Freie Universität Berlin

Institut für Informatik
AG Software Engineering

Automatisches Erkennen von Trial-and-Error Episoden beim Programmieren

Bachelorarbeit
im Fach Informatik

eingereicht von
Hannes A. Restel
(Matrikelnummer 3896781)

August 2006

Erstprüfer: Prof. Dr. Lutz Prechelt
Zweitprüfer: Prof. Dr. Carsten Schulte

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides Statt, dass ich die vorliegende Bachelorarbeit bis auf die offizielle Betreuung selbst und ohne fremde Hilfe angefertigt habe und die benutzten Quellen und Hilfsmittel vollständig angegeben sind.

Berlin, den 22. August 2006 ....

Unterschrift

Inhaltsverzeichnis

Teil A)Einleitung.. 4
Teil B)Theorie der Episoden.. 5
1.Was ist Trial-and-Error?.. 5
2.Einfluss von Entwicklungsumgebungen auf Trial&Error-Episoden............................. 5
3.Typen des Trial-and-Error.. 7
4.Debugging vs. Trial&Error.. 8
5.Einige einfache Videobeispiele von Trial&Error-Episoden.. 9
6.Eingesetzte Methoden zum Finden der Episoden.. 10
7.Der EventViewer – Ein Werkzeug zur Lokalisierung der Episoden............................ 12
8.Qualitatives und quantitatives Auftreten von T&E-Episoden.................................... 12
9.Aufbau von Episoden.. 14
9.1.Startindikatoren.. 14
9.2.Folgeindikatoren... 15
9.3.Abbruchindikatoren.. 15
9.4.Nichtindikatoren.. 15
9.5.Präindikatoren... 15
10.Fallbeispiel „Video005": Finden der notwendigen Granularität für den Episodenerkenner... 15
11.Vom Suchen und Finden der Indikatoren.. 18
12.Verlegenheitsänderungen und Semantische Codeabhängigkeiten.......................... 19
13.Finden von Abbruchindikatoren... 21
14.Fallbeispiel: „Video006": Technische Fokussierung auf den Episodenerkenner........ 21
Teil C)Technische Grundlagen für das Formalisieren der Trial&Error-Episoden............... 24
1.Beschreibung: Arbeitsablauf und Kommunikationswege des Episodenerkenners....... 25
2.Erläuterung der ECG-Ereignis-Handhabung und Bedeutung für die Arbeit................ 26
3.Erläuterung des Stellenverfolger-Moduls und Bedeutung für die Arbeit................... 26
3.1.Verwandtschaft von Codestellen.. 28
4.Erläuterung von ALED und Bedeutung für die Arbeit... 29
4.1.Anpassen der theoretischen Überlegungen an die Praxis in ALED.......................... 31
4.2.Hierarchien von Erkennern.. 31
4.3.Vorfilterung – eine Erweiterung der ALED-Spezifikation... 32
4.4.Abbruch eines Erkenners aufgrund eines Abbruchindikators.................................. 33
4.5.Definierte Objekte eines ALED-Rahmenwerkes... 34
4.5.a.Das episode-Objekt:... 35
4.5.b.Das event-Objekt:.. 36
4.5.c.Das output-Objekt:... 36
4.5.d.Das Schlüsselwort ABORT:.. 36
Teil D)Formalisierung der Trial&Error-Episoden... 37
1.Formalisierung der allgemeinen Trial-and-Error-Episode... 38
2.Formalisierung der Indikator-Erkenner.. 38
2.1.T&E-Indikator: Vereinfachung komplexer Ausdrücke.. 39
2.2.T&E-Indikator: Auskommentieren von Code.. 39
2.3.T&E-Indikator: Einfügen von Testausgaben... 40
3.Formalisierung der speziellen Trial-and-Error-Episoden.. 40
3.1.T&E-Episode: Schleifenbedingungen mehrfach verändert....................................... 40
3.2.Heuristische T&E-Episode aufgrund des Auftretens von T&E-Indikator-Ereignissen.... 41
4.Formalisierung der Über-Episoden... 42
4.1.merging-Erkenner: Zusammenschließen (merging) von Trial-and-Error-Episoden....... 42
Teil E)Zusammenfassung und Ausblick.. 43

Teil F)Verzeichnisse und Anhänge.. 44
1.Literaturverzeichnis.. 44
2.Materialverzeichnis... 45
3.Anhänge.. 45
3.1.Danksagung.. 45
3.2.Dokumentation Event Viewer... 46
3.3.Formular für die Observierungen.. 48
3.4.Übersicht der Syntax von Java-ALED.. 49

Teil A) Einleitung

Fehler können während der Programmierung auf vielfältigste Weise begangen werden.

Ein nicht unbedeutender Teil dieser Fehler ist auf Unsicherheiten des Entwicklers zurückzuführen, er weiß einfach nicht, wie das Problem zu lösen ist: Heißt es *length()* oder *size()*? Geht die Schleife bis *n* oder bis *n-1*? Ist die quantenmechanische Eignungskonstante von Baryonen 32,465 oder war es doch 34,265? „Lass es uns einfach probieren und so lange raten, bis es zu stimmen scheint!", mag die Antwort vieler Programmierer darauf sein.

Unsicherheiten dieser Art führen unweigerlich zu Defekten[1] im Programm, welche zu einem Versagen[2] führen können. Um diesen unabsichtlichen Fehlern[3] vorzubeugen, wird der Mikroprozess der Softwareentwicklung[4] betrachtet, also das Vorgehen einzelner Programmierer bei ihrer Implementierungsarbeit.

Einzelne Phasen während der Implementierungsarbeit nennen sich *Episoden*. Werden Episoden mit Hilfe geeigneter Werkzeuge und Techniken betrachtet, analysiert und ausgewertet, so lassen sich daraus erfolgreich Methoden und Techniken zur Defekterkennung und Defektvermeidung entwickeln.

Im Rahmen dieser Arbeit wird eine Teilmenge der bereits oben erwähnten Episoden diskutiert, nämlich die so genannten *Trial&Error-Episoden* (Abkürzung *T&E-Episoden*). Am Ende dieser Arbeit soll eine Sammlung formalisierter und kategorisierter T&E-Episoden stehen, welche sich mit einem entsprechenden Programm ausführen lassen, um auftretende Episoden tatsächlich in der Praxis erkennen zu können.

Diese Arbeit beschäftigt sich nicht ausschließlich mit Formalisierungen von Trial-and-Error-Episoden, sondern will einen Beitrag zum Verständnis und den Charakteristika aller Episoden leisten.

---

[1] Ein Defekt entsteht aufgrund eines Fehlers des Softwareentwicklers
[2] Ein Versagen ist ein im Sinne der Spezifikation falsches Verhalten des Programms
[3] Ein Fehler ist entweder ein *Falschtun* (commission) oder ein *Versäumnis* (ommission), wobei einem Fehler entweder ein *Irrtum* (bewusstes, absichtliches Handeln; engl. mistake) oder ein *Versehen* (blunder) zu Grunde liegt.
[4] Siehe: http://projects.mi.fu-berlin.de/w/bin/view/SE/MicroprocessHome (von Sebastian Jekutsch)

Teil B) Theorie der Episoden

1. Was ist Trial-and-Error?

Zuerst gebe ich eine Definition zum Verständnis des Trial&Error an: „Unter Versuch und Irrtum (englisch trial-and-error) versteht Edward Lee Thorndike eine Problemlösungsmethode, bei der so lange zulässige Lösungsmöglichkeiten probiert werden, bis die gewünschte Lösung gefunden wird. Dabei wird bewusst auch die Möglichkeit von Fehlschlägen in Kauf genommen“[5].

Ich konnte während meiner Arbeiten zwei Arten des Versuchs und Irrtums identifizieren: Das *bewusste Trial&Error* ist eine echte Problemlösungs*methode*, bei der sich der Entwickler seines Versuchs und Irrtums und den daraus resultierenden Gefahren der erhöhten Defektlastigkeit bewusst ist, während dessen das *unbewusste Trial&Error* spontan und zufällig auftritt, in dem der Programmierer das Programm einfach solange ändert und testet, bis ein Defekt scheinbar behoben wurde, ohne einen Gedanken an mögliche negative Auswirkungen auf die Stabilität des Programms aufzuwenden.

Somit lässt sich *bewusstes T&E* beispielsweise für einen spontanen Test einsetzen oder für das schnelle Auffinden eines Defekts, was mit anderen Techniken wie etwa einem Debugger zu lange dauern würde oder zu „schwergewichtig“ für das betrachtete Problem wäre. Die Anwendung eines *unbewussten T&E* hingegen wird vom Entwickler nicht bemerkt und ist somit die größere Gefahr für ein Versagen der Software.

In der Praxis sind diese beiden Möglichkeiten des Versuchs-und-Irrtums nur sehr schwer differenzierbar, unter anderem deshalb, weil eine Identifizierung von T&E-Episoden auf technischer Ebene nur auf Basis phänomenologischer Beobachtungen getroffen werden kann, insbesondere wenn ein Computeralgorithmus diese Identifizierungen zu treffen hat. Dies bedeutet, dass nur die Aktionen eines Programmierers betrachtet werden können, jedoch seine innere Motivation – also seine Gedanken - nicht zugänglich sind. Deshalb geht diese Arbeit nur relativ begrenzt auf eine Unterscheidung zwischen bewusstem und unbewusstem T&E ein.

2. Einfluss von Entwicklungsumgebungen auf Trial&Error-Episoden

Glücklicherweise wurde bereits eine Fülle recht ausgefeilter Entwicklungsumgebungen (so genannte *IDE*s – Integrated Development Environments) entwickelt[6], welche in der Lage sind einen Großteil der vom Entwickler produzierten Fehler bereits vor dem Ausführen und Testen des Programms abfangen zu können und sogar das Entstehen von Defekten zu vermeiden.

IDEs sind zu einer neuen Schicht zwischen dem Programmierer und seinem erzeugten Code geworden, so dass der Entwickler mehr und mehr mit der IDE anstatt direkt mit seinem Quellcode interagiert. Moderne IDEs geben beispielsweise hinreichend Informationen über die korrekte Verwendung von Methoden und Attributen, beugen (unter anderem durch Unterstützung beim Refactoring[7]) falscher Verwendung von Variablennamen und Namensräumen vor, verhindern die falsche Benutzung von Klammersetzung und erleichtern den Umgang mit objektorientiertem „Spaghetticode“[8].

---

[5] Quelle: http://de.wikipedia.org/wiki/Trial_and_error , 15.07.2006

[6] Einige Vertreter von IDEs: Eclipse, NetBeans, KDevelop, Microsoft Developer Studio, Borland Delphi

[7] Refactoring: das projektweite semantikerhaltende automatische Restrukturieren des Codes, z.B. die Umbennenung von Variablen

[8] Bsp: „differenz = this.berechne(alleEvents.elementAt(jListEpisodeList.getSelectedIndex())[1]).getSeconds()“

Oft werden IDEs von Programmierern als ein „erweitertes Gedächtnis" eingesetzt, welches sich für den Entwickler die korrekte Schreibweise von Variablen, Methodenköpfen und -parametern von APIs[9] vollständig merken kann. IDEs haben also in den letzten Jahren den gesamten Codierprozess stark beeinflusst und die Art und Weise des Programmierens nachhaltig verändert. Bereits bei der Verwendung einer modernen Entwicklungsumgebung führt der Programmierer stets eine Art Trial&Error durch, da es beispielsweise ausreichend ist die ersten Lettern eines Variablennamens zu schreiben und sich anschließend für einen der von der IDE vorgegebenen Vorschläge zu entscheiden (siehe Abbildung 1).

Abbildung 1: Beispiel Hilfe von IDEs bei Bezeichnern

Auch können bewusst Fehler erzeugt werden, um auf Verbesserungsvorschlägen der IDE zu hoffen (siehe Abbildung 2 und 3).

Abbildung 2: Hilfe der IDE bei API-Feinheiten

Es werden also bestimmte Annahmen getroffen, eine bestimmte Schreibweise ausprobiert (Trial) und diese Annahmen werden sofort von der IDE bestätigt oder widerlegt (Error bzw. Success).

Das Videobeispiel *Video001* zeigt sehr anschaulich, wie die IDE den Entwickler beim Einsatz einer Klasse unterstützt und ihn so lange begleitet, bis die gewünschte Methode der gewünschten Klasse erfolgreich verwendet wird.

Diese neue Schicht der maschinellen Intelligenz – oder wenigstens des maschinellen Codeverständnisses – beugt einem Auftreten vieler Fehler und damit dem aktiven Einsatz vieler Trial&Error-Fälle präventiv vor, so dass diese Fälle, wenn auch vor ein paar Jahren noch intensiv genutzt, in der Praxis heute als ausgestorben gelten dürfen. Im Folgenden werden alle präventiv verhinderten Fehler als *inhibitorische Fehler*[10] deklariert.

[9] API = Application Programming Interface; eine Sammlung von Bibliotheken einer Programmiersprache

[10]Inhibition (lat. inhibere) bedeutet Unterbinden, Hemmung; siehe http://de.wikipedia.org/wiki/Inhibitorisch, 20.08.2006

Abbildung 3: Vorschläge der IDE bei Unsicherheit

Alles in Allem jedoch vermögen IDEs nur die Behebung syntaktischer Fehler, also Fehler welche die erfolgreiche Kompilierung und/oder Ausführung des Programms verhindern. Auch wenn sich die IDEs in Zukunft weiterentwickeln, so werden sie wohl nicht in der Lage sein, inhaltliche oder semantische Fehler zu erkennen, also zu verstehen was der Entwickler implementieren möchte.

3. Typen des Trial-and-Error

Aus diesem Grund ist eine weitere Differenzierung des Begriffs des Trial&Error notwendig: Als *syntaktische Fehler* definiere ich jene Fehler, welche ein erfolgreiches Starten und Ausführen des Programms verhindern. Folglich sind *syntaktische T&E-Episoden* die Episoden, in denen der Entwickler versucht diese Fehler zu beheben, sich aber der korrekten Syntax der eingesetzten Sprache nicht sicher ist (siehe dazu Videobeispiel *Video002*: „Erfolgloses Starten eines Kommandozeilenprogramms"). Die inhibitorischen Fehler fallen in die Klasse der syntaktischen Fehler, da sie sich auf die Syntax beziehen. Diese Arbeit geht davon aus, dass in naher Zukunft beinahe keine syntaktischen T&E-Episoden mehr in der Praxis auftreten werden, da deren Entstehen durch moderne Entwicklungsumgebungen verhindert werden wird.

Das Auffinden und Beheben von syntaktischen Fehlern trägt nur unwesentlich zum Ziel der Defektvermeidung bei: Erstens weist das Programm bei Eintreten eines syntaktischen Fehlers sofort ein offensichtliches Versagen auf (beispielsweise durch einen Absturz des Programms) anstatt nur einen potentiell gefährlichen und unentdeckten Defekt zu beinhalten, und zweitens lassen sich syntaktische Fehler sehr leicht und vollständig lokalisieren, so dass auf diesem Gebiet keine Forschung mehr notwendig ist. Aus diesem Grund sind syntaktische Fehler und syntaktische T&E-Episoden in dieser Arbeit nicht von Belang!

Die in dieser Arbeit betrachteten T&E-Fälle sind vielmehr die *semantischen T&E-Fehler*, welche sich auf den Inhalt und die Logik des Programms beziehen, also auf den Versuch, ein Programm in ein der Spezifikation getreues Verhalten zu überführen. Eine *semantische T&E-Episode* ist eine Phase des Programmierprozesses, in welcher der Entwickler die

inhaltlichen Fehler seines Programms entfernt bzw. versucht zu entfernen, wobei er aber keine wohl überlegten Codeänderungen tätigt, sondern den Code beständig modifiziert und so lange probiert, bis alle Fehler beseitigt zu sein scheinen.

Diese Arbeit geht von der Theorie aus, dass die meisten T&E-Episoden intuitiv gesteuert sind - sich der Programmierer seiner momentanen Trial-and-Error-Tätigkeit nicht bewusst ist - und nur ein kleiner Teil aller T&E-Episoden auf bewusstes T&E zurückzuführen ist: Der Entwickler ist so sehr auf die Lösung des Problems fixiert, dass er zunächst die „rationalen" Methoden der Problemlösung wie ein „sich Zurücklehnen und über den Code nachdenken" oder die Erzeugung einfacher Testfälle zur Defektlokalisierung ignoriert bzw. ihm zunächst nicht in den Sinn kommen. Erst nach einigen erfolglosen Versuchen nimmt der Entwickler Abstand und beginnt, analytisch nachzudenken (siehe Videobeispiel *VIDEO003*). Das Beispiel zeigt, wie der Programmierer trotz Hilfe der IDE ein Problem der Objektorientierung nicht lösen kann. Erst nach gut drei Minuten stoppt der Programmierer sein Raten und schaut in eine andere Stelle des Quellcodes, wo das gleiche Problem bereits gelöst wurde, und übernimmt die Lösung.

Es leuchtet sofort ein, dass unbewusstes Trial-and-Error prekär bezüglich der Defektlastigkeit eines Programms sein kann, da sich der Programmierer seines unsicheren Handelns nicht bewusst ist. Ein erster positiver Eindruck bezüglich der Korrektheit genügt, um Folgeaktivitäten – wie etwa die Zuwendung zum nächsten algorithmischem Problem – anzustoßen. Eine der Episode nachgeschaltete Qualitätssicherungsphase findet (wenn überhaupt) nur in einer ungenügenden Sorgfalt statt.

Der Vollständigkeit halber sei erwähnt, dass ich zusätzlich noch Episoden definiert habe, welche ich *Trial&Success* (T&S) und *Trial&Ignore* (T&I) nenne. T&S macht eine Irrtumsphase nichtig, da das Programm auf Anhieb korrekt zu laufen scheint; T&I ist die Nichtbeachtung bzw. willentliche Nichtbehebung eines gefundenen Fehlers. Diese beiden Arten scheinen etwas ungewöhnlich zu sein, traten während meinen Forschungen aber durchaus einige Male auf.

4. Debugging vs. Trial&Error

Eine Beschäftigung mit den Begriffen Irrtum, Fehler und Defekt führt unweigerlich zu einer Auseinandersetzung mit der Idee des *Debugging*. Debugging ist „the process of finding and removing the causes of failures in software"[11]. Der Artikel zu Debugging auf en.wikipedia.org erweitert diese Definition um den Zusatz des *methodischen* Prozesses[12].

Debugging hat also die aktive und insbesondere die bewusste Defektlokalisierung und anschließende Korrektur des Defekts zum Ziel. Debugging wird mit definierten „Tools & Techniques"[13] wie etwa einem Debugger[14] oder Codedurchsichten durchgeführt.

Debugger kamen bei meinen Beobachtungen jedoch nur bei einer einzigen Testperson zum Einsatz, dafür aber sehr intensiv. Im Laufe meiner Forschungen stellte ich fest, dass ein einfacher und oft zeitsparender Ersatz für den Einsatz eines Debuggers das Einpflegen

[11]Definition Debugging: http://www.testingstandards.co.uk/living_glossary.htm vom 23.07.2006
[12]Artikel zum Debugging: http://en.wikipedia.org/wiki/Debugging vom 23.07.2006
[13]Ausdruck aus dem PmBoK („Project Management Body of Knowledge")
[14]Ein Debugger ist ein Programm zur schrittweisen Ausführung eines Programms mit dem Ziel, Momentanaufnahmen der einzelnen Programmzustände (z.B. der Variableninhalte) festhalten zu können

von für den Entwickler nachvollziehbaren Testausgaben in den Quellcode ist.

Techniken des Debugging werden nicht nur zur Fehlerbehebung eingesetzt, sondern dienen häufig auch zur Etablierung oder Erhöhung des Verständnisses des Programmverhaltens.

Der Defektlokalisierung schließen sich Defektbehebungsstrategien an. Die verwendete Strategie zur Defektbehebung kann im Gegensatz zu der vorhergehenden systematischen Defektlokalisierung auch ein unstrukturiertes Trial&Error sein. T&E-Episoden treten also auch als Teilaktivität eines Debugging-Prozesses in Erscheinung und sind oft von einem Debugging-Prozess umschlossen.

Bezug nehmend auf diese Arbeit wird der Begriff des Debugging abgeschwächt und in ihm nur der *Versuch* der Behebung eines Defekts gesehen. Eine systematische Vorgehensweise für die Lokalisierung und Behebung ist nicht notwendig. Debugging bleibt trotz der Abschwächung weiterhin ein zweigeteilter Prozess: Erstens die Defektlokalisierung und zweitens die Defektbehebung. Die Defektlokalisierung hat aufgrund nicht vorhandener Codeänderungen keinen Einfluss auf das semantische Verhalten des Programms[15]. Während dieser Phase ist die Interaktion zwischen Entwickler und Computer bezüglich der Benutzung von Eingabegeräten also sehr beschränkt: Ein Durchlesen des Codes genügt, weshalb ein auf Tastatur- und Mauseingaben des Benutzers angewiesener Erkenner-Algorithmus die Defektlokalisierungsphase nicht oder nur sehr unvollständig erkennen kann.

Erst der zweite Teil – die Defektbehebung - modifiziert das Programm.

Lokalisierung und Defektbehebung treten häufig gemeinsam auf und überschneiden sich sogar. In der Praxis lassen sich beide Aktivitäten nur sehr schwer voneinander trennen. Deshalb habe ich mich entschlossen, meine formalisierten Episoden allgemeiner zu fassen, so dass diese auch vorgeschaltete oder zwischengeschaltete Lokalisierungsphasen als Teil einer T&E-Episode identifizieren werden. Somit werden Debugging und Trial-and-Error in den formalisierten Episoden-Erkennern zu einem Prozess vereint.

Video010 verdeutlicht dies. Aufgrund vieler ähnlicher Anweisungen fällt die genaue Positionierung des Defekts schwer. Der Entwickler schaut sich mehrere Male den Code relativ ausführlich an, ändert eine kleine Stelle, führt das Programm aus und wiederholt diesen Zyklus.

5. Einige einfache Videobeispiele von Trial&Error-Episoden

Bevor ich zu dem tiefer gehenden theoretischen Teil der Arbeit gelange, nämlich der genauen Formalisierung der T&E-Episoden inklusive der resultierenden Probleme, möchte ich einige Beispiele von Episoden geben sowie genauer erläutern, wie das Vorgehen zum Auffinden der Episoden im Allgemeinen ausgesehen hat.

Zunächst einmal sei hier ein gemeinsames Merkmal – gewissermaßen die „Oberklasse" - aller T&E-Episoden genannt: Jede T&E-Episode ist im engen zeitlichen Umfeld einer Ausführung des Programms angesiedelt, da nur aufgrund eines Testlaufs ein semantischer

[15]Einzige zulässige Ausnahme der Codemanipulation während der Defektlokalisierung ist das Einfügen von Testausgaben und das eventuelle spätere Entfernen derselbigen nach erfolgreicher Lokalisierung. Wenn zusätzlich in einem angemessenen Zeitrahmen auf das Einfügen der Testausgaben keine weitere Modifikation des Codes erfolgte, handelt es sich um Etablierung des Codeverständnisses und nicht um ein vollständiges Debugging mit Defektbehebung.

Fehler erkannt werden kann. Im Anschluss findet eine Modifikation des Codes aufgrund des semantischen Fehlers statt und endet in einer erneuten Ausführung. Dieser Ausführen-Modifizieren-Ausführen-Zyklus geschieht solange, bis der Fehler behoben zu sein scheint.

Ein einfaches Beispiel ist ein mehrmaliges Verändern eines Schleifenkopfs einer for-Schleife, welcher erst bei der dritten Änderung die gewünschte Anzahl an Durchläufen erreicht.

Weil ein Bild bekanntlich mehr als „1000 Worte" sagt, ist den meisten Beispielepisoden ein Video beigelegt, welches die entsprechende Situation visualisiert. Die Beispiele sind durchnummeriert. Im Materialverzeichnis dieser Arbeit finden sich die entsprechenden Verknüpfungen zum Videomaterial. Das for-Schleifen-Beispiel ist *VIDEO004* zu finden.

Diese Videos sind entweder konstruierte Beispiele und dienen dem besseren Verständnis einer Situation, andere sind direkt der Realität entnommen. So können viele Überlegungen und Entscheidungen dieser Arbeit durch Anschauungsmaterial belegt und die hier präsentierten Theorien gestützt werden.

VIDEO005 ist bereits etwas komplexer: Aufgrund eines Copy-and-Paste-Fehlers[16] ist die korrekte Programmausführung nicht mehr gegeben: Zur Fehlerbehebung wird eine Testausgabe auf dem Bildschirm erzeugt, der Quellcode nach dem Fehler durchsucht und der Fehler schließlich behoben. Nach der Erörterung der verwendeten Methoden im folgenden Kapitel werde ich eine Diskussion der aufgetretenen Probleme anhand des Beispiels *VIDEO005* führen.

VIDEO006 schließlich ist eine komplexe Episode. In diesen neun Minuten werden sehr viele Aktivitäten ausgeführt, darunter auch eine Vielzahl von Aktivitäten welche nicht primär der Episode zugeordnet sind, sondern parallel zu ihr ablaufen und ein „Rauschen" darstellen. Die Behebung des Defekts aus *VIDEO006* ist trivial (eine Null in eine Eins umwandeln), aber die Lösungsfindung währt sehr lange, da sich der Programmierer einmal mehr der Position des Defekts im Code nicht sicher ist. Dieses Beispiel wird weiter unten im Kapitel 14 fallstudienartig erläutert werden, da es geradezu ein Paradebeispiel für eine Fülle der auftretenden Probleme für die Automatisierung der Trial&Error-Erkennung darstellt.

6. Eingesetzte Methoden zum Finden der Episoden

Wie nun lassen sich Episoden überhaupt identifizieren?

Ich musste mir hier unter Anderem folgende Gedanken machen: Handelt es sich um T&E, um Debugging oder um sonstiges Vorgehen? Was möchte der Programmierer mit seiner eben getätigten Codeänderung erreichen? Bezieht sich die Änderung noch auf die betrachtete Episode? Wann ist das Ende einer Episode erreicht? Wie genau sehen eigentlich T&E-Episoden aus?

In diesem Abschnitt der Arbeit habe ich mir noch keinerlei Gedanken über einen möglichen Erkenner-Algorithmus, also das „Automatische Erkennen von Trial-and-Error-Episoden" (siehe Titel dieser Arbeit) gemacht, sondern einen Top-Down-Ansatz verfolgt, um einerseits ein allgemeines Verständnis für Episoden zu generieren und andererseits bereits Episoden identifiziert zu haben, um später die Komplexität eines automatischen Episodenerkenners reduzieren zu können.

[16]Mehr zum das Thema Copy&Paste-Episoden: http://projects.mi.fu-berlin.de/w/bin/view/SE/ThesisCopyTracking vom 20.08.2006

In dieser Arbeit kamen vier Methoden zum Einsatz, um mögliche Episoden zu identifizieren.

Der erste Ansatz war, durch eigene Überlegungen und Selbstbeobachtungen mögliche Episoden zu entwerfen und diese durch die nächsten drei Schritte erweitern oder widerlegen zu lassen.

Ein zweiter geplanter Schritt war das Durchführen von Befragungen anderer Programmierer (ausschließlich Studenten), wobei sich jedoch schnell ein großer Einarbeitungsaufwand für die Befragten feststellen ließ: Erkenntnisreiche Antworten könnten sie nur geben, wenn sie sich bereits im Vorfeld der Befragungen mit dem Thema Trial&Error auseinander gesetzt hätten.

Deshalb habe ich die Interviews in die dritte Methode integriert, nämlich das Beobachten – also „über die Schulter schauen" - von Entwicklern während ihrer Arbeit. Diese Live-Observierungen waren nicht rein passiv, da ich aktiv in den Mikroprozess der Programmierung der Testpersonen eingegriffen habe: So habe ich vor jeder Beobachtungen zur Sensibilisierung einen kleinen Diskurs zum Thema mit der Testperson geführt und während der Beobachtung Fragen gestellt, Anmerkungen gemacht und dadurch mehr Verständnis für die Vorgehensweisen der Entwickler erlangt, als ohne ein „Stören" meinerseits. Diese Methode ist nicht wissenschaftlich oder empirisch, da ich die Testumgebung und wahrscheinlich auch die beobachteten Ergebnisse beeinflusst habe, aber einen empirischen Anspruch war diesen Beobachtungen auch nicht zugeschrieben: Sie dienten vielmehr dem Erkenntnisgewinn und der Inspiration zu möglichen T&E-Episoden. Zusätzlich konnte ich durch diese Beobachtungen bereits entworfene T&E-Episoden wieder verwerfen, wenn sie sich in der Realität nicht bestätigten. Wann immer es mir möglich war, habe ich ein kleines Programm auf den Computern der Testpersonen installiert, welches den Bildschirm als Video abfilmte (ein so genanntes *Screen Video Capturing Programm*), um eine fundierte Basis für meine vorgestellten T&E-Episoden geben zu können.

Um vergleichbare Resultate jeder Beobachtung zu erhalten, habe ich im Vorfeld der Beobachtungen ein Formular erstellt, welches mir ein „normiertes" Protokollieren ermöglichte. Das Formular enthielt unter anderem die Einträge: Datum, Dauer Arbeitsumfeldbeschreibung, Freitextfelder sowie Strichlisten zum Quantifizieren des Auftretens bereits definierter Episoden[17].

Meine ursprüngliche Bedenken, dass die Teilnehmer aufgrund meiner Präsenz und dem daraus resultierenden Beobachtet-werden-Gefühl konzentrierter und weniger hastig arbeiten würden und somit eine geringere Trial&Error-Tätigkeit aufweisen würden als unter normalen Umständen, stellten sich glücklicherweise recht schnell als unbegründet heraus. Im Gegenteil: Viele der spontanen Ausrufe und Kommentare (wie etwa „ich bin ein Dussel!", „bitte, bitte, bitte.." oder nach einer erfolgten Änderung: „Hmm.. Scheint zu gehen...") halfen mir, die inneren Gedanken der Testpersonen deuten zu können.

Als Vergleichsbasis für diese Behauptungen dient die Auswertung von zahlreichen bereits aufgezeichneten Videos, welche Mitschnitte vom Bildschirm des Entwicklers während des Implementierungsprozesses zeigen, womit sich auch langwierige Sitzungen beobachten ließen. Dieses war die vierte meiner verwendeten Methoden.

Bei den Live-Observierungen und den Videos habe ich Studenten meines Jahrgangs, einige Mitglieder des ACM Programming Contest Teams und zwei professionelle Programmierer

[17]Ein Formularvordruck findet sich im Anhang

beobachtet. Die von den Probanden verwendeten Programmiersprachen waren sehr vielfältig: C, C++, Qt, PHP, SQL, Delphi und Java. Auch die Orte der Beobachtungen (Universität, Arbeitsplatz und private Wohnungen) deckten alle möglichen Arbeitsumfelder ab.

7. Der EventViewer – Ein Werkzeug zur Lokalisierung der Episoden

Um ein schnelles Navigieren der zum Teil mehrstündigen Videos zu ermöglichen und damit eine effizientere Ausnutzung meiner aufgewendeten Zeit zu erreichen, habe ich als Teil meiner Arbeit ein Programm namens *EventViewer*[18] entwickelt. Es verbindet aufgezeichnete Ereignisse (in einer Textdatei gespeichert) und das dazugehörige Video derart, dass durch Anwahl eines beliebigen Ereignisses zur korrespondierenden Position im Video gesprungen wird. Die Ereignis-Datei oder auch „Lesezeichen"-Datei (analog zu den Kapiteln einer DVD) wird „on-the-fly" während der Benutzung der IDE von einem Eclipse-Plugin namens *ElectroCodeoGram*[19] erzeugt. Ebenfalls möglich ist es, die Lesezeichen manuell zu erzeugen und in eine Datei zu schreiben. Gefiltert werden können die Lesezeichen durch die Angabe eines Ausdrucks (z.B. „codechange"), wobei dann nur die den Ausdruck enthaltenen Lesezeichen beachtet werden.

Im Anhang findet sich eine genauere Erläuterung des *EventViewers*.

Genutzt habe ich das Programm, indem ich die Videos nach Auftreten der Ereignisse „Programm ausführen" bzw. „Debugger ausführen" durchsucht habe und mir die Stellen in unmittelbarer Umgebung betrachtete. Durch dieses Vorgehen wurden natürlich viele Episoden ignoriert, welche sich nicht in einem „Programm Ausführen"-Radius befunden haben. Dieses Vorgehen ist dadurch gerechtfertigt, dass laut Definition des T&E ein Versuch nur dann auftreten kann, wenn tatsächlich versucht bzw. getestet wird. Die einzige Möglichkeit bei der Programmierung einen Versuch zu starten, ist ein Ausführen des Programms, weshalb auch nur diese Stellen relevant für mich waren und die übersprungenen Codestellen somit keine semantischen T&E-Episoden sein konnten.

8. Qualitatives und quantitatives Auftreten von T&E-Episoden

Die Anzahl auftretender T&E-Episoden variiert je nach Art und Größe der Programmieraufgaben, der Phase in dem sich ein Projekt befindet sowie der Erfahrung und Persönlichkeit des Entwicklers.

Selbstverständlich können die Ergebnisse meiner wenigen Beobachtungen nicht repräsentativ sein, sie geben die groben Tendenzen aber recht gut wieder.

So beobachtete ich bei den professionellen Programmierern wesentlich weniger T&E-Episoden als bei den Studenten. Nur eine einzige Testperson – ein Entwickler - setzte einen Debugger und fortgeschrittene Konstrukte wie Assertions ein.

Wesentlich abhängig von der Häufigkeit der T&E-Episoden ist auch die Art der Programmieraufgaben: In domänenfremden Aufgaben ist die Anzahl der T&E-Vorgänge erhöht, wo hingegen ich bei der Beobachtung eines zweiten Berufsprogrammierers – welcher tagtäglich dieselben Aufgaben löst – keinen einzige unbewusste T&E-Episode beobachten konnte.

Besonders groß war der Unterschied zwischen API-intensiven und Algorithmen-intensiven

[18]Der *EventViewer* ist zu beziehen unter: http://projects.mi.fu-berlin.de/w/bin/view/SE/ThesisTrialErrorEventViewer vom 21.08.2006

[19]ElectroCodeoGram: http://projects.mi.fu-berlin.de/w/bin/view/SE/ThesisEclipsemonitor vom 21.08.2006

Aufgaben: Ein Student, welcher ausschließlich mit den ihm bis dahin unbekannten Qt-Bibliotheken arbeitete, verbrachte einen Großteil seiner Zeit mit Durchlesen und Suchen in der API und vollführte anschließend eine beeindruckende Anzahl syntaktischer Fehler, bis die gewünschten Klassen und Methoden zufriedenstellend genutzt werden konnten. Semantische Episoden ließen sich bei ihm jedoch nicht finden[20]. Algorithmisch anspruchsvoll waren seine Aufgaben nicht.

Den Studenten des ACM-Programming Contests hingegen, welchen algorithmische Probleme gestellt werden, verbrachten einen großen Teil ihrer Zeit mit der Korrektur ihrer Algorithmen, oftmals durch Benutzung von T&E. Bei diesen Aufgaben war eine korrekte Implementierung beim ersten Anlauf (ein Trial&Success) nur äußerst selten.

Wie von mir erwartet, arbeiteten alle von mir beobachteten Programmierer mit einer modernen Entwicklungsumgebung (überwiegend mit Eclipse).

Es stellte sich heraus, dass ein überaus großer Teil aller begangenen T&E-Phasen der Klasse der syntaktischen Episoden angehören und nur ein kleiner Teil auf die kritischen semantischen Episoden entfällt. Die Dominanz syntaktischer T&E-Episoden hielt ich Vorfeld der Beobachtungen zwar persönlich als sehr wahrscheinlich, aber dass die semantischen T&E-Episoden nur einen so kleinen Teil einnehmen hat mich überrascht. Im Schnitt entfielen auf eine semantische Episode fünf syntaktische Episoden. Als Hauptgrund hierfür mache ich die intensive Nutzung der IDEs mit dem resultierenden reichhaltigen Auftreten der inhibitorischen T&E-Phasen verantwortlich, welche laut Definition in die Klasse der syntaktischen Episoden fallen.

Bevor ein Programm einen semantischen Nutzen entfalten kann, muss meist viel Code geschrieben werden, welcher nur als Fundament für das eigentliche Problem dient und somit im Hinblick auf eine semantische Betrachtung uninteressant ist. In dieser zumeist trivialen und sich oft wiederholenden Codierungsarbeit werden viele der syntaktischen Fehler begangen. So können die semantischen T&E-Episoden erst in einer späten Phase des Codierungsprozesses eintreten, wenn eine gewisse Semantik hergestellt ist, so dass das Programm getestet werden kann[21].

So traten die ersten T&E-Episoden bei den ACM-Contest Testpersonen im Schnitt erst nach der Hälfte der zur Verfügung stehenden Zeit auf. Bis zu diesem Punkt wurde zunächst die „Infrastruktur" für den algorithmisch anspruchsvollen Teil sowie eine erste ungetestete Implementierung der Lösung geschrieben.

Da sich meine Beobachtungen nicht auf große Projekte bezogen haben, kann ich keine Aussagen über die Verteilung der verschiedenen Fehler in einem großen Rahmen machen, wie er in der Berufswelt sehr viel häufiger auftritt. Idealerweise ist ein Projekt aber so gut modularisiert, dass die zu programmierenden Abschnitte klein bleiben und sich somit die Bedingungen ähnlich meinen Beobachtungen verhalten.

---

[20]Es scheint, dass eine gut dokumentierte API viele syntaktische unnötige Fehler – und damit Zeitaufwendung – zu unterbinden vermag. Eine Beschäftigung mit dieser Idee ist jedoch nicht Teil meiner Arbeit..

[21]Für die weitere Forschung auf diesem Gebiet wäre es durchaus interessant zu erfahren, wie hoch die Quote dieser Routine-Codierungsphasen ist (beispielsweise gemessen an der durchschnittlichen Anzahl syntaktischer Fehler pro Zeile), wie viele Prozent des Gesamtcodes auf diese Routinephasen entfällt und ab welchem Zeitpunkt die ersten semantischen T&E-Episoden auftreten.

9. Aufbau von Episoden

Die Beobachtungen führten mich zu dem Schluss, dass es für die Formalisierung der Episoden von Vorteil ist, diese in einem ersten Schritt nicht nach strengen Regeln zu verfassen, sondern eine Episode – beziehungsweise ihren Episodenerkenner - in bestimmte Zustände einzuteilen. Bestimmte Abfolgen von Programmiertätigkeiten überführen eine Episode in einen neuen Zustand, wobei jeweils nur eine bestimmte Menge aller möglichen Programmiertätigkeiten einen Übergang auslösen kann und die zeitliche Reihenfolge der einzelnen Tätigkeiten nur eine untergeordnete Rolle spielt.

Jeder Erkenner kennt folgende Zustände für eine Episode: inaktiv – gestartet – bestätigt – beendet.

Der Mikroprozess nennt solche Programmiertätigkeiten und Abfolgen von Programmiertätigkeiten *Aktivitäten* (auch activity, event oder Tätigkeit).

Die für eine erfolgreiche Identifikation von Episoden zwingend notwendigen *Aktivitäten* nenne ich *Indikatoren*. Ein *Indikator* kann aus einer Abfolge mehrerer *Aktivitäten* zusammengesetzt sein.

Es gibt *Start-*, *Folge-*, *Abbruchindikatoren* sowie *Prä-* und *Nichtindikatoren*.

Je nach augenblicklichem Zustand des Episodenerkenners üben nur bestimmte Indikatoren Einfluss auf den Zustand des Erkenners aus, können also einen Übergang zum nächsten Zustand bewirken, so dass sich mit aufeinander folgenden Zuständen eine Episode ergibt. Als erkannt gilt eine Episode ab dem Zustand *bestätigt*.

Das Wort Indikator habe ich gewählt, da immer erst im Anschluss an das Erkennen einer Episode eindeutig ist, welche Aktivitäten eine Episode geformt haben. Die Aktivitäten sind also nur Indikatoren dafür, dass momentan eventuell eine Episode abläuft. Zum Zeitpunkt des Auftretens der Aktivität kann dieser noch keine Episode zugeordnet werden, da eine Episode insbesondere ein definiertes Ende besitzt, welches zum Zeitpunkt des Auftretens einer Aktivität noch unbekannt ist.

Es ist nur feststellbar, dass sich eine Abfolge von Indikatoren eventuell später zu einer Episode vereinigen wird. Solange eine Episode aufgrund einer nicht ausreichenden Abfolge von Indikatoren noch nicht bestätigt ist, wird solch eine mögliche Episode *Hypothese*[22] genannt.

9.1. Startindikatoren

Der wohl wichtigste *Startindikator* ist das Starten bzw. Ausführen des Programms, welches immer Voraussetzung für Identifikation einer T&E-Episode ist. Zum Zeitpunkt einer Programmausführung enthält das Programm keine syntaktischen Fehler mehr, aufgrund derer der Programmablauf unterbrochen werden könnte. Erst nach einem Testlauf lässt sich das Vorhandensein eines Defekts erkennen, und erst nach dieser Erkenntnis ist überhaupt ein Versuch und Irrtum seitens des Entwicklers notwendig.

Ein wesentliches Charakteristikum eines semantischen Fehlers besteht darin, dass der Programmablauf bei Eintreten eines semantischen Fehlers – also eines Fehlers im Sinne der Spezifikation – meistens nicht unterbrochen wird, sondern trotzdem fortgesetzt wird und das Programm dadurch nicht korrekte Zustände annehmen kann.

Es gibt jedoch Ausnahmen: Ein sehr häufig auftretender semantischer Fehler ist der

[22]Hypothese: Begriff vorgeschlagen und verwendet von Sebastian Jekutsch

Zugriff auf nicht vorhandene Elemente in dynamischen Datenstrukturen zur Laufzeit (wie etwa einen Vektor), welches bei Nicht-Abfangen einen Laufzeitfehler ergibt und das Programm beendet wird. Obwohl derartige Laufzeitfehler zum Abbruch des Programms führen, fallen in meinen Betrachtungen in die Kategorie der semantischen Fehler, da sie zur Kompilationszeit nicht geprüft werden können.

9.2. Folgeindikatoren

In den meisten Fällen besteht eine Episode aus mehr als nur einem Indikator: *Folgeindikatoren* sind alle notwendigen Aktivitäten oder Mengen von Aktivitäten welche auf einen Startindikator folgen, damit eine Hypothese verfolgt wird und sich eine Episode formen kann.

9.3. Abbruchindikatoren

Abbruchindikatoren sind Aktivitäten, welche bei Auftreten die Fortführung einer Hypothese verhindern, den Aufbau einer Episode also unterbrechen. Sollte bei Auftreten eines Abbruchindikators die Hypothese bereits bestätigt sein, ist die Episode erfolgreich abgeschlossen und wird als Episode erkannt. Beispiele für Abbruchindikatoren sind eine bestimmte Zeit der generellen Inaktivität der betrachteten Datei bzw. des betrachten Blocks, welches auf ein Überlegen des Entwicklers hindeutet (oder gar auf seine Abwesenheit), also kein spontanes Trial&Error mehr ist.

9.4. Nichtindikatoren

Aktivitäten, welche keinen Einfluss auf den Aufbau einer Episode haben und parallel zur Episodenformung auftreten, sind die *Nichtindikatoren*. Dies könnte etwa das Einfügen einer Leerzeile oder der kurzzeitige Wechsel in ein anderes Fenster oder anderes Programm sein.

9.5. Präindikatoren

Zwar beginnen Episoden erst mit Eintreten eines Startindikators, jedoch können zur vollständigen Erkennung der Episode auch jene Aktivitäten wesentlich sein, welche kurz vor Beginn des Startindikators auftraten. Das sind die *Präindikatoren*.

Entscheidend für das Erkennen einer Episode sind also drei Punkte: Der Punkt an dem die Hypothese startet, der Punkt an dem eine Hypothese erfolgreich bestätigt wird und der Punkt an dem ein Abbruchindikator auftritt. Treten alle Punkte in dieser Reihenfolge auf, so wurde eine Episode mit Anfang und Ende erfolgreich identifiziert.

10. Fallbeispiel „*VIDEO005*“: Finden der notwendigen Granularität für den Episodenerkenner

Für uns als Menschen lassen sich all diese Indikatoren relativ leicht durch Beobachten und logisches Verknüpfen der Aktivitäten feststellen, wie aber kann ein Algorithmus Indikatoren und Episoden identifizieren?

Einem Erkenner müssen viele Regeln eingespeichert werden, welche er zur Identifikation

möglicher Indikatoren und zur Konstruktion von Hypothesen nutzt. Um diese Regeln auf den Quelltext des Entwicklers anwenden zu können, muss der Algorithmus allerdings über sehr genaue Kenntnis dieses Codes verfügen. Es ist unumgänglich, dass der Erkenner-Algorithmus weiß, mit welchem Codeabschnitt sich der Entwickler momentan beschäftigt, und ob dieser Codeabschnitt in semantischer Abhängigkeit zur begonnenen Hypothese steht.

Dieses Kapitel nimmt sich mit Hilfe eines Beispiels genau dieser Problematik an.

Die folgende Tabelle ist eine Aufschlüsselung des bereits vorgestellten Videobeispiels *VIDEO005 („TYPISCHER T&E-FEHLER")*. Anhand der Tabelle zeige ich die verschiedenen Sichten auf die Codeänderungen und stelle diese Sichten einander gegenüber. Somit lässt sich leicht die Komplexität einer Formalisierung von Episoden aufgrund von Unzulänglichkeiten der zur Verfügung stehenden einsetzbaren Analysemodule begreifbar machen. Ich beginne mit der knappen Darstellung der Sichten in einer Tabelle und lasse eine schriftliche Diskussion folgen.

Schritt Nr.	***Aktivität des Programmierers***	***Art des Indikators***	***Geworfenes Ereignis****	***Details***
1	Copy Zeile x in Block A	Prä-Indikator	-	
2	3 mal Paste Zeile x in Block A	Prä-Indikator	Block A changed	
3	Anpassen der eingefügten Zeilen x in Block A zu x'	Prä-Indikator	Block A changed	An dieser Stelle wurde vergessen, das Eingefügte x vollständig anzupassen
4	Änderung in Block B	Prä-Indikator	Block B changed	Anpassen der Testausgabe
5	Programm starten und testen	Startindikator	run	
6	Versagen erkannt	-	-	Erkannt aufgrund von Testausgabe
7	Debugging: Defekt lokalisieren	-	-	Schneller Erfolg, da zu betrachtender Block sehr klein ist
8	Defektbehebung: Änderung von x' zu x'' in Block A	Folgeindikator	Block A changed	
9	Programm starten und testen	Folgeindikator	run	Richtige Testausgabe, also Ende der Episode
10+x	Zuwendung zu anderen Dingen	Abbruchindiaktor	?	Terminalsymbol -> Ende der Episode erreicht

*) Für die Ereignisse steht ein höheres Detailniveau als nur die des geänderten Blocks zur Verfügung, die Block-Granularität reicht jedoch für die Betrachtungen in diesem Beispiel aus.

Bei den Schritten 1 bis 10 handelt es sich aufgrund der Abgeschlossenheit des betrachteten Problems um eine geschlossene Programmiereinheit, also um eine Episode. Ob diese Episode allerdings auch eine T&E-Episode ist, möchte ich erst am Ende der Diskussion klären.

Der Ausführung eines Programms während der Implementierungsphase gehen natürlich Codeänderungen voraus. Somit ist zwar der Startindikator für diese Episode das Ausführen des Programms, allerdings sind die vorangehenden Änderungen ebenfalls relevant, um ein Grundverständnis für die bei Schritt 5 gestartete Episode aufbringen zu können. Ohne die Betrachtung der Schritte 1 bis 4 wäre die Episode für einen Erkenner-Algorithmus nicht als

Episode identifizierbar, da er lediglich die drei *Ereignisse*[23] „run – Block *A* changed – run" erhalten würde, was viel zu wenig Informationen sind, um adäquate Rückschlüsse auf eine T&E-Episode ziehen zu können. Schritte Eins bis Vier sind deshalb als *Präindikatoren* bezeichnet: Sie lösen zwar nicht den Beginn einer Episode aus, werden aber nach Auftreten des Startindikators in die Betrachtung eingeschlossen.

Ein Trial&Error-Episodenerkenner muss sich also eine bestimmte Anzahl Aktivitäten merken, welche vor Auftreten eines Startindikator auftreten. Das „Merken" manifestiert sich in einem *Puffern* (engl. *caching*) der letzten *x* Modifikationen an den *y* zuletzt veränderten Blöcken. Sind diese Blöcke mit denen identisch, welche nach Eintreten des Startindikators modifiziert werden, ist die Zusammengehörigkeit von Präindikatoren und Folgeindikatoren sehr wahrscheinlich und somit ist die Voraussetzung für eine Hypothese gegeben.

Nicht alle Präindikatoren sind für die Hypothese relevant: Das Erzeugen der Testausgabe in Schritt 4 ist unwichtig, da Block *B* im restlichen Verlauf der Episode nicht mehr berührt wird. Demnach ist Schritt 4 nicht nur ein Präindikator, sondern auch ein Nichtindikator oder ein „Rauschen".

Die sich dem ersten Ausführen anschließende Debugging- und Defektbehebungsphase ist einem menschlichen Beobachter zwar leicht zugänglich, einem Erkenner-Algorithmus jedoch verborgen, da dieser Teil der Episode keinerlei Ereignisse wirft. Eine Maschine kann nicht feststellen, dass ein Programmierer aufgrund einer Testausgabe das falsche Verhalten eines Programms erkannt hat, und eine Maschine kann auch nicht erkennen, dass der Programmierer mittels Sichten des Codes versucht, einen Defekt zu lokalisieren.

In Schritt 8 schließlich behebt der Programmierer den in Schritt 7 lokalisierten Fehler.

Mit dem Schritt 9 wird die Hypothese erfolgreich bestätigt, da nun ausreichend Folgeindikatoren aufgetreten sind. Allerdings ist das Ende der Episode noch nicht erreicht, da bisher kein Abbruchindikator aufgetreten ist. Abbruchindikatoren sind meist die Zuwendung des Entwicklers zu anderen Problemen und damit anderen Blöcken im Quellcode.

Möglicherweise jedoch findet diese Zuwendung im gleichen Bereich des Quellcodes statt, mitunter wird sogar der eben defekt-bereinigte Code erneut verändert. Die Differenzierung der unterschiedlichen Belange wiederum sind für einen menschlichen Beobachter leicht zugänglich, wie aber kann ein Algorithmus diese Zuwendung des Programmierer zu neuen Problemen (insbesondere bei identischer Codestelle) erkennen? Diese Frage löst ein Punkt des nächsten Kapitels.

Bisher blieb offen, ob die begonnene Episode (oder bestätigte Hypothese) eine T&E-Episode ist oder nicht. Es gab zwar nur eine einzige Änderung des relevanten Blocks *A* (in Schritt 8) und damit keinen sicheren Hinweis auf ein mehrmaliges Probieren, dennoch wurde diese eine Änderung aus Schritt 8 erneut getestet, und somit einem weiteren Versuch (*trial*) unterzogen, was auf Unsicherheit des Programmierers an seiner Änderung schließen lassen kann. Dieser letzte Schritt 9 ist dann auch für mich der finale Folgeindikator, um die Schritte 1 bis 9 als den Beginn einer T&E-Episode zu deuten, also die Hypothese zu bestätigen.

[23]Der Ausdruck *Ereignis* kennzeichnet eine Aktivität des Entwicklers, so wie sie von einer Maschine interpretiert wird.

11. Vom Suchen und Finden der Indikatoren

Ein großes Problem meiner Arbeit lag in der beschränkten Aussagekraft der mir zur Verfügung stehenden Informationen über die Codeänderungen begründet, so dass sich Indikatoren nur sehr schwer finden ließen: Die vom ElectroCodeoGram übertragenden Ereignisse sind zwar sehr zahlreich, bieten in ihrer jetzigen Form aber zu wenig Informationen, um die in der Theorie formulierbaren Episoden in ihrer ganzen Komplexität auf ein Computermodell übertragen zu können. Um Indikatoren erkennen zu können, müssen die eingehenden Ereignisse in geeigneter Weise gefiltert und anschließend kombiniert werden, so dass sie zu neuen aussagekräftigeren Ereignissen zusammen geschlossen werden können. Insbesondere müssen sich Ereignisse auf eindeutige Positionen im Quellcode beziehen lassen. Ein Ereignis wie „file *x* changed" ist viel zu grob und nicht hilfreich.

Glücklicherweise stand ein von Sebastian Jekutsch programmierter Algorithmus zur Verfügung, welcher dynamische Codeänderungen auf Block-Niveau nachvollzieht. In einem Abschnitt aus dem Praxisteil dieser Arbeit werde ich dieses Modul namens *Stellenverfolger* genauer erläutern.

Die mir zur Verfügung stehende Granularität der Informationen ist also auf einer Block-Ebene angesiedelt. Ein *Block* ist eine semantische oder syntaktische Einheit wie etwa eine Methode, eine for-Schleife, eine Bedingung oder eine Zuweisung. Je nach Art des Blocks müssen andere Regeln der Formalisierungen angewendet werden. Ein Block besteht aus Zeilen, wobei im Allgemeinen jede Zeile jeweils eine Anweisung oder Zuweisung enthält, aber nicht muss. Zeilen sind keine syntaktischen Konstrukte, sondern erleichtern dem menschlichen Entwickler lediglich die Übersicht über den Code. Ins Extrem getrieben ist es durchaus möglich, den Code eines gesamten Programms in nur einer entsprechend langen Zeile zu schreiben. Zeilen sind deshalb nur beschränkt zuverlässige Anhaltspunkte für die Analyse von Code. Dennoch werden Zeilenbetrachtungen aufgrund der leichteren Analyse in Erkennern eingesetzt, so auch im Stellenverfolger.

Die Block-Ebene ist jedoch noch zu ungenau, um exakte Änderungen des Codes nachvollziehen zu können und erreicht das notwendige Detailniveau auf Zeichenebene nicht. Das Block-Niveau kann lediglich Aussagen darüber treffen, ob eine Block geändert wurde, jedoch nicht die exakte Modifikation (analog zum Entscheidungsproblem der theoretischen Informatik).

Zeichen-Niveau bedeutet, dass Informationen darüber zur Verfügung stehen, welches die konkreten Änderungen einer Codestelle über die Zeit hinweg sind (analog zum Optimierungsproblem der theoretischen Informatik), so zum Beispiel: *„Um 12:34h und 42 Sekunden wurde der Wert der Variablen* dummy *von* 23 *auf* 5 *verändert."*

In der Evolution eines Programms ändern sich Zeichen, Zeilen und Blöcke, sowohl in ihrem Inhalt als auch in ihrer Position. Zeilen werden eingefügt, kopiert, verschoben oder neu generiert. Ein über die Zeit hinweg betrachteter Block wird *Stelle* (engl. *location*)[24] genannt.

---

[24]Stelle: Begriff vorgeschlagen und verwendet von Sebastian Jekutsch

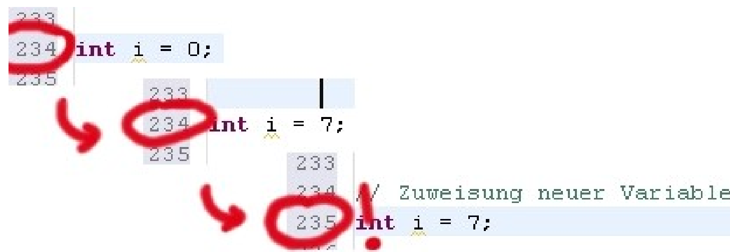

Abbildung 4: Beispiel: dynamische Zeilenverfolgung

Eine Analyse der Evolution des Programmcodes über die Zeit ist schwer zu erreichen, da nicht die statischen Blöcke selbst betrachtet werden dürfen – beispielsweise immer nur die Zeile 234 des Programms – sondern vielmehr die Evolution jeder Zeile seit Entstehen dynamisch verfolgt werden muss. Bei Neuentstehen der Zeile 234, was etwa die Zuweisung *„int i= 0"* sein könnte, muss diese Zeile indiziert werden (nennen wir sie Zeile *y*) und anschließend verfolgt werden können, auch wenn sie nicht mehr die hart kodierte Zeile 234 ist. Durch Einfügen eines Kommentars oberhalb der 234 verschiebt sich die Position der betrachteten Zeile *y* natürlich um Eins nach hinten, weshalb *y* jetzt nicht mehr in Zeile 234 steht, sondern in Zeile 235.

In der Praxis treten natürlich sehr viel kompliziertere Fälle auf, weshalb die Anwendung eines simplen Diff-Algorithmus' hier nicht ausreicht. Der Stellenverfolger nimmt sich jedoch auch dieses Problems an und kann Stellenänderungen über die Zeit hinweg identifizieren.

Dass die Granularität des Zeilen-Niveaus zur exakten Bestimmung der Episoden überhaupt notwendig ist, verdeutlicht folgendes einfaches Beispiel: Zum Erkennen einer T&E-Episode kann relevant sein, wie häufig sich die Bedingungen in einem Schleifenkopf ändern (siehe *VIDEO007*). Da sich eine Schleife selbst immer in einem Block befindet, wäre auf Block-Niveau nur feststellbar, dass sich etwas im Block verändert hat, nicht jedoch die greifbare Ausprägung der Änderung. Ob es nun die besagte Änderung des Schleifenkopfs oder eine unwichtige Änderung im Schleifenrumpf war, ließe sich nicht feststellen.

Selbst wenn das Ereignis die Information enthalten würde, in welcher Zeile des Blocks sich eine Änderung ereignet hat, so ist diese Information ungenügend, da beispielsweise ein besagtes Einfügen eines Kommentars alles verschiebt.

12. Verlegenheitsänderungen und Semantische Codeabhängigkeiten

Sind sich Entwickler in einer Situation unsicher und möchten ihr Programm einfach „zum Laufen bekommen" ohne jedoch eine konkrete Problemlösungsstrategie zu haben, ändern sie an vielen Stellen den Code ab, und seien es nur eine kleine „Verlegenheitsänderungen" wie etwa das Einfügen einer Leerzeile oder die Umbenennung einer Variablen „für die bessere Übersicht"[25]. Diese Änderungen tragen nicht zur Lösung des Problems bei und das unnötige Auftreten dieser Nichtindikatoren lässt gut die Hilflosigkeit des Programmierers erahnen und erinnert beinahe an das biologische Muster der Übersprungshandlungen (siehe Videobeispiel *VIDEO008*). Im Beispiel ersetzt der Entwickler nach einem Testlauf ein „`==`" durch ein „`!=`" und testet erneut. Da der Defekt weiterhin auftritt, überfliegt er kurz den Code, löscht eine Leerzeile (Verlegenheitsänderung), stellt anschließend das ersetzte „`!=`" durch das ursprüngliche „`==`" wieder her und testet erneut. Dieses erneute Testen ist natürlich unnötig, da das Programm nun wieder seinen

[25]Zitat aus einer Beobachtung

falschen Originalzustand besitzt.

Die Verlegenheitsänderungen treten oftmals nicht nur im Prozess der Fehlerbehebung selbst auf, sondern auch im Prozess der Fehlerlokalisierung. Diese kann ein durchaus komplexer Prozess sein: Oftmals existieren Codeabhängigkeiten zwischen den einzelnen Blöcken des Codes, so dass die Lokalisierung semantischer Fehler oftmals schwer ist.

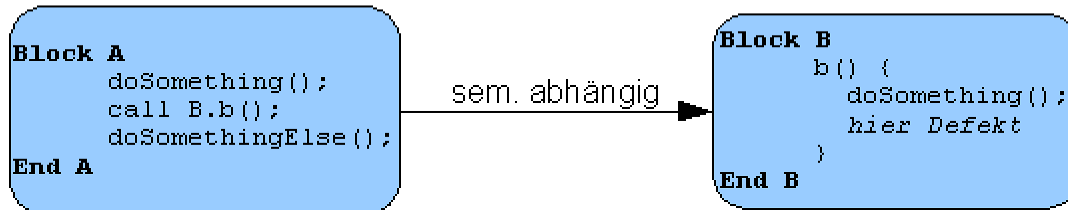

Abbildung 5: semantische Codeabhängigkeit existiert

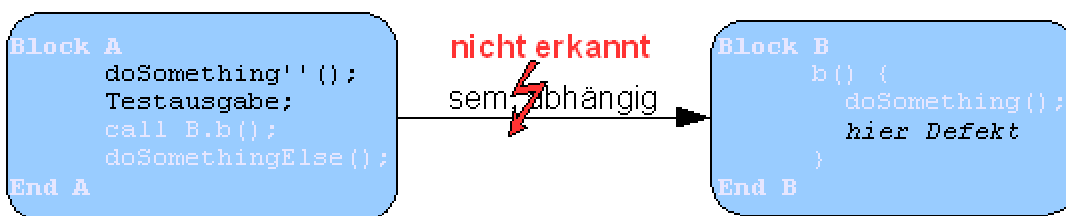

Abbildung 6: falsches Debugging: sem. Codeabhängigkeit nicht erkannt (grau: nicht modifiziert)

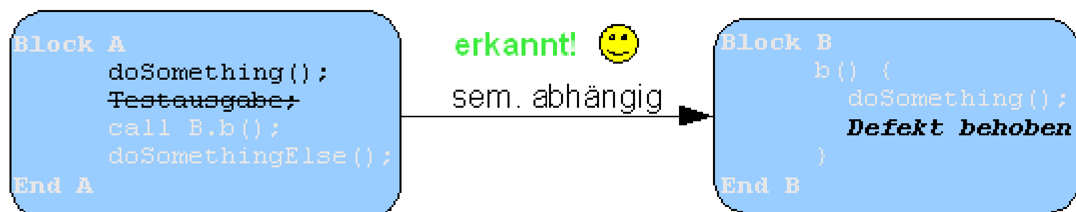

Abbildung 7: korrektes Debugging und Fehlerkorrektur der sem. Abhängigkeit

Wie Abbildungen 5 bis 7 zeigen sollen, sucht der Programmierer beispielsweise den Defekt in Codeabschnitt *A* und modifiziert *A* mehrfach, obgleich sich der Defekt in Methode *b* des Codeabschnitts *B* befindet, welche von *A* aufgerufen wird.

Vor der ersten Modifikation von *A* fand ein Testlauf mit anschließender falscher Defektlokalisierung statt. Die Modifikationen in *A* können nur Trial&Error sein, da Änderungen in Block *A* den Fehler niemals beheben können. Erst nach diesen Änderungen inklusive Testen der Änderungen in *A* wird der Defekt in Block *B* lokalisiert und dort behoben.

Ein Spezialfall der semantischen Codeabhängigkeiten ist eine Mischung aus sich überlappenden Defektlokalisierungs- und Trial&Error-Tätigkeiten.

Die Episode besteht aus zwei abhängigen Teilen, wobei sich jeweils ein Teil auf die Tätigkeiten in Block *A* und Block *B* bezieht. Das ist insbesondere dann problematisch, wenn die Veränderung in *B* alleine nicht als eine Episode erkennbar ist, sondern nur in Kombination mit den Änderungen in *A* eine T&E-Episode ergibt. Andererseits können die Änderungen in *A* alleine noch keine Episode ergeben, da die zugehörigen notwendigen Folgeindikatoren erst in Block B auftreten. Kommen zusätzlich die oben beschriebenen

Verlegenheitsänderungen hinzu, welche möglicherweise in den weiteren Blöcken *C* oder *D* liegen, so stößt ein automatischer Episodenerkenner schnell an die Grenze des technisch Analysierbaren. Der rein phänomenologisch arbeitende Algorithmus des Erkenners, kann Folgeindikatoren und Nichtindikatoren nicht mehr auseinander halten, da er den Änderungen der diversen Stellen keinen Blockabhängigkeiten zuordnen kann.

Ich mache deshalb folgende stark vereinfachende Annahme, welche sich später in Regeln für den Erkenner wiederfinden wird: Das Erkennen von T&E-Episoden ist losgelöst von semantischen Codeabhängigkeiten. Ich hoffe darauf, dass die falschen Änderungen im zuerst betrachteten Block so gravierend sind, dass sie eine eigenständige Trial&Error-Hypothese auslösen. Andernfalls wird keine Episode erkannt.

13. Finden von Abbruchindikatoren

Ein weiteres Problem semantischer Codeabhängigkeiten tritt auf, wenn ein Abbruchindikator für eine bereits bestätigte Hypothese erkannt werden soll, also das Abschließen der Episode. In meinen Beobachtungen waren die Episoden-Enden stets eine Zuwendung des Programmierers zu neuen Problemen, da er das alte soeben gelöst hat. Meistens befinden sich diese Probleme in anderen Stellen des Codes, manchmal jedoch wird ein und dieselbe Stelle für die Lösung des neuen Problems weiter bearbeitet, mitunter wird der soeben geänderte Code sogar erneut modifiziert. Im ersten Fall ist die Identifizierung des Abbruchindikators sehr leicht: Tritt eine bestimmte Zeit der Inaktivität in einer Stelle auf, so wird der sich auf diese Stelle beziehende Erkenner abgebrochen.

Ein weiterer Abbruchindikator ist ein bestimmter maximaler Zeitraum ohne Ausführen des Programms. Dieser Abbruchindikator wird im zweiten Falle der identischen Codestelle eingesetzt.

14. Fallbeispiel: *„Video006"*: Technische Fokussierung auf den Episodenerkenner

Nachdem auf den letzten Seiten einige der notwendigen Informationen für das Verständnis von Trial&Error-Episoden sowie deren charakteristischen Eigenschaften erläutert wurden, werden mit diesem Fallbeispiel zwei weitere Punkte eingebracht sowie die oben beschriebenen Erkenntnisse in diesem Beispiel angewandt.

Zum Ersten wird anhand dieses Beispiels verdeutlicht, ab wann sich eine Anzahl Indikatoren zu einer T&E-Episode verbindet und zum Zweiten wird die Problematik der technischen Umsetzung eines Episodenerkenners vertieft.

Wie im obigen Beispiel wird zuerst eine Tabelle der verschiedenen auftretenden Tätigkeiten angegeben, gefolgt von einer schriftlichen Diskussion. Die Aufstellung der Tabelle bildet den Blick der menschlichen Sichtweise auf den Code dar und nicht den eines Erkenner-Algorithmus'.

Schritt Nr.	*Aktivität des Programmierers*	*Art des Indikators*	*Geworfenes Ereignis**	*Details*
1	byte1 in byte0 umgewandelt	Prä-Indikator	Block A changed	Änderung unabsichtlich u .falsch; Ist Auslöser für Folgeaktivitäten
2	Einen Wert von „.255“ auf „.254“ geändert	Prä-Indikator	Block A changed	war ursprünglich geplante Änderung; Schritt 1 war unnötig
3	Testlauf #1	Start-Indikator	run	Testlauf ergibt Defekt
4	Wert von „.254“ wieder auf „.255“ geändert	Folge-Indikator	Block A changed	
5	Testlauf #2	Folge-Indikator	run	Testlauf ergibt Defekt
6	1. Wechsel zu Input-Datei	Nicht-Indikator	File-Focus Lost	Prüfen, ob wirklich falsch
7	Testausgaben einfügen	Nicht-Indikator	Block B changed	Testausgabe nicht in Block A
8	Testlauf #3	Folge-Indikator	run	Testlauf ergibt Defekt
9	2. Wechsel zu Input-Datei	Nicht-Indikator	File-Focus Lost	prüfen, ob Output wirklich falsch
10	Vier Testausgaben einfügen	Folge-Indikator	Block A changed	Änderung diesmal in Block A
11	Testlauf #4	Folge-Indikator	run	Testlauf ergibt Defekt
12	Debugging: Kontrollieren der Testausgaben	-	- bzw. File-Focus Lost	auch Kontrolle mittels des Windows-Calculators
13	Ändern der Testausgaben	Folge-Indikator	Block A changed	Problem eingegrenzt
14	Testlauf #5	Folge-Indikator	run	Testlauf ergibt Defekt
15	relativ ausführliches Debugging	-	-	inklusive anderer Methoden
16	Zuweisung von Variablen zu „0“	Nicht-Indikator	Block C changed	unnötige Verlegenheitsänderung
17	hastige Codedurchsicht	-	-	
18	neue Änderungen in Block A	Folge-Indikator	Block A changed	plus: alte Testausgaben entfernt
19	Testlauf #6	Folge-Indikator	run	Testlauf ergibt Defekt
20	Debugging: erneute hastige Codedurchsicht	-	-	keine neuen Erkenntnisse
21	eine weitere Zuweisungsänderung	Folge-Indikator	Block A changed	plus: die alten Testausgaben wieder eingefügt
22	Testlauf #7	Folge-Indikator	run	Testlauf ergibt Defekt
23	Debugging: Prüfen von Testausgaben u Code	-	-	Defekt entdeckt
24	Umwandeln von byte0 in byte1	Folge-Indikator	Block A changed	Defekt entfernt
25	Diverse Änderungen	Nicht-Indikator		Änderungen sind wohl überlegt
26	Testlauf #8	Folge-Indikator	run	Testlauf ergibt Defekt

*) wiederum hohe Abstraktionsstufe; Die tatsächlich geworfenen Ereignisse sind exakter, hier aber nicht von Belang.
(Ich habe mich bemüht, die Tabelle so klein wie möglich zu halten. Allerdings hat die Episode eine Länge von neun Minuten!)

Bereits auf den ersten Blick fällt die große Anzahl an Testläufen auf. Es stellt sich unweigerlich die Frage, ob eine so große Anzahl von Testläufen notwendig ist, um eine einzige T&E-Episode zu erkennen. Dies ist nicht der Fall: Spätestens nach dem vierten Testlauf (Schritt 11) ist eine Trial&Error-Hypothese bestätigt. Alle anderen Ereignisse

außer Acht lassend, wurde Block *A* häufig genug modifiziert und ist von vielen Testläufen durchsetzt, um eine abgeschlossene Hypothese zu rechtfertigen.

Erst die vollständige Abarbeitung aller 26 Schritte löst das in Schritt 1 aufgetretene semantische Problem. Wiederum ist dies für Menschen leicht erkennbar, für eine Maschine nicht. Eine Herausforderung in der Konstruktion eines geeigneten Episodenerkenners besteht nun darin, den Erkenner nach erfolgreicher Identifizierung einer Episode (z.B. nach besagtem Schritt 11) nicht in den verbleibenden Schritten nach neuen Episoden suchen zu lassen, sondern die Folgeschritte als Teil der ersten erkannten Episode zu sehen. Zweifellos lassen sich in den Schritten 12 bis 26 viele andere Trial&Error-Episoden finden, auch wenn diese Teil der in Schritt 11 erkannten Episode sind.

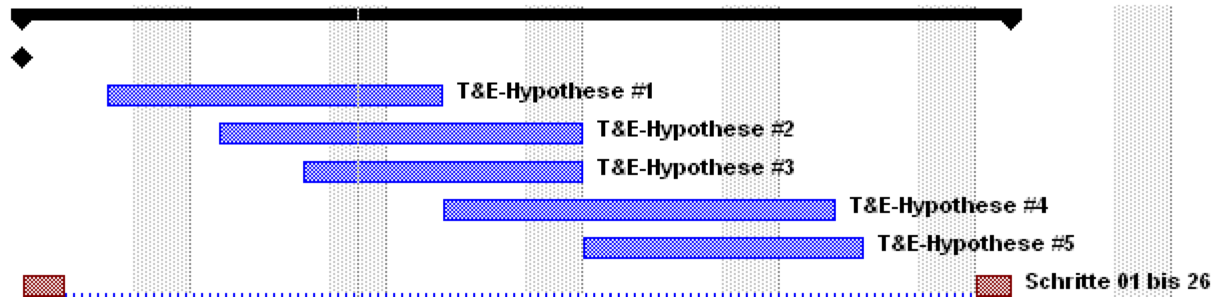

Abbildung 8: Beispiel: sich überlagernde Hypothesen

Wie aus Abbildung 8 ersichtlich wird, startet nach Eintreffen jedes neuen Ereignisses ein neuer Erkenner, so dass mehrere parallel arbeitende Hypothesen gebildet werden. Jeder blaue Balken kennzeichnet eine erfolgreich validierte (bestätigte) Hypothese. Das Ende jedes blauen Balkens kennzeichnet also den Zeitpunkt des sicheren Erkennens einer Episode.

Schritt 26 markiert das Episoden-Ende jedes einzelnen der 5 laufenden Erkenner. Bei acht Testläufen – also acht potenziellen Start-Indikatoren – formen die Erkenner fünf verschiedene Episoden, deren Beginn sich gegenseitig überlappt. Obwohl alle gefundenen T&E-Episoden formal korrekt sind, ist deren Auftreten nicht wünschenswert, da die zusätzlich erkannten Episoden (alle Episoden nach Auftreten von *T&E-Hypothese #1*) Teil der ersten erkannten Hypothese *T&E-Hypothese #1* sind.

Zwei einfache aber effiziente Möglichkeiten zur Vermeidung dieser multiplen Erkennung sind zum einen das generelle Unterbinden einer in Block *A* entstehenden neuen T&E-Hypothese für die nächsten *x* eintreffenden Ereignisse, oder zum anderen eine erneute Erkennerphase erst nach Auftreten eines Abbruchindikators zu erlauben, etwa einer Inaktivität in Block *A* von *y* Sekunden. Im Beispiel von Abbildung 8 würde mit diesen Techniken nur noch *T&E-Episode #1* erkannt werden und unter Umständen *T&E-Episode #5*, da diese weit genug nach der Identifikation von *T&E-Episode #1* auftritt.

Da beide Techniken in der Praxis leider nur sehr schwer umsetzbar sind, wird eine dritte Methode angewendet, um Episodenüberlappungen zu handhaben: Im Anschluss an jeden Abbruchindikator wird ein weiterer Erkenner gestartet, welcher sich die letzten T&E-Episoden in einem gewissen Zeitfenster anschaut und diese aufgrund ihrer Zugehörigkeit zur gleichen Stelle (location) zu einer großen Episode verschmilzt (joinen, mergen).

Selbstverständlich werden jene Indikatoren weiterhin betrachtet, welche sich auf einem anderen Bereich als Block *A* und damit auf eine andere semantische Einheit beziehen.

Wenden wir uns nun einem anderen Aspekt des Beispiels zu, der Klassifizierung der Aktivitäten als Indikatoren. Wie bereits oben im Kapitel 12 zu semantischen Codeabhängigkeiten erläutert, werden alle die Episode nicht betreffenden Stellen ausgeblendet und Aktivitäten in diesen Stellen werden als Nichtindikatoren gedeutet. Welche Stelle für eine mögliche begonnene Episode relevant ist, lässt sich an ihrer Änderungshäufigkeit erkennen und idealerweise an der Zugehörigkeit der *Testausgaben* zu dieser Stelle.

Das Erkennen der Zugehörigkeit von Testausgaben zu einem Block ist nur in der Theorie vorstellbar. Ich verfolge es nicht weiter, weil in der Praxis zu vielfältige Formen von Testausgaben existieren, als dass sie alle identifiziert und ausgewertet werden könnten: Nicht immer werden die Ausgaben in eine Konsole geschrieben. Häufig werden sie auf einem GUI dargestellt, finden sich als HTML-formatiertes Konstrukt wieder oder sind Getter/Setter eines komplexen Darstellungsobjekts. Des Weiteren sind Testausgaben für meine formalisierten Erkenner nur sichtbar, wenn sie innerhalb der IDE auftreten, das Programm also von der IDE kompiliert und ausgeführt wird, was nicht immer der Fall ist (man denke etwa an dynamische Webseitengestaltung mit JavaServerPages).

Eine weniger komplexe und in meinen Augen wesentlich besser geeignete Methode zur Bestimmung der Indikatoren bietet das oben beschriebene Caching der letzten *x* Änderungen: Betrachten wir den Startindikator „Testlauf Vier in Schritt 11“ und gehen von einer Pufferung der letzten zehn Events aus, so ergeben sich vier vergangene Änderungen in Block *A*, eine Änderung in Block *B* sowie drei „Programm Ausführen“-Ereignisse, was aufgrund des Schwerpunkts aller Änderungen in Block *A* einen Fokus auf Block *A* rechtfertigt. Eine genauere Analyse der Änderungen in Stelle *A* ergibt, dass sich lediglich zwei Zeilen in *A* geändert haben und zwar mehrfach. Dies sind ausreichende Informationen, um die T&E-Hypothese zu bestätigen. Die Änderung in Block *B* und die kurzzeitige Betrachtung einer anderen Datei werden als Nichtindikatoren eingestuft.

In diesem Fall ist interessant, dass aufgrund der gepufferten Änderungen in *A* mit den vielen Testläufen bereits die Betrachtung des Startindikators genügt, um eine Episode erkennen und eine Ausgabe generieren zu können.

Teil C) Technische Grundlagen für das Formalisieren der Trial&Error-Episoden

Der theoretisch angelegte Teil *B* hat eine Einführung in die Welt der Episoden gegeben sowie Modelle zum Auffinden von Episoden erläutert. Zusätzlich wurden bereits einige technische Einschränkungen bezüglich der Realisierung eines automatischen Episodenerkenners vorgestellt.

Der nun folgende Abschnitt *C* beschäftigt sich mit der Umsetzung und Adaption der aus Teil *B* gewonnenen Kenntnisse auf die technische Ebene und geht detailliert auf die zur Verfügung stehenden Techniken ein.

Das Ergebnis dieser Arbeit ist keine konkrete Implementierung eines Episodenerkenners in

einer bestimmten Programmiersprache, sondern eine Spezifikation bzw. *Formalisierung* der Episoden in einer speziellen Syntax namens *ALED*. In ALED formalisierte Episoden werden von einem geeigneten Programm - wie dem von Christian Kopf vorgeschlagenen *ALED-Rahmenwerk* – direkt ausgeführt. Letztendlich verfasse ich eine interpretierbare Syntax. Ein großer Vorteil dieser Vorgehensweise ist die abstrakte und weitgehend generische Spezifikation der Episoden bei gleichzeitiger Möglichkeit des Ausführens der Formalisierungen.

Der von mir häufig verwendete Ausdruck des *Episodenerkenners* ist nun auf technischer Ebene betrachtet kein einzelnes Modul oder Programm, sondern die Gesamtheit aller an der Erkennung von Episoden beteiligten Prozesse, Module und Programme, wie sie der nächste Abschnitt erläutert.

1. Beschreibung: Arbeitsablauf und Kommunikationswege des Episodenerkenners

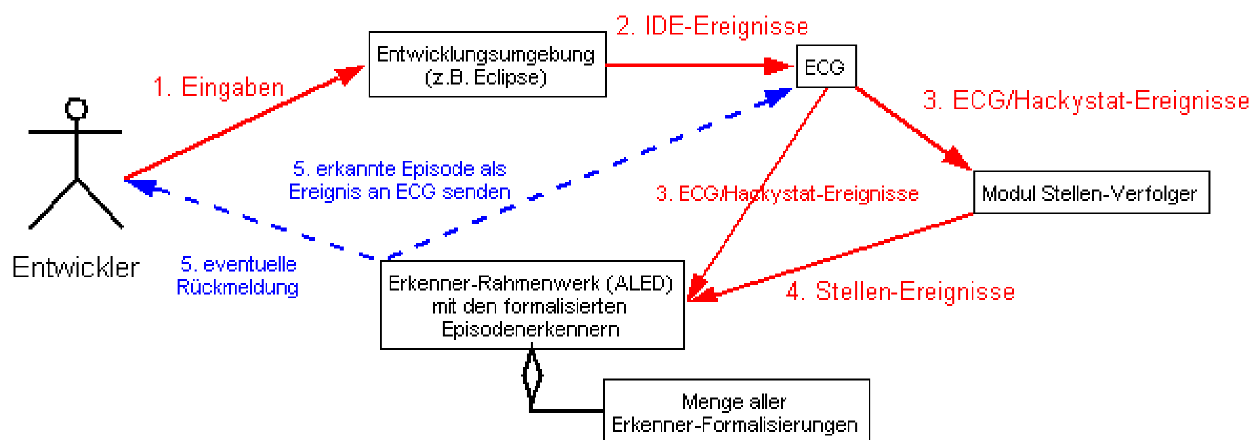

Abbildung 9: Arbeitsablauf des Episodenerkenners

Zunächst erläutere ich den groben Arbeitsablauf des Episodenerkenners anhand von Abbildung 9 und beschreibe anschließend die einzelnen Komponenten des Erkenners.

Der *Arbeitsablauf* (oder englisch *workflow*) beginnt bei dem Entwickler, welcher Eingaben in seiner Entwicklungsumgebung tätigt. Die IDE protokolliert diese Tätigkeiten zu einem gewissen Grad und übergibt diese in einem Datenstrom als so genannte *Ereignisse* an das *ElectroCodeoGram* (*ECG*). Das ECG ist ein Plug-In für die Entwicklungsumgebung, welche die Ereignisse auf eine bestimmte Weise aufbereitet und an Erkenner weiterleitet, in diesem Falle an das Stellenverfolger-Modul und das ALED-Rahmenwerk.

Der Stellenverfolger bereitet die Ereignisse wiederum auf und sendet seine Informationen direkt an das Erkenner-Rahmenwerk. Meine Formalisierungen – also die Regeln für alle T&E-Episoden - sind in der Sprache ALED definiert. Das ALED-Rahmenwerk schließlich prüft im Ereignis-Strom anhand der spezifizierten Episoden-Regeln, ob eine Episode aufgetreten ist.

Erkannte Episoden werden als neue Ereignisse auf den Ereignisstrom geschrieben, wo die erkannte Episoden anderen Erkennern (und insbesondere ALED selbst) zur weiteren Verarbeitung zur Verfügung steht.

Anschließend gibt es mehrere Möglichkeiten der weiteren Kommunikation der Ausgabe-Ereignisse des ALED-Rahmenwerks. So könnte der Entwickler mittels eines Dialogfensters

auf die eben aufgetretene Episode aufmerksam gemacht werden. Diese zusätzlichen Verarbeitungsschritte finden in dieser Arbeit allerdings keine Beachtung.

Das Erkennen und anschließende Übersenden der neuen Episoden-Ereignisse an den Ereignisstrom erfolgt in Echtzeit.

In dieser Arbeit werden primär zwei Klassen von Ereignissen betrachtet: Erstens die *run*-Ereignisse, welche ein Ausführen und Testen des Programms repräsentieren und zweitens die *codechange*-Ereignisse, welches die Ereignismenge aller den Code manipulierenden und verändernden Ereignisse darstellt.

2. Erläuterung der ECG-Ereignis-Handhabung und Bedeutung für die Arbeit

Moderne IDEs verwenden für die interne und externe Kommunikation oft ereignisbasierte Architekturen. Es ist möglich, diesen Ereignisströme „anzuzapfen" und auszulesen, was die Idee hinter dem *ElectroCodeoGram* ist: Die im ECG eintreffenden Ereignisse werden in ein Hackystat[26]-kompatibles XML-Format umgewandelt.

Das ECG schreibt den Ereignisstrom entweder direkt in eine Textdatei bzw. Datenbank oder verarbeitet ihn weiter: Mit Hilfe der so genannten ECG-Module (das sind Eclipse-Plug-ins) werden „on-the-fly" IDE-Ereignisse ausgewertet, kombiniert und zu neuen Ereignissen zusammengefasst, welche sofort auf den Ereignisstrom geschrieben werden.

Diese neuen Ereignisse sind beispielsweise Informationen zu neu erkannten T&E-Episoden.

Das ECG ist zwar speziell für die Eclipse-Entwicklungsumgebung erstellt worden, aufgrund der allgemein lesbaren XML-Konstrukte können mit dem ECG vergleichbare Programme aber Ereignisse beliebiger IDEs als Hackystat-kompatible XML-Ereignisse aufbereiten und anschließend durch die weiter unten beschriebenen Erkenner weiterverarbeitet werden.

Die folgende Abbildung zeigt den schematischen Aufbau eines vom ECG in XML umgewandelten Eclipse-Ereignisses:

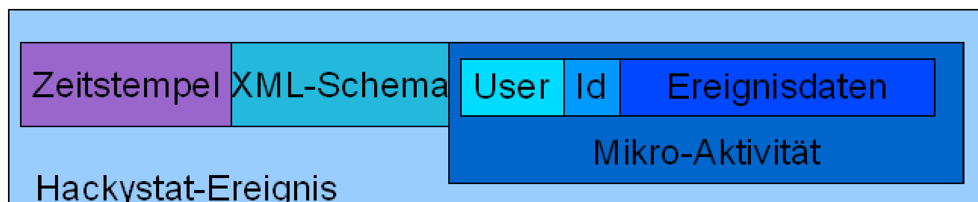

Abbildung 10: schematische Darstellung eines im ECG verwendeten Hackystat-Ereignisses

3. Erläuterung des Stellenverfolger-Moduls und Bedeutung für die Arbeit

Ein von mir genutzter ECG-Sensor ist der Stellenverfolger von Sebastian Jekutsch.

Ich habe den Ist-Status der Entwicklung des Stellenverfolgers zu meinen Gunsten in einen Soll-Status verändert. Nicht jede Eigenschaft die ich im Folgenden beschreibe oder verwende ist bereits implementiert. Da sich der Stellenverfolger jedoch noch in einem frühen Zustand seiner Entwicklung befindet bin ich zuversichtlich, dass meine hier geforderten Eigenschaften in Zukunft implementiert werden. Soweit es mir möglich ist, verwende ich die von Sebastian Jekutsch vorgeschlagene Syntax für den Stellenverfolger

[26]Hackystat: http://hackydev.ics.hawaii.edu/hackyDevSite/home.do vom 21.08.2006

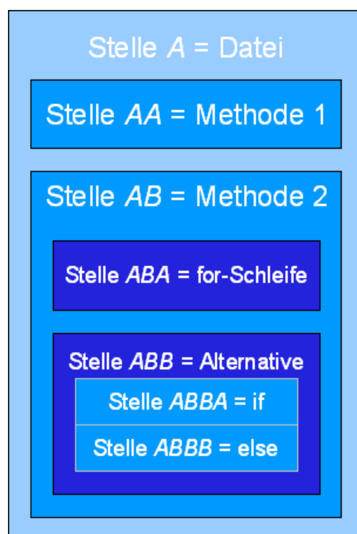

Abbildung 11: Hierarchie von Stellen

Eine Stelle ist entweder ein syntaktisches Konstrukt wie eine Klasse, Methode oder Schleife oder eine leicht erkennbare semantische Zusammengehörigkeit wie eine Liste von Variablendeklarationen.

Der Algorithmus identifiziert *Stellen* im Quellcode und verfolgt die Modifikationen dieser Stellen über die Zeit hinweg. Sollte eine Stelle zu komplex werden, so wird die Stelle in weitere Unterstellen unterteilt. Auf diese Weise entsteht eine baumartige Hierarchie von Stellen.

Der Baum besitzt als *Wurzel* die betrachtete Datei, welches das gröbste Niveau darstellt[27]. Anschließend folgen die einzelnen *Äste* (oder *Zwischenstellen*), welche wiederum Unterstellen in sich vereinen können (siehe Abbildung 12). Die feinste Stufe, welche selbst keine Unterstellen mehr enthält, werden *Blätter* genannt.

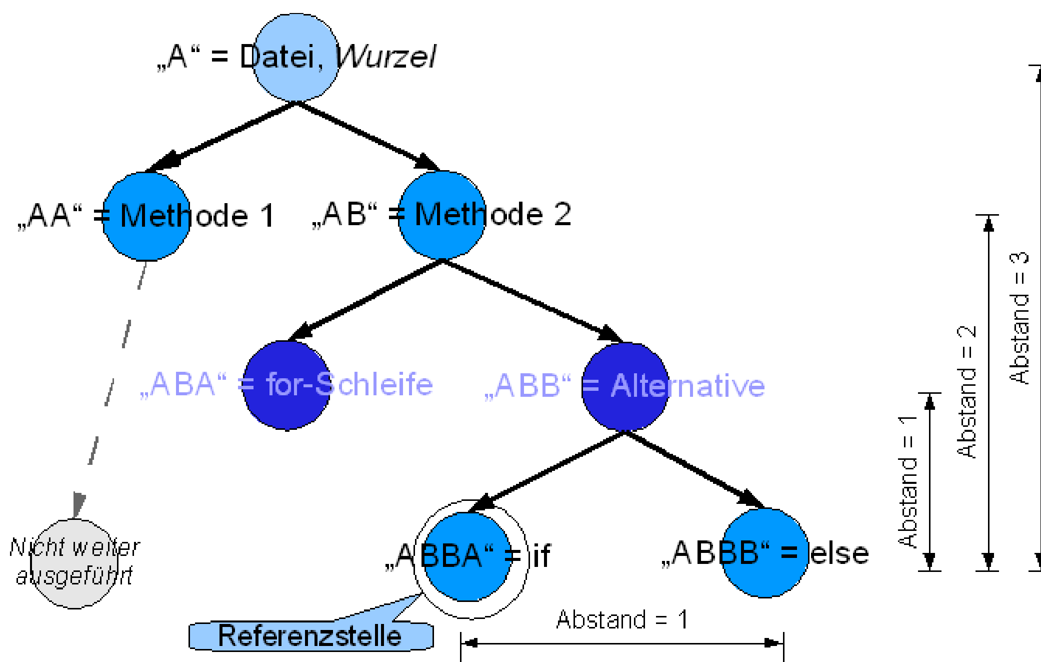

Abbildung 12: Hierarchie von Stellen als Baum; Nur die rechte Seite wurde vollständig ausgeführt

Zwischenstellen bestehen aus zwei Teilen: Der *Kopf* (*head*) enthält genaue Informationen über den Typ der Stelle (wie etwa die Schleifenparameter des Typs for-Schleife) und der *Rumpf* (*body*) enthält die weiteren Unterstellen (wie etwa weitere Unteräste oder ein Blatt). Der eigentliche Quellcode – also die Anweisungen und die Zuweisungen – steht ausschließlich in Blättern.

Jede Stelle besitzt als Parameter eine eindeutige Identifikation (*id*), eine eindeutige Position innerhalb der Hierarchie (repräsentiert als verkettete Liste), einen eindeutigen Typ sowie die Art der letzten Codeänderung.

[27]Es ist auch möglich, das aktuelle Projekt als Wurzel zu betrachten und die einzelnen Dateien als Unterstrukturen. Dieses habe ich nicht getan, da seine Auswertung die Komplexität steigert und T&E-Episoden über ein gesamtes Projekt hinweg schwer zu identifizieren sind.

Bei jeder Änderung einer Stelle werden neue Ereignisse generiert und auf den Ereignisstrom geschrieben. Die Ereignisse werden dabei die Hierarchie-Ebene des Stellen-Baums hinauf gereicht und lösen in den höheren Ebenen ebenfalls Ereignisse aus, da sich durch eine Änderung in einer Unterstelle auch der Zustand der umschließenden Stellen ändert.

Wird etwa ein Blatt verändert, so werden *x* Ereignisse bei *x* Ebenen im Baum geworfen.

3.1. Verwandtschaft von Codestellen

Wie die Baumstruktur der Hierarchie bereits nahe legt, sind Codestellen miteinander verwandt. Je „näher" sich eine Codestelle der Anderen befindet, desto höher ist der Grad der Verwandtschaft.

Es gibt stets eine Referenzstelle, von der aus gesehen der Verwandtschaftsgrad zu anderen Stellen gemessen wird (entspricht einem *single-source-multiple-path*-Verfahren). Den Grad der Verwandtschaft (die Nähe zueinander) definiere ich als Anzahl der zu durchlaufenden Zwischenknoten, bis der erste gemeinsame Vaterknoten von Referenzstelle und Zielstelle erreicht ist (siehe Abbildung 13).

Eine Stelle besitzt zu sich selbst einen Abstand der Größe 0. Die engste mögliche Verwandtschaft mit Abstand 1 zwischen zwei Stellen sind zwei Geschwisterblätter, also Blätter mit exakt demselben Vaterknoten (Vaterstelle).

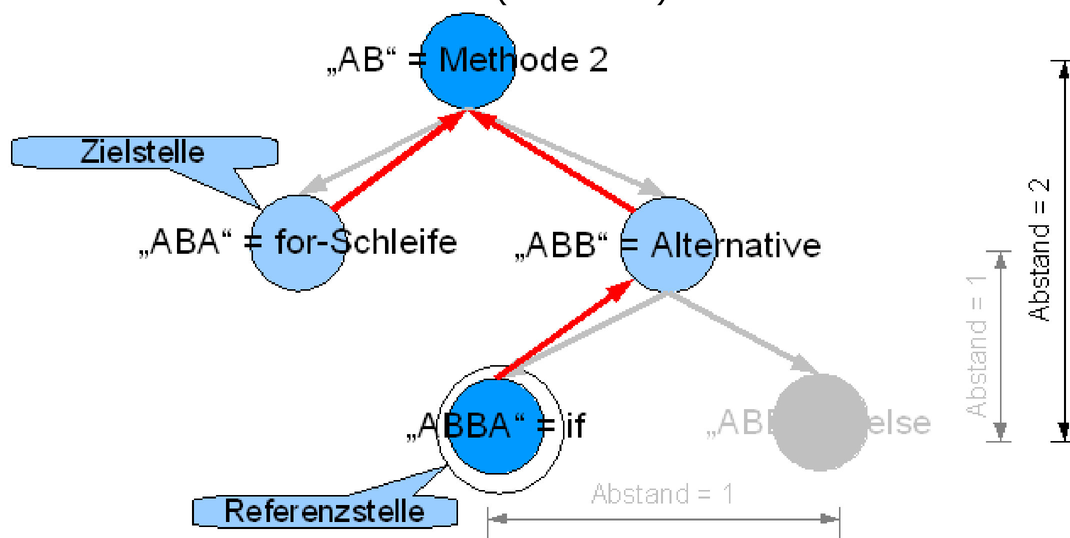

Abbildung 13: Beispiel Verwandtschaftsgrad: Referenzstelle hat zu Zielstelle einen Abstand von 2, da von der Referenzstelle aus zwei Knoten passiert werden müssen, bis der gemeinsame Vaterknoten von Referenzstelle und Zielstelle erreicht ist

Bezüglich der Verwandtschaft von Codestellen definiere ich in den Erkennern folgende Bedingungen:

- Stellen werden nur dann als verwandt erachtet, wenn sie sich in der gleichen Methode befinden. Diese Einschränkung verhindert zwar das oben beschriebene Erkennen semantischer Codeabhängigkeiten, verringert aber dadurch die Komplexität der Formalisierungen erheblich.
- Die Wurzel ist immer eine Datei.
- Die Wurzel nimmt eine Sonderstellung ein: Sie kann als einzige Stelle zusätzliche Ereignisse werfen, welche sich nicht auf den Code beziehen, sondern den Zustand der

Wurzel beschreiben. Diese Ereignisse sind u.a. „Datei gespeichert", „Datei geschlossen" und „Datei inaktiv". Ein Teil der vom ECG geworfenen Ereignisse wird somit bereits vom Stellenverfolger gekapselt und muss nicht einzeln in meinen Formalisierungen ausgedrückt werden, was die Beschreibung der Episoden in der ALED-Syntax vereinfacht.
Wird beispielsweise das Ereignis „Datei geschlossen" geworfen, so wird automatisch jede in dieser Datei laufende Hypothese unterbrochen, da das Schließen einen Abbruchindikator darstellt.

Zu beachten ist, dass eine textuelle Nähe zweier Codestellen gemessen an der zwischen den beiden Stellen liegenden Anzahl der Codezeilen nicht unbedingt eine nahe Verwandtschaft bezeugt: Anweisungen zweier textuell aufeinander folgender Methoden sind beispielsweise nur drei Zeilen voneinander entfernt, können aber einen großen Abstand im Hierarchie-Baum zueinander besitzen.

Es folgt eine Liste aller *Parameter* eines Stellen-Ereignisses (ausgenommen der Wurzel):

- *id* – eindeutige Identifikation des Ereignisses
- *file* – die eine Stellenänderung betreffende Datei
- *ancestor* – Position im Hierarchie-Baum als Liste aus Zeigern der Vater und Vorvater-Stellen (Vorväter dargestellt in Objektschreibweise „ancestor.ancestor")
- *timestamp* – Zeitpunkt zu dem die Stellenänderung auftrat
- *user* – Benutzer der Datei, d.h. der momentan am System arbeitende Entwickler
- *locationType* – Typ der Stelle (z.B. for-Schleife, if-Block, Methode, Normalcode, etc.)
- *changeType* – Typ der Codeänderung (z.B. Modifizieren, Einfügen, Anhängen, Löschen, Verschieben, etc.)
- *text* – der Quelltext der Stelle, sofern vorhanden

4. Erläuterung von ALED und Bedeutung für die Arbeit

ALED (Automation-Language for Episode Description) ist eine funktionale Beschreibungssprache für Episoden. Sie wurde zeitgleich mit meiner Arbeit als Bachelorarbeit von Christian Kopf erarbeitet. Das folgende Kapitel ist eine kompakte Erläuterung von ALED und der ALED-Syntax. Für weitere Informationen sei explizit auf die Arbeit von Christian Kopf verwiesen[28].

Jede Formalisierung einer Episode geschieht in ALED mit Hilfe von *Anweisungen* (Regeln), welche in der Art regulärer Ausdrücke formuliert werden und deren zeitliches Auftreten durch sequentielle Angabe der Anweisungen festgelegt ist. Eine Abfolge sequentieller Anweisungen bildet einen *Episodenerkenner*. Jeder Erkenner ist in einer separaten Datei abgespeichert.

[28]Christian Kopf Arbeit ist zu finden unter: http://projects.mi.fu-berlin.de/w/bin/view/SE/ThesesEpisodeRecognizerFramework

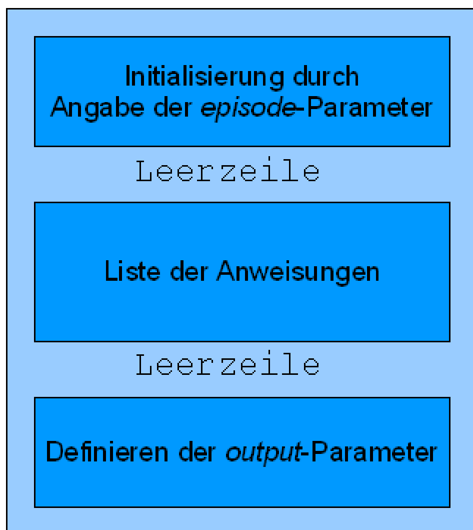

Abbildung 14: Strukturierung einer ALED-Formalisierung als Datei

Jede Datei einer ALED-Formalisierung besteht aus drei Teilen, welche jeweils durch eine Leerzeile getrennt sind (siehe Abbildung 14). Im ersten Teil werden die globalen Parameter gesetzt, wie z.B. die Abbruchbedingungen. Den zweiten Teil stellen die Anweisungen dar und im dritten Teil werden die Parameter des Ausgabe-Ereignisses definiert.

Jede Anweisung bezieht sich auf ein bestimmtes Ereignis, welches entweder auftreten oder nicht auftreten darf sowie eine Möglichkeit der Angabe von *Bedingungen* pro Anweisung. Mit Hilfe von „*Mengenoperationen*" (Vereinigung, Schnitt, Negation) können mehrere Ereignisse pro Anweisung kombiniert werden.

Weiterer Bestandteil jeder Anweisung ist die Angabe, wie oft (angegeben in *Mindest- und Maximalhäufigkeit*) die Anweisung hintereinander auftreten darf.

Tritt das in einer Anweisung spezifizierte Ereignis auf, so wird ein für jede Anweisung eindeutig formuliertes Codefragment – das so genannte *Aktionsstatement* - ausgeführt.

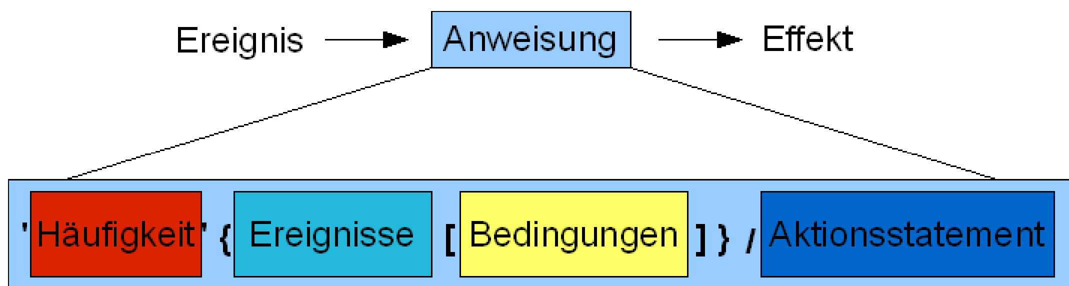

Abbildung 15: Aufbau einer Anweisung in ALED (mit Trennsymbolen)

Die Beschreibungssprache ALED ist zwar abstrakt, dennoch werden Bedingungen und Aktionsstatements in einer konkreten (aber beliebigen) Programmiersprache definiert. In meinem Fall war dies Java, weshalb der von mir eingesetzte Dialekt von ALED auch *Java-ALED* genannt wird.

Die vollständige Beschreibung der Syntax ist im Anhang zu finden.

Während ALED nur eine Metasprache zur Beschreibung einer Episode darstellt, ist ein *ALED-Rahmenwerk* ein Programm, welches Ereignisse als Eingabe erhält und die in ALED-Syntax formalisierten Erkenner tatsächlich nachverfolgen und die Aktionsstatements ausführen kann. Somit ist ein ALED-Rahmenwerk gewissermaßen eine Implementierung der ALED-Spezifikation.

Bedingungen und Aktionsstatements werden in der Implementierungssprache des ALED-Rahmenwerkes verfasst. Das Rahmenwerk führt die Aktionsstatements direkt aus, weshalb die in ALED definierten Anweisungen sehr mächtig sein können.

4.1. Anpassen der theoretischen Überlegungen an die Praxis in ALED

Leider lag bis zum Abschluss der Arbeit keine Implementierung eines Rahmenwerkes vor, so dass die formalisierten Erkenner nicht eingesetzt werden können.

Meine Anforderungen an ein ALED-Rahmenwerk sowie meine (weiter unten erläuterten) Erweiterungen in ALED habe ich deshalb so formuliert, dass diese ohne größeren Mehraufwand in ein bestehendes original-ALED Rahmenwerk integriert werden können.

Meine in ALED formulierten Episodenerkenner weisen zwei große Unterschiede zu den von mir im theoretischen Teil *B* dieser Arbeit geforderten Episodenerkennern auf: Erstens gibt es entgegen meinen Überlegungen keine Pufferung (caching) der vorangegangenen Ereignisse und zweitens ist das Konzept der Startindikatoren stark vereinfacht.

Wir Menschen bewerten den augenblicklichen Ist-Zustand also aufgrund von Geschehnissen aus der Vergangenheit (denken also in „Puffern"), ALED sieht jedoch keinen Pufferung vor. Ich sehe dennoch von der Forderung eine Puffers in einem ALED-Rahmenwerk ab, da dieses Konstrukt die Komplexität einer Episodenformalisierung noch steigert. Statt dessen habe ich die Spezifikation von ALED um ein simpleres Konstrukt als den Puffer erweitert, die *Vorfilterung:* Bestimmte im Initialisierungsteil des Episodenerkenners beschriebenen Parameter ermöglichen es dem Vorfilterungs-Mechanismus, Nichtindikatoren von Folgeindikatoren zu unterscheiden. Somit ist eine Betrachtung der Präindikatoren nicht mehr notwendig und der *Puffer* wurde um ein äquivalentes Konstrukt ersetzt (Details dazu siehe weiter unten im Kapitel 4.3).

Dies hat allerdings zur Folge, dass der Start einer Hypothese nicht mehr bei einem *run*-Ereignis („Ausführen-Ereignis") möglich ist: Aufgrund des nicht vorhandenen Puffers wäre unklar (bzw. vollkommen nichtdeterministisch), auf welche Datei oder Stelle sich das *run* bezieht, da ein *run*-Ereignis keinerlei Parameter bezüglich der Codestelle oder Datei enthält. Startet ein Erkenner ohne Puffer weiterhin erst nach einem *run*, so ist die Episode möglicherweise bereits zu weit voran geschritten, um noch erkannt zu werden.

Deshalb vereinfache ich das Konstrukt der Startindikatoren dahingehend, dass jeder im Rahmenwerk geladene Erkenner bei jedem Auftreten eines *codechange*-Ereignisses („Programm-Manipulations-Ereignis") startet. Der einzige Startindikator ist somit ein *codechange*-Ereignis. Die von mir häufig verwendeten Stellenverfolger-Ereignisse sind *codechange*-Ereignisse.

Dieses Vorgehen ist wahrscheinlich ressourcenintensiv, aber ohne Puffer unvermeidbar. Allerdings werden viele Erkenner bereits sehr schnell – etwa nach dem fünften Folge-Ereignis nach dem Start des Erkenners – wieder verworfen. Bei den 7 definierten Erkennern laufen so zu jedem Zeitpunkt etwa 35 parallele Erkenner.

4.2. Hierarchien von Erkennern

Das ALED-Rahmenwerk lädt beim Starten des Rahmenwerks die vom Benutzer gewählten Episodenerkenner (also Dateien), nach denen es im Ereignisstrom suchen soll.

Bei Auftreten jeden Ereignisses prüft das Rahmenwerk, ob das Ereignis in einem Erkenner definiert ist; Es wird bei jedem eintreffenden Ereignis jeder geladene Erkenner auf einen möglichen Zustandsübergang kontrolliert. Nach Vollendung eines Erkenners generiert das Rahmenwerk ein *Ausgabe-Ereignis* und integriert es in den Ereignis-Datenstrom. Somit steht das Ausgabe-Ereignis weiteren Erkennern zur Verfügung.

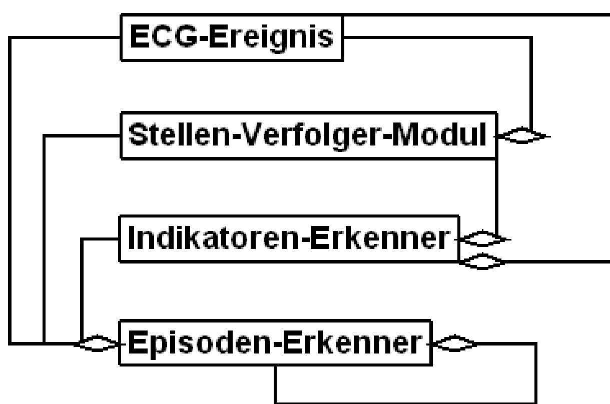

Abbildung 16: Zusammensetzung von Erkennern aus Ereignissen; in UML-artiger Notation

Sich anschließende Erkenner können Ausgabe-Ereignisse vorhergehender Erkenner in ihren Anweisungen nutzen und auf diese Weise wird eine Hierarchie von Erkennern geschaffen (siehe Abbildung 16).

Wie abgebildet existieren vier Arten von Ausgabe-Ereignissen – jeweils eine pro Komplexitäts-Ebene.

Ein Episoden-Erkenner etwa basiert auf ECG-Ereignissen, Ereignissen des Stellenverfolgers und auf Ereignissen eines Indikator-Erkenners.

Ein Indikatorerkenner ist ein Erkenner, welcher bestimmte Aktivitäten des Benutzers zusammenfasst, ohne dass diese eine Episode ergeben müssen. Beispiel ist das Erkennen, ob eine Stelle auskommentiert wurde. Die Ausgabe-Ereignisse eines Indikator-Erkenners werden von einem Episodenerkenner genutzt, um komplexere Folgeindikatoren festzustellen.

Diese Modularisierung erhöht die Wiederverwendbarkeit einzelner Erkenner und senkt gleichzeitig die Komplexität, da die Anweisungen nun kürzer ausfallen können.

Auf diese Weise habe ich das Problem der sich überlagernden T&E-Hypothesen aus Teil *B* gelöst: Die erste T&E-Episode und alle folgenden unnötig erkannten T&E-Episoden werden als neue Ereignisse auf den Datenstrom geschrieben. Ein weiterer sich anschließender Episodenerkenner lauscht auf diese Episoden-Ereignisse und fügt sie zu einer großen T&E-Episode zusammen.

Eine kleine Randbemerkung: Ereignisse dürfen niemals vom Ereignisstrom gelöscht werden, da sie jedem möglichen Erkenner zur Verfügung stehen müssen. Mit zunehmender Anzahl hintereinander geschalteten Erkennern steigt die Anzahl der Ereignisse auf dem Strom.

4.3. Vorfilterung – eine Erweiterung der ALED-Spezifikation

ALED sieht vor, dass alle erwünschten sowie alle unerwünschten Ereignisse als Anweisungen formuliert werden. Darf ein bestimmtes Ereignis nicht auftreten, so wird dies explizit als Anweisung (Regel) definiert.

Dies wirft jedoch einige Probleme auf:

- Verschiedene Ereignisse auf dem Strom können mehreren Dateien, Benutzern, Projekten, etc. zugeordnet sein. Hier sind jedoch nur Ereignisse eines Benutzers in einem Projekt in einer Datei relevant.
  Es muss also für jede einzelne Regel des Erkenners eine entsprechende Bedingung formuliert werden, welche alle zu einer anderen Datei gehörigen Ereignisse ausfiltert. Gleiches gilt für eine Filterung irrelevanter Benutzer oder Projekte. Dies steigert den Aufwand und die Komplexität der Formulierung enorm.
- Zwischen zwei Folgeindikatoren einer Episode dürfen eine gewisse Anzahl Nichtindikatoren auftreten, ohne dass der Erkenner verwirft. Deshalb muss zwischen jeder Anweisung des Erkenners eine zusätzliche Anweisung eingefügt werden, welche

diese Nichtindikatoren (also Aktionen anderer Benutzer, Dateien, etc.) inklusive ihrer erlaubten Häufigkeit explizit berücksichtigt.

Die Spezifikation der Erkenner wird auf diese Weise unnötig komplex. Deshalb habe ich ALED um das zusätzliche Konstrukt der *Vorfilterung* erweitert.

Die Parameter *Benutzer, Zeitstempel, Dateiname* und *Projekt* des Startindikators (des ersten eintreffenden Ereignisses) werden als Referenz-Parameter im Episodenerkenner gespeichert und dienen als Vergleichsbasis für alle folgenden eintreffenden Ereignisse: Nur wenn die Parameter der Folge-Ereignisse mit den Referenz-Parametern *Benutzer* und *Projekt* identisch sind, werden sie in die Auswertung der Anweisungen hinein bezogen (werden also zu Folgeindikatoren). Alle anderen Ereignisse werden ausgefiltert und nicht weiter betrachtet!

Zusätzlich wird die Vorfilterung nur auf *codechange*-Ereignisse angewendet, also auf Ereignisse welche den Quellcode modifizieren. Beispiele für *codechange*-Ereignisse sind *Stellen*-Ereignisse und *copy&paste*-Ereignisse. Alle anderen Ereignisse erreichen den Erkenner ohne Vorfilterung (z.B. *run*-Ereignisse).

Die *Vorfilterung* hat den großen Vorteil, dass ein Programmierer eines ALED-Episodenerkenners simplere Erkenner verfassen kann, da nicht bei jeder Anweisung mittels einer Bedingung geprüft werden muss, ob das soeben eingetroffene Ereignis die geforderten Bedingungen erfüllt (wie etwa einen bestimmten Benutzer x oder ein bestimmter Stellenbereich *y*). Somit werden die unerwünschten Nichtindikatoren bereits vor Eintritt in den Erkenner erkannt und verworfen!

4.4. Abbruch eines Erkenners aufgrund eines Abbruchindikators

Ein Erkenner – bzw. die in ihm laufende Hypothese – wird abgebrochen bzw. beendet, sobald ein Abbruchindikator auftritt.

Dies ist im seltensten Fall ein einzelnes Ereignis (wie etwa das Schließen der aktuellen Datei), sondern eine Abfolge von Ereignissen. Abbruchindikatoren werden als Abbruchbedingungen in den Initialisierungsteil der Erkenner-Datei geschrieben. Das Rahmenwerk überprüft unentwegt ob eine Abbruchbedingung eingetreten ist und beendet den laufenden Erkenner darauf hin sofort.

Ob anschließend ein Ausgabe-Ereignis generiert wird, ist durch den Zustand der Hypothese bedingt: Nur wenn sie bei Abbruch bestätigt ist, wird ein neues Ausgabe-Ereignis generiert.

Welcher Art die gefundene Episode ist und welche Parameter das resultierende zu werfende Ausgabe-Ereignis trägt, wird im letzten Abschnitt einer Erkenner-Datei definiert.

Es folgt eine Auflistung aller definierten Abbruchindikatoren:

- *Inaktivität der Datei für einen definierten maximalen Zeitraum -* Inaktivität ist ein Zeitraum ohne Modifikationen einer Stelle oder Datei. Während dieser Zeit können natürlich vielfältige Ereignisse anderer Art auftreten.
- *Inaktivität der relevanten Stellen für einen definierten maximalen Zeitraum* – Inaktivität ist ein Zeitraum ohne Modifikationen der Stelle. Während dieser Zeit können natürlich vielfältige Ereignisse anderer Art auftreten.
  Die relevanten Stellen ergeben sich aus der Verwandtschaft der Stellen bezüglich der Referenzstelle des Startindikators.
- *Schließen der Datei (implizit)*

- *Beenden der IDE (implizit)*
- *Löschen der betrachteten Stelle (implizit)*
- *kein* run*-Ereignis oder* debug*-Ereignis für einen definierten maximalen Zeitraum*
- *Auftreten einer definierten maximalen Anzahl aufeinander folgender Nichtindikatoren* – Vorgefilterte Ereignisse anderer Benutzer und Projekte zählen nicht als Nichtindikatoren, sondern werden überhaupt nicht betrachtet.

Einige der Abbruchbedingungen werden implizit vom Rahmenwerk durchgeführt, ohne dass eine Angabe eines Parameters mit Wert seitens des Entwicklers notwendig ist. Ein Erkenner wird immer dann implizit beendet, wenn die betrachtete Datei geschlossen wird oder die Entwicklungsumgebung beendet wird.

Das Schlüsselwort *ABORT* bewirkt einen sofortigen Abbruch der Episode. Dies ist der „manuelle" Weg, einen Erkenner abzubrechen und das evtl. anschließende Werfen des Ausgabeereignisses auszulösen.

Die Werte der Parameter sind für verschiedene T&E-Episoden zwar unterschiedlich, liegen aber alle relativ nahe beieinander: Der Zeitraum der maximalen Inaktivität einer Stelle liegt stets zwischen zwei bis fünf Minuten.

Der Fokus dieser Arbeit liegt auf der Erkennung kurzer T&E-Episoden mit einer maximalen Länge von etwa 10 Minuten. Es gibt zwar auch Fälle von zwei Tagen währenden Episoden (wie mir ein Entwickler berichtete), Episoden dieser Länge sind viel jedoch zu umfangreich und individuell, um sie generalisieren und somit in einem Erkenner formalisieren zu können.

Ein Beispielszenario für Abbruchindikatoren:

> *„Der Erkenner wird verworfen, wenn die relevanten Stellen (also die verwandten Stellen) länger als 120 Sekunden nicht mehr verändert wurden und/oder das Programm für mindestens drei Minuten nicht mehr getestet (also ausgeführt) wurde und/oder die Anzahl aufeinander folgender Nichtindikatoren größer als 20 war (was auf eine Konzentration des Entwicklers auf andere Codestellen und damit andere Probleme hindeutet). Eintreffende Stellen-Ereignisse werden dabei als Nichtindikatoren gewertet, wenn ihr Abstand zur Referenzstelle größer als Vier ist."*

Die im Szenario genannten Bedingungen werden folgendermaßen beschrieben (ich nehme hier bereits die später erklärte ALED-Syntax vorweg):

```
episode.max_inactiveTime = 120
episode.max_inactiveTimeRun = 180
episode.max_locationDistance = 4
episode.max_nonindicators = 20
```

Wie genau das Erkenner-Rahmenwerk die Parameter ausliest, speichert und verarbeitet erklärt der folgende Abschnitt.

4.5. Definierte Objekte eines ALED-Rahmenwerkes

Das Ansprechen der globalen Parameter und der Ereignis-Parameter jedes einzelnen Ereignisses geschieht in ALED anhand dreier so genannter Objekte. Dies sind die reservierten Schlüsselwörter *episode*, *event* und *output*.

Die Parameter der Objekte werden durch die gewohnte objektorientierte Punkt-Schreibweise angesprochen:

4.5.a. <u>Das *episode*-Objekt:</u>

Das *episode*-Objekt enthält alle globalen Parameter sowie die aus den im ersten eintreffenden Ereignis (dem Startindikator) ausgelesenen Referenz-Parameter, mit denen die späteren Ereignisse in der Vorfilterung verglichen werden. Dieses Objekt wurde von mir der ursprünglichen ALED Spezifikation hinzugefügt.

- implizit durch den Startindikator geerbte Parameter:
  - *Benutzername (user)*
  - *Dateiname (file)*
  - *Projekt (project)*
  - *Zeitstempel (timestamp)*
  - *Stelle (location)*
- explizite Parameter:
  - *max_nonindicators* – maximale Anzahl aufeinander folgender Nichtindikatoren bis der Erkenner verworfen wird
  - *max_locationDistance* – maximaler erlaubter Abstand (Verwandtschaftsgrad) der Stellen, bis diese als Nichtindikator eingestuft werden
  - *max_inactiveTime* – maximaler Zeitraum der Nichtmodifikation der relevanten Stellen, bis die Episode verworfen wird; Angabe in Sekunden
  - max_inactiveTimeRun – maximaler Zeitraum zwischen zwei *run*-Ereignissen, bis die Episode verworfen wird; Angabe in Sekunden
  - *begin_timestamp –* Zeitstempel des Beginns der Episode (ist gleich dem Zeitstempel des Ereignisses des Startindikators)
  - *end_timestamp* – Zeitstempel des Endes der Episode (ist gleich dem Zeitstempel des letzten Ereignisses bevor die Episode abgebrochen wurde)
- spezielle Parameter:
  - *id* – eindeutige Identifikation; wird automatisch generiert
  - *validated* – Status, ob Hypothese bereits bestätigt (*true* wenn ja, *false* sonst); initialisiert mit *false*
- <u>Definition</u>: Die expliziten Parameter werden nur dann vom Rahmenwerk verarbeitet, wenn sie im Initialisierungsteil der Datei aufgelistet werden. Nicht gelistete Parameter werden nicht überprüft.
  id und *validated* sind besondere Parameter, da sie als einzige Parameter automatisch vom Rahmenwerk initialisiert werden: *id* wird automatisch generiert und *validated* ist mit *false* initialisiert.

Die vier Parameter *max_nonindicators, max_locationDistance, max_incativeTime* und *max_inactiveTimeRun* bilden ein Maß für die „Strenge" oder Genauigkeit eines Erkenners. Durch eine enge Belegung lässt sich so die Wahrscheinlichkeit der *falschen positiven Erkennungen* (*false positives*) verringern, während eine weitere Auslegung eine größere Anzahl an Episoden erkennt.

4.5.b. Das *event*-Objekt:

Dieses Objekt wird bei Eintreffen jeden neuen Folgeindikators - also mit Eintreffen jeden relevanten Ereignisses - neu generiert, die Parameter ändern sich von Ereignis zu Ereignis.

Feste Parameter des *event*-Objekts:

- *name* – Name bzw. Typ des Ereignisses (z.B. *codechange* oder *trial-and-error-episode*)
- *id* – eindeutige Identifizierung des Ereignisses
- *user* – Benutzername des Entwicklers
- *project* – Projektnamen, an dem der Entwickler momentan arbeitet
- *file* – Dateiname, in dem das Ereignis liegt; Nicht bei jedem Ereignis verfügbar (z.B. bei einem *run*-Ereignis)
- *timestamp* – Zeitstempel, zu dem das Ereignis auftrat
- *location* – die zum eintreffenden Ereignis zugehörige Stelle; Nicht bei jedem Ereignis verfügbar (z.B. bei einem *run*-Ereignis)

Die restliche Liste der Parameter ist dynamisch, da jedes Ereignis über individuelle Parameter verfügt. Von mir häufig genutzte Ereignisse waren natürlich die Ereignisse des Stellenverfolgers, dessen Parameter bereits weiter oben in Kapitel 3.1 aufgelistet sind.

4.5.c. Das *output*-Objekt:

Dieses Objekt enthält alle Parameter des Ausgabe-Ereignisses – also des Ereignisses welches nach Abbruch bzw. Beendigung eines Erkenners vom Rahmenwerk auf den Ereignisstrom geschrieben wird. Dieses Objekt wurde von mir zur ALED Spezifikation hinzugefügt.

Das Ausgabe-Ereignis wird nur dann geworfen, wenn bei Auftreten eines Abbruchindikators der *episode.validated*-Parameter auf *true* gesetzt ist.

Sollte der Episodenerkenner abbrechen, ohne dass *validated* gesetzt ist, bricht die Episode ohne Werfen eines Ereignisses ab.

Feste Parameter des *output*-Objekts:

- *eventName* – der Name des zu werfenden Ereignisses (und damit der offizielle Name der zu werfenden Episode)
- *id* – die eindeutige Identifizierung des Ereignisses; wird automatisch generiert
- *begin_timestamp* – ist gleich *episode.begin_timestamp*; automatisch generiert
- *end_timestamp* – ist gleich *episode.end_timestamp*; automatisch generiert

Alle anderen Parameter sind individuell für jede Episode. Das resultierende Ausgabe-Ereignis ist - wie bereits oben beschrieben – ein XML-Konstrukt. Sämtliche Werte der Parameter müssen deshalb Zeichenketten (Strings) sein oder Zahlen (wobei ich davon ausgehe, dass das Rahmenwerk diese Zahlen automatisch in Strings konvertiert).

4.5.d. Das Schlüsselwort *ABORT*:

Dieses Schlüsselwort bewirkt das sofortige Abbrechen der Episode und steht im Anweisungsstatement. Der Einsatz ist dann sinnvoll, wenn nicht auf einen der im Vorhinein definierten Abbruchbedingungen gewartet werden soll (wie etwa zwei Minuten der Inaktivität).

Teil D) Formalisierung der Trial&Error-Episoden

In diesem nun abschließenden Teil der Arbeit werden die formalisierten Erkenner vorgestellt.

Zusätzlich zur Formalisierung in Java-ALED wird für jeden Erkenner eine Motivation und eine Beschreibung angegeben. Soweit es möglich war, ist jedem Erkenner ein Videoausschnitt beigelegt, um das Vorkommen des Erkenners in der Realität zu bekräftigen.

Die erste Formalisierung stellt die *allgemeine T&E-Episode* dar. Die Idee dahinter ist, dass sich jede weitere Formalisierung einer T&E-Episode auf diese allgemeine Episode zurückführen lässt, weil die allgemeine Episode das Grundgerüst aller T&E-Episoden beschreibt, nämlich alternierende Codeänderungen und Testläufe. Es gibt auch T&E-Episoden, welche sich nicht explizit einer *speziellen Episode* zuordnen lassen. Diese Episoden werden von der *allgemeinen T&E-Episode* aufgefangen und erkannt.

Anschließend werden die Indikator-Erkenner aufgelistet, welche zum Teil Voraussetzungen für die im Anschluss niedergeschriebenen speziellen Episoden-Erkenner sind.

Auf die in Abschnitt 3 beschriebenen Episoden-Erkenner aufbauend folgt ein letzter Abschnitt, welcher die Über-Episoden beschreibt, also Erkenner welche T&E-Episoden selbst zu einer neuen Episode zusammenfassen.

Einen guten Einstieg bietet die Formalisierung 4.1, da sie leicht verständlich und zudem ausführlich beschrieben ist.

Legende:

- `schwarz`: Mengenangabe und Trennsymbole
- `blau`: Ereignisse
- `grün`: Bedingungen
- `rot`: Aktionsstatements
- `//` : Kommentar

Verwendete Werte für die Episoden- bzw. Ereignisparameter sind selbsterklärend gewählt, so dass sie in Ihrer Bedeutung nicht explizit aufgelistet sind.

Eine Beschreibung der Syntax von ALED befindet sich im Anhang (Abschnitt F.3.4).

1. Formalisierung der *allgemeinen Trial-and-Error-Episode*

Diese Formalisierung eines *allgemeinen Erkenners* definiert eine allgemeine T&E-Episode. „Allgemein" bedeutet, dass jede T&E-Episode grob die hier präsentierte Struktur besitzt.

Diese Episode deckt alle „normalen" T&E-Episoden ab. Dies sind jene Episoden, welche sich nicht durch besondere Merkmale wie das mehrmalige Ändern einer Schleife oder Testausgaben hervortun und somit nicht von den im nächsten Abschnitt beschriebenen *speziellen Erkennern* identifiziert werden können.

Beispiel für eine solche normale Episode ist das *Video010*.

```
// die Initialisierungsparameter:
episode.max_nonindicators = 5
episode.max_inactiveTime = 180
episode.max_inactiveTimeRun = 240
episode.max_locationDistance = 4

// die Anweisungen:
// Benutzer, Datei und Projekt werden impliziert durch das erste Ereignis (den
//      Startindiktor) definiert.
// Nur wenn Folgeereignisse mit diesen Parametern des Startindikators identisch sind, werden sie
//      betrachtet. Alle anderen Ereignisse werden ausgefiltert.
'1,1' {<locationchange>} /    ;
'0,4' {<locationchange>} /    ;
'1,2' {<run>} /               ;
'1,2' {<locationchange>} /    ;
'1,2' {<run>} /               episode.validated = true;     // ab hier ist die Hypothese bestätigt
                              Time lastTime = event.timestamp;
'0,*' {{<locationchange>} + {<run>}} /              lastTime = event.timestamp;
'1,1' {<run> [(event.timestamp > lastTime+100)]} /  ABORT;

// Definition des Ausgabeereignisses
output.eventName = „trial-and-error-episode"
output.type = „general"
// Die id, die Referenzstelle und die Beginn-, Endezeitstempel werden automatisch generiert
```

Formalisierung der allgemeinen T&E-Episode

Eine Abfolge von Ereignissen wird dann als allgemeine T&E-Episode erkannt, wenn

- Es mindestens eine und maximal vier Codeänderungen in einem Abstand der Referenzstelle von maximal 4 gibt
- Anschließend das Programm einmal bis zweimal ausgeführt (getestet) wird
- Anschließend ein bis zwei Codeänderungen im erlaubten Bereich stattfinden
- Anschließend das Programm einmal bis zweimal ausgeführt wird. Ab diesem Zeitpunkt ist die Hypothese bestätigt.
- Anschließend Null bis beliebig viele *run-* und/oder *codechange*-Ereignisse auftreten, wobei zwischen einem *run* und dem vorhergehenden Ereignis weniger als 100 Sekunden vergehen dürfen. Ansonsten wird die Episode beendet.

Zusätzlich beendet der Erkenner, wenn mehr als 5 aufeinander folgende Nichtindikatoren auftreten und/oder innerhalb von 3 Minuten keine Codeänderung im erlaubten Bereich erfolgt und/oder in einem Zeitraum von 4 Minuten keine zwei Testläufe stattfinden.

2. Formalisierung der *Indikator-Erkenner*

Diese Formalisierungen beschreiben keine kompletten Episoden-Erkenner, sondern fassen lediglich bestimmte Ereignisse so zusammen, dass ein T&E-Erkenner diese als ein einziges Folge-Ereignis verwenden kann.

2.1. T&E-Indikator: Vereinfachung komplexer Ausdrücke

Code kann deshalb fehlerbehaftet sein, weil ein Ausdruck zu komplex und damit zu unübersichtlich geschrieben wurde. Eine Defektlokalisierung fällt schwer, da nicht sofort eindeutig ist, welcher Teil des Ausdrucks fehlerbehaftet ist. Entwickler wirken dem entgegen, indem sie die Komplexität des Codes temporär reduzieren bis der Defekt gefunden bzw. behoben ist.

Vereinfachungen sind das Einschränken der Funktionalität oder die Verwendung von statischen Werten (so genannten Dummies) anstatt Domänenparametern.

Dieser Indikator ist gegenüber der Realität stark vereinfacht. Ich definiere eine Vereinfachung einer Stelle so, dass ihre Länge in Zeichen verringert wird und sich direkt an die Verkürzung des Codes ein Testlauf anschließt.

```
// die Initialisierungsparameter:
episode.max_nonindicators = 3
episode.max_inactiveTime = 100
episode.max_inactiveTimeRun = 100
episode.max_locationDistance = 0     // nur die Referenzstelle selbst wird betrachtet

// die Anweisungen:
'1,1' {<locationchange>}      / ;
'1,3' {<locationchange>}      / int length = event.text.length();
'1,2' {<run>}                 / ;
'1,2' {<locationchange> [event.text.length() < length]}    / length = event.text.length();
'1,1' {<run>}                 / episode.validated = true; ABORT;

// Definition des Ausgabeereignisses
output.eventName = „trial-and-error-indicator‟
output.type = „simplification‟
// Die id, die Referenzstelle und die Beginn-, Endezeitstempel werden automatisch generiert
```

Formalisierung des T&E-Indikators „Codevereinfachungen"

2.2. T&E-Indikator: Auskommentieren von Code

Vor seiner Behebung muss ein Defekt erst gefunden werden. Um die Zahl der möglichen Fehlerstellen zu reduzieren, kommentieren Entwickler Code aus und verringern somit die Komplexität.

Vermehrtes Aus- und wieder Einkommentieren mit dazwischen liegenden Testläufen weist auf eine Debugging- bzw. T&E-Tätigkeit hin, weshalb ich dieses als Indikator formalisiert habe.

```
// die Initialisierungsparameter:
episode.max_nonindicators = 3
episode.max_inactiveTime = 30
episode.max_inactiveTimeRun = 30
episode.max_locationDistance = 0     // nur die Referenzstelle selbst wird betrachtet

// die Anweisungen:
'1,1' {<locationchange>}      / ;
'1,2' {<run>}                 / ;
'1,1' {<locationchange> [(event.changeType == „modified‟) &&
                      ((event.text.trim().startsWith(„//‟)) || (event.text.trim.startsWith(„/*‟))]}
                              / ;
'1,2' {<run>}                 / episode.validated = true; ABORT;

// Definition des Ausgabeereignisses
output.eventName = „trial-and-error-indicator‟
output.type = „comment-out‟
// Die id, die Referenzstelle und die Beginn-, Endezeitstempel werden automatisch generiert
```

Formalisierung des T&E-Indikators „Auskommentieren von Code"

2.3. T&E-Indikator: Einfügen von Testausgaben

Wie bereits erwähnt, kann ich keine gute Formalisierung für das Einfügen von Bildschirmausgaben geben, da diese auf zu vielfältige Weise (Kommandozeile, GUI, Getter/Setter) auftreten.

Ich präsentiere hier nur eine sehr grobe Formalisierung, die davon ausgeht, dass die Testausgabe auf der Konsole statt findet, die Stelle der Testausgabe und des fehlerbehafteten Codes nahe verwandt sind, sowie jede Testausgabe in einer eigenen Zeile steht und mit „System.out.println" beginnt.

```
// die Initialisierungsparameter:
episode.max_nonindicators = 5
episode.max_inactiveTime = 40
episode.max_inactiveTimeRun = 60
episode.max_locationDistance = 2     // nur die Referenzstelle selbst wird betrachtet

// die Anweisungen:
'1,1' {<locationchange> [(event.changeType == „inserted" || event.changeType == „modified") &&
                (event.text.trim.contains(„\nSystem.out.println(")]}       / ;
'1,2' {<run>}                / ;
'1,1' {<locationchange> [(event.changeType == „modified")] / ;
'1,2' {<run>}                / episode.validated = true; ABORT;

// Definition des Ausgabeereignisses
output.eventName = „trial-and-error-indicator"
output.type = „testoutput"
// Die id, die Referenzstelle und die Beginn-, Endezeitstempel werden automatisch generiert
```

Formalisierung des T&E-Indikators „Testausgaben werden eingefügt"

Erläuterung des Beispielvideos *Video009*, welches diesen Indikator visualisiert:

Der Entwickler führt das Programm aus und es erscheint - anders als gewünscht - keine Ausgabe auf der Kommandozeile. Darauf hin fügt er an der für die Ausgabe verantwortlichen Stelle ein „*a*" als Test ein und führt das Programm erneut aus. Da wiederum kein Effekt zu Beobachten ist, ergänzt er den Code um eine weitere mit Bestimmtheit anzuzeigende Ausgabe. Der folgende Testlauf zeigt die Testausgaben weiterhin nicht an. Somit ist sich der Entwickler sicher, dass der Ablauf des Programms die für die Ausgabe verantwortliche Stelle nicht erreicht und sucht den Defekt an anderer Stelle, was das Video jedoch nicht mehr zeigt.

-> Siehe *Video009*

3. Formalisierung der *speziellen Trial-and-Error-Episoden*

Die spezialisierten T&E-Erkenner sind komplexer strukturiert als der *allgemeine Erkenner*.

Bestimmte Abschnitte dieser Episoden können deshalb in weitere kleinere Teilerkenner – die oben beschriebenen Indikator-Erkenner – ausgelagert sein.

Diese Modularisierung erzeugt eine Hierarchie von Erkennern, an deren Ende die Über-Erkenner stehen, die keine Indikatoren betrachten, sondern vollständige T&E-Episoden zusammenfassen.

3.1. T&E-Episode: Schleifenbedingungen mehrfach verändert

Ein klassischer Fall eines T&E stellt diese Episode dar: Eine Schleife erreicht nicht die gewünschte Anzahl an Durchläufen und muss deshalb angepasst werden. Es müssen nun also entweder die Schleifenbedingungen selbst geändert werden (*head*), die Variablen im Schleifenrumpf (*body*), oder beides. In meinen Beobachtungen kam diese Art des T&E

zwar nur selten vor, ich selbst habe diese Schleifen-Episoden aber schon sehr oft durchlaufen und insbesondere Programmieranfänger machen diese Fehler häufig, weshalb ich diese Formalisierung aufgenommen habe.

Diese Formalisierung ist nur eine grobe Annäherung an die Realität. Es ist lediglich feststellbar, dass momentan eine Schleifen-Stelle verändert wird sowie die Position der Änderung (Kopf oder Rumpf). Die exakten Änderungen auf Zeilen- oder gar Zeichenniveau lassen sich nicht feststellen. Es ist also möglich, dass sich eine Änderung im Schleifenrumpf nicht auf die Schleifenvariablen bezieht und somit eine *falsche positive Erkennung* (englisch *false positives*) erkannt wird. Ich reduziere dieses Risiko, indem ich nur Modifikationen bereits bestehenden Codes betrachte (*changeType==modified*) und nicht das Einfügen neuen Codes (*changeType==inserted*).

-> Siehe *Video007*

```
// die Initialisierungsparameter:
episode.max_nonindicators = 5
episode.max_inactiveTime = 120
episode.max_inactiveTimeRun = 180
episode.max_locationDistance = 2

// die Anweisungen:
'1,1' {<locationchange> [event.locationType == „loop“]}          / String id = event.id;
'1,2' {<run>}         / ;
'1,2' { {<locationchange> [(event.id == id) && (event.changeType == „headChanged“)]} +
       {<locationchange> [((event.ancestor == id) || (event.ancestor.ancestor == id)) &&
                            event.changeType == „modified“]}}    / ;
'1,2' {<run>}         / ;
'1,2' { {<locationchange> [(event.id == id) && (event.changeType == „headChanged“)]} +
       {<locationchange> [((event.ancestor == id) || (event.ancestor.ancestor == id)) &&
                            event.changeType == „modified“]}}    / ;
'1,2' {<run>}         / episode.validated = true;
'1,*' { {<locationchange> [(event.id == id) && (event.changeType == „headChanged“)]} +
       {<locationchange> [((event.ancestor == id) || (event.ancestor.ancestor == id)) &&
                            event.changeType == „modified“]} + {<run>}}  / ;

// Definition des Ausgabeereignisses
output.eventName = „trial-and-error-episode“
output.type = „loop“
// Die id, die Referenzstelle und die Beginn-, Endezeitstempel werden automatisch generiert
```

Formalisierung der speziellen T&E-Episode „Schleifendurchlauf fehlerhaft“

3.2. Heuristische T&E-Episode aufgrund des Auftretens von T&E-Indikator-Ereignissen

Diese Episode habe ich nie beobachtet, sondern sie basiert vollständig auf heuristischen Annahmen, so dass sie einen ähnlichen Stellenwert wie die *allgemeine T&E-Episode* einnimmt. Das Auftreten diverser T&E-Indikator-Erkenner in einem begrenzten Zeit- und Stellenraum löst diese Episode aus. Diese Episode lässt die Betrachtung der Ausführen-Ereignisse außer Acht, da diese bereits von den Indikator-Erkennern berücksichtigt werden. Der Abbruch findet automatisch statt.

```
// die Initialisierungsparameter:
episode.max_nonindicators = 10
episode.max_inactiveTime = 120
episode.max_locationDistance = 2

// die Anweisungen:
'1,1' {<trial-and-error-indicator>} /       ;
'2,3' {<trial-and-error-indicator>} /       ; episode.validated = true;

// Definition des Ausgabeereignisses
output.eventName = „trial-and-error-episode“
output.type = „general“
// Die id, die Referenzstelle und die Beginn-, Endezeitstempel werden automatisch generiert
```

Formalisierung der Über-Episode „Zusammenschließen mehrerer T&E-Episoden“

4. Formalisierung der *Über-Episoden*

4.1. *merging*-Erkenner: Zusammenschließen (*merging*) von Trial-and-Error-Episoden

```
// die Initialisierungsparameter:
episode.max_nonindicators = 5
episode.max_inactiveTime = 300
episode.max_locationDistance = 1

// die Anweisungen:
// Benutzer, Datei, Projekt und Zeitstempel werden impliziert durch das erste Ereignis (den
//     Startindiktor) definiert
// Nur wenn Folgeereignisse mit diesen Parametern des Startindikators identisch sind, werden sie
//     betrachtet. Alle anderen Ereignisse werden ausgefiltert.
// „trial-and-error-episode"-Ereignisse sind genau wie „codechange"-Ereignisse einer Stelle
//     (location) zugeordnet, weshalb der Vorfilterungsparameter „max_locationDistance" angewendet
//     werden kann
'1,1' {<trial-and-error-episode>} /  Time t = event.end_timestamp;
                                     String members = event.id;
'1,*' {<trial-and-error-episode> [(event.end_timestamp <= t+60)]} /
                                     members.append(„, " + event.id);
                                     t += (event.timestamp – ereignis.timestamp);
                                     episode.validated = true;

// Definition des Ausgabeereignisses
output.eventname = „merged-trial-and-error-episodes"
output.members = members
// Die id und die Beginn-, Endezeitstempel werden automatisch generiert
```

Formalisierung der Über-Episode „Zusammenschließen mehrerer T&E-Episoden"

Das Zusammenführen von T&E-Episoden ist immer dann notwendig, wenn mehrere T&E-Episoden hintereinander in kurzem zeitlichem Abstand erkannt werden und diese Episoden zur gleichen Stelle gehören. In diesem besonderen Fall handelt es sich um eine einzige große T&E-Episode und die erkannten Einzelepisoden sind nur Teile dieser Episode und werden deshalb zusammen gefasst.

-> Siehe dazu Videobeispiel *Video006*.

Aufgrund der besseren Übersicht erfolgt die Erklärung der Episode in Stichpunkten:

- Der Anweisungsteil des Erkenners besteht nur aus zwei Zeilen. Die erste Zeile des Anweisungsteils ist der Startindikator. Bei jedem Auftreten des Ereignisses „trial-and-error-episode" wird ein neuer *merging*-Erkenner gestartet, bzw. eine neue Instanz des Typs *merging*-Erkenner. Bereits gestartete und noch nicht verworfene Erkenner werden selbstverständlich fortgesetzt.
- Bei Instanziierung werden dem Erkenner die Parameter der ersten drei Zeilen zugeordnet, sowie weitere Parameter, welche das Rahmenwerk aus dem Startindikator ausliest (Benutzer, Zeitstempel, Datei und Projekt) und als Referenzparameter zur Vorfilterung dienen.
- Der Startindikator darf genau einmal auftreten (*'1,1'*). Nach Auftreten wird der in rot gekennzeichnete Code des Anweisungsstatements ausgeführt (die Erzeugung der Variablen *time* und *members)*.
- Die zweite Anweisung darf beliebig oft auftreten, jedoch mindestens ein Mal (*'1,*'*). Das zugehörige Anweisungsstatement wird aufgrund der in grün gezeichneten Bedingung jedoch nur dann ausgeführt, wenn das Folge-Ereignis maximal 60 Sekunden nach dem vorgehenden Folgeindikator auftritt, was in der Neuzuweisung der Variablen *t* definiert ist.
- Bei jedem Vorkommen der zweiten Anweisung (unter Berücksichtigung der Bedingung) wird das Aktionsstatement ausgeführt. So wird der Parameter *episode.validated* bereits nach dem ersten Durchlauf der zweiten Anweisung gesetzt. Dies bedeutet, dass die

Hypothese des *merging*-Erkenners bereits nach zwei *trial-and-error-episode*-Ereignissen bestätigt wurde.
- Welche Episoden zusammengefasst wurden, steht in der sich stets verlängernden Textliste *members.*
- *Vorfilterung:* Nur diejenigen Stellen können im Voraus ausgefiltert werden, die einen Stellen-Parameter besitzen, für die also eine Stellenverwandtschaft festgestellt werden kann. Dies ist bei allen *codechange*-Ereignissen als auch bei allen T&E-Episoden-Ereignissen der Fall, da sich jede abgeschlossene Episode immer auf eine Stelle bezieht.
- Der Abbruch bzw. das Ende der Episode tritt bei Auftreten eines Abbruchindikators ein. Die Abbruchindikatoren sind im Initialisierungsteil festgelegt.
  Der Erkenner bricht ab, wenn
  - die Referenz-Stelle und alle verwandten Stellen maximal zulässigen Abstands für mindestens 300 Sekunden nicht mehr modifiziert wurden
  - und/oder mindestens 5 aufeinander folgende Nichtindikatoren auftreten
- Nichtindikatoren: Nur *trial-and-error-episode*-Ereignisse sind in den Anweisungen definiert. Deshalb sind alle anderen Ereignisse Nichtindikatoren für den Erkenner und werden deshalb nicht betrachtet.
  Auch *trial-and-error-episode*-Ereignisse werden zu Nichtindikatoren, wenn sie die Bedingung (zeitlich nicht älter als 60 Sekunden bezüglich auf die vorhergehende Anweisung) nicht erfüllen.
- Ein Ausgabe-Ereignis wird erzeugt wenn *validated* gesetzt ist und ein Abbruchindikator eingetreten ist.
  Das neu erzeugte Ereignis heißt *merged-trial-and-error-episodes*. Es bekommt eine Reihe automatisch zugewiesener Parameter sowie den Parameter *members*, welcher eine Liste aller eben zusammengefassten Episoden enthält.

Teil E) Zusammenfassung und Ausblick

Die Arbeit hat gezeigt, dass es möglich ist, im Programmieralltag auftretende semantische T&E-Episoden formal zu beschreiben. Obwohl diese Arbeit nur eine Möglichkeit der Formalisierung anhand spezieller Instrumente (ECG, Java-ALED, Stellenverfolger) aufgezeigt hat, lässt sich die Schwierigkeit derartiger Formalisierungen festhalten: Die Interpretation des Verhaltens und der Absichten eines Entwicklers mit Mitteln der elektronischen Datenverarbeitung ist noch sehr grob. Viele der hier vorgestellten Formalisierungen reichen nicht an ein detailreiches Niveau heran, wie es menschlichen Betrachtern aufgrund ihrer Auffassungsgabe zugänglich ist.

Für die Zukunft gilt es also, Ereignisse aus dem ECG zu verfeinern und weitere Erkenner zu schreiben, so dass sich Benutzeraktionen besser erfassen und zusammenfassen lassen.

Dennoch ließen sich alle beobachteten kürzeren T&E-Episoden in nur wenigen Formalisierungen definieren, dem ein erreichter hoher Grad der Abstraktion zu Grunde liegt.

Aufgrund eines nicht vorhandenen einsatzbereiten ALED-Rahmenwerks ließen sich die Formalisierungen leider nicht testen. Es ist davon auszugehen, dass die Formalisierungen

nach einigen Tests noch weiter verfeinert werden müssen und insbesondere die Rate der falschen positiven Erkennungen weiter optimiert werden muss. Ob die Erkenner in ihrem momentanen Zustand eher zu viele oder zu wenige *false positives* erkennen, vermag ich nicht vorher zu sagen. Die Struktur der in dieser Arbeit erweiterten ALED-Syntax lässt eine einfache Optimierung und Modifikation der formalisierten Erkenner mit nur einem minimalen Aufwand leicht zu.

Diese Arbeit war nicht nur auf die Formalisierung von T&E-Episoden fixiert, sondern hat sich mit dem noch wenig erforschten Feld der Episoden des Mikroprozesses befasst, so dass weiteren Forschungen auf diesem Gebiet von dieser Arbeit profitieren können und diese Arbeit als eine Vorlage zum Formalisieren weiterer Episoden dienen kann.

Teil F) Verzeichnisse und Anhänge

Zu aller erst sei auf die Wiki-Seite der Bachelorarbeit verwiesen:

http://projects.mi.fu-berlin.de/w/bin/view/SE/ThesisTrialError.

Dort finden sich unter Anderem ein Glossar der verwendeten Begriffe, eine Seite zum EventViewer sowie die Beispielvideos.

1. Literaturverzeichnis

- Jekutsch, Sebastian: http://projects.mi.fu-berlin.de/w/bin/view/SE/MicroprocessHome, Freie Universität Berlin, 2005
- Kim,M., Bergman,L., Lau,T. und Notkin,D.: *„An Ethnographic Study of Copy and Paste Programming Practices in OOPL"*, University of Washington, 2004
- Kopf, Christian: *„Entwicklung eines Rahmens zur einfachen Erstellung und Pflege von Episodenerkennern"*, Freie Universität Berlin, 2006
- Pavlina, Steve: *„Trial and Error, Ego and Awareness"*, http://www.stevepavlina.com, 2005
- Prechelt, Lutz: *„Vorlesung SWT – Kapitel 11 - Analytische Qualitätssicherung 1"*, Freie Universität Berlin, 2005
- Schlesinger, Frank: *„Protokollierung von Programmiertätigkeiten in der Eclipse-Umgebung"*, Freie Universität Berlin, 2005
- diverse Seiten des AG SE Wiki (http://projects.mi.fu-berlin.de/w/bin/view/SE/), Freie Universität Berlin, 2006

2. Materialverzeichnis

- *Video001:* Beispiel *IDE hilft Entwickler beim korrekten Einsatz einer Klasse*
- *Video002*: syntaktische T&E-Episode: *Erfolgloses Starten eines Kommandozeilenprogramms*
- *Video003*: Beispiel eines unbewussten Trial-and-Errors. Lösung erfolgt mit Hilfe bereits vorhandenen Codes erst nach drei Minuten des Ratens.
- *Video004*: einfaches künstliches Beispiel der T&E-Episode *FOR-Schleife*
- *Video005*: Defektbehebung aufgrund eines Copy&Paste-Fehlers
- *Video006*: komplexe Episode als Fallbeispiel
- *Video007*: Beispiel einer T&E-Schleifen-Episode. Beinahe sämtliche mögliche Belegungen für den Parameter einer while-Schleife werden probiert.
- *Video008*: Beispiel einer Verlegenheitsänderung und eines überflüssigen Testlaufs
- *Video009*: Beispiel für den Indikator-Erkenner *Einfügen von Testausgaben*
- *Video010*: Beispiel *Überschneidung von Defektlokalisierung und Trial-and-Error*. Eine ArrayIndexOutOfBounds-Exception muss behandelt werden. Die zugehörige Stelle besteht aus vielen gleichartigen und deshalb unübersichtlichen Anweisungen.

Die Videos sind auf der zugehörigen CD im MPEG4-Format DIVX abgelegt bzw. auf der Wiki-Seite dieser Arbeit zu finden.

3. Anhänge

3.1. Danksagung

Ich bedanke mich zu allererst bei Sebastian Jekutsch für seine gute Betreuung, sein offenes Ohr und seine flinken Finger bei der Beantwortung zahlreicher e-Mails.

Christian Kopf für die Zusammenarbeit und gegenseitige Bereicherung unserer Themen.

Weiteren Dank gilt allen meinen zahlreichen „Versuchspersonen" von Dahlem bis Steglitz bis Wilmersdorf über Schöneberg, Kreuzberg und Blankenfelde bei Berlin, welchen ich bei der Programmierung über die Schulter schauen durfte.

Schließlich danke ich Martin, Jacqueline und Juliane für das Sichten der Rechtschreibfehler und ihre kritischen Anmerkungen.

3.2. Dokumentation *Event Viewer*

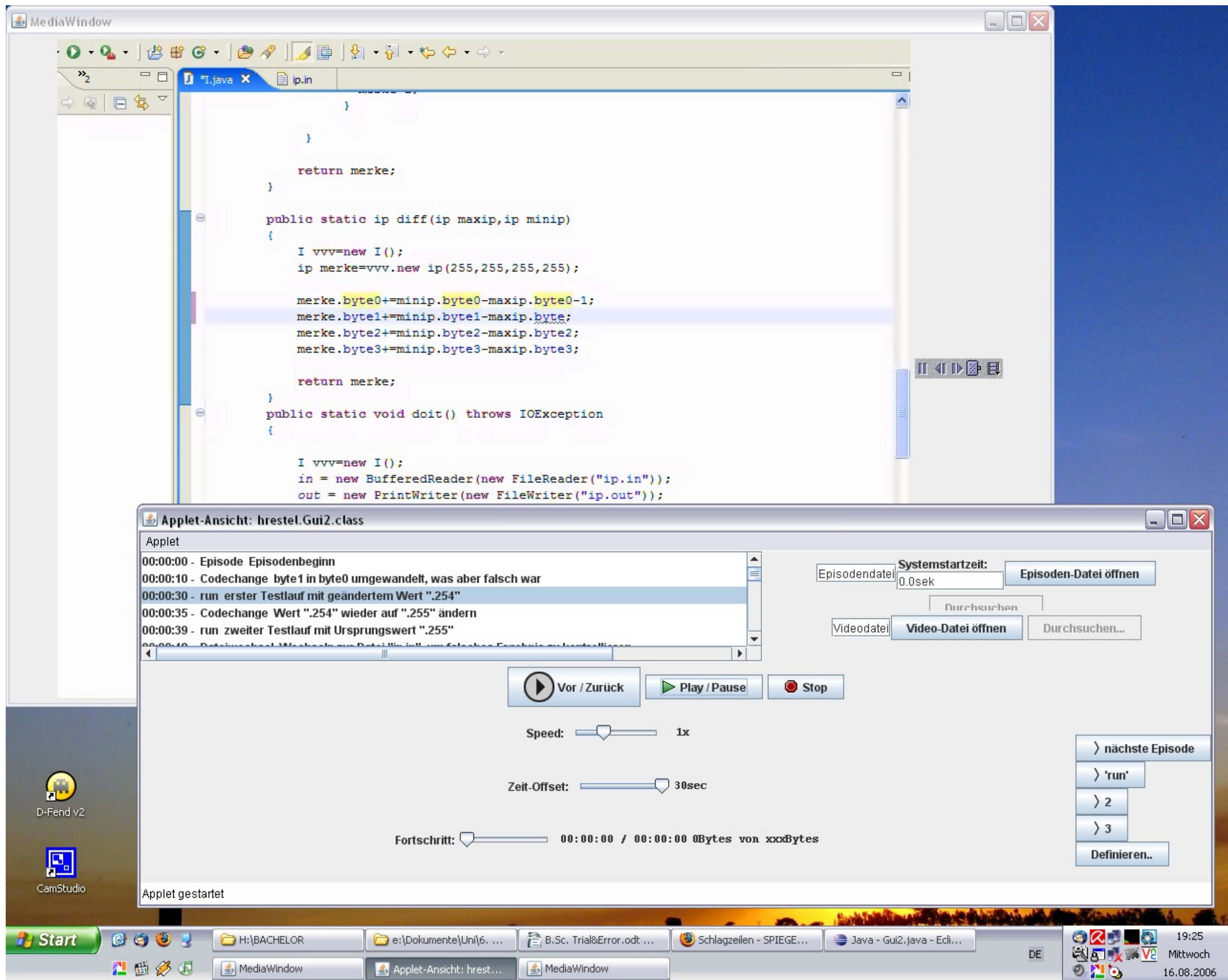

Abbildung 17: Bildschirmfoto des EventViewers

Systemanforderungen:

- Der *EventViewer* ist komplett in Java geschrieben und benötigt die *Java Runtime Environment (JRE Version5)* sowie das *Java Media Framework (JMF)* .

Ausgeführt wird das Programm in einem Webbrowser oder in einem AppletViewer. Dazu wird die mitgelieferte HTML-Datei im gewünschten Programm ausgeführt.

Navigation:

- Oben rechts können Video und die entsprechende Lesezeichen-Datei geladen werden.
- Mit Klick auf den Knopf *Play/Pause* startet das Video in einem separaten Fenster.
- Auch wenn der Rückwärts-Knopf bereits existiert: Momentan kann das Video nur vorwärts abgespielt werden!
- Mit dem Schieberegler *Speed* lässt sich die Abspielgeschwindigkeit einstellen, von einem Bruchteil der Originalgeschwindigkeit (Regler ist ganz links) bis zu 10facher Geschwindigkeit (Regler ist ganz rechts). Mit steigender Geschwindigkeit steigt die Prozessorauslastung.
- Der *Offset* -Schieberegler gibt den Zeitvorlauf an, um den das Video früher einsetzt, wenn zu einem Lesezeichen gesprungen wird.

- Die vier Knöpfe rechts in der Mitte ermöglichen das Definieren von Sprungmarkenregeln: Ein Klick auf den Knopf *Definieren* lässt ein neues Fenster erscheinen. Nun kann jedem der Vier Knöpfe eine Überschrift sowie ein Ausdruck zugewiesen werden. Der Ausdruck (eine Zeichenkette) bewirkt, dass beim Klicken auf den entsprechenden Knopf zu dem nächsten Lesezeichen in der Liste gesprungen wird, welches diesen Ausdruck textuell enthält.
  Der Knopf mit der Sprungmarkenregel *Nächste Episode* existiert immer und lässt das Video zum nächsten Lesezeichen in der Liste springen.

Personalisierte Lesezeichen-Datei:

Es ist leicht, sich selbst eine Lesezeichen-Datei zu erzeugen, und zwar für jedes beliebige Video. Dazu wird vom Benutzer eine Textdatei mit folgender Struktur angelegt:

- Die erste Zeile der Textdatei enthält die Überschriften für jede Spalte getrennt durch das Tabulator-Zeichen.
- Folgende zwei Spalten in folgender Reihenfolge mit folgendem Inhalt müssen existieren:
- Lesezeichenname (z.B. "trial-and-error-episode")
- Zeitstempel mit Format: "`TT.MM.JJJJ hh:mm:ss`"
- Anschließend können beliebig viele Spalten folgen, z.B. Spalten für Inhalt und Beschreibung .
- **Wichtig**: Jeder Eintrag jeder Zeile wird mit dem Tabulator-Zeichen getrennt!

Besonderheiten und Hinweise:

- Der *EventViewer* ist ein Applet. Die Sicherheitsrichtlinien der Applets sind für gewöhnlich so gesetzt, dass das Applet nicht (weder lesend noch schreibend) auf die lokale Festplatte zugreifen darf. Der EventViewer setzt deshalb alle Sicherheitsvorschriften (Policies) außer Kraft, um korrekt Arbeiten zu können.
- Es kann evtl. Probleme mit der Policies-Datei geben, so dass keine Daten von der Festplatte gelesen werden dürfen. In diesem Falle sollte das Programm in eine IDE (z.B. Eclipse) geladen werden und darüber ausgeführt werden.
- Das vom *EventViewer* verwendete JavaMediaFramework ist für das Abspielen der Videos notwendig.
  Wichtig: Videos, welche im DivX oder XVid-Codec komprimiert sind, müssen unbedingt "DIVX" im FourCC-Code stehen haben! Das JavaMediaFramework (JMF) erkennt die Videos nur in diesem Falle an. Ist der Code beispielsweise auf "DX50", "XVID" oder "DIV3" gestellt, so kann das Video nicht abgespielt werden!
  Das setzen des Codes ist mit dem Programm *AVI FourCC Changer* möglich.
- Erfolgreich getestet wurde das Programm unter Windows und Linux. MacOS X kann aufgrund von Problemen mit dem JMF keine Videos abspielen.

Verweise:

- Download des *EventViewer*s
- alles über FourCC Codes
- Download des AVI Four CC Changer
- Java Media Framework

3.3. Formular für die Observierungen

Formular und Protokoll (Maßstab wurde verkleinert)
zur Observierung von semantischen T&E Vorgängen beim Programmieren

Name der Testperson:
Ort:
Datum:
Beginn: Ende:
Observierung mit CapturingProgramm: ja / nein

kurze Beschreibung des Arbeitsumfelds und der aktuellen Programmiertätigkeit: ...
...

Beobachtete semantische T&E Episoden und Indikatoren:
Vereinfachung komplexer Ausdrücke:
Auskommentieren von Code:
Schleifenbedingungen fehlerhaft:
Einfügen von Testausgaben/Zustandsausgaben:
Überschneidung von T&E mit Debugging-Phase:
Debugger benutzt:
semantische Codeabhängigkeiten verhängnisvoll:
sonst:

Beobachtete syntaktische T&E Episoden:
IDE hilft:
verschachtelte Schleifen falsch geschlossen:
komplexer unübersichtlicher Einzeiler:
Auto-Vervollständigung benutzt:
IDE entfernt Fehler / Hilfe beim Refactoring:
bewusst Fehler produziert:
sonst:
in API/Dokumentation/Anforderung schauen:
Kommandozeile falsch benutzt:

von Testperson selbst erkannte T&E-Episoden: ...
...

Ich habe Episode fälschlicherweise als T&E-Episode gedeutet: ...
...

Notizen: ...
...

3.4. Übersicht der Syntax von Java-ALED

Diese Übersicht entstammt einer Vorlage von Christian Kopf, welche für diese Arbeit modifiziert und erweitert wurde.

```
Formalisierung        :=   Initialisierung Leerzeile Episode Leerzeile output-Beschreibung

Initialisierung       :=   keine, einige oder alle der folgenden Parameter samt Wert, jeder
                           Parameter jedoch nur einmal und getretnnt durch eine Leerzeile:
                           max_nonindicators , max_locationDistance, max_inactiveTime,
                           max_inactiveTimeRun

output-Beschreibung   :=   eventName = <Wert> EndOfLine [Menge beliebiger anderer Parameter
                           getrennt durch eine Leerzeile]

Episode               :=   Anweisung
                           {Anweisung EndOfLine} Anweisung

Anweisung             :=   Auftrittshäufigkeit {Ereignismenge} / [Aktionsstatement]

Auftrittshäufigkeit   :=   'Mindesthäufigkeit, Höchsthäufigkeit'

Mindesthäufigkeit     :=   {0,1,2,3,...}

Höchsthäufigkeit      :=   {0,1,2,3,...,*}       // * bedeutet: beliebig oft

Ereignismenge         :=   Ereignistyp
                           Ereignismenge Mengenoperator Ereignismenge
                           {Ereignismenge}[Bedingung]

Mengenoperator        :=   {+, \, ~}
                           „+" bedeutet ODER
                           „\" bedeutet OHNE
                           „~" bedeutet SCHNITTMENGE

Ereignistyp           :=   Ereignisname[Bedingung]

Ereignisname          :=   <Namensattribut eines Ereignisses>
                           #                     // # bedeutet beliebiges Ereignis

Bedingung             :=   [boolscher Ausdruck in Javaschreibweise]

Aktionsstatement      :=   Statement in Form von Java-Code
```

Legende:

- rot: Zeichen, die lediglich zur Verdeutlichung der Formalisierung benutzt werden und nicht der eigentlichen Syntax entsprechen
- blau: Formalisierungsbeschreibung
- grün: Erläuterungen (ggf. Verweis auf weitere Formalisierungen z.B. Java)
- schwarz: endgültige Werte und Zeichen