Masterarbeit

Automatisierung von Regressionstests auf Basis von Benutzungsmodellen

Mathias Winder

Institut für Softwaretechnologie, Technische Universität Graz

Omicron Electronics GmbH, Abteilung für Software Qualitätssicherung

Begutachter/Betreuer (TU Graz): Univ.-Prof. Dipl.-Ing. Dr.techn. Franz Wotawa
Betreuer (Omicron): Dr.-Ing. Vladimir Entin und Dipl.-Inf. Stephan Christmann

Graz, im November 2011

Abstract

Regression testing is an efficient technique to discover failures and side effects dynamically during the software development process. This is achieved by retesting the SUT (System Under Test) after changes (e.g. bugfixes or new functionality). Executing these regression tests manually is very time consuming, so it is recommended to automate this process, especially the creation of the tests.

The goal of this master thesis is the development of a *model-based* system for automated test case generation which supports regression testing. Existing methods of test case generation should be researched and evaluated (for its suitability for regression testing), with focus on statistical testcase generation. It should be defined, how the interaction between user and SUT can be described as a usage model, which is the basis for the test case generation. The usage model should be abstract (no implementation knowledge), implementation changes shouldn‘t require regenerations of testcases. Optionally simple methods for test data generation should be treated. Finally a working prototype of the system will be implemented and evaluated.

Kurzfassung

Regressiontesten ist ein effizientes Verfahren, um möglichst schnell und dynamisch Nebenwirkungen im Softwareentwicklungsprozess zu entdecken. Dies wird durch das wiederholte Testen des gesamten SUT (System Under Test), z.B. nach einem Bugfix oder einem neuen Feature, erreicht. Das manuelle Durchführen dieses Verfahrens ist jedoch sehr zeitaufwändig, weshalb eine automatische Durchführung und vor allem Testfallerzeugung angestrebt werden sollte.

Ziel dieser Diplomarbeit ist es, ein *modellbasiertes* System für Testfallgenerierung zu entwickeln, welches die automatisierte Erzeugung von Testfällen und Ausführung von Regressionstests unterstützt. Dabei sollen bereits bestehende Methoden zur Testfallgenerierung recherchiert und (auf den Einsatz für Regressionstests) adaptiert werden. Der Fokus liegt bei dieser Arbeit auf statistischer Testfallgenerierung. Desweiteren soll definiert werden, wie die Interaktion des Benutzers mit dem SUT als Benutzungsmodell beschrieben werden kann, welches die Basis für die Testfallgenerierung darstellt. Das Modell soll dabei abstrakt (d.h. ohne direktes Wissen über die Implementation) sein, sodass bei Änderungen der Implementation möglichst keine Neugenerierung der Testfälle notwendig ist. Desweiteren sollen auch einfache Methoden zur Testdatengenerierung behandelt werden. Anschließend erfolgt die prototypische Implementierung und Evaluierung des Systems.

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Ort	Datum	Unterschrift

Danksagung

Zuerst möchte ich mich beim Institut für Softwaretechnologie (IST) und Herrn Prof. Franz Wotawa bedanken, welche die Durchführung dieser Masterarbeit im Rahmen meines Informatikstudiums an der Technischen Universität Graz ermöglichten. Herr Wotawa stand mir dabei als Betreuer immer wieder beratend zur Seite und war dabei sehr offen gegenüber neuer Ideen.

Meinem Auftraggeber, der Firma *Omicron electronics GmbH*, möchte ich vor allem für die reibungslose und unkomplizierte Organisation danken. Spezieller Dank gilt hier auch dem gesamten SWQA-Team des *Business Unit Primary*, welches mir bei der Konzeptionierung und speziell bei der Implementierung immer beratend zur Seite stand und mit konstruktivem Feedback für viele neue Ideen und Verbesserungen sorgte.

Im speziellen möchte ich mich hier noch bei meinen beiden firmenseitigen Betreuern bedanken. Vladimir Entin, welcher mein direkter Betreuer bei Omicron war, definierte damals mit mir zusammen das Thema der Masterarbeit. Mit immer wieder neuen Literaturvorschlägen half er mir im Anfangsstadium der Arbeit, mich in die Thematik einzuarbeiten. Im weiteren Verlauf und speziell im Endstadium half er mir immer wieder mit hilfreichen Feedbacks, das Erarbeitete zu verbessern. Stephan Christmann, Leiter des SWQA-Teams, hat mit seinen Ideen das TAI-Projekt überhaupt erst ermöglicht und uns jegliche Freiheiten bei der Auswahl des Masterarbeit-Themas gelassen. Er half uns dabei jedoch mit seinen Ideen und Ratschlägen, die richtige Richtung zu finden und motivierte mich stets, neues auszuprobieren.

Graz, im November 2011 Mathias Winder

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Test Automation Infrastructure . . . 1
  - 1.1.1 Tools und Plugins . . . 1
- 1.2 Ziel der Arbeit . . . 2

2 Grundlagen **5**
- 2.1 Blackbox Testing . . . 5
- 2.2 Modellbasiertes Testen . . . 6
  - 2.2.1 Modell als Abstraktion der Wirklichkeit . . . 6
  - 2.2.2 Modelle und Adaptoren . . . 6
  - 2.2.3 Graphenbasierte Pfadgenerierung . . . 8
- 2.3 Regressionstesten . . . 11
  - 2.3.1 Regression Testing im Allgemeinen . . . 11
  - 2.3.2 Automatisierung von Regressionstests . . . 12

3 Benutzungsmodell **13**
- 3.1 Semantik eines ALTS . . . 15
- 3.2 Beschreibung mittels UML Statemachine . . . 17
  - 3.2.1 Eingeschränkte UML Statemachine . . . 18
  - 3.2.2 UML Statemachine als ALTS . . . 20

4 Statistische Testverfahren **23**
- 4.1 Statistisches Testen . . . 23
  - 4.1.1 Statistische Testautomatisierung mit Benutzungsmodellen . . . 23
- 4.2 Markov Chains . . . 24
  - 4.2.1 Markov Chain Usage Model (MCUM) . . . 25
  - 4.2.2 Algorithmen für faire Verteilung . . . 27
- 4.3 Wahrscheinlichkeitsbasierte Algorithmen zur Pfadgenerierung . . . 37
  - 4.3.1 Random Walk . . . 37
  - 4.3.2 Most-Probable-Neighbor Walk . . . 38
  - 4.3.3 Regression Testing mit probabilistischen Algorithmen . . . 39
- 4.4 Risikobasiertes Testen . . . 41
  - 4.4.1 Berechnung von Risiken . . . 42
  - 4.4.2 Abschätzen von Risikofaktoren . . . 43
  - 4.4.3 Modellieren von Risiken im Benutzungsmodell . . . 45
- 4.5 Testfallerzeugung mittels risikobasierten Modellen . . . 48
  - 4.5.1 Risikobasierte Testfallpriorisierung . . . 48
  - 4.5.2 Risikobasierte Testfallgenerierung . . . 53

5 Implementierung **59**
5.1 Plugin für Statistisches Testen 59

6 Evaluation **62**
6.1 Fallstudie 1: Einfaches Beispiel 62
6.1.1 SUT: Car Alarm System 62
6.1.2 Beispielimplementierung 62
6.1.3 Ergebnisse 63
6.2 Fallstudie 2: Praxisbeispiel 64
6.2.1 SUT: Primary Test Manager 65
6.2.2 Erstellung des Modells und des Adapters 68
6.2.3 Vorgehensweise bei der Bewertung der Testfälle 68
6.2.4 Ergebnisse und Bewertung 72

7 Konklusion **77**

Analyse: Erstellung von GUI-Benutzungsmodellen mittels UML Statemachines **78**
A.1 Verschachtelung von Teilansichten 78
A.2 Richtiges Beschreiben von Modalen Fenstern 80

Analyse: Manuelle Risikoabschätzung und -modellierung in der Praxis **83**
B.3 Risikomodellierung 83
B.4 Risikoabschätzung 84

Verwendete Akronyme **85**

Literaturverzeichnis **86**

Abbildungsverzeichnis

1.1 Workflow von TAI 2
1.2 TAIsuite kombiniert mit TAIvinci und CodedUI [Cod] 3

2.1 Modellbasiertes GUI-Testing 5
2.2 Testmodellierung im allgemeinen (Quelle: [RBGW10]) 7
2.3 Testautomatisierung mittels Modell und Adaptor 7
2.4 Beispiel für einen gerichteten Graphen 8
2.5 Chinese Postman Problem 10
2.6 Testen bei inkrementeller Entwicklung (Quelle: [SL10]) 12

3.1 Beispiel-SUT Car Alarmäls ALTS 16
3.2 Alarm-Beispiel als UML Statemachine 17
3.3 Verwendetes Subset des UML Metamodells 19
3.4 Auflösen eines Composite States 21
3.5 Auflösen einer Rekursiven Submachine 22

4.1 Übergangswahrscheinlichkeiten einer Markov Chain als Graph 25
4.2 Markov Chain mit Standardverteilung 28
4.3 Markov Chain mit Gleichverteilung 29
4.4 Übergangswahrscheinlichkeiten mit Chinese Postman Adaptierung 31
4.5 Testbaum und Wahrscheinlichkeiten nach Chow‘s Algorithmus 32
4.6 Car Alarm mit berechneten Wahrscheinlichkeiten 36
4.7 Zustand *Unarmed* mit wegführenden Transitionen 38
4.8 Kantenauswahl Random Walk 39
4.9 Pseudozufallszahlen bei der Testfallgenerierung 40
4.10 Einfache Risikoannotierung in Benutzungsmodellen 46
4.11 Regionale Risikoannotierung in Benutzungsmodellen 47
4.12 Risikopriorisierung für n Testfälle 49
4.13 Car Alarm Benutzungsmodell mit Risikoannotierungen 50
4.14 The Adventurer‘s Journey 55
4.15 The Adventurer‘s Journey Approximation 57

5.1 Pluginstruktur von TAIsuite 59
5.2 Klassenstruktur des Plugins für statistisches Testen 60
5.3 Ansicht des Plugins für statistisches Testen in TAIsuite 61

6.1 Einfache Beispielimplementation des Car Alarm System 62
6.2 Übergangswahrscheinlichkeiten bei Zustand *Unarmed* 65

6.3 Workflow des Primary Test Manager (Quelle: Omicron electronics GmbH [Omi]) 66
6.4 Teilansichten bei der Erstellung eines Jobs 67
6.5 Ansicht zum Suchen von Objekten 67
6.6 Konfiguration von Vektorgruppen 68
6.7 Benutzungsmodell für die Objektsuche 69
6.8 Erreichbarkeit der Transitionen bei Gleichverteilung 73
6.9 Gleichverteilung wahrscheinlichkeitsbasierter Testsuites 74
6.10 Risikofokusierung risikobasierter Testsuites 76

A.1 Schablonen einer graphischen Oberfläche 79
A.2 Schlechtes Modellieren der Oberfläche 79
A.3 Verschachtelte Modellierung der Oberfläche 80
A.4 Modales Fenster 81
A.5 Fehlerhaftes Modellieren des Modalen Fensters 81
A.6 Korrektes Modellieren des Modalen Fensters 82

Tabellenverzeichnis

4.1 Pfadwahrscheinlichkeiten der einzelnen Verfahren 35
4.2 Pfadwahrscheinlichkeiten des Car Alarm Beispiels 35
4.3 Statische risikobasierte Priorisierung 51
4.4 Dynamische risikobasierte Priorisierung 52

6.1 Ergebnisse der ersten Fallstudie 64

B.1 Beispieltabelle für die Abschätzung von Risikofaktoren 84

1 Einleitung

1.1 Test Automation Infrastructure

Test Automation Infrastructure (TAI) ist ein im Sommer 2010 gestartetes Projekt von *Omicron Electronics* und besteht unter anderem aus einer ständig wachsenden Sammlung von selbst entwickelten Softwaretools und Plugins, welche die Tester bei modellgestützter Testautomatisierung unterstützen sollen. Dafür wurde zuerst ein eigener Workflow [EWZC11] (siehe Bild 1.1) ausgearbeitet, welcher die modellgestützte Testautomatisierung in TAI beschreibt. Zusätzlich wurde ein XML-Schema erstellt, welche als Speicher- und Austauschformat zwischen den Tools/Plugins fungiert. Spezieller Fokus wurde vorerst auf GUI-Testing (dt. Oberflächentesten) gelegt, jedoch sollen später auch andere Testarten unterstützt werden.

1.1.1 Tools und Plugins

Da TAI noch relativ jung ist, ist die Anzahl der Tools/Plugins und deren Möglichkeiten noch recht begrenzt, jedoch existiert bereits für jeden Schritt des Workflows mindestens ein Tool. Das Herzstück von TAI ist dabei ein Tool namens *TAIsuite* (siehe Bild 1.2). Nachdem TAI Anfangs nur aus einer losen Tool-Chain bestand, entschied man sich später zu dieser zentralen Komponente, welche aus technischer Sicht einen Pluginloader darstellt und die einzelnen Tools als Plugins zur Verfügung stellt. Diese können dann innerhalb des Workflows aufgerufen werden und die Daten automatisch miteinander austauschen.

TAI besteht derzeit aus folgenden Plugins/Tools, welche den Workflow unterstützen:

- **TAIspy:** Ein GUI-Tracker, welcher Nutzer-Aktivitäten (z.B. Mausklicks) automatisch erfasst und diese in das TAI-interne XML-Format umwandelt. Die erfassten Aktivitäten können dann für fertige Testcases verwendet werden, welche die erfassten Aktivitäten mittels einer speziellen UI-Library (basierend auf dem Framework *White* [Whi]) simulieren können. Dadurch werden GUI-tests ermöglicht.
- **Reflector:** Generiert aus .NET-Klassen (welche z.B. mit GUI-Trackern wie Coded UI [Cod] oder Ranorex [Ran] automatisch erstellt wurden) eine passende XML-Library und abstrahiert aus den einzelnen Methoden logische Aktivitäten. Diese Libraries können in TAI als Adaptoren verwendet werden, um abstrakte Testfälle in ausführbare umzuwandeln.
- **C#-Exporter:** Ein Gegenstück zum (bereits erwähnten) Reflector, welches die abstrakten Testfälle wahlweise in (ausführbare) *Coded UI Tests* oder *Unit Tests* umwandelt.

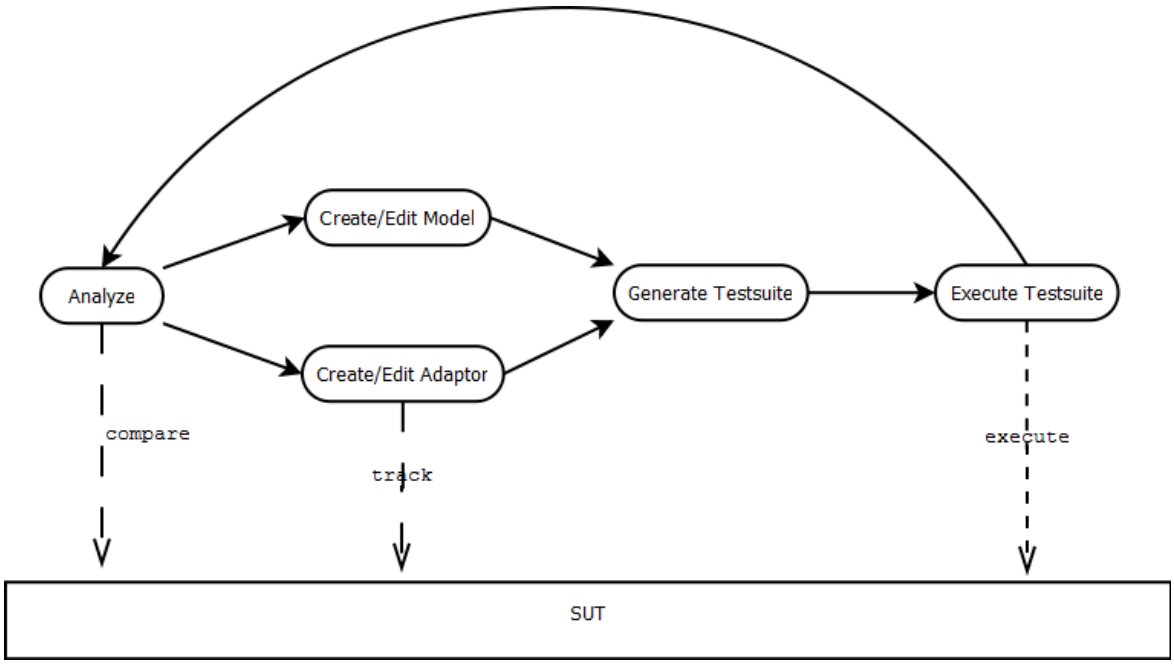

Abbildung 1.1: Workflow von TAI

- **TAIvinci:** Ein Plugin für Magicdraw [Mag], welches die Erstellung von speziellen UML-Aktivitätsdiagrammen, aus denen in TAI Testfälle generiert werden können. Es besteht grundsätzlich aus einer eigenen Toolbox (dt. Werkzeugleiste) und speziell angepassten GUI-Dialoge zum bearbeiten einzelner UML-Elemente in Magicdraw. Mit TAIvinci erstellte Modelle können in Magicdraw im XMI-Format [OMG] abgespeichert werden. TAIvinci ist derzeit das einzige Tool, dass nicht innerhalb von TAIsuite aufgerufen werden kann.
- **XMI-Importer:** Schnittstelle für den Import von XMI-Dateien, welche mit TAIvinci erstellt wurden.
- **Generator:** Einfaches Plugin zur Testfallgenerierung, welches aus einem einfachen Graphen (beschrieben als Aktivitätsdiagramm) Testfälle generieren kann (mittels einfacher Algorithmen wie z.B. Random Walk oder einer einfachen Approximation des Chinese Postman Problems 2.2.3)

1.2 Ziel der Arbeit

Nachdem der Grundstein für TAI gelegt worden war und die ersten Versuche zur praktischen Anwendung durchgeführt wurden, stellten sich schnell die Grenzen der einzelnen Tools heraus. Speziell die zur Verfügung stehenden Modellierungsmöglichkeiten stellten sich als unzureichend heraus. Mit den Aktivitätsdiagrammen war nur die Erstellung eines einfachen Graphes (ohne zusätzliche logische Bedingungen) möglich. Außerdem wünschte man sich die Möglichkeit, das

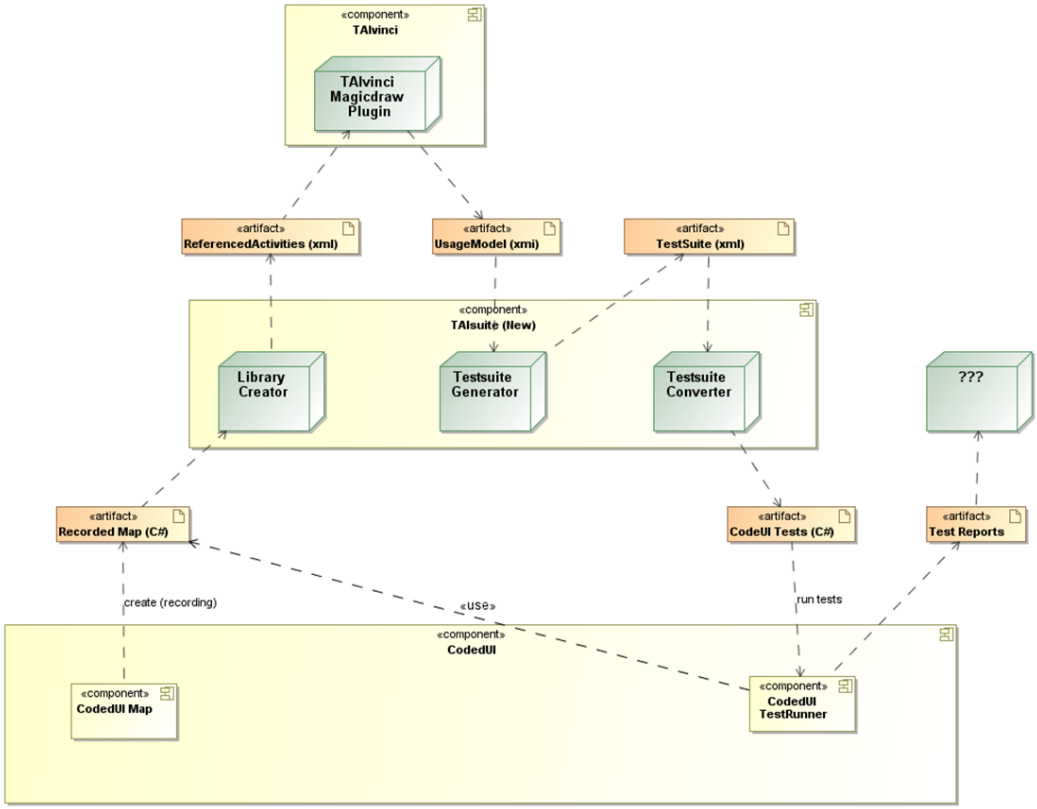

Abbildung 1.2: TAIsuite kombiniert mit TAIvinci und CodedUI [Cod]

SUT auch als Zustandsautomat darzustellen. Die bisher implementierten Algorithmen waren für einen sinnvollen und praktischen Einsatz auch weniger geeignet, vor allem in Hinblick auf Regression Testing bei inkrementeller Entwicklung. Deshalb sollte in dieser Arbeit TAI speziell in diesen Bereichen erweitert werden.

Das vorgegebene Ziel von Seiten der Firma *Omicron Electronics* war es, die Funktionen zur Modellierung und Testfallgenerierung von TAI zu erweitern. Es sollte vor allem eine Möglichkeit geschaffen werden, logische Bedingungen zu modellieren.

Es war relativ schnell klar, sich an *Labelled Transition Systems* [KSA09] und *Symbolic Transition Systems* [RdBJ00] zu orientieren, da diese schon relativ bekannt sind und genügend semantische Möglichkeiten bieten. Für eine einfachere Modellierung sollte das Modell jedoch mittels UML-Zustandsdiagrammen beschrieben werden können, welches aber nicht Inhalt dieser Arbeit ist. Die verwendete Modelltransformation wird in dieser Arbeit deshalb nur kurz erwähnt ohne darauf näher einzugehen.

Weiters sollten Methoden und Algorithmen zur modellbasierten Ableitung von Testfällen recherchiert (und optional selbst weiterentwickelt/angepasst) werden. Spezieller Fokus bei den Methoden sollte auf die Möglichkeit gelegt werden, dass sie auch sinnvoll im Bereich Regression Testing eingesetzt werden können. Grundsätzlich sollte dafür ein Konzept erstellt werden, wie die ausgewählten Methoden in TAI integriert werden können. Optionalerweise sollten die recherchierten Konzepte auch implementiert und mittels Fallstudien evaluiert werden.

Bei den recherchierten Methoden zur Testfallgenerierung einigte man sich schon sehr früh darauf, den Fokus auf statistische (speziell risikobasierte) Methoden zu legen. Hierbei sollte vor allem auch ausgearbeitet werden, wie statistische Daten ermittelt, modelliert und berechnet werden können.

Im Kapitel 2 werden die Grundlagen vermittelt, um die erarbeiteten Inhalte dieser Arbeit verstehen zu können. Kapitel 3 beschreibt die grundlegenden Elemente des in TAI verwendeten Benutzungsmodells. Kapitel 4 beinhaltet die recherchierten und erarbeiteten Konzepte der statistischen Testverfahren, im speziellen auch Algorithmen zur wahrscheinlichkeitsbasierten und risikobasierten Testfallgenerierung sowie Testfallpriorisierung. In Kapitel 5 wird der implementierte Prototyp vorgestellt, welcher innerhalb dieser Masterarbeit entstand und die vorgestellten Algorithmen aus Kapitel 4 beinhaltet. Zum Schluss werden in Kapitel 6 die Ergebnisse vorgestellt, welche man durch eine Evaluierung des Prototyps anhand 2 verschiedener Benutzungsmodelle erhielt.

2 Grundlagen

Im folgenden wird grundlegendes Wissen vermittelt, welche zum Verstehen dieser Arbeit benötigt wird.

2.1 Blackbox Testing

Blackbox Testing [SL10] ist ein Testverfahren, welches verwendet wird, wenn der innere Aufbau des Testobjekts nicht bekannt bzw. (bewusst) nicht herangezogen wird. Das Gegenstück zum Blackbox Testing ist das Whitebox Testing, bei welchem der innere Aufbau der Implementierung getestet wird.

Man beschränkt sich beim Testen auf die Inputs (dt. Eingaben) und Outputs (dt. Ausgaben), welche die *Blackbox* zur Verfügung stellt. Ein vollständiger Test wäre hier die Kombination aller möglichen Eingaben und Ausgaben. Ein bekannter Vertreter des Blackbox Testing ist der Systemtest einer GUI (Grafische Benutzeroberfläche). Hierbei wird eine Software direkt über die grafische Schnittstelle/Oberfläche (Graphical User Interface) getestet. Der Zugang zur Software ist also der gleiche, wie ihn schlussendlich auch der Nutzer hat, weshalb sich diese Art des Testens speziell auch für Benutzungstests eignet. Diese Arbeit befasst sich hauptsächlich mit dieser Art von Blackbox-Tests.

Bei modellbasierten GUI-Systemtests (siehe Abbildung 2.1) werden meist spezielle Adaptoren verwendet. Diese greifen nicht direkt auf Methoden des SUT's zu, sondern simulieren Benutzereingaben (z.B. Mausklicks) bzw. überprüfen Bildschirmausgaben.

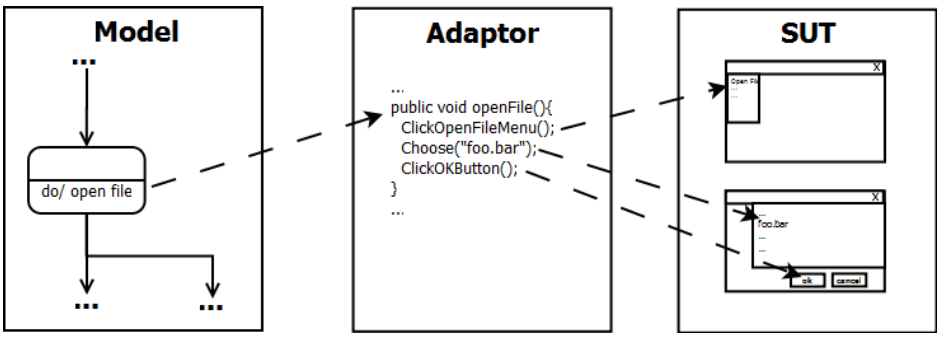

Abbildung 2.1: Modellbasiertes GUI-Testing

2.2 Modellbasiertes Testen

Unter dem Begriff *Modellbasiertes Testen* [RBGW10] im Kontext des Softwaretestens [SL10] versteht man das Testen eines Systems (bzw. Software) anhand eines Modells, welches das zu testende System (SUT) in vereinfachter Form beschreibt.

2.2.1 Modell als Abstraktion der Wirklichkeit

Laut [Sta73] verkürzt bzw. abstrahiert eine Modell das Original, d.h. es weist nicht alle Eigenschaften und Merkmale davon auf. Es hebt vielmehr diejenigen hervor, welche im Modellierungskontext wichtig sind, andere werden nur angedeutet (falls nötig) oder ganz vernachlässigt.

Testmodelle besitzen also eine abstrakte Beschreibung über das zu testende System (siehe Abbildung 2.2), beinhalten also ein begrenztes Wissen über das System (laut [Bin99] ist Testen streng genommen immer modellbasiert, da schon die Definition von Testfällen ein Modell ist). Die von Testmodellen abgeleiteten Testfälle können deshalb auch nicht das ganze System testen, sondern nur den beschriebenen Teil.

Testmodelle können durch unterschiedliche Modellierungswerkzeuge beschrieben werden (z.B. Zustandsdiagramme, Petrinetze, Sequenzdiagramme, etc.). Je nach verwendetem Werzkeug (bzw. Modelltyp) können unterschiedliche Aspekte eines Systems getestet werden, beispielsweise Zustandsdiagramme zum Testen grafischer Oberflächen (GUI-Test).

Meist wird nicht die technische Umsetzung des Systems beschrieben, sondern man beschreibt die Benutzung des Systems (also aus Anwendersicht). Ein solches Modell bezeichnet man als Usage Model (dt. Benutzungsmodell). Aus diesem Benutzungsmodell lassen sich durch unterschiedliche Verfahren einzelne Testcases (dt. Testfälle) ableiten, welche eine konkrete Benutzung (Eine Folge von Eingaben und erwarteten Ausgaben an das SUT) beschreiben.

2.2.2 Modelle und Adaptoren

Modelle können je nach Abstraktionsstufe unterschiedlich verwendbar sein:

- **Rein informelle Modelle** enthalten nur (semantisch korrekte) Beschreibungen des SUT. Der Tester kann anhand dieser das SUT manuell testen.
- **Technische Modelle** enhalten zusäztliche Informationen über die Implementation bzw. wie sie zu verwenden ist (z.B. das Klicken eines Buttons)
- **Modelle mit Adaptoren** (siehe Abbildung 2.3): Hier enthalten die Modelle keine direkten Informationen über die Implementationen. Es werden Adaptoren verwendet, welche die Informationen über die Implementation bereitstellen. Dies hat den Vorteil, dass das Modell von der Implementation unabhängig ist. Wenn sich die Implementation des SUT ändert, muss im optimalen Fall nur der Adaptor angepasst/ausgetauscht werden.

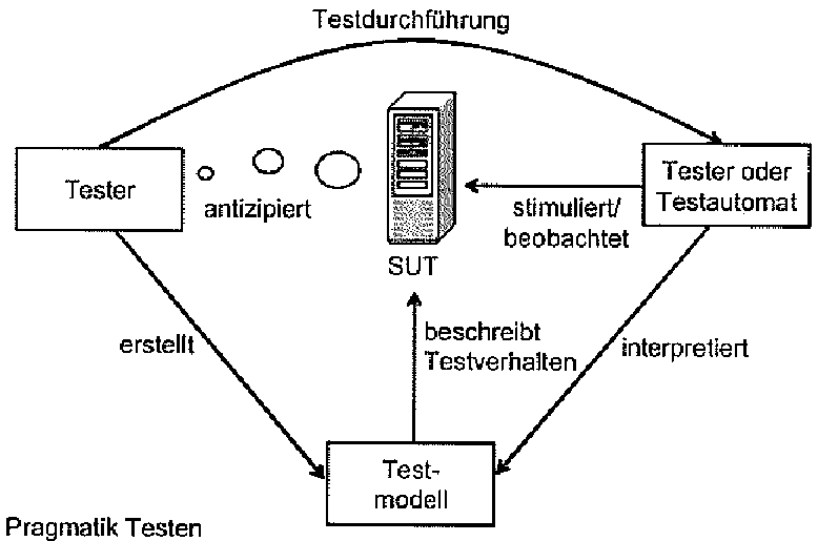

Abbildung 2.2: Testmodellierung im allgemeinen (Quelle: [RBGW10])

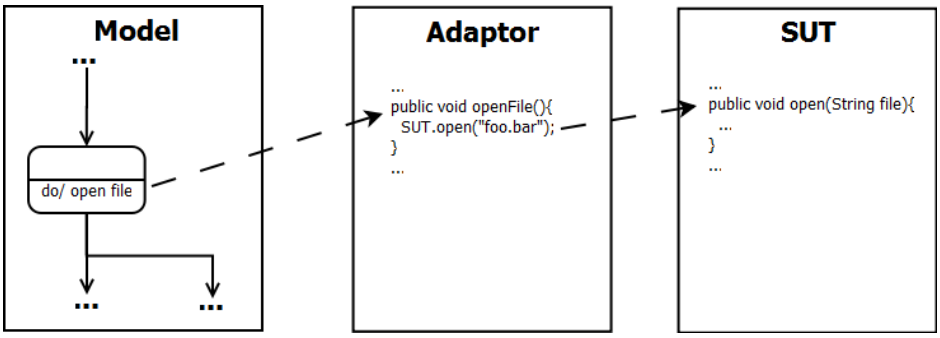

Abbildung 2.3: Testautomatisierung mittels Modell und Adaptor

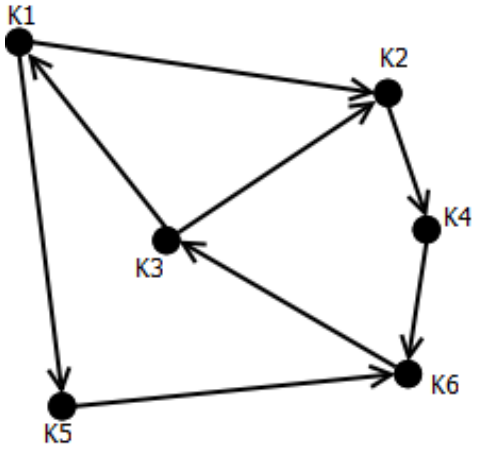

Abbildung 2.4: Beispiel für einen gerichteten Graphen

Gerade in Bezug auf Regressionstests ist es sehr wichtig, dass das Benutzungsmodell und Adaptor möglichst gut getrennt sind und bei Änderung im SUT optimalerweise nur Anpassungen im Adaptor notwendig werden.
Beispiel: Man modelliert für GUI-tests einen Anwendungsfall (z.B. das Abspeichern eines Dokuments) durch das Klicken mehrerer *Buttons*, anstatt dass ein Adaptor verwendet wird. Wird nun dieser Vorgang in einer neuen Version des SUT durch einen einzelnen Menüaufruf ersetzt, so muss das Benutzungsmodell ebenfalls abgeändert werden.

2.2.3 Graphenbasierte Pfadgenerierung

Ein Graph kann beschrieben werden als Paar $G = (V, E)$

- V eine endliche Menge von Knoten (*vertices*)
- $E = (v_i, v_j), v_i \in V \wedge v_j \in V$ eine Menge von Kanten (*edges*)

In ungerichteten Graphen ist sowohl der Übergang der Knoten v_i nach v_j als auch von v_j nach v_i möglich. In gerichteten Graphen ist nur ein der Übergang von v_i nach v_j möglich.

Eine bekannte Methode zur Generierung der Tests ist die graphenbasierte Pfadgenerierung, sofern die Modelle sich als Graph interpretieren lassen. Dies ist auch bei den in dieser Arbeit behandelten Modellen der Fall. Sowohl Transitionssysteme als auch Zustandsdiagramme lassen sich als gerichtete Graphen (siehe Bild 2.4) interpretieren. Die Modelle lassen sich jedoch meist nicht als reiner Graph interpretieren. Durch zusätzliche Bedingungen kann es der Fall sein, dass manche Teilpfade nicht immer verwendet werden können, weshalb Pfadgenerierungsmethoden für die Modelle adaptiert werden müssen.

Man erhält aus einem Graphen einen gültigen Testfall, indem man einen beliebigen Pfad zwischen (vorher festgelegten) Start- und Endknoten wählt.

Random Walk

Die einfachste Methode zur Pfadgenerierung in einem Graphen ist der Random Walk (dt. Zufallslauf).

Algorithmus 1 (Einfacher Random Walk Algorithmus) *Gehe wie folgt vor:*

1. *Starte bei einem gewählten Startknoten.*
2. *Von allen möglichen Übergängen, welche von dem aktuellen Knoten aus möglich sind, wähle einen per Zufall. Der Knoten am Ende des Übergangs ist der neue aktuelle Knoten (Merke dir alle verwendeten Übergänge).*
3. *Wiederhole Punkt 2 bis der gewählte Endknoten erreicht wird. Die verwendete Folge von Übergängen bilden den fertigen Pfad.*

Diese Methode ist in ihrer Reinform natürlich ungeeignet für das Thema Regression Testing, da bei jedem Anwenden unterschiedliche Pfade generiert werden. Diese Tests/Pfade sind also grundsätzlich nicht reproduzierbar. Ein Ansatz, wie solche zufallsbasierten Methoden für Regression Testing doch verwendet werden können, wird in Abschnitt 4.3.3 erörtert.

Chinese Postman Problem

Das Chinese Postman Problem [Thi03] (dt. Briefträgerproblem) sucht in einem gewichteten Graphen die kürzeste Route (beliebige Start- und Endknoten) oder den günstigsten Pfad (festgelegte Start- und Endknoten), welcher alle Kanten des Graphen abdeckt (komplette Kantenabdeckung). Als Analogie hierzu dient ein Briefträger, welcher seine Post in allen Strassen eines bestimmten Ortes ausliefern muss und hierfür die kürzeste Route sucht. Dabei wird das zu beliefernde Strassennetz als Graph repräsentiert, wobei Kanten die jeweiligen Strassen und Knoten die Kreuzungen repräsentieren. Das Chinese Postman Problem ist sehr stark mit dem bekannten Traveling Salesman Problem verwandt, welches jedoch eine komplette Knotenabdeckung anstrebt. Mit der Lösung dieses Problems können alle modellierten Komponenten eines Systems in der kürzest möglichen Zeit getestet werden (jedoch nicht in allen Kombinationen).

Beispiel 1 *Bild 2.5 zeigt einen Beispielgraphen für das Chinese Postman Problem. Die Kante zwischen $K3$ und $K4$ kann nur in eine Richtung gewählt werden (Einbahnstrasse), alle anderen Kanten können von beiden Seiten gewählt werden. Der kürzeste Pfad zwischen Punkt P und Punkt P (von Postamt bis zurück zum Postamt, das Postamt bild also hier Start- und Zielknoten), bei welcher der Briefträger alle Strassen/Kanten besucht, wäre in diesem Beispiel*
$P -- > K3 -- > K4 -- > K7 -- > K6 -- > K3 -- > K4 -- > K2 -- > P -- > K1 -- > K5 -- > K6 -- > K3 -- > P$
wobei hier die Kanten $K3 - K4$, $K6 - K3$ und $K3 - P$ zwar doppelt besucht werden müssen, aber immer noch die kürzeste mögliche Route ergeben.

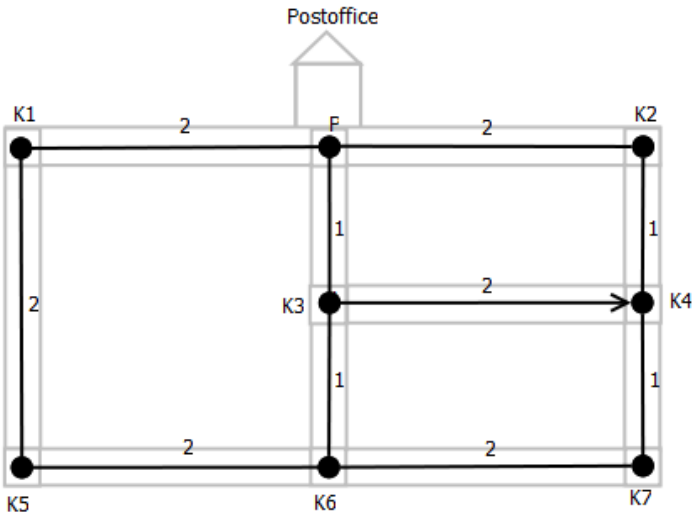

Abbildung 2.5: Chinese Postman Problem

Das Chinese Postman Problem ist nur für ungerichtete Graphen in polynomialer Zeit lösbar. In gerichteten Graphen (siehe [Thi03]) stellt es ein NP-vollständiges Problem dar, weshalb zur Lösung meist approximative Algorithmen verwendet werden, welche mit Heuristiken arbeiten.

Eine bekannte und einfache approximative Lösung ist die Nearest-Neighbor-Heuristik (Ein Erklärung der des Algorithmus findet man unter [Mon01]), welche auch gerne als Basisalgorithmus für komplexere Verfahren hergenommen wird.

Die Nearest-Neighbor-Heuristik wählt bei jedem Knoten jeweils die Kante mit der niedrigsten Kantengewichtung um zum nächsten Knoten zu gelangen. Um eine approximative Lösung für das Chinese Postman Problem zu erhalten, kann man diese Heuristik dahingehend anpassen, dass in jedem Schritt eine Kante gewählt wird, welche noch nicht besucht wurde oder das niedrigste Kantengewicht besitzt (wenn mehrere in Frage kommen, wird eine zufällige Kante gewählt). Dieses Verfahren wird dann einfach so lange angewandt, bis alle Kanten mindestens einmal besucht wurden und der Endknoten erreicht wurde.

Im Rahmen dieser Arbeit wurde eine erweiterte Chinese Postman Approximation auf Basis der Nearest-Neighbor-Heuristik erstellt und implementiert.

Algorithmus 2 (Erweiterter Nearest-Neighbor Algorithmus) *Erweitere die Nearest-Neighboor-Heuristik um folgende Eigenschaften:*

1. *Bei jeder Kante wird mitgezählt, wie oft sie bisher besucht wurde. Dabei wird für jede Kante die Heuristik*

$$H = Kantengewicht \cdot 2^{AnzahlderBesuche}$$

bestimmt. Die Kante mit dem niedrigsten Wert wird dadurch als nächste gewählt.

2. *Erweiterung des Algorithmus um einen Lookahead: Der Lookahead ist dabei eine Zahl > 0 welche bestimmt, wie weit der Algorithmus vorrausblicken kann um die beste Kante zu bestimmten. Dabei wird ein Spannbaum mit der Tiefe des Lookahead's erstellt, welcher die Kantengewichtung der möglichen Wege vom derzeitigen Knoten enthält. Mithilfe dieses Spannbaums kann dann die nächste Kante bestimmt werden, indem man den günstigsten Pfad (geringste Gesamtheuristik des Teilpfads) bestimmt, was in der Regel zu einem besseren Ergebnis führt.*

2.3 Regressionstesten

Unter Regression Testing [SL10] (dt. Regressionstesten) versteht man das wiederholte Durchführen von Testfällen an einem SUT, an welchem Änderungen vorgenommen wurden. Es dient hauptsächlich zum Aufspüren von sogenannten Nebeneffekten des SUT. Regression Testing gehört zu der Kategorie der *dynamischen Testverfahren* [SL10].

2.3.1 Regression Testing im Allgemeinen

Man kann zwischen folgenden Änderungen des SUT's unterscheiden [SL10]:

- **Testen nach Softwarewartung:** Hier geht es darum, bestehende Software an geänderte Einsatzbedingungen (z.B. Updates des Betriebssystems, Datenbanksystems, etc.) angepasst wird. Durch die Wiederholung der Regressiontests wird gewährleistet, dass die Software nach den Änderungen die gleiche Funktionalität aufweist wie die alte Software.
- **Testen nach Weiterentwicklung:** Bestehende Software wird hier um neue Funktionen zu erweitert. Jedoch reicht es dabei nicht, nur die Erweiterung zu testen, da durch das Ändern der Software auch (als Nebeneffekt) Fehler in bereits bestehenden Funktionen auftreten können. Durch die Regressiontests wird gewährleistet, dass die bisherige Funktionalität der Software aufrecht erhalten wurde (Die Menge der Regressiontests muss natürlich um weitere Tests erweitert werden, welche die neuen Funktionen testet).
- **Testen bei inkrementeller Entwicklung:** Bei inkrementeller Entwicklung wird ein Produkt (die Software) nicht in einem Stück erstellt, sondern in mehreren Versionsständen. Dabei wächst die Funktionalität und die Zuverlässigkeit mit der Zeit. Das Ziel ist dabei, dass das System nicht an der Kundenerwartung vorbeientwickelt wird. Dadurch ist schon in einem relativ frühem Stadium des Projekts ein großer Teil des Systems testbar, wodurch Fehler frühzeitig entdeckt werden können. Bei Weiteren Releasezyklen müssen diese Test jedoch wiederholt werden, um Nebeneffekte ausschließen zu können (siehe Abbildung 2.6).

Omicron Electronics betreibt inkrementelle Softwareentwicklung und nutzt dazu Scrum [Pic09]. Diese Arbeit nimmt deshalb hauptsächlich Bezug auf Regression Testing bei inkrementeller Entwicklung.

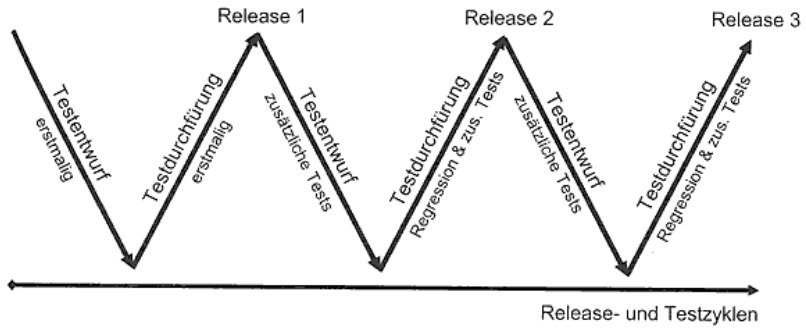

Abbildung 2.6: Testen bei inkrementeller Entwicklung (Quelle: [SL10])

2.3.2 Automatisierung von Regressionstests

Unter *Testautomatisierung* versteht man das Erstellen von Tests in maschinenlesbarer Sprache, welche automatisch ausgeführt (d.h. ohne Notwendigkeit von menschlichem Eingreifen) werden können. Bekannte Vertreter sind z.B. Unit-Testing, welche Funktionen der Implementation direkt aufrufen. Jedoch existieren auch Testautomatisierungstools für Blackbox-Testing, speziell auch für den Bereich GUI-Testing (z.B. Ranorex [Ran]). Der Vorteil von Testautomatisierung ist, dass bei wiederholtem Testbedarf kein menschliches Eingreifen mehr notwendig ist. Dadurch bleibt dem Tester monotone, sich wiederholende Arbeit erspart und seine Ressourcen können effizienter eingesetzt werden. Weiters kann, unterstützt durch Modelle, auch das Erstellen von Testfällen automatisiert werden.

Beim Regression Testing ist eine Automatisierung von Testfällen natürlich sehr naheliegend, da die Testfälle immer wieder durchgeführt werden müssen. Aufgrund des Aufwandes scheuen jedoch in der Praxis viele Firmen die Automatisierung. Gerade bei inkrementellem Entwicklungsprozess stellt Automatisierung ein großes Problem dar. Die Software ändert sich gerade Anfangs meist sehr stark (speziell die Benutzeroberfläche), weshalb Anpassungen der Tests notwendig werden, speziell wenn diese von Hand erstellt wurden.

Weniger aufwändige Änderungen (z.B. rein technische Änderungen) sollten nur Anpassungen in den Adaptoren benötigen, nicht im Modell. Jedoch kann es auch vorkommen, dass sich ganze Arbeitsschritte ändern, wodurch auch Änderungen im Modell notwendig werden. Dadurch müssen die Tests neu generiert werden. Da beim Regression Testing jedoch immer die (zumindest inhaltlich) gleichen Tests durchgeführt werden sollten, sollten Methoden zur Testfallgenerierung, welche hierfür eingesetzt werden, reproduzierbar sein. Der Random Walk Algorithmus 2.2.3 (in Reinform) wäre hierfür beispielsweise nicht geeignet.

3 Benutzungsmodell

Das verwendete Benutzungsmodell (im folgenden als *Activity Labelled Transition System* bzw. *ALTS* bezeichnet) kann als ein um Aktivitäten erweitertes *Labelled Transition System* [KSA09] beschrieben werden. Dabei wird nun in diesem Abschnitt ein allgemein gültiges Basismodell beschrieben (welches für alle behandelten Testmethoden gilt), weitere notwendige Erweiterungen (wie z.B. die Annotation von Risikofaktoren) werden in den entsprechenden Abschnitten eingeführt.

Definition 1 (Activity Labelled Transition System) *Das ALTS kann als ein 7-Tupel*

$$ALTS = (A, V, S, s_i, S_f, L, T)$$

beschrieben werden. Es gilt dabei

- *A ist die Menge aller Aktivitäten (mit $\epsilon \in A$)*
- *V ist die Menge aller Variablen*
- *S ist die Menge aller Zustände*
- *s_{init} ist der Startzustand ($s_{init} \in S$)*
- *S_f ist die Menge von Endzuständen ($S_f \subseteq S$)*
- *L ist die Menge von Labels*
- *$T \subseteq S \times L \times S$ ist die Menge von Transitionsrelationen*

Eine Aktivität $a \in A$ beschreibt einen Vorgang im System. Sie besteht aus einem Identifikator und eine Menge an Parametern (ähnlich einer Funktion). Die Aktivität kann als Regulärer Ausdruck $z*'('[z(,z)*]')'$ (z ist hierbei ein beliebiges Zeichen) geschrieben werden, z.B. *after(1)* (wobei *after* der Identifikator ist und *1* ein Parameter).

Ein Zustand ist ein Paar $(s, a) \in S$ mit $a \in A$ (s ist hierbei einfach nur ein Identifikator), ein Zustand kann also eine Aktivität enthalten. Für den Startzustand und die Endzustände gilt jedoch, dass diese kein Aktivität beinhalten können ($s_i = (s, \epsilon)$ und $(s, a) \in S_f \rightarrow a = \epsilon$).

Eine Transitionsrelation kann als Tripel (s,l,g) geschrieben werden, wobei s als *Quellzustand*, l das *Label* und g als *Zielzustand* bezeichnet wird.

Definition 2 (Label) *Ein Label ist ein Tripel $L = (c, t, E)$, wobei gilt:*

- $c : D^* \rightarrow \{true, false\}$ *ist ein boolescher Ausdruck (wobei D eine Konstante oder eine Variable sein kann, also* $V \subset D$*) und wird Condition genannt. Ein leerer Ausdruck (d.h. es ist keine Condition definiert) evaluiert nach true* ($\epsilon \rightarrow true$)
- $t \in A$ *ist eine Aktivität und wird in diesem Kontext Trigger genannt*
- E *ist eine Menge an Zuweisungen für die Variablen* V *und wird Effect genannt (z.B* $\$A = true, \$B = 3$ *für Variablen $A und $B).*

Die Syntax, in welcher eine Condition (Nichtterminalsymbol *condition*) bzw. ein Effect (Nichtterminalsymbol *effect*) beschrieben werden können, ist mit folgender EBNF definiert:

```
notzerodigit = '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
digit =  '0' | notzerodigit
number = '0' | [ '-' ] notzerodigit {digit}

symbol = a-z | A-Z | digit
word = symbol {symbol}

variable = '$' word

bool = 'true' | 'false'

operator = '+' | '-' | '*' | '/'

operation = (variable | number) { operator (variable | number) }

allocation = variable '=' ('"' word '"' | bool | operation)

effect = allocation { ',' allocation }


simpleequation = (variable | '"' word '"' | number | bool) ('==' |
    '!=') (variable | '"' word '"' | number | bool)

unequation = (variable | number) ('>' | '<') (variable | number)

equation = simpleequation | unequation

condition = equation { ('&&' | '||') equation}
```

> Anmerkung: Das *ALTS* ist auch einem I/O Symbolic Transition System [RdBJ00] sehr ähnlich, wobei sowohl Inputs als auch Outputs als Aktivität beschrieben werden. Weiterer markanter Unterschied ist jedoch, dass auch Zustände Aktivitäten enthalten können.

3.1 Semantik eines ALTS

Das ALTS beschreibt in diesem Kontext die Benutzung eines Systems. Die Semantik eines ALTS basiert grundsätzlich auf der Semantik von Symbolic Transitions Systems [RdBJ00] und Labelled Transition Systems [KSA09]. Im folgenden werden kurz die Semantiken der einzelnen Elemente beschrieben.

Eine **Aktivität** $a \in A$ beschreibt eine Aktivität, welche in dem beschriebenen System vorkommen kann. Eine Aktivität kann dabei sowohl ein Input (z.B. Mausklick oder Tastatureingabe) als auch ein Output (z.B. Bildschirmausgabe) darstellen.

Eine **Variable** $v \in V$ beschreibt eine Variable, welche zu unterschiedlichen Zeitpunkten bzw. Positionen im ALTS unterschiedliche Werte annehmen kann.

Eine **Condition** c ist ein normaler logischer Ausdruck, welcher entweder nach $true$ oder $false$ evaluiert.

Ein **Effect** $e \in E$ ist eine einfache Zuweisung oder mathematische Operation für eine Variable v, welche den von e definierten bzw. berechneten Wert nach dessen Auftreten annimmt.

Eine **Transitionsrelation** $tr = (s, l, g)$ beschreibt einen Übergang (Transition) von dem Quellzustand s in den Zielzustand g. Dabei ist die *Verwendung* einer Transitionsrelation tr nur möglich, wenn die Condition c des Labels $l = (c, t, E)$ nach $true$ evaluiert. Transitionsrelationen ohne Condition können immer verwendet werden. Wird tr verwendet, werden die Variablen V bezüglich den Zuweisungen aus E definiert (Anwendung der *Effects*). Die Verwendung von tr beschreibt das triggern der Aktivität t, die Aktivität wird also ausgeführt.

Eine **Aktivität** a in einem Zustand $(s, a) \in S$ wird durchgeführt, wenn dieser Zustand betreten wird. Damit können z.B. Bildschirmausgaben beschrieben werden, durch welchen ein Zustand definiert ist.

Ein **Pfad** P ist eine (geordnete) Folge von Transitionsrelationen t_i (mit $0 \leq i < n$, $n = |P|$ und $t_i \in T$). Dabei muss der Quellzustand jeder Transitionsrelation dem Endzustand der vorangegangenen entsprechen (mit Ausnahme der ersten und letzten Relation). Es gilt also für jede Transitionsrelation t_i mit $0 < i < n - 1$ des Pfades P:

$$t_i = (\_, \_, s_{i+1}) \rightarrow t_{i+1} = (s_{i+1}, \_, \_)$$

d.h. der Zielzustand einer Relation ist immer gleich dem Quellzustand der darauffolgenden Relation.

Ein **Testfall** TC ist ein Pfad, wenn der erste Anfangszustand des Pfades (Quellzustand der ersten Transition) dem Startzustand s_{init} des ALTS und der Endzustand des Pfades (Zielzustand der letzten Transition) einem der Endzustände S_f des ALTS entspricht. Es werden also folgende Bedingungen erfüllt:

- $t_0 = (s_0, \_, \_) \rightarrow s_0 = s_{init}$, d.h. der Pfad beginnt beim Startzustand
- $t_{n-1} = (\_, \_, s_{n-1}) \rightarrow s_{n-1} \in S_f$, d.h. der Pfad endet mit einem Endzustand

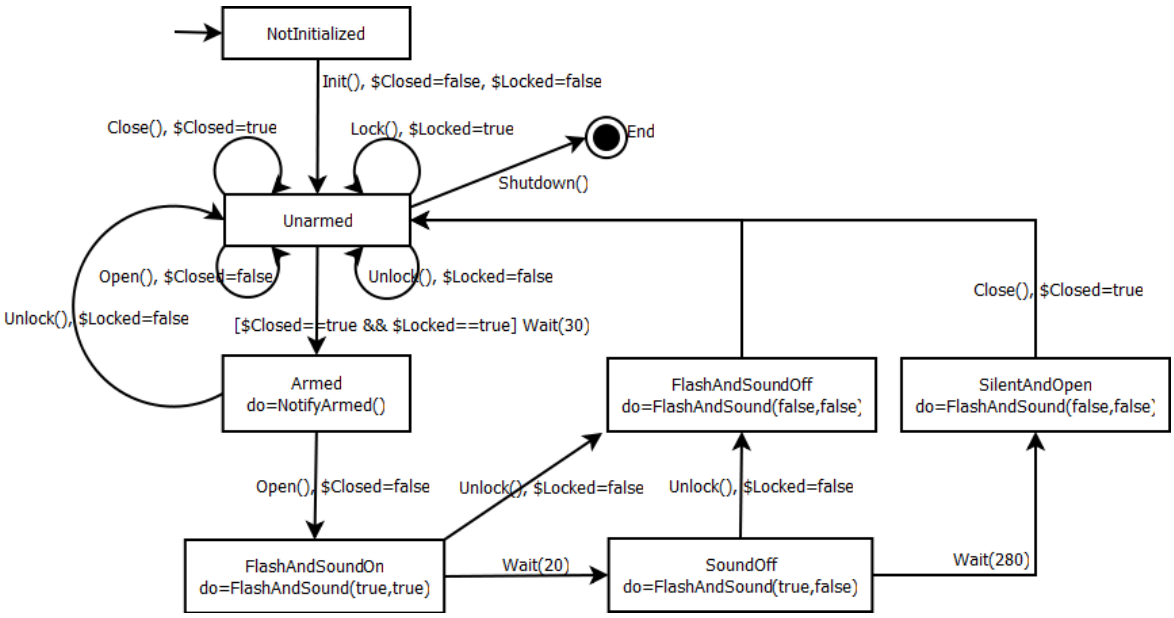

Abbildung 3.1: Beispiel-SUT Car Alarmäls ALTS

Eine **Testsuite** *TS* ist eine (geordnete) Menge von unterschiedlichen Testfällen.

Beispiel 2 (Beispiel für ein ALTS) *Bild 3.1 zeigt ein einfaches Beispiel als ALTS in graphischer Form:*

- *Aktivitäten: Init(), Shutdown(), Close(), Lock(), Open(), Unlock(), Wait(seconds), FlashAndSound(fl*
- *Variablen: $Closed, $Locked*
- *Zustände: NotInitialized, Unarmed, Armed, FlashAndSoundOn, SoundOff, FlashAndSoundOff, SilentAndOpen, End*
- *Startzustand: NotInitialized*
- *Endzustand: End*

Das Beispiel 2 zeigt die Benutzung eines Alarmsystems für Autos und ist eine leicht modifizierte Version des Modells aus [Aic10]. Ist das Auto abgeschlossen ($\$Closed == true$) und abgesperrt ($\$Locked == true$), wechselt das System nach 30 Sekunden in einen Armed-Zustand. Wird dann versucht, dass Auto zu öffnen, wird der Alarm ausgelöst, welcher ein Tonsignal für 20 Sekunden und ein Blinksignal für 5 Minuten (300 Sekunden) auslöst. Beide Signale werden durch das Aufsperren ($Unlock()$) beendet. Dieses Beispiel wird im Folgenden immer wieder als Referenzbeispiel herangezogen werden.

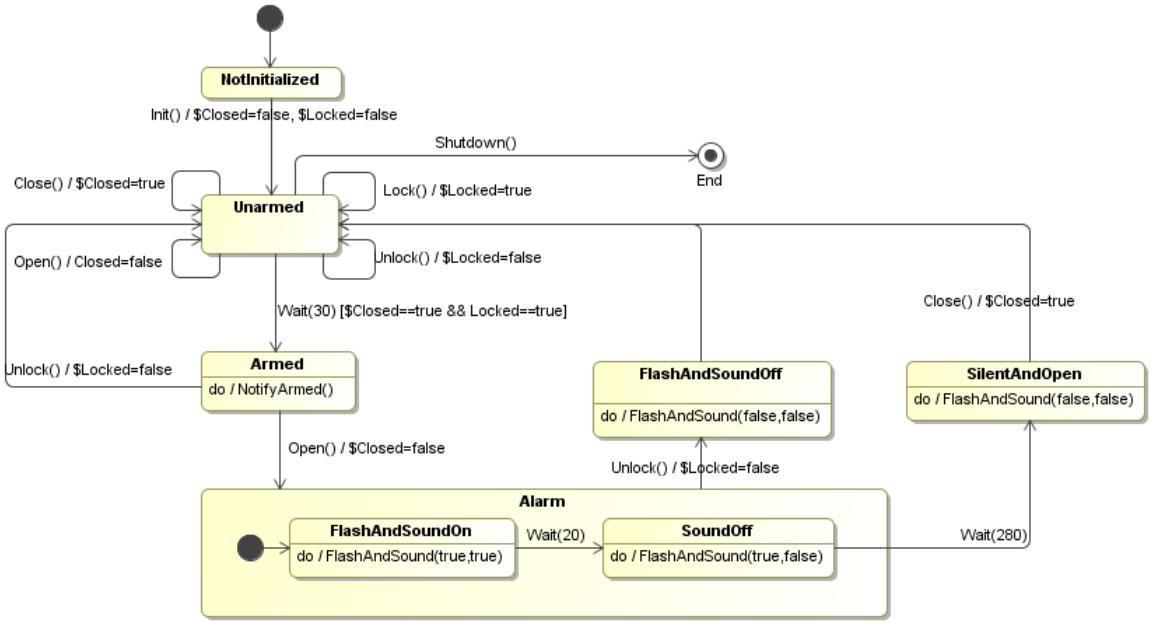

Abbildung 3.2: Alarm-Beispiel als UML Statemachine

3.2 Beschreibung mittels UML Statemachine

Der im vorherigen Abschnitt definierte ALTS ist mächtig genug, um die Benutzung eines Systems zu beschreiben. Jedoch hat er den Nachteil, dass er bei komplexeren (großskalierten) Systemen wegen fehlenden Stukturierungsmöglichkeiten bzw. Subtypen (z.B. unterschiedliche Arten von Zuständen) sehr schnell unübersichtlich wird. UML Statemachines bieten ein großes Set an Elementtypen und Strukturierungsmöglichkeiten.
In diesem Abschnitt wird ein Subset der *UML Statemachine Superstructure* [OMG07] mit zusätzlichen Einschränkungen beschrieben, welches es ermöglicht, großskalierte Modelle zu erstellen, welche in ein ALTS umgewandelt werden können. Sie erhöhen also nicht die Mächtigkeit der ALTS und werden hier nur erwähnt, da sie für die Modellierung der durchgeführten Casestudies (dt. Fallstudien) verwendet wurden.

Beispiel 3 *Bild 3.2 zeigt das Alarm-Beispiel als UML Statemachine. Was sofort auffällt ist die Composite State, welche den Alarm (Licht- und Tonsignal) strukturiert. Die Verwendung des Zustands* ***not initialized*** *ist übrigens eine Notwendigkeit, da ein Initial State nur ein Pseudostate darstellt und dort deshalb keine direkte Definition eines Triggers möglich ist (siehe UML Superstructure [OMG07]).*

3.2.1 Eingeschränkte UML Statemachine

Im folgenden wird auf das unterstützte Subset von UML-Elementen und deren Einschränkungen (definiert als OCL Constraints) eingegangen. Eine genauere Beschreibung/Definition der Elemente findet man in [OMG07].

Verwendetes Subset

Bild 3.3 zeigt das verwendete Subset, mit welchem Statemachines (dt. Zustandsdiagramme) erstellt werden können, die in ALTS umgewandelt werden können.

Grundsätzlich bestehen **Statemachines** aus 1 oder mehreren **Regions** (in unserem Fall nur eine direkte, da keine Parallelität behandelt wird), welche mindestens 1 **Vertex** (Knoten) und beliebig viele **Transitions** enthalten. Aus den Vertices leiten sich unterschiedliche Typen von **Pseudostates** und **States** ab. Transitions beschreiben die Übergänge zwischen den Vertices, bestehen also aus einem *source*-Vertex und einem *target*-Vertex.

Pseudostates beschreiben einfache Knoten und sind keine ëchtenSZustände, da das System diese Zustände nicht annehmen kann (d.h. ein Pseudozustand wird sofort wieder verlassen, weshalb Transitions mit einem Pseudozustand als *source*-Vertex keine Trigger besitzen können). Sie können von folgendem Typ (*PseudostateKind*) sein:

- *initial*: Markiert den Anfang (Startzustand) einer Region
- *junction*: Einfacher Knoten ohne genauere Funktion, hauptsächlich zur Zusammenführung mehrerer Transitions verwendet.
- *choice*: Ermöglicht das Treffen einer Entscheidung

States beschreiben echte Zustände, welche ein System annehmen kann (ein State ist dann *aktiv*). Man unterscheidet 3 Arten von Zuständen:

- *Simple State*: Einfacher Zustand welcher im System angenommen werden kann.
- *Composite State*: Zustand, welcher eine Region beinhalten kann, d.h. in diesem können Unterzusände definiert werden, welche aktiv werden können, solange der Composite State aktiv ist. Wird in der beinhaltenden Region ein Final States (dt. Endzustand) erreicht, wird ein im Composite State ein *completion event* ausgelöst, wodurch dieser wieder verlassen (inaktiv) wird.
- *Submachine State*: Zustand welcher auf eine andere Statemachine referenziert (solange dieser Zustand aktiv ist, können Zustände der referenzierten Substatemachine aktiv werden). Auch hier wird beim Erreichen eines Final States ein *completion event* ausgelöst.

Statemachines sind hierarchisch aufgebaut, States können also auch rekursiv andere Statemachines enthalten. Dies wird so realisiert, dass States (Ausnahme: FinalStates) beliebige Regionen enthalten/referenzieren können.

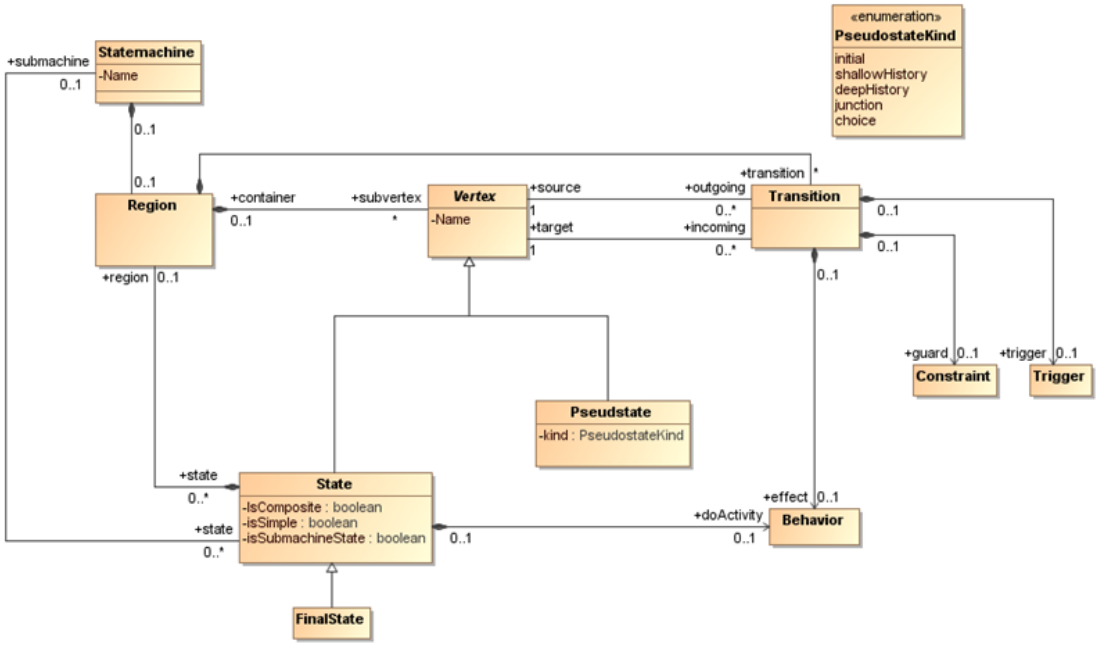

Abbildung 3.3: Verwendetes Subset des UML Metamodells

Zusätzliche Einschränkungen einzelner Elemente

Im folgenden werden nur zusätzliche Einschränkungen (informell und in OCL-Format) der Elemente erwähnt. Die genaue Semantik der Elemente findet man auch hier in [OMG07].

Vertex

- Ein Vertex muss mindestens eine outgoing-Transition besitzen, sofern es sich bei dem Vertex nicht um einen FinalState handelt

  ```
  not(type(FinalState)) implies (outgoing->size() >= 1)
  ```

- Ein Vertex kann maximal eine Region enthalten

  ```
  (region->size() <= 1)
  ```

State

- Ein Composite state hat genau eine Region

  ```
  isComposite = (region->size() == 1)
  ```

- Rekursive Submachine States (Submachine States, welche selber in der von ihnen referenzierten Submachine enthalten sind - direkt oder indirekt) dürfen keine wegführenden (outgoing) Transitions enthalten.

Pseudostate

- Folgende PseudostateKind's sind erlaubt: initial, junction, choice

```
(self.kind = #initial) or (self.kind = #junction)
or (self.kind = #choice)
```

Transition

- Der BehaviorType für die Association *effect* muss FunctionBehavior sein

3.2.2 UML Statemachine als ALTS

Damit aus dem als UML Statemachine beschriebenen Benutzungsmodell Testfälle generiert werden können (mit den in dieser Masterarbeit beschriebenen Algorithmen), muss die Statemachine in ein (semantisch äquivalentes) ALTS umgewandelt werden.

Grundsätzlich können alle Vertex-Elemente als Zustände und die Transitions als Transitionsrelationen interpretiert werden. Dabei werden dann die Guards, Trigger und Effects der Transitionen zu den Conditions, Trigger und Effects des jeweiligen Labels der Transitionsrelation. Hat ein State eine doActivity definiert, wird diese zur Aktivität des jeweiligen Zustands im ALTS.

Eine komplexere Umwandlung stellt das Auflösen von *Composite States* und *Submachine States* dar.

Auflösen von Composite States

Für jeden Composite State C:

1. Wenn C einen Initial pseudostate I beinhaltet: Ersetze I durch einen junction pseudostate J. Leite jede Transition, welche zu C führt nach J um (für jede Transition mit C als Target wird J das neue Target)
2. Ersetze jeden Final State F in einem durch einen junction pseudostate. Ersetze den Source jeder Transition T, welche als Source C hat und durch ein *completion event* ausgelöst wird (d.h. keinen speziellen Trigger definiert), durch F.
3. Jede Transition T mit einem individuellen Trigger (kein *completion event*), welche von C wegführt, muss für jeden (in C eingeschlossenen) (Sub)State dupliziert werden und diesen State als neue Source definieren (da T jederzeit gefeuert werden könnte, solange C aktiv ist). Die Original-Transition T wird gelöscht.

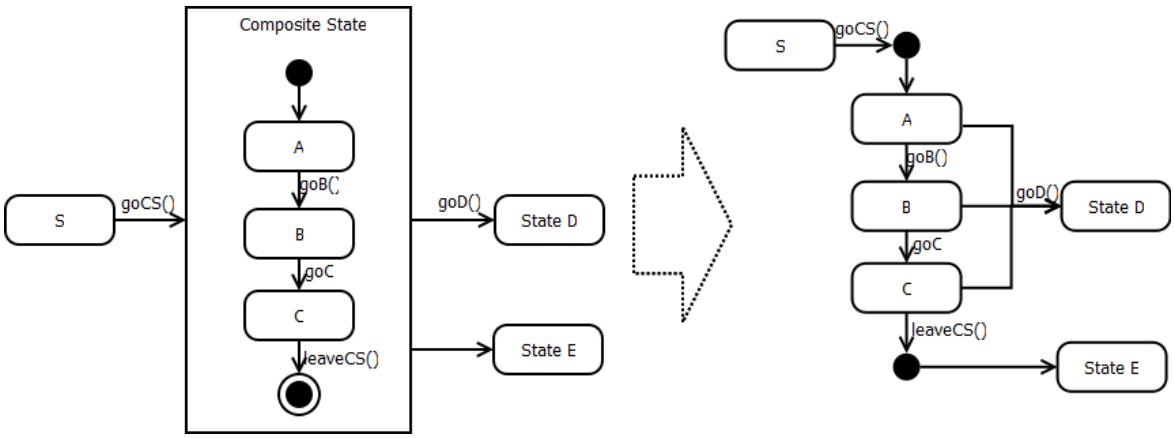

Abbildung 3.4: Auflösen eines Composite States

Auflösen von Submachine States

Es müssen zuerst 2 Arten von Statemachines unterschieden werden, welche von Submachine State referenziert werden können:

- Einfache Submachines
- Rekursive Submachines: Eine Submachine S ist rekursiv, wenn diese (direkt oder indirekt) eine Referenz zu sich selbst hat (d.h. in einer von S eingeschlossen Region gibt es einen Submachine State, welcher S referenziert)

Generelle Regeln für Submachines

1. Für jeden Submachine State S ersetze diesen durch einen Composite State C und kopiere die Region von S referenzierten Submachine M zu der eingeschlossenen Region von C (Achtung: es können mehrere Duplikate der Region entstehen, falls M öfters referenziert wird)
2. Wende die Regeln für Composite States an

Zusätzliche Regeln für Rekursive Submachines

1. Wenn sich der Submachine State S (mit Referenz auf Statemachine M) in einer duplizierten Region von M befindet (direkt oder indirekt), dann ändere die Targets aller Transitions, welche S als Target besitzen, auf den Composite State, welcher diese duplizierte Region beinhaltet

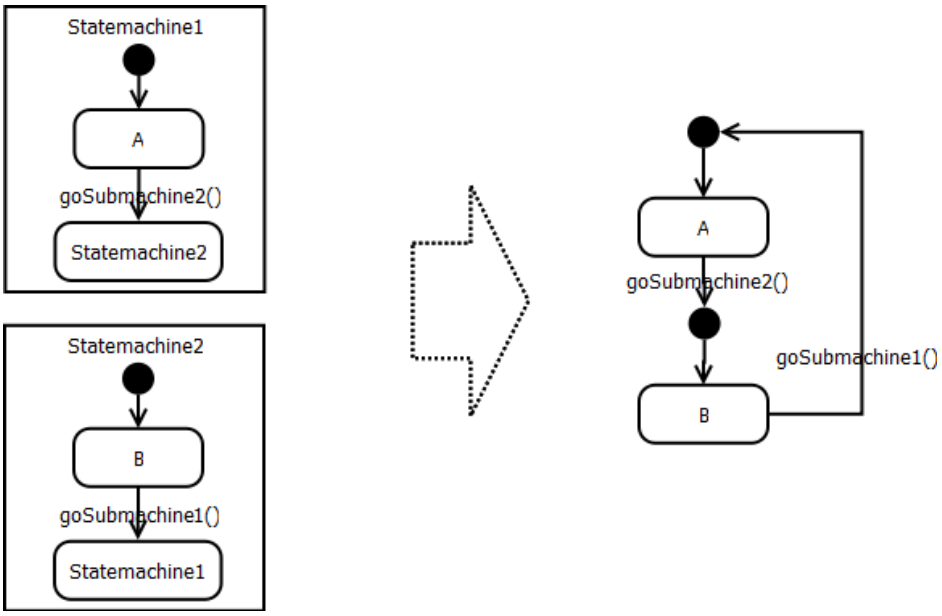

Abbildung 3.5: Auflösen einer Rekursiven Submachine

4 Statistische Testverfahren

Dieser Abschnitt befasst sich mit automatischer Testfallgenerierung und -selektion auf Basis von statistischen Methoden. Dabei wird auf Testfallgenerierung mit wahrscheinlichkeitsbasierten und risikobasierten Methoden eingegangen. Bei beiden Varianten werden jeweils zuerst Möglichkeiten gezeigt, wie die dazu benötigten Werte (Übergangswahrscheinlichkeiten bzw. Risikofaktoren) ermittelt werden können (manuell, semi- oder vollautomatisch). Danach werden Algorithmen zur Testfallgenerierung gezeigt, welche diese Werte nützen.

4.1 Statistisches Testen

Unter statistischen Testen versteht im allgemeinen das Nutzen statistischer Methoden (z.B. Wahrscheinlichkeitsberechnungen, Statistische Metriken, ...) innerhalb des Test-Workflows. Dabei können statistische Methoden sowohl bei der Erstellung von Testfällen als auch bei der Analyse von Testergebnissen genutzt werden. Statistisches Testen ist dabei weniger eine eigenständige Testart, sondern kann zur Erweiterung/Verbesserung vorhandener Testarten (z.B. Regression Testing, Back-To-Back Testing, ...) verwendet werden.

4.1.1 Statistische Testautomatisierung mit Benutzungsmodellen

Ein von einem Benutzungsmodell abgeleiteter Testfall kann als Sequenz aus Inputs und erwarteten Outputs beschrieben werden. Ein Benutzungsmodell kann dabei im Bereich des statistischen Testens um 2 wichtige Punkte erweitert werden:

- Testsequenzen (Testfälle) werden basierend auf einer Wahrscheinlichkeitsverteilung, welche ein Profil für die Benutzung der Software repräsentiert, generiert.
- Eine statistische Analyse wird auf die Testhistorie (d.h. am SUT durchgeführte Testfälle) angewandt, was eine Messung unterschiedlicher Wahrscheinlichkeitsaspekte des Testablaufs ermöglicht.

Statistisches Testen kann also als Sequenzgenerierungs- und Analyseproblem betrachtet werden. Dieses Kapitel befasst sich mit dem Sequenzgenerierungsproblem (bzw. Pfadgenerierung im Kontext eines Graphen).

4.2 Markov Chains

Die folgenden mathematischen Grundlagen über *Markov Chains* (dt. Markovketten) stellen eine vereinfachte Erklärung aus [CT06] dar und fokusieren Eigenschaften, welche für darauf aufbauende Benutzungsmodelle (siehe 4.2.1) und Testverfahren relevant sind.

Im folgenden beschreibt $Pr(X = x|Y = y)$ die Wahrscheinlichkeit, dass $X = x$ unter der Bedingung, dass $Y = y$. Weiters beschreibt $p_n(..)$ die Wahrscheinlichkeit im n-ten Schritt.

Ein stochastischer Prozess X_i ist eine Folge von zeitlich geordneten Zufallsvariablen (bzw. zufälligen Vorgängen). Ein stochastischer Prozess X_i mit $i > 0$ mit Werten in Z (endlich oder abzählbar - wir betrachten nur endliche) wird Markov Chain genannt, wenn für alle $x_1, x_2, ..., x_{i+1} \in Z$ (mit $i = 1, 2, ...$) gilt

$$Pr(X_{n+1} = x_{n+1}|X_n = x_n, ..., X_1 = x_1) = Pr(X_{n+1} = x_{n+1}|X_n = x_n) = p_{n+1}(x_{n+1}|x_n)$$

sofern

$$Pr(X_n = x_n, ..., X_1 = x_1) > 0$$

d.h. bei zwei aufeinanderfolgenden Zuständen x_i, x_{i+1} ist die Übergangswahrscheinlichkeit für den Folgezustand x_{i+1} nur von dem direkt vorher kommenden Zustand x_i abhängig, nicht jedoch von *älteren* Zuständen in der Folge X_i.

Z heißt dabei der *state space* (dt. Zustandsraum) der Markov Chain.

Die Markov Chain heisst *time homogeneous* (dt. Zeit-homogen), wenn

$$p_n(y|x) = p(y|x) = p_{x,y}$$

d.h. es liegt eine einstufige Übergangswahrscheinlichkeit vor. Vereinfacht ausgedrückt heißt das, dass die Übergangswahrscheinlichkeit von Zustand x nach Zustand y sich nie ändert (unabhängig davon, wie oft und zu welcher Zeit der Übergang auftritt).

Die Übergangswahrscheinlichkeiten $p_{x,y}$ mit $x, y \in Z$ lassen sich entweder als stochastische Matrix

$$P = (p_{x,y})_{x,y \in Z}$$

oder als Graph (siehe Bild 4.1 aus dem Beispiel) darstellen.

Beispiel 4 *Ein Zustandsraum $Z = a, b, c$ hat folgende Übergangswahrscheinlichkeiten*

$$p_{a,a} = 0, p_{a,b} = p_{a,c} = \frac{1}{2}, p_{b,a} = p_{b,c} = \frac{1}{4}, p_{b,b} = p_{c,c} = \frac{1}{2}, p_{c,a} = p_{c,b} = \frac{1}{4}$$

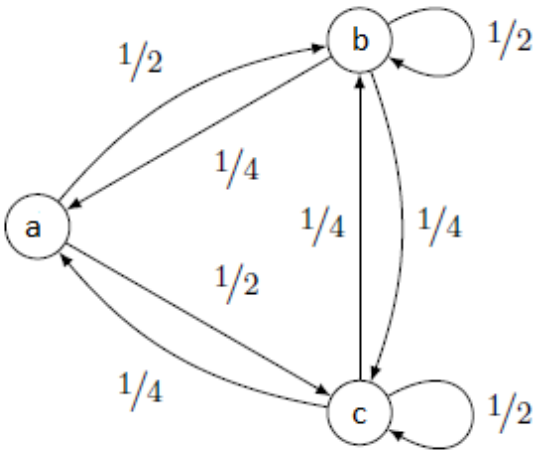

Abbildung 4.1: Übergangswahrscheinlichkeiten einer Markov Chain als Graph

Dies kann als folgende Matrix dargestellt werden:

$$P = \begin{pmatrix} 0 & 1/2 & 1/2 \\ 1/4 & 1/2 & 1/4 \\ 1/4 & 1/4 & 1/2 \end{pmatrix}$$

Bild 4.1 zeigt den dazu passenden Graphen. Nichtmögliche Übergänge (wie $p_{a,a} = 0$) werden nicht visualisiert.

4.2.1 Markov Chain Usage Model (MCUM)

Bereits im Jahre 1994 stellten James A. Whittaker und Michael G. Thomason ein Benutzungsmodell basierend auf Markov Chains vor [WT94], welches als *Markov Chain Usage Model* (MCUM) oder als *Usage Markov Chain* bekannt ist. Markov Chains werden hier zum einen als Sequenzgenerator (damals wurden nur Sequenzen von Inputs betrachtet) für statistisches Testen verwendet, weiters werden sie auch zur statistischen Analyse verwendet (Resultate werden zusammen mit den Markov Chains zur Testplanung verwendet).

Aufbau des MCUM

Ein MCUM besteht aus einer Menge an Zuständen (states) und Transitionen (transitions). Die Zustände stellen die unterschiedlichen Zustände dar, welche die Software annehmen kann. Transitionen beschreiben die Übergänge zwischen den Zuständen und beinhalten Systeminputs (welche den Übergang auslösen) sowie Übergangswahrscheinlichkeiten (transition probabilities). Zusätzlich werden aus der Menge von States ein Startzustand (start state)

und Terminierungszustand (termination state) gewählt, welche den Anfang und Ende der Benutzung markieren. Das MCUM kann also als 5-Tupel .

Definition 3 (MCUM) *Ein MCUM kann als 5-Tupel* $MCUM = S, L, T, s_s, s_t$ *beschrieben werden, wobei*

- S *ist eine Menge an Zuständen*
- L *ist eine Menge von Markierungen (labels), bestehend aus einem Systeminput sowie einer Übergangswahrscheinlichkeit*
- $S \times L \times S \rightarrow T$ *ist eine Menge an Transitionen, welche die Übergänge zwischen den States beschreiben*
- $s_s \in S$ *ist der Startzustand des MCUM*
- $s_t \in S$ *ist der Terminierungszustand (bzw. Endzustand) des MCUM*

Die Übergangswahrscheinlichkeiten (transition probabilities) stellen die Wahrscheinlichkeit dar, dass dieser Übergang (ausgehend von dem Zustand, aus welchem die Transition genutzt werden kann) gewählt wird. Bei der Generierungen von Sequenzen (Testfällen) können diese Wahrscheinlichkeiten genutzt werden um bessere (sinnvollere) Benutzerwege zu simulieren.

Anpassungen für das ALTS als MCUM

Das ALTS und MCUM sind sich vom Grundaufbau sehr ähnlich, der Unterschied liegt hauptsächlich in der Markierung. In MCUM bestehen Markierungen nur aus einem Input und einer Übergangswahrscheinlichkeit. Eine Markierung (Label) in einem ALTS besteht hingegen aus einer Bedingung (condition), einem Trigger (Aktivität, welche auch ein Input darstellen kann) und einer Menge von resultierenden Effekten (Effects). MCUM‘s enthalten weder Bedingungen noch Effekte (Übergänge sind immer möglich). Damit eine ALTS auch als MCUM interpretiert werden kann, muss man diese Markierung um die Übergangswahrscheinlichkeit erweitern. Es gilt also

$$L = (c, t, E, p)$$

eine Markierung (Label) besteht also wie bisher aus einer Condition c, einem Trigger t, einer Menge von Effekten E und nun zusätzlich einer Übergangswahrscheinlichkeit p.

> Anmerkung: Durch die gerade beschriebene Erweiterung kann das ALTS nur bedingt als Markov Chain interpretiert werden. Das Problem ist, dass diese Markov Chain nicht Zeit-homogen ist, da die Übergangswahrscheinlichkeit einer Transition auf 0 sinkt, wenn die Bedingung nicht erfüllt wird (d.h. die Transition kann dann nicht getriggert werden). Diese Problematik wird nochmals in 4.2.2 genauer erörtert.

4.2.2 Algorithmen für faire Verteilung

Nachdem nun der Sinn von Markov Chains bzw. Übergangswahrscheinlichkeiten erklärt wurden, stellt sich die Frage, wie man diese Wahrscheinlichkeiten definiert. Hierfür gibt es grundsätzlich 3 verschiedene Ansätze:

1. **Manuelles Definieren der Wahrscheinlichkeiten:** Ein Systemexperte, welcher Wissen über das Nutzerverhalten hat, definiert die einzelnen Wahrscheinlichkeiten manuell.
2. **Automatisches Definieren durch Nutzungsanalyse:** Nutzungsverhalten wird automatisch aufgenommen (z.B. durch *Datalogging*) und daraus wird ein Nutzungsprofil erstellt, welches aus den einzelnen Wahrscheinlichkeiten besteht.
3. **Automatisches Definieren durch Gleichverteilung (faire Verteilung):** Hier wird nicht Nutzungsverhalten simuliert, jedoch wird versucht, eine Gleichverteilung des Markov Chains zu erreichen (d.h. jeder Pfad innerhalb des Markov Chains ist gleich wahrscheinlich).

Der erste Ansatz, manuelles Definieren, ist ein leicht zu realisierender Ansatz und kann immer durchgeführt werden, sofern ein Systemexperte mit dem nötigen Wissen zur Verfügung steht. Jedoch ist dieser Ansatz sehr fehleranfällig (menschliche Fehler) und mit viel Aufwand und dadurch hohen Kosten verbunden.

Automatisches Definieren durch Nutzungsanalyse ist ein sehr interessanter Ansatz, da hier durch die Analyse von echtem Nutzern auch ein sehr realistisches Ergebnis auf automatisiertem Weg erzielt werden kann, entweder als Durchschnitt von vielen Nutzern oder mehrere Nutzerprofile einzelner Nutzer bzw. Nutzergruppen. Jedoch bringt dieser Verfahren rechtliche Gefahren mit sich, da Nutzerverhalten erfasst wird. Man muss also hierfür freiwillige Nutzer finden. Weiters ist es nötig, dass die Software entweder schon besteht oder zumindest ein Betastadium erreicht hat, damit dieses Verfahren angewendet werden kann.

In den meisten Fällen genügt es jedoch schon, wenn alle Komponenten mit der gleichen Wahrscheinlichkeit getestet werden (siehe [FG10] [WP00]). In diesem Abschnitt werden nun mehrere (approximative) Algorithmen zur automatischen Generierung von Übergangswahrscheinlichkeiten vorgestellt, welche eine solche Gleichverteilung (im folgenden auch *faire Verteilung* genannt) ermöglichen.

Formulierung des Problems

In diesem Abschnitt wird erklärt, was man überhaupt unter einer gleichverteilten Markov Chain versteht und weshalb diese sinnvoll ist. Das allgemeine Ziel ist es, aus einem gegebenen Benutzungsgraphen (Usage Graph) ein MCUM zu erstellen, bei welcher theoretisch jeder Pfad, welcher zwischen Start- und Endknoten möglich ist, mit der gleichen Wahrscheinlichkeit auftritt.

Grundsätzlich würde man die Wahrscheinlichkeitsverteilung eines Graphen als fair betrachten, wenn an jedem Knoten alle wegführenden Transitionen mit der gleichen Wahrscheinlichkeit

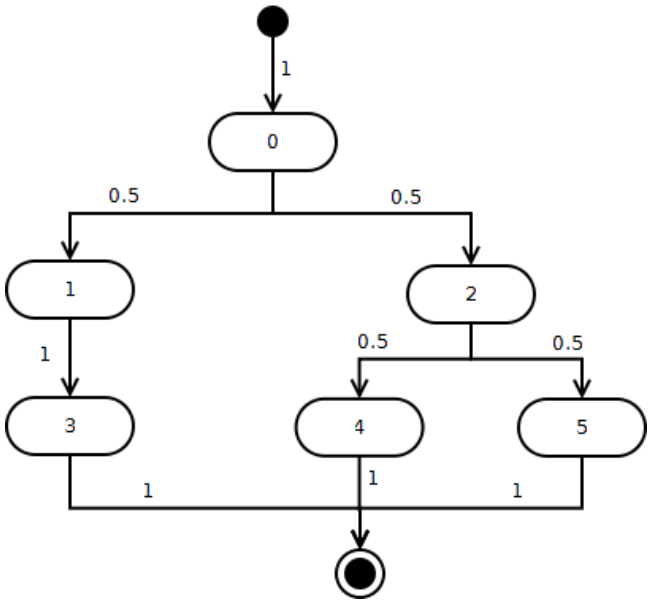

Abbildung 4.2: Markov Chain mit Standardverteilung

auftreten. Dies wäre auch zu jenem Zeitpunkt, wenn man sich in diesem Zustand befindet, ein faires Auswahlverfahren. Jedoch gilt dies nur für diesen Augenblick bzw. Pfade der Länge 1, man nenn eine solche Verteilung auch *Standardverteilung*.

Beispiel 5 *Bild 4.2 zeigt ein sehr einfaches Beispiel, in welchem schon das Problem einer solchen Standardverteilung erkennbar ist. Es ist recht leicht zu erkennen, dass die 3 möglichen Pfade zwischen Start- und Endknoten mit unterschiedlicher Wahrscheinlichkeit auftreten (Pfad* $0 \to 1 \to 3$ *zu 50% und die Pfade* $0 \to 2 \to 4$ *bzw.* $0 \to 2 \to 5$ *jeweils nur zu 25%).*

Um eine faire Verteilung über den gesamten Graphen (d.h. alle möglichen Pfade sollten mit der gleichen Wahrscheinlichkeit auftreten) zu erhalten, müssen streng genommen alle möglichen Pfade miteinander verglichen werden, wodurch man ein NP-vollständiges Problem erhält (insbesondere da Zyklen möglich sind). Das Ziel ist es deshalb in der Praxis meistens, eine gute Approximation für dieses Problem zu finden. Bild 4.3 zeigt das vorherige Beispiel mit einer Gleichverteilung.

In dem (einfachen) Beispiel 5 ist die Berechnung der Übergangswahrscheinlichkeiten relativ einfach, da eine endliche Anzahl von Pfaden existiert. Man muss dazu einfach nur mitzählen, wie oft jede Transition insgesamt (in allen möglichen Pfaden) vorkommt und wie oft jeder Knoten vorkommt.

Die Übergangswahrscheinlichkeit für eine Transition ist einfach mit der Formel

Formel 1

$$p_{i,j} = \frac{count(t_{i,j})}{count(i)}$$

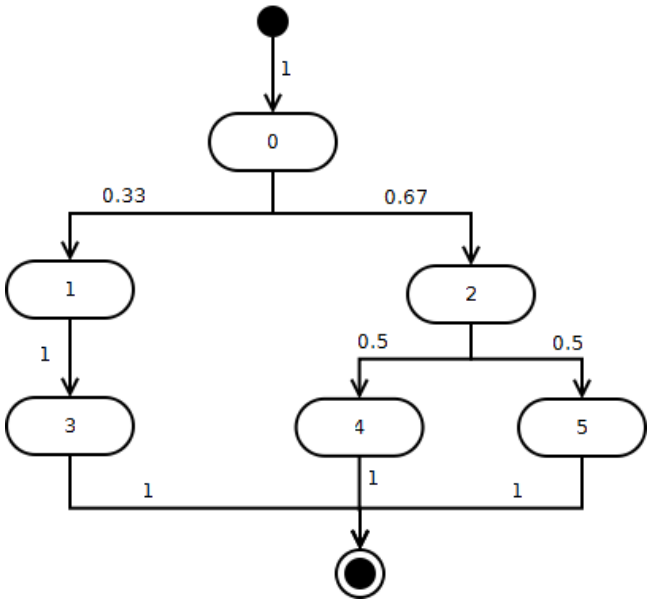

Abbildung 4.3: Markov Chain mit Gleichverteilung

zu berechnen, wobei i der Knoten ist, von welchem die Transition $t_{i,j}$ wegführt und $count(x)$ die Anzahl des Auftretens von x (Transition oder Knoten) in allen Pfaden ist.

Beispiel 6 *Für das vorherige Beispiel ergeben sich für die Pfade*

$$Paths = \{\{0,1,3\},\{0,2,4\},\{0,2,5\}\}$$

folgende Anzahl der Knoten

$$count(0) = 3, count(1) = 1, count(2) = 2, count(3) = 1, count(4) = 1, count(5) = 1$$

und folgende Anzahl der Transitionen

$$count(t_{0,1}) = 1, count(t_{1,3}) = 1, count(t_{0,2}) = 2, count(t_{2,4}) = 1, count(t_{2,5}) = 1$$

daraus erhält man folgende Übergangswahrscheinlichkeiten

$$p_{0,1} = \frac{1}{3} = 0.33, p_{1,3} = \frac{1}{1} = 1, p_{0,2} = \frac{2}{3} = 0.67, p_{2,4} = \frac{1}{2} = 0.5, p_{2,5} = \frac{1}{2} = 0.5$$

Es genügt jedoch schon ein einfacher Zykel im Graphen, damit diese Berechnungsmethode nicht mehr möglich ist (da die Anzahl der möglichen Pfade dann unendlich wird). Aber auch bei endlichen Mengen von Pfaden ist es bei komplexeren Beispielen in der Regel sehr aufwändig, alle Pfade zu vergleichen.

Im folgenden werden nun 3 Algorithmen zur (approximierten) Berechnung/Generierung solcher Wahrscheinlichkeitsprofile vorgestellt. Grundlagen zu den Algorithmen sind aus [FG10] sowie in [Thi03] und [Cho78]. Weitere Algorithmen und Methoden dieser Art sind zum Beispiel in [WP00] beschrieben.

Adaptierung des Chinese Postman Problems

Eine einfache Methode für eine approximative Berechnung ist eine Adaptierung des Chinese Postman Problems, welches in Abschnitt 2.2.3 erklärt wurde (Grundlagen hierzu sind in [FG10] zu finden). Im Gegensatz zu der im letzten Abschnitt vorgestellten Methode werden nicht alle Pfade zur Berechnung herangezogen, sondern nur eine minimale Menge an Pfaden, mit welchem jede Transition zumindest einmal abgedeckt wurde (durch Anwenden des Chinese Postman).

Benutzungsgraphen (bzw. Transitionssysteme) sind normalerweise gerichtete Graphen und verfügen über einen Startknoten, welcher nur aus ausgehenden Transitionen besteht und über einen Endknoten, welcher nur aus einkommenden Transitionen besteht. Damit dieser also überhaupt als Chinese Postman Problem definiert werden kann, muss eine fiktive Transition (bzw. gerichtete Kante) von dem Endknoten zum Startknoten hinzugefügt werden (diese Transition signalisiert auch das Ende bzw. Beginn eines neuen Testfalls).

Algorithmus 3 (Chinese Postman Gleichverteilung) *Eine möglichst faire Wahrscheinlichkeitsverteilung für einen Graphen erhält man wie folgt:*

1. *Ermittle den kürzesten Pfad als Lösung des Chinese Postman Problem und dabei zähle mit, wie oft jede Transition und wie oft jeder Knoten vorkommt.*
2. *Berechne die Transitionswahrscheinlichkeiten (transition probabilities) mit der Formel* $p_{i,j} = \frac{count(t_{i,j})}{count(i)}$*, wobei* i *und* j *Knoten sind und* $t_{i,j}$ *die Transition von Knoten* i *nach Knoten* j *darstellt.*

Beispiel 7 *Für das vorhin verwendete Beispiel 5 würde man mit dieser Methode die gleiche Lösung erhalten (da alle möglichen Pfade nötig sind um alle Transitionen abzudecken). Bild 4.4 zeigt einen (zyklenfreien) Graphen (mit fiktiver Transition zwischen End- und Startknoten), bei dem nicht alle Pfade nötig sind für eine komplette Transitionsabdeckung (transition coverage). Mit der Menge*

$$\{\{1,2,4\},\{1,3,5\},\{1,2,3,4\}\}$$

erhält man den kürzesten Weg um alle Transitionen abzudecken, die Pfade $\{1,3,4\}$ *und* $\{1,2,3,5\}$ *werden nicht berücksichtigt. Man erhält also die Anzahl der Knoten*

$$count(1) = 3, count(2) = 2, count(3) = 2, count(4) = 2, count(5) = 1$$

und Anzahl der Transitionen

$$count(t_{1,2}) = 2, count(t_{1,3}) = 1, count(t_{2,3}) = 1,$$

$$count(t_{2,4}) = 1, count(t_{3,4}) = 1, count(t_{3,5}) = 1$$

daraus erhält man folgende Übergangswahrscheinlichkeiten

$$p_{1,2} = 2/3 \approx 0.67, p_{1,3} = 1/3 \approx 0.33, p_{2,3} = 1/2 = 0.5,$$

$$p_{2,4} = 1/2 = 0.5, p_{3,4} = 1/2 = 0.5, p_{3,5} = 1/2 = 0.5$$

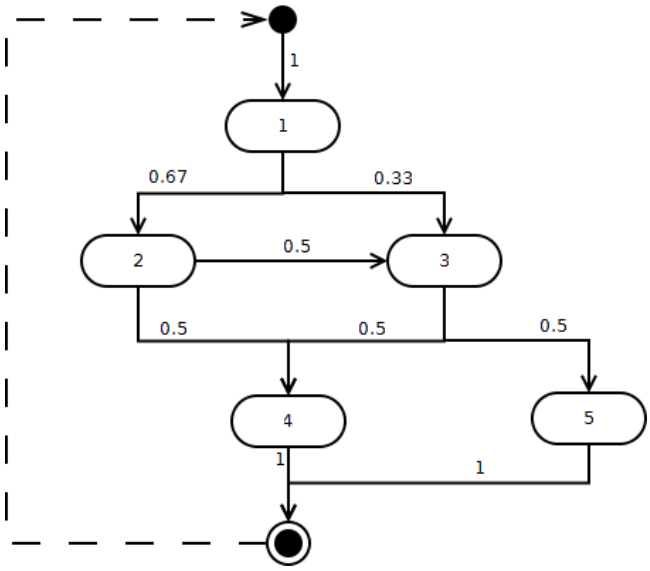

Abbildung 4.4: Übergangswahrscheinlichkeiten mit Chinese Postman Adaptierung

Im Gegensatz dazu würde man bei der Berücksichtigung aller Pfade folgende Werte erhalten: Anzahl der Knoten

$$count(1) = 5, count(2) = 3, count(3) = 4, count(4) = 3, count(5) = 2$$

Anzahl der Transitionen

$$count(t_{1,2}) = 3, count(t_{1,3}) = 2, count(t_{2,3}) = 2,$$

$$count(t_{2,4}) = 1, count(t_{3,4}) = 2, count(t_{3,5}) = 2$$

Übergangswahrscheinlichkeiten

$$p_{1,2} = 3/5 = 0.6, p_{1,3} = 2/5 = 0.4, p_{2,3} = 2/3 \approx 0.67,$$

$$p_{2,4} = 1/3 = 0.33, p_{3,4} = 2/4 = 0.5, p_{3,5} = 2/4 = 0.5$$

Dieses Verfahren kann zwar mit Zyklen umgehen, jedoch werden diese nicht speziell behandelt (sie werden nur einmal durchlaufen), wodurch nicht sichergestellt ist, dass man bei zyklischen Graphen ein gutes Ergebnis erhält.

Chow's Algorithmus

Chow's Algorithmus (Grundlagen: siehe [Cho78]) geht noch etwas einfacher vor. Es werden nicht mögliche Pfade erstellt, sondern es wird zuerst ein möglichst kleiner Testbaum (test tree) aus dem Graphen/Modell generiert, welcher alle Transitionen abdeckt (man benötigt also keine kompletten Pfade, wie es bei der vorherigen Version nötig war). Dabei wird der Baum nach der Methode der Breitensuche traversiert (*breadth first traversal*).

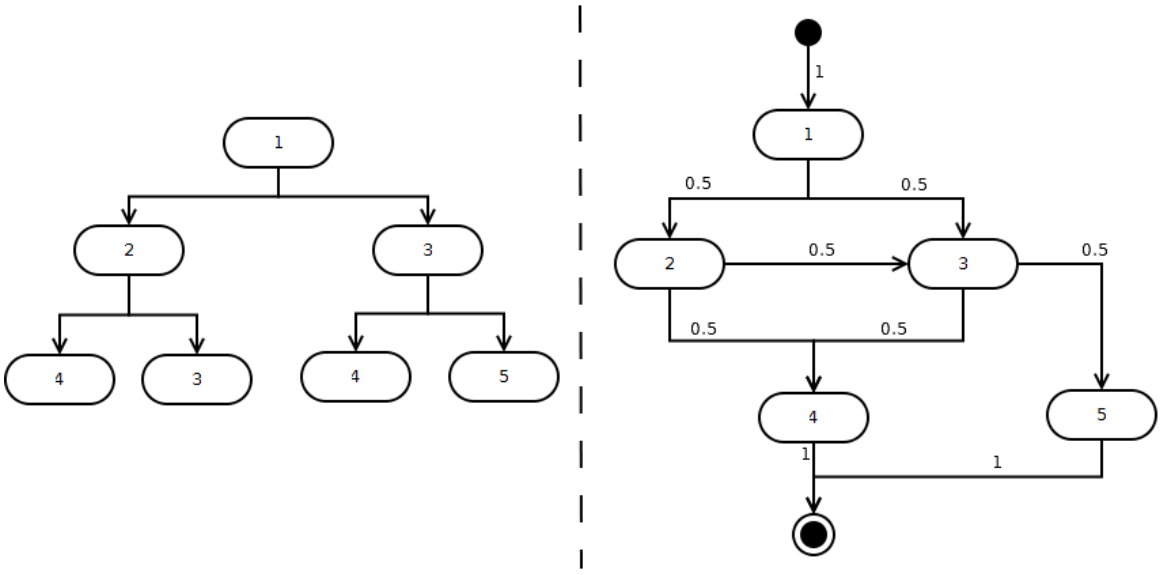

Abbildung 4.5: Testbaum und Wahrscheinlichkeiten nach Chow‘s Algorithmus

Algorithmus 4 (Chow‘s Algorithmus) *Ausgehend von einem gegebenen Graphen mit Startknoten s, führe folgende Schritte durch:*

1. *Erstelle einen leeren Baum tree*
2. *Erstelle eine Liste todo und füge den Startknoten s hinzu*
3. *Entferne den ersten Knoten k aus der Liste todo*
4. *Für jede ausgehende Transition t am Knoten k, welche sich t noch nicht im Baum tree befindet:*
   a) *Füge die Transition t dem Baum tree hinzu*
   b) *Füge den Zielknoten der Transition t der Liste todo hinzu*
5. *Falls die Liste todo nicht leer ist, gehe zu Schritt 3 zurück. Ansonsten endet der Algorithmus.*
6. *Berechne die Anzahl der Knoten und Transitionen. Gezählt wird jedoch, indem man den Baum für alle Blätter jeweils von der Wurzel aus traversiert und mitzählt, wie oft jede Transition und jeder Zustand/Knoten vorkommt (Blätter werden nicht mitgezählt).*
7. *Berechne die Transitionwahrscheinlichkeit einer Transition $t_{i,j}$ mit der Formel $p_{i,j} = \frac{count(t_{i,j})}{count(i)}$*

Beispiel 8 *Bild 4.5 (links) zeigt den Testbaum, welchen man durch das Anwenden der Breitensuche für das Beispiel aus dem vorherigen Abschnitt (siehe Bild 4.4) erhält.*

Man erhält dadurch folgende Anzahl der Knoten

$$count(1) = 4, count(2) = 2, count(3) = 2$$

und folgende Anzahl der Transitionen

$$count(t_{1,2}) = 2, count(t_{1,3}) = 2, count(t_{2,3}) = 1, count(t_{2,4}) = 1,$$

$$count(t_{3,4}) = 1, count(t_{3,5}) = 1$$

daraus erhält man folgende Übergangswahrscheinlichkeiten

$$p_{1,2} = 2/4 = 0.5, p_{1,3} = 2/4 = 0.5, p_{2,3} = 1/2 = 0.5, p_{2,4} = 1/2 = 0.5,$$

$$p_{3,4} = 1/2 = 0.5, p_{3,5} = 1/2 = 0.5$$

In diesem kleinen Beispiel erhält man als Resultat sogar die Standardverteilung, was daran liegt, dass der Testbaum symmetrisch ist. Jedoch stellt die Standardverteilung in diesem Beispiel auch keine schlechte Lösung dar.

Algorithmus für zyklische Graphen

Die beiden vorangegangenen Algorithmen nehmen beide (wie schon erwähnt) keinen speziellen Bezug auf Zyklen, auch wenn sie damit prinzipiell umgehen können (beide Algorithmen terminieren auch bei Zyklen, jedoch kann auf Zyklen nicht direkt eingegangen werden). Deshalb wird in diesem Abschnitt noch ein Algorithmus aus [FG10] präsentiert, welcher es ermöglicht, genauer auf Zyklen einzugehen. Dadurch kann ein besseres Testen von Zyklen während der Generierungsphase ermöglicht werden.

Bei diesem Algorithmus wird die Berechnung für die Wahrscheinlichkeit einer Transition $t_{i,j}$ mit der Formel

Formel 2

$$p_{i,j} = \frac{T_{i,j}}{N_i}$$

durchgeführt, wobei

- N_i angibt, wieviel mögliche Pfade von Zustand i zum Endzustand existieren
- $T_{i,j}$ angibt, wieviel mögliche Pfade von Transition $t_{i,j}$ zum Endzustand exisitieren

Algorithmus 5 (Zyklenbezogene Pfadwahrscheinlichkeiten) *Der Algorithmus berechnet rekursiv einen Testbaum mit allen möglichen nichtzyklischen Pfaden. Aufkommende Zyklen werden mit einer Gewichtung i belegt (i kann als Parameter für den Algorithmus angegeben werden). Folgender Pseudocode [FG10] beschreibt den Algorithmus:*

```
IN:     graph G; Int i;
OUT:    weight matrix T; weight vector N, E
        s: State; Ch: list of State;

BEGIN
        s := initial state ;
        Ch: = empty list;
        T and E are initialized to 0;
        N [final state] :=1;
        Generate(s, Ch, i);
END


FUNCTION Generate (s, Ch, i)
BEGIN
        Add s to Ch;
        FOR each sx successor of s do
          IF Ch does not contains sx then
            IF E[sx] = 0 then // Unprocessed paths
              Generate(sx, Ch);
            END IF
            N [s] := N[s] + N[sx];
            T[s, sx] := N[sx] ;
          ELSE // cycle
            N[s] := N[s] + i; //  i: weight given to cycles
            T[s, sx] := i;
          END IF
        END FOR
END
```

Beispiel 9 *Angewandt an das vorangegangene Beispiel (welches jedoch keine Zyklen enthält, deshalb wird $i = 1$ gesetzt), erhält man folgende Werte:*
Anzahl von möglichen Pfaden ausgehend von Knoten

$$N_1 = 5, N_2 = 3, N_3 = 2, N_4 = 1, N_5 = 1$$

Anzahl von möglichen Pfaden ausgehend von Transitionen

$$T_{1,2} = 3, T_{1,3} = 2, T_{2,3} = 2, T_{2,4} = 1, T_{3,4} = 1, T_{3,5} = 1$$

Übergangswahrscheinlichkeiten

$$p_{1,2} = 3/5 = 0.6, p_{1,3} = 2/5 = 0.4, p_{2,3} = 2/3 \approx 0.67, p_{2,4} = 1/3 = 0.33,$$

$$p_{3,4} = 1/2 = 0.5, p_{3,5} = 1/2 = 0.5$$

In diesem Beispiel erhält man sogar das optimale Resultat, was daran liegt, dass alle möglichen Pfade in der Berechnung berücksichtigt wurden.

Vergleich der Algorithmen

Als einfacher Test, ob ein Verfahren ein brauchbares Resultat liefert (zumindest für das verwendete Beispiel), kann man die Wahrscheinlichkeit des Auftretens der einzelnen Pfade berechnen (mit $prob(Path) = \prod p_{i,j}, \forall t_{i,j} \in Path$, also das Produkt aller Übergangswahrscheinlichkeiten eines Pfades) und miteinander vergleichen. Bei einem guten Resultat weichen die einzelnen Pfadwahrscheinlichkeiten nicht stark voneinander ab. Tabelle 4.1 zeigt die resultierenden Wahrscheinlichkeiten der einzelnen Pfade für das verwendete Beispiel.

Pfad	Standard	Optimal	CPA 4.2.2	Chow 4.2.2	Zyklen 4.2.2
$\{1,2,4\}$	0.25	0.2	0.335	0.25	0.2
$\{1,3,4\}$	0.25	0.2	0.165	0.25	0.2
$\{1,3,5\}$	0.25	0.2	0.165	0.25	0.2
$\{1,2,3,4\}$	0.125	0.2	0.1675	0.125	0.2
$\{1,2,3,5\}$	0.125	0.2	0.1675	0.125	0.2

Tabelle 4.1: Pfadwahrscheinlichkeiten der einzelnen Verfahren

Dieses kurze Beispiel ist aufgrund seiner Einfachheit natürlich nicht gerade sehr aussagekräftig und enthält auch keine Zyklen. Für einen besseren Vergleich wurden die Algorithmen an dem *Car Alarm* Beispiel (siehe 2) verglichen. Bild 4.6 zeigt das Beispiel mit den berechneten Wahrscheinlichkeiten. Tabelle 4.2 zeigt die resultierenden Wahrscheinlichkeiten einzelner (stichprobenartig ausgewählter) Pfade. Die Pfade werden dabei durch ihre Trigger beschrieben (I...Init(), O...Open(), C...Close(), L...Lock(), U...Unlock, S...Shutdown(), W...Wait()).

Pfad	Standard	CPA 4.2.2	Chow 4.2.2	Zyklen (i=3) 4.2.2
{I,C,O,L,S}	0.00084	0.00014	0.00014	0.000069
{I,C,L,W,U,S}	0.00042	0.00015	0.00015	0.000069
{I,C,L,W,O,W,U,S}	0.00010	0.00015	0.00015	0.000069
{I,C,L,W,O,W,W,C,W,U,S}	0.0000089	0.000017	0.000017	0.0000084
{I,C,O,C,L,U,O,L,C,W,U,L,S}	0.0000000017	0.00000000003	0.00000000003	0.000000000025

Tabelle 4.2: Pfadwahrscheinlichkeiten des Car Alarm Beispiels

Grenzen der Algorithmen

Mit den 3 vorgestellten Algorithmen erhält man relativ gute Übergangswahrscheinlichkeiten, die in Folge zur Optimierung bei der Testgenerierung genutzt werden können. Jedoch bestehen 2 grundsätzliche Problematiken bei den Resultaten:

- Wahrscheinlichkeit von zyklischen Pfaden sinkt sehr schnell

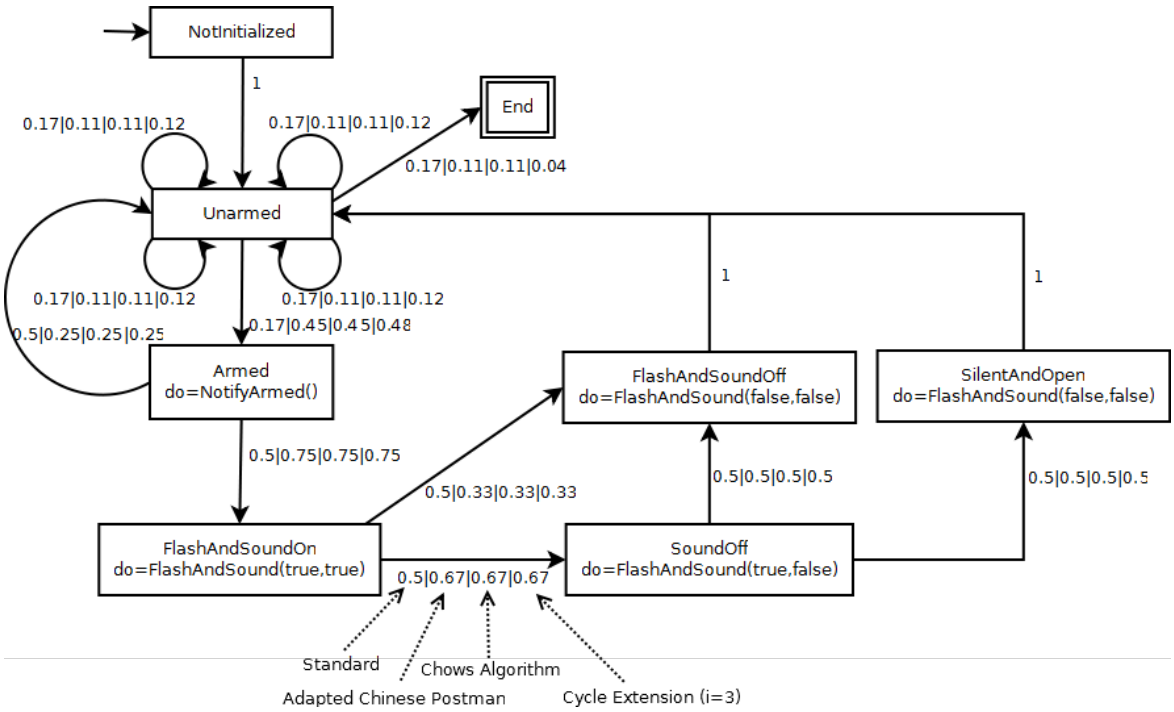

Abbildung 4.6: Car Alarm mit berechneten Wahrscheinlichkeiten

- Ergebnisse sind auf zeithomogene Markov Chains optimiert, ALTS-Modelle sind jedoch meist zeitinhomogen (aufgrund von *Guards*)

Die stark sinkende Wahrscheinlichkeit von zyklischen Pfaden ist ein generelles Problem, da bei öfterem Wiederholen von Zyklen die Wahrscheinlichkeit des Pfades immer sinkt. Es ergibt sich überhaupt schon die Frage, wie oft es sinnvoll ist, eine Kette von Aktionen zu wiederholen. Der 3te Algorithmus 4.2.2 gibt hier schon einen recht interessanten und steuerbaren Ansatz, indem man Zyklen eine spezielle Gewichtung vorgeben kann. Dadurch werden Pfade, welche wiederholende Zykeln enthalten, schon etwas mehr berücksichtigt (weniger Wiederholungen sind dementsprechend wahrscheinlicher und mehr Wiederholungen unwahrscheinlicher). Die anderen Algorithmen optimieren die Übergangswahrscheinlichkeiten immer auf nur einen Durchlauf, was jedoch nicht immer am sinnvollsten ist (manche Fehler entstehen z.B. erst durch wiederholte Ausführung der gleichen Aktionskette).

Das größere Problem ist die Tatsache, dass mit den Algorithmen die Übergangswahrscheinlichkeiten nur auf zeithomogene Markov Chains optimiert werden. ALTS-Modelle können jedoch Bedingungen enthalten, durch welche Transitionen zu gewissen Zeiten (abhängig von Testdaten und vorher durchlaufenen Teilpfaden) eventuell nicht passierbar sind, wodurch deren Übergangswahrscheinlichkeit auf 0 sinkt. Dadurch wird das Modell zeitinhomogen und die Wahrscheinlichkeiten (zu diesem Zeitpunkt) möglichen Transitionen ändern sich ebenfalls (da zu jedem Zeitpunkt die Bedingung $\Sigma p_i = 1$ gelten muss). Einzig mit Algorithmus 3 können diese Bedingungen berücksichtigt werden (durch einen angepassten Chinese Postman), jedoch erhält man auch hier nicht mit Sicherheit ein gutes Resultat. Grundsätzlich lässt sich sagen,

dass die vorgestellten Algorithmen nur bei Modellen ohne (bzw. mit sehr wenigen) *Guards* eingesetzt werden sollten.

4.3 Wahrscheinlichkeitsbasierte Algorithmen zur Pfadgenerierung

Im folgenden werden nun Algorithmen zur Pfadgenerierung vorgestellt, welche die Resultate und Konzepte des vorherigen Abschnitts nutzen bzw. als Ausgangsbasis für spätere Weiterverarbeitungsschritte (z.B. Testfallpriorisierung) dienen können.

4.3.1 Random Walk

Der Random Walk (dt. Zufallslauf) ist einer der einfachsten Algorithmen um Pfade (oder in unserem Fall Testfälle) zu generieren. Dabei wird zuerst der Startknoten gewählt und dann immer per Zufall eine der derzeit möglichen Kanten/Transitionen gewählt. Dabei kann prinzipiell zwischen zwei Arten von Random Walk‘s unterschieden werden:

- Rein zufallsbasierter Random Walk: Es wird per Zufall eine möglichen Kanten gewählt, wobei keine Kanten bevorzugt werden (jede Kante wird mit der gleichen Wahrscheinlichkeit gewählt).
- Wahrscheinlichkeitsbasierter Random Walk: Die möglichen Kannten werden bzgl. ihrer Wahrscheinlichkeitsverteilung bevorzugt gewählt (eine Kante mit höherer Wahrscheinlichkeit wird eher gewählt als eine mit niedriger Wahrscheinlichkeit). Handelt es sich bei der Wahrscheinlichkeitsverteilung um die Standardverteilung bekommt man einen *rein zufallsbasierten Random Walk*.

Die Zufallsläufe terminieren jeweils nach einer vorgegebenen Schrittanzahl.

Algorithmus 6 (Wahrscheinlichkeitsbasierter Random Walk) *Gehe wie folgt vor:*

1. *Wähle einen Zahl $D > 0$ (z.B. 100) und ermittle mittels Zufallsgenerator eine zufällige ganze Zahl R, welche sich $0 \leq R < D$.*
2. *Normalisiere die Übergangswahrscheinlichkeiten aller derzeit möglichen Kanten T (sodass $\sum p_i = 1$, wobei p_i die Übergangswahrscheinlichkeit der Kante $t_i \in T$).*
3. *Setze $i = 0$*
4. *Berechne eine Schranke $M = D \cdot \sum_{j=0}^{i} p_j$*
5. *Falls $R \leq M$ wähle Kante t_i als nächste Kante, ansonsten erhöhe i mit $i = i + 1$ und kehre zu Schritt 4 zurück.*

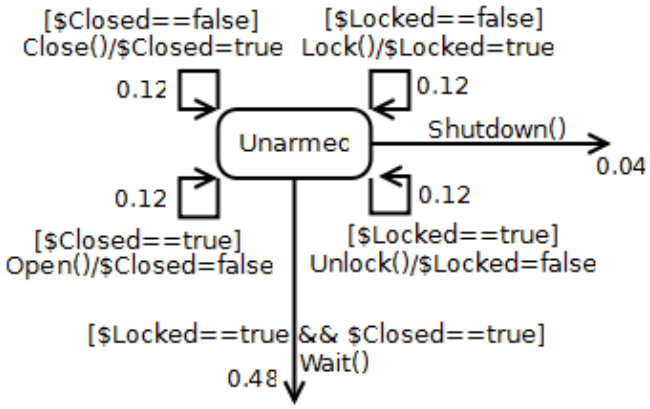

Abbildung 4.7: Zustand *Unarmed* mit wegführenden Transitionen

Beispiel 10 *Man betrachtet den Knoten Unarmed aus dem Car Alarm Beispiel (siehe Bild 4.7). Er enthält 6 ausgehende Transitionen, welche jedoch nie alle gleichzeitig wählbar sind. Man nimmt an, die Variablen sind gesetzt auf* $\$Locked = false$ *und* $\$Closed = false$*, dann sind die Transitionen* $T = \{t_{Lock()}, t_{Close()}, t_{Shutdown()}\}$ *(Reihenfolge zufällig gewählt) mit den Übergangswahrscheinlichkeiten* $P = \{p_{Lock()}, p_{Close()}, p_{Shutdown()}\}$ *möglich. Zuerst müssen ihre Übergangswahrscheinlichkeiten normalisiert werden:*

$$p_{Lock()} := \frac{p_{Lock()}}{\sum_{p_i \in P} p_i} = \frac{p_{Lock()}}{p_{Lock()} + p_{Close()} + p_{Shutdown()}} = \frac{0.12}{0.12 + 0.12 + 0.04} = 0.4285,$$

$$p_{Close()} := 0.4285, p_{Shutdown()} := 0.143$$

Man wählt nun als $D = 100$ *und ermitteln mit dem Zufallsgenerator die Zufallszahl* $R = 80$*. Nun setzt man* $i = 0$ *und berechnen die Schranke*

$$M = D \cdot \sum_{j=0}^{0} p_j = 100 \cdot 0.4285 = 42.85$$

Da 80 nicht kleiner oder gleich 42.85 ist, erhöht man $i = i + 1 = 1$ *und wiederholt die Berechnung der Schranke*

$$M = D \cdot \sum_{j=0}^{1} p_j = 100 \cdot (0.4285 + 0.4285) = 85.7$$

Es gilt $80 \leq 85.7$*, weshalb man* $t_1 = t_{Close()}$ *als nächste Tansition wählt. Bild 4.8 zeigt das Auswahlverfahren als Grafik.*

4.3.2 Most-Probable-Neighbor Walk

Der *Most-Probable-Neighbor Walk* ist ein einfacher Algorithmus, welcher für TAI selbst entwickelt wurde und im Gegensatz zu einem *Random Walk* nicht zufallsbasiert ist. Er basiert grundsätzlich auf dem Algorithmus 2 wie der Chinese Postman, jedoch terminiert er nach Erreichen einer vorgegebenen Schrittanzahl (und nicht bei 100% Kantenabdeckung).

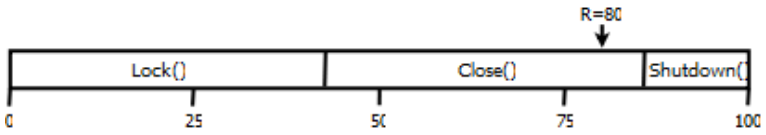

Abbildung 4.8: Kantenauswahl Random Walk

Algorithmus 7 (Most-Probable-Neighbor) *Diese Version basiert auf der Erweiterten Heuristik aus Algorithmus 2. Das Gewicht einer Transition t wird hierbei mit der Formel*

$$G(t) = \frac{1}{P(t)} \cdot 2^{Visits(t)}$$

berechnet, wobei P(t) die Übergangswahrscheinlichkeit der Transition t darstellt und Visits(t) die Anzahl, wie oft die Transition in dem Pfad schon besucht wurde. Bei jedem Schritt des Most-Probable-Neighbor Walk wird nun die Transition gewählt, für welche die kleinste Gewichtung G(t) berechnet wurde. Der Algorithmus terminiert jedoch nicht nach dem Besuch aller Transitionen, sondern nach dem Erreichen einer vorgegebenen Schrittanzahl.

Die Gewichtung ist also umso kleiner, je höher die Übergangswahrscheinlichkeit ist und wird bei jedem Besuch verdoppelt. Durch die dynamische Änderung der Gewichtung wird unter anderem mehr Variation erreicht.

> Anmerkung: Denkbar wäre auch eine Erweiterung der Gewichtungsformel um einen Zufallsfaktor (wodurch etwas mehr Streuung der Testfälle erreicht werden könnte).

4.3.3 Regression Testing mit probabilistischen Algorithmen

In diesem Abschnitt wird darauf eingegangen, wie die gerade vorgestellten probabilistische Algorithmen (wie z.B. Random Walk 4.3.1) für Regression Testing eingesetzt werden können.

Das grundsätzliche Problem ist, dass beim Regression Testing immer die gleichen Testfälle durchgeführt werden müssen/sollten. Wenn also Testfälle neu generiert werden (da Änderungen im Modell notwendig waren, z.B. weil sich gewisse Anforderungen geändert haben), sollten sich neu generierte Testfälle optimalerweise von den alten nur in den geänderten Bereichen unterscheiden. Grundsätzlich ist es aber vor allem wichtig, dass nach einer Neugenerierung die gleichen Aktionsabläufe vorhanden sind, welche die Software testen. Probabilistische Algorithmen sind in ihrer Reinform jedoch meist nicht reproduzierbar. Dies liegt an folgenden Problematiken:

- durch Zufallsfaktoren erzielt man bei jedem Anwenden unterschiedliche Resultate
- grobe Änderungen in der Modellstruktur können auch grundsätzlich zu anderen Resultaten führen

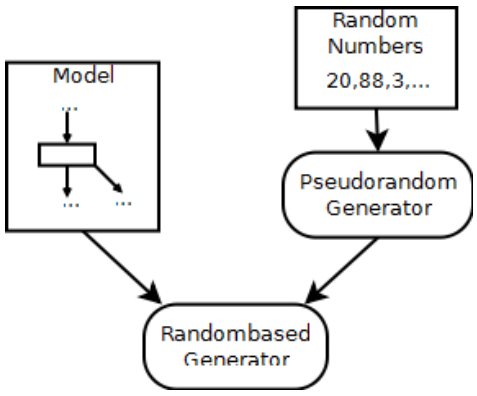

Abbildung 4.9: Pseudozufallszahlen bei der Testfallgenerierung

Probabilistische Verfahren sind in den meisten Fällen vom Zufall abhängig. Die Steuerung des Zufalls mit probabilistischen Werten ist eine große Stärke dieser Algorithmen, da sie trotz einer gewissen Kontrolle immer noch einen Spielraum für Unvorhersehbarkeit zulassen. Dadurch können unter Umständen Fehler entdeckt werden, welche exakte Algorithmen nicht finden würden, jedoch sind die generierten Testfälle nicht reproduzierbar. Dies ist bei manchen Testverfahren sogar ein großer Vorteil, beim Regression Testing ist dies jedoch eher ein Problem.

Um grundsätzlich Testfälle für das Regression Testing zu erhalten, hat es jedoch durchaus Sinn, solche Algorithmen zu verwenden (da sehr effizient relativ gute Testfälle generiert werden können). Bei einer nötigen Neugenerierung würden jedoch mit großer Wahrscheinlichkeit andere Testfälle generiert werden, was nicht im Sinne dieses Testverfahrens liegt.

Ein Vorschlag, um dieses Problem in den Griff zu bekommen, ist die Verwendung eines Pseudozufallsgenerators, welcher die Reproduzierung von Zufallswerten ermöglicht (siehe Bild 4.9). Realisiert kann das z.B. werden, indem man im ersten Schritt mit einem herkömmlichen Zufallszahlengenerator eine Menge von n Zufallszahlen $X = X_1, ..., X_n$ generiert. Bei der Generierung der Testfälle wird nun der Pseudozufallsgenerator verwendet, welcher jeweils die nächste Zahl von X ausgibt (X_i beim i-ten Aufruf, also $X_1, X_2, ...$, nach dem n-ten Aufruf wird wieder bei 1 begonnen, also $..., X_{n-1}, X_n, X_1, ...$). Dadurch erhält man bei jedem Anwenden die gleichen Testfälle, solange sich die Modellstruktur nicht zu stark ändert.

Das größere Problem, welches grundsätzlich für alle *nichtgesteuerten* Pfadgenerierungsalgorithmen (d.h. ohne Vorgabe, was getestet werden soll) gilt, ist die Tatsache, dass bei größeren Änderungen der Struktur meist auch komplett andere Pfade generiert werden. Dies ist jedoch davon Abhängig, wie stark die Algorithmen von der Struktur des Modells abhängig sind. Probabilistische Algorithmen sind meist sehr stark von der Struktur abhängig, da die Pfade meist nur nach statistischen Werten beeinflusst werden.

Bei der Generierung von Regressionstests sollten deshalb Testziele vorhanden sein, um gleiche (oder zumindest äquivalente) Testfälle bei wiederholtem Generieren zu erhalten. Solche Testfälle können z.B. Zustände oder Transitionen sein, welche für einen (in dieser Testfallmenge) akzepierten Testfall erreicht werden müssen. In [SW] wird beispielsweise ein

recht gutes Verfahren beschrieben, um aus UML Statemachines auf diese Art und Weise Testziele zu definieren. Hier werden sogenannte *Test Definitions* erstellt, indem man eine geordnete Folge von Transitionen (für eine bestimmte Statemachine) definiert. Diese *Test Definition* beschreibt aber nicht ganze Testfälle, sondern nur, welche Transitionen in den generierten Testfälle vorkommen müssen (und zwar in der vorgegebenen Reihenfolge). Bei der Generierung werden dann alle möglichen Testfälle erstellt (innerhalb einer maximalen Anzahl an Zyklen), welche die *Test Definition* erfüllen.

Jedoch gibt es auch statistische Mögichkeiten, um die Generierung von Testfällen mit Vorgaben zu beeinflussen. Ein Beispiel hierfür ist das Risikobasierte Testen, welches im nächsten Abschnitt 4.4 behandelt wird.

4.4 Risikobasiertes Testen

Softwaretesten bzw. Testen im allgemeinen hat das grundsätzliche Problem, dass das zu testende System (SUT) ab einer gewissen Komplexität nie zu 100% getestet werden kann. Ein Modell beschreibt im Allgemeinen schon mal gar nicht das gesamte System und aus dem Modell können in der Regel nicht alle möglichen Testfälle durchgeführt werden. Es gilt

$$Durchführbare Testfälle \subseteq Modell \subseteq SUT$$

Die Anzahl der durchführbaren Testfälle hängt dabei in der Praxis von den zur Verfügung stehenden Ressourcen (z.B. verfügbare Zeit, Tester, ...) ab, welche aus wirtschaftlichen Gründen meist recht knapp gehalten werden. Gerade bei solch aufwendigen Testverfahren wie dem Regression Testing, in welchem die Testfälle sehr oft wiederholt werden müssen, sollte die Anzahl der durchzuführenden Testfälle so gering wie möglich gehalten werden (Testfallautomatisierung ermöglicht hier zwar eine Erhöhung der durchführbaren Testfälle, aber auch hier sollte ein sinnvolles Maximum eingehalten werden). Die durchführbare Menge von Testfällen sollte dabei das System so gut wie möglich testen. Grundsätzlich sollte man dabei folgende Vorgaben erfüllen:

- Alle Bereiche des Systems sollten zumindest einmal getestet werden (Abdeckung)
- Wichtigere/Kritischere Bereiche sollten genauer gestestet werden (Fokusierung)

Bei der Generierung bzw. Auswahl von Testfällen für ein komplettes SUT oder einzelne Komponenten werden also Verfahren benötigt, welche Möglichkeiten bieten, solche Vorgaben zu erfüllen. Eine Abdeckung des gesamten SUT ist relativ einfach überprüfbar bzw. durch *Transition Coverage*-Algorithmen (z.B. Chinese Postman) auch gut automatisierbar. Schwieriger ist das Erreichen des zweiten Punkts, das Bevorzugen von kritischeren Bereichen. Man braucht hierfür

- Die Möglichkeit zur Kennzeichnung bzw. Modellierung wichtiger/kritischer Bereiche
- Bewertungsverfahren für die Bevorzugung dieser Bereiche

Risikobasiertes Testen ist ein solches Verfahren, welches Möglichkeiten bietet, diese Vorgaben zu erreichen. Wichtigere Bereiche können hier mittels Risikoabschätzung und -modellierung ermittelt werden. Unterschiedliche Bewertungsverfahren helfen dann bei der Generierung bzw. Auswahl der durchführbaren Testfälle.

4.4.1 Berechnung von Risiken

Der folgende beschriebene Ansatz zur Risikomodellierung und -abschätzung basiert grundsätzlich auf der Recherche aus [Aml99] und [Bac99] bzw. den eigenen Erfahrungen, welche bei einer Fallstudie (siehe A.2) gewonnen wurden.

Modellierung von einzelnen Risiken

Einfach ausgedrückt beschreibt das Risiko (z.B. eines Systems, einer Aktivität, ...) das Auftreten eines ungewolltes Verhaltens, welches mit einer gewissen Wahrscheinlichkeit auftritt und Schaden bzw. Kosten verursacht. Dabei spricht man von einem hohen Risiko, wenn dieses Verhalten entweder sehr wahrscheinlich auftritt und/oder wenn das Problem hohe Kosten verursacht. Dadurch erkennt man schon, aus welchen Faktoren ein Risiko (*Risikofaktoren*) grundsätzlich besteht:

- Wahrscheinlichkeit, dass ein Problem beim Ausführen einer bestimmten Aktivität auftritt (Probability of Fault)
- Kosten, welches ein Problem bei einer bestimmten Aktivität verursacht (Cost)

Je nach Art des Risikos können diese Faktoren dabei unterschiedlich stark gewichtet werden (beispielsweise sollte bei einem Atomkraftwerk der Schaden stärker gewichtet werden als die Wahrscheinlichkeit des Auftretens).

Es gibt natürlich auch weitere Risikofaktoren wie z.B. entstehende Schäden, Performanceeinflüsse, zeitliche Verzögerungen, etc. Weiters können auch die bestehenden Faktoren weiter verfeinert werden (z.B. Aufteilung von *Kosten* in *Wartungskosten* und *Entstehende Kosten während des Ausfalls*).

Um Risiken automatisiert zu verarbeiten bzw. für Algorithmen zu nutzen, wird ein Risikomodell benötigt, mit welchem ein Risiko mathematisch modelliert werden kann. Grundsätzlich ist es hierfür erstmals wichtig, eine sinnvolle Skala zu definieren, mit welcher die Risikofaktoren als Zahlen repräsentiert werden können (z.B. ganze Zahlen von 1 bis 5, wobei 1 einen niedrigen Risikofaktor und 5 einen hohen Risikofaktor darstellt). Für jeden Risikofaktor kann hierbei eine eigene Skala definiert werden. Ist man z.B. der Meinung, dass in dem zu testenden System die Kosten entscheidender für das Risiko wie die Auftrittswahrscheinlichkeiten sind (z.B. in Finanzsoftware, bei der z.B. eine Fehlerhafte Buchung extremen Schaden verursachend würde), kann man für Kosten eine größere Skala verwenden als für die Auftrittswahrscheinlichkeit, dadurch wird der Kostenfaktor bei der Risikoberechnung stärker gewichtet.

Mit den skalierten Risikofaktoren kann nun eine mathematische Risikoformel definiert werden, welche beschreibt, wie die entsprechenden Risikofaktoren zusammenhängen. Eine einfache

und deshalb gern verwendete Risikoformel (unter anderem in [YR03], [Hem] und [Aml99]) ist

Formel 3

$$R(a) = P(a) \cdot C(a)$$

wobei a eine beliebige Aktivität und $R(a)$ dessen Risiko darstellt, bestehend aus den Faktoren

- $P(a)$, der Wahrscheinlichkeit, dass die Aktivität a einen Fehler verursacht
- $C(a)$, die Kosten, welche bei einm Fehler in Aktivität a entstehen

Beispiel 11 *Möchte man die Kosten z.B. durch Wartungskosten ($MC(a)$) und Kundenschaden ($CD(a)$) verfeinern, könnte ein entsprechenden Modell wie folgt aussehen:*

$$R(a) = P(a) \cdot \frac{MC(a) + CD(a)}{2}$$

Je nach Art des zu testenden Systems sollte immer individuell entschieden werden, welche Risikofaktoren für den Erfolg des Systems entscheidend sind und wie stark dabei ihre Gewichtung ist. Eine Social Media Plattform hat zum Beispiel ganz andere Risikofaktoren und Gewichtung (Datenschutz) als eine Bankomatsoftware (fehlerhafte Abbuchung), weshalb bei beiden ein anderes Risikomodell verwendet werden sollte.

4.4.2 Abschätzen von Risikofaktoren

Um für das Risiko einen sinnvollen Wert zu erhalten, welcher das Risiko in Relation mit den restlichen Risiken im System gut repräsentiert, muss man die Risikofaktoren mit sinnvollen Werten belegen. Solche Faktoren kann man aber nicht einfach messen, sie müssen abgeschätzt werden. Hierfür gibt es 2 Ansätze:

1. Abschätzen mittels Softwaremetriken, z.B. für Codelänge, Codecomplexität, Bughistory, etc.
2. Manuelles Abschätzen, bei dem Projektbeteiligte (Entwickler, Tester, etc.) eine Schätzung abgeben

Abschätzen mittels Softwaremetriken

Risikoabschätzung mittels Softwaremetriken werden unter anderem in [FHB11] und [Aml99] behandelt. Bei diesem Verfahren werden einzelne Metriken der Software zur Berechnung der Risikofaktoren verwendet.

Beispiel 12 *Es wird eine Formel für den Risikofaktor Probability of Failure definiert, welche den Faktor mittels Codelänge, Codekomplexität und Bughistory (bisheriger Versionen der Software) berechnet:*

$$ProbabilityOfFailure(a) = Codelength(a) \cdot 0.2 + Complexity(a) \cdot 0.4 + Bughistory(a) \cdot 0.4$$

Bei diesem Beispiel wird fällt die Metrik für die Codelänge weniger ins Gewicht(wobei natürlich die einzelnen Metriken zuvor noch im Verhältnis zur Skala angepasst werden müssen). Für die Berechnung von Wartungskosten könnte ebenfalls die Codecomplexität verwendet werden (komplexerer Code ist aufwändiger zu Warten).

Der Vorteil des Abschätzens mittels Softwaremetriken ist, dass diese zum Teil sehr gut automatisierbar sind und dadurch weniger aufwändig ermittelbar sind (Entwicklungsumgebungen wie Visual Studio stellen beispielsweise statistische Daten über die Implementation zur Verfügung, welche verwendet werden können). Jedoch hat dieser Ansatz mehrere Nachteile:

- Nur möglich, wenn entsprechende Metriken existieren (eine Bughistory beispielsweise existiert nur bei späteren Versionen einer Software und ist nur sinnvoll, wenn ein gutes Bugreporting betrieben wird)
- Aussagekraft von manchen Metriken sind sehr fragwürdig (inwiefern beeinflusst beispielsweise die Codelänge die Fehlerwahrscheinlichkeit)
- Manche Risikofaktoren können nicht mit Softwaremetriken berechnet werden (Kosten für den Kunden bei Fehlerfall ist beispielsweise nicht innerhalb der Software erfassbar)
- Manche Aktivitäten, für welche das Risiko berechnet werden soll, sind schwer bis unmöglich als Code in der Implementation erfassbar

Grundsätzlich sollte man nie rein mit Softwaremetriken Risikofaktoren abschätzen, sondern diese eher als Unterstützung für manuelles Abschätzen verwenden.

Manuelles Abschätzen von Risikofaktoren

Das manuelle Abschätzen von Risikofaktoren ist ein in der Praxis recht beliebtes Verfahren und ermittelt die einzelnen Risikofaktoren durch die Erfahrung einzelner Projektbeteiligter. Um ein gutes Resultat zu erhalten, ist es aber sehr entscheidend, wie man die Erfahrung der einzelnen Beteiligten für die Risikoabschätzung nutzt. Die im Folgenden beschriebene Vorgangsweise basiert grundsätzlich auf [Aml99], wurde jedoch für eigene Zwecke angepasst und erweitert.

Als erstes muss man sich die Frage stellen, welche Projektbeteiligten (sofern sie zur Verfügung stehen) über welche Risikofaktoren bescheid wissen:

- Entwickler sind Experten für die Implementation und können dadurch beispielsweise die Fehlerwahrscheinlichkeit und Wartungskosten gut abschätzen
- Der Productowner ist Experte für die Anforderungen und Wirtschaftlichkeit der Software, er kann jegliche Art von Kosten und Schäden gut abschätzen
- Tester kennen die Software aus Nutzer- und Entwicklersicht, können also prinzipiell jede Art von Risikofaktor abschätzen

Wichtig ist dabei auch, dass man eine Gewichtung der einzelnen Beteiligten festlegt. Ein Productowner weiss natürlich mehr über anfallende Kosten beim Ausfall einzelner Softwareteile bescheid als ein Tester, er sollte also bei dieser Frage deshalb auch stärker gewichtet werden. Ein Entwickler kennt die technischen Details der Implementation und kann dadurch deren Komplexität und Fehleranfälligkeit gut abschätzen. Andererseits hat ein Tester eine objektivere Sicht und kann die Fehleranfälligkeit durch seine Erfahrungen mit älteren Versionen (sofern vorhanden) besser abschätzen. Gerade bei solchen Fragestellungen ist es oft nicht einfach, eine passende Gewichtung zu finden. Jedoch ist nicht nur die Projektrolle entscheidend über die Gewichtung sondern auch, wie lange ein Beteiligter schon im Projekt arbeitet (jemand, der beispielsweise schon 3 Jahre an einem Projekt arbeitet, kann über gewisse Faktoren bessere Schätzungen abgeben, als jemand, der neu dazu gekommen ist).

Bevor man nun aber ein Meeting durchführt, bei dem die Beteiligten ihre Schätzungen abgeben, sind noch folgende Vorbereitungen notwendig:

- Festlegen, für welche Aktivitäten oder Softwarebereiche ein Risiko ermittelt werden soll
- Genaue Beschreibung der einzelnen Risikofaktoren und der Aktivitäten/Softwarebereiche, welche abgeschätzt werden sollen
- Festlegung der Vorgehensweise zum Abschätzen (unterschiedliche Rollen einzeln befragen, ermitteln durch gemeinsame Diskussion oder Durchnittswert, etc.)

Eine speziell angepasste und (innerhalb des Fallbeispiels) durchgeführte Variante der manuellen Risikoabschätzung wird im Anhangskapitel A.2 beschrieben.

4.4.3 Modellieren von Risiken im Benutzungsmodell

Wie vorher schon öfters erwähnt wurde, bezieht sich ein Risiko immer auf eine bestimmte Aktivität oder einen Teil der Software. Der Bezug der Risiken zu den dabei durchgeführten Aktivitäten muss irgendwie beschrieben werden. Einfache, nicht modellbasierte, Verfahren sind beispielsweise das Beschreiben der Risiken in Tabellenform. Im Bereich des modellbasierten Testens ist es natürlich naheliegend, Risiken direkt auf den Modellen zu definieren, zumal gerade Benutzungsmodellen selber schon Aktivitäten beschreiben.

Bei graphenähnlichen Benutzungmodellen wie Statemachines oder Transitionssystemen, welche also grundsätzlich aus Knoten und Kanten bestehen, werden im Normalfall einfach die

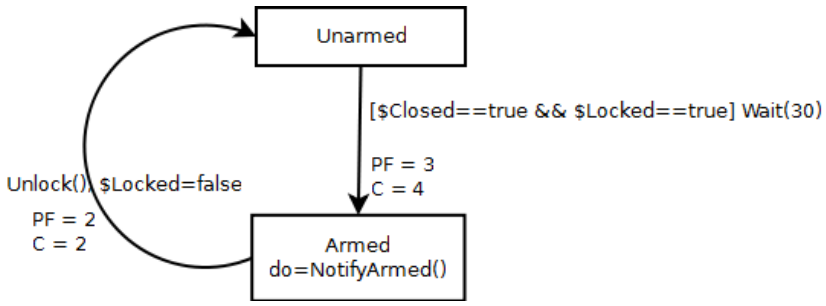

Abbildung 4.10: Einfache Risikoannotierung in Benutzungsmodellen

entsprechenden Elemente (Knoten und/oder Kanten), welche die Aktivität repräsentieren, mit den Risikofaktoren annotiert.

Beispiel 13 *Bild 4.10 zeigt ein Beispiel für solch einfache Risikoannotierungen, wobei PF für Probability of Failure (Wahrscheinlichkeit, dass ein Fehler auftritt) und C für Cost (Kosten, falls der Fehler auftritt) steht.*

Da meistens aber schon relativ kleine Benutzungsmodelle aus relativ vielen einzelnen Aktivitäten bestehen (welche im Modell als Transitionen repräsentiert werden) und diese oftmals sehr atomar sind (z.B. Mausklick), ist es oft sehr schwer, Risiko sinnvoll auf einzelne Aktivitäten aufzuteilen. Deshalb kam die Idee auf, ein Konzept für *Regionale Risiken* zu definieren, welche sich auf ganze Bereiche beziehen. Innerhalb von UML Statemachines können diese bereichsbezogenen Risiken einfach an eine *Region* (also auf Substatemachines oder Composite States) annotiert werden, im ALTS beziehen sich diese Risiken dann einfach auf die entsprechende Menge von Transitionen.

Beispiel 14 *Bild 4.11 zeigt ein Beispiel mit regionalen Risiken.*

Das Risiko einer Region bezieht sich auf alle Transitionen, welche zu Elementen innerhalb dieser Region führen (Ziel der Transition). Von der Region wegführende Transitionen sind nicht davon betroffen. Dies wird dadurch gerechtfertigt, dass Aktivitäten, welche zu einer Risikoregion führen, selber schon Teil der Risikoregion sind. Man befindet sich durch das Ausführen dieser Aktivtät danach in der Region. Von der Risikoregion wegführende Transitionen gehören nicht zu der Risikoregion.

Es ist dadurch natürlich auch möglich, dass mehrere Risiken für eine Transition/Aktivität relevant sind (ein eigenes Risiko und eine oder mehrere Risiken von Regionen, in welchen die Transition auftritt). Dies kann Sinn machen, wenn man ein Risiko innerhalb einer Risikoregion besonders hervorheben möchte. Man berechnet dazu die Summe aller Risiken, welche für die Transition relevant sind.

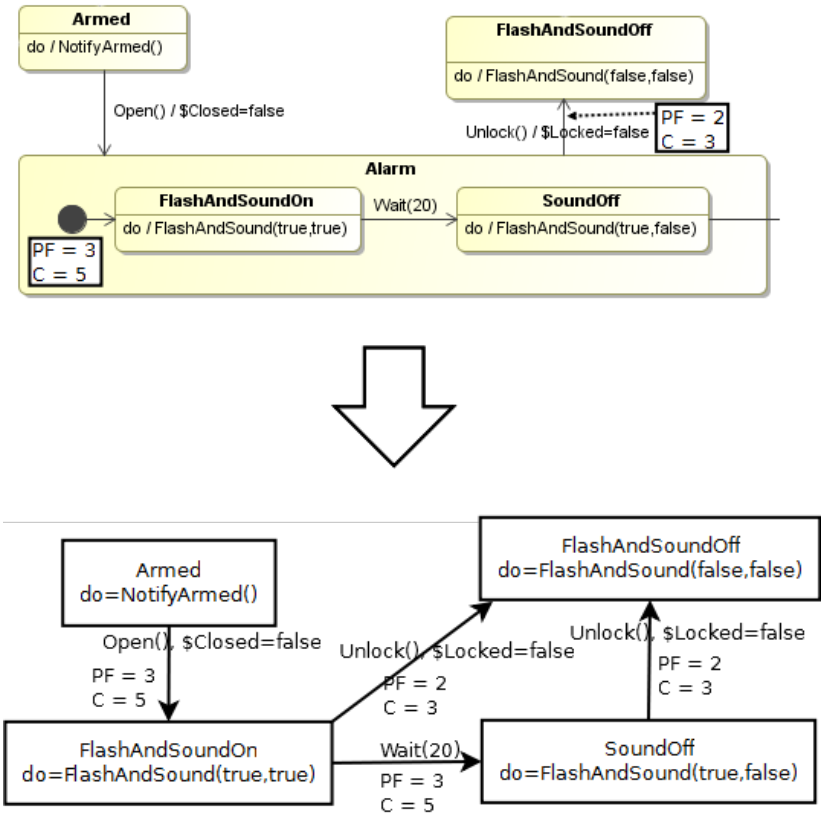

Abbildung 4.11: Regionale Risikoannotierung in Benutzungsmodellen

Erweiterung des ALTS für Risikomodellierung

Damit die Risiken, wie gerade vorgestellt, innerhalb eines ALTS modelliert werden können, muss das ALTS dahingend erweitert werden. Das in 3 beschriebene 7-Tupel ALTS=(A,V,S, s_i, S_f, L, T) wird hierbei zu einem 8-Tupel ALTS=(A,V,S, s_i, S_f, L, T, RF) erweitert, wobei RF eine Menge von annotierten Risikofaktoren darstellt, definiert als

$$RF = \{(type, value, RT) | RT \subseteq T\}$$

Ein Risikofaktor besteht also aus einem Risikofaktortyp (*type*, z.B. *Probability of Failure*), einem Risikofaktorwert (*value*) und einer Menge an Transition (RT), auf welche sich der Risikofaktor bezieht. Ein Beispiel für die graphische Darstellung dieser Erweiterung ist in Bild 4.11 dargestellt.

Im Modell werden nur einzelne Risikofaktoren modelliert und nicht das fertige Risiko. Dadurch kann die Risikoformel (zur Berechnung des eigentlichen Risikos) jederzeit ausgetauscht werden, wodurch die Modellierung flexibler wird.

4.5 Testfallerzeugung mittels risikobasierten Modellen

Nachdem man nun Benutzungsmodelle erstellt und diese mit Risiken versehen hat, möchte man diese Daten natürlich dafür nutzen, um daraus passende Testfälle abzuleiten. Das Ziel ist es hierbei, eine Menge an Testfällen bzw. eine Testsuite zu erzeugen, welche in möglichst kurzer Zeit, also mit einer minimalen Anzahl an nötigen Testschritten, das SUT möglichst gut bezüglich dessen Risiko testet.

Hierfür wird nun zuerst der übliche Weg, bei dem die Risiken für eine Testfallpriorisierung 4.5.1 genutzt werden, beschrieben. Danach wird ein eigener Ansatz vorgestellt, welcher die Risiken schon direkt bei der Generierung berücksichtig 4.5.2.

4.5.1 Risikobasierte Testfallpriorisierung

Der übliche Weg, risikobasierte Testfälle zu erhalten, ist die risikobasierte Testfallpriorisierung. Die Risikoannotierungen des Modells werden hierbei noch nicht bei der Testfallgenerierung verwendet, sondern erst bei einer späteren Priorisierung erzeugter Testfälle. Die übliche Vorgangsweise ist dabei folgende:

- Erstelle ein Modell (z.B. als Statemachine) mit Risikoannotierungen
- Nutze bekannte Algorithmen (z.B. Random Walk, Chinese Postman, etc.) um Testfälle aus dem Modell zu generieren
- Analysiere die generierten Testfälle (bzgl. annotierten Risiken) und priorisiere die Testfälle nach ihrem Gesamtrisiko

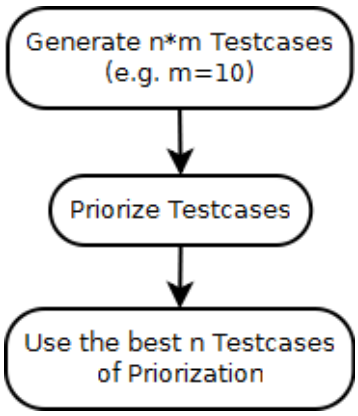

Abbildung 4.12: Risikopriorisierung für n Testfälle

Da die Algorithmen zur Testfallerzeugung das Risiko nicht berücksichtigen, werden in der Regel auch viele Testfälle erzeugt, welche sich weniger gut für risikobasiertes Testen eignen. Deshalb werden meist viel mehr Testfälle als nötig erzeugt und dann später nur die besten Teställe genutzt. Würden beispielsweise 100 Testfälle benötigt werden, könnte man zuerst 500 Testfälle generieren und diese dann nach Risiken priorisieren. Danach würde man nur die ersten 100 Testfälle auch wirklich nutzen und den Rest entfernen (dabei erhält man im allgemeinen ein besseres Ergebnis, je mehr Testfälle generiert werden). Bild 4.12 zeigt einen üblichen Workflow bei Risikopriorisierung, m repräsentiert dabei den Faktor, um wieviel mehr Testfälle generiert werden als man schlussendlich benötigt.

Im einfachsten Fall werden Testfälle bezüglich ihres Gesamtrisikos bewertet, indem man die Summe der Risiken aller Aktivitäten, welche während des Testfalles durchgeführt werden, berechnet. Grundsätzlich kann man dabei zwischen 2 Arten von Priorisierung unterscheiden:

- Statische (bzw. totale) Priorisierung, bei der die Testfälle eine feste Bewertung erhalten (unabhängig von anderen Testfällen)
- Dynamische Priorisierung, bei der die Bewertung der Testfälle davon abhängt, welche Risiken schon in zuvor gereihten Testfällen vorkommen

Im folgenden werden nun diese beiden Priorisierungsmethoden, jeweils anhand eines konkreten Algorithmus, erklärt. Die verwendeten Algorithmen hierfür stammen aus [HAK08] und [YR03], in [Hem] beispielsweise wird das Thema der Testfallpriorisierung etwas allgemeiner behandelt.

Statische Priorisierung: Total Risk Score Priorization

Bei statischer (bzw. totaler) Priorisierung handelt es sich um ein eher einfaches Verfahren für risikobasierte Testfallpriorisierung. Es werden im ersten Schritt (unabhängig voneinander) alle vorhandenen Testfälle danach bewertet, wie gut sie das SUT bezüglich dessen Risiken testen. Danach werden die Testfälle nach diesen Werten geordnet (beginnend mit dem besten/höchsten Wert).

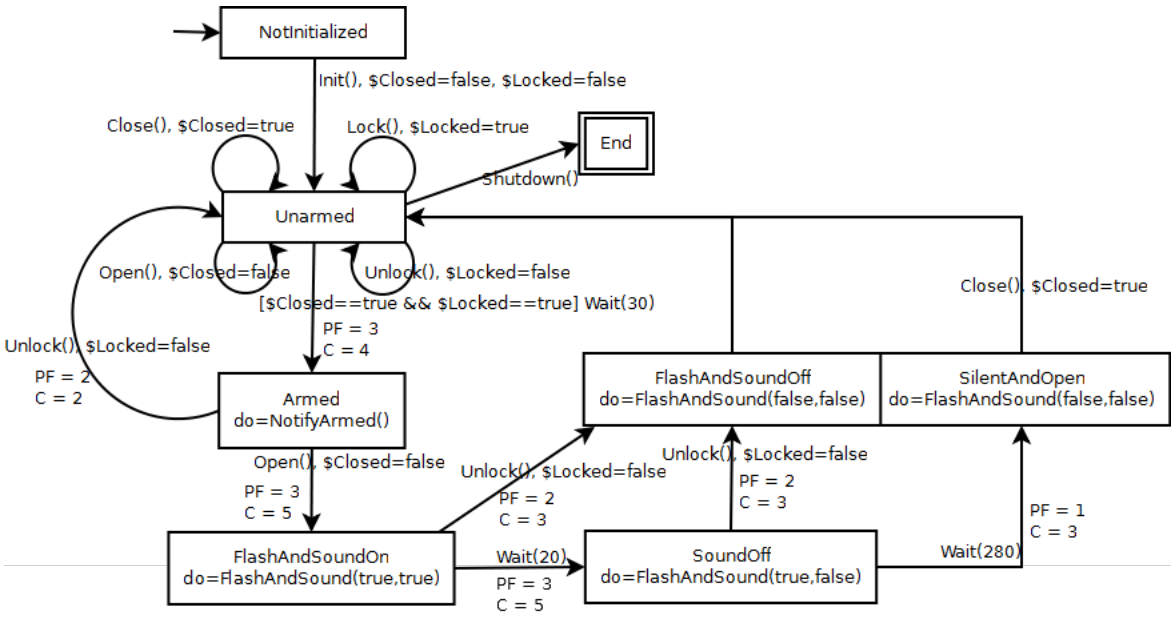

Abbildung 4.13: Car Alarm Benutzungsmodell mit Risikoannotierungen

Algorithmus 8 (Total Risk Score Priorization) *Dieser sehr einfache Algorithms aus [HAK08] geht wie folgt vor:*

1. *Erstelle mit einem beliebigen Algorithmus die Menge TCS, welche aus Testfällen $TC_i \in TCS, 0 \leq i < [TCS]$ besteht, wobei ein Testfall TC_i aus einer Menge an Transitionen T_i besteht*
2. *Berechne für jeden Testfall $TC_i \in TCS$ mit den Transitionen T_i eine Risk Exposure RE wie folgt:*
   - *Berechne für jede Transition $t \in T_i$ das Risiko ($R(t)$ mithilfe der vewendeten Risikoformel, z.B. $R(t) = PF(t) \cdot C(t)$ (PF... Probability of Failure, C... Cost)*
   - *Berechne die Risk Exposure des Testfalls TC_i mit $RE(TC_i) = \sum_{t \in T_i} R(t)$*
3. *Sortiere (mit einem beliebigen Sortierungsalgorithmus, z.B. Quicksort) die Menge TCS in eine geordnete Liste PTCS um, sortiert nach der Risk Exposure in absteigender Reihenfolge*

Beispiel 15 *Als Beispiel dient wieder das Car Alarm Benutzungsmodell, Bild 4.13 zeigt das Beispiel mit annotierten Risikofaktoren (PF...Probability of Fault, C...Cost), für Transitionen ohne angegebene Risikofaktoren gilt das Minimum (=1). Das Risiko einer Transition wird dabei mit der Formel*

$$R(t_{i,j}) = PF(t_{i,j}) \cdot C(t_{i,j}$$

berechnet. Tabelle 4.3 zeigt stichprobenartig gewählte Testfälle, welche z.B. mit einem Random Walk generiert werden könnten. Die Testfälle werden dabei durch ihre Trigger beschrieben (I...Init(), O...Open(), C...Close(), L...Lock(), U...Unlock, S...Shutdown(), W...Wait()).

Testfall	Testpfad	Risk Exposure
TC_0	{I,C,O,L,S}	5
TC_1	{I,C,L,W,U,S}	20
TC_2	{I,C,L,W,O,W,U,S}	52
TC_3	{I,C,L,W,O,W,W,C,W,U,S}	66
TC_4	{I,C,O,C,L,U,O,L,C,W,U,L,S}	27

Tabelle 4.3: Statische risikobasierte Priorisierung

Nach der statischen Priorisierung erhält man demnach folgende geordnete Menge:

$$TCS = \{TC_3, TC_2, TC_4, TC_1, TC_0\}$$

Statische Priorisierungen sind zwar sehr einfach zu realisieren, haben aber das Problem, dass man nur eine Liste an Testfällen zurückbekommt, welche nach dem totalen Risiko der Testfälle geordnet wurden. Das Problem ist dabei, dass diese Bewertung keine Rücksicht darauf nimmt, wie oft eine risikobehaftete Aktivität schon in den bisher gereihten Testfällen vorgekommen ist. Dadurch kann es z.B. vorkommen, dass nur Aktivitäten mit sehr hohem Risiko (immer wieder) und Aktivitäten mit geringerem Risiko unter Umständen nie getestet werden (obwohl diese Aktivitäten auch getestet werden sollten, nur dementsprechend später bzw. weniger oft).

Dynamische Priorisierung: Additional Risk Score Prioritization

Um zu verhindern, dass eine Risikopriorisierung zu sehr die Transitionen mit dem höchsten Risiko fokusiert, gibt es auch dynamische Priorisierungsverfahren. Bei diesen Verfahren wird die Risikobewertung der Testfälle davon abhängig gemacht, welche Testfälle schon in die *Priorisierungsliste* aufgenommen wurden. Dabei wird das Risiko einer Transition weniger stark gewichtet, je öfters die Transition schon in den vorher priorisierten Testfällen vorgekommen ist.

Algorithmus 9 (Additional Risk Score Prioritization (ARSP)) *Die Grundsätzliche Vorgehensweise des ARSP ist folgende:*

1. *Gegeben ist eine ungeorndete Liste TCS mit Testfällen und eine leere Liste PTCS*
2. *Berechne die Risk Exposure aller Testfälle in TCS (wie bei statischer Priorisierung)*
3. *Entferne den Testfall mit dem höchsten Risk Exposure aus TCS und füge ihn der Liste PTCS hinzu*
4. *Falls TCS noch nicht leer ist, gehe zurück zu Schritt 2, ansonsten ist PTCS die priorisierte Liste*

In [HAK08] wird eine einfache Variante dieser dynamischen Verfahren beschrieben, die *Additional Risk Score Prioritization*. Dabei werden schon verwendete Transitionen mit 0 gewichtet (jedes Risiko wird also nur einmal bewertet). Wenn mehrere Testfälle die gleiche Risk Exposure besitzen, wird per Zufall entschieden, welcher als nächstes der Priorisierungsliste hinzugefügt wird.

Beispiel 16 *Angewandt an die gleichen Testfälle wie bei dem vorherigen Beispiel 15, würde man folgende Resultate erhalten:*

- *Der erste Durchlauf führt zu den gleichen Risk Exposures wie bei der statischen Variante, TC_3 wird aus TCS entfernt und zu PTCS hinzugefügt*
- *Die einzigen nicht genutzten Transitionen sind nun die beiden Transitionen mit den Triggern Open() und Unlock() beim Zustand Unarmed, sowie die beiden Transitionen mit den Triggern Unlock(), welche zu Zustand FlashAndSoundOff führen. Alle anderen werden nun mit 0 bewertet.*
- *Tabelle 4.4 zeigt die Risk Exposures (RE) der einzelnen Durchläufe, die nächsten priorisierten Testfälle sind also TC_2 und TC_4.*
- *Die 2 übrigen Testfälle sind nun beide gleich stark gewichtet (beide 0), per Zufall wird z.B. TC_0 zuerst priorisiert und dann TC_1*

Testfall	Testpfad	RE (1)	RE (2)	RE (3)	RE (4)
TC_0	{I,C,O,L,S}	5	1	1	0
TC_1	{I,C,L,W,U,S}	20	0	0	0
TC_2	{I,C,L,W,O,W,U,S}	52	6	-	-
TC_3	{I,C,L,W,O,W,W,C,W,U,S}	66	-	-	-
TC_4	{I,C,O,C,L,U,O,L,C,W,U,L,S}	27	2	2	-

Tabelle 4.4: Dynamische risikobasierte Priorisierung

Die resultierende Priorisierungsliste mit diesem Verfahren ist also:

$$PTCS = \{TC_3, TC_2, TC_4, TC_0, TC_1\}$$

Dynamische Priorisierung mittels alternativen Heuristiken

Das zuvor beschriebene dynamische Verfahren zur risikobasierten Priorisierung verhindert zwar, dass risikoreichere Transitionen unter Umständen zu stark priorisiert werden, jedoch ist es immer noch etwas starr da jedes vorkommende Risiko nur einmal gewertet wird. Wenn also alle Risiken schon einmal in den Testfällen vorgekommen sind, werden die übrigen Testfälle willkürlich priorisiert, ohne Rücksicht auf das Risiko (wie man z.B. nach dem 4ten Durchlauf in Tabelle 4.4 sieht, da dort die beiden übrigen Testfälle TC_0 und TC_1 mit 0 gewertet werden, wodurch die weitere Priorisierung rein zufällig ist).

In diesem Abschnitt wird deshalb ein eigener Ansatz vorgestellt, dynamische Priorisierung mittels Heuristiken zu verfeinern. Als Heuristik dient hierfür eine Interessensformel, welche

beschreibt, wie stark das Interesse an einer Aktivität ist bzgl. ihres Risikos und der Anzahl bisheriger Durchführungen dieser Aktivität. Die Basisformel des Interesses I für eine Aktivität a ist

Formel 4

$$I(a) = R(a) \cdot \left(\frac{1}{2}\right)^{Visits(a)}$$

wobei *R(a)* das Risiko und *Visits(a)* die Anzahl bisheriger Besuche der Aktivität a beschreiben. Jedoch ist diese Formel beliebig modifizierbar, um z.B. das Interesse an einer Aktivität schneller oder langsamer sinken zu lassen.

Die daraus resultierende Testfallpriorisierung ist dann nicht mehr direkt vom Risiko abhängig, die Testfälle werden nur über das Gesamtinteresse an einem Testfall priorisiert. Bei einem ALTS werden dabei (ähnlich wie beim Berechnen der *Risk Exposure*) alle berechneten Interessen einer Liste von Transitionen T, welche in einem Testall TC vorkommen, mit

Formel 5

$$I(TC) = \sum\nolimits_{t \in T} I(t)$$

summiert, wobei sich die Anzahl der Besuche (*Visits(a)*) aus den schon priorisierten Testfällen ergibt. Ähnlich wie beim ARSP-Verfahren wird nach jedem Berechnungsdurchlauf (Berechnen des Interesses aller Testfälle) nur der Testfall mit dem höchsten Interesse zur priorisierten Liste hinzugefügt und ein neuer Berechnungsdurchlauf wird gestartet.

Die beiden Verfahren aus dem vorherigen Abschnitten sind natürlich auch schon heuristische Verfahren, welche ein optimales Testen von riskanten Aktivitäten zum Ziel hat. Beide Verfahren können auch mittels einer Interessenformel beschrieben werden. Für das TRSP-Verfahren 4.5.1 lässt man hierfür einfach die Anzahl der Besuche weg, was zur Formel

$$I(a) = R(a)$$

führt, das ARSP-Verfahren 4.5.1 kann durch die Formel

$$I(a) = R(a) \cdot 0^{Visits(a)}$$

beschrieben werden (wobei $0^0 = 1$ gilt).

4.5.2 Risikobasierte Testfallgenerierung

Bisher wurde nur auf Verfahren eingegangen, welche auf Basis von beliebigen Testfallgenerierungsmethoden eine risikobasierte Priorisierung durchgeführt haben, um risikobasierte Testfälle zu erhalten. Im Folgenden wird nun ein eigener Ansatz präsentiert, welcher die Generierung von Testfällen direkt anhand von risikobasierten Faktoren ermöglicht.

Das Verfahren der Testfallpriorisierung hat zwar den Vorteil, bestehende Testfallgenerierungsmethoden verwenden zu können, wodurch lediglich ein vergleichsweise einfacher Priorisierungsalgorithmus eingeführt werden muss, jedoch hat man dadurch auch folgende Nachteile:

- Das Ergebnis kann in der Regel nicht optimal sein (sofern nicht alle möglichen Testfälle der Priorisierung zur Verfügung stehen).
- Niedrigere Effizienz, da
  1. 2 Verfahren angewendet werden müssen (Generierung + Priorisierung)
  2. Mehr Testfälle als nötig generiert werden sollten, um ein besseres Ergebnis zu erhalten
- Testfallpriorisierung kann nur für *Offlinetesten* (Erstellen von Testfällen, welche später durch einen Testrunner ausgeführt werden) und nicht für Onlinetesten (Ausführung direkt am Modell) verwendet werden. Dies ist jedoch in Hinblick auf Regressionstests nicht relevant.

Echte Optimalität bei risikobasiertem Testen scheitert natürlich schon im Grundansatz, da schon die Risikomodellierung bzw. -abschätzung nur Heuristiken sind. Jedoch kann theoretisch aus den vorliegenden Daten ein optimales Ergebnis berechnet werden.

Grundidee: Ermitteln der interessantesten Pfade

Angenommen, man hat ein ALTS-Benutzungsmodell M mit Risikoannotierungen. Weiters beschreibt man das Interesse, eine Aktivität zu testen, mit der Formel 4

$$I(a) = R(a) \cdot \left(\frac{1}{2}\right)^{Visits(a)}$$

Hat man nun nur die Ressourcen, n Aktivitäten durchzuführen, dann ist es das Ziel, die interessanteste Menge an Testfällen zu finden, welche eine Gesamtlänge von maximal n Aktivitäten aufweist. Alternativ kann man auch anstatt der Anzahl von Aktivitäten eine Anzahl von Testfällen als Ressourcenbegrenzung definieren.

Problemstellung: The Adventurer‘s Journey

Die Problemstellung zum ermitteln der interessantesten Pfade, im folgenden *The Adventurer‘s Journey* genannt, basiert auf dem *Chinese Postman Problem*. Dabei wird als fiktive Problemstellung die Reiseplanung eines Abenteurer‘s beschrieben. Der Abenteurer möchte dabei aus einem Graphen mit möglichen Reisezielen, welche durch verschiedene Wege/Pfade verbunden sind, die für ihn interessanteste Route ermitteln, wobei er jedoch nur eine begrenzte Reisezeit zur Verfügung hat.

Beispiel 17 *Ein Abenteurer möchte eine möglichst abenteuerliche Reise (bzw. Ausflug) planen, er hat für die Reise jedoch nur eine gewisse Zeitspanne. Außerdem kann er nicht beliebig zwischen den zur Verfügung stehenden Standorten wechseln, es stehen ihm hierfür jeweils unterschiedliche Aktivitäten zur Verfügung (welche unterschiedlich riskant sind und unterschiedlich lange benötigen). Dazu hat er sich eine Karte (siehe Bild 4.14) angefertigt, in*

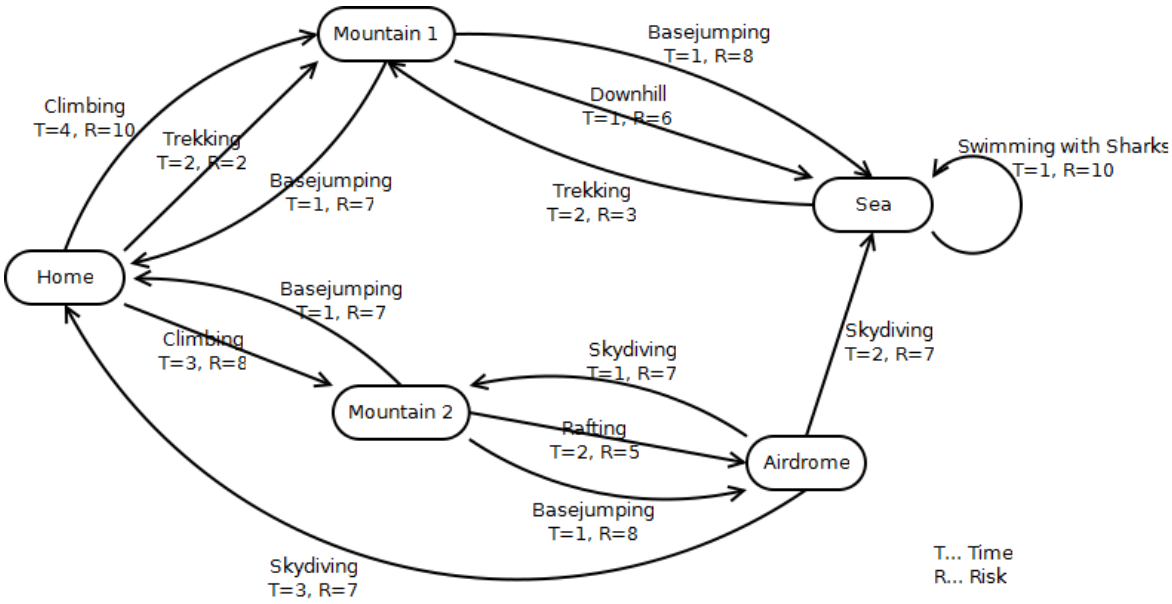

Abbildung 4.14: The Adventurer‘s Journey

welcher er an die unterschiedlichen Aktivitäten Risikofaktoren vergeben hat und wie lange er dafür benötigt. Die Karte besteht aus einem gerichteten Graphen, bei dem die Knoten die Standorte repräsentieren und die Kanten (bzw. Transitionen) die Aktivitäten zwischen den Standorten. Die Kanten besitzen zusätzlich einen Wert für die benötigte Zeit in Stunden (T) und einen Wert für das dabei entstehende Risiko (R).

Er möchte nun die interessanteste/abenteuerlichste Tour finden, welche

- *nicht länger dauert, als dass er Zeit zur Verfügung hat*
- *(optional) Zuhause (Home) beginnt und auch dort wieder endet (bei einer Rundreise)*

Dabei kann er die gleichen Aktivitäten mehrmal durchführen, jedoch wird eine Aktivität uninteressanter, je öfters er sie wiederholt. Sein Interesse, eine Aktivität a durchzuführen, beschreibt er dabei mit der Standardformel

$$I(a) = R(a) \cdot \left(\frac{1}{2}\right)^{Visits(a)}$$

(das Interesse an einer Aktivität sinkt also jeweils um die Hälfte, wenn er sie wiederholt durchführt)

Die Problemstellung unterscheidet sich also grundsäzlich vom Chinese Postman Problem durch folgende Änderungen:

- *Anstatt des kürzesten wird der interessanteste Pfad gesucht*
- *Der Problem bezieht sich auf eine bestimmte Pfadlänge (Ressourcen) und nicht auf die Abdeckung aller Wege (transition coverage)*

Würde der Abenteurer beispielsweise maximal 12 Stunden (Tagesreise) Zeit haben und möchte am Schluss seiner Tour wieder Zuhause ankommen (Rundreise), wäre die folgende Tour die Abenteuerlichste:

1. *Climbing nach Mountain 2 (Zeit = 3, Gesamtinteresse = 8)*
2. *Basejumping nach Airdrome (Zeit = 4, Gesamtinteresse = 8 + 8 = 16)*
3. *Skydiving nach Mountain 2 (Zeit = 5, Gesamtinteresse = 16 + 7 = 23)*
4. *Basejumping nach Airdrome (Zeit = 6, Gesamtinteresse = 23 + 1/2*8 = 27)*
5. *Skydiving nach Sea (Zeit = 8, Gesamtinteresse = 27 + 7 = 34)*
6. *Swimming with Sharks bei Sea (Zeit = 9, Gesamtinteresse = 34 + 10 = 44)*
7. *Trekking nach Mountain 1 (Zeit = 11, Gesamtinteresse = 44 + 3 = 47)*
8. *Basejumping nach Home (Zeit = 12, Gesamtinteresse = 47 + 7 = 54)*

Wäre dem Abenteurer der Endzustand egal, wäre folgende Tour interessanter:

1. *Climbing nach Mountain 2 (Zeit = 3, Gesamtinteresse = 8)*
2. *Basejumping nach Airdrome (Zeit = 4, Gesamtinteresse = 8 + 8 = 16)*
3. *Skydiving nach Mountain 2 (Zeit = 5, Gesamtinteresse = 16 + 7 = 23)*
4. *Basejumping nach Home (Zeit = 6, Gesamtinteresse = 23 + 7 = 30)*
5. *Climbing nach Mountain 1 (Zeit = 10, Gesamtinteresse = 30 + 10 = 40)*
6. *Basejumping nach Sea (Zeit = 11, Gesamtinteresse = 40 + 8 = 48)*
7. *Swiming with Sharks bei Sea (Zeit = 12, Gesamtinteresse = 48 + 10 = 58)*

Algorithmus 10 (Approximative Lösung des Adventurer‘s Journey) *Diese Approximation basiert auf der erweiterten Nearest-Neighbor-Heuristik und speziell auf dem Algorithmus 2. Es wird also auch hier ein Lookahead angeboten, um das Ergebnis zu verbessern. Die aktuelle Kantengewichtung wird dabei wie folgt berechnet:*

1. *Berechne für jede Kante a das Interesse I(a) mit einer Interessensheuristik (z.B.* $I(a) = R(a) \cdot \left(\frac{1}{2}\right)^{Visits(a)}$*)*
2. *Optional besitzen die Kanten auch eine benötigte Durchführungszeit T(a) (ansonsten:* $T(a) = 1$*)*
3. *Berechne die Kantengewichtung durch die Heuristik* $H(a) = \frac{T(a)}{I(a)}$

Der Algorithmus endet, wenn die maximale Durchführungszeit (bzw. Ressourcen) erreicht wurde ($\sum T(a) \geq Ressources$*).*

Diese Approximation ist grundsätzlich auf Touren mit undefiniertem Reiseziel ausgelegt (d.h. der Endzustand ist egal).

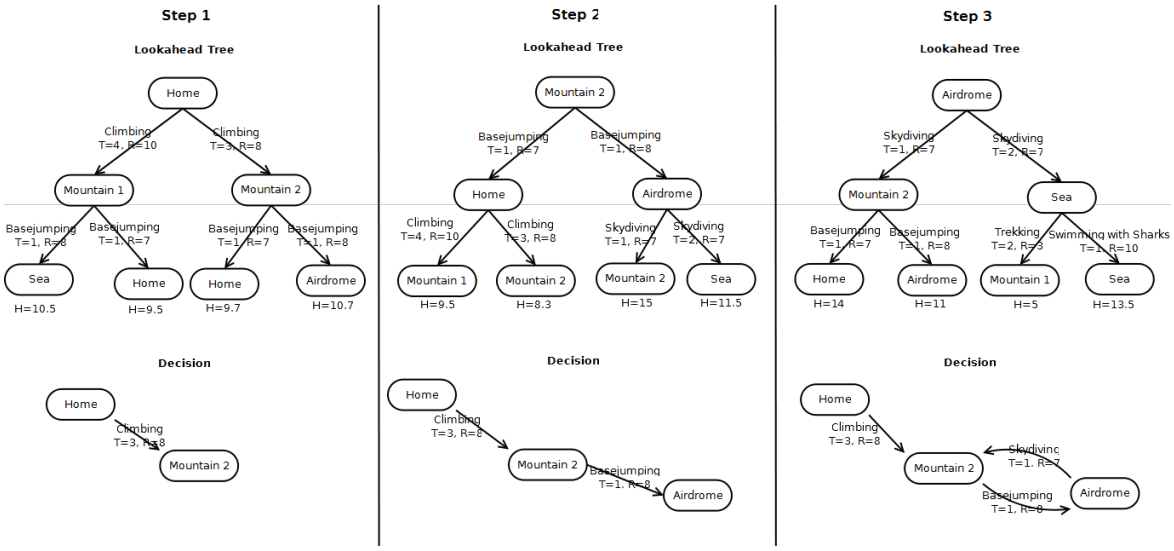

Abbildung 4.15: The Adventurer‘s Journey Approximation

Algorithmus 11 (Erreichen des korrekten Endzustandes) *Falls ein Endzustand vorgegeben wurde, kann am Ende des Algorithmus 10 noch folgendes durchgeführt werden:*

1. *Falls der letzte Zustand des berechneten Pfades P dem Endzustand entspricht, ist der Algorithmus beendet*
2. *Falls vom letzten Zustand von P ein Teilfad TP zum definierten Endzustand exisistiert und die Gesamtzeit von P und TP zusammen nicht die vorgegebene Maximalzeit überschreitet, füge diesen Teilpfad zu P hinzu und beende den Algorithmus*
3. *Ansonsten entferne die letzte Kante von P und gehe zurück zu Punkt 2*

Beispiel 18 *Bild 4.15 zeigt die ersten 3 Schritte der Approximation mit Lookahead=2. Bis dahin wird noch die optimale Lösung verfolgt, im 4ten Schritt wird zum Zustand Home zurückgekehrt, da dieser Weg bei einem Lookahead von 2 mehr Erfolg verspricht. Die fertige Approximation hat dann den Pfad*

1. *Climbing nach Mountain 2 (Zeit = 3, Gesamtinteresse = 8)*
2. *Basejumping nach Airdrome (Zeit = 4, Gesamtinteresse = 8 + 8 = 16)*
3. *Skydiving nach Mountain 2 (Zeit = 5, Gesamtinteresse = 16 + 7 = 23)*
4. *Basejumping nach Home (Zeit = 6, Gesamtinteresse = 23 + 7 = 30)*
5. *Climbing nach Mountain 1 (Zeit = 10, Gesamtinteresse = 30 + 10 = 40)*
6. *Basejumping nach Home (Zeit = 11, Gesamtinteresse = 40 + 7 = 47)*

als Lösung, wobei der Schritt 6 erst durch die Suche des Endzustandes ermittelt wird.

Ist kein Endzustand definiert (da es sich um eine Rundreise handelt), bevorzugt der Algorithmus stattdessen *Basejumping* nach *Sea* und *Swimming with Sharks* bei *Sea*. Dies ist in diesem Fall sogar die optimale Lösung. Wenn man den Lookahead mindestens so groß wählt, wie die Pfadlänge der optimalen Lösung ist, würde man für Fälle ohne bestimmten Endzustand auch immer die optimale Lösung erhalten.

Nutzen des Adventurer‘s Journey bei risikobasierter Testfallgenerierung

Ähnlich wie der Abenteurer müssen auch Softwaretester beim risikobasierten Testen vorgehen. Sind einmal die einzelnen Komponenten/Abläufe eines SUT identifiziert und mit Risikofaktoren versehen, werden Testfälle definiert, welche möglichst riskante Komponenten bevorzugen. Jedoch darf man sich dabei nicht zu sehr auf solche Komponenten fixieren, weil dann zu einseitig getestet wird. Der Tester interessiert sich also zuerst für möglichst risikoreiche Teile, je öfters er sie jedoch getestet hat, desto uninteressanter werden sie für weitere Tests. Desweiteren besteht in der Praxis meist auch ein Ressourcen-Problem, da man nur begrenzte Zeit zum Testen zur Verfügung hat (speziell auch beim Regressiontesting).

In den bisherigen Ansätzen wurden zuerst möglichst viele Testfälle mit einer beliebigen Testfallgenerierungs-Methode erstellt, welche in der Regel keine Risikofaktoren berücksichtigen. An der generierten Liste an Testfällen wird dann eine Priorisierung vorgenommen. Dabei werden Testfälle mit möglichst risikoreichen Pfaden in der Liste vorgereiht. Es sollten dabei möglichst viele Testfälle generieren werden um danach überhaupt eine gute Priorisierung vornehmen zu können.

Bei randomisierten Algorithmen stellt sich die Frage, wann abgebrochen werden kann (wann sind genug Testfälle generiert?). Anderen Methoden, wie zum Beispiel *Traveling Salesman* und *Chinese Postman*, haben hier generell das Problem, dass sie meist zu wenig Testfälle generieren (im optimalen Fall kommt jede Transition nur einmal vor, was beim Testen jedoch oft nicht gewünscht ist, da zu wenig Variation besteht).

Die Anwendung des *Adventurer‘s Journey* kann bei solchen Modellen verwendet werden, um risikoreiche Testfälle zu generieren, da er diese schon bei der Generierung berücksichtigt. Dadurch, dass die Gesamtlänge aller Pfade angegeben wird, können die optimierten Testfälle auch auf die zur Verfügung stehenden Ressourcen optimiert werden. Entscheidend für der Generierung einer möglichst guten Testsuite ist dabei vor allem auch eine gute Interessensformel, um damit die Variation der unterschiedlichen Aktivitäten zu steuern.

Um den *Adventurer‘s Journey* bei einem ALTS verwenden zu können, muss dieses als Graph mit Kantenlängen interpretiert werden, wobei im einfachen Fall für alle Transitionen als Zeitfaktor T=1 angenommen werden kann. Jedoch wäre auch eine Erweiterung des ALTS denkbar. Ebenfalls muss man auch hier eine virtuelle Kante zwischen End- und Anfangszustand einführen. Wenn die virtuelle Kante zwischen End- und Anfangszustand durchlaufen wird, ist ein Testfall beendet und ein neuer beginnt.

5 Implementierung

Im folgenden wird kurz beschrieben, wie die erarbeiteten Konzepte implementiert wurden, sodass sie für die *Test Automation Infrastructure* nutzbar sind.

Um das Benutzungsmodell als UML Statemachine beschreiben zu können, wurde das entsprechende Subset des UML Metamodells als Klassenstruktur 1:1 übernommen (siehe Bild 3.3). Die Struktur des ALTS wurde sehr einfach gehalten (Jeweils eine Klasse für Zustände und Transitionen mit entsprechenden Annotationen), die Umwandlung der Statemachine in ein ALTS wurde mit dem in 3.2.2 beschriebenen Regelwerk realisiert.

Die Klassenstruktur für UML Statemachine und das ALTS wurde direkt in den *Core* von TAI übernommen, damit diese Modelle auch für andere Plugins nutzbar sind.

5.1 Plugin für Statistisches Testen

Die in Kapitel 4 vorgestellten Algorithmen wurden als TAI-Plugin implementiert, sodass es innerhalb des vordefinierten Workflows (*Testfallgenerierung*) verwendet werden kann.

Bild 5.1 zeigt die Pluginstruktur von TAIsuite. Das Interface *IPlugable* identifiziert generell ein Plugin für TAIsuite, die abstrakte Klasse *BasePlugin* bietet Grundfunktionalitäten für Plugins. Die von *IPlugable* ableitenden Plugins definieren die Art/Rolle des Plugins:

- *IImportService*: Für den Import von Benutzungsmodellen oder fertigen Testfällen
- *IProcessService*: Für das Ableiten von Testfällen aus Benutzungsmodellen

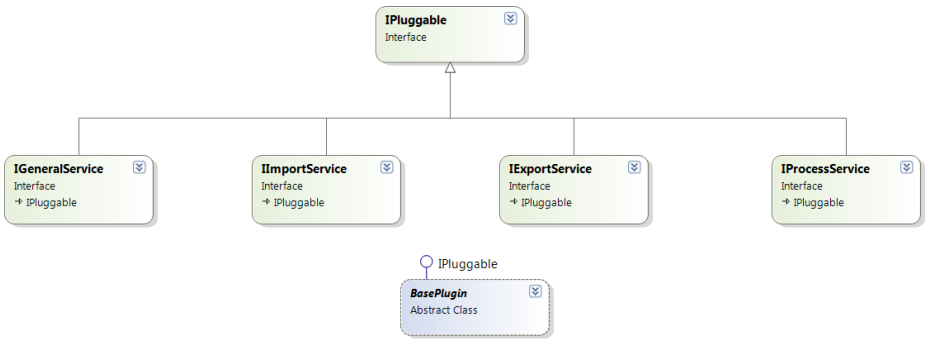

Abbildung 5.1: Pluginstruktur von TAIsuite

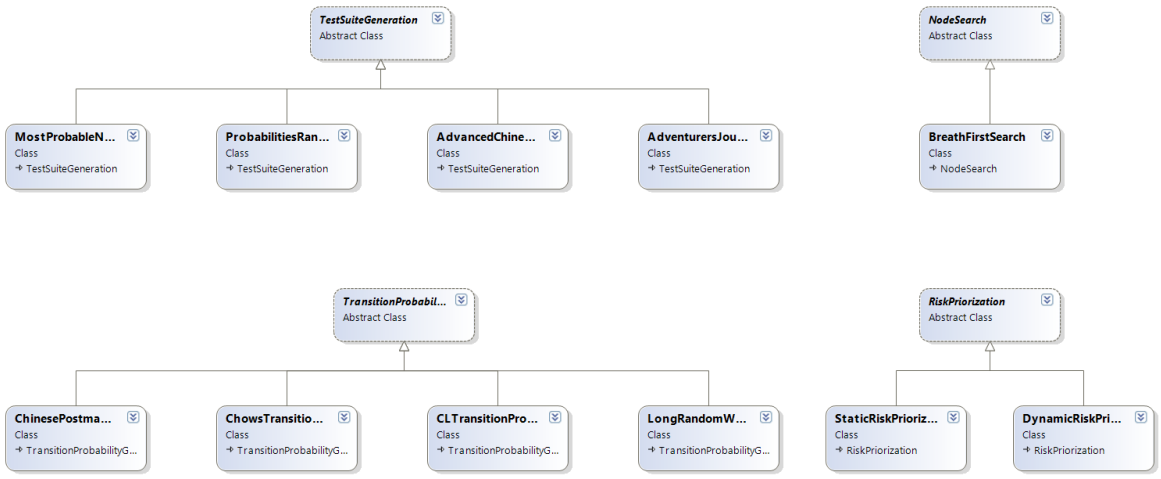

Abbildung 5.2: Klassenstruktur des Plugins für statistisches Testen

- *IExportService*: Für den Export generierter Testfälle in unterschiedliche Formate (z.B. CodedUI-Tests)
- *IGeneralService*: Für allgemeine Plugins, für welche es (noch) keine definierte Rolle gibt (z.B. ein Visualisierungsplugin für ALTS)

Das erstellte Plugin für statistisches Testen leitet sich von *IProcessService* ab, da es für die Testfallgenerierung genutzt wird. Das Plugin bietet hierfür 3 verschiedene Typen von Algorithmen an, welche hintereinander durchgeführt werden:

- Generieren von Übergangswahrscheinlichkeiten im Benutzungsmodell (optional): siehe Algorithmen 3, 4 und 5
- Generieren von Testfällen aus dem Benutzungsmodell: siehe Algorithmen 6 und 7 (wahrscheinlichkeitsbasiert), sowie Algorithmus 10 (risikobasiert) bzw. Algorithmus 2 als Chinese Postman Approximation
- Priorisieren der Testfälle (optional): siehe Algorithmen 8 und 9

Bild 5.2 zeigt die Struktur des Plugins. Es werden jeweils abstrakte Klassen angeboten, welche schon grundlegende Funktionalitäten besitzen. Die Klassen für die Testfallgenerierung müssen z.B. nur noch eine Entscheidungsmethode für eine Liste von Transitionen anbieten, die eigentliche Generierung der Testfälle (Filtern der möglichen Transitionen bezüglich *Guards*, anwenden von *Effects*, Erstellung des Pfades, ...) übernimmt die Basisklasse *TestSuiteGeneration*.

Die Klasse *NodeSearch* kann für Finden eines Pfades zu einem bestimmten Knoten (Zustand) genutzt werden (z.B. das Finden des Endzustandes für das Vervollständigen eines Testfalls).

Bild 5.3 zeigt die Ansicht des Plugins in TAIsuite. Der Nutzer kann einen oder mehrere Startzustände (links) wählen und eine Kombination der jeweiligen Algorithmen (rechts). In der Tabelle *Parameters* (unten) können die für die ausgewählten Algorithmen passenden Parameter eingestellt werden.

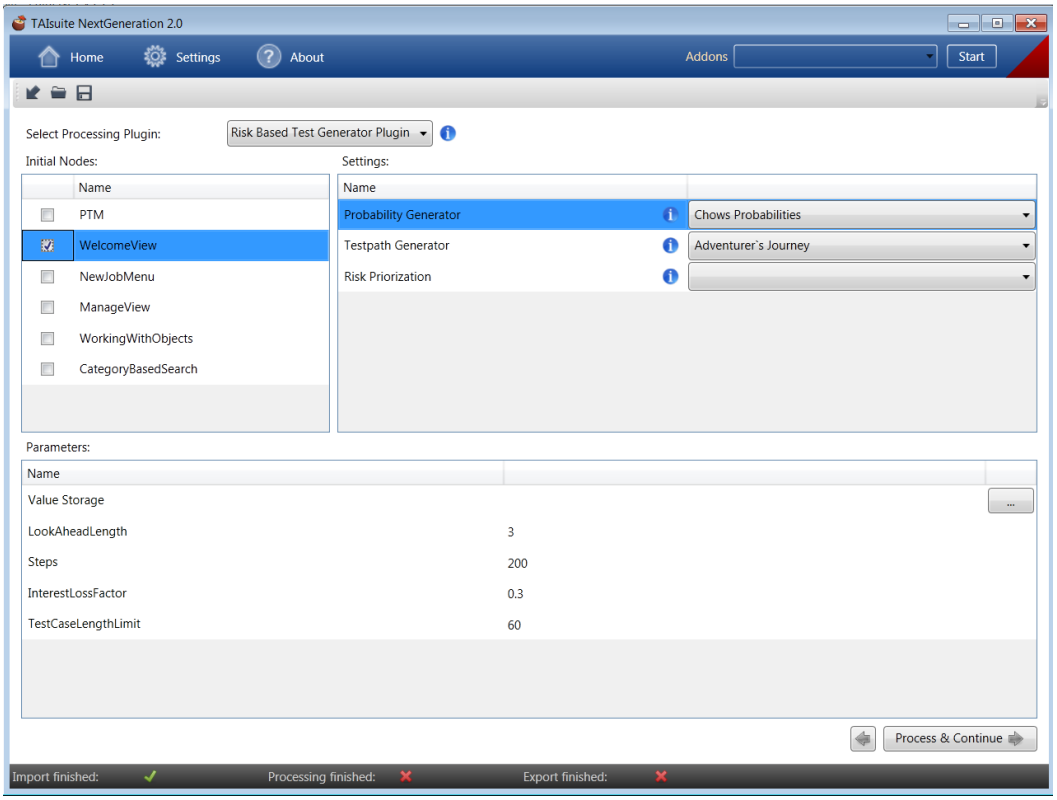

Abbildung 5.3: Ansicht des Plugins für statistisches Testen in TAIsuite

6 Evaluation

Im Folgenden werden die vorgestellten Verfahren und Algorithmen anhand von 2 durchgeführten Fallstudien bewertet.

6.1 Fallstudie 1: Einfaches Beispiel

Im der ersten Fallstudie wurden die Algorithmen an einem vergleichsweise einfachen Beispiel angewendet. Dadurch war es einfacher möglich, zu überprüfen, ob die Algorithmen grundsätzlich funktionieren.

6.1.1 SUT: Car Alarm System

Als einfaches Beispiel-SUT diente das *Car Alarm System*, welches in den vorherigen Kapiteln schon mehrmals verwendet wurde. Die Beschreibung des SUT ist in Abschnitt 2 zu finden, Bild 4.13 zeigt das verwendete Benutzungsmodell mit Risikoannotierungen.

6.1.2 Beispielimplementierung

Für das Modell wurde eine einfache Beispielimplementation mit grafischer Oberfläche programmiert (siehe Abbildung 6.1), über welche mittels Buttons die Aktionen (*Open, Lock, ...*) bzw. über Textfelder die Ausgaben (*Flash,Sound,...*) simuliert werden.

Zusätzlich wurden 4 Mutanten der Beispielimplementation erstellt. Ein Mutant ist eine Version der Implementation, in welche absichtlich ein Fehler eingebaut wurde. Die Fehler der einzelnen Mutanten sind:

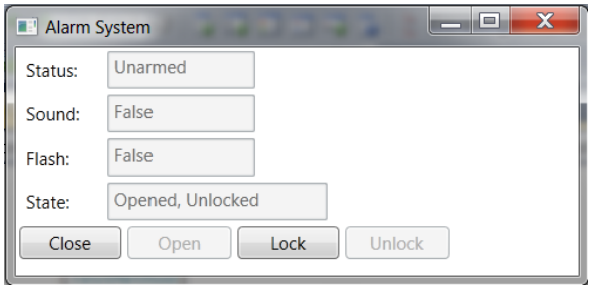

Abbildung 6.1: Einfache Beispielimplementation des Car Alarm System

- Mutant 1: Das Tonsignal (*Sound*) während des Alarms stellt sich zu spät ab.
- Mutant 2: Der Status stellt nicht automatisch auf *Unarmed* zurück, wenn der Alarm ausgelöst wurde.
- Mutant 3: Wenn der Alarm automatisch ausgeht (durch warten), wird das System entriegelt (*$Locked==false*).
- Mutant 4: Während des Alarms ist kein *Unlock* möglich.

Die generierten Testfälle sollten zuerst bei der originalen Implementation fehlerfrei durchlaufen. Danach sollten sie bei den Mutanten die eingebauten Fehler entdecken.

6.1.3 Ergebnisse

Exemplarisch werden hier nur die Ergebnisse von 6 generierten Testsuites vorgestellt. Die Testsuites wurden mit folgenden Konfigurationen erstellt:

- Testsuite 1: Implementierung des *Advanced Chinese Postman*
- Testsuite 2: Einfacher *Random* Algorithmus (ohne Übergangswahrscheinlichkeiten) mit 250 Testschritten und einer maximalen Testfalllänge von 40 Testschritten.
- Testsuite 3: Erweiterter *Random* Algorithmus mit 250 Testschritten und einer maximalen Testfalllänge von 40 Testschritten. Übergangswahrscheinlichkeiten wurden mit der *Chinese Postman* Variante erstellt.
- Testsuite 4: *Most Probable Neighboor* Algorithmus mit 125 Testschritten und einer maximalen Testfalllänge von 40 Testschritten. Übergangswahrscheinlichkeiten wurden mit der *Chinese Postman* Variante erstellt.
- Testsuite 5: *Most Probable Neighboor* Algorithmus mit 125 Testschritten und einer maximalen Testfalllänge von 40 Testschritten. Übergangswahrscheinlichkeiten wurden mit dem zyklenbasierten Algorithmus erstellt.
- Testsuite 6: *Adventurer's Journey* Algorithmus mit 125 Testschritten und einer maximalen Testfalllänge von 40 Testschritten.

Tabelle 6.1 zeigt die Anzahl der jeweils generierten Testfälle und deren Ergebnisse (wobei jeweils die Rangnummer der jeweiligen Testfälle innerhalb der Testsuite angegeben wird, welche einen Fehler gefunden haben).

Was sofort auffällt ist, dass auch bei der nichtmutierten Implementation ein Fehler gefunden wurde. Dieser Fehler wurde unbewusst durch schlampige Programmierung erzeugt, da die Wartezeiten (wie z.B. für das automatische Ausschalten des Tonsignals) mittels Threads implementiert wurden. Da jedoch keine Synchronisierung der Threads vorgenommen wurde, konnte es durch die richtige Kombination von Eingaben dazu kommen, dass das System nicht korrekt in den *Armed*-Zustand übergeht. Dies erwies sich für die Evaluierung als Glücksfall, da dadurch ein (im Vergleich zu den Mutanten) komplexerer Fehler gefunden werden konnte.

	#TC	Original	Mutant 1	Mutant 2	Mutant 3	Mutant 4
Testsuite 1	4		1	1	1	
Testsuite 2	48					
Testsuite 3	53		34	34		
Testsuite 4	7	4,5,6	1,3,4,5,6	1,3,4,5,6	1,3,4,5,6	4,5,6
Testsuite 5	8	2	2,4,5,6	2,4,5,6	2,4,5,6	2,4,5,6
Testsuite 6	7	3,4	2,3,4,5,6	2,3,4,5,6	2,3,4,5	0,3,4,5

Tabelle 6.1: Ergebnisse der ersten Fallstudie

Ebenfalls sehr auffällig ist die Tatsache, dass die beiden rein zufallsbasierten Algorithmen bei diesem Beispiel sehr schlecht abschneiden. Dies hat seine Ursache vor allem schon in der Struktur des Benutzungsmodells, da schon im Zustand *Unarmed* der Endzustand viel wahrscheinlicher erreicht wird als der Zustand *Armed*, zu welchem gewechselt werden muss, damit man überhaput die Chance hat, die Fehler zu finden.

Abbildung 6.2 verdeutlicht dies. Da nach der Initialisierung nur die Trigger *Lock()*, *Close()* und *Shutdown()* möglich sind (weil *$Locked==false* und *$Closed==false*), ist die Wahrscheinlichkeit eines sofortigen *Shutdown()* bei 14%. Damit überhaupt ein Übergang in den Zustand *Armed* möglich wird, sind mindestens 2 Inputs nötig (im optimalen Fall wird *Lock()* und *Close()* direkt hintereinander gewählt, was eine Wahrscheinlichkeit von $0.42 \cdot 0.42 \approx 0.18 \Rightarrow 18\%$ hat) und nach jedem dieser Inputs ist wieder ein *Shutdown()* möglich. Dadurch kommt es also in vielen Fällen vor, dass der Testfall endet, bevor der Übergang in den Zustand *Armed* überhaupt möglich ist.

Man sieht bei diesem Beispiel die Vorteile des Adventurer‘s Journey und des Most Probable Neighboor, da beide ihre Entscheidung unter anderem dadurch treffen, wie oft ein Übergang schon gewählt wurde (ein *Shutdown()* wird also jedes mal unwahrscheinlicher, je öfters dieser gewählt wurde).

Für eine sinnvolle Bewertung der risikobasierten Priorisierung war dieses Beispiel eher weniger geeignet, da es hierfür zu klein ist. Außerdem sind die Fehler der Mutanten nur bedingt auf das Risiko bezogen. Einzig erwähnenswert ist vielleicht die Tatsache, dass speziell sehr kurze Testfälle, welche nie in den Zustand *Armed* wechseln, bei der Priorisierung nach hinten gereiht wurden, wodurch die Fehler etwas früher gefunden wurden.

6.2 Fallstudie 2: Praxisbeispiel

Die erste Fallstudie aus dem vorherigen Abschnitt wurde speziell zum Testen der richtigen Funktionsweise der implementierten Algorithmen erstellt. Aufgrund dessen recht kleinen Modells ist es mit dieser Fallstudie jedoch nur schwer möglich, die Algorithmen bezüglich ihrer Eignung für statistisches (insbesondere risikobasiertes) Testen zu bewerten.

Als zweite Fallstudie sollte deshalb ein komplexeres Modell erstellt werden, idealerweise mit einem SUT aus der Praxis. Verwendet wurde hierfür die neueste Version des *Primary Test*

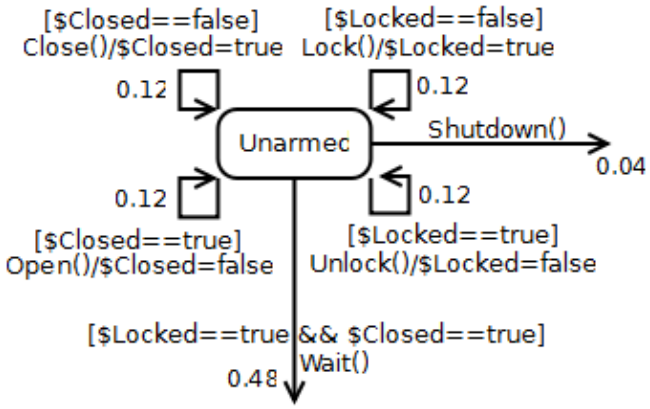

Abbildung 6.2: Übergangswahrscheinlichkeiten bei Zustand *Unarmed*

Manager (Stand: Oktober 2011) der Firma *Omicron electronics*, wobei das Modell schon in einem recht frühen Stadium der Software (April 2011) erstellt wurde und im weiteren Verlauf im Normalfall nur die Adaptoren (technische Realisierung der abstrakten Aktivitäten) neu angepasst wurden. Minimale Änderungen im Modell waren aufgrund der Entfernung eines Features notwendig.

6.2.1 SUT: Primary Test Manager

Im folgenden wird nun zuerst grob das SUT selbst erklärt und dann wird auf die getesteten Bereiche (bzw. Benutzungsszenarien) eingegangen.

Der *Primary Test Manager* (im folgenden kurz PTM genannt) ist eine von *Omicron electronics* erstellte Software, welche in Kombination mit dem ebenfalls von *Omicron electronics* entwickelten Prüfgerätes CPC 100 (mehr Informationen hierzu siehe [CPC]) verschiedenen Prüfungen ermöglicht, um den Zustand von Transformatoren, Durchführungen und Stufenschaltern zu diagnostizieren.

Der Benutzer kann damit Prüfobjekte (digitale Repräsentationen des Prüflings, z.B. Transformator) erstellen und verwalten, hierfür Prüf- bzw. Testpläne (sogenannte Jobs) definieren und diese dann später (vor Ort) automatisiert durchführen. Abbildung 6.3 zeigt den generellen Workflow zur Benutzung der Software.

Für die Fallstudie wurden 3 Benutzungsszenarien modelliert. Die ersten beiden (*Erstellen von Jobs* bzw. *Suchen und bearbeiten von Jobs*) beschreiben (grob) die grundlegende Benutzung des SUT. Das dritte (*Konfigurieren von Transformatoren*) ist eine Verfeinerung eines Teiles des ersten Benutzungsszenarios, womit die Wartbarkeit bzw. Updatefähigkeit des Modells gezeigt wird (siehe dazu Abschnitt 7).

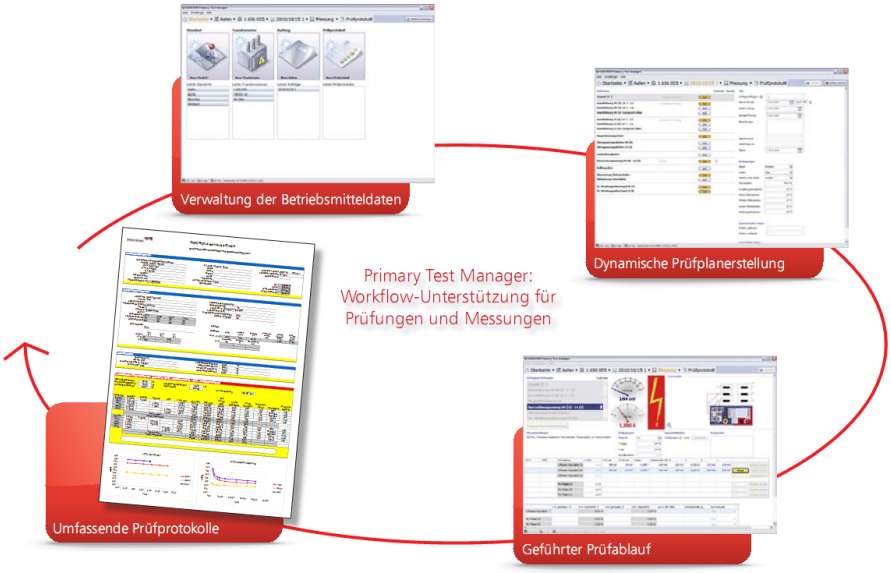

Abbildung 6.3: Workflow des Primary Test Manager (Quelle: Omicron electronics GmbH [Omi])

Erstellen von Jobs

Ein Job dient zur Verwaltung ein Tests, welcher einem *Asset* (z.B. ein elektrischer Transformator) und dessen Standort (*Location*) zugeordnet werden kann. Bei der Erstellung des *Jobs* können die entsprechenden *Assets* bzw. *Locations* entweder neu erstellt werden oder bestehende aus der Datenbank geladen werden. Bild 6.4 zeigt die Basisansichten der GUI zur Erstellung eines Jobs.

Suchen und Bearbeiten

Die Objekte der durchgeführten Tests (Locations, Assets, Job und Report) können über eine Suchmaske gesucht und anschließend bearbeitet werden. Bild 6.5 zeigt die Suchmaske (im Modus *Expanded*), das Bearbeiten der Objekte funktioniert gleich wie das Erstellen (Teilansichten siehe Bild 6.4).

Konfigurieren von Transformatoren

Durch die Konfiguration eines Transformators kann dessen genaue technische Spezifikation definiert werden, wodurch sich dann die Auswahl der durchführbaren Tests ergibt. Dazu gehört unter anderem die Frequenz, die Anzahl der Phasen und die Konfiguration der Vektorgruppen. Bild 6.6 zeigt ein Beispiel für eine Konfiguration von Vektorgruppen.

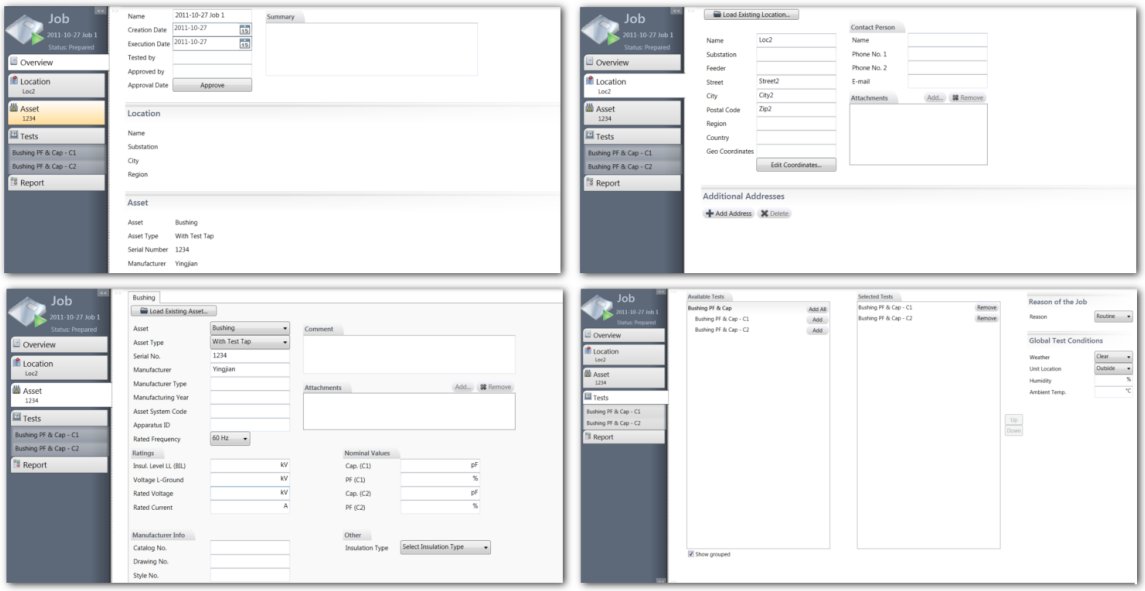

Abbildung 6.4: Teilansichten bei der Erstellung eines Jobs

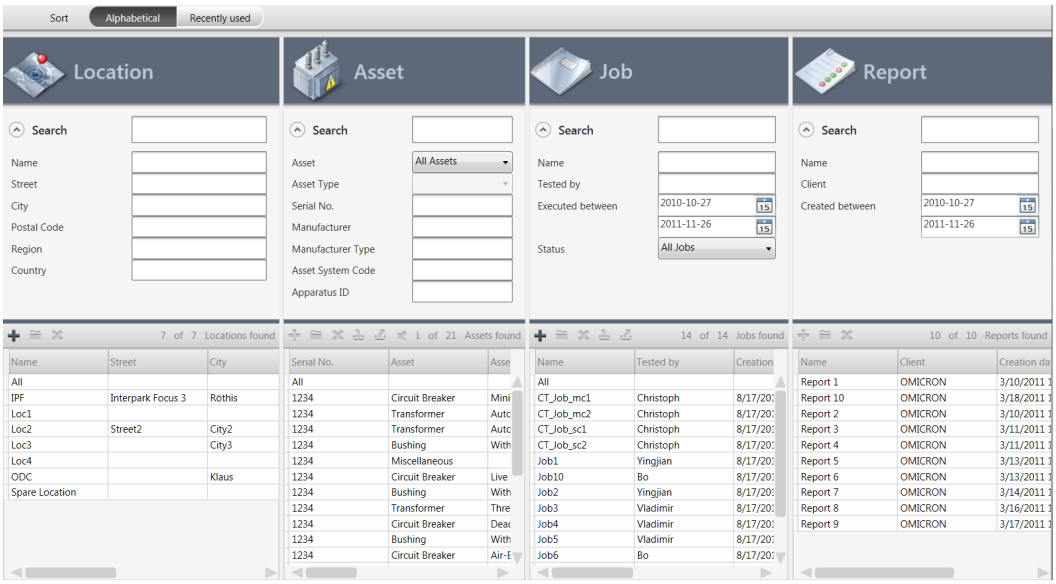

Abbildung 6.5: Ansicht zum Suchen von Objekten

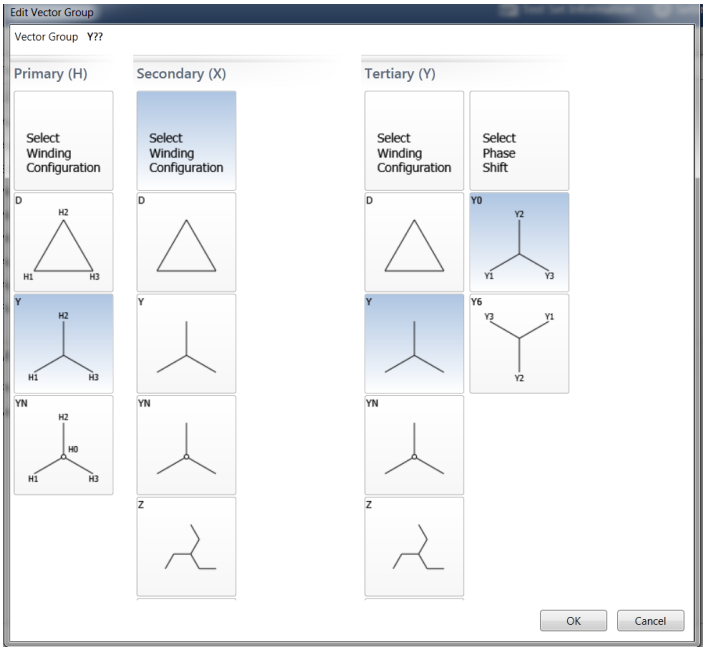

Abbildung 6.6: Konfiguration von Vektorgruppen

6.2.2 Erstellung des Modells und des Adapters

Bei der Erstellung des Benutzungsmodells wurden die in 7 beschriebenen Modellierungsmuster angewandt. Es wurde also jede der vorhin beschriebenen Ansichten der GUI als eigenes Zustandsdiagramm modelliert. Einzelne Teilansichten bzw. unterschiedliche Zustände der Ansichten (z.B. für einfache Suche, Detailsuche, ...) wurden innerhalb dieser Diagramme definiert.

Beispiel 19 *Bild 6.7 zeigt das Zustandsdiagramm für die Benutzung der Objektsuche (siehe Bild 6.5).*

Da für das SUT schon in einem sehr frühen Stadium ein Konzept für die GUI erstellt wurde, konnte das Benutzungsmodell schon relativ früh erstellt werden, ohne das die fertige Oberfläche vorhanden war. Lediglich der Adapter, welche die technische Realisierung der Aktivitäten (z.B. Klicken eines Buttons) beinhaltet, musste nachträglich (von Version zu Version) neu angepasst werden.

Die Vorgehensweise zur Modellierung des Benutzungsmodells und zur manuellen Abschätzung der Risiken wird in den Anhangskapiteln 7 und A.2 beschrieben.

6.2.3 Vorgehensweise bei der Bewertung der Testfälle

Risikobasiertes Testen hat zum Ziel, die Bereiche eines SUT genauer zu testen, von welchen ein höheres Risiko ausgeht. Auf der anderen Seite heißt das auch, man vernachlässigt eher

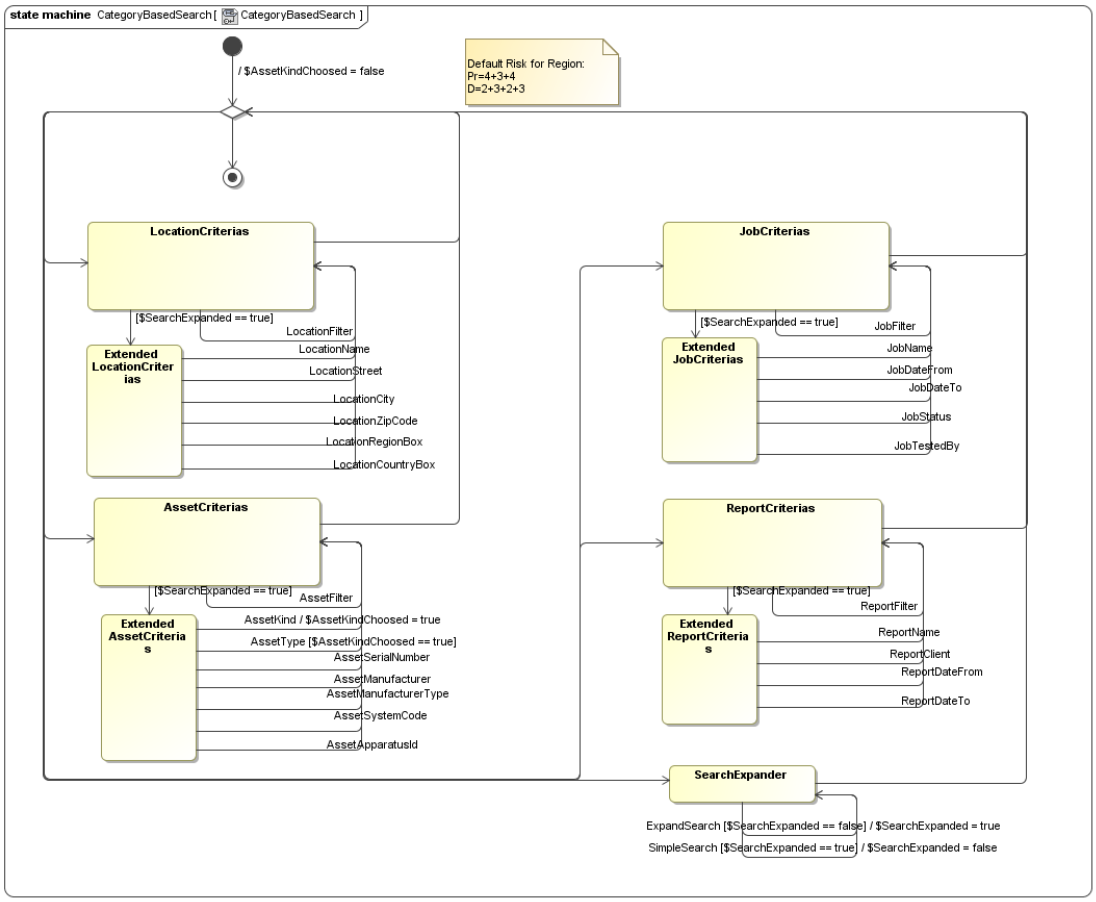

Abbildung 6.7: Benutzungsmodell für die Objektsuche

unwichtigere (bzw. weniger risikobehaftete) Bereiche und akzeptiert, dass dort weniger Fehler gefunden werden (als durch gleichmäßiges Testen des SUT).

Wenn man nun ein Testverfahren bezüglich risikobasierten Testen bewerten möchte, stellt sich hier schonmal die Frage, wie dies zu geschehen hat. Die Anzahl gefundener Fehler ist nicht wirklich ausschlaggebend, die gefunden Fehler müssten zumindest bezüglich ihres Risikos bewertet werden. Weiters erhält man nur bei genügend gefundenen Fehlern ein stichhaltiges Ergebnis. Hierfür ist das für diese Fallstudie verwendete Modell zu klein und zu einfach gehalten, ein größeres würde jedoch den Rahmen dieser Masterarbeit sprengen.

Denkbar wäre natürlich auch hier der gezielte Einbau von Mutanten, was jedoch beim vewendeten SUT auch nur sehr schwer möglich gewesen wäre.

Deshalb wurde eine eher vereinfachte Vorgehensweise für die Fallstudie gewählt. Dabei wurden die generierten Testfälle eines jeweiligen Algorithmus (im folgenden *Testsuite* genannt) nach 3 Kriterien bewertet:

1. Alle generierten Testfälle müssen am SUT automatisch mit einem Testrunner durchführbar sein, damit die Verfahren auch für Automatisierung als geeignet gelten. Ein Testfall gilt als durchführbar, wenn alle Aktivitäten wie erwartet durchgeführt wurden oder ein Fehler entdeckt wurde.
2. Für die Evaluierung wahrscheinlichkeitsbasierter Algorithmen werden die Testfälle mit gleichverteilten Übergangswahrscheinlichkeiten generiert. Dafür werden zuerst die Übergangswahrscheinlichkeiten mit den in 4.2.2 vorgestellten Algorithmen generiert. Das Ergebnis gilt dann als erfolgreich, wenn die Aktivitäten möglichst gleichverteilt sind (d.h. möglichst gleich oft in den Testfällen vorkommen). Dabei wird hier nicht nur der wahrscheinlichkeitsbasierte Algorithmus bewertet, sondern auch der verwendete Algorithmus zum Generieren der gleichverteilten Übergangswahrscheinlichkeiten.
3. Alle mit risikobasierten Algorithmen generierten Testfälle werden miteinander verglichen, indem die Fokusierung der einzelnen Aktivitäten (je früher und öfter eine Aktivität vorkommt desto höher die Fokusierung) bezüglich ihres definierten Risikos verglichen werden.

Berechnen der Fokusierung einzelner Aktivitäten

Wie stark eine einzelne Aktivität (bzw. eine Komponente) bei den Tests fokusiert wird, hängt in erster Linie davon ab, wie oft diese durchgeführt wird. Desweiteren kann auch entscheidend sein, wie früh (d.h. ob in priorisierten Testfällen) eine Aktivität ausgeführt wird, z.B. wenn aus Mangel an Ressourcen nicht immer alle Testfälle durchgeführt werden können und in manchen Fällen nur priorisierte Testfälle durchgeführt werden.

Deshalb wurden 2 Formeln definiert, um die Fokusierung von Aktivitäten zu bewerten. Für beide Formeln bezeichnet $Count(a, t)$ die Anzahl, wie oft die Aktivität a in Testfall t vorkommt. $Contained(a, t)$ hat den Wert 1, wenn die Aktivität a in Testfall t vorkommt, ansonsten den Wert 0.

Die erste Formel für eine Testsuite $T = t_1, ..., t_n$ lautet

Formel 6

$$Focus1(a) = \sum_{i=1..n} Count(a, t_i)$$

und berechnet die Anzahl des Auftretens der Aktivität in einer berechneten Testsuite. Auch für die Überprüfung der risikobasierten Priorisierung hat diese Formel Sinn, sofern nach der Priorisierung nur die ersten Testfälle verwendet werden.

Die zweite Formel für eine geordnete Testsuite $T = t_1, ..., t_n$ lautet

Formel 7

$$Focus2(a) = \sum_{i=1..n} Count(a, t_i) * 0.8^i$$

und berücksichtigt auch die Reihung der Testfälle, in denen sie vorkommt. Der Faktor *0.8* verringert den berechneten Wert je nach Position des Testfalls. Dadurch kann die Fokusierung von Aktivitäten stärker bewertet werden (Aktivitäten, welche in priorisierten Testfällen vorkommen, werden stärker gewichtet).

Vergleich der berechneten Werten

Die Fokusierung der einzelnen Aktivitäten des Benutzungsmodells direkt innerhalb eines Verfahrens zu vergleichen würde eher wenig Sinn haben. Dies ist hauptsächlich deshalb so, weil die meisten Aktivitäten sehr stark von anderen Aktivitäten abhängig sind. Kann z.B. eine Aktivität B nur durchgeführt werden, wenn zuvor die Aktivität A durchgeführt wurde, dann wird die Aktion A mindestens so oft durchgeführt werden, wie Aktivität B (auch wenn Aktivität B ein hohes Risiko hat und Aktivität A nur ein geringes).

Deshalb mussten die einzelnen Testsuites (welche mit unterschiedlichen Verfahren/Algorithmen generiert wurden) miteinander verglichen werden. Dabei wurden die Fokusierung der Aktivitäten in den Testsuites verglichen und zwar relativ zu deren definierten Risikofaktoren. Als Vergleichsbasis waren dabei speziell 2 Testsuites geeignet:

1. Eine Testsuite, welche mit einem rein zufallsbasiertem Algorithmus generiert wurde (für die Bewertung von der wahrscheinlichkeitsbasierten Algorithmen).
2. Eine unpriorisierte Testsuite, welche mit dem *Most Probable Neighbor* Algorithmus erstellt wurde, da dieser am ein möglichst gleichmäßiges Durchführen aller Aktivitäten ermöglicht (und keine Risikofaktoren berücksichtigt).

Eine weitere gute Vergleichsreferenz für die Gleichverteilung war die Testsuite, welche mittels der erweiterten Chinese Postman Approximation generiert wurde, da dadurch recht gut erkennbar war, welche Aktivitäten zwangweise (wegen der Modellstruktur) bevorzugt werden.

6.2.4 Ergebnisse und Bewertung

Im diesem Abschnitt werden nun die Ergebnisse der 3 Kriterien, nach denen sie bewertet wurden, präsentiert. Dabei wird jedoch zur besseren Übersicht nur ein Teil aller bewerteten Testsuites näher erläutert.

Automatische Durchführbarkeit der Testfälle

Für das erstellte Modell wurde ein Adapter für den Zugriff auf mit Ranorex erstellten Funktionen erstellt. Die generierten Testfälle wurden mittels TAIsuite in Unit-Tests konvertiert. Diese Unit-Tests konnten daraufhin alle fehlerfrei durchgeführt werden, womit das Benutzungsmodell als automatisierbar bezeichnet werden kann.

Vergleich der Übergangswahrscheinlichkeiten für Gleichverteilung

Abbildung 6.8 zeigt, wie oft die einzelnen Transitionen (bzw. deren Aktivitäten) im Vergleich zu den anderen vorkommen (in Prozent), wenn deren Übergangswahrscheinlichkeiten als Basis bei der Generierung dienen. Für die Ermittlung der Werte wurde Formel 6 verwendet. Hierfür wurde der wahrscheinlichkeitsbasierte Zufallsalgorithmus mit einer sehr hohen Anzahl an Schritten (100000) durchgeführt und das vorkommen der einzelnen Transitionen (innerhalb der generierten Pfade) miteinander verglichen.

Die rote Linie (**Universal Distribution**) stellt die (nicht existierende) perfekte Gleichverteilung dar und dient als Referenz. Die anderen Linien zeigen jeweils das Vorkommen der Transitionen bei

- gleichen Übergangswahrscheinlichkeiten (**Standard**), d.h. dass alle Transitionen jeweils gleich wahrscheinlich sind
- Übergangswahrscheinlichkeiten, welche mit dem Chinese Postman Verfahren 4.2.2 erstellt wurden (**Chinese Postman Probabilities**)
- Übergangswahrscheinlichkeiten, welche mit dem Verfahren nach Chow 4.2.2 erstellt wurden (**Chow‘s Probabilities**)
- Übergangswahrscheinlichkeiten, welche mit dem zyklenbasierten Verfahren 4.2.2 erstellt wurden (**Cycle Optimized Probabilities**)

Wie in Abbildung 6.8 unschwer zu erkennen ist, waren für dieses Beispiel nur die mit dem Chinese Postman Verfahren erstellten Übergangswahrscheinlichkeiten wirklich nutzbar. Speziell in der Mitte des Diagramms (Transition *JobDateFrom* bis *LocationZipCode*) erkennt man, dass die anderen Verfahren zu schwache Wahrscheinlichkeitswerte hatten, damit diese Transitionen überhaupt erreicht werden konnten.

Dies liegt speziell an dem schon in 4.2.2 erwähnten Problem, dass die Verfahren für zeithomogene Graphen (d.h. Wahrscheinlichkeiten der Transitionen ändern sich nicht) erstellt wurden, jedoch ist dies bei diesem Benutzungsmodell durch die zahlreich verwendeten *Guards*

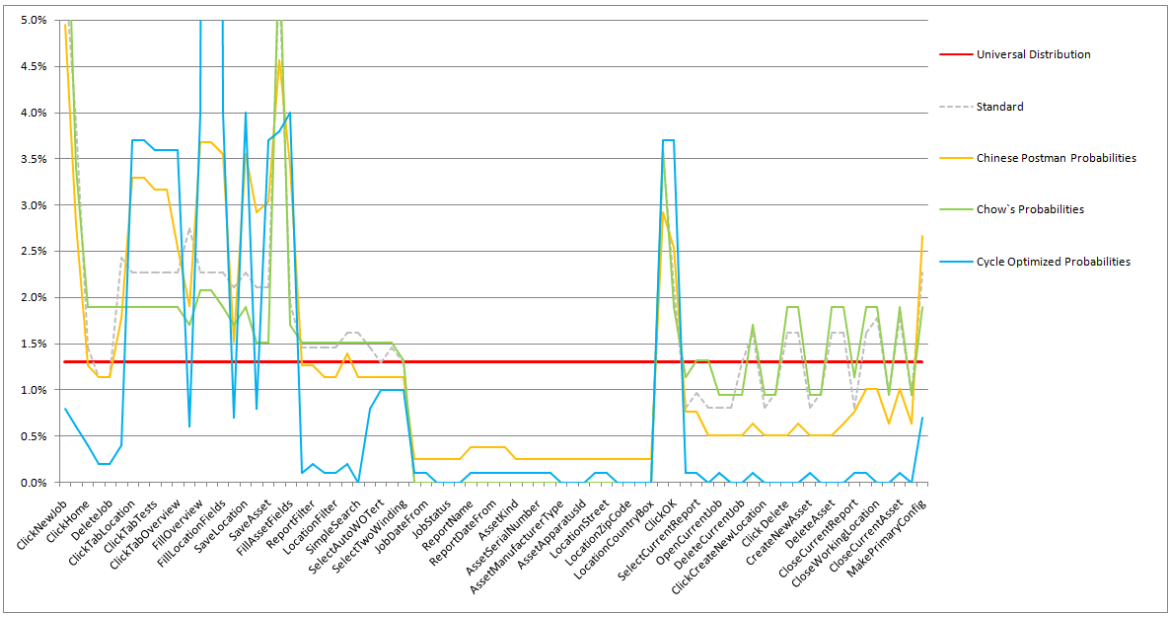

Abbildung 6.8: Erreichbarkeit der Transitionen bei Gleichverteilung

nicht der Fall. Einzig der Chinese Postman Algorithmus konnte für die Implementation dahingehend angepasst werden, dass solche Guards berücksichtigt werden. Für ein ALTS kann dieses Verfahren also als einziges wirklich eingesetzt werden (im folgenden wurde bei der Generierung auch nur dieses Verfahren angewandt.

Gleichverteilung wahrscheinlichkeitsbasierter Testsuites

Abbildung 6.9 zeigt das relative Auftreten der Transitionen (in Prozent) in den Testsuites, welche mit wahrscheinlichkeitsbasierten Algorithmen erstellt wurden. Hier wurde ebenfalls Formel 6 zur Berechnung der Werte verwendet. Als Übergangswahrscheinlichkeiten wurden die mit dem *Chinese Postman Verfahren* 4.2.2 generierten Werte verwendet, das Ziel ist also eine Gleichverteilung. Als Vergleichsreferenz dienen die Gleichverteilung selber (*universal distribution*), ein rein zufallsbasierter Algorithmus und der *Chinese Postman Algorithmus*, da mit diesem schon recht gut erkennbar ist, welche Transitionen zwangsweise öfters vorkommen müssen (es ist sozusagen die *bestmögliche erreichbare Gleichverteilung*).

Interpretation der Resultate:

- **Random**: Beim rein zufallsbasierten Algorithmus treten nur rund ein Drittel der Transitionen auf (schlechte Abdeckung) und diese nicht sehr gleichmäßig.
- **Random with Probabilities**: Beim wahrscheinlichkeitsbasierten Zufallsalgorithmus treten ein Großteil der Transitionen auf, was schon eine deutliche Verbesserung darstellt. Jedoch ist das Auftreten nur partiell gleichmäßig.

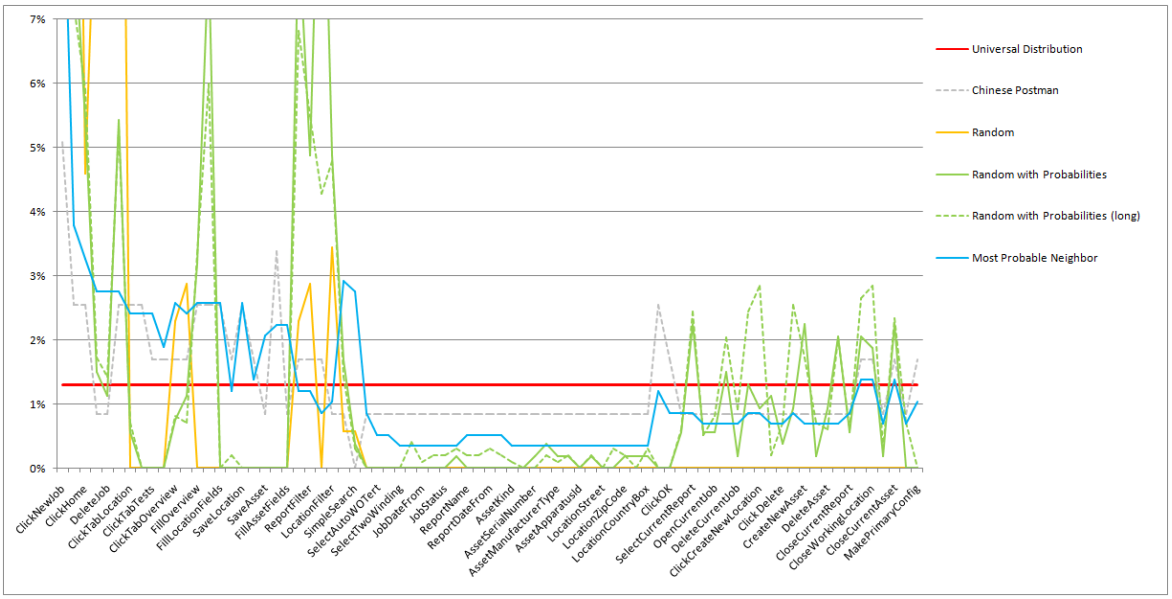

Abbildung 6.9: Gleichverteilung wahrscheinlichkeitsbasierter Testsuites

- **Random with Probabilities (long)**: Hier wurde der Algorithmus 5 mal länger durchgeführt, wodurch eine sichtbare Verbesserung erzielt wurde.
- **Most Probable Neighbor**: Mit diesem Algorithmus wurde hier das mit Abstand beste Resultat erzielt, sowohl bei der Abdeckung der Transitionen (100%) als auch bei der Gleichmäßigkeit.

  Anmerkung: In dieser Evaluation wurde nur das Auftreten einzelner Transitionen überprüft, jedoch nicht Kombinationen mehrerer Transitionen (Es wurden lediglich gleiche Testfälle herausgefiltert). Der *Most Probable Neighbor Algorithmus* könnte eine ungleichmäßige Verteilung solcher Kombinationen haben, wobei dies jedoch wegen der *Interessensheuristik* (durch welche eine Streuung dieser Kombinationen bewirkt wird) eher nicht der Fall sein sollte.

Risikofokusierung risikobasierter Testsuites

Schlussendlich wurden noch die risikobasierten Testsuites bewertet. Abbildung 6.10 zeigt eine Auswahl (wobei hier der Fokusierungsformel *focus3(a)* verwendet wurde). Zum Vergleich dient diesmal das jeweils definierte Risiko einer Transition (individuelles + regionales Risiko) und eine Testsuite (generiert mit Most Probable Neighbor) ohne Risikopriorisierung. Dabei gilt das Risiko (*Risk*) als zu erreichendes (optimales) Ziel und die nicht priorisierte Testsuite als untere Schranke (eine erfolgreiche risikobasierte Testsuite sollte der Risikoverteilung ähnlicher sein als diese nicht priorisierte Testsuite). In diesem Fall wurde die Formel 7 verwendet, um die entsprechenden Werte zu berechnen.

Interpretation der Resultate:

- **Most Probable Neighbor (Dynamic/Static Priorization)**: Dynamische und Statische Priorisierung lieferten in diesem Beispiel das exakt gleiche Resultat. Gefühlsmäßig scheint das Resultat etwas besser zu sein als das Original, jedoch nicht signifikant.
- **Adventurer's Journey (lossfactor=0.8)**: Ein sehr gutes Ergebnis, welches sich schon sehr stark an die Risikoverteilung annähert. Zu Bedenken ist hier jedoch, dass durch die vergleichsweise geringe Abschwächung der *Interessensformel* (hier $Interest(a) = Risk * lossfactor^{visits}$, der lossfactor (0..1) gibt also an, wie stark das Interesse nachlässt) auch eine kleinere Streuung unterschiedlicher Kombinationen verursacht wird.
- **Adventurer's Journey (lossfactor=0.3)**: Bis auf wenige Stellen (keine 100% Abdeckung der Transitionen) ein recht gutes, im Vergleich zum vorherigen Resultat jedoch schlechteres (durch die stärkere Abschwächung), Ergebnis. Man erkennt dadurch die Wichtigkeit der passenden *Interessensformel*.

Bei der hier verwendeten Evaluierungsmethode hat der Adventurer's Journey Algorithmus insgesamt wesentlich bessere Ergebnisse geliefert als die Algorithmen mit Testfallpriorisierung. Jedoch muss hier auch berücksichtigt werden, dass die Evaluierungsmethode sehr einfach gehalten wurde.

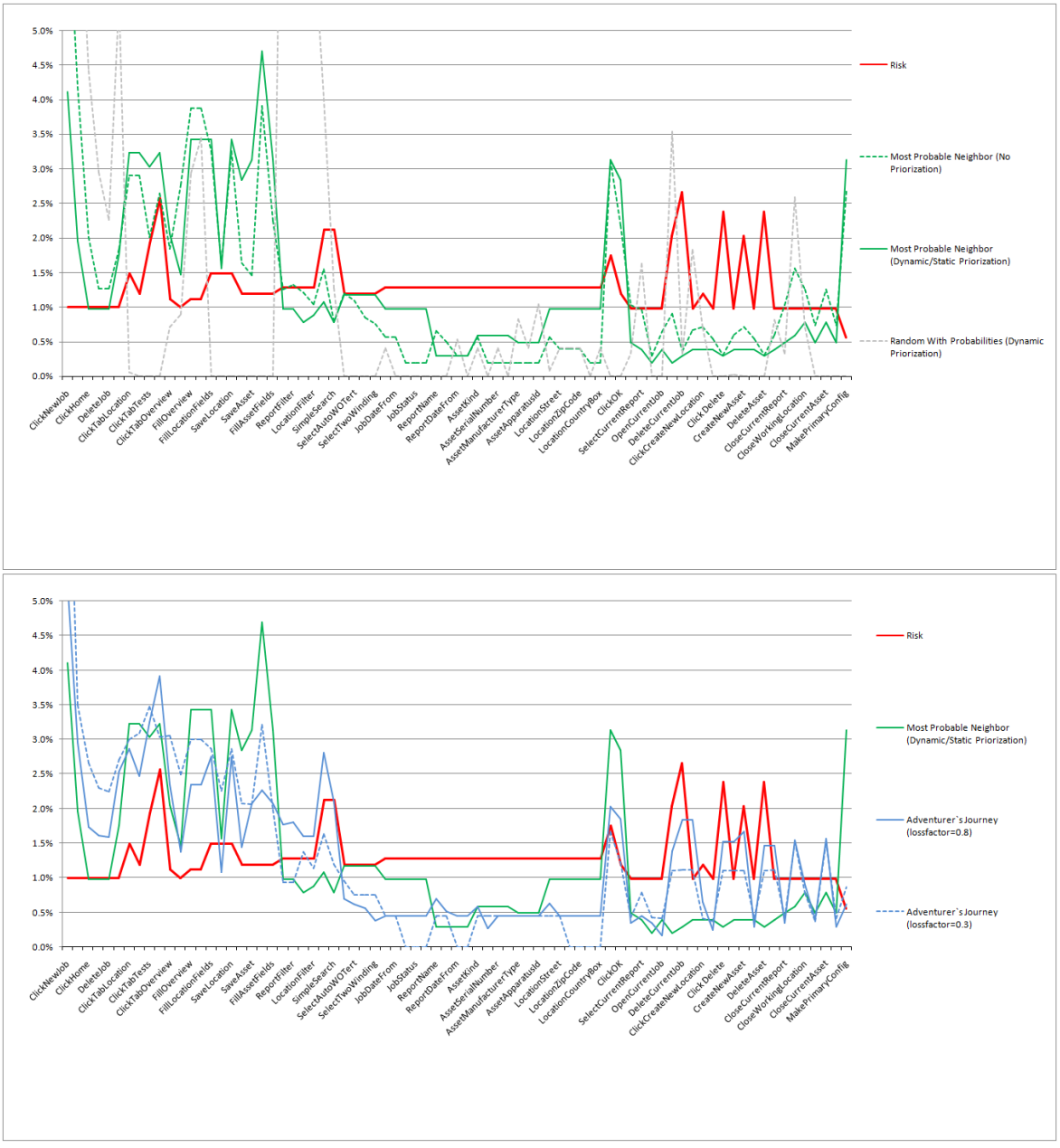

Abbildung 6.10: Risikofokusierung risikobasierter Testsuites

7 Konklusion

Dieses Kapitel stellt die Ergebnisse dieser Masterarbeit noch einmal zusammenfassend dar. Darüber hinaus wird dargelegt, inwiefern das erarbeitete Konzept noch verbessert werden kann.

Grundsätzlich wurden die Ziele der Arbeit erreicht. Es wurden zuerst unterschiedliche Methoden zur modellbasierten Testfallgenerierung recherchiert, wobei man sich schon recht früh auf statistische Methoden (siehe Kapitel 4) konzentrierte. Daraus wurde dann ein Konzept für statistische Testfallgenerierung, basierend auf dem eigenen Benutzungsmodell ALTS (siehe Kapitel 3), erstellt. Dieses Konzept wurde dann als Plugin für die *Test Automation Infrastructure* (TAI) realisiert und durch 2 Fallstudien evaluiert.

Speziell die verwendeten Algorithmen zur Generierung von *fairen* Übergangswahrscheinlichkeiten stellen ein noch recht großes Problem dar, da diese nur gute Resultate liefern, wenn das Benutzungsmodell keine *Guards* definiert hat (d.h. alle Transitionen sind immer möglich). Einzig der implementierte Algorithmus, welcher auf dem Chinese Postman Problem beruht, konnte relativ gute Ergebnisse liefern (da der verwendete Chinese Postman Algorithmus *Guards* berücksichtigte). Hier fehlen eindeutig noch Erweiterungen, welche solche *zeitinhomogenen* Graphen berücksichtigen.

Mit den wahrscheinlichkeitsbasierten Zufallsalgorithmen konnte eine signifikante Verbesserung gegenüber rein zufallsbasierten Algorithmen erreicht werden (speziell in Bezug auf Kantenabdeckung). Dadurch konnten die Vorteile von kontrollierten und zufallsbasierten Algorithmen gut kombiniert werden. Jedoch wurde nicht immer eine 100% Kantenabdeckung erreicht und auch bei sehr großen Testsuites wurde keine sehr gute Gleichverteilung erreicht. Mit der selbstentwickelten *Most-Probable-Neighboor* konnte eine bessere Gleichverteilung erreicht werden, jedoch hat dieser keinen Zufallsfaktor. Eine Erweiterung um einen zufallsbasieten Mechanismus wäre hier wünschenswert, damit dieser Algorithmus nicht zu *starr* ist.

Mit den risikobasierten Algorithmen konnten Testsuites generiert werden, welche eine bessere Fokusierung bezüglich des definierten Risiko‘s zulassen. Speziell der selbstentwickelte Algorithmus *Adventurer‘s Journey* konnte hier sehr gute Ergebnisse liefern. Jedoch war dies natürlich eine sehr theoretische Evaluierung und den Erfolg von risikobasierten Testverfahren wird man frühestens mach mehreren Jahren Einsatz bewerten können.

Analyse: Erstellung von GUI-Benutzungsmodellen mittels UML Statemachines

In diesem Abschnitt wird darauf eingegangen, wie mittels UML Statemachines ein Benutzungsmodell für die graphischen Oberfläche (GUI) des zweiten Fallbeispiels 6.2 erstellt wurde. Dafür wurden Muster für die Erstellung von Benutzungsmodellen mittels UML Statemachines konzipiert.

Bei der Erstellung der Modelle fiel die Notwendigkeit auf, spezielle Muster (ähnlich zu Design Patterns in der Softwareentwicklung) für die Erstellung von Benutzungsmodellen zu definieren, um diese Modelle für andere verständlicher und dadurch einfacher wartbar zu machen. Zwei der wichtigsten Muster (Verschachtelung und Modale Fenster) werden im folgenden vorgestellt.

Man muss jedoch dabei anmerken, dass mit diesen Mustern die Struktur der Oberfläche recht genau beschrieben wird. Bei einem Neudesign der Oberfläche könnte dies zur Notwendigkeit von aufwändigen Änderungen des Modells führen. Die fertige Struktur der Oberfläche sollte also bei der Erstellung eines solchen Benutzungsmodells bereits feststehen. Ansonsten empfiehlt es sich ein abstrakteres Modell zu erstellen, welches sich z.B. von den Requirements (dt. Anforderungen) des SUT ableitet.

A.1 Verschachtelung von Teilansichten

GUI's teilen sich meist schablonenartig in mehrere Verschiedene Teilansichten auf, welche ineinander verschachtelt werden. Bild A.1 zeigt ein Beispiel für so eine Verschachtelung. Ein beliebter Fehler bei der Beschreibung von GUI-Benutzungsmodellen ist es, die einzelnen Zustände der Oberfläche als ganzes zu unterscheiden (entweder als einfache Zustände oder in getrennte Statemachines aufgeteilt). Dadurch wird es meist nötig, die gleichen Vorgänge öfters zu beschreiben (siehe Bild A.2), wodurch das Modell unnötigerweise anwächst und dadurch sowohl unübersichtlicher als auch schlechter wartbar wird.

Ein Ansatz zur Lösung dieses Problem ist es, sich die Verschachtelung der Oberfläche als Vorbild zu nehmen und die einzelnen Teilansichten entweder als Composite States oder Submachine States (je nach Komplexität) zu verschachteln. Bild A.3 zeigt eine solche Verschachtelung mit Composite States.

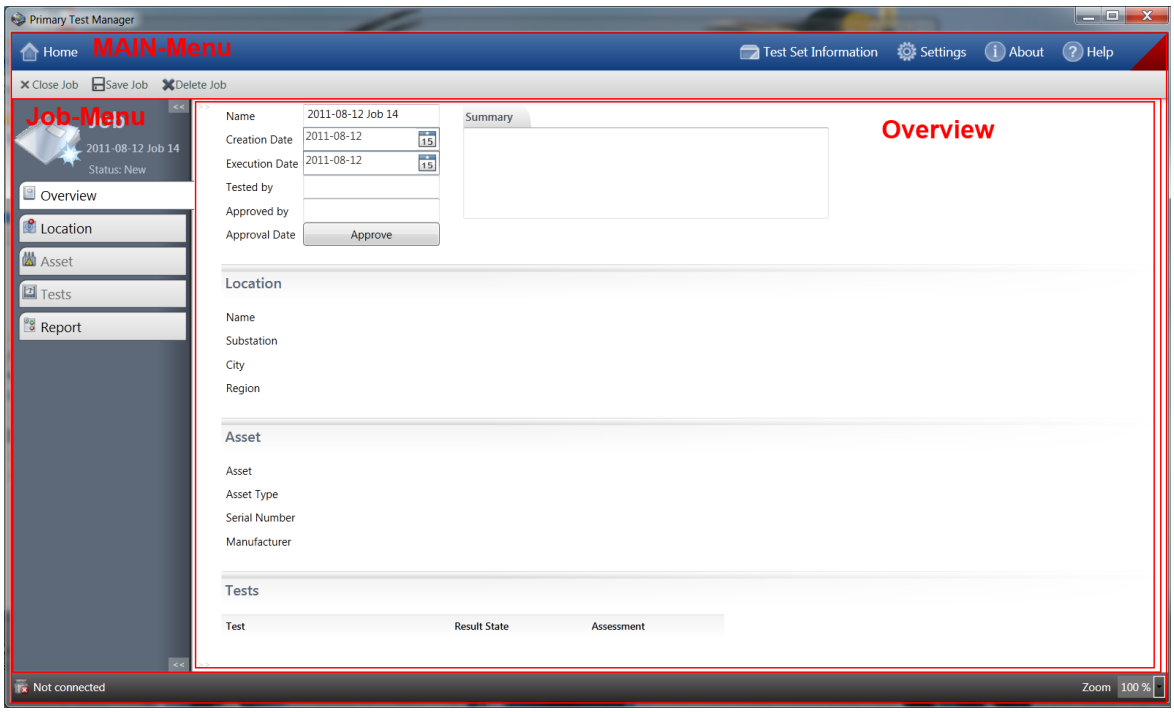

Abbildung A.1: Schablonen einer graphischen Oberfläche

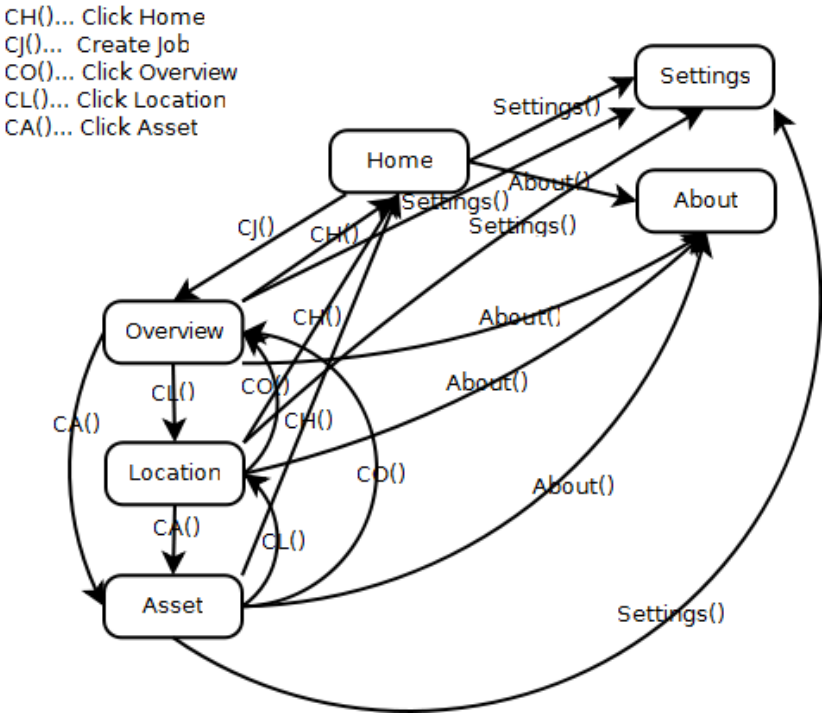

Abbildung A.2: Schlechtes Modellieren der Oberfläche

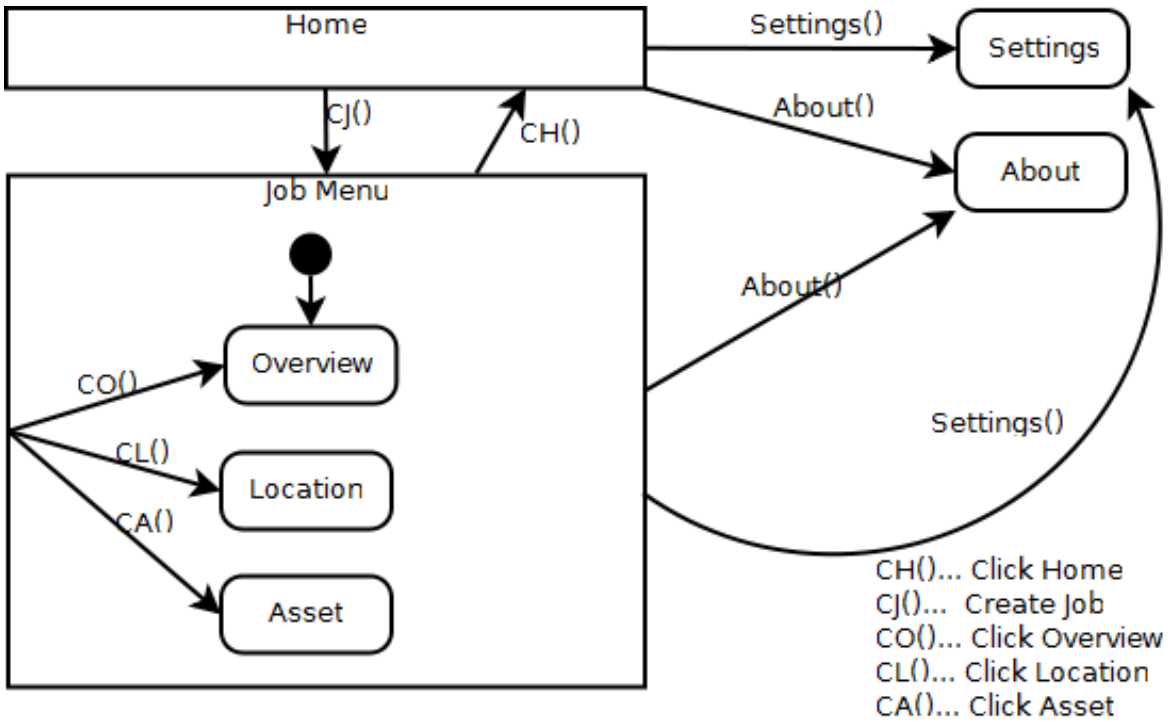

Abbildung A.3: Verschachtelte Modellierung der Oberfläche

A.2 Richtiges Beschreiben von Modalen Fenstern

Eine weitere typische Fehlerquelle beim Erstellen von GUI-Benutzungsmodellen sind modale Fenster wie z.B. Fehlermeldungen oder Konfigurationsdialoge (siehe Beispiel in Bild A.4). Modale Fenster sperren den Zugriff auf darunterliegende Fenster (meist die Hauptapplikation), gehören aber gerne zum Arbeitsablauf einer verschachtelten Teilansicht. Ein typischer Fehler ist hierbei das Modellieren des modalen Fensters innerhalb des verschachtelten States, welcher die Teilansicht repräsentiert, in welcher es Auftritt. Es sind jedoch alle States, in denen ein aktiver Substate verschachtelt ist, währenddessen auch aktiv und davon wegführende Transitionen können deshalb getriggert werden. Bild A.5 zeigt ein Beispiel für eine semantisch fehlerhafte Modellierung von modalen Fenstern. Der Substate *Edit Vector Group* repräsentiert hierbei ein modales Fenster. Wenn dieses Fenster aktiv ist, kann nur innerhalb dieses Fenster agiert werden. Jedoch ist es bei dieser Modellierung laut UML-Semantik möglich, den Trigger *Home()* auszulösen und so den State *JobView* (und damit auch den Substate *Edit Vector Group*) zu verlassen.

Man muss also bei der Modellierung von sochen Fenstern immer sicherstellen, dass keine alternativen Abläufe durchgeführt werden können, da diese semantisch falsch sind. Hierbei muss man zwischen 2 Arten von Verschachtelungen in Statemachines unterscheiden:

- Composite State: Liegt nur eine Verschachtelung mit Composite States vor, kann man das modale Fenster auf der obersten Region modellieren. Dadurch müssen alle

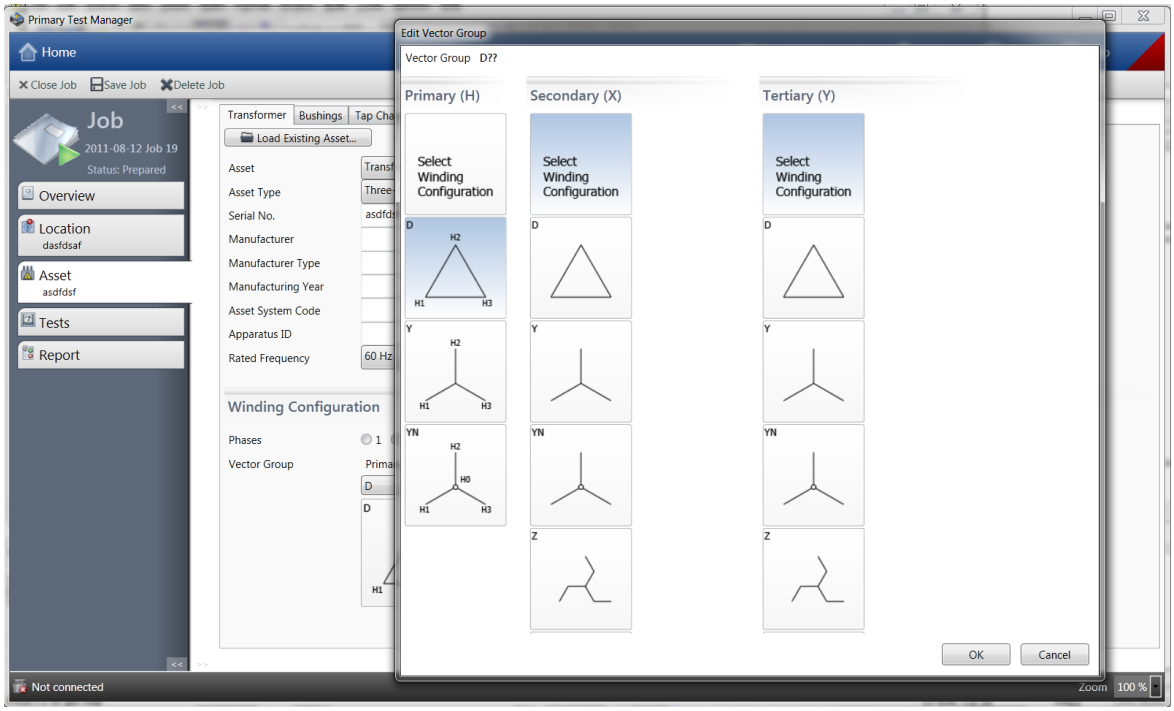

Abbildung A.4: Modales Fenster

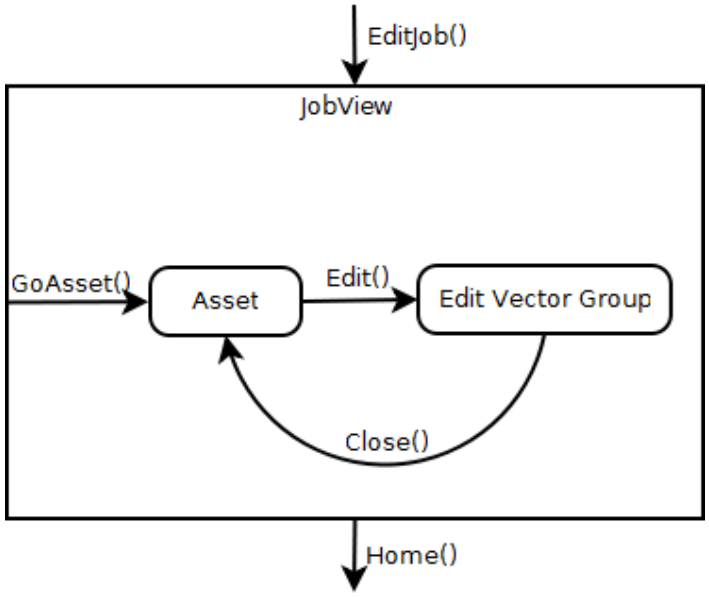

Abbildung A.5: Fehlerhaftes Modellieren des Modalen Fensters

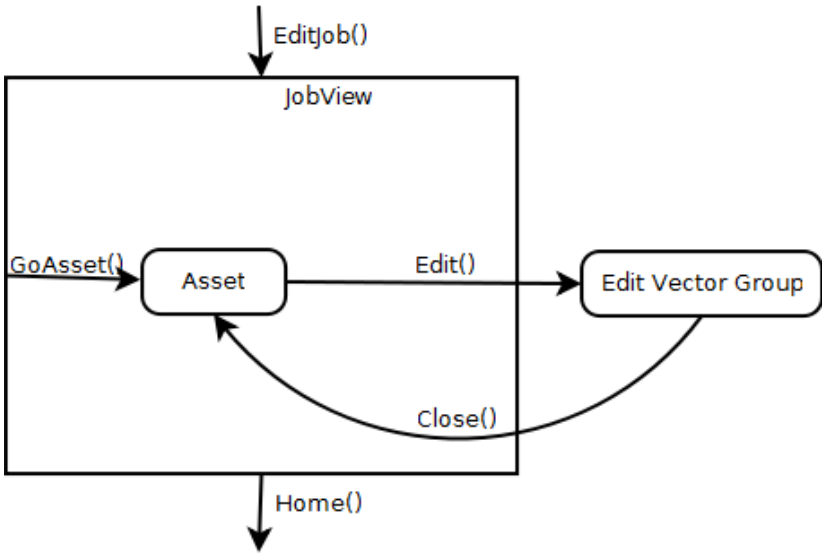

Abbildung A.6: Korrektes Modellieren des Modalen Fensters

verschachtelten Zustände verlassen werden (d.h. sie werden inaktiv) und es sind keine falschen Abläufe mehr möglich. Bild A.6 zeigt eine solche Modellierung.

- Statemachine: Bei einer Verschachtelung mit Statemachines kann man nicht auf die oberste Region zugreifen. Hier bleibt nur die Möglichkeit mögliche, darüberliegende Abläufe (Transitionen) mittels Guards zu sperren (das Modale Fenster setzt eine Flag, wodurch diese Transitionen nicht mehr getriggert werden können). Da dies jedoch keine sehr schöne Lösung darstellt, ist es zu empfehlen, auf Statemachines so gut wie möglich zu verzichten, wenn modale Fenster auftreten.

Analyse: Manuelle Risikoabschätzung und -modellierung in der Praxis

Im folgenden wird erklärt, wie die Risikofaktoren für das Beispiel-SUT der zweiten Fallstudie (siehe Abschnitt 6.2) modelliert und in der Gruppe abgeschätzt wurden.

B.3 Risikomodellierung

Im ersten Schritt wurde dabei das zuvor erstellte Benutzungsmodell (siehe 7) analysiert und überlegt, welche Bereiche man am sinnvollsten mit Risikofaktoren belegt. Einzelne Transitionen wurden hierbei eher selten belegt, da das vorliegende Modell sehr atomare Aktivitäten (wie z.B. Mausklicks) beschreibt, welche alleine eher schwer bezüglich eines Risikos abzuschätzen sind.

Es ist gerade bei der manuellen Risikoabschätzung einfacher, logisch zusammenhängende Aktivitäten (wie z.B. die Menge aller Aktivitäten, welche zum Erstellen eines Tests benötigt werden) abzuschätzen. Da bei der Erstellung des Benutzungsmodells darauf geachtet wurde, dass solche Aktivitätsmengen als Composite State oder Substatemachine modelliert werden (was allein schon aus Verständnis- und Wartungsgründen sinnvoll ist), war die Belegung des Benutzungsmodells mit solchen Risikofaktoren problemlos möglich. Hierfür wurden die entsprechenden Regionen zuerst mal mit (noch leeren) Risikofaktoren und Kommentaren, welche den logischen Zusammenhang der Aktivitäten beschreiben, belegt.

Als Risikoformel wurde die einfache Formel 3 verwendet, bestehend aus den Faktoren

- Probability of Fault (P), also der Wahrscheinlichkeit, dass ein Fehler auftritt
- Cost (C), also den dadurch verursachten Kosten

Für beide Faktoren wurde hierfür eine Bewertungsskala von 1 (wenig) bis 5 (viel) festgelegt. Da für die Fallstudie nur Tester zur Verfügung standen, welche alle ähnlich viel Erfahrung mit dem SUT hatten, wurde festgelegt, dass ein Faktor jeweils über den Durchschnittswert aller abgegebenen Bewertungen ermittelt wird. Jedoch war es jedem Beteiligten möglich, Bewertungen für einzelne Faktoren auszulassen, falls er der Meinung war, zu wenig Erfahrung mit dem entsprechnden Bereich zu haben.

B.4 Risikoabschätzung

Das Abschätzen der einzelnen Risikofaktoren sollte in einem Meeting erfolgen. Hierfür wurde zuerst in einem Probemeeting die Vorgehensweise erprobt und dann in dem eigentlichen Meeting die Risikofaktoren abgeschätzt.

Im Probemeeting mit 3 Personen wurde versucht, die Risikoabschätzungen direkt im Modell einzutragen. Dies gestaltete sich jedoch als eher umständlich und schwierig, da die einzelnen Diagramme nicht immer für jeden Beteiligten sofort verständlich waren. Das Modell selber reichte nicht aus, damit sofort jedem Beteiligten klar war, wofür genau der jeweilige Risikofaktor steht. Desweiteren kostete das Eintragen direkt über das Modellierungstool (Magicdraw) zusätzlich Zeit.

Ein weiteres Problem waren teilweise lange Diskussionen über einzelne Risikobereiche, da keine direkte Vorgehensweise für die Abstimmung definiert wurde. Als störend wurde dabei vor allem empfunden, dass man dadurch zu stark bei der Bewertung beeinfluss wurde, wodurch keine wirkliche individuelle Bewertung entstand.

Teilnehmer	Aktivität 1 PoF	Aktivität 1 Cost	Aktivität 2 PoF	Aktivität 2 Cost	...
Franz	3	4	3	1	...
Max	2	4	-	-	...
John	5	3	4	2	...
Average	3.34	3.67	3.5	1.5	...

Tabelle B.1: Beispieltabelle für die Abschätzung von Risikofaktoren

Für das richtige Meeting (mit 5 Personen) wurde deshalb folgende Vorgehensweise festgelegt:

- Ermitteln der Risikofaktoren in Tabellenform (Beispiel: siehe Tabelle B.1), dessen Werte später in das Modell übertragen werden
- Einfache Beschreibung des Bereichs bzw. der Aktivität, für welche ein Risikofaktor abgeschätzt werden soll. Falls vorhanden, kann hierfür die Verwendung eines GUI-Prototyps hilfreich sein.
- Für die Abschätzung geht man wie folgt vor:
  - Jeder gibt gleichzeitig (und unabhänig) seine erste Abschätzung ab (nach seinem ersten Eindruck)
  - Kurze (moderierte) Diskussion über die Abstimmung. Hier werden auch speziell Ausreisser (welche stark vom Durchschnitt abweichen) befragt. Dadurch können recht leicht Missverständnisse beseitigt werden.
  - Jeder kann nun seine Abschätzung korrigieren, falls er es für nötig hält.

Durch diese Vorgehensweise konnte dann das Meeting recht schnell durchgeführt werden. Die dort ermittelten Faktoren wurden dann in das Benutzungsmodell aufgenommen, welches dann bei der durchgeführten Fallstudie verwendet wurde.

Verwendete Akronyme

TAI	Test Automation Infrastructure
LTS	Labelled Transition System
ALTS	Activity Labelled Transition System
STS	Symbolic Transition System
SUT	System Under Test
PTM	Primary Test Manager
UML	Unified Modeling Language
GUI	Graphical User Interface (grafische Benutzeroberfläche)
XMI	XML Metadata Interchange
XML	eXtensible Markup Language
NP	Nichtdeterministisch Polynomiell
OMG	Object Modeling Group
OCL	Object Constraint Language
MCUM	Markov Chain Usage Model
TRSP	Total Risk Score Priorization
ARSP	Additional Risk Score Prioritization

Literaturverzeichnis

[Aic10] Bernhard K. Aichernig. Uml in action: A two-layered interpretation for testing. *Proceedings of UML&FM 2010, the third IEEE International Workshop on UML and Formal Methods*, 2010.

[Aml99] Stale Amland. Risk based testing and metrics: Risk analysis fundamentals and metrics for software testing including a financial application case study. *The Journal of Systems and Software*, 1999.

[Bac99] James Bach. Heuristic risk-based testing. *Software Testing and Quality Engineering Magazine*, 1999.

[Bin99] R. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 1999.

[Cho78] T. S. Chow. Testing software design modeled by finite-state machines. 1978.

[Cod] Coded ui test basic walkthrough: http://www.dotnetfunda.com/articles/article1241-coded-ui-test-basic-walkthrough-.aspx.

[CPC] Cpc 100 information: http://www.omicron.at/de/products/pro/primary-testing-diagnosis/cpc-100-series/cpc-100/. last visited Oktober 2011.

[CT06] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory, Second Edition*. JOHN WILEY & SONS, INC., PUBLICATION, 2006.

[EWZC11] Vladimir Entin, Mathias Winder, Bo Zhang, and Stephan Christmann. Combining model-based and capture-replay testing techniques of graphical user interfaces. *TAIC PART: Academic & Industrial Conference, Practice and Research Techniques*, 1:6, 2011.

[FG10] Abderrahmane Feliachi and Helene Le Guen. Generating transition probabilities for automatic model-based generation. *2010 Third International Conference on Software Testing, Verification and Validation*, 2010.

[FHB11] Michael Felderer, Christian Haisjackl, and Ruth Breu. Towards integrating manual and automatic risk assessment into a risk-based testing methodology. Technical report, Institute of Computer Science, University of Innsbruck, 2011.

[HAK08] Stallbaum Heiko, Metzger Andreas, and Pohl Klaus. An automated technique for risk-based test case generation and priorization. Technical report, Software Systems Engineering, University of Duisburg-Essen, 2008.

[Hem] Srikanth Hema. Requirement-based test case priorization. Technical report, Department of Computer Science, North Carolina State University.

[KKP08] Matthew Kaplan, Tim Klinger, and Amit Paradkar. Less is more: A minimalistic approach to uml model-based conformance test generation. *2008 International Conference on Software Testing, Verification and Validation*, 2008.

[KSA09] Willibald Krenn, Rupert Schlick, and Bernhard K. Aichernig. Mapping uml to labeled transition systems for test-case generation. *FMCO'09 Proceedings of the 8th international conference on Formal methods for components and objects*, pages 186–207, 2009.

[Mag] No Magic. Magic draw: https://www.magicdraw.com/.

[Mon01] Jerome Monnot. Approximation results toward nearest neighbor heuristic, July 2001.

[MXX06] Chen Mingsong, Qiu Xiaokang, and Li Xuandong. Automatic test case generation for uml activity diagrams. *Proceedings of the 2006 international workshop on Automation of software test*, 2006.

[OMG] OMG. Xmi 2.1 specification: http://www.omg.org/spec/xmi/2.1/.

[OMG07] OMG. Uml superstructure specification: http://www.omg.org/spec/uml/2.1.2/, 2007.

[Omi] Omicron electronics gmbh: http://www.omicron.at/en/. last visited Oktober 2011.

[Pic09] Roman Pichler. *Scrum: Agiles Projektmanagement erfolgreich einsetzen.* dpunkt.verlag, 2009.

[PRB10] Hyuncheol Park, Hoyeon Ryu, and Jongmoon Baik. Historical value-based approach for cost-cognizant test case prioritization to improve the effectiveness of regression testing. Technical report, School of Engineering, Information and Communications University, 2010.

[Ran] Ranorex. Ranorex webside: http://www.ranorex.de/.

[RBGW10] T. Roßner, C. Brandes, H. Götz, and M. Winter. *Basiswissen Modellbasierter Test.* dpunkt.verlag, 2010.

[RdBJ00] Vlad Rusu, Lydie du Bousquet, and Thierry Jeron. An approach to symbolic test generation. *IRISA/INRIA*, 2000.

[SDG09] Sebastian Siegl, Winfried Dulz, and Reinhard German. Model-driven testing based on markov chain usage models in the automotive domain. Technical report, University of Erlangen-Nuremberg, 2009.

[SL10] A. Spillner and T. Linz. *Basiswissen Softwaretest.* dpunkt.verlag, 2010.

[Stö05] Harald Störrle. *UML 2 für Studenten.* Pearson Studium, 2005.

[Sta73] H. Stachowiak. *Allgemeine Modelltheorie*. Springer-Verlag, 1973.

[SW] Christian Schwarzl and Franz Wotawa. Test case generation in practice for communicating embedded systems. Technical report, The Virtual Vehicle Competence Center, Graz University of Technology (IST).

[SW05] Hema Srikanth and Laurie Williams. On the economics of requirements-based test case prioritization. Technical report, North Carolina State University, 2005.

[Thi03] H. W. Thimbleby. The directed chinese postman problem. 2003.

[Whi] White (ui automation framework): http://white.codeplex.com/.

[WP00] G.H. Walton and J.H. Poore. Generating transition probabilities to support model-based software testing. 2000.

[WSS08] Stephan Weißleder, Dehla Sokenou, and Bernd-Holger Schlinglo. Reusing state machines for automatic test generation in product lines. Technical report, Humboldt-Universität zu Berlin, Institut für Informatik, 2008.

[WT94] James A. Whittaker and Micheal G. Thomason. A markov chain model for statistical software testing. *IEEE*, 1994.

[YR03] Chen Yanping and Probert Robert. A risk-based regression test selection strategy. Technical report, University of Ottawa, 2003.