENSURE      <acceptance test>
     BY     < alternative 1 >
ELSE BY     < alternative 2 >
       :
ELSE BY     < alternative n >
ELSE        < error handling >.
```

Mit Hilfe dieses Sprachelements kann der Programmierer mehrere Programmstücke vorgeben, um ein Problem zu lösen. Wenn dabei eine Alternative nicht einer vorgegebenen Bedingung (acceptance test) genügt, wird angenommen, daß während der Ausführung dieser Alternative ein Fehler auftrat, und es wird zur nächsten Alternative übergegangen, oder die Fehlerbehandlung wird eingeleitet.
Eine Implementierung der Recovery-Block-Technik muß Rekonstruktionsmöglichkeiten für frühere Zustände von fehlerhaften Alternativen vorsehen. Dies ist in der Praxis sowohl zeitlich als auch speicherplatzmässig sehr aufwendig.

Während im Hardwarebereich zum Schutz gegen alterungsbedingte und physikalische Ausfälle eine Redundanz durch Verdoppelung identischer Komponenten (Produktredundanz) erreicht wird, nützt diese Technik im Bereich der Software nur bedingt. Identische Software läßt sich nur dann einsetzen, wenn sie sehr intensiv überprüft wurde. In allen anderen Fällen muß die Diversität als spezielle Form der Redundanz verwendet werden. D. h. man muß sicherstellen, daß der Softwareentwicklungsprozeß für zwei funktional identische Komponenten oder Algorithmen unabhängig und mit unterschiedlichen Hilfsmitteln (z. B. Sprachen, Betriebssysteme, Personen) durchgeführt wird.

Diversitäre Implementierungen /Gmei 80/ bieten den Vorteil, daß Common-Mode-Fehler während der Implementierung weitgehend ausgeschlossen werden können. Der Schwachpunkt dieses Ansatzes liegt neben den höheren Entwicklungskosten in der gemeinsamen Problemspezifikation, von der jede der diversitären Implementierungen auszugehen hat. In der Testphase kann man die diversitäre Implementierung vorteilhaft ausnutzen, indem man ohne Vorausberechnung der erwarteten Testresultate einen Ergebnisvergleich der verschiedenen Implementierungen durchführt. Unterschiedliche Ergebnisse der einzelnen Programme deuten dann entweder auf Implementierungsfehler hin, oder beruhen auf Fehlern bzw. Mißverständnissen der Spezifikation. Fehler aus dieser zweiten Gruppe sind normalerweise nur durch eine detaillierte und zeitaufwendige Spezifikationsanalyse zu finden, deshalb ist die Möglichkeit, Fehler in der Spezifikation über eine diversitäre Implementierung zu finden, besonders hervorzuheben.

Die konstruktive Maßnahme, die zeitlich als letzte während der Softwareentwicklung eingesetzt wird, ist die Verwendung einer geeigneten (höheren) Programmiersprache. Der Einfluß der Implementierungssprache auf die Zuverlässigkeit eines Programms ist unbestritten und hat immer wieder die Entwicklung neuer Sprachen herausgefordert. Moderne Sprachkonzepte, wie etwa in MODULA-2 /Wirt 80/ oder ADA /Ada 81/, schränken einerseits die Freiheiten des Programmierers ein, können aber andererseits eine ganze Reihe von Prüfungen zur Übersetzungszeit machen, die in vielen anderen Sprachen entweder nur zur Laufzeit oder überhaupt nicht durchgeführt werden können.

Der Einsatz konstruktiver Maßnahmen hat neben der Zuverlässigkeit des zu erstellenden Programmprodukts auch auf das Qualitätsmerkmal "Wartbarkeit" einen direkten Einfluß. Die Erfahrung mit existierenden Programmsystemen zeigt, daß immer wieder Programmänderungen aufgrund von Fehlern oder erweiterten Anforderungen notwendig werden. Die Auswirkungen solcher Änderungen lassen sich leichter abschätzen oder kontrollieren, wenn man konstruktive Verfahren zur Verfügung hat, die in derselben Art und Weise wie bei der originären Softwareentwicklung auch bei den Änderungen anwendbar sind. Somit ist festzustellen, daß die Anwendung konstruktiver Verfahren eine Nachvollziehbarkeit der Softwareentwicklung erleichtert oder überhaupt erst ermöglicht.

Wir sind in diesem Abschnitt auf die vier wichtigsten Gruppen von konstruktiven Maßnahmen näher eingegangen. Sowohl die strukturierte Programmierung als auch die (höheren) Programmiersprachen sind inzwischen allgemein bekannte und anerkannte Techniken, die in nahezu allen Softwareprojekten einsetzbar sind.
Die redundante Auslegung von System(teil)en hat sich insbesondere in Realzeitsystemen bewährt. Demgegenüber haben sich die angebotenen Spezifikations- und Entwurfsverfahren noch nicht allgemein durchgesetzt, obwohl deren grundsätzliche Notwendigkeit unbestritten ist.

Zusammenfassend ist zu sagen, daß die Anwendung der hier angesprochenen konstruktiven Maßnahmen zwar keine Garantie für zuverlässige Programme bietet, daß durch ihren Einsatz jedoch der Erstellungsprozeß von Software systematischer wird und außerdem eine Basis geschaffen wird, auf der die analytischen Maßnahmen aufbauen können.

3.2 Analytische Maßnahmen

Jede analytische Maßnahme ist in eine der drei Untergruppen

Programmbeweis, Reviews oder Testverfahren

einzuordnen.

Während die Reviews als management-orientierte Maßnahme anzusehen sind, ist der Programmbeweis eine rein technische Maßnahme. Das Testen ist eine Mischung zwischen mangagement-orientierten und technischen Maßnahmen.
Nachdem der Programmbeweis und die Reviews in den beiden folgenden Kapiteln kurz behandelt werden, legen wir den Schwerpunkt unserer Betrachtungen auf das im Kapitel 3.2.3 beschriebene Gebiet des Testens.

3.2.1 Programmbeweis

Der Begriff 'Programmbeweis' suggeriert die absolute Korrektheit (und damit Fehlerfreiheit) eines bewiesenen Programmes.
In Wirklichkeit verwendet man beim Programmbeweis jedoch einen eingeschränkten Korrektheitsbegriff, der sich zumeist nur auf eine formale Problemspezifikation bezieht und wichtige Dinge wie die Wortlänge der Rechenanlage etc. nicht berücksichtigt.

Diese kurze Vorbemerkung soll verhindern, daß der Leser den Eindruck gewinnt, der Programmbeweis biete bereits zum heutigen Zeitpunkt für realistische Problemstellungen brauchbare Lösungen oder sei gar eine Alternative zum Testen.

Nichtsdestoweniger bin ich der Meinung, daß die Beschäftigung mit dem Programmbeweis positive Auswirkungen auf den Programmierstil hat, weil die Beweistechnik den Programmierer zwingt formaler und präziser zu denken.

Der Programmbeweis hat zum Ziel, die Korrektheit eines Programms gegenüber der Spezifikation nachzuweisen. In /Goos 80/ wird ein Programm als korrekt definiert, wenn es zu jeder zulässigen Eingabe die durch die Spezifikation geforderte Ausgabe erzeugt.

Um einen Beweis überhaupt führen zu können, muß eine formale Spezifikation zusätzlich zum Programmtext vorliegen. Die Spezifikationssprache muß dabei nicht den detaillierten Algorithmus beschreiben, sondern es genügt, daß man damit die funktionalen Zusammenhänge, sowie die Ein- und Ausgabebereiche formuliert.

Basierend auf dieser Spezifikation muß gezeigt werden, daß das Programm (in der Implementierungssprache oder einer Zwischenform) die beschriebene Aufgabe erfüllt. Wichtig ist dabei, daß die verwendete Programmier- und Spezifikationssprache auf derselben logischen Basis (vgl. /Endr 76/ S. 311,326) aufsetzen.

Die meisten Autoren, die sich mit dem Programmbeweis auseinandersetzen, verweisen immer wieder darauf, daß die Beweistechnik sinnvollerweise während der Programmentwicklung und nicht erst beim fertigen Programm angewandt werden soll (/Lond 77/ p.113, /Myer 76/ p. 326).
Beispielsweise kann man sich vorstellen, daß während der Programmentwicklung Invarianten auf "natürliche" Weise entstehen und nicht erst wieder mühevoll aus dem existierenden Programmtext rekonstruiert werden müssen.

Eine der ersten und wohl auch die bekannteste Beweismethode ist die Methode der induktiven Assertionen /Floy 67/. Diese Methode ist die Grundlage für die meisten automatischen Programmbeweissysteme /Lond 77/. Im Verlauf der letzten Jahre wurden eine Reihe von Abwandlungen dieser Methode veröffentlicht (siehe als Übersicht /Gerh 79/ p. 96-97). Die Floydsche Beweismethode baut auf einer (sehr niedrigen) flußdiagrammähnlichen Beschreibung des Programms auf. Dagegen gehen spätere Ansätze mehr in die Richtung, den Beweis auf der Ebene einer höheren Programmiersprache oder gar einer abstrakten Spezifikation zu führen.

Der größte Fortschritt auf dem Gebiet des Beweisens hat sich in letzter Zeit beim Beweisen von Datenstrukturen ergeben. Das wurde möglich durch die Verwendung abstrakter Datentypen (ADT) /Gutt 77/, welche die Beweisführung erheblich vereinfachen. Jedem ADT sind Axiome zugeordnet, deren konkrete Realisierungen bewiesen werden müssen. Die Axiome beschreiben die Wirkung von Operationen und Operationsfolgen auf die Datenstruktur.

Der Begriff 'Programmbeweis' erweckt bei vielen Leuten den Eindruck, daß ein bewiesenes Programm fehlerfrei ist. Daß dies nicht unbedingt so sein muß, zeigen Beispiele aus der Literatur, wo in mehrfach als korrekt qualifizierten Programmen immer wieder Fehler gefunden wurden /Gerh 76/.

Diese Fehler zeigen exemplarisch zweierlei auf: zum einen beruhen viele Fehler auf Mißverständnissen der Spezifikation; zum anderen zeigt es sich, daß die Beweistechnik sehr kompliziert ist und nur von den wenigsten Programmierern beherrscht wird. Einen guten Überblick über die kontroverse Diskussion zum Thema 'Programmbeweis' sowie eine Gegenüberstellung von Stärken und Schwächen findet man in /Myer 76/ p. 319 ff.

3.2.2 Reviews

Synonyme für Review sind auch die Begriffe Inspections /Faga76, Faga 77/ und Walk-Throughs /Myer 79/. Man versteht darunter formalisierte Methoden zur manuellen Analyse von Softwareentwicklungsdokumenten. Reviews basieren darauf, daß ein Programm oder auch ein Entwurf von einem Team, bestehend aus mehreren Personen, gelesen und analysiert wird.

Eine charakteristische Review-Gruppe besteht aus ca. 4 Personen: einem Moderator, der die Review-Gruppe leitet, dem Programmierer, einem Entwerfer und einem Testspezialisten. Der Ablauf eines Review ist durch Prozeduren und Richtlinien genau festgelegt. Für die einzelnen Phasen existieren sogar Checklisten, die systematisch durchgearbeitet werden müssen mit dem Ziel, bestimmte Fehlerklassen auszuschließen. Eine detaillierte Beschreibung dieser Technik erfolgt in /Faga 76/ .

Die vorhandenen Arbeiten, insbesondere die aus der Fa. IBM, berichten von positiven Erfahrungen mit der Review-technik.

Als Vorteil von Reviews wird zumeist angeführt, daß man damit sehr effektiv Fehler findet. Darüberhinaus ist die Flexibilität beim Einsatz hervorzuheben, die es erlaubt, neben formalen Dokumenten wie dem Programmcode auch informale Dokumente wie Entwurfsbeschreibungen zu untersuchen.

Zusammenfassend ist zu sagen, daß die Review-Technik nicht als Ersatz, sondern als Ergänzung der auf der Maschine ablaufenden Testläufe zu sehen ist.

3.2.3 Testen

Ziele des Testens sind der Nachweis

- der funktionalen Korrektheit eines Programm(system)s,
- von Leistungskenngrößen (z. B. Laufzeit),
- der korrekten Funktion innerhalb einer bestimmten Systemumgebung.

Wir werden uns hier mit der funktionalen Korrektheit befassen und beschränken uns weiterhin auf sequentielle Programme. Besonderes Augenmerk werden wir dabei auf das Teilgebiet der Testplanung richten, die unserer Meinung nach für die Aussagekraft und Effizienz der Tests eine entscheidende Bedeutung hat.

Das Testen von Programmen ist jedem Programmierer geläufig und wird täglich praktiziert. Man kann uneingeschränkt sagen, daß der Programmtest innerhalb der analytischen Maßnahmen in der Praxis eine herausragende Stellung einnimmt. Wir wollen hier unter Testen verstehen, daß ein Programmstück in einer bestimmten Umgebung mit Daten versorgt und dann ausgeführt wird. Die dabei anfallenden Zwischen- und Endergebnisse werden mit den erwarteten Ergebnissen verglichen, wobei eine Differenz auf einen Fehler hindeutet.

Allgemein bekannt ist die Aussage von Dijkstra, daß nämlich das "Testen zwar das Vorhandensein von Fehlern nachweisen kann, nicht aber das Nichtvorhandensein". Aus diesem Dilemma und aus der Tatsache, daß das Testen trotz dieser Schwächen unumgänglich ist, kann man nur die Schlußfolgerung ziehen, daß das Testen systematischer betrieben werden muß (siehe auch /Endr 76/ S. 333). Durch diese Systematik ist es ungeachtet von Dijkstras Aussage möglich, den Verifikationswert des Testens zu erhöhen und beim Anwender das Vertrauen in die Zuverlässigkeit des Produkts zu erhöhen.

Da die Technik des Testens in den unterschiedlichsten Anwendungsgebieten, in verschiedenen Programmier- und Projektumgebungen und unter Verwendung unterschiedlicher Techniken und Werkzeuge durchgeführt wird, ist es schwer, ein einheitliches und vollständiges Bild des Testens zu entwerfen. Das Buch von Myers /Myer 79/ liefert aus der Sicht eines Praktikers einen guten Überblick über dieses Themengebiet. Daneben gibt es eine Reihe von Konferenzberichten und Sammelbänden, die sich mit diesem Themenkreis beschäftigen (/Hetz 73/, /Mill 77/, /Info 78/, /Comp 78/).

Bezeichnend für den Zustand, in dem sich die Technik des Testens im Augenblick befindet, ist dabei der Titel "The Art of Software Testing", den G. J. Myers für sein Buch gewählt hat. D. h., das Testen wird immer noch mehr als eine (schwarze) Kunst denn als ingenieurmäßiges Vorgehen angesehen.

Die komplexe Tätigkeit des Testens ist in die drei Aufgabengebiete

- Testplanung (Testfallauswahl, Testdatenerstellung etc.)
- Testdurchführung
- Testauswertung

zu unterteilen, die in den folgenden Kapiteln näher betrachtet werden. Wir gehen davon aus, daß in den Softwareentwicklungszyklus Meilensteine (Kontrollpunkte) integriert sind, an denen gewisse Testaktivitäten abzuwickeln oder zu überwachen sind.

3.2.3.1 Testplanung

Die Testplanung ist eine Aktivität, die überwiegend in der Entwurfsphase stattfindet und deren Ergebnis ein Testplan ist.

Der Testplan ist ein Dokument, in dem alle für die Durchführung der Tests relevanten Informationen niedergelegt sind. Die folgende Liste von Punkten wird in ähnlicher Form auch von /Myer 79/, /Hart 77/ und /Kope 75/ für einen Testplan gefordert.

1) Die Testziele jeder Testphase müssen festgelegt werden.
   Z. B. Test aller Funktionen der Anforderungsspezifikation.

2) Testabbruchkriterien legen fest, wann das Testen zu beenden ist. Die Notwendigkeit solcher Abbruchkriterien ergibt sich aus der Tatsache, daß ein vollständiger Test - d. h. ein Test aller möglichen Eingangskombinationen - selbst für einfache Programme nicht mehr in vertretbarer Zeit möglich ist.
   Gängige Abbruchkriterien orientieren sich an der Programmstruktur und lauten dann beispielsweise: 'Alle Verzweigungen müssen im Verlauf der durchgeführten Testläufe mindestens einmal ausgeführt sein'. Selbstverständlich sind Abbruchkriterien dieser Art nicht voll befriedigend, da sie kaum auf die Semantik des Testobjekts eingehen.

3) Kontrollpunkte: Zeitpunkte, an denen definierte Testdokumente vorliegen oder bestimmte Testaktivitäten abgeschlossen sein müssen.

4) Personalplanung (verantwortlicher Manager oder Programmierer).

5) Rechenzeit, die für die Tests benötigt wird.

6) Testfallbibliothek: Sätze von Testeingabedaten und erwarteten Testresultaten; Archivierung der Testergebnisse.

7) Testwerkzeuge: Z. B. Ablaufüberwacher zum Zählen der Ausführungshäufigkeit.

8) Hardware- und Software-Konfiguration.

9) Integrationsreihenfolge der Programmteile.

Durch den Testplan wird hauptsächlich der organisatorische Rahmen beschrieben, in dem die Tests durchzuführen sind. Dies ist eine in weiten Teilen managementorientierte Aufgabe, unter die hauptsächlich die Punkte 1 - 5 fallen. Dagegen sind die Punkte 6 - 9 eher dem technischen Teil der Testplanung zuzurechnen. Der technische Teil bildet den eigentlichen Kern des gesamten Testplans, denn dort werden letztlich die Testfälle beschrieben. D. h. die Testeingabedatensätze und die erwarteten Testergebnisse werden explizit (eventuell in programmiersprachlicher Notation) aufgeführt.

Da die Qualität des Testens letztlich daran zu messen ist, wie systematisch Testfälle und Testdaten erzeugt werden, wollen wir uns im nächsten Kapitel ausschließlich mit der Testfall- und Testdatengenerierung beschäftigen.

Zunächst weisen wir aber nochmals auf die unterschiedliche Bedeutung von Testfall- und Testdatenerzeugung hin. Testfallgenerierung ist der umfassende Begriff. Wir verstehen darunter die (funktionale) Zerlegung des Eingabedatenraumes in Äquivalenzklassen. Die Testfallgenerierung kann sowohl auf der Spezifikation als auch dem Programmcode aufsetzen und umfasst immer auch die Beschreibung der erwarteten Testausgabe. Dagegen verstehen wir unter der Testdatenerzeugung die Generierung eines speziellen Satzes von Eingangsdaten für jeden Testfall.

3.2.3.1.1 Testfall- und Testdatenerzeugung

Im Kapitel 2 haben wir festgestellt, daß die Bestimmung von Testdaten ein zweistufiger Prozeß ist. Auf der ersten Stufe werden aus beliebigen Softwareentwicklungsdokumenten Testfälle (d. h. Äquivalenzklassen) abgeleitet. Auf der zweiten Stufe werden für diese Testfälle die eigentlichen Testdaten erzeugt.

Im Verlauf des Softwareentwicklungszyklus wird eine Reihe von Dokumenten erstellt, die man ganz grob mit den einzelnen Phasen des SW-Entwicklungszyklus assoziieren kann. Die folgende Übersicht stellt den Softwareentwicklungsdokumenten jeweils gegenüber, welche Arten von Testdaten aus diesen Dokumenten gewonnen werden können.

Softwarentwicklungsdokument	Art der Testfall- bzw. Testdatenerzeugung
Anforderungsspezifikation	funktionsorientiert
Programmspezifikation und Entwurf	funktionsorientiert datenorientiert
Programmcode	funktionsorientiert datenorientiert strukturorientiert

Je formaler die Entwicklungsdokumente werden, desto umfangreicher werden die Möglichkeiten zur Testdatengewinnung. Diese Formalität wiederum ist die Voraussetzung, daß die Testdatenauswahl zumindest teilweise automatisiert werden kann.

Während auf der Ebene der Anforderungsspezifikation und des Entwurfs die Testobjekte zumeist als 'schwarzer Kasten' betrachtet werden, ist auf Programmcodeebene das Testobjekt auch als 'weißer Kasten' anzusehen. Dabei wird beispielsweise die Strukturinformation (u. a. Verzweigungsbedingungen) eines Programms für die Testdatenerzeugung ausgenutzt.

Im folgenden werden eine Reihe von Ansätzen beschrieben, die sich mit der Testfall- bzw. Testdatenerzeugung beschäftigen. Die Ansätze lassen sich in (teil-) automatische und manuelle Verfahren unterteilen, von denen die folgenden behandelt werden :

Wir beginnen mit den Verfahren, deren Automatisierungsgrad am stärksten ausgeprägt ist, und enden bei Verfahren, die ausschließlich manuell betrieben werden.

. Testdatengenerierung aus formalen Sprachbeschreibungen

. Testfallgenerierung mittels endlicher Automaten

. Symbolische Ausführung und Testdatenerzeugung

. Testdateigenerierung

. Testdatenerzeugung auf der Basis von Zufallszahlen

. Manuelle Techniken der Testfall- und Testdatengenerierung

Testdatengenerierung aus formalen Sprachbeschreibungen

Bereits relativ früh hat man sich im Bereich des Übersetzerbaus Gedanken gemacht, wie die formale Sprachbeschreibung für die Erzeugung von Testdaten verwendet werden kann. Hanford /Hanf 70/ beschreibt ein Programm, genannt "Syntaxmaschine", womit die syntaktische Korrektheit von Programmen überprüft wird. Die Syntaxmaschine erzeugt zufällige Testdaten, von denen bekannt ist, ob sie gültige oder ungültige Eingaben für ein Compiler-Front-End darstellen. Die erzeugten Testdaten sind möglicherweise semantisch nicht sinnvoll und liefern bei Ausführung nicht vorhersagbare Ergebnisse. Die Syntaxmaschine verlangt als Eingabe eine Definition der zu testenden Syntax in formaler Notation (erweiterte BNF). Man kann mit einer solchen Maschine sehr wohl die syntaktische Korrektheit einer Eingabe überprüfen, nicht aber die Korrektheit des gesamten Übersetzungsprozesses.

Testfallgenerierung mittels endlicher Automaten

Endliche Automaten sind ein mächtiges Modellierungshilfsmittel, das sich in so unterschiedlichen Bereichen wie der Beschreibung von Schaltwerken, der Definition von Übertragungsprotokollen, bei der Symbolentschlüsselung in Übersetzern und bei der Spezifikation von Dialogen bewährt hat.
Diese unterschiedlichen Anwendungsgebiete korrespondieren mit unterschiedlichen Interpretationen der Eingangs- und Ausgangsdaten des Automaten. So bestehen die Eingangsdaten bei der Symbolentschlüsselung beispielsweise aus Zeichen eines bestimmten Zeichenvorrats.

Gemeinsam ist allen hier aufgezählten Einsatzgebieten, daß die endlichen Automaten als Entwurfshilfsmittel eingesetzt werden.

In /Baue 79/ wird am Beispiel des Entwurfs elektronischer Wählsysteme ein weiteres Anwendungsgebiet endlicher Automaten vorgestellt. Interessant an diesem Beitrag ist der Ansatz, aus der Automatenbeschreibung des Wählsystems automatisch Testfälle zu generieren. Ein Testfall ist eine Sequenz von Benutzeraktionen und Systemreaktionen, die im Startzustand des Automaten beginnt und in einem Endzustand endet. Die Eingangsdaten des Automaten sind in diesem Fall die Benutzeraktionen (z. B. Telefonhörer abnehmen), und die Ausgabedaten sind die Systemreaktionen (z. B. 'Besetzt'-Zeichen).

Wir werden später aufzeigen, wie sich diese Idee, den endlichen Automaten als Testfallgenerator einzusetzen, ebenfalls im Bereich des Softwareentwurfs anbietet. Selbstverständlich erfordern die unterschiedlichen Beschreibungsobjekte (hier: Telefonwählsystem, dort: Softwareentwurf) eine unterschiedliche Interpretation der Testfälle sowie der Eingabe- und Ausgabedaten des endlichen Automaten.

Symbolische Ausführung und Testdatenerzeugung

Auf Programmcodeebene ist als strukturorientiertes Verfahren zur automatischen Testdatenerzeugung die symbolische Ausführung (für eine Definition siehe /Darr 78/) hervorzuheben. Ramamoorthy und Ho /Rama 76/ verwenden die symbolische Ausführung, um für einzelne Programmpfade symbolische Bedingungen zu ermitteln . Die Programmpfade sind dabei Knotenfolgen vom Eingangs- zum Ausgangsknoten eines Programmgraphen. Der vorgeschlagene Ablauf der zweistufigen Testfall- und Testdatenerzeugung ist:

1. Stufe:
   - Auswahl eines Programmpfades
   - Erstellung eines symbolischen Ausdrucks für den Pfad (Testfall)
   - Vereinfachung dieses symbolischen Ausdrucks
2. Stufe:
   - Generierung von echten Testdaten aus dem vereinfachten Ausdruck (mittels Einsetzverfahren und/oder Lösen von (Un-)Gleichungssystemen).

Ein ähnlicher Ansatz wie der hier beschriebene, wird auch von L. Clarke /Clar 76/ in ihrer Dissertation vorgeschlagen.

Probleme ergeben sich bei der Anwendung der symbolischen Ausführung bei bestimmten Programmanweisungen wie Schleifen mit variablen Schleifenobergrenzen, Feldern und bei Unterprogrammaufrufen. Nicht zuletzt aufgrund dieser Mängel hat sich die symbolische Ausführung bisher nicht in großem Stil in der Praxis durchgesetzt.

Es ist denkbar, die symbolische Ausführung und die damit verbundene Testdatenerzeugung nicht nur als eigenständiges Werkzeug bzw. Verfahren zu betreiben, sondern in ein umfassendes Werkzeugsystem als eine Komponente zu integrieren. Dies geschieht beispielsweise im System SADAT /Voge 80/, in dem die symbolische Ausführung neben Hilfsmitteln für den statischen und dynamischen Programmtest angeboten wird. Eine solche Kombination von Werkzeugen bietet sich schon deswegen an, weil von den einzelnen Werkzeugen oftmals auf dieselben Basisinformationen zugegriffen wird, die dann nur einmal erzeugt und abgespeichert sein müssen.

Ein Vorteil der symbolischen Ausführung besteht darin, daß ein symbolischer Ausdruck (Pfadprädikat) in geschlossener Form alle die Testdaten, die genau einen bestimmten Pfad zur Ausführung bringen, beschreibt. Damit hat man eine ganze Klasse sich identisch verhaltender Testdaten (einen Testfall) erfaßt. Da man sich dann beim Testen auf einen Repräsentanten dieser (Äquivalenz-)Klasse beschränken kann, erreicht man gegenüber dem vollständigen Testen eine bedeutende Reduktion der Testläufe.

Testdateigenerierung

Die Testdateigenerierung ist der eigentlichen Testdatengenerierung zuzuordnen. Ihr liegen folgende Überlegungen zugrunde:

Die meisten Programme, insbesondere in der kommerziellen Datenverarbeitung, arbeiten mit Dateien. Diese Dateien müssen zu Testzwecken mit Werten vorbesetzt werden, was bei einem manuellen Verfahren für große Dateien sehr mühsam ist. Daher wurden Werkzeuge entwickelt, die speziell Testdateien generieren. Ein charakteristischer Vertreter davon ist PRO/TEST /Zeda 76/. Mit diesem Werkzeug ist es möglich, Satzformate zu beschreiben, Datenbereiche und Datentypen festzulegen und für diese Festlegungen beliebige Mengen von Datensätzen zu generieren. Die Generierung großer Mengen unterschiedlicher Datensätze ist dabei über Parameter einfach steuerbar.

Eine mögliche Erweiterung dieses Ansatzes ist, daß direkt aus den Datenbeschreibungen eines Programmes (z. B. aus der File-Section eines COBOL-Programms) der Satzaufbau für den Testdateigenerator entnommen wird.

Ein Werkzeug dieser Art ist zwar in der Lage, die Testeingabedateien zu beschreiben, dagegen müssen die erwarteten Testresultate nach wie vor manuell erstellt werden, und ein Vergleich der erwarteten und tatsächlichen Resultate muß ebenfalls manuell durchgeführt werden.

Testdatenerzeugung auf der Basis von Zufallszahlen

Wir haben in den zuvor beschriebenen Verfahren gesehen, daß man durch Klassenbildung und Auswahl eines Repräsentanten dieser Klasse anstrebt, den Umfang der notwendigen Testdaten zu minimieren. Im Gegensatz zu diesen Verfahren wird bei der Erzeugung von Testdaten mittels Zufallszahlen keine Klassenbildung vorgenommen. Vielmehr geht man bei den zufällig erzeugten Testdaten davon aus, denselben Validationseffekt durch eine größere Menge von Testdaten zu erreichen.

Zufallszahlen ermöglichen es, in kürzester Zeit große Mengen von Testdaten zu generieren. Voraussetzung hierfür ist lediglich, daß die Typen der Eingangsvariablen und eventuell deren Verteilungen bekannt sind.

S. F. Lundstrom /Lund 78/ kombiniert Zufallszahlengeneratoren mit der Überwachung und Aufzeichnung der Testausführung. Aus so gewonnen Ausführungscharakteristiken werden für die Generierung neuer Testdatensätze Modifikationen der Verteilungsfunktionen abgeleitet. Ziel dieser Modifikationen ist es, die zum Erreichen eines Testziels (z.B. Ausführung aller Programmverzweigungen) insgesamt nötigen Testdatensätze zu reduzieren.

Im Zusammenhang mit der zufälligen Testdatenerzeugung ergeben sich zwei Probleme.
Erstens: die geringe 'Fehlersensitivität' von zufälligen Testdaten. Beispielsweise kann man sich vorstellen, daß die Variable N als zulässigen Definitionsbereich den gesamten INTEGER-Zahlenbereich umfaßt und nur im Fall N = 2 eine fehlerhafte Ausgabe erzeugt. In diesem Fall ist die Wahrscheinlichkeit der Fehlerentdeckung durch Zufallszahlen sehr gering.
Zweitens: die erwarteten Testresultate. Bei großen Mengen zufällig erzeugter Testdaten ist es in den meisten Fällen nicht einfach, die erwarteten Testresultate zu beschaffen.

Eine mögliche Lösung des letztgenannten Problems bietet die Technik des 'modellreferenzierten Testens' /Geig 79/. Hierbei wird ein Programm zweifach von unterschiedlichen Personen implementiert, wobei eine der Implementierungen das Testobjekt und die andere Implementierung ein vereinfachtes Modell des Testobjekts beschreibt. Beide Implementierungen werden nun mit zufälligen Testdaten versorgt. Eine genaue manuelle Untersuchung der Testresultate ist nur notwendig, wenn Testobjekt und Modell unterschiedliche Ergebnisse liefern. Bei Ergebnisgleichheit geht man davon aus, daß die Implementierung korrekt ist.

Manuelle Techniken der Testfall- und Testdatenerzeugung

Diese Art von Testfall- und Testdatenerzeugung lebt weitgehend von der Intuition und Erfahrung des Testers. Je nachdem, ob der Tester das Testobjekt als schwarzen oder weißen Kasten betrachtet, kommen dabei für die Testfall- und Testdatenerzeugung unterschiedliche Dokumente in Betracht, die manuell analysiert werden müssen.

Die Betrachtung als weißer Kasten setzt für die Testdatenerzeugung die Kenntnis des Programmcodes voraus. Innerhalb des Codes sind neben den Datendefinitionen insbesondere die Verzweigungsbedingungen interessant. Diese Verzweigungsbedingungen unterstützen die Einteilung des Eingabebereichs eines Programmes in Äquivalenzklassen. Aus den Datendefinitionen lassen sich Rand- und Spezialwerte ableiten (z. B. 0, 0.0, MAXINT).

Die Betrachtung als schwarzer Kasten liefert funktionale Testwerte, die aus der Spezifikation abzuleiten sind. Ausgehend von der Spezifikation ist ebenso wie im Falle eines weißen Kastens eine Einteilung der Eingabedatenbereiche in Äquivalenzklassen möglich. Es ist zu beachten, daß die unter der Betrachtung als weißer und als schwarzer Kasten gewonnen Äquivalenzklassen nur im Idealfall identisch sein werden.

In der Literatur (/Myer 79/ S. 37 ff,/Howd 78/ S. 384,/Goos 80/ S. 6-28) wird zumeist eine Reihe von Faustregeln angegeben, die sich in der Praxis als 'fehlersensitiv' herausgestellt haben.
Die Checklisten, die bei Inspections und Walkthroughs Verwendung finden, geben ebenfalls Hinweise für die Testdatenauswahl.
Darüberhinaus lassen sich für manche Implementierungssprachen Sonderregeln angeben, die bekannte Unzulänglichkeiten der Sprache durch Testfälle abdecken (in FORTRAN müssen diese Sonderregeln beispielsweise die Überschreitung von Indexgrenzen bei Reihungen einschließen).

Wir haben in diesem Abschnitt sechs verschiedene Ansätze zur Testfall- bzw. Testdatengenerierung beschrieben, die sich bezüglich der Rechnerunterstützung unterscheiden.

Die Testfallgenerierung aus formalen Sprachbeschreibungen ist völlig automatisierbar, da neben den eigentlichen Testdaten auch die erwarteten Testresultate (korrekte bzw. inkorrekte Programme) automatisch erzeugt werden. Demgegenüber ist der Automatisierungsgrad der Testfallgenerierung mittels endlicher Automaten geringer, da lediglich Testfälle (und keine Testdaten) automatisch erzeugt werden. Wenn es sich bei dem zugrundeliegenden Automaten um einen Automaten mit Ausgabe handelt, sind auch die erwarteten Ergebnisse eines Testfalls automatisch generierbar. Die Umsetzung der Testfälle in Testdaten erfolgt jedoch manuell. Die symbolische Ausführung ist ein Verfahren zur automatischen Testfallgenerierung, an das unter bestimmten Voraussetzungen eine automatische Testdatengenerierung anschließbar ist. Auf dieses Verfahren wurden anfänglich große Erwartungen gesetzt, die sich jedoch in der Praxis aufgrund verfahrenseigener Mängel (z. B. Schleifen mit variablen Obergrenzen) nicht erfüllten. Die Testdateigenerierung ist insbesondere im kommerziellen Bereich eine gängige Technik, um große Testdateien mit minimalem Aufwand parametergesteuert zu erstellen. Ebenso wie bei der Testdatenerzeugung mittels Zufallszahlen muß der Benutzer noch manuell die erwarteten Ergebnisse bestimmen, was aufgrund der großen Menge generierter Daten sehr aufwendig sein kann.

Als Fazit der (halb-)automatischen Verfahren zur Testfall- bzw. Testdatengenerierung ist festzustellen, daß die Verfahren mit sehr ausgeprägter Rechnerunterstützung nur ein begrenztes Anwendungsgebiet abdecken. Universeller einsetzbare Verfahren zeichnen sich durch eine geringere Rechnerunterstützung aus. Das Spektrum reicht bis zu den manuellen Verfahren der Testfall- und Testdatengenerierung, deren Anwendungsgebiet unbegrenzt ist.

3.2.3.1.2 Integrierte Verfahren der Testfall- und Testdatenerzeugung

Im letzten Kapitel wurden einzelne Ansätze zur Testfall- und/oder Testdatenerzeugung diskutiert. Als Fazit ist zu sagen, daß die meisten dieser Verfahren den Programmcode als Basisinformation verwenden. Demgegenüber ist festzustellen, daß Entwicklungsdokumente der frühen Softwareentwicklungsphasen - wie Anforderungsspezifikation und Entwurf - nur in bescheidenem Umfang für die Testfall- bzw. Testdatenerzeugung ausgenutzt werden.

In der Literatur werden sehr wenige für eine größere Menge von Problemstellungen geeignete Ansätze beschrieben, die in systematischer Form Informationen während der frühen SW-Entwicklungsphasen sammeln und daraus Testfälle und Testdaten ableiten. Eine solche phasenübergreifende Methodologie setzt voraus, daß die Tätigkeiten und Verfahren in den einzelnen Phasen aufeinander abgestimmt sind. Wir bezeichnen eine solchermaßen abgestimmte Vorgehensweise als integriertes Verfahren der Testfall- und Testdatenerzeugung.

Testfall- und Testdatenerzeugung mittels Cause-Effect-Graphen

Ein gutes Beispiel für ein integriertes Verfahren zur Testfall- und Testdatenerzeugung ist in dem Buch von Myers (/Myer 79/ S. 36-76) beschrieben.

Hierzu ist es notwendig, die natürlichsprachliche Spezifikation in eine formale Sprache, den Cause-Effect-Graph /Elme 73/, umzusetzen.

Die systematische Erzeugung von Testfällen läuft dann in folgenden Schritten ab:

a) Unterteilung der Spezifikation in überschaubare und abgeschlossene Teile.

b) Die 'Causes' und 'Effects' der Spezifikation werden identifiziert und eindeutig gekennzeichnet. 'Causes' sind unterschiedliche Eingangsbedingungen oder Äquivalenzklassen von Eingangsbedingungen. 'Effects' sind Ausgangsbedingungen oder Zustandsänderungen.

c) Die semantische Bedeutung der Spezifikation wird analysiert und zur Vervollständigung des Cause-Effect-Graphen herangezogen. So wird langsam die Lücke zwischen den 'Causes' und den 'Effects' durch Zwischenzustände geschlossen.

d) Aufgrund von Restriktionen in der Spezifikation werden einzelne 'Causes' zu Kombinationen zusammengefaßt.

e) Der so erstellte Cause-Effect-Graph wird systematisch in eine Entscheidungstabelle (ET) umgesetzt. Die Zeilen der ET repräsentieren die (Elementar-) Bedingungen und die Spalten die Testfälle.

f) Die einzelnen Spalten (Testfälle) der ET werden in echte Testdaten umgesetzt.

Die Verwendung der Entscheidungstabellentechnik gestattet es, die Vollständigkeit der Testfälle in Bezug auf die Elementarbedingungen nachzuweisen.

Testfallermittlung in modularen Softwareentwicklungen

Als weiteres Beispiel für ein integriertes Verfahren der Testfallermittlung sei das Projektmodell /Dene 80/ der Firma Softlab vorgestellt. Dieses Projektmodell enthält als wesentliche Komponente ein Modularisierungskonzept /Dene 79/, das auf den Ideen der abstrakten Datentypen aufbaut. Assoziiert mit dieser Modularisierung ist eine Teststrategie.

Diese beruht darauf, die Moduln nur über ihre in der Spezifikation festgelegten Schnittstellen zu testen. Hierfür werden Testfälle spezifiziert, die später mit einem Testtreiber ausgeführt werden. Ein Testfall ist durch eine Folge von Aufrufen der zu testenden Moduloperationen festgelegt. Die Eingabeparameter für die Aufrufe der Moduloperationen werden in der jeweiligen Implementierungssprache formuliert. Testeingabedaten von außen, etwa aus einer Datei oder von einer interaktiven Eingabe des Testers, werden nicht benutzt.

Die Herleitung der Testfälle erfolgt aus der Spezifikation, wofür jedoch kein formalisiertes Verfahren angegeben wird. Vielmehr wird die Übereinstimmung mit den Zielen der Spezifikation durch Inspektionen und Reviews überprüft.
Das hier beschriebene Verfahren ist insofern als integriertes Verfahren anzusehen, weil das Testkonzept auf die modulorientierte Spezifikation abgestimmt ist.

Testfallermittlung aus algebraischen Spezifikationen

Für algebraische Spezifikationen läßt sich ein Testfallgenerierungsverfahren angeben, das auf diese Spezifikationstechnik abgestimmt ist und damit als integriertes Generierungsverfahren anzusehen ist. Algebraische Spezifikationen zeichnen sich durch Axiome aus, die den Zusammenhang zwischen den Operationen eines abstrakten Datentyps formal beschreiben. Im einfachsten Fall gilt, daß ein Axiom einem Testfall entspricht. In /Gann 81/ wird das System DAISTS beschrieben, welches die formal beschriebenen Axiome einer algebraischen Spezifikation für die Generierung von Testrahmen ausnutzt und diese Axiome gleichzeitig als Vorgabe für die erwarteten Ergebnisse verwendet. In DAISTS werden die Resultate auf beiden Seiten der Spezifikationsaxiome durch den Testrahmen auf Gleichheit überprüft. Falls bei der Programmausführung keine Übereinstimmung erzielt wird, werden die Axiomennamen und die Testfallnummern gemeldet. Testdaten für die einzelnen Testfälle gewinnt man durch eine entsprechende Parameterversorgung der Moduloperationen.

In /Comm 82/ wird eine erste Idee skizziert, wie auf der Basis der Spezifikations- und Entwurfssprache SLAN-4 /Beic 80/ eine Testfallgenerierung aussehen könnte. Die in /Comm 82/ beschriebene Vorgehensweise für die Testfallgenerierung ist weitgehend aus dem System DAISTS entnommen. Darüberhinaus ist vorgesehen, die Vor- und Nachbedingungen (Assertionen) der Prozedurspezifikationen (in SLAN-4 = Modulspezifikationen) zur Testzeit ausführbar zu machen. Damit lassen diese Assertionen sich ebenso wie die Axiome in DAISTS als Vorgabe für die erwarteten Ergebnisse heranziehen.

3.2.3.2 Testdurchführung

Unter Testdurchführung verstehen wir den dreiteiligen Prozeß:

- Übernahme der Testeingangsdaten
- Ausführung des Testobjekts
- Aufzeichnen von Zwischen- und Endergebnissen sowie von testspezifischen Ausgaben.

Die Testeingangsdaten werden aus dem Testplan entnommen. Um die Vorbereitung der Testausführung zu erleichtern und die eigentliche Testausführung meßbar zu machen, stehen eine Reihe von Testhilfsmitteln und Werkzeugen zur Verfügung.

So werden beispielsweise Modultestrahmen (/Panz 78/, /Heue 74a/, /Heue 74b/) angeboten, die die Parameterversorgung eines Moduls übernehmen und den allein nicht ablauffähigen Modul in ein komplettes Programm integrieren. Andere Werkzeuge zeichnen das dynamische Verhalten des Testobjekts während der Ausführung auf (/Huan 78/, /RXVP 81/, /Stuc 77/). Diese Systeme zur Ausführungsüberwachung basieren zum Teil auf Überlegungen, an welchen Stellen innerhalb eines Programms Instrumentierungspunkte optimal anzulegen sind /Rama 76a/.

Eine ganze Reihe von Fähigkeiten dieser Testwerkzeuge sind heute bereits in modernen Compilern verfügbar (z. B. stellt der IBM-PASCAL-Compiler /IBM 81/ neben anderen Fähigkeiten eine Möglichkeit zur Überwachung der Ausführungshäufigkeit zur Verfügung).

Jegliche Instrumentierung eines Programms verändert eben dieses Programm teils signifikant. So haben wir im IDT beim Programmsystem RXVP für sequentielle Programme eine Vergrößerung des Speicherplatzbedarfs um 30 - 100 Prozent und der Laufzeit um 50 - 100 Prozent beobachtet. Während sich diese Veränderungen bei sequentiellen Programmen meist noch tolerieren lassen, muß man sich in der Realzeitprogrammierung (oder auch bei parallelen Programmen) andere und erweiterte Testtechniken einfallen lassen. Hierbei kommt man bei Realzeitproblemen oftmals nicht daran vorbei, spezielle Hardware zu entwickeln, die das Testobjekt zeitlich und speicherplatzmäßig gar nicht oder nur minimal belasten.

3.2.3.3 Testauswertung

Die Auswertung von Testausgabedaten erfordert Kriterien, die in einem Testplan niedergelegt werden. Diese Auswertekriterien können von exakten Ausgabewerten bis zur Kontrollsummenbildung oder zur Beschreibung von Abhängigkeiten zwischen Ausgabewertepaaren reichen.

Man stellt fest, daß die Testauswertung sehr stark von den verfügbaren Dokumenten und Informationen aus Spezifikation und Entwurf abhängt. Die Testauswertung erfolgt in den meisten Anwendungen noch manuell durch einen Vergleich der aus der Spezifikation abgeleiteten Werte mit den tatsächlich gemessenen Werten. In manchen Anwendungen ist auch eine mechanische Auswertung der Ergebnisse möglich. Dies gilt insbesondere dann, wenn ein einfacher funktionaler Zusammenhang zwischen Testeingabe und Testausgabe besteht. Bei Wiederholungstests (Regressionstests) bietet sich der Einsatz von automatischen Vergleichsprogrammen an. Damit lassen sich Diskrepanzen der Testausgaben zwischen dem ursprünglichen Programm und dem modifizierten Programm feststellen. Thayer et. al. (/Thay 78/ p. 198) berichten, daß gerade in den von ihnen untersuchten großen Projekten ein Datenbasisvergleichsprogramm für die Testunterstützung extrem nützlich war. Den praktischen Stellenwert solcher Vergleichsprogramme unterstreicht die Tatsache, daß unter fortgeschrittenen Betriebssystemen bereits Kommandos (z. B. unter UNIX das Kommando DIFF) dafür angeboten werden.

3.2.3.4 Zusammenhang zwischen Testen und Beweisen

Für die Verifikation (d. h. Demonstration der Konsistenz zwischen Spezifikation und Programm) kommen sowohl der Programmbeweis als auch das Testen als Methode in Frage. Je nach Standpunkt wird von den Praktikern oft die Meinung vertreten, daß das Testen "die allein seligmachende Methode" ist, während die mehr theoretisch orientierten Personen "auf den Programmbeweis schwören". Solche extremen Positionen lassen sich bei genauerem Hinsehen selbstverständlich nicht aufrechterhalten. Wir wollen hier anhand von Veröffentlichungen von S. Gerhart /Gerh 77/ und M. Geller /Gell 78/ zeigen, wie beide Techniken nützlich einzusetzen sind, und sich sogar gegenseitig ergänzen.

So beschreibt Susan Gerhart /Gerh 77/, wie Testen und Beweisen als komplementäre Methoden demselben Ziel, nämlich der Verifikation, dienen können. Sie charakterisiert dabei als Besonderheit des Testens die "systematische, detaillierte, konkrete und fallweise Vorgehensweise", während die Besonderheit des Beweisens in seiner "systematischen, allgemeinen, abstrakten, deduktiven und induktiven Vorgehensweise" liegt.

Zur Veranschaulichung dieser Aussagen werden im folgenden die komplementären Eigenschaften erläutert. Wir stützen uns hierbei auf /Gerh 79/.

a) Testen ist konkret: D. h., es wird das Programm zusammen mit aktuellen Daten in einer Umgebung (inkl. Software und Hardware) als Testobjekt verwendet. Dabei wird das echte Verhalten des Testobjekts beobachtet. Beweisen ist abstrakt, insofern als dabei die allgemeinen Eigenschaften der Spezifikation und des Programms verwendet werden. Der Beweis liefert also Erkenntnisse, ob für die Gesamtproblemstellung eine plausible Lösung gefunden wurde, während das Testen nachweist, daß Teile des Problems, aber nicht notwendig das Ganze, korrekt implementiert wurden.

b) Das Testen ist fallorientiert, insofern als es für individuelle Testfälle exakte Ergebnisse liefert. Demgegenüber liegt der Schwerpunkt des Beweises, unabhängig von individuellen Testfällen, auf einer allgemeinen Betrachtungsweise.

c) Durch sukzessives Testen kann man zwar für mehr und mehr Fälle die (punktweise) Korrektheit nachweisen, jedoch gibt es i. a. keinen Induktionsschluß, der darüberhinausgehende, allgemeinere Aussagen zuließe. Demgegenüber kann mit Hilfe des Beweisens, vorausgesetzt die Verifikationsbedingungen und die Durchführung sind korrekt, geschlossen werden, daß auch das Programm korrekt ist.

Testen und Beweisen können in einem komplementären Ansatz zur Erkennung und Vermeidung unterschiedlicher Fehlerklassen herangezogen werden.

Geller /Gell 78/ beschreibt in seiner Arbeit einen Ansatz, wie mit Hilfe von Testdaten ein Programm zu beweisen ist. Er geht dabei davon aus, daß es einfacher ist, das Verhalten eines Programms durch seine Reaktion auf Testdaten zu beschreiben, als etwa durch eine formale Spezifikationsmethode.

In seinem Vorschlag führt er drei Arten von Assertionen ein : die 'synthetisierten Assertionen', die 'Testdatenassertionen' und die 'Verallgemeinerungsassertionen'. Die Testdatenassertionen beschreiben die Werte, die die Programmvariablen am Ein- und Ausgang eines Programmstückes haben sollen. Diese Assertionen lassen sich durch Ausführung mit einigen Testdaten überprüfen. Mit Hilfe der Verallgemeinerungsassertion schließt man aus den (punktuellen) Ergebnissen der Testdatenassertion auf größere Bereiche (Klassen von Daten). Die synthetische Assertion wird aus den Testdaten- und den Verallgemeinerungsassertionen abgeleitet und ist mit einer Floyd'schen /Floy 67/ Assertion vergleichbar.

Aus diesen beiden hier vorgestellten Veröffentlichungen könnte man entnehmen, daß Testen und Beweisen von ihrer praktischen Bedeutung her gleichwertig sind. Daß dies nicht so ist, liegt an einer Reihe von pragmatischen Gründen:

- Beweisen ist schwierig und wird nur von einigen hochqualifizierten Spezialisten beherrscht.

- Beim Testen läßt sich das Programm in seiner Betriebsumgebung (zusammen mit Betriebssoftware, Hardware und Bediener) überprüfen.

- Testen ist eine interdisziplinär verständliche Tätigkeit, zu der auch Auftraggeber und Endbenutzer ihren Beitrag leisten können.

3.2.3.5 Theoretische Fundierung des Testens

In diesem Abschnitt gehen wir auf zwei Ansätze ein, von denen der erste von Goodenough und Gerhart /Good 77/ große Allgemeingültigkeit besitzt, während der zweite Ansatz von White und Cohen /Whit 80/ bereits wesentlich spezieller ist. Der letztgenannte Ansatz benutzt den Programmcode und erlaubt es, bestimmte Fehlerklassen (z. B. Fehler im Programmablauf) gezielt zu betrachten.

Goodenough und Gerhart /Good 77/ beschreiben den augenblicklichen Stand mit dem Satz : "Wir wissen weniger über die Theorie des Testens, was wir oft tun, als über die Theorie des Beweisens, was wir selten tun." Allgemein bekannt ist, daß nur ein erschöpfender Test, d. h. ein Test mit allen möglichen Eingangsdatenkombinationen, eine vollständige Verifikation garantieren kann. Da ein solcher Test aufgrund der sehr großen Zahl von Möglichkeiten nicht durchführbar ist, muß man Kriterien finden, anhand derer eine Untermenge der insgesamt möglichen Testfälle ausgewählt wird. Diese Untermenge sollte möglichst denselben Verifikationswert besitzen wie ein vollständiger Test. Ziel einer theoretischen Fundierung des Testens (und insbesondere der Testdatenauswahl) muß es sein, eine Begründung für die Auswahl bestimmter Testfälle zu liefern und eine Beziehung zwischen Testzielen und den dazu notwendigen Testdaten herzustellen. Diese Beziehung kann nicht alleine durch eine Betrachtung des Programms als schwarzer Kasten hergestellt werden, sondern sie benutzt Informationen aus dem gesamten Softwareentwicklungszyklus.

Die Hauptaussage dieser von Goodenough und Gerhart aufgestellten Testtheorie ist in einem "Fundamentalsatz" zusammengefaßt und lautet in nichtformaler Notation :

Satz: Ein erfolgreich ausgeführter Test ist äquivalent zu einem Korrektheitsbeweis, wenn der Test einem Testdatenselektionskriterium genügt, das wirksam und zuverlässig ist.

Im folgenden werden die Begriffe in diesem Satzes erläutert:

- Ein Test ist eine Untermenge der insgesamt zulässigen Eingangsdaten eines Programms.

- Ein Testdatenselektionskriterium ist eine Vorschrift, wie Teilmengen der zulässigen Eingangsdaten auszuwählen sind.
  (Anmerkung: Ein Testdatenselektionskriterium entspricht in unserer Terminologie einem Testfall).

- Ein Test gilt als erfolgreich ausgeführt, wenn die Ausgangswerte "normal" sind. D. h., die Ergebnisse von Spezifikation und Implementierung stimmen für alle durch das Testdatenselektionskriterium bestimmten Testdatensätze überein.

- Ein Testdatenselektionskriterium ist wirksam, wenn mindestens ein durch das Testkriterium beschriebener Testdatensatz einen Fehler, sofern überhaupt ein Implementierungsfehler vorliegt, erkennt.

- Ein Testdatenselektionskriterium ist zuverlässig, wenn alle durch das Testkriterium beschriebenen Testdatensätze zum gleichen Ergebnis kommen. D. h., die Testläufe müssen entweder alle erfolgreich sein oder alle müssen einen Fehler erkennen.

In der Praxis kommt es nun darauf an, die Zuverlässigkeit und Wirksamkeit eines Testdatenselektionskriteriums nachzuweisen. Goodenough und Gerhart geben ein konstruktives Verfahren an, wie aus Spezifikation, Entwurf und Implementierung ein "zuverlässiges" Testdatenselektionskriterium gewonnen werden kann. Dieses Verfahren basiert darauf, Prädikate (aus Fallunterscheidungen der Spezifikation und/oder der Implementierung) zu erstellen und diese Prädikate in geeigneter Weise zu komplexeren Prädikaten zusammenzubauen. Dabei ist es wichtig, die Prädikate so auszuwählen, daß sie "unabhängig" sind. D. h., zwei Testdatensätze, die dieselbe Prädikatenmenge erfüllen, müssen auch zum selben Ergebnis führen (Hinter dieser Überlegung steht die Einteilung des Eingabedatenraumes in Äquivalenzklassen. Die einzelnen Elemente einer Äquivalenzklasse verhalten sich bezüglich des Ergebnisses vollkommen gleich.).

White und Cohen /Whit 80/ schlagen eine Strategie für den Programmtest vor, die aber nur für eine eingeschränkte Klasse von Programmen (Programme, deren Pfadprädikate durch Linearkombinationen der Eingangsdaten darstellbar sind) gilt. Mit Hilfe dieser Strategie ist es möglich, Fehler im Ablauf eines Programms zu finden. Die Anweisungen eines Programms, die den Ablauf beeinflussen (z. B. Verzweigungen), teilen dabei den Raum der Eingangsdaten in Bereiche (domains), die sich wechselseitig ausschließen. Jeder dieser Bereiche korrespondiert mit einem bestimmten Programmpfad. Assoziiert mit diesem Programmpfad ist eine Menge von Eingangsdaten, die genau zu diesem Programmpfad führen. Die Teststrategie besteht nun darin, Testpunkte so festzulegen, daß Bereichsverletzungen entdeckt werden. Durch die Beschränkung auf lineare Prädikate ist sichergestellt, daß der Aufwand an Testfällen nur linear mit der Zahl der Eingangsvariablen und der Zahl der Pfadprädikate des Testobjekts wächst.

Nachteilig an diesem Verfahren ist neben der Beschränkung auf die relativ kleine Klasse von Programmen mit linearen Pfadprädikaten vor allem die große Zahl von Testfällen bei strukturell komplexen Programmen.

3.2.3.6 Zusammenhang zwischen Test und Dokumentation

Für eine Programmdokumentation ist zu fordern, daß sie u. a. Testfälle und Testdaten enthält. Auf der Basis solcher Dokumentationsinformationen sind dann bei Programmänderungen Wiederholungstests (Regressionstests) durchführbar. Diese Tests werden notwendig, wenn ein Fehler verbessert wurde oder aber ein bestehendes Programm um eine zusätzliche Funktion ergänzt wurde. Danach ist nachzuweisen, daß diese Änderung keine anderen Programmfunktionen in unerwünschter Weise beeinflußt hat. Um diesen Nachweis führen zu können, müssen früher durchgeführte Testläufe in geeigneter Weise archiviert werden.

Die Praxis zeigt, daß insbesondere bei Programmänderungen die Gefahr, neue Fehler zu machen, groß ist (laut /Myer 76/ p. 252, besteht eine 20 bis 50-prozentige Chance, beim Verbessern eines Fehlers wieder einen Fehler einzubauen). Hilfsmittel für den Regressionstest reichen von einfachen Testdateivergleichsprogrammen bis zu Systemen, wie etwa AUTORETEST (/Houg 80/ p. 12), die die Testwiederholung weitgehend automatisieren.

Über diese Regressionstests hinaus ist die Dokumentation selbst als Testdatenquelle anzusehen. Die hieraus gewonnen Testdaten entdecken Diskrepanzen zwischen Benutzerdokumentation und tatsächlicher Implementierung. Daß diese Diskrepanzen einerseits trivial, andererseits die tägliche Praxis sind, ist an nahezu jedem Handbuch überprüfbar.

3.3 Softwareproduktionsumgebungen (SPU'en)

Alle Entwicklungsdokumente, die als Zwischen- oder Endprodukte von Testaktivitäten anfallen, sind ebenso wie die übrigen Entwicklungsdokumente als Bestandteile von größeren Softwareproduktionsumgebungen (/Haus 81/, /Hess 81/ s. 139 ff) anzusehen. Motivation für die Beschäftigung mit SPU'en ist u. a. die Erkenntnis, daß zur Qualitätsverbesserung der Software ein aufeinander abgestimmter Satz von Methoden, Verfahren und Werkzeugen notwendig ist.
Ein erster Schritt in diese Richtung sind die integrierten Verfahren zur Testfall- und Testdatenerzeugung (vgl. Kapitel 3.2.3.1.2).

In bereits existierenden SPU'en spielt nicht zuletzt der Validations- und Testaspekt eine wesentliche Rolle. Nach /Haus 81/ stellt beispielsweise das System SWB /Mats 81/ der Firma Toshiba die am weitesten entwickelten Testsysteme innerhalb einer SPU zur Verfügung. SWB bietet Instrumente und Verfahren, um Tests zu planen, zu spezifizieren und die Ergebnisse zu dokumentieren. Dies ist für Teilsysteme und auch für komplette Systeme möglich. Darüberhinaus werden neben Testwerkzeugen auf dem Entwicklungsrechner auch solche auf dem Zielrechner bereitgestellt.

Obwohl zum Thema SPU'en in letzter Zeit sehr viele Veröffentlichungen erschienen sind, ist man noch weit davon entfernt, ein marktgerechtes und allgemein anerkanntes System anbieten zu können. So behelfen sich viele Firmen damit, daß sie sich auf ihre lokalen Bedürfnisse zugeschnittene SPU'en aufbauen. Dies setzt zunächst voraus, daß man sich auf ein Projektmodell (d. h. einen Softwarelebenszyklus, Validations- und Testpunkte sowie Entwicklungsdokumente) einigt.

3.4 Konklusion aus dem Stand der Technik

In den letzten Kapiteln haben wir einen Überblick über den Stand der Technik bei der Erstellung von zuverlässiger Software gegeben. Schwerpunktmäßig haben wir dabei das Gebiet des 'Programmtestens' behandelt.

Auf den ersten Blick bekommt man den Eindruck, daß im Bereich des Testens sehr viele Einzelansätze existieren. Erst in jüngster Zeit wird verstärkt an integrierten Ansätzen für das Software-Engineering (Softwareproduktionsumgebungen) im allgemeinen und das Testen im besonderen gearbeitet.

Das Testen erfordert einen hohen Aufwand und ist bei manueller Tätigkeit sehr fehlerträchtig. Durch Automatisierung und den Einsatz von Werkzeugen ist eine Verbesserung erzielbar. Existierende Testwerkzeuge basieren bisher zumeist auf dem Programmcode als formalem Dokument. Dies gilt insbesondere auch für Werkzeuge zur Testfallgenerierung und Testdatenauswahl. Für die automatische Testfallgenerierung während der Entwurfsphase gibt es dagegen nur wenige Vorschläge (vgl. 3.2.3.1.2), die zudem nicht die Vollständigkeitseigenschaft bei der Testfallgenerierung garantieren.

Der zentrale Punkt, der die Güte des Testens unserer Meinung nach entscheidend beeinflußt, ist die systematische Auswahl signifikanter und 'fehlersensitiver' Testfälle. In den vorigen Kapiteln wurde aufgezeigt, daß bisher hauptsächlich der Programmcode als Basis für Werkzeuge zur Testfallgenerierung diente. Dagegen wurden die Spezifikations- bzw. Entwurfsdokumente vernachlässigt, wobei die Ursachen dafür in der geringen Formalität dieser Dokumente liegen.

Wenn man den Einsatz von Testwerkzeugen in den frühen Phasen der Softwareentwicklung vorantreiben will, muß man zwei Probleme angehen:
Erstens muß eine gewisse Formalität der Spezifikations- und Entwurfsdokumente gewährleistet sein.
Zweitens muß die Schnittstelle zwischen Spezifikations- bzw. Entwurfswerkzeugen und den Testwerkzeugen kompatibel sein.

In vielen Papieren, die sich mit Methoden des Software-Engineering beschäftigen, wird eine entwicklungsbegleitende Testplanung und Testdatenerzeugung gefordert. Trotz dieser Forderung findet man in der Literatur nur wenige ausgereifte Verfahren, die ein Konzept zur Testfall- und Testdatenerzeugung auf der Basis von Spezifikation und Entwurf anbieten. Die Lage ist so zu charakterisieren, daß zwar der Wunsch nach einer entwicklungsbegleitenden Testplanung besteht, daß man aber nicht so richtig weiß, wie man diesen Wunsch mit Hilfe eines allgemein gültigen Verfahrens in die Tat umsetzen kann.

Die Datenabstraktion als Spezifikations- und Entwurfstechnik bildet eine geeignete, formale Schnittstelle, auf der Testwerkzeuge aufbauen können. Im folgenden Kapitel wird deshalb vorschlagen, wie ein auf abstrakten Datentypen basierender Entwurf als Basis für eine automatische, entwurfsbegleitende Testfallgenerierung aussieht.

4. TESTPLAN - Ein System zur entwurfsbegleitenden Testplanung und Testdurchführung

Im letzten Kapitel haben wir einige Bereiche herausgearbeitet, für die es keine oder nur unbefriedigende Lösungen gibt. Man kann einen Teil dieser Probleme durch folgende Vorgehensweise lösen:

> Man verwendet als Spezifikations- und Entwurfstechnik die Datenabstraktion und bekommt dadurch als Testobjekt einen abstrakten Datentyp (ADT), den wir im folgenden synonym auch als Modul bezeichnen. Der Vorteil dieses Objekts liegt darin, daß der ADT dem Benutzer nur über Zugriffsfunktionen zugänglich ist. Diese Eigenschaft läßt sich neben der Implementierung auch vorteilhaft für einen funktionalen Test ausnutzen. Die formal beschriebene, funktionale Schnittstelle ermöglicht damit die Weiterentwicklung der Implementierung und parallel dazu die Erstellung eines Testplans.
>
> Über die gängigen Eigenschaften eines ADT's hinaus lassen sich Aussagen zur Aufrufreihenfolge von Zugriffsfunktionen machen. Diese Aussagen werden in Form von Assertionen mit Hilfe regulärer Ausdrücke formuliert und sind Teil der Entwurfsbeschreibung. Auf der Basis dieser Assertionen, von denen wir annehmen, daß sie die Aufrufbeziehungen der Funktionen eines ADT's vollständig beschreiben, werden aus der Entwurfsbeschreibung Testfälle automatisch abgeleitet. Somit ist festzustellen, daß die Anbindung der Testwerkzeuge an die Entwurfsbeschreibung über die Assertionen erfolgt. Für die Umsetzung der Assertionen in Testfälle verwenden wir aus der Automatentheorie bekannte Algorithmen. Die automatisch generierten Testfälle sind vom Benutzer noch interaktiv in konkrete Testdaten umzusetzen.
>
> Quasi als 'Abfallprodukt' der formalen Entwurfsbeschreibung lassen sich aus dem Entwurf auch noch automatisch Testrahmen ableiten. Diese Testrahmen dienen zur Aufnahme der parallel von einer unabhängigen Person implementierten Funktionen des ADT's.

Für diese hier prinzipiell vorgestellte Vorgehensweise wurde ein System zur entwicklungsbegleitenden Testplanung und Testdurchführung (TESTPLAN) entwickelt. Das System TESTPLAN besteht aus drei Komponenten:

- Einem Verfahren zur Erstellung eines Testplans parallel zum Programmentwurf (Kap 4.2).
- Einer Sprache zur Formulierung des Softwareentwurfs und der Testpläne (Kap 4.3).
- Einem Werkzeug zur Umsetzung der sprachlichen Formulierung in eine interne Darstellung, sowie deren weitere Verarbeitung (Kap 4.4).

4.1 Ziele und Abgrenzung des Systems TESTPLAN

Aus den eingangs erwähnten, unbefriedigend gelösten Testaktivitäten ergeben sich für das System TESTPLAN folgende detaillierte Ziele:

- Die Testpläne sollen auf der Basis der Entwurfsbeschreibung parallel zur Programmentwicklung erstellt werden. Damit soll die Unabhängigkeit von Test- und Implementierungsperson gewährleistet bzw. ermöglicht werden.

- Aus dem Entwurf sollen automatisch Testfälle abgeleitet werden. Diese Testfälle bilden das Grundgerüst der Testpläne.

- Die Testpläne sollen in einer maschinell verarbeitbaren Form (Sprache) beschrieben werden. Diese Testplansprache ist in die Entwurfssprache integriert.

- Aus den Testplänen sollen automatisch ablauffähige Testrahmen generiert werden.

- Eine Speicherung der Testpläne auf dem Rechner soll die Wiederholbarkeit der Testläufe sicherstellen.

- Die Zuordnung von einzelnen Funktionen der Anforderungsspezifikation zu Testfällen muß möglich sein. (Damit ist eine Verfolgungsmöglichkeit von der Systemanforderung über die Implementierung bis in die Testphase gegeben.)

- Informationen, die in der Entwurfsbeschreibung bereits vorliegen, sollen für die Testplanung verwendet werden.

Da kein einzelnes Verfahren oder Werkzeug dieses breitgefächerte Zielspektrum überdeckt, ist ein Netz sich ergänzender konstruktiver und analytischer Maßnahmen erforderlich.

Die wichtigsten Maßnahmen, die in TESTPLAN eingearbeitet wurden, sind:

Konstruktive Maßnahmen:
- formale Entwurfsbeschreibungssprache
- Modularisierung und Datenabstraktion
- Assertionentechnik

Analytische Maßnahmen:
- Testfallgenerierung aus dem Entwurf
- Archivierung der Testfälle
- automatisch generierte Testrahmen.

Bevor TESTPLAN gegenüber vergleichbaren Entwicklungen abgegrenzt wird, erläutern wir noch den Assertionen-Begriff.

Assertionen sind Ausdrücke, die den erwarteten Zustand eines Objekts (Programm, Teile eines Entwurfs, ...) zu einem bestimmten Zeitpunkt beschreiben. Dies geschieht in Form von Bedingungen, denen diese Objekte genügen müssen. Im System TESTPLAN bilden die Assertionen den wesentlichen Teil der Schnittstelle zwischen dem formal beschriebenen Entwurf und dem darauf aufbauenden Testplan. Wir unterscheiden zwei Arten von Assertionen, die auf die verwendete Entwurfstechnik (Datenabstraktion) abgestimmt sind:
Erstens, globale Assertionen zur Formulierung funktionaler Zusammenhänge zwischen den Zugriffsfunktionen eines ADT's.
Zweitens, lokale Assertionen zur Beschreibung des Eingabe-/Ausgabeverhaltens der einzelnen Zugriffsfunktionen.
Diese beiden Assertionenarten sind vollkommmen unabhängig voneinander. Sie bilden jeweils die Basis für eigenständige Testfallgenerierungsverfahren, die in den Kapiteln 4.2.2.1 bzw. 4.2.2.2 behandelt werden.

Es gibt meines Wissens zwei Ansätze, die basierend auf abstrakten Datentypen eine Testfallgenerierung aus dem Entwurf anstreben und am ehesten mit dem System TESTPLAN vergleichbar sind:

1) Die Softlab-Produktionsumgebung S/E/TEC enthält unter anderem die Teilsysteme EDDA (Entwurfsdialekt zur Datenabstraktion) und TUS (Testunterstützungssystem). Diese beiden Teilsysteme werden von Hesse (/Hess 81/ S. 125, S. 145 ff) beschrieben.

2) Das System DAISTS /Gann 81/ (vgl. Kap 3.2.3.1) zur Implementierung, Spezifikation und zum Testen von Datenabstraktionen. In dieselbe Richtung wie DAISTS geht die Firma IBM, indem sie auf ihrer Spezifikations- und Entwurfssprache SLAN-4 (/Beic 80/, /Comm 82/) aufbaut. Diese Sprache unterstützt die algebraische Spezifikation.

Abgrenzung gegenüber dem Softlab-Ansatz
Sowohl TESTPLAN als auch der Softlab-Ansatz gehen davon aus, daß der Entwurf in Form von Moduln (im Sinne der abstrakten Datentypen) beschrieben ist. Innerhalb des Teilsystems TUS ist zwar eine Testfall- und Testdatengenerierung geplant, es wird aber nicht näher beschrieben, was darunter zu verstehen ist.

TESTPLAN geht über die von der Kombination (EDDA,TUS) angebotenen Fähigkeiten in drei Punkten hinaus:

Erstens, im Grad der Formalisierung des Entwurfs. Das System TESTPLAN setzt voraus, daß über die formal beschriebenen Daten, Import- und Exportoperationen hinaus Assertionen (in formaler Sprache) vorhanden sind, die das statische Verhalten des Moduls beschreiben.

Zweitens, in der Art und Weise, wie Testfälle und Testdaten aus dem Entwurf abgeleitet werden. Die formal beschriebenen Assertionen dienen hierbei als Basis für eine systematische Testfallgenerierung und Testdatenerzeugung. Dieser Vorgang ist unter bestimmten Umständen (beispielsweise, wenn es sich bei den Assertionen um lineare Prädikate oder reguläre Ausdrücke handelt) automatisierbar.

Drittens, in der Möglichkeit zur (syntaktischen) Prüfung von Plausibilität und Konsistenz zwischen Entwurf und Testplan. Diese Prüfung ist möglich aufgrund der integrierten Sprache, die sowohl die Entwurfs- als auch die Testplanbeschreibungssprache umfaßt.

Abgrenzung gegenüber DAISTS
TESTPLAN unterscheidet sich von DAISTS hauptsächlich in drei Punkten:

Erstens, kann das System TESTPLAN automatisch Testfälle generieren, die erlauben, das Axiomensystem (in TESTPLAN mit Hilfe des ASSERT-Statements formuliert) auf Vollständigkeit zu überprüfen. D. h. Es können Testfälle auch für solche Teile generiert werden, die in der Entwurfsbeschreibung vergessen wurden.

Zweitens, in der Möglichkeit (System-) Anforderungen über den Entwurf bis auf Testplanebene zu verfolgen (vgl. 4.3.2.7.3).

Drittens, durch das zweistufige Assertionenkonzept. TESTPLAN sieht Assertionen auf Modul- und Prozedurebene vor. Die Assertionen der Modulebene entsprechen den Axiomen in DAISTS, während es sich auf Prozedurebene um Input-/Outputassertionen handelt, die in DAISTS keine Entsprechung haben.

Abgrenzung gegenüber SLAN-4
In /Comm 82/ wird ein erster Ansatz zur Erweiterung der Entwurfs- und Spezifikationssprache SLAN-4 um eine Testfallgenerierung vorgeschlagen. SLAN-4 wurde bisher bei IBM 'in mehreren Projekten mit positiven Ergebnissen für Spezifikation und Entwurf' eingesetzt. TESTPLAN hat mit SLAN-4 eine Reihe von Gemeinsamkeiten. Insbesondere die entwurfsspezifischen Teile von TESTPLAN und SLAN-4 sind in ihrer Grundstruktur ähnlich, obwohl beide unabhängig voneinander entwickelt wurden. Auch das zweistufige Assertionenkonzept ist beiden Systemen gemeinsam.

Für die Hauptunterschiede zwischen TESTPLAN und der um eine Strategie zur Testfallgenerierung erweiterten Sprache SLAN-4 sei auf die beiden ersten Unterscheidungspunkte bei DAISTS hingewiesen.

4.2 Das Verfahren TESTPLAN-V

In diesem Kapitel wird das Verfahren zur entwurfsbegleitenden Testplanung und Testdurchführung behandelt. Nachdem die Einbettung des Verfahrens TESTPLAN-V in den gesamten Softwareentwicklungszyklus kurz beschrieben ist, gehen wir im Unterkapitel 4.2.2 ausführlich auf das Kernstück des Verfahrens - die automatische Ableitung von Testfällen aus der Entwurfsbeschreibung - ein.

Wenn auch eine voll automatisierte Herleitung von Testfällen aus einer allgemeinen Entwurfsbeschreibung nicht möglich sein wird, so ist es doch denkbar, aus dem Entwurfsdokument interaktiv Testdaten und erwartete Testresultate abzuleiten und in Form eines Testplans dann wieder als Teil der Entwurfsbeschreibung abzuspeichern. Der Testplan wird in einer formalen Sprache beschrieben und ist in dieser Form vom Rechner direkt verarbeitbar. Der Testplan ist damit als Ergänzung zur Entwurfsbeschreibung anzusehen. Auf der Basis dieser formalen Testplanbeschreibung können die weiteren Schritte dann automatisch ablaufen. Insbesondere kann aus der Entwurfs- und Testplanbeschreibung ein Testrahmen generiert werden.

Bisherige Ansätze für die Beschreibung von Testplänen und die automatische Erzeugung von Testrahmen /Panz 78/ orientieren sich überwiegend am Programmcode (entweder auf Objektcode- oder Quellsprachenebene). In unserem Ansatz ist die Testplanbeschreibung in die Entwurfsbeschreibung integriert. Dadurch können Informationen aus der Entwurfsbeschreibung direkt übernommen werden, was einerseits den Schreibaufwand für den Testplan reduziert und andererseits auch Plausibilitätskontrollen und Konsistenzprüfungen zwischen Entwurfs- und Testplanbeschreibung ermöglicht.

Bevor das Verfahren zur entwurfsbegleitenden Testplanung und Testdurchführung detailliert wird, werden wir noch kurz die Einschränkungen präzisieren, denen dieses Verfahren unterliegt:

- Wir betrachten ausschließlich sequentielle Programme.

- Unsere Entwurfs- und Testobjekte sind abstrakte Datentypen (ADT'en), für die folgendes gilt. Wir nehmen an, daß das dynamische Verhalten eines ADT's mittels regulärer Ausdrücke über der Grundmenge der Zugriffsoperationen eines ADT's beschrieben ist. Dabei sind die Zugriffsoperationen parameterlose Prozeduraufrufe, was zunächst wie eine starke Einschränkung der durch das Verfahren verarbeitbaren Programmklasse aussieht. Diese Einschränkung ist letztlich jedoch nicht gravierend, da wir die durch den Parameterverzicht verlorene Information stattdessen in informeller Notation zu formulieren erlauben.

Während in Abb. 4.1 die entwurfsbegleitende Testplanung und Testdurchführung in Form eines Flußdiagramms dargestellt ist, wollen wir hier schrittweise das Verfahren TESTPLAN-V betrachten.

1. Unsere Entwurfsbeschreibung liegt in Form von abstrakten Datentypen (ADT'en) vor. Die Beschreibung besteht aus formalen und informellen Teilen, wobei sich die formalen Teile an die algebraische Spezifikation von ADT'en anlehnen. Wie in algebraischen Spezifikationen üblich, wird der Definitions- und Wertebereich jeder Zugriffsfunktion beschrieben. Synonym für Folgen von Zugriffsfunktionen werden wir den Begriff 'Operationsfolge' verwenden.

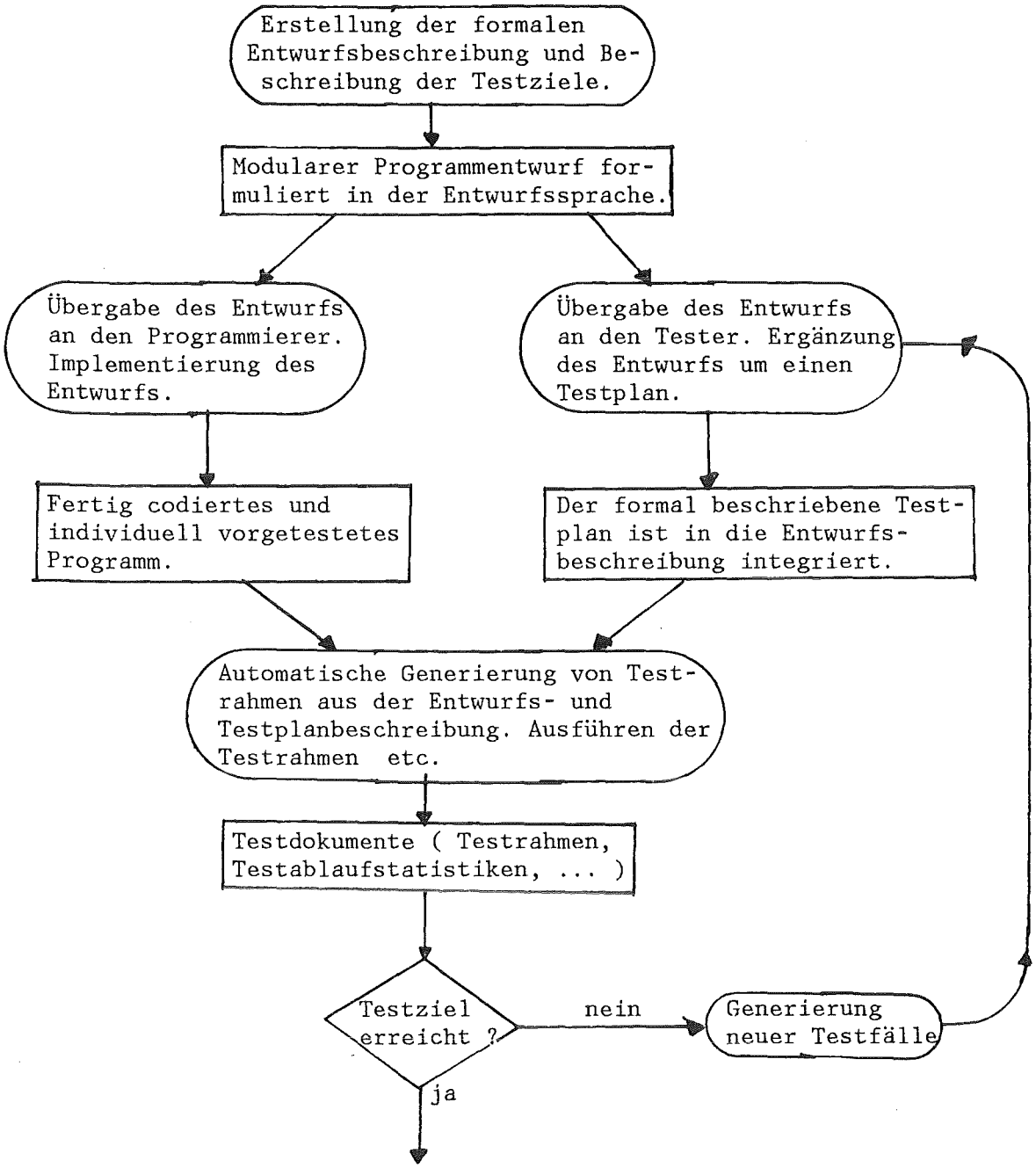
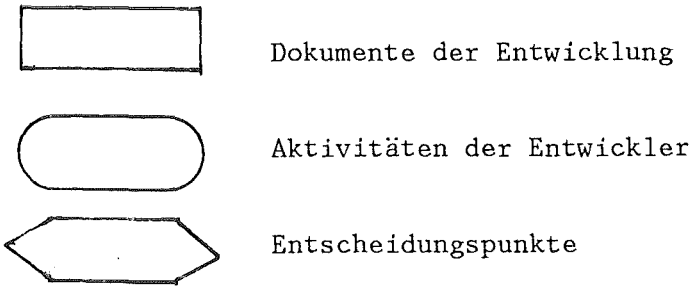


Abb. 4.1: Schema der entwicklungsbegleitenden Testplanung und Testdurchführung.

Legende:



Im Gegensatz zu der algebraischen Spezifikation lassen wir für die Beschreibung von Zusammenhängen zwischen den Funktionen eines ADT's auch informelle Sprachelemente zu. So beschreiben wir beispielsweise die Operationsfolgen selbst formal und die aus diesen Operationsfolgen resultierenden Effekte - d. h. die gewünschte Semantik der Operationsfolge - informell.

Bsp.: ASSERT op1 op2 = 'unzulässige Operationssequenz';

Operationsfolge — gewünschte Semantik dieser Operationsfolge

Die Entwurfsbeschreibung erfolgt in der Sprache TESTPLAN-S, wobei die informellen Teile der Entwurfsbeschreibung syntaktisch als 'String' dargestellt sind. Die Einhaltung der Sprachkonventionen wird durch den TESTPLAN-S-Übersetzer sichergestellt.
Die informellen Teile werden automatisch abgespeichert, weiterverarbeitet und später bei der Ausformulierung der generierten Testfälle in geeigneter Form interpretiert.

2. Assertionen sind spezielle Sprachelemente der Entwurfsbeschreibung, die das (dynamische) Verhalten eines ADT's beschreiben. Sie dienen als Grundlage einer automatischen Testfallgenerierung. Wie bereits einführend beschrieben, unterscheiden wir zwei Arten von Assertionen, wobei im folgenden hauptsächlich auf die Assertionen zur Beschreibung von Operationsfolgen eingegangen wird.
Wenn man den typischen Aufbau solcher Operationsfolgen analysiert, so stellt man fest, daß es sich um reguläre Ausdrücke handelt. Diese Erkenntnis ist wichtig, da für das Gebiet der regulären Ausdrücke eine gut entwickelte Theorie vorliegt, die wir uns für die Testfallgenerierung nutzbar machen werden.

   Die regulären Ausdrücke haben innerhalb der Entwurfsbeschreibung eine bestimmte Interpretation. Wenn wir von dieser Interpretation abstrahieren, können wir einige Transformationen und Folgerungen aus der Theorie der regulären Ausdrücke und Automaten anwenden. Daran anschließend ist eine Rückinterpretation der Ergebnisse auf die Entwurfsebene nötig.

3. Jeder ADT sei durch eine endliche Menge von Operationsfolgen - also ein System von regulären Ausdrücken - unter Berücksichtigung der eingangs erwähnten Einschränkungen vollständig im Entwurf beschrieben. Dieses System von regulären Ausdrücken wird in einen endlichen Automaten transformiert (vgl. das in Anhang B ausführlich beschriebene Verfahren). Die Zustandsübergänge des Automaten entsprechen jeweils einer einzelnen Operation. Folgen von Zustandsübergängen charakterisieren eine Operationsfolge. Die Rückinterpretation geschieht dann so, daß wir den Automaten als Testfallgenerator betrachten, aus dem automatisch eine endliche Menge von Testfällen ableitbar ist. Jeder Testfall ist eine Operationsfolge vom Anfangs- zu einem der Endzustände des Automaten. Die Endlichkeit der Testfallmenge wird durch verschiedene Testabbruchkriterien garantiert, auf die später noch eingegangen wird.

4. Die Testfälle (Operationsfolgen) müssen von einer Testperson noch in einer beliebigen Programmiersprache - wir haben in unserer Realisierung aus den in Kapitel 4.3 beschriebenen Gründen PASCAL gewählt - ausformuliert werden. Bei dieser interaktiven Vervollständigung der Testfälle kann auf bereits in der Entwurfsbeschreibung vorliegende Informationen zurückgegriffen werden (beispielsweise kann auf bereits eingeführte Datentypen Bezug genommen werden). Als Ergebnis liegt dann der Entwurf ergänzt um programmiersprachlich formulierte Testfälle vor. Die Gesamtheit dieser Testfälle bildet den Testplan.

5. Diese ergänzte Entwurfsbeschreibung wird syntaktisch und semantisch wiederum durch den TESTPLAN-S-Compiler überprüft und übersetzt. Ist die Übersetzung fehlerfrei, so ist sichergestellt, daß danach auch syntaktisch korrekte Testrahmen aus dieser Beschreibung automatisch generiert werden können.

7. Der generierte Testrahmen muß zusammen mit dem unabhängig davon erstellten Testobjekt gebunden und ausgeführt werden. Die Testergebnisse werden archiviert und stehen für spätere Auswertungen zur Verfügung. Vor der Ausführung ist vorgesehen, das Testobjekt automatisch zu instrumentieren. Für diese Instrumentierung sind Standardtechniken anwendbar, wie sie beispielsweise in /Huan 78/ beschrieben sind.

Zusammenfassend ist festzustellen, daß die ADT'en sowohl unsere Entwurfs- als auch unsere Testobjekte sind. Die Unterklasse von ADT'en, die sich mit dem vorgestellten Verfahren bearbeiten läßt, ist begrenzt durch die Verwendung regulärer Ausdrücke als Beschreibungsmittel für Folgen von Zugriffsoperationen. Reguläre Ausdrücke sind von ihrer Mächtigkeit her nur in der Lage, solche Folgen von Zugriffsoperationen zu beschreiben, die auf reduzible Graphen zurückführbar sind. Diese Einschränkung ist in der Praxis nicht schwerwiegend, da alle nach dem Prinzip der strukturierten Programmierung erstellten Programme auf reduzible Graphen führen.

4.2.1 Einbettung des Systems TESTPLAN in den Softwareentwicklungszyklus

Der Softwareentwicklungszyklus wurde u. a. ausführlich von Boehm /Boeh 76/ und Endres /Endr 80/ behandelt. In Abb. 4.2 halten wir uns weitgehend an Boehm, wobei wir die Testphasen etwas detaillierter auffächern und insbesondere zwischen Entwurf und Implementierung eine Testplanungsphase einschieben.

In dieser Abbildung sind auf der linken Seite die einzelnen Phasen der Softwareentwicklung und auf der rechten Seite die dabei anfallenden Entwicklungsdokumente dargestellt. In der Mitte sind die im Rahmen dieser Arbeit wichtigen Hilfsmittel beschrieben. Durch die senkrechten Balken wird angedeutet, in welchen Phasen diese Hilfsmittel einsetzbar sind. Die Hilfsmittel für den Entwurf und die Testplanung schließen unmittelbar aneinander an. Damit wird angedeutet, daß der Entwurf und die darauf aufbauende Testplanung nahtlos ineinander übergehen und sich ergänzen.

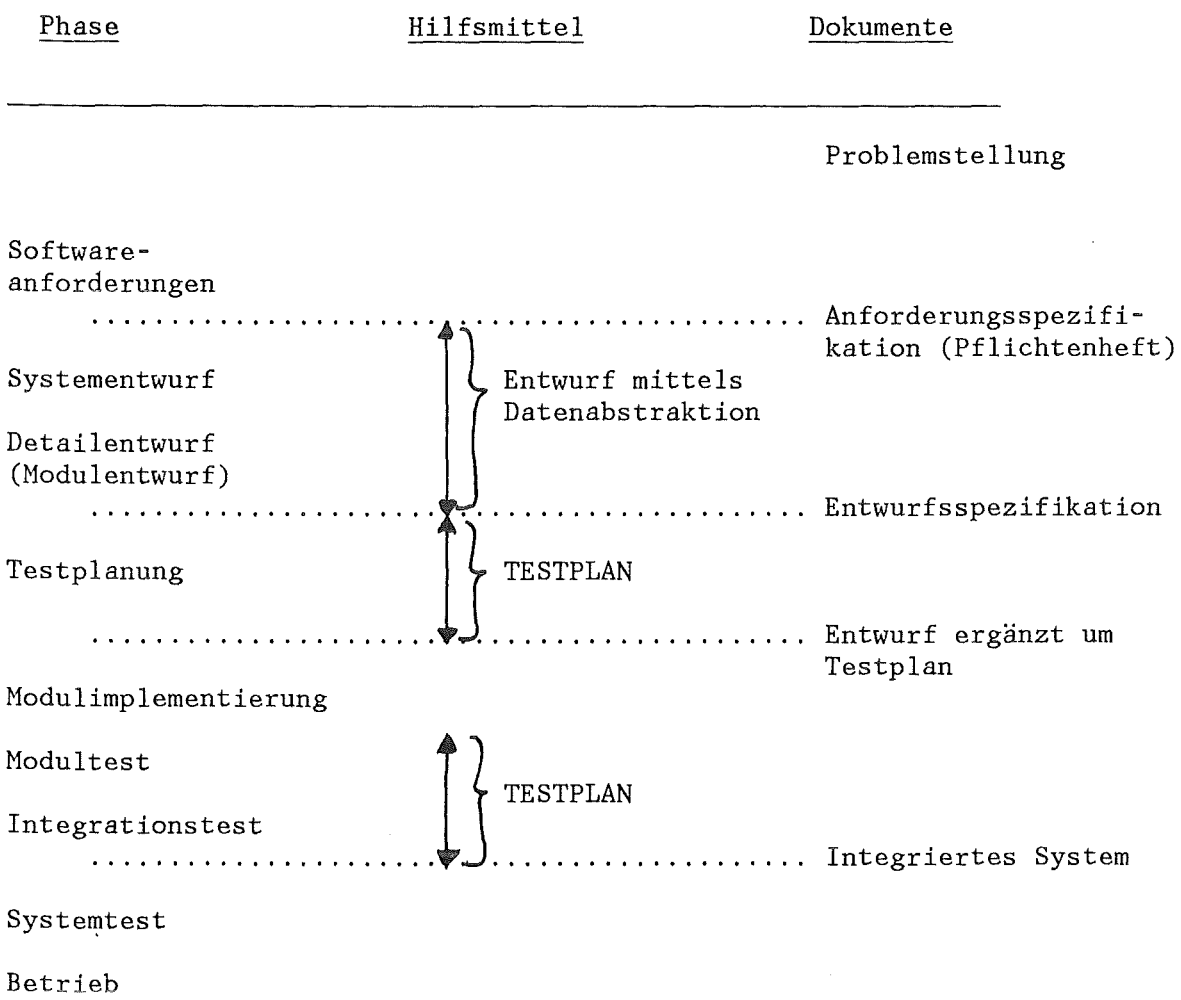


Abb. 4.2: Der Softwareentwicklungszyklus

Wir wollen hier nicht detailliert betrachten, wie der dynamische Prozeß des Entwurfs abläuft. Wir nehmen jedoch an, daß der Entwurf nach den Prinzipien der Modularisierung und Datenabstraktion durchgeführt wurde. Weiterhin gehen wir davon aus, daß die Entwurfsspezifikation am Ende des Detailentwurfs in einer formalen Darstellung vorliegt und damit eine gute Schnittstelle für das System TESTPLAN darstellt.

Das System TESTPLAN unterstützt den Modul- und Integrationstest. Dagegen ist es im Systemtest sinnvoll, die dafür nötigen Testdaten aus der Anforderungsspezifikation abzuleiten.

Aufgrund der formalen Beschreibung ist eine Archivierung der Testfälle leicht möglich, was in der Wartungsphase für Regressionstests vorteilhaft ausgenutzt werden kann. Werkzeuge, die die übrigen Phasen des Softwareentwicklungszyklus abdecken, werden hier nicht betrachtet.

4.2.1.1 Anforderungen an die Entwurfsbeschreibung

Da nicht jede Entwurfsbeschreibung gleichermaßen geeignet ist, um daraus einen Testplan abzuleiten, muß überlegt werden, welche Anforderungen an eine Entwurfsbeschreibung zu stellen sind, damit auf der Basis eines solchen Entwurfs Testdaten und Testresultate für einen 'Schwarzen-Kasten-Test' ableitbar sind. Die Erläuterungen innerhalb der Klammern zeigen auf, wo formal beschriebene Teile des Entwurfs für die Testplanung verwendbar sind:

- Die Entwurfsbeschreibung muß ein Konzept zur Modularisierung enthalten. (Die einzelnen Moduln sind dann die Objekte für den Schwarzen-Kasten-Test.)

- Das Verhalten eines Moduls muß durch Assertionen formulierbar sein. (Diese Assertionen sind in zweifacher Weise in der Testphase verwendbar: Zum einen können daraus spezielle Testdaten und Testergebnisse abgeleitet werden, zum anderen können diese Assertionen - sofern es sich um Eingabe-/Ausgabeassertionen von Zugriffsfunktionen handelt - zur Testausführungszeit in ausführbare Assertionen überführt werden.)

- Die Entwurfsbeschreibungssprache muß detaillierte Möglichkeiten zur Schnittstellenbeschreibung und Formulierung der Datenstruktur besitzen. (Aus diesen detaillierten Datenbeschreibungen, etwa in der Art von PASCAL, lassen sich datenorientiert Testfälle aus dem Entwurf ableiten.)

- Die Entwurfsbeschreibung muß um die Testplanbeschreibung erweiterbar sein. (Aus dieser integrierten Beschreibungsform ergeben sich Möglichkeiten zur Konsistenzprüfung und Plausibilitätskontrolle zwischen Entwurf und Testplan.)

4.2.1.2 Anforderungen an die Implementierung

Die in der Entwurfsbeschreibung festgelegten Schnittstellen müssen von der Implementierung streng eingehalten werden. Sollten trotzdem während der Implementierung Änderungen notwendig werden, so müssen die daraus resultierenden Modifikationen in der Entwurfsbeschreibung und anderen Dokumenten nach einer genau festzulegenden Änderungsprozedur ablaufen.

Weiterhin sind organisatorische Maßnahmen zu fordern, die sicherstellen, daß Implementierung und Test von unterschiedlichen und unabhängigen Personen durchgeführt werden. Die Implementierungssprache sollte entweder direkt als Schnittstellenbeschreibungssprache innerhalb der Entwurfssprache verwendet werden, oder es sollte eine einfache Abbildung zwischen der Schnittstellenbeschreibung des Entwurfs und der Implementierungssprache möglich sein.

4.2.1.3 Sonstige Anforderungen

Zu diesen zählen die interaktive Bedienbarkeit und die Existenz einer Datenbank.

Die interaktive Bedienbarkeit ist eine wesentliche Voraussetzung für die Benutzerfreundichkeit des Systems.
Ein Datenbanksystem vereinfacht die Verwaltung der umfangreichen Entwicklungsdokumente.

4.2.2 Automatische Ableitung von Testfällen aus der Entwurfsbeschreibung

Innerhalb des Verfahrens TESTPLAN-V kommt der automatischen Testfallgenerierung aus der Entwurfsbeschreibung eine zentrale Bedeutung zu. Die Entwurfsbeschreibung enthält Assertionen als Grundlage für die Testfallgenerierung.

Wir unterscheiden globale und lokale Assertionen, die auf
- regulären Ausdrücken bzw.
- logisch/arithmetischen Prädikaten

basieren. In diesem zweistufigen Assertionenkonzept beschreiben die globalen Assertionen das Zusammenspiel mehrerer Operationen eines Moduls, wogegen die lokalen Assertionen sich ausschließlich auf eine einzige Operation beziehen.

In den globalen Assertionen lassen sich Aufrufreihenfolgen für die Zugriffsfunktionen eines Moduls - ähnlich der Semantikbeschreibung in algebraischen Spezifikationen - ausdrücken. Diese Aufrufreihenfolgen sind immer dann durch reguläre Ausdrücke beschreibbar, wenn der zugrundeliegende Ablaufgraph reduzibel ist. (Für Programme, die nach den Regeln der strukturierten Programmierung erstellt wurden, ist diese Forderung nach einem reduziblen Ablaufgraphen immer erfüllt.) Wir nehmen an, daß alle globalen Assertionen zusammen die Aufrufbeziehungen vollständig beschreiben.

Die lokalen Assertionen geben das statische Ein-/Ausgabeverhalten jeder einzelnen Zugriffsfunktion durch logisch/arithmetische Prädikate wieder. Im Gegensatz zu den globalen Assertionen, muß man zur Formulierung der lokalen Assertionen über Kenntnisse der Modulinterna verfügen. Dies gilt insbesondere für die lokale Datenstruktur und den internen Aufbau der einzelnen Zugriffsfunktionen.

Für die automatische Testfallgenerierung wird je nach Assertionentyp folgende Vorgehensweise vorgeschlagen:

Die Testfallgenerierung aus globalen Assertionen stützt sich auf Algorithmen aus der Automatentheorie. Da die Anzahl der generierbaren Testfälle i. a. sehr groß oder gar unendlich werden kann, müssen Testabbruchkriterien eingeführt werden. In diesem Zusammenhang wird ein auf stochastischen Automaten basierendes Testabbruchkriterium vorgeschlagen, das es erlaubt, einen Zusammenhang zwischen der Anzahl der generierten Testfälle und der dadurch erreichten Zuverlässigkeit herzustellen.

Die Testfallgenerierung aus lokalen Assertionen basiert auf der Zerlegung der logisch/arithmetischen Prädikate in Elementarbedingungen. Aus diesen Elementarbedingungen werden mit Hilfe von Wahrheitstafeln systematisch Testfälle konstruiert.

Nach dieser prinzipiellen Darstellung der Vorgehensweise wollen wir zunächst vorausschicken, was wir unter regulären Ausdrücken bzw. logisch/arithmetischen Prädikaten verstehen. Im Anschluß daran gehen wir detailliert auf die Testfallgenerierungsalgorithmen ein.

Reguläre Ausdrücke

Reguläre Ausdrücke bestehen aus Operations-, Grund- und Klammersymbolen.
Die Grundsymbole sind Prozeduraufrufe, die innerhalb des ADT's definiert sein müssen. Die Operationssymbole sind

- die Konkatenation (ausgedrückt durch eine Folge von Aufrufen),
- die Auswahloperation ("|"),
- die unendliche Wiederholung ("*").

Die Klammersymbole dienen dazu, Prioritäten auszudrücken sowie den Gültigkeitsbereich von Operationssymbolen zu beschreiben.

Über diese gängige Darstellung regulärer Ausdrücke hinaus werden die Erweiterungen a) - c) eingeführt.

a) $a\ b^{+}\ c$ entspricht a b c, a b b c, ...
d. h. b wird mindestens einmal ausgeführt.

b) $a\ b^{4}\ c$ entspricht a b b b b c
d. h. endliche und feste Anzahl von Wiederholungen einer Operation (in unserem Beispiel 4-mal).

c) $a_{0}^{3}\ c$ entspricht c, a c, a a c, a a a c

d. h. Wiederholung der Operation a mit der Untergrenze 0 und der Obergrenze 3.

Reguläre Ausdrücke werden in der Entwurfsbeschreibung mit Hilfe der ASSERT-Anweisung (vgl. Kapitel 4.3.2.2) formuliert.

Logisch/arithmetische Prädikate

Die logisch/arithmetischen Prädikate bestehen aus:

Eingangsvariablen, Konstanten,
arithmetischen-, logischen- und Relationsoperatoren,
Existenz- und Allquantoren.

Bsp.: $\underbrace{(\ item < 20\ )\ \ OR\ (\ item = 100\ )}_{\text{Prädikat}};$

Logisch/arithmetische Prädikate werden in der Entwurfsbeschreibung mit Hilfe der INASSERT- und OUTASSERT-Anweisung (vgl. Kapitel 4.3.2.3) formuliert.

4.2.2.1 Automatische Ableitung von Testfällen aus regulären Ausdrücken

Reguläre Ausdrücke bilden ein sehr gutes Hilfsmittel während der Spezifikation und des Entwurfs, weil damit einerseits die umgangssprachliche Bedeutung ausgedrückt werden kann und andererseits bereits eine sehr formale Beschreibung vorliegt, die eine automatische Weiterverarbeitung ermöglicht. Um die enge Beziehung zwischen Umgangssprache und formaler Sprache zu illustrieren, wollen wir ein Beispiel aus /Shaw 79/ anführen.

Beispiel:

Die erlaubten Operationen auf Objekten des Moduls 'Datei' können durch den regulären Ausdruck

open (read ((update write) | print))* close

beschrieben werden, die die Sequenzen

[oc, oruwc,orpc, oruwrpc, ...] repräsentieren.

(o steht für open, r steht für read, ...).

Die beabsichtigte Interpretation ist, daß das Open-Kommando gefolgt wird von Null oder mehr (Read update write)'s und/oder (read/prints)'s, und schließlich durch das Close-Kommando abgeschlossen wird.

Wir nehmen an, daß alle regulären Ausdrücke zusammen das gewünschte Systemverhalten vollständig beschreiben. Außerdem gehen wir davon aus, daß die regulären Ausdrücke jeweils im Startzustand des Systems beginnen, was insofern auch sinnvoll ist, da man, um eine Operation testen zu können, zunächst die gesamte Initialisierungssequenz durchlaufen muß.

In diesem Kapitel beschreiben wir zunächst die Konstruktion eines deterministischen endlichen Automaten aus den regulären Ausdrücken der Entwurfsbeschreibung. Da die aus einem endlichen Automaten ableitbare Testfallmenge im allgemeinen unendlich ist, müssen Testabbruchkriterien angegeben werden. Wir befassen uns hier hauptsächlich mit einem auf Wahrscheinlichkeiten basierenden Testabbruchkriterium.

4.2.2.1.1 Konstruktion eines deterministischen Automaten

Der Ansatz zur automatischen Testfallgenerierung beruht nun darauf, daß wir für jeden regulären Ausdruck einen äquivalenten endlichen Automaten konstruieren und diese einzelnen Automaten dann zu einem großen Automaten zusammenbauen.
Der daraus resultierende Automat beschreibt alle in den regulären Ausdrücken formulierten Operationsfolgen. Dieser Automat läßt sich durch Einfügen eines Zustands 'undefiniert' zu einem vollständigen Automaten ergänzen. Alle Operationsfolgen, die im Zustand 'undefiniert' enden, sind in der Entwurfsbeschreibung nicht enthalten.
Dieser Konstruktionsalgorithmus ist detailliert in Anhang B beschrieben.

Wenn wir den so gewonnen Automaten als Testfallgenerator betrachten, so können wir eine endliche Menge von Testfällen - d. h. Operationsfolgen vom Startzustand zu den Endzuständen des Automaten - erzeugen. Für das obige Beispiel des Moduls 'Datei' würden unter anderem folgende 2 Testfälle generiert werden:

1) o r u w c
2) o u

Im ersten Fall liegt eine zulässige Operationsfolge vor, während der zweite Fall im zugrundeliegenden regulären Ausdruck nicht beschrieben ist. Dieses Beispiel ist insofern einfach, da es nur von einem einzigen regulären Ausdruck ausgeht.

Da im allgemeinen Fall eine unendliche Menge von Testfällen ableitbar ist, müssen Abbruchkriterien für die Testfallgenerierung angegeben werden.

4.2.2.1.2 Testabbruchkriterien

Aus der Literatur ist eine Reihe von Testabbruchkriterien (/Mill 79/, /Mill 81/) bekannt. Zu den einfachsten - und am meisten angewendeten - zählt die mindestens einmalige Ausführung einer Anweisung oder einer Verzweigung im Programm. Wendet man beispielsweise diese beiden letztgenannten Kriterien auf unseren im vorigen Kapitel konstruierten Automaten an, so kann man sich folgende Testabbruchkriterien vorstellen:

- Alle Endzustände des Automaten müssen mindestens einmal durch Testfälle erreicht werden.

- Alle Kanten des Automaten (Operationen) müssen mindestens einmal ausgeführt worden sein.

Diese beiden Kriterien sind linear mit der Anzahl der Endzustände bzw. mit der Anzahl der Kanten des Automaten. Diese Linearität bewirkt, daß keine Unterschiede zwischen kritischen und unkritischen Programmteilen in Bezug auf die Intensität des Testens gemacht werden. D. h., das Testobjekt wird vollkommen abstrakt und ohne Bezug auf die Semantik betrachtet.

4.2.2.1.3 Testabbruchkriterien in stochastischen Automaten

Wir wollen im folgenden ein Testabbruchkriterium vorschlagen, das Bezug auf die Semantik und das Anforderungsspektrum der zu testenden Software nimmt. Unter dem Anforderungsspektrum verstehen wir die Verteilung der tatsächlichen Eingangsdatenkombinationen während des Betriebs der Software. Zwei wichtige Punkte, die das Anforderungsspektrum charakterisieren, sind zum einen die Wahrscheinlichkeit für den Aufruf einer Operation in einem bestimmten Systemzustand, zum anderen der Risikofaktor, der mit dem Aufruf einer Operation verbunden ist.

Wir gehen aus von

- einem vollständigen endlichen Automaten mit ausgezeichnetem Startzustand und bekannten Endzuständen (vgl. Algorithmus GEN_TESTFALLAUTOMAT aus Anhang B).

- Wahrscheinlichkeiten, die mit den Zustandsübergängen des Automaten (Operationen) assoziiert sind und vom Benutzer interaktiv einzugeben sind.

Diese beiden Eigenschaften charakterisieren bei geeigneter Definition der Wahrscheinlichkeiten einen stochastischen Automaten /Göss 75/.

Die Anzahl und Länge der Zufallsfolgen (Testfälle), die sich aus einem stochastischen Automaten ableiten lassen, ist im allgemeinen unendlich. Wenn man jedoch eine Grenzwahrscheinlichkeit für jede Zufallsfolge vorgibt, erhält man eine in Anzahl und Länge endliche Menge von Zufallsfolgen.

Jede Zufallsfolge setzt sich aus einer Reihe von Zustandsübergängen zusammen, die mit Wahrscheinlichkeiten belegt sind. Wir bilden das Produkt dieser Wahrscheinlichkeiten und brechen dann ab, wenn entweder die Produktwahrscheinlichkeit kleiner als die Grenzwahrscheinlichkeit ist, oder aber ein Endzustand des Automaten erreicht wurde.

Bevor wir den Algorithmus zur Bestimmung des Testabbruchs angeben, schicken wir noch einige Betrachtungen voraus, die erläutern sollen, wie man von einem deterministischen endlichen Automaten auf einen stochastischen Automaten kommt und wie die dabei verwendeten Wahrscheinlichkeiten zu interpretieren sind.

Wir gehen aus von einem Automaten mit einer endlichen Zustandsmenge $Z = \{ z_1, .. , z_n \}$.

Mit jedem Zustandsübergang (z_i --> z_j) assoziieren wir zwei Wahrscheinlichkeiten P_{ij} und Q_{ij}.

Die Übergangswahrscheinlichkeiten $P(z_i \text{ --> } z_j) = P_{ij}$ werden vor Ausführung des Algorithmus durch den Benutzer interaktiv bereitgestellt. (Man beachte, daß die Übergangswahrscheinlichkeit für ein und dieselbe Operation in verschiedenen Zuständen durchaus unterschiedlich sein kann).

Für alle Zustände i des Automaten gilt :

$$\sum_j P_{ij} = 1 . \qquad j \in \{ \text{Menge der Zustandsübergänge, die vom Zustand i ausgehen} \}$$

Die Risikowahrscheinlichkeiten $Q(Z_i \text{ ---> } Z_j) = Q_{ij}$ charakterisieren die Fehleranfälligkeit der einzelnen Operationen. Diese Wahrscheinlichkeiten lassen sich im Gegensatz zu den Übergangswahrscheinlichkeiten automatisch aus der Entwurfsbeschreibung ableiten. Zur Berechnung der Komplexität einer Modul- oder Prozedurschnittstelle können Entwurfsmetriken (/Gilb 76/, /Keut 81/) herangezogen werden. Wir nehmen an, daß die Komplexität einer Operation direkt proportional zu deren Fehleranfälligkeit ('Risiko') ist.

Ein Modul enthalte m Operationen . Für die Risikowahrscheinlichkeiten $\{ r_1, \ldots r_m \}$ dieser Operationen gilt:

$$\sum_i r_i = 1 . \qquad i \in \{ 1, \ldots ,m \}$$

An allen Zustandsübergängen (i, j) des Automaten, an denen eine Operation k aufgerufen wird, werden folgende Zuweisungen vorgenommen:

$$Q_{ij} := k \quad \text{mit} \quad k \in \{ r_1, \ldots r_m \}$$

Damit sind die Wahrscheinlichkeiten P_{ij} und Q_{ij} eingeführt, mit deren Hilfe wir den Algorithmus zur Bestimmung des Testabbruchs angeben können. Dieser Algorithmus 'TESTLIMIT' besteht aus fünf Schritten:

1. Wir starten mit einem vollständigen endlichen Automaten, wie er nach Schritt fünf des Algorithmus 'GEN_TESTFALLAUTOMAT' vorliegt.

2. Wir assoziieren mit den Zustandsübergängen (i, j) die Wahrscheinlichkeiten P_{ij} und Q_{ij} .

3. Für alle $i,j \in Z$ gilt dann: $R_{ij} := P_{ij} * Q_{ij}$;
   Die entstehenden Wahrscheinlichkeiten R_{ij} werden in jedem Zustand i auf 1 normiert, sodaß man die Wahrscheinlichkeit R_{ij}' erhält und für alle Zustände i gilt:

   $$\sum_j R_{ij}' = 1.$$

   $j \in \{$ Menge der Zustandsübergänge, die vom Zustand i ausgehen $\}$

   Als Ergebnis liegt jetzt ein stochastischer Automat vor.

4. Dieser stochastische Automat läßt sich in Form eines Baumes 'abrollen'. Von der Wurzel ausgehend kann man die Produktwahrscheinlichkeiten einer Zufallsfolge (Pfad) sukzessive berechnen.

5. Wir brechen die Pfadgenerierung ab, wenn entweder alle Pfade in einem Endzustand sind, oder aber die Produktwahrscheinlichkeiten eines jeden Pfades kleiner einer vorgegebenen Grenzwahrscheinlichkeit sind.

Somit liegt am Ende dieses Algorithmus eine endliche Menge von Testfällen (Operationsfolgen) folgender Bauart vor:

```
a b c d b     (* a,b,c ... sind Operationen eines Moduls *)
a d d
:
```

Im weiteren Verlauf muß jeder dieser Testfälle manuell in eine Testprozedur (z. B. in PASCAL) umgesetzt werden, die dann in den Testplan aufgenommen werden kann. Bei der Erstellung der Testprozeduren kann auch auf Informationen zurückgegriffen werden, die im informellen Teil der regulären Ausdrücke (vgl. S. 36) beschrieben sind.

Der Übergang von einem deterministischen auf einen stochastischen Automaten hat den Effekt, daß man einen quantitativen Zusammenhang zwischen dem Testabbruchkriterium und der erreichten Zuverlässigkeit herstellen kann. Durch eine Variation der Grenzwahrscheinlichkeit (vgl. Schritt 5 im Algorithmus TESTLIMIT) läßt sich die zu generierende Zahl von Testfällen der jeweils geforderten Zuverlässigkeit anpassen. Eine Verkleinerung der Grenzwahrscheinlichkeit resultiert in einer größeren Menge von Testfällen.

Nehmen wir an, wir hätten aufgrund einer Grenzwahrscheinlichkeit 'Gr' insgesamt 'n' Testfälle generiert und eine Ausführung dieser Testfälle hätte 'nf' fehlerhafte Testläufe erbracht, so kann man diese Daten unmittelbar in die Formel für die Zuverlässigkeit (R = 1 - (nf/n)) einsetzen. Ist die geforderte Zuverlässigkeit noch nicht erreicht, muß der Anpassungsprozeß der Grenzwahrscheinlichkeit wiederholt werden. Dies kann - je nach Fehlerhaftigkeit der neu hinzugekommenen Testfälle - zu einer Verbesserung oder Verschlechterung der Maßzahl für die erreichte Zuverlässigkeit führen.

4.2.2.2 Automatische Ableitung von Testfällen aus log./arithm. Prädikaten

Ein gängiges Verfahren, um in der Anforderungsspezifikation von seiten des Auftraggebers Funktionen zu definieren, ist die paarweise Beschreibung von Eingangsbedingungen und erwarteten Resultaten. Diese Form der Beschreibung kann ohne große Modifikationen in die Entwurfsspezifikation übernommen werden. Innerhalb einer formalen Entwurfsbeschreibung lassen sich die Eingangsbedingungen und erwarteten Resultate einer Funktion in Form von logisch/arithmetischen Prädikaten beschreiben.

Beispiel: Es ist ein Sortierprogramm zu schreiben. Die zu sortierenden Objekte sind ganze Zahlen zwischen -10 und 1000. Die unsortierte und die sortierte Liste stehen jeweils in der Reihung TYPE a = ARRAY (. 1 .. 100 .) OF INTEGER.

Das logisch/arithmetische Prädikat für die Eingangsbedingung lautet:

```
(-10 <= a(i) <= 1000) FOR ALL i = 1 TO 100 .
```

Das logisch/arithmetische Prädikat für die Ausgangsbedingung lautet:

```
( a(i) <= a(i + 1) ) FOR ALL i = 1 TO 99 .
```

Wir nehmen an, daß für jede Zugriffsfunktion eines ADT's eine Beschreibung des Ein-/Ausgabeverhaltens mittels logisch/arithmetischer Prädikate vorliegt.

Eine Testfallgenerierung auf der Basis dieser Prädikate läuft in drei Schritten ab:

- Vereinfachen des logisch/arithmetischen Prädikats.
- Zerlegung eines Prädikats in seine Elementarbedingungen.
- Systematische Synthese von Testfällen aus den Elementarbedingungen mit Hilfe von Wahrheitstafeln.

Als weiteren Schritt, der über die Testfallgenerierung hinausgeht, ist daran eine

- automatische Testdatenerzeugung

anzuschließen.

4.2.2.2.1 Vereinfachung des logisch/arithmetischen Prädikats

Da die aus dem Entwurf abgeleiteten logisch/arithmetischen Prädikate redundante und eventuell widersprüchliche Informationen enthalten, liegt eine Vereinfachung dieser Prädikate nahe. Hierfür kann man Techniken anwenden, wie sie auch in Systemen zur symbolischen Ausführung /Boye 75/ Anwendung finden.

Da manuell eingegebene Prädikate im allgemeinen bereits einfach sind, gilt unser Hauptinteresse dem Nachweis der Widerspruchsfreiheit innerhalb eines Prädikats.

Aufgrund von Entscheidbarkeitsproblemen /Loos 81/ gibt es nicht für alle Klassen von Prädikaten ein mechanisches Verfahren zur Vereinfachung. Diese Einschränkung gilt insbesondere auch für die hier betrachteten logisch/arithmetischen Prädikate (bestehend aus INTEGER, REAL,... - Konstanten und - Variablen). Zur Vereinfachung existiert eine Reihe algebraischer Manipulationssysteme (/Clar 81/, /Rich 78/), die meistens auf heuristischen Umformungsregeln basieren und darüberhinaus oftmals auch noch einer interaktiven Unterstützung durch den Benutzer bedürfen.

Die Anwendung solcher Heuristiken oder auch kompletter Manipulationssysteme führt auf zwei mögliche Ergebnisse:

- Eine Vereinfachung ist nicht möglich.

- Eine Vereinfachung ist möglich.
  . Man erhält als Ergebnis ein einfacheres Prädikat.
  (Bsp.: A < 10 AND A = 5 wird vereinfacht zu A = 5)
  . In günstigen Fällen kann das Prädikat so vereinfacht werden, daß die Nichterfüllbarkeit nachgewiesen wird.

4.2.2.2.2 Zerlegung eines Prädikats in seine Elementarbedingungen

Das Ergebnis der Vereinfachung ist ebenso wie das Ausgangsprodukt wiederum ein logisch/arithmetisches Prädikat, das nun noch in seine Elementarbedingungen zerlegt wird. Wir wollen im folgenden annehmen, daß das Prädikat als Baum dargestellt ist. Dann bezeichnen wir als Elementarbedingungen diejenigen Unterbäume, deren Wurzel aus einem logischen oder Relationsoperator besteht und die neben dieser Wurzel keine weiteren logischen bzw. Relationsoperatoren enthalten. Die Elementarbedingungen sind die Atome, aus denen die Testfälle zusammengebaut werden. Für ein Beispiel sei auf Kap. 4.2.2.2.5 verwiesen.

4.2.2.2.3 Systematische Synthese von Testfällen

Durch Aufstellen einer Wahrheitstafel lassen sich aus den Elementarbedingungen systematisch und vollständig alle Testfälle konstruieren. Desweiteren können die Verknüpfungen des Prädikats, die geringere Priorität haben (und im Baum weiter oben stehen), ebenfalls in die Wahrheitstafel aufgenommen werden.

Jede Zeile der Wahrheitstafel skizziert einen Testfall, für den vom Programmierer noch explizit die Testeingaben und die erwarteten Testresultate formuliert werden müssen (vgl. TEST(1),...).

Aus der Beziehung

$$\text{Anzahl der Testfälle} = 2^{\text{Anzahl der Elementarbedingungen}}$$

kann man entnehmen, daß dieses Verfahren nur bei einer relativ kleinen Zahl von Elementarbedingungen ohne Einschränkungen angewandt werden kann.

Da wir als ersten Schritt in unserem Verfahren bereits eine Vereinfachung durchgeführt haben, können wir an dieser Stelle annehmen, daß damit auch die Anzahl der Elementarbedingungen (fast) minimal ist.

Darüberhinaus ist es denkbar, daß der Benutzer aus der restlichen Entwurfsbeschreibung noch 'Nebenbedingungen' ableiten kann, die vielleicht nicht oder nur umständlich im Prädikat beschrieben werden können. Diese Nebenbedingungen können vom Benutzer zur interaktiven Reduktion der Testfälle benutzt werden, so daß in der obigen Formel das '='-Zeichen durch ein '<='-Zeichen ersetzt werden kann.

4.2.2.2.4 Testdatenerzeugung aus logisch/arithmetischen Prädikaten

Die logisch/arithmetischen Prädikate dienen als Ausgangspunkt für die Testdatenerzeugung. In diesen Prädikaten können Variablen unterschiedlicher Datentypen über Operatoren miteinander verknüpft sein. Je nachdem, welche Datentypen in einem Prädikat vertreten sind, ist eine unterschiedliche Klassifikation denkbar. In /Davi 73/ wird festgestellt, daß es kein allgemeingültiges, mechanisches Lösungsverfahren für alle Klassen von Prädikaten gibt. Wenn wir jedoch die Prädikate als (Un-)Gleichungssysteme allgemeinster Art auffassen, existieren für Prädikate mit bestimmten Eigenschaften algorithmische Lösungsverfahren.

Beispielsweise können Verfahren der linearen Optimierung für lineare Prädikate mit INTEGER- Variablen angewandt werden /Clar 75/. Erweiterungen solcher Algorithmen auf lineare Systeme mit REAL-Variablen sind ebenfalls bekannt.

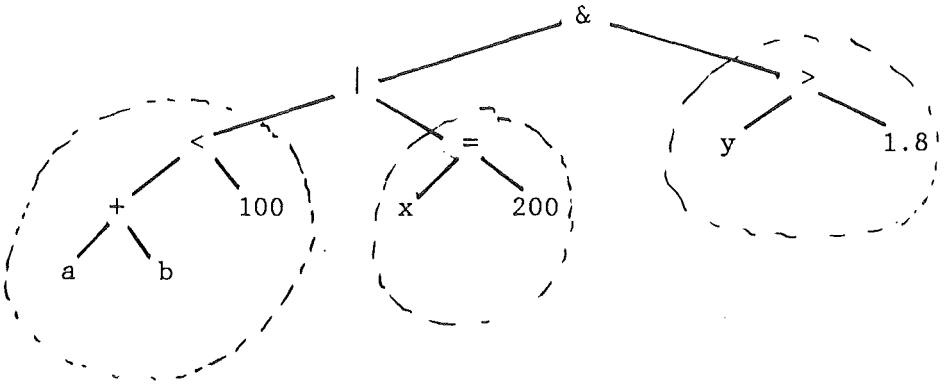
In /Elsp 74/ wird mit dem 'Hill Climbing Algorithmus' ein Verfahren benutzt, das aus den (Un-)Gleichungen eine Funktion konstruiert, die anschließend minimiert wird. Das Verfahren läßt alle Arten von (Un-)Gleichungen und alle einfachen Variablentypen zu, jedoch ist es nicht immer in der Lage, selbständig zu einer Lösung zu kommen und braucht eventuell die Hilfe des Benutzers.

4.2.2.2.5 Beispiel für eine Zerlegung in Elementarbedingungen und für eine Synthese der Testfälle mittels Wahrheitstafeln

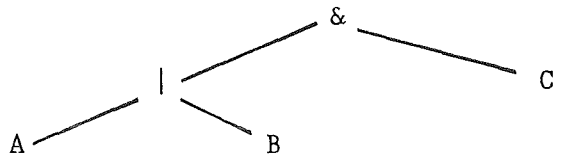
Es sei folgendes Prädikat gegeben

(a + b < 100 | x = 200) & y > 1.8

Dieser Ausdruck sieht als Baum formuliert so aus:



Die Elementarbedingungen sind jeweils durch gestrichelte Kreise umrandet.
Aus dem obigen Baum kann man einen vereinfachten abstrakten Baum gewinnen, indem man die Elementarbedingungen durch die Symbole A,B,C ersetzt.



mit A: a + b < 100; B: x = 100; C: y > 1.8 .

Für die Elemente dieses abstrakten Baumes kann man eine Wahrheitstafel aufstellen, in der dann systematisch alle Permutationen durchgespielt werden.

Fall	A	B	C	A | B	&	
1	0	0	0	0	0	Fehlerausgang
2	0	0	L	0	0	Fehlerausgang
3	0	L	0	L	0	Fehlerausgang
4	0	L	L	L	L	reguläre Ausgabe
5	L	0	0	L	0	Fehlerausgang
6	L	0	L	L	L	reguläre Ausgabe
7	L	L	0	L	0	Fehlerausgang
8	L	L	L	L	L	reguläre Ausgabe

A = 0 bedeutet : a + b >= 100
B = 0 bedeutet : x =/ 200
C = 0 bedeutet : y <= 1.8

4.2.2.3 Bewertung der beiden Ansätze zur automatischen Ableitung von Testfällen

Wir geben hier einen tabellarischen Überblick über die in den beiden letzten Kapiteln erarbeiteten Ergebnisse. Dabei legen wir besonderen Wert darauf, wie stark die einzelnen Verfahrensschritte automatisiert werden können.

Generier-verfahren	Verfahrensschritte	Möglicher Automatisierungsgrad
Testfälle aus regulären Ausdrücken	Erstellung des Testfallautomaten	voll
	einfaches Testabbruchkriterium	voll
	Testabbruchkriterium mit Wahrscheinlichkeiten	eingeschränkt
	Testdatenerzeugung	manuell
Testfälle aus logisch/arithmetischen Prädikaten	Vereinfachung des logisch/arithmetischen Prädikats	eingeschränkt
	Zerlegung in Elementarbedingungen	voll
	Synthese der Testfälle	voll
	Testdatenerzeugung	eingeschränkt

Im folgenden wollen wir einige mit diesen Verfahren zusammenhängende Probleme und Eigenschaften besonders hervorheben.

- Betrachtet man das auf Wahrscheinlichkeiten basierende Testabbruchkriterium, so erweist es sich als vorteilhaft, daß - im Gegensatz zu rein syntaktischen Abbruchkriterien - auch die Semantik der Operationen und Operationsfolgen eine stärkere Rolle spielt. Diese Semantik wird in den stochastischen Automaten in Form von Übergangs- bzw. Risikowahrscheinlichkeiten eingebracht. Durch eine Verschiebung der Grenzwahrscheinlichkeit kann man die Anzahl der zu generierenden Testfälle leicht variieren und damit die Intensität des Testens bestimmen. Damit lassen sich dann, je nach Kritikalität der Anwendung, maßgeschneiderte Testabbruchkriterien vorgeben.

- Man entnimmt aus der obigen Tabelle, daß eine vollständige Automatisierung bis zur Testfallgenerierung auf jeden Fall möglich ist. Will man aber an die automatische Testfallgenerierung auch noch eine automatische Testdatenerzeugung anschließen, so ist dies im allgemeinen Fall nicht mehr lösbar. Man kann dies nur für eine sehr eingeschränkte Klasse von Prädikaten (z. B. lineare Prädikate mit INTEGER-Variablen) mit vernünftigem Aufwand realisieren.
  Damit ist an dieser Stelle auf jeden Fall eine interaktive Unterstützung durch den Benutzer nötig. Es hat sich auf Gebieten mit ähnlicher Problemstellung (vgl. Programmbeweissysteme) gezeigt, daß auch durch einen großen Programmier- und Rechenzeitaufwand, etwa bei der Vereinfachung allgemeiner Prädikatklassen, keine entscheidende Verbesserung mehr erzielt werden kann. Deshalb begnügt man sich sinnvollerweise damit, einige einfache Regeln und Heuristiken zu programmieren.

4.3 Die Sprache TESTPLAN-S

Aufgrund der vorausgehenden Ausführungen ist klar, daß man zur Formulierung des Softwareentwurfs und der darin zu integrierenden Testpläne eine Sprache benötigt. Die im Verfahren TESTPLAN-V vorgeschlagene Vorgehensweise erfordert eine formale Beschreibung des Entwurfs. Diese formale Beschreibung wird von der Sprache TESTPLAN-S geleistet. Im Gegensatz zu graphischen Sprachen basiert TESTPLAN-S ausschließlich auf einer textuellen Darstellungsform. Hierzu wird der bei Programmiersprachen übliche Zeichensatz verwendet.

Als erste Idee für eine solche Sprache denkt man sofort an eine existierende Programmiersprache mit Erweiterungen. Aufgrund der umfangreichen Anforderungen (wie Modulkonzept, Assertionen, Testplanbeschreibung, etc.) scheidet eine solche einfache Erweiterung, beispielsweise über Precompiler, aus. Anstatt dessen definieren wir eine eigene Sprache mit einem eigenen Übersetzer. Da es andererseits aber nicht sinnvoll ist, bereits vorhandene Dinge nochmal zu erfinden, wählen wir aus folgenden Überlegungen PASCAL als Kern unserer Sprache TESTPLAN-S:

> Obwohl wir anstreben, den Entwurf weitgehend implementierungsunabhängig zu halten, muß man letztlich doch einen Übergang von der Entwurfs- zur Implementierungssprache herstellen. Als Implementierungssprache für das Testobjekt ist in unserem Fall PASCAL vorgesehen. Da wir Teile der Entwurfsbeschreibung zur Generierung von Testrahmen ausnutzen wollen, liegt es nahe, im Entwurf auch schon so weit als möglich PASCAL-Notation zu verwenden. Der Vorteil einer solchen Vorgehensweise liegt darin, daß man ein einheitliches System, bestehend aus PASCAL-Testrahmenprogramm und PASCAL-Testobjekt, erhält. Innerhalb der Entwurfssprache lassen wir den gesamten PASCAL-Sprachumfang zu.
>
> Da wir die abstrakten Datentypen syntaktisch wie einen Definitionsmodul in MODULA-2 behandeln, ergibt sich aufgrund der Verwandtschaft dieser Sprache mit PASCAL eine gute Kompatibilität.
>
> Die Wahl von PASCAL als Implementierungssprache bedeutet zwar eine Einschränkung. Eine Übertragung des vorgeschlagenen Verfahrens auf andere Implementierungssprachen ist aber ohne weiteres vorstellbar.

Aus der Abbildung 4.3 ersieht man, daß ein Modul aus Datendefinitionen, Prozeduren (Zugriffsfunktionen) und Attributen (Moduleigenschaften) besteht. Die Moduln stehen jeweils auf derselben Ebene und lassen sich nicht schachteln. Dies ist keine Einschränkung, da jede beliebige Beziehung zwischen Moduln über die Import- bzw. Exportoperationen (vgl. Kap. 4.3.2.1) realisierbar ist.

Um die Übersichtlichkeit zu gewährleisten, wurde der schematische Aufbau der Entwurfsbeschreibung möglichst einfach gehalten.

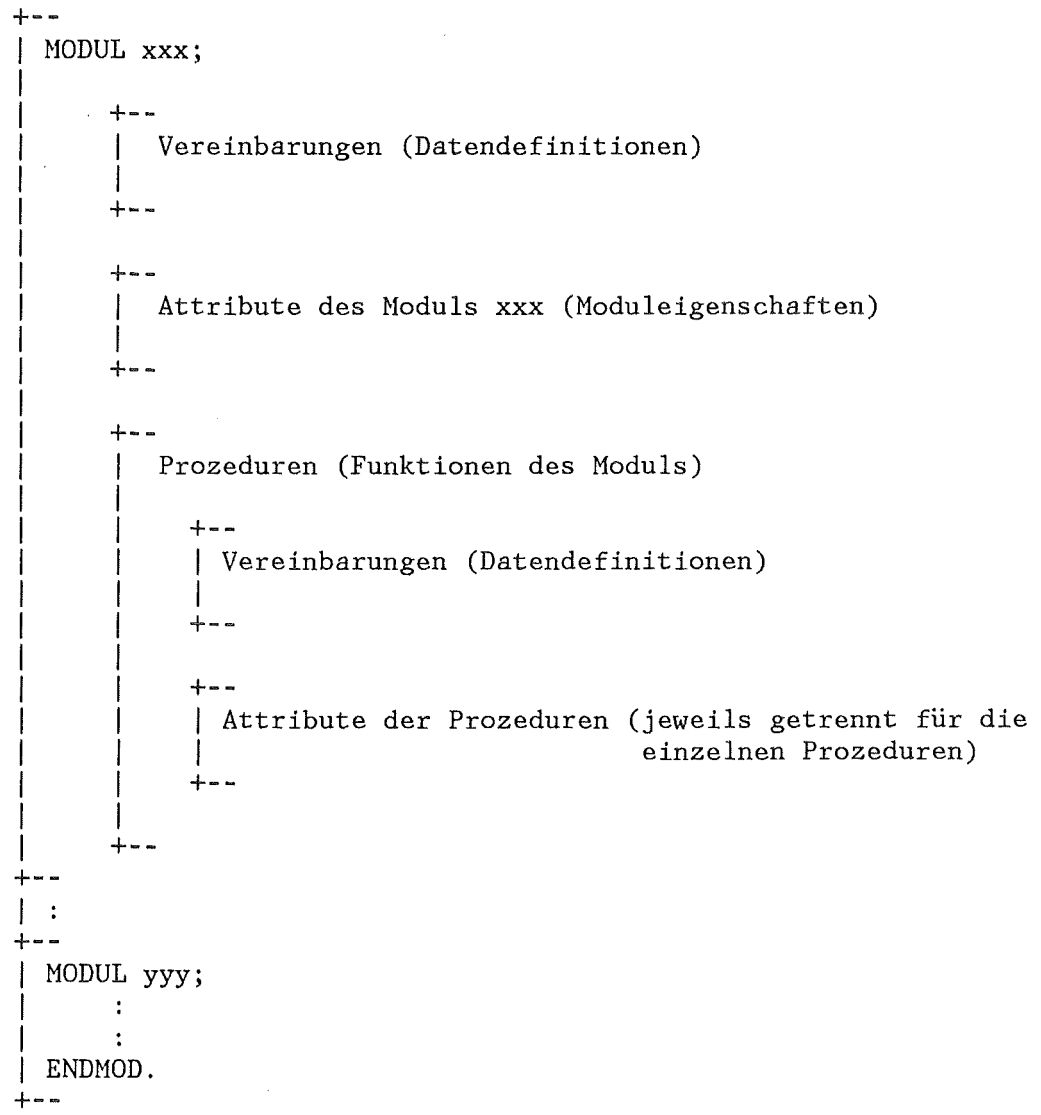


Abb. 4.3: Schema einer Entwurfsbeschreibung in TESTPLAN-S.

Programmiersprachen werden im allgemeinen durch ihre Syntax- und Semantikbeschreibung charakterisiert. Dieselbe Einteilung wählen wir auch für die Beschreibung der Sprache TESTPLAN-S.

Die Syntax wird in Kapitel 8 in erweiterter Backus-Naur-Form dargestellt. Die Semantik der einzelnen Sprachanweisungen wird in Kapiteln 4.3.2 und 4.3.3 erläutert. Darüberhinaus wurde eine formale Semantikbeschreibung in Form einer attributierten Grammatik /Gmei 82/ erstellt.

4.3.1 Sprachkonzepte

Der erste Eindruck, den man gewöhnlich von einer Sprache (sei es einer Entwurfs- oder einer beliebigen Programmiersprache) gewinnt, bezieht sich meistens auf die syntaktische Form. Wichtiger und schwerwiegender sind jedoch die Sprachkonzepte, die sich hinter dieser äußeren Darstellungsform verbergen. Zu den in TESTPLAN-S enthaltenen Konzepten zählen insbesondere :

- Die Verwendung von Moduln im Sinne der abstrakten Datentypen;
- Eine Mischung aus struktureller und relationaler Darstellung;
- Assertionen;
- Integration der Entwurfs-, Testplanungs- und Kommandosprache in einer einheitlichen Sprachform.

4.3.1.1 Abstrakte Datentypen

Die zentralen Beschreibungsobjekte der Sprache TESTPLAN-S sind Moduln. Wir verwenden den Begriff Modul im Sinne der abstrakten Datentypen (ADT'en) /Gutt 77/. Die ADT'en haben in den letzten Jahren eine zunehmende Beachtung als Entwurfs- und Abstraktionstechnik gefunden. Der Vorteil dieses Verfahrens ist der gegenüber Prozeduren und Makros höhere Abstraktionsgrad.

Motivation für die Entwicklung der ADT'en war die Zusammenfassung logisch zusammengehöriger Datentypen und Funktionen, sowie der Wunsch, Informationen gegenüber der Umwelt zu verbergen (information hiding). Die Philosophie der Datenabstraktion wird von einer Reihe neuerer Programmiersprachen unterstützt. (Beispiele hierfür sind: CLU /Lisk 77/, ALPHARD /Wulf 76/, ADA /Ledg 80/ und MODULA-2 /Wirt 80/).

Bei ADA und MODULA-2 kommt darüberhinaus eine weitere Neuerung hinzu : Diese beiden Sprachen teilen einen Modul textuell in einen Definitions- und Implentierungsmodul auf.
Der Definitionsmodul enthält im wesentlichen die Schnittstellen und kann quasi als Entwurf der zu implementierenden Software angesehen werden. Hauptsächlich wegen dieser Schnittstelleneigenschaften bietet sich für TESTPLAN-S an, das Definitionsmodulkonzept von MODULA-2 zu übernehmen. Syntaktisch sieht damit ein Modul ähnlich wie eine Prozedur in PASCAL aus. Jedoch unterscheiden sie sich in unterschiedlichen Regeln bezüglich der Sichtbarkeit lokaler Variablen /Wirt 81/.

Neben der Verbesserung der Entwurfsqualität haben die ADT'en einen positiven Einfluß auf die Testbarkeit der Software. Dies gilt hauptsächlich für den funktionalen oder 'Schwarzer-Kasten-Test', der naturgemäß in einem nach funktionalen Kriterien aufgebauten ADT besonders gut durchführbar ist.

4.3.1.2 Strukturelle und relationale Darstellung

Wir verwenden in TESTPLAN-S sowohl strukturelle als auch relationale Sprachelemente. Strukturelle Sprachelemente werden beispielsweise verwendet, um Gültigkeitsbereiche von Variablen abzugrenzen oder die Zugehörigkeit von Prozeduren zu einem Modul zu demonstrieren. Relationale Sprachanweisungen lassen sich durch ihre weitgehende Reihenfolgeunabhängigkeit charakterisieren. Beispiele für relationale Sprachanweisungen in TESTPLAN-S sind die Modulattribute, welche von der Aufschreibungsreihenfolge unabhängig und durch ihre Schlüsselworte selbstidentifizierend sind. Relationale Sprachen sind leicht erweiterbar, was man sich am Beispiel der Modulattribute gut veranschaulichen kann. Wenn eventuell zusätzliche Modulattribute notwendig werden, so können diese einfach über neue Schlüsselworte in die Sprache eingebaut werden.

Für diese Mischung aus struktureller und relationaler Darstellung diente die Sprache SSL (Software Specification Language) /Aust 76/ als Vorbild. TESTPLAN-S geht über SSL hauptsächlich in zwei Punkten hinaus. Erstens wird der einfache Modulbegriff (Modul = Prozedur oder Subroutine) von SSL durch die Betrachtung eines Moduls als ADT abgelöst. Zweitens wird einem Modul als weiteres Modulattribut ein Testplan zugeordnet.

4.3.1.3 Assertionen

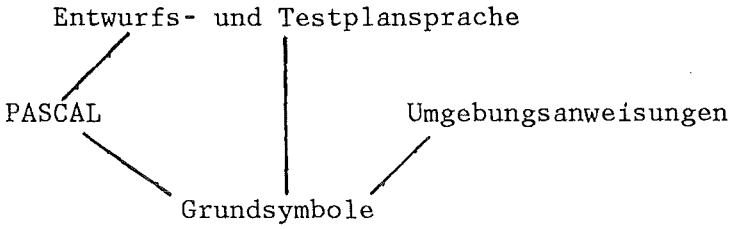
Das in TESTPLAN-S integrierte Assertionenkonzept bildet die Basis für alle Validations- und Testaktivitäten. Die Assertionen beschreiben in formaler - und zur restlichen Entwurfsbeschreibung teilweise redundanter - Weise das statische Verhalten eines Moduls. Wir unterscheiden in TESTPLAN-S zwei Ebenen, auf denen Assertionen zum Einsatz kommen : Die Modul- und die Prozedurebene. Auf der Ebene des Moduls werden die Assertionen hauptsächlich dazu benutzt, um Reihenfolgebedingungen von Zugriffsoperationen auszudrücken. Auf Prozedurebene wird damit das Input/Output-Verhalten einer einzelnen Prozedur beschrieben.

4.3.1.4 Integration verschiedener Teilsprachen zur Sprache TESTPLAN-S

Innerhalb der Sprache TESTPLAN-S lassen sich vier weitgehend unabhängige Teilkomponenten identifizieren :

- Eine Entwurfs- und Testplanbeschreibungssprache
  zur Beschreibung des Entwurfs und der ergänzenden Testpläne;
- PASCAL
  unter anderem zur Beschreibung der Datenstrukturen;
- Umgebungsanweisungen
  zur Steuerung der Entwurfs- und Testaktivitäten;
- Grundsymbole der Sprache.

Es gilt nun, dem Benutzer gegenüber eine einheitliche und einfache Schnittstelle anzubieten, in der diese Teilkomponenten zu einer homogenen Sprache zusammengefaßt sind. Der Zusammenhang dieser vier Teilkomponenten ist aus folgendem Diagramm zu entnehmen.



Die Verbindungslinien sind dabei so zu interpretieren, daß tieferstehende Teilkomponenten von den darüberstehenden benutzt werden. Beispielsweise werden in der Teilkomponente 'Entwurfs- und Testplansprache' die Datendefinitionen von PASCAL verwendet.

Die komplette Sprache TESTPLAN-S ist im Anhang A (Kap. 8.1) in BNF beschrieben. Diese Beschreibung gliedert sich analog den Teilkomponenten in vier Teile. In den beiden folgenden Kapiteln 4.3.2 und 4.3.3 wollen wir uns mit der Semantik der 'Entwurfs- und Testplansprache' sowie den 'Umgebungsanweisungen' beschäftigen. Auf die Teilkomponenten 'PASCAL' und 'Grundsymbole' werden wir nicht weiter eingehen, da deren Semantik aus der einschlägigen Literatur leicht zu entnehmen ist.

4.3.2 Entwurfs- und Testplanbeschreibung in TESTPLAN-S

Aus der Abbildung 4.3 können wir den schematischen Aufbau einer Entwurfsbeschreibung entnehmen. Wir gehen davon aus, daß ein Entwurf nach funktionalen Kriterien erfolgt und daß jede funktionale Einheit (= Modul) durch ihre Schnittstellen, Datenstrukturen und Invarianten charakterisiert ist. Eine Verfeinerung dieser schematischen Darstellung führt auf Abb. 4.3.2.

Aus diesem Beispiel ersehen wir, daß jeder Modul durch einen Modulbezeichner identifiziert wird. Die logischen Namen der in diesem Modul verwendeten Dateien werden durch Klammern eingerahmt im Anschluß an den Modulbezeichner angeführt.

Vereinbarungen und Datendefinitionen halten sich an die PASCAL-Konvention und sind an mehreren Stellen der Entwurfsbeschreibung möglich. Wir unterscheiden Datendefinitionen auf Modul-, Prozedur- und Testplanebene. Auf die Datendefinitionen innerhalb des Testplans gehen wir weiter unten ein. Die Gültigkeitsbereiche der Datendefinitionen sind durch die Blockschachtelung sowie durch die EXPORT- und IMPORT-Anweisungen festgelegt. Diese Regelung bedeutet insbesondere, daß bei Mehrfachdefinition ein und desselben Bezeichners immer die Definition in der innersten Schachtel gültig ist.

```
MODULE knk (eingabedatei, fehlerdatei);
     CONST ..
     TYPE ..    } (* Vereinbarungen nach PASCAL-Konvention *)
     VAR ..
     IMPORT ..
     EXPORT ..
     REFERTO ..  } (* Modulattribute, deren Semantik in den   *)
     ASSERT ..   } (* folgenden Kapiteln näher erläutert wird *)
     TESTPLAN ..

     PROCEDURE kühlmittel; (* Prozeduren gemäß PASCAL *)
         CONST ..
         TYPE ..    } (* Vereinbarungen nach PASCAL-Konvention *)
         VAR ..
         INASSERT  a < 10;
         OUTASSERT  b = 100;
         IMPORT ..                (* Prozedurattribute, deren Seman- *)
         EXPORT ..                (* tik in den folgenden Kapiteln   *)
         REFERTO ..               (* näher erläutert wird            *)
         INPUT ..
         OUTPUT ..
         CALL ..
         TESTPLAN ..
     ENDPROC (* kühlmittel *);
     .
     .
     PROCEDURE meldung(...);
     :
     ENDPROC (* meldung *);
ENDMOD (* knk *);
:
MODULE processteuerung;
       :
       (* Modulrumpf *)
       :
ENDMOD (* processteuerung *).
```

Abb. 4.3.2: Erweitertes Beispiel einer Entwurfsbeschreibung in TESTPLAN-S

Die Prozeduren werden durch das Schlüsselwortpaar PROCEDURE .. ENDPROC begrenzt, wobei im Anschluß an das Schlüsselwort PROCEDURE die Deklaration des Prozedurkopfes gemäß der PASCAL-Syntax steht. Innerhalb der Prozeduren gibt es keine weiteren Hierarchiestufen (Schachtelung).

Sowohl die Moduln als auch die Prozeduren umfassen neben den Datendefinitionen eine Reihe von Eigenschaften, die wir als Modul- bzw. Prozedurattribute bezeichnen.
Diese Attribute werden über Schlüsselworte identifiziert und sind jeweils mit bestimmten Sprachanweisungen assoziiert:

Import-Anweisung	(IMPORT)
Export-Anweisung	(EXPORT)
Assert-Anweisung	(ASSERT)
Inassert-Anweisung	(INASSERT)
Outassert-Anweisung	(OUTASSERT)
Trace-Anweisung	(REFER TO)
Input-Anweisung	(INPUT)
Output-Anweisung	(OUTPUT)
Call-Anweisung	(CALL)
Testplan-Anweisung	(TESTPLAN)

Im folgenden werden diese Sprachanweisungen erläutert.

4.3.2.1 Die Import- und Export-Anweisungen

Die Import- und Export-Anweisungen bilden zwei spiegelbildliche Operationen, die im wesentlichen aus MODULA-2 übernommen wurden. Im Gegensatz zur Sprache MODULA-2 lassen wir in TESTPLAN-S diese beiden Anweisungen sowohl auf Modul- als auch auf Prozedurebene zu. Die Import-Anweisung beschreibt dabei alle Objektbezeichner, die außerhalb deklariert sind, aber innerhalb des Moduls (der Prozedur) benutzt werden. Wenn ein Bezeichner in einer Import-Anweisung aufgeführt ist, kann das dadurch bezeichnete Objekt innerhalb des Moduls (der Prozedur) so verwendet werden, als ob die Modulgrenzen nicht existieren würden.

Die Export-Anweisung beschreibt alle Objektbezeichner, die innerhalb eines Moduls (einer Prozedur) spezifiziert sind und außerhalb benutzt werden sollen. Wir nehmen an, daß Prozeduren, die exportiert werden sollen, auf Blocklevel 1 des Moduls definiert und nicht rekursiv sind.

Für die Erstellung des Testplans kann man aus den Import- bzw. Export-Anweisungen Informationen über globale Variablen ableiten. Bei der Erstellung des Modul- oder Prozedurtestplans ist dann darauf zu achten, daß alle globalen Variablen mit Daten versorgt sind, oder aber deren Ergebnisse überwacht werden.

```
Beispiel:    IMPORT temperatur, druck;
             EXPORT mittelwert, referenzwert;
```

Die im Beispiel verwendeten Bezeichner 'temperatur' und 'druck' müssen selbstverständlich in einem anderen Modul deklariert sein. Ist dies nicht der Fall, so erhält man beim Generieren des Testrahmens aus der Entwurfsbeschreibung eine Fehlermeldung. Für die Bezeichner der Import-Anweisung einer Prozedur und der Import-Anweisung des umfassenden Moduls muß eine strenge Inklusionsbeziehung bestehen.

4.3.2.2 Die Assert-Anweisung

Die Assert-Anweisung beschreibt auf Modulebene in teils formaler, teils informeller Weise die Reihenfolgebedingungen für Zugriffsoperationen. Beispielsweise kann man sich vorstellen, daß in einem Modul 'Filesystem' folgende Reihenfolgebedingung spezifiziert ist :

```
ASSERT open ( read print )* close = 'korrekte Aufrufsequenz';
```

Diese Schreibweise ist so zu interpretieren, daß die Operation 'open' gefolgt wird von null oder mehreren Aufrufen der Sequenz 'read print' und durch die Operation 'close' abgeschloßen wird.

Die Bezeichner innerhalb der Assert-Anweisung müssen als moduleigene oder importierte Zugriffsfunktionen deklariert sein. Darüberhinaus gibt es innerhalb der Assert-Anweisung noch einen informellen Teil, in den der Entwerfer textuelle Erläuterungen schreiben kann, die vom Testplanersteller später ausgewertet werden können. Dieser Text wird als 'string' im Anschluß an das '='-Zeichen formuliert.

Zwischen der Assert-Anweisung und der Testplan-Anweisung besteht ein enger semantischer Zusammenhang. In Kapitel 4.1 haben wir einen Algorithmus beschrieben, der aus den regulären Ausdrücken der Assert-Anweisung Testfälle automatisch ableitet, die dann in der Testplan-Anweisung noch ausgearbeitet und in PASCAL-Code umgesetzt werden.

4.3.2.3 Die Inassert- und Outassert-Anweisungen

Während die Assert-Anweisung nur auf Modulebene (globale Assertionen) zulässig ist, werden die Inassert- und die Outassert-Anweisungen ausschließlich auf Prozedurebene (lokale Assertionen) angewandt.

Diese beiden Anweisungen beschreiben in statischer Weise die Datenbedingungen (Assertionen) vor und nach der Ausführung eines Prozeduraufrufs. Syntaktisch handelt es sich bei diesen Assertionen um logisch/arithmetische Ausdrücke, wie sie beispielsweise aus der PASCAL-Ausdrucksgrammatik bekannt sind. Darüberhinaus sind noch Existenzquantoren (FOR SOME) und Allquantoren (FOR ALL) zulässig. Für die Formulierung dieser Quantoren wurde auf die Assertionenform der Sprache PL/CS (/Zelk 79/ S. 96) zurückgegriffen.

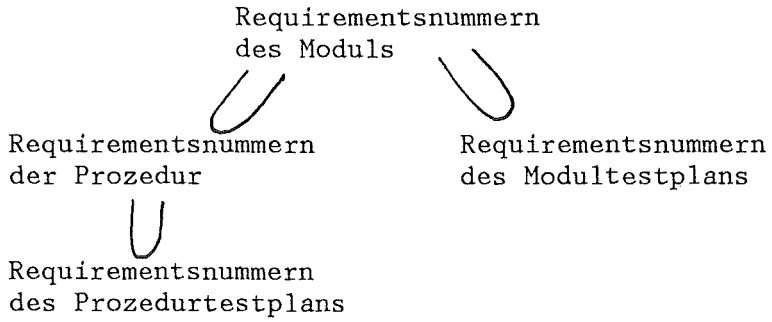
Das Assertionenpaar (INASSERT, OUTASSERT) bildet die Grundinformation für einen prozedurspezifischen Testplan. In Kapitel 4.2 haben wir beschrieben, wie man aus den logisch/arithmetischen Prädikaten automatisch Testfälle und eventuell sogar Testdaten ableiten kann. Wir wollen an dieser Stelle nur noch darauf hinweisen, daß man mittels der logisch/arithmetischen Prädikate die Eingangsdaten einer Prozedur in Eingabedatenklassen zerlegen kann. Jede Klasse bildet einen Testfall, der stellvertretend durch einen Repräsentanten zu testen ist. Diese Vorgehensweise führt zu einer beträchtlichen Reduktion der notwendigen Testdaten.

```
Beispiel:  INASSERT ( temp(.i.) > 100 ) AND ( temp(.i.) < 200 )
                    FOR ALL i = 1 TO 100;
           OUTASSERT mittelwert = 170;
```

4.3.2.4 Die Trace-Anweisung

Die Trace-Anweisung stellt eine Beziehung zwischen der Anforderungsspezifikation und dem Entwurf her. Diese Anweisung erlaubt es, Systemanforderungen über den Entwurf bis hin zur Implementierung und zum Testplan zu verfolgen (vgl. 4.3.2.7.3). Wir setzen dabei voraus, daß die einzelnen Anforderungen durch Nummern (z. B. 11) identifiziert sind.

In TESTPLAN-S ist die Trace-Anweisung auf Modul-, Prozedur- und Testplanebene zulässig, wobei zwischen den Trace-Anweisungen dieser drei Beschreibungseinheiten folgende Inklusionsbeziehungen bestehen :



Anhand des folgenden Beispiels wollen wir den Aufbau einer Trace-Anweisung demonstrieren :

```
REFER TO 3, 4, 16;
```

Auf weitere Eigenschaften und Zielsetzungen der Trace-Anweisung wird im Kapitel 4.3.2.7 noch genauer eingegangen.

4.3.2.5 Die Call-Anweisung

Die Call-Anweisung ist nur als Prozedurattribut erlaubt und beschreibt in dieser Funktion die Aufrufbeziehungen zu anderen Prozeduren. Diese aufgerufenen Prozeduren können entweder im selben Modul vereinbart sein oder aber aus einem anderen Modul importiert werden. Somit können mit Hilfe der Import-Anweisung und der lokaldefinierten Prozeduren Plausibilitätskontrollen durchgeführt werden, die bestimmen, ob ein Bezeichner innerhalb der Call-Anweisung zulässig ist.

Darüberhinaus kann man beschreiben, ob andere Prozeduren unbedingt, bedingt (COND, d. h. in Abhängigkeit von bestimmten Variablen) oder zyklisch (CYCL) aufgerufen werden.

Beispiel: CALL meldung CYCL lieseingabe;

Dieses Beispiel ist so zu interpretieren, daß die Prozedur 'meldung' unbedingt und die Prozedur 'lieseingabe' zyklisch aufgrufen wird.

Die Call-Anweisung beschreibt externe Prozeduraufrufe und kann in dieser Eigenschaft zum Binden des generierten Testrahmens mit dem Testobjekt sowie den zusätzlichen Moduln und Bibliotheken benutzt werden.

4.3.2.6 Die Input- und Output-Anweisungen

Es gibt zwei Arten, Daten von außerhalb in eine Prozedur einzubringen. Die erste Möglichkeit über globale Variable läßt sich durch die Import- bzw. Export-Anweisungen beschreiben. Die zweite Möglichkeit, die mit Hilfe der Input- und Output-Anweisungen formuliert wird, besteht im Lesen von bzw. Schreiben auf Dateien.

Die Input- bzw. Output-Anweisungen stellen eine Beziehung her zwischen den Datenobjekten einer Prozedur und dem logischen Dateinamen, unter dem die Daten gehalten werden. Man kann sich vorstellen, daß einer Input-Anweisung (Output-Anweisung) der Entwurfsbeschreibung eine READ-Anweisung (WRITE-Anweisung) der Implementierung entspricht.

```
Beispiel:  INPUT startwert, schrittweite  FROM referenzdatei;
           OUTPUT mittelwert TO druckdatei;
```

Man muß beachten, daß die verwendeten Datenobjekte ('startwert', 'schrittweite' und 'mittelwert') innerhalb des Prozedurentwurfs entweder explizit vereinbart wurden, oder über eine Import-Anweisung innerhalb der Prozedur sichtbar sind. Die Bezeichner nach den Schlüsselwörtern FROM und TO müssen jeweils logische Dateinamen sein. Diese Dateinamen müssen in der umfassenden Moduldefinition in Form von Parametern zusammen mit dem Modulnamen aufgeführt sein (z. B. MODUL xyz (referenzdatei, druckdatei, fehlerfile);).

4.3.2.7 Die Testplan-Anweisung

Die Testplan-Anweisung erlaubt, Testpläne auf Prozedur- und Modulebene zu beschreiben und daraus direkt ausführbare Testrahmen abzuleiten. Die Testplan-Anweisung basiert auf den Informationen der Entwurfsbeschreibung, insbesondere den Datendefinitionen, der Assert-, Inassert- und Outassert-Anweisung und ist als Ergänzung der Entwurfsbeschreibung anzusehen. Aus der Entwurfsbeschreibung werden (automatisch) Testfälle abgeleitet, die ein Gerüst bilden, das in der Testplan-Anweisung dann mit "Fleisch", d. h. einer programmiersprachlichen Beschreibung der Testfälle umgeben wird. Für die einzelnen Testfälle werden explizite Testdaten bereitgestellt, die entweder über Zuweisungen oder über Lesebefehle von Dateien in das Programm eingebracht werden.

Die Abbildung 4.3.2 zeigt eine schematische Darstellung des Entwurfs, woraus die syntaktische Einordnung der Testplan-Anweisung entnommen werden kann. Im Bild 4.3.3 beschreiben wir eine Verfeinerung der Testplan-Anweisung und damit den Aufbau eines Testplans.

```
:
TESTPLAN
   SPECIFIER 'xyprogrammierer 16. 2. 1982';
   REFERTO 2, 3;
   TESTBEG
      (* Vereinbarungen gemäß PASCAL-Konvention *)
      (* Makrovereinbarungen *)
      TEST(. 1 .)
         TESTSPEC ... ; (* Beschreibung des Testfalls, wie er aus dem *)
                        (* Entwurf automatisch abgeleitet wurde *)
         (* Vereinbarungen gemäß PASCAL-Konvention *)
         (* Makrovereinbarungen *)
         REFERTO 3;
         IMPORT random, poisson;
         PASCBEG
            (* PASCAL-Programm, das den unter TESTSPEC spezifizierten #)
            (* Testfall implementiert. *)
            :
         PASCEND;
      :
      TEST(. n .)
         TESTSPEC ...
         :
   TESTEND;
   :
```

Abb. 4.3.3: Schema eines Testplans

Innerhalb des Testplans kann optional zunächst allgemeine Information über den Testplanersteller, das Datum etc. beschrieben werden. Vereinbarungen sind sowohl global für den gesamten Testplan als auch lokal für die einzelnen Testfälle möglich.
Der gesamte Testplan besteht aus einer Folge von Testfällen (TEST(. i .)), die durch die Umgebungsanweisungen (siehe Kapitel 4.3.2.8) in beliebiger Reihenfolge manipuliert werden können.

Jeder Testfall enthält neben der Testspezifikation (TESTSPEC), die automatisch aus dem Entwurf abgeleitet wird, insbesondere noch ein PASCAL-Programmstück, welches den in der Testspezifikation beschriebenen Testfall implementiert.

Im folgenden wollen wir einige Spracheigenschaften, die sich hinter der Testplan-Anweisung verbergen, näher betrachten.

4.3.2.7.1 Die Makrovereinbarung

Bei der Anwendung der Sprache auf einige Beispiele zeigte es sich, daß die Formulierung solcher Testpläne einen großen Schreibaufwand erfordert. Da die Folgen von Programmanweisungen in den einzelnen Testfällen häufig fast identisch sind, bot sich die Einführung eines Makrokommandos (DEFINE) an. Diese Makroschreibweise erlaubt eine kompakte und prozedurähnliche Schreibweise.

Man unterscheidet die Makrodeklaration und den Makroaufruf. Die Makrodeklaration assoziiert einen Text aus dem Sprachumfang der Sprache TESTPLAN-S mit einem Bezeichner. Der Makroaufruf führt zur Ersetzung des Bezeichners durch den vordefinierten Text (eventuell mit Ersetzung der formalen Parameter). Somit wird der Makroaufruf zur Übersetzungszeit textuell expandiert.

Es ist nicht erlaubt, in der Makrodefinition Vereinbarungen zu verwenden. Die Verwendung geschachtelter Makros ist zulässig, sofern alle verwendeten Makrobezeichner vorher definiert wurden.

Oftmals kann man feststellen, daß die verschiedenen Testfälle sich nur an wenigen Positionen oder in wenigen Variablenwerten unterscheiden. Es liegt dann nahe, die Makrotechnik zu verwenden, um jeweils einen Referenzdatensatz als Makroanweisung vorzugeben, in dem dann gezielt die fraglichen Variablen zu verändern sind.

Beispiel: Die Makrovereinbarung möge folgendermaßen aufgebaut sein:

```
DEFINE zuweisung(par1,par2) =
       feld(.1.) := par1; feld(.2.) := par2#;
```

Der Makroaufruf

```
zuweisung(8,9);
```

bewirkt dann

```
feld(.1.) := 8; feld(.2.) := 9;
```

4.3.2.7.2 Testhilfsfunktionen

Die Testdurchführung erfordert eine Reihe von Testhilfsfunktionen für sich wiederholende Operationen. Zu diesen Hilfsfunktionen zählen beispielsweise Zufallszahlengeneratoren und Prozeduren für den Ergebnisvergleich zwischen erwarteten und tatsächlichen Ausgabewerten.

Wir nehmen an, daß diese Testhilfsfunktionen in einem eigenen Modul 'testsupport' zusammengefaßt sind und in vorübersetzter Form vorliegen. Der Modul 'testsupport' ist dann zusammen mit dem Testrahmen und dem Testobjekt zu binden.

4.3.2.7.2.1 Zufallszahlengeneratoren

Zufällig erzeugte Testdaten haben insbesondere dann ihre Bedeutung, wenn man ein Programm mit einer großen Zahl von Testfällen überprüfen will, wobei man die erwarteten Ergebnisse zumeist nicht kennt oder aber aus Aufwandsgründen nicht vorherbestimmen will.

Tests mit Zufallszahlen sind oft auf die Robustheit eines Programms ausgerichtet und können insofern auch als 'Crash-Tests' bezeichnet werden. Eine weitere Anwendung für zufällig erzeugte Testdaten ergibt sich dann, wenn neben dem Testobjekt noch ein Modell des Testobjekts zur Verfügung steht, die dann beide mit den Zufallszahlen versorgt werden und deren Ergebnisse auf Gleichheit überprüft werden.

Nach der Erfahrung, die man in SIMULA67 mit Zufallszahlengeneratoren gemacht hat, erweist sich die Einführung folgender Generatorfunktionen als ausreichend:

Verteilungstyp	Notation
Erlangverteilung	ERLANG (Miwert,Abweichung,Startwert)
Normalverteilung	NORMAL (Miwert,Abweichung,Startwert)
Negativ-Exponentielle Verteilung	NEGEXP (Mittelwert,Startwert)
Poissonverteilung	POISSON (Mittelwert,Startwert)
Gleichverteilung für ganze Zahlen	RANDINT (Intervallanfang,Intervallende, Startwert)
Gleichverteilung für reelle Zahlen	UNIFORM (Intervallanfang,Intervallende, Startwert)

4.3.2.7.2.2 Ergebnisvergleich zwischen erwarteten und tatsächlichen Ausgabewerten

Die Ergebnisvergleiche zwischen erwarteten und tatsächlichen Ausgabewerten (Prozedur COMPARE) ist eine oft wiederkehrende Funktion, die hauptsächlich den Schreibaufwand vermindern soll. Die Prozedur COMPARE hat drei Parameter: die zu überwachende Variable, das erwartete Ergebnis und eine Datei, auf die eventuelle Differenzen ausgegeben werden.

Beispielsweise ist das Programmstück

```
VAR teinmittel, tmittel : INTEGER;
:
tmittel :=360; COMPARE( teinmittel, tmittel, fehlerfile);
```

identisch mit dem folgenden Programmstück.

```
VAR teinmittel, tmittel : INTEGER;
:
tmittel :=360;
IF teinmittel <> tmittel THEN
   BEGIN
   WRITE(fehlerfile,'Fehler ! teinmittel : Soll= ',tmittel:6);
   WRITE(fehlerfile,'Ist = ',teinmittel:6); WRITELN;
   END;
```

4.3.2.7.3 Verfolgen von Systemanforderungen bis zum Testplan

Eine wichtige Eigenschaft des Verfahrens TESTPLAN und insbesondere auch der Testplan-Anweisung ist die Verfolgbarkeit der Anforderungen über den Entwurf und Testplan bis auf die Ebene der einzelnen Testfälle. Aus diesen Informationen lassen sich eine Reihe aussagekräftiger Reports ableiten, die für eine Teststatistik herangezogen werden können. So können beispielsweise folgende, für den Abnahmetest interessierende Fragen beantwortet werden:

- Wurden alle in der Anforderungsspezifikation vorkommenden Systemanforderungen im Entwurf eingearbeitet?
- In welchem Baustein des Entwurfs oder der späteren Realisierung werden die einzelnen Anforderungen behandelt?
- Mit welchen Testfällen wurden einzelne Systemanforderungen überprüft?

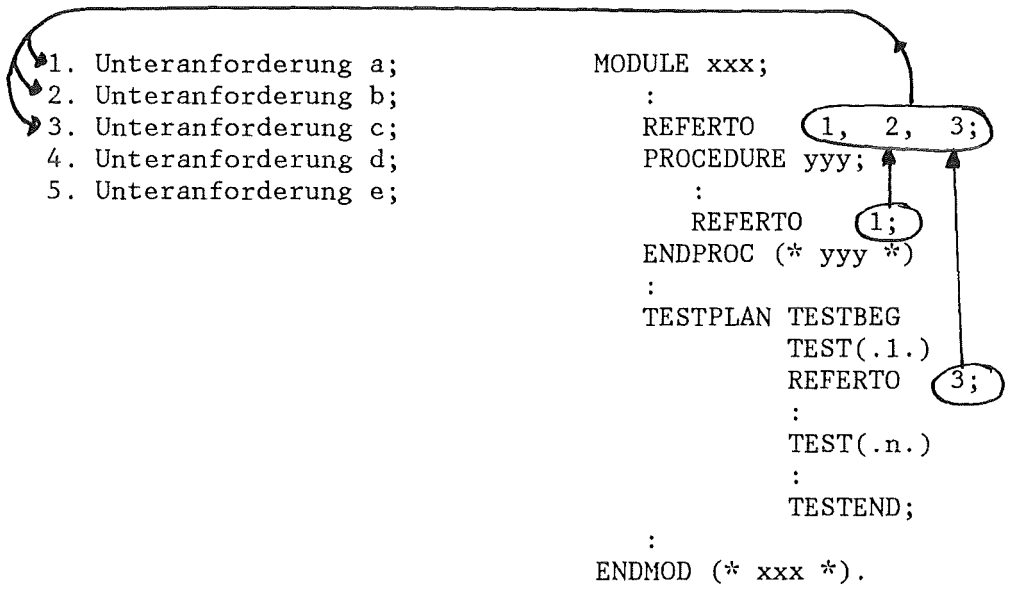


Abb. 4.3.4: Beziehungen zwischen Systemanforderungen, Entwurf und Testplan.

Wir gehen davon aus, daß die Systemanforderungen in natürlicher Sprache vorliegen und daß die einzelnen Unteranforderungen durch eindeutige Numerierungen identifiziert sind.
Damit haben wir einen festen Bezugspunkt innerhalb der Systemanforderung und können aufgrund der im Kapitel 4.3.2.4 geschilderten Inklusionsbeziehungen die oben gestellten Fragen beantworten.

4.3.2.7.4 Allgemeine Bemerkungen zur Testplan-Anweisung

Die Verwendung von PASCAL-Datendeklarationen und die explizite Beschreibung der Testfälle in PASCAL-Notation stellt sicher, daß die mit der Sprache TESTPLAN-S formulierbaren Testpläne in ihrer Komplexität nur durch die Mächtigkeit von PASCAL selbst eingeschränkt sind.

Die Generierung eines Testrahmens stützt sich hauptsächlich auf die in der Testplan-Anweisung enthaltenen Informationen. Darüberhinaus werden die Deklarationen des Testrahmens aus den einzelnen Datendefinitonen des Moduls und der Prozedur zusammengebaut. Dabei haben bei Doppeldefinitionen die im Sinne der Blockschachtelung weiter innen stehenden Definitionen Vorrang über die außenstehenden. Innerhalb der Testplan-Anweisung unterscheiden wir nochmals zwei mögliche Datendefinitionsebenen: eine für den gesamten Testplan und eine für einzelne Testfälle gültige Datendefinition.

4.3.3 Umgebungsanweisungen in TESTPLAN-S

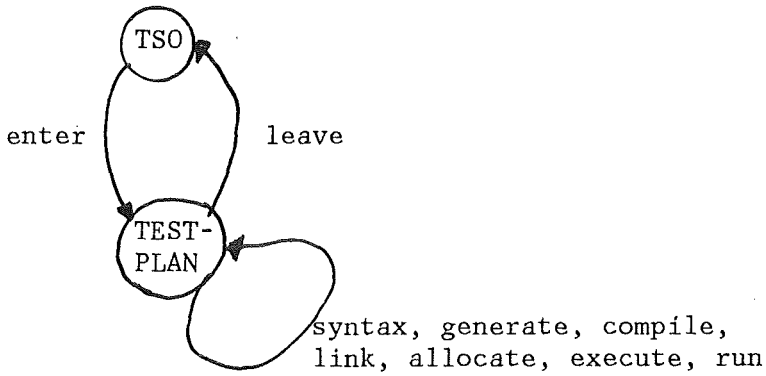
Die Sprache TESTPLAN-S stellt eine Reihe von Umgebungsanweisungen zur Verfügung, welche eine TESTPLAN-eigene Kommandosprache bilden. Gegenwärtig sind folgende Kommandos vorgesehen:

```
ENTER                 (* Übergang von TSO[1] nach TESTPLAN *)

LEAVE                 (* Übergang von TESTPLAN nach TSO *)

SYNTAX ...            (* Übersetzen einer in TESTPLAN-S *)
                      (* formulierten Entwurfsbeschreibung *)

GENERATE ...          (* Erstellen eines Testrahmens aus *)
                      (* der Zwischensprache             *)

COMPILE  ...          (* Übersetzen eines Testrahmens mit *)
                      (* dem PASCAL-Compiler              *)

LINK ...              (* Binden der Testobjekte mit dem  *)
                      (* Testrahmen und ext. Prozeduren  *)

ALLOCATE ...          (* Herstellung einer Verbindung    *)
                      (* zwischen log. und physikalischer *)
                      (* Filebeschreibung                *)

EXECUTE ...           (* Ausführen des Testrahmens       *)

RUN ...               (* COMPILE + LINK + EXECUTE        *)
```

1) TSO ist ein Time-sharing-Betriebssystem der Fa. IBM

Diese Kommandos bilden lediglich eine minimale Untermenge. Erweiterungen, insbesondere um Kommandos zur Überwachung des Testfortschritts (Ausführungshäufigkeit, etc.), sind jederzeit denkbar und leicht integrierbar.

Formuliert man die Dialogkommandos und Systemzustände in Form eines endlichen Automaten, so ergibt sich folgendes Bild:



Das SYNTAX-Kommando veranlaßt eine syntaktische und semantische Prüfung des Quelltextes in einer vorgegebenen Datei. Dabei wird eine Eingabe auf Übereinstimmung mit der TESTPLAN-S-Sprachkonvention überprüft. Falls das Quellprogramm Fehler enthält, werden entsprechende Fehlermeldungen abgesetzt, ansonsten wird für dieses Quellprogramm eine Zwischensprache erzeugt. Diese Zwischensprache kann interpretativ weiterverarbeitet werden (durch das GENERATE-Kommando).

Ausgehend von der im SYNTAX-Kommando erstellten Zwischensprache werden Testrahmen generiert. Je nach dem Aufbau des GENERATE-Kommandos kann man eine unterschiedliche Anzahl von Testrahmen generieren. Es ist möglich Testrahmen für einzelne Moduln, Prozeduren oder auch Gruppen von Testfällen zu generieren.

Das COMPILE-Kommando initiiert die Übersetzung eines Testrahmens durch den PASCAL-Compiler. Es sind dieselben Parameter zulässig wie im GENERATE-Kommando. Das COMPILE-Kommando setzt voraus, daß der Testrahmen zuvor mittels des GENERATE-Kommandos erstellt wurde. Falls der Testrahmen noch nicht existiert, wird die Anforderung zur Übersetzung ignoriert.

Das LINK-Kommando erzeugt aus übersetzten Testrahmen und den vorübersetzten Bausteinen (Moduln) ein gebundenes und ablauffähiges Programm.

Das EXECUTE-Kommando startet ein gebundenes Programm.

Das ALLOCATE-Kommando stellt eine Verbindung her zwischen den Dateinamen im Programm (Testobjekt und Testrahmen) und der physikalischen Filebezeichnung. Das ALLOCATE-Kommando muß auf jeden Fall vor dem EXECUTE-Kommando ausgeführt werden.

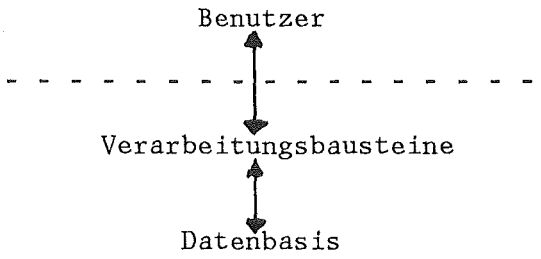
Das RUN-Kommando steht abkürzend für die Kommandofolge
COMPILE - LINK - EXECUTE.

4.4 Das Werkzeug TESTPLAN-W

Das Werkzeug TESTPLAN-W ist ein Instrument zur Speicherung, Prüfung, Dokumentation und Verarbeitung von Entwurfs- und Testplanbeschreibungen, die gemäß dem Verfahren TESTPLAN-V erstellt und in der Sprache TESTPLAN-S formuliert sind.

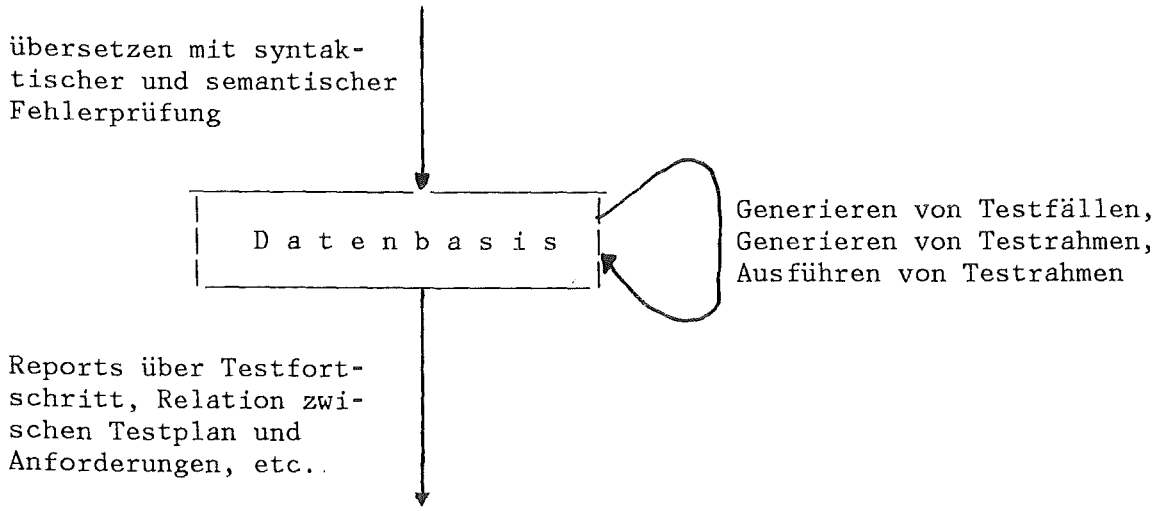
Mit Hilfe dieses Werkzeugs ist es möglich, die Einhaltung des Sprachumfangs von TESTPLAN-S zu überwachen, Testfälle aus dem Entwurf abzuleiten und automatisch Testfälle zu generieren sowie den Testfortschritt zu dokumentieren.

Eine erste Problemanalyse für das Werkzeug TESTPLAN-W führt auf folgende - einem allgemeinen Dialogsystem ähnliche - Grundstruktur.



Wenn man diese Strukur weiter verfeinert, gelangt man zu dem schichtenweisen Aufbau und der Modularisierung, wie sie in Kapitel 5 beschrieben ist.

TESTPLAN-W besteht aus einer Reihe von Funktionen (Verarbeitungsbausteinen), die alle nach folgendem Schema auf einer gemeinsamen Datenbasis operieren.



Im Kapitel 5 wird die Realisierung der Datenbasis detailliert beschrieben, während wir uns hier auf einige allgemeine Überlegungen zur Datenbasis des Werkzeugs TESTPLAN-W beschränken. Abschließend werden wir kurz die verschiedenen Funktionen charakterisieren, die auf der Datenbasis operieren.

4.4.1 Die Datenbasis

In der Datenbasis, dem Kern des Werkzeugs TESTPLAN-W, wird die Entwurfsbeschreibung und die daraus abgeleitete Zwischensprache gespeichert. Die Datenbasis wird gebildet von einer Reihe einzelner Datenstrukturen, die wiederum zumeist mit den Sprachanweisungen der Sprache TESTPLAN-S assoziiert sind.

Informationen werden in die Datenbasis eingebracht durch Übersetzen eines TESTPLAN-S-Quellprogramms, durch das Generieren von Testrahmen oder aber das Ausführen eines Testrahmenprogramms.
Die Datenbasis stellt sich gegenüber dem Benutzer als schwarzer Kasten dar, auf dessen Inhalt er nur über Funktionen zugreifen kann. Dieses Prinzip des 'information hiding' erleichtert vor allem die Konsistenzhaltung der Datenbasis.

Wir gehen im Verfahren TESTPLAN-V davon aus, daß der Entwurf um einen Testplan ergänzt wird. Parallel zur schrittweisen Entwicklung des Entwurfs würde sich auch eine inkrementelle Übersetzung des Quelltexts in die Zwischensprache anbieten. Aus Gründen einer einfacheren Realisierung gehen wir dennoch davon aus, daß Entwurf und Testplan zusammen vorliegen und in einem Schritt übersetzt werden. Eventuell von früher her vorhandene Übersetzungen eines Moduls mit demselben Namen werden überschrieben.

4.4.2 Die Funktionen des Werkzeugs

Wir unterscheiden in TESTPLAN-W drei Gruppen von Funktionen:

- Eine Funktion zur Prüfung der Syntax und statischen Semantik;
- Testspezifische Funktionen;
- Funktionen für den Aufbau und die Manipulation der Datenbasis.

Alle Funktionen, die darüberhinaus in den Bereich des Editierens fallen, werden in der normalen Editierumgebung des Gastsystems durchgeführt.

Die Funktionen des Werkzeugs TESTPLAN-W entsprechen den Umgebungsanweisungen (vgl. Kap. 4.3.3) in der Sprache TESTPLAN-S.
D. h., man hat innerhalb dieser Sprache Kommandos, um die Funktionen des Werkzeugs zu aktivieren. Einschränkend ist noch zu sagen, daß nur ein Teil der weiter unten beschriebenen Funktionen als Umgebungsanweisungen in die Sprache TESTPLAN-S integriert ist.
Es gibt aber - außer der Aufblähung des Sprachumfangs - keine prinzipiellen Schwierigkeiten, Kommandos für alle Funktionen in die Sprache einzubauen.

4.4.2.1 Eine Funktion zur Prüfung der Syntax und der statischen Semantik

Zur Prüfung einer TESTPLAN-S-Entwurfsbeschreibung auf syntaktische und (statisch) semantische Korrektheit wird die Funktion

SYNTAX (<Dateiname> // <,>)*

bereitgestellt.

Falls die Entwurfsbeschreibung korrekt ist, wird eine Übersetzung des TESTPLAN-S-Quelltexts in eine Zwischensprache vorgenommen, anderenfalls werden Fehlermeldungen ausgegeben.
Nur auf der Basis einer fehlerfreien Übersetzung ist es möglich, die übrigen Funktionen von TESTPLAN-W anzuwenden, denn alle diese Funktionen (z. B. testspezifische Funktionen) benötigen die vom SYNTAX-Kommando erzeugte Zwischensprache.

4.4.2.2 Testspezifische Funktionen

4.4.2.2.1 Generierung von Testfällen

Wir nehmen an, daß in der Entwurfsbeschreibung in Form von mehreren regulären Ausdrücken Reihenfolgebedingungen für Operationsfolgen definiert sind. Die Funktion 'Testfallgenerierung' baut gemäß dem in Kapitel 4.1 beschriebenen Verfahren aus den regulären Ausdrücken einen endlichen Automaten auf und leitet aus diesem Testfälle ab.
Als Abbruchkriterium für die Testfallgenerierung wählen wir die mindestens einmalige Ausführung jedes Übergangs innerhalb des endlichen Automaten. Damit liefert diese Funktion eine endliche Menge von Testfällen, die innerhalb des Testplanstatements (TESTPLAN) unter verschiedenen Testnummern (TEST(.i.)) als Kommentar abgespeichert werden.

Der Benutzer hat nun noch die als Kommentar vorliegenden Testfälle in PASCAL zu formulieren. Hierzu müssen Testdaten bereitgestellt werden, mit denen die Prozeduraufrufe zu versorgen sind. Ebenso ist es nötig, die erwarteten Ergebnisse mit den tatsächlichen Resultaten einer Operation oder Operationsfolge zu vergleichen.

4.4.2.2.2 Generierung eines Testrahmens

Diese Funktion wird durch das Kommando GENERATE implementiert. Im Testplanstatement stehen PASCAL-Anweisungen zur Versorgung des Testobjekts und zum Überprüfen der erwarteten Ergebnisse. Diese Anweisungen werden zusammen mit den Vereinbarungen des Testplans und den Vereinbarungen der umfassenden Beschreibungseinheiten (Prozeduren, Modulen) zu einem korrekt aufgebauten PASCAL-Programm zusammengebaut. Letztlich muß noch ein Programmkopf generiert werden, dessen Programmname aus dem Modulnamen und der Testnummer (TEST(.i.)) zusammengesetzt ist.

4.4.2.2.3 Übersetzen, Binden und Ausführen von Testrahmen

Diese Funktionen sind durch die Kommandos COMPILE, LINK, EXECUTE, RUN und ALLOCATE (vgl. Kap. 4.3.3) implementiert. In den meisten Time-Sharing-Systemen gibt es solche oder ähnliche Kommandos, so daß auf deren Semantik nicht näher eingegangen werden muß.

4.4.2.2.4 Testauswertung und Ausgabe von testrelevanter Information

Um eine Testauswertung zu ermöglichen, wird während der Übersetzung einer TESTPLAN-S-Eingabe und zur Ausführungszeit (durch Instrumentierung) statische und dynamische Testinformation in der Datenbasis akkumuliert. Diese Informationen sind vom Benutzer abrufbar und können in Form folgender Reports ausgegeben werden:

- Auflisten aller Testrahmen eines Moduls;
- Aufzählen aller Testfälle, die eine bestimmte Anforderung überprüfen (Damit können nicht getestete Anforderungen identifiziert werden.);
- Aufzählen aller fehlerhaft verlaufenen Tests;
- Bestimmung der Testüberdeckung auf Prozedur-, Statement- und Verzweigungsebene. Es sind sowohl Einzelreports als auch kumulierte Reports für mehrere Testfälle möglich. Um die Testüberdeckung überhaupt bestimmen zu können, muß zuvor eine Instrumentierung des Testobjekts erfolgt sein.

4.4.2.2.5 Archivierung und Restauration der Testfälle

Da die Testläufe sich oftmals über einen längeren Zeitraum hinziehen und später zu Dokumentationszwecken und für Regressionstests wieder zur Verfügung stehen müssen, sind Vorkehrungen für eine Langzeitdatenhaltung zu treffen. Zu diesem Zweck sind Funktionen für die Archivierung und Restauration von Testfällen, Testrahmen etc. auf externen Speichermedien vorgesehen.

4.4.2.3 Funktionen für den Aufbau und die Manipulation der Datenbasis

Zur Verwaltung der Datenbasis dienen folgende Funktionen:

a) Funktionen zum Aus- und Einlagern der gesamten Datenbasis;

b) Funktionen zum Löschen von Einträgen in der Datenbasis;

c) Funktionen zur Speicherverwaltung (Beschaffen und Freigeben);

d) Funktionen für die Manipulation der Zwischensprache (insbesondere des Strukturbaums, der Tabellen und Listen).

Während die Funktionsgruppen a) und b) nach außen sichtbar und damit dem Benutzer zugänglich sind, dienen die Funktionen c) und d) rein internen Zwecken. Die letztgenannten Funktionen werden nur implizit von anderen Funktionen benötigt.

5. Realisierung des Systems TESTPLAN

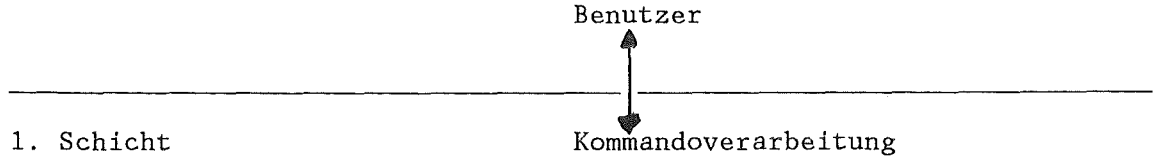
In diesem Kapitel behandeln wir die Implementierung des Werkzeugs TESTPLAN-W. Der weitaus aufwendigste Teil der Implementierung betrifft die Übersetzung der Sprache TESTPLAN-S. Diese Sprache ist im Kap. 8.1 als kontextfreie Grammatik beschrieben. Um ein übersetzererzeugendes System verwenden zu können, wurde eine äquivalente attributierte Grammatik für TESTPLAN-S entwickelt /Gmei 82/.

In den folgenden Unterkapiteln wollen wir
- die Realisierungskonzepte,
- den Datenfluß,
- den Einsatz von attributierten Grammatiken und Übersetzererzeugenden Systemen sowie
- den Stand der Implementierung beschreiben.

5.1 Realisierungskonzepte

Um angestrebte Programmqualitäten, wie Portabilität, Strukturiertheit und Änderungsfreundlichkeit zu unterstützen, wurde bei der Realisierung u. a. nach folgenden Konzepten vorgegangen:

Schichtenweiser Systemaufbau und Modularisierung
Eine Problemanalyse und schrittweise Verfeinerung führt auf folgenden Systemaufbau:



1. Schicht	Kommandoverarbeitung
2. Schicht	Funktionsmodule
3. Schicht	Datenmodule
4. Schicht	Speicherverwaltung

Der Benutzer aktiviert die Systemfunktionen über Kommandos, die in der ersten Schicht zusammengefaßt sind.

In der zweiten Schicht liegen eine Reihe von Funktionsmodulen:
- Ein 'Übersetzungsmodul'
  mit den Submodulen 'Symbolentschlüsselung' und 'Protokollierung';
- Ein Modul 'Testspezifische Funktionen'
  mit den Submodulen 'Testfallgenerierung' und 'Testrahmengenerierung';
- Ein 'Testauswertungsmodul';

. Ein 'Testausführungsmodul' mit Submodul 'Instrumentierung';

. Ein Modul 'Hilfsfunktionen' (beispielsweise für die Auslagerung von Dateien auf Hintergrundspeicher).

Die dritte Schicht besteht aus Datenmodulen für den Zugriff und die Manipulation der Datenstruktur. In unserer Realisierung sind diese Module, aufgrund der Verwendung automatischer Hilfsmittel, teilweise in die Funktionsmodule integriert(beispielsweise sei auf die vom GAG-System erzeugten Verwaltungsfunktionen für die Symboltabelle hingewiesen).

Die vierte Schicht ist für den Benutzer nicht sichtbar.

PASCAL als Implementierungssprache
Die Implementierung wurde in PASCAL-VS auf der IBM 3033 des Kernforschungszentrums Karlsruhe durchgeführt. PASCAL eignet sich insbesondere aufgrund seiner mächtigen Datenstrukturierungsmöglichkeiten für eine solche Implementierung. Außerdem bietet diese Sprache gute Eigenschaften bei der Laufzeitfehlerüberwachung, sowie einen interaktiven Debugger, der insbesondere während der Implementierung vorteilhaft ist.

Automatische Hilfsmittel
Da an die Implementierung des Systems TESTPLAN in weiten Teilen ähnliche Anforderungen wie an die Programmierung eines Compilers gestellt werden, bietet es sich an, automatische Hilfsmittel aus dem Compilerbau einzusetzen. An der UNI Karlsruhe steht hierfür das Compilererzeugende System GAG /Kast 82/ zur Verfügung. Der Einsatz des GAG-Systems beschleunigt die Implementierung wesentlich und erhöht gleichzeitig deren Zuverlässigkeit.
Auf dieses System und die von ihm verlangten Randbedingungen gehen wir in Kapitel 5.3 noch genauer ein.

5.2 Datenfluß im System Testplan

In Abb 5.1 Wird der Datenfluß für das TESTPLAN System beschrieben. Die gestrichelte Linie umschließt alle bisher implementierten Systemteile.

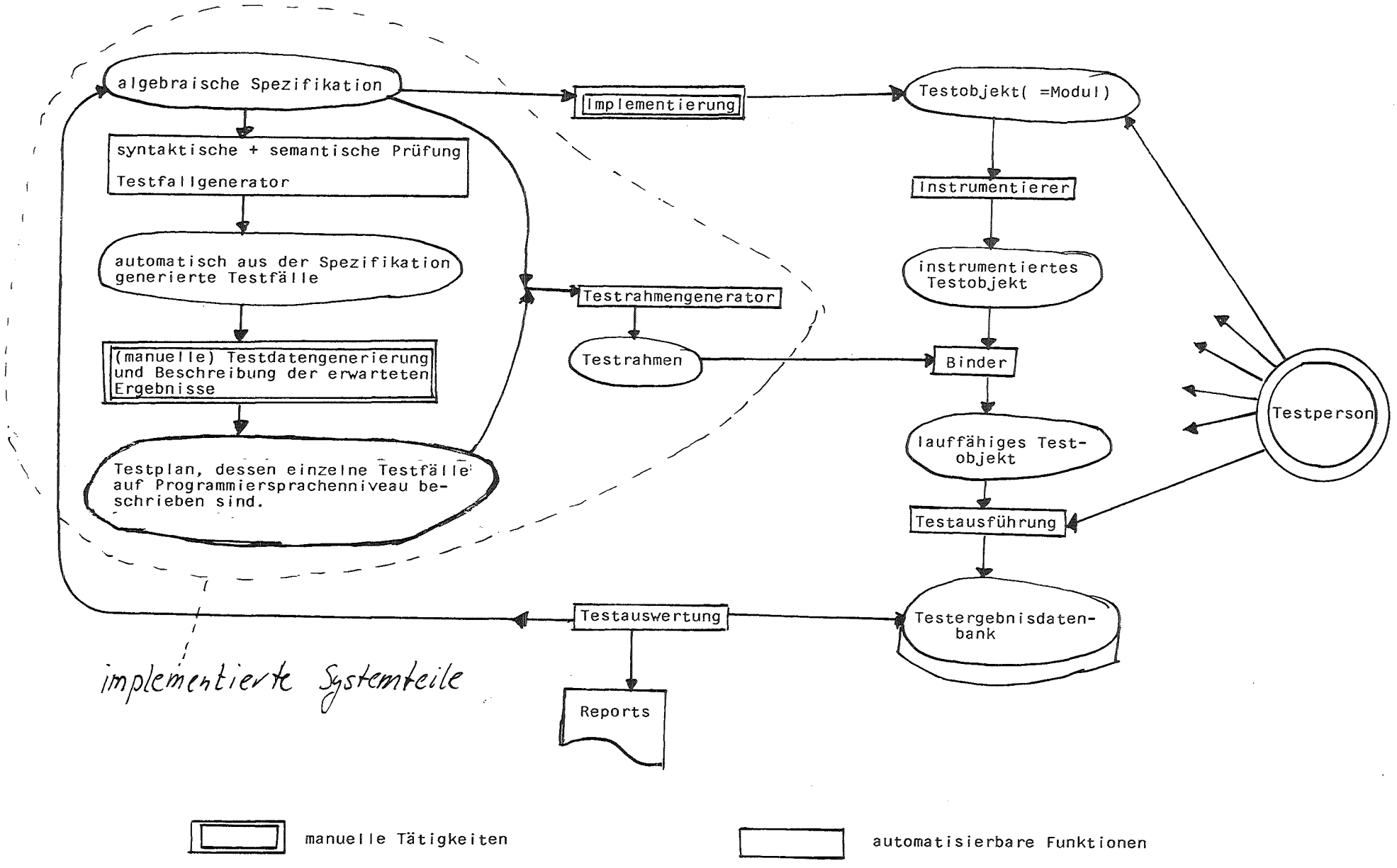


Abb 5.1: Datenfluß im TESTPLAN-System

5.3 Attributierte Grammatiken und Übersetzererzeugende Systeme

Die Übersetzung des Quellprogramms zusammen mit einer syntaktischen und semantischen Prüfung erfordert vom Implementierungsumfang des gesamten TESTPLAN-Systems her den größten Aufwand.
Für eine solche Übersetzerimplementierung wurden zwei verschiedene Techniken betrachtet:

1) Eine konventionelle Übersetzung nach dem Modell des rekursiven Abstiegs;
2) Die Verwendung eines Übersetzererzeugenden Systems.

Wir haben uns für den Fall 2 entschieden, da am Institut für Informatik II der Universität Karlsruhe mit dem GAG-System /Kast 82/ ein leistungsfähiges System zur Übersetzererzeugung zur Verfügung steht. Das GAG-System basiert auf attributierten Grammatiken (AG'en), die in der Definitionssprache ALADIN /Kast 79/ zu formulieren sind. Solchermaßen beschriebene Sprachdefinitionen können automatisch auf Vollständigkeit und formale Korrektheit überprüft werden.

AG'en basieren auf kontextfreien Grammatiken, deren Sprachumfang durch Kontextbedingungen eingeschränkt wird. Ein syntaktisch korrekter Satz gehört genau dann zur definierten Sprache, wenn für alle Attribute die Kontextbedingungen erfüllt sind.

Um einen ersten Eindruck zu vermitteln, wie eine in ALADIN formulierte AG aussieht, wollen wir an dieser Stelle ein kurzes Beispiel aus der Sprachdefinition von TESTPLAN-S anführen. Es handelt sich um die Regel für den 'import_part'.

```
RULE rr102a: import_part ::= 'IMPORT' ( name // ',' ) ';'
(*--------*)
STATIC

(* Eindeutigkeit der Listenelemente bestimmen *)
CONDITION UNIQUE_ELEMS(name.at_name)
MESSAGE "IMPORT-IDENTIFIER MUST BE DISTINCT";

import_part.at_import_list := name.at_name;

CONDITION f_keys_in_list(name.at_name,
                         (INCLUDING module_def_body.at_exlist_in));
  MESSAGE is given within Funktion

import_part.at_import_defs := f_construct_import_defs(name.at_name,
                          INCLUDING module_def_body.at_exlist_in)
END;
```

Eine komplette Beschreibung der AG für die Sprache TESTPLAN-S findet man in /Gmei 82/.

Das GAG-System erzeugt für eine in ALADIN geschriebene AG automatisch die Attributauswerteroutinen. Damit kann der Benutzer dieses Systems von konkreten Implementierungsdetails abstrahieren und sich voll auf die Definition von semantischen Spracheigenschaften konzentrieren.

Wenn man einmal eine lauffähige AG hat, zeigt sich ein weiterer Vorteil dieses automatischen Hilfsmittels: Im Vergleich zu herkömmlichen Compilern lassen sich Modifikationen schnell und zuverlässig durchführen, und man erhält unmittelbar wieder einen lauffähigen neuen Compiler.

Bevor das GAG-System eingesetzt werden konnte, stellte sich die Aufgabe, für die kontextfreie Grammatik der Sprache TESTPLAN-S (vgl. Kap. 8.1) eine äquivalente attributierte Grammatik zu definieren. Um die Verarbeitbarkeit durch das GAG-System sicherzustellen, mußte diese kontextfreie Grammatik zunächst in einem Zwischenschritt in eine äquivalente Grammatik mit der LALR(1)-Eigenschaft überführt werden. Diese Transformation war notwendig, da im GAG-System ein LALR(1)-Parsergenerator /Denc 77/ verwendet wird.

Zusammen mit dem GAG-System ist an der Universität Karlsruhe auch eine AG für die Sprache PASCAL verfügbar. Da die Sprache TESTPLAN-S eine Obermenge von PASCAL ist, konnte diese PASCAL-AG vorteilhaft als Kern der zu entwickelnden AG verwendet werden. Der Umfang der PASCAL-AG (2997 Zeilen) erhöhte sich bei der AG für TESTPLAN-S auf 4769 Zeilen (jeweils mit Kommentaren und Leerzeilen).

Auf der Basis dieser AG wurde dann vom GAG-System ein Compiler erzeugt und getestet. Dieser erzeugte Übersetzer mußte noch von der SIEMENS 7760 der Universität Karlsruhe auf die IBM 3033 des Kernforschungszentrums Karlsruhe übertragen werden. Die Portabilität war aufgrund der Implementierungssprache PASCAL gegeben.

5.4 Stand der Implementierung

Ziel unserer Implementierung ist es, zu zeigen, daß die gegenüber konventionellen Programmentwicklungstechniken neuen Verfahrensteile mechanisierbar sind.

Zu diesen neuen Teilen zählen die automatische Testfall- und Testrahmengenerierung. Um die Testfall- und Testrahmengenerierung sinnvoll demonstrieren zu können, muß zuvor eine syntaktische und semantische Prüfung der Entwurfsbeschreibung erfolgen. Hierfür lassen sich bekannte Techniken aus dem Übersetzerbau einsetzen.

Für die nicht implementierten Systemteile existieren entweder bereits Testwerkzeuge (z. B. RXVP /RXVP 80/ für Instrumentierung, Testausführung und Testauswertung), oder aus einer Implementierung dieser Teile ergeben sich keine zusätzlichen Erkenntnisse. Aus dem letztgenannten Grund wurde auf eine Implementierung

- der Umgebungsanweisungen,
- der Makrovereinbarung sowie
- der Prozedurattribute

verzichtet.

Die Umgebungsanweisungen beschreiben die Kommandoschnittstelle zwischen System und Benutzer. Anstelle der durch die Umgebungsanweisungen beschriebenen komfortablen Schnittstelle bieten wir eine stark vereinfachte Benutzerschnittstelle an. So haben wir für typische Aktionsfolgen (z.B. Übersetzen - Protokollieren - Testfallgenerierung) Kommandoprozeduren bereitgestellt, die es dem Benutzer erlauben, mit dem TESTPLAN-System zu arbeiten, ohne auf die IBM-Steuersprache zurückgreifen zu müssen.

Da die Makrovereinbarung lediglich den Umfang der Entwurfsbeschreibung, nicht aber deren sprachliche Mächtigkeit beeinflußt, wird sie in dieser Implementierung nicht berücksichtigt. Erwähnt werden soll aber noch, daß eine Implementierung der Makrovereinbarung ausschließlich den Modul 'Symbolentschlüsselung' betreffen würde.

Wie im Kapitel 4.2.2 beschrieben unterscheiden wir zwei Ansätze zur automatischen Testfallgenerierung auf Modul- und Prozedurebene. Wir werden uns in dieser Implementierung zunächst auf die Modulebene beschränken. In Analogie zu den Modulattributen ließen sich aber auch die Prozedurattribute implementieren.

5.4.1 Kurzbeschreibung der Moduln

Die in Kapitel 5.1 aufgezählten Moduln sind jeweils als eigenständige PASCAL-Programme realisiert. Die Kommunikation zwischen diesen Programmen erfolgt über Dateien.

Modul Übersetzung:
Dieser Modul hat die Funktionen:

a) Übersetzung der formalen Entwurfsbeschreibung mit syntaktischer und semantischer Prüfung;
b) Separierung aller ASSERT-Statements in einer Datei;
c) Selektion der für die Testrahmengenerierung notwendigen Information in einer Datei.

Die Funktion a erstellt die Grundlagen, ohne die die Funktionen b und c nicht durchführbar sind.

In der Funktion a wird zunächst ein separates Symbolentschlüsselungsprogramm aufgerufen, welches die Grundsymbole in sequentieller Form auf eine Datei schreibt. Von dieser Datei liest der eigentliche Übersetzer die Grundsymbole wieder ein. Anschließend wird vom Parser ein Strukturbaum aufgebaut und gemäß den Regeln der attributierten Grammatik eine semantische Analyse durchgeführt. Eventuelle Fehler sowie ein Übersetzungsprotokoll werden von einem separaten Protokollierprogramm auf eine Datei ausgegeben.

Funktion b schreibt die syntaktisch und semantisch korrekten ASSERT-Statements der Entwurfsbeschreibung in eine Datei. Diese Datei bildet die Grundlage für eine automatische Testfallgenerierung.

Funktion c schreibt alle PASCAL-Programmstücke der Entwurfsbeschreibung auf eine Datei. Diese Datei dient als Eingabe für die Testrahmengenerierung.

Modul 'Testspezifische Funktionen':
Dieser Modul besteht aus den beiden Teilen 'Testfallgenerierung' und 'Testrahmengenerierung'.

Die Funktion 'Testfallgenerierung' erzeugt auf der Basis der im Modul 'Übersetzung' durch Funktion b erstellten Datei automatisch Testfälle. Der dabei verwendete Algorithmus ist in Kapitel 4.2.2.1.1 skizziert. Das Ergebnis der Testfallgenerierung sind Aufrufsequenzen, die wiederum in einer Datei abgelegt werden, wo sie dem Benutzer zur weiteren Verarbeitung zur Verfügung stehen.

Die Funktion 'Testrahmengenerierung' erzeugt auf der Basis der im Modul 'Übersetzung' durch Funktion c erstellten Datei automatisch für jeden Modul der Entwurfsbeschreibung einen Testrahmen. Aufgrund der in Teilaufgabe a durchgeführten syntaktischen und semantischen Prüfungen ist sichergestellt, daß der generierte Testrahmen fehlerfrei übersetzt werden kann. Die erzeugten Testrahmen werden in einer Datei abgelegt.

Die Modulen 'Testauswertung', 'Testausführung' und 'Hilfsfunktionen' sind nicht implementiert.

5.4.2 Implementierungsdaten

Insgesamt besteht das System TESTPLAN aus 38350 Zeilen PASCAL-Programmcode, davon wurden 27477 Zeilen durch das GAG-System generiert.

Das Programmsystem benötigt an der IBM 3033 einen Laufbereich von ca. 1000k Byte.

Wir haben an der IBM 3033 des Kernforschungszentrums im interaktiven Betrieb deutlich längere Antwortzeiten für das System TESTPLAN beobachtet, als dies an der SIEMENS 7760 der Universität Karlsruhe der Fall war.

Die Übertragung des TESTPLAN-Systems von der SIEMENS 7760 auf die IBM 3033 verlief - abgesehen von Compilerfehlern und Beschränkungen des PASCAL-VS-Systems - problemlos.

6. Beispiel und Erfahrungswerte

6.1 Beispiel: Warteschlange

In diesem Kapitel wird das Verfahren TESTPLAN an dem einfachen Beispiel einer Warteschlange erläutert. Dieses Beispiel wurde gewählt, weil die Problemstellung allgemein bekannt ist und der Leser sich damit mehr auf den skizzierten Lösungsweg konzentrieren kann.

Es wird herausgearbeitet, wie Konsistenz und Vollständigkeit einer Entwurfsbeschreibung getestet werden können. Die praktische Erfahrung zeigt, daß der Nachweis der Vollständigkeit einer Entwurfsbeschreibung schwieriger ist als der Konsistenznachweis /Gutt 77/. Das Konsistenzproblem läßt sich ähnlich wie in Programmiersprachen (z. B. durch Datentypen) abhandeln, wogegen der Vollständigkeitsnachweis aufgrund vergessener Systemeigenschaften größere Probleme bereitet.

Meistens lassen sich aus einer Problemstellung die 'elementaren' Systemeigenschaften (Funktionen) noch einfach herausarbeiten. Bereits bei der Kombination solcher Elementarfunktionen ist im allgemeinen keine vollständige Beschreibung mehr gegeben. Auf der Basis von Ausführungssequenzen der elementaren Funktionen wird gezeigt, wie eine vollständige Menge von Testfällen generiert werden kann. Diese Testfälle überdecken neben den spezifizierten auch die nicht-spezifizierten Aufrufsequenzen und beschreiben - unter der Annahme, daß alle Elementarfunktionen gefunden wurden und als Zugriffsfunktionen in einem ADT spezifiziert sind - ein Problem vollständig.

6.1.1 Problembeschreibung

Die zu realisierende "Warteschlange" ist eine Datenstruktur samt Zugriffsfunktionen mit den Eigenschaften:

1) Eine neue Warteschlange ist leer;

2) Eine Warteschlange ist nicht leer unmittelbar nach einer Add-Operation;

3) Eine neue Warteschlange hat kein 'Erstes Element';

4) Von einer leeren Warteschlange kann kein Element entnommen werden;

5) Falls die Warteschlange vor einer Add-Operation leer ist, so liefert eine Front-Operation genau das hinzugefügte Element;

6) Wenn eine Warteschlange vor einer Add-Operation leer ist, dann ist sie auch nach einer Remove-Operation leer, sofern zwischen diesen beiden Operationen keine anderen Operationen durchgeführt werden;

7) Wenn eine Warteschlange nicht leer ist , dann hat eine Add-Operation gefolgt von einer Remove-Operation dieselbe Wirkung wie eine Remove- gefolgt von einer Add-Operation.

In algebraischen Spezifikationen wird die Syntax und Semantik dieser natürlichsprachlichen Beschreibung formalisiert, was dann auf folgende Darstellung führt.

Die syntaktische Beschreibung legt den Definitions- und Wertebereich der Operationen fest. Diese Operationen bilden zusammen mit den zugrundeliegenden Datenstrukturen den Abstrakten Datentyp (ADT) 'Warteschlange'.

SYNTAX:

Funktionsname	Definitionsbereich		Wertebereich
NEW	( )	----->	QUEUE
ADD	(QUEUE,ITEM)	----->	QUEUE
FRONT	(QUEUE)	----->	ITEM
REMOVE	(QUEUE)	----->	QUEUE
EMPTY	(QUEUE)	----->	BOOLEAN

Die semantische Beschreibung spezifiziert den Effekt von Aufrufsequenzen der Operationen eines ADT's.

SEMANTIK:

```
1)     IS_EMPTY(NEW) = true
2)     IS_EMPTY(ADD(q,i)) = false
3)     FRONT(NEW) = error
4a+b)  FRONT(ADD(q,i)) = if IS_EMPTY(q) then i
                                       else FRONT(q)
5)     REMOVE(NEW) = error
6a+b)  REMOVE(ADD(q,i)) = if IS_EMPTY(q)
                          then NEW else ADD(REMOVE(q),i)
```

Dabei sind 'q' und 'i' Variablen des Typs 'queue' bzw. 'item'; 'error' ist ein spezieller Wert.

6.1.2 Entwurfsbeschreibung der Warteschlange in TESTPLAN-S

Eine Entwurfsbeschreibung in der Sprache TESTPLAN-S lehnt sich stark an die zuvor dargestellte algebraische Spezifikation eines ADT an.

Für die obige Syntax- und Semantikbeschreibung finden sich in TESTPLAN-S äquivalente Sprachkonstrukte.

So wird die Syntax durch Deklarationen nachgebildet (in unserer Abb. 6.1 durch Typ- und Prozedurvereinbarungen. Die Operationen des ADT werden mit EXTERNAL vereinbart, weil sie später auf der Grundlage dieser Entwurfsbeschreibung von einem unabhängigen Programmierer implementiert werden und an einer anderen Stelle archiviert werden).

```
0000010001MODULE schlange;
0000020001TYPE pt = @ item;
0000030001
0000040001     item  = RECORD
0000050001             suc,pre : pt;
0000060001             info     : INTEGER;
0000070001             END;
0000080001
0000090001     anker = RECORD
0000100001             first,last: pt;
0000110001             END;
0000120001
0000130001     fehler = (emptyqueue, implerror, noerror);
0000140001
0000200001PROCEDURE add(nutzinfo : INTEGER); EXTERNAL;
0000210001PROCEDURE front(VAR element : item; VAR err : fehler); EXTERNAL;
0000220001PROCEDURE new_ws; EXTERNAL;
0000230001FUNCTION is_empty : BOOLEAN; EXTERNAL;
0000240001PROCEDURE remove(VAR err : fehler); EXTERNAL;
0000250001
0000260001ASSERT new_ws  is_empty  =' entspricht Regel 1',
0000270001       new_ws (front | remove) =' entspricht Regel 3 + 5',
0000280001       new_ws (add)+ is_empty = 'FALSE, entspricht Regel 2',
0000290001       new_ws add remove is_empty = 'TRUE, entspricht Regel 6a',
0000300001       new_ws add front ='Durch add eingefuegtes Element,Regel 4a',
0000310001       new_ws add (add |front)* ='Regel 4b',
0000320001       new_ws (add remove)* = 'Regel 6b';
0000330001
0000340001TESTPLAN
0000351001  TESTBEG
0000360001    (* später ausfüllen *)
0000371001  TESTEND
0000380001ENDMOD.
```

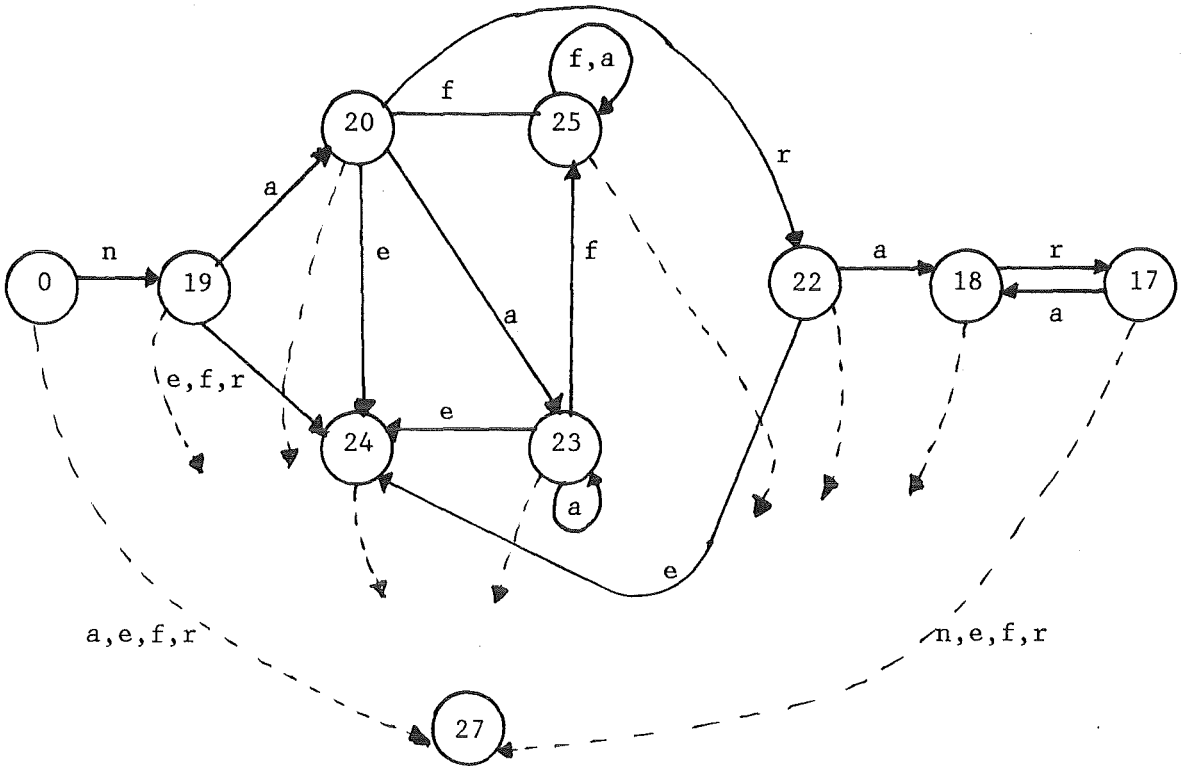
Abb. 6.1: Entwurfsbeschreibung der Warteschlange in TESTPLAN-S

Die Semantik wird im ASSERT-Statement durch reguläre Ausdrücke beschrieben. Der Effekt der regulären Ausdrücke bzw. ihre Beziehungen zu der Semantikbeschreibung der algebraischen Spezifikation lassen sich, eingeschlossen in Hochkommata, als Kommentar formulieren.

Der Testplan bildet den letzten Teil der Entwurfsbeschreibung und ist zu diesem Zeitpunkt noch leer.

6.1.2.1 Generierung von Testfällen aus der Entwurfsbeschreibung

Die automatische Testfallgenerierung basiert auf den regulären Ausdrücken des ASSERT-Statements. Wir nehmen an, daß alle regulären Ausdrücke zusammen das gewünschte Systemverhalten beschreiben. Das Werkzeug TESTPLAN-W konstruiert gemäß Kapitel 4.2.2.1.1 einen deterministischen endlichen Automaten (Abb. 6.2), aus dem sich nach verschiedenen Kriterien (vgl. 4.2.2.1.2) Testfälle generieren lassen.



Legende: Anfangszustand: 0
Endzustände: 17, 19, 20, 22, 23, 24, 25, 27
a (add), f (front), e (is_empty), n (new_ws), r (remove);

Abb. 6.2: Endlicher Automat, in dem alle regulären Ausdrücke integriert sind.

Die gestrichelten Kanten des Automaten wurden aus Gründen der Übersichtlichkeit nicht vollständig gezeichnet. Mit diesen Kanten, die im Zustand 27 enden, sind alle Funktionen assoziiert, die zur Vervollständigung des Automaten notwendig sind.

Der Testfallgenerierungsalgorithmus liefert für den ADT 'Warteschlange' mit dem Testabbruchkriterium 2 (alle Übergänge mindestens einmal ausgeführt) insgesamt 48 Testfälle in Form von Aufrufsequenzen.

Aus diesen Aufrufsequenzen wählen wir aus Platzgründen die zwei Sequenzen

1) (* new_ws add add is_empty = FALSE, entspricht Regel 2 *)
2) (* new_ws add remove remove = NICHT SPEZIFIZIERTE AUFRUF-SEQUENZ *)

aus.

6.1.2.2 Ergänzung der Entwurfsbeschreibung um einen Testplan

Die automatisch ermittelten Aufrufsequenzen bilden das Grundgerüst eines jeden Testfalls. Um dieses Gerüst herum muß nun die testende Person PASCAL-Programmstücke schreiben. Dies bedeutet, daß Prozeduren mit Parametern versorgt und eventuell Ergebnisse mit vorgegebenen Resultaten verglichen werden müssen.

```
0000000001*****************************************************************************
0000000002*                                                                           *
0000000003*                          T E S T P L A N                                  *
0000000004*                                                                           *
0000000005*                                                                           *
0000000006*                                                                           *
0000000007* COMPILER GENERATION DATE :  07/26/82                                      *
0000000008* PROTOCOL GENERATION DATE :  10/26/82                                      *
0000000009*                                                                           *
0000000010* NUMBER OF MESSAGES:                                                       *
0000000011*                                                                           *
0000000012*     INFORMATIONS  :   0 (I)                                               *
0000000013*     ERRORS        :   0 (E)                                               *
0000000014*     SYSTEM ERRORS :   0 (S)                                               *
0000000015*     SYSTEM LIMITS :   0 (L)                                               *
0000000016*                                                                           *
0000000017*                                                                           *
0000000018*****************************************************************************
0000000019
0000010001MODULE schlange(OUTPUT);
0000020001
0000030001     :  (* analog Abb. 6.1 *)
0000330001
0000340001TESTPLAN
0000351001  TESTBEG VAR fehltyp : fehler;
0000360001    TEST(.1.)
0000360101    (* new_ws add add is_empty = FALSE, entspricht Regel 2 *)
0000360201       PASCBEG BEGIN
0000361001       new_ws;
0000362001       add(201);
0000363001       add(202); (* Warteschlange enthält jetzt 2 Elemente *)
0000364001       IF is_empty THEN WRITELN(OUTPUT,'is_empty = FALSE,',
0000365001                        ' d. h. Ergebnis o. K.')
0000366001                   ELSE WRITELN(OUTPUT,' Ergebnis ist nicht o. K.');
0000366101       END PASCEND;
0000367001
0000368001    TEST(.99.)
0000368101    (* new_ws add remove remove = NICHT SPEZIFIZIERTE ABLAUFSEQUENZ *)
0000368201       PASCBEG BEGIN
0000369001         new_ws;
0000369101         add(303);
0000369201         remove(fehltyp);
0000369301         IF (fehltyp = noerror) AND is_empty
0000369401         THEN WRITELN(OUTPUT, 'Ergebnis o. K.')
0000369501         ELSE WRITELN(OUTPUT, 'Ergebnis falsch');
0000369601         remove(fehltyp);
0000369701         IF (fehltyp = emptyqueue)
0000369801         THEN WRITELN(OUTPUT, 'Ergebnis o. K.')
0000369901         ELSE WRITELN(OUTPUT, 'Ergebnis falsch');
0000370001       END PASCEND;
0000371001  TESTEND
0000380001ENDMOD.
```

Abb. 6.3: Ergänzte und durch das TESTPLAN-System überprüfte Entwurfsbeschreibung der Warteschlange.

Für das Warteschlangenbeispiel ergibt sich die in Abb. 6.3 dargestellte erweiterte Entwurfsbeschreibung. Zur Formulierung der PASCAL-Programmstücke können entweder die bereits im Entwurf vorhandenen Datenstrukturen und Prozeduren verwendet werden, oder es können neue hinzugefügt werden.

Als Hilfestellung bei der Formulierung der Testprozeduren können die Kommentare (Text hinter dem '='-Zeichen in der Aufrufsequenz) herangezogen werden, die mit den generierten Aufrufsequenzen assoziiert sind.

Die Ergänzung des Testplans wird zwischen den Schlüsselworten TESTPLAN und TESTEND gemäß den Sprachkonventionen von TESTPLAN-S vorgenommen. Falls zusätzliche Dateien in der Testprozedur eingeführt werden, muß die Parameterliste des Moduls entsprechend ergänzt werden.

6.1.3 Automatische Generierung von Testrahmen

Aus der ergänzten Entwurfsbeschreibung in Abb. 6.3 wird automatisch ein syntaktisch korrekter Testrahmen in der Sprache PASCAL generiert (vgl. Abb 6.4). Der generierte Testrahmen ist zwar syntaktisch korrekt, aber schwer lesbar. Eine Verbesserung der Lesbarkeit ließe sich aber ohne prinzipielle Schwierigkeiten durch ein Formatierprogramm erreichen.

```
PROGRAM
schlange ( OUTPUT ) ; TYPE pt = @ item ; item = RECORD suc , pre : pt ;
info : INTEGER ; END ; anker = RECORD first , last : pt ; END ; fehler
= ( emptyqueue , implerror , noerror ) ; PROCEDURE add ( nutzinfo :
INTEGER ) ; EXTERNAL ; PROCEDURE front ( VAR element : item ; VAR err :
fehler ) ; EXTERNAL ; PROCEDURE new_ws ; EXTERNAL ; FUNCTION is_empty :
BOOLEAN ; EXTERNAL ; PROCEDURE remove ( VAR err : fehler ) ; EXTERNAL ;
PROCEDURE TESTPLAN;
VAR fehltyp : fehler ;
PROCEDURE TEST1;
BEGIN new_ws ; add ( 201 ) ; add ( 202 ) ; IF is_empty THEN WRITELN (
OUTPUT , 'is_empty = FALSE,' , ' d. h. Ergebnis o. K.' ) ELSE WRITELN (
OUTPUT , ' Ergebnis ist nicht o. K.' ) ; END
;
PROCEDURE TEST99;
BEGIN new_ws ; add ( 303 ) ; remove ( fehltyp ) ; IF ( fehltyp =
noerror ) AND is_empty THEN WRITELN ( OUTPUT , 'Ergebnis o. K.' ) ELSE
WRITELN ( OUTPUT , 'Ergebnis falsch' ) ; remove ( fehltyp ) ; IF (
fehltyp = emptyqueue ) THEN WRITELN ( OUTPUT , 'Ergebnis o. K.' ) ELSE
WRITELN ( OUTPUT , 'Ergebnis falsch' ) ; END
;
   BEGIN (*TESTPLAN-BODY*)
   TEST1;
   TEST99;
   END ; (*TESTPLAN-BODY*)
BEGIN TESTPLAN; END.
```

Abb. 6.4: Testrahmen für den Test des ADT 'Warteschlange'

In Kapitel 4.2 wurde das Verfahren TESTPLAN-V als paralleles Entwicklungsschema vorgeschlagen, wobei Testpläne und Testrahmen parallel zu den eigentlichen Testobjekten entwickelt werden. Gegenstand dieses Beispiels war die Entwicklung der Testpläne und Testrahmen aus der Entwurfsbeschreibung. Für die eigentliche Implementierung des Testobjekts, sowie das anschließende Binden und Ausführen des Testobjekts zusammen mit dem Testrahmen lassen sich bekannte Techniken anwenden. Eine detaillierte Ausführung dieser Teile würde den Rahmen dieser Arbeit sprengen.

6.2 Erfahrungswerte

Das TESTPLAN-System wurde bisher anhand einiger aus der Literatur über algebraische Spezifikationen bekannter Beispiele erprobt. Dabei ergaben sich folgende Daten:

	Keller	Keller mit Limit	Warte-schlange	Symbol-tabelle
Rechenzeit für				
Scanner	0.92 s	1.00 s	1.20 s	1.16 s
Übersetzung	1.63 s	2.09 s	3.79 s	3.40 s
Protokollierung	1.09 s	1.27 s	1.81 s	1.65 s
Testfall-generierung	0.78 s	1.64 s	1.43 s	5.24 s
Testrahmen-generierung	0.12 s	0.14 s	0.28 s	0.20 s
Anzahl der erzeugten Testfälle bei				
Alg. 1 (*)	4	6	8	7
Alg. 2 (**)	36	52	48	57
Anzahl der Zustände des Testfallautomaten	8	10	10	10

(*) Alle Endzustände werden mindestens einmal erreicht;
(**) Alle Kanten des Automaten (Operationen) müssen mindestens einmal ausgeführt sein.

Neben der zuvor beschriebenen 'Warteschlange' wurden als weitere Beispiele der 'Keller' (mit und ohne Beschränkung) sowie eine 'blockstrukturierte Symboltabelle' verarbeitet.

Die Erstellung einer Entwurfsbeschreibung in der Sprache TESTPLAN-S ist vom Aufwand her mit einer algebraischen Spezifikation vergleichbar. Als Ergebnis erhält man eine präzise Entwurfsbeschreibung, die sich für eine automatische Testfallgenerierung eignet und gleichzeitig einer Testperson als einzige Arbeitsunterlage dient.
Das Ausarbeiten der automatisch generierten Testfälle zu PASCAL-Programmstücken durch die Testperson erweist sich als relativ einfache Tätigkeit, da der Tester lediglich noch für die Datenversorgung der Prozedur- und Funktionsaufrufe sorgen bzw. die Korrektheit der Ergebnisse überwachen muß.

Der Rechenzeitbedarf für Übersetzung, Protokollierung und Testrahmengenerierung steigt linear mit der Größe (in Grundsymbolen) der Entwurfsbeschreibung.

Betrachtet man den gesamten Softwareentwicklungsaufwand, so ist festzustellen, daß durch den Einsatz des Systems TESTPLAN gegenüber einer konventionellen Vorgehensweise in etwa dieselben Kosten entstehen. Der höhere Aufwand in der Entwurfsphase wird durch einen geringeren Aufwand in der Testphase kompensiert.

Wenn für eine Problemstellung bereits eine algebraische oder dazu vergleichbare Spezifikation vorliegt, so kann man mit geringem Zusatzaufwand diese Problemstellung in TESTPLAN-S umformulieren und daraus dann automatisch Testfälle und Testrahmen generieren.

Da die generierten Testfälle noch manuell ergänzt werden müssen, ist darauf zu achten, daß die Testabbruchkriterien eine 'vernünftige' Anzahl von Testfällen liefern. Diese Randbedingung wird durch die beiden implementierten Testabbruchkriterien erfüllt.

7. Zusammenfassung und Ausblick

In der vorliegenden Arbeit wird eine Möglichkeit vorgestellt, systematische Testmethoden in die frühen Softwareentwicklungsphasen zu integrieren. Die dabei vertretene Grundphilosophie besteht darin, möglichst große Teile zu automatisieren. Bisherige Entwicklungen von automatischen Testwerkzeugen konzentrierten sich weitgehend auf den Programmcode, wogegen in dieser Abhandlung gezeigt wird, daß sich auch die frühen Softwareentwicklungsphasen für derartige Testwerkzeuge eignen.

Die wichtigsten Ergebnisse dieser Arbeit sind:

- die Entwicklung eines neuartigen Algorithmus zur automatischen Testfallgenerierung für eine bestimmte Klasse von Entwurfsbeschreibungen;
- die automatische Ableitung von lauffähigen Testrahmen aus der Entwurfsbeschreibung;
- die vollständige, formale Definition einer Entwurfs- und Testplansprache auf der Grundlage Abstrakter Datentypen.

Im zentralen Teil dieser Arbeit haben wir das integrierte System TESTPLAN zur entwurfsbegleitenden Testplanung und Testdurchführung entwickelt.
Es besteht aus:

- einem Verfahren zur Erstellung des Testplans parallel zur Implementierung,
- einer Sprache zur Formulierung des Softwareentwurfs und der Testpläne und
- einem Werkzeug zum Umsetzen der sprachlichen Formulierung in eine interne Darstellung und zur Generierung der Testfälle sowie deren weitere Verarbeitung.

Verfahren, Sprache und Werkzeug sind aufeinander abgestimmt.

Eine Version des Systems TESTPLAN wurde auf der IBM 3033 des Kernforschungszentrums Karlsruhe in PASCAL implementiert. Diese Implementierung wurde wesentlich erleichtert und beschleunigt durch das Übersetzererzeugende System GAG, das freundlicherweise vom Institut für Informatik II der Universität Karlsruhe zur Verfügung gestellt wurde.

Weiterführende Arbeiten sollten das System TESTPLAN an größeren Beispielen erproben, da bisherige Erfahrungen mit dem System nur für einige kleinere Beispielprogramme vorliegen.
Darüberhinaus wäre es interessant, den eng mit dem Testen verbundenen Wartungsaspekt näher zu untersuchen.
Zukünftigen Arbeiten ist es vorbehalten, einen Teil der hier vorgeschlagenen Techniken auf nicht-sequentielle Programme zu übertragen.

8. Anhänge:

8.1 Anhang A: Sprachdefinition in BNF

In diesem Abschnitt definieren wir die entwurfsorientierte Testplansprache (TESTPLAN-S). Die Beschreibung basiert auf der aus der Algol-60-Beschreibung bekannten Backus-Naur-Form. Darüberhinaus werden folgende Erweiterungen eingeführt :

1) Alle Produktionen werden mit Punkt abgeschlossen.

2) Die Optionalklammern '[...]' haben folgende Bedeutung :

```
A ::= a [ b ] c      wird interpretiert als
A ::= a c | a b c.
```

3) Die Sternoperation im Zusammenhang mit einer runden Klammer bedeutet, daß der geklammerte Ausdruck mindestens einmal zur Anwendung kommt.

```
A ::= a ( b )* c      wird interpretiert als
A ::= a b c | a b b c | a b b b c |   ...    .
```

Ebenso ist eine Anwendung der Sternoperation zusammen mit der Optionalklammer möglich.

```
A ::= a [ b ]* c     wird interpretiert als
A ::= a c | a b c | a b b c |   ...          .
```

4) Listen werden durch (a // b)* beschrieben, wobei a eine beliebige syntaktische Konstruktion ist, während b ein Trennzeichen darstellt.
Ausführlich bedeutet A ::= (a // b)*
dann A ::= a | aba | ababa | ...   .

Wortsymbole, Sonderzeichen und Sonderzeichenkombinationen sind durch Apostroph eingeschlossen. Erläuternde Texte sind durch Klammerpaare '< ... >' begrenzt. Die übrigen Zeichenkombinationen sind Metasymbole der Sprache TESTPLAN-S.
Zeilen, die mit '¢¢' beginnen, markieren die Sprachkonstruktionen, die in einer ersten Implementierung weggelassen wurden.

```
**********************************************************************************
***                  Entwurfs- und Testplansprache                             ***
**********************************************************************************


startproduktion ::=
      ( module_def // ';' )*  '.' | Umgebungsanweisungen .

module_def ::=
      'MODULE' name [ '(' prog_params ')' ] ';' module_def_body
     'ENDMOD'.
```

```
module_def_body ::=
      import_part  export_part  const_def_part  type_def_part
     var_decl_part  prochead_decl_part  module_attr_list.

import_part ::=
      'IMPORT' ( name // ',')* ';' |    .

export_part ::=
      'EXPORT' ( name // ',')* ';' |    .

prochead_decl_part ::=
      [ prochead_decl ]*.

prochead_decl ::=
      ('PROCEDURE' | 'FUNCTION' ) name  proc_type ';' 'EXTERNAL' ';'
¢¢    const_def_part type_def_part var_decl_part
¢¢    proc_attr_list ( 'ENDPROC' | 'ENDFUNC' ) ';'
      .

module_attr_list ::=
      assert_statement  trace_statement  testplan_statement.

trace_statement ::=
      'REFERTO' ( integer_number // ',')* ';' |    .

assert_statement ::=
      'ASSERT' assert_list_part ';'  |    .

assert_list_part ::=
      ( assert_list_elem ',' )*.

assert_list_elem ::=
      regular_expr [ '=' string ].

regular_expr ::=
      ( regular_term )*.

regular_term ::=
      ( regular_faktor // '|' )*.

regular_faktor ::=
      '(' regular_expr ')' [ wiederholung ] | name .

wiederholung ::=
      '+'  |  '*'  |  integer_number [ integer_number ].

testplan_statement ::=
      'TESTPLAN' specifier_statement  trace_statement 'TESTBEG'
      const_def_part  type_def_part  var_decl_part
¢¢    Makrovereinbarungen
      testfaelle 'TESTEND'.

specifier_statement ::=
      'SPECIFIER' string ';'    |    .
```

```
testfaelle ::=
      [ einzel_testfall ]*.

einzel_testfall ::=
      'TEST' '(.' integer_number '.)'  testspec_statement
      trace_statement 'PASCBEG'  ( block |
¢¢                                 block_mit_Makro
      ) 'PASCEND' ';'.

testspec_statement ::=
      'TESTSPEC' string_list ';'    |    .

string_list ::=
      ( string  )*.

¢¢ block_mit_Makro ::=
¢¢       label_decl_part  const_def_part  type_def_part  var_decl_part
¢¢       proc_decl_part  Makrovereinbarungen
¢¢       'BEGIN'  statement_seq 'END'.
¢¢
¢¢ proc_attr_list ::=
¢¢     ( InputAssertionStatement | OutputAssertionStatement |
¢¢     export | import | TestplanStatement | TraceStatement |
¢¢     InputStatement | OutputStatement | CallStatement )*.
¢¢
¢¢ CallStatement ::=
¢¢     'CALL' ( [ 'COND' | 'CYCL' ] ( var_denot // ',' )* )* ';' |    .
¢¢
¢¢ InputStatement ::=
¢¢     'INPUT' (( [ ( var_denot // ',' )* ] 'FROM' name) // ',' )* ';' |  .
¢¢
¢¢ OutputStatement ::=
¢¢     'OUTPUT' ( ( [ ( var_denot // ',' )* ] 'TO' name) // ',' )* ';' | .
¢¢
¢¢ InputAssertionStatement ::=
¢¢     'INASSERT' Assertionbody ';'  |   .
¢¢
¢¢ OutputAssertionStatement ::=
¢¢     'OUTASSERT' Assertionbody ';'   |   .
¢¢
¢¢ Assertionbody ::=
¢¢     expr [ ( 'FOR' 'ALL' | 'FOR' 'SOME' ) var_denot
¢¢     [ '=' expr 'TO' expr [ 'BY' expr ] ] ].
¢¢
¢¢ Makrovereinbarungen ::=
¢¢     'DEFINE' ( ( name [ '(' (name // ',')* ')' ] '='
¢¢     Makrotext '#' ) // ',' )* ';'.
¢¢
¢¢ Makrotext ::=
¢¢     <Beliebiger Text ohne das Endezeichen '#'>.
```

```
*****************************************************************************
***                  PASCAL - Sprachkonstrukte                            ***
*****************************************************************************


block ::=
      label_decl_part  const_def_part  type_def_part  var_decl_part
      proc_decl_part 'BEGIN'  statement_seq 'END'.

prog_params ::=
      ( name //',').

label_decl_part ::=
      'LABEL' ( label_decl//',') ';'    |    .

label_decl ::=
      integer_number.

const_def_part ::=
      'CONST' const_def_list    |    .

const_def_list ::=
      ( const_def ';' )*.

const_def ::=
      name '='  constant.

type_def_part ::=
      'TYPE' type_def_list      |     .

type_def_list ::=
      ( type_def ';' )*.

type_def ::=
      name '='  type_denoter.

var_decl_part ::=
      'VAR'  var_decl_list     |    .

var_decl_list ::=
      ( var_decl ';' )*.

var_decl ::=
      ( name // ',')* ':' type_denoter.

proc_decl_part ::=
      [ proc_decl ]*.
```

```
type_denoter ::=
      type_identifier                                                          |
      record_type                                                              |
      pointer_type                                                             |
      constant '..' constant                                                   |
      '(' enumeration ')'                                                      |
      [ 'PACKED' ] 'ARRAY' '(.' subscr_tp_list '.) 'OF' type_denoter           |
      [ 'PACKED' ] 'SET' 'OF' type_denoter                                     |
      [ 'PACKED' ] 'FILE' 'OF' type_denoter                                    |
      [ 'PACKED' ] 'RECORD' field_list 'END'.

type_identifier ::=
      name.

enumeration ::=
      ( name // ',' )*.

pointer_type ::=
      '@' name

subscr_tp_list ::=
      ( type_denoter // ',' )*.

field_list ::=
      record_section | variant_part |
      record_section ';' field_list  |    .

record_section ::=
      ( name // ',')* ':' type_denoter.

variant_part ::=
      'CASE' tag_field 'OF' variants [ '; '].

tag_field ::=
      [ name ':' ] type_identifier.

variants ::=
      ( variant // ';')*.

variant ::=
      ( case_label //',')* ':' '(' field_list ').

proc_decl ::=
      ('PROCEDURE' | 'FUNCTION') name proc_type ';' ( block | name ) ';'.

proc_type ::=
      [ '(' form_parms ') ] [ ':' type_identifier ].

form_parms ::=
      (form_parm_sect // ';').

form_parm_sect ::=
      ( 'PROCEDURE' | 'FUNCTION' ) name proc_type  |
      [ 'VAR' ] ( name // ',')* ':' type_identifier.
```

```
call_with_params ::=
      identifier '('  expr [ format ] act_params ').

act_params ::=
      [ ',' expr [ format] [ act_params ]  ].

format ::=
      ':' expr [ ':' expr ].

expr ::=
      simp_expr [ rel_opr  simp_expr ].

simp_expr ::=
      [ add_opr ] term [ add_opr  term  ]* .

term ::=
      factor [ mul_opr  factor ]*.

factor ::=
      '(' expr ')'                    |
      call_with_params                |
      'NOT' factor                    |
      var_denot                       |
      identifier                      |
      'NIL'                           |
      '(.' [ element_list] '.)'       |
      integer_number                  |
      real_number                     |
      character                       |
      string.


variable ::=
      var_denot   |  identifier.

var_denot ::=
      variable '.'  name                    |
      variable '(.'  subscript_list '.)'    |  variable '@'.

element_list ::=
      ( element //  ',')*.

element ::=
      expr [ '..' expr ].

subscript_list ::=
      ( subscript // ',' )*.

subscript ::=
      expr.

identifier ::=
      name.

constant_ident ::=
      name.
```

```
constant ::=
      [ add_opr ]  unsigned_const.

unsigned_const ::=
      constant_ident | integer_number | real_number | character |
      string.

rel_opr ::=
      '=' | '<>' | '<=' | '>=' | '<' | '>' | 'IN'.

add_opr ::=
      '+' | '-' | 'OR'.

mul_opr ::=
      '*' | '/' | 'DIV' | 'MOD' | 'AND'.

statement_seq ::=
      ( statement // ';')*.

label ::=
      integer_number.

statement ::=
      label ':'  statement                                                    |
      var_denot ':='  expr                                                    |
      identifier ':='  expr                                                   |
      call_with_params                                                        |
      identifier                                                              |
      'GOTO' integer_number                                                   |
      'BEGIN' statement_seq 'END'                                             |
      'CASE' expr 'OF'  case_limbs [ ';' ] 'END'                              |
      'WHILE' expr 'DO'  statement                                            |
      'REPEAT' statement_seq 'UNTIL'  expr                                    |
      'FOR' identifier ':='  expr  TO_DOWNTO  expr 'DO'  statement            |
      'WITH' variable  WITH_clause                                            |
      'IF' expr 'THEN'  statement [ 'ELSE' statement ]                        |
      ',' variable  WITH_clause                                               |
      .

WITH_clause ::=
      'DO' statement.

case_limbs ::=
      ( case_limb // ';')*.

case_limb ::=
      ( case_label // ',')* ':' statement.

case_label ::=
      constant.

TO_DOWNTO ::=
      ( 'DOWNTO' | 'TO' ).
```

```
*****************************************************************************
***                        Umgebungsanweisungen                           ***
*****************************************************************************

¢¢ Umgebungsanweisungen ::=
¢¢      Syntaxanweisung | Generateanweisung | Enteranweisung |
¢¢      Leaveanweisung | Compileanweisung | Linkanweisung |
¢¢      Allocateanweisung | Executeanweisung | Runanweisung.
¢¢
¢¢ EntityName ::=
¢¢      Modulname [ '.' Prozedurname ] '(' integer_number [ '..'
¢¢      integer_number ] ')'.
¢¢
¢¢ Modulname ::=
¢¢      name.
¢¢
¢¢ Prozedurname ::=
¢¢      name.
¢¢
¢¢ Syntaxanweisung ::=
¢¢      'SYNTAX' ( Dateiname // ',' )*.
¢¢
¢¢ Dateiname ::=
¢¢      <Name gemäß den Dateinamenskonventionen der Implementierungs->
¢¢      <maschine z. B. 'EXAMPLE.DATA'>.
¢¢
¢¢ Generateanweisung ::=
¢¢      'GENERATE' ( EntityName // ',' )*.
¢¢
¢¢ Enteranweisung ::=
¢¢      'ENTER'.
¢¢
¢¢ Leaveanweisung ::=
¢¢      'LEAVE'.
¢¢
¢¢ Linkanweisung ::=
¢¢      'LINK' EntityName [ 'WITH' ( Modulname // ',' )* ].
¢¢
¢¢ Executeanweisung ::=
¢¢      'EXECUTE' EntityName.
¢¢
¢¢ Runanweisung ::=
¢¢      'RUN' EntityName.
¢¢
¢¢ Compileanweisung ::=
¢¢      'COMPILE' ( EntityName // ',' )*.
¢¢
¢¢ Allocateanweisung ::=
¢¢      'ALLOCATE' LogischeFileId  Physikalische_Filebeschreibung.
¢¢
¢¢ LogischeFileId ::=
¢¢      name.
¢¢
¢¢ Physikalische_Filebeschreibung ::=
¢¢      Dateiname.
```

```
*****************************************************************************
***       Grundsymbole                                                    ***
*****************************************************************************


Grundsymbole ::=
    Wortsymbole | Sonderzeichen | Sonderzeichenkombinationen |
    Kommentar | integer_number | real_number | name | string |
    character.

name ::=
    letter [ letter | digit | '_' ]*.

integer_number ::=
    digit [ digit ]*.

digit ::=
    0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9.

letter ::=
    A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q |
    R | S | T | U | V | W | X | Y | Z |
    a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q |
    r | s | t | u | v | w | x | y | z.

string ::=
    ( '''' [ character ]* '''' ).

character ::=
    <Ein Zeichen aus dem implementierten Zeichensatz>.

real_number ::=
    digit [ digit ]* '.' ( digit )* [ ScaleFactor ].

ScaleFactor ::=
    'E' [ ( '+' | '-' ) ] 'digit' [ 'digit' ]*.

Kommentar ::=
    '(*' <Beliebige Zeichenkombination ausser "*)"> '*)'.

Sonderzeichen ::=
    ';' | '.' | ':' | '(' | ')' | '=' | '#' | '<' | '>' | '+' | '-' |
    '*' | '/' | '@' | '|' | ','.

Sonderzeichenkombination ::=
    '<>' | '>=' | '<=' | '..' | '(.' | '.)' | '(*' | '*)'.
```

```
Wortsymbole ::=
    'ALL' | 'ALLOCATE' | 'AND' | 'ARRAY' | 'ASSERT' |
    'BEGIN' | 'BY' |
    'CALL' | 'CASE' | 'COMPILE' | 'COND' | 'CONST' | 'CYCL' |
    'DEFINE' | 'DIV' | 'DO' | 'DOWNTO' |
    'FILE' |
    'ELSE' | 'END' | 'ENDFUNC' | 'ENDMOD' | 'ENDPROC' |
    'ENTER' | 'EXECUTE' | 'EXPORT' | 'EXTERNAL' |
    'FILE' | 'FOR' | 'FROM' | 'FUNCTION' |
    'GENERATE' | 'GOTO' |
    'IF' | 'IN' | 'INASSERT' | 'INPUT' | 'IMPORT' |
    'LABEL' | 'LEAVE' | 'LINK' |
    'MOD' | 'MODULE' |
    'NIL' | 'NOT' |
    'OF' | 'OR' | 'OUTASSERT' | 'OUTPUT' |
    'PACKED' | 'PASCBEG' | 'PASCEND' | 'PROCEDURE' |
    'RECORD' | 'REFERTO' | 'REPEAT' | 'RUN' |
    'SET' | 'SOME' | 'SPECIFIER' | 'SYNTAX' |
    'TEST' | 'TESTBEG' | 'TESTEND' | 'TESTPLAN' |
    'TESTSPEC' | 'TO' | 'TYPE' |
    'UNTIL' |
    'VAR' |
    'WHILE' | 'WITH'.
```

8.2 Anhang B: Ein Testfallgenerierungsalgorithmus auf der Basis regulärer Ausdrücke

Wir gehen von einer Menge regulärer Ausdrücke aus, die alle nach folgendem Muster aufgebaut sind:

$$\underbrace{\text{new (add | remove)}}_{\text{formaler Teil}} = \underbrace{\text{Fehlerzustand}}_{\text{informeller Teil}}$$

Wir unterscheiden bei den regulären Ausdrücken einen formalen und einen informellen Teil. Aus dem formalen Teil werden die Testfälle automatisch abgeleitet, während der informelle Teil im Verlauf der manuellen Erstellung der Testprozeduren und erwarteten Testergebnisse herangezogen wird.

Der Algorithmus 'GEN_TESTFALLAUTOMAT' läuft in 6 Schritten ab und verwendet dabei drei andere Algorithmen, nämlich

- Alg1: Generierung eines Automaten aus einem regulären Ausdruck. (/Goos 81/ S. 517).
- Alg2: Generierung eines deterministischen endlichen Automaten aus einem indeterministischen endlichen Automaten. (/Goos81/ S. 513).
- Alg3: Generierung eines reduzierten Automaten aus einem vollständigen Automaten. (/Goos81/ S. 605 + S. B17 ff.).

Für jeden regulären Ausdruck müssen die Schritte 1 und 1.1 wiederholt werden.

1. Generierung eines endlichen Automaten aus einem regulären Ausdruck nach Alg1.

1.1 Falls ein indeterministischer Automat vorliegt, muß mit Hilfe von Alg2 ein deterministischer Automat generiert werden.

2. Aus den verschiedenen deterministischen endlichen Automaten wird ein einzelner endlicher Automat zusammengebaut. Beim Zusammenbau kann es passieren, daß aus mehreren deterministischen Automaten ein indeterministischer Automat entsteht. So wird nach jedem Hinzufügen eines Teilautomaten Alg2 aufgerufen werden. Der Zusammenbau beginnt mit dem Teilautomaten, dessen regulärer Ausdruck am längsten ist.
   Als Ergebnis liegt schließlich ein deterministischer endlicher Automat vor.

3. Dieser Automat wird zu einem vollständigen Automaten erweitert. Dies erfordert die Einführung eines Zustandes 'undefiniert'.

4. Mit Hilfe der regulären Ausdrücke, von denen der ganze Algorithmus ausgeht, werden die Endzustände des Automaten bestimmt.

5. Der Automat wird mit Hilfe von Alg3 reduziert.

6. Es wird automatisch eine endliche Menge von Pfaden vom Startzustand zu den Endzuständen ermittelt (Pfad = Aufruffolge von Operationen). Der Zustand 'undefiniert' ist per Definition ein Endzustand.

9. Literatur

/Ada 81/ "The Programming Language ADA, Reference Manual"
Lecture Notes in Computer Science, Springer
Berlin, Heidelberg, New York, Bd. 106, 1981.

/Ande 77/ Andersen, T.
"Software Fault-Tolerance : A System Supporting Fault-Tolerant Software".
Proceedings INFOTECH State of the Art Conference on Reliable Software, London 28. 2. - 2. 3. 1977, 1977, p. 141-154.

/Aust 76/ Austin, S.L.; Buckles, B.P.; Ryan, J.P.
"SSL - A Software Specification Language"
Science Application Inc., Huntsville, SAI-77-537-HU, 1976.

/Baue 79/ Bauer, J. A; Finger, A. B.
"Test Plan Generation Using Formal Grammars"
Proc. of 4th Conf. on Software Engineering, Munich 1979, p. 425-432.

/Beic 80/ Beichter, F; Buchegger, O; Fuchs, N. E; Herzog, O.
"SLAN-4, eine Softwareentwurfs- und Spezifikationssprache"
Proceedings Tagung Software Engineering - Entwurf und Spezifikation, German Chapter of the ACM, Berlin, September 1980.

/Boeh 76/ Boehm, B. W.
"Software Engineering"
IEEE Trans. on Computers, Vol. C-25, No. 12, December 1976, p. 1226-1241.

/Boeh 78/ Boehm, B. W; Brown, J. R; Kaspar, H; Lipow, M; MacLeod, G. J; Merrit, M. J.
"Characteristics of Software Quality".
North Holland, TRW Series in Software Technology, Vol. 1,1978.

/Boye 75/ Boyer, R.; Elspas, B.; Levitt, K.
"SELECT - A Formal System for Testing and Debugging Programs by Symbolic Execution".
Proc. International Conference on Reliable Software, Los Angeles, April 1975, p. 234-245.

/Clar 76/ Clarke, L.
"Test Data Generation and Symbolic Execution of Programs as an Aid in Program Validation"
Ph. D. Thesis, Univ. of Colorado, 1976.

/Clar 81/ Clarke, L. A.; Richardson, D. J.
"Symbolic Evaluation Methods for Program Analysis"
p. 264-300, in Muchnik/Jones (Eds) "Program Flow Analysis"
Prentice Hall Software Series, Englewood Cliffs, 1981.

/Comm 82/ Commentz-Walter, B; Frischkorn, H. G.
unveröffentlichter Bericht, Januar 1982.

/Comp 79/ COMPUTER
Volume 12, No. 8, August 1979.

/Darr 78/ Darringer, J. A; King, J. C.
"Application of Symbolic Execution to Program Testing"
COMPUTER, April 1978, p. 51-60

/Davi 73/ Davis, M.
"Hilbert's Tenth Problem is Unsolvable".
Am. Math. Mon., 80, No. 3, March 1973, p. 233- 269.

/Denc 77/ Dencker, P;
"Ein neues LALR-System".
Fakultät für Informatik, Universität Karlsruhe,
Diplomarbeit, 1977.

/Dene 79/ Denert, E;
"Software-Modularisierung"
Informatik Spektrum, Band 2, Heft 4, Oktober 1979, S. 204-218.

/Dene 80/ Denert, E; Hesse, W;
"Projektmodell und Projektbibliothek"
Informatik Spektrum, Band 3, Heft 4, November 1980, S. 215-228.

/Dijk 68/ Dijkstra, E. W;
"The structure of the THE-multiprogramming system".
CACM 11, 5, 1968, S. 341 -346.

/Elme 73/ Elmendorf, W. E.
"Cause-Effect Graphs in Functional Testing"
TR-00.2487, IBM Systems Dev. Division, 1973.

/Elsp 74/ Elspas, B .; Green, M.; Korsak, A.; Wong, P.
unveröffentlichter Bericht, Oktober 1974.

/Endr 75/ Endres, A;
"An Analysis of Errors and their Causes in System Programs",
in IEEE Trans. on Software Engineering , Vol 2, No. 4,
December 1976, p. 327-336.

/Endr 76/ Endres, A.
"Formale Analyse und Verifikation von Programmen".
Dissertation, Universität Stuttgart, Juni 1976.

/Endr 80/ Endres, A.
"Methoden der Programm- und Systemkonstruktion:
Ein Statusbericht".
Informatik-Spektrum, Vol. 3, Bd. 3, S. 156-171, 1980.

/Faga 76/ Fagan, W. E;
"Design and Code Inspections to Reduce Errors in Program
Development"
IBM Systems Journal, 1976, p. 182 - 211;

/Faga 77/ Fagan, W. E;
"Inspecting Software Design and Code"
DATAMATION, October 1977, p. 133 - 144;

/Floy 67/ Floyd, R. W.
"Assigning Meanings to Programs".
Proc. of a Symposium on Applied Math. New York, April 5-7,
1976, Vol 19, Am. Math. Society, p. 19-32, 1967

/Gann 81/ Gannon, J; McMullin, P; Hamlet, R.
"Data-Abstraction Implementation, Specification and Testing"
ACM Transactions on Programming Languages and Systems,
Vol 3, No. 3, July 1981, p. 211-223.

/Geig 79/ Geiger, W; Gmeiner, L; Trauboth, H; Voges, U;
"Program Testing Techniques for Nuclear Reactor
Protection Systems".
COMPUTER, August 1979, S. 10 - 18;

/Gerh 76/ Gerhart, S. L; Yelowitz L.
"Observations of Fallibility in Applications of Modern
Programming Methodologies"
IEEE Trans. on SE, Vol. 2, No. 3, September 1976, p. 195-207.

/Gerh 77/ Gerhart, S. L.
"A Unified View of Current Program Testing and Proving:
Theory and Practice".
INFOTECH State of the Art Conf. on Reliable Software,
London 28.2. - 2. 3. 1977.

/Gerh 79/ Gerhart, S. L.
"Program Validation"
in "Computing Systems Reliability" Ed. T. Anderson and
B. Randell,
Cambridge University Press 1979, S. 66 - 108.

/Gilb 76/ Gilb, T.
"Software Metrics"
Studentlitteratur, p. 185 ff., 1976.

/Gmei 80/ Gmeiner, L; Voges, U;
"Software Diversity in Reactor Protection Systems :
An Experiment".
In Lauber (Ed.) : 'Safety of Computer Control Systems'
S. 75 - 80, 1980, Pergamon Press.

/Gmei 82/ Gmeiner, L;
unveröffentlichter Bericht, August 1982

/Göss 75/ Gössel, M.
"Wahrscheinlichkeitsautomaten und Zufallsfolgen".
Akademie-Verlag, Berlin, WTB Band 109, 1975.

/Good 77/ Goodenough, J. B; Gerhart, S. L.
"Toward a Theory of Testing: Data Selection Criteria"
in "Current Trends in Programming Methodology", Vol. II,
p. 44 - 79, 1977

/Goos 80/ Goos, G.
"Programmkonstruktion".
Skriptum Vorlesung WS 80/81, Institut für Informatik,
Universität Karlsruhe.

/Goos 81/ Goos, G.
"Compilerbau"
Skriptum Vorlesung 1981, Institut für Informatik II,
Universität Karlsruhe.

/Gutt 77/ Guttag,J.
"Abstract Data Types and the Development of Data Structures"
CACM, June 1977, Vol. 20, No. 6, p. 396-404.

/Hanf 70/ Hanford, K. V.
"Automatic Generation of Test Cases"
IBM Systems Journal, No. 4, p. 242-258, 1970.

/Hart 77/ Hartwick, D;
"Test Planning"
AFIPS PRESS, 1977 National Computer Conference, S. 285-294.

/Haus 81/ Hausen, H. L; Müllerburg, M.
"Software Produktionsumgebungen: Entwicklungsstand und Trends"
in Informatik-Fachberichte 43 "Werkzeuge der
Programmiertechnik, G. Goos (ED), 1981, S. 1- 27.

/Hess 81/ Hesse, W.
"Methoden und Werkzeuge zur Softwareentwicklung:
Einordnung und Überblick".
in Informatik-Fachberichte 43 "Werkzeuge der
Programmiertechnik, G. Goos (ED), 1981, S. 113- 154.

/Hetz 73/ Hetzel, W. C. (Editor)
"Program Test Methods"
Prentice Hall 1973.

/Heue 74a/ Heuermann, C. A; Myers, G. J; Winterton, H. J.
"Automated Test and Verification"
IBM Technical Disclosure Bulletin, Vol 17, No. 7, p. 2030-2031

/Heue 74b/ Heuermann, C. A; Myers, G. J; Winterton, H. J.
"Automated Test with Interface Verification Simulation"
IBM Technical Disclosure Bulletin, Vol 17, No. 7, p. 2032-2033

/Homm 80/ Hommel, G. (Editor)
"Vergleich verschiedener Spezifikationsverfahren am Beispiel
einer Paketverteilanlage".
KfK-PDV 186, Kernforschungszentrum Karlsruhe 1980.

/Houg 80/ Houghton, R. C; Oakley, K. A. (Eds.)
"NBS Software Tools Database"
NBSIR 80-2159, US Departement of Commerce, National Bureau of Standards, Oktober 1980.

/Howd 78/ Howden, W. E.
"Evaluation of the Effectiveness of Symbolic Testing"
Software Practice and Experience, Vol. 8, 381-397,1978

/Huan 78/ Huang, J. C.
"Program Instrumentation and Software Testing"
COMPUTER, p. 25-32, April 1978.

/IBM 81/ "PASCAL/VS Programmers Guide".
IBM Corp. Program-number 5796-PNQ, April 1981.

/Info 78/ Proceedings INFOTECH State of the Art Conf. on Software Testing, London 20. - 22. 9. 1978.

/Kast 79/ Kastens, U;
"ALADIN- Eine Definitionssprache für attributierte Grammatiken".
Fakultät für Informatik, Universität Karlsruhe,
Bericht 7/79, 1979.

/Kast 82/ Kastens, U; Hutt, B; Zimmermann, E.
"GAG: A Practical Compiler Generator"
Lecture Notes in Computer Science, Springer, Band 141
1982.

/Keut 81/ Keutgen, H.
"Eine Metrik zur Bewertung der Modularisierung"
Informatik-Fachberichte 50, GI-11.-Jahrestagung, München,
S. 191 - 199, Oktober 1981.

/Kope 76/ Kopetz, H;
"Softwarezuverlässigkeit"
C. Hanser Verlag, München, Wien, 1976.

/Ledg 80/ Ledgard, H.
"ADA- An Introduction & ADA Reference Manual"
Springer Verlag Berlin, Heidelberg, New York, 1980.

/Lisk 77/ Liskov,B.; Snyder, A.; Atkinson, R.; Schaffert, C.
"Abstraction Mechanisms in CLU"
CACM 20, 564-576, 1977.

/Lond 77/ London,R. L.
"Perspectives on Program Verification"
in "Current Trends in Programming Methodology", Vol. II,
p. 151 - 172, 1977.

/Loos 81/ Loos.
"private communication" 13. 11. 81

/Lund 78/ Lundstrom, S. F.
"Adaptive Random Data Generation for Computer Software Testing"
AFIPS 1978, p. 505-512.

/Mats 81/ Matsumoto, Y; Sasaki, O; Nakajima, S; Takezewa, K;; Yamamoto, S; Tanaka, T.
"SWB: A Software Factory"
in Hünke (Ed.)"Software Engineering Environments"
North Holland publ. Co., Amsterdam, January 1981, p. 396-401.

/Mill 77/ Miller, E. F. (Editor)
"Program Testing Techniques".
IEEE No. EHO 130-5, New York 1977.

/Mill 79/ Miller, E.F.
"Test Coverage" Diskussionsbeitrag in INFOTECH State of the Art Report Software Testing.
Vol 1: Analysis and Bibliography.
p. 73 ff. , Maidenhead, England, 1979.

/Mill 81/ Miller, E.F; Henderson, J.B; Mapp, T. E.
"Application of Structural Quality Standards to Software"
Software Engineering Standards Workshop, San Francisco,
IEEE No. 81CH1633-7, p. 51-57, August 1981.

/Myer 78/ Myers, G. J.
"A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections"
CACM, 21, p. 760 - 768, 1978;

/Myer 79/ Myers, G. J.
"The Art of Software Testing"
Wiley, 1979.

/Panz 78/ Panzl, D. J.
"Automatic Software Test Drivers"
COMPUTER, p. 44-50, April 1978.

/Parn 72/ Parnas, D. L.
"A Technique for Software Module Specification with Examples".
CACM 15, 5, 1972, S.330 - 336.

/Rama 74/ Ramamoorthy, C. V.; Cheung, R.C; Kim, K. H.
"Reliability and Integrity of large Computer Programs"
in Lecture Notes in Computer Science, Vol 12, S. 86 - 161, Juni 1974.

/Rama 76/ Ramamoorthy, C. V; Ho, S. F; Chen, W. T.
"On the Automated Generation of Program Test Data"
in IEEE Trans. on Software Engineering , Vol 2, No. 4, December 1976, p. 95-102.

/Rama 76a/ Ramammorthy, C. V; Kim, K. H; Chen, W. T.
"Optimal Placement of Software Monitors Aiding Systematic Testing".
IEEE Trans. on Software Engineering, Vol 1, No. 4, Dec. 1975. p. 403-410.

/Rama 82/ Ramammorthy, C. V; Bastani F. B.
"Software Reliability - Status and Perspectives"
IEEE Trans. on Software Engineering, Vol 8, No. 4, Jul. 1982. p. 354-371.

/Rich 78/ Richardson, D. J.; Clarke, L. A.; Bennet, D. L.
" SYMPLR, SYmbolic Multivariate Polynomial Linearization and Reduction".
TR-78-16, 1978, Dept. of Computer Science, Univ. of Mass.

/Rohl 73/ Rohlfing, H.
"Simula 67"
BI-Hochschultaschenbücher, Bd. 747, 1973.

/RXVP 80/ "RXVP-80 User Manual"
General Research Corp., Santa Barbara, California, Oct. 1980.

/Shaw 79/ Shaw, A. C.
"Software Specification Languages Based on Regular Expressions"
ETH Zürich, Institut für Informatik, Bericht 31, June 1979.

/Snee 82/ Sneed, H. M; Wiehle, H. R. (Hrsg)
"Software-Qualitätssicherung"
GACM-Tagung, 25/26. 3. 1982, Neuherberg
Teubner Verlag, Stuttgart, 1982.

/Stuc 77/ Stucki, L. G.
"New Directions in Automated Tools for Improving Software Quality"
in Current Trends in Programming Methodology, R. T. Yeh(Ed.), Vol. II, P. 80-111, 1977.

/Thay 78/ Thayer, T. A; Lipow, M; Nelson, E. C.
"Software Reliability - A Study of Large Project Reality"
TRW Series of Software Technology, Vol. 2,
North-Holland Publishing Company, Amsterdam, 1978.

/Trau 79/ Trauboth, H.
"Software Testing and Validation Techniques for Highly Reliable Process-Information-Systems"
Computer Audit and Control Conference,
Proceedings 65. INFOTECH State of the Art Conference, Paris, 14-16 November 1979, p. 2/3 - 2/27.

/Voge 80/ Voges, U; Gmeiner, L; v. Mayrhauser, A;
"SADAT - An Automated Testing Tool"
IEEE Trans. on Software Engineering, Vol. 6, No. 3, May 1980, p. 286-290.

/Whit 80/ White, L.J; Cohen, E. I.
"A Domain Testing Strategy for Computer Program Testing"
IEEE Trans. on SE, Vol 6, No. 3, May 1980, p. 247-257.

/Wirt 80/ Wirth, N.
"MODULA-2"
ETH-Zürich, Institut für Informatik, Band 36, March 1980.

/Wirt 81/ Wirth, N.
"Lilith: A Personal Computer for the Software Engineer"
Proceedings of a Course on 'Microcomputer System Design',
Trinity College, University of Dublin, June 1981, p. 333-381.

/Wulf 76/ Wulf, W. A.; London, R. L.; Shaw, M.
"An Introduction to the Construction and Verification of
ALPHARD Programs"
IEEE Trans. on Software Engineering, No. 2, 253-265, 1976.

/Zeda 76/ "PRO/TEST Testdatengenerator"
Informationsschrift der Fa. ZEDA Gesellschaft für
Datenverarbeitung und EDV-Beratung, Wuppertal, 1976.

/Zelk 79/ Zelkowitz, M. V.; Shaw, A. C.; Gannon, J. D.
"Principles of Software Engineering and Design"
Prentice Hall, Englewood Cliffs, N.Y., 1979.