Diplomarbeit

Automatisierung von Strukturtests mit evolutionären Algorithmen

André Baresel
Matrikelnummer: 111 0 44
baresel@informatik.hu-berlin.de

Betreuung:
Herr J. Wegener (DaimlerChrysler AG)
Professor Dr. K. Bothe (Humboldt Universität)

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einführung

1.1 Motivation

Um den korrekten Ablauf von Software zu überprüfen und während der Entwicklung frühzeitig Fehler erkennen zu können, sind Tests in allen Entwicklungsphasen von Softwaresystemen notwendig. Die Bestimmung einer repräsentativen Menge von Testdaten, welche der stichprobenartigen Prüfung des Verhaltens einer Software dienen, ist aufwendig und teuer. Eine Automatisierung von Softwaretests verspricht die Vereinfachung von Teilprozessen des Tests und daraus resultierend die Verringerung von Kosten sowie eine Qualitätsverbesserung der getesteten Softwarekomponenten.

Die *Strukturtests* bilden eine Gruppe von Verfahren, welche sich bei den Festlegungen einer zu bestimmenden Testdatenmenge an der Struktur der zu testenden Software orientieren. Durch das Überdeckungsmaß wird ausgedrückt, in welchem Umfang eine Menge von Testdaten ein bestimmtes Testkriterium erfüllt. Eine hohe Überdeckung des jeweiligen Testkriteriums stellt sicher, daß durch die Testdatenmenge viele Programmteile durchlaufen werden.

Beim dynamischen Strukturtest kann die Erfüllung von Testkriterien während der Ausführung des *Testobjekts* geprüft werden. Dazu sind Anpassungen der Software notwendig. Die Überwachung der Testkriterien kann auch mit Hilfe einer Instrumentierung erfolgen, welche die Protokollierung des *Testverlaufs* übernimmt. Die angepaßte Software muß semantisch äquivalent sein, das heißt sich logisch genauso verhalten, wie das Originalprogramm.

Ziel dieser Arbeit ist die Automatisierung von dynamischen Strukturtests. Dazu wird der folgende Lösungsansatz genutzt. Die Suche nach einer Menge von Testdaten, welche ein Strukturtestkriterium abdeckt, kann als Optimierungsaufgabe formuliert werden, die unter Einsatz von evolutionären Algorithmen gelöst wird. Existierende Vorschläge zu einzelnen Strukturtestkriterien weisen Schwachpunkte auf, wenn die zu testende Software komplex ist. Zielstellung der vorliegenden Arbeit ist daher die Behebung dieser Probleme und eine Verallgemeinerung für alle gängigen Strukturtestkriterien. Für die Instrumentierung einer Software stehen eine Reihe von Methoden zur Auswahl. Entwicklungsziel ist eine für alle Testkriterien nutzbare Instrumentierung.

1.2 Inhalt der Arbeit

Die Arbeit beschreibt den Aufbau und die Arbeitsweise des entwickelten Werkzeugs für den automatischen Strukturtest *ASTELA* (*Automatic Structural Testing with Evolutionary Algorithms*). Dieses System generiert automatisch für eine zu testende Software Eingabedaten, welche die Kriterien gängiger Strukturtestverfahren erfüllen. Das Testwerkzeug umfaßt Komponenten für die Testvorbereitung und die Testdurchführung. In der *Testvorbereitung* wird die zu testende Software für die Testdatengenerierung angepaßt. Aufgabe der Testdurchführung ist die Suche nach einer Testdatenmenge zur Erfüllung des vom Nutzer gewählten Testkriteriums.

Die Arbeit beschreibt auf Basis der Anforderungen an die zu erzeugenden Testdatenmengen der verschiedenen Testkriterien eine Zerlegung des Problems in Optimierungen von einzelnen *Teilzielen*, welche unabhängig voneinander bearbeitet werden können.

Bei der Testdatensuche für die einzelnen *Teilziele* werden evolutionäre Algorithmen eingesetzt, weil diese Methode die besten Erfolgschancen bei der Optimierung von schwierigen Zielfunktionen (viele lokale Optima sowie Plateaus) aufweist. Für die automatische Suche von Testdaten werden zu jedem Testkriterium Zielfunktionen entwickelt. Eine Zielfunktion hängt vom Testkriterium und der zu testenden Software ab. Die Daten für die Berechnung liefern Messungen, welche bei der Ausführung der vorbereiteten Software zusammengestellt werden. Im Kapitel zur automatischen Vorbereitung (*Instrumentierung*) der zu testenden Software werden die notwendigen Quelltexttransformationen erläutert.

Des weiteren stellt diese Arbeit eine effiziente Bewertungsmethode vor, welche für die Beschleunigung des Gesamtprozesses Startwerte für alle Teilziele sammelt. Mit dieser Methode können zufällig erreichte Teilziele erkannt und der Gesamtaufwand reduziert werden, weil durch die Nutzung von guten Eingabedaten für den Optimierungsbeginn die Suche verkürzt werden kann.

Bei der Entwicklung des Systems stand der Test von *AnsiC* Programmen im Vordergrund. Im Vorfeld bestand jedoch eine weitere Anforderung an das Testwerkzeug dahingehend, den Aufwand für Anpassungen an andere Quellsprachen möglichst gering zu halten, da Softwaretests für alle gängigen Programmiersprachen notwendig sind.

1.3 Gliederung

Kapitel 2 führt in die Begriffe des Strukturtests ein und erläutert die Kriterien der verschiedenen Testverfahren sowie die Anforderungen an die zu erzeugenden Eingabemengen. Anhand einer Klassifikation der Testkriterien am Ende des Kapitels werden verschiedene Arten von Teilzielen erläutert. Auf dieser Klassifikation basieren die entwickelten Zielfunktionen. Kapitel 3 beschreibt die Anwendung von evolutionären Algorithmen für den Strukturtest. Hierzu werden die Grundideen dieser Algorithmen dargestellt, ein Überblick zu den Lösungsansätzen in der Literatur gegeben und einige Schwachpunkte identifiziert. Nach der Vorstellung des Konzepts für das System *ASTELA* zu Beginn von Kapitel 4 werden die Lösungsvorschläge für die technische Umsetzung von evolutionären Strukturtests erläutert. Ein wichtiger Bestandteil des Kapitels sind die verschiedenen Zielfunktionen und die Methode zur gleichzeitigen Bewertung mit Hilfe eines speziellen Graphen. Inhalt von Kapitel 5 sind erste Experimente mit *ASTELA*. Eine Diskussion der Zielfunktion folgt im Kapitel 6. Die Zusammenfassung, Wertung und ein Ausblick auf mögliche Verbesserungen sowie Erweiterungen des Testsystems wird im Kapitel 7 vorgenommen.

2 Dynamische Strukturtests

Dynamische Tests umfassen eine Vielzahl von Testverfahren, welche die folgenden gemeinsamen Merkmale besitzen:

- Das übersetzte und ausführbare Programm wird mit konkreten *Testdaten* (Eingabewerten) ausgeführt.
- Es handelt sich um Stichprobenverfahren, das heißt die Korrektheit des getesteten Programms läßt sich durch einen Test nicht beweisen.

Werden die Testdaten auf Basis des Kontroll- oder Datenflusses einer zu testenden Software ermittelt, liegt ein dynamischer Strukturtest vor. Dieser wird in der Literatur häufig *White-Box-Test* genannt [Balzert98].

Es folgt eine Erläuterung der strukturellen Programmelemente, welche bei der Definition der Strukturtestkriterien benutzt werden.

2.1 Grundbegriffe

Die Strukturtestverfahren definieren Kontroll- und Datenflußkriterien über Graphen einer zu testenden Software. Ein *Kontrollflußgraph* zu einem Testobjekt (zu testende Funktion) ist ein gerichteter Graph $G=(N, A, s, e)$ mit N, der Menge der Knoten, einer zweistelligen Relation A zwischen den Knoten (Menge der *Zweige,* ist Teil von $N \times N$) sowie dem *Startknoten* 's' und dem *Endknoten* 'e' ($e, s \in N$) ([Korel90]).

Jeder *Knoten* des Kontrollflußgraphen außer Start- und Endknoten korrespondiert mit mindestens einer Anweisung. Bilden Anweisungen im Testobjekt eine unbedingte Abfolge, können diese einem gemeinsamen Knoten zugeordnet werden. Eine Sonderform bilden bedingte Einzelanweisungen, deren Darstellung durch mehrere Knoten erfolgt.

Jeder *Zweig* des Graphen, also die gerichtete Kante von Knoten i zu Knoten j, entspricht einem möglichen *Kontrollfluß* (Steuerungsübergang) der letzten Anweisung von Knoten i zur ersten Anweisung von j.

Ein Knoten heißt *Verzweigungspunkt oder -knoten*, wenn von ihm mehrere Zweige ausgehen. In diesem Fall sind verschiedene Kontrollflüsse möglich. Knoten dieser Art werden von Schleifenanweisungen, bedingten Einzelanweisungen und Verzweigungsanweisungen gebildet.

Während der Ausführung einer Software wird der Kontrollfluß an Verzweigungspunkten durch die Auswertung von Bedingungen festgelegt. Eine solche Bedingung berechnet für die binäre Verzweigung einen Wahrheitswert, der den Kontrollfluß zum sogenannten *Wahr-* oder *Falschzweig* bestimmt. Eine weitere Form ist die Mehrfachverzweigungsanweisung, zu der Bedingungen für eine Entscheidung zwischen mehreren Zweigen angegeben werden.

Ein *Weg* im Kontrollflußgraphen ist eine Sequenz $\langle (k_i, z_i)^* \rangle$ von *Knoten* $k_i \in N$ und *Zweigen* $z_i \in A$. Er drückt die Ausführung einer Abfolge von Knoten aus. Der Begriff

Segment steht für einen Weg zwischen zwei Verzweigungsknoten, in dem keine weitere Verzweigung enthalten ist. *Segmente* im Kontrollflußgraphen werden in der Literatur auch häufig als *Entscheidungs-Entscheidungswege* (*dd-path* ≙ *decision-decision path*) bezeichnet, weil sie von einer Entscheidung zur nächsten führen.

Ein Weg vom Anfangsknoten zum Endknoten wird Pfad genannt. Der Pfad heißt *möglich* (*feasible*), wenn eine Eingabe existiert, welche bei Ausführung des Testobjekts den Pfad nachvollzieht. Dazu müssen alle Anweisungen der im Pfad enthaltenen Knoten in der vorgeschriebenen Reihenfolge abgearbeitet werden und die Auswertung der Verzweigungsbedingungen zu einem Kontrollfluß durch die festgelegten Zweige führen.

Die *Kontrollflußabhängigkeit* ist eine Beziehung zwischen den Knoten des Kontrollflußgraphen. Ein Knoten *y* heißt *abhängig* von einem Knoten *x* g.d.w. ein Weg von *x* nach *y* führt, und für mindestens einen direkten Nachfolger von *x* gilt, daß alle Wege zum Endknoten den Knoten *y* nicht enthalten (siehe [Schaeff73] "back-dominance").

Der *Datenflußgraph* ist eine Form des Kontrollflußgraphen, bei dem in jedem Knoten *k* die Mengen *DEF*(k), *UNDEF*(k) und *USE*(k) zugeordnet werden. Diese Mengen drücken Eigenschaften von Variablen der zu testenden Software aus. *DEF*(k) ist die Menge aller der Variablen, für die eine Anweisung des Knotens *k* einen Wert zuweist und die nach Abschluß der Anweisungen definiert bleibt. Des weiteren ist *UNDEF*(k) die Menge aller Variablen, die durch den Knoten *k* ihre Gültigkeit verlieren (*undefiniert*), ohne daß ein neuer Wert zugewiesen wird. USE(k) ist die Menge aller der Variablen, deren Werte die Anweisungen des Knotens *k* auslesen, ohne daß sie zuvor ihre Gültigkeit verlieren.

2.2 Gängige Strukturtestverfahren

Üblicherweise werden Strukturtests in kontrollflußorientierte und datenflußorientierte Testverfahren unterteilt [Ligges93].

Typische Vertreter der kontrollflußorientierten Tests sind Anweisungs-, Zweig- und Pfadüberdeckung. Durch die Kriterien dieser Verfahren wird festgelegt, welche Pfade, Wege oder Elemente des Kontrollflußgraphen ausgeführt werden müssen, damit eine Testdatenmenge die vollständige Überdeckung erreicht. Die verschiedenen Methoden des kontrollflußbezogenen Testens unterscheiden sich in der Festlegung der Art und Häufigkeit der Ausführung der Wege oder Programmkonstrukte [Riedm97].

Datenflußbezogene Testverfahren definieren die zugehörigen Kriterien über die im Programmablauf enthaltenen Zugriffe auf Informationsstrukturen [Ligges93], zum Beispiel Variablen, Felder und Listen. Die Interaktion zwischen wertzuweisenden und wertbenutzenden Anweisungen zu prüfen, ist die Idee des datenflußorientierten Testens. Unterschiede zwischen den einzelnen Methoden ergeben sich im auszuführenden Teil der Interaktionen [Riedm97].

2.2.1 Kontrollflußorientierte Verfahren

Kontrollflußorientierte Verfahren benutzen zur Definition der Testkriterien Strukturelemente einer Software, wie Anweisungen, Zweige oder Bedingungen. Die bekanntesten Verfahren sind Anweisungsüberdeckungs-, Zweigüberdeckungs- und Pfadüberdekkungstests. Ihr Ziel ist es, mit einer Anzahl von Testdaten alle vorhandenen Anweisungen, Zweige bzw. Pfade auszuführen [Balzert98]. Ein weiteres Verfahren ist der Bedingungsüberdeckungstest. Die verschiedenen Formen dieser Methode können immer dann sinnvoll angewendet werden, wenn die zu testende Software zusammengesetzte Verzweigungsbedingungen enthält.

Es werden nun die einzelnen Testkriterien unter Angabe eines Überdeckungsgrades (Testwirksamkeit) beschrieben. Der Überdeckungsgrad drückt die Vollständigkeit der Testdatenmenge zum jeweiligen Testkriterium aus. Die tatsächliche Wirksamkeit in bezug auf die Fehleraufdeckung kann damit jedoch nur indirekt erfaßt werden [Riedm97].

Anweisungsüberdeckung

Das Kriterium der Anweisungsüberdeckung hat die Ausführung aller Anweisungen einer zu testenden Software zum Ziel und ist wie folgt definiert:

Definition 1.1 Anweisungsüberdeckung

Es wird die Menge aller Testdaten gesucht, die zur vollständigen Überdeckung aller Programmanweisungen führt. Die Anweisungsüberdeckung wird auch **C_0-Überdeckung** genannt [Riedm97].

Zur Testdatenmenge kann ein Überdeckungsgrad bestimmt werden:

$$\ddot{U}berdeckungsgrad\ C_0 = \frac{Anzahl\ der\ \ddot{u}berdeckten\ Anweisungen}{Anzahl\ aller\ Anweisungen}$$

Zweigüberdeckung

Die Forderung an die Testdatenmenge des Kriteriums für die Zweigüberdeckung ist die Ausführung aller Zweige einer zu testenden Software.

Definition 1.2 Zweigüberdeckung

Es wird die Menge aller Testdaten gesucht, die Pfade durch das Programm erzeugt, in denen alle Zweige des Kontrollflußgraphen mindestens einmal enthalten sind (vollständige Zweigüberdeckung) [Riedm97].

Die Zweigüberdeckung wird als **C_1-Überdeckung** bezeichnet. Der Überdeckungsgrad einer Testdatenmenge kann mit der nachstehenden Formel berechnet werden.

$$\ddot{U}berdeckungsgrad\ C_1 = \frac{Anzahl\ der\ \ddot{u}berdeckten\ Zweige}{Anzahl\ aller\ Zweige}$$

Bedingungsüberdeckung

Die verschiedenen Kriterien der Bedingungsüberdeckung legen die Vollständigkeit einer Testdatenmenge auf Basis der Verzweigungs- und Wiederholungsbedingungen der zu testenden Software fest. Drei bekannte Ausprägungen sind die *einfache Bedingungsüberdeckung*, die *Mehrfach-Bedingungsüberdeckung* und die *minimale Mehrfach-Bedingungsüberdeckung* [Balzert98].

Die Bedingungen der Verzweigungs- oder Wiederholungsanweisungen können aus einer Reihe von Prädikaten bestehen. Den Grundbaustein bilden die atomaren Bedingungen (*Basisterme*), welche durch relationale Terme also *Vergleichsoperationen* beschrieben werden. Die Zusammensetzung, das heißt die logische Verknüpfung von Bedingungen, bildet den Gesamtausdruck einer Verzweigungsbedingung. Jede Verzweigungsbedingung ist damit eine hierarchische Gliederung von logischen Operationen auf der Basis atomarer Bedingungen.

Die *atomare Bedingungsüberdeckung* stellt das einfachste Kriterium dar. Sie wird **C_2-Überdeckung** genannt und folgendermaßen definiert:

Definition 1.3 atomare Bedingungsüberdeckung:

Eine Testdatenmenge erfüllt die **C_2-Überdeckung** g.d.w. es für jede Verzweigung in der zu testenden Software, für jede atomare Bedingung dieser Verzweigung und jeden Wahrheitswert ein Testdatum gibt, bei dessen Ausführung die atomare Bedingung diesen Wert annimmt [Riedm97].

Für die atomare Bedingungsüberdeckung ist nicht garantiert, daß beide Wahrheitswerte der Gesamtbedingung ausgewertet werden, weil nur Teile der Bedingung zur Testdatenbestimmung betrachtet werden. Diese Form der Bedingungsüberdeckung garantiert damit nicht die Zweigüberdeckung, was sich leicht an einem Beispiel zeigen läßt:

Beispiel:

Wenn A>0 **und** B>0 **dann**...

- Die Testdaten (A=2,B=0) und (A=0,B=2) reichen zur atomaren Bedingungsüberdeckung aus.
- In beiden Testfällen wird die Verzweigung mit Falsch ausgewertet !

Das umfangreichste Kriterium in der Gruppe der Bedingungsüberdeckungstests ist die *Mehrfach-Bedingungsüberdeckung*. Sie stellt noch stärkere Anforderungen an die Testdatenmenge als die Kriterien *C_1*- und *C_2-Überdeckung* [Riedm97]. Sie wird häufig als **C_3-Überdeckung** bezeichnet [Pagel94]. Die Definition lautet:

Definition 1.4 Mehrfach-Bedingungsüberdeckung

Eine Testdatenmenge erfüllt die *Mehrfach-Bedingungsüberdeckung* g.d.w. für jede Verzweigungsbedingung jede mögliche Kombination der Wahrheitswerte der atomaren Bedingungen ausgeführt wird [Riedm97].

Beispiel:

Wenn (A **oder** B) **und** (A **oder** C) **dann** ...

Zu den vier atomaren Bedingungen sind $2^4 = 8$ Testfälle zu erzeugen.

Die logischen Kombinationen der Basisprädikate erzeugen auch alle Wahrheitswerte des Gesamtterms, wodurch die Zweigüberdeckung in das Kriterium eingeschlossen ist.

Eine Abschwächung des genannten Kriteriums ist die *minimale Mehrfach-Bedingungsüberdeckung*. Sie ist wie folgt definiert:

Definition 1.5 minimale Mehrfach-Bedingungsüberdeckung

Eine Testdatenmenge erfüllt die *minimale Mehrfach-Bedingungsüberdeckung* g.d.w. für jede Verzweigungsbedingung und jede enthaltene *n*-stelligende logische Verknüpfung alle diejenigen Kombinationen von Wahrheitswerten der Operatoren getestet werden, für die eine Wertänderung eines Operators den Wahrheitswert der Verknüpfung ändern kann [Riedm97].

Zweistellige logische Operationen müssen folgendermaßen getestet werden:

UND(t1,t2) : (Wahr,Wahr) (Wahr,Falsch) (Falsch,Wahr)
ODER(t1,t2) : (Falsch,Falsch) (Wahr,Falsch) (Falsch,Wahr)

Beispiel:

Wenn (A **oder** B) **und** (A **oder** C) **dann** ...
Der Gesamtterm besitzt die folgende hierarchische Struktur:
T_1 = *(A oder B)* ; T_2 = *(A oder C)* ; $T = t_1$ *und* t_2

Die zu erzeugenden Testfälle sind für:
T : (wahr,wahr); (wahr,falsch); (falsch,wahr)
T_1 : (falsch,falsch) (wahr,falsch) (falsch,wahr)
T_2 : (falsch,falsch) (wahr,falsch) (falsch,wahr)

Segment-Überdeckung

Zu den Testmethoden der Segment-Überdeckung zählt die *LCSAJ*-Überdeckung (*linear code sequence and jump*). Das ist eine Gruppe der Strukturtestverfahren, die sich bei der Bildung einer Testdatenmenge nicht nur an einzelnen Programmelementen, sondern an Wegen im Kontrollflußgraphen orientieren. Zunächst sei die einfachste Form, die Segmentpaarüberdeckung (**C_{sp}-Überdeckung**), definiert:

Definition 1.6 Segmentpaarüberdeckung

Eine Testdatenmenge erfüllt die **C_{sp}-Überdeckung** g.d.w. es für jedes Paar von Segmenten *q* und *r*, die im Kontrollflußgraphen direkt aufeinanderfolgen, ein Testdatum gibt, das die Folge $q \circ r$ ausführt (das heißt die Folge ist Teil des durchlaufenen Pfades) [Riedm97].

Eine Verallgemeinerung der **C_{sp}-Überdeckung** bildet das nächste Kriterium durch die Zusammenstellung von *n* aufeinanderfolgenden Segmenten.

Definition 1.7 Segmentfolgenüberdeckung

Eine Testdatenmenge erfüllt die **C_s(n)-Überdeckung** g.d.w. es für jedes *n*-Tupel von Segmenten $S_1, \ldots, S_n$, die im Kontrollflußgraphen direkt aufeinanderfolgen, ein Testdatum gibt, das die Folge $S_1, \ldots, S_n$ ausführt (d.h. die Folge ist Teil des durchlaufenen Pfades) [Riedm97].

Eine Variante der $C_s(n)$-Überdeckung ist die *LCSAJ*-Überdeckung. Sie orientiert sich bei der Bildung der Segmentfolgen an den Sprunganweisungen in der zu testenden Software. Zu den offensichtlichen, expliziten Sprüngen wie Schleifen- oder Programmabbruch kommen solche, die durch strukturierte Programmelemente, wie zum Beispiel Verzweigungs- und Schleifenanweisungen implizit generiert werden.

Die Definition der *LCSAJ*-Wege erfolgte in [Hennel76] anhand der Quelltextstruktur (*Zeilennumerierung*) und bedarf deshalb einer Anpassung, die eine Festlegung für alle Programmiersprachen gestattet. Dazu werden die Knoten des Kontrollflußgraphen entsprechend der Anweisungsreihenfolge im Programmquelltext numeriert, so daß eine Identifikation von Sprüngen aus den Informationen des Kontrollflußgraphen möglich ist. Optimierungen des Compilers in Form einer Umstellung der Anweisungen werden bei der Bildung der *LCSAJ*-Wege nicht betrachtet.

Definition 1.8 LCSAJ-Weg

Ein *Sprung* liegt immer dann vor, wenn ein Zweig von Knoten i zu Knoten j existiert und $i \neq j+1$ gilt.

Startknoten sind der Eintrittspunkt des Testobjekts sowie alle Knoten zu denen ein Sprung führt. *Endknoten* sind alle Knoten von denen ein Sprung ausgeht.

Eine *LCSAJ* besteht aus einem Tripel (*Startknoten*, *Endknoten*, *Sprungziel*) derart, daß der Weg zwischen Start- und Endknoten keine Sprünge enthält. Jeder Knoten mit einer Sprunganweisung ist ein *LCSAJ*.

Definition 1.9 LCSAJ-Überdeckung

Eine Testdatenmenge erfüllt die LCSAJ-Überdeckung g.d.w. jeder LCSAJ-Weg durch mindestens ein Testdatum ausgeführt wird.

Strukturierte Pfadüberdeckung

Die umfangreichste Forderung für Strukturtests ist die Ausführung aller möglichen Pfade in der zu testenden Software. Das entsprechende Kriterium heißt *Pfadüberdeckung* (*C_7-Überdeckung*). Diese Methode ist jedoch wegen der unpraktikabel hohen Anzahl von Pfaden in realen Programmen nicht sinnvoll einsetzbar. Für eine Software mit Schleifen ohne obere Grenze der Iterationszahl gibt es zudem unendlich viele Wege.

Die *strukturierte Pfadüberdeckung* führt eine Beschränkung des Tests auf eine endliche Zahl von Pfadklassen durch Schleifen ein. Gesucht wird die Testdatenmenge, die zu jeder Pfadklasse mindestens einen Repräsentanten (*Pfad*) ausführt.

Für die Bildung von Pfadklassen wird ein Äquivalenzkriterium festgelegt (aus [Riedm97]):

Definition 1.10 k-Äquivalenz

Zwei Pfade im Kontrollflußgraphen heißen ***k-äquivalent***, wenn für jede Schleife gilt:

Sie durchlaufen die Schleife...

1. weniger als k-mal und sind identisch (bzw. k-äquivalent für innere Schleifen)

oder

2. mindestens k-mal, wobei
   - die ersten k-1 Durchläufe identisch (k-äquivalent) sind,
   - der k-te Durchlauf identisch (k-äquivalent) ist, sich aber darin unterscheidet, daß eine Kante zum Schleifeneingang fehlt, wenn die Schleife genau k-mal durchlaufen wird,
   - weitere Durchläufe (k+1,...) beliebig aussehen oder nicht vorhanden sind.

Die *strukturierte Pfadüberdeckung*, welche auch als **C_i(k)-Überdeckung** bezeichnet wird, kann damit wie folgt definiert werden:

Definition 1.11 strukturierte Pfadüberdeckung:

Für k>0 erfüllt eine Testdatenmenge die **C_i(k)-Überdeckung** g.d.w. die aus den Testdaten resultierende Menge von ausgeführten Pfaden mindestens ein Element aus jeder Klasse von k-äquivalenten Pfaden enthält [Riedm97].

Eine abgeschwächte Form der strukturierten Pfadüberdeckung ist die **C_{GI}-Überdeckung** oder *Boundary-Interior-Coverage***.** Dazu wird die **schwache '2'-Äquivalenz** definiert:

Definition 1.12 Schwache '2'-Äquivalenz

Zwei Pfade heißen **schwach '2'-äquivalent,** wenn für jede Schleife gilt:

Sie durchlaufen die Schleife...

1. höchstens einmal und sind identisch (bzw. innere Schleifen schwach 2-äquivalent) *oder*
2. mindestens zweimal, wobei der erste Durchlauf beider Pfade identisch (schwach 2-äquivalent) ist.

Definition 1.13 Boundary-Interior-Coverage

Eine Testdatenmenge erfüllt die **C_{GI}-Überdeckung** g.d.w. die resultierende Menge von ausgeführten Pfaden Elemente aus jeder Klasse von schwach 2-äquivalenten Pfaden enthält [Riedm97].

Der Überdeckungsgrad für die strukturierten Pfadtests läßt sich wie folgt bestimmen:

$$\ddot{U}berdeckungsgrad = \frac{Anzahl\ der\ \ddot{u}berdeckten\ Pfadklassen}{Anzahl\ aller\ Pfadklassen}$$

2.2.2 Datenflußorientierte Verfahren

Datenflußorientierte Testverfahren stellen Forderungen an die zu bildende Testdatenmenge auf Grundlage des Datenflußgraphen. Ein solcher Graph ist ein modifizierter Kontrollflußgraph mit zusätzlichen Angaben über Wertzuweisungen (*DEF*), Wertbenutzungen (*USE*) sowie das Ungültigwerden (*UNDEF*) von Variablen (Definitionen siehe Abschnitt 2.1).

Das folgende Kriterium "*alle Definitionen*" fordert für die Vollständigkeit eines Tests, daß jeder zugewiesene Wert einer Variablen (*DEF*) mindestens einmal in einem Ausdruck benutzt (*USE*) wird.

Definition 1.14 "alle Definitionen" (all-defs)

Eine Testdatenmenge erfüllt das Kriterium *alle Definitionen* g.d.w. für jede Variable und jede Definition dieser Variablen mindestens ein Pfad ausgeführt wird, bei dem nach dieser Definition eine Benutzung des Variablenwertes folgt (darf zwischendurch nicht neu definiert oder ungültig werden).

Eine Forderung, alle Paare von Definition und Benutzung einer Variablen zu testen, umfaßt das folgende Kriterium:

Definition 1.15 "alle DU-Interaktionen" (All-DefUse-Chains)

Eine Testdatenmenge erfüllt das Kriterium *alle DU-Interaktionen* g.d.w. für jede Variable mindestens ein Weg zwischen jeder Definition und jeder von dieser Definition erreichbaren Benutzung der Variable ausgeführt wird, auf dem die Variable keinen neuen Wert erhält, benutzt oder ungültig wird.

Das Kriterium "*alle-Referenzen*" testet, ob alle Wege zwischen der Benutzung einer Variablen (z.B. in einer Verzweigungsbedingung) und der Definition der gleichen Variable ausgeführt werden können.

Definition 1.16 "alle Referenzen" (All-Uses)

Eine Testdatenmenge erfüllt das Kriterium *alle Referenzen* g.d.w. für jede Variable, jede Definition der Variablen in Knoten *d*, jede davon zu erreichende Referenz der Variable in Knoten *r* und für jeden von diesem Knoten ausgehenden Zweig *z* mindestens ein Pfad ausgeführt wird, der in Reihenfolge die Knoten *d*, *r* und den Zweig *z* enthält, ohne daß zwischen Knoten *d* und *r* die Variable einen neuen Wert erhält, benutzt oder ungültig wird.

2.3 Klassifikation nach Zielstellungen

Anhand der Zielstellungen der verschiedenen Testverfahren kann eine Klassifikation vorgenommen werden. In der vorliegenden Arbeit werden Verfahren unterschieden, die das Erreichen bestimmter *Knoten* fordern und solchen, welche die Ausführung bestimmter *Pfade* verlangen. Des weiteren gibt es einige Verfahren, die diese Forderungen kombinieren. Das sind die *knoten-weg-orientierten Verfahren*, welche das Erreichen bestimmter Knoten und die Verfolgung eines festgelegten Weges beinhalten sowie die *knoten-knotenorientierten* Verfahren mit dem Anspruch der Ausführung

bestimmter Knotenpaare in Reihenfolge. Bei den letztgenannten Testmethoden können die erlaubten Pfade eingeschränkt werden.

Knotenorientierte Verfahren

Zu den knotenorientierten Verfahren zählt die Anweisungsüberdeckung. Es wird die Ausführung aller Anweisungen angestrebt, was gleichbedeutend mit der Suche nach Pfaden durch alle Knoten des Kontrollflußgraphen ist. Zielstellung sind damit die Knoten des Graphen.

Sonderformen der knotenorientierten Verfahren sind die Bedingungsüberdeckung und Zweigüberdeckung, da die Zielstellung zusätzliche Forderungen enthält. Bei der Zweigüberdeckung zählt neben dem Erreichen eines bestimmten Knotens die Ausführung einer bestimmten Verzweigung dieses Knotens. Die Bedingungsüberdeckung wird über die Verzweigungen der zu testenden Software definiert. Neben dem Erreichen des Verzweigungsknotens wird der Ergebniswert bestimmter Teilbedingungen als Zielstellung betrachtet.

Pfadorientierte Verfahren

Pfadorientierte Verfahren sind die Formen der strukturierten Pfadüberdeckung. Gesucht werden Testdaten, welche alle Pfade beziehungsweise je ein Element aller Pfadklassen ausführen. Zur Festlegung der Pfadklassen eignet sich eine Zerlegung in Wegstücke. Die in einer Klasse enthaltenen Pfade können damit durch die Beschreibung der Reihenfolge, gegebenenfalls der Wiederholung von Wegstücken, spezifiziert werden.

Knoten-Wegorientierte Verfahren

Zu den knoten-wegorientierten Testverfahren gehören die verschiedenen Arten der Segmentüberdeckung und das LCSAJ-Kriterium. Die einzelnen Teilziele, für die Testdaten gesucht werden, umfassen bei diesen Methoden jeweils einen zu erreichenden Knoten des Kontrollflußgraphen und ein an diesem Knoten beginnendes Wegstück.

Knoten-Knotenorientierte Verfahren

Zu den *knoten-knotenorientierten* Testverfahren gehören alle Testmethoden, deren Ziel eine Testdatenmenge ist, die Pfade durch bestimmte Paare von Knoten ausführt. Dies gilt für die beiden Datenflußkriterien "*alle DU-Interaktionen*" und "*alle Referenzen*". Die Zielstellung kann Einschränkungen der Wege zwischen den beiden Knoten einschließen.

Das Datenflußkriterium "*alle Definitionen*" ist eine Sonderform, die sich sowohl zu den knoten-knotenorientierten wie auch den knoten-wegorientierten Methoden zuordnen läßt. Die Zielstellung fordert neben dem Erreichen bestimmter Knoten die Ausführung eines Knotens aus einer Menge.

3 Evolutionäre Strukturtests

Die Grundidee des evolutionären Strukturtests ist die Konstruktion einer Optimierungsaufgabe für das Problem der Bestimmung einer Testdatenmenge zum jeweiligen Testkriterium. Zur Lösung dieser Aufgabe werden evolutionäre Algorithmen eingesetzt. Solche Algorithmen haben sich schon in vielen unterschiedlichen Anwendungsfeldern für die Lösung komplexer Optimierungsaufgaben bewährt (zum Beispiel [Michal92] und [Schwefl77]).

Ein Lösungsansatz zur Erzeugung von Testdaten für die Anweisungs- und Zweigüberdeckung wird bereits in [Jones96] beschrieben. Der dort vorgestellte Ansatz bedarf jedoch einiger Erweiterungen, damit die gängigen Strukturtestverfahren ausgeführt und komplexere Programmstrukturen getestet werden können (siehe Abschnitt 3.3 "Arbeiten zur Thematik").

Für den evolutionären Strukturtest bilden die Programmparameter der zu testenden Software den Suchraum für eine Testdatenmenge, die eine Ausführung aller durch das Testkriterium beschriebenen Elemente der Software erlaubt. Als *Teilziel* der Suche wird die Ausführung bestimmter Teile der Software, wie zum Beispiel Anweisungen oder Pfade, bezeichnet.

Durch die Festlegung einer *Zielfunktion* besteht die Möglichkeit die einzelnen Teilziele mit Hilfe eines evolutionären Kreislaufs nacheinander oder parallel zu *optimieren*. Das Optimum der Zielfunktion entspricht dabei der Ausführung der spezifizierten Programmelemente. Ein Teilziel gilt als *erreicht*, wenn ein optimales Testdatum gefunden wurde. Die Testdatenmenge für ein Testkriterium wird durch die Zusammenfassung der Testdaten aller *erreichten* Teilziele gebildet. Der Überdeckungsgrad des Testkriteriums läßt sich aus der Anzahl der erreichten Teilziele entsprechend der Formeln aus Kapitel 2 ableiten.

Jedes *Individuum* eines evolutionären Kreislaufs stellt ein Eingabedatum der zu testenden Software dar. Das Testobjekt wird mit jedem Individuum (*Testdatum*) ausgeführt. Die Zielfunktion nimmt eine Bewertung des Individuums in bezug auf die Annäherung an ein Teilziel vor. Die Bestimmung des Zielfunktionswertes erfolgt über die bei der Ausführung des Testobjektes durchlaufenen Anweisungen und Verzweigungen.

Der erste Abschnitt dieses Kapitels erläutert das Konzept evolutionärer Algorithmen und deren Nutzung für ein Optimierungsproblem. Auf dieser Grundlage wird im Abschnitt 3.2 die Umsetzung eines evolutionären Testprozesses beschrieben. Abschnitt 3.3 beschäftigt sich mit bestehenden Lösungsansätzen zur Nutzung evolutionärer Algorithmen für Strukturtests. Nach der Erläuterung der Grundideen und Ergebnisse der verschiedenen Arbeiten folgt eine Bewertung. Am Ende dieses Abschnitts werden vor allem Schwachpunkte bestehender Lösungsansätze aufgedeckt, die es bei der Entwicklung des *ASTELA*-Systems zu beheben galt.

3.1 Evolutionäre Algorithmen

Evolutionäre Algorithmen sind stochastische Suchstrategien, die das natürliche Phänomen der biologischen Evolution nachbilden. Sie basieren auf den Gedanken der Vererbung und Darwins Theorie der natürlichen Auslese *(Survival of the Fittest)* [Darwin, 1859].

Darwin formuliert in seinen Arbeiten drei Grundgedanken als zentrale Mechanismen der Evolution. Die natürliche Auslese (*Selektion*) dient der Auswahl der überlebenden Individuen nach bestimmten Kriterien. *Variationen* innerhalb der Individuen einer Art schaffen unterschiedliche Anpassungen an einen Lebensraum. Damit haben einige Individuen bessere Überlebenschancen als andere. Die dritte Grundansicht Darwins war, daß sich erbliche Varianten, die sich im Kampf ums Überleben bewährt haben, in den Folgegenerationen wiederfinden. So können sich kleine Variationen der Individuen einer Art quasi addieren und im Laufe der Generationen zur Vervollkommnung und Optimierung von Eigenschaften der Lebewesen führen [Schönb94].

Die Evolution kann als Suchstrategie interpretiert werden, bei der die Individuen einer Art versuchen, sich optimal an die jeweiligen Lebensräume anzupassen. Evolutionäre Algorithmen übertragen (*Adaption*) diese Mechanismen zur Optimierung technischer Systeme. Dementsprechend werden die Begriffe aus der Evolutionstheorie und der Genetik verwendet, wie beispielsweise *Selektion*, *Individuum* und *Fitneß*.

Evolutionäre Algorithmen sind durch ein iteratives Vorgehen gekennzeichnet. Jede Iteration umfaßt eine Generation von Individuen. Sie enthält damit eine Reihe von Lösungsvorschlägen. Ein Individuum repräsentiert eine zulässige Einstellung für die Variablen des zu optimierenden Systems und wird durch die Zielfunktion bewertet. Diese Bewertung spiegelt das Optimierungsproblem in einer mathematischen Funktion wieder, wobei als Lösung die Minimierung oder Maximierung angestrebt wird.

Evolutionäre Algorithmen beruhen auf den Grundprinzipien *Selektion*, *Rekombination* und *Mutation*. Die Selektion trifft die Auswahl der Individuen für die Weiterentwicklung. Es wird festgelegt, welche Individuen sich stark vermehren und welche weniger. Eine Selektion für die Erzeugung von Nachkommen basiert auf der Bewertung der Individuen (*Fitneß*). Durch die Rekombination von guten Individuen sollen besser angepaßte Lösungen hervorgebracht werden. Mit Hilfe der Mutation werden Individuen zufällig verändert, um neue oder im Laufe der Suche verlorengegangene Eigenschaften zu erzeugen.

Die Bildung einer neuen Generation von Individuen bildet den Abschluß des evolutionären Kreislaufs. Viele der existierenden Algorithmen lassen eine Parameterisierung dieses Teilprozesses zu. Es gilt zu entscheiden, welche der Individuen in die neue Generation einfließen und welche entfallen.

Im Vergleich zu konventionellen Optimierungsverfahren verwenden evolutionäre Algorithmen stochastische Operatoren und realisieren eine parallele Durchmusterung des Suchraumes durch den Einsatz vieler Lösungsvorschläge. Die Wahrscheinlichkeit in

lokalen Optima zu verbleiben, ist gegenüber konventionellen Verfahren vergleichsweise gering, wie ein Vergleich der Methoden in [Schwefl77] zeigt.

Im weiteren wird der allgemeine Aufbau evolutionärer Algorithmen erläutert. Der Abschnitt endet mit einer Charakterisierung von zwei konkreten Ausprägungen dieser Suchstrategie aus der Literatur.

3.1.1 Grundlagen evolutionärer Algorithmen

Der in den evolutionären Algorithmen ablaufende Optimierungsprozeß läßt sich, wie Abbildung 3.1 zeigt, in Teilschritte zerlegen.

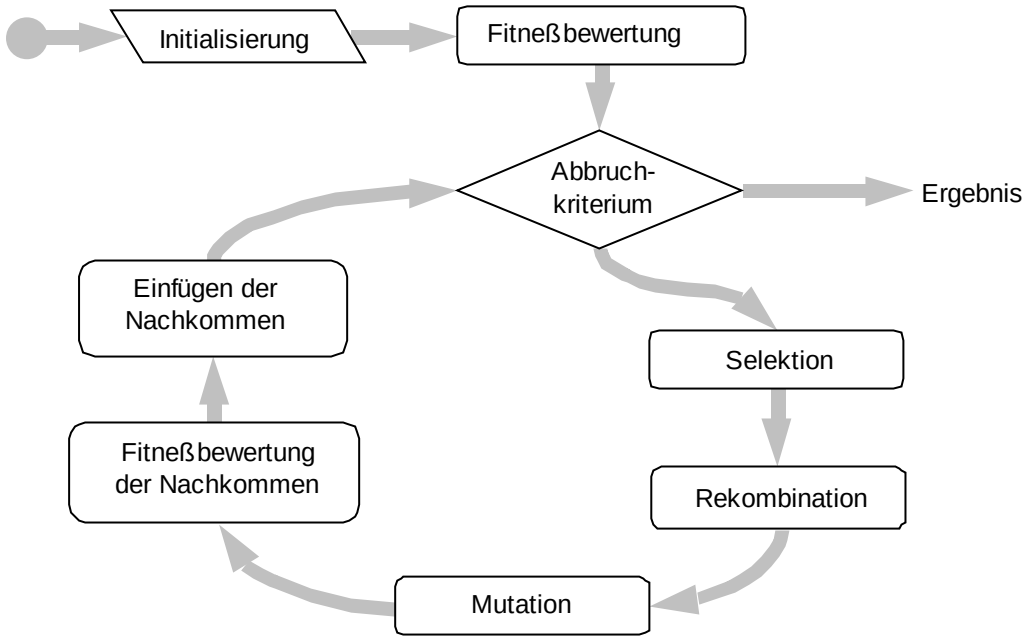


Abbildung 3.1: Aufbau evolutionärer Algorithmen [nach Wegener, 1999]

Die Initialisierung eines evolutionären Suchprozesses beginnt mit der Bildung einer zufälligen Startpopulation. Sind bereits gute Startwerte für die Optimierung bekannt, können diese in die erste Generation von Individuen einfließen und damit den Prozeß beschleunigen.

Eine Zielfunktion dient der Bewertung der Individuen und muß entsprechend des Optimierungsproblems festgelegt werden. Diese Funktion berechnet die Güte eines Lösungsvorschlags (Individuum) im Hinblick auf das Teilziel. Eine geeignete Bewertung der Individuen ist entscheidend für den Erfolg der evolutionären Optimierung. Sie muß so gestaltet sein, daß Individuen, die weit von einer optimalen Lösung entfernt liegen, eine schlechtere Bewertung erhalten, als solche, die näher am Optimum liegen [Sthamer96].

Im ersten Schritt des Algorithmus wird eine Fitneßbewertung der Individuen vorgenommen. Aus dem Fitneßwert eines Individuums resultiert seine Fortpflanzungs-

wahrscheinlichkeit. Im Gegensatz zum Zielfunktionswert, der eine Eignung des Individuums für das Optimierungsproblem ausdrückt, wird die Fitneß durch den Vergleich aller Individuen einer Population berechnet. Dafür existieren eine Reihe von verschiedenartigen Verfahren [Goldbg89].

Eine Auswahl der Elternindividuen zur Bildung von Nachkommen erfolgt im zweiten Schritt: der Selektion. Die verschiedenen Methoden zu Selektion berücksichtigen bei der Zusammenstellung der Elternpaare die berechneten Fitneßwerte. Weitverbreitet ist die Roulettselektion, welche den Individuen Abschnitte auf einem Wertebereich [0,L] zuordnet (Größe entspricht den Fitneßwerten). Die Elternindividuen werden durch die Generierung von Zufallszahlen in diesem Bereich gewählt [Baker87].

Im dritten Schritt entstehen durch Rekombination der ausgewählten Elternindividuen Nachkommen. Die existierenden Operatoren entwickeln aus zwei oder mehreren Elternindividuen durch Kombination der Variablenwerte neue Individuen. Hierfür kommen verschiedene Verfahren zum Einsatz.

Bei der *diskreten Rekombination* wird für jede kodierte Variable eines Individuums per Zufallsgenerator geregelt, von welchem Elternindividuum der Variablenwert verwendet wird. In einem n-dimensionalen Raum, n entspricht dabei der Anzahl der Variablen eines Individuums, können bei der diskreten Rekombination nur Nachkommen erzeugt werden, die auf den Eckpunkten des durch die Elternindividuen aufgespannten Hyperwürfels liegen. Die Verfahren der intermediären Rekombination können beliebige Individuen innerhalb des Hyperwürfels hervorbringen. Durch entsprechende Parameterisierung sind sogar Individuen außerhalb dieses Würfels möglich [Pohlhm99].

Die Mutation nimmt eine zufällige Veränderung des genetischen Materials der neu gebildeten Individuen vor. Auf diese Weise können Informationen entstehen, die im Laufe der Generationen verloren gegangen sind oder noch nicht vorhanden waren. Die Größe der zufälligen Änderung eines Individuums wird als Mutationsschritt und die Wahrscheinlichkeit einer Mutation als Mutationsrate bezeichnet.

Entsprechend der Repräsentation der Variablen eines Individuums gibt es verschiedene Mutationsoperatoren. In der binären Darstellung kann die Mutation durch zufälliges Umschalten von Bits erfolgen. Bei der Darstellung der Variablen in Form von reellen Zahlen wird die Mutation durch Addition eines zufällig erzeugten Wertes realisiert. Je nach Problemstellung sind unterschiedlich große Mutationsschrittweiten gebräuchlich. Für beide Darstellungsformen finden sich in der Literatur eine Reihe von Operatoren, die sowohl kleine als auch große Mutationsschritte mit einer bestimmten Wichtung zulassen [Pohlhm99].

Nach der Bestimmung der Fitneß der so erzeugten Nachkommen kann die neue Generation zusammengestellt werden. Hierfür ist festzulegen, wieviele Nachkommen in die Population eingefügt werden und welche Individuen der Elterngeneration ersetzt werden. Es gibt Strategien, welche die neue Generation komplett aus den Nachkommen aufbauen. Ein Individuum existiert in diesem Fall nur für eine Generation. Andere Verfahren wählen die neue Generation zufällig aus den Nachkommen und den

Individuen der vorherigen Population aus oder tauschen nur die schlechtesten Individuen der alten Population durch neu gebildete aus [Wegener, 1999].

Mit der Zusammenstellung einer neuen Generation schließt sich der Kreis des evolutionären Algorithmus. Für die entstandene Population wird das Abbruchkriterium überprüft und falls notwendig die nächste Generation erzeugt.

3.1.2 Genetische Algorithmen vs. Evolutionsstrategie

Im Rahmen der Entwicklung von Algorithmen zur Simulation der Evolution für die Optimierung technischer Systeme entstanden verschiedene Methoden, die in den vorangegangenen Abschnitten mit dem Oberbegriff evolutionäre Algorithmen zusammengefaßt wurden. In [Schönb94] werden unter dem Begriff der evolutionären Modelle zwei Verfahren erläutert. Das sind die *Evolutionsstrategie* (siehe [Rechb73] und [Schwefl77]) und die *genetischen Algorithmen* (siehe [Hollnd75] und [Michal92]).

Genetische Algorithmen gehen von einer binären Kodierung der Individuen in sogenannten Bitstrings aus. Die Operationen Rekombination und Mutation werden auf Bitebene definiert. Wobei die Rekombination die zentrale Rolle bei der Produktion von Nachkommen spielt.

Für die Evolutionsstrategien erfolgt eine Kodierung der Individuen durch einen Vektor von reellen Werten. In diesem Modell spielt die Rekombination eine untergeordnete Rolle. Sie wird anhand der Verknüpfung von Vektorelementen beschrieben. Die Erzeugung von Nachkommen konzentriert sich auf den Einsatz der Mutation vorhandener Individuen. Dazu werden leistungsfähige Mutationsoperatoren mit Schrittweitenadaption genutzt.

3.2 Strukturtests als evolutionärer Prozeß

Die Verfahren des dynamischen Strukturtests haben die Bestimmung von Testdatenmengen zum Ziel, die ein Strukturtestkriterium erfüllen. Solche Kriterien fordern, wie Kapitel 2 zeigt, die Ausführung bestimmter Anweisungen, Pfade usw. Die Ermittlung der zu einem Kriterium gehörenden Testdatenmenge läßt sich auf die Bestimmung der Testdaten zu einzelnen *Teilzielen* zurückführen. Zum Beispiel ist für die Anweisungsüberdeckung die Ausführung einer bestimmten Anweisung ein Teilziel.

Versteht man die Bestimmung einer Testeingabe zu einem Teilziel als Optimierungsproblem, lassen sich die evolutionären Algorithmen leicht für den Strukturtest adaptieren. Für den Test werden zum jeweiligen Kriterium Teilziele gebildet. *Evolutionäre Tests* beschreiben einen Prozeß bei dem das Testobjekt in mehreren Zyklen (Generationen) mit Testdaten daraufhin überprüft wird, ob ein Teilziel erreicht oder wie nah die Eingabe dem Ziel ist. Es erfolgt eine Evolution der Testeingaben [Wegener, 1998].

Der evolutionäre Strukturtest umfaßt damit eine Folge von evolutionären Optimierungen für die einzelnen Teilziele. Im Gesamtprozeß werden Testdaten entwickelt, welche ein bestimmtes Strukturtestkriterium erfüllen. Mehr Details zur *Bestimmung der*

Teilziele und die Festlegung einer geeigneten Reihenfolge bei der Abarbeitung enthält Abschnitt 4.1.

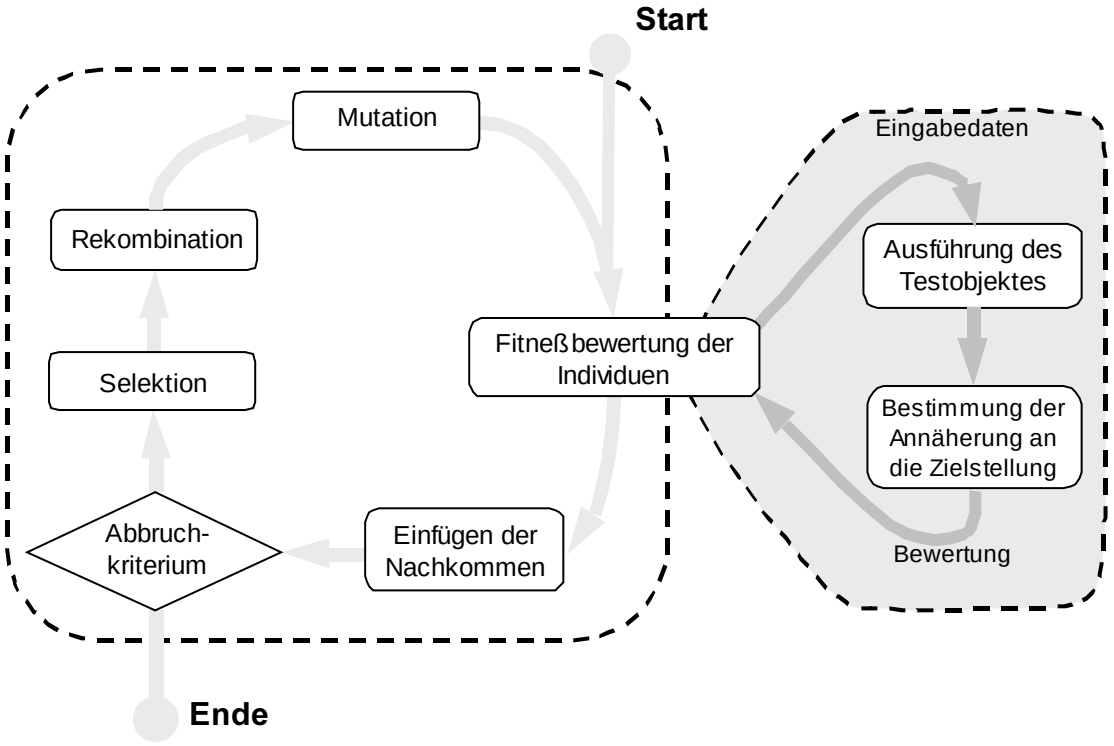


Abbildung 3.2: evolutionärer Optimierungsprozeß für ein Teilziel

Das zweite Kernproblem der Adaption evolutionärer Algorithmen ist aus Abbildung 3.2 ersichtlich. Im dargestellten Prozeß bedarf es einer Bewertung der Individuen anhand einer Annäherung an das Teilziel. Dafür ist eine Zielfunktion zu entwickeln, die eine Berechnung auf Basis der Informationen zu den ausgeführten Programmelementen gestattet. Das Teilziel wird mit dem globalen Optimum der Funktion erreicht.

Die Kriterien der betrachteten Testverfahren orientieren sich an den Strukturinformationen der zu testenden Software. Deshalb ist es naheliegend, die Zielfunktion mit Hilfe dieser Angaben auszudrücken. Im folgenden Abschnitt werden einige Lösungsansätze aus der Literatur erläutert.

3.3 Arbeiten zur Thematik

Zur Automatisierung von Softwaretests existiert eine breite Palette von Arbeiten mit verschiedenen Ansätzen. Dieser Abschnitt stellt eine kleine Auswahl dieser Artikel mit nützlichen Ideen und Vorschlägen zur Lösung des Problems vor.

Nach der Betrachtung einer Methode zur Testdatengenerierung von Bogdan Korel [Korel90] wird die Veröffentlichung von Jones, Sthamer und Eyres [Jones96] erläutert. Der Beitrag veranschaulicht einen Lösungsansatz für die automatische Durchführung von Zweigtests auf Basis genetischer Algorithmen. Das dort ausgearbeiteten Grundgerüst wurden im Rahmen dieser Diplomarbeit weiterentwickelt. In der Arbeit [Tracey98] wird ein System zur Testdatengenerierung für Softwaretests im allgemeinen entworfen.

Ein weiterer Artikel beschreibt dessen Anwendung für den Strukturtest ([Tracy98b]). Insbesondere der Aufbau der Kostenfunktion in der zweitgenannten Arbeit wird im folgenden näher erläutert. [Pargas99] schlägt ein Konzept auf Basis der Kontrollflußabhängigkeiten vor. Die Idee wird im folgenden beschrieben. Eine Untersuchung der Ergebnisse bildet den Abschluß der hier analysierten Veröffentlichungen.

Bogdan Korel: Automated Software Test Data Generation

Bogdan Korel [Korel90] beschreibt in seiner Arbeit einen Ansatz zur Erzeugung von Testdaten für bestimmte Strukturtestkriterien. Grundlage der Methode bildet die Identifikation einer Menge von Pfaden, aus der eine Überdeckung des Kriteriums resultiert. Zu jedem Pfad (*Teilziel*) der Menge wird ein Testdatum gesucht, das den Pfad im Testobjekt ausführt. Für die Bestimmung dieses Testdatums wird eine *Verzweigungsfunktion* gebildet und eine Optimierung vorgenommen.

Durch ein *iteratives Vorgehen*, ausgehend von einer Testeingabe, deren resultierender Ausführungspfad ein möglichst langes, identisches Anfangsstück mit dem Zielpfad besitzt, erfolgt eine Suche nach dem Testdatum zum Teilziel. Die Abweichung in der Verzweigung nach dem identischen Wegstück drückt der Autor durch die Bildung einer Funktion auf Basis der Verzweigungsbedingungen aus.

In den Iterationsschritten der Optimierungsmethode kann durch Variation des Testdatums und Überwachung der Verzweigungsfunktion eine Annäherung an den Zielpfad gefunden werden. Wenn die Suche erfolgreich ist, führt das resultierende Testdatum einen Pfad mit längerem identischen Anfangsstück aus. Zur Optimierung der Verzweigungsfunktion kommt ein Verfahren zur Minimierung, die "*direct-search*" Methode, zum Einsatz. Diese Strategie beschreibt die schrittweise Verbesserung des Verzweigungsfunktionswertes durch die getrennte Einstellung der einzelnen Variablenwerte. Schrittweise heißt, daß immer nur eine Variable in Richtung des Optimums verändert wird.

Der Autor stellt fest, daß eine Datenflußanalyse die Suche unterstützen kann und zeigt erste Erfahrungen mit einem Prototypen auf. Eine solche Analyse kann die auf die Verzweigungsfunktion einflußnehmenden Variablen der Testeingabe bestimmen und so die Suche effizienter gestalten, da nur ein Teil der Variablen den Suchraum für die Minimierung bildet.

Ein weiterer Bestandteil der Arbeit sind Überlegungen für den Fall dynamischer Eingabestrukturen, also Bäume und Listen in Form von Pointerstrukturen (mehr dazu siehe Ausblick Abschnitt 7.3 "Erweiterungen für komplexe Testeingabeschnittstellen").

Prather and Myers: The Path Prefix Software Testing Strategy

Ronald E. Prather und J.Paul Myers setzen sich in ihrer Arbeit [Prather87] mit der effizienten Testdatengenerierung für Strukturtestverfahren auseinander. Die Verfasser erarbeiten eine adaptive Methode zur Ermittlung von Testdaten für die Zweigüberdeckung. Ausgehend von vorliegenden Testverläufen wird die am besten für die Erfüllung noch offener Teilziele geeignete Eingabe ausgewählt und modifiziert.

Für die Bestimmung der Testdatenmenge zur Zweigüberdeckung beschreiben die Autoren einen iterativen Algorithmus, der eine schrittweise Erzeugung von Testdaten zuläßt. Die vorgestellte *Pfadpräfixstrategie* geht von einer Nachbarschaftsbeziehung, also der Nähe von Teilzielen (*test goal closeness phenomen*), aus. In jedem Iterationsschritt werden die vorliegenden Testverläufe auf die Ausführung von Verzweigungen untersucht, bei denen noch nicht alle Zweige ausgeführt werden. Die Methode ermittelt auf diesem Weg Teilziele, welche in der näheren Umgebung bereits durchlaufener Pfade liegen.

Zur optimalen Auswahl des nächsten zu variierenden Testdatums schlagen die Autoren den kürzesten Weg zu einer *offenen Verzweigung* vor. Sie bezwecken damit eine Beschleunigung des Gesamtprozesses. Das Vorgehen entspricht einer Breitensuche über den Zweigen des Kontrollflußgraphen.

Ein Problem der pfadorientierten Testverfahren ist die große Anzahl von Pfaden bei einer komplexen Software und die Auswahl von Zielpfaden aus dieser Menge für die Testausführung. Die Pfadpräfixstrategie nutzt die Ergebnisse vorhandener Testverläufe für die Auswahl des nächsten Zielpfades. Von den Autoren wird diese Strategie deshalb als adaptiv bezeichnet. Sie weisen nach, daß sich durch ihr Vorgehen die Anzahl der für den Test auszuführenden Pfade und damit der zu bestimmenden Testdaten reduziert. Als Obergrenze der Anzahl kann die zyklische Komplexität (*cyclomatic measure*) der zu testenden Software in der Definition von McCabe [McCabe76] angegeben werden. Auch McCabe betrachtet die notwendigerweise zu testenden Pfade einer Software.

Jones et al.: Automatic structural testing using genetic algorithms

B. Jones, H. Sthamer und D. Eyres entwickeln in ihren Arbeiten [Jones96] und [Jones98] ein Konzept der Nutzung genetischer Algorithmen zur automatischen Testdatenerzeugung für die Zweigüberdeckung. Dabei wird ähnlich wie bei den zuvor beschriebenen Arbeiten versucht, ein Testdatum für einen *Entscheidungsweg* zu generieren. Die Autoren gestalten den Suchprozeß eines Testdatum für den auszuführenden Weg als Optimierungsproblem. Für die Optimierung setzen sie genetische Algorithmen ein.

Zur Bestimmung einer Fitneß während der Ausführung des zu testenden Programms ist eine Instrumentierung des Quelltextes notwendig. Die während der Laufzeit des Testobjekts dynamisch bestimmten Fitneßwerte dienen dem genetischen Algorithmus als Selektionskriterium für die Bildung neuer Testdaten.

Eine Zielfunktionsberechnung wird in dieser Arbeit ähnlich dem Ansatz von [Korel90] auf die Auswertungsergebnisse der Verzweigungsbedingungen eines Testobjekts zurückgeführt. Der Unterschied zur Methode von Korel, der eine Variation der Testdaten im Rahmen der Beibehaltung des Anfangspfadstücks vornimmt und die Zielfunktion nur auf die letzte zu verändernde Verzweigungsbedingung bezieht, drückt die Funktion von Jones, Sthamer und Eyres eine Bewertung des Entscheidungsweges bis zur ersten unerwünschten Verzweigung aus. Auf diese Weise kann zu allen getesteten Pfaden eine Annäherung zum Zielpfad gemessen werden.

Neben dem Testkriterium der Zweigüberdeckung analysieren die Autoren die Nutzung des Verfahrens zur Generierung von Testdaten, die eine bestimmte Anzahl von Schleifenwiederholungen (null, ein- und zweimaliger Durchlauf) bewirken. Dies ist kein spezielles Strukturtestkriterium, aber ein Lösungsansatz für die Überprüfung der Pfadüberdeckung, die bestimmte Wege durch Schleifeniterationen fordert.

Da sich ein Pfad im Falle einer Schleifenwiederholung nicht im Kontrollflußgraphen überprüft läßt, schlagen die Autoren als Lösung ein Ausrollen des Schleifenkörpers in einem sogenannten Kontrollflußbaum vor. Damit kann eine Bewertung des Pfades durch Schleifen genau wie bei der Zweigüberdeckung anhand der Verzweigungen des Kontrollflußbaums erfolgen.

Die Arbeit beschäftigt sich im weiteren ausführlich mit den Operatoren des genetischen Algorithmus und analysiert die beste Kodierung der Variablen sowie die Bildung einer guten Abstandsfunktion. Dazu testen die Autoren das Verhalten des genetischen Algorithmus für die nachstehenden Funktionen:

$$\text{Abstand(A .equal B)} = \frac{1}{|A - B|} \tag{3.1}$$

$$\text{Abstand(A .equal B)} = \text{Hamming Distanz(A-B)} \tag{3.2}$$

Tracey et.al.: An Automated Framework for Structural Test-Data Generation

Der Artikel von N.Tracey, J.Clark, K.Mander und J.McDermid [Tracy98b] beschreibt in Anlehnung an die Ergebnisse von [Jones96] ein Konzept zur automatischen Testdatengenerierung für die verschiedenen Strukturtestverfahren. Dabei wird die Bildung einer Zielfunktion, hier Kostenfunktion genannt, als entscheidend für die Optimierung erkannt und für den Fall der Zweigüberdeckung eine Lösung angeboten. Die vorgeschlagene Kostenfunktion orientiert sich an dem von den Autoren entwickelten System zur Automatisierung von allgemeinen Softwaretests [Tracey98]. Als Optimierungsmethode benutzt das entwickelte System die Methode *Simulated Annealing*.

Die Verfasser entwickeln in Erweiterung zum Ansatz von [Jones96] eine Kostenfunktion für zusammengesetzte Verzweigungsbedingungen, also für logische Verknüpfungen von Relationen durch *And*, *Or* und *Not*. Eine solche Funktion führt den Suchalgorithmus zum Zielknoten und wird ähnlich wie beim Ansatz von [Jones96] über einen Pfad vom Eingangs- zum Zielknoten definiert. Sie entspricht damit dem pfadorientierten Ansatz von [Korel90]. Existieren mehrere Pfade zu einem Ziel, erfolgt ein separater Test. Die Autoren betrachten den Fall gesondert, daß der zu erreichende Knoten innerhalb einer Schleife liegt. Sie schlagen vor, die Kosten auf Basis der besten Annäherung aller Schleifenwiederholungen zu beschreiben, da nicht feststeht, in welcher Iteration eine Ausführung möglich ist.

Der entwickelte Prototyp fügt automatisch eine Instrumentierung entsprechend der Knoten des Kontrollflußgraphen in den Quelltext ein. Die Funktionsaufrufe der Instrumentierung werten die Bedingungen noch vor der Verzweigungsanweisung aus und protokollieren diese Informationen bei Abweichung vom Teilziel. Ein Abbruch des

Tests erfolgt für den Fall des Verfehlens des Teilziels. Für Schleifen integriert das Testsystem zusätzliche Zähler in die zu testende Software, so daß ein Schleifenkriterium überprüft werden kann.

Pargas et al.: Test-Data Generation Using Genetic Algorithms

Roy Pargas, Mary Jean Harrold und Robert Peck [Pargas99] stellen einen zielorientierten Ansatz (siehe auch [Jasper94]) zur Testdatengenerierung vor, der anders als das pfadorientierte Testen nicht die Ausführung eines bestimmten Pfades zum Ziel hat, sondern das Erreichen eines Zielknoten über eine Menge von gleichwertigen Wegen. Die Wegmenge ist durch *unerwünschte* Verzweigungen eingegrenzt, deren resultierender Pfad am Zielknoten vorbeiführt. Zur Identifikation der unerwünschten Verzweigungen verwenden die Autoren den Kontrollflußabhängigkeitsgraph (ein Konzept aus der *Compilerbautechnik* z.B. [Schaeff73] "control dependence graph").

Die Abhängigkeiten der Ausführung einer Anweisung von bestimmten Verzweigungsknoten können anhand des Kontrollflußgraphen berechnet werden. Auf Basis dieser Information führen die Autoren den sogenannten *control-dependence-predicate-path* ein. Dabei handelt es sich um die Liste aller Verzweigungsbedingungen, von denen eine Anweisung abhängig ist. Die enthaltenen Bedingungen müssen erfüllt werden, um keine unerwünschte Verzweigung auszuführen. Eine Definition der Zielfunktion erfolgt über die Anzahl der identisch ausgewerteten Prädikate von *control-dependence-predicate-path* und Bedingungswerten des Testverlaufs.

Ferner beschäftigen sich die Verfasser von [Pargas99] mit dem Gesamtprozeß der Testdatengenerierung für alle Teilziele. Beim Start der Testdatensuche zu einem noch nicht erreichten Teilziel bewerten sie alle gesammelten Testverläufe neu, da die Bestimmung der Eignung der Testdaten für jedes Teilziel individuell gestaltet ist.

Die Arbeit stellt ein Prototypsystem für den automatischen Strukturtest vor. Das System stützt sich auf ein Analyse- und Instrumentierungswerkzeug (*Aristotle*) sowie ein Programmpaket für die gleichzeitige Optimierung mehrerer Teilziele (*TGen*). Der parallele Strukturtest wird durch nebenläufige Programminstanzen erreicht. Die einzelnen Instanzen arbeiten über einen zentralen Datenpool zusammen, so daß eine gemeinsame Nutzung der Testergebnisse möglich ist.

Wertung

Bogdan Korel: Automated Software Test Data Generation

In dieser Veröffentlichung wird der Ansatz des pfadorientierten Testens verwirklicht. In einem späteren Artikel [Fergus96] erläutert der Autor die Probleme seines Vorschlags. Die wesentlichen Elemente, nämlich die Berechnung der Menge der Pfade zu einem Teilziel (z.B. Knoten) und die Auswahl des Pfades als Zielstellung, sind im allgemeinen schwer zu realisieren. Aus einer großen Menge konstruierbarer Pfade müssen diejenigen gefunden werden, die tatsächlich ausführbar (*möglich*) sind.

Für komplexe Software, d.h. mit vielen Verzweigungen und Schleifen, steigt die Anzahl der aus dem Kontrollfluß berechenbaren Pfade, wie schon beim Pfadüberdeckungs-

kriterium in Teilabschnitt 2.2.1 beschrieben. Ein Teil davon ist aufgrund der Kombination der Verzweigungsbedingungen nicht *möglich*. Eine Reihe von Arbeiten beschäftigen sich mit der Bestimmung von ausführbaren (*feasible*) Pfaden. Zum Beispiel werden in [Jasper94] die möglichen Pfade anhand der Erfüllbarkeit der Kombination der Bedingungen in den Verzweigungspunkten eines Pfades identifiziert. Für komplexe Testobjekte mit Rekursion, Schleifen, dynamischen Datenstrukturen und Arrays ergeben sich jedoch Probleme bei der Analyse.

Die Schwäche der Methode von Korel liegt in der Auswahl des richtigen Pfades zum Zielknoten. Wie die Verfasser einräumen haben auch die verschiedenen Algorithmen zur Überprüfung der Pfade über die symbolische Ausführung wenig Erfolg, sobald die Komplexität der verwendeten Datenstrukturen hoch ist (Felder, Listen, Bäume).

Prather and Myers: The Path Prefix Software Testing Strategy

Wie der Name schon andeutet, handelt es sich bei der Methode in [Prather87] um eine pfadorientierte Strategie. Hervorzuheben ist dabei jedoch die effektive Nutzung vorhandener Testergebnisse für das Vorgehensmodell. Bei dieser Lösung stellt sich das Problem, daß eine Automatisierung des Strukturtests versucht, Testeingaben für unmögliche Pfade zu generieren, weniger gravierend dar, weil nur ausgeführte Pfade in Richtung Ziel variiert werden.

Als kritisch erweist sich meiner Meinung nach die Auswahl des Pfades für eine offene Verzweigung, sobald es mehrere Wege dorthin gibt. Wenn zu einer Entscheidung mit einem offenen Zweig mehrere Testfälle existieren, legen sich die Autoren auf den Pfad mit dem kürzesten Präfix, also der geringsten Anzahl von Verzweigungen bis zum Teilziel, fest. Hierfür bilden die Autoren ein Prädikat, welches sich aus allen im Pfad enthaltenen Bedingungen zusammensetzt. Eine Änderung der Variablen ist nur in den durch dieses Prädikat erlaubten Grenzen zugelassen. Mit der Anzahl der Bedingungen nimmt die Länge eines solchen Prädikats zu. Die Autoren wählen den kürzesten Präfixpfad, um die Suche zu erleichtern, weil eine kürzerer Präfixpfad weniger Verzweigungsbedingungen enthält. Es kann jedoch keinesfalls davon ausgegangen werden, daß die Anzahl der Prädikate in Zusammenhang mit der Lösbarkeit eines Gesamtprädikats steht. Es ist sogar möglich, daß ein längeres Präfix weniger Einschränkungen auf die Variablenwerte vornimmt oder sich die Teilbedingungen bei einem kürzeren Präfix gegenseitig ausschließen. In beiden genannten Fällen sind die Erfolgschancen für eine Suche mit dem längeren Präfix höher.

Jones et al.: Automatic structural testing using genetic algorithms

In den Ausführungen von Jones, Sthamer und Eyres wird ein erster Vorschlag für die Bildung einer Zielfunktion zur mathematischen Beschreibung des Optimierungsproblems dargestellt. Zur Vereinfachung wurden nur atomare Verzweigungsbedingungen betrachtet. Prinzipiell setzen sich Bedingungen jedoch aus mehreren logisch verknüpften Bedingungen zusammen, so daß hier Erweiterungen notwendig sind.

Eine Schwäche des Lösungsansatzes ist wie schon bei [Korel90] das Problem der Auswahl eines ausführbaren Pfades aus einer Menge der konstruierbaren Wege. Wird für

die Optimierung ein *unmöglicher* Pfad gewählt, bleibt die evolutionäre Suche stecken. Dabei kann das Testsystem nicht entscheiden, ob eine ungünstige Population vorliegt oder das Teilziel tatsächlich nicht erreichbar ist.

Die Auswahl eines speziellen Entscheidungswegs grenzt die Möglichkeiten für die Suche der Testeingaben unnötig ein. Gibt es zum Beispiel zehn Wege zu einem Teilziel werden im ungünstigsten Fall zehn getrennte Optimierungen ausgeführt. Dadurch gestaltet sich der Gesamtprozeß der Testdatengenerierung zu einem Teilziel wesentlich aufwendiger.

Als sehr nützlich erweist sich die Idee des Ausrollens der Schleifen für den strukturierten Pfadtest. Die Kriterien in diesem Testverfahren fordern die Wiederholung bestimmter Wege durch das Schleifeninnere. Dies kann durch das Entrollen in einem Graphen überprüft werden. Mehr dazu bei den Ausführungen zur Zielfunktion für pfadorientierte Testverfahren.

Tracey et al.: An Automated Framework for Structural Test-Data Generation

In Erweiterung zu den Vorschlägen von [Jones96] entwerfen die Autoren die Zielfunktion für den Fall zusammengesetzter Verzweigungsbedingungen. Eine Verbesserung ist die Idee der Bewertung für Anweisungen in Schleifen, bei denen eine wiederholte Annäherung an das Teilziel (ein Knoten) erfolgt.

Schwachpunkt dieser Arbeit bleibt das Problem der Festlegung auf einen bestimmten Pfad. Die Instrumentierung der Autoren sieht vor, daß das Teilziel direkt zur Laufzeit zu überprüfen und gegebenenfalls der Test abzubrechen ist. Mit diesem Ansatz kann immer nur ein Teilziel überwacht werden. Zufällig gefundene Eingaben für andere Ziele kann das System nicht erkennen.

Pargas et al.: Test-Data Generation Using Genetic Algorithms

Die Autoren dieser Arbeit setzen bei der Bestimmung der Zielfunktion auf den Kontrollflußabhängigkeitsgraph. Gegenüber den anderen Arbeiten kann mit Hilfe dieses Konstrukts die Abhängigkeit eines Teilziels für die knotenorientierten Testverfahren vollständig beschrieben werden, da ein solcher Graph schleifenfrei und damit die Reihenfolge der Verzweigungen ablesbar ist.

Als Schwäche erweist sich jedoch, die Zielfunktion in Form einer Abdeckungszahl auszudrücken. Dadurch ergibt sich eine unzureichende Führung der Optimierung. Sobald zwei Lösungen die gleiche Anzahl identisch ausgewerteter Prädikate besitzen, ist eine zielgerichtete Optimierung unmöglich. Mit dem *control-dependence-predicate-path* erfolgt eine Festlegung auf genau einen Entscheidungsweg. Auch wenn die Anzahl der möglichen Entscheidungswege gegenüber den konstruierbaren Pfaden sehr viel geringer ist, besteht weiterhin eine Einschränkung des erlaubten Suchbereiches. Zielknoten in Schleifen werden in diesem Lösungsansatz nicht behandelt, weil das Bewertungskriterium diesen Fall nicht berücksichtigt.

4 Automatisierung von evolutionären Strukturtests

Für die Automatisierung von dynamischen Softwaretests wurde im Rahmen dieser Arbeit das System ASTELA (*Automatic Structural Testing with Evolutionary Algorithms*) entworfen und implementiert. Dieses System wird im folgenden vorgestellt. Die grundlegenden Ideen basieren auf den Ergebnissen der Arbeit [Jones96]. Das dort beschriebene Konzept wurde für die Unterstützung anderer gängiger Testverfahren erweitert und dazu eine Reihe von Bewertungsmethoden entwickelt. Ein weiterer Anspruch an die im Rahmen dieser Arbeit entworfene Bewertungsmethode bestand darin, aus allen generierten Testdaten eine Zusammenstellung guter Startwerte für die verschiedenen Teilziele eines Tests zu ermöglichen um damit die Effizients des Gesamttests zu erhöhen. Das hier vorgestellte Verfahren erlaubt die parallele Bewertung eines Testdatums für alle Teilziele.

Zu einer Eingabe in Form eines Quelltextes (*Testobjekt*) führt das System *ASTELA* automatisch einen dynamischen Strukturtest durch. Dem Nutzer stehen mehrere Strukturtestverfahren zur Auswahl, die er in den Benutzereinstellungen für die Testausführung spezifizieren kann. Wenn der Strukturtest erfolgreich ist, liefert das System als Ergebnis eine Testdatenmenge, welche das festgelegte Testkriterium erfüllt oder den bis dahin erreichten Überdeckungsgrad repräsentiert. Für die Generierung der Testdaten verwendet das System evolutionäre Algorithmen.

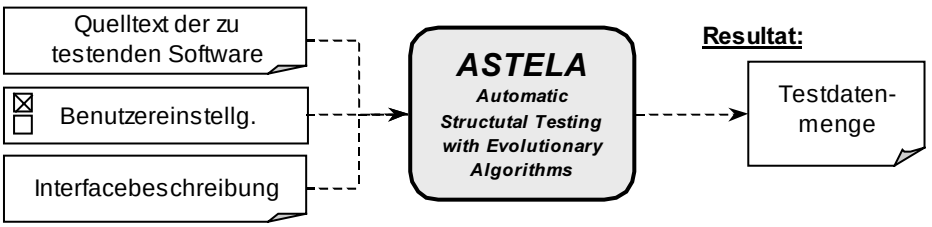


Abbildung 4.1: Ein/Ausgaben des Systems

Einen Überblick zu den Systemkomponenten und der Verteilung der Aufgaben gibt Abschnitt 4.1. Der nächste Abschnitt erläutert die zentrale Teststeuerungskomponente. Für die evolutionäre Optimierung ist die Entwicklung geeigneter Zielfunktionen notwendig. Die Betrachtungen zur Bewertung von generierten Testdaten sind Inhalt von Abschnitt 4.3. Eine besondere Rolle bei der Berechnung der Zielfunktionswerte spielen Abstandsfunktion und Erreichbarkeitsgraph. Mit diesen zwei Konzepten beschäftigen sich die Abschnitte 4.4 und 4.5. Die Komponenten zur Testvorbereitung behandelt Abschnitt 4.6 am Ende des Kapitels, nachdem die notwendigen Vorbereitungen des Testobjekts für den evolutionären Test in den vorangegangenen Ausführungen beschrieben wurden.

4.1 System für den evolutionären Strukturtest (ASTELA)

ASTELA besteht aus einer Reihe von Softwarekomponenten, welche bei der Ausführung des evolutionären Strukturtests zusammenwirken. Die *Teststeuerung* übernimmt die Koordination der Arbeit des in Abbildung 4.2 dargestellten Systems. Alle anderen

Systemelemente lassen sich nach ihren Aufgaben in zwei Gruppen unterteilen. Das sind die Werkzeuge zur Vorbereitung des Tests und die Komponenten für die Durchführung der evolutionären Optimierung.

Folgenden Teilabschnitte charakterisieren die in Abbildung 4.2 gezeigten Komponenten. Im System arbeiten Teststeuerung (Teilabschnitt 4.1.1), evolutionärer Algorithmus (Teilabschnitt 4.1.2) und *Testtreiber/Modulstubs* (Teilabschnitt 4.1.4) eng zusammen. Letztere dienen der dynamischen Testdurchführung und werden automatisch generiert. Teilabschnitt 4.1.5 stellt die dafür konzipierten Werkzeuge vor.

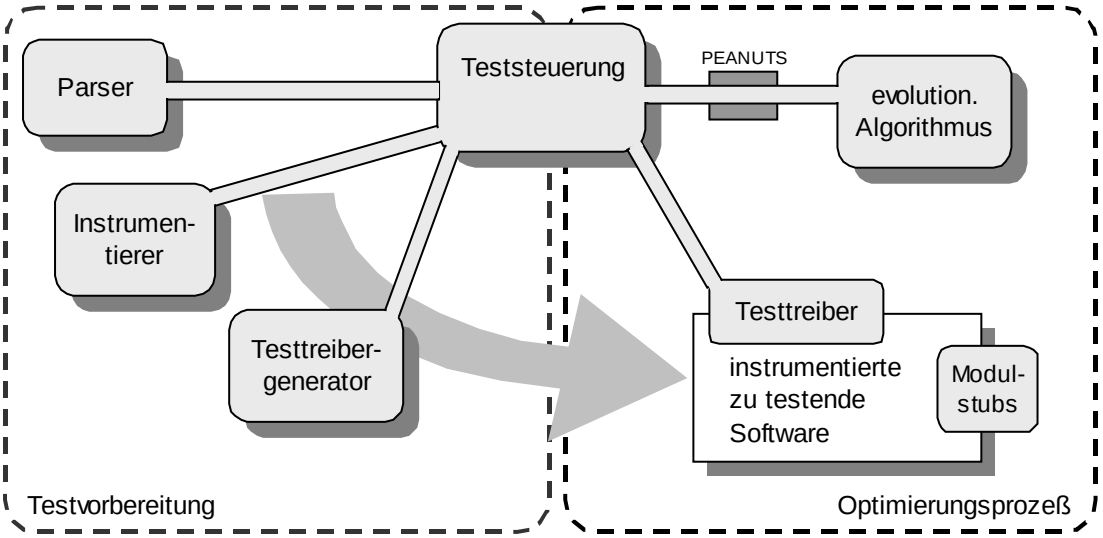


Abbildung 4.2: Aufbau von ASTELA – ein System für den evolutionären Strukturtest

Für eine Kommunikation zwischen der Teststeuerung und den evolutionären Algorithmen wurde im Rahmen dieses Projekts die Schnittstelle *PEANUTS (Program for Evolutionary Algorithm Network Communications)* entworfen. Diese Kommunikationsschnittstelle beschreibt Teilabschnitt 4.1.3.

4.1.1 Teststeuerung

Die Teststeuerung bestimmt zur Vorbereitung des Strukturtests Teilziele für die Optimierung auf Basis des gewählten Testverfahrens und der zu testenden Software. Für die *Identifikation* der Teilziele sind Strukturinformationen zum Testobjekt notwendig. Diese Informationen werden durch die *Parser-Komponente* bereitgestellt. Der dynamische Strukturtest mit generierten Testdaten erfordert die Anpassung der zu untersuchenden Software. Das Testobjekt wird mit Hilfe eines automatisch erzeugten *Testtreibers* ausgeführt. Durch eine Instrumentierung erfolgen während der Laufzeit Messungen für die Bewertung der Testdaten. Die Ausführung der nötigen Anpassungen delegiert die Teststeuerung an die *Instrumentierer-Komponente* und den *Testtreibergenerator*.

Nach der Bestimmung von Teilzielen entscheidet die Teststeuerung über deren Reihenfolge bei der Abarbeitung. Für jedes Teilziel werden in Zusammenarbeit mit der evolutionären Optimierungskomponente Testdaten generiert, mit dem Ziel ein Test-

datum zu finden, welches das Teilziel erfüllt (Konzept siehe Abschnitt 3.2). Die Berechnung der Zielfunktion ist die Hauptaufgabe der Teststeuerung während dieses Suchprozesses. Abschnitt 4.3 erläutert die Entwicklung geeigneter Funktionen für die einzelnen Testverfahren.

Wenn alle Teilziele erreicht sind, also das gesamte Testkriterium erfüllt ist, oder bestimmte Abbruchkriterien, wie zum Beispiel eine maximale Testdauer, wirksam werden, beendet die Teststeuerung den Test. Als Ergebnis liefert sie eine Eingabemenge, welche das Kriterium erfüllt, beziehungsweise den während des gesamten Suchprozesses erreichten Überdeckungsgrad repräsentiert. Die Eingabemenge ist eine Zusammenstellung aller Eingaben der einzelnen erfüllten Teilziele, wobei mehrere Teilziele durch dasselbe Testdatum erfüllt sein können.

Eine detaillierte Beschreibung der technischen Umsetzung der *Teststeuerungs-Komponente* findet sich in den Abschnitten 4.2 "*Aufgaben der Teststeuerung*", 4.3 "*Zielfunktion*", 4.4 "Abstandsfunktion" und 4.5 "*Erreichbarkeitsgraph*".

4.1.2 Evolutionäre Algorithmen

Das System ASTELA nutzt für den evolutionären Strukturtest die *GEA-Toolbox (Genetic and Evolutionary Algorithm Toolbox for Use with Matlab)* von Hartmut Pohlheim [Pohlhm99]. Dieses Werkzeug basiert auf Matlab [Mathworks].

Die GEA-Toolbox erlaubt die ganzzahlige oder reellwertige Darstellung von Individuen. Über zahlreiche Optionen kann die Auswahl der evolutionären Algorithmen gesteuert werden. Es lassen sich sowohl genetische Algorithmen als auch Evolutionsstrategien implementieren. Darüber hinaus sind nahezu beliebige Mischformen evolutionärer Algorithmen möglich. Die GEA-Toolbox ist damit ein ideales Werkzeug zur Umsetzung der evolutionären Strukturtests.

Für jede Variable der Individuen läßt sich ein zulässiger Wertebereich angeben. Bei der Generierung der Individuen durch die Toolbox wird automatisch auf die wertmäßige Gültigkeit der Variablen geachtet.

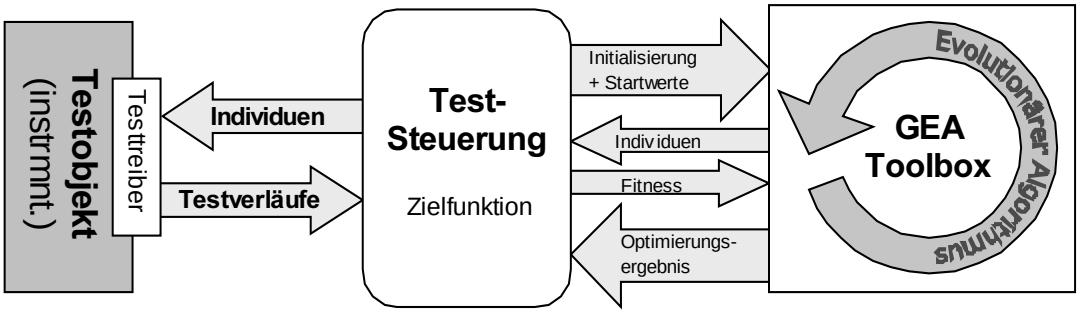


Abbildung 4.3: Umsetzung des Optimierungsprozesses eines Teilziels

Zeitlicher Ablauf der Optimierung mit der GEA-Toolbox

Die Prozesse *Teststeuerung*, *GEA-Toolbox* und *instrumentiertes Testobjekt* kooperieren bei der Durchführung des Strukturtests. Abbildung 4.3 visualisiert die Interaktionen zwischen den Teilprozessen. Eine Synchronisation regelt die Kommunikations-

schnittstelle *PEANUTS* und die Festlegungen in der Teststeuerung zur Ausführung des Testobjekts. Das Testobjekt wird im System *ASTELA* für jede Generation von Individuen erneut aufgerufen. Den zeitlichen Ablauf der Operationen eines Ausschnittes der Optimierung eines Teilziels stellt das Diagramm in Abbildung 4.4 dar.

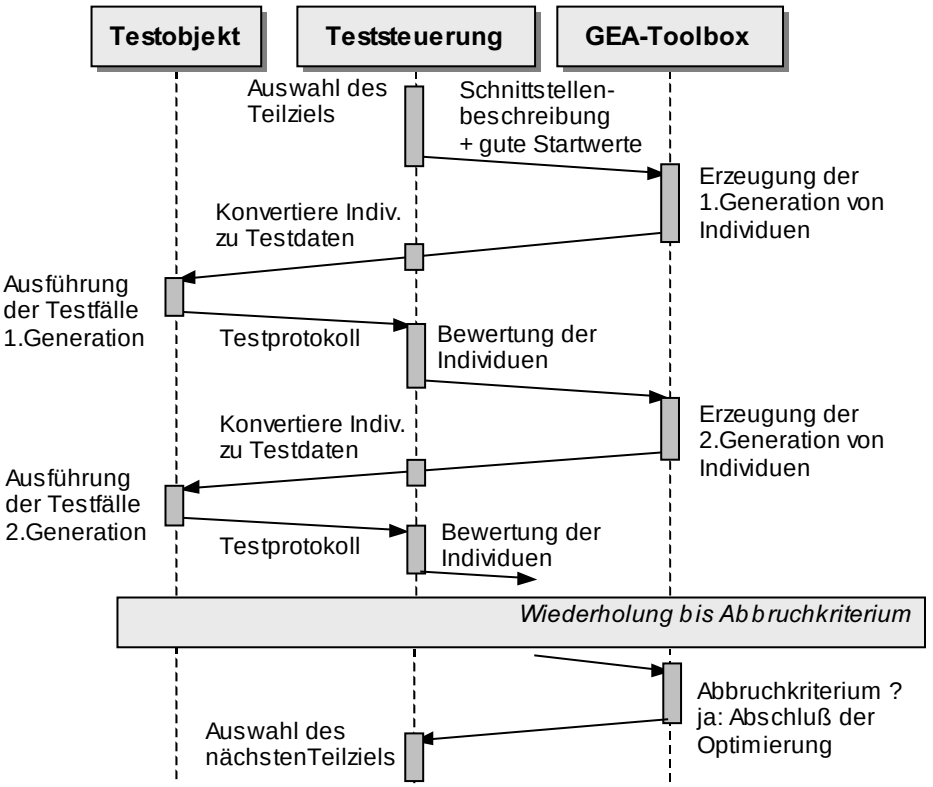


Abbildung 4.4: Sequenzdiagramm für Optimierungsprozeß

Nach der Auswahl eines Ziels übermittelt die Teststeuerung eine Beschreibung der Schnittstelle des Testobjekts und gegebenenfalls gute Startwerte an die *GEA-Toolbox*, die daraufhin die erste Generation von Individuen erzeugt. Bis zum Ende der Optimierung verschickt die Toolbox die Individuen der erzeugten Generationen an die Teststeuerung und wartet auf eine Bewertung. Die Teststeuerung nimmt eine Konvertierung der Individuen in Testdaten vor und startet den Testtreiber, der das Testobjekt mit den Testdaten ausführt. Eine in das Testobjekt integrierte Instrumentierung protokolliert den Testverlauf. Mit den aufgezeichneten Messungen berechnet die Teststeuerung eine Bewertung der Individuen und überträgt diese an die GEA-Toolbox. Solange kein Abbruchkriterium erfüllt ist, erzeugt die Toolbox auf Basis der Bewertungen neue Individuen. Diese Individuen müssen wiederum auf ihre Eignung für das Teilziel überprüft werden. So schließt sich der evolutionäre Kreis.

4.1.3 Kommunikationsschnittstelle (PEANUTS)

Zur Integration von Teststeuerung und evolutionärem Algorithmus wurde ein Client/Servermodell und eine Schnittstelle entwickelt. Das Modell erlaubt die Kommunikation zwischen Teststeuerung und modifizierter GEA-Toolbox auf

getrennten Rechnerplattformen. Als Kommunikationskanal dienen die üblichen Netzwerkprotokolle, so daß eine weitreichende Portabilität des Testsystems sichergestellt ist.

Program for Evolutionary Algorithm Network Communication (*PEANUTS*) stellt ein Client-Server-Protokoll für die Durchführung von Optimierungen zur Verfügung. Der Server ist eine modifizierte Version der *GEA-Toolbox* von Hartmut Pohlheim [Pohlh-GEA]. Dieses Werkzeug wurde für die speziellen Anforderungen an den Test von Softwaremodulen um die notwendigen Schnittstellen erweitert. Als Client operiert die Teststeuerung, welche die zu lösende Aufgabe beschreibt und die Berechnung der Zielfunktion durchführt. Im folgenden wird kurz auf die Bestandteile der Schnittstelle eingegangen.

Der Client spezifiziert den Suchraum des Problems (① Initialisierung). Die angepaßte *GEA-Toolbox* arbeitet als Server, welcher die Testdaten generiert. Während der Optimierung nimmt die *Teststeuerung* die zu testenden Individuen vom Server entgegen (④ Individuen), ruft das Testobjekt mit den umgewandelten Testdaten auf und berechnet die Zielfunktionswerte. Die Bewertung der Individuen wird an den Server gesendet (③ Bewertung). Der Server erzeugt daraufhin neue Individuen oder beendet den Prozeß, in dem er ein Endergebnis an den Client verschickt, welches die optimalen beziehungsweise besten gefundenen Testdaten enthält (② Endergebnis).

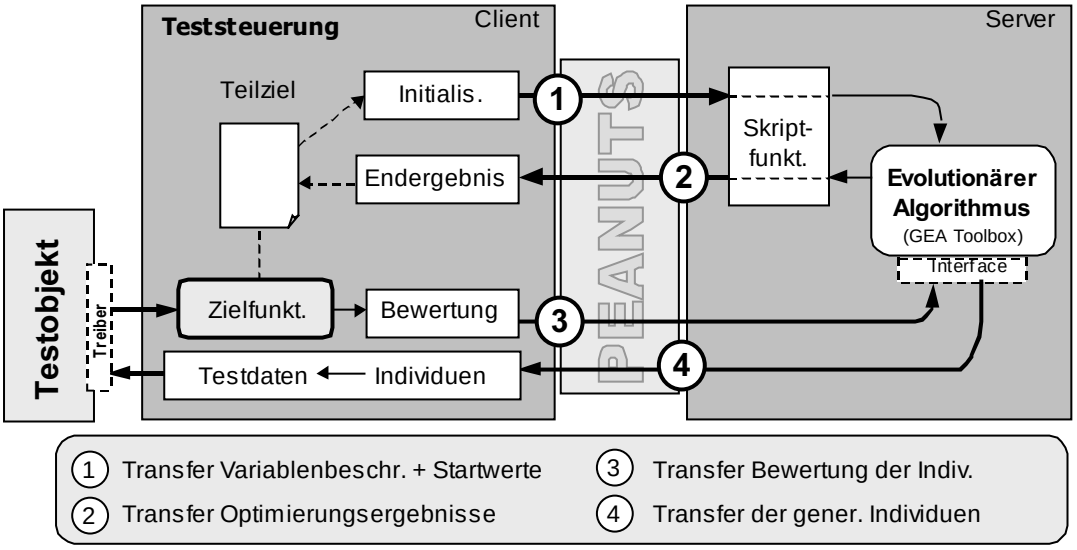


Abbildung 4.5: Struktur des Client-Server-Systems von PEANUTs

Abbildung 4.5 zeigt die Bestandteile von *PEANUTS* und den geschilderten Datenfluß. Für die Beschreibung des Optimierungsproblems erwartet der Server folgende Informationen:

- eine Charakterisierung der Schnittstelle des Testobjekts – hierzu gehören Anzahl und Wertebereiche der Variablen – sowie
- eine Aufstellung bisheriger guter Lösungen (diese Angabe ist optional).

4.1.4 Testtreiber und Modulstubs

Der Testtreiber (*Testrahmen, Aufrufschnittstelle*) verbindet die Teststeuerung mit der zu testenden Funktion (*Testobjekt*). Er ruft das Testobjekt unter Verwendung der generierten Testdaten als Eingabeparameter auf. Die Eingaben eines Testobjekts sind durch die Funktionsparameter und die benutzten globalen Variablen festgelegt. Der Testtreiber muß die zugehörigen dynamischen und nutzerdefinierten Datenstrukturen initialisieren und einlesen können. Eine kurze Erläuterung zur Kodierung von solchen Datenstrukturen in Testdaten findet sich am Ende dieses Teilabschnitts.

Für den dynamischen Strukturtests müssen der Testverlauf analysiert und die Ausführung bestimmter Komponenten des Testobjekts überprüft werden. Im System *ASTELA* findet diese Prüfung mit der Auswertung der Testprotokolle in der Teststeuerung statt. Eine Aufnahme eines Testverlaufs (*Testmonitoring*) erfolgt durch die Instrumentierung im Testobjekt. Die formatierte Ausgabe der Ergebnisse der Instrumentierung in Form des Testprotokolls übernimmt der Testtreiber nach Rückkehr aus dem Testobjekt.

Der Aufwand für die Ausführung des Testobjekts ist zur effizienten Testgestaltung möglichst gering zu halten. Deshalb erlaubt der entwickelte Testtreiber die Ausführung mehrerer Testdaten in Reihenfolge, vereinigt die aufgenommenen Testprotokolle und übergibt sie an die Teststeuerung. Auf diese Weise werden Ladezeiten reduziert.

Für den Test einer Software stehen während der Entwicklungsphase häufig nicht alle Module des zu testenden Softwaresystems zur Verfügung. Es werden sogenannte *Stubs* / *Dummies* (Platzhalter) benötigt. Die Generierung von Stubs gestaltet sich schwieriger als die Testtreibergenerierung und kann nur mit einer Spezifikation der Funktionalität erfolgen. Eine Erzeugung funktionstüchtiger Stubs ist erforderlich, falls Datenflüsse aus den *Stub*-Aufrufen für den Kontrollfluß des Testobjekts relevant sind.

Formen von Platzhaltern (Modulstubs)

Modulstubs besitzen eine Reihe von unterschiedlichen Anwendungsgebieten. Ein Platzhalter kann zum Beispiel für eine Hardwaresteuerungskomponente eingesetzt werden, wenn die entsprechende Hardware dem Testsystem nicht zur Verfügung steht. In diesem Fall übernimmt der Stub die Simulation einer Ansteuerung der Hardware und die Erzeugung von Rückantworten. Die Veränderungen der Reaktion der Hardware sind dabei Bestandteil des Tests, also 'Eingabewerte' des Testobjekts. Neben Hardwarekomponenten können so externe Datenquellen (*Filesystem*, *Datenbank*) oder Benutzeroberflächen simuliert werden [Riedm97].

Eine andere Form eines Stubs ist die beschränkte Nachbildung einer Funktionalität der zu ersetzenden Komponente anhand einer vorliegenden Spezifikation des Verhaltens. Der Leistungsumfang wird dabei nur zum Teil nachgebildet. [Riedm97] schlägt die Generierung eines Stubs anhand von Ein-/Ausgabepaaren vor. So kann zum Beispiel für den Test der Funktionen einer Client-Server-Schnittstelle ein Teil der Funktionalität der Serverkomponente nachgebildet werden, falls diese nicht zur Verfügung steht.

Der einfachste Platzhalter für eine Funktion ist die Null-Funktion, also ein Modulstub ohne Funktionalität. Ein solcher Modulstub kann immer dann eingesetzt werden, wenn die Aufrufe keinen direkten Einfluß auf den Kontrollfluß der zu testenden Funktion haben.

Prinzipiell besteht das Problem, daß Modulstubs die Ergebnisse eines Tests verfälschen können, wenn ersetzte Funktionen nicht korrekt simuliert werden. In diesem Fall muß ein erneuter Test bei Integration der fehlenden Module erfolgen [Riedm97].

Kodierung von Datenstrukturen

Für den dynamischen Strukturtest müssen die erzeugten Testdaten eingelesen und in die Datenstrukturen des Testobjekts umgewandelt werden. Dabei liegen die Testdaten in Form einer Liste von Variablen vor. Sie können im einfachsten Fall als direkte Eingabeparameter des Testobjekts dienen. Für komplexe Datenstrukturen, wie Listen, Arrays oder Bäume, muß eine Kodierung bestimmt und die Daten eines Individuums entsprechend dieser Kodierung in die Datenstrukturen eingelesen werden.

Die Datenstrukturen des Testobjekts sind hierzu in einen Vektor von einzelnen Variablen zu wandeln. Dynamische Strukturen, wie Arrays und Bäume, müssen begrenzt werden, da für die Testdaten eine feste Anzahl von Variablen vorgeschrieben ist. Eine Kodierung von Listen und Bäumen sollte so gestaltet sein, daß bei der Generierung von Testdaten verschiedene Listen und Baumstrukturen entstehen können. Es muß dabei zum Beispiel auch möglich sein, daß ein Testdatum erzeugt wird, welches eine leere Liste repräsentiert.

Die Kodierung von zusammengesetzten Datenstrukturen ist Betrachtungsgegenstand der Testtreibergenerierung im Abschnitt 4.6. Für die Unterstützung von dynamischen Datenstrukturen bei der automatischen Erstellung eines Testtreibers ist eine Einschränkung der Variabilität der Datenstrukturen auf Grund der festen Struktur der generierten Testdaten vorzunehmen. Mögliche Strategien für solche Festlegungen beschreibt Kapitel 7.3 im Ausblick.

4.1.5 Weitere Werkzeuge

Parser-Komponente

Aufgabe der Parser-Komponente ist die Extraktion der Strukturinformationen aus dem Quelltext eines Testobjekts. Diese Informationen dienen der Durchführung von Strukturtests und werden unabhängig von der verwendeten Programmiersprache in Form eines Kontrollflußgraphen und einer Beschreibung des Aufbaus der Verzweigungsbedingungen von der Parser-Komponente zusammengestellt.

Der Parser beherrscht die lexikalischen Festlegungen, den Syntax sowie die semantischen Regeln einer Quellsprache und kann während der Analyse der zu testenden Software Informationen für die Konstruktion des Kontrollflußgraphen sammeln. In Vorbereitung zur *Instrumentierung* und *Testtreibergenerierung* übernimmt der Parser aufgrund der engen Verknüpfung mit der Quellsprache die Bestimmung der Datenbeschreibungen im Testobjekt. Mit Hilfe dieser semantischen Informationen

können Datentypen für die Messungen der Instrumentierung und die Schnittstelle für den Testtreiber identifiziert werden.

Die Ergebnisse der Parser-Komponente sind ein Kontrollflußgraph, die Beschreibung der Verzweigungsbedingungen, Informationen über die im Quelltext definierten Datentypen sowie die Zusammenstellung aller Eingabeparameter eines Testobjekts.

Instrumentierer-Komponente

Aufgabe des Instrumentierers ist das Einfügen von Meßpunkten in die zu testende Software. Dazu nimmt er eine Quelltextmodifikation des Testobjekts vor. Für die Transformationen stehen der Komponente auf Grund der engen Zusammenarbeit mit dem Parser die syntaktische Struktur und die Semantikinformation zum Testobjekt zur Verfügung. Der syntaktische Aufbau dient der Identifikation der Meßpunkte. Um typgerechte Funktionsaufrufe generieren zu können, werden die Datentypinformationen zu den Variablen der Software genutzt (*Semantik*).

Eine Messung wird mit Hilfe von Funktionsaufrufen der für die Programmiersprache bereitgestellten Instrumentierungs-Bibliothek initiiert. Die Protokollierung des gesamten Testverlaufs, in der Literatur häufig als *Testmonitoring* oder *Execution monitoring* bezeichnet, übernehmen weitere Bibliotheksfunktionen. Bei der Übersetzung des Testobjekts wird die Bibliothek der instrumentierten Software beigefügt.

Eine Instrumentierung der Software wird unabhängig vom Testkriterium vorgenommen, weil die Berechnungen aller entwickelten Zielfunktionen auf den gleichen Meßinformationen basieren.

Testtreibergenerator-Komponente

Der Testtreibergenerator ist für die Vorbereitung des Testobjekts für dynamische Tests verantwortlich. Das Werkzeug erstellt dazu eine Schnittstelle, mit der die generierten Testdaten von der Teststeuerung in die Eingabeparameter der zu testenden Software konvertiert werden. Mit Hilfe der vom Parser zusammengestellten Datentypinformationen und den Verweisen auf Eingabevariablen des Testobjekts kann eine automatische Kodierung eines solchen Testrahmens erfolgen.

Der Testtreibergenerator erzeugt einen Testrahmen, welcher eine Reihe von Funktionen für die Umwandlung eines Testdatums in die nutzerdefinierten Datentypen beinhaltet. Für jeden Datentyp, der in mindestens einem Eingabeparameter verwendet wird, erstellt die Komponente Programmcode zur Initialisierung und Konvertierung der Testdaten. Dabei wird die Zuordung zwischen den Variablen eines Testdatums und den Datenstrukturelementen der Parameter des Testobjekts festgelegt. Die Konvertierungsreihenfolge bestimmt den Aufbau eines Testdatums. In gleicher Abfolge wie die Parameter beim Test eingelesen werden, ordnet der Testtreibergenerator die Wertebereiche der einzelnen Testdatenvariablen. Die Beschreibung der einzelnen Variablen zum Testobjekt erhält die Teststeuerung mit den anderen Strukturinformationen.

Parser-Komponente, Instrumentierungs- und Testtreibergenerierungs-Komponente dienen der Testvorbereitung. Die technische Umsetzung dieser Komponenten wird im Abschnitt 4.6 erläutert.

4.2 Aufgaben der Teststeuerung

Die Teststeuerung verbindet die einzelnen Komponenten des Systems für den automatischen Strukturtest. Sie nimmt die Ergebnisse der Testvorbereitung entgegen, stellt mit den gesammelten Informationen Optimierungsziele zusammen und realisiert den vom Benutzer gewählten dynamischen Strukturtest. Bei der Testdurchführung übernimmt die Steuerung die in Abbildung 4.6 dargestellten Aufgaben. Im einzelnen sind das die Identifikation der zu erfüllenden Teilziele, die Festlegung der Bearbeitungsreihenfolge und die Berechnung der Zielfunktionswerte während der Optimierung eines Teilziels.

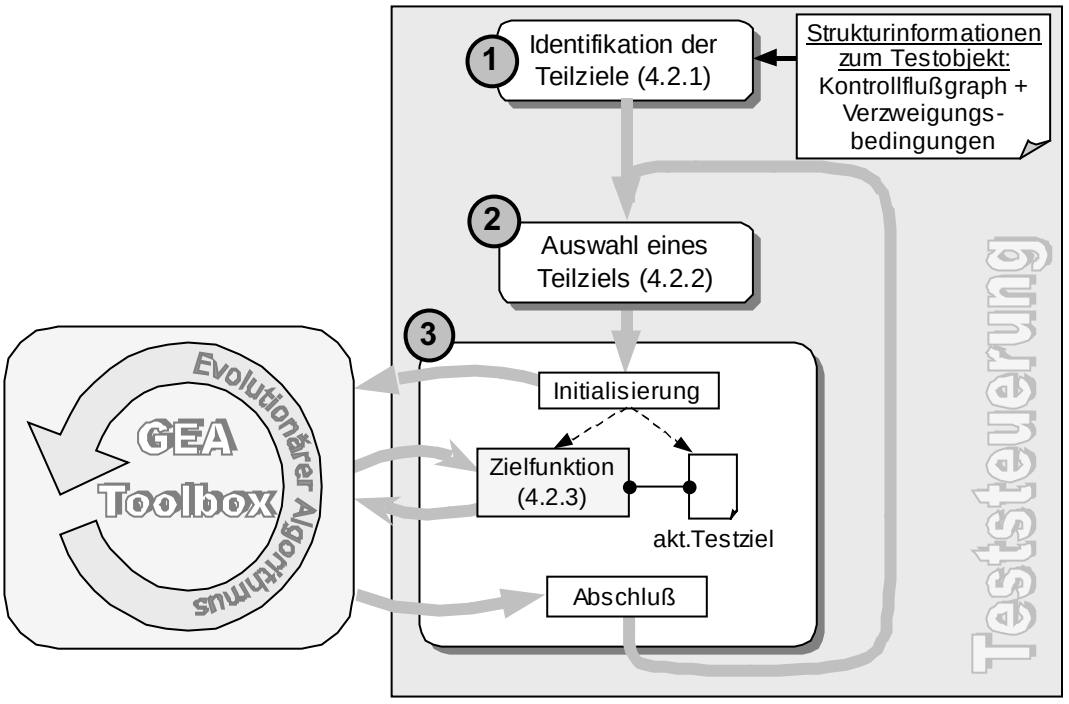


Abbildung 4.6: Teststeuerungsaufgaben

Den Inhalt dieser drei Teilaufgaben erläutern die folgenden Teilabschnitte. Der Teststeuerung stehen, wie Abbildung 4.6 zeigt, Strukturinformationen zum Testobjekt zur Verfügung. Die Daten werden im Vorfeld durch die Systemkomponenten zur Testvorbereitung zusammengestellt. Im ersten Schritt werden daraus je nach gewähltem Testverfahren Teilziele identifiziert (Teilabschnitt 4.2.1).

Die Bestimmung der optimalen Reihenfolge der Teilziele durch die Nutzung der im Optimierungsprozeß gesammelten Testergebnisse beschreibt Teilabschnitt 4.2.2. Es wird eine effiziente Gestaltung des Gesamtprozesses angestrebt.

Die Zielfunktion für die Optimierung eines Teilziels wird anhand von Informationen gebildet, welche bei Ausführung des Testobjekts erfaßt werden. Teilabschnitt 4.2.3 beschreibt die Anforderungen an die Zielfunktionen der einzelnen Strukturtestverfahren.

Für die Berechnung der Funktion und zur Bestimmung von Startwerten für die Optimierung der anderen Teilziele setzt die Teststeuerung einen sogenannten Erreichbarkeitsgraphen ein. Die Methode der Bestimmung der Zielfunktionswerte über einen derartigen Graphen erläutert Teilabschnitt 4.2.4. Der letzte Teil des Abschnittes beschreibt den Optimierungsprozeß sowie Möglichkeiten der Parallelisierung von automatischen Strukturtests.

4.2.1 Identifikation der Teilziele

Die Strategie für die Bestimmung einer Testdatenmenge zum gewählten Testkriterium ist eine Identifikation von Teilzielen, deren Ausführung mit Testeingaben eine Erfüllung des Kriteriums zur Folge hat. Die Teilziele berechnet die Teststeuerung auf Basis der Strukturinformation der zu testenden Software. Es folgt eine gesonderte Betrachtung für die einzelnen Testverfahren.

Anweisungsüberdeckung

Bei der Anweisungsüberdeckung wird eine Testdatenmenge gesucht, die in ihrer Gesamtheit alle Anweisungen des Programms mindestens einmal ausführt. Die Teilziele sind damit die Knoten des Kontrollflußgraphen, da jede Anweisung der zu testenden Software genau einem Knoten zugeordnet ist. Das Durchlaufen eines Knotens ist mit der Ausführung der zugehörigen Anweisungssequenz gleichzusetzen. Die Informationen zu den Anweisungen eines Programms liefert der Kontrollflußgraph, der von der Testvorbereitung erstellt wird. Aus diesem Graphen können die Teilziele entnommen werden.

Zweigüberdeckung

Bei der Zweigüberdeckung wird eine Testdatenmenge gesucht, für die gilt, daß jede Kante des Kontrollflußgraphen mit mindestens einer Eingabe ausgeführt wird. Die einzelnen Teilziele sind damit die Zweige des Kontrollflußgraphen. Ein Teilziel ist durch einen Startknoten und den in diesem Knoten beginnenden Zweig charakterisiert. Wie schon bei der Anweisungsüberdeckung resultieren diese Informationen aus den Strukturinformationen der Testvorbereitung.

Formen der Bedingungsüberdeckung

Es existieren eine Reihe verschiedener Formen der *Bedingungsüberdeckung*. Sie basieren auf der Struktur der Verzweigungsbedingungen der zu testenden Software. Ziel ist die Generierung von Testdaten, die in ihrer Gesamtheit alle Verzweigungsbedingungen ausführen und dabei jeweils Teile der zusammengesetzten Verzweigungsbedingungen mit bestimmten Werten ausführen.

Es sind Teilziele für alle Verzweigungen eines Programms zu bilden. Diese gehen aus dem von der Testvorbereitung erstellten Kontrollflußgraphen hervor. Jeder Verzweigungsanweisung ist ein entsprechender Knoten zugeordnet. Eine weitere Grundlage für die Zusammenstellung der Teilziele sind die Daten zum Aufbau der einzelnen Verzweigungsbedingungen eines Programms. Dazu nötige Angaben enthalten die vom Parser erarbeiteten Strukturinformationen zum Testobjekt. Die Bildung der Teilziele in

den einzelnen Verzweigungspunkten eines Programms beschreiben die folgenden Ausführungen zu den verschiedenen Formen der Bedingungsüberdeckung.

Atomare Bedingungsüberdeckung

Die Teilziele der Verzweigungen einer zu testenden Software bei der *atomaren Bedingungsüberdeckung* bilden die Basisrelationen (*atomare Terme*) einer Bedingung. Einzelne Terme müssen mit Testdaten jeweils einmal zu *Wahr* und einmal zu *Falsch* ausgewertet werden. Nach der Bestimmung aller Verzweigungen werden zu jeder Verzweigungsbedingung die atomaren Terme anhand der Information zum Aufbau der Bedingungen identifiziert und jeweils ein Teilziel mit der Zielstellung Auswertung zu *Wahr* und eins für den Wert *Falsch* gebildet.

Mehrfach-Bedingungsüberdeckung

Ein Verfahren zur Bestimmung der Teilziele für die *Mehrfach-Bedingungsüberdeckung* stützt sich auf die Bildung aller möglichen Kombinationen der atomaren Terme der Verzweigungen eines Testobjekts. Bei der Konstruktion der Teilziele ist zu beachten, daß wegen compilerbedingten Optimierungen bei Programmiersprachen wie *AnsiC* und *Pascal* gegebenenfalls nicht alle Kombinationen von Bedingungen tatsächlich ausgewertet werden können, weil die Berechnung vorzeitig endet und die Werte der restlichen Teilbedingung wegen möglicher Seiteneffekte nicht bestimmt werden können. Solche nicht ausführbaren Teilziele entfallen beim Strukturtest.

Die Ziele der *minimalen Mehrfach-Bedingungsüberdeckung* beziehen sich auf die Auswertungsmerkmale der logischen Verknüpfungen einer Verzweigungsbedingung. Für die n-*stelligen* Und- sowie Oder- Verknüpfungen werden in den Teilzielen Forderungen an die Auswertung der einzelnen Operanden gestellt. Entsprechend der logischen Operationen in der Bedingung einer Verzweigung sind Teilziele für bestimmte Wahrheitswertkombinationen zu bilden.

logische Op.	**zu testende Kombinationen**
Und	- alle Operatoren '*wahr'* - ein Operator '*falsch*', alle anderen '*wahr*'
Oder	- alle Operatoren 'falsch' - ein Operator 'wahr', alle anderen 'falsch'

Tabelle 4.1: Wahrheitswertkombinationen

Eine Einschränkung der Menge der Kombinationsmöglichkeiten ergibt sich auch hier durch die compilerbedingte Optimierung in vielen Programmiersprachen. Für die Operation Und bricht im optimierten Fall die Auswertung ab, sobald ein Operator mit '*falsch*' ausgewertet wird, weil damit der Gesamtwert *'falsch*' feststeht.

Deshalb wird in der *modifizierten minimalen Mehrfach-Bedingungsüberdeckung* nur für diejenigen Operatoren eine Forderung erhoben, die in der jeweiligen Situation ausgewertet werden können [Riedm97]. Das sind für eine Und-Operation alle Kombi-

nationen, bei denen die ersten *k* Operatoren mit *Wahr* ausgewertet werden und der k+1-Operator mit *Falsch*. Eine analoge Festlegung wird für die Oder-Operationen getroffen. Die so entstehenden Kombinationen bilden die Teilziele zu einem Verzweigungsknoten.

Formen der Pfadüberdeckung

Zielstellung der automatischen Pfadüberdeckungstests ist, Testdaten zur Ausführung einer Menge von Pfaden im Kontrollflußgraphen des Testobjekts zu generieren. Die abzuarbeitenden Teilziele des Testverfahrens umfassen damit alle durch das Testkriterium bezeichneten Pfade. Für die Berechnung der Zielpfade erweist sich eine Zerlegung des Kontrollflußgraphen in Abschnitte als günstig. Diese wird zudem später genutzt, um Beziehungen zwischen den Teilzielen bestimmen zu können. Hierzu gehören zum Beispiel ein gleicher Anfangspfad oder gleiche Wegstücke im Kontrollflußgraphen.

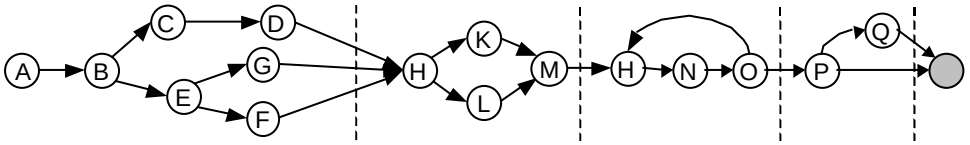


Abbildung 4.7: Kontrollflußgraph mit Abschnitten

Innerhalb der einzelnen Abschnitte des Kontrollflußgraphen, wie Abbildung 4.7 zeigt, können alle Wege identifiziert werden. Ein Teilziel, welches einen vollständigen Pfad umfaßt, setzt sich aus Wegen der einzelnen Abschnitte zusammen. Eine Methode zur Konstruktion aller Teilziele ist die Bildung aller Wegkombinationen in den einzelnen Abschnitten. Im Beispiel der Abbildung 4.7 gibt es drei Wege im ersten Abschnitt, zwei Wege im zweiten Abschnitt und zwei Wege im letzten Abschnitt. Der dritte Abschnitt enthält eine Schleife und wird deshalb gesondert betrachtet.

Eine Schleife im Testobjekt kann eine unbegrenzte Anzahl von Wegen hervorrufen. Für die auszuführenden Pfade der verschiedenen Strukturtestverfahren wird deshalb nur der Weg der ersten *k* Schleifeniterationen festgelegt. Ein vollständiger Pfad setzt sich aus Wegen in den einzelnen Iterationen zusammen. Für die Teilziele werden nur Wegkombinationen der ersten *k* Schleifeniterationen zusammengestellt.

Eine Beschreibung der Abschnittserzeugung erläutert Abschnitt 4.3.4. Die Berechnung der in einem schleifenlosen Abschnitt enthaltenen Wege entspricht einer Tiefensuche in einem gerichteten Graphen. Ein Weg wird aus der Abfolge von durchlaufenen Knoten und Zweigen bis zum Ausgang des Abschnitts gebildet. Enthält ein Abschnitt eine Schleife, so sind alle Wege, welche genau eine Schleifeniteration ausführen, zu bestimmen. Die auszuführenden Pfade in einem solchen Graphen werden durch die Verknüpfung von k Wegen einer Iteration gebildet. Einige Strukturtestkriterien fordern die identische Wiederholung, so daß sich die Anzahl der Möglichkeiten reduziert.

4.2.2 Strategie für die Teilzielauswahl

Zielsetzung bei der Entwicklung des Strukturtestsystems ASTELA ist die optimale Gestaltung des Suchprozesses einer Testdatenmenge zur Erfüllung des gewählten Strukturtestkriteriums. Dazu wurden die Methoden anderer Arbeiten betrachtet. Das

Vorgehensmodell bei der Bearbeitung der Teilziele in der Arbeit [Jones96] entspricht der sogenannten Breitensuche (*breath-first-search*). Die Testergebnisse der Optimierung zu einem Teilziel fließen im dort entwickelten Prototypen nicht in die Optimierung des nächsten Teilziels ein. Deshalb wirkt sich hier die Reihenfolge der Bearbeitung nur auf das zufällige Erreichen von Teilzielen in tiefer gelegenen Knoten des Kontrollflußgraphen aus.

Die Arbeit von [Prather87] schlägt als Reihenfolge für die Testdatengenerierung im Fall der Zweigüberdeckung die Anordnung im Kontrollflußgraphen vor. Beginnend beim Eingangsknoten werden die Zweige der Knoten des Graphen sukzessive abgearbeitet. Diese Reihenfolge verspricht, daß bei der Testdatengenerierung zu einem Zweig am Anfang des Kontrollflußgraphen nachfolgende Zweige zufällig erreicht oder gute Näherungen an diese gefunden werden. Die Autoren nutzen die Testergebnisse für die im Kontrollflußgraphen nachfolgenden Teilziele.

Die Suche nach Testdaten, welche ein bestimmtes Teilziel erfüllen, kann effektiver gestaltet werden, wenn vorher zufällig gefundene Individuen mit guter Bewertung in die Optimierung einfließen können. Zu diesem Zweck muß das System bei der Optimierung eines Teilziels gleichzeitig eine Bewertung für alle anderen Teilziele vornehmen können, um gute Startwerte zu bemerken.

Das System ASTELA verwendet eine dynamische Anpassung der Reihenfolge anhand der laufend gesammelten Startwerte für jedes Teilziel. Bei der Auswahl des nächsten zu optimierenden Teilziels werden die vorliegenden Startwerte analysiert. Dieses Vorgehen läßt eine noch effizientere Gestaltung des gesamten Prozesses erwarten. Es bedarf keiner Betrachtung von Nachbarschaftsbeziehungen[1] zwischen den Teilzielen, wie bei [Prather87] speziell für Zweige (*closeness phenomen of test goals*).

Eine Abfolge der Optimierungen der Teilziele kann dynamisch mit den jeweils vorliegenden Testdaten entschieden werden. Die jeweils nächste Optimierung wird mit dem Teilziel begonnen, für das die am besten bewerteten Testdaten vorliegen. Der Grundgedanke dieser Methode ist, daß eine gute Bewertung der vorliegenden Testdaten für ein Teilziel auf eine schnelle Bestimmung des Optimums hindeutet. Währenddessen können für alle anderen Teilziele, vor allem für solche, die in der Nähe des ausgewählten Teilziels liegen, weiterhin gute Startwerte gesammelt werden.

Falls die Suche nach einer geeigneten Eingabe für ein Teilziel fehlschlägt, bedeutet dies nicht, daß es keine passende Eingabe gibt – das Teilziel also nicht erfüllbar ist. Möglicherweise führt ein zweiter Optimierungsversuch zum Ziel. Das System ASTELA markiert fehlgeschlagene Teilziele für einen erneuten Versuch, nachdem alle Ziele abgearbeitet wurden. Wird für ein fehlgeschlagenes Teilziel während einer Optimierung zufällig eine bessere Annäherung gefunden, so wird das Teilziel erneut für eine Optimierung freigegeben.

[1] Zwei Teilziele sind benachbart, wenn das Erreichen des einen Ziels auch in die Nähe des anderen Teilziels führt.

4.2.3 Berechnung von Zielfunktionswerten

Für eine evolutionäre Optimierung (Abschnitt 3.2) der Teilziele eines Strukturtestverfahrens wird eine geeignete Bewertung der generierten Testdaten benötigt. Sie muß eine Annäherung der Testdaten an ein Teilziel ausdrücken. Die Konstruktion einer solchen Zielfunktion richtet sich deshalb nach den Zielstellungen eines Testkriteriums. Für die verschiedenen Arten von Teilzielen wurde in Abschnitt 2.3 eine Klassifikation vorgestellt, an der sich die Bildung der Zielfunktion orientiert.

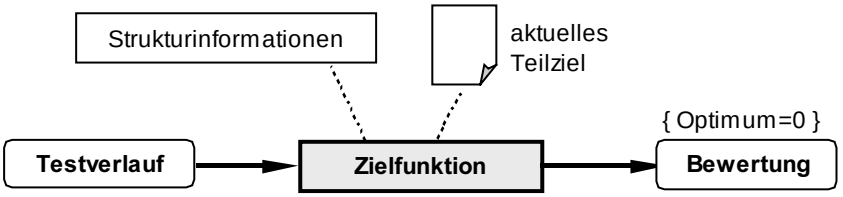


Abbildung 4.8: Zielfunktion und die Parameter einer Bewertung

Wie Abbildung 4.8 verdeutlicht, basiert die Bewertung von Testdaten im System *ASTELA* bei allen strukturorientierten Verfahren auf dem protokollierten *Testverlauf*. Die Zielfunktion vergleicht den Verlauf und das aktuelle Teilziel und leitet daraus eine Bewertung ab. Für die Berechnung der Zielfunktionswerte stehen ihr die Strukturinformationen zum Testobjekt zur Verfügung. Damit eine einfache Formulierung des Abbruchkriteriums möglich ist, werden die Zielfunktionen für alle Testverfahren so konstruiert, daß das Optimum bei Null liegt.

Ein *Testverlauf* ist aus strukturorientierter Sicht die Folge von ausgeführten Knoten und Verzweigungen. Abweichungen eines Testverlaufs von einem Teilziel werden durch *unerwünschte* Verzweigungen verursacht. Wie in [Korel90] vorgeschlagen, dient eine *Abstandsfunktion* (dort *Verzweigungsfunktion*) der Bedingungen einer unerwünschten Verzweigung zur Formulierung des Optimierungsziels. Die Abstandsfunktion ist jedoch nur ein Teil einer Zielfunktion, weil sie die Annäherung an ein Teilziel in nur einem Verzweigungspunkt ausdrückt. Mit der Entwicklung einer zweckmäßigen Abstandsfunktion beschäftigt sich Abschnitt 4.4.

Weil die Zielstellungen in den Verzweigungen für die einzelnen Klassen von Testverfahren sehr unterschiedlich definiert sind, wird im folgenden eine getrennte Betrachtung der Zielfunktionen vorgenommen. Zum Beispiel ist bei einem *pfadorientierten* Testverfahren die Ausführung bestimmter Pfade der zu testenden Software notwendig. Eine Abweichung des Testverlaufs kann durch direkten Vergleich mit dem Zielpfad gemessen werden. Dagegen spielt bei der Bewertung eines Individuums für *knotenorientierte* Teilziele der genaue Weg im Testobjekt keine Rolle. Es zählt die Ausführung des Zielknotens. In diesem Fall lassen sich Verzweigungen identifizieren, von deren Entscheidung ein Pfad durch den Knoten abhängig ist. Diese dienen der Bildung einer Zielfunktion für knotenorientierte Verfahren.

Zielfunktion für knotenorientierte Testverfahren

Teilziel der knotenorientierten Testverfahren ist die Ausführung eines bestimmten Knotens unabhängig von dem zuvor durchlaufenen Pfad. Um den Suchraum nicht unnötig einzuschränken, werden die möglichen Wege zu einem Zielknoten anders als bei [Jones96] und [Pargas99] nicht getrennt betrachtet.

Die Abbildung 4.9 zeigt ein Beispiel mit drei Testverläufen. Deutlich wird die besondere Stellung der Verzweigungsknoten 2, 4 und 5. Sie werden als *entscheidende* Verzweigungen bezeichnet, da sie Zweige besitzen, deren Ausführung ein Verfehlen des Zielknotens bewirkt. Ein Kontrollfluß über solche *unerwünschten* Zweige führt in Bereiche des Kontrollflußgraphen, von denen aus der Zielknoten nicht erreichbar ist. Zu jedem Testdatum, das ein Teilziel nicht erreicht, kann die erste *entscheidende* Verzweigung identifiziert werden, bei der ein unerwünschter Zweig ausgeführt wird. In den Beispielen aus Abbildung 4.9 sind das gemäß der Reihenfolge der Testfälle die Knoten 2, 4 und 5.

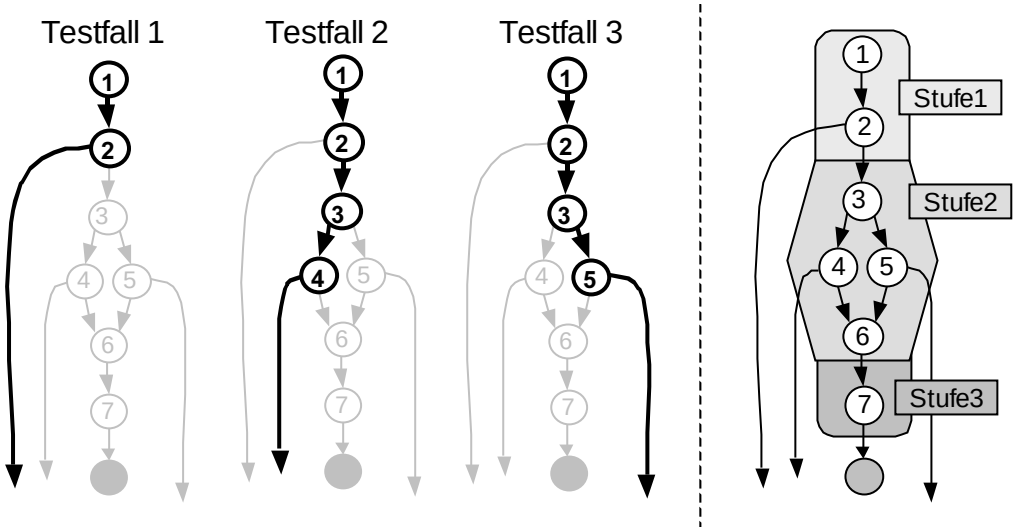


Abbildung 4.9: Beispiel Annäherungsstufen

Die Annäherungsstufe, welche jeder Verzweigung zugeordnet ist, ermöglicht einen Vergleich von Testfällen, die in unterschiedlichen Verzweigungen des Testobjekts das Teilziel verfehlen. Diese Stufen geben eine Optimierungsrichtung vor. Je höher die erreichte Annäherung ist, desto besser wird das Testdatum bewertet. Die Abstufung richtet sich nach der Anordnung der entscheidenden Verzweigungen auf den Wegen zum Zielknoten. Eine Vergabe der Annäherungsstufen ist so abzustimmen, daß alle möglichen Wege zu einem Zielknoten bei der Bewertung *gleichrangig* bleiben. Das bedeutet, daß kein Weg bessere Zielfunktionswerte erhält, weil er kürzer ist oder weniger Verzweigungen enthält.

Dieser Bewertungsansatz erfüllt die Anforderungen an eine Zielfunktion, eine Wertung unabhängig vom Weg zum Zielknoten vorzunehmen. Es erhält das Testdatum die beste Bewertung, das die höchste Annäherungsstufe und den kleinsten Abstand zu einer erwünschten Verzweigung erreicht.

Die *entscheidenden* Verzweigungen in Abbildung 4.9 sind durch unterschiedliche Annäherungsstufen gekennzeichnet. Testfall 1 erhält eine schlechtere Bewertung als Testfall 2, weil die Annäherung in Knoten 2 geringer ist als die von Knoten 4. Die Testfälle 2 und 3 scheitern auf der gleichen Annäherungsstufe.

Für die genaue Festlegung der Zielfunktion muß die Bildung der Annäherungsstufen beschrieben werden. Die Herleitung der Stufen auf Grundlage einer Analyse der Ausführungsbeziehungen der Verzweigung im Testobjekt erläutert Abschnitt 4.3.1, der sich neben diesem Problem auch mit der Formulierung einer vollständigen Zielfunktion beschäftigt. Für zusätzliche Forderungen eines knotenorientierten Teilziels, wie zum Beispiel bei Bedingungsüberdeckungstests, muß der vorgestellte Bewertungsansatz geringfügig erweitert werden. Die Teilabschnitte 4.3.2 und 4.3.3 untersuchen zwei Sonderfälle der knotenorientierten Testverfahren.

Zielfunktion für pfadorientierte Testverfahren

Teilziel der pfadorientierten Testverfahren ist die Ausführung eines bestimmten Weges durch den Kontrollflußgraphen. Es gibt zwei Methoden die Näherung eines Testverlaufs an das Teilziel zu beschreiben. Die erste Möglichkeit vergleicht Testverlaufs- und Zielpfad beginnend beim Startknoten und vergibt eine Annäherungsstufe anhand der Länge des identischen Anfangsstücks. Eine zweite Möglichkeit ist die Betrachtung des Gesamtpfades und der Bewertung aller identischen Wegstücke von Testverlaufs- und Zielpfad.

Die erste Methode geht davon aus, daß für eine weitere Annäherung die Fehlverzweigung nach einem gewissen identischen Anfangsstück zu korrigieren ist. Sie entspricht damit dem pfadorientierten Bewertungsansatz von [Korel90]. Eine zielgerichtete Optimierung in der Fehlverzweigung kann auf Grundlage der Abstandsfunktion zur Verzweigungsbedingung erfolgen. Der Zielfunktionswert wird aus der Länge des identischen Anfangsstückes und dem Abstand zum auszuführenden Zweig gebildet.

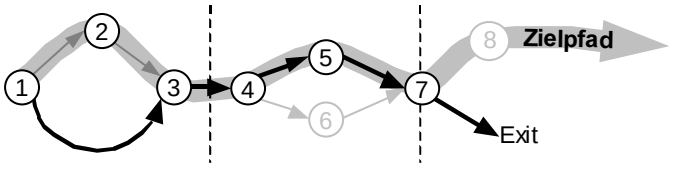


Abbildung 4.10: Vergleich zweier Pfade in Abschnitten des Kontrollflußgraphen

Mit der zweiten Methode wird eine Annäherung über die Vergrößerung der identischen Wegstücke im Gesamtpfad angestrebt. Für diesen Bewertungsansatz erweist sich eine Einteilung des Kontrollflußgraphen in Abschnitte als günstig, wie sie schon für die Identifikation der Teilziele bei pfadorientierten Testverfahren eingesetzt wird.

Die Bewertung eines Testdatums für einen Abschnitt wird, wie bei der ersten Methode, durch die Länge des identischen Weges bestimmt. Der Zielfunktionswert berechnet sich aus der Summe der Bewertungen für alle durchlaufenen Abschnitte.

Die Bestimmung der Abschnitte sowie weitere Festlegungen für die verschiedenen Pfadtestverfahren beschreibt Abschnitt 4.3.4.

Zielfunktion für knoten-wegorientierte Testverfahren

Ein Teilziel von knoten-wegorientierten Testverfahren umfaßt zwei Forderungen, welche in die Bewertung von Testdaten einbezogen werden müssen. Zunächst verlangt ein Teilziel das Erreichen eines bestimmten Knotens. Durchläuft ein Testdatum einen solchen Pfad, fordert das Teilziel im weiteren die Ausführung eines bestimmten Weges beginnend in diesem Knoten.

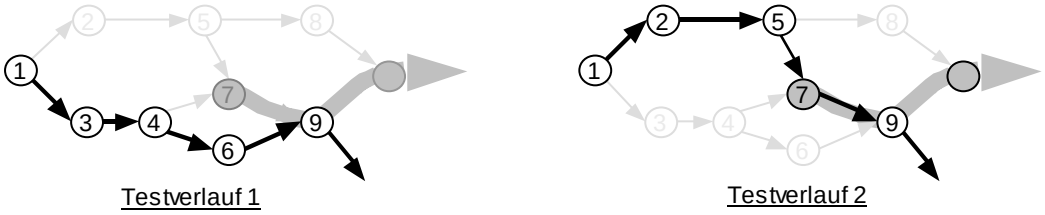


Abbildung 4.11: Testszenario knoten-wegorientierter Verfahren

Die Bewertung von Testverläufen, welche die erste Forderung nicht erfüllen, kann entsprechend der knotenorientierten Testverfahren erfolgen. Diesen Fall stellt die Abbildung 4.11 im ersten Testverlauf dar. Wird der Zielknoten durch ein Testdatum erreicht, wie die Abbildung anhand des zweiten Beispiels zeigt, muß die Methode zur Zielfunktionsberechnung den nachfolgenden Weg mit der Zielsetzung vergleichen. Dafür kann der Bewertungsansatz der pfadorientierten Testverfahren genutzt werden.

Zielfunktion für knoten-knotenorientierte Testverfahren

Die Teilziele der knoten-knotenorientierten Testverfahren beinhalten bestimmte Knotenkombinationen. Nach der Ausführung des ersten Knoten ist im Gegensatz zu den knoten-wegorientierten Teilzielen ein zweiter Knoten ohne Vorgabe des Weges zu durchlaufen.

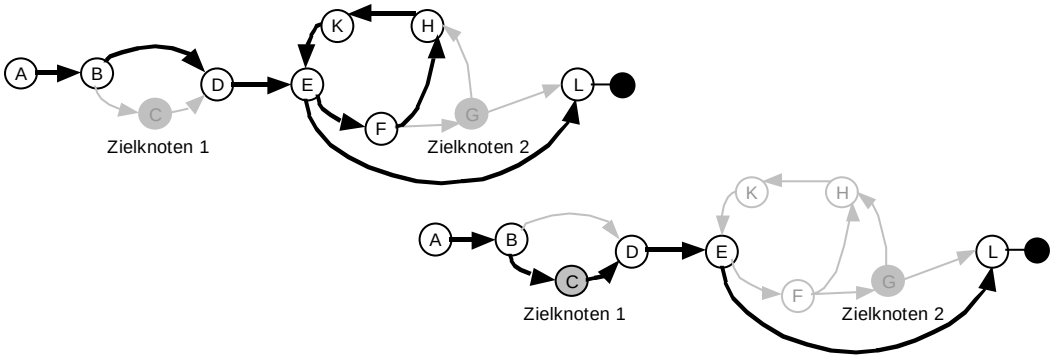


Abbildung 4.12: Testszenario für knoten-knotenorientierte Verfahren

Die Annäherung an den ersten Zielknoten kann wie bei den knotenorientierten Verfahren bewertet werden. Einen beispielhaften Testverlauf zeigt Abbildung 4.12 im ersten Graphen. Für alle Testdaten, deren ausgeführter Pfad durch den ersten Zielknoten geht, muß der Zielfunktionswert die Annäherung an den zweiten Knoten zum Ausdruck bringen. Diesen Fall stell der zweite Testverlauf in Abbildung 4.12 dar.

4.2.4 Rolle des Erreichbarkeitsgraphen

Die im vorangegangenen Teilabschnitt beschriebenen Zielfunktionen berechnen die Bewertung eines Testdatums durch den Vergleich des ausgeführten Kontrollflusses (Testverlauf) mit dem gewählten Teilziel. Bei diesem Vergleich wird der Kontrollfluß in bestimmten Verzweigungen des Programms überprüft und bei einer Abweichung von der Zielstellung ein Zielfunktionswert gebildet. Je nach Teilziel sind die zu überwachenden Zweige sehr unterschiedlich.

Ein sogenannter Erreichbarkeitsgraph bildet die Grundlage der Methode zur gleichzeitigen Zielfunktionsberechnung für alle Teilziele. Während der Kontrollfluß in diesem Graphen nachvollzogen wird, können die Ausführung oder die Abweichung von Teilzielen festgestellt und bewertet werden. Der Erreichbarkeitsgraph vereinigt die Berechnung für alle Teilziele eines Testverfahrens. Mit diesem Bewertungsverfahren bietet sich die Möglichkeit, zufällig erreichte Teilziele zu erkennen sowie für jedes Teilziel die Testdaten mit der besten Bewertung zusammenzustellen. Diese Daten dienen als *Startwerte* für die Optimierung eines Teilziels. Auf Grund der unterschiedlichen Arten von Zielfunktionen für die jeweiligen Testverfahren gestalten sich die Knoten und Zweige des Erreichbarkeitsgraphen verschieden.

Knotenorientierte Verfahren

Für die Zielfunktionsberechnung der knotenorientierten Testverfahren ist das Durchlaufen sogenannter *entscheidender* Verzweigungen zu überwachen. Führt ein Testfall eine für das aktuelle Teilziel entscheidende Verzweigung mit einem unerwünschten Zweig aus, so wird das Ziel verfehlt. Ansonsten erreicht er den Zielknoten.

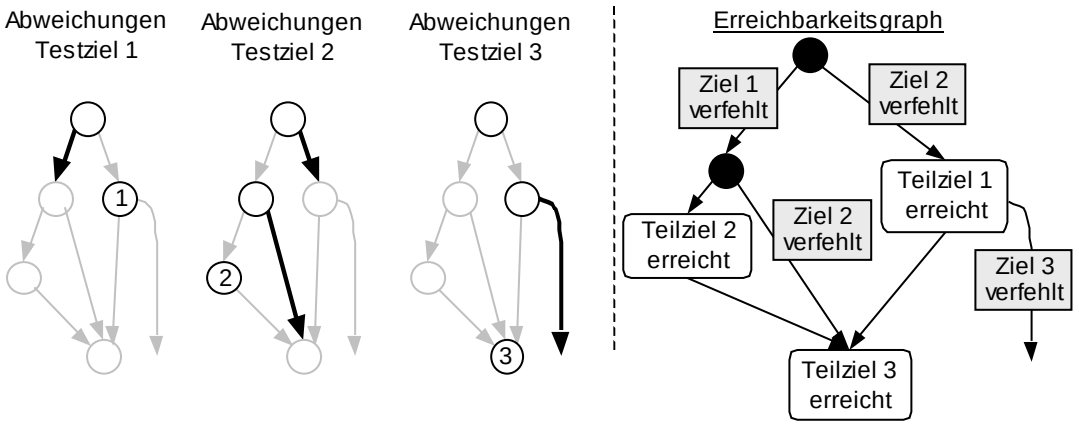


Abbildung 4.13: Graph zur Bestimmung der Abweichung von Teilzielen

Die Abbildung 4.13 zeigt ein Beispiel mit drei Zielknoten und den möglichen Abweichungen (*hervorgehobene Zweige*). Diese Information kann leicht in einem zum Kontrollflußgraphen ähnlichen Graphen zusammengefaßt werden, wie die Abbildung auf der rechten Seite demonstriert. In den Knoten und Zweigen sind erreichte Teilziele und unerwünschte Zweige (*mit Angabe des Teilziels*) vermerkt.

Weil die Knoten- und Zweigabfolge in einem solchen Graphen identisch mit dem Kontrollflußgraphen ist, kann die Folge der Verzweigungsentscheidungen eines beliebigen Tests nachvollzogen werden. Während der Rekonstruktion des Testverlaufs im Erreichbarkeitsgraphen werden anhand der Eintragungen alle erreichten und verfehlten Teilziele identifiziert sowie die Zielfunktionswerte für *alle* Teilziele bestimmt.

Pfadorientierte Verfahren

Die Zielfunktionen der pfadorientierten Testverfahren vergleichen einen Testverlaufspfad mit einem Zielpfad. Bei dieser Form des Strukturtests dient der Erreichbarkeitsgraph nur indirekt der gleichzeitigen Bewertung eines Testverlaufs für alle Teilziele. Hauptaufgabe ist die Erkennung zufällig erreichter Zielpfade sowie die Zusammenstellung von Startwerten für die Ausführung von Wegstücken des Kontrollflußgraphen.

Auf Grundlage der durch den Erreichbarkeitsgraphen erfaßten Startwerte für einzelne Wegabschnitte des Testobjekts kann für die Initialisierung der Optimierung eines Teilziels eine Menge von Testdaten mit guter Bewertung gebildet werden. Mit dieser Methode wird eine aufwendige Erfassung von Startwerten für jedes einzelne Teilziel umgangen. Der Aufwand ergibt sich aus der im allgemeinen sehr großen Anzahl zu bearbeitender Teilziele.

Wie in Abbildung 4.14 dargestellt, repräsentieren die Knoten des Erreichbarkeitsgraphen die Abschnitte des Kontrollflusses. Die Zweige entsprechen den möglichen Wegen im zugehörigen Abschnitt.

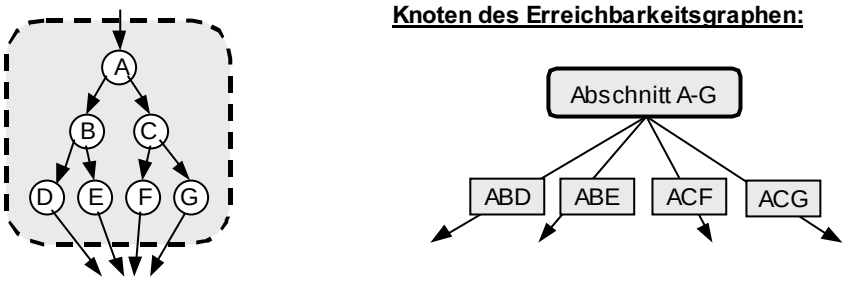


Abbildung 4.14: Knoten des Erreichbarkeitsgraphen für pfadorientierte Teilziele

Der Erreichbarkeitsgraph umfaßt alle Kombinationen der Wege in den einzelnen Abschnitten des Kontrollflußgraphen. Er ist baumartig gegliedert. Die Teilziele bilden die Blattknoten. Der so aufgebaute Graph erlaubt die Rekonstruktion des Testverlaufs und Bewertung der Annäherungen an die Wege der einzelnen Abschnitte. Aus diesen Ergebnissen wird der Zielfunktionswert ermittelt.

Die folgenden zwei Abbildungen zeigen einen Kontrollflußgraphen und einen Ausschnitt vom zugehörigen Erreichbarkeitsgraphen.

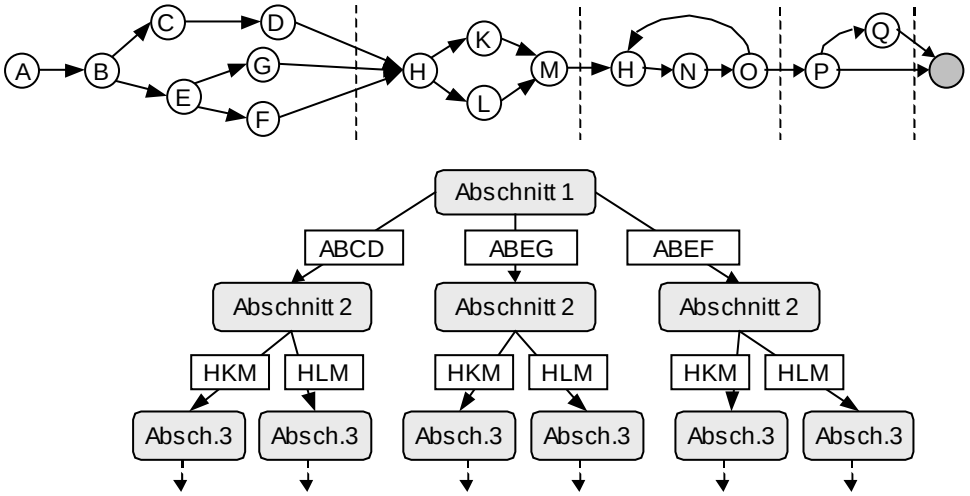


Abbildung 4.15: Ausschnitt des Erreichbarkeitsgraphen für den pfadorientierten Test

Zur Erfassung von Startwerten für die Berechnungsmethode des identischen Anfangsstücks kann eine getrennte Aufnahme der Startwerte in den Knoten des Erreichbarkeitsgraphen vorgenommen werden. Das hat den Effekt, daß nur Testdaten mit einem identischen Anfangsstück zusammengestellt werden. Dagegen sammelt die andere Methode die Testdaten entsprechend der Abschnitte, also Knoten übergreifend.

Mit der Konstruktion und Nutzung des Graphen im System *ASTELA* beschäftigt sich der Abschnitt 4.5 "Erreichbarkeitsgraph".

4.2.5 Optimierungsprozeß

Die im vorangegangenen Teilabschnitt beschriebene Methode zur Berechnung der Zielfunktionswerte für alle Teilziele in einem Graphen erlaubt dem Optimierungsprozeß neben der Bewertung der Testdaten für das gewählte Teilziel die Bestimmung der Eignung für andere Teilziele.

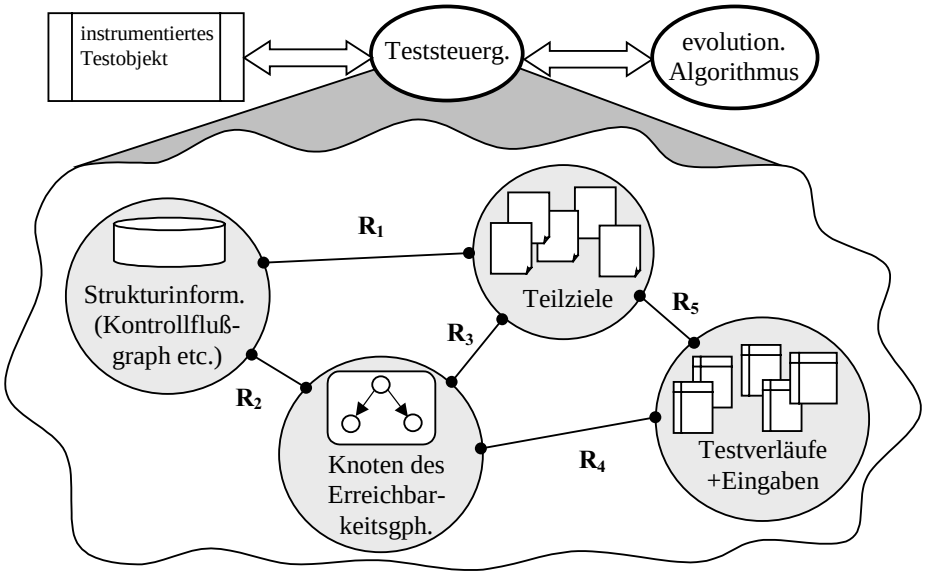


Abbildung 4.16:Datenmodell Teststeuerung

Für den Optimierungsprozeß im System ASTELA wurde das in Abbildung 4.16 dargestellte Datenmodell (*Repository*) entworfen. Das Schema zeigt die Teststeuerung in Zusammenarbeit mit dem evolutionären Algorithmus und dem instrumentierten Testobjekt. Des weiteren werden in der Abbildung die verschiedenen Datenquellen dargestellt. Dazu gehören die Strukturinformationen zum Testobjekt, die generierten Teilziele, eine Sammlung der protokollierten Testverläufe sowie der konstruierte Erreichbarkeitsgraph. Diese Datenquellen stehen durch die Relationen R_1 bis R_5 in enger Verknüpfung.

Relation R_1 drückt die Beschreibung der Teilziele anhand von Strukturinformation aus. Die Verweise zwischen den beiden verbundenen Datenquellen ist je nach Testverfahren unterschiedlich. Zum Beispiel verweisen die Teilziele der Bedingungsüberdeckung auf die strukturelle Information der ihnen zugeordneten Bedingungen.

Die Relation R_2 steht für die Beschreibung der Knoten des Erreichbarkeitsgraphen durch Strukturinformationen zum Testobjekt. Beim knotenorientierten Test entsprechen die Knoten des Graphen den Verzweigungen des Testobjekts. Andererseits geben die Knoten bei pfadorientierten Tests ganze Abschnitte im Kontrollflußgraphen wieder, die durch die Relation beschrieben werden.

Eine Zuordnung der Teilziele zu den Knoten des Erreichbarkeitsgraphen realisiert die Relation R_3. Sie dient der Berechnung der Zielfunktion für die Teilziele mit Hilfe des Erreichbarkeitsgraphen (siehe Teilabschnitt 4.2.4).

Die Relationen R_4 und R_5 werden entsprechend des gewählten Testverfahrens gebildet. Im Fall eines pfadorientierten Tests drückt die Relation R_4 eine Beziehung zwischen den Testverläufen und den Knoten des Erreichbarkeitsgraphen aus. Damit werden gute Startwerte für bestimmte Wegstücke zusammengefaßt. Für die knotenorientierten Verfahren wird eine Relation R_5 konstruiert. Sie ordnet den Teilzielen die besten Testverläufe direkt zu. Mit Hilfe dieser Relation können während der Optimierung an jedes Teilziel gute Startwerte gebunden werden.

Der Prozeß

Im Optimierungsprozeß der Teststeuerung werden *offene*, *erreichte* und *verfehlte* Teilziele unterschieden. Konnte ein Teilziel ausgeführt werden, so liegen Testdaten vor und das Ziel gilt als *erreicht.* Ein Teilziel wird als *verfehlt* bezeichnet, wenn eine Optimierung nicht erfolgreich war und die Suche abgebrochen wurde. Teilziele, die noch nicht optimiert wurden, besitzen den Status *offen*.

Die Reihenfolge der Abarbeitung der Teilziele bestimmt sich während der einzelnen Optimierungen und ist damit dynamisch. Dies gilt auch für den Status der Teilziele. Findet ein Suchprozeß zufällig für ein *verfehltes* Teilziel einen besseren Startwert, so merkt sich die Steuerung das Teilziel für eine erneute Optimierung vor, in dem der Status wieder auf *offen* gesetzt wird. Die Teststeuerung beendet die Testdatengenerierung, wenn nur noch *erreichte* und *verfehlte* Teilziele vorliegen.

Überlegungen zur Parallelisierung

Die Parallelisierung der Optimierungsprozesse verschiedener Teilziele bedarf der folgenden Anpassungen:

- zentrale Datenverwaltung (Repository) für parallelen Zugriff auf die Relationen
- Einstreuen (*Seeding*) guter Startwerte während der Optimierung eines Teilziels (Zufallsergebnisse anderer Optimierungen)
- Erweiterung des Teilzielstatus auf *erreicht*, *verfehlt*, *offen* und *wird optimiert*

Folgende Regeln werden im Falle der Bestimmung besserer Startwerte für ein beliebiges Teilziel eingehalten:

(1.) Findet ein Optimierungsprozeß einen besseren Startwert für ein *verfehltes* Teilziel, so wird dessen Status wieder auf den Wert *offen* geändert.

(2.) Findet ein Prozeß für ein Teilziel, welches gerade durch einen anderen Prozeß optimiert wird, bessere Startwerte, so werden diese Werte in die laufende Optimierung einbezogen (*Seeding - Einstreuen*).

(3.) Gute Startwerte für offene Teilziele werden zentral gesammelt, bis ein Optimierungsprozeß für das Teilziel gestartet wird.

4.3 Zielfunktion

Die Bestimmung von Zielfunktionswerten, die eine Näherung der generierten Testdaten an das Teilziel beschreiben, basiert auf Messungen während der Ausführung des Testobjekts. Dabei werden für jedes generierte Testdatum durchlaufene Programmstrukturen erfaßt und Informationen zu den ausgewerteten Verzweigungsbedingungen zusammengestellt.

J.C. Huang vergleicht in seiner Arbeit [Huang79] das Instrumentieren von Software mit der Überprüfung technischer Systeme durch Meßgeräte. Der Test der Funktionalität erfolgt dort durch eine Auswahl von Messungen an verschiedenen Teilkomponenten des Systems. Das Problem der Auswahl der zu messenden Komponenten ergibt sich auch für den Test von Software. Häufig wird in der Literatur eine Instrumentierung mit Hilfe von Zählern vorgeschlagen ([Balzert98] und [Huang79]). Bei diesem Ansatz wird im Testobjekt geprüft, welche Systemkomponenten (*Anweisungen*) ausgeführt werden. Diese Information reicht für evolutionäre Tests nicht aus, um eine geeignete Zielfunktion bilden zu können.

Für eine zweckmäßige Zielfunktion wird in Anlehnung an die Arbeit von [Jones96] eine Instrumentierung der Verzweigungsbedingungen vorgenommen. Diese Messungen werden in einer Abstandsfunktion normalisiert und in die Berechnung der Zielfunktion einbezogen. Mit dieser Methode fließen die Daten in die zu optimierende Zielfunktion ein, die auf Entscheidungen im Kontrollfluß wirken. Eine Instrumentierung der Verzweigungsbedingungen ermöglicht daneben auch die Rekonstruktion des ausgeführten Kontrollflusses. Wie beim Test technischer Systeme dürfen die Messungen keinen

Einfluß auf die Funktionalität der zu untersuchenden Software nehmen. Dies betrifft insbesondere Seiteneffekte in der Auswertung von Verzweigungsbedingungen durch enthaltene Funktionsaufrufe sowie Variablenzuweisungen. Im Abschnitt 4.6 werden die notwendigen Testvorbereitungen, darunter auch die *Instrumentierung* des Testobjekts, erläutert.

Wie schon in der Einführung zur Berechnung der Zielfunktionswerte in Abschnitt 4.2.3 erläutert, richtet sich die Bewertung einer Testeingabe nach der Klasse des Testverfahrens. Die folgenden Teilabschnitte stellen die für die verschiedenen Verfahren entwickelten Zielfunktionen vor. Alle Funktionen sind so formuliert, daß ihr Optimum im Wert Null liegt. Auf diese Weise läßt sich leicht ein Abbruchkriterium für die evolutionären Algorithmen formulieren.

Einen wichtigen Bestandteil der Zielfunktion bildet eine Abstandsfunktion für bestimmte Verzweigungen einer zu testenden Software. Die folgenden Ausführungen gehen von der Existenz einer solchen Funktion aus. Sie wird auf Grundlage von Messungen an den Teilbedingungen der Verzweigungen berechnet. Abbildung 4.17 zeigt eine Verzweigung mit ausgeführtem 'wahr'-Zweig. In diesem Beispiel ist der Abstand zum 'falsch'-Zweig funktional auszudrücken. Wie eine solche Abstandsfunktion gebildet werden kann, zeigt Abschnitt 4.4.

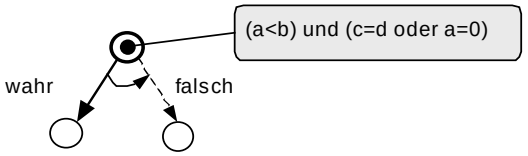


Abbildung 4.17: Abstandsfunktion in einer Verzweigung

4.3.1 Zielfunktion für knotenorientierte Testverfahren

Bei den knotenorientierten Tests richtet sich die Zielfunktion nach bestimmten Verzweigungen des Testverlaufs. Zu jedem Testdatum, welches den Zielknoten nicht ausführt, kann eine Verzweigung bestimmt werden, nach deren Ausführung kein Weg mehr zum Ziel möglich ist. Diese *entscheidende* Verzweigung dient als Richtungsvorgabe für die Optimierung. Es sollen Testdaten generiert werden, die möglichst nah an den Zielknoten gelangen. Das Optimum ist die Ausführung des Teilziels. Auf Grundlage der entscheidenden Verzweigungen kann die folgende Zielfunktion gebildet werden. Sie stützt sich auf eine Annäherungsstufe und einer zu den Verzweigungsbedingungen gebildeten Abstandsfunktion.

Formel (4.3)

Zielfunktion(Testverlauf) : = *höchste Annäherungsstufe*+1 - *Annäherung*(Testverlauf)

Annäherung(Testverlauf) : = *Abstand zum erwünschten Zweig* +
Annäherungsstufe der entscheidenden Verzweigung

Die Annäherungsstufe ergibt sich aus der Kontrollflußanalyse des Testverlaufs zum Testdatum. Sie wird durch die erste *entscheidende* Verzweigung bestimmt, in der ein unerwünschter Zweig ausgeführt wird. Die Auswahl des erwünschten Zweigs mit

dessen Ausführung weiter in Richtung Zielknoten vorangeschritten wird und die Bestimmung eines Abstands zu diesem Zweig erfolgt auf Grundlage der Messungen an den Verzweigungen des durchlaufenen Pfads. Die Abstandsfunktion drückt einen Übergang zwischen den Annäherungsstufen aus und ist damit wichtiger Bestandteil einer geeigneten Zielfunktion.

Kann ein Testdatum generiert werden, das den erwünschten Zweig durchläuft, so führt der resultierende Weg weiter in Richtung des Zielknoten. Dabei erreicht der Testverlauf das Ziel beziehungsweise eine höhere Annäherungsstufe. In beiden Fällen erhält er eine bessere Bewertung und dient der weiteren Optimierung mit dem evolutionären Algorithmus.

Bei der Optimierung gilt es in erster Linie eine möglichst große Annäherungsstufe zu erlangen. Die Stufe erhält deshalb in der Formel das größere Gewicht. Eine Abstandsfunktion zum erwünschten Zweig beschreibt hingegen den Übergang zwischen den Annäherungsstufen. Die folgenden Ausführungen verdeutllichen die Berechnung dieser Funktion. Danach wird die Methode zur Festlegung der Annäherungsstufen in einem beliebigen Kontrollflußgraphen beschrieben.

Abstandsfunktion zum erwünschten Zweig

Eine Abstandsfunktion leitet die Optimierung zwischen zwei Annäherungsstufen. Sie wird anhand der Verzweigungsbedingungen im Programm gebildet, die den Kontrollfluß in den entscheidenden Verzweigungen steuern. Ein Testverlauf umfaßt im allgemeinen mehrere Verzweigungen, die angepaßt werden können, damit eine höhere Annäherungsstufe erreicht wird. Abbildung 4.18 zeigt Beispiele mit den zugehörigen Abstandsmessungen.

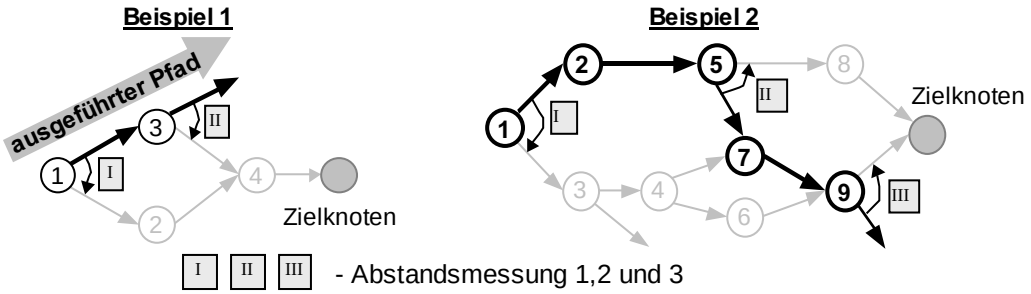


Abbildung 4.18: Erwünschter Zweig

Eine Anpassung der Testdaten an eine bestimmte Verzweigungsentscheidung des Testverlaufs wird erreicht, wenn die entsprechende Abstandsmessung in der Zielfunktion verwendet wird. Auf diese Weise werden alle diejenigen Testfälle besser bewertet, welche den kleinsten Abstand zur gewünschten Entscheidung besitzen. Es muß geklärt werden, welcher Abstand der durchlaufenen Verzweigungen für die Zielfunktion verwendet wird, damit eine gerichtete Optimierung möglich ist.

Im ersten Beispiel der Abbildung 4.18 verfehlt der ausgeführte Pfad das Teilziel im Verzweigungsknoten 3. In diesem Fall hat sowohl eine Veränderung der Verzweigung im Knoten 1 wie auch eine andere Entscheidung im Knoten 3 die Verbesserung des

Testverlaufs zur Folge. In beiden Fällen bewirkt eine erfolgreiche Anpassung die Ausführung des Zielknoten. Deshalb kann der minimale Abstand beider Messungen in die Zielfunktion einbezogen werden.

Für eine Optimierung der Annäherung sind jedoch nicht in jedem Fall alle Verzweigungen des ausgeführten Pfades zu betrachten. Das zweite Beispiel der Abbildung 4.18 verdeutlicht ein solches Problem. Mit der zum Testverlauf vorliegenden Information ist nicht entscheidbar, ob eine Veränderung der Verzweigung in Knoten 1 zu einer besseren Annäherung an den Zielknoten führt, weil keine Daten zum Verhalten im danach auszuführenden Knoten 3 vorliegen. Es wird von einer Anpassung in Knoten 1 abgesehen, weil der darauffolgende entscheidende Knoten 3 auf einer geringeren Annäherungsstufe steht als die Stufe des ausgeführten Testverlaufs.

Es bleibt in Experimenten zu prüfen, ob eine Berücksichtigung der Abstandsmessungen derjenigen Verzweigungsbedingungen sinnvoll ist, die zu einer besseren Annäherungsstufe führen. Bei diesem Vorgehen besteht jedoch die Gefahr, daß durch die resultierenden Zielfunktionswerte ein kurzer Weg zum Zielknoten angestrebt wird, der aber wegen logischen Widerspruchs der auszuwertenden Bedingungen nicht ausgeführt werden kann. Eine Abstandsfunktion, welche auf Basis der letzten entscheidenden Verzweigung gebildet wird, besitzt dieses Problem nicht. Deshalb verwendet das System ASTELA vorerst nur diese Form der Bewertung.

Annäherungsstufen im Kontrollflußgraphen

Die Abbildung 4.19 zeigt ein Beispiel für den knotenorientierten Test mit drei Testfällen, die das Teilziel verfehlen. Der Zielknoten ist grau gekennzeichnet. Zu jedem der drei Testfälle läßt sich der erste *entscheidende* Verzweigungsknoten bestimmen, welcher einen unerwünschten Zweig ausführt und damit in einen Teil des Kontrollflußgraphen führt, von dem aus das Teilziel nicht erreichbar ist. Die Abbildung verdeutlicht *unerwünschte* Zweige durch gerichtete Zweige mit fehlendem Endknoten.

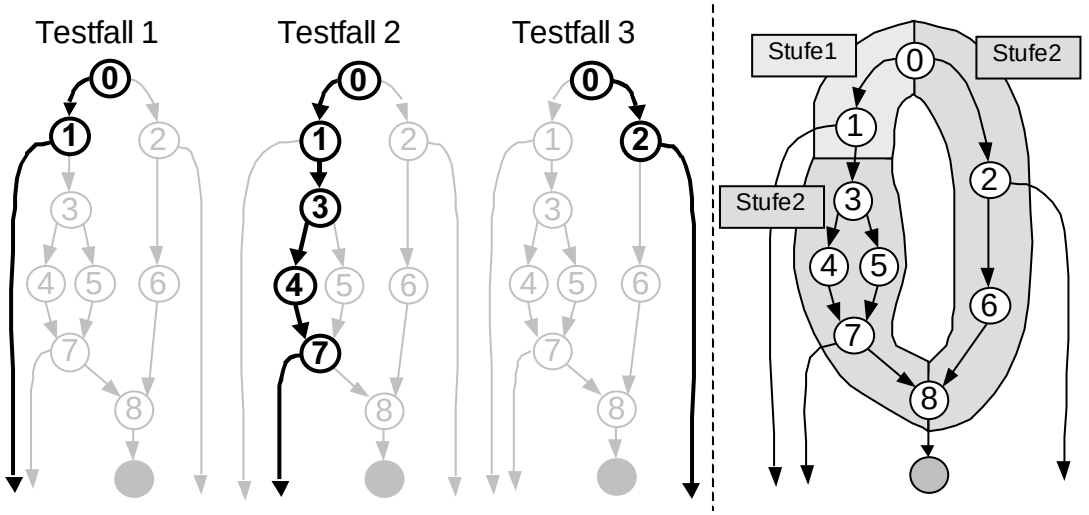


Abbildung 4.19: Beispiel Annäherungsstufen

Der Zielknoten ist von der Verzweigung in den *entscheidenden* Knoten *abhängig*. Diese sogenannte *Kontrollflußabhängigkeit* (Definition Seite 10) läßt sich aus jedem

gerichteten Graphen berechnen. Sie basiert auf der *back-dominance*, einer Beziehung zwischen den Knoten eines gerichteten Graphen, die mit Hilfe eines iterativen Algorithmus bestimmt werden kann (siehe [Schaeff73] Seite 15ff).

Für die Festlegung von Annäherungsstufen, welche einen Vergleich zwischen den einzelnen Testfällen bezüglich der Eignung für das Teilziel zulassen, bedarf es einer Analyse sämtlicher Wege und der Bestimmung aller möglichen Ausführungssequenzen der entscheidenden Verzweigungen. Über die Untersuchung der Wege kann die Ausführungsreihenfolge der entscheidenden Verzweigungsbedingungen aufgedeckt werden. Anhand dieser Ordnung kann eine Aussage zur Qualität von Testdaten getroffen werden, welche in unterschiedlichen Verzweigungen den Zielknoten verfehlen. Dazu werden den Verzweigungen Stufen zugeteilt, welche eine Optimierungsrichtung zum Teilziel vorgeben.

Die Analyse der Verzweigungen in Abbildung 4.19 ergibt, daß in jedem Fall der Knoten 7 nach dem Knoten 1 ausgeführt wird. Bei den dargestellten Testfällen besteht daher ein qualitativer Unterschied, der sich in der Bewertung widerspiegeln muß. Der Knoten 7 erhält aus diesem Grund die höhere Annäherungsstufe 2. Neben dem Weg über die beiden entscheidenden Verzweigungen 1 und 7 existiert im Beispiel noch ein alternativer Pfad über die Knoten 2 und 6. Dieser Weg weist nur eine entscheidende Verzweigung auf. Es gibt daher nur eine Möglichkeit das Teilziel zu verfehlen. Die gleiche Situation gilt für Testfälle, die bis zur Entscheidung in Knoten 7 gelangen. Deswegen wird Knoten 2 die gleiche Annäherungsstufe wie Knoten 7 zugeordnet.

Aus den so festgelegten Annäherungsstufen resultiert eine deutlich bessere Bewertung für die Testfälle 2 und 3, weil dort nur noch eine Entscheidung angepaßt werden muß, um das Testziel zu erreichen. Dagegen ist bei Testfall 1 das Verhalten nach der Anpassung in der darauffolgenden Verzweigung 7 ungewiß und bedarf möglicherweise weiterer Veränderungen der Testdaten.

Eine Bestimmung der Annäherungsstufe durch den Vergleich aller Wege im Kontrollflußgraphen ist im allgemeinen sehr aufwendig, weil je nach Komplexität des Graphen die Anzahl der Wege groß ist. Deswegen wurde für die Implementierung in ASTELA eine Vereinfachung dieser Methode erarbeitet. Für die Erläuterungen wird vorerst angenommen, daß der Zielknoten nicht in einer Schleife liegt. Eine Betrachtung dieses speziellen Falls folgt im Anschluß.

Reduktion des Kontrollflußgraphen

Ziel einer Reduktion des Kontrollflußgraphen ist eine Bestimmung der Annäherungsstufen, ohne dabei alle konstruierbaren Wege des Graphen zu analysieren. Dafür werden die Knoten in Gruppen zusammengefaßt, welche eine grobe Einteilung der Stufen ermöglichen.

Die Betrachtung der Ausführungsreihenfolge muß damit nur noch innerhalb der Gruppen vorgenommen werden. Der Aufwand erheblich reduziert sich erheblich, weil innerhalb einer Gruppe weniger Wege konstruierbar sind. Es können Sonderfälle wie Schleifen analysiert und das Problem durch die Fortsetzung von Reduktionen des

Graphen weiter vereinfacht werden. Es folgt eine Betrachtung der Teilgraphen von Knotengruppen. Die Erläuterungen zur Bildung der Gesamtordnung der Annäherungsstufen aus den Informationen der Teilgraphen schließen den Teilabschnitte ab.

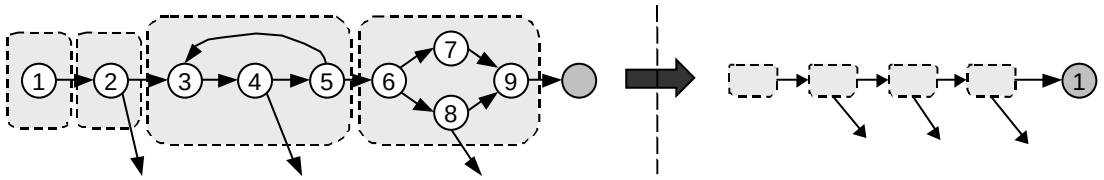


Abbildung 4.20: Reduktion des Kontrollflußgraphen

Die Abbildung 4.20 zeigt die Zusammenfassung von Knotengruppen mit folgenden Eigenschaften:

(1.) Jede Gruppe besitzt genau einen Eingangsknoten.

(2.) Alle Ausgänge einer Gruppe, sofern sie nicht das Teilziel verfehlen, führen zum Eingang der nächsten Gruppe.

(3.) Die Gruppen werden nur aus Knoten gebildet, die Bestandteil mindestens eines Pfades vom Eingangsknoten zum Zielknoten sind.

(4.) Eine Gruppe ist *minimal*, daß heißt, eine weitere Teilung ist nicht möglich ohne die ersten beiden Eigenschaften zu verletzen.

Die so gebildeten Gruppen formen einen Weg vom Eingangsknoten zum Zielknoten. Jede Gruppe repräsentiert einen Schritt in Richtung Zielstellung. Für zwei beliebige Verzweigungen unterschiedlicher Knotengruppen resultiert die Ausführungsreihenfolge aus der Anordnung der Gruppen. Bei der Betrachtung spielen dabei nur die für den Zielknoten *entscheidenden* Verzweigungen eine Rolle.

Numerierung als Mittel für die Bestimmung der Annäherungsstufen

Eine Bestimmung der Annäherungsstufen erfolgt über die Numerierung der entscheidenden Verzweigungen. Die Vergabe der Nummern beginnt bei den Verzweigungen, welche in der Nähe des Zielknotens liegen und endet am Eingangsknoten des Graphen. Für die Knoten mit der kleinsten Nummer wird die höchste Annäherungsstufe vergeben, wohingegen die Knoten mit der größten Nummer, welche somit gegenüber den anderen Verzweigungen vom Zielknoten ‘weit weg’ liegen, die kleinste Annäherungsstufe erhalten.

Nach der Zerlegung des Gesamtgraphen in Knotengruppen, wird die Numerierung in der Gruppe begonnen, welche direkt am Zielknoten liegt. Die Vergabe der Nummern wird dann iterativ in den davor liegenden Knotengruppen fortgesetzt. Wenn die Numerierung am Eingangsknoten des Gesamtgraphen beendet ist, können die Annäherungsstufen in genau umgekehrter Reihenfolge der Nummern zugeordnet werden.

Annäherungsstufen in Knotengruppen

Um für die Festlegung der Annäherungsstufen die Ausführungsreihenfolge der *entscheidenden* Verzweigungen innerhalb einer Gruppe zu klären, ist der Aufbau des

Graphen in der Gruppe zu betrachten. Einen guten Überblick zur Komplexität von Graphen gibt die Arbeit [McCabe76]. Vor allem die Fälle von unstrukturiertem Programmcode lassen sich dort gut nachvollziehen.

Für die Vergabe von Annäherungsstufen innerhalb der Gruppe ist die Beziehung der entscheidenden Verzweigungen hinsichtlich ihrer Ausführungsreihenfolge bei allen möglichen Wegen zu bestimmen. Kann eine Ordnung zwischen den entscheidenden Verzweigungen bestimmt werden, so legt diese die Annäherungsstufen fest.

Bewertungsregel:

Existiert ein Weg zwischen zwei Verzweigungen X und Y, der zuerst durch X und dann durch Y führt, so muß ein Testdatum, das in der Verzweigung Y das Teilziel verfehlt, eine bessere, oder mindestens die gleiche Annäherungsstufe erhalten, wie ein Testdatum, das in der Verzweigung X am Teilziel scheitert.

Eine Betrachtung der beiden Testfälle aus Abbildung 4.21 verdeutlicht den Grundgedanken dieser Regel. Der Testfall 1 scheitert in der Verzweigung ①. Dagegen führt Testfall 2 durch einen Zweig von Knoten ① weiter in Richtung Zielknoten, durchläuft dann aber eine unerwünschte Verzweigung in Knoten ② und geht damit am Teilziel vorbei. Die Zielfunktion muß nach der Bewertungsregel Testfall 2 mit einer besseren Bewertung belohnen, da das Testdatum im Gegensatz zum Testfall 1 in Knoten ① eine erwünschte Verzweigung ausführt und damit erst '*später*' am Teilziel scheitern.

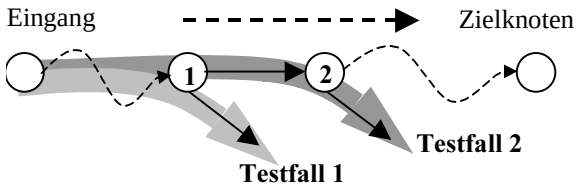


Abbildung 4.21: Grundidee der Bewertungsregel

In einem beliebigen Graphen ist für die Festlegung von Annäherungsstufen, welche die Bewertungsregel einhalten, eine Analyse aller Wege notwendig. Im folgenden wird das Vorgehen in einem beliebigen Graphen festgelegt. Weil die Bestimmung aller Wege jedoch im allgemeinen sehr aufwendig ist, werden im Nachgang einfachere Methoden für zwei häufig auftretende spezielle Graphenarten, die *strukturierte Verzweigungshierarchie* und *Graphen mit Schleifen*, erläutert.

Annäherungsstufen in Gruppen mit beliebigem Graphen

Für zwei *entscheidende* Verzweigungen X und Y einer Gruppe können folgende Ausführungsbeziehungen festgestellt werden:

(1) Die Verzweigungen werden stets in gleicher Reihenfolge ausgeführt.

(2) Die Verzweigungen werden in einer Reihenfolge oder alternativ ausgeführt.

(3) Die Verzweigungen werden in unterschiedlicher Reihenfolge ausgeführt.

Diese Beziehungen faßt eine Ordnungsrelation zwischen allen entscheidenden Verzweigungen einer Gruppe zusammen. Mit Hilfe der Relation kann die Numerierung der

entscheidenden Verzweigungen vorgenommen werden. Dafür werden im ersten Schritt aus der Gesamtmenge der entscheidenden Verzweigungen disjunkte Teilmengen M_i gebildet, welche jeweils nur Verzweigungen enthalten, die durch eine der Beziehungen (1), (2) oder (3) verknüpft sind. Zwischen den zwei Elementen verschiedener Mengen dürfen keine Beziehungen bestehen.

Die Vergabe der Nummern kann innerhalb der Teilmengen M_i unabhängig erfolgen, weil die Verzweigungen unterschiedlicher Mengen in Testfällen stets alternativ ausgeführt werden (keine Ausführungsbeziehung) und die resultierenden Annäherungsstufen der Bewertungsregel entsprechen.

Für die Festlegung von Annäherungsstufen gibt es zwei Vorschläge. Die Verfahren unterscheiden sich in der Vorgehensweise, wenn im Graphen Verzweigungen mit der Beziehung (2) auftreten, weil in diesem Fall eine Ordnung nach der Bewertungsregel nicht explizit vorgeschrieben ist. Nach der ersten Methode erhalten solche Verzweigungen unterschiedliche Annäherungsstufen, da es Wege gibt, bei denen die in Beziehung stehenden Verzweigungen nacheinander ausgeführt werden. Es wird eine *pessimistische Bewertung* vorgenommen, um eine Optimierungsrichtung für diesen ungünstigen Testverlauf angeben zu können. Bei der zweiten Methode bekommen solche Verzweigungen dieselbe Annäherungsstufe, weil es Wege gibt, in welchen jeweils nur eine der Verzweigungen auftaucht (*optimistische Bewertung*). Die aus beiden Verfahren resultierenden Annäherungsstufen verstoßen nicht gegen die Bewertungsregel, die in diesem Fall eine "*bessere oder mindestens die gleiche*" Annäherungsstufe fordert.

Abbildung 4.22 zeigt ein Beispiel für das Bewertungsproblem. Scheitert das Testdatum in Knoten 4, kann entweder eine optimistische oder eine pessimistische Einschätzung der Annäherungsstufe gegeben werden.

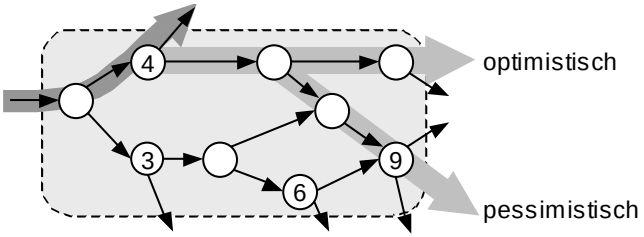


Abbildung 4.22: Optimistische und pessimistische Sichtweise

Bei der *pessimistischen* Bewertung wird angenommen, daß der schlechteste Fall eintreten könnte, in welchem Testdaten in einer Verzweigung X angepaßt werden und dann in Verzweigung Y das Teilziel verfehlen. Deshalb werden auch für den Fall, daß die Verzweigung X und Y durch Beziehung (2) in Relation stehen, unterschiedliche Annäherungsstufen vergeben. Das heißt X erhält eine geringere Annäherungsstufe als alle im Kontrollfluß folgenden entscheidenden Verzweigungen.

Grundgedanke der *optimistischen* Variante ist, daß bei erfolgreicher Generierung von Testdaten, die in Verzweigung X den gewünschten Zweig ausführen, nicht in jedem Fall Testverläufe entstehen, deren Weg durch Y verläuft. Deshalb werden die Annäherungsstufen von Verzweigungen, welche durch Beziehung (2) in Relation stehen, gleichgestellt.

Numerierung für die Vergabe der Annäherungsstufen (pessimistische Variante)

Alle Verzweigungen, welche durch Beziehung (3) verbunden sind und keinen Nachfolger entsprechend der Beziehungen (1) oder (2) haben, erhalten innerhalb der Gruppe die Nummer 1. Die Nummer (k+1) erhalten alle diejenigen Verzweigungen, welche durch die Beziehung (3) verknüpft sind und deren Nachfolger entsprechend der Beziehungen (1) oder (2) die Nummer k oder eine kleinere besitzen.

Numerierung für die Vergabe der Annäherungsstufen (optimistische Variante)

Alle Verzweigungen, welche durch die Beziehungen (3) oder (2) verbunden sind und keinen Nachfolger entsprechend der Beziehung (1) haben, erhalten innerhalb der Gruppe die Nummer 1. Die Nummer (k+1) erhalten alle diejenigen Verzweigungen, welche durch die Beziehungen (3) oder (2) verknüpft sind und deren Nachfolger entsprechend der Beziehung (1) die Nummer k oder eine kleinere trägt.

Die Numerierung der in der Abbildung 4.22 dargestellten Knoten wird wie folgt vorgenommen (Beziehung (2) besteht zwischen den Knoten 3 und 6 sowie den Knoten 4 und 9):

- *Bei der pessimistischen Version erhält Knoten 3 die Nummer 3, die Knoten 4 und 6 die Nummer 2 sowie der Knoten 9 die Nummer 1.*
- *Die optimistische Methode vergibt nur zwei Nummern. Die Nummer 1 für die Knoten 4 und 9 sowie Nummer 2 für die Knoten 3 und 6.*

Die Analyse aller Wege und die darauf basierende Aufstellung der Beziehungen (1), (2) und (3) ist je nach Komplexität von großen Verzweigungsgraphen sehr aufwendig. Für Knotengruppen, welche bei strukturierter Programmierung entstehen, wurde eine einfachere Methode der Numerierung entwickelt, welche im folgenden vorgestellt werden soll. Die Beziehung (3) gilt für entscheidende Verzweigungen innerhalb von Schleifenstrukturen. Für Graphen, welche Zyklen enthalten, wurde deshalb ein weiteres vereinfachtes Verfahren entwickelt.

Annäherungsstufen in strukturierten Verzweigungshierarchien

Die strukturierte Verzweigungshierarchie stellt neben Schleifen bei systematischer Programmierung des Testobjekts die wohl häufigste Form ein Knotengruppierung dar. Hier läßt sich, wie im Anschluß gezeigt wird, sehr leicht eine Ordnung der entscheidenden Verzweigungen finden und die Numerierung festlegen.

In strukturierten Verzweigungshierarchien lassen sich sehr einfach disjunkte Mengen von Wegen identifizieren, weil Verzweigungen den Kontrollfluß aufteilen. Eine Knotengruppe kann dabei mehrere sequentiell angeordnete oder geschachtelte Verzweigungen umfassen. Diese Regel gilt nur für den Fall, daß im Testobjekt keine expliziten Sprunganweisungen angewendet werden.

Liegt eine strukturierte Verzweigungshierarchie vor, kann die hierarchische Ordnung bestimmt werden. Die verschiedenen Wege einer strukturierten Verzweigung können

für die Festlegung der Annäherungsstufen getrennt betrachtet werden und so das Problem der Festlegung der Annäherungsstufen im Graphen der gesamten Gruppe auf Teilgraphen reduziert werden. Diese Zerlegung entspricht der Bildung der Teilmengen M_i in beliebig aufgebaute Knotengruppen.

Abbildung 4.23 zeigt eine Knotengruppe in Form einer Verzweigungshierarchie.

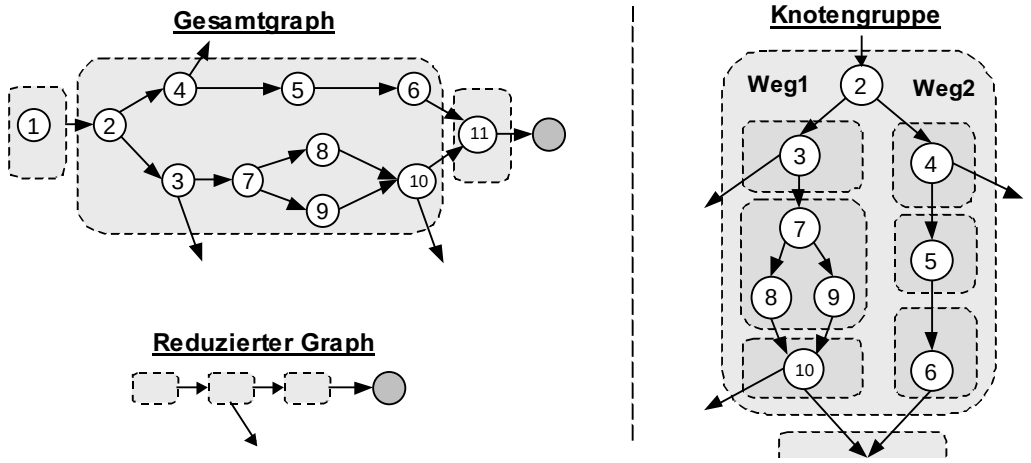


Abbildung 4.23: Strukturierte Verzweigungshierarchie

Für die Bestimmung der Annäherungsstufen innerhalb eines Weges der Verzweigung wird genau wie bei der Reduktion des Gesamtgraphen (vergleiche Abbildung 4.20 und Text) eine Zerlegung in Abschnitte vorgenommen. Die Stufung zwischen den Abschnitten entspricht der Ausführungsreihenfolge innerhalb des Weges. Für die gebildeten Abschnitte des Weges kann durch eine weitere Unterteilung in Teilgraphen rekursiv vorgegangen werden.

Im Beispiel aus Abbildung 4.23 liegen nach der Zerlegung nur noch Untergruppen mit einer oder keiner *entscheidenden* Verzweigung vor. Damit stehen die Annäherungsstufen fest. Die Knoten 10 und 4 erhalten demnach Stufe 1 sowie der Knoten 3 die Stufe 2.

Numerierung zur Vergabe der Annäherungsstufen:

Die Numerierung wird für die disjunkten Mengen von Wegen getrennt vorgenommen. Dafür erfolgt, analog zum Vorgehen beim Gesamtgraphen, eine Zerlegung des Teilgraphen in Abschnitte. Die Numerierung wird bei dem Abschnitt begonnen, der am nächsten zum Ausgang der Knotengruppe in Richtung Zielknoten liegt. Die Numerierung innerhalb der so gebildeten Untergruppen wird rekursiv fortgesetzt, wenn entscheidende Verzweigungen enthalten sind. Ansonsten entfällt der Abschnitt für die weiteren Betrachtungen.

Annäherungsstufen für eine Knotengruppe mit Schleifen

Für die Vereinfachung des Problems der Festlegung der Annäherungsstufen in Graphen mit Zyklen wird eine Zusammenfassung aller Knoten des Schleifenkörpers vorgenommen. Durch eine solche Zusammenfassung wird der Teilgraph der Gruppe in einen zyklenlosen Verzweigungsgraphen umgewandelt. In diesem schleifenlosen Graphen

können die Annäherungsstufen mit den bereits beschriebenen Verfahren bestimmt werden. Entscheidende Verzweigungen, welche innerhalb einer Schleife liegen, erhalten die gleiche Annäherungsstufe, da für die Knoten im Schleifenkörper die Ausführungsbeziehung (3) gilt.

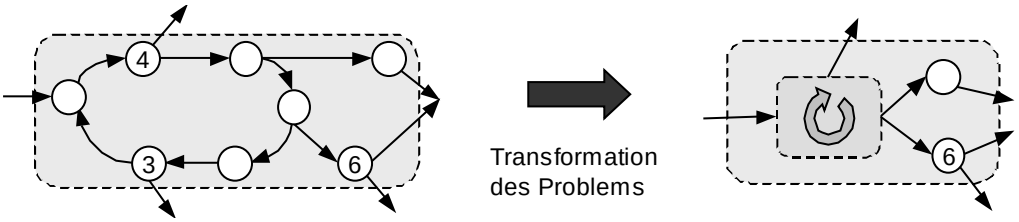


Abbildung 4.24: Schleifen in einer Knotengruppe

In dem geänderten Graphen können die Annäherungsstufen einfacher bestimmt werden. Falls *entscheidende* Verzweigungen im Schleifenkörper vorliegen, wird im transformierten Graphen eine Annäherungsstufe für den Ersatzknoten vergeben und die aus den Festlegungen resultierende Stufe allen *entscheidenden* Verzweigungen der Schleife zugewiesen.

Die Abbildung 4.24 verdeutlicht diesen Lösungsansatz. Knoten 3 und 4 sind Bestandteil des Schleifenkörpers und erhalten nach Bestimmung der Annäherungsstufen im rechts dargestellten zyklenfreien Graphen die gleiche Stufe.

Vergabe der Annäherungsstufen:

Der Graph einer Gruppe mit Schleife wird so umgewandelt, daß ein azyklischer Graph entsteht. Die Transformation faßt die Knoten des Schleifekörpers in einem Ersatzknoten zusammen. In dem gebildeten azyklischen Graphen können die Annäherungsstufen, wie bereits beschrieben, berechnet werden. Alle in der Schleife enthaltenen entscheidenden Verzweigungen bekommen die Stufe, welche dem Ersatzknoten zugeordnet wird.

Sonderfall Zielknoten in einer Schleife

Die vorhergehenden Teilabschnitte beschreiben die Festlegung der Annäherungsstufen in Knotengruppen des Kontrollflußgraphen. Dabei wurde der Fall, daß der Zielknoten Bestandteil einer Schleife ist, noch nicht betrachtet. Mit Hilfe einer einfachen Modifikation des Kontrollflußgraphen läßt sich der Zielknoten für diese Betrachtungen aus der Schleife abspalten, um damit die Bestimmung der Annäherungsstufen in das zuvor beschriebene Problem (Seite 58) "*Annäherungsstufen in Gruppen mit beliebigem Graphen*" zu transformieren.

Wenn sich der Zielknoten in einer Schleife befindet, kann in einem Testlauf durch Schleifeniterationen eine wiederholte ‚Annäherung' an das Ziel auftreten. Wie Abbildung 4.25 linke Seite verdeutlicht, läßt sich die Sicht auf das Problem wandeln, indem der Zielknoten für die Analyse der Annäherungsstufen aus der Schleife herausgelöst wird. Bei dieser Transformation werden die vom Zielknoten wegführenden Zweige für die Betrachtung entfernt. In dem so modifizierten Graphen ist der Zielknoten nicht mehr Bestandteil eines Zyklus. Der übrig gebliebene Teil des Schleifenkörpers bildet

eine vorgelagerte Knotengruppe. Eine Bestimmung der Annäherungsstufen kann in dem geänderten Graphen so erfolgen, wie in den vorangegangenen Ausführungen zu den Knotengruppen beschrieben wurde.

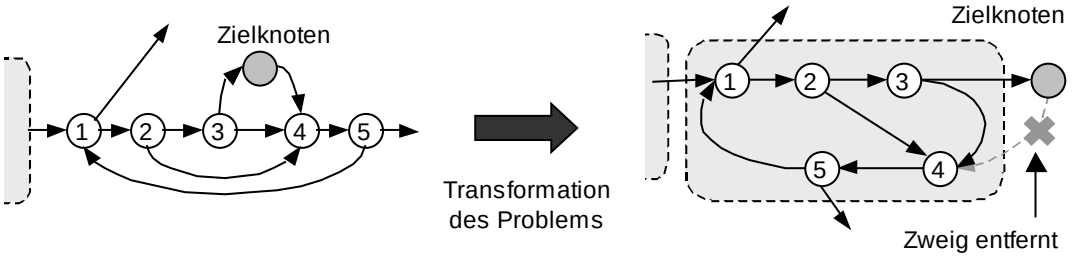


Abbildung 4.25: Zielknoten in einer Schleife

Durch diese Betrachtungsweise des Problems wird deutlich, daß alle Testfälle, welche den Zielknoten verfehlen, nur einen Teil des Schleifenkörpers wiederholen. Dieser Teil des Schleifenkörpers wird bis zur Abweichung vom Ziel wiederholt. Durch die Transformation wird der vorgelagerte Teil des Schleifenkörpers in einer Knotengruppe zusammengefaßt und für die Bestimmung der Annäherungsstufen wie die anderen Abschnitte behandelt.

Experimente müssen zeigen, ob die bessere Bewertung von Testfällen, welche Knoten mit direkt zum Zielknoten verweisenden Zweigen ausführen (*im Beispiel der Knoten 3*), für die Optimierung hilfreich sein kann. Es sollte geprüft werden, ob sich die Abwandlung der Bewertung in diesem speziellen Fall eignet, um die Optimierung zu beschleunigen oder ob es genügt, die unerwünschten Verzweigungen aus der Schleife mit schlechteren Bewertungen zu ‚bestrafen'.

Annäherungsstufen im Gesamtkontext des Kontrollflußgraphen

Für die Vereinfachung der Methode zur Festlegung der Annäherungsstufen wurde eine Zerlegung der Gesamtgraphen in Knotengruppen vorgeschlagen. Innerhalb der Knotengruppen wird das Problem gegebenenfalls weiter zerlegt (zum Beispiel in Verzweigungshierarchien).

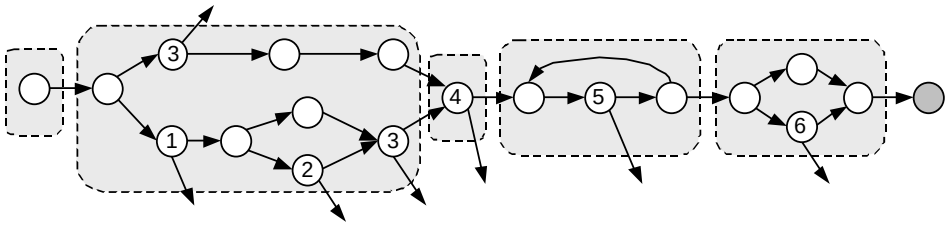


Abbildung 4.26: Annäherungsstufen der entscheidenden Verzweigungen

Die Vergabe der Stufen erfolgt genau umgekehrt zur Numerierung der Knoten. Den entscheidenden Verzweigungen mit den höchsten Nummern werden die geringsten Annäherungsstufen zugeteilt. In der Abbildung wird das Ergebnis der Verteilung von Annäherungsstufen an einem Beispiel verdeutlicht. Die Annäherungsstufen sind in den Verzweigungen vermerkt.

4.3.2 Zielfunktion für Zweigüberdeckung

Die Zweigüberdeckung gehört, wie bei der Klassifikation in Abschnitt 2.3 erläutert, zu den knotenorientierten Verfahren. In erster Linie hat die Zielfunktion deshalb einen Testverlauf daraufhin zu bewerten, wie weit eine Annäherung an den Knoten besteht, von dem der Zielzweig ausgeht. Erst wenn der Knoten erreicht wird, muß mit Hilfe einer Abstandsfunktion die Annäherung an den Zielzweig ausgedrückt werden.

Die folgende Abbildung verdeutlicht die zwei möglichen Bewertungssituationen für den Zweigüberdeckungstest. Teilziel ist in diesem Beispiel der Zweig 7→9. Während der erste Testfall den Startknoten des Zielzweigs (Knoten 7) nicht erreicht, durchläuft der zweite Testfall diesen Knoten und führt dann jedoch den falschen Zweig aus.

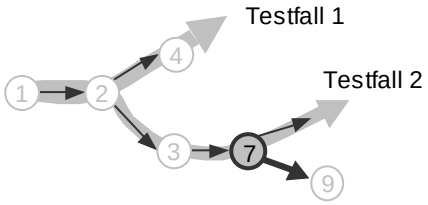


Abbildung 4.27: Bewertungssituationen der Zweigüberdeckung

Die Zielfunktion wird wie folgt definiert:

Zielfunktion(Testverlauf) : = (höchste vergebene Annäherungsstufe+1) – Annäherung´(Testverlauf)

Für den Fall, daß der Verzweigungsknoten nicht erreicht wird:

Annäherung´(Testverlauf) : =
1 + *Annäherung zum Verzweigungsknoten siehe Formel* (4.3)

Für den Fall, daß der Verzweigungsknoten erreicht wird:

Annäherung´(Testverlauf) : =
Abstand zum erwünschten Zweig (siehe Abschnitt 4.4)

4.3.3 Zielfunktion für Bedingungsüberdeckung

Bei der Bedingungsüberdeckung handelt es sich um ein knotenorientiertes Verfahren. Das bedeutet, daß eine Berechnung der Zielfunktion entsprechend der Festlegungen in Teilabschnitt 4.3.1 erfolgen muß, damit die Annäherung an einen Verzweigungsknoten beschrieben werden kann.

Für ein Teilziel der Bedingungsüberdeckung genügt das Erreichen eines bestimmten Verzweigungsknotens nicht. Es werden zusätzliche Forderungen an die Auswertung der Bedingungen in diesem Knoten gestellt. Die Erfüllung dieser Forderungen muß in die Bewertung einbezogen werden, so daß eine Optimierung auch für diesen Teil der Zielstellung möglich ist.

Zielfunktion(Testverlauf) : = (höchste vergebene Annäherungsstufe+1) – Annäherung´(Testverlauf)

Für den Fall, daß der Verzweigungsknoten nicht erreicht wird:

Annäherung´(Testverlauf): =
1 + Annäherung zum Verzweigungsknoten siehe Formel (4.3)

Für den Fall, daß der Verzweigungsknoten erreicht wird:

Annäherung´(Testverlauf): = *Bewertungsfunktion der Verzweigungsbedingung*

Die *Bewertungsfunktion* der Verzweigungsbedingung richtet sich nach der Form der Bedingungsüberdeckung. Es folgt eine Betrachtung der Funktionen für die verschiedenen Überdeckungskriterien.

Atomare Bedingungsüberdeckung

Zielstellung bei der atomaren Bedingungsüberdeckung ist die Bestimmung von Eingaben für jeden Wahrheitswert der atomaren boolschen Terme einer Verzweigungsbedingung. Moderne Compiler (z.B. *AnsiC* und *Pascal*) optimieren die Berechnung der Verzweigungsbedingungen. Es wird nur der Teil der Bedingung ausgewertet, welcher zur Bestimmung der Entscheidung notwendig ist. Deshalb liegen nur die Werte einiger atomarer Bedingungen bei jedem Testverlauf vor. Die Voraussetzungen, bei denen eine Auswertung der Zielbedingung erfolgt, müssen in die Bewertung der generierten Testdaten einfließen.

Folgende Ausführungsregelungen gelten für *n*-stellige logische Operatoren:

(1) der *i*-te Term einer UND-Verknüpfung wird nur ausgewertet, wenn die ersten (*i*-1) Terme *wahr* ergeben.

(2) der *i*-te Term einer ODER-Verknüpfung wird nur ausgewertet, wenn die ersten (*i*-1) Terme *falsch* ergeben.

Für die Bewertung eines Testdatums ist zu prüfen, in welchem Teil der Bedingung sich der atomare Term des Teilziels befindet. In die Bewertungsfunktion müssen alle im Testobjekt vor dem atomaren Term auszuwertenden Operanden einfließen.

Beispiel:

Bed. = (A1 oder A2) und (B1 oder B2 oder (C1 und C2)) und A3
Der atomare Term "C2" wird in diesem Term nur ausgewertet, wenn A1 oder A2 wahr ergibt, desweiteren B1 und B2 das Ergebnis falsch sowie der Ausdruck C1 den Wert wahr liefert. Dies sind die Vorbedingungen für die Auswertung von C2.

Es sei Term A der atomare Term des Teilziels und W_A das angestrebte Ergebnis bei der Ausführung des Testobjekts. Für die Bewertung eines Testdatums sind folgende Formeln anzuwenden:

Bewertung der Vorbedingungen für die Ausführung des Terms A (rekursiv)

Wenn A∈T, das heißt, Term A ist Teilbedingung des zu bewertenden Terms T, dann ∃ k *mit* A∈T_k: (*je eine Formel für T als Und bzw. Oder-Verknüpfung*)

$$Bewertung(Und(T_1,\ldots,T_n)) = \frac{\sum_{i=1}^{k-1} Abstand(T_i, wahr) + Bewertung(T_k)}{k} \quad (4.4)$$

$$Bewertung(Oder(T_1,\ldots,T_n)) = \frac{\left(\sum_{i=1}^{k-1} Abstand(T_i, falsch)\right) + Bewertung(T_k)}{k}$$

Bewertung für den atomaren Term A selbst:

$$Bewertung(A) = Abstand(A, W_A) \quad \text{(siehe Abschnitt 4.4)}$$

Die Funktion *Abstand(Term A, Wert W_A)* entspricht den Festlegungen einer Abstandsfunktion für die Auswertung der Bedingungen einer Verzweigung. Diese Funktion wird in Abschnitt 4.4 genauer betrachtet. Sie liefert je nach Testverlauf ein Ergebnis zwischen 0 und 1, welches die Annäherung des ausgeführten Tests an den boolschen Wert W_A in der Bedingung *A* ausdrückt. Das schlechteste Funktionsergebnis '0' erhält ein Term, wenn er nicht ausgewertet wird.

Die folgende Abbildung zeigt ein Beispiel für ein Teilziel der atomaren Bedingungsüberdeckung in einer zusammengesetzten Verzweigungsbedingung und die Bewertung eines Testdatums, das einen Teil der Verzweigungsbedingung auswertet.

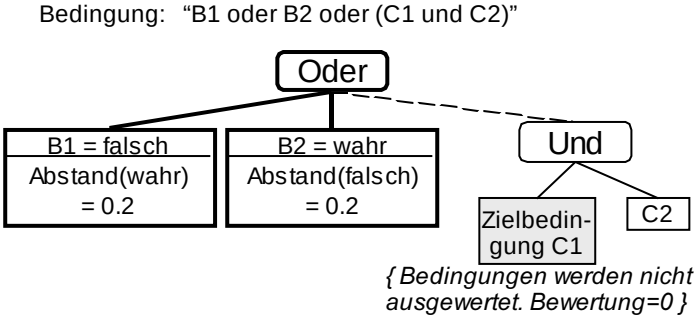


Abbildung 4.28: Auswertungsbeispiel atomare Bedingungsüberdeckung

Teilziel (grau) ist die atomare Bedingung C1, welche mit 'wahr' ausgewertet werden soll. Im dargestellten Beispiel liegen nur die Wahrheitswerte der Teilbedingungen B1 und B2 vor. Die restlichen Elemente der Bedingung wurde bei diesem Testfall nicht ausgewertet. Die Bewertung lautet:

Bewertung =(Abstand(B1,falsch)+Abstand(B2,falsch)+Bewertung(Und(c1,c2))÷3

Bewertung = (1 + 0.1 + 0) ÷ 3

Mehrfach-Bedingungsüberdeckung

Bei der Mehrfach-Bedingungsüberdeckung C_3 besteht ein Teilziel aus einer bestimmten Kombination von Wahrheitswerten für die atomaren Bedingungen. Aufgrund der

Optimierungen durch den Compiler bei den meisten modernen Programmiersprachen, beispielsweise *AnsiC* und *Pascal*, wird häufig nur ein Teil der Bedingung ausgewertet, so daß nicht alle Kombinationen getestet werden können. Für diesen Fall beschreiben [Riedm97] und [Jasper94] eine modifizierte Form der *minimalen Mehrfach-Bedingungsüberdeckung* (*MCDC - modified condition decision coverage)*.

Ein *Teilziel* (*T, k*) wird folgendermaßen beschrieben:

Term T ist der betrachtete Teil der Verzweigungsbedingung

- *wenn der Term T eine n-stellige UND-Operation ist:*

  Teilziel (T, k) ist die Auswertung der ersten k-Operatoren des Terms mit *wahr* und des (k+1)-ten Operators mit *falsch*.

- *wenn der Term eine n-stellige ODER-Operation ist:*

  Teilziel (T, k) ist die Auswertung der ersten k-Operatoren des Terms mit *falsch* und des (k+1)-ten Operators mit *wahr*.

Damit ein bestimmter Teil der Verzweigungsbedingung in einem optimierten Testobjekt ausgewertet wird, müssen Vorbedingungen für die anderen Terme der Gesamtbedingung gelten. Es läßt sich eine Bewertung der Testdaten ähnlich der atomaren Bedingungsüberdeckung definieren.

Statt des atomaren Terms A muß in der Berechnungsvorschrift '*Term T* ' verwendet werden. Die nachstehenden Formeln beschreiben die Bewertungsfunktion für den Fall, daß der Term T ausgewertet wird.

Sei (Term T , k) das Teilziel der Optimierung. Die Bewertungsvorschrift der Annäherung an das Teilziel im Kontext der gesamten Verzweigungsbedingung wird nach Formel (4.4)*. rekursiv definiert, wobei statt des Terms A nun der Term T verwendet wird.*

(A) Term T ist UND-Operation:

$$Bewertung(Term\ T) = \frac{\left(\sum_{i=1}^{k} Abstand(Op_i, wahr)\right) + Abstand(Op_{k+1}, falsch)}{k+1}$$

(B) Term T ist ODER-Operation:

$$Bewertung(Term\ T) = \frac{\left(\sum_{i=1}^{k} Abstand(Op_i, falsch)\right) + Abstand(Op_{k+1}, wahr)}{k+1}$$

Wird ein Operand des Terms nicht ausgewertet, so ist für die Abstandsfunktion der Wert Null einzusetzen. Das folgende Beispiel zeigt die Ausdruckshierarchie einer Verzweigungsbedingung. Ein Teilziel der Mehrfach-Bedingungsüberdeckung ist grau gekennzeichnet. In die Bewertung werden alle in der Abbildung hervorgehobenen Teile der Bedingung einbezogen. Die Terme A2 und C3 werden nicht für die Bewertung benutzt.

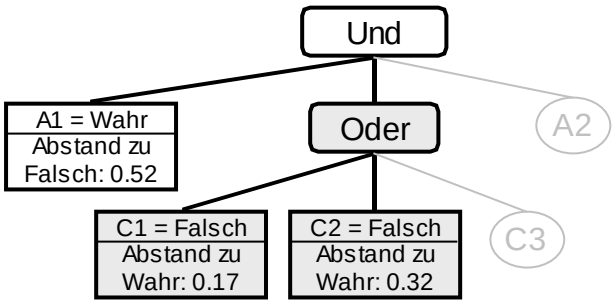


Abbildung 4.29: Auswertungsbeispiel Mehrfach-Bedingungsüberdeckung

Teilziel: Term T = "C1 oder C2 oder C3" ; der erste Operand (k=1) soll mit falsch und der Operand danach mit wahr ausgewertet werden.

$$Bewertung(Gesamt) := \frac{Abstand(A1, wahr) + Bewertung(T)}{2}$$

$$Bewertung(T) := \frac{Abstand(C1, falsch) + Abstand(C2, wahr)}{2}$$

Bewertung(Gesamt) = (1+ (1+0.32) ÷ 2) ÷ 2 = 0.83

4.3.4 Zielfunktion für pfadorientierte Testverfahren

In Teilabschnitt 4.2.3 wurden zwei grundlegende Methoden zur Bewertung von Testdaten für pfadorientierte Teilziele erläutert. Die erste Methode definiert die Zielfunktion über den identischen Anfangspfad:

Zielfunktion(Testverlauf) : = Zielpfadlänge – Annäherung(Testverlauf, Zielpfad)
Annäherung(Testverlauf, Zielpfad) : =

Abstand zum erwünschten Zweig +
Länge(*identischer Anfangspfad*) (4.5)

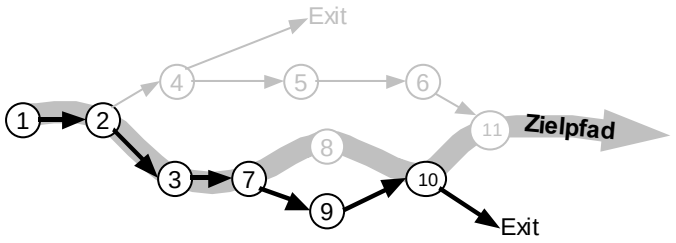


Abbildung 4.30: Beispiel zur Abweichung des Testverlaufs vom Zielpfad

Abbildung 4.30 zeigt einen Kontrollflußgraphen mit Testverlauf «1, 2, 3, 7, 9, 10» und den Zielpfad «1, 2 ,3 7, 8, 10, 11». Die Abweichung des Testverlaufs vom Zielpfad nach dem Knoten 7 ergibt die folgende Bewertungsformel:

Annäherung(Testverlauf) = *Abstand zum Zweig*(7→8) + *Länge*(«1, 2, 3, 7»)

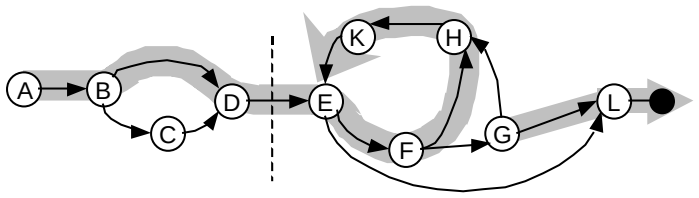
Für den alternativen Bewertungsvorschlag wird zunächst die Beschreibung des Zielpfades durch Wegstücke in den Abschnitten des Kontrollflußgraphen erläutert. Die Spezifikation des Pfades erlaubt zudem die Festlegung von Teilzielen für strukturierte Pfadtests, bei denen nur ein Teil des Weges durch Schleifeniterationen vorgeschrieben ist. Zu dieser Form der Festlegung des Zielpfades wird die Formel (4.5) angepaßt, so

daß die Zielfunktion auch für Pfadtestkriterien genutzt werden kann, welche Wegeklassen als Teilziele definieren.

Pfadbeschreibung durch Wegstücke

Umfaßt die Zielstellung nicht nur einen bestimmten Pfad, sondern eine Äquivalenzklasse von Pfaden, wie bei den Kriterien des strukturierten Pfadtests ("Strukturierte Pfadüberdeckung" Seite 14), muß die Berechnung geringfügig angepaßt werden. Die Zielfunktion wird auf Basis der festgelegten Teile des Gesamtpfades definiert und ist so gestaltet, daß die variablen Pfadstücke einer Äquivalenzklasse, wie zum Beispiel nicht betrachtete Schleifeniterationen, keinen Einfluß auf den Funktionswert haben.

Ein Zielpfad besteht aus einer Abfolge von Wegstücken, welche in einer festgelegten Reihenfolge ausgeführt werden müssen. Die folgende Abbildung zeigt einen Kontrollflußgraphen, zu dem auf der rechten Seite beispielhaft ein Zielpfad angegeben ist.



Zielpfad:
Wegstück1: 'ABD'
Wegstück2: 'EFHK'
Wegstück3: 'EFHK'
beliebige Schleifenwiederhlg.
Wegstück4: 'GL' [Ende]

Abbildung 4.31: Zielpfad definiert durch Wegstücke

Im dargestellten Zielpfad der Abbildung 4.31 wird das Wegstück bis zum Schleifeneintritt, der Weg für die ersten beiden Schleifeniterationen und das Wegstück nach dem Verlassen der Schleife festgelegt. Die Bestimmung dieser Wegstücke bei der Identifikation der Teilziele für den strukturierten Pfadtest (siehe Teilabschnitt 4.2.1).

Es folgt die Definition der Bewertungsmethode "identisches Anfangswegstück" für den Fall, daß der Zielpfad über Wegstücke festgelegt wird:

- *Seien W_1 , ... , W_n die festgelegten Wegstücke des Zielpfades in den Abschnitten des Kontrollflußgraphen.*
- *O.b.d.A.: Der Vergleich des Testverlaufs mit dem geforderten Zielpfad ergebe eine Übereinstimmung der Wegstücke W_1 bis W_k (im Wegstück W_{k+1} erfolgt eine Abweichung). Die Zielfunktion berechnet sich damit wie folgt* :

  Zielfunktion(Testpfad) : = Zielpfadlänge – Annäherung(Testpfad)

  $$\text{Annäherung(Testpfad)} : = \left(\sum_{i=1}^{K} Länge(W_i) \right) + Annäherung\ (W_{K+1}, Testpfad) \qquad (4.6)$$

  Annäherung(W_{k+1} , Testpfad) : = *Abstand zum erwünschten Zweig* +
  Länge des identischen Wegstücks im Abschnitt

  Der erwünschte Zweig und die Länge des identischen Teils richten sich nach der festgestellten Abweichung vom Wegstück W_{k+1} im zugehörigen Abschnitt.

Schleifen mit mehreren Austrittswegen

Falls die Schleife mehrere Austrittswege besitzt, wie beispielsweise die Abbildung 4.31 zeigt, muß eine Annäherung an den gewünschten Schleifenaustritt bestimmt werden. Wird der falsche Weg aus der Schleife gewählt, so muß das zugehörige Testdatum eine schlechtere Bewertung erhalten.

Ein Austrittsweg führt von einem Verzweigungsknoten, der Bestandteil des Schleifenkörpers ist, zum Ausgang der Knotengruppe. Die Schleife in Abbildung 4.31 hat genau zwei solcher Wege ("EL" und "GL"). Damit ein bestimmter Austrittsweg durchlaufen wird, muß der Startknoten dieses Weges, welcher als einziger Knoten noch im Schleifenkörper liegt, mit dem gewünschten Zweig ausgeführt werden. Um Testfälle bewerten zu können, die nicht den gewünschten Austrittsweg nutzen, wird eine eigenständige Abstandsfunktion gebildet, welche die Annäherung an den Startknoten dieses Wegs ausdrückt.

Mit der Bestimmung einer Abstandsfunktion für die Ausführung eines bestimmten Knotens in einer Schleife beschäftigt sich der Teilabschnitt knotenorientierte Teilziele (siehe "Sonderfall Zielknoten in einer Schleife"; Seite 62). Wird der Startknoten des Austrittsweges erreicht, kann die Annäherung an den Austrittsweg mit der Formel (4.5) bewertet werden.

Alternative Bewertung für pfadorientierte Testverfahren

Die Idee der Zerlegung des Pfades in Teilabschnitte läßt eine weitere Form der Beschreibung einer Annäherung an einen Zielpfad zu. Dazu werden Zielpfad und Testverlauf in den Abschnitten des Kontrollflußgraphen unabhängig voneinander verglichen. Der Gesamtwert der Zielfunktion ergibt sich aus der Summe der Abweichungen in den durchlaufenen Abschnitten.

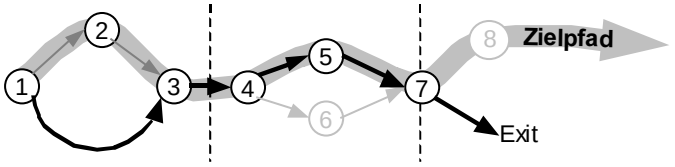


Abbildung 4.32: Abschnitte des Zielpfades

Die Abbildung zeigt einen Testverlauf über die Knoten 1, 3, 4, 5, 7 und einen Zielpfad, von dem bereits im ersten Abschnitt abgewichen wird. Mit der Formel (4.6) erhält das zugehörige Testdatum eine sehr geringe Bewertung, obwohl der Testverlauf im zweiten Abschnitt wieder mit dem Zielpfad übereinstimmt. Die alternative Bewertung liefert für solche Testverläufe einen höheren Zielfunktionswert.

- *Seien W_1,.. W_N die festgelegten Wegstücke des Zielpfades in den Abschnitten des Kontrollflußgraphen.*

Zielfunktion(Testverlauf) : = Zielpfadlänge – *Annäherung*(Testverlauf)

$$\text{Annäherung}(\text{Testverlauf}) := \left(\sum_{i=1}^{N} Annäherung(W_i, Testverlauf) \right) \qquad (4.7)$$

1. Fall: Der Testverlauf ist im Wegstück W_i identisch:

Annäherung(W_i, Testverlauf) : = Länge (W_i)

2. Fall: Der Testverlauf weicht vom Wegstück W_i ab:

Annäherung(W_i, Testverlauf) : = *Abstand zum erwünschten Zweig* +
Länge des identischen Wegstücks W_i

4.3.5 Zielfunktion für knoten-wegorientierte Testverfahren

Die Teilziele der knoten-wegorientierten Verfahren bestehen aus einem zu erreichenden Zielknoten und einem dort beginnenden Wegstück, welches auszuführen ist. Die Zielfunktion ist damit eine Verknüpfung der Funktionen aus den vorangegangenen Abschnitten 4.3.1 (knotenorientierte Tests) und 4.3.4 (pfadorientierte Verfahren).

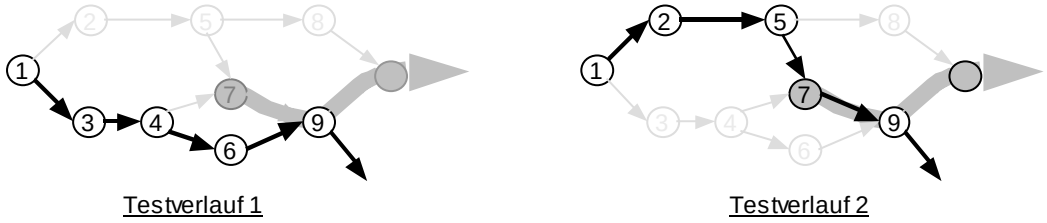


Abbildung 4.33: Zwei Testverläufe mit knoten-wegorientiertem Testziel

Die Zielfunktion für knotenorientierte Tests wird für die Bewertung von Testdaten eingesetzt, die den Zielknoten nicht erreichen (siehe Testfall 1 in der Abbildung 4.33). Durchläuft ein Testfall hingegen den gewünschten Knoten, wie Testfall 2 in der Abbildung 4.33, so ist die Eignung des zugehörigen Testdatums für die Ausführung des nachfolgenden Wegstücks funktional auszudrücken. Hierfür eignet sich die Bewertungsmethode der pfadorientierten Testverfahren. Die Zielfunktion lautet damit:

Zielfunktion(Testverlauf): = ω + max. Annäherungsstufe zum Zielknoten –
Annäherung(Testverlauf)

(ω *ist die Länge des nachfolgenden Wegstückes*)

Für den Fall, daß der Knoten nicht erreicht wird:

Annäherung(Testverlauf) : = ω + *Annäherung(Zielknoten)* *siehe Formel (4.3)*

Für den Fall, daß der Knoten erreicht wird:

Annäherung(Testverlauf) : = *Annäherung(Wegstück)* *siehe Formel (4.5)*

4.3.6 Zielfunktion für knoten-knotenorientierte Testverfahren

Die Bewertung der Eignung von Testdaten für die Teilziele der knoten-knotenorientierten Testverfahren muß eine Annäherung an die Zielstellung der Ausführung von zwei Zielknoten als Zielfunktionswert ausdrücken.

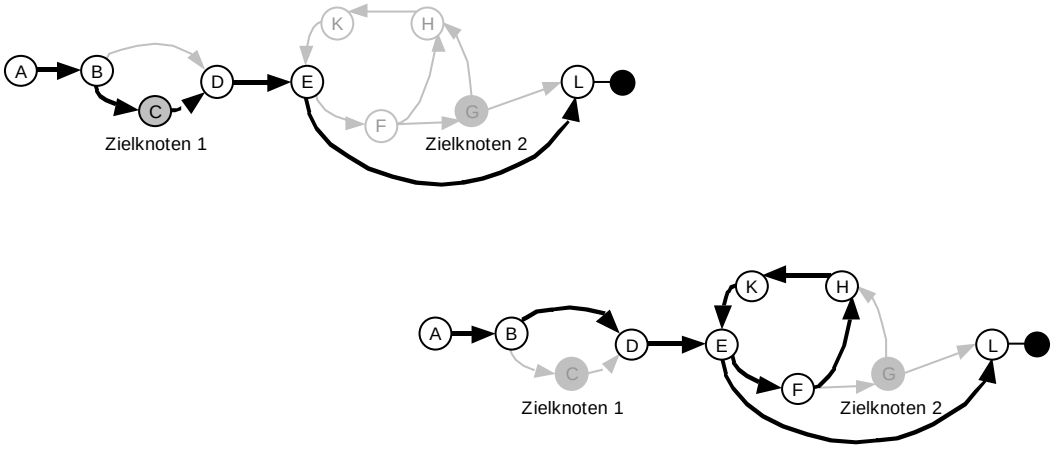


Abbildung 4.34: Beispielhafte Testfälle knoten-knotenorientierter Tests

Für Testverläufe, welche den ersten Zielknoten nicht erreichen (siehe ersten Testfall in Abbildung 4.34), kann zur Bewertung die Zielfunktion der *knotenorientierten* Verfahren aus Abschnitt 4.3.1 genutzt werden. Andernfalls wird die Annäherung eines Testverlaufs an den zweiten geforderten Knoten bestimmt. Die Zielfunktion gestaltet sich dabei wiederum *knotenorientiert*, mit dem Unterschied, daß Annäherungsstufen zwischen dem ersten und zweiten Knoten für die Berechnung verwendet werden.

Teilziel ist ein Weg zwischen den Knoten K_1 und K_2. Die Zielfunktion lautet:

$\underline{Zielfunktion}$(Testverlauf) : = ($\mu + \omega$) – $\underline{Annäherung}$(Testverlauf)
$\mu \mathrel{\hat{=}}$ Anzahl der Annäherungsstufen zwischen Eingang und K_1
$\omega \mathrel{\hat{=}}$ Anzahl der Annäherungsstufen zwischen K_1 *und* K_2

Falls der Knoten K_1 nicht erreicht wird:

$\underline{Annäherung}$(Testverlauf) : = ω + $\underline{Annäherung}$(Zielknoten K_1) siehe Formel (4.3)

Falls der Knoten K_1 erreicht wird:

$\underline{Annäherung}$(Testverlauf) : = $\underline{Annäherung}$(Zielknoten K_2) siehe Formel (4.3)

Für die letzte Formel sind die *entscheidenden* Verzweigungen und Annäherungsstufen zwischen Knoten K_1 und K_2 zu bestimmen. In dem Bereich zwischen den beiden Knoten kann es für bestimmte Strukturtestverfahren weitere Einschränkungen der erlaubten Wege geben. Diese sind in die Berechnung der entscheidenden Verzweigungen einzubeziehen.

4.4 Abstandsfunktion

Für den automatischen Strukturtest wird eine Funktion benötigt, welche die Optimierung von Testdaten für das Auswertungsergebnis bestimmter Verzweigungsbedingungen des Testobjekts zuläßt. Werden bei der Suche Testdaten gefunden, die im Optimum der Abstandsfunktion liegen, so wird das gewünschte Ergebnis der Bedingung ausgewertet und das Testobjekt führt den für das aktuelle Teilziel zu durchlaufenden

Zweig aus. Lösungsansätze hierfür beschreiben die Arbeiten [Jones96], [Korel90] und [Tracy98b]. Grundlegende Idee ist die Verknüpfung der in der Bedingung enthaltenen Relationsoperanden zu einer Funktion, deren optimaler Wert beim gewünschten boolschen Ergebnis der einzelnen Relation liegt. Zu jeder Relation sind damit zwei Abstandsfunktionen, jeweils eine für den Ergebniswert 'wahr' und eine für den Wert 'falsch', zu bilden.

Für die weiteren Betrachtungen sind Verzweigungen zu differenzieren, welche durch binäre Entscheidungen gekennzeichnet sind, sowie solche, die eine Mehrfachverzweigung zu lassen.

4.4.1 Binäre Verzweigungsanweisungen

In der Abbildung 4.35 ist eine binäre Verzweigung mit einer zusammengesetzten Bedingung dargestellt. Die beiden angegebenen Testfälle sind für die evolutionäre Optimierung auf ihre Eignung zur Ausführung des ‘falsch‘-Zweiges zu prüfen. Dazu liefert die Instrumentierung des Testobjekts Messungen an den Relationen der Verzweigungsbedingung. Die Meßwerte sind in der Abbildung zu den beiden Testfällen vermerkt.

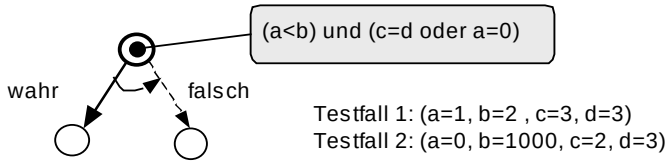


Abbildung 4.35: Zu bildende Abstandsfunktion in einer Verzweigung

Folgende Anforderungen werden an die Abstandsfunktion **Φ** gestellt:

- Sie muß eine Distanz zur gewünschten Entscheidung der Bedingung funktional ausdrücken.
- Der Wertebereich ist auf [0,1] festgelegt. Je kleiner der Wert, desto weniger ist das Testdatum geeignet.
- Mit dem Optimum in '1' erfolgt die Ausführung der gewünschten Verzweigung.
- Der Eingabebereich der Relationsoperanden kann sehr groß sein (z.B. $\pm 10^{+300}$).

Im weiteren werden die Relationen gezeigt und die jeweiligen Funktionen erläutert, welche die oben genannten Forderungen erfüllen. Die Bildung einer Abstandsfunktion für zusammengesetzte Verzweigungsbedingungen mit mehreren beteiligten Relationen wird im Anschluß beschrieben.

Relation 'A=B':

$$\Phi(A,B) = (1+\varepsilon)^{-|A-B|} \quad \text{, mit } 0 < \varepsilon < 1 \tag{4.8}$$

Das Optimum dieser Funktion liegt bei (A – B) = 0, also A = B.

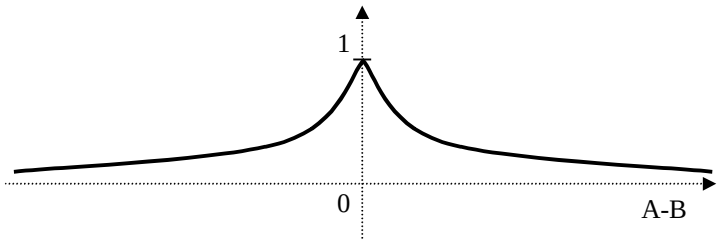


Abbildung 4.36: Abstandsfunktion für "A=B"

Relation 'A≠B':

$$\Phi_{A\neq B}(A,B)=\begin{cases}1 & falls\ A-B\neq 0\\ 0 & sonst\end{cases} \tag{4.9}$$

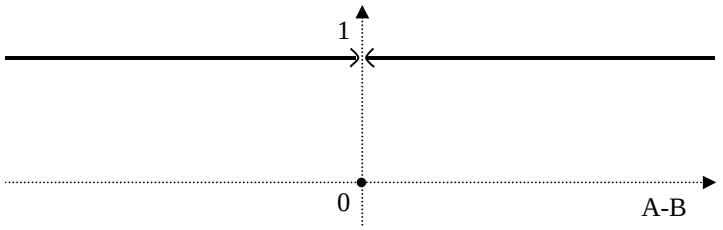


Abbildung 4.37: Abstandsfunktion für "A≠B "

In dieser Abstandsfunktion gibt es nur einen *nichtoptimalen* Punkt in A – B = 0. Dies ist die einzige Stelle, wo A≠B nicht gilt.

Es folgen die Abstandsfunktionen für die Relationen A≤B und A≥B:

$$\Phi_{A\geq B}(A,B)=\begin{cases}(1+\varepsilon)^{A-B} & falls\ A-B\leq 0\\ 1 & sonst\end{cases} \tag{4.10}$$

$$\Phi_{A\leq B}(A,B)=\begin{cases}(1+\varepsilon)^{B-A} & falls\ A-B\geq 0\\ 1 & sonst\end{cases} \tag{4.11}$$

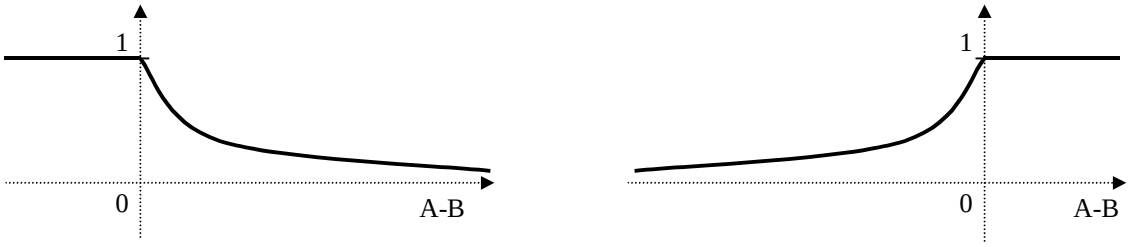


Abbildung 4.38: Abstandsfunktion für A≤B (links) und A≥B (rechts)

Für die Relationen A>B und A<B ist eine Abstandsfunktion zu konstruieren, die insbesondere an der Stelle A=B keinen optimalen Wert besitzt. Die Funktion muß bei der kleinsten Abweichung von diesem Punkt in die gewünschte Richtung das Optimum zurückliefern. Es sei κ ein sehr kleiner Wert entsprechend der Typdefinition der Abstandsfunktion (ASTELA verwendet *double precision floating point* also $\kappa<10^{-200}$).

$$\Phi_{A>B}(A,B) = \begin{cases} (1+\varepsilon)^{A-B} \cdot (1-\kappa) & falls\ A-B \le 0 \\ 1 & sonst \end{cases} \tag{4.12}$$

$$\Phi_{A<B}(A,B) = \begin{cases} (1+\varepsilon)^{B-A} \cdot (1-\kappa) & falls\ A-B \ge 0 \\ 1 & sonst \end{cases} \tag{4.13}$$

Mit dieser Funktion wird erreicht, daß an der Stelle A=B die Abstandsfunktion das Ergebnis „1 – κ“ liefert und das Optimum mit dem Funktionswert „1“ erst bei der kleinsten Abweichung von A = B gefunden wird.

	Abstand zum Ergebniswert			
	Wahr		**Falsch**	
Term: rel	Φ_{wahr}(rel)		Φ_{Falsch}(rel)	
boolean	0		0	
a = b	$\Phi_{a=b}$ (a,b)	(4.8)	$\Phi_{a\neq b}$ (a,b)	(4.9)
a ≠ b	$\Phi_{a\neq b}$ (a,b)	(4.9)	$\Phi_{a=b}$ (a,b)	(4.8)
a < b	$\Phi_{a<b}$ (a,b)	(4.13)	$\Phi_{a\geq b}$ (a,b)	(4.10)
a ≤ b	$\Phi_{a\leq b}$ (a,b)	(4.11)	$\Phi_{a>b}$ (a,b)	(4.12)
a > b	$\Phi_{a>b}$ (a,b)	(4.12)	$\Phi_{a\leq b}$ (a,b)	(4.11)
a ≥ b	$\Phi_{a\geq b}$ (a,b)	(4.10)	$\Phi_{a<b}$ (a,b)	(4.13)

Tabelle 4.2: Abstandsfunktionen für Basisprädikate

Abstandsfunktion für zusammengesetzte Bedingungen

Setzt sich die Bedingung in einem Verzweigungspunkt aus mehreren Relationen zusammen, die miteinander logisch verknüpft sind, so muß diese Verbindung in der Abstandsfunktion ausgedrückt werden.

Logische UND-Verknüpfung

Eine binäre UND-Verknüpfung wird immer dann mit Wahr ausgewertet, wenn beide Operanden Wahr ergeben. Diese Verknüpfung läßt sich am besten mit der Summenbildung der Abstandsfunktionen der Operanden ausdrücken.

Logische ODER-Verknüpfung

Die funktionale Beschreibung einer ODER-Verknüpfung, welche Wahr ergibt, wenn einer der beiden Operanden ‘Wahr’ liefert, läßt sich als Maximum der Abstandsfunktionen der Operanden ausdrücken.

Für die logische Negation einer Teilbedingung ist die Abstandsfunktion des entgegengesetzten logischen Wertes anzusetzen. Die folgende Tabelle faßt die Regeln zur Bildung der zusammengesetzten Abstandsfunktion zusammen:

Bedingung	Abstand zum Ergebniswert	
	Wahr	Falsch
Term:	$\Phi_{wahr}(Term)$	$\Phi_{Falsch}(Term)$
$A \vee B$	$Max(\Phi_{wahr}(A), \Phi_{wahr}(B))$	$\frac{\Phi_{Falsch}(A) + \Phi_{Falsch}(B)}{2}$
$A \wedge B$	$\frac{\Phi_{wahr}(A) + \Phi_{wahr}(B)}{2}$	$Max(\Phi_{Falsch}(A), \Phi_{Falsch}(B))$
$\neg A$	$\Phi_{Falsch}(A)$	$\Phi_{wahr}(A)$

Tabelle 4.3: Abstandsfunktionen für zusammengesetzte Bedingungen (in Anlehnung an [Tracy98b])

Beispiel: (siehe Abbildung 4.35) *Berechnungen für ε=0.0001*

Bedingung: "(a<b) und (c=d oder a=0)"
Gewünschtes Ergebnis: 'Falsch'

(W_1) $\Phi_{Falsch}(Term) = max\ (\Phi_{Falsch}(a<b)\ ,\ \Phi_{Falsch}(\text{"c=d oder a=0"}))$
(W_2) $\Phi_{Falsch}(\text{"c=d oder a=0"}) = (\Phi_{Falsch}(c=d) + \Phi_{Falsch}(\text{"a=0"})) \div 2$
(W_3) $\Phi_{Falsch}(a<b) = \Phi_{a \geq b}(a,b)$ (4.10)
(W_4) $\Phi_{Falsch}(c=d) = \Phi_{c \neq d}(c,d)$ (4.9)
(W_5) $\Phi_{Falsch}(a=0) = \Phi_{a \neq 0}(a,0)$ (4.9)

Testfall 1 (a=1, b=2, c=3, d=3)
(W_3)=0.9999
(W_4)=0
(W_5)=1
(W_2)=0.5
(W_1)=0.9999

Testfall 2 (a=0, b=1000, c=2, d=3)
(W_3)=0.9049
(W_4)=1
(W_5)=0
(W_2)=0.5
(W_1)=0.9049

Ergebnis: Testfall 1 ist näher am Teilziel "Bedingung=Falsch" als Testfall 2.

4.4.2 Abstandsfunktion für Mehrfachverzweigungen

Die Mehrfachverzweigung ist eine besondere Form der Verzweigungsanweisung. Im Kontrollflußgraphen wird diese Verzweigungsart durch einen Knoten mit mehreren ausgehenden Zweigen dargestellt. Die Abstandsfunktion muß je nach ausgeführtem Zweig einen Abstandswert zu allen anderen Zweigen berechnen. Dieser Abstand ist genau wie bei den binären Verzweigungen anhand der Verzweigungsbedingungen zu bilden.

Für die betrachtete Programmiersprache *AnsiC* ist die Abstandsbestimmung sehr einfach zu formulieren, weil die Bedingungen einer Mehrfachauswahl auf die Äquivalenzrelation eingeschränkt sind. Bei der *Switch-Case*-Anweisung von *AnsiC* können außerdem nur Aufzählungstypen verwendet werden. Die Verzweigungsentscheidung erfolgt zur Laufzeit auf Grundlage der Werte der einzelnen *Case*-Anweisungen. Für jeden *Case* wird im Kontrollflußgraphen ein Zweig gebildet.

Beispiel:

```
switch (A) {
        case 1: .... break;             /* Wenn A==1 */
        case 2: case 3: .... break;     /* Wenn A==2 oder A==3 */
        default: .... break;            /* sonst */
}
```

Die Abstandsfunktion für den jeweiligen Zweig lautet deshalb $\Phi_{a=b}$ (a,b) (siehe Formel (4.8)), wobei 'b' der Wert des jeweiligen Zweiges ist. Erlaubt die Programmiersprache die Ausführung eines *Default*-Zweigs, welcher immer dann ausgeführt wird, wenn keiner der spezifizierten Werte zutrifft, muß auch für diesen Zweig eine Abstandsfunktion gebildet werden.

Es seien dazu $B_1, \ldots, B_n$ die Zweigbedingungen der Mehrfachverzweigung. Der Defaultzweig wird immer dann ausgeführt, wenn gilt: $\neg (B_1 \vee \ldots \vee B_n)$. Für eine solche Verzweigungsbedingung kann sehr einfach der Abstand anhand Tabelle 4.3 gebildet werden:

$$\Phi_{\text{default}} = \frac{\Phi_{\text{Falsch}}(B_1) + .. + \Phi_{\text{Falsch}}(B_n)}{n} \tag{4.14}$$

4.5 Erreichbarkeitsgraph

Mit der Motivation jeden Testverlauf für jedes Teilziel mit möglichst wenig Aufwand bewerten zu können, entstand die Idee der Konstruktion eines speziellen Graphen (*Erreichbarkeitsgraph*), welcher die gleichzeitige Bewertung aller Teilziele erlaubt. Der Erreichbarkeitsgraph enthält Knoten und Zweige derart, daß eine individuelle Bewertung aller Teilziele entsprechend der Zielfunktion erfolgen kann.

Die *Knoten* des Graphen stellen die Stufen auf dem Weg zu einem Teilziel dar, deren Inhalt je nach Testverfahren sehr unterschiedlich ist. Während der Kontrollfluß in den Knoten des Graphen nachvollzogen wird, können die Ausführung oder das Abweichen von Teilzielen festgestellt und bewertet werden. Durch dieses Bewertungsverfahren bietet sich die Möglichkeit, zufällig erreichte Teilziele zu erkennen sowie für jedes Teilziel die Testdaten mit der besten Bewertung zusammenzustellen. Die gesammelten Daten dienen als *Startwerte* für die Optimierung eines Teilziels.

Der Erreichbarkeitsgraph vereinigt die Berechnung aller Teilziele eines Testverfahrens. Auf Grund der unterschiedlichen Arten von Zielfunktionen für die jeweiligen Testverfahren gestalten sich die Knoten und Zweige des Erreichbarkeitsgraphen verschieden.

Bei knotenorientierten Tests sind den Knoten des Erreichbarkeitsgraphen die Verzweigungen des Testobjekts und deren Annäherungsstufen zugeordnet. Der Graph für die pfadorientierten Testmethoden gestaltet sich anhand der zu durchlaufenden Abschnitte im Kontrollflußgraphen. Diese werden durch Knoten repräsentiert.

Die *Zweige* des Erreichbarkeitsgraphen entsprechen dem Kontrollfluß des Testobjekts. Dies gilt sowohl für den knotenorientierten Fall, bei dem die Zweige den Kontrollfluß

zwischen den Verzweigungen darstellen, wie auch für den pfadorientierten Test, wo die Zweige den Kontrollfluß von einem Abschnitt in den nächsten kennzeichnen.

Es folgt eine Beschreibung der Umsetzung der Kontrollflußgraphen und der Methode zur Zielfunktionsberechnung für knoten- sowie pfadorientierte Testverfahren und einer besonderen Ausprägung des Graphen für Bedingungsüberdeckungstests.

4.5.1 Erreichbarkeitsgraph für knotenorientierte Testverfahren

Bei den knotenorientierten Testverfahren bilden die Entscheidungen in den Verzweigungsknoten des Kontrollflußgraphen die Grundlage der Berechnung der Zielfunktion. Deshalb ähnelt der Erreichbarkeitsgraph für diese Testmethoden stark dem Kontrollflußgraphen des Testobjekts. Jeder Verzweigung des Kontrollflußgraphen wird ein Knoten im Erreichbarkeitsgraphen zugeordnet. Die Zweige des Graphen sind so einzutragen, daß sie den Kontrollflüssen zwischen den Verzweigungen im Testobjekt entsprechen. In den Knoten des Erreichbarkeitsgraphen werden die Annäherungsstufen und unerwünschten Verzweigungen zu jedem Teilziel vermerkt.

Führt ein Testfall eine bestimmte Verzweigung im Testobjekt aus, können die Eintragungen im zugehörigen Knoten des Erreichbarkeitsgraphen analysiert werden. Anhand der zum durchlaufenen Zweig vermerkten Informationen ist feststellbar, ob Teilziele erreicht wurden oder eine unerwünschte Verzweigung für eines der Teilziele ausgeführt wurde.

Die Nutzung des Erreichbarkeitsgraphen ermöglicht eine effiziente Bestimmung der Annäherungsstufe zu jedem Teilziel für einen beliebigen Testverlauf. Das gilt nicht nur für das aktuell zu optimierende Teilziel sondern auch für alle anderen, so daß eine Zusammenstellung guter Startwerte für jedes Ziel möglich ist.

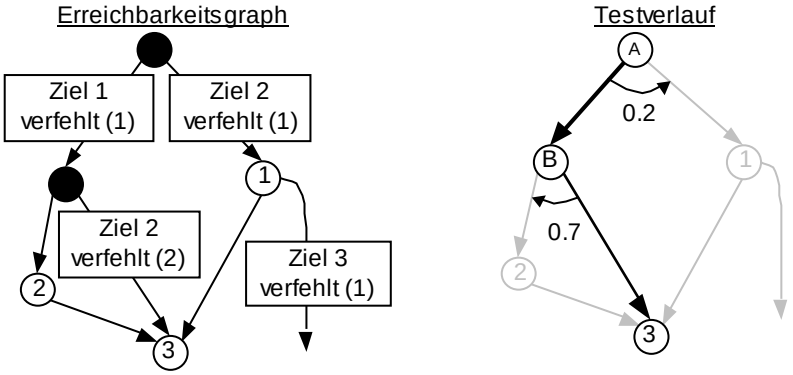


Abbildung 4.39: Beispiel Erreichbarkeitsgraph

Zur Berechnung der Zielfunktion

Über die Rekonstruktion des Kontrollflusses eines Testverlaufs im Erreichbarkeitsgraphen lassen sich die ausgeführten Teilziele feststellen und für alle anderen Teilziele die verantwortlichen Verzweigungen identifizieren. Die Bewertung des Testdatums für die Teilziele kann anhand der jeweils vermerkten Annäherungsstufen berechnet werden. In Abbildung 4.39 wird zur Verdeutlichung dieses Vorgangs ein Beispiel dargestellt.

Die rechte Seite der Abbildung 4.39 zeigt einen Testverlauf mit den durch die Instrumentierung gemessenen Abstandswerten in den ausgeführten Verzweigungsbedingungen. Der Testfall durchläuft im ersten Schritt den Zweig "A→B", welcher den Vermerk "Teilziel 1 verfehlt (Stufe 1)" trägt. Mit dieser Information kann die Bewertung des Testdatums für Teilziel 1 berechnet werden. Der Zielfunktionswert wird aus Annäherungsstufe (1) und Abstandswert (0.2) bestimmt.

$$\text{Zielfunktion}_{\text{Teilziel1}}(\text{Testverlauf}) = 2 - (1 + 0.2) = 0.8$$

Im zweiten Schritt des Testverlaufs wird der Zweig "B→3" ausgeführt. Damit verfehlt der Testfall Teilziel 2 mit Annäherungsstufe 2. Die Bewertung des Testverlaufs für Ziel 2 ist somit „3 – (2 + 0.7)“. Der dargestellte Testverlauf erreicht Teilziel 3.

Zusammenstellung von Startwerten

Durch die Zielfunktionsberechnung im Erreichbarkeitsgraphen werden zu jedem Testverlauf alle erreichten Teilziele und die Annäherung des Testverlaufs an alle nicht erreichten Teilziele bestimmt. Somit ist es der Teststeuerung möglich, neben der Optimierung des momentanen Ziels, gute Startwerte für die anderen Teilziele zu sammeln.

4.5.2 Erreichbarkeitsgraph für pfadorientierte Testverfahren

Der Erreichbarkeitsgraph für pfadorientierte Testverfahren übernimmt zwei Aufgaben der Teststeuerung. Das sind die Berechnung der Zielfunktionswerte für das aktuelle Teilziel und die Zusammenstellung von Startwerten für die anderen Teilziele.

Zur Berechnung der Zielfunktion wurden zwei Methoden vorgestellt. Beide Ansätze vergleichen den Testverlauf mit dem Zielpfad in bestimmten Abschnitten des Kontrollflußgraphen. Ein Teilziel legt für jeden Abschnitt den auszuführenden Pfad fest. Die Zielfunktion prüft abschnittsweise eine Abweichung des Testverlaufs und bewertet die Länge des identischen Wegstücks. Diese Bewertung kann in den Abschnitten unabhängig erfolgen, wobei je nach Zielfunktionsvariante die Ergebnisse unterschiedlich im Gesamtwert zusammengefaßt werden.

Bei der Ermittlung von Startwerten wird auf Grund der größeren Anzahl von Teilzielen gegenüber knotenorientierten Tests keine separate Zusammenstellung von Testdaten für jedes einzelne Ziel vorgenommen. Die Eingabedaten werden entsprechend ihrer Eignung für bestimmte Abschnitte des Gesamtpfades erfaßt.

Wie bereits erläutert, läßt sich der Kontrollfluß eines Testobjekts in Abschnitte zerlegen, welche jeder Testfall durchläuft. Als Teilziele ergeben sich alle Kombinationsmöglichkeiten von Wegen in den einzelnen Abschnitten. Die folgende Abbildung zeigt die Unterteilung des Kontrollflußgraphen.

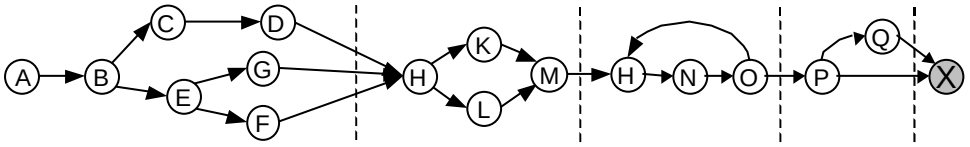


Abbildung 4.40: Kontrollflußgraph mit Abschnitten

In jedem der gebildeten Abschnitte ist, mit Ausnahme von enthaltenen Schleifen, die gesondert betrachtet werden, eine bestimmte Anzahl von Wegen möglich. Alle möglichen Pfade lassen sich hierarchisch nach der Abfolge der einzelnen Abschnitte in einem Baum darstellen. An den Zweigen des Baums sind in der Abbildung die jeweiligen Wege vermerkt.

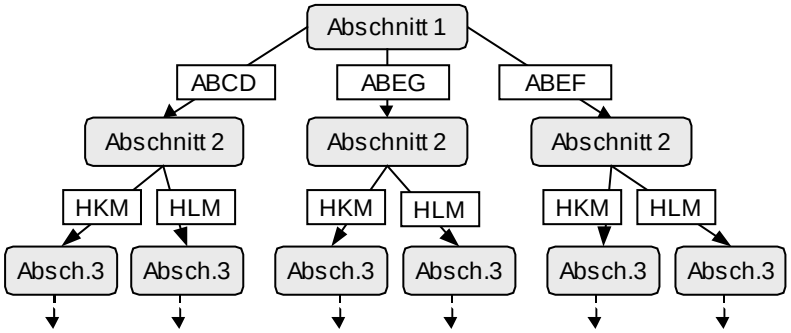


Abbildung 4.41: Aufsplittung der Wege in den Abschnitten aus Abbildung 4.40

Ein Weg vom Wurzelknoten des Baumes zu einem Blattknoten beschreibt einen Pfad durch die zu testende Software. Die *Blätter* dieses Baumes sind die einzelnen Teilziele des Pfadtests. Der Baum gliedert somit die Teilziele hierarchisch nach gleichartigen Anfangspfaden. Alle Teilziele, welche mit einem gemeinsamen Knoten des Baumes verbunden sind, besitzen das gleiche zu diesem Knoten gehörende Anfangswegstück. Dieses Anfangswegstück bildet sich aus den durchlaufenen Wegstücken beginnend beim Wurzelknoten bis zum gemeinsamen Knoten.

Die Knoten eines so konstruierten Erreichbarkeitsgraphen eignen sich zur Erfassung von Startwerten für die beiden vorgestellten Bewertungsmethoden. Für die erste Variante, welche die Länge des identischen Anfangspfads mißt, werden die generierten Testdaten in jedem durchlaufenen Knoten zusammengestellt. Je nach Zielpfad des nächsten zu optimierenden Teilziels werden Startwerte von den auszuführenden Knoten des Erreichbarkeitsgraphen bereitgestellt. Bei der zweiten Methode werden Testdaten besser bewertet, die möglichst viele deckungsgleiche Wegstücke im Gesamtpfad besitzen. Die Startwerte werden deshalb nicht separat in jedem Knoten gesammelt, sondern den Abschnitten des Kontrollflußgraphen zugeordnet.

In beiden Formen müssen gute Startwerte, nicht wie bei knotenorientierten Verfahren, an alle Teilziele propagiert werden. Die Testdaten für ein bestimmtes Teilziel werden erst zu Beginn der Optimierung zusammengestellt.

Der in Abbildung 4.40 gezeigte Graph ließ sich relativ leicht in Abschnitte einteilen, die wenige Knoten enthalten. Bei komplexeren Programmstrukturen entstehen zum Teil große Abschnitte mit sehr vielen Knoten und einer daraus resultierenden hohen Anzahl

möglicher Wege in einem Abschnitt. In diesem Fall ist eine weitere Zerlegung solcher Abschnitte anzustreben, um eine bessere Zusammenstellung von Startwerten zu erreichen. Für die Zerlegung können die Methoden aus Teilabschnitt 4.3.1 "Reduktion des Kontrollflußgraphen" zur Bildung von Knotengruppen verwendet werden.

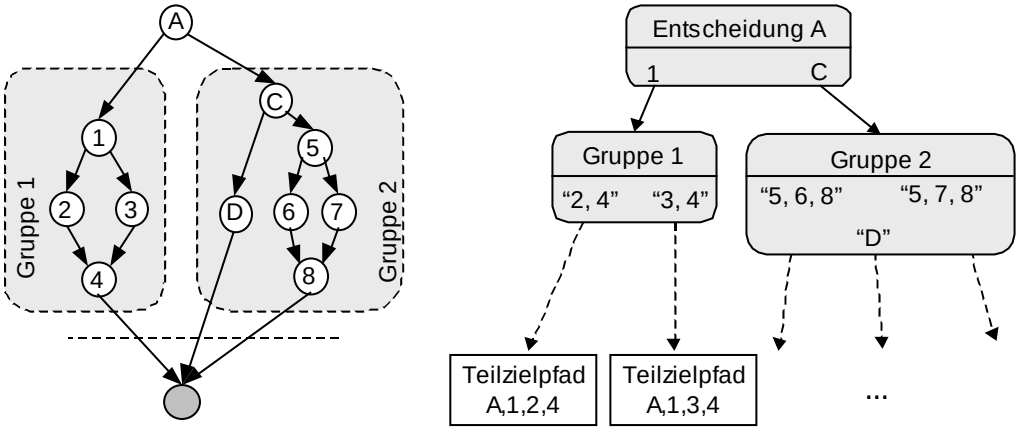


Abbildung 4.42: Vertikale Aufteilung für den Erreichbarkeitsgraphen

Abbildung 4.42 linke Seite zeigt ein Beispiel für den Fall, daß nur ein Abschnitt gebildet werden kann. Dieser Abschnitt enthält alle Knoten vor dem Endknoten (A bis D und 1 bis 8). Die Abbildung zeigt auf der rechten Seite einen Vorschlag zur Unterteilung des Abschnittes mit Hilfe von zwei Knotengruppen (1 bis 4 sowie C, D, 5 bis 8). Mit dieser Zerlegung lassen sich Startwerte getrennt nach Teilwegen in dem Abschnitt zusammentragen.

Das Beispiel ist bewußt einfach gehalten, um die Vorgehensweise zu verdeutlichen. Auf Basis der Zerlegung kann der Erreichbarkeitsgraph gebildet werden, wie die rechte Seite der Abbildung zeigt. In der Abbildung sind aus Platzgründen nur zwei Beispiele angegeben – Teilzielpfad "A, 1, 2, 4" und "A, 1, 3, 4". Weitere Zielpfade sind "A, C, 5, 6, 8" und "A, C, 5, 7, 8" und "A, C, D", was sich leicht aus dem Graphen ableiten läßt.

Die Startwerte können auch für eine solche vertikale Aufsplittung eines Abschnittes getrennt nach den Anfangspfaden erfaßt werden.

Berechnung der Zielfunktion

Die Berechnung der Zielfunktion nach den beiden vorgestellten Methoden orientiert sich an den einzelnen Annäherungen eines Testverlaufs in den Knoten des Erreichbarkeitsgraphen.

Dabei bewertet die erste Methode nur die Länge des identischen Anfangsstücks. Diese Information kann aus dem Baum des Erreichbarkeitsgraphen entnommen werden. Es wird dazu die Abfolge der Erreichbarkeitsgraph-Knoten von Teilziel und Testverlauf verglichen. Kommt es zu einer Abweichung in einem Knoten, so berechnet sich die Annäherung aus der Summe der Weglängen in den vorher durchschrittenen Knoten. Sie erfaßt damit nur den identischen Anfangspfad. Zusätzlich fließt in die Zielfunktion die Abweichung vom Wegstück aus dem letzten gemeinsamen Knoten ein. Das Vorgehen entspricht Formel (4.6) auf Seite 69.

Das zweite Bewertungsverfahren betrachtet die Abschnitte getrennt und summiert die jeweiligen Annäherungen. Jeder Knoten des Erreichbarkeitsgraphen von Testverlauf und Zielpfad wird dazu auf Übereinstimmung geprüft und gegebenenfalls eine Abweichung bewertet. Der Zielfunktionswert ist die Summe aller Einzelwerte der Abschnitte (siehe Formel (4.7) Seite 70).

Es folgt eine Beispielberechnung:

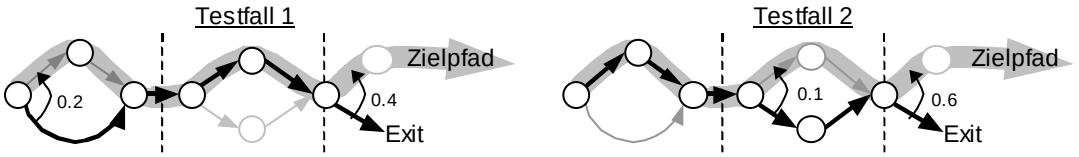


Abbildung 4.43: Zwei Testfälle mit einem Zielpfad (grau)

Methode 1: (Länge der identischen Abschnitte + Annäherung)
Testfall 1: ZF = 0 + 0.2 = 0.2
Testfall 2: ZF = Länge(Abschnitt1)+0.1 = 3 + 0.1

Methode 2: (drei Abschnitte – jeweils ein Annäherungswert)
Testfall 1: ZF = 0.2 + 3 + 0.4 = 3.6
Testfall 2: ZF = 3 + 0.1 + 0.6 = 3.7

Abschnitt mit Schleife

Für Abschnitte mit Schleifen lassen sich nicht ohne weiteres alle möglichen Wege bestimmen. Daher beschränken sich die Kriterien der Pfadüberdeckung auf die ersten *k* Iterationen einer Schleife. Es werden die möglichen Wege einer Iteration betrachtet:

(1) Eintrittswege der Schleife – dazu gehören die Wege der ersten Wiederholung und Wege, die zum sofortigen Austritt aus der Schleife führen

(2) Wege, welche die Schleife einmal wiederholen

(3) Wege, welche die Schleife verlassen

Für die Berechnung der Zielfunktion wird die Überprüfung dieser Wegstücke in *k* Iterationen gefordert. Dazu sind je nach Testkriterium eine Sequenz von Wegstücken (1) 'Eintritt', (2) 'Wiederholung' und (3) 'Austritt' zu bilden. Es ist zu berücksichtigen, daß vor dem Austritt aus einer Schleife eine beliebige Wiederholung der Schleife erlaubt ist.

Der Erreichbarkeitsgraph zu einem Abschnitt mit einer Schleife wird somit aus den Verkettungen der Wege (1), (2) und (3) gebildet. Das folgende Beispiel in Abbildung 4.44 verdeutlicht den Erreichbarkeitsgraphen für die Schleife in Abschnitt II. Wie die Darstellung zeigt, sind innerhalb einer Iteration drei Wege möglich: 'EFHK', 'EFGHK', 'EFGK'. Die Schleife besitzt einen Austrittsweg ('EFGL'). Für den Eintritt in die Schleife (siehe [2] in der Abbildung) ergeben sich vier konstruierbare Wege: "EFHK", "EFGHK","EFGK" und "EFGL". Letzterer verläßt die Schleife sofort wieder.

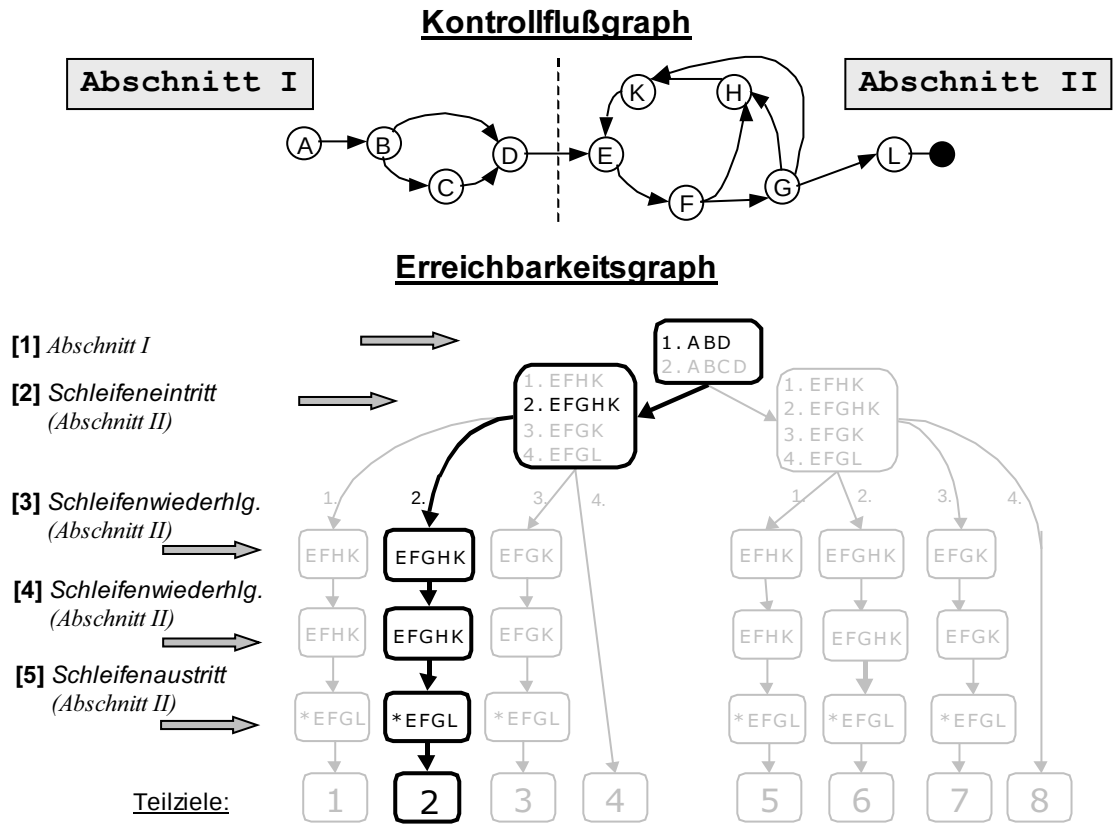


Abbildung 4.44: Abschnitt mit Schleife

Für die Wiederholung (siehe [3], [4]) gibt es drei Wege: "EFHK", "EFGHK" sowie "EFGK". Die Teilziele der strukturierten Pfadüberdeckung schreiben identische Schleifeniterationen vor. Deshalb splittet sich der Baum nach der ersten Iteration nicht weiter auf. Der Erreichbarkeitsgraph in dieser Form ermöglicht jedoch auch die Darstellung von Teilzielen anderer Testkriterien, die Wegvariationen zwischen den Iterationen zu lassen. Den Abschluß jedes Pfades bildet eine beliebige Wiederholung der Schleife bis der Austrittsweg genommen wird (siehe [5]).

In der Abbildung ist der Zielpfad von Teilziel 2 hervorgehoben. Er lautet: "ABD-EFGHK-EFGHK-EFGHK-(*)EFGL". Bei einem Testverlauf 'ABD-EFGL' ergibt sich damit eine Abweichung vom Zielpfad bei der Abstandsmessung [2]. Für die erste Bewertungsmethode muß der Abstand zwischen dem genommenen Weg "EFGL" und dem Weg des Teilziels "EFGHK" gemessen werden:

Annäherung(Teilziel 2)= Länge(Abschnitt I *) + Länge(* 'EFG' *) +*
Abstandsfunktion(G→H *))*

In einem weiteren Testverlauf "ABD-EFGHK-EFGK-EFGL" ergibt sich eine Abweichung bei der zweiten Wiederholung, also Abstandsmessung [3].

Annäherung(Teilziel 2)= Länge(Abschnitt I *) + Länge(* 'EFGHK' *) +*
Länge('EFG' *) + Abstandsfunktion(* G→H *))*

Verschachtelung von Schleifen

Für geschachtelte Schleifen sind sowohl Abweichungen vom Weg der inneren wie auch vom Weg der äußeren Schleifeniterationen zu prüfen. Der Zielpfad beschreibt jeweils den Weg für die ersten *k* Iterationen. Es bietet sich an, den Erreichbarkeitsgraphen für diesen Fall ähnlich zu schachteln. Dazu werden Knoten für die innere Schleife gebildet. Diese Knoten prüfen den Weg im Teilgraphen des inneren Schleifenkörpers und messen gegebenenfalls Abweichungen vom Zielpfad. Sie enthalten hierfür Informationen zum Teilgraphen der inneren Schleife und sind intern genauso strukturiert wie ein Erreichbarkeitsgraph. Abbildung 4.45 rechter Teil stellt einen solchen Knoten für die Schleife 'ABCD' dar.

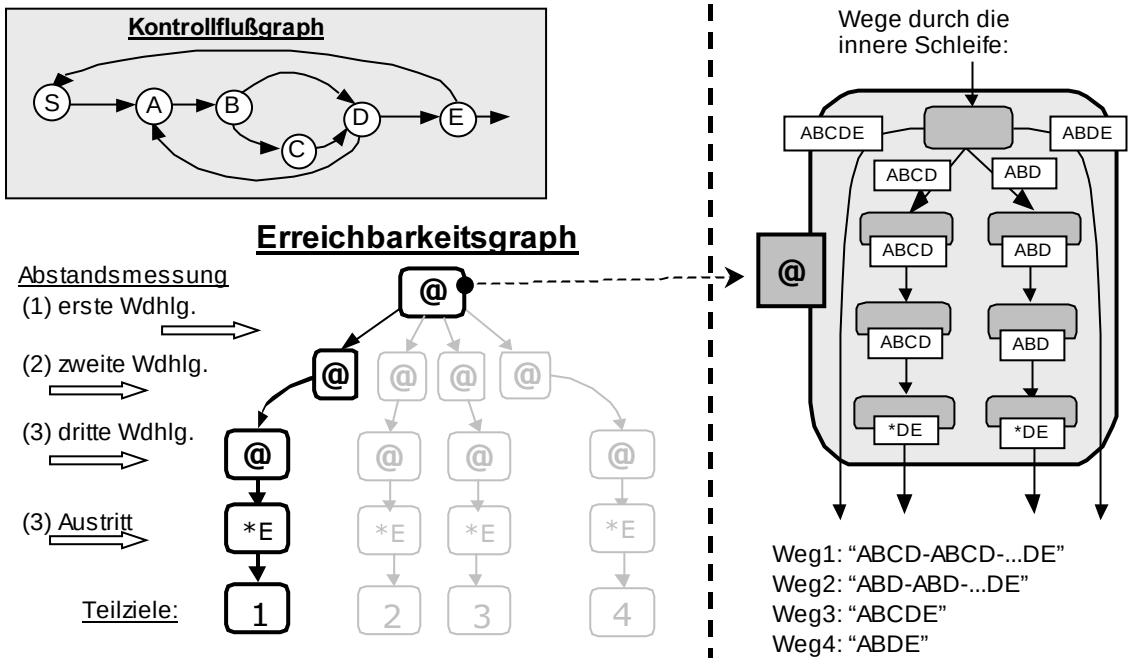


Abbildung 4.45: Verschachtelte Schleifen

Abbildung 4.45 zeigt ein einfaches Beispiel einer Verschachtelung. Die innere Schleife enthält vier Wege. Zwei davon führen zur Wiederholung, die anderen beiden zum Austritt aus der inneren Schleife. Anhand dieser Information kann der *Erreichbarkeits-graph-Knoten* auf der rechten Seite der Abbildung entworfen werden. Ein solcher erweiterter Knoten enthält einen Teilgraphen, welcher die Wege der inneren Schleife beschreibt.

Der auf der linken Seite der Abbildung dargestellte Erreichbarkeitsgraph zeigt die vier Teilziele zur Ausführung der Schleifenverschachtelung für das Testkriterium: ‚*strukturierte Pfadüberdeckung* (k=3; also dreimalige identische Wiederholung)'. Entsprechend des Weges der ersten Schleifeniteration ist der Erreichbarkeitsgraph im Wurzelknoten verzweigt. Alle folgenden Knoten prüfen die Übereinstimmung der weiteren zwei Iterationen und das Verlassen der Schleife nach beliebiger Wiederholung. Auf diese Weise wird die 3-Äquivalenz der Schleifenwiederholungen überprüft (Definition siehe Teilabschnitt 2.2.1).

Gute Startwerte für Teilziele

Als Startwerte für die evolutionäre Optimierung sollen die Testdaten mit den besten Zielfunktionswerten genutzt werden. Anders als bei den knotenorientierten Testverfahren wird auf Grund der großen Anzahl von Teilzielen von einer individuellen Berechnung der Zielfunktion abgesehen. Statt dessen werden die Informationsträger des Erreichbarkeitsgraphen genutzt, während der laufenden Optimierung gute Testdaten zu erfassen. Zu Beginn der nächsten auszuführenden Optimierung werden dann die besten Testdaten ausgelesen. Die beiden vorgestellten Varianten der Zielfunktion für pfadorientierte Tests bewerten unterschiedliche Kriterien. Deshalb richtet sich die Erfassung der Testdaten auch nach der verwendeten Bewertungsmethode.

Für die erste Variante der Zielfunktionsberechnung sind Testdaten zusammenzustellen, welche ein möglichst langes identisches Anfangsstück des Zielpfads ausführen. Weil zu jedem Knoten des Erreichbarkeitsgraphen genau ein Anfangsweg im Testobjekt gehört, liegt es nahe, den Knoten die Aufgabe zu übertragen, solche Testdaten aufzunehmen, welche den zugehörigen Weg durchlaufen. Für die Erfassung der Testdaten wird einfach jeder Test entsprechend des ausgeführten Kontrollflusses im Erreichbarkeitsgraphen nachvollzogen und das zum Test gehörende Eingabedatum in den durchlaufenen Knoten vermerkt. Der Aufwand ist damit gegenüber einer individuellen Zielfunktionsberechnung für jedes Teilziel erheblich reduziert.

Zu Beginn der Optimierung eines Teilziels werden Startwerte aus den zu durchlaufenden Knoten des Erreichbarkeitsgraphen entnommen. Die besten Testdaten ergeben sich aus dem Knoten am Ende des Zielpfades, weil dort der längste identische Anfangsweg vorliegt. Sind in diesem Knoten keine Startwerte vorhanden, so wurde der zugehörige Anfangspfad bis zu diesem Punkt noch nicht ausgeführt. In diesem Fall sind die vorhergehenden Knoten zu untersuchen. Sie beschreiben ein kürzeres Anfangsstück des Zielpfades, wodurch sich die Wahrscheinlichkeit erhöht, daß für den Pfad Eingaben existieren.

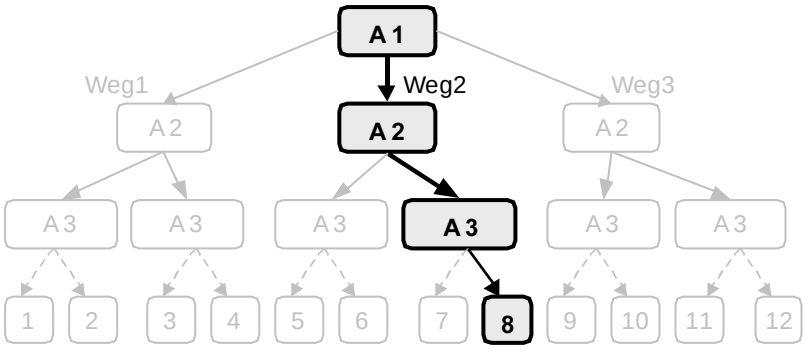


Abbildung 4.46: Einfacher Erreichbarkeitsgraph mit eingezeichnetem Testverlauf

Die Abbildung 4.46 zeigt einen Erreichbarkeitsgraphen mit einem hervorgehobenen Testverlauf. Aus den durchlaufenen Knoten des Erreichbarkeitsgraphen ist erkennbar, daß der Test das Teilziel 8 erreicht und sehr gut für Teilziel 7 geeignet ist, da ein identisches Anfangswegstück in den Abschnitten A1 und A2 vorliegt. Im Zweig von Knoten A3 zum Teilziel 7 werden bei der Optimierung die besten Startwerte für den

vollständigen Zielpfad gesammelt. Die Eignung des Testdatums für die Teilziele 5 und 6 ergibt sich aus Knoten A2. Dort wird vom Pfad dieser beiden Ziele abgewichen. Der Zielpfad der anderen acht Teilziele ist bereits im ersten Abschnitt anders als gewünscht.

Für die zweite Bewertungsmethode sind Startwerte zusammenzustellen, welche möglichst viele identische Wegstücke in den einzelnen Abschnitten des Kontrollflußgraphen besitzen. Weil die Bewertung unabhängig von der Lage der identischen Teile erfolgt, ist eine Erfassung von Startwerten schwierig.

Ein Vorschlag ist die abschnittsweise Zusammenstellung der besten Testdaten. Dazu wird für jeden Weg in jedem Abschnitt eine Liste der Testdaten erfaßt. Die Bestimmung der Startwerte zu Beginn einer Optimierung greift auf die Testdatenlisten der auszuführenden Wegstücke zurück und berechnet für jeden Eintrag den Zielfunktionswert zum gewünschten Zielpfad. Bei dieser Berechnung werden die Testdaten mit den besten Bewertungen in einer Startmenge für das Teilziel zusammengefaßt. Folgende Abbildung zeigt zwei Testfälle und den gewünschten Zielpfad:

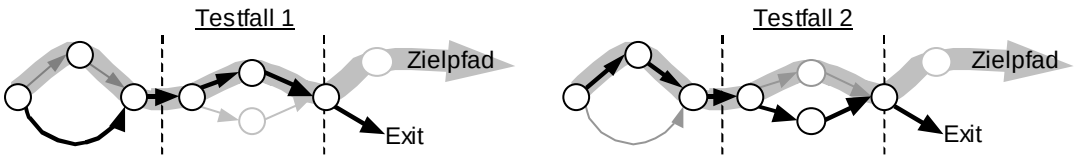


Abbildung 4.47: Zwei Testfälle mit unterschiedlichem Verhalten in den Abschnitten

Der erste Testfall führt im mittleren Abschnitt ein identisches Wegstück aus, wohingegen der zweite Testfall im Anfangsstück mit dem Zielpfad zusammenfällt. Die beiden Testfälle sind damit in den Listen der Abschnitte 1 und 2 erfaßt. Eine Synthese beider Eigenschaften kommt dem Zielpfad sehr nahe, deshalb werden die zugehörigen Testdaten in die Menge der Startwerte einbezogen, so daß der evolutionäre Algorithmus die Möglichkeit erhält, eine Kombinationen zu erzeugen.

4.5.3 Erreichbarkeitsgraph für Bedingungsüberdeckung

Eine Besonderheit dieses Testverfahrens gegenüber den anderen knotenorientierten Verfahren ist, daß sogenannte Teilzielgruppen zu jedem Verzweigungsknoten gebildet werden können. Alle in der Gruppe enthaltenen Teilziele beziehen sich auf die Ausführung desselben Verzweigungsknotens. Sie sind deshalb an den gleichen entscheidenden Verzweigungen einzutragen und bei der Berechnung der Zielfunktion gemeinsam zu berücksichtigen. Unterschiede im Zielfunktionswert ergeben sich erst, wenn der zugehörige Zielverzweigungsknoten erreicht wird.

Für die effiziente Berechnung der Zielfunktion bei Bedingungsüberdeckungstests wird der in Abschnitt 4.5.1 beschriebene Erreichbarkeitsgraph leicht modifiziert. Die Knoten und Zweige verweisen auf Teilzielgruppen, deren Elemente gemeinsam betrachtet werden. Die erweiterte Zielfunktionsberechnung des Bedingungsüberdeckungstests wird erst bei Erreichen des Zielknotens für jedes Gruppenelement ausgeführt. Dazu sind die Formeln der Bewertungsfunktion für die ausgeführten Teilbedingungen einer Verzweigung zu verwenden (siehe Teilabschnitt 4.3.3).

Durch die Ausbildung von Teilzielgruppen reduziert sich die Anzahl der Verweise auf Teilziele erheblich. Folgende Abbildung zeigt dies an einem einfachen Beispiel. Anstelle von drei Verweisen an den entscheidenden Zweigen (siehe ‚*verfehlt*') und den daraus resultierenden Berechnungen für jeden Verweis, enthält der Erreichbarkeitsgraph nur jeweils einen Verweis für alle Teilziele der betrachteten Bedingung (*A und B*).

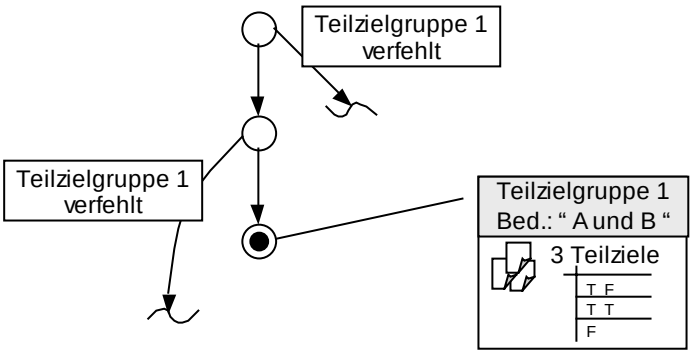


Abbildung 4.48: Modifizierter Erreichbarkeitsgraph mit Teilzielgruppen

Die Bestimmung der Startwerte ist so zu modifizieren, daß erst bei Erreichen des Verzweigungsknotens eines Teilziels separat Testdaten gesammelt werden. Wird der Knoten nicht erreicht, ist der Zielfunktionswert für alle Gruppenelemente gleich und die Startwerterfassung kann für die gesamte Gruppe erfolgen.

4.6 Werkzeuge für den dynamischen Strukturtest

Das System ASTELA nutzt zur Vorbereitung des automatischen Strukturtests die drei Komponenten Parser, Instrumentierer und Testtreibergenerator. Sie liefern die notwendigen Informationen für die Durchführung der Tests. Weil diese drei Systemkomponenten eng zusammenarbeiten, wurden sie in einem Werkzeug integriert. Die folgende Abbildung verdeutlicht den Informationsfluß zwischen den Systemelementen.

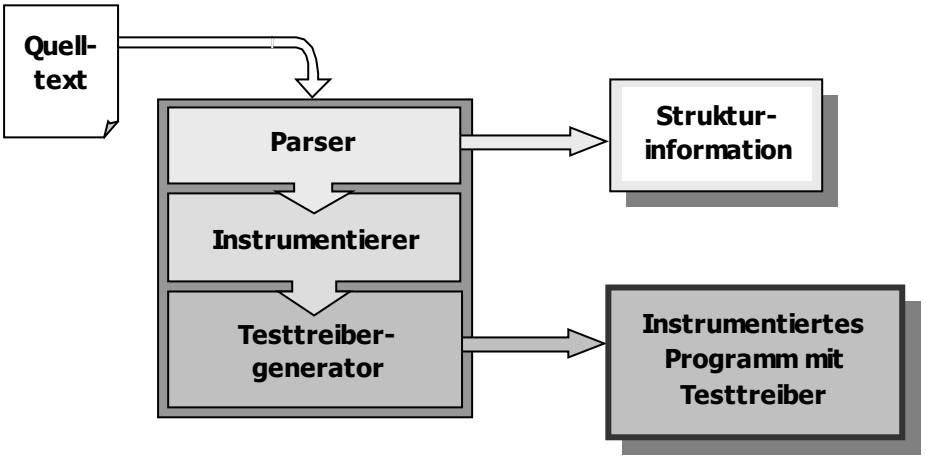


Abbildung 4.49: Parsieren, Instrumentieren und Testtreiber generieren

Ausgehend vom Quelltext werden die Strukturinformationen des Testobjekts durch den Parser identifiziert. Die Informationen umfassen Daten zum Kontrollflußgraphen, zur

Eingabeschnittstelle und zu den Verzweigungsbedingungen. Der Instrumentierer übernimmt diese sowie weitere semantische Daten und führt eine Quelltexttransformation durch. Dabei wird an bestimmten Stellen des Quelltextes Programmcode eingefügt. Die eingefügte Instrumentierung nimmt Messungen bei der Ausführung des Testobjekts vor und dient damit der Berechnung der Zielfunktion. Abschließend erzeugt der Testtreibergenerator einen Testtreiber für die zu testende Software. Die generierte Schnittstelle wird für die Kommunikation zwischen Testobjekt und Teststeuerung benutzt.

Während bei der Testausführung von der verwendeten Programmiersprache abstrahiert werden kann, ist die Testvorbereitung eng an die Quellsprache gebunden. Das Werkzeug zur Testvorbereitung muß anhand der Syntax der Sprache die Kontroll- und Datenstrukturen des Testobjektes identifizieren (*Parser*). Auch der *Instrumentierer* ist auf die Programmiersprache des Testobjekts festgelegt, weil die Transformationen der Syntax und Semantik der Sprache folgen müssen. Die Änderungen des Quelltextes betreffen vor allem die Ausdrücke der Verzweigungsbedingungen, für die detaillierte Typinformationen der beteiligten Bezeichner vorliegen müssen.

Der *Testtreibergenerator* hat die Aufgabe, die zu testende Software so zu ergänzen, daß ein direkter Aufruf des Testobjekts möglich ist. Diese Ergänzungen erfolgen in der Programmiersprache der zu testenden Software, womit auch für diese Komponente eine enge Bindung an die Quellsprache besteht.

Bei der Entwicklung der Komponenten für die Testvorbereitung bestand das Interesse, die Integration von weiteren Programmiersprachen so einfach wie möglich zu gestalten. Die Arbeit [Baresel] beschreibt einen Lösungsansatz zur Reduktion des Implementationsaufwands bei der Erweiterung des Werkzeugs um neue Quellsprachen. Der Parsergenerator [YACC99] bildete dazu die Entwicklungsgrundlage.

4.6.1 Parser

Für den Strukturtest einer Software bedarf es in der Vorbereitung der Identifikation von Kontroll- und Datenstrukturen. Mit diesen Informationen werden die Testziele bestimmt. Zu jedem Testobjekt ist ein sogenannter Kontrollflußgraph zu konstruieren. Dazu muß der Aufbau der zu testenden Software mit Hilfe eines Parsers analysiert werden. Mit diesem Werkzeug ist es möglich, die Elemente einer Sprache sowie ihre syntaktische Zusammenstellung zu bestimmen und daraus die möglichen Kontrollflüsse abzuleiten.

Der Parser benötigt einen sogenannten *Scanner*, der die Symbole der Quellsprache identifiziert. Die mit Hilfe des Scanners bestimmte Symbolfolge wird durch den Parser interpretiert und daraus jedem Symbol die syntaktische Bedeutung zugeordnet. Zum Beispiel kann eine Klammer einen Ausdruck vervollständigen oder den Abschluß einer Parameterliste beschreiben. Die Testvorbereitung verwendet den Parser für die Identifikation der Quelltextabschnitte und die Vorbereitung der Transformationen. Dazu gehört die Zusammenstellung semantischer Informationen, wie zum Beispiel Typbeschreibungen von verwendeten Variablen, sowie der Parameter einer zu testenden Funktion.

Folgende Informationen müssen für die Testvorbereitung mit Hilfe des Parsers erfaßt werden:

- der Kontrollflußgraph (*Strukturinformation für die Testausführung*)
- die semantischen Informationen der verwendeten Datenstrukturen, dazu gehört der Aufbau der nutzerdefinierten Typen im Quelltext (*wird für die Instrumentierung und den Testtreibergenerator benötigt*)
- die Eingabeschnittstelle eines Testobjekts, dazu gehören alle Variablen die während der Ausführung gelesen werden (*Nutzung durch Testtreibergenerator*)
- die Verzweigungsbedingungen und deren Aufbau bis zu den Relationen (*Instrumentierung und Strukturinformation für die Testausführung*)

Zielstellung bei der Entwicklung des Parsers für das System ASTELA war eine Komponente, welche durch wenige Anpassungen auch für andere Programmiersprachen verwendet werden kann. Zu diesem Zweck wurde ein objektorientiertes Konzept entwickelt, daß eine Grundfunktionalität für die Erfassung von Datenbeschreibungen, Konstruktion des Kontrollflußgraphen und Instrumentierung des Quelltextes bereitstellt.

Der hier vorgestellte Parser wandelt die Symbole eines Quelltextes in Objekte eines allgemeinen Sprachkonzepts. In [Ghezzi87] wird ein Überblick zu den Kontrollstrukturelementen von verschiedenen Programmiersprachen gegeben. Dazu gehören zum Beispiel Anweisungsfolgen, Schleifen und Verzweigungen. Aus den Verknüpfungen der Objekte kann der Kontrollfluß abgeleitet und damit der Kontrollflußgraph konstruiert werden.

Bei dem entwickelten Basismodell wird von der sprachspezifischen Beschreibung der Programmelemente abstrahiert. Diese Methode erlaubt die Implementation der Erstellung des Kontrollflußgraphen, der Identifikation von Verzweigungsbedingungen sowie der Bestimmung der Meßpunkte unabhängig von der verwendeten Programmiersprache der zu testenden Software. Die Funktionen der *Instrumentierung* und *Testtreibergenerierung* werden im Basismodell offen gelassen. Sie sind für jede Sprache unterschiedlich und müssen deshalb in den sprachspezifischen Erweiterungen integriert werden.

Parser-Generator

Eine sehr einfache Methode der Erstellung eines *Parserautomaten* ist die Verwendung sogenannter *Parser-Generatoren*. Diese Werkzeuge können anhand einer Beschreibung der Syntax einen Automaten erzeugen, der die Parsierung übernimmt.

Ein Parsergenerator aus der UNIX-Werkzeugsammlung ist YACC. Dieses Programm wurde in der ersten Version von *S. C. Johnson* für eine UNIX-Plattform geschrieben. Es ist eng mit LEX, einem Generator für die lexikalische Analyse (Scanner), verbunden. YACC ist ein Werkzeug, das zu einer gegebenen Spezifikation einer Sprache in der *Backus-Naur Form* (*BNF*) den zugehörigen Parser erzeugen kann. Viele moderne Programmiersprachen sind in der BNF beschrieben. Der Parsergenerator YACC [YACC99] bietet zudem den Vorteil, daß er für eine breite Palette von Betriebssystemen zur Verfügung steht. Damit wird eine gute Portierbarkeit des Werkzeugs zur Testvorbereitung sichergestellt.

Durch besondere Angaben in der Beschreibung der Quellsprache des Parsers erlaubt YACC zusätzliche Funktionalität in den zu erzeugenden Automaten zu integrieren (siehe [LexYacc]). So kann ein Werkzeug erstellt werden, das während der syntaktischen Überprüfung Informationen aus dem Quelltext zusammenstellt. Von der syntaktischen Korrektheit der Software wird dabei ausgegangen, so daß der Parsierungsprozeß nur der Bestimmung der Softwarestruktur dient.

Für die Testvorbereitung wurde dazu ein Objektmodell entworfen, daß die Repräsentation von Programmstrukturelementen, wie zum Beispiel Schleifenanweisungen, zuläßt. In den Parser-Automaten wird die Konstruktion der verschiedenen Objekte je nach Syntax der Sprache integriert. Als Ergebnis der Parsierung entsteht eine Verknüpfung von Objekten, auf deren Basis der Kontrollflußgraph und die Schnittstellen eines Testobjekts bestimmt werden können.

Das Objekt-Modell für den Parser

Die Kontrollstrukturen einer Programmiersprache erlauben die Identifizierung von Softwarekomponenten (*Funktionen* und *Datenstrukturen*) und deren Ausführungsreihenfolge. In den verschiedenen Sprachen existieren Beschreibungsmechanismen auf modularer und anweisungsorientierter Ebene [Ghezzi87]. Die modulare Ebene beschreibt die Zerlegung eines Programms in Funktionseinheiten und Datenstrukturdefinitionen. Hierzu gehört beispielsweise die Festlegung einer Schnittstelle für den Aufruf der Funktion und deren Funktionskörper sowie die Definitionen von globalen Variablen.

- Das *Anweisungskonzept* legt die Reihenfolge der Ausführung einzelner Anweisungen fest.
- Anweisungen und Bedingungen können mathematische Terme enthalten, die während der Laufzeit ausgewertet werden. Für die Beschreibung wird ein *Ausdruckskonzept* verwendet.
- Durch das *Datenstrukturkonzept* werden Festlegungen zu Datentypen und Aufrufschnittstellen durch ein Objektmodell abgebildet.

Das *Anweisungskonzept* stellt Objekte für die verschiedenen Arten von Anweisungsstrukturen zur Verfügung. Als Grundlage des entwickelten Objektmodells wurden die drei in [Ghezzi87] identifizierten traditionellen Strukturierungsformen verwendet. Die drei Formen sind Sequenzbildung (*sequencing*), Auswahl (*selection*) und Wiederholung (*repetition*). Für die sprachunabhängige Darstellung der Kontrollflußstrukturierung eines Testobjekts werden die folgenden Konzepte bereitgestellt:

- Einzelanweisung (*statement*)
- bedingte Einzelanweisung (*selection*)
- Sequenz von Anweisungen (*sequencing*)
- Schleifenanweisung (*repetition*)
- binäre Verzweigungsanweisung (*selection*)
- Mehrfachverzweigung (*selection*)

Für diese Strukturierungskonzepte kann die Beschreibung der Konstruktion eines Kontrollflußgraphen unabhängig von der Programmiersprache des Testobjekts festgelegt werden. Im weiteren lassen sich in einer solchen Strukturbeschreibung die Verzweigungspunkte des Testobjekts anhand der jeweiligen Anweisungsformen identifizieren.

Das *Ausdruckskonzept* beschreibt Form und Aufbau von mathematischen Termen in Quellsprachen. Ausdrücke können in einer hierarchischen Struktur von Operationen und Operanden dargestellt werden. Dies ist Aufgabe der Objekte des Ausdruckskonzepts. Eine besondere Ausprägung sind boolsche Ausdrücke (Bedingungen). Sie zeichnen sich durch logische Operationen und relationale, also vergleichende Operanden, aus. Boolsche Ausdrücke sind für die Festlegung der Meßpunkte bei der Instrumentierung näher zu betrachten.

Ein *Datenstruktur-Konzept* dient der Beschreibung der benutzerdefinierten Datentypen eines Testobjekts und der allgemeinen Beschreibung einer Funktionalität zur Datentypbestimmung von Ausdrücken. Dieses Konzept wird außerdem bei der Testtreibergenerierung für die Kodierung der Datentypen der Eingabeparameter eines Testobjekts benötigt. Auf Basis der Beschreibung werden Funktionen zur Konvertierung der Testdaten in die im Quelltext definierten Datenstrukturen generiert.

[Ghezzi87] beschreibt eine Reihe von Methoden zur Datendefinition in den analysierten Quellsprachen. Ausgehend von verschiedenen *Build-In-Types* werden zusammengesetzte Datentypen (*Data Aggregats*) erläutert.

Das entwickelte Datenstruktur-Konzept umfaßt ein Basismodell für die Abbildung der Datentypenbeschreibungen auf Objekte. Es stellt Basistypen, Aufzählungstypen (*Build-In-Types*), zusammengesetzte Typen (*Cartesian-Product*), vereinigte Typen (*discriminated union*) sowie Mengentypen (*Powerset*) bereit. In diese Objekte ist die Datenstrukturierung durch Sequenzbildung von Typen und die rekursive Gestaltung von Datentypen integriert. Die Festlegung von Namen für Datentypen und deren Verwendung bei der Beschreibung von Variablen und Schnittstellen ist Aufgabe einer Symboltabelle, die im folgenden vorgestellt wird.

Aufbau einer universellen Symboltabelle

Eine Symboltabelle beschreibt die semantischen Aspekte einer Software. Sie enthält die verwendeten Bezeichner eines Quellprogramms. Die Verwaltung einer solchen Tabelle zählt für gewöhnlich nicht zu den Aufgaben des Parsers, sondern ist Bestandteil der semantischen Analyse (Compilerbau). Für die Testvorbereitung spielt jedoch die Identifikation der Bezeichner in einem Programm und die Zuordnung der Beschreibung eine wesentliche Rolle. Zudem benötigen einige Sprachen, wie zum Beispiel *AnsiC*, schon für das Parsen des Quelltextes Symbolinformationen, um eine Unterscheidung zwischen bestimmten Arten von Bezeichnern vornehmen zu können. Deshalb wurde die Erfassung der Symboltabelle in den Parser integriert.

Unter einem *Symbol* wird im entwickelten Parser ein Name und eine Beschreibung des Symbols verstanden. Symbole können zum Beispiel Variablen und Funktionsnamen, aber auch Typen- oder Klassenbezeichner sein.

Eine Symboltabelle bildet die Sichtbarkeiten (*Scope*) von Variablen in einem Quellprogramm nach. Ein *Scope* ist der Bereich, in dem eine Variable definiert ist, also mit ihrem Namen verwendet werden kann. In [Ghezzi87] werden zwei Konzepte der Sichtbarkeit von Variablen in Programmiersprachen beschrieben. Danach gibt es die statische und die dynamische Variablensichtbarkeit (*static and dynamic scope binding*).

Die dynamische Sichtbarkeit liegt vor, wenn Variablen zur Laufzeit in beliebigen Programmteilen benutzt werden können und sich diese Eigenschaft bei Programmablauf je nach den zuvor ausgeführten Anweisungen ändert. Bei der Entwicklung der hier vorgestellten Symboltabelle wird nur das statische Konzept unterstützt. Die statische Bindung geht von der Sichtbarkeit der Variablen in bestimmten Bereichen des Quelltextes aus.

Strukturierte Sprachen wie *Pascal*, *Modula* und *AnsiC* definieren in ihrer Syntax eine Blockhierarchie der Kontrollstrukturen. Die Sichtbarkeiten (Scope) von Symbolen werden in Form dieser Hierarchie gekapselt. Ein einfaches Beispiel dafür sind globale und lokale Variablen. Die eindeutige Bestimmung eines Bezeichners ist anhand der Verschachtelung der Scopes festgelegt. Der innerste Sichtbarkeitsbereich hat immer Vorrang.

Die Aufgabe der Symboltabelle ist die eindeutige Zuordnung von Namen und Beschreibung eines Bezeichners an jeder Position im Quelltext. Derselbe Name darf unter Umständen mehrfach verwendet werden. Die Symboltabelle führt dafür verschiedene Namensbereiche. In *AnsiC* ist es zum Beispiel möglich, daß im selben Quelltextbereich ein Bezeichner einer Struktur (*struct* name) und eine gleichnamige Variable existieren. Eine eindeutige Identifikation ist anhand des Schlüsselwortes *struct* zugesichert.

Typbestimmung von Ausdrücken

Für die Quelltexttransformation in der Instrumentierung ist es notwendig, den Ergebnistyp eines Ausdrucks bestimmen zu können, weil die Form der Messung von den verglichenen Typen abhängig ist. Das Konzept der Typauflösung in einem Ausdruck umfaßt neben der Behandlung zusammengesetzter Typen die impliziten und expliziten Typumwandlungen. Diese sind je nach Programmiersprache festgelegt.

Abstrakte Datentypen und die Zuordnung von eigenen Operationen wurden nicht implementiert. Dies ist zum Beispiel für Java und C++ notwendig. Abstrakte Datentypen erlauben ein komplettes Überladen aller Operationen einschließlich der Relationen, so daß neben einer umfangreichen Erweiterung des Datenstruktur-Konzepts die Methodik der Durchführung von Messungen (Instrumentierung) überarbeitet werden muß.

4.6.2 Instrumentierer

In Analogie zur Überprüfung technischer Systeme durch Messungen an verschiedenen Teilkomponenten, können Softwaresysteme instrumentiert, das heißt mit Meßpunkten versehen werden [Huang79]. Bei der Instrumentierung von Software ist genau wie bei

anderen technischen Systemen darauf zu achten, daß eine Messung nicht die Funktionalität ändert, beziehungsweise die Ergebnisse verfälscht.

Formen der Instrumentierung

Die Arbeit von *J.C. Huang* gibt einen guten Überblick über die Möglichkeiten von Instrumentierungen mit verschiedenen Prüfzielen des Softwaretests. Für die Überwachung der Ausführung von Anweisungen und Zweigen wird zum Beispiel das Einfügen von Zählern an Positionen der Software vorgeschlagen, die Bestandteil von bestimmten Segmenten des zugehörigen Kontrollflußgraphen sind. Diese Messungen geben Auskunft über die ausgeführten Programmteile und können als Testwirksamkeitsmaß verwendet werden [Riedm97].

Aufgrund der Ausführungsabhängigkeiten (*control flow dependence* - Seite 10) zwischen den Anweisungen eines Programms ist es möglich, nur einen Bruchteil der Zweige zu instrumentieren und daraus die Ausführung aller anderen Zweige abzuleiten. [Chusho87] erläutert eine Methode zur Identifikation der sogenannten wesentlichen Zweige (*essential branches*) eines Programms.

Ein weiterer Vorschlag von [Huang79] zur Instrumentierung von Programmen ist das *Datamonitoring*. Dabei handelt es sich um die Protokollierung des Werteverlaufs von Variablen sowie deren Definition und Benutzung zur Laufzeit. Mit sogenannten *Assertions* können bestimmte Bedingungen, zum Beispiel eine Relation zwischen zwei Variablen, während des Programmablaufs überprüft werden.

[Riedm97] schlägt für die Kontrolle der Bedingungsüberdeckung eine Messung durch Funktionsaufrufe an Stelle der Verzweigungsbedingungen vor. Diese Funktionsaufrufe protokollieren den Wert (*wahr* oder *falsch*) einer Bedingung und können so eine Testwirksamkeit aufzeichnen. Ein ähnlicher Ansatz wurde für *ASTELA* realisiert.

Eine Messung der Wirksamkeit für die Segmentfolgenüberdeckung und die verschiedenen Datenflußüberdeckungskriterien gestaltet sich schwieriger. Die Überwachung für solche Tests ist umfangreicher und kann nicht im Rahmen einer Instrumentierung erfolgen. Das hier vorgestellte Verfahren unter Verwendung des Erreichbarkeitsgraphen stellt einen Weg vor, bei dem erst nach der Ausführung des Testobjekts das Testkriterium geprüft wird. Diese Methode stützt sich auf die Protokollinformationen der Instrumentierung der getesteten Software.

Instrumentierung für automatische Strukturtests

Die für das System ASTELA entwickelte Instrumentierung dient der Berechnung der Zielfunktion für das gewählte Testkriterium. Eine Instrumentierung der Software erfolgt unabhängig vom Testverfahren. Mit dem Testobjekt können im Anschluß an die Vorbereitung beliebige Strukturtests durchgeführt werden.

Die Berechnung aller vorgestellten Zielfunktionen basiert auf einer Abstandsfunktion in den Verzweigungspunkten der Software (siehe Abschnitt 4.4). Für diese Berechnung sind Messungen an den Verzweigungsbedingungen notwendig. Ein Protokoll der ausgewerteten Relationen der Verzweigungen gibt zudem Aufschluß darüber, welche Zweige

und Anweisungen ausgeführt wurden. Daraus kann der vollständige Kontrollfluß in der getesteten Software bestimmt und so das vom Nutzer gewählte Strukturtestkriterium überprüft werden. Als Meßpunkte werden die atomaren Bedingungen (*Relationen*) der Verzweigungen eines Programms genutzt.

Die technische Umsetzung der Instrumentierung hängt von den Möglichkeiten der Quellsprache der zu testenden Software ab. Auf Grundlage des durch den Parser aufgebauten Objektbaums (siehe *Objekt-Modell* des Parsers) zur analysierten Software können die verzweigenden Kontrollstrukturen eines Programms bestimmt und Transformationen durchgeführt werden.

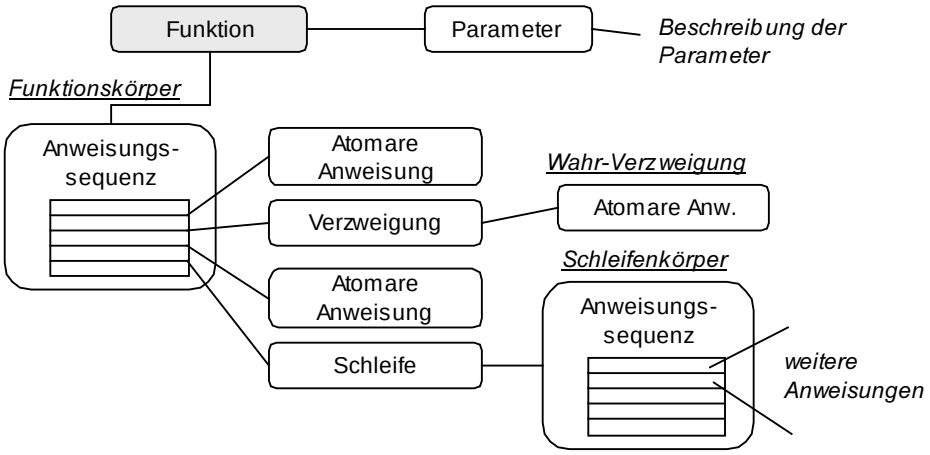


Abbildung 4.50: Objektbaum zu einer Beispielfunktion

Wenn die Programmiersprache eine Implementation von nutzerdefinierten Funktionen ermöglicht, können die Relationen der atomaren Bedingungen durch Funktionsaufrufe ersetzt werden. Diese Bibliotheksfunktionen liefern dieselben boolschen Werte wie die ersetzte Relation und protokollieren bei der Ausführung die Werte der Operanden.

Beispiel:

WENN (A < B und V>0) **DANN** ...

wird transformiert in:

WENN (Less(A,B) und Greater(V,0)) **DANN** ...

Unterstützt die Quellsprache ein solches *Funktionskonzept* nicht, muß die Messung durch Unterprogrammaufrufe vor der Verzweigung ausgeführt werden.

Beispiel:

WENN (A < B und V>0) **DANN** ...

wird transformiert in:

Protokolliere_Kleiner(A,B) → Messung "A<B"
Protokolliere_Groesser(V,0) → Messung "V>0"
WENN (A<B und V>0) **DANN** ...

In jedem Fall ist bei allen Transformationen darauf zu achten, daß keine Seiteneffekte erzeugt oder bestehende Ausführungsabhängigkeiten gewandelt werden und damit das Verhalten des Testobjekts geändert wird.

Bei der Transformation der Relationen in Funktionsaufrufe ist insbesondere für typisierte Sprachen wie *AnsiC* eine Bestimmung der Ergebnistypen der beteiligten Operanden einer Relation notwendig. Entsprechend der an der Relation beteiligten Typen müssen Funktionen mit gleichen Parametertypen eingefügt werden. Bei der Instrumentierung für *AnsiC* wurde das sogenannte *Namemangling* für die Bildung von Funktionsnamen genutzt. Dazu wird der Typ der Operanden im Funktionsnamen kodiert. Eine Instrumentierungsbibliothek stellt Funktionen für alle möglichen Basistypen bereit.

Beispiel:

```
int a,b;
IF (a<b) { ....          wird transformiert:   IF ( LESS_INT(a,b) ) { ....
```

Instrumentierung von Mehrfachverzweigungen

Einige Sprachen, wie zum Beispiel *AnsiC, Pascal* und *Modula,* erlauben die Nutzung von Mehrfachverzweigungen. Diese werden durch Beschreibung der einzelnen Fälle spezifiziert. Für eine Instrumentierung des Quelltextes ergibt sich das Problem, daß die einzelnen Vergleichsoperationen, die erst bei der technischen Umsetzung solcher Konstrukte im Maschinencode erzeugt werden, nicht beeinflußbar sind.

In der Literatur wird für diesen Fall meist eine Transformation in einzelne binäre Verzweigungen vorgeschlagen. Diese Umwandlung des Programmcodes ist im Einzelfall sehr aufwendig, da beispielsweise die Struktur einer Mehrfachverzweigung in *AnsiC* sehr offen gestaltet ist, wie folgendes Quellprogramm zeigt: (*Sprung in eine Schleife*):

```
SWITCH (A)
        case 0: for (i=0;i<10;i++)
        {
                case 3: v=i;
                case 1: i=v;
        };
```

Für *AnsiC* wird bei der Testvorbereitung ein anderer Weg gegangen. Die Idee ist die Konstruktion einer Funktion ähnlich der Instrumentierung von binären Verzweigungen. Genauso wie bei der binären Variante müssen dieser Funktion alle Auswertungsfälle (*Cases*) bekannt sein, um den Abstand zu den zugehörigen Zweigen protokollieren zu können. Im folgenden Beispiel übernimmt diese Aufgabe die Funktion "Eval_SWITCH":

Beispiel:

```
SWITCH (A)  {       /* Originalprogramm */
        case 1:     ...
        case 2:     ...
        case 5:     ...
}
```

Das Beispiel wird transformiert in:

```
struct  Switch_Data_Typ
        data_1 = ( 1, 2, 5 }; /* Datenstruktur für die Auswertungsfälle */
```

```
SWITCH ( EVAL_SWITCH( A , data_1 ) ) {
        case 1:     ...
        case 2:     ...
        case 5:     ...
}
```

Auf Grundlage der Datenstruktur '*data_1*' kann innerhalb der Funktion *‚Eval_Switch‘* geprüft werden, welcher der Fälle ausgeführt wird und wie der Abstand zu allen anderen Verzweigungsentscheidungen ist. Diese Form der Transformation ist einfacher als eine Umwandlung in binäre Verzweigungen.

Zusammenstellung eines Testprotokolls

Die Funktionen der Instrumentierung sind für die Zusammenstellung eines Testprotokolls verantwortlich. Eine Ausgabe des Protokolls erfolgt durch den zu generierenden Testtreiber. Den Inhalt des Testprotokolls bildet die Abfolge der Messungen während der Ausführung des Programms. Ein Protokolleintrag muß deshalb neben dem Meßergebnis die Position der Messung eindeutig festhalten, so daß zu einem späteren Zeitpunkt Verzweigungsentscheidungen nachvollzogen und der Programmablauf rekonstruiert werden können. Die Meßposition wird als Parameter der eingefügten Funktionen übergeben. Der Instrumentierer von ASTELA ordnet jeder Verzweigungsbedingungen eindeutige Nummern zu, die auch bei der Auswertung des Testprotokolls benutzt werden.

Beispiel – Identifikationsnummern:

```
IF ( LESS_INT ( 101 , A , B ) and GRT_LONG( 220, V, 0 ) )
{
        IF (EQU_CHAR( 224 , D , 10 ) ) ...
```

Zugehörige Strukturinformationen:

```
KNOTEN 1:  VERZWEIGUNG mit Bedingung "100"   (siehe unten)
KNOTEN 2:  VERZWEIGUNG mit Bedingung "224"   (siehe unten)
BEDINGUNG 100: UND ( Bedingung 101, Bedingung 220 )
BEDINGUNG 101: "A<B"
BEDINGUNG 220: "V>0"
BEDINGUNG 224: "D==0"
```

Mit Hilfe der Strukturinformationen können aus einem Protokoll, das nur die Identifikationsnummern und gemessenen Werte enthält, die Verzweigungsentscheidungen berechnet werden. Aus dem Protokoll "101: 34,20" ist zum Beispiel ersichtlich, daß von Knoten 1 der *‚falsch‘*-Zweig ausgeführt wird ("34 < 20" ergibt falsch).

Bei der Umsetzung der Instrumentierung in *AnsiC* wurden auf Grund der engen Zusammenarbeit von Testtreiber und Instrumentierung Funktionen in einer Testtreiberbibliothek integriert. Diese Bibliothek wird zusammen mit dem transformierten Quelltext der zu testenden Software compiliert.

4.6.3 Testtreibergenerator

Der Testtreibergenerator hat die Aufgabe für die zu testende Software einen Testrahmen zu erzeugen. Dieser Testrahmen (*Testtreiber, Aufrufschnittstelle*) verbindet die Teststeuerung und die Funktionen des *Testobjekts*. Er ruft das Testobjekt mit den generierten Testdaten auf und gibt für jedes Datum ein Protokoll des Testverlaufs an die Teststeuerung zurück.

Die Parameter eines Testobjekts können beliebige Datenstrukturen, wie dynamische Listen, Bäume oder zusammengesetzte Typen, sein. Neben den Parametern einer Funktion können globale Variablen Eingaben eines Testobjekts sein, wenn sie innerhalb der Funktion benutzt werden.

Der Testtreiber muß dynamische, komplexe und nutzerdefinierte Datenstrukturen erzeugen und mit den Testdaten ausfüllen können. Diese Testdaten werden von der Teststeuerung als Zahlenvektor übergeben. Der Aufbau des Vektors ist bei der Treibergenerierung zu bestimmen. Dabei stehen dem Testtreibergenerator Typinformationen zu den Eingabeparametern eines Testobjekts zur Verfügung. Die Objekte des *Datenstrukturkonzepts* (siehe Parser Teilabschnitt 4.6.1) beschreiben die Zusammensetzungen der benutzten Datenstrukturen. Auf dieser Grundlage legt der Testtreibergenerator die Reihenfolge der Testdateneinträge fest und erzeugt Programmcode, der die Datenstrukturen dieser Abfolge einliest.

Das Vorgehen der Testtreibergenerierung kann in drei Schritte gegliedert werden:

1. Bestimmung der Eingabeparameter (einzulesende Variablen des Testobjekts); Dabei werden Zugriffe auf globale Variablen kontrolliert.
2. Bestimmung des Aufbaus der zugehörigen Datenstrukturen und Festlegung der Reihenfolge beim Einlesen der Daten
3. Erzeugung von Programmcode zum Einlesen der Datenstrukturen (Konvertierung des Zahlenvektors)

Die Datenstrukturen der Eingabeparameter eines Testobjekt können geschachtelt, als Sequenzen oder Bäume organisiert sein. Es ist deshalb vorteilhaft, für jede Datenstruktur separate Programmteile zu generieren. Auf diese Weise können Sequenzen eines Datentyps durch wiederholten Aufruf der zugehörigen Funktion eingelesen werden. Diese Organisationsform wird im folgenden Absatz beschrieben.

Hierarchische Organisation der Konvertierung

Für jeden Datentyp wird eine eigenständige Funktion erzeugt. Das Einlesen enthaltener Substrukturen wird durch Funktionsaufrufe realisiert. Für die sogenannten Build-In-Types der Quellsprache stehen im Rahmen der Testtreiberbibliothek vorgefertigte Funktionen zur Verfügung. Der Programmcode für einen nutzerdefinierten Datentyp gestaltet sich daher rekursiv. Jede zusammengesetzte Datenstruktur basiert auf Unterstrukturen und kann deshalb durch eine Sequenz von Aufrufen der zugehörigen Konvertierungsfunktion eingelesen werden.

Bei den typenvariablen Strukturen (*Unions*), Listen, dynamischen Arrays, Bäumen und Pointerstrukturen kann die Festlegung einer Konvertierung nur mit Hilfe zusätzlicher Angaben zu den Testbeschränkungen erfolgen. Deshalb wurde die Testtreibergenerierung für dynamische Datenstrukturen bei der Entwicklung des Systems ASTELA vorerst nicht realisiert.

Für dynamische Datentypen müssen die Grenzen der Variabilität festgelegt werden, da bei evolutionären Tests von einem festen Zahlenvektor als Eingabe ausgegangen wird. Die Testvorgaben dürfen jedoch nicht einen Aufbau für solche Strukturen fixieren. Vielmehr gilt es, die oberen Schranken für den Test zu vereinbaren. Eine Variabilität der dynamischen Strukturen kann durch eine entsprechende Kodierung umgesetzt werden. Mehr zu den möglichen Erweiterungen des Systems bei der Testtreibergenerierung beschreibt der Ausblick im Teilabschnitt 7.3.

Beschreibung der Reihenfolge der Testdaten für die Steuerung

Sowohl die Teststeuerung als auch die evolutionären Algorithmen behandeln die Eingaben eines Testobjekts als Zahlenvektor. Der Aufbau eines solchen Vektors richtet sich nach den Datentypen der Eingabeschnittstelle des Testobjekts und den zu generierenden Konvertierungsfunktionen. Letztere legen die Reihenfolge fest, in der die Elemente der Datenstrukturen aus dem Zahlenvektor entnommen werden.

Aus der Konvertierungsreihenfolge resultieren die Datentypen und Wertebereiche der einzelnen Vektorelemente. Diese Typinformationen muß der Testtreibergenerator der Teststeuerung mitteilen, damit nur solche Testdaten erzeugt werden, die den Wertebereichen der zugehörigen Elemente der Datenstrukturen entsprechen.

Erwartet ein Testobjekt zum Beispiel drei Parameter mit den Typen Fließkomma, Integer 16bit und Integer 8bit, sind nur Testdaten mit diesem Format zu erzeugen. Der Testtreibergenerator bestimmt anhand der Lesereihenfolge der Eingabeparameter die Sequenz der einzulesenden *Build-In*-Typen und übergibt diesen Vektor zusammen mit den anderen Strukturinformationen an die Teststeuerung. Aufgabe der Teststeuerung ist die Weitergabe der Datentypinformationen an die evolutionären Algorithmen.

5 Erste Experimente

5.1 Strukturtests der Beispielapplikationen

Die folgenden Teilabschnitte erläutern erste Testergebnisse mit Beispielapplikationen. Im Rahmen der hier beschriebenen ersten Experimente konnten nur Zweig- und Anweisungsüberdeckungstests durchgeführt werden. Die Teilabschnitte erläutern im einzelnen den Kontrollflußgraphen der Testobjekte sowie das Optimierungsverhalten in verschiedenen Testsituationen.

5.1.1 Beispiel 1 – Dreiecktypbestimmung

Beschreibung der Applikation

Das Testobjekt '*Dreiecktyp*' hat drei Eingangsparameter, welche die Seitenlängen eines Dreiecks bezeichnen. Durch das Programm wird der Typ des gebildeten Dreiecks bestimmt. Dazu wird in den ersten fünf Bedingungen die Gültigkeit der Angaben, also der Wertebereich der Parameter, sowie das Zutreffen der Dreiecksungleichungen geprüft. Ist eine der Forderungen nicht erfüllt, wird die Applikation verlassen, da kein Dreieck vorliegt.

Den nächsten Teil des *'Dreiecktyp'*-Programms bildet die Sortierung der Seitenlängen für die spätere Typbestimmung. Die Werte werden so getauscht, daß die Variable 'c' den größten und 'a' den kleinsten Wert erhält. Danach wird durch den Vergleich der Seitenlängen bestimmt, ob das spezifizierte Dreieck gleichseitig oder gleichschenklig ist. Den Abschluß des Programms bildet die Prüfung der Eigenschaften von rechtwinkligen und stumpfwinkligen Dreiecken.

Der Kontrollflußgraph

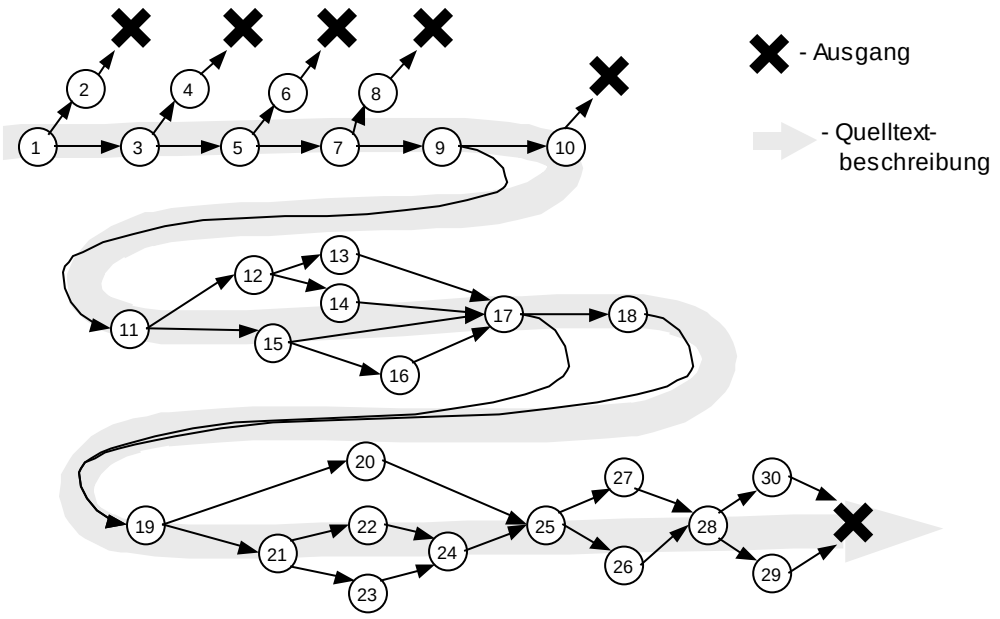


Abbildung 5.1: Kontrollflußgraph für Dreiecktypbestimmung

Die Verzweigungsbedingungen

Die folgenden Bedingungen werden für die Entscheidungen in der verzweigenden Knoten herangezogen:

Knoten	Bedingung	Knoten	Bedingung
1	a≤0 oder b≤0 oder c≤0	**15**	c<a
3	a>1000 oder b>1000 oder c>1000	**17**	a>b
5	c>=a+b	**19**	a=b und b=c
7	a>=b+c	**21**	a=c oder b=c oder a=b
9	b>=a+c	**25**	$a^2 + b^2 = c^2$
11	c<b	**28**	$a^2 + b^2 < c^2$
12	b<a		

Tabelle 5.1: Verzweigungsbedingungen 'Dreieckstypbestimmung'

Optimierungsbeispiel Zweigüberdeckung - Zielzweig 21→22

Tabelle 5.2 zeigt eine beispielhafte Optimierung des Zielzweigs 21→22. Die Spalten bezeichnen der Reihe nach die Generationsnummer, den Zielfunktionswert, die Variablen des besten Individuums, den durchschnittlichen Zielfunktionswert aller Individuen der Generation und in der letzten Spalte die Bewertung des schlechtesten Individuums.

Gen.	bestes Indiv.	a	b	c	∅	schlechtestes Indiv.
1	5.6543	217	19600	8596	6,52403	6,98276
3	5,6193	589	16486	13387	6,09352	6,8852
5	4,0644	2700	407	3773	5,70983	6,81122
7	4,0339	2067	1337	3749	5,38861	6,77931
9	0,20584	4123	4331	8271	4,32345	6,75542
11	0,010346	1959	2063	3836	3,27041	6,78622
13	0,0092564	4426	4333	7310	1,91663	6,79899
15	0,0043901	2098	3380	2142	2,12437	6,79386
17	0,0015986	5403	5387	8352	2,40233	6,81312
19	0,00099945	4585	4575	7387	1,49052	6,83973
21	0,00099945	4585	4575	7387	1,62981	6,81332
23	0,00099945	4585	4575	7387	1,5642	6,79959
25	0,00099945	4585	4575	7387	1,47974	6,78836
27	0,00099945	4585	4575	7387	2,04908	6,8094
29	0,00099945	4585	4575	7387	1,10486	6,0515
31	0,00009999	4585	4584	6147	1,07859	6,79363
34	0	4574	4574	7526	0,97284	6,53296

Tabelle 5.2: Beispieloptimierung für Zweig 21→22

Aus dem Kontrollflußgraphen in Abbildung 5.1 ist zu entnehmen, daß die Vorbedingungen, das sind die Verzweigungen in den Knoten 1, 3, 5, 7, 9, 19, mit *falsch*

ausgewertet werden müssen, damit der Knoten 21 erreicht werden kann. Daraus ergeben sich sechs Annäherungsstufen für Knoten 21.

In der Beispieloptimierung von Tabelle 5.2 wird deutlich, daß die besten Individuen der 1. bis 4. Generation in Knoten 3 auf Annäherungsstufe 1 mit den Abständen 0,34 und 0,38 scheitern [*Zielfunktionswert*: 1 + 6 - (1+0.34)]. Bei der weiteren Optimierung verfehlt das jeweils beste Individuum der 5. bis 7. Generation den Zielzweig in der Verzweigung 5 (*Annäherungsstufe 2*). Dort gilt es, die Bedingung "c>a+b" mit *falsch* auszuwerten [ZF = 1 + 6 - (2+0.94)]. Die besten Testdaten der nächsten Generationen (>8) erreichen den Knoten 21 (*Annäherungsstufe 6*), werten die Verzweigungsbedingung jedoch mit Falsch aus. Sie verfehlen damit das Teilziel '21→22'. In der 34. Generation wird ein Individuum gefunden, das den gewünschten Zweig durchläuft.

Das Testdatum lautet: (4574, 4574, 7526)

Optimierung aller Teilziele

Die zuvor beschriebene Optimierung wurde zu Demonstrationzwecken mit einer sehr kleinen Population und ohne Verwendung von Startwerten ausgeführt. Bei der Optimierung aller Teilziele unter Nutzung von zufällig gefundenen Testdaten als Ausgangswerte werden die Ergebnisse gewöhnlich schneller gefunden, weil die vorliegenden Testergebnisse wiederverwendet werden und damit viele Individuen bereits innerhalb der ersten Generationen sehr viel näher am Optimum getestet werden, als eine zufällig generierte Startpopulation. Eine Optimierung ohne Einstreuung von gut bewerteten Testdaten benötigt einige Generationen mehr, um etwa auf den gleichen Stand zu gelangen, wie eine Population bestehend aus Startwerten.

Dieses Verhalten läßt sich gut anhand der durchschnittlichen Bewertung der Individuen einer Generation erkennen (*vergleiche dazu Spalte 6 – Beispieloptimierung ohne Startwerte in Tabelle 5.2 und die folgende Tabelle)*

Gen.	Bestes Indiv.	A	b	c	∅	schlechtestes Indiv.
Start	0.01365	358	464	428		
1	0.01055	176	114	119	0.82278	3.05353
2	0.01039	733	736	989	2.90738	6.50255
3	0.01208	943	813	833	3.13159	6.50255
4	0.0116815	556	572	120	1.13706	3.03502
5	0.0107916	727	720	986	2.51687	5.50257
6	0.0111872	376	643	365	0.69118	4.03319
7	0.0103959	656	774	771	0.572794	3.01375
8	0	735	735	840	0.237755	2.02464

Tabelle 5.3: Optimierung mit Startwerten

Während sich die ersten Generationen bei der Testdatensuche ohne Startwerte noch sehr weit von Optimum entfernt bewegen, hat die Optimierung mit Startwerten einen Vorsprung und erreicht schneller das Ziel.

Für den Zweigüberdeckungstest des *‚Dreiecktyp‘* – Programms ergaben sich häufig vier Optimierungsläufe. Der erste Lauf erzeugt eine zufällige Population für den Zweig 1→2. Alle weiteren Optimierungen werden für noch nicht durchlaufene Zweige gestartet. In den meisten Fällen sind das die Zweige 21→22 (*zwei Seiten gleich lang*), 19→20 (*alle Seiten gleich lang*) und 25→26 (*Bedingung* $a^2+b^2=c^2$). Testdaten für die anderen Zweige wurden bei allen Testläufen zufällig gefunden.

Die GEA-Toolbox wurde für die Verwendung von sieben Populationen mit je 20 Individuen konfiguriert. In den Einstellungen unterscheiden sich die einzelnen Subpopulationen in der Mutationsschrittweite, also in der Feinheit der Suche. Es folgt die Analyse eines beispielhaften Laufs der Testdatengenerierung für die Zweigüberdeckung.

1. Lauf (Ziel: Ausführung des Zweigs 1→2)

Im ersten Testlauf wird die Ausführung des Zweigs 1→2 angestrebt. Da keine Startwerte vorliegen, wird die erste Generation vollständig zufällig erzeugt. Einige der generierten Testdaten lagen damit schon sehr nahe am auszuführenden Zweig und so konnte in der zweiten Generation ein Testdatum gefunden werden. Die Ausführung des Testobjekts mit den generierten Daten ergab nach diesen zwei Generationen mit zusammen 266 Individuen die Ausführung von sehr vielen anderen Zweigen des Testobjekts. Das Ergebnis[1]:

(905.527 566.656 -1) : 1→2, 2 → Exit
(273.38 382.613 829.253) : 1→3 , 3→5 , 5→6, 6→ Exit
(563.088 37.7482 332.908) : 5→7, 7→8 , 8→ Exit
(4.52041 593.771 440.585) : 7→9, 9→10, 10→Exit
(361.728 442.284 595.582) : 9→11, 11→15, 15→17, 17→19, 19→21, 21→23, 23→24, 24→25, 25→27, 27→28, 28→30, 30→Exit
(457.579 373.215 430.017) : 15→16, 16→17
(848.352 713.607 305.268) : 11→12 , 12→13, 13→17
(805.129 981.539 490.217) : 12→14, 14→17, 17→18, 18→19
(713.277 898.636 743.736) : 28→29, 29→Exit

35 von 43 Zweigen ausgeführt: 81% Zweigüberdeckung

Nach diesem Lauf fehlen noch Testdaten für die Ausführung von acht Zweigen: 3→4, 4→Exit, 19→20, 20→25, 21→22, 22→24, 25→26 und 26→Exit. Die beste Annäherung an den Zweig 21→22 wird mit dem Testdatum (194.146 134.265 134.143) erreicht. Für diesen Zweig wird im zweiten Optimierungslauf ein Testdatum gesucht.

2. Lauf (Ziel: Ausführung des Zweigs 21→22)

Für die Optimierung dieses Ziels wurden 23 Generationen mit insgesamt 2912 Individuen erzeugt, bis ein Testdatum gefunden wurde, welches den Zweig durchläuft. Das gefundene Testdatum lautet (195.161; 135.397; 135.397). Mit dieser Eingabe wurden sowohl der Zweig 21→22 wie auch 22→24 ausgeführt.

[1] Die Testeingaben werden als Vektor (x; y; z) oder (x y z) angegeben. Bei der Zahlendarstellung ist „21.2“ als „21 Komma 2“ zu lesen.

Neben der Optimierung des aktuellen Teilziels konnten zufällig Testdaten für zwei weitere Zweige gefunden werden. Mit dem erzeugten Individuum „*323.867; 777.181; 1000.34*“ werden die Zweige 3→4 und 4→Exit ausgeführt. Nach diesem Lauf wurde eine Zweigüberdeckung von 91% erreicht.

Der beste zufällig gefundene Startwert für ein offenes Teilziel ist das Testdatum (195.161; 135.397; 135.397) zum Zweig 19→20. Mit diesem und einigen anderen gut bewerteten Eingaben initialisiert die Teststeuerung die dritte Optimierung.

3. Lauf (Ziel: Ausführung des Zweigs 19→20)

Dieser Lauf erforderte mit 89 Generationen die höchste Anzahl von Optimierungsschritten. Es mußte ein Testdatum gefunden werden, bei dem alle drei Variablen den gleichen Wert besitzen. Die Verzweigungsbedingung in Knoten 19 besteht aus zwei Teilbedingungen, welche mit UND verknüpft sind. Der verwendete Compiler optimiert die Auswertung dieser Bedingung, das bedeutet, daß die zweite Teilbedingung nur ausgeführt wird, wenn die erste den Wert *wahr* ergibt.

Die große Anzahl von Generationen resultiert aus vielen erzeugten Individuen, die schon die erste Teilbedingung nicht erfüllen und damit eine sehr schlechte Bewertung erhalten. Trotz dieses Bewertungsproblems (siehe Diskussion im Abschnitt 6.1) konnte ein Testdatum auch für dieses Teilziel gefunden werden. Mit der Eingabe (143.965; 143.965; 143.965) führt das Testobjekt die Zweige 19→20 und 20→25 aus.

Der vierte und letzte Lauf optimiert Testdaten für die Ausführung des Zweigs 25→26. Als vielversprechendes Testdatum wurden die Eingabe (91.3349; 200.286; 178.264) und einige weitere Daten in der näheren Umgebung dieser Werte benutzt.

4. Lauf (Ziel: Ausführung des Zweigs 25→26)

Für die Optimierung der Testdaten dieses Teilziels wurden 46 Generationen mit zusammen 5810 Individuen gebildet. Bei einem Aufruf des Testobjekts mit der Eingabe (224.894 269.112 147.797) werden die Zweige 25→26 und 26→28 ausgeführt.

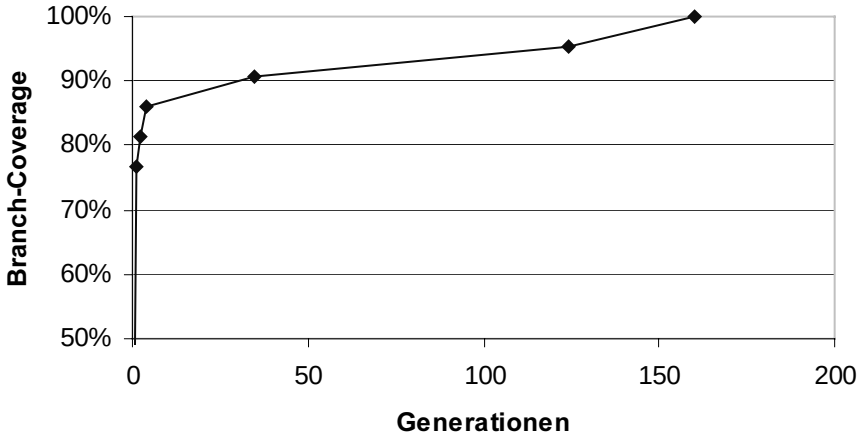
Damit ist die 100% Zweigüberdeckung erreicht. Die gesamte Optimierung benötigte 160 Generationen mit insgesamt 20216 erzeugten Individuen. Es wurden Testdaten zur Ausführung aller Zweige des Testobjekts gefunden.

Ergebnis:

(4.52041	593.771	440.585)	(143.965	143.965	143.965)
(195.161	135.397	135.397)	(224.894	269.112	147.797)
(273.38	382.613	829.253)	(323.867	777.181	1000.34)
(361.728	442.284	595.582)	(457.579	373.215	430.017)
(563.088	37.7482	332.908)	(713.277	898.636	743.736)
(805.129	981.539	490.217)	(848.352	713.607	305.268)
(905.527	566.656	-1.0000)			

100 % branch coverage 100 % statement coverage

Überdeckungsgrad im Testverlauf



5.1.2 Beispiel 2 – Komplexe Verzweigungshierarchie

Dieses Testobjekt wurde für die Entwicklung und den Test der Komponenten des Systems ASTELA eingesetzt. Es sind Mehrfachverzweigungen, Verzweigungen aus Schleifen und Verzweigungshierarchien enthalten.

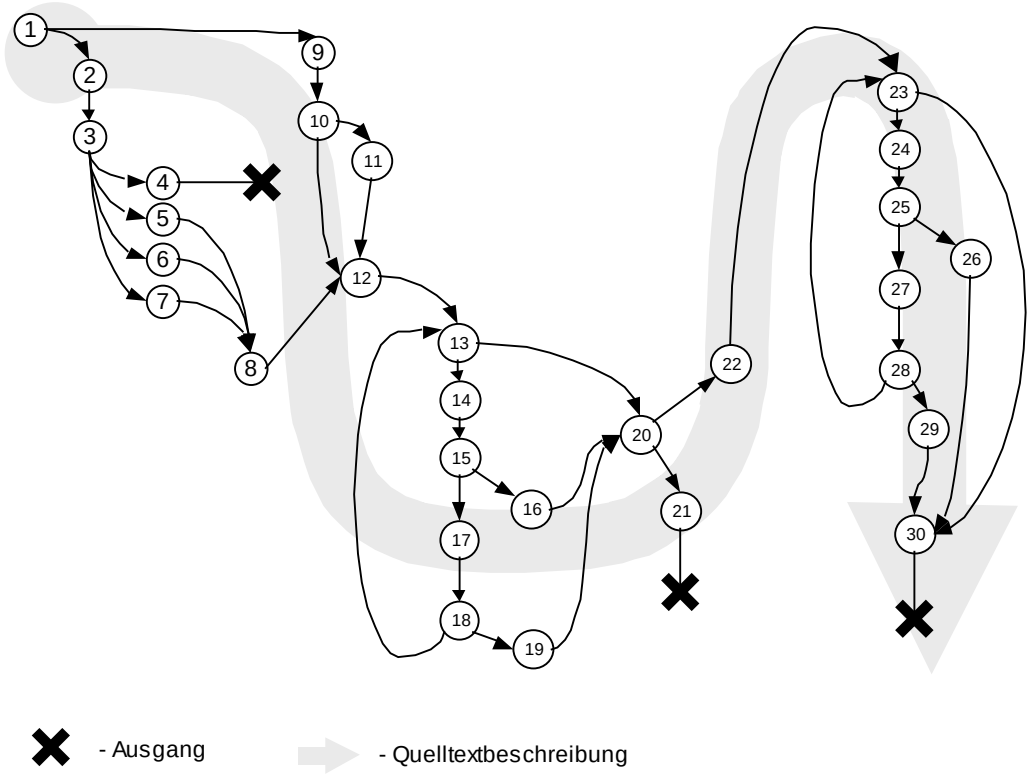


Abbildung 5.2: Kontrollflußgraph komplexe Verzweigungshierarchie

Die Verzweigungsbedingungen

Knoten	Bedingung	Knoten	Bedingung
1	a < e oder e+a ≠ b+c oder e+c < a+b+d	**18**	f=a
3	a+b : case "=0" ; case "=10" ; case "=23"	**20**	(a<b oder b<c) und (a<c oder e<d)
10	c<b oder (a>b und e<d) oder a=e	**23**	e+f<a+b und z<100
13	f<d und z<100	**25**	e<f
15	e<f	**28**	f=a

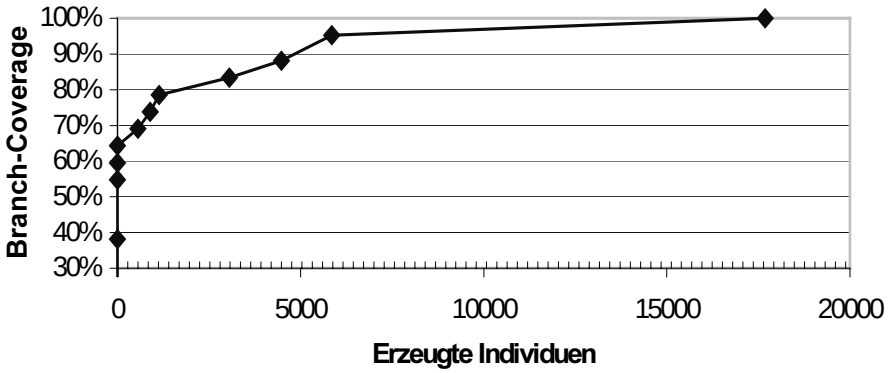
Automatischer Zweigüberdeckungstest

Für die Untersuchung des Strukturtests dieser Funktion waren insbesondere die Zweige 10→12 (2 Level), 18→19 (3 Level; Schleife) und 28→29 (4 Level; Schleife) von Interesse.

Verschiedene Testläufe mit ASTELA haben gezeigt, daß nicht nur die Verzweigungen am Ende des Programms schwer optimierbar sein können. Anders als erwartet wird zum Beispiel für die Verzweigung in Knoten 28 sehr schnell ein Testdatum gefunden. Die Suche einer Eingabe für die Verzweigung in Knoten 10 schreitet dagegen im Vergleich zu den anderen Optimierungen langsam voran. Als problematisch hat sich bei der Optimierung dieses Beispiels die Abstandsfunktion an den Typgrenzen erwiesen. Bei der Formel "e+a = c+b" in Verzweigungsknoten 1 wird mit 16bit Werten gerechnet. Sind die Werte entsprechend groß, erzeugt die Summe aus zwei positiven Werten ein negatives Ergebnis. An solchen Grenzwerten entstehen lokale Optima, die im Wertebereich der Variablen sehr weit vom globalen Optimum entfernt sind. Der Gesamtprozeß mußte hier häufig bei minimalen Annäherungsschritten nach 1000 Generationen abgebrochen werden.

Wurde der Wertebereich der Variablen soweit eingeschränkt, daß Zahlenbereichsgrenzen den Test nicht beeinflussen, konnte die Zweigüberdeckung innerhalb von 50 bis 250 Generationen mit je 100 Individuen gefunden werden.

Das folgende Diagramm zeigt ein Beispiel für eine Optimierung. Der Testlauf benötigte zur Bestimmung von Eingaben für alle Teilziele 177 Generationen.



Mit der folgenden Eingabemenge wird zum Beispiel 100-prozentige Zweigüberdeckung erreicht:

3906	8561	-8060	6782	-2608	9398
5014	4228	5725	1188	4939	-2069
-3944	3943	-8612	4925	-3361	4318
-3901	3910	3344	1618	-8589	6606
2277	-2255	-6234	-2617	5563	6945
4974	4219	5729	1184	4974	-7808
-3661	7703	7778	8182	6116	6962
-4462	3067	-2600	-1925	-739	-8243
-5131	-2411	4268	6865	3492	-5361
9409	4149	-4021	-9048	-3387	-9002
4930	3591	6251	-2421	5300	3548

5.1.3 Beispiel 3 - Line Covered by Rectangle

Das Testobjekt bestimmt für eine Eingabe, das sind die Koordinaten einer Linie und eines Rechtecks, ob ein Teil der Linie im Rechteck enthalten ist oder die Linie vollständig außerhalb liegt.

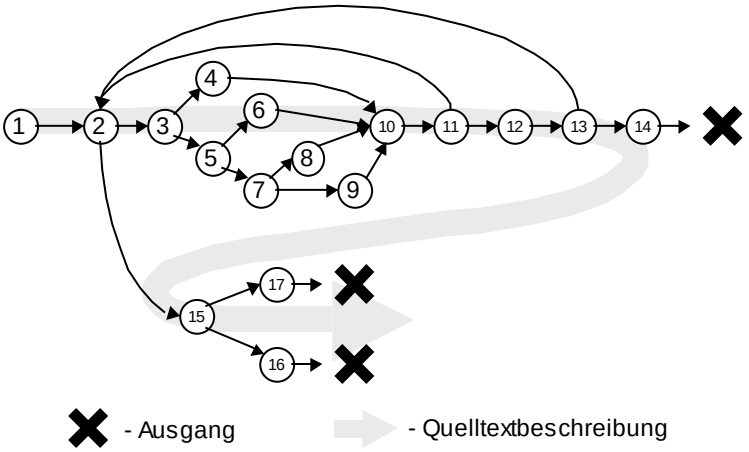


Abbildung 5.3:Kontrollflußgraph zum Line-Covered-by-Rectangle

Die Verzweigungsbedingungen

Knoten	Bedingung	Knoten	Bedingung
2	i ≤ 3	**11**	det != 0
3	i = 0	**13**	m ≥ 0 und m ≤ 1 und n ≥ 0 und n ≤ 1
5	i = 1	**15**	rec.p1.x ≤ line.p1.x und line.p2.x ≤ rec.p2.x und
7	i = 2		rec.p1.y ≤ line.p1.y und rec.p1.y ≤ line.p1.y

Ergebnisse

Aus den zufällig generierten Eingaben resultierte zumeist eine hohe Zweigüberdeckung. Dieses Verhalten ergibt sich aus der Struktur des Testobjektes, da bereits eine Eingabe genügt um 90 % der Zweige auszuführen und diese Eingabe (Koordinaten einer Linie, die außerhalb des Rechtecks liegen) sehr leicht zu erzeugen ist. Optimierungen fanden bei wenigen Testläufen nur für die Zweige 15→16 und 11→ 12 statt.

Bei diesem Testobjekt konnte auf Grund der geringen Komplexität mit einer sehr kleinen Populationsgröße (5-10 Individuen) gearbeitet und trotz dessen mit wenigen Generationen Optimierungserfolge erzielt werden. Bei zehn Individuen pro Generation genügten zumeist bereits 4 bis 5 Generationen, bis Zweig 11→12 durchlaufen wurde.

Für Zweig 15→16, der nur ausgeführt wird, wenn sich die Linie mit keiner Seite des Rechtecks schneidet (dieser Test findet innerhalb der Schleife statt), wurden im ungünstigsten Fall 20 Generationen benötigt. Als Startwert für die Optimierung diente ein Testfall, bei dem die Linie außerhalb des Rechtecks lag. Dieser Testfall wurde zufällig ermittelt und erhielt eine gute Bewertung, weil Knoten 15 ausgeführt werden konnte. Mit diesem Startwert erzeugten die evolutionären Algorithmen neue Testdaten, bei denen sich der Abstand von Linie und Rechteck verringerte. Daraus resultieren bessere Abstandswerte für die Teilbedingungen in Knoten 15. Der abschließende Optimierungserfolg entstand durch eine Mutation, bei der die Größe des Rechtecks so verändert wurde, daß die Linie in das Innere des Rechtecks gelangte. Die folgende Abbildung zeigt die Optimierungsfortschritte:

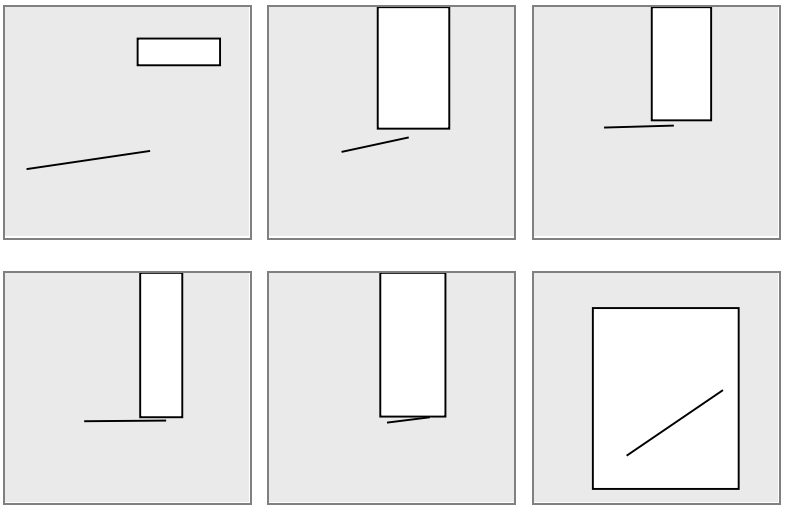


Abbildung 5.4: Beispiel eines Optimierungsprozesses

Gene-ration	echteck x	Rechteck y	Höhe	Breite	X1	Y1	X2	Y2	Ziel-funktion
1	176	41	35	109	192	190	31	214	0,011
5	144	0	160	94	172	184	98	153	0,0057
8	157	0	149	78	185	156	95	153	0,0054
12	180	0	190	55	213	195	107	194	0,0053
15	148	0	189	85	212	191	158	198	0,0052
21	**79**	**46**	**239**	**192**	**250**	**241**	**125**	**155**	**0**

Mit folgender Eingabemenge wird 100-prozentige Zweigüberdeckung erreicht:

70	212	65	203	247	154	60	89
250	66	82	51	5	142	5	222
193	183	147	43	4	233	237	217
79	46	239	192	250	241	125	155
0	81	95	232	33	97	46	231

5.1.4 Beispiel 4 - Netflow

Das Testobjekt ist ein Algorithmus zur Bestimmung der besten Flußverteilung (*flow pattern*) in einem Netzwerk. Die Flußkapazitäten zwischen den Knoten sind als Ober- und Untergrenzen parameterisiert. Als Eingabe werden die Vernetzung der Knoten, die Kostenmatrix, die Ober-/Untergrenzen der Netzflußkapazitäten und Startwerte für die Flußoptimierung übergeben. Der Algorithmus wurde für ein beliebiges Netzwerk konzipiert. Für den Test mit ASTELA erfolgte eine Einschränkung auf 6 Knoten und 11 Zweige.

Erste Tests mit diesem Beispiel haben gezeigt, wie wichtig die Überwachung aller Teilziele für die Bestimmung der Zweigüberdeckung ist. Viele der Zweige konnten bereits mit den ersten zehn Testfällen ausgeführt werden. Auf diesem Weg wurde sehr schnell eine Zweigüberdeckung von 90% erreicht. Dieses Verhalten liegt an der Struktur des Testobjekts. Es enthält hauptsächlich Schleifenstrukturen mit Verzweigungen, welche je nach Iteration verschieden ausgewertet werden. Bei einigen Testeingaben wird während der Programmausführung bereits ein Großteil der Zweige durchlaufen.

Die folgende Abbildung stellt den Kontrollflußgraphen zum Testobjekt dar. In der Grafik sind des weiteren die wichtigsten Schleifen gekennzeichnet.

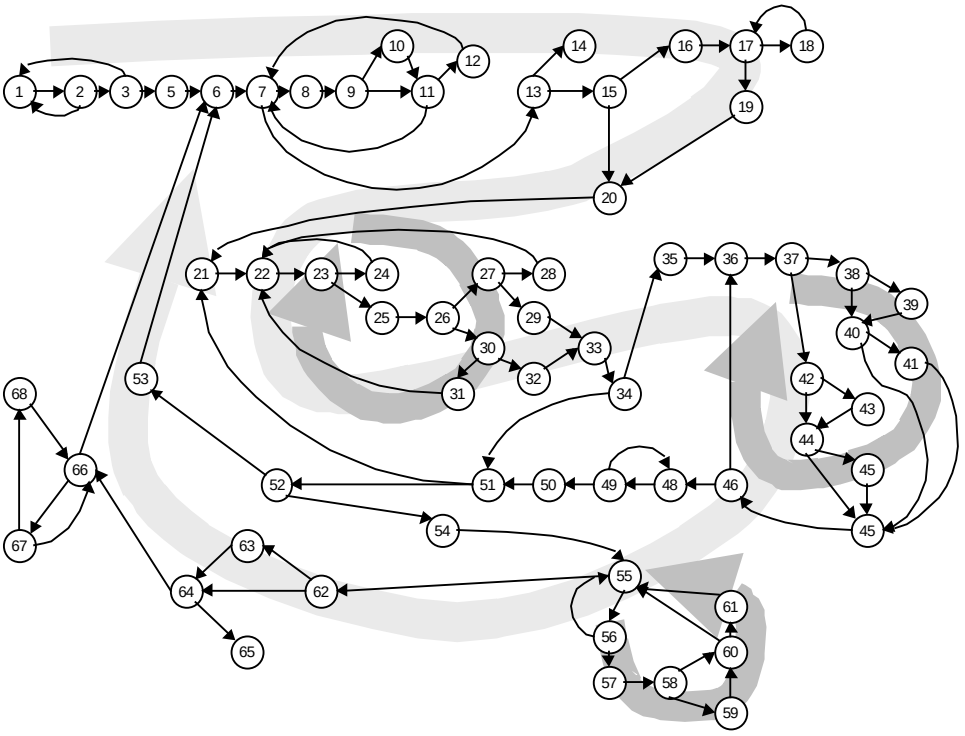


Abbildung 5.5: Kontrollflußgraph zum Netflow-Algorithmus

Die Analyse der nicht ausgeführten Zweige zeigt, daß genau vier Verzweigungen für die restlichen 10% der Überdeckung verantwortlich sind. Das sind die Verzweigungen in Knoten 7, 13, 42 und 62. Dabei stehen die Zweige 7→13, 13→14, 14→Exit in

direktem Zusammenhang, daß heißt ein Kontrollfluß über Zweig 7→13 führt durch die anderen beiden Zweige.

Eine Eingabe für die Verzweigung in Knoten 62 zur Anweisung in 63 konnte in den meisten Fällen nach wenigen Generationen gefunden werden. Jedoch war es der Teststeuerung in keinem der durchgeführten Testläufe möglich, eine Eingabe für den Zweig 42→43 zu finden. Wie eine nähere Analyse des Testobjekts zeigt, ist der Programmcode nicht ausführbar.

Durch die Berechnung der Variable 'c' in Knoten 36 und der Vorbedingung 'nb[ni]<0' für die Ausführung von Knoten 42, ist der Wert 'c' in Knoten 42 in jedem Fall größer Null. Damit ist der erste Teil der Bedingung immer falsch. Es kann kein Testdatum gefunden werden. Die beste Annäherung erzeugt 'c' gleich 0.

Bemerkungen zum Testergebnis[2]

Netflow ist ein iterativer Optimierungsalgorithmus. Die *Netflow*-Lösung wird mit Hilfe der Schleifen im Programm angenähert. Durch die Wiederholung der Schleifen wird ein großer Teil der Zweige durch die meisten Testeingaben ausgeführt. Es müssen nur Eingaben für einige spezielle Fälle, wie zum Beispiel dem Optimierungsabbruch in Knoten 13, oder für spezielle Optimierungssituationen (*Verzweigung in Knoten 62*) gesucht werden. Aus diesem Grund genügt eine sehr kleine Menge von Testdaten, um eine große Überdeckung zu erreichen.

Bemerkung zum Test von Netflow

Der Algorithmus von Netflow ist sehr offen gestaltet. Die Eingabeschnittstelle umfaßt die Beschreibung der Netzstruktur und der Flußkosten, die Begrenzung des Flusses zwischen den Netzknoten sowie Startwerte für den Fluß durch die Knoten. Anhand dieser Daten wird eine iterative Optimierung durchgeführt. Im vorliegenden Programm wurde die Netzgröße, daß heißt die Anzahl der Knoten und Zweige, vor der Instrumentierung im Quellprogramm fest eingestellt. Der Grund dafür liegt in den eingeschränkten Möglichkeiten der Testtreibergenerierung im Testsystem.

5.2 Visualisierung einer Zielfunktion

Für die Verdeutlichung des Optimierungsproblems zur Testdatengenerierung eines knotenorientierten Teilziels zeigt die Abbildung 5.6 zu einem Testobjekt (siehe folgender Quelltext in *AnsiC*) eine 3-dimensionale Ansicht der Zielfunktion in der Nähe des Optimums.

```
        Void fct(int x, int y)
        {
1:              if (x<-0.3 || y<-0.3)
2:                      return;
3:              if (x>10   || y>10)
4:                      return;
```

[2] wegen der großen Variablenanzahl für jede Testeingabe wurde an dieser Stelle auf eine Beispielmenge verzichtet

```
5:              if (x*x + y*y == 1.5)
6:                      print("(%g,%g)",x,y);          /* Zielknoten */
        }
```

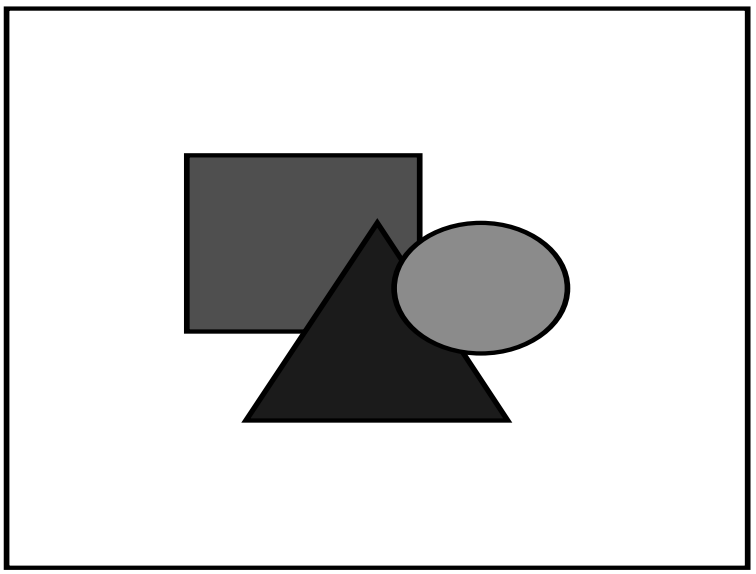


Abbildung 5.6: Zielfunktion im Bereich x, y: [-20, +40]

Die Abbildung zeigt den Eingabebereich -30 bis +40 und die errechneten Zielfunktionswerte. Es folgt eine Darstellung der gleichen Zielfunktion im Bereich -10 bis +15.

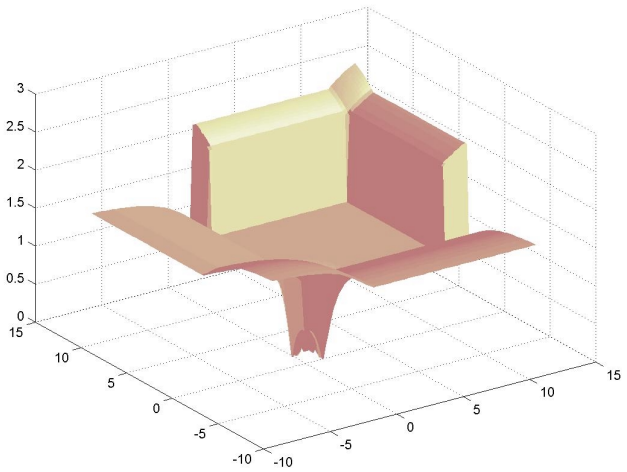


Abbildung 5.7: Zielfunktion im Bereich x, y: [-10, +15]

6 Diskussion der Zielfunktion

Der Erfolg einer Automatisierung von Softwaretests mit Hilfe evolutionärer Optimierungen ist von der Beschreibung des Suchproblems abhängig. Gibt eine Zielfunktion kaum Hinweis auf die Annäherung der Testdaten an das zu erreichende Teilziel, zum Beispiel durch eine wenig aussagekräftige Abstandsfunktion, wird die Optimierung zur Zufallssuche. Zielstellung ist deshalb die Festlegung von Zielfunktionen zur schnellen Optimierung von Testdaten für die einzelnen Teilziele des Strukturtests. Dazu werden im folgenden einige Problemsituationen der ausgearbeiteten Funktionen geschildert und mögliche Auswirkungen diskutiert.

Im Teilabschnitt 6.1 wird das Konvergenzverhalten von evolutionärere Algorithmen für die Zielfunktionen *knotenorientierter* Strukturtests analysiert. Es wird das Suchverhalten für verschiedene Situationen betrachtet. Teilabschnitt 6.2 beschäftigt sich abschließend mit der Diskussion der Zielfunktion für die pfadorientierten Tests.

6.1 Optimierungsverhalten bei knotenorientierten Tests

Um das Optimierungsverhalten des Testsystems näher analysieren zu können, wurden verschiedene Beispiel-Testobjekte konstruiert. Damit ist es möglich, spezielle Probleme der derzeitigen Zielfunktionen zu betrachten, bei denen die Optimierung mit der bisherigen Implementation gar nicht oder nur sehr langsam voranschreitet. Es werden im folgenden drei Situationen näher erläutert.

Zusammengesetzte Bedingungen

Ein einfaches Beispiel, das mit der bislang genutzten Zielfunktion sehr schlecht optimiert wird, ist eine zusammengesetzte Bedingung der folgenden Form:

```
Eingabe: A,B,C,D: integer

Bedingung: A=0 und B=0 und C=0 und D=0
```

C-Compiler optimieren die Auswertung aller Ausdrücke in einem Programm. Daher werden bei boolschen Ausdrücken immer nur die nötigen Teilbedingungen ausgeführt. Eine solche Optimierung bewirkt, daß zur Laufzeit nicht in jedem Fall alle Teilbedingungen einer Verzweigungsentscheidung ausgewertet werden. Diese Eigenschaft vieler moderner Compiler nutzen viele Programme aus, so daß eine Veränderung dieses Auswertungsverhaltens mögliche Seiteneffekte bewirken kann und damit die Semantik des Programms verfälscht.

Die für ASTELA entwickelte Instrumentierung nimmt keinen Einfluß auf das Bewertungsverhalten von Verzweigungsbedingungen. Dies hat den Nachteil, daß nur ein Teil der Messungen an einer Bedingung für die Abstandsfunktion und damit für die Bewertung der Testdaten genutzt werden können.

Das speziell konstruierte Beispiel mit der Bedingung "A=0 und B=0 und ..." wird auf Grund der teilweisen Auswertung der Bedingung nur sehr schlecht optimiert, weil

immer nur eine Variable zur Verbesserung der Bewertung beitragen kann. Das ist immer diejenige Variable von A bis D, die als erste ungleich Null ist. Der Vergleich dieser Variable mit Null ist die zuletzt ausgewertete Teilbedingung.

Während in der Anfangsphase sehr schnell Eingaben mit A gleich Null gefunden werden, verlangsamt sich die Optimierung jeder weiteren Variable, weil die Population iterativ zuerst gegen (0, x, y, z) dann gegen (0, 0, y', z') konvergiert und sich dabei der Suchraum immer weiter künstlich einschränkt (*eine Variation der ersten Variablen wird sehr schlecht bewertet*).

Werden in die Testdatenbewertung alle Teilbedingungen einbezogen, kann das Konvergenzverhalten wesentlich verbessert werden. Dazu müssen die Messungen vor der Verzweigung aufgenommen werden. Dies setzt jedoch voraus, daß bei der Auswertung der Teilbedingungen keine Variablen zum Beispiel durch Funktionsaufrufe verändert werden, weil sich sonst durch die Auswertung aller Teilbedingungen das Verhalten des Testobjekts gegenüber der nicht instrumentierten Programmversion ändern kann.

Eine Analyse der Datenflüsse im Programm kann prüfen, ob die Messung an den Teilbedingungen vor der Verzweigung durchgeführt werden können. Die Einbeziehung aller Teilbedingungen in die Abstandsfunktion beschleunigt die Optimierung, weil sich der Raum für mögliche Verbesserungen der vorliegenden Testdaten erweitert. So stehen im Beispiel bei der Zielfunktion "|a| + |b| + |c| + |d|" alle vier Variablen für die Verbesserung zur Auswahl.

Verzweigungshierarchien

Das zuvor beschriebene Beispiel behandelt das Optimierungsverhalten einer einzelnen Verzweigungsbedingung, wenn sich diese aus mehreren Teilbedingungen zusammensetzt. Der Selektionsprozeß findet aufgrund der Auswertungsreihenfolge immer nur anhand der zuletzt ausgewerteten Teilbedingung statt. Diese schrittweise Optimierung bewirkt eine starke Einschränkung der Möglichkeiten für Verbesserungen der vorliegenden Testdaten. Dasselbe Problem tritt für Teilziele auf, die von den Entscheidungen mehrerer aufeinander folgender Verzweigungen abhängig sind. Die Bewertung erfolgt in diesem Fall immer nur anhand der zuletzt ausgewerteten Verzweigungsbedingung. Zur Erläuterung folgt ein Beispiel:

```
If (a<0 || b<0)          return falsch;
If (a>100 || b>100)      return falsch;
x = f1(a); y = f2(a,b);
If (x==0)
        Anweisung1;
        if (y > 10)   /* < auszuführender Zielknoten > */;
```

Das Beispiel zeigt ein Testobjekt mit dem auszuführenden Zielknoten. Es muß eine Eingabe (a,b) gefunden werden, die im ersten Schritt die Bedingung "a<0 || b<0" nicht erfüllt. Der zweite Optimierungschritt betrifft die Verzweigungsbedingung "a>100 || b>100", die ebenfalls mit falsch durchlaufen werden muß. Nach der Berechnung von x und y auf Basis von a und b folgen die jeweils mit *wahr* auszuwertenden Bedingungen "x=0" und "y>10".

Bei der Optimierung von Eingaben kann immer nur der Teil der Messungen einbezogen werden, der tatsächlich ausgeführt wird. Damit schränkt sich ähnlich wie für die zusammengesetzten Bedingungen der Suchraum ein und die Optimierung schreitet gegebenenfalls nur sehr langsam fort. Je höher die Anzahl der Hierarchieebenen ist, desto schlechter sind die Erfolgschancen in einer vorgegebenen Anzahl von Generationen das Testdatum zu bestimmen.

Wie die folgende Transformation des Quelltextes zeigt, kann jedoch ein Teil der Messungen vorgezogen werden, so daß eine entsprechend angepaßte Berechnung der Zielfunktion das Konvergenzverhalten verbessert.

```
Messung1('a<0', 'b<0', 'a>100', 'b>100');
If (a<0 || b<0)          return falsch;
If (a>100 || b>100)      return falsch;
x = f1(a); y = f2(a,b);
Messung2('x==0', 'y>10');
If (x==0)
        Anweisung1;  // keine Änderungen von "y"
        if (y > 10)
                f(a); /* Zielknoten */;
```

Der transformierte Quelltext verhält sich genauso wie das Originalprogramm. Alle Messungen an den Teilbedingungen der Verzweigungen sind möglichst weit nach vorn gezogen. Damit wird erreicht, das Testdaten auch für noch nicht ausgeführte Bedingungen optimiert werden können. Beispielsweise werden durch die Messung 2 die Abstände für x gleich Null und y größer 10 bestimmt. Bei der Variation der Testdaten können beide Abstände berücksichtigt werden.

Schleifen im Testobjekt

Ein besonderes Problem bei der Suche von Testdaten sind Schleifen die Einfluß auf das Erreichen eines Zielknotens haben. Eine geeignete Zielfunktion muß eine Bewertung für alle Testfälle vornehmen, die den Zielknoten verfehlen, und dabei eine Unterschiedliche Annäherung zwischen den Schleifeniterationen bestimmen. In der für diese Arbeit realisierten Variante werden die Messungen der Verzweigungen innerhalb der letzten Schleifenwiederholung herangezogen. Es wird bei diesem Bewertungsverfahren immer nur die Messung benutzt, die an der Verzweigung durchgeführt wurde, die zum Verlassen der Schleife führt. Folgendes Beispiel läßt sich mit diesem Bewertungsansatz jedoch nur sehr schlecht optimieren (*sei A[1..20] Testeingabe*):

```
sum = 0;
for (i=1;i<20;i++)
   {    if (A[i]<0)  exit(-1);   }          ← ' exit ' verläßt die Funktion
/* ... auszuführender Zielknoten ...*/
```

Wird bei der Ausführung des Testobjektes mit einem Testdatum die Anweisung "exit(-1)" erreicht, so führt der Kontrollfluß am Zielknoten vorbei. Die daraus resultierende Bewertung nutzt nur die Abstandsfunktion von "A[i]<0" und betrachtet

dabei nicht, wie weit das zugehörige Testdatum von einem Abschluß der Schleife bei "i=20" entfernt ist. Eine Messung an dieser Bedingung würde zusätzlich Aufschluß über einen Optimierungsfortschritt geben.

Die Auswahl der zu benutzten Meßwerte beeinflußt damit den Verlauf der Zielfunktion und wirkt sich damit auf das Konvergenzverhalten der evolutionären Algorithmen bei der Suche nach Testdaten aus. Welche Messung für den Fortschritt steht, ist jedoch nicht in jedem Fall klar, wie das folgende Beispiel einer Schleife mit mehreren Austrittswegen zeigt:

```
while (g(A))
{
        if (A==10) return -3;          ← Anweisung verfehlt Zielknoten
        ... Anweisungen, die den Wert von "res" und "A" ändern...
        switch (res)
        {  case 1: return -1;          ← Anweisung verfehlt Zielknoten
           case 2: continue;
           case 0: break;              ← Anweisung erreicht Zielknoten
        }
}
f(A); /* auszuführender Zielknoten */
```

Im Gegensatz zum ersten Beispiel liegt hier eine Schleife vor, bei der nicht von einer festen Anzahl von Wiederholungen ausgegangen werden kann und mehrere Bedingungen zum Zielknoten verzweigen. Ein Optimierungsziel, im Sinne eines gewünschten Auswertungsergebnisses einer bestimmten Bedingung, welche weiter in Richtung Zielknoten führt, ist damit nicht gegeben. Unklar ist auch, ob eine Kombination der Meßwerte von verschiedenen Verzweigungen, welche zum Zielknoten führen, sinnvoll ist, um das Konvergenzverhalten zu verbessern.

6.2 Optimierung von Eingaben für Zielpfade

Bei den pfadorientierten Testverfahren wurden noch keine Experimente durchgeführt, deshalb können für die Diskussion der Zielfunktion auch nur die Beobachtungen der knotenorientierten Tests herangezogen werden. Dabei ist anzumerken, daß sich die Optimierung knoten- und pfadorientierter Teilziele nur darin unterscheiden, welche Messungen an den Verzweigungen in die Zielfunktion einbezogen werden.

In dieser Arbeit wurden zwei Bewertungsmethoden vorgeschlagen. Die möglichen Vor- und Nachteile sollen in diesem Teilabschnitt diskutiert werden. Prinzipiell können für die Optimierung von Verzweigungen mit zusammengesetzten Bedingungen die gleichen Aussagen wie bei den knotenorientierten Tests getroffen werden. Das heißt die dort diskutierten möglichen Verbesserungen sind auch für die pfadorientierten Test anwendbar.

Die erste Variante der Zielfunktion, das ist die Optimierung der Länge des *identischen Anfangspfads*, lehnt sich dabei an die Ergebnisse aus der Literatur. Hier wurde bereits von [Korel90] vorgeschlagen, die Bestimmung eines Testdatums zur Ausführung eines bestimmten Pfades durch die schrittweise Anpassung einer Eingabe zu realisieren, bei

welcher der durchlaufene Pfad vorliegt zu realisieren. Hierzu wird das Eingabedatum entsprechend einer Funktion zur ersten abweichenden Verzweigung variiert. Dies geschieht solange, bis diese Verzweigung, wie gewünscht, durchlaufen wird und ein längeres identisches Anfangsstück entsteht. Diese Bewertungsform wurde bereits erfolgreich für den evolutionären Test eingesetzt (siehe [Tracy98b]).

Der Vorteil des Einsatzes von evolutionären Algorithmen ist die technische Realisierung einer gerichteten Suche nach Testdaten, die ein immer längeres Pfadstück ausführen. Die evolutionären Algorithmen erweisen sich dabei als sehr robustes Optimierungsverfahren (siehe [Schwefl77]). Vor allem deshalb weil der Zielfunktionsverlauf von der Struktur und Semantik des Testobjekts abhängig ist, das heißt zum Beispiel viele lokalen Optima oder Plateaus besitzen kann.

In Anlehnung zu den Überlegungen zur Verbesserung der Abstandsfunktion bei zusammengesetzten Verzweigungsbedingungen entstand eine zweite Bewertungsidee für pfadorientierte Tests. Der Zielfunktionswert entsteht durch den Vergleich von ausgeführtem und gewünschten Pfad in mehreren Kontrollflußabschnitten. Auf diese Weise wird die Eignung eines Testdatums anhand einer Anzahl von Abweichungen bestimmt. Die Bewertungsmethode kann sich ähnlich wie bei den zusammengesetzten Verzweigungsbedingungen positiv auf das Konvergenzverhalten auswirken, weil die Möglichkeiten der Verbesserung eines Testdatums nicht nur auf eine Verzweigung beschränkt sind, sondern auf mehrere anzupassende Merkmale.

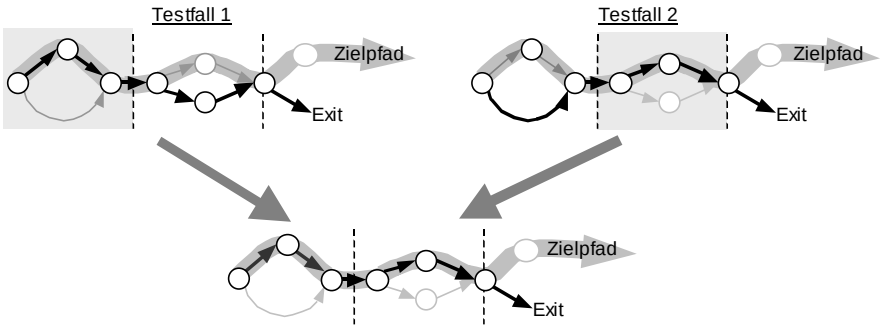


Abbildung 6.1: Kombination von zwei Testfällen

Aus dem Vergleich in mehreren Pfadabschnitten resultiert eine gleichberechtigte Bewertung der Individuen unabhängig davon welches der Pfadstücke identisch mit dem Zielpfad ist. Damit ist es den evolutionären Algorithmen möglich, Individuen zu kombinieren, die in verschiedenen Abschnitten mit dem Zielpfade übereinstimmen. Durch die Rekombination könnten dabei Individuen entstehen, welche die gewünschten Eigenschaften verbinden, das bedeutet das Testdaten erzeugt werden, die bei Ausführung des Testobjektes einen Pfad durchlaufen, der noch besser mit dem Zielpfad übereinstimmt. Das Bewertungsverfahren ermöglicht solche Optimierungsfortschritte. Die Häufigkeit dieses Erfolgs hängt jedoch ganz vom Aufbau des Testobjekts und den eingesetzten Rekombinationsoperatoren der evolutionären Algorithmen ab.

7 Zusammenfassung, Wertung und Ausblick

7.1 Zusammenfassung

Das im Rahmen dieser Arbeit entstandene System *ASTELA* realisiert den automatischen Strukturtest für *AnsiC*-Programme. Für die Testvorbereitung wurde ein Werkzeug entwickelt, welches die Parsierung, Instrumentierung und Testtreibergenerierung übernimmt. Die Instrumentierung der zu testenden Software erfolgt unabhängig vom gewählten Testverfahren.

Das Testsystem kann verschiedene Strukturtestverfahren automatisieren. Für die vom Benutzer gewählte Testmethode wird über Optimierungsprozesse eine Testdatenmenge gesucht, welche zur Überdeckung der Testkriterien führt. Dazu erfolgt die Identifikation von Teilzielen, die in einzelnen Optimierungen gelöst werden.

In dieser Arbeit wurden Zielfunktionen für die verschiedenen Testverfahren entwickelt. Die Optimierung der Teilziele von Anweisungs- und Zweigüberdeckungstests konnte in ersten Experimenten erprobt werden. Für die *Optimierung* wurden evolutionäre Algorithmen eingesetzt. Der so konzipierte evolutionäre Test durchmustert den Suchraum der Eingabeparameter eines Testobjekts. Jedes Individuum der durch den Algorithmus erzeugten Generation repräsentiert ein Testdatum, mit dem das Testobjekt über den automatisch generierten *Testtreiber* aufgerufen wird. Die bei der Testvorbereitung eingefügte *Instrumentierung* erstellt bei jeder Ausführung des Testobjekts ein Protokoll zu den durchlaufenen Meßpunkten. Aus diesen Messungen berechnet die Teststeuerung mit Hilfe der Zielfunktion eine Bewertung der Testdaten. Die Bewertung der Individuen wird zur Erzeugung besserer Testdaten verwendet.

Zur *Bewertung* der Testdaten wurden unterschiedliche Konzepte für die verschiedenen Testverfahren entwickelt. Sie stützt sich auf die zu Beginn eingeführte Klassifikation in *knotenorientierte*, *pfadorientierte*, *knoten-weg-* und *knoten-knotenorientierte* Verfahren. Bei den knotenorientierten Verfahren werden anhand der Meßdaten sogenannte Annäherungsstufen und eine Abstandsfunktion in Verzweigungsbedingungen bestimmt und daraus der Zielfunktionswert gebildet. Für die pfadorientierten Testverfahren werden zwei Methoden zur Bewertung vorgeschlagen. Die erste Methode berechnet den Zielfunktionswert anhand der Länge des identischen Anfangspfadstücks. Der ausgeführte Pfad kann dabei anhand des Meßprotokolls rekonstruiert werden. Eine zweite Methode betrachtet den durchlaufenen Gesamtpfad und bestimmt die Summe aller identischen Teilstücke im Vergleich zum Zielpfad.

Die Effizienz einer Optimierung kann durch die Einstreuung von guten Startwerten gesteigert werden. Deshalb wurde für die Bestimmung der Bewertung und die Zusammenstellung guter Startwerte ein sogenannter *Erreichbarkeitsgraph* konzipiert. In diesem Graphen können durch die Rekonstruktion des Testverlaufs erreichte Teilziele und die Bewertung für jedes nicht ausgeführte Teilziel bestimmt werden. Ein Vergleich der Testergebnisse für jedes Teilziel ermöglicht die Zusammenstellung der besten Testdaten. Diese werden als Startwerte benutzt.

Eigenschaften des Testsystems

Das erstellte Testsystem ASTELA führt Strukturtests von *AnsiC*-Software vollständig automatisch aus. Das System identifiziert für das Testproblem einzelne Teilziele und optimiert diese unabhängig voneinander. Zufällig erreichte Teilziele werden erkannt. Das Testsystem paßt die Reihenfolge der Abarbeitung der Teilziele entsprechend der vorliegenden Ergebnisse an. Es wird immer das Teilziel optimiert, für das die besten Startwerte vorliegen. Die Optimierung wird für den Fall, daß die Eingabe für ein Teilziel nicht gefunden werden kann, nach einer maximalen Anzahl von Versuchen abgebrochen. Als Ergebnis des Strukturtests wird ein Testdatum für jedes ausgeführte Teilziel sowie je ein Testdatum mit der besten Annäherung für jedes nicht ausgeführte Teilziel ausgegeben.

Die evolutionäre Optimierung wird mit der GEA-Toolbox [Pohlh-GEA] realisiert. Während der Optimierung übernimmt die Teststeuerung die Aufgabe der Bewertung der über die Toolbox generierten Testdaten. Dazu führt die Teststeuerung das Testobjekt mit Hilfe eines zum Testobjekt generierten Testtreibers aus und berechnet den Zielfunktionswert auf Grundlage des Testprotokolls.

Mit Hilfe eines sogenannten *Erreichbarkeitsgraph* konnte eine Verfahren entwickelt werden, das die parallele Bewertung von Testergebnissen für alle Teilziele ermöglicht. Auf diesem Weg können gute Startwerte für die verschiedenen Optimierungen gesammelt werden. Dies bewirkt in vielen Fällen eine Beschleunigung des Gesamtprozesses der Testdatengenerierung.

7.2 Wertung

Das System für den automatischen Strukturtest unterteilt sich in zwei Hauptkomponenten - das Werkzeug zur Testvorbereitung und die Teststeuerung für die Ausführung des Strukturtests.

Im Rahmen dieser Arbeit wurde die Testvorbereitung von *AnsiC*-Programmen erfolgreich implementiert. Parser und Instrumentierer arbeiten uneingeschränkt für alle Formen von *AnsiC*-Quellprogrammen. Das Werkzeug stützt sich auf ein objektorientiertes Konzept für die Darstellung der Sprachelemente. In dieses Modell konnten sehr leicht die Instrumentierung und Testtreibergenerierung integriert werden.

Einzelne Aufgaben wie zum Beispiel die Konstruktion des Kontrollflußgraphen konnten in einem sprachunabhängigen Basismodell realisiert werden, so daß bei der Einbettung neuer Quellsprachen der Aufwand für die Implementation dieser Teilaufgaben entfällt.

Das Konzept sieht eine einmalige Instrumentierung des Quelltextes der zu testenden Software unabhängig vom gewählten Testkriterium vor, weil sich die notwendigen Eingangsdaten aller gebildeten Zielfunktionen auf die gleichen Meßpunkte in der zu testenden Software stützen. Nach erfolgter Instrumentierung können mit der Software beliebige Strukturtests durchgeführt werden.

Die Testtreibergenerierung konnte in eingeschränktem Umfang integriert werden. Nicht unterstützt sind dynamische Datenstrukturen, wie Listen und Bäume sowie dynamische

Arrays und variable Datentypen. Für solch komplexe Datenstrukturen sind zusätzliche manuelle Spezifikationen notwendig. Eine automatische Generierung von Modulstubs, die dem Test von noch nicht vollständig implementierter Software dienen, wurde noch nicht umgesetzt. Dies schränkt die Möglichkeiten des Test jedoch kaum ein, da die manuelle Integration funktionsloser Modulstubs mit wenig Aufwand verbunden ist.

Die Berechnung der Zielfunktion in einem Erreichbarkeitsgraphen erwies sich als geeigneter Weg zur gleichzeitigen Bewertung aller offenen Teilziele. Damit konnte eine Methode zur Zusammenstellung guter Startwerte für alle noch nicht bearbeiteten Teilziele verwirklicht werden.

Das System ASTELA verwendet für Anweisungs- und Zweigtests eine Zielfunktion, die unabhängig vom durchlaufenen Pfad eine Bewertung der Testdaten für das jeweilige Teilziel vornimmt. Dieser Ansatz führte bei der getesteten Software zu guten Ergebnissen. Es bedarf jedoch noch weiterer Untersuchungen mit anderen Testobjekten.

Für die netzwerkbasierte Kommunikation zwischen Teststeuerung und evolutionären Algorithmen konnte eine neue Schittstelle (*PEANUTs*) erarbeitet werden. Mit dieser Komponente ist es möglich einen beliebigen Optimierungsprozeß mit der Teststeuerung zu verbinden. Die Trennung der Beschreibung des Optimierungsproblems und des Lösungsverfahrens erweist sich für Weiterentwicklungen als nützlich. Beispielsweise können so mehrere Optimierungen parallel über verschiedene Prozesse ablaufen oder andere Optimierungsmethoden zur Anwendung kommen.

7.3 Ausblick

Das Testsystem ASTELA automatisiert die Testdatenbestimmung für strukturierte Tests. Der Abschnitt beschreibt Ideen für mögliche Erweiterungen des Testsystems, unter anderem durch den Ausbau der Testtreiberfunktionalität, die bessere Beschreibung des Suchproblems und den Einsatz weiterer Optimierungsverfahren zur Verbesserung der Erfolgschancen bei der Testdatenbestimmung. Des weiteren wird der Einsatz einer Visualisierung von Softwaretests und zum Abschluß weitere Anwendungsgebiete für den automatischen Strukturtest beschrieben.

Erweiterungen für komplexe Testeingabeschnittstellen

Die evolutionären Algorithmen gehen von einer festen Struktur und Kodierung der Eingaben aus. Dagegen können Listen, Arrays und Bäume sowie andere dynamische Datenstrukturen eine beliebige Komplexität annehmen. Zur Nutzung des Systems für Tests von Funktionen mit entsprechenden Schnittstellen, dazu gehören zum Beispiel auch typvariable Daten, ist für den Test eine Eingabespezifikation notwendig. So ist für einen Test folgendes festzulegen:

1. Ober- und Untergrenze der Variabilität dynamischer Datenstrukturen. Hierzu zählt die maximale Anzahl von Elementen bei Listen, Arrays und Bäumen
2. Abhängigkeiten zwischen Datenelementen von zusammengesetzten Strukturen
3. Einschränkungen von Wertebereichen beziehungsweise die Festlegung auf bestimmte Werte von Eingabedatenstrukturen

Entsprechend der Spezifikation der Schnittstelle ist durch den Testtreibergenerator Programmcode zu erzeugen, welcher die Konvertierung eines Zahlenvektors in die dynamischen Strukturen übernimmt. Dafür sind entsprechende Kodierungen zu entwerfen und das Verhalten während des Tests zu prüfen.

Verbesserung der Zielfunktion

Schnellere Optimierung durch Messungen an allen Teilbedingungen

In *ASTELA* erfolgt die Bestimmung der Zielfunktion nur anhand der tatsächlich ausgewerteten Bedingungen. Diese Einschränkung resultiert aus dem Problem der Veränderung des Verhaltens durch Seiteneffekte, wenn die Auswertung aller Teilbedingungen erzwungen wird. Folgendes Beispiel verdeutlicht das Problem:

Beispielquellcode mit Seiteneffekten:

```
If (a++ && a++) a=a+10;
```

Die zweite Bedingung a++ wird nur ‘ausgeführt’ wenn das erste 'a++' wahr ergibt. Dieses Verhalten muß nach der Instrumentierung erhalten bleiben. Unter diesen Umständen ist eine vorgezogene Messung an allen Teilbedingungen nicht möglich.

Wie bereits bei der Diskussion der Zielfunktion für knotenorientierte Tests bemerkt, kann sich die Verwertung der Messungen an allen Teilbedingungen vorteilhaft auf die Konvergenz während der Optimierung auswirken. Für alle seiteneffektfreien Bedingungen ist deshalb auf jeden Fall eine Instrumentierung vorzunehmen, welche unabhängig von den Optimierungen durch den Compiler die Messungen an allen Teilbedingungen liefert.

Als gleichwertige Lösung für Verzweigungsbedingungen mit Seiteneffekten bietet sich die Anwendung von Programmtransformationen an. Solche Quelltextänderungen, welche die Semantik beibehalten, jedoch Seiteneffekte beseitigen, ermöglichen Messungen an allen Teilbedingungen ohne das Verhalten des Testobjektes zu beeinflussen. So kann der vorangegangene Beispielquellcode wie folgt gewandelt werden:

Transformation zu einer seiteneffektfreien Bedingung

```
Messung(a==0); Messung (a+1==0);
If (a && a+1)
      { a+=2;  a=a+10; }
else  { a++; if (a) a++; }
```

Die Idee könnte auf die Transformation von Bedingungen mit Funktionsaufrufen erweitert werden. Das Problem wird damit aber wesentlich umfangreicher. Zu analysieren ist, ob diese Funktionen globale Variablen oder ihre Parameter ändern. In diesem Fall müßte eine Transformation der Funktion in verschiedene Versionen erfolgen. Der Aufwand kann dabei jedoch erheblich steigen.

Verbesserung der Optimierung durch vorgezogene Messungen

Zur Optimierung von Teilzielen, welche durch viele Hierarchieebenen gekennzeichnet sind, kann ähnlich wie bei zusammengesetzten Bedingungen eine Verbesserung der Optimierung über das Vorziehen von Messungen erreicht werden. Der günstigste Fall tritt ein, wenn alle Messungen direkt zu Beginn stattfinden könnten. Diese Lösung ähnelt dem Ansatz der symbolischen Ausführung, bei der versucht wird die Bedingungen zur Ausführung einer Anweisung als logischen Term auszudrücken. Die Konstruktion eines solchen Terms ist jedoch sehr schwierig, existierende Lösungsansätze beschränken sich auf bestimmte Konstrukte von Programmiersprachen. Vor allem Schleifen, Arrays und Pointer bereiten Probleme.

Das möglichst weite vorziehen von Messungen stellt einen Zwischenschritt zur Bildung einer quasi vollständigen Zielfunktion dar und läßt auf Grund der Erfolge bei zusammengesetzten Bedingungen gute Ergebnisse für schwer zu optimierende Teilziel erwarten.

Anpassung der Zielfunktion für Schleifen

Bei der Diskussion zur Handhabung von Messungen aus mehreren Schleifeniterationen wurden die Probleme der Erkennung eines Optimierungsfortschritts erläutert. Pauschal läßt sich kein Weg zur Definition einer Zielfunktion festlegen, weil sehr unterschiedliche Schleifenkonstruktionen möglich sind. Eine semantische Analyse der Strukturen und die Erkennung der Meßpunkte, welche den Fortschritt einer Optimierung ausdrücken, würde die Bildung einer Funktion ermöglichen, die eine gezielte Bewertung zuläßt.

Messungen an boolschen Variablen und Aufzählungstypen

Für den Fall, daß eine Bedingung aus boolschen Variablen oder solchen Variablen, die nur wenige Werte annehmen (z.B. Aufzählungstypen), besteht, kann keine ausreichend aussagekräftige Zielfunktion gebildet werden, weil die Messung eine Abstandsfunktion mit nur wenig Orientierung zur Folge hat.

Ein Vorschlag zur Lösung dieses Problems ist die Analyse aller Anweisungen, die für die Bildung der problematischen Variablenwerte verantwortlich sind. Dies kann durch die Bestimmung des Datenflusses realisiert werden.

Welche Probleme durch wenig aussagekräftige Messungen an Variablen entstehen, die sich auf eine kleine Anzahl von möglichen Werten beschränken, zeigt folgendes Beispiel:

```
Dreieck(a,b,c):
    match=0
    if a=b then match++
    if b=c then match++
    if a=c then match++

    if (match=1) print "gleich-schenklig"
    if (match=3) print "gleich-seitig"
```

Die Suche nach Testdaten, welche 'match=3' erfüllen, kommt bei der derzeitigen Implementation einem Zufallstest gleich. Über eine Erweiterung des Testsystems um eine Datenflußanalyse kann in diesem Beispiel jedoch aus der Programmstruktur hergeleitet werden, welche Voraussetzungen für den Wert 'match=3' gültig sein müssen, mit dem Ziel daraus eine Bewertung der Testdaten zu konzipieren.

Einschränkung des Suchbereichs

Mit Hilfe einer Datenflußanalyse können für jede der Bedingungen eines Programms alle ausschlaggebenden Eingabevariablen identifiziert werden. Damit wird es möglich für die Optimierung der Eingaben den Suchbereich einzuschränken. Durch die Begrenzung der zu variierenden Parameter ist eine Beschleunigung der Optimierung zu erwarten, weil nur tatsächlich wirksame Änderungen getestet werden. Vergleichbar ist dies mit einer Zielfunktion, die 10 Eingabevariablen besitzt, von denen jedoch nur eine zur Berechnung herangezogen wird.

Ausbildung von Populationsarten

Motiviert von dem Gedanken, daß die Kombination von Individuen mit bestimmten Eigenschaften in einigen Situationen nicht sinnvoll erscheint, entstand das Konzept der Ausbildung sogenannter Arten von Individuen. Die Individuen verschiedener Arten versuchen parallel das gesetzte Teilziel auf unterschiedlichen Wegen zu erreichen.

Grundidee ist, daß eine Populationsart versucht den Zielknoten über einen bestimmten Pfad oder eine Gruppe von Pfaden zu erreichen. Die Kombination von Individuen, welche verschiedene Pfade ausführen, wird unterbunden oder durch eine sehr kleine Migrationsrate nur selten zugelassen. Die Populationsgrößen könnten je nach Optimierungsfortschritt variabel gestaltet und auf diese Weise eine Konkurrenz zwischen den Arten realisiert werden.

Kombination von Optimierungsverfahren

Motiviert durch die Ergebnisse der Arbeit [McGraw97], ist eine effektivere Gestaltung des Suchprozesses durch Kombination von Optimierungsmethoden denkbar. Neben den evolutionären Algorithmen sollten andere Optimierungsmethoden wie beispielsweise das *Hill-climbing* oder ein *Gradient-Descent* Algorithmus parallel angewendet werden. Auf diese Weise kann die für das Problem am besten arbeitende Methode der Testdatengenerierung dienen.

Visualisierung von Softwaretests

Das Testwerkzeug ASTELA arbeitet ohne eine Visualisierung der Testdatensuche. Eine Erweiterung des Systems könnte die Darstellung von Optimierungserfolgen sein, so daß der überwachende Tester aus der graphischen Darstellung Problemzonen, das heißt nicht durchlaufene Anweisungen, Zweige oder Programmpfade, und die möglichen Ursachen erkennen kann. Diese Visualisierung läßt die Verbindung von Testwerkzeug und Debugging zu. Der Tester erhält die Möglichkeit, das Verhalten an Programmverzweigungen zu prüfen und auf diese Weise Fehlerquellen im Programm aufzuspüren.

Erschließung neuer Anwendungsfelder

Strukturtests auf funktionaler Ebene bilden nur einen kleinen Teil des Testumfangs bei der Entwicklung von Softwareprodukten. So sind neben der Überprüfung von einzelnen Funktionen auch Tests auf modularer Ebene notwendig. Für solche Tests wird der sogenannte *procedure call graph* benutzt. Dieser Graph dient der Definition von Testfällen, also Szenarios, bei denen eine Reihe von Funktionen ausgeführt werden. Die Suche nach Testdaten, welche bestimmte Testfälle ausführen, könnte genauso wie bei den hier vorgestellten einfachen Strukturtests als Optimierungsproblem mit einer Zielfunktion formuliert und mit Hilfe des Testsystems automatisiert werden. Anstatt des Kontrollflußgraphen einer einzelnen Funktion kommt dabei ein zusammengesetzter Graph von allen betroffenen Funktionen zur Anwendung.

Die Entwicklung von modernen Softwareprodukten basiert heute schon häufig auf objektorientierten Sprachen. Damit entsteht die Notwendigkeit des Tests von OO-Architekturen. In der Literatur existieren verschiedene Ansätze für dieses Problem. Ein Ansatz ist die Anpassung der klassische Testmethoden für die neuen Anforderungen. Das bedeutet zum Beispiel den Strukturtest von einzelnen Objektmethoden. Diese Form des Tests erfaßt jedoch nicht alle Formen von Fehlerquellen, die in objektorientierten Systemen auftreten können. Nötig ist zum Beispiel der Test von Objektschnittstellen, das heißt das Verhalten des Objektes in verschiedenen Zuständen. Es stellt sich damit das Problem der Definition von Testzielen. Ein sehr schwieriges Problem ergibt sich auch für die Messungen in der zu testenden Software, weil in den meisten modernen Programmiersprachen alle Operationen und damit auch die Relationen in Verzweigungsbedingungen im Programm nahezu beliebig implementiert werden können. Hierfür ist die Abstandsbestimmung neu zu überdenken.

8 Quellenverzeichnis

[AhoSet92] A.Aho, R.Sethi, J.Ullmann, *Compilerbau, Addison-Wesley, Bonn 1992, (engl.Titel: "Compilers: Principles, Techniques and Tools", 1986)*

[Baker87] Baker, J., *Reducing Bias and Inefficiency in the Selection Algorithm, Proceedings of 2nd International Conference on Genetic Algorithms (ICGA'87), Cambridge, Massachusetts, USA, 1987, pp. 14-21*

[Balzert93] Helmut Balzert (Hrsg.), *CASE Systeme und Werkzeuge, Wissenschaftsverlag 1993.*

[Balzert98] Helmut Balzert, *Lehrbuch der Software-Technik: Software-Managment, Software-Qualitätssicherung, Unternehmensmodellierung, Spektrumverlag 1998*

[Baresel] Baresel, André, *Instrumentierung von Quellprogrammen für evolutionäre Strukturtests ein sprachunabhängiger Ansatz, Studienarbeit, Humboldt Universität, Jan.2000*

[CACM] *Collected Algorithms from the Communications of the ACM, Volume I,II und III, "http://www.acm.org/calgo" oder "http://www.netlib.org/toms"*

[Chusho87] Chusho T., *Test Data Selection and Quality Estimation Based on the Concept of Essential Branches for Path Testing, IEEE Transactions on Software Engineering Vol.13,No.5, May 1987, pp 509-517*

[Davis96] Davis, L., *Handbook of Genetic Algorithms, International Thomson Computer Press. 1996*

[DeMillo87] R.A. DeMillo, W.M. McCracken, R.J. Martin, J.F. Passafiume, *Software Testing and Evaluation, The Benjamin/Cummings Publishing Comp., Inc. 1987*

[Fergus96] Roger Ferguson and Bogdan Korel, *The Chaining Approach for Software Test Data Generation, ACM Transactions on Software Engineering and Methodology, Vol. 5 No.1, January 1996, pp.63-86*

[Fischer91] C.Fischer, R.LeBlanc, *Crafting a Compiler with C, The Benjamin/Cummings Publishing Comp., Inc., California 1991*

[Ghezzi87] Carlo Ghezzi and Mehdi Jazayeri, *Programming Language Concepts, John Wiley & Sons, Inc. 1987*

[Goldbg89] Goldberg, D.E., *Genetic Algorithms in Search, Optimization and Machine Learning, Addison Wesley Publishing Comp., Inc. 1989(Reprint), 1953(Original)*

[Hennel76] M.A.Hennel, D.Hedley and M.R.Woodward, *Quantifying the Test Effectiveness of Algol 68 Programs, The Computer Journal, 1976; pp.36-41*

[Hollnd75] John H. Holland, *Adaption in natural and artificial Systems, University of Michigan, 1975*

[Huang79] Huang J.C., *Programm Instrumentation: A Tool for Software Testing, Software Testing, Infotech State of the Art Report, Vol. 2, Maidenhead 1979, pp. 147-159*

[Ince87] D.C. Ince, *The Automatic Generation of Test Data, The Computer Journal, Vol.30, No. 1, 1987*

[Jasper94] Robert Jasper, Mike Brennan,Keith Williamson,Bill Currier, *Test data generation and feasible path analysis, ISSTA '94. Proceedings of the 1994 international symposium on software testing and analysis, pp.95-107*

[Jones96] B.Jones, H.Stahmer, D.Eyres, *Automatic Structural Testing Using Genetic Algorithms.Software Engineering Journal, September 1996, pp.299-306*

[Jones98] B.Jones, H.Sthamer, D.Eyres, *A Strategy for using Genetic Algorithms to Automate Branch and Fault-based Testing, The Computer Journal, Vol. 41 No. 2, 1998, pp.97-107*

[Korel90] Bogdan Korel, *Automated Software Test Data Generation, IEEE Transactions on Software Engineering, August 1990. vol.16 no.8, pp.870-879*

[LexYacc] Herold, Helmut, *Lexx und Yacc: lexikalische und syntaktische Analyse, Addison-Wesley, 1992, 1.Auflage (UNIX und seine Werkzeuge)*

[Ligges93] Liggesmeyer Peter, *Wissensbasierte Qualitätsassistenz zur Konstruktion von Prüfstrategien für Softwarekomponenten,BI-Wiss.-Verl. 1993*

[Mathworks] The MathWorks, Inc., *MATLAB 5.x http://www.mathworks.com/products/matlab*

[McCabe76] Thomas J. McCabe, *A Complexity Measure, IEEE Transactions on Software Engineering, Dec.1976, Vol 2 No. 1*

[McGraw97] Gary McGraw und Christoph Michael, *Opportunism and Diversity in Automated Software Test Data Generation*

[Michal92] Zbigniew Michaelewicz, *Genetic Algorithms+Datastructures=Evolution Programs, Springer-Verlag Berlin Heidelberg 1992*

[Myers79] Glenford J. Myers, *The Art of Software Testing, John Wiley & Sons, Inc., New York, 1979*

[Pagel94] Pagel, Bernd-Uwe, *Software Engineering: Die Phasen der Software Entwicklung, B.-U.Pagel und H.-W. Six, Addison-Wesley, 1994, 1.Auflage*

[Pargas99] Roy. P. Pargas, Mary Jean Harrold, Robert R. Peck, *Test-Data Generation Using Genetic Algorithms, Journal of Software Testing, Verification and Reliability, 1999 (to appear)*

[Pohlhm99] Pohlheim, H., *Evolutionäre Algorithmen – Verfahren, Operatoren und Hinweise für die Praxis. Springer Verlag, 1999*

[Pohlh-GEA] Pohlheim, H., *Genetic and Evolutionary Algorithm Toolbox for use with Matlab. Technical Report, Technical University Ilmenau, 1994-1998, http://www.geatbx.com oder http://www.pohlheim.com*

[Prather87] Ronald E. Prather and J.Paul Myers, Jr. *The Path Prefx Software Testing Strategy, IEEE Transactions on Software Engineering, July 1987, Vol.13 No. 7*

[Rechb73] I. Rechenberg, *Evolutionsstrategie – Optimierung technischer System nach Prinzipien der biologischen Evolution, Friedrich Frommann Verlag, Stuttgart-Bad Cannstatt, 1973*

[Riedm97] Eike Hagen Riedemann, *Testmethoden für sequenzielle und nebenläufige Software-Systeme, B.G.Teubner Stuttgart 1997*

[Schaeff73] Schaeffer, Marvin, *A mathematical theory of global program optimization, Prentice-Hall Inc., 1973*

[Schönb94] Schöneburg, Eberhard, *Genetische Algorithmen und Evolutionsstrategien, Addison-Wesley, 1994*

[Schrein85] A.T.Schreiner, G.Friedman, *Compiler bauen mit UNIX: Eine Einführung, Carl Hanser Verlag, 1985*

[Schwefl77] Hans-Paul Schwefel, *Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie, 1.Aufl. Birkhäuser Verlag Basel, 1977*

[Sthamer96] Sthamer, H.-H., *The Automatic Generation of Software Test Data Using Genetic Algorithms., Dissertation, University von Glamorgan, Wales, Großbritanien, 1996*

[Tracy98b] Nigel Tracey, John Clark, Keith Mander, John McDermid, *An Automated Framework for Structural Test-Data Generation, Proceedings of the 13th IEEE Conference on Automated Software Engineering (ASE), Oct.1998,*

[Tracey98] Nigel Tracey, John Clark, Keith Mander, *The Way Forward for Unifying Dynamic Test Case Generation – The Optimisation-Based Approach, International Workshop on Dependable Computing and Ist Appl. (IFIP'98), Jan.1998, pp.169-180*

[Wilhlm92] R.Wilhelm, D. Maurer: *Übersetzerbau: Theorie, Konstruktion, Generierung, Springer Verlag, 1992*

[YACC99] Philip Stearns, *Parser Generator Version 0.62 (Beta Release) www.cix.co.uk/~phil-stearns*

Anhang A Quelltexte der Testobjekte

Beispiel 1: Dreieckstypbestimmung

```
void DreiecksTyp(double a,double b, double c)
{
1         if (a<=0 || b<=0 || c<=0)
2                 { error=1; return; } // negative seitenlängen
3         if (a>10000 || b>10000 || c>10000)
4                 { error=1; return; } // zu große seitenlängen
5,6       if (c>=a+b) { error=2; return; } // kein dreieck
7,8       if (a>=b+c) { error=3; return; } // kein dreieck
9,10      if (b>=a+c) { error=4; return; } // kein dreieck
          // Sortierung der Seiten
11        if (c<b)
12        {     if (b<a)
13                 {  double h2=a; a=c;c=h2; }
                else
14                 {  double h2=b; b=c;c=h2; }
          } else
15        {     if (c<a)
16                 {  double h2=c; c=a;a=h2; }
          }
17        if (a>b)
18              {  double h2=a; a=b;b=h2; }
          // Dreiecks-typ-bestimmung:
19        if (a==b && b==c)
20              {  ist_gleichschenklig =1; ist_gleichseitig =1; }
          else
21        {     if (a==c || b==c || a==b)
22                 { ist_gleichschenklig = 1; }
23              else
                   { ist_gleichschenklig =0; }
24              ist_gleichseitig =0;
          }
25        if (a*a+b*b==c*c)
26              ist_rechtwinklig = 1;
27        else  ist_rechtwinklig = 0;
28        if (a*a+b*b<c*c)
29              { ist_stumpwinklig = 1; }
30        else  { ist_stumpwinklig = 0; }
}
```

Beispiel 2: Komplexe Verzweigungshierarchie

```
int complex_branching(short a,short b, short c, short d, short e, short f)
{           char z;
1           if (a<e || e+a != c+b || e+c < a+b+d )
            {
2                 a++;
3                 switch ((int)a+(int)b)
                  {
4                 case 0: return 2;
5                 case 10: a = b-e; break;
6                 case 23: e = a+b+d; break;
7                 default: c = a; a = b;
                  }
8                 c--;
            } else
            {
9                 b++;
10,11             if (c<b || (a>b && e<d) || a == e) c++;
            }
12          a = b+c; z=0;
13          while (f < d && z<100)
            {
14                z++; e = c+e+b; f-=2*a+e+1;
15,16             if (e < f) break;
17                a = d+a;
18,19             if (f == a) break;
            }
20,21       if ((a<b || b<c) && (a<c || e<d)) return 1;
22          a = b+c; z=0;
23          while (f+e < a+b && z<100)
            {
24                z++; e = c+e+b; f-=2*a+e;
25,26             if (e < f) break;
27                a = d+a;
28,29             if (f == a) break;
            }
30          return 0;
}
```

Beispiel 3: Line covered by rectangle

Quelle: Joachim Wegener

```
struct point { int x, y; };
struct Line {  struct point p1, p2; };
struct Rectangle { struct point rect_p; int width, heigth; };

enum {no, yes}
is_line_covered_by_rectangle (struct Rectangle rectangle, struct Line line)
{          struct rectanglePoints { struct point p1, p2, p3, p4; } rec;
           int i, p1x, p1y, p2x, p2y;
           float det, m , n;
           /* Punkte des Rectangles berechnen */
1          rec.p1.x = rectangle.rect_p.x; rec.p1.y = rectangle.rect_p.y;
           rec.p2.x = rec.p1.x + rectangle.width; rec.p2.y = rec.p1.y;
           rec.p3.x = rec.p1.x; rec.p3.y = rec.p1.y + rectangle.heigth;
           rec.p4.x = rec.p1.x + rectangle.width;
           rec.p4.y = rec.p1.y + rectangle.heigth;
           /* Determinante für line und die vier Linien des rectangles berechnen:*/
2          for (i=0; i <= 3; i++)
           { /* setzen der Hilfsgerade auf jeweils eine der vier Aussenlinie des rectangle */
3               if (i == 0)
4               { p1x = rec.p1.x; p1y = rec.p1.y; p2x = rec.p2.x; p2y = rec.p2.y;}
5               else if (i == 1)
6               { p1x = rec.p1.x; p1y = rec.p1.y; p2x = rec.p3.x; p2y = rec.p3.y;}
7               else if (i == 2)
8               { p1x = rec.p2.x; p1y = rec.p2.y; p2x = rec.p4.x; p2y = rec.p4.y;}
                else /* if (i == 3) */
9               { p1x = rec.p3.x; p1y = rec.p3.y; p2x = rec.p4.x; p2y = rec.p4.y;}
                /* Berechnung der Determinante der beiden Geraden  */
10              det = (((line.p2.x - line.p1.x) * (-p2y + p1y))
                      - ((-p2x + p1x)  * (line.p2.y - line.p1.y)));
11              if (det != 0)
                { /* es gibt einen eindeutigen Schnittpunkt ->
                        Berechnung der Lambda-Werte der Geraden-Vektoren-Matrix */
12                m = (((p1x - line.p1.x) * (-p2y + p1y))
                        - ((-p2x + p1x) * (p1y - line.p1.y))) / det;
                  n = (((line.p2.x - line.p1.x) * (p1y - line.p1.y))
                        - ((p1x - line.p1.x) * (line.p2.y - line.p1.y))) / det;
                  /* Liegt der Schnittpunkt der Geraden auch auf den Linien? */
13                if ((m >= 0) && (m <= 1) && (n >= 0) && (n <= 1))
14                  return yes; /* line wird vom rectangle ueberdeckt*/
                }
           }
           /* es gibt keinen Schnittpunkt zwischen der line und allen Geraden des  rectangles
           -> liegt line auf einem Aussenrand des rectangles oder vollstaendig im rectangle?*/
15         if ( (rec.p1.x <= line.p1.x) && (line.p1.x <= rec.p2.x)
                 && (rec.p1.y <= line.p1.y) && (line.p1.y <= rec.p3.y))
16           return yes; /* Linie liegt auf oder im rectangle */
17         else return no;  /* line wird nicht vom rectangle ueberdeckt*/
}
```

Beispiel 4: Netflow

Quelle: Collected Algorithms of the ACM [CACM] Algorithm 248

```
// Netflow: 248-P 1 - 0
#undef _MSC_VER
#include "AnsiCInstrLib.h"
#include <stdlib.h>
#include <stdio.h>
#define NUM_ARCS 10
#define NUM_NODES 6

typedef int arc_array[NUM_ARCS+1];
typedef int node_array[NUM_NODES+1];

void NetFlow(arc_array I/*BRANCH from node*/, arc_array J/* BRANCH to node*/,
             arc_array cost /* flow costs*/,
             arc_array hi /*maximum flow*/, arc_array lo /* minimum flow*/,
             arc_array flow /* real flow */, node_array pi /* node input */)
{  int nodes = NUM_NODES;   int arcs  = NUM_ARCS;
   int a,aok,c,cok,del,e,eps,inf,lab,n,ni,nj,src,snk; node_array na,nb;
1          for (a=1;a<=arcs;a++)
2,3             if (lo[a]>hi[a]) { int v = lo[a];lo[a]=hi[a];hi[a]=v; }
5          inf = 99999999; aok = 0;
6          while (1) // LABEL 'SEEK'
           {
7               for (a=1;a<=arcs;a++)
8               {  c=cost[a]+pi[I[a]]-pi[J[a]];
9                  if (flow[a]<lo[a] || (c<0 && flow[a]<hi[a]))
10                    { src=J[a]; snk=I[a]; e=1; break; } // finish loop
11                 if (flow[a]>hi[a] || (c>0 && flow[a]>lo[a]))
12                    { src=I[a]; snk=J[a]; e=-1; break; } // finish loop
                }
13,14           if (a==arcs) return; // if loop has not been finished before
15              if (!(a==aok && na[src]!=0))
16              {  aok=a;
17                 for (n=1;n<=nodes;n++)
18                    { na[n]=0; nb[n]=0; }
19                 na[src]=abs(snk)*e;nb[src]=abs(aok)*e;
                }
20              cok=c;
                do {
21                 lab=0;
22                 for (a=1;a<=arcs;a++)
23                 {  if ((na[I[a]] == 0 && na[J[a]]==0) ||
                          (na[I[a]] != 0 && na[J[a]]!=0))
24                    continue; // goto XC is LOOP !
25                    c=cost[a]+pi[I[a]]-pi[J[a]];
26                    if (na[I[a]]!=0) // if == 0 goto XA
27                    {  if (flow[a]>=hi[a] || (flow[a]>=lo[a] && c>0))
28                          continue;
29                       na[J[a]]=I[a]; nb[J[a]]=a; // goto 'XB' implizit now!
                      } else
```

```
                    { // LABEL 'XA'
30,31                  if (flow[a]<=lo[a] || (flow[a]<=hi[a]&&c<0)) continue;
32                     na[I[a]]=-J[a]; nb[I[a]]=-a;
                    }
                    // LABEL 'XB'
33                  lab=1;
34                  if (na[snk]!=0) // WAS JUMP ! CODE FROM END INSERTED HERE!
35                  {  eps = inf; ni=src;
                       do {
36                        nj=abs(na[ni]); a=abs(nb[ni]);
                          c=cost[a]-abs(pi[ni]-pi[nj])*sign(nb[ni]);
37                        if (nb[ni]>=0)
38                        {  if (c>0 && flow[a]<lo[a])
39                              eps=min(eps,lo[a]-flow[a]);
40                           if (c<=0 && flow[a]<hi[a])
41                              eps=min(eps,hi[a]-flow[a]);
                          } else // XE:
42                        {  if (c<0 && flow[a]>hi[a])
43                              eps=min(eps,flow[a]-hi[a]);
44                           if (c>=0 && flow[a]>lo[a])
45                              eps=min(eps,flow[a]-lo[a]);
                          }
46                        ni=nj;
47                     } while (ni!=src);
                       do {
48                        nj=abs(na[ni]); a=abs(nb[ni]);
                          flow[a]=flow[a]+eps*sign(nb[ni]);
                          ni=nj;
49                     } while (ni!=src);
50                     aok=0; lab = -10000000; break; // GOTO 'SEEK' Node 6
                    }
                 } // END Of 'for' NODE 22
51            } while (lab!=0); // XC - GOTO Node 21
52,53         if (lab == -10000000) continue; // GOTO 'SEEK';
54            del=inf;
55            for (a=1;a<=arcs;a++)
56            {  if (!((na[I[a]]==0 && na[J[a]]==0) ||
                       (na[I[a]]!=0 && na[J[a]]!=0)))
57                {  c=cost[a]+pi[I[a]]-pi[J[a]];
58,59              if (na[J[a]]==0 && flow[a]<hi[a]) del = min(del,c);
60,61              if (na[J[a]]!=0 && flow[a]>lo[a]) del = min(del,-c);
                 } // XD:
              }
62            if (del==inf && (flow[aok]==hi[aok] || flow[aok]==lo[aok]))
63               del = abs(cok);
64,65         if (del==inf) return; // GOTO 'IS-INFEASABLE'
66            for (n=1;n<=nodes;n++)
67,68            if (na[n]==0) pi[n]=pi[n]+del;
         } // end of while - NODE 6
}
```

Eigenständigkeitserklärung

Hiermit versichere ich, die vorliegende Arbeit eigenständig erstellt und nur die angegebenen Quellen und Hilfsmittel verwendet zu haben.

Berlin, den

André Baresel

Einverständniserklärung

Ich erkläre mich damit einverstanden, daß ein Exemplar meiner Diplomarbeit in der Institutsbibliothek der Humboldt-Universität zu Berlin aufgestellt wird.

Berlin, den

André Baresel