AG Software Engineering ◇ Institut für Informatik

Freie Universität Berlin

Entwicklung einer Sprache zur einfachen Beschreibung und Implementierung von Episodenerkennern

Bachelorarbeit (Informatik)

eingereicht durch

Christian A. Kopf (Matr.-#: V3893232)

Institut für Informatik, Freie Universität Berlin

am

18. September 2006

Inhaltsübersicht

1 Abriss....3
2 Einleitung....3
3 Ähnliche Arbeiten....3
4 Beschreibungsmechanismen....4
4.A Reguläre Ausdrücke....5
4.B Zustandsübergangsautomaten....6
4.C Eigener Beschreibungsmechanismus....7
5 Analyse und Vergleich....9
5.A Anforderungen an die Mächtigkeit des Formalismus....9
5.B Aufwand der Eingabe....10
5.C Vorgehen bei der Entwicklung....12
5.D Weitere Anforderungen....12
5.D.I Aufwand der Implementierung....12
5.D.II Unterstützung bei der Fehlersuche....13
5.D.III Effizienz....13
5.D.IV Spezifische Nachteile....14
5.D.V Subjektive Bewertungskriterien....14
5.E Ablauf der Episodenerkennung....15
5.F Überblick....16
6 Folgerungen und Fazit des Vergleichs....17
6.A Erweiterte reguläre Ausdrücke....17
6.B Erweiterte Zustandsübergangsautomaten....18
6.C Eigener, neuer Notationsentwurf....18
6.D Fazit....18
7 Ein komplexeres Beispiel....18
7.A Modellierung als erweiterter Zustandsübergangsautomat (Bsp 7-1a)....19
7.B Modellierung mit Hilfe erweiterter regulärer Ausdrücke (Bsp 7-1b)....20
7.C Modellierung mit der neu entwickelten Notation (Bsp 7-1c)....20
8 Programmiertechnische Umsetzung von Zustandsübergangsautomaten....21
8.A Entwurf eines geeigneten Klassenpakets....21
8.A.I Der Entwurf....21
8.A.II Der Ablauf....22
8.A.III Die Eingabe....24
8.A.IV Zusammenfassung....25
8.B SCXML....26
9 Formalisierung des neu entwickelten Beschreibungsmechanismus (ALED)....27
9.A Formalisierung der Syntax....28
9.B Die drei Abfolgeoperationen....31
9.B.I Chronologischer Sequenzoperator....31
9.B.II Chronologischer Alternativenoperator....32
9.B.III Chronologischer Kombinationsoperator....33
9.B.IV Bindungsstärke und Klammerung....34
9.C Erläuterung und Abbildung der Auftrittshäufigkeiten auf Zustandsübergangsautomaten....35
9.D Weitere Ideen/Ausblick....37
9.D.I Sonderbefehle in den Aktionsanweisungen....38
9.D.II Metainformationen....39
9.D.III Allgemeine Anmerkungen....39
10 Anhänge....39
10.A Begriffserklärungen....39
10.B Literatur- und Quellenhinweise....41
10.C Copyrightangaben zu Beispiel 8.B-1....41
10.D Danksagungen....42
10.E Eidesstattliche Erklärung....42

1 Abriss

Ziel der Bachelorarbeit war die Entwicklung einer Sprache zur einfachen Beschreibung und Implementierung von Episodenerkennern, um diese bei der Entwicklung eines Rahmenwerkes zum Bau und zur Pflege von Episodenerkennern nutzen zu können. Um einem Nutzer sowohl den Entwurf eines Episodenerkenners als auch dessen Eingabe in das erwähnte Rahmenwerk so einfach und intuitiv wie möglich zu machen, ist es notwendig einen geeigneten Beschreibungsmechanismus für Ereignisepisoden zu finden, der beides ermöglicht. In dieser Ausarbeitung vergleiche ich zwei gängige und vielversprechende Beschreibungsmechanismen mit einem völlig Neuen, extra für das vorliegende Problem entworfenen, und versuche ihre Vor- und Nachteile aufzuzeigen. Die Entscheidung für vorerst einen dieser Beschreibungsmechanismen ist für das weitere Vorgehen bei der Entwicklung eines Rahmens zur einfachen Erstellung und Pflege von Episodenerkennern richtungsweisend.

2 Einleitung

Dieses Dokument stellt die Ausarbeitung der Bachelorarbeit „Entwicklung einer Sprache zur einfachen Beschreibung und Implementierung von Episodenerkennern" dar, welche im Rahmen des Studiums des Mikroprozesses der Softwareentwicklung [1] entsteht. Die Episodenerkennung, also das Auffinden und Analysieren von bestimmten Aktivitätsmustern aus einem Strom von Basisereignissen, ist eine bedeutende Voraussetzung zur Untersuchung des Mikroprozesses. Im hier vorliegenden Anwendungsfall werden Aktivitäten eines Softwareentwicklers mit Hilfe der Entwicklungsumgebung Eclipse und dem ElectroCodeoGram (ECG) aufgezeichnet und gespeichert. Es ist nun von Interesse, bestimmte Muster in dem so entstehenden Ereignisstrom zu erkennen und ggf. ein weiteres Ereignis („Muster XYZ erkannt") dem Ereignisstrom hinzuzufügen. Hierbei ist zu beachten, dass solche Episoden teilweise große Komplexität annehmen können (z.B.: Trial-Error-Analyse [2]). Folglich kann es sinnvoll sein, mehrere Episodenerkenner sequentiell nacheinander abzuarbeiten, da sie sich auf Ereignisse beziehen können, die erst von einem anderen Episodenerkenner „geworfen", also entdeckt werden müssen. Ziel der Bachelorarbeit ist es, eine ihr folgende Entwicklung eines Rahmenwerkes zur einfacheren Erstellung und Pflege von Episodenerkennern zu optimieren, bzw. zu ermöglichen. In einem solchen Rahmenwerk soll es möglich sein, Episoden als Module in das ECG zu integrieren. Essentiell für die Lösung dieser Aufgabe ist das Verwenden eines intuitiven und ausreichend mächtigen Beschreibungsmechanismus für Episoden, so dass der spätere Nutzer des Rahmenwerkes schnell und einfach, ohne Einschränkung der Möglichkeiten, Episoden für die Analyse beschreiben kann. Dieses Dokument geht auf einige besonders vielversprechende Beschreibungsmechanismen ein, zeigt ihre Vorteile und ihre Schwachstellen auf und versucht außerdem Empfehlungen für ein weiteres Vorgehen in diesem Bereich zu geben.

3 Ähnliche Arbeiten

Hongbing Kou von der University of Hawai'i und sein Team entschieden sich bei der Entwicklung eines Episodenerkenners für das Tool Zorro [3][4] für die Nutzung eines

regelbasierten Systems, da dieses kleine und einfache Iterationen, an Stelle von komplexen Prozessen, betont. Im Dokument „Studying Micro-Processes in Software Development Stream“ [5] begründet er im Abschnitt 3.3 auf Seite 4 die Entscheidung gegen die Verwendung von endlichen Zustandsautomaten (Finite State Machine, FSM) und für regelbasierte Systeme damit, dass er und das Team von Prof. Philip Johnson den Fokus nicht auf das Aufdecken von Episoden, sondern auf Übereinstimmung mit dem vorgeschriebenen Prozess legten. Insbesondere das von ihm untersuchte Test-Driven Development betont kleine und simple Iterationen. Der entscheidende Unterschied zwischen Kous Arbeit und der Vorliegenden liegt im zukünftigen Anwendungsfokus: Das Team von der Universität Hawai'i beschäftigt sich damit welche Veränderungen an Softwareartefakten vorgenommen wurden, unser Fokus liegt jedoch darauf, wie diese vom Entwickler vorgenommen werden. Da dadurch in unserem Fall häufig komplexe oder umfangreiche Episoden erkannt werden sollen (anders als bei Kou, der einen Entwicklungsprozess bezüglich seiner Durchführung evaluiert), werde ich die Verwendung von deterministischen, endlichen Zustandsautomaten, im weiteren Zustandsübergangsautomaten genannt, in Betracht ziehen.

Das World-Wide-Web-Consortium (W3C) entwickelte im Jahr 2005 eine auf der XML-Technologie beruhende Beschreibungssprache für Zustandsübergangsautomaten nach der Definition von D. Harel [5][6]: SCXML (State Chart XML) [7]. Da Zustandsübergangsautomaten einen geeigneten Beschreibungsmechanismus zur Episodenerkennung darstellen, wie in den folgenden Kapiteln gezeigt wird, bietet diese Sprache Möglichkeiten für eine simplere Umsetzung von Episodenerkennern.

Ferner sei auf zwei Arbeiten hingewiesen, die derzeit an der Freien Universität Berlin unter Prof. Dr. Lutz Prechelt von Hannes Restel [2] und Sofoklis Papadopoulos [8] angefertigt werden und sich mit zwei speziellen Episoden, beziehungsweise Episodenschemata, beschäftigen. Anhand dieser Arbeiten ist leicht zu erkennen, welch weites Spektrum an Möglichkeiten Episodenerkenner bei der Untersuchung des Mikroprozesses der Softwareentwicklung abdecken können müssen.

Des Weiteren wurde darauf geachtet, dass das Umsetzen, der in dieser Arbeit erzielten Ergebnisse, mit der ElectroCodeoGram-Software (ECG) möglich ist. Das ECG entstand im Rahmen der Diplomarbeit von Frank Schlesinger: „Protokollierung von Programmiertätigkeiten in der Eclipse-Umgebung“ [9].

4 Beschreibungsmechanismen

Auf Grund der Tatsache, dass man das Erkennen von Episoden von zwei unterschiedlichen Standpunkten aus betrachten kann, ergeben sich auch zwei grundlegende Möglichkeiten Episoden zu beschreiben. Dem einen Standpunkt zufolge geht man von einer vorhandenen, chronologisch sortierten Menge von Basisereignissen aus, die einem vorher fest definiertem Alphabet entstammen. Diese Menge möchte ich nun auf sequentielle Übereinstimmungen mit einem bestimmten Muster - auch Episodenschema genannt - überprüfen. Dieser Standpunkt legt die Nutzung von regulären Ausdrücken oder ähnlichen Mechanismen zur Beschreibung einer Episode nahe. Ein anderer Standpunkt legt seinen Fokus auf die sequentielle Abarbeitung eines Datenstroms, so dass man festhält in wieweit der bisherige Datenstrom bereits dem gesuchten Muster, also der Episode, entspricht. Je

nach dem, bis zu welcher Stelle das Muster bereits erfüllt ist, wird überprüft, ob das nächste Ereignis auf dem Datenstrom auch einem der nächsten möglichen Ereignisse im gesuchten Muster entspricht. Dieser Standpunkt ist in unserem Fall der Intuitive, da wir auf einem Echtzeit-Datenstrom arbeiten und das just-in-time-Erkennen (auch „on-the-fly" genannt) einer Episode eine primäre Aufgabe des gewünschten Episodenerkenners sein soll. Er kann zum Beispiel recht einfach an Zustandsübergangsautomaten verdeutlicht werden.

Damit liegen uns also bereits zwei Ideen für mögliche Beschreibungsmechanismen von Episoden vor: reguläre Ausdrücke und Zustandsübergangsautomaten. In diesem Kapitel werde ich beide kurz einführen und einen weiteren, selbst entworfenen Beschreibungsmechanismus, vorstellen.

4.A Reguläre Ausdrücke

Reguläre Ausdrücke bilden die Familie der regulären Sprachen und beschreiben eine Menge von Zeichenketten, die bestimmten im regulären Ausdruck festgehaltenen syntaktischen Regeln folgen. Ein solcher regulärer Ausdruck enthält nur Zeichen des zugrunde liegenden Alphabets sowie einige Metazeichen, die Operationen und Ähnliches darstellen. Innerhalb eines regulären Ausdrucks werden drei Operationen genutzt: wiederholen, alternativ und folgend (also das Aneinanderreihen von Teilausdrücken). Außer den Metazeichen steht jedes Zeichen des Alphabets für sich selbst, der Punkt "." steht für ein beliebiges Zeichen des zugrunde liegenden Alphabets.

Um eine mögliche Menge von Alternativen festzulegen werden eckige Klammern ("[" und "]") benutzt, das "-"-Zeichen definiert einen Bereich und "^" negiert eine Angabe.

Beispiel (Bsp 4.A-1): *.[^0-9] heißt: ein beliebiges Zeichen, jedoch keine Ziffer!*

Es gibt außerdem in erweiterten Implementierungen regulärer Ausdrücke vordefinierte Zeichenklassen (z.B.: /d = Dezimalziffern), die jedoch im Allgemeinen nur der Vereinfachung dienen.

Die Anzahl von Wiederholungen wird durch die Angabe von Wiederholungsfaktoren angegeben. Diese stehen in geschweiften Klammern ("{" und "}"). Außerdem gibt es hierfür die Sonderzeichen "?" (optional, also ein oder kein Vorkommen), "+" (mindestens ein Vorkommen) und "*" (beliebig oft, also ggf. auch kein Vorkommen).

Gruppierungen von Zeichen werden mit runden Klammern ("(" und ")") zusammengefasst. In vielen Implementierungen kann auf jede dieser Gruppierungen als Variable zurückgegriffen werden.

Alternativen werden durch "|" voneinander getrennt.

Ein Beispiel (Bsp 4.A-2):

.+(abc|Abc)d[aAbc]{3,4}*

Bedeutet:

Mindestens ein beliebiges Zeichen (ggf. auch mehrere) gefolgt von "abc" oder "Abc", darauffolgend beliebig viele "d" (ggf. auch kein einziges) und anschließend 3 oder 4 Zeichen aus der Menge {a,A,b,c} (mehrfaches Vorkommen des selben Zeichens ist dabei

erlaubt).

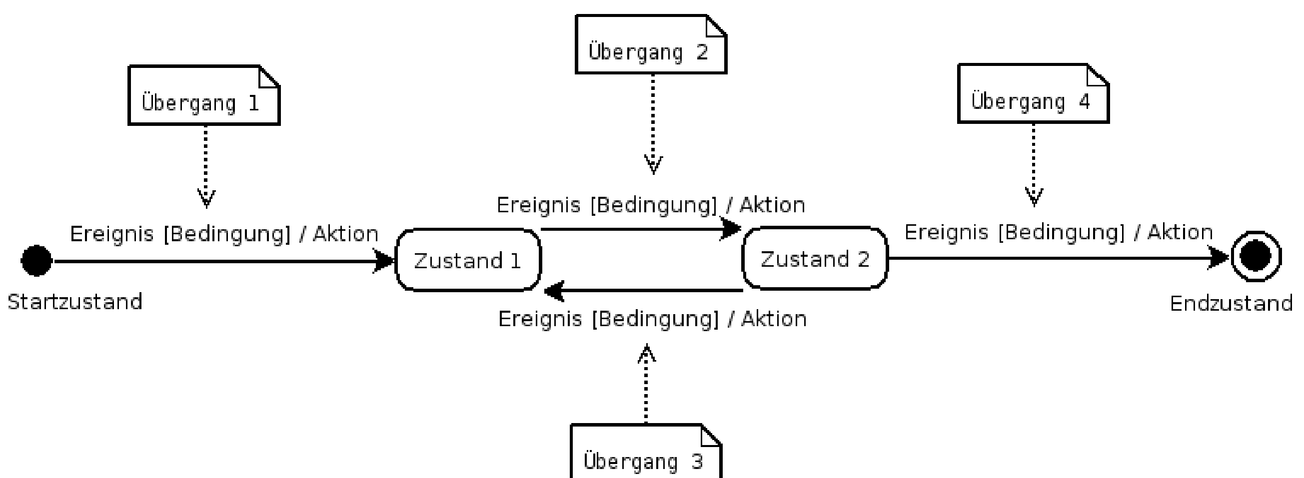
4.B Zustandsübergangsautomaten

Zustandsübergangsautomaten stellen eine weitere Möglichkeit dar, formelle Sprachen zu beschreiben. Ebenso wie bei den regulären Ausdrücken ist es möglich, mit Hilfe einiger Erweiterungen, Sprachen über den Rahmen des Typ3 der Chomsky-Hierarchie hinaus darzustellen. Ebenso wie bei den regulären Ausdrücken nutzen wir von Beginn an eine erweiterte Variante. Zustandsübergangsautomaten sind ein besonders zur Visualisierung von Abläufen geeigneter Beschreibungsmechanismus und finden in vielen Bereichen der Softwaretechnik Verwendung, darunter oft auch in Entwurfsphasen.

Ein solcher Automat wird durch Zustände (also Knoten) und ihre Übergänge, die in der Regel beim Auftreten von Ereignissen stattfinden, dargestellt. Zu jedem Übergang gehört sowohl ein Start- als auch ein Endknoten. Ein Automat besitzt immer einen Startzustand, und mindestens einen Endzustand, nach dessen Erreichen er als erfüllt oder verworfen gilt. In unserer weiteren Darstellung werden nur solche Endzustände betrachtet, bei denen der Automat als erfüllt definiert wird. Sollten Ereignisse zu keinem gültigen Übergang führen, so wird in unserem Beschreibungsmechanismus der Automat automatisch verworfen. Neben dem auslösenden Ereignis kann ein Übergang im erweiterten Modell noch andere Bedingungen (zum Beispiel bestimmte Variablen innerhalb eines Geltungsbereiches oder ähnliches) besitzen und bei seiner Durchführung Aktionen (z.B.: das Setzen von Variablen) ausführen. Damit erreichen wir, wie leicht zu erkennen ist, die Mächtigkeit von sogenannten Kellerautomaten, die alle kontextfreien Sprachen (Chomskyhierarchie-Typ2) darstellen können.

In der UML-Notation stellen Ovale oder Kreise die Zustände dar. Startzustände sind ausgefüllte kleine Kreise; Endzustände kleine ausgefüllte Kreise umgeben mit einem weiteren Kreis. Übergänge werden durch Pfeile dargestellt, die die Richtung des Zustandswechsels angeben.

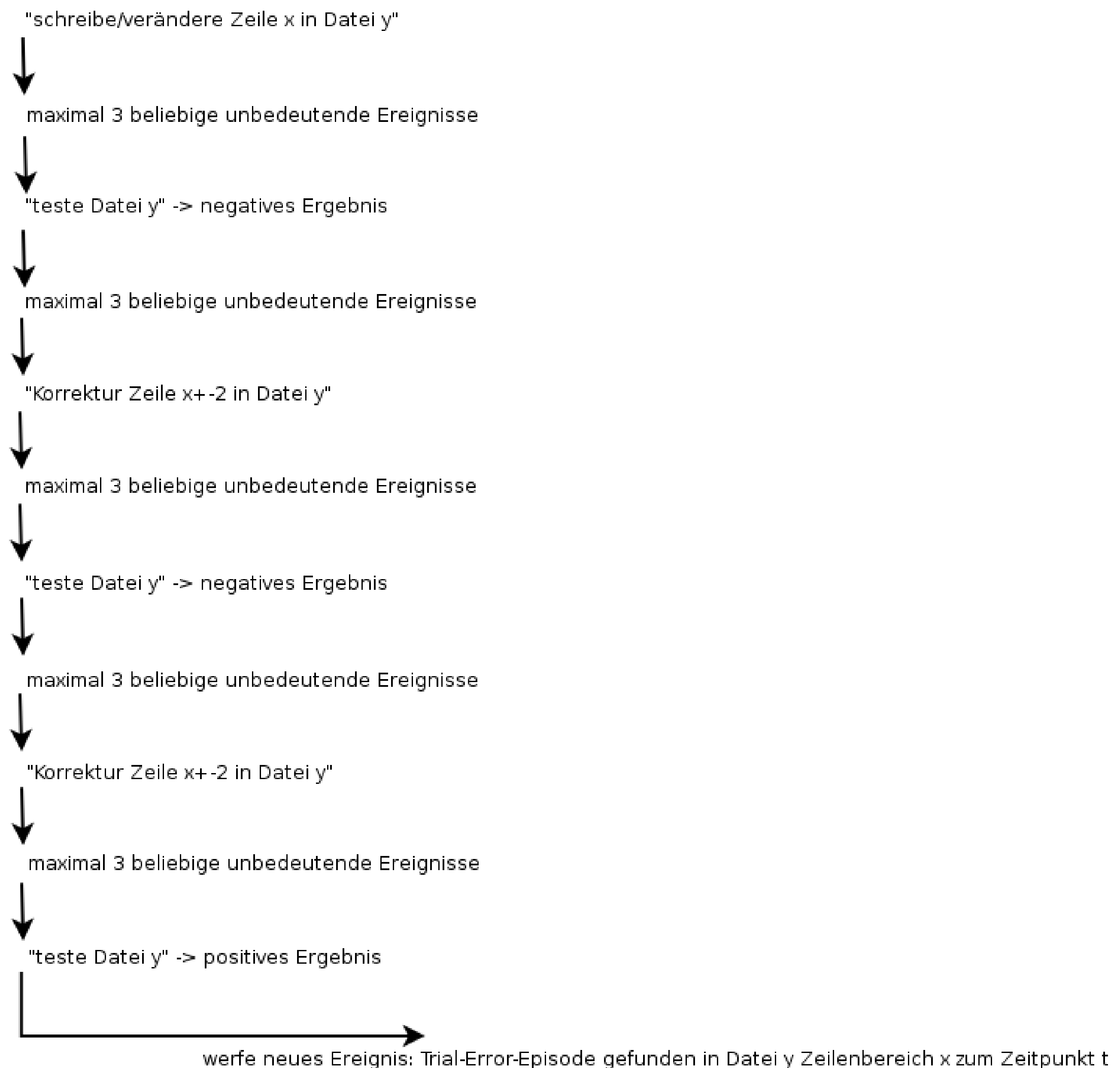
Beispiel (Bsp 4.B-1):



4.C Eigener Beschreibungsmechanismus

Da sowohl reguläre Ausdrücke, als auch Zustandsübergangsautomaten, mit einem anderen Fokus als der Darstellung von Episoden entwickelt wurden, liegt es nahe, einen der intuitiven, ausformulierten Schreibweisen ähnlichen Beschreibungsmechanismus zu entwickeln, welcher möglichst optimal auf dieses Problem ausgerichtet ist. Beim Erarbeiten von Beispielepisoden verwendete ich unbeabsichtigt, also intuitiv, einen solchen Beschreibungsmechanismus. Es erschien der Übersicht halber sinnvoll, auftretende Ereignisse beziehungsweise Ereignismengen von oben nach unten in chronologischer Abfolge zu notieren.

Beispiel (ein positives Ergebnis eines Tests meint hier debug in Eclipse ohne Fehler absolviert) (Bsp 4.C-1):



Aus dieser groben Struktur und mit Hilfe genauerer Kenntnis der benötigten Komplexitäten lässt sich folgende Skriptnotation entwickeln:

- die zeitliche Abfolge wird von oben nach unten dargestellt

- 'int a, int b' meint: Auftreten mindestens a-mal und maximal b-mal (Das Zeichen * steht hierbei für die Wildcard, also "beliebig")

- {...} geschwungene Klammern umschließen Ereignismengen

- <...> spitze Klammern umschließen einzelne Ereignisse

- [...] eckige Klammern umschließen Bedingungen

- eine Ereignismenge besteht aus einzelnen Ereignissen, einer Wildcard (hier: #) ggf. mit Ausschluss einer Ereignismenge oder ist leer, Außerdem kann sie weitere Bedingungen enthalten.

- das Zeichen \ entspricht der Mengenoperation "ohne" bzw. "exklusive"

- das Zeichen / stellt den Beginn der Aktionsanweisungen dar, die bei Auftreten eines gültigen Ereignisses durchgeführt werden sollen, respektive müssen

- eine Zeile repräsentiert ein Vergleichsmuster und besteht aus Häufigkeitsangabe des Auftretens, Ereignismenge (zum Vergleich) und möglichen Aktionen (letztere beiden dürfen leer sein, ersteres ist als '0,#' darzustellen, falls beliebig)

- event.xyz spricht das Attribut xyz des soeben erhaltenen Ereignisses an.

Als weitere Operatoren sind für die Mengendarstellung hilfreich:

+ als "Vereinigung"

~ als "Schnittmenge"

Diese Notation wird in der weiteren Arbeit auch als ALED (Automaton-Language for Episode Description) bezeichnet. Eine ausführliche Formalisierung ist in Kapitel 9 dieser Arbeit zu finden.

ein einfaches Beispiel (Bsp 4.C-2):

```
'1, 1' {<fileopen [true]} / String datei = event.file; int count = 0;
'0,*' {# \ {<fileclose> [event.file == datei]} [true]} / count++;
'1,1' {<codechange> [event.file == datei]} / count++;
'0,*' {# [true]} / count++;
'1,1' {<fileclose> [event.file == datei]} / throw '<open-change-close>,
file=datei, numberofevents=count';
```

Diese Beispielepisode wirft ein neues Ereignis, sobald eine Datei geöffnet, mindestens einmal geändert und wieder geschlossen wurde. Attribute des neuen Ereignisses sind die Datei auf die sich die Episode bezieht und die Anzahl der aufgetretenen Ereignisse, während diese Datei geöffnet war. Die Episode aus Bsp 4 ist unter Bsp 8.3 in dieser Schreibweise notiert.

5 Analyse und Vergleich

Um die oben vorgestellten Beschreibungsmechanismen zu vergleichen, also ihre jeweiligen Vor- und Nachteile bezogen auf die Darstellung von Ereignisepisoden beschreiben zu können, ist es von essentieller Bedeutung, die möglichen Anforderungen zu verdeutlichen.

5.A Anforderungen an die Mächtigkeit des Formalismus

Um festzustellen, ob ein Beschreibungsmechanismus geeignet ist beliebige Episoden zu repräsentieren, ist es notwendig, die mögliche Komplexität einer Episode abzuschätzen. Das folgende Beispiel wird zeigen, dass ein Beschreibungsmechanismus, der lediglich erlaubt reguläre Sprachen (Sprachen des Typ 3 der Chomsky-Hierarchie) darzustellen, für unsere Aufgabe nicht ausreichend ist:

Folgende Typ2-Sprache könnte zum Beispiel eine Episode beschreiben, die dann existiert, wenn jede geöffnete Datei auch wieder geschlossen wurde (stark vereinfacht) *(Bsp 5.A-1)*:

L:=A^n B^n (mit n>=1 und n Element aus Z)

Diese Sprache enthält alle Wörter, die aus einer Anzahl aus A gefolgt von der selben Anzahl B besteht.

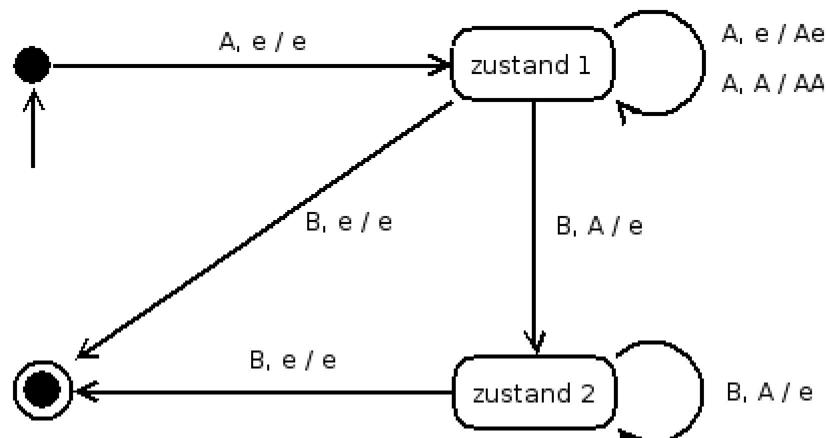
Da während der Untersuchungen keine Episoden erarbeitet werden konnten, die über die Anforderung von Typ2-Sprachen hinausgingen, stellt sich nun folgende Frage:

Ist es möglich, eine solche Sprache mit Hilfe der oben eingeführten Beschreibungsmechanismen darzustellen?

Erweiterte reguläre Ausdrücke: Zwar lässt sich unter Verwendung von regulären Ausdrücken der Rahmen der Typ-3-Sprachen leicht überschreiten (wie das Pumping Lemma auf diesem Beispiel zeigt: (A*)B*\1), jedoch lässt sich kein Zähler, hier n, darstellen oder verwenden. Damit sind reguläre Ausdrücke nicht geeignet, Episoden zu beschreiben, die Konstrukte wie das der Sprache L enthalten.

Erweiterte Zustandsübergangsautomaten: Der folgende Kellerautomat entspricht in etwa der obigen Definition und erkennt alle Wörter der Sprache L des zugrundeliegenden Alphabetes. e sei das leere Wort:

(Bsp 5.A-1a)



eigene Notation (ALED): Folgende Formulierung beschreibt die Sprache L und zeigt damit,

dass ALED an die benötigte Komplexität heranreicht:

(Bsp 5.A-1b)

'1, 1' {<A>[true]} / int count = 0;

'0, *' {<A>[true]} / count++;

'0, *' {<B>[count > 1]} / count--;

'1,1' {<B>[count == 0]} / throw '<elem_L>';

Aufgrund der Tatsache, dass die Aktionsanweisungen (Erläuterung und Formalisierung ab Seite 18) hier Java-Anweisungen sind, erreichen wir Turing-Mächtigkeit. Betrachten wir die Notation jedoch indem wir die Aktionsanweisungen nicht als Java-Code, sondern als Kelleranweisungen notieren, bleibt die benötigte Typ2-Mächtigkeit dennoch erhalten:

(Bsp 5.A-1c)

'1, 1' {<A>[top == e]} /

'0, *' {<A>[(top == e) || (top == A)]} / push A

'0, *' {<B>[top == A]} / pop

'1,1' {<B>[top == e]} /

5.B Aufwand der Eingabe

Abgesehen von einer ausreichenden Mächtigkeit sind auch die Auswirkungen auf die Nutzbarkeit und Realisierbarkeit des resultierenden Rahmenwerkes von großer Bedeutung für die Auswahl eines geeigneten Beschreibungsmechanismus. Insbesondere der Aufwand bei der Erstellung eines neuen Episodenerkenners ist entscheidend. Hierbei sind einige Eigenschaften von Episodenerkennern zu berücksichtigen. So kann eine semantische Episode mehrere syntaktische Formen, im Weiteren als Alternativen bezeichnet, besitzen. Zum Beispiel muss ein Episodenerkenner für die Verfolgung von Codekopien sowohl Copy-Change-Episoden für ganze Methoden, als auch solche für kleinere Codefragmente oder gar ganze Klassen erkennen. Da diese zwar im chronologischen Verlauf ähnlich sein können, auf jeden Fall jedoch unterschiedliche Ereignisse nutzen, ist es ggf. nötig diese als eigene Episodenerkenner zu implementieren. Jedoch ist es unter Umständen wünschenswert, diesen „Satz von Episodenerkennern" gemeinsam als Copy-Change-Episodenerkenner zu betrachten. So gesehen besteht ein Episodenerkenner ggf. aus mehreren kleineren Episodenerkennern, welche jeweils eine der möglichen Alternativen erkennen (Dieses lässt sich zum Beispiel unter Verwendung des Kompositummusters realisieren, siehe auch Seite 14.). Jedoch hat diese Situation direkte Auswirkungen auf das spätere Umsetzen der formalen Beschreibung, da in solchen Fällen mehrere eigentliche Erkenner eingegeben werden müssen. Wenn also der Aufwand für das Realisieren eines Erkenners durch Eingabe eines Zustandsübergangsautomaten mehr Aufwand benötigt als selbige in Form eines regulären Ausdruckes, kann sich hier der Mehraufwand vervielfachen.

Beispiel (5.B-1): Nehmen wir an, das Realisieren(Entwurf und Eingabe in das Rahmenwerk) einer Episode als Zustandsübergangsautomat würde um 50% mehr

Aufwand kosten, als selbige Realisierung als regulärer Ausdruck. Nehmen wir ferner an, dass nun eine Episode aus 5 Alternativen besteht, dann ist es leicht ersichtlich, dass der Mehraufwand in der Realisierung ein hohes Maß an Zeit und Arbeit beanspruchen kann.

Folglich ist die folgende Frage von Bedeutung bei der Wahl eines geeigneten Beschreibungsmechnanismus:

Wie viel objektive Arbeit entsteht in der Beschreibung und Realisierung eines Episodenerkenners bei den unterschiedlichen Beschreibungsformalismen? Das heißt allgemeiner: Wie hoch ist der Aufwand für den Entwurf und die spätere Eingabe eines Episodenerkenners?

Erweiterte reguläre Ausdrücke: Im Extremfall wäre für jede Alternative ein eigener regulärer Ausdruck notwendig. Da die Abarbeitung der regulären Ausdrücke durch bereits existierende Bibliotheken durchgeführt werden könnte, wäre lediglich das Schreiben der Ausdrücke nötig. Der größere Mehraufwand läge hier eindeutig beim Entwurf der regulären Ausdrücke und nicht bei deren Eingabe in das Rahmenwerk.

Beispiel (5.B-2): Möchte ich also einen Episodenerkenner bauen der aus fünf Alternativen besteht, so müsste ich im Extremfall (alle Alternativen sind völlig unterschiedlich, haben also keine gemeinsamen Teilsequenzen) für jede Alternative den regulären Ausdruck entwickeln (der Aufwand hierfür ist schwer abschätzbar, da subjektiv) und dann lediglich die regulären Ausdrücke in das Rahmenwerk eingeben (vergleichsweise sehr geringer Aufwand).

Erweiterte Zustandsübergangsautomaten: Für jede Alternative ist es unter Umständen nötig, einen eigenen Automaten zu formulieren, dessen Implementierung teilweise recht komplexe Objektstrukturen benötigt. Wollten wir zum Beispiel eine Episode erkennen lassen, für die zwei völlig voneinander verschiedene Alternativen mit jeweils relativ großer Länge existieren, so wäre es bei dem in Kapitel 8.A erläuterten Entwurf notwendig, zwei Automaten zu implementieren. Das ist - wie man unschwer am Entwurf (Kapitel 8.A) erkennen kann - vergleichsweise aufwendig. Also entsteht hierbei ggf. ein nicht unerheblicher Mehraufwand bei der Umsetzung im Rahmenwerk, da mehrfach Klassen erweitert und Objekte erzeugt werden müssen.

Eine Möglichkeit Zustandsübergangsautomaten in vergleichsweise einfacher Form programmiertechnisch umzusetzen, bietet State Chart XML (SCXML; kurze Vorstellung in Kapitel 8.B). Mit Hilfe einer Java SCXML Engine, wie zum Beispiel Commons SCXML [10], lassen sich in Zukunft aus SCXML Dokumenten automatisch objektorientierte (Java) Zustandsübergangsautomaten generieren. Dadurch wird der Mehraufwand stark verringert.

Eigene Notation (ALED): Ähnlich wie bei den regulären Ausdrücken ist es auch hier gegebenenfalls sinnvoll für jede Alternative eine eigene Beschreibung (je eine ALED-Datei [siehe Kapitel 9]) anzufertigen. Alternativ dazu würde auch eine Beschreibung mit den entsprechenden Abfolgeoperatoren (siehe dazu Abschnitt 9.A und 9.B) genügen. Im Extremfall wäre der Schreibaufwand in beiden Fällen ungefähr gleich groß. Auf Grund der größeren Darstellung ist somit eine geringfügige Mehrarbeit als bei den regulären Ausdrücken zu erwarten. Dennoch ist die Eingabe in das Rahmenwerk recht simpel und auch zeitlich unproblematisch.

5.C Vorgehen bei der Entwicklung

Für diese Arbeit lag ein weiterer Schwerpunkt darauf, die Entwicklung und Umsetzung eines Episodenerkenners in einem Arbeitsschritt durchführen zu können, das heißt, dass der Entwickler möglichst direkt in der später verwendeten Darstellung entwickelt.

Erweiterte reguläre Ausdrücke: Das Entwickeln einer Episode direkt in der Schreibweise von regulären Ausdrücken ist je nach individuellen Fähigkeiten zwar möglich, bleibt jedoch unübersichtlich. Der fertige Ausdruck könnte allerdings ohne Weiteres in einem Rahmenwerk für reguläre Ausdrücke direkt umgesetzt werden, ohne dass er vom Entwickler selbst in den Programmcode übersetzt werden muss.

Erweiterte Zustandsübergangsautomaten: Bei Automaten muss die grafische Abbildung des Entwurfs vom Entwickler entweder in SCXML übersetzt oder aber direkt in ein Klassenmodell, wie das im Kapitel „bisheriger Entwurf (Zustandsübergangsautomaten)" beschriebene, umgebaut werden. Eine Entwicklung des Episodenschemas direkt in SCXML ist schwer handzuhaben. Das direkte Entwickeln in einer turing-mächtigen Programmiersprache ist ebenfalls vergleichsweise aufwendig. In beiden Fällen ist jedoch die Abstraktion zwischen dem Entwickeln von Episoden und dem Programmieren nicht im wünschenswerten Maße gegeben.

Eigene Notation (ALED): Das Schaffen einer solchen Abstraktion war Ziel bei der Entwicklung von ALED. Dem Entwickler eines Episodenerkenners sollte es möglich sein, das benötigte Episodenschema in ALED zu entwickeln und anschließend keine Übersetzungen in andere computerverständliche Sprachen oder Dialekte vornehmen zu müssen. Tatsächlich ist es nach wie vor notwendig, Bedingungen und Aktionsanweisungen in einer anderen Notation, wie zum Beispiel Kellernotation oder unter Verwendung einer turing-mächtigen Sprache, zu formulieren. Da dies jedoch direkt während des Entwicklungsprozesses geschieht und die Wahl der Notation (also des ALED-Dialekts/der zugrunde liegenden Programmiersprache) dem Entwickler überlassen ist, erfüllt ALED am ehesten die Ziele der Arbeit, bezogen auf das Vorgehen bei der Entwicklung eines Episodenerkenners.

5.D Weitere Anforderungen

Im Weiteren werde ich kurz auf einige weitere Anforderungen und Eigenschaften eingehen, die für die Auswahl eines geeigneten Beschreibungsmechanismus eine Rolle spielen. Sie konnten im Rahmen der Bachelorarbeit nicht schwerpunktsmäßig untersucht werden und werden folglich weniger ausführlich betrachtet.

5.D.I Aufwand der Implementierung

Ebenfalls nicht ganz unbedeutend ist der Implementierungsaufwand und die Radikalität des Vorgehens, die bei der Entwicklung des Rahmenwerkes anfallen würden. Dabei ist zu berücksichtigen, inwiefern Wiederverwendung bzw. Reimplementierung in Betracht kommen und abzuschätzen, wie komplex die Implementierung wird.

Erweiterte reguläre Ausdrücke: Für reguläre Ausdrücke existieren zahlreiche Bibliotheken für beinahe jede gängige Programmiersprache. Viele davon enthalten bereits einige

Erweiterungen. Somit ist ein hoher Grad an Wiederverwendung möglich. Der Implementierungsaufwand würde sich vor Allem darauf konzentrieren, nach jedem neu aus dem Datenstrom erhaltenen Ereignis, den bisher empfangenen Datenstrom zu "scannen". Darauf wird später noch genauer eingegangen.

Erweiterte Zustandsübergangsautomaten: Bereits zu einem früheren Zeitpunkt meiner Bachelorarbeit habe ich einen groben Rahmen aus Klassen und Schnittstellen vorgestellt, um Episodenerkenner mit Hilfe dieses Beschreibungsmechanismus zu implementieren. Dieser Rahmen müsste noch erweitert und eingebunden werden. Da hier einige Klassen und Methoden erst durch den Nutzer erweitert und geschrieben werden, halten sich die weiteren notwendigen Implementierungsarbeiten (für die Entwicklung des Rahmenwerks) in Grenzen. Nutzt man für die Entwicklung eine bereits existierende Engine (wie zum Beispiel Common SCXML) um SCXML-Beschreibungen von Automaten zu verwenden, so ist der Implementierungsaufwand nochmals deutlich geringer, da lediglich eine Einbindung in das ECG vorzunehmen ist.

Eigene Notation (ALED): Da ALED noch nicht verbreitet ist, ist hier eine komplette Implementierung nötig. Dabei ist zunächst ein Parser zu schreiben. Dieser könnte dann die Notation in SCXML oder Ähnliches umformen oder auch direkt einen lauffähigen Episodenerkenner erzeugen. Auf jeden Fall benötigt dieser Beschreibungsmechanismus die meiste Implementierungsarbeit.

5.D.II Unterstützung bei der Fehlersuche

Da der Entwurf eines Episodenerkenners semantisch oft schwierig ist und es bei der Umsetzung in einen Beschreibungsmechanismus leicht zu Fehlern kommen kann, liegt ein weiterer Fokus bei der Entwicklung eines geeigneten Rahmenwerkes auf den Möglichkeiten zur Unterstützung der Defektvermeidung und -behebung während der Nutzung. Welche Möglichkeiten im späteren Rahmenwerk zur Verfügung stehen, hängt nicht zuletzt auch von der Auswahl des Beschreibungsmechanismus ab. Da bisher lediglich für die Zustandsübergangsautomaten ein Rahmenwerksentwurf vorliegt, gestaltet sich der Vergleich der Beschreibungsmechanismen in diesem Punkt jedoch schwierig und beruht fast ausschließlich auf Vermutungen. Da die Aktionsanweisungen von ALED in Java-Syntax geschrieben werden können, lässt er sich besonders einfach mit Hilfe von Codegenerierung umsetzen. Hierbei ist zu beachten, dass dieses später die Möglichkeiten zur Unterstützung bei der Defektvermeidung und -behebung beeinträchtigen kann.

5.D.III Effizienz

Ein offensichtliches Kriterium, die Auswahl eines Formalismus betreffend, scheint seine Effizienz, beziehungsweise die Effizienz der späteren Implementierung zu sein. Auch hier lassen sich kaum Aussagen treffen, da keine funktionstauglichen Rahmenwerke vorliegen. Es ist davon auszugehen, dass bei der späteren Anwendung mehrere Episodenerkenner gleichzeitig genutzt werden und der Datenstrom mitunter viele schnell aufeinanderfolgende Ereignisse beinhaltet, weshalb eine hohe Effizienz von Vorteil ist.

5.D.IV Spezifische Nachteile

Einige Nachteile die sich auf ein Rahmenwerk auswirken können sind spezifisch und resultieren entweder aus den ursprünglichen Anwendungsgebieten oder aus der zugrunde liegenden Struktur der Beschreibungsmechanismen.

Erweiterte reguläre Ausdrücke: Bei der Verwendung von regulären Ausdrücken zur Episodenerkennung stellt die notwendige Flexibilität des zugrunde liegenden Alphabetes ein weiteres Problem dar: Jedem Basisereignis müsste ein Kürzel oder ein Zeichen zugewiesen werden. Dabei kann die Anzahl an möglichen Basisereignissen bei der Implementierung leicht die Anzahl verfügbarer Zeichen überschreiten.

Beispiel (5.D.IV-1): Selbst wenn man jedes ASCII-Zeichen zur Abbildung eines Ereignisses verwenden könnte, wären maximal 128 unterschiedliche Ereignisse möglich. Würde man bei der Abbildung auf mehr als ein Zeichen beinhaltende Zeichenketten zurückgreifen, so würden reguläre Ausdrücke schnell unübersichtlich.

Eine große Menge an unterschiedlichen möglichen Ereignissen ist also bei regulären Ausdrücken ausgesprochen unhandlich.

Erweiterte Zustandsübergangsautomaten: Insbesondere bei längeren zu erkennenden Episoden werden in der Darstellung als Zustandsübergangsautomat wahrscheinlich viele Objekte benötigt, da sämtliche Übergänge als Objekt modelliert werden müssen. (Erläuterungen des zu Grunde liegenden Entwurfes folgen weiter unten. Zum Rahmenwerk von Commons SCXML kann ich keine Aussage treffen, da ich keinerlei Codeeinsicht genommen habe.) Verwendet man SCXML, so ist darauf hinzuweisen, dass die Fähigkeiten von SCXML weit über die geforderten hinausgehen und somit eine Menge von Notationsanweisungen existieren, die für die Beschreibung von Episoden überflüssig sind.

Eigene Notation (ALED): Die spezifische Schwäche der neu entwickelten Darstellung besteht darin, dass es rasch zu Doppeldeutigkeiten kommen kann. So müssen zum Beispiel zwei aufeinander folgende Zeilen mit der Auftrittshäufigkeit '0,#' disjunkte Ereignismengen besitzen. Zu lösen ist dieses Problem zwar recht einfach, hier durch Vereinigen der beiden Zeilen zu einer, jedoch stellt es ggf. eine Schwierigkeit dar, die beim Entwurf beachtet werden muss.

5.D.V Subjektive Bewertungskriterien

Subjektive Bewertungskriterien wie zum Beispiel die Erlernbarkeit, die Verständlichkeit und die Nachvollziehbarkeit sind bei der Entwicklung eines Rahmenwerkes zur einfachen Erstellung und Pflege von Episodenerkennern besonders wichtig. Denn der Fokus bei einer solchen Arbeit liegt darauf, dem späteren Nutzer des Rahmenwerkes die Möglichkeit zu geben, erdachte Episoden intuitiv, schnell und ohne großen Aufwand als ECG-Modul einzubinden.

Solche Bewertungskriterien sind allerdings relativ schwer zu messen.

Sowohl reguläre Ausdrücke als auch Zustandsübergangsautomaten sind in der Informatik lange etablierte und bekannte Beschreibungsmechanismen, die jedoch nicht optimal auf das Themenfeld der Episodenerkennung angepasst sind. Der neue Beschreibungsmechanismus hingegen ist aus dem Wunsch entstanden, Episoden

beschreiben zu können, jedoch Außenstehenden noch völlig unbekannt. Entgegen den Erwartungen hat sich allerdings bei den ersten Vorstellungen vor Kommilitonen gezeigt, dass der neue Notationsentwurf als intuitiver und weniger stark abstrakt empfunden wird, als vorher angenommen.

5.E Ablauf der Episodenerkennung

Neben den Anforderungen haben auch die sich aus den unterschiedlichen Beschreibungsmechanismen ergebenden Abläufe während der Episodenerkennung starken Einfluss auf deren Eignung für diese Aufgabe. Da davon auszugehen ist, dass zum einen der bereits empfangene Datenstrom relativ lang sein kann, zum zweiten viele Ereignisse (auch Lärm) in kurzen Zeiträumen auftreten und on-the-fly in den Erkennungsprozess eingebunden werden sollen und drittens mehrere, wahrscheinlich sogar relativ viele Episodenerkenner, sequentiell nacheinander "geschaltet" sein werden, ist die Geschwindigkeit möglichst hoch und der für eine Episodenanalyse nötige Rechenaufwand möglichst niedrig zu halten. Auch hier unterscheiden sich die drei betrachteten Varianten erheblich:

<u>Erweiterte reguläre Ausdrücke:</u> Reguläre Ausdrücke sind an sich eher dazu geeignet, auf statischen, das heißt bereits abgeschlossenen Datenmengen, Vergleiche zu einer Struktur anzustellen. In unserem Fall soll allerdings ein dynamischer Datenstrom analysiert werden, also on-the-fly. Bei einer Umsetzung mit regulären Ausdrücken müsste jeder Episodenerkenner nach jedem neu aus dem Datenstrom erhaltenen Ereignis, jeden Teilstring des bisher empfangenen Datenstroms, der an diesem neuen Ereignis endet, erneut überprüfen.

Ein Beispiel (5.E-1): Der bereits empfangene Datenstrom ist 'ABBDCF'; das neue Ereignis ist 'G'; wir suchen nach dem Schema '(^A)IG'. Der dazugehörige Erkenner müsste nun folgende Strings mit dem Schema vergleichen: 'ABBDCFG', 'BBDCFG', 'BDCFG', 'DCFG', 'CFG', 'FG', 'G'. (Das Beispiel ist stark vereinfacht!)

Dieses Problem lässt sich durch die Verwendung von sogenannten „Fenstern" vermeiden, wobei immer nur ein String der letzten x Zeichen (x fest, aber episodenabhängig, also für jeden Episodenerkenner statisch) untersucht wird. Allerdings ist auch diese Lösung nicht praktikabel, da Episodenerkenner teilweise sehr weit auseinander liegende Ereignisse analysieren sollen.In einer „Dateiaktivitäts-Episode" können beispielsweise zwischen zwei bedeutenden auch viele „uninteressante" Ereignisse (sogenannter Lärm) auftreten.

<u>Erweiterte Zustandsübergangsautomaten:</u> (betrachtet wird der Entwurf aus Kapitel 8.A.) Hierbei müssten lediglich beim Eintreffen eines neuen Ereignisses aus dem Datenstrom alle Übergänge vom derzeit aktiven Zustand überprüft werden. Diese Menge ist selbst bei komplexen Automaten nicht sonderlich groß, und stets unabhängig von der Länge des bisher empfangenen Datenstroms. Gegebenenfalls kann es vorkommen, dass mehrere Instanzen eines Erkenners vorliegen. Dieses kann immer dann der Fall sein, wenn ein neues Ereignis des Datenstroms einen Übergang aus dem Startzustand des Automaten auslöst. Im Regelfall bleibt die Anzahl von gleichzeitig aktiven, aber evtl. in verschiedenen Zuständen befindlichen Instanzen gleicher Erkennerklassen, allerdings überschaubar.

<u>Eigene Notation (ALED):</u> Bei ALED ist der Ablauf stark von der zugrunde liegenden

Umsetzung abhängig. Nimmt man zum Beispiel an, dass man auch hier mit Zuständen arbeitet, das heißt, man weiß, in welcher Zeile man sich bisher befindet, so müssen lediglich die noch zulässigen Zeilen überprüft werden. Also ggf. die Zeile in der wir uns befinden (bei mehrfachem erlaubten Auftreten), sowie die darunter liegenden Zeilen (falls die nächste Zeile auch kein Auftreten, also Häufigkeit Null, erlaubt, muss auch die übernächste Zeile betrachtet werden, usw.). Es ist darauf zu achten, dass ggf., wie bei den Automaten, mehrere Instanzen eines Erkenners vorliegen können.

5.F Überblick

	Erw. Reguläre Ausdrücke	***Erw. Zustands-übergangs-automaten***	***Eigener, neuer Notationsentwurf (ALED)***
Subjektive Bewertungs-kriterien	(+) allg. in der Informatik bekannt (-) sicherer Umgang und gute Kenntnisse nötig	(+) leicht zugänglich (-) aufwendiger Entwurf	(+) problemspezifisch (-) komplett neu, und bisher nicht bekannt
Spezifisches	- Verwaltung des Alphabets gestaltet sich schwierig	- große Objektmengen möglich - SCXML -> viele überflüssige Features	- noch nicht ganz ausgereift
Erkennungs-Ablauf	- ggf. sehr hohe Anzahl an Vergleichen nötig, da an sich nicht dynamisch	- mehrere Instanzen des selben Episoden-erkenners möglich	- entweder hohe Anzahl an Vergleichen oder mehrere Instanzen möglich (Abhängig von Implementierung)
Implementier-ungsaufwand für das Rahmenwerk	+ geringer Implementier-ungsaufwand, da zahlreiche Bibliotheken vorhanden	* grobe Struktur bereits implementiert, Logik fehlt noch + unkomplizierte Umsetzung in Commons SCXML	- hoher Implementier-ungsaufwand (Parser, etc.)
Aufwand für Entwickler von Episoden-erkennern	+ geringer Eingabeaufwand - Entwurf gestaltet sich schwierig	- relativ hoher Eingabe- bzw. Umsetzungs-aufwand + leichterer Entwurf	+ noch relativ niedriger Eingabeaufwand + vgl. geringer Entwurfsaufwand + Entwurf = Umsetzung

	Erw. Reguläre Ausdrücke	***Erw. Zustands-übergangs-automaten***	***Eigener, neuer Notationsentwurf (ALED)***
Mächtigkeit	- keine komplette Typ2-Mächtigkeit - Modellierung von Bedingungen und Aktionen nur begrenzt möglich	+ mindestens Typ2-Mächtigkeit	+ mindestens Typ2-Mächtigkeit

6 Folgerungen und Fazit des Vergleichs

Wie bereits im letzten Abschnitt beschrieben und aus der Tabelle ersichtlich, bietet die neue Darstellungsform einige Vorteile gegenüber den anderen beiden Beschreibungsmechanismen. Nachteile, wie unter Anderem die benötigte Sicherheit im Umgang mit regulären Ausdrücken, deren unzureichende Mächtigkeit, sowie der arbeitsaufwendige Entwurf unter Zuhilfenahme von Zustandsübergangsautomaten, sprechen für eine weitere Verfolgung des neu entwickelten Ansatzes. In diesem Kapitel möchte ich nochmals kurz auf wichtige Merkmale der unterschiedlichen Beschreibungsmechanismen zurückkommen und meine Entscheidung für den neu entwickelten Beschreibungsmechanismus (ALED) begründen.

6.A Erweiterte reguläre Ausdrücke

Insbesondere wegen der zu geringen Mächtigkeit bzw. dem hohen nötigen Aufwand zur Steigerung dieser und die nicht auf dynamische Überprüfung ausgelegte Struktur, sprechen gegen reguläre Ausdrücke als allgemeinen Beschreibungsformalismus für Episodenerkenner. Auch die Tatsache, dass sich der sichere Umgang mit regulären Ausdrücken bei komplexeren Episoden schwierig und unübersichtlich gestaltet, ist nicht optimal. Allerdings eignen sie sich aufgrund der vorhandenen effizienten Implementierungen für den direkten Vergleich von Ereignissen (z.B. neues Datenstrom-Ereignis wird mit Ereignismuster für Zustandsübergang verglichen). Sie könnten deshalb sowohl in einer möglichen Implementierung des Zustandsübergangsautomaten-Beschreibungsmechanismus als auch in einer Umsetzung der neuen Notationsform benötigt werden.

Beispiel (6.A-1): Um von Zustand eins zu Zustand zwei zu gelangen, wird das Auftreten eines Ereignisses des Typs <run-debug> mit der Bedingung [event.file == „a.txt"] benötigt. Das nächste im Datenstrom auftretende Ereignis (angenommen: <file-close> mit dem Attribut file="a.txt") muss nun mit dem Ereignistyp <run-debug>[event.file=="a.txt"] verglichen werden. Diese Aufgabe kann ggf. von regulären Ausdrücken übernommen werden. Zuerst sollte dabei der Ereignisname (hier also file-close mit run-debug) verglichen werden. Bei Übereinstimmung müssen dann anschließend noch die weiteren Bedingungen (hier: file ist gleich a.txt) überprüft werden. Es ist zu bemerken, dass sowohl exclude-Fälle (also alle Ereignisse, außer jene, die <run-debug> entsprechen, führen zu Zustand zwei) als auch include-Fälle (also nur Ereignisse, die <run-debug> entsprechen führen zu Zustand zwei) auftreten können. (Exclude-Fälle

werden in der neuen Notationsform wie folgt dargestellt: #\<run-debug>)

6.B Erweiterte Zustandsübergangsautomaten

Zustandsübergangsautomaten stellen eine leicht verständliche und einfach nachvollziehbare Möglichkeit dar, Episoden visuell zu modellieren. Auf Grund der teilweise recht großen benötigten Objektmengen und dem hohen Eingabeaufwand für Episodenerkenner eignen sie sich jedoch nur bedingt für die Nutzung im Rahmenwerk.

6.C Eigener, neuer Notationsentwurf

Insbesondere die Tatsache, dass ALED einen geringen Entwurfs- und Umsetzungsaufwand für den Endbenutzer des Rahmenwerkes mit mindestens Typ2-Mächtigkeit verbindet, ist hervorzuheben. Ebenso zeichnet sich der neue Beschreibungsmechanismus durch seine Abstraktion von Entwicklung des Episodenschemas und Programmiertätigkeit aus. Die einzige Aufgabe des Episodenentwicklers besteht im Entwurf des Episodenschemas, da der neue Beschreibungsmechanismus direkt in ein passendes Rahmenwerk eingegeben werden kann.

6.D Fazit

Entscheidend für das weitere Vorgehen ist also der Entwicklungsfokus. Soll "lediglich" ein möglichst intuitiver Entwurfsmechanismus in ein Modul umgesetzt werden, sind Zustandsübergangsautomaten die erste Wahl. Ist jedoch das Ziel eher ein Rahmenwerk zur Verfügung zu stellen, in das man besonders schnell bereits entwickelte Episodenerkenner integrieren kann (also besonders einfache Einbindung eines neuen Episodenerkenners), so sind reguläre Ausdrücke zu bevorzugen (Reguläre Ausdrücke würden aufgrund der geringen Mächtigkeit auch hierfür nicht ausreichen). Die neue Entwurfsnotation versucht beiden Ansätzen gerecht zu werden, dafür nimmt man allerdings in Kauf, einen bis dato komplett unbekannten Beschreibungsmechanismus zu verwenden. Dafür ermöglicht der neue Ansatz dem späteren Benutzer des Rahmenwerkes das Arbeiten mit einem intuitiven Entwurfsmuster und gleichzeitig eine einfache Einbindung in das ECG. Es ist außerdem noch festzustellen, welche Muster bei der Anwendung bevorzugt werden, wenn Nutzer alle drei Beschreibungsmechanismen beherrschen. In jedem Fall ist bei der Entwicklung einer geeigneten Software darauf zu achten, dass ein breites Spektrum an unterschiedlichen Episoden für eine Analyse in Frage kommt. Zum Beispiel können sich Episoden stark in ihrer Dauer unterscheiden, was unter Umständen eine gewisse Persistenz nötig macht. Auch die bereits erwähnte Tatsache, dass mehrere Episoden eines Episodenschemas überlappend existieren und somit gegebenenfalls mehrere Instanzen des dazu gehörigen Automaten (mit unterschiedlichen aktuellen Zuständen) vorliegen können, ist zu berücksichtigen.

7 Ein komplexeres Beispiel

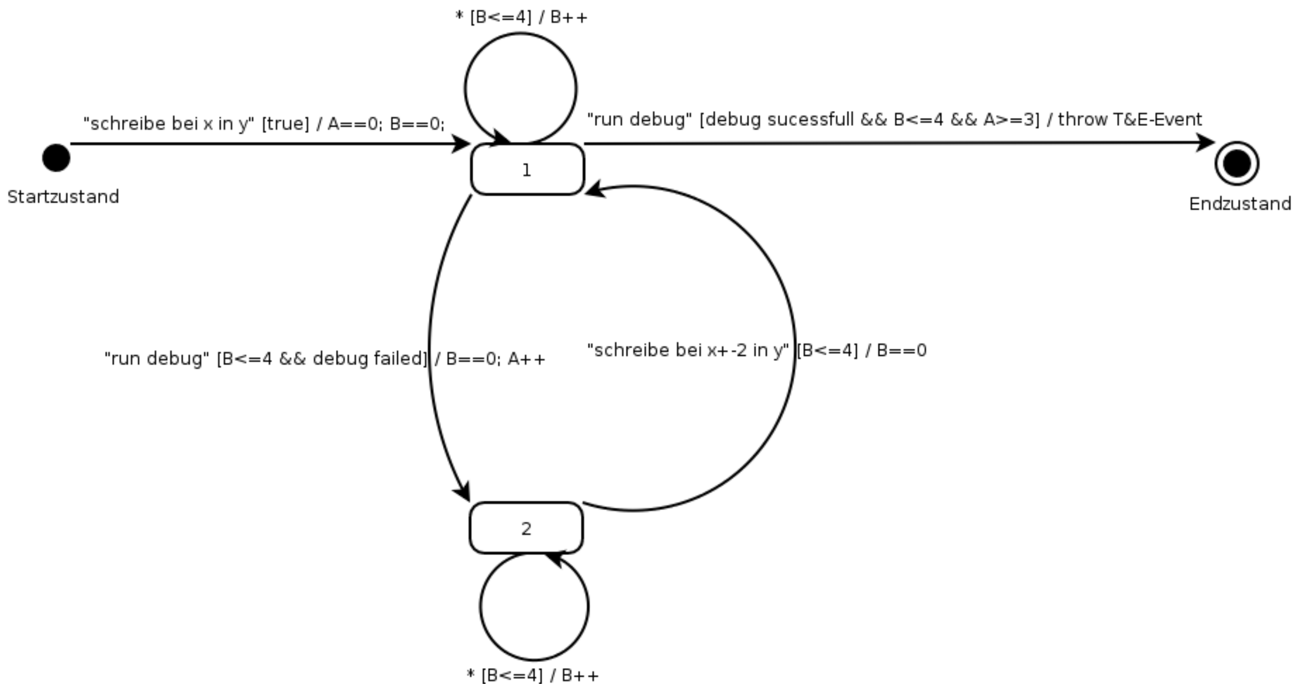
In diesem Abschnitt werde ich ein komplexeres Beispiel vorstellen, um einen kleinen Eindruck von der Komplexität realistischer Anwendungen zu ermöglichen. Hierbei handelt es sich trotz des größeren Umfanges (im Vergleich zu den bisherigen Beispielen) um eine

relativ simple Episode. Kompliziertere, wie zum Beispiel Trial-and-Error-Episoden (Hannes Restel, FU-Berlin 2006; siehe auch: http://projects.mi.fu-berlin.de/w/bin/view/SE/ThesisTrialError) oder Episoden zum Verfolgen von Codekopien (Sofoklis Papadopoulos, FU-Berlin 2006; siehe auch: http://projects.mi.fu-berlin.de/w/bin/view/SE/ThesisCopyTracking), ermöglichen Interessierten einen tiefergehenden und ausführlicheren Einblick.

Betrachtet wird folgende Episode (missglücktes Debugging bedeutet hier: Fehler im Code gefunden) : *(Bsp 7-1)*

Zuerst schreibt oder verändert der beobachtete Entwickler eine Zeile (im weiteren mit dem Wert x versehen) in einer Datei (im weiteren mit y bezeichnet). Nun führt er maximal 4 für die Episode unbedeutende Aktionen aus (Achtung: hierbei darf kein erfolgreiches Debuggen der Datei y stattfinden, da dieses den Automaten verwerfen würde). Nach missglücktem Debugging der Datei y dürfen weitere unbedeutende Aktionen auftreten (maximal 4). Nun erfolgt eine Änderung im unmittelbaren Umfeld von Zeile x in Datei y (z.B.: x+-2), wiederum gefolgt von maximal 4 unbedeutenden Operationen (Events), u.s.w. Nach dreimaligem negativem Debugging und den darauffolgenden, erneuten Veränderungen im Umfeld von x in Datei y wird ein erfolgreiches Debugging der Datei y durchgeführt. Damit soll die Episode erkannt und ein neues Ereignis "geworfen" werden mit den folgenden zusätzlichen Parametern Datei und ursprünglich veränderte/geschriebene Zeile.

7.A Modellierung als erweiterter Zustandsübergangsautomat *(Bsp 7-1a)*



7.B Modellierung mit Hilfe erweiterter regulärer Ausdrücke *(Bsp 7-1b)*

Die vollständige Darstellung als regulärer Ausdruck ist nicht möglich, da sich Bedingungen, wie z.B. Attributwerte, nicht darstellen lassen. Dennoch will ich kurz zumindest die Ereignissequenz als regulären Ausdruck vorstellen; hierbei sei w ein write-Ereignis, s ein debug-success-Ereignis und f ein debug-fail-Ereignis (es ist allerdings davon auszugehen, dass der Erfolg oder Nichterfolg eines Debugvorganges als Attribut innerhalb des debug-Ereignises gespeichert ist):

w[^s]{0,4}f[^s]{0,4}w[^s]{0,4}f[^s]{0,4}w[^s]{0,4}f[^s]{0,4}w[^f]{0,4}s

7.C Modellierung mit der neu entwickelten Notation *(Bsp 7-1c)*

'1,1' {< write >[true]} / x:= event.line; y:= event.file;

'0,4' {#\{< run-debug >[event.result=="successfull"&&event.file==y]}[true]} /

'1,1' {< run-debug >[event.result=="failed"&&event.file==y]} /

'0,4' {#\{< run-debug >[event.result=="successfull"&&event.file==y]}[true]} /

'1,1' {< write >[(x-2<=event.line<=x+2)&&event.file==y]} /

'0,4' {#\{< run-debug >[event.result=="successfull"&&event.file==y]}[true]} /

'1,1' {< run-debug >[event.result=="failed"&&event.file==y]} /

```
'0,4' {#\{< run-debug >[event.result=="successfull"&& event.file==y]}[true]} /
'1,1' {< write >[(x-2<=event.line<=x+2)&&event.file==y]} /
'0,4' {#\{< run-debug >[event.result=="successfull"&&event.file==y]}[true]} /
'1,1' {< run-debug >[event.result=="failed"&&event.file==y]} /
'0,4' {#\{< run-debug >[event.result=="successfull"&&event.file==y]}[true]} /
'1,1' {< write >[(x-2<=event.line<=x+2)&&event.file==y]} /
'0,4' {#\{< run-debug >[event.result=="failed"&&event.file==y]}[true]} /
'1,1' {< run-debug >[event.result=="successfull"&&event.file==y]} /
throw '< trial-error >, file=y, line=x, time=event.time';
```

Anmerkung:

Diese Modellierung kann unter der Zuhilfenahme von Abfolgeoperatoren (siehe Kapitel 9.B) noch deutlich kürzer und übersichtlicher dargestellt werden, da sich wiederholende Teilsequenzen eingeklammert und mit einer Auftrittshäufigkeit versehen werden können.

Dennoch wird hier erneut deutlich, dass bei einer programmiertechnischen Umsetzung ein Kompositummuster von Vorteil ist, da es eine weitere Möglichkeit bietet, solche Teilsequenzen zu behandeln, indem man sie als eigene Episoden vorher erkennen lässt und lediglich nach den so geworfenen Ereignissen sucht.

Die Orientierung an diesem Entwurfsmuster würde also zwei Vorteile mit sich bringen: Erstens können mehrere logisch zusammenhängende Erkenner als Alternativen eines Episodenerkenners gebündelt und zusammengefasst werden. Zweitens können wiederholt auftretende Teile eines Episodenerkenners ausgegliedert und somit leichter wiederverwendet werden.

8 Programmiertechnische Umsetzung von Zustandsübergangsautomaten

Ich werde in diesem Kapitel auf den Entwurf des Rahmenwerkes unter Nutzung von Zustandübergangsautomaten und SCXML eingehen, da diese als zwei unterschiedliche Ansätze einer programmiertechnischen Umsetzung des Beschreibungsmechanismus „Zustandsübergangsautomaten" bei der obigen Analyse herangezogen wurden. Zum besseren Verständnis der Ergebnisse ist es folglich notwendig, sie kurz zu erläutern.

Da ein Zustandsübergangsautomat nicht in schriftlicher, sondern in grafischer Form vorliegt kann er nicht einfach in einen Parser oder etwas ähnliches eingegeben werden, wie es z.B. bei regulären Ausdrücken der Fall ist. Eine programmiertechnische Umsetzung kann entweder durch die Verwendung einer geeigneten Beschreibungssprache (z.B.: SCXML) oder durch die Nutzung von passenden, vordefinierten Klassenpaketen geschehen.

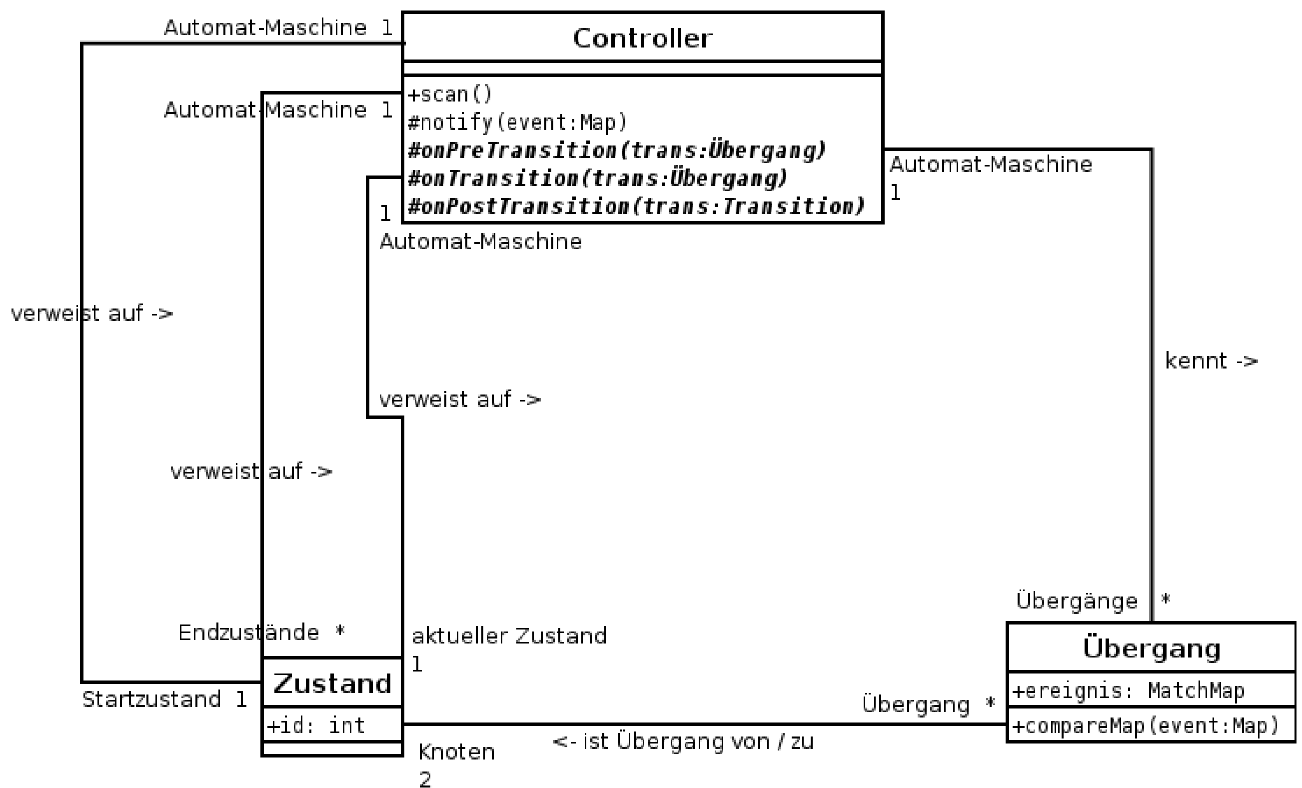
8.A Entwurf eines geeigneten Klassenpakets

Zur Umsetzung wird eine Abbildung auf ein Objektmodell genutzt, das heißt, dass der Zustandsübergangsautomaten-Entwurf mit Hilfe eines Rahmenwerkes ausprogrammiert werden muss. Ich stelle hier kurz einen von mir in der Programmiersprache Java geschriebenen Entwurf vor.

8.A.I Der Entwurf

Ein Zustandsautomat wird durch Knoten (also Zustände) und ihre Übergänge (in der Regel beim Auftreten von Ereignissen) dargestellt. In meinem ersten Entwurf wird der eigentliche Automat durch einen "Controller" abgebildet, der sich den aktuellen Zustand merkt. Dabei ist jeder Zustand (auch Knoten genannt) mit all seinen Übergängen assoziiert. Zu jedem Übergang gehören ein Start- und ein Endknoten. Jeder Übergang enthält eine Vergleichsstruktur für aus dem Datenstrom stammende Ereignisse als Attribut, im Weiteren wird der Typ dieser Vergleichsstruktur als „MatchMap" bezeichnet.

Das Klassendiagramm würde in einem solchen Fall wie folgt aussehen:



Das Objektdiagramm:

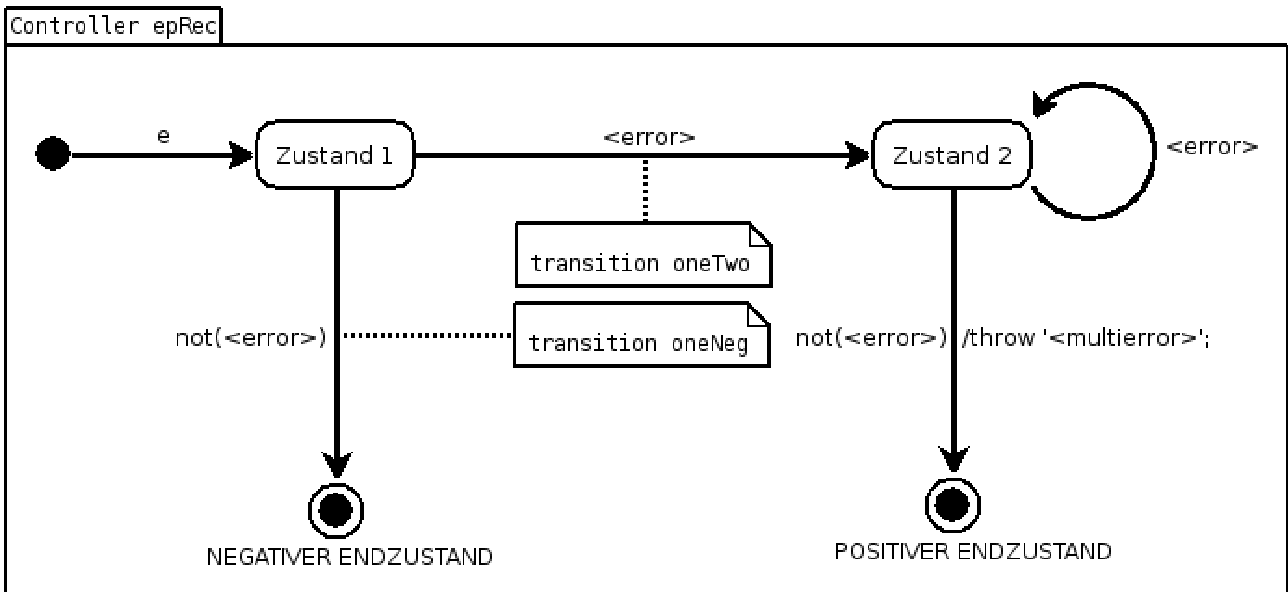
8.A.II Der Ablauf

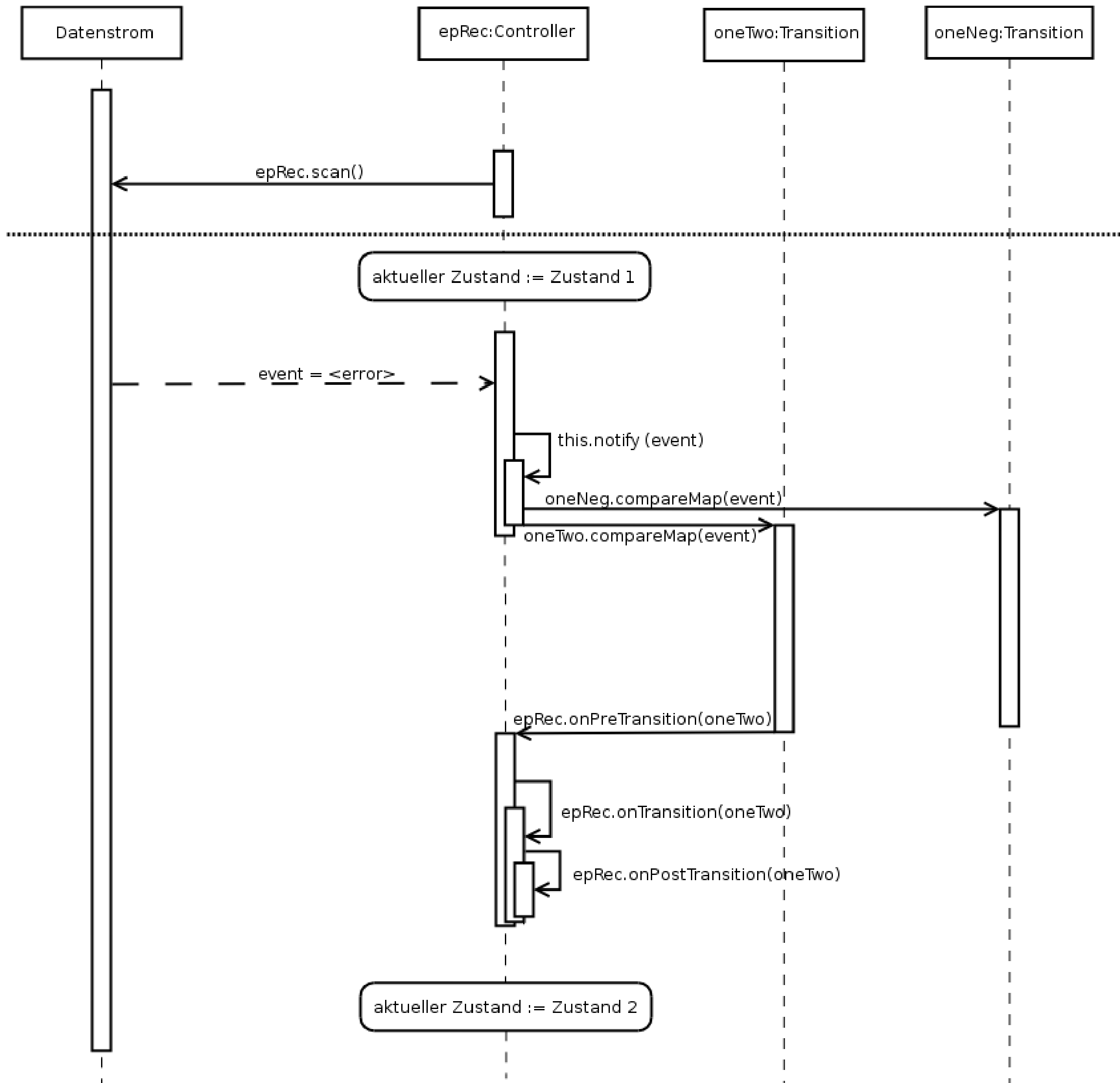
Der Controller erhält ein Event (also ein Ereignis mit mehreren Attributen) aus dem Datenstrom und gibt es via notify() an den aktuellen Zustand und alle dazugehörigen (möglichen) Übergänge. Der Typ dieses Ereignisses wird, in Hinblick auf die Ähnlichkeiten mit dem Java-Objekttyp gleichen Namens, schlicht als „Map" bezeichnet. Alle so erreichten Übergänge prüfen nun ihre MatchMap mit der erhaltenen Map. Bei Übereinstimmung wird die abstrakte (also vom späteren User zu implementierende) Methode onTransition() ausgeführt. Sie setzt den aktuellen Zustand des Controllers neu, startet ggf. parallel einen neuen Controller oder lässt ein neues Event an den Datenstrom übergeben und schließt den aktiven Controller.

Folgende Notation bzw. folgender Zustandsübergangsautomat kann zum Beispiel zu dem

darauffolgenden Sequenzdiagramm (Seite 24) führen (Bsp. 8.A.II-1):

'1,*' {<error>} /
'1,1' {#\<error>} / throw '<multierror>';





8.A.III Die Eingabe

Die Umsetzung eines neuen Episodenerkenners in diesem Rahmenwerk ist wie folgt vorzunehmen:

Der Nutzer muss für jeden Automaten die abstrakte Klasse „Controller" erweitern, in dem er die Methoden „onPreTransition", „onPostTransition" und „onTransition" für jeden möglichen Übergang definiert. Außerdem würde der Nutzer alle nötigen Objekte der Klassen „Übergang" und „Zustand" erzeugen lassen.

Beispiel (8.A.III-1): Nehmen wir folgende leicht veränderte Teil-Episode aus Beispiel 7-1c:

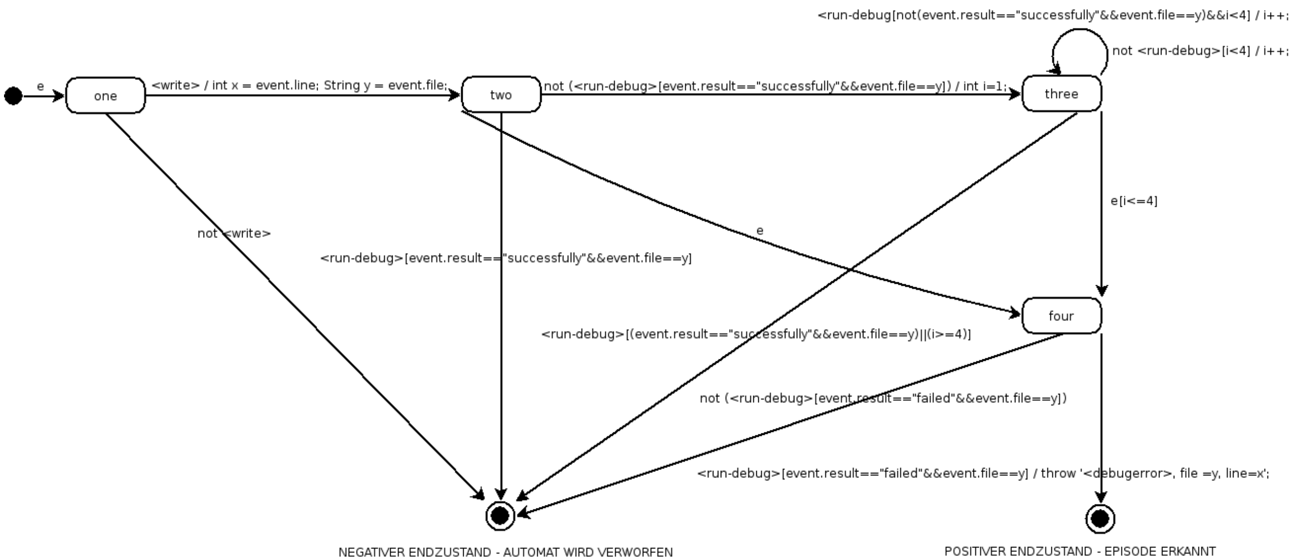
'1,1' {< write >} / int x = event.line; String y = event.file;

'0,4' {#\< run-debug >[event.result=="successfull"&&event.file==y]} /

'1,1' {< run-debug >[event.result=="failed"&&event.file==y]} /

throw '< debugerror >, file=y, line=x';

Diese Episode kann zum Beispiel durch den folgenden Zustandsübergangsautomaten erkannt werden:



Um diesen Automaten nun zu implementieren, müssen sämtliche Zustandsobjekte und Übergangsobjekte (mit ihren Vergleichsmustern) erzeugt und der Controller erweitert werden. So muss zum Beispiel in der Methode onTransition folgender if-Fall implementiert werden, um den Übergang von Zustand two nach Zustand three zu realisieren :

```
...
        if (trans == twoThree)
        {
                int i=1;
                actualState = three;
        }
...
```

Wir nehmen hierbei an, dass das zum Übergang gehörige Zustandsobjekt twoThree genannt wurde; actualState sei das Attribut des Controllers, das die Assoziation zum aktuellen Zustand darstellt.

Dem Nutzer stehen dabei mehrere Varianten zur Verfügung, er könnte beispielsweise die Anweisung int i=1; auch erst in der onPostTransition-Methode realisieren.

8.A.IV Zusammenfassung

Übergänge und Zustände werden durch Klassen bzw. deren Objekte repräsentiert. Sie

verweisen jeweils auf einen Start- und einen Zielzustand. Die Ereignis-Bedingung des Übergangs ist Attribut (Typ „Match-Map") des zugehörigen Objekts und wird mit auftretenden Ereignissen (Typ „Map") verglichen. Sonstige Bedingungen sind entweder in der Match-Map-Struktur enthalten oder werden im Controller, z.B. durch die Methode onPreTransition abgefragt (Java-Mächtigkeit). Aktionen werden durch den Controller eingeleitet, bzw. durchgeführt. Der Keller bzw. die Variablen können beliebig innerhalb des Controllers definiert und genutzt werden. Der Controller ist mit dem Datenstrom des ECG verbunden und wird vom Benutzer erweitert. Er reicht jedes auftretende Ereignis als „Map"-Struktur an die mit dem aktuellen Zustand assoziierten Übergänge weiter. Diese Überprüfen die „Map" auf Übereinstimmung mit ihrem „Match-Map"-Attribut. Falls eine solche Übereinstimmung vorliegt, wird die zugehörige Aktion in den Methoden des Controllers („onPreTransition", dann „onTransition", dann „onPostTransition") ausgeführt.

8.B SCXML

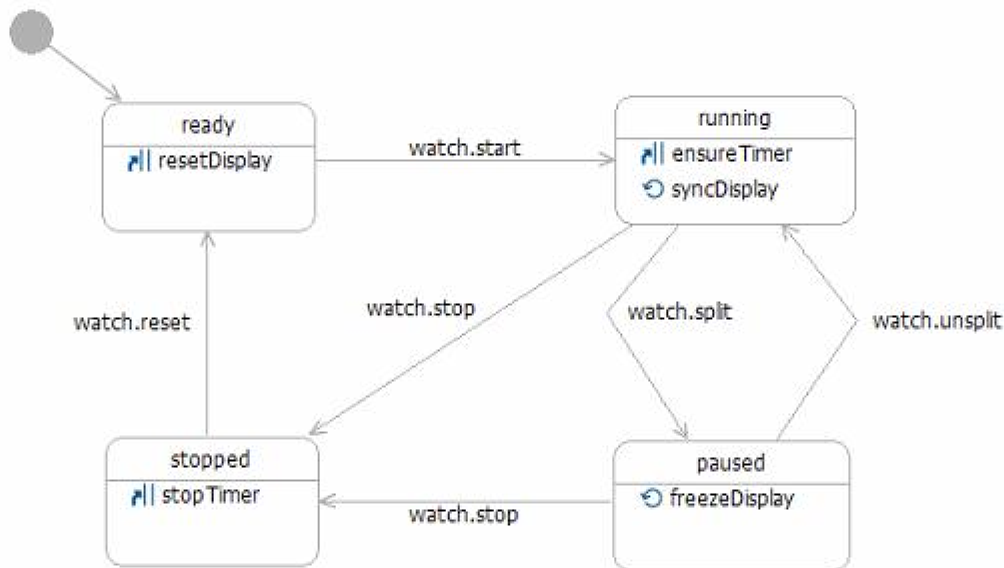
Eine Alternative zu diesem Entwurf bietet die Nutzung von SCXML. Diese Beschreibungssprache bezieht sich auf die Notation von Zustandsübergangsautomaten des Mathematikers David Harel, den sogenannten Harel State Tables, die auch in UML 2.0 Verwendung finden.

Die Anwendung würde wie folgt ablaufen:

Der Entwickler eines Episodenerkenners beschreibt den grafischen Entwurf des Automaten in einer SCXML-Datei. Ein Programm wie Commons SCXML könnte anschließend automatisch die SCXML-Angaben in einen Erkenner umsetzen.

Ohne weiter auf die Sprache als solche einzugehen, möchte ich ein kurzes Beispiel zeigen um einen kleinen Eindruck von SCXML zu vermitteln.

Beispiel (8.B-1) (Angaben zum Copyright des Beispiels finden Sie am Ende der Ausarbeitung):



Derselbe Automat in SCXML beschrieben:

```
<?xml version="1.0"?>
<scxml xmlns="http://www.w3.org/2005/07/SCXML"
      version="1.0"
      initialstate="reset">
   <state id="reset">
      <transition event="watch.start"   target="running"/>
   </state>
   <state id="running">
      <transition event="watch.split"   target="paused"/>
      <transition event="watch.stop"    target="stopped"/>
   </state>
   <state id="paused">
      <transition event="watch.unsplit" target="running"/>
      <transition event="watch.stop"    target="stopped"/>
   </state>
   <state id="stopped">
      <transition event="watch.reset"   target="reset"/>
   </state>
</scxml>
```

Weitere Bedingungen werden durch das Attribut „cond" im „transition"-Tag deklariert.

Aktionsanweisungen werden in weiteren Tags definiert, z.B. wie folgt (Bool'sche Variable „door_closed" wird auf „true" gesetzt)(Bsp. 8.B-2):

```
...
<transition event="door_close" target="cooking" cond="${'door_closed' eq '${false}'}">
    <var name="door_closed" expr="${true}"/>
</transition>
...
```

9 Formalisierung des neu entwickelten Beschreibungsmechanismus (ALED)

In diesem Kapitel wird der bereits erwähnte neu entwickelte Beschreibungsmechanismus formal spezifiziert. Er wird im Weiteren als ALED (Automaton-Language for Episode Description/Automatensprache zur Episodenbeschreibung) bezeichnet, wobei der konkrete Name von der Schreibweise und Formalisierung der Aktionsanweisungen und der Bedingungen (Erläuterungen weiter unten) vorgegeben wird, weshalb eigentlich von Java-ALED gesprochen werden müsste.

ALED ist eine Beschreibungssprache für Automaten, die dadurch Turingmächtigkeit erhält, dass sowohl ihre Aktionen als auch ihre Bedingungen in einer turingmächtigen Sprache formuliert werden. (Prinzipiell lassen sich sowohl die Bedingungen als auch die Aktionen in allen erdenklichen Sprachen notieren. Hiervon hängt jedoch die Mächtigkeit der gesamten Beschreibungssprache ab. Würde man also z.B. anstelle von Java (turingmächtig) eine Kellernotation für die Aktionen nutzen, so würde man lediglich Typ2-Mächtigkeit erhalten.)

9.A Formalisierung der Syntax

Bei der ALED gilt ein Dokument immer als Beschreibung genau einer Episode, beziehungsweise einer Episodenalternative (siehe auch Abschnitt 5.B). Die Zeilen werden chronologisch von oben nach unten abgearbeitet bzw. überprüft. EOL („End of line") steht dabei für einen Zeilenumbruch.

Eine solche Episode besteht aus einem oder mehreren Blöcken:

```
Episode :=      Block {Block}
```

Ein Block ist dabei entweder eine nicht-leere Menge von Anweisungen (durch Zeilenumbrüche voneinander getrennt) oder aber ein von Klammerzeilen umfasster Block, beziehungsweise zwei Blöcke durch einen Abfolgeoperator voneinander getrennt:

```
Block :=        '(' [Auftrittshäufigkeit] 'EOL' Block ')' 'EOL'
oder            Block Abfolgeoperator [Auftrittshäufigkeit] 'EOL' Block
oder            Anweisung 'EOL' {Anweisung 'EOL'}
```

Eine Episode könnte also wie folgt aussehen (Bsp. 9.A-1):

Anweisung

oder so:

Anweisung1

Anweisung2

oder auch so:

Anweisung1

(

Anweisung2

Abfolgeoperator Auftrittshäufigkeit

Anweisung3

Anweisung4

)

Anweisung5

Es gibt drei Abfolgeoperatoren, die im Abschnitt 9.B genauer erläutert werden. Dies sind:

Abfolgeoperator :=	'SEQ'
oder	'ALTER'
oder	'COMB'

Werden zwei Anweisungen lediglich durch einen Zeilenumbruch voneinander getrennt, so entspricht dies der Aussage erste Anweisung SEQ zweite Anweisung. Man spricht deshalb auch von dem impliziten bzw. dem expliziten chronologischen Sequenzoperator (Kapitel 9.B.I).

Satz:

Anweisung1

Anweisung2

entspricht:

(

Anweisung1

SEQ

Anweisung2

)

Eine Anweisung besteht aus einer Auftrittshäufigkeit, einer durch geschweifte Klammern eingefassten Ereignismenge, einem Trennsymbol und ggf. einer oder mehreren Aktionsanweisungen.

Anweisung := Auftrittshäufigkeit '{' Ereignismenge '}' '/' {Aktionsanweisung}

Eine Auftrittshäufigkeit besteht aus Mindest- und Höchsthäufigkeit, die durch ein Komma voneinander getrennt und zusammen von Hochkommata eingefasst werden. (In Abschnitt 9.C wird nochmals genauer auf die Auftrittshäufigkeiten eingegangen.)

Auftrittshäufigkeit :='" Mindesthäufigkeit ',' Höchsthäufigkeit '"

Die Mindesthäufigkeit ist eine natürliche Zahl oder Null.

Die Höchsthäufigkeit ist ebenfalls eine natürliche Zahl (außer 0) oder aber unendlich (bzw. beliebig) repräsentiert durch die Wildcard *.

Mindesthäufigkeit :=	'0'
oder	'1'
oder	'2'
oder	...
Höchsthäufigkeit :=	'1'
oder	'2'
oder	'3'
oder	...
oder	'*'

Die Höchsthäufigkeit ist immer größer oder gleich der Mindesthäufigkeit oder aber sie ist beliebig.

Satz: (Höchsthäufigkeit >= Mindesthäufigkeit) oder (Höchsthäufigkeit == *)

Eine Ereignismenge kann entweder aus einem Ereignistyp oder aus einer Ereignismenge (eingefasst in geschweifte Klammern) oder aus zwei Ereignismengen, die durch einen Mengenoperator verbunden sind, bestehen. Eine Ereignismenge kann außerdem auch aus einer in geschweiften Klammern eingefassten Ereignismenge und anschließend einer Bedingung bestehen.

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

Als Mengenoperatoren fungieren:

+ als Vereinigung

\ als „ohne"

~ als Schnittmenge

Mengenoperator :=	'+'
oder	'\'
oder	'~'

Ein Ereignistyp besteht aus Ereignisname und ggf. einer Bedingung.

Ereignistyp :=	Ereignisname [Bedingung]

Ein Ereignisname ist entweder das Zeichen #, das für ein beliebiges Ereignis steht oder aber ein mögliches Namensattribut eines Ereignisses - eingefasst in spitze Klammern (größer-/kleiner-als-Zeichen).

Ereignisname := '#'

oder '<' mögliches Namensattribut eines Ereignisses '>'

Eine Bedingung ist ein Boole'scher Ausdruck in der Schreibweise der zugrunde liegenden Turing-mächtigen Sprache - eingefasst in eckige Klammern.

Bedingung := '[' Boole'scher Ausdruck ']'

Es gilt: Ein Ereignisname, der ungleich # ist, entspricht auch immer dem Ereignisnamen # mit einer Bedingung die das Namensattribut des auftretenden Ereignisses betrifft.

Satz: <abc> == #[event.name == „abc"]

Eine Aktionsanweisung ist eine Anweisung in der Syntax der zugrunde liegenden Sprache (in unserem Fall Java).

Aktionsanweisung := Anweisung in Form der zugrunde liegenden Sprache

Beispiel (9.A-2):

Betrachten wir folgende Java-ALED-Anweisung:

'1, 5' {#|<run-debug>[event.file == „xyz"]} / String s = „habe-was";

'1, 5' ist hierbei die Auftrittshäufigkeit mit 1 als Mindesthäufigkeit und 5 als Höchsthäufigkeit. Ein bis fünf Ereignisse müssen demnach aus der darauffolgenden Ereignismenge auftreten.

#|<run-debug>[event.file == „xyz"] ist die Ereignismenge, bestehend aus einer Ereignismenge #, einem Mengenoperator | und einer zweiten Ereignismenge <run-debug>[event.file == „xyz"]. Erstere besteht aus einem Ereignistyp ohne Bedingung (Ereignisname ist hierbei #), letztere ebenfalls aus einem Ereignistyp, wobei hier <run-debug> ein Ereignisname und event.file == „xyz" eine Bedingung ist. String s = „habe-was"; ist eine Aktionsanweisung in Java-Syntax.

9.B Die drei Abfolgeoperationen

Die Abfolgeoperationen legen fest, in welcher chronologischen Reihenfolge bestimmte Ereignisse auftreten müssen, damit die gesuchte Episode vorliegt. Sie bieten oftmals die Möglichkeit mehrere Episodenalternativen in einer ALED-Beschreibung zu vereinen. Ist nach der öffnenden Klammer, beziehungsweise nach einem explizit angegebenen Abfolgeoperator, keine Auftrittshäufigkeit angegeben, so gilt immer die Auftrittshäufigkeit '1,1'. Dies stellt eine häufige Fehlerursache bei der Verwendung von Abfolgeoperatoren dar.

Beispiel (9.B-1):

(

'2,3' {<a>} /;

SEQ

```
'1,*' {<c>} /;
)
```

ist also gleichbedeutend mit:

```
('1,1'
'2,3' {<a>} /;
SEQ '1,1'
'1,*' {<c>} /;
)
```

9.B.I Chronologischer Sequenzoperator

Es gilt, sofern durch das Einklammern und den expliziten Einsatz von Abfolgeoperatoren nicht anders festgelegt, immer, dass die Anweisungen einer Episode (im Sinne der ALED-Formalisierung) von oben nach unten, durch chronologische Sequenz, miteinander verbunden sind. Folgendes Beispiel würde also dann erfolgreich eine Episode erkennen, wenn zuerst zwei- bis dreimal Ereignis a und anschließend einmal Ereignis b auftritt.

Beispiel (9.B.I-1):

```
'2,3' {<a>} /;
'1,1' {<b>} /;
```

Selbiges gilt allgemein für Blöcke. In einigen Fällen ist es dennoch sinnvoll Blöcke explizit mit dem Sequenzoperator SEQ zu verknüpfen, um zum Beispiel das mögliche mehrfache Auftreten eines Blocks zu beschreiben: *(Bsp. 9.B.I-2)*

```
( '1,1'
'2,3' {<a>} /;
'1,1' {<b>} /;
SEQ '1,4'
'1,*' {<c>} /;
'1,2' {<d>} /;
)
'1,1' {<e>} /;
```

Heißt folglich: Eine Episode wird immer dann erkannt, wenn zwei- bis dreimal a, anschließend b und dann ein- bis viermal ein Block aus mindestens einem c, gefolgt von einem oder zwei Ereignissen des Typs d auftreten, und letztendlich noch ein e Ereignis auftritt.

Achtung: Bei der impliziten Sequenzoperation zwischen einzelnen Anweisungen ist darauf zu achten, dass Anweisungen mit der Mindesthäufigkeit 0 zu den mit ihnen sequenziell verbundenen Anweisungen (bis zur ersten solchen mit Mindesthäufigkeit >0) disjunkte Ereignismengen beschreiben. Sollte dies nicht der Fall sein würde die deterministische

Eigenschaft der Episode verloren gehen.

Beispiel (9.B.II-1):

'0,*' {A} /;

'0,1' {B} /;

'1,1' {C} /;

Hierbei müssten die Ereignismengen A, B und C disjunkt sein, da sonst unterschiedliche „Übergänge“ möglich wären. Also ein Ereignis x aus A und B bei seinem Auftreten beiden Zeilen zugeschrieben werden könnte und somit nicht eindeutig festzustellen ist, an welchem Punkt man sich in der chronologischen Abfolge befindet.

9.B.II Chronologischer Alternativenoperator

Der chronologische Alternativenoperator findet zum Beispiel immer dann Verwendung, wenn entweder ein Ereignis x mit einer Aktionsanweisung y oder aber ein Ereignis a mit einer von Aktionsanweisung b (b ungleich y) eintreten darf, um eine Episode zu erfüllen. Das folgende Beispiel erkennt also immer dann eine Episode, wenn entweder ein- bis dreimal ein Ereignis aus A und anschließend ein Ereignis aus C oder genau ein Ereignis aus B und anschließend ein Ereignis aus C eintreten (A,B,C seien dabei Ereignismengen).

Beispiel (9.B.II-1):

(

'1,3' {A} /e;

ALTER

'1,1' {B} /f;

)

'1,' {C} /x;

Achtung: Da ALED deterministisch ist, müssen A, B und C voneinander disjunkte Ereignismengen sein.

9.B.III Chronologischer Kombinationsoperator

Der chronologische Kombinationsoperator gibt an, dass die durch ihn getrennten Blöcke oder Anweisungen in beliebiger (Achtung: Auftrittshäufigkeiten beachten) Kombination vorkommen können, um eine Episode zu erkennen.

Beispiel (9.B.III-1):

(„1,3“

„2,2“ {<a>} /e;

COMB „2,4“

„1,1“ {<b>} /f;

)

'1,1' {<c>} /x;

Obiges Beispiel erkennt sowohl bei <b><a><a><b><c> als auch bei <a><a><b><a><a><b><a><a><c> die Episode. Hier ist leicht zu merken, dass die explizite Verwendung von Abfolgeoperatoren leicht zu Fehlern führen kann. So wird im obigen Beispiel auch bei <b><b><a><a><c> eine Episode erkannt.

ACHTUNG:

Abfolgeoperatoren stellen also eine häufige Fehlerquelle dar. Im Allgemeinen gilt demnach: Gibt es bezogen auf eine Ereignissequenz irgendeine Möglichkeit der beschriebenen Episode zu entsprechen, so wird diese auch erkannt. Folgendes Beispiel erkennt also auch „bbb“, weil das eine vorgeschriebene a-Feld (implizite Auftrittshäufigkeit '1,1' nach der öffnenden Klammer) auch leer sein darf.

Beispiel (9.B.III-2):

(

'0,' {<a>} /;*

COMB '1,'*

'2,4' {<b>} /;

)

Ebenso wird auch bei<b><b><b><b><b><b><b><b> die gesuchte Episode erkannt, jedoch wegen der impliziten Maximalangabe in der ersten Zeile nicht bei <b><b><b><b><b><b><b><b><b>.

9.B.IV Bindungsstärke und Klammerung

Allgemein gilt, dass der chronologische Sequenzoperator sowohl im- als auch explizit stärker bindet als die beiden anderen Abfolgeoperatoren. Der Alternativenoperator bindet außerdem stärker als der Kombinationsoperator. Um explizit eine andere Bindungsstärke festzulegen, dienen die Klammerzeilen.

Beispiel (9.B.IV-1):

Block1 SEQ Block2 ALTER Block3 COMB Block4

entspricht also:

(((Block1 SEQ Block2) ALTER Block3) COMB Block4)

Alle drei Operatoren sind assoziativ, wenn die Auftrittshäufigkeiten implizit, also '1,1' sind. ALTER- und COMB-Operatoren sind außerdem kommutativ, der SEQ-Operator nicht, wie folgendes Beispiel zeigt *(Bsp. 9.B.IV-2):*:

Ereignissequenz: <a><b>

wird durch folgende Episodenbeschreibung in ALED erkannt:

(

'1,1' {<a>} /;

SEQ

'1,1' {<b>} /;

)

nicht jedoch durch folgende:

(

'1,1' {<b>} /;

SEQ

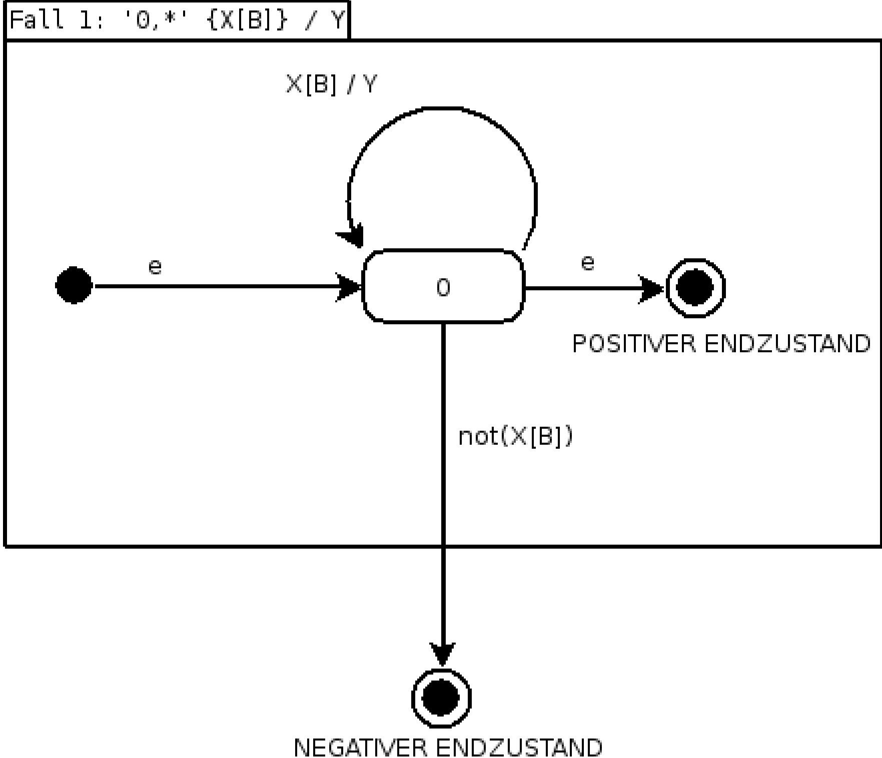
'1,1' {<a>} /;

)

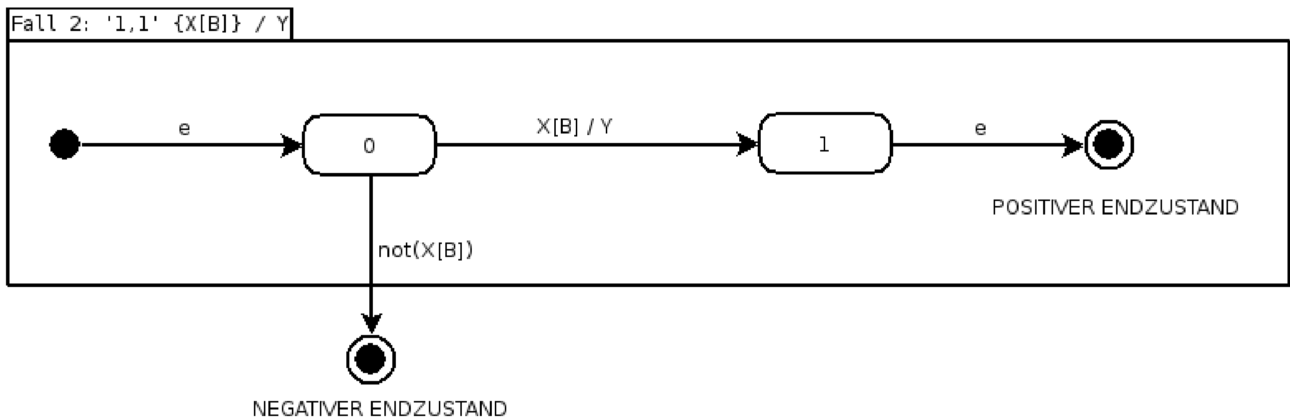
9.C Erläuterung und Abbildung der Auftrittshäufigkeiten auf Zustandsübergangsautomaten

Im Folgenden werden die unterschiedlichen Möglichkeiten von Auftrittshäufigkeiten in ALED-Anweisungen auf das Zustandsautomatenmodell abgebildet. Dies dient der Formalisierung dieser Häufigkeitsangaben und der Verdeutlichung der Übersetzbarkeit von einem auf das andere Modell. Die Ereignismenge in einer ALED-Anweisung beschreibt die Menge an Ereignissen, bei denen ein Übergang im zugehörigen Automaten stattfindet und bietet zusätzlich die Möglichkeiten weitere Bedingungen, wie sie auch beim Automatenmodell vorgesehen sind, zu formulieren. Die mögliche Aktionsanweisung am Ende einer Anweisung ersetzt die Aktionen, die bei einem Übergang stattfinden können. Interessant und ungeklärt bleibt damit bisher allerdings die Abbildung der Auftrittshäufigkeit in das Zustandsübergangsautomatenmodell. Diese lässt sich in den folgenden Bildern ablesen (Achtung: Ein Automat erkennt ein Wort, sobald es einen möglichen Pfad durch den Automaten gibt, der durch das Wort durchlaufen wird und zu einem positiven Endzustand führt):

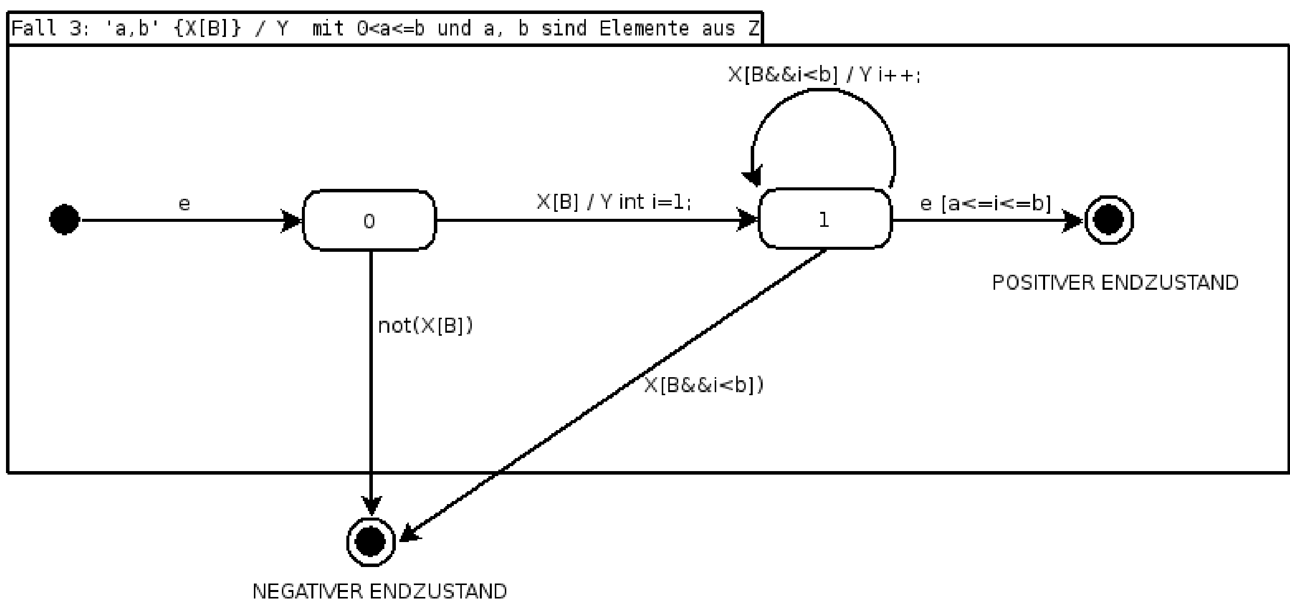
Fall 1: Beliebig viele Ereignisse aus einer Ereignismenge dürfen auftreten:


Fall 1: '0,*' {X[B]} / Y
X[B] / Y
e
0
e
POSITIVER ENDZUSTAND
not(X[B])
NEGATIVER ENDZUSTAND

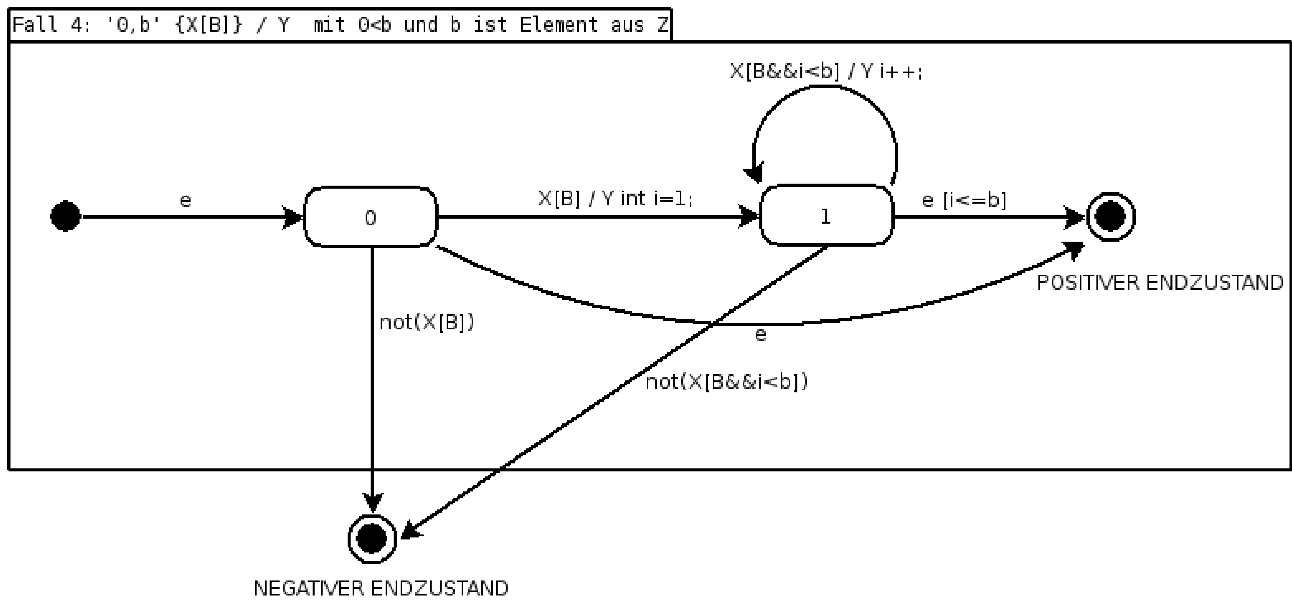
Fall 2: Genau ein Ereignis aus einer Ereignismenge muss auftreten:



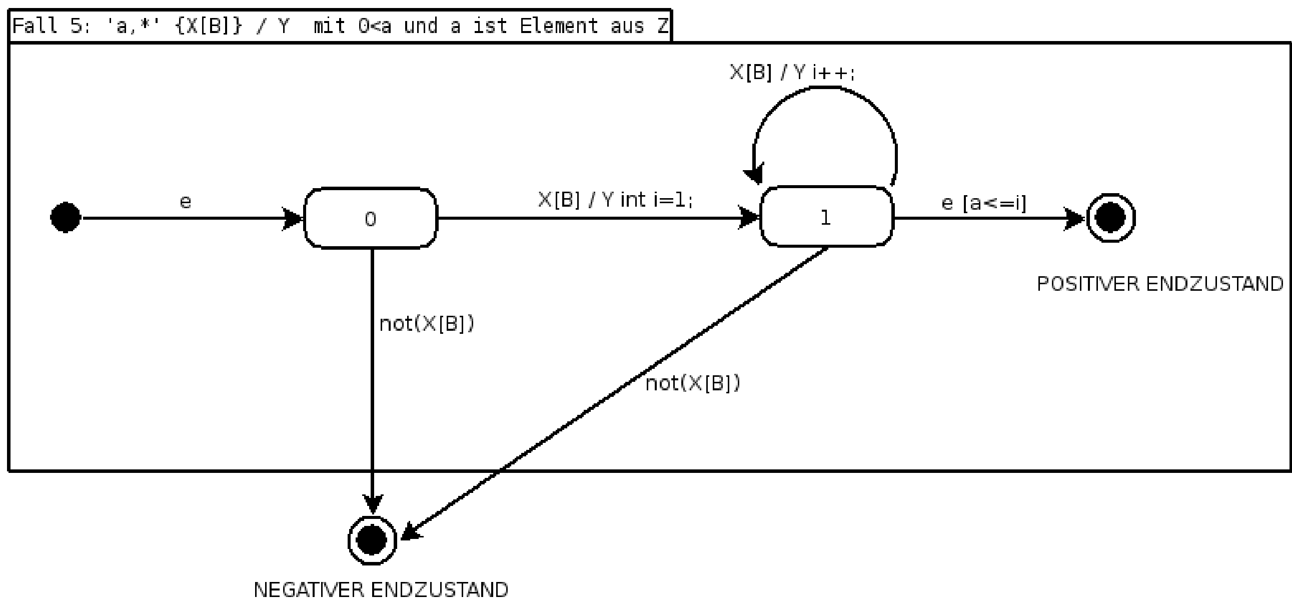
Fall 3: Mindestens a Ereignisse aus einer Ereignismenge müssen und maximal b dürfen auftreten:



Fall 4: Höchstens b Ereignisse aus einer Ereignismenge dürfen auftreten:



Fall 5: Mindestens a Ereignisse aus einer Ereignismenge müssen auftreten:



Der negative Endzustand wird immer dann erreicht, wenn ein nicht den möglichen ALED-Anweisungen entsprechendes Ereignis auftritt, also der Automat verworfen werden muss, weil die Episode vorerst nicht mehr vollendet werden kann.

9.D Weitere Ideen/Ausblick

Insbesondere in den letzten Wochen dieser Arbeit wurden viele weitere Ideen zur Umsetzung eines Rahmenwerks zur Episodenerkennung und zu ALED entwickelt. Die meisten davon finden in den obigen Kapiteln der Arbeit keine Erwähnung. Dies ist vor allem darauf zurückzuführen, dass es mir wichtig war, ALED in einer möglichst einfachen

und ausreichenden Form zu spezifizieren. Da ALED an sich eine Sammelbeschreibung vieler ALED-Dialekte (z.B.: Java-ALED, Pearl-ALED, ...) darstellt, erschien es mir sinnvoll, einen festen und unbedingt notwendigen Rahmen an Befehlen, Operatoren und Notationsvorschriften zu entwickeln, wohl wissend, dass viele interessante oder arbeitserleichternde Features so nicht enthalten sein würden. Die bis hierher nicht erwähnten Ideen erweitern weder die Mächtigkeit der ALED-Spezifikation, noch vergrößern sie die Episodenmenge, die beschrieben werden kann. Dennoch können einige davon im praktischen Gebrauch aus unterschiedlichsten Gründen vorteilhaft sein, weshalb ich kurz auf zwei eingehen möchte.

9.D.I Sonderbefehle in den Aktionsanweisungen

Bisher werden die Aktionsanweisung ausschließlich in der verwendeten Programmiersprache verfasst. Dies bietet den Vorteil, dass für den Entwickler eines Episodenerkenners die Trennung zwischen ALED-Notation und verwendeter Programmiersprache relativ klar und somit leicht nachvollziehbar ist. Allerdings könnte es wünschenswert sein Aktionen, wie zum Beispiel das Werfen eines neuen Ereignisses, in einer allgemein gültigen Notation auszudrücken. So wären zum Beispiel folgende Sonderbefehle denkbar:

ABORT;

-> Abbruch der Episodenerkennung, z.B. durch Auftreten eines unerwünschten Ereignisses

NEW x (name = variable1, param1 = variable2, ...);

-> Erstellen einer Ereignisinstanz/eines Ereignisobjekts x, mit Liste der zugehörigen Attribute in der Form „Bezeichner = Wert"

THROW x;

-> Übergabe der Ereignisinstanz/des Ereignisobjekts x an den Datenstrom

WAIT (int);

-> Anweisung die nächsten int auftretenden Ereignisse in der Episodenerkennung nicht zu berücksichtigen.

Sollte man die Einführung solcher Sonderbefehle, im Folgenden Aktionsbefehle genannt, wünschen, so sind damit auch Veränderungen an der obigen Formalisierung von ALED nötig:

Aktionsanweisung :=	Anweisung in Form der zugrunde liegenden Sprache
oder	Aktionsbefehl ';'
Aktionsbefehl :=	'ABORT'
oder	'THROW' Bezeichner
oder	'WAIT (' ganze positive Zahl ')'

oder 'NEW' Bezeichner '(name =' Wert [',' {Zuweisungen}] ')'

Die hier aufgeführten Formalisierungen von Aktionsanweisung und Aktionsbefehl sind nur beispielhaft. Sie sollen zeigen, dass konkretere Überlegungen nötig sind, um Aktionsbefehle direkt in ALED zu definieren.

9.D.II Metainformationen

Ebenso kann es sinnvoll zu sein Metainformationen zu Beginn oder am Ende einer Episodenbeschreibung anzugeben. Solche könnten sein: Informationen zum sogenannten Vorfiltern, das heißt Angaben über bestimmte Ereignisse, die für die gesuchte Episode völlig ohne Belang sind, sollen nicht beachtet werden, oder Informationen über den Verfasser, die Verwendung und den Grund für die Episodenbeschreibung beziehungsweise für den Einsatz zur Episodenerkennung. Auch können hier Informationen über Ereignisse, die gegebenenfalls geworfen werden, enthalten sein. (Hierbei ist darauf hinzuweisen, dass ein Episodenerkenner auch mehrere unterschiedliche Ereignisse werfen kann.) Konkretere Ideen zu solchen Metadaten beschreibt H. Restel in seiner Bachelorarbeit „Automatisches Erkennen von Trial-and-Error Episoden beim Programmieren“ [2] in Kapitel 4.

9.D.III Allgemeine Anmerkungen

Wie die beiden obigen Beispiele zeigen, ist es möglich und gegebenenfalls auch sinnvoll, ALED um weitere Befehle, Operatoren etc. zu erweitern. Die Weiterentwicklung von ALED steht dabei jedem frei, sollte jedoch auf den in diesem Dokument erarbeiteten Erkenntnissen beruhen, beziehungsweise diese berücksichtigen. ALED versteht sich, wie bereits erwähnt, als grober Rahmen zur Episodenbeschreibung, der auf seiner hier spezifizierten Form auch weiterhin beruhen sollte, um die Inkompatibilität unterschiedlicher Weiterentwicklungen zu vermeiden.

10 Anhänge

Weitere Informationen zu der Bachelorarbeit „Entwicklung einer Sprache zur einfachen Beschreibung und Implementierung von Episodenerkennern“ können unter der folgenden Wiki-Internetseite abgerufen werden:

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

Unter anderem sind hier Dokumente und Bilder zu den unterschiedlichen Kapiteln verfügbar. Auch der chronologische Ablauf der Arbeit kann eingesehen werden.

10.A Begriffserklärungen

Einige in diesem Dokument verwendete Begriffe existieren in unterschiedlichen Bereichen der Informatik in anderen Bedeutungen beziehungsweise wurden erst durch diese Arbeit eingeführt. In diesem Abschnitt sollen solche Begriffe kurz zur Vermeidung von Missverständnissen mit ihrer hier genutzten Bedeutung erklärt werden.

Episode

Eine Episode ist eine bestimmte Ereignissequenz, die einem Muster, dem Episodenschema,

entspricht oder einen bestimmten Effekt hervorruft und somit von besonderem Interesse ist. Der Begriff Episode taucht auch in der ALED-Formalisierung auf. Hier steht er stellvertretend für „Episodenschema" oder „Episodenmuster".

Episodenschema

Ein Episodenschema beschreibt eine Episode oder eine bestimmte Episodenmenge. Sie gibt vor, welchem Muster eine Ereignissequenz entsprechen muss, um als zugehörige Episode zu gelten.

Episodenerkenner

Programm oder Programmfragment, welches es ermöglicht, Episoden an Hand von Episodenschemata auf einem Datenstrom oder innerhalb einer chronologischen Ereignismenge aufzufinden und gegebenenfalls ein entsprechendes Ereignis an der korrekten Position einzufügen.

Beschreibungsmechanismus

Beschreibungsmechanismen bieten die Möglichkeit, Mengen, Sequenzen und Ähnliches abstrakt zu beschreiben. Episodenschemata können mit ihnen, ausreichende Mächtigkeit vorausgesetzt, verfasst werden.

Formalismus

Der Begriff Formalismus wird in dieser Arbeit gleichbedeutend mit „Beschreibungsmechanismus" verwendet.

Skriptnotation

Skriptnotation meint hier einen groben, noch nicht ausgereiften Entwurf eines Beschreibungsmechanismus.

Ereignis

Der Ontologie zufolge spiegeln Ereignisse Prozessentitäten, also Geschehen oder bestimmte Handlungen wieder. In diesem Dokument wird oft auch von Ereignis gesprochen, wenn eigentlich ein Datensatz gemeint ist, der ein Ereignis beschreibt.

Ereignisstrom

Ein Ereignisstrom ist ein Datenstrom, der ausschließlich aus Ereignissen beziehungsweise Ereignisbeschreibungen besteht.

Test-Driven Development

Als testgetriebene Entwicklung (auch testgesteuerte Programmierung, engl. test-driven development) ist eine Methode zur Entwicklung eines Computerprogramms. Sie ist Teil der agilen Softwareentwicklung. (Mehr Informationen zu testgetriebener Entwicklung findet man unter http://www.testdriven.com.)

Zustandsübergangsautomat

Zustandsübergangsautomat wird hier gleichbedeutend mit deterministischer endlicher Zustandsautomat verwendet. In dieser Arbeit wird ein Mischmodell aus Mealy- und Moore-Automaten genutzt.

Komplexität

Die Komplexität einer Sache meint im Allgemeinen den größtmöglichen Umfang/das größtmögliche Ausmaß, den/das diese Sache annehmen kann.

10.B Literatur- und Quellenhinweise

- [1] S. Jekutsch: http://projects.mi.fu-berlin.de/w/bin/view/SE/MicroprocessHome, AG Software Engineering, Institut für Informatik, Freie Universität Berlin, Versionen: 1. Februar 2006 bis 18. September 2006
- [2] H. Restel: Automatisches Erkennen von Trial-and-Error Episoden beim Programmieren, AG Software Engineering, Institut für Informatik, Freie Universität Berlin, 2006, http://projects.mi.fu-berlin.de/w/bin/view/SE/ThesisTrialError
- [3] Hongbing Kou: Studying Micro-Processes in Software Development Stream, Collaborative Software Development Laboratory, Department of Information and Computer Sciences, University of Hawai'i
- [4] Hongbing Kou, Philip M. Johnson: Automated Recognition of Low-Level Process: A Pilot Validation Study of Zorro for Test-Driven Development, Collaborative Software Development Laboratory, Department of Information and Computer Sciences, University of Hawai'i
- [5] D. Harel and M. Politi: Modeling Reactive Systems with Statecharts: The STATEMATE Approach, McGraw-Hill, 1998. (Siehe auch: http://www.wisdom.weizmann.ac.il/~dharel/reactive_systems.html.)
- [6] David Harel: Statecharts: A visual Formalism For Complex Systems, Dep. Of Applied Mathematics, The Weizmann Institute of Science, Rehovot, Israel, 1984/1986, © 1987 Elsevier Science Publishers B.V. (Nord-Holland)
- [7] SCXML, ©2006 W3C, http://www.w3.org/TR/scxml/
- [8] S. Papadopoulos: Verfolgen von Codekopien zur Defektvermeidung, AG Software Engineering, Institut für Informatik, Freie Universität Berlin, voraussichtlich 2006, http://projects.mi.fu-berlin.de/w/bin/view/SE/ThesisCopyTracking
- [9] F. Schlesinger: Protokollierung von Programmiertätigkeiten in der Eclipse-Umgebung, AG Software Engineering, Institut für Informatik, Freie Universität Berlin, 2006, http://projects.mi.fu-berlin.de/w/bin/view/SE/ThesisEclipsemonitor
- [10] Commons SCXML, © 2005-2006, The Apache Software Foundation, http://jakarta.apache.org/commons/scxml/
- U. Schöning: Theoretische Informatik – kurzgefasst (4. Auflage),Spektrum Akademischer Verlag Heidelberg Berlin, © 2001/2003
- U. Schöning: Ideen der Informatik – grundlegende Modelle und Konzepte (2. Auflage), Oldenbourg Verlag München Wien, © 2006
- H. Wittenbrink, W. Köhler et al.: XML, TEIA Lehrbuch Verlag, © 2003

10.C Copyrightangaben zu Beispiel 8.B-1

Copyright 2006 The Apache Software Foundation

Licensed under the Apache License, Version 2.0 (the "License");

you may not use this file except in compliance with the License.

You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.

10.D Danksagungen

Ich möchte mich bei Herrn Prof. Schulte für die Zweitkorrektur dieser Arbeit, Christopher Oezbek für die Buchausleihe und allen Rechtschreibkorrektoren für ihre Mühe bedanken.

Besonderer Dank gilt:

Hannes Restel für die gute Zusammenarbeit und die reichhaltige gegenseitige Befruchtung unserer Themen;

Prof. Dr. Prechelt für die Betreuung dieser Arbeit;

und im besonderen Maße Sebastian Jektusch für viel konstruktive Kritik und gelegentlichen Mutzuspruch, sowie seine organisatorischen Bemühungen.

Ferner gilt Dank all jenen, die es mir möglich machten diese Bachelorarbeit zu schreiben.

10.E 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 18.September 2006

....

(Unterschrift Christian A. Kopf)