(19)

REPUBLIK
ÖSTERREICH
Patentamt

(10) Nummer: **AT 411 802 B**

# (12) PATENTSCHRIFT

(21) Anmeldenummer: A 861/2001
(22) Anmeldetag: 01.06.2001
(42) Beginn der Patentdauer: 15.10.2003
(45) Ausgabetag: 25.05.2004

(51) Int. Cl.$^{7}$: **G06F 17/60**

(56) Entgegenhaltungen:
HARTMAN J. ET AL., UML-BASED INTEGRATION TESTING, IN: PROCEEDINGS OF THE INTERNATIONAL SYMPOSIUM ON SOFTWARE TESTING AND ANALYSIS, 2000 PORTLAND, OREGON, US, PP. 60 - 70. ISBN: 1-58113-266-2 TEXTRONIX K1297 PROTOCOL TESTER, K1297-G20 BASE SOFTWARE VON 06/2001 UND FRÜHERE VERSIONEN MEMON A.M. ET AL., USING A GOAL-DRIVEN APPROACH TO GENERATE TEST CASE FOR GUIS, IN PROCEEDINGS OF THE 21ST INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 1999, LOS ANGELES, CALIFORNIA, US, PP. 257 – 266 MEMON A.M. ET AL., AUTOMATED TEST ORACLES FOR GUIS IN: PROCEEDINGS OF THE EIGHT INTERNATIONAL SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING FOR TWENTY-FIRST CENTURY APPLICATIONS, 2000, SAN DIEGO CALIFORNIA, US, PP. 30 - 39

(73) Patentinhaber:
SIEMENS AG ÖSTERREICH
A-1210 WIEN (AT).

(72) Erfinder:
BEER ARMIN
BADEN, NIEDERÖSTERREICH (AT).
MANZ JOACHIM
MÜNCHEN (DE).
MOHACSI STEFAN
WIEN, NIEDERÖSTERREICH (AT).
STARY CHRISTIAN
WIEN (AT).

## (54) VERFAHREN ZUM TESTEN VON SOFTWARE

(57) Die Erfindung betrifft eine Verfahren zum automatisierten Testen von Software, welche eine grafische Benutzeroberfläche aufweist. Mit zumindest einem grafischen Editor wird zumindest das dynamische und das semantische Verhalten der Benutzeroberfläche der Software spezifiziert. Von einer der Testfallgenerator-Software werden an Hand des so spezifizierten Verhaltens der Benutzeroberfläche Testfälle generiert, welche unmittelbar anschließend oder in einem abgesetzten Schritt von einer Software zur automatischen Testausführung ausgeführt werden.

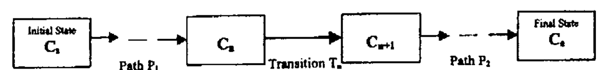


Fig. 12

AT 411 802 B

Die Erfindung betrifft ein Verfahren zum automatisierten Testen von Software, welche eine grafische Benutzeroberfläche aufweist, wobei eine auf einem Datenverarbeitungsgerät ausführbare Testfallgenerator-Software verwendet wird, mittels welcher Testfälle generiert und diese mit einer Software zur automatischen Testausführung auf einem Datenverarbeitungsgerät überprüft werden.

5 Weiters betrifft die Erfindung ein Verfahren zum Testen von Software mit einer grafischen Benutzeroberfläche, wobei mit einer Software zur automatischen Testausführung auf einem Datenverarbeitungsgerät Testfälle überprüft werden, welche mit einer Testfallgenerator-Software erzeugt werden, wobei zum Testen eines Überganges zwischen zwei Zuständen der Benutzeroberfläche der zu testenden Software zumindest ein Testfall erzeugt wird, welcher den entsprechenden Über-
10 gang enthält.

Schließlich betrifft die Erfindung auch noch ein Verfahren zur Ermittlung eines Pfades zu einem vorgebbaren Übergang in einem erweiterten Zustandsdiagramm beispielsweise bei einer Software mit einer grafischen Benutzeroberfläche. Testen ist im allgemeinen eine Tätigkeit mit dem Ziel, Fehler in einer Software zu finden und Vertrauen für deren korrekte Arbeitsweise zu bilden. Der
15 Test ist eine der wichtigsten Qualitätssicherungs-Maßnahmen in der Softwareentwicklung. Im Software-Entwicklungsprozess wird der Test jedoch häufig hinsichtlich Zeit, Kosten und Systematik unterschätzt.

Der Entwurf von effektiven Testfällen, d.h. solchen, die

- kundenrelevante Fehler finden,
20 • wahlweise eine mehr oder weniger vollständige Abdeckung des Testlings ermöglichen,
- auch komplexe Testszenarien enthalten, deren Erstellung nicht nur viel Vorbereitungszeit, sondern auch teures und seltenes Expertenwissen erfordert, und
- auch für automatische Regressionstest einsetzbar sind,

ist eine sehr anspruchsvolle und geld- und zeitaufwendige Tätigkeit.

25 Ein Testfall ist nach IEEE90 folgendermaßen definiert: "A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path to verify compliance with a specific requirement." Die Möglichkeit, eine derart anspruchsvolle Tätigkeit mit einem Softwaretool durchführen zu können, ist daher für jedes Software-Entwicklungsprojekt aus den drei bekannten Schlüsselaspekten Funktionalität, Termintreue und
30 Kosten von herausragender Bedeutung.

Semantik-orientierte Testszenarien gewährleisten den fehlerfreien Ablauf der vereinbarten Funktionalität entsprechend den von Kunden gewünschten Abläufen. Die Erzeugung und Ablauffähigkeit der Testszenarien über ein Softwaretool tragen erheblich zur Terminsicherheit bei und sparen darüber hinaus Entwicklungskosten.

35 Eine weitere Schwäche bei der Freigabe vieler Software-Projekte besteht darin, dass am Ende einer oft mehrjährigen Entwicklungszeit nicht mehr transparent ist, inwieweit das freigegebene Produkt die zu Beginn verabredeten und in den Spezifikationen dokumentierten Eigenschaften überhaupt erfüllt. Das heißt, es fehlt eine Brücke zwischen Design- und Test-Dokumentation, wodurch genaue Qualitätsaussagen über das auszuliefernde Produkt erschwert oder vielfach
40 unmöglich gemacht werden.

Zur Zeit werden verschiedenen Verfahren zum Testen von Software verwendet, beispielsweise StP-T *(Poston R.M., Automated Testing from object models; Comm. of the ACM, Sept. 1994, Vol.* 37, *No. 9, pp. 48 - 58)* oder Rational Test Factory *(Rational, User Manual, TestFactory, 1999).* Bei diesen Verfahren laufen allerdings komplizierte Prozesse mit abwechselnd manuellen und automa-
45 tisierten Tätigkeiten ab.

Bei anderen Softwaretools wie etwa Mockingbird *(Wood J., Automatic Test Generation Software Tools; Siemens Corporate Research, Technical Report 406, Dec. 1992)* können keine ausführbaren Testfälle generiert werden, hingegen werden bei dem Tool Titan *(Wood J., Automatic Test Generation Software Tools; Siemens Corporate Research, Technical Report 406, Dec. 1992)*
50 aus einem Testschema und Testmatrizen Testdaten generiert. Allerdings sind diese Verfahren zu wenig benutzerfreundlich, um sie bei komplexen Softwaresystemen erfolgreich einzusetzen.

Zur Generierung von Testfällen werden ebenfalls verschiedene Methoden verwendet, beispielsweise die Testfallgenerierung mit Hilfe von Suchmechanismen der Künstlichen Intelligenz, wobei der Backtracking-Mechanismus von PROLOG verwendet wird. Eine andere Methode be-
55 steht in der Erzeugung von einzelnen Zustands-Übergangs-Sequenzen aus einem komplexen

Zustands-Übergangs-Graphen mit Zyklen von einem Start-Zustand bis zu einem Ziel-Zustand, wobei die Zustandswechsel durch Benutzereingaben getriggert werden. Nachteilig an diesen bekannten Verfahren ist insbesondere, dass diese mit dem Problem der großen Anzahl von redundanten Testfällen kämpfen. Außerdem fehlen intelligente Algorithmen für die Testfallgenerierung, mit denen neben dem Generieren von "Gutfällen" auch das Generieren von "Schlechtfällen" sowie das Aufdecken spezifischer Fehler möglich ist.

Als allgemeiner Stand der Technik weiters bekannt sind die Dokumente

[1] HARTMAN Jean, IMOBERDORF Claudio, MEISINGER Michael, UML-based integration testing, in: Proceedings of the international Symposium on Software Testing and Analysis, 2000, Portland, Oregon, US. pp. 60-70. ISBN: 1-58113-266-2,

[2] Tektronik K1297 Protocol tester, K1297-G20 Base Software von 06/2001,

[3] MEMON Atif M, POLLACK Martha E., SOFFA Mary Lou. Using a goal-driven approach to generate test case for GUIs, in: Proceedings of the 21st international conference on Software engineering, 1999, Los Angeles, California, US. pp. 257-266,

[4] MEMON Atif M, POLLACK Martha E., SOFFA Mary Lou. Automated test oracles for GUIs, in: Proceedings of the eight international symposium on Foundations of software engineering for twenty-first century applications, 2000, San Diego, California, US. pp. 30-39.

Die Dokumente [1] und [2] befassen sich allerdings nicht mit dem Test grafischer Benutzeroberflächen und stellen lediglich einen allgemeinen Stand der Technik da. Insbesondere offenbart das Dokument [1] ein Verfahren zum Testen von Corba/COM („Common Object Request Broker Architecture"), während sich Dokument [2] sowie ältere Versionen dieses Dokumente auf das Testen von Protokollen beschränken.

Die Dokumente [3] und [4] behandeln das Testen von grafischen Benutzeroberflächen, eine Vermeidung der beschriebenen Nachteile ist aber auch mit den hier beschriebenen Verfahren und Applikationen nicht möglich.

Nach dem oben gesagten ist es eine Aufgabe der Erfindung, Verfahren anzugeben, mit denen ein benutzerfreundliches Testen von Software möglich ist, bei denen obige Nachteile vermieden werden.

Weiters ist es eine Aufgabe der Erfindung, Design- und Testprozesse auch in großen Projekten unter hohem Termin- und Kostendruck zu ermöglichen.

Diese Aufgaben werden mit einem eingangs erwähnten Verfahren zum automatisierten Test von Software dadurch gelöst, dass erfindungsgemäß

a) mit zumindest einem Editor zumindest das dynamische und das semantische Verhalten der Benutzeroberfläche der Software spezifiziert wird, wobei als Editor ein grafischer Editor verwendet wird, und

b) von der Testfallgenerator-Software an Hand des so spezifizierten Verhaltens der Benutzeroberfläche Testfälle generiert werden, welche unmittelbar anschließend oder in einem abgesetzten Schritt

c) von der Software zur automatischen Testausführung ausgeführt werden.

Durch die Verwendung eines grafischen Editors kann auf äußerst benutzerfreundliche Art und Weise das Verhalten der Benutzeroberfläche der zu testenden Software spezifiziert werden.

In [4] wird zwar ein Verfahren zur Simulation von Benutzeroberflächen beschrieben, eine Testfallgenerierung ist in diesem Dokument aber nicht beschrieben. Eine solche Generierung von Testfällen ist allerdings zentraler Bestandteil der vorliegenden Anmeldung.

Ebenso werden in Dokument [3] keine automatisch ausführbaren Testfälle generiert, wie dies bei der vorliegenden Anmeldung gemäss Anspruch 1 beansprucht wird, und ebenso werden zur Spezifizierung des Verhaltens der Oberfläche der zu testenden Software keine grafischen Editoren verwendet.

Zweckmäßigerweise werden vor Schritt a) des erfindungsgemäßen Verfahrens statische Informationen der Benutzeroberfläche von dem Editor eingelesen. Üblicherweise werden dabei die statischen Informationen mittels einer Bildschirmanalyse-Software oder aus einer Ressourcendatei eingelesen.

Dabei umfassen die statischen Informationen zumindest ein Layout und/oder Attribute der Elemente der grafischen Benutzeroberfläche.

Um das erfindungsgemäße Verfahren flexibel zu gestalten und Eingriffe von einem Benutzer im

Sinne eines möglichst effektiven Tests zu erlauben, können die statischen Informationen hinsichtlich des Layouts und/oder der Attribute von einem Benutzer ergänzt werden.

Besonders benutzerfreundlich lässt sich das Verfahren nach der Erfindung gestalten, wenn das dynamische Verhalten der Software/Benutzeroberfläche über die Eingabe von Zustandsübergängen spezifiziert wird, insbesondere wenn die Zustandsübergänge mittels grafischer Symbole dargestellt werden.

Auf diese Weise hat man das originalgetreue Bild eines Dialogs vor sich, und es können die einzelnen Zustandsübergänge beispielsweise durch Zeichnen von Pfeilen besonders einfach definiert werden.

Bei einer besonders vorteilhaften Ausführungsform der Erfindung werden die Zustandsübergänge mit semantischen Bedingungen und/oder mit syntaktischen Bedingungen verknüpft, und zur Spezifizierung des dynamischen Verhaltens der Benutzeroberfläche müssen nur die Zustandsübergänge gezeichnet werden, bei denen die Anreize mit syntaktischen bzw. semantischen Bedingungen verknüpft sind.

Die nun in Form eines Zustand-Übergangs-Diagrammes vorliegende formale Spezifikation beschreibt das dynamische Verhalten der Benutzeroberfläche in exakter Form und ist der Input für einen Testfallgenerator.

Ein Testfall-Generierungsalgorithmus sucht - wie weiter unten beschrieben - nach passenden Wegen im Zustands-Übergangs-Graph, wobei alle Elemente der grafischen Benutzeroberfläche von der Testfallgenerator-Software zumindest einmal angesprochen werden und alle von semantischen und/oder syntaktischen Bedingungen abhängenden Zustandsübergänge von der Testfallgenerator-Software mit zumindest einem richtigen und zumindest einem falschen Übergangswert abgedeckt werden.

Weiters werden die oben angesprochen Aufgaben mit einem eingangs erwähnten Verfahren zum Testen von Software mit einer grafischen Benutzeroberfläche dadurch gelöst, dass erfindungsgemäß zur Erzeugung des zumindest einen Testfalls

a) ein erster Pfad von Übergängen erzeugt wird, welcher bei einem Ausgangszustand der Benutzeroberfläche startet und in einem Zwischenzustand endet, wobei der Zwischenzustand ein Zustand ist, welcher alle für den zu überprüfenden Übergang notwendigen Eingabebedingungen erfüllt, und
b) zumindest ein weiterer Pfad von Übergängen erzeugt wird, der in dem von dem zu testenden Übergang erzeugten Zustand beginnt und im Endzustand der grafischen Benutzeroberfläche endet, und
c) die beiden Pfade mit dem Übergang miteinander verbunden werden.

Zweckmäßigerweise wird dabei der erzeugte Testfall in einer Testfall-Datenbank gespeichert.

Dabei erfolgt die Erzeugung eines Pfades zu einem vorgebbaren Übergang mit einem eingangs erwähnten Verfahren, bei dem gemäß der Erfindung

a) zumindest ein Satz von erlaubten Eingabebedingungen ermittelt wird, für den der zu testende Übergang ausführbar ist,
b) für alle Variablen, von denen die Eingabebedingungen abhängen, geeignete Werte ermittelt werden, sodass alle Eingabebedingungen erfüllt sind, und für jede Variable, von der die Bedingung abhängt, beginnend bei einer ersten Variable
c) zumindest ein Übergang gesucht wird, welcher die Variable auf den gewünschten Wert setzt, anschließend der Zustand des Zustandsdiagramms auf einen dem Wert der geänderten Variable entsprechenden Wert verändert wird und
d) Punkt c) für die nächste Variable der Bedingung durchgeführt wird.

Bei einer Ausführungsform der Erfindung wird der Pfad durch Aufrufen einer Suchfunktion ermittelt.

Günstigerweise wird dabei im Falle der Übereinstimmung des gegenwärtigen Zustandes des Zustandsdiagramms der Benutzeroberfläche mit einem Satz von erlaubten Eingabebedingungen kein Pfad erzeugt.

Bei einer besonders vorteilhaften Ausführungsform der Erfindung weisen die Variablen eine vorgebbare Reihenfolge auf und die Variablen werden gemäß Punkt c) und d) in einer bestimmten Reihenfolge abgearbeitet.

Weiters wird in Schritt c) bei einem Übereinstimmen des Wertes einer Variable mit dem ge-

wünschten Wert mit der nächsten Variable fortgefahren, wobei, wenn in Schritt c) keine geeigneten Werte gefunden werden, ein Fehler ausgegeben wird.

Das erfindungsgemäße Verfahren erweist sich dadurch als besonders effektiv, weil im Falle, dass für eine Variable kein Übergang gefunden wird, zumindest zu der unmittelbar vorher abgearbeiteten Variable zurückgegangen, für diese ein neuer Übergang erzeugt, und anschließend wieder für die Variable ein Übergang nach Schritt c) gesucht wird. Außerdem wird zu jedem Übergang ein Pfad ermittelt.

Dabei wird bei einer konkreten Ausführungsform der Erfindung der Pfad durch rekursives Aufrufen der Suchfunktion ermittelt.

Weiters wird für den Fall, dass kein Pfad zu dem Übergang gefunden wird, ein anderer Übergang ermittelt.

Außerdem wird im Falle eines Auffindens eines Pfades noch überprüft, ob durch den Pfad eine oder mehrere bereits auf einen gewünschten Wert gesetzte Variablen verändert werden, und im Falle der Veränderung von zumindest einer Variable durch einen Pfad wird ein neuer Pfad zu dem Übergang gesucht.

Schließlich wird noch im Falle des Nicht-Auffindens einer Lösung die Reihenfolge für die Abarbeitung der Variablen geändert, bzw. im Falle des Nicht-Auffindens einer Lösung in Schritt b) werden andere Variablen gesucht.

Abschließend wird ein ermittelter Pfad einem Ergebnispfad hinzugefügt und nach Hinzufügen aller Pfade wird der Ergebnispfad ausgegeben.

Notwendig ist es dann noch, einen Pfad zu einem Endzustand des Zustandsdiagramms zu ermitteln. Dazu wird gemäß dem erfindungsgemäßen Verfahren ein Übergang gesucht, welcher die Anwendung unmittelbar beendet, und ausgehend von einem gegenwärtigen Zustand des Zustandsdiagramms wird ein Pfad zu dem Übergang gesucht.

Zweckmäßigerweise wird dabei für den Fall, dass der gegenwärtige Zustand der Anwendung der Endzustand ist, kein Pfad gesucht wird.

Im folgenden ist die Erfindung an Hand der Zeichnung näher erläutert. In dieser zeigen die

- Fig. 1 einen prinzipiellen Ablauf eines erfindungsgemäßen Verfahrens,
- Fig. 2 eine beispielhafte Ansicht eines Fensterhierarchie-Editors zum Editieren der statischen Informationen einer grafischen Benutzeroberfläche,
- Fig. 3 eine beispielhafte Ansicht eines Fenstereigenschaften-Editors zum Editieren der statischen Informationen eines Fensters einer grafischen Benutzeroberfläche,
- Fig. 4 eine beispielhafte Ansicht eines Fenster-Editors zum Editieren der dynamischen Informationen eines Fensters einer grafischen Benutzeroberfläche, weiters
- Fig. 5 eine Ansicht eines Menü-Editors zum Editieren der dynamischen Informationen des Menüs eines Fensters einer grafischen Benutzeroberfläche,
- Fig. 6 eine Ansicht eines Bedingungs-Editors zum Editieren der semantischen Informationen einer grafischen Benutzeroberfläche, weiters
- Fig. 7 eine Ansicht eines Aktions-Editors zum Editieren von semantischen Informationen einer grafischen Benutzeroberfläche,
- Fig. 8 eine beispielhafte Eingabemaske zur Testfallerzeugung mit einem Softwaretool basierend auf dem erfindungsgemäßen Verfahren,
- Fig. 9 ein beispielhaftes Ausgabefenster für Testfälle, wobei auch ein nachträgliches Editieren der Testfälle möglich ist,
- Fig. 10 ein Beispiel für ein mit einer Software zur automatischen Testausführung erzeugtes Ausgabefile, welches an Hand von mit dem erfindungsgemäßen Verfahren erzeugten Testfällen generiert wurde,
- Fig. 11 ein Beispiel für die Erzeugung eines Testfalles,
- Fig. 12 ein Beispiel einer Struktur eines Testfalles,
- Fig. 13 ein Beispiel einer Struktur eines Funktionsaufrufes,
- Fig. 14 ein Beispiel für einen Bedingungsbaum,
- Fig. 15 eine beispielhafte Ansicht eines Login-Fensters einer grafischen Benutzeroberfläche, und
- Fig. 16 eine Struktur der Funktionsaufrufe bei einer beispielhaften Testfallgenerierung.

Im folgenden ist an Hand der Figuren 1 - 16 das erfindungsgemäße Verfahren und eine entsprechend angepasste Software zum Durchführen des Verfahrens eingehend erläutert. Die Fig. 1

zeigt dabei den grundsätzlichen Ablauf des Verfahrens, wobei der erfindungswesentliche Anteil der mit IDATG bezeichnete Bereich ist. Gemäß der Fig. 1 wird vorerst eine zu testende grafische Benutzeroberfläche (GUI) hinsichtlich ihrer statischen Eigenschaften beschrieben, beispielsweise unter Verwendung einer entsprechenden Software, wie etwa einem sogenannten "GUI Builder".
5 Diese statischen Informationen werden dann in einer Ressourcendatei abgespeichert. Eine andere Möglichkeit besteht darin, die statischen Informationen mittels einer Bildschirmanalyse-Software ("GUI Spy") zu ermitteln. Eine eingehende Erklärung der statischen Struktur einer GUI erfolgt weiter unten.

Die in der Ressourcendatei gespeicherten bzw. mit der Bildschirmanalyse-Software eingelese-
10 nen statischen Informationen werden nun in die verwendete erfindungsgemäße Software IDATG eingelesen, mit den dynamischen und semantischen Informationen über die GUI ergänzt, und an Hand all dieser Informationen werden wie weiter unten noch eingehend erläutert, Testfälle generiert, die schließlich mit einem entsprechenden Programm, wie etwa "WinRunner", ausgeführt werden können.

15 Die Figuren 2 - 8 zeigen verschiedene Editoren und Dialogfenster zum Beschreiben der grafischen Benutzeroberfläche, worauf im folgenden noch im Detail eingegangen wird. Zum nachträglichen Editieren der statischen Informationen der GUI wird dabei ein Fensterhierarchie-Editor wie beispielhaft in Fig. 2 gezeigt verwendet, in dem die Fensterhierarchie der GUI als ein Baum angezeigt wird. Diese Hierarchie kann dann mittels Drag and Drop mit dem Editor bearbeitet werden.

20 Die Figuren 3 und 4 zeigen einen Fenstereigenschaften-Editor zum Editieren der statischen Informationen eines Fensters einer grafischen Benutzeroberfläche sowie einen Fenster-Editor zum Editieren der dynamischen Eigenschaften eines Fensters der grafischen Benutzeroberfläche. Mit den in Fig. 4 gezeigten Pfeilen des grafischen Editors wird das dynamische Verhalten des OK-Buttons beschrieben. Ist dabei die Benutzereingabe korrekt, so springt der Fokus zurück auf
25 das erste Feld, in diesem Fall "Name", und es kann in der Eingabemaske eine neue Person mit den entsprechenden Daten eingegeben werden. Findet hingegen eine falsche Eingabe statt, beispielsweise ein negatives Alter im Feld "Age", wird eine entsprechende Nachricht in einem Fehlermeldungsfenster ausgegeben.

Mit dem in Fig. 5 gezeigten Menü-Editor können Menüs einer GUI editiert werden, und es kön-
30 nen Übergänge gezeichnet werden, die durch Anwählen des entsprechenden Menüeintrages ausgelöst werden (in dem gezeigten Beispiel wird durch Anwählen von "Close" eine Datei geschlossen und je nachdem ob die Datei zuvor editiert wurde oder nicht, zu verschiedenen Fenstern verzweigt).

Mit dem in Fig. 6 gezeigten Bedingungs-Editor können außerdem noch semantische Informati-
35 onen editiert werden, z.B., dass bei Eingabe eines Mädchennamens (#MaidenName#) das angegebene Geschlecht nicht männlich sein darf.

Die Fig. 7 zeigt einen Aktions-Editor, auf den weiter unten noch eingegangen wird, und Fig. 8 zeigt ein typisches Auswahlfenster der verwendeten Software, dem zu entnehmen ist, dass bei einer erprobten Ausführungsform der Erfindung zwei Arten von Testfällen generiert werden kön-
40 nen, nämlich ein Übergangstest für Übergänge zwischen bestimmten Übergängen und ein Syntax-Test für Eingabefelder.

Schließlich zeigt die Fig. 9 noch einen Ausgabeeditor für erzeugte Testfällen, wobei mit diesem Editor die Testfälle nachträglich noch editiert und auch noch eigene, beispielsweise manuell erstellte Testfälle hinzugefügt werden können, und Fig. 10 zeigt das Ergebnis eines mit einer Software
45 zur automatischen Testausführung - beispielsweise WinRunner - ausgeführten Tests an Hand von mit dem erfindungsgemäßen Verfahren erzeugten Testfällen.

Ein einfaches Beispiel für die Generierung eines Testfalls ist in Fig. 11 dargestellt. Ziel des Testfalls ist es, den "Delete"-Knopf einer Anwendung zu testen. Dieser ist jedoch nur aktiv, wenn zuvor ein Datensatz mittels "Search" gesucht wurde. Das Suchen wiederum ist nur möglich, wenn
50 zuvor ein Name als Suchbegriff eingegeben wurde. All diese Informationen wurden zuvor als Bedingungen und Aktionen spezifiziert. Der Generierungsalgorithmus ist in der Lage, aus diesen Informationen Schritt für Schritt einen korrekten Testfall zu erstellen.

Zum näheren Verständnis der Erfindung soll im folgenden eine grafische Benutzeroberfläche an Hand formaler Ausdrücke beschrieben werden.

55 Grafische Benutzeroberflächen bestehen aus Objekten, den sogenannten "Fenstern" oder

"Windows". Es existieren verschiedene Definitionen für den Begriff "Fenster". In der folgenden Beschreibung werden sämtliche GUI-Objekte als "Fenster" bezeichnet, d. h. unabhängig von der tatsächlichen Position des Fensters in der Hierarchie der grafischen Benutzeroberfläche werden auch Dialoge, Knöpfe, Eingabefelder und auch statischer Text als Fenster bezeichnet werden. Jedem Fenster ist dabei eine eindeutige ID zugeordnet. Eine grafische Benutzeroberfläche kann somit als Menge von Fenstern beschrieben werden: GUI_Objects = $\{W_1, W_2, ...W_n\}$, wobei die $W_i$ die entsprechenden Fenster-IDs darstellen.

Jedes Fenster kann durch einen Satz von Eigenschaften beschrieben werden, die im folgenden als "Designatoren" bezeichnet werden und immer von einem '#' Charakter eingeschlossen sind. Man kann dabei zwischen drei Grundtypen unterscheiden:

- Designatoren, die von der Klassenbibliothek der GUI vordefiniert werden. Diese beinhalten Strings wie die ID oder die caption, Zahlen wie die Koordinaten und Boolsche Werte, die angeben, ob das Fenster aktiv ist oder nicht. Die Namen dieser Designatoren haben folgendes Muster: #WindowID:$PropertyName#. z. B., #IDOK:$Enabled#. Das Zeichen '$' wird benutzt um vordefinierte Designatoren von anderen Typen zu unterscheiden.
- Manche Fenster-Typen akzeptieren Benutzereingaben, die als Inhalt des Fensters angezeigt werden. Beispielsweise können Eingabefelder Strings oder Zahlen enthalten, Checkboxen können Boolsche Werte enthalten. Um diese Werte anzusprechen, ist die Fenster-ID ausreichend, z. B. #IDC_NAME#.
- Zusätzlich kann ein Benutzer zusätzliche Designatoren für ein Fenster definieren, um bestimmte anwendungsspezifische Eigenschaften zu beschreiben. Beispielsweise kann ein Dialog verschiedene Modi aufweisen, beispielsweise einen Modus für die Erstellung eines neuen Datensatzes und einen anderem zum Editieren eines bestehenden Datensatzes. In diesem Fall ist es beispielsweise für den Benutzer zweckmäßig, einen neuen Boolschen Designator zu definieren, der den gegenwärtigen Modus angibt. Als Syntax wird dabei folgendes Muster verwendet: #WindowID:PropertyName#. z.B., #IDD_HUMAN:Mode#. In diesem Fall enthält der Eigenschaftsname kein '$'.

Ein Fenster W wird nun von einem n-Tupel von Designatoren (Eigenschaften) definiert: $W = (D_1, D_2, ...D_n)$. Die Anzahl und die Typen der Designatoren hängen von der für die GUI benutzten Klassenbibliothek und weiteren applikationsspezifischen Eigenschaften ab. Dieses n-Tupel von Designatoren beschreibt einen momentanen Zustand, da sich die Werte der Designatoren dynamisch ändern können, wenn die Applikation ausgeführt wird. Beispielsweise kann ein Anwender den Inhalt eines Fensters oder dessen Größe verändern.

Nachdem die GUI ausschließlich aus Fenstern besteht und jeder Zustand mit einem Tupel repräsentiert werden kann, kann auch der gesamte Zustand der GUI als eine Kombination C all dieser Tupel beschrieben werden:

$C = (W_1, W_2, ...W_n) = (D_{1,1},D_{1,2}, ...D_{1,m},D_{2,1},D_{2,2}, ...D_{2,p},D_{n,1},D_{n,2}, ...D_{n,q})$.

Der Anfangszustand der GUI wird als Start-Kombination $C_s$ bezeichnet, er enthält alle Anfangswerte der GUI-Designatoren.

Bei Beendigung der GUI-Anwendung existiert kein Fenster mehr, die End-Kombination ist leer: $C_e = ()$.

**Statische Struktur einer GUI**

Jedes Fenster einer GUI kann eine unbegrenzte Anzahl sogenannter "Kindfenster" aufweisen. Umgekehrt hat jedes "Kindfenster" exakt ein "Elternfenster", bzw. im Fall eines Top-Level-Fensters kein Elternfenster. Die Vater-Kind-Relation R zwischen zwei Fenstern mit den IDs p und c kann folgendermaßen definiert werden: pRc, wobei p das Elternfenster von c ist. Zyklen wie R = {(a,b), (b,c), (c,a)} sind nicht erlaubt. Somit sind die Fenster einer GUI hierarchisch in der Form eines Baumes angeordnet. Eigentlich handelt sich dabei mehr um einen Wald als um einen Baum, da mehrere Unterbäume existieren können, die nicht miteinander verbunden sind.

Die semantischen Zusammenhänge einer Eltern-Kind-Beziehung sind die folgenden: Ein Kind kann nur existieren, wenn auch sein Vater existiert. Ebenso kann ein Kind nur aktiviert werden, wenn auch der Vater aktiviert ist. Hingegen kann natürlich der Vater existieren, ohne dass die Existenz des Kindes notwendig ist. Außerdem ist es nicht möglich, dass ein Kind seinen Vater dynamisch ändert.

**Verhalten einer GUI**

Unter Verwendung von Kombinationen kann das Verhalten einer GUI als eine Maschine endlicher Zustände (Zustandsautomat) ausgedrückt werden. Allerdings muss berücksichtigt werden, dass die Anzahl der möglichen Zustände auch für kleine GUI's sehr groß werden kann und somit die Darstellung in einem gewöhnlichen Zustands-Übergangs-Diagramm praktisch unmöglich macht. Aus diesem Grund ist es notwendig, dass einige Erweiterungen des Konzepts eines Zustandautomaten gemacht werden, um diese Komplexität behandeln zu können.

In den vorangegangenen Absätzen wurde die Beschreibung der Momentanzustände einer GUI erörtert. Weiters ist es noch notwendig, die Zustandsänderungen zu beschreiben, die während der Ausführung einer GUI auftreten. Diese Zustandsänderungen werden als Übergänge (T) bezeichnet und werden durch eine Benutzereingabe oder ein internes Ereignis ausgelöst. Ein Übergang ist ein 3-Tupel $T=(E, S, \tau)$, welches beinhaltet

- das Ereignis E welches den Übergang auslöst,
- einen Satz S von richtigen (gültigen) Eingabekombinationen für diesen Übergang, und
- eine Funktion $\tau(C) \rightarrow C$, die für jede gültige Eingabekombination definiert ist. Diese transformiert die Eingabekombination in eine neue Kombination.

Beispiel: Wenn der Benutzer den OK-Button drückt (Ereignis wird ausgelöst) und alle Felder richtig ausgefüllt sind (Definition der richtigen Eingabezustände), sollte der Eingabefokus zurück auf das erste Eingabefeld zurückgehen (Definition der Transformationsfunktion).

Anstelle der Aufzählung aller gültigen Eingabekombinationen ist es gewöhnlich einfacher, den gültigen Satz von richtigen Eingabekombinationen mittels Bedingungen zu beschreiben. Demgemäß ist eine Kombination für einen bestimmten Übergang gültig, wenn alle Bedingungen für den Übergang von den Werten der Designatoren in der Kombination erfüllt sind. Andernfalls ist die Kombination ungültig. Üblicherweise haben nicht alle Designatoren direkten Einfluss auf die Bedingung.

Beispiel: die Bedingung #IDOK:$Enabled# = TRUE bezieht sich nur auf einen einzigen Designator der Kombination, die andern Werte sind nicht von Relevanz. Somit sind alle Kombinationen gültig, für welche #IDOK:$Enabled# = TRUE gilt.

Gleichfalls wirken sich die meisten Übergangsfunktionen nicht auf alle Werte der Eingabekombination aus. Somit kann eine Funktion einfacher durch eine Anzahl von elementaren Werteänderungen ausgedrückt werden, welche als "Aktionen" bezeichnet werden. Beispielsweise wirkt sich die Aktion SetAttribute(#IDOK:$Enabled#, TRUE) nur auf einen Designator der Kombination aus. In vielen Fällen hängen die Designatoren voneinander ab, was bedeutet, dass bei einem Setzen eines Designators auf einen neuen Wert auch ein oder mehrere andere Designatoren auf einen neuen Wert gesetzt werden. Wird beispielsweise ein Fenster geschlossen, so werden auch alle seine Kinder geschlossen.

Spezielle Sprachen sind notwendig, um Ereignisse, Bedingungen und Aktionen zu beschreiben. Sie sind im folgenden im Detail erklärt.

**Ereignissprache**

Jeder Übergang wird von einem Ereignis ausgelöst. Ein Übergang wird genau dann ausgeführt, wenn das Ereignis auftritt und alle Bedingungen für den Übergang erfüllt sind. Somit kann ein Ereignis als Vorbedingung für einen Übergang betrachtet werden. Der Unterschied zu den anderen Bedingungen liegt darin, dass Ereignisse momentan sind (sie weisen keine Dauer auf), während andere Bedingungen für eine gewisse Zeitdauer vorhanden sind.

Die Ereignisse, die bei einer GUI auftreten können, lassen sich in zwei Gruppen aufteilen:

- Ereignisse die von einem Benutzer ausgelöst werden, beispielsweise durch einen Mausklick oder das Drücken einer Taste der Tastatur
- Ereignisse die von dem System ausgelöst werden, wie das Signal eines Zeitgebers oder eine interne Nachricht.

Beide Typen hängen stark von der Hard- und Software des zu testenden Systems ab, beispielsweise von dem Layout des Keyboards oder vom Betriebssystem. Daher ist es kaum möglich, eine Sprache zu entwickeln, die alle möglichen Ereignisse eines Computersystems beschreiben kann. Die für die im Rahmen der Erfindung verwendete Software (IDATG) entwickelte Sprache deckt alle Keyboard- und Mausereignisse ab, die von einem Benutzer auf einem Personalcomputer

unter MS Windows ® ausgeführt werden, aber sie kann auf einfache Weise für andere Systeme adaptiert werden.

Jedes Benutzerereignis bezieht sich auf ein bestimmtes Fenster, für welches das Ereignis bestimmt ist. Die grundlegende Syntax für die Beschreibung eines Ereignisses ist Event/WindowID. Beispielsweise bezeichnet <MClickL>/IDOK einen Klick auf den linken Mausbutton, während der Mauscursor über dem OK-Button positioniert ist. Wenn keine Fenster-ID spezifiziert ist, nimmt die erfindungsgemäße Software an, dass das betroffene Fenster jenes ist, in welchem sich im Moment der Eingabefokus befindet. (Diese Information ist in der Eingabekombination des Übergangs enthalten.)

Die Ereignissprache macht keinen Unterschied zwischen Groß- und Kleinschreibung (<MClickL> und <mclickl> bedeuten dasselbe). Allerdings ist es wichtig, die korrekte Schreibweise in String-Konstanten, die in der getesteten Anwendung auftreten, zu verwenden (d.h. <select"ListItem"> bedeutet nicht dasselbe wie <select"listitem">. Ein Übergang ohne ein Auslöseereignis kann in seltenen Fällen notwendig sein und wird mit <> ausgedrückt.

Mittels der Tastatur ausgelöste Ereignisse weisen bei der erfindungsgemäßen Software die folgende Syntax auf: Wenn der Tastenname eine Länge von mehr als einem Zeichen aufweist, muss er in spitzen Klammern angegeben werden, beispielsweise <Enter>. Gruppen von Tasten, die gleichzeitig zu drücken sind, werden mit den selben spitzen Klammern eingeschlossen und sind durch Bindestriche getrennt, beispielsweise <Ctrl-Shift-F10>.

Eine detailliertere Darstellung soll hier nicht erfolgen, da dies für den Erfindungsgedanken von geringer Bedeutung ist, und es sind hier nur noch einige Beispiele angegeben, und zwar für

- Funktionen und Cursortasten:

    <Esc>, <F1> - <F12>, <PrtScr>, <ScrlLock>, <Pause>
    <Ins>, <Del>, <Home>, <End>, <PgUp>, <PgDn>
    <Left>, <Right>, <Up>, <Down>

- wichtige Tasten der Haupttastatur:

    <Backspace>, <Tab>, <CapsLock>, <Enter>, <Space>
    <Divide> (/), <Minus> (-), <Greater> (>), <Less> (<)

- spezielle Tasten (normalerweise reicht der einfache Name. Falls es von Bedeutung ist, ob die rechte oder die linke Taste zu drücken ist, wird L oder R hinzugefügt):

    <Shift>, <ShiftL>, <ShiftR>
    <Ctrl>, <CtrlL>, <CtrlR>
    <Alt>, <AltL>, <AltR>, <AltGr> (auf deutschen Tastaturen)
    <Win>, <WinL>, <WinR>, <Menu> (Zusatztasten beispielsweise für Win95/98);

Mittels der Maus ausgelöste Ereignisse werden ebenfalls in spitzen Klammern geschrieben:

| | |
|---|---|
| <MClickL> | Klick mit linker Maustaste |
| <MClickR> | Klick mit rechter Maustaste |
| <MDblClickL> | Doppelklick mit linker Maustaste |
| <MDblClickR> | Doppelklick mit rechter Maustaste |
| <MPressL> | Drücken und Halten der linken Maustaste |
| <MPressR> | Drücken und Halten der rechten Maustaste |
| <MReleaseL> | Loslassen der linken Maustaste |
| <MReleaseR> | Loslassen der rechten Maustaste |
| <MMove> | Bewegen der Maus |

**Bedingungssprache**

Zur Definition eines Satzes gültiger Eingabekombinationen für einen Übergang sind Bedingungen notwendig. Ein solcher Satz gültiger Eingabekombinationen wird implizit durch eine Spezifizierung bestimmter Einschränkungen für einige oder alle Designatoren in der Kombination definiert, wodurch der Satz möglicher GUI-Zustände auf jene Zustände, die für den Übergang gültig sind, eingeschränkt wird. Im folgenden soll kurz auf die notwendige Syntax zur Beschreibung solcher Bedingungen eingegangen werden.

Bei der verwendeten, auf dem erfindungsgemäßen Verfahren basierenden Software, werden sowohl Groß- als auch Kleinbuchstaben akzeptiert, Abstände zwischen Operatoren und Operanden können, müssen aber nicht verwendet werden. Ebenso können Klammern verwendet werden, diese sind aber nur notwendig, um die Priorität der Operatoren zu ändern.

Die Sprache beachtet die mathematischen und logischen Prioritätsregeln. Ausdrücke werden in "Infix"-Notation geschrieben, was bedeutet, dass binäre Operatoren zwischen ihren beiden Operanden stehen und unäre Operatoren vor ihren Operanden stehen. Bedingungen müssen immer einen Boolschen Wert ausgeben, da Bedingungen nur TRUE oder FALSE sein können.

Der Sprache der verwendeten Software (IDATG) kennt vier grundlegende Typen von Werten, nämlich:

| | |
|---|---|
| NUM | ein Integer-Wert (32 Bit) |
| BOOL | in Boolscher Wert, der TRUE oder FALSE sein kann |
| STRING | ein String (maximale Länge = 255 Zeichen) |
| DATE | eine gültige Datumsangabe im Format TT.MM.JJJJ |

Operatoren, die von der Software akzeptiert werden, können in vier Klassen unterteilt werden:

- Logische Operatoren
  IDATG akzeptiert die Standard-Operatoren AND, OR, XOR und NOT. Dabei bedeutet OR ein inklusives Oder, welches TRUE ergibt, wenn zumindest einer seiner Operanden TRUE ist; XOR ist ein exklusives Oder, welches TRUE ergibt, wenn genau ein Operand TRUE und der andere FALSE ist.
- Vergleichsoperatoren
  Operanden können unter Verwendung der Operatoren =, !=, <, >, <= und >= verglichen werden. Während die letzten vier Operatoren nur für numerische Ausdrücke und Datumsangaben erlaubt sind, können das Gleich- (=) und das Ungleichzeichen (!=) für alle Datentypen verwendet werden. Die Software entscheidet dabei automatisch, ob ein mathematischer oder ein String-Vergleich durchgeführt wird.
- Numerische Operatoren
  Es können die grundlegenden Operatoren +, -, * und / verwendet werden.
- Spezielle Operatoren
  Der Operator SYN überprüft, ob der gegenwärtige Inhalt eines Feldes der spezifizierten Syntax desselben entspricht oder nicht. Der Operator erwartet die ID des Eingabefeldes als Argument. Ist für das Eingabefeld IDC_NAME beispielsweise die Syntax C2 (2 Zeichen) definiert, dann ergibt der Ausdruck SYN #IDC_NAME# TRUE, falls das Feld etwa "ab", enthält, und FALSE, wenn es "abc" beinhaltet.

Weiters existieren drei verschiedene Typen von Operanden:

- Konstante Werte: die Notation hängt von der Art des Wertes ab. NUM-Werte werden wie gewohnt geschrieben (45, -3), BOOLsche Werte können TRUE oder FALSE sein. STRING-Werte werden zwischen Anführungszeichen geschrieben ("text", "Jerry Steiner"). Ein Backslash (\) sorgt dafür, dass das nächste Zeichen als Text interpretiert wird (z.B. "te\"xt"). DATE-Werte werden als TT.MM.JJJJ geschrieben (z.B. 08.03.1994, 29.02.2000).
- Designatoren (Variablen): Designatoren können adressiert werden, indem der entsprechende Name zwischen '#'-Zeichen geschrieben wird (z.B. #IDC_NAME#). Dabei ist es wichtig, dass jede Variable genau den Typ aufweist, den der jeweilige Operator verlangt. Beispielsweise ist es nicht möglich, den Designator #IDOK:$Enabled# (Typ BOOL) mit dem konstanten Wert 5 (Typ NUM) zu vergleichen.
- Zusammengesetzte Ausdrücke: hinsichtlich der Komplexität von Ausdrücken existieren keine Beschränkungen; demgemäß ist es möglich, Operanden zu verwenden, die zusammengesetz-

te Ausdrücke sind und selbst Operatoren enthalten. Beispielsweise kann der BOOLsche Ausdruck '#IDC_NAME# = "Mr. Holmes"' mit jedem logischen Operator verwendet werden und '(#IDC_AGE# * 3) + 5' mit jedem numerischen.

5 **Aktionssprache**

Aktionen werden verwendet, um die Übergangsfunktion zu definieren, die eine Eingabekombination in eine Ausgabekombination umwandelt, wozu beispielsweise ein Aktions-Editor wie in der Fig. 7 gezeigt verwendet wird. Die einfachste Möglichkeit, um eine solche Funktion zu spezifizieren, besteht darin, einen Satz grundlegender Aktionen zu definieren, die jeweils einen einzelnen 10 Designator der Kombination ändern. Beispielsweise ändert SetAttribute(#IDOK:$Enabled#, TRUE) nur den Booleschen Designator #IDOK:$Enabled#.

Allerdings ist es oftmals komfortabler, komplexere Aktionen zu spezifizieren, die eine Veränderung von mehr als einem Designator verursachen. Diese Aktionen hängen von der Funktionalität der verwendeten Klassenbibliothek der GUI ab, da diese typische Prozesse der GUI beschreiben 15 sollen. Beispielsweise setzt die Aktion CallDialog(#IDD_HUMAN#) nicht nur den Designator #IDD_HUMAN:$Exist# auf TRUE, sondern auch den $Exist Designator aller Kinder des Dialogs. In diesem Fall ist es offensichtlich einfacher, eine einzige Aktion zu definieren, anstatt für jedes Kindfenster eine einzelne Aktion zu definieren.

Zusätzlich ist es noch wichtig, die Reihenfolge zu definieren, in welcher die Aktionen ausge-20 führt werden sollen, da es möglich ist, dass zwei Aktionen den Wert ein und desselben Designators festlegen. Außerdem ist es möglich, dass eine Aktion von einem Ergebnis einer vorangegangenen Aktion abhängt.

Grundsätzlich hat jede Aktion einen eindeutigen Namen und erwartet eine bestimmte Anzahl von Argumenten, sehr ähnlich einem Funktionsaufruf in einer Programmiersprache. Argumente 25 müssen mit dem vorgesehenen Typ übereinstimmen und können jeder von der Bedingungssprache abgedeckte Ausdruck sein. Damit wird es möglich, mit Aktionsargumenten auf Designatoren Bezug zu nehmen oder komplexe Ausdrücke zu verwenden. Beispiel: SetCheckBox(#IDC_PHD#, NOT #IDC_MBA#), SetInputField(#IDC_AGE#, 5*3+4).

Zusammenfassend lässt sich festhalten, dass zur formalen Beschreibung einer GUI folgende 30 Informationen notwendig sind: Ein Satz W, der alle Fenster der GUI enthält, eine Startkombination $C_s$, die den Anfangszustand für alle Eigenschaften der Fenster in W definiert, eine binäre Relation R auf W, die das Mutter-Kind-Verhältnis zwischen den Fenstern beschreibt sowie ein Satz von Übergängen T, welcher das dynamische Verhalten der GUI beschreibt. Somit kann eine GUI formal geschrieben werden als GUI = (W, $C_s$, R, T).

35

**ALGORITHMUS FÜR DIE TESTFALLGENERIERUNG**

**Testfallerzeugung**

Der folgende Abschnitt behandelt eine mögliche Anwendung der formalen GUI-40 Spezifikationssprache, nämlich die Testfallerzeugung.

Eine geordnete Sequenz von Übergängen $P=(T_1, T_2, \ldots T_n)$ wird als Pfad (P) bezeichnet, wenn sie die folgenden Bedingungen erfüllt:

$\forall\ i\geq 1;\ i\leq n\ \tau_i(C_i) = C_{i+1};\ C_i \in S_i$ (jeder Übergang produziert eine Kombination, die eine gültige Eingabe für den nächsten Übergang darstellt). $C_{n+1}$ ist die Ausgabekombination des Pfades. Somit 45 kann der Pfad auch als ein Meta-Übergang mit der Funktion $\varphi(C_1) = C_{n+1} = \tau_n(\tau_{n-1}(\ldots(\tau_2(\tau_1(C_1)))))$ angesehen werden.

Ein Pfad wird als Testpfad (TC) bezeichnet, wenn er im Ausgangszustand der GUI beginnt und im Endzustand der GUI endet, was bedeutet, dass die Anwendung beendet wird. $TC = (T_1, T_2, \ldots T_n)$, $C_1 = C_s$, $C_{n+1}= C_e$. Das Ziel der Testfallerzeugung (TCG) ist es, einen Satz von 50 Testfällen zu finden, der alle spezifischen Übergänge und somit die gesamte GUI abdeckt. Um einen speziellen Übergang zu testen, ist ein Algorithmus notwendig, der Testfälle findet welche diesen speziellen Übergang beinhalten.

**Auffinden eines Testfalls für einen bestimmten Übergang**

55 Zwei Pfade müssen gefunden werden, um einen Testfall zu finden, der einen speziellen Über-

gang $T_n = (E_n, S_n, \tau_n)$ enthält:

- Ein Pfad $P_1=(T_1, T_2, \dots T_{n-1})$, der im Ausgangszustand der GUI beginnt und in einem gültigen Eingangszustand für den zu testenden Übergang endet. $C_1 = C_s$, $C_n \in S_n$. Der Pfad kann leer sein wenn $C_s \in S_n$.
- Ein Pfad $P_2=(T_{n+1}, T_{n+2}, \dots T_m)$, der in jenem Zustand beginnt, der von dem zu testenden Übergang erzeugt wird, und welcher im Endzustand der GUI endet. $C_{n+1} \in S_{n+1}$, $C_{m+1}= C_e$. Der Pfad ist leer, wenn $C_{n+1} = C_e$.

Diese Situation ist in Fig. 12 dargestellt. Somit arbeitet der Erzeugungsalgorithmus wie folgt (in Pseudo-Code):

Funktion **GenerateTC**(Eingabe: Übergang $T_n$)

- Initialisieren der GUI-Variablen entsprechend $C_s$. Anmerkung: Der Zustand der getesteten GUI wird von diesen Variablen simuliert.
- Suchen des ersten Pfades $P_1$ durch Aufrufen der Funktion **SearchPathToTrans**($T_n$, $C_s$)
- Falls kein Pfad gefunden wird, wird ein Fehler ausgegeben (inkonsistente Spezifikation)
- Suchen des zweiten Pfades $P_2$ durch Aufrufen der Funktion **SearchPathToEnd**($C_{n+1}$)
- Falls kein Pfad gefunden wird, wird ein Fehler ausgegeben (inkonsistente Spezifikation)

**Auffinden eines Pfades zu einem bestimmten Übergang**

Die Funktion SearchPathToTrans versucht, einen Pfad $P_1$ zu finden, der im Anfangszustand der GUI beginnt und in einem Zustand endet, der die Ausführung des speziellen Übergangs $T_n$ erlaubt. Viele Graph-Suchalgorithmen gehen von einem Anfangszustand des Systems aus und versuchen über Zufallspfade den gewünschten Zustand zu erreichen. Die enorme Anzahl möglicher Kombinationen und Benutzereingaben macht es allerdings unmöglich, in einer vernünftigen Zeit unter Verwendung dieser Technik zu einem Ergebnis zu kommen. Somit ist ein Algorithmus notwendig, mit dem man im Stande ist, einen bestimmten Zustand systematisch zu erreichen, in dem alle Bedingungen für diesen Zustand einer nach dem anderen erfüllt werden.

Funktion **SearchPathToTrans**(Eingabe: Übergabe $T_n$, gegenwärtiger GUI-Zustand $C_i$)

1. Bestimmen des Satzes gültiger Eingabekombinationen $S_n$, oder anders ausgedrückt, der Bedingungen, die erfüllt werden müssen, damit $T_n$ ausgeführt werden kann
2. Wenn $C_i \in S_n$, ist kein Pfad notwendig => erfolgreicher Abschluß
3. Suchen von geeigneten Werten für alle Variablen, von denen die Bedingung abhängt, sodass die Bedingung TRUE wird. Das kann durch Aufrufen der Funktion **FulfillCondition**($S_n$, TRUE) erreicht werden.
4. Wenn keine Lösung gefunden wurde, wird ein Fehler ausgegeben
5. Für alle Variablen, von denen die Bedingung abhängt, wird folgendes durchgeführt:

    {

    6. Wenn der gegenwärtige Wert der Variable mit dem gewünschten übereinstimmt, wird mit der nächsten Variable fortgefahren
    7. Suchen eines Übergangs $T_x$, der die Variable auf den gewünschten Wert setzt
    8. Wenn kein Übergang gefunden wurde, wird Backtracking gestartet, indem zu der vorherigen Variable zurückgegangen wird
    9. Rekursiver Aufruf von **SearchPathToTrans**($T_x$, $C_i$), um einen Pfad zu $T_x$ zu finden
    10. Wenn keine Lösung gefunden wird, Suchen eines anderen Übergangs durch Zurückspringen auf Punkt 7
    11. Überprüfen, ob der neue Pfad Variablen verändert, die bereits in einem früheren Durchlauf gesetzt wurden
    12. Wenn ja, suchen eines neuen Pfades durch Zurückspringen auf Punkt 9
    13. Hinzufügen des neuen Pfades zum Ergebnispfad, entsprechendes Setzen von $C_i$ und Fortsetzen mit der nächsten Variable

    }

14. Wenn keine Lösung gefunden wurde, wird versucht, die Reihenfolge der Variablen zu ändern oder andere geeignete Variablen zu suchen, indem zu Punkt 3 zurückgesprungen wird
15. Ausgabe des Ergebnispfades

Wie zu sehen ist, verwendet der Algorithmus Backtracking, d.h. das Zurückwechseln zu einem früheren Programmzustand, wenn keine Lösung gefunden werden konnte. Außerdem kann, wenn

die Funktion ein unpassendes Ergebnis liefert, die Funktion erneut aufgerufen werden, um eine alternative Lösung auszugeben. Dies kann sooft wiederholt werden, bis keine alternativen Lösungen mehr existieren.

Eine andere Eigenschaft des Algorithmus ist seine komplexe rekursive Struktur. Im Gegensatz zu konventionellen rekursiven Funktionen, die eine lineare Sequenz von Funktionsaufrufen produzieren, gleicht die Struktur der Funktionsaufrufe bei diesem Algorithmus einem Baum, wie dies in Fig. 13 dargestellt ist. Jede Instanz der Funktion startet eine Rekursion für jede Variable, die gesetzt werden muss. Die Rekursion endet, wenn kein Pfad erforderlich ist um die Variable zu setzen, weil diese Variable bereits den korrekten Wert aufweist. Der resultierende Pfad kann dadurch bestimmt werden, dass die Blätter des Baumes von links nach rechts miteinander verbunden werden.

Backtracking in Kombination mit der baumähnlichen Rekursionsstruktur machen es außerordentlich schwierig, den Erzeugungsprozess zu verfolgen. Aus diesem Grunde sollte eine Logging-Technik verwendet werden, die Informationen über den Prozess in ein Logfile schreibt.

Außerdem müssen geeignete Maßnahmen ergriffen werden, um endlose Rekursionen zu vermeiden. Erstens kann die maximale Rekursionstiefe mit einem einfachen Zähler begrenzt werden. Wenn die Grenze erreicht wird, gibt die Funktion SearchPathToTrans einen Fehler aus. Zweitens kann die Anzahl des Auftretens eines bestimmten Übergangs in einem Pfad begrenzt werden. Die Grenze wird nach dem Suchen eines Übergangs in Schritt 7 überprüft. Wenn die Grenze erreicht ist, wird der Übergang zurückgewiesen und eine Alternative wird gesucht.

**Auffinden eines Pfades zum Endzustand**

Nachdem SearchPathToTrans einen Pfad $P_1$ wiedergegeben hat, der mit dem Anfangszustand der GUI beginnt und mit dem gewünschten Übergang $T_n$ endet, ist es notwendig, den Testfall durch das Auffinden eines Pfades $P_2$ von $T_n$ zum Endzustand $C_e$ der GUI zu finden. Dieses Ziel kann durch die Funktion SearchPathToEnd erreicht werden, welche im Grunde eine vereinfachte Version von SearchPathToTrans ist.

Funktion **SearchPathToEnd** (Eingabe: Zustand der GUI $C_{n+1}$)

1. Wenn $C_{n+1} = C_e$, ist kein Pfad notwendig => erfolgreicher Abschluß
2. Suchen eines Übergangs $T_x$, welcher die Anwendung beendet
3. Wenn kein Übergang gefunden wird, wird ein Fehler ausgegeben (inkonsistente Spezifikation)
4. Aufrufen der Funktion **SearchPathToTrans**($T_x$, $C_{n+1}$), um einen Pfad zu $T_x$ zu finden
5. Wenn keine Lösung gefunden wird, wird durch Zurückspringen zu Schritt 2 ein neuer Übergang gesucht
6. Ausgabe des Ergebnispfades

**Erfüllen einer Bedingung**

Um einen Pfad zu einem bestimmten Übergang zu finden, muß man wissen, wie die Bedingung, von welcher der Übergang abhängt, erfüllt werden kann. Das bedeutet, dass ein geeigneter Wert für alle Variablen, die in der Bedingung auftreten, gefunden werden muß, sodass die Bedingung erfüllt ist.

Eine Bedingung kann als Baum dargestellt werden, indem Operanden als Kind-Knoten ihrer Operatoren dargestellt werden, wie dies beispielsweise in Fig. 14 dargestellt ist. Wieder wird ein rekursiver Algorithmus, der Backtracking verwendet, zur Auffindung von Lösungen für diesen Bedingungs-Baum verwendet.

Das Herz des Algorithmus ist die Prozedur **FulfillCondition**(Knoten, Wert). Diese wird rekursiv für jeden Knoten des Bedingungbaumes aufgerufen. Die Eingabeparameter sind der gegenwärtige Knoten und der erforderliche Wert für den Unterausdruck, der durch diesen Knoten repräsentiert wird. Der Algorithmus kann durch Aufrufen der Funktion FulfillCondition(RootNode, "TRUE") gestartet werden. Jeder Knoten versucht, den von ihm geforderten Wert zu liefern, indem er geeignete Werte von seinen Kind-Knoten fordert. Abhängig von der Art des Knotens werden unterschiedliche Strategien angewandt, um die gewünschte Bedingung zu erfüllen. Die Rekursion endet bei den Blattknoten des Baumes, welche entweder konstante Werte oder Variablen sind. Während konstante Werte nicht geändert werden können, um eine Bedingung zu erfüllen, ist es natürlich mög-

lich, Variablen neue Werte zuzuordnen.

Als Beispiel sei die Bedingung (#Age# > 60) AND (#Female# XOR #Male#) für einen Eintrag in einem Datenformular angenommen; diese Situation ist durch den Bedingungsbaum in Fig. 14 dargestellt. Der Baum wird von oben nach unten wie folgt abgearbeitet:

- Der Ursprungsknoten 'AND' wird mit dem erforderlichen Wert 'TRUE' aufgerufen. Um die Bedingung zu erfüllen, verlangt er ebenso den Wert 'TRUE' von seinen Kind-Knoten.
- Der linke Kind-Knoten '>' überprüft, ob einer seiner eigenen Kind-Knoten einen konstanten Wert aufweist. Nachdem sein rechter Nachfolger immer 60 zurückgibt, besteht die einzige Möglichkeit, die Bedingung (#Age# > 60) zu erfüllen darin, einen geeigneten Wert (z.B. 70) von seinem linken Nachfolger zu verlangen.
- Der Knoten #Age# repräsentiert den Inhalt eines Eingabefeldes und ist nun auf den Wert 70 fixiert. Falls dieser Wert sich im Laufe der Testfallgenerierung als nicht geeignet erweist, und Backtracking gestartet wird, versucht der Elternknoten dieselbe Prozedur mit anderen möglichen Werten wie 61,1000,10000 etc. Nur wenn alle diese Versuche scheitern, gibt auch der Elternknoten einen Fehler aus.
- Der Knoten 'XOR' (exclusive or) besitzt zwei Möglichkeiten, um die Bedingung zu erfüllen, nachdem beide Kinderknoten keine konstanten Werte haben. Zuerst wird versucht, von dem linken Knoten 'TRUE' zu verlangen und 'FALSE' von dem rechten Zweig. Wenn dies nicht zum gewünschten Erfolg führt, werden die gewünschten Werte umgekehrt.
- Die Knoten #Female# und #Male# repräsentieren die Werte von zwei Check-Boxen. Sehr ähnlich zu dem Eingabefeld #Age#, werden ihre Werte vom Elternknoten fixiert.

Wenn alle Knoten es geschafft haben, die geforderten Werte zu liefern, gibt der Quellknoten schließlich eine Erfolgsmeldung an die aufrufende Funktion zurück.

Wenn eine Variable öfter als einmal in einer Bedingung auftritt, müssen semantische Widersprüche vermieden werden. So würde z.B. der Wert 70 ungültig sein in einer Bedingung wie (#Age# > 60) AND (#Age# < 65). In diesem Fall passiert das folgende:

- Der Wert von #Age# wird durch den ersten Unterbaum (#Age# > 60) auf 70 fixiert
- Der zweite Unterbaum (#Age# < 65) stellt fest, dass der Wert von #Age# durch einen anderen Knoten fixiert wurde und dass dieser Wert nicht geeignet ist, um die Bedingung zu erfüllen. Er gibt einen Fehler aus.
- Backtracking wird gestartet und der erste Unterbaum versucht einen anderen Wert festzulegen (z.B. 61)
- Nun ist die auch Bedingung des zweiten Unterbaums erfüllt und die Funktion wird erfolgreich beendet.

Oftmals ist es auch wünschenswert, Testfälle zu erzeugen, welche bestimmte Bedingungen nicht erfüllen. Das Ziel einer solchen Vorgangsweise besteht darin, zu testen, wie die GUI auf falsche Benutzereingaben reagiert. Dieses Ziel kann auf einfache Weise dadurch erreicht werden, dass anstelle von 'TRUE' von einem Bedingungsknoten 'FALSE' verlangt wird.

**Fallstudie**

In diesem Abschnitt soll im folgenden die Methodologie zur Darstellung einer GUI und der anschließenden Testfallgenerierung an einem einfachen Beispiel erklärt werden. In der Praxis bleibt dabei ein Großteil des im folgenden beschriebenen Formalismus für den Anwender verborgen, da die erfindungsgemäße Software leistungsstarke visuelle Editoren zur Beschreibung der GUI bietet. Im folgenden soll davon ausgegangen werden, dass ein Login-Fenster (siehe Fig. 15) spezifiziert und getestet werden soll.

Zuerst ist die Definition der GUI-Objekte notwendig, d.h. der Menge von Fenstern: W = {Login-Dialog, Username, Password, OK}. Im folgenden werden dafür kurz die Abkürzungen {L,U,P,O} verwendet. Um diese Fenster zu beschreiben, sind die folgenden Designatoren notwendig:

- Für alle Fenstertypen: *(Caption [String], Enabled [Boolean], Visible [Boolean], Focused [Boolean], Coordinates [4 Integers]).*
- Zusätzlich haben die beiden Eingabefelder U und P einen Designator *Value [String].*

Bei der verwendeten Software können Informationen über das Window-Layout vorteilhafterweise von Ressource-Files oder mit Hilfe eines "GUI Spys" eingelesen werden.

Als nächster Schritt ist die Definition des Anfangszustandes der GUI notwendig, indem der

Startwert jedes Designators festgelegt wird.
L = ("Login",TRUE, TRUE, TRUE, 0,0,139,87)
U = ("Username",TRUE,TRUE, TRUE, 7,16,132,31,"")
P = ("Password",FALSE,TRUE, FALSE, 7,45,132,60,"")
O = ("OK",FALSE,TRUE, FALSE, 43,67,93,81)

Wie man erkennen kann, sind P und O anfänglich *enabled* und der Fokus befindet sich auf U (und ebenso auf L, welches das Kind von U ist).

Nun kann eine Startkombination definiert werden, indem alle Designatoren miteinander verbunden werden:
$C_s$ = ("Login", TRUE, ... 43,67,93,81).

Wie bereits weiter oben ausgeführt, kann bei der verwendeten Software diese Information importiert oder manuell mit den Eigenschaftseditoren editiert werden.

Weiters sind noch die Eltern-Kind-Relationen notwendig, die bei dieser "kleinen" Anwendung relativ einfach sind: L ist die Mutter von U, P und O.
R = {(L,U), (L,P), (L,O)}.

Bei der verwendeten Software werden dabei die Eltern-Kind-Relationen in einer Baumansicht visualisiert und editiert.

Des weiteren ist es noch notwendig, das dynamische Verhalten der GUI zu beschreiben. Dazu werden die Übergänge, die bei dieser beispielhaften Anwendung auftreten können, spezifiziert. Allerdings ist ein beträchtlicher Teil des Verhaltens der GUI bereits durch die verwendete Plattform und den Fenstertyp definiert, und es soll im folgenden daher nur auf jene Übergänge eingegangen werden, die zusätzliche GUI-Eigenschaften, die durch einen Programmierer implementiert werden, repräsentieren.

Der erste Übergang $T_1$ beschreibt das Verhalten des OK-Buttons. Das Ereignis ist ein Mausklick auf OK:

$E_1$ = <MClickL>/#O#

Der Satz möglicher Eingabekombinationen wird durch folgenden Bedingungen definiert:

$S_1$ = #O:$Enabled# AND #O:$Visible#

Der Übergang wir durch folgende Aktion beschrieben:

$\tau_1$ = CloseApplication()

Wie man erkennen kann, muss der OK-Button *enabled* sein, bevor er betätigt werden kann. Nun muss spezifiziert werden, wie er diesen Wert annehmen kann:

Der Übergang $T_2$ beschreibt das Feld "Password": Das Ereignis wird frei gelassen, da es hier kein entsprechendes Ereignis, etwa mit der Bedeutung "Eingabe eines Wertes in das Feld" existiert. Man könnte lediglich ein Ereignis definieren, wenn ein spezieller Wert für P verwendet werden würde. Allerdings impliziert dies, dass nur dieser spezielle Wert in das Feld eingegeben werden kann, was aber nicht der Fall ist. Um dieses Problem zu lösen, nämlich dass ein beliebiger Wert eingegeben werden kann, wird folgende Schreibweise verwendet:

$E_2$ = <>
$S_2$ = #P:$Enabled# AND #P:$Visible# AND #P# != ""

Nachdem ein Passwort eingegeben wurde, wird der OK-Button enabled:

$\tau_2$ = SetAttribute(#O:$Enabled#, TRUE)

Schließlich muss noch spezifiziert werden, wie P enabled werden kann: Der Übergang $T_3$ bezieht sich auf das Verhalten des Feldes "Username". Wieder wird das Ereignis indirekt über folgende Bedingung beschrieben:

$E_3$ = <>
$S_3$ = #U:$Enabled# AND #U:$Visible# AND #U# != ""

Nachdem ein Benutzername eingegeben wurde, wird P enabled:

$\tau_3$ = SetAttribute(#P:$Enabled#, TRUE)

Dabei ist zu beachten, dass es nicht möglich ist, zu spezifizieren, welcher Benutzername und welches Passwort tatsächlich von der Anwendung akzeptiert werden, da diese Information in einer Datenbank gespeichert sind und sich dynamisch ändern. Allerdings existieren nun ausreichend Informationen, um einen beispielhaften Testfall für eine GUI zu erzeugen. Nach der Erzeugung kann der Tester die erzeugten Werte entweder mit aktuellen Werten aus einer Datenbank ersetzen, oder er kann diese Werte in die Spezifikation der GUI als $E_2$ und $E_3$ einsetzen.

**Erzeugung eines GUI-Testfalls**

Im folgenden wird die Testfallgenerierung für $T_1$ demonstriert, wodurch ein ungefährer Eindruck von den Schwierigkeiten auch bei einfachen GUIs vermittelt wird. Die Erzeugung ist ziemlich aufwendig, obwohl nur wenige Übergänge auftreten und kein Backtracking notwendig ist. In Fig. 16 ist die Struktur der Funktionsaufrufe zum leichteren Verständnis dargestellt:

**Funktion GenerateTC($T_1$)**

Initialisieren der GUI-Variablen entsprechend $C_s$
Suchen des Pfades $P_1$ durch Aufrufen der Funktion **SearchPathToTrans**($T_1$, $C_s$)

**Funktion SearchPathToTrans($T_1$, $C_s$)**

**[Rekursionstiefe 1]**

1. Bestimmen der Menge erlaubter Eingangskombinationen $S_1$ = #O:$Enabled# AND #O:$Visible#
2. $C_s \notin S_1$ => ein Pfad ist notwendig
3. Suchen geeigneter Werte für alle Variablen durch Aufrufen der Funktion **FulfillCondition**($S_1$,TRUE).
4. Die Funktion gibt eine Lösung aus: #O:$Enabled# und #O:$Visible# erfordern den Wert TRUE
5. Für beide Variablen wird folgendes ausgeführt:
   {
   (Erste Schleife für #O:$Enabled#):
6. Der gegenwärtige Wert der Variable (FALSE) stimmt nicht mit dem notwendigen Wert (TRUE) überein => ein vorangegangener Pfad muss gesucht werden, der die Variable setzt
7. Ein geeigneter Übergang wird gesucht
8. Eine Lösung wurde gefunden: $T_2$ aktiviert O!
9. Rekursiver Aufruf von **SearchPathToTrans(**$T_2$, $C_s$), um einen Pfad zu $T_2$ zu finden

**Funktion SearchPathToTrans($T_2$, $C_s$)**

**[Rekursionstiefe 2]**

1. Bestimmen der Menge der gültigen Eingangskombinationen
   $S_2$ = #P:$Enabled# AND #P:$Visible# AND #P# != ""
2. $C_s \notin S_2$ => ein Pfad ist notwendig
3. Suchen geeigneter Werte für alle Variablen durch Aufrufen der Funktion **FulfillCondition**($S_2$, TRUE).
4. Die Funktion gibt eine Lösung aus: #P:$Enabled# und #P:$Visible# verlangen den Wert TRUE, #P# muss auf den Wert "x" gesetzt werden.
5. Für alle drei Variablen wird folgendes durchgeführt:
   {
   (Erste Schleife für #P:$Enabled#):
6. Der gegenwärtige Wert der Variable (FALSE) stimmt nicht mit dem erforderlichen Wert (TRUE) überein => ein vorangegangener Pfad muss gesucht werden, der die Variable entsprechend setzt
7. Ein geeigneter Übergang wird gesucht
8. Eine Lösung wurde gefunden: $T_3$ aktiviert P!
9. Rekursiver Aufruf von **SearchPathToTrans**($T_3$, $C_s$), um einen Pfad zu $T_3$ zu finden

**Funktion SearchPathToTrans($T_3$, $C_s$)**

**[Rekursionstiefe 3]**

1. Bestimmen der Menge der gültigen Eingangskombinationen
   $S_3$ = #U:$Enabled# AND #U:$Visible# AND #U# != ""
2. $C_s \notin S_3$ => ein Pfad ist notwendig

3. Suchen geeigneter Werte für alle Variablen durch Aufrufen der Funktion **FulfillCondition**($S_3$, TRUE).
4. Die Funktion gibt eine Lösung aus: #U:$Enabled# und #U:$Visible# verlangen den Wert TRUE, #U# muss auf den Wert "x" gesetzt werden.
5. Für alle drei Variablen wird folgendes durchgeführt:

{

(Erste Schleife für #U:$Enabled#):

6. Der gegenwärtige Wert der Variable stimmt mit dem erforderlichen überein => kein vorangegangener Pfad ist notwendig, es wird mit der nächsten Variable fortgesetzt.

(Zweite Schleife für #U:$Visible#):

6. Der gegenwärtige Wert der Variable stimmt mit dem erforderlichen überein => kein vorangegangener Pfad ist notwendig, es wird mit der nächsten Variable fortgesetzt.

(Dritte Schleife für #U#):

6. Der gegenwärtige Wert der Variable ("") stimmt nicht mit dem erforderlichen Wert ("x") überein => ein vorangegangener Pfad muss gesucht werden, der den Wert entsprechend setzt
7. Ein geeigneter Übergang wird gesucht
8. Eine Lösung wurde gefunden: Eingabefelder erlauben die direkte Manipulation ihres Inhalts durch den Benutzer. Im folgenden wird dieser Übergang mit $T_u$ bezeichnet.
9. Rekursiver Aufruf von **SearchPathToTrans**($T_u$, $C_s$), um einen Pfad zu $T_u$ zu finden

**Funktion SearchPathToTrans($T_u$, $C_s$)**

**[Rekursionstiefe 4]**

1. Bestimmen der Menge der gültigen Eingangskombinationen
   $S_u$ = #U:$Enabled# AND #U:$Visible#
2. $C_s \not\in S_u$ => erfolgreicher Abschluß

**[Rekursionstiefe 3 fortgesetzt]**

10. Eine Lösung wurde gefunden. Lösungspfad = ($T_u$)
11. Überprüfen ob der neue Pfad irgendeine Variable verändert, die in einer früheren Schleife gesetzt wurde.
12. #U:$Enabled# und #U:$Visible# durch den Pfad unverändert
13. Hinzufügen des neuen Pfades zum Ergebnispfad (der gegenwärtig leer ist), setzten des gegenwärtigen Zustandes $C_i$ auf $\tau_u(C_s)$. Da keine weiteren Variablen gesetzt werden müssen, wird die Schleife verlassen

}

14. Ein Lösung wurde gefunden
15. Ausgabe des Ergebnispfades

**[Rekursionstiefe 2 fortgesetzt]**

10. Eine Lösung wurde gefunden. Lösungspfad = ($T_u$, $T_3$)
11. Überprüfen ob der neue Pfad irgendeine Variable verändert, die in einer früheren Schleife gesetzt wurde.
12. Nachdem es sich um die erste Schleife handelt, wird der Pfad akzeptiert
13. Hinzufügen des neuen Pfades zum Ergebnispfad (gegenwärtig leer), setzen des gegenwärtigen Zustandes $C_i$ auf $\tau_3(\tau_u(C_S))$

(Zweite Schleife für #P:$Visible#):

6. Der gegenwärtige Wert der Variable stimmt mit dem erforderlichen überein => kein vorangegangener Pfad ist notwendig, es wird mit der nächsten Variable fortgesetzt

(Dritte Schleife für #P#):

6. Der gegenwärtige Wert der Variable ("") stimmt nicht mit dem erforderlichen Wert ("x") überein => ein vorangegangener Pfad muss gesucht werden, der den Wert entsprechend setzt
7. Ein geeigneter Übergang wird gesucht

8. Eine Lösung wurde gefunden: Eingabefelder erlauben die direkte Manipulation ihres Inhalts durch den Benutzer. Im folgenden wird dieser Übergang mit $T_p$ bezeichnet.
9. Rekursiver Aufruf von **SearchPathToTrans**($T_p$, $C_i$), um einen Pfad zu $T_p$ zu finden

5 **Funktion SearchPathToTrans($T_p$, $C_i$)**

**[Rekursionstiefe 3]**
1. Bestimmen der Menge der gültigen Eingabekombinationen
   $S_p$ = #P:$Enabled# AND #P:$Visible#
2. $C_i \notin S_p$ => erfolgreicher Abschluß

**[Rekursonstiefe 2 fortgesetzt]**
10. Eine Lösung wurde gefunden. Lösungspfad = ($T_p$)
11. Überprüfen ob der neue Pfad irgendeine Variable verändert, die in einer früheren Schleife gesetzt wurde.
12. #P:$Enabled# und #P:$Visible# bleiben durch den Pfad unverändert
13. Hinzufügen des neuen Pfades zum Ergebnispfad ($T_u$, $T_3$), setzten des gegenwärtigen Zustandes $C_i$ auf $\tau_p(\tau_3(\tau_u(C_s)))$. Nachdem keine weiteren Variablen mehr vorhanden sind, wird die Schleife verlassen.

}

14. Eine Lösung wurde gefunden
15. Ausgabe des Ergebnispfades

**[Rekursionstiefe 1 fortgesetzt]**
10. Eine Lösung wurde gefunden. Lösungspfad = ($T_u$, $T_3$, $T_p$, $T_2$)
11. Überprüfen ob der neue Pfad irgendeine Variable verändert, die in einer früheren Schleife gesetzt wurde.
12. Nachdem es sich um die erste Schleife handelt, wird der Pfad akzeptiert
13. Hinzufügen des neuen Pfades zum Ergebnispfad, setzen des gegenwärtigen Zustandes $C_i$ auf $\tau_2(\tau_p(\tau_3(\tau_u(C_s))))$

(Zweite Schleife für #O:$Visible#):

6. Der gegenwärtige Wert der Variable stimmt mit dem erforderlichen überein => kein vorangegangener Pfad ist notwendig, es wird mit der nächsten Variable fortgesetzt.

}

14. Eine Lösung wurde gefunden.
15. Ausgabe des Ergebnispfades

**[Funktion GenerateTC fortgesetzt]**

Eine Lösung wurde gefunden. Lösungspfad $P_1$= ($T_u$, $T_3$, $T_p$, $T_2$, $T_1$)

Suchen des Pfades $P_2$ durch Aufrufen der Funktion **SearchPathToEnd**($\tau_1(\tau_2(\tau_p(\tau_3(\tau_u(C_s)))))$)

**Funktion SearchPathToEnd($\tau_1(\tau_2(\tau_p(\tau_3(\tau_u(C_s)))))$)**

Erfolgreicher Abschluß, da die Eingabekombination = $C_e$ (Der letzte Übergang $T_1$ schließt die Anwendung)

**[Funktion GenerateTC fortgesetzt]**

Eine Lösung wurde gefunden. Lösungspfad $P_2$= ()

Ausgabe des gesamten Testfalles = ($T_u$, $T_3$, $T_p$, $T_2$, $T_1$)

Nun ist der gesamte Testfall fertiggestellt:
1. Eingabe der Benutzernamens "x", dadurch Aktivierung des Feldes "password"
2. Eingabe des Passwortes "x", dadurch Aktivierung des OK-Buttons
3. Drücken des OK-Buttons, dadurch Schließen der Anwendung

**Abschließende Bemerkungen**

Im Vergleich zu anderen Algorithmen zur Testfallerzeugung ist der hier vorgestellte Algorithmus wesentlich effizienter. Beispielsweise existieren Lösungen basierend auf einem Prolog-Algorithmus. Dieser Algorithmus führt eine rekursive Suche nach einem Pfad durch, der zu einem Übergang führt. Die Suche wird allerdings relativ ziellos durchgeführt, was bedeutet, dass korrekte Pfade nur durch Zufall gefunden werden. Außerdem kann die Rekursionstiefe durch den Benutzer nicht eingeschränkt werden, was zu sehr langen Bearbeitungszeiten führt und manchmal sogar in Endlosschleifen endet. Dieser Prolog-basierte Algorithmus eignet sich daher vor allem für kleine, Kommandozeilen-orientierte Benutzerschnittstellen, bei denen die Anzahl der möglichen Benutzeraktionen sehr eingeschränkt ist. Für moderne GUIs ist dieses Verfahren allerdings nicht geeignet.

Im Gegensatz dazu bestimmt das vorliegende, erfindungsgemäße Verfahren bzw. die darauf basierende Software nicht nur die für die GUI notwendigen Werte automatisch, sondern versucht auch, diese Werte systematisch, einen nach dem anderen, zu setzen. Die maximale Rekursionstiefe kann vom Benutzer kontrolliert werden, und Endlosschleifen sind unmöglich.

Die Hauptstärke des erfindungsgemäßen Verfahrens und des daraus abgeleiteten Algorithmus besteht darin, dass nicht nach zufälligen Pfaden gesucht wird, und anschließend überprüft wird, ob diese das gegebene Problem lösen, sondern es wird zuerst bestimmt, wie eine korrekte Lösung auszusehen hat, und erst dann wird ein Weg gesucht, um diese Lösung zu suchen.

## PATENTANSPRÜCHE:

1. Verfahren zum automatisierten Testen von Software, welche eine grafische Benutzeroberfläche aufweist, wobei eine auf einem Datenverarbeitungsgerät ausführbare Testfallgenerator-Software verwendet wird, mittels welcher Testfälle generiert und diese mit einer Software zur automatischen Testausführung auf einem Datenverarbeitungsgerät überprüft werden, **dadurch gekennzeichnet, dass**

   a) mit zumindest einem Editor zumindest das dynamische und das semantische Verhalten der Benutzeroberfläche der Software spezifiziert wird, wobei als Editor ein grafischer Editor verwendet wird, und

   b) von der Testfallgenerator-Software an Hand des so spezifizierten Verhaltens der Benutzeroberfläche Testfälle generiert werden, welche unmittelbar anschließend oder in einem abgesetzten Schritt

   c) von der Software zur automatischen Testausführung ausgeführt werden.
2. Verfahren nach Anspruch 1, **dadurch gekennzeichnet, dass** vor Schritt a) statische Informationen der Benutzeroberfläche von dem Editor eingelesen werden.
3. Verfahren nach Anspruch 2, **dadurch gekennzeichnet, dass** die statischen Informationen aus einer Ressourcendatei eingelesen werden.
4. Verfahren nach Anspruch 2, **dadurch gekennzeichnet, dass** die statischen Informationen mittels einer Bildschirmanalyse-Software eingelesen werden.
5. Verfahren nach einem der Ansprüche 2 bis 4, **dadurch gekennzeichnet, dass** die statischen Informationen zumindest umfassen ein Layout und/oder Attribute der Elemente der grafischen Benutzeroberfläche.
6. Verfahren nach einem der Ansprüche 2 bis 5, **dadurch gekennzeichnet, dass** die statischen Informationen hinsichtlich des Layouts und/oder der Attribute von einem Benutzer ergänzt werden.
7. Verfahren nach einem der Ansprüche 1 bis 6, **dadurch gekennzeichnet, dass** das dynamische Verhalten der Software/Benutzeroberfläche über die Eingabe von Zustandsübergängen spezifiziert wird.
8. Verfahren nach Anspruch 7, **dadurch gekennzeichnet, dass** die Zustandsübergänge mittels grafischer Symbole dargestellt werden.
9. Verfahren nach Anspruch 7 oder 8, **dadurch gekennzeichnet, dass** die Zustandsübergänge mit semantischen Bedingungen verknüpft werden.
10. Verfahren nach einem der Ansprüche 7 bis 9, **dadurch gekennzeichnet, dass** die Zustandsübergänge mit syntaktischen Bedingungen verknüpft werden.

11. Verfahren nach einem der Ansprüche 1 bis 10, **dadurch gekennzeichnet, dass** alle Elemente der grafischen Benutzeroberfläche von der Testfallgenerator-Software zumindest einmal angesprochen werden.
12. Verfahren nach einem der Ansprüche 9 bis 11, **dadurch gekennzeichnet, dass** alle von semantischen und/oder syntaktischen Bedingungen abhängenden Zustandsübergänge von der Testfallgenerator-Software mit zumindest einem richtigen und zumindest einem falschen Übergangswert abgedeckt werden.
13. Verfahren zum Testen von Software mit einer grafischen Benutzeroberfläche (GUI), wobei mit einer Software zur automatischen Testausführung auf einem Datenverarbeitungsgerät Testfälle überprüft werden, welche mit einer Testfallgenerator-Software erzeugt werden, wobei zum Testen eines Überganges ($T_n$) zwischen zwei Zuständen ($C_n$, $C_{n+1}$) der Benutzeroberfläche (GUI) der zu testenden Software zumindest ein Testfall (TC) erzeugt wird, welcher den entsprechenden Übergang ($T_n$) enthält, **dadurch gekennzeichnet, dass** zur Erzeugung des zumindest einen Testfalls (TC)
    a) ein erster Pfad ($P_1$) von Übergängen ($T_1$, $T_2$,.....$T_{n-1}$) erzeugt wird, welcher bei einem Ausgangszustand ($C_s$) der Benutzeroberfläche (GUI) startet und in einem Zwischenzustand ($C_n$) endet, wobei der Zwischenzustand ($C_n$) ein Zustand ist, welcher alle für den zu überprüfenden Übergang ($T_n$) notwendigen Eingabebedingungen ($C_n \in S_n$) erfüllt, und
    b) zumindest ein weiterer Pfad ($P_2$) von Übergängen ($T_{n+1}$, $T_{n+2}$, ... $T_m$) erzeugt wird, der in dem von dem zu testenden Übergang ($T_n$) erzeugten Zustand ($C_{n+1}$) beginnt und im Endzustand ($C_e$) der grafischen Benutzeroberfläche (GUI) endet, und
    c) die beiden Pfade ($P_1$, $P_2$) mit dem Übergang ($T_n$) miteinander verbunden werden.
14. Verfahren nach Anspruch 13, **dadurch gekennzeichnet, dass** der Testfall (TC) in einer Testfall-Datenbank gespeichert wird.
15. Verfahren zum Testen von Software nach Anspruch 13 oder 14, **dadurch gekennzeichnet, dass** zur Ermittlung eines Pfades ($P_x$) zu einem vorgebbaren Übergang in einein erweiterten Zustandsdiagramm,
    a) zumindest ein Satz von erlaubten Eingabebedingungen ($S_n$) ermittelt wird, für den der zu testende Übergang ($T_n$) ausführbar ist,
    b) für alle Variablen, von denen die Eingabebedingungen ($S_n$) abhängen, geeignete Werte ermittelt werden, sodass alle Eingabebedingungen erfüllt sind ($S_n$ = TRUE), und für jede Variable, von der die Bedingung ($S_n$) abhängt, beginnend bei einer ersten Variable
    c) zumindest ein Übergang ($T_x$) gesucht wird, welcher die Variable auf den gewünschten Wert setzt, anschließend der Zustand ($C_i$) des Zustandsdiagramms auf einen dem Wert der geänderten Variable entsprechenden Wert verändert wird und
    d) Punkt c) für die nächste Variable der Bedingung ($S_n$) durchgeführt wird.
16. Verfahren nach Anspruch 15, **dadurch gekennzeichnet, dass** der Pfad ($P_x$) durch Aufrufen einer Suchfunktion (SearchPathToTrans ($T_x$, $C_i$)) ermittelt wird.
17. Verfahren nach Anspruch 15 oder 16, **dadurch gekennzeichnet, dass** im Falle der Übereinstimmung des gegenwärtigen Zustandes ($C_i$) des Zustandsdiagramms mit einem Satz von erlaubten Eingabebedingungen ($C_i \in S_n$) kein Pfad ($P_x$) erzeugt wird.
18. Verfahren nach einem der Ansprüche 15 bis 17, **dadurch gekennzeichnet, dass** die Variablen eine vorgebare Reihenfolge aufweisen und dass die Variablen gemäß Punkt c) und d) in einer bestimmten Reihenfolge abgearbeitet werden.
19. Verfahren nach einem der Ansprüche 15 bis 18, **dadurch gekennzeichnet, dass** in Schritt c) bei einem Übereinstimmen des Wertes einer Variable mit dem gewünschten Wert mit der nächsten Variable fortgefahren wird.
20. Verfahren nach einem der Ansprüche 15 bis 19, **dadurch gekennzeichnet, dass** in Schritt c) keine geeigneten Werte gefunden werden, ein Fehler ausgegeben wird.
21. Verfahren nach einem der Ansprüche 15 bis 19, **dadurch gekennzeichnet, dass** im Falle, dass für eine Variable kein Übergang ($T_x$) gefunden wird, zumindest zu der unmittelbar vorher abgearbeiteten Variable zurückgegangen, für diese ein neuer Übergang erzeugt, und anschließend wieder für die Variable ein Übergang nach Schritt c) gesucht wird.
22. Verfahren nach einem der Ansprüche 15 bis 21, **dadurch gekennzeichnet, dass** zu je-

dem Übergang ($T_x$) ein Pfad ermittelt wird.

23. Verfahren nach Anspruch 22, **dadurch gekennzeichnet, dass** der Pfad durch rekursives Aufrufen der Suchfunktion (SearchPathToTrans ($T_x$, $C_i$)) ermittelt wird.
24. Verfahren nach Anspruch 22 oder 23, **dadurch gekennzeichnet, dass** für den Fall, dass kein Pfad zu dem Übergang ($T_x$) gefunden wird, ein anderer Übergang ($T_x'$) ermittelt wird.
25. Verfahren nach einem der Ansprüche 22 bis 24, **dadurch gekennzeichnet, dass** im Falle eines Auffindens eines Pfades überprüft wird, ob durch den Pfad eine oder mehrere bereits auf einen gewünschten Wert gesetzte Variablen verändert werden.
26. Verfahren nach Anspruch 25, **dadurch gekennzeichnet, dass** im Falle der Veränderung von zumindest einer Variable durch einen Pfad ein neuer Pfad zu dem Übergang ($T_x$) gesucht wird.
27. Verfahren nach einem der Ansprüche 16 bis 26, **dadurch gekennzeichnet, dass** im Falle des Nicht-Auffindens einer Lösung die Reihenfolge für die Abarbeitung der Variablen geändert wird.
28. Verfahren nach einem der Ansprüche 16 bis 27, **dadurch gekennzeichnet, dass** im Falle des Nicht-Auffindens einer Lösung in Schritt b) andere Variablen gesucht werden.
29. Verfahren nach einem der Ansprüche 16 bis 28, **dadurch gekennzeichnet, dass** ein ermittelter Pfad einem Ergebnispfad hinzugefügt und nach Hinzufügen aller Pfade der Ergebnispfad ausgegeben wird.
30. Verfahren nach einem der Ansprüche 16 bis 29, **dadurch gekennzeichnet, dass** zum Ermitteln eines Pfades ($P_2$) zu einem Endzustand des Zustandsdiagramms ein Übergang ($T_y$) gesucht wird, welcher die Anwendung unmittelbar beendet, und ausgehend von einem gegenwärtigen Zustand ($C_{n+1}$) des Zustandsdiagramms ein Pfad zu dem Übergang ($T_y$) gesucht wird.
31. Verfahren nach Anspruch 30, **dadurch gekennzeichnet, dass** für den Fall, dass der gegenwärtige Zustand der Anwendung der Endzustand ist ($C_{n+1} = C_e$), kein Pfad gesucht wird.

**HIEZU 7 BLATT ZEICHNUNGEN**

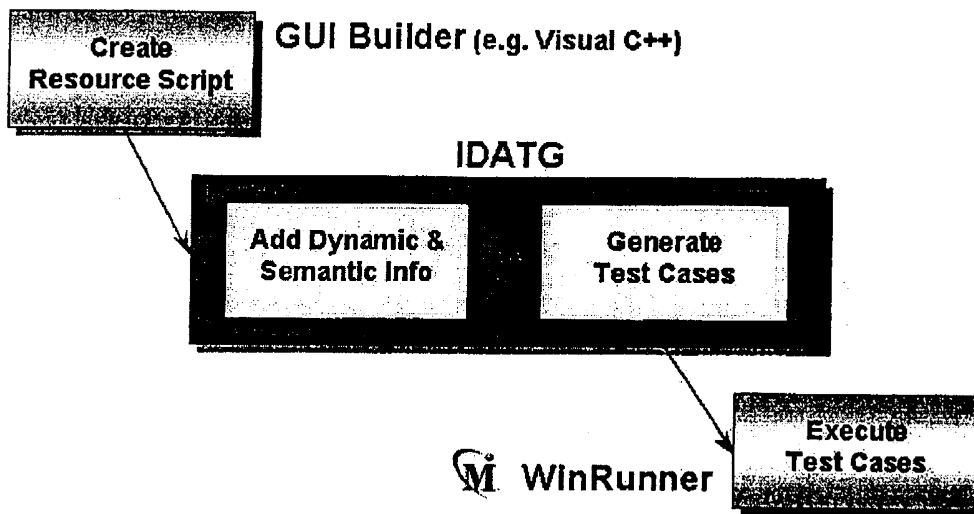


Fig. 1

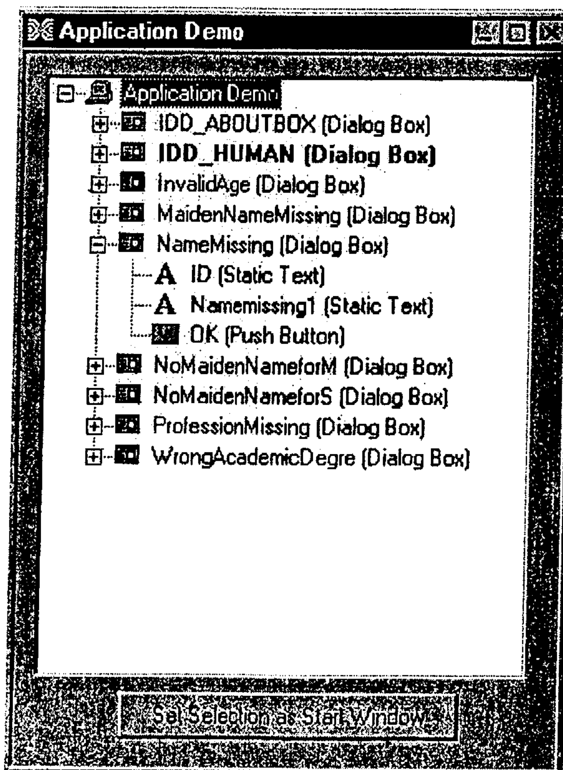


Fig. 2

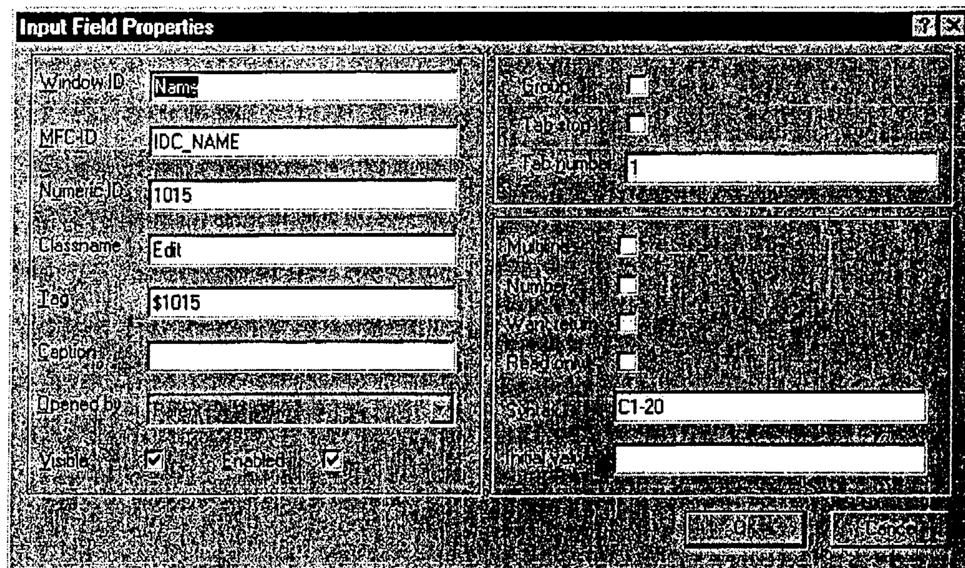


Fig. 3

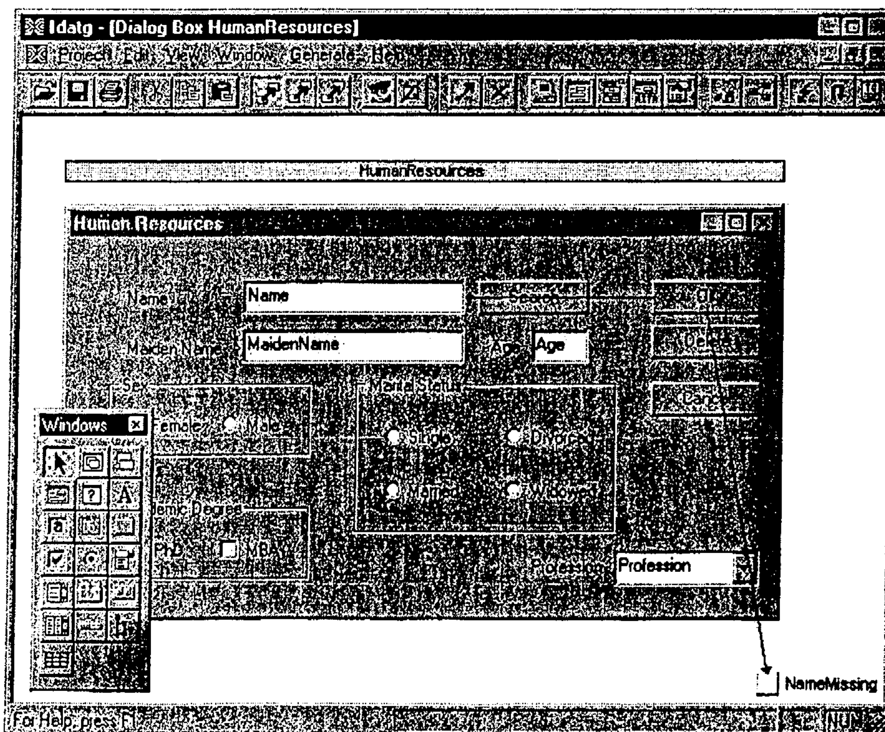


Fig. 4

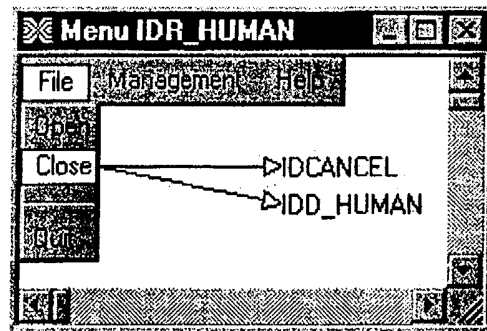


**Fig. 5**

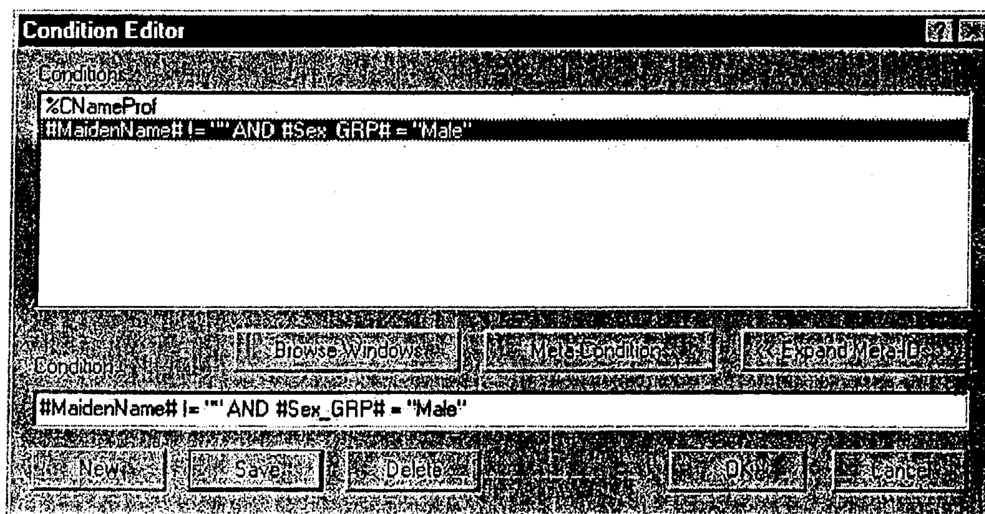


**Fig. 6**

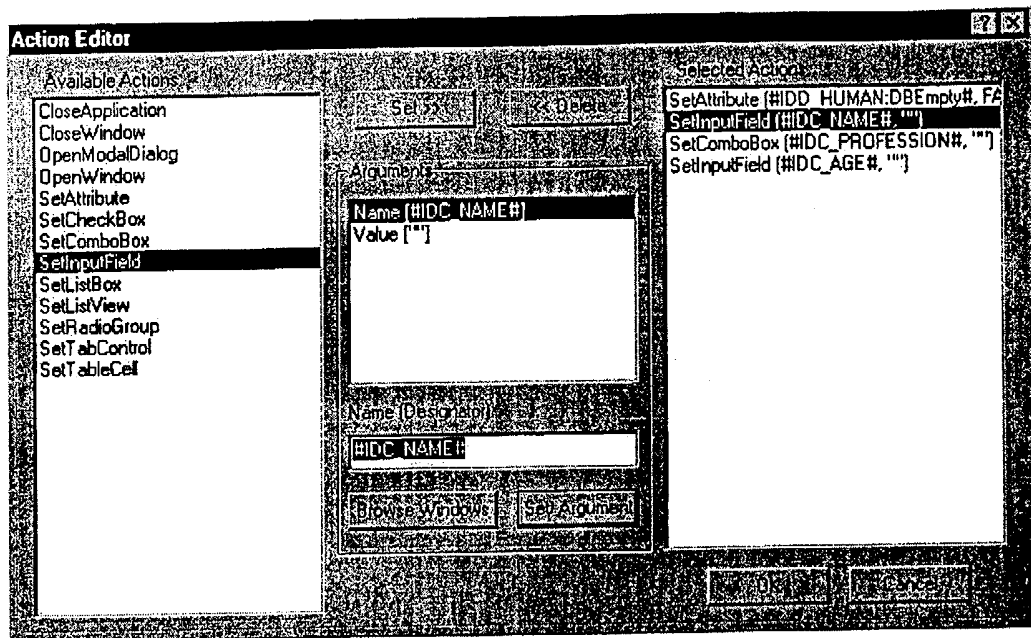


**Fig. 7**

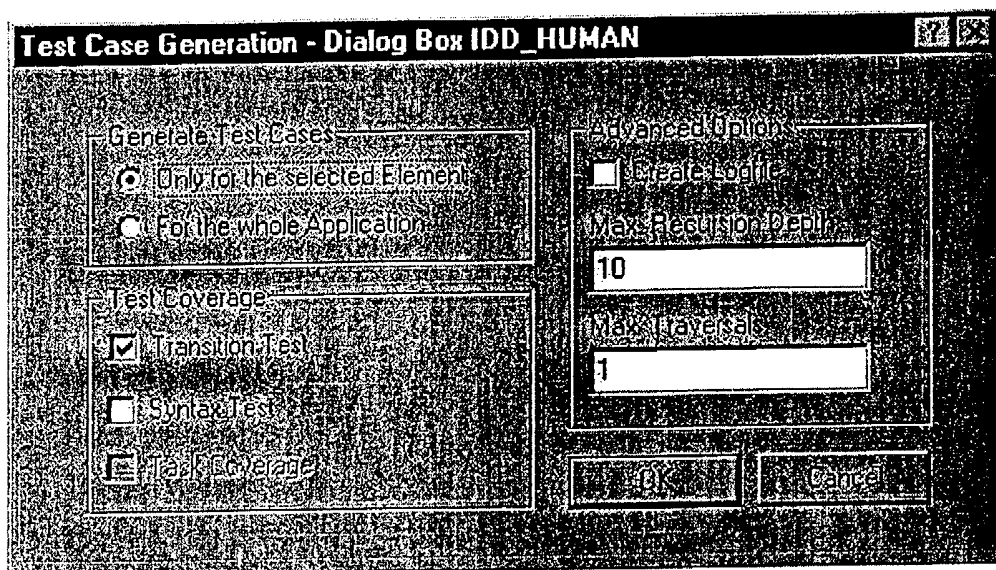


**Fig. 8**

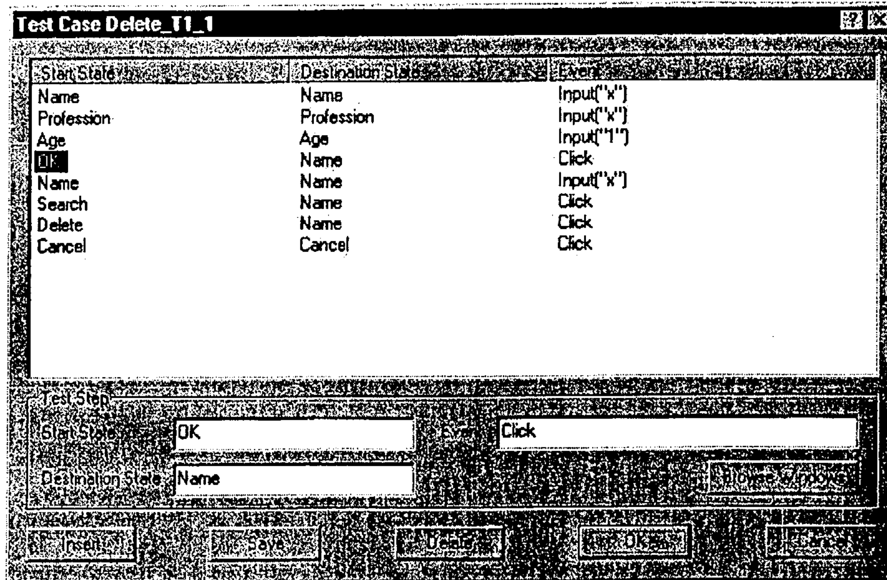


**Fig. 9**

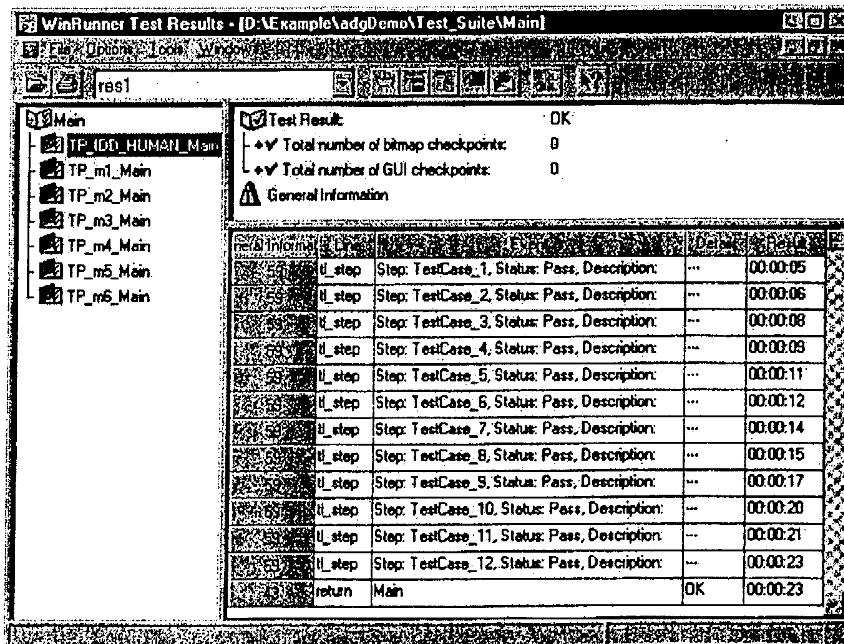


**Fig. 10**

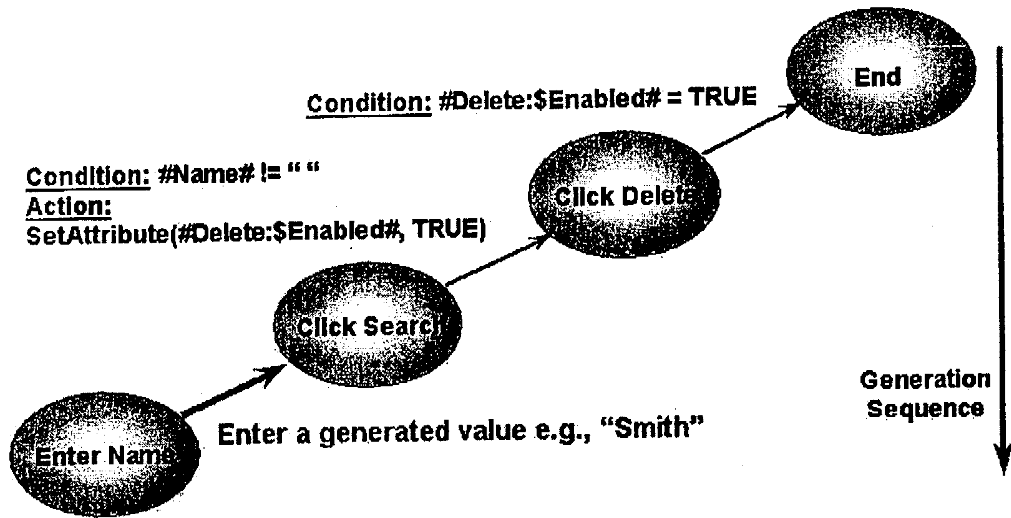


Fig. 11

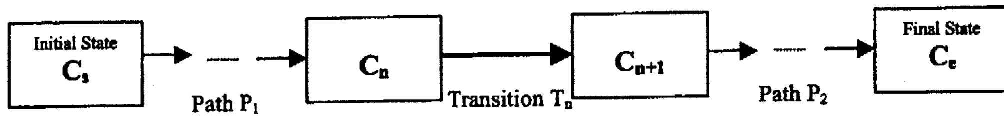


Fig. 12

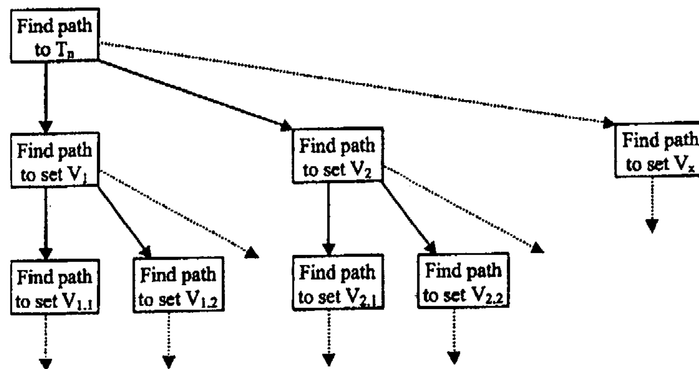


Fig. 13

Patentschrift Nr.: **AT 411 802 B**
Int. Cl. 7: **G06F 17/60**

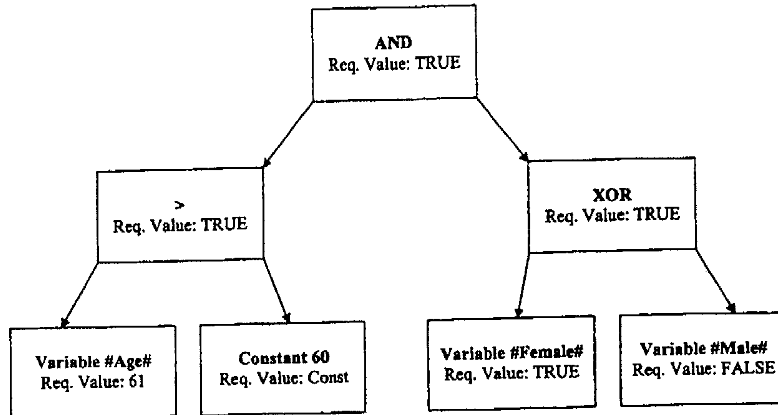


**Fig. 14**

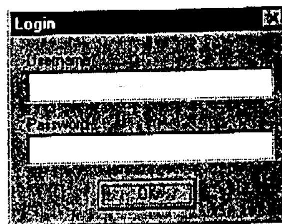


**Fig. 15**

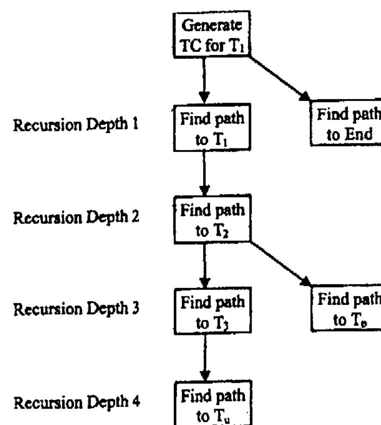


**Fig. 16**