PG Live

Endbericht

Mohsen Ahyaie, Ahmed Alami,
Said Chihani, Christian Frost,
Jochen Gerlach, Falk Howar,
David Karla, Christian May,
Svetla Nikolova, Marc Peschke

6. August 2007

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung und Übersicht

Falk Howar

Alternde Software-Systeme stellen für die Industrie zunehmend große Probleme dar. Es gibt an vielen, für die Unternehmen zentralen Stellen, Systeme, die mehrere Jahre oder vielleicht sogar Jahrzehnte lang in der Produktion eingesetzt wurden und während dessen von vielen verschiedenen Personen gepflegt und gewartet wurden. Diese Systeme sind oft in Sprachen entwickelt worden, die heute erstens nicht mehr Stand der Technik sind und zweitens nur mit großem Aufwand neu zu erlernen sind. Es gibt zwar noch Experten für diese Sprachen bzw. Systeme, jedoch sind gute Quellcode-Dokumentation und Modell-getriebene Entwicklung vergleichsweise junge Trends in der Informatik. Je älter solche Systeme werden, desto höher wird der Druck auf die Unternehmen, auf neue Technolgie umzusteigen. Dabei soll natürlich das in der alten Software versteckte Wissen möglichst mit-migriert werden. Eine komfortable Lösung für das Problem der Migration des in alten Systemen enthaltenen Wissens könnte das rechnergestützte Lernen dieser Systeme sein.

Hat man das Zielsystem erst einmal durch das Lernen in eine Darstellung überführt, welche eine spätere Migration auf neuere Technik erlaubt, kann man mit der Systembeschreibung zum Beispiel Modelchecking durchführen. Dies bedeutet die Gültigkeit bestimmter globaler Aussagen über das System rechnergestützt zu überprüfen. Wenn man aber aus einem beliebigen technischen System eine systematische Beschreibung lernen könnte, warum sollte man sich auf alte Systeme beschränken? Man könnte das Verfahren auch in der Entwicklung neuer Software zum Testen nutzen oder in Bereichen, in denen Systeme bisher so beschrieben werden, dass Modell-Checking gar nicht oder nur eingeschränkt möglich ist, wie z.B. bei der Beschreibung von Geschäftsprozessen durch ECA-Regelsysteme.

LiVe steht für „Lernen impliziter Verhaltensmuster“, also für das Lernen nicht ausdrücklich formulierter Eigenschaften von Systemen. Ziel eines solchen Erlernens von nicht (sofort) sichtbaren Systemeigenschaften kann es sein, diese in eine Darstellung zu überführen, welche die Systembeschreibung vollständig explizit macht.

Eine solche vollständige Beschreibung kann z.B. ein regulärer Ausdruck oder ein endlicher Automat sein. Implizites Wissen versteckt sich z.B. in ECA-Regelsystemen. Um eine Systembeschreibung in Form eines ECA-Regelsystems in eine Darstellung als Automat zu überführen, ist es notwendig, eine Transformation vorzunehmen, welche die vesteckten Eigenschaften offenlegt. Wahrscheinlich gibt es mehrere Möglichkeiten einer solchen Überführung von der einen in die andere Darstellung. Es ist anzunehmen, dass bei einem allgemeinen formalen Ansatz die „Zustands-Explosion“ ein ernstzunehmendes Problem darstellt. Hier bietet der Lern-Ansatz ei-

ne nützliche Garantie. Der gelernte Automat - wenn man davon ausgeht, dass sich das gelernte System als Automat beschreiben lässt - ist auf jeden Fall minimaler Größe. Der Ansatz ermöglicht es außerdem, anstelle des vollständigen Systems nur ein Teil des Systems zu erlernen. Man kann durch Abstraktion der bekannte Symmetrien vereinfachen oder nur die Teile des Systems erlernen, die von Interesse sind.

1.1 Ziele der Projektgruppe

Marc Peschke

Die Projektgruppe 507 an der Universität Dortmund hat die Aufgabe erhalten, eine formale Sprache zur Beschreibung von Systemen auf der Basis von ECA-Regeln zu entwickeln.
Mit Hilfe der entwickelten Sprache sollen beliebige Systeme simuliert werden können. Mittels eines Lernalgorithmus soll das simulierte System dann in eine Darstellung als einen endlichen Automaten überführt werden. Wichtigster Bestandteil der Aufgabe ist das Lernen.

Im Rahmen der Projektgruppe soll ein Framework entstehen, das es erlaubt, aus ECA-Regelsystemen globale Modelle zu generieren. Wichtig ist dabei die Möglichkeit, mit dem Abstraktionsniveau der Modellbildung zu „spielen“, da „naive“

Ansätze notwendig an der generierten Modellgröße scheitern (Zustandsexplosion). Um das zu erreichen soll die am Lehrstuhl V entwickelte LearnLib [20] eingesetzt werden, eine Bibliothek für praxisorientiertes Lernen von Verhaltensspezifikationen. Sie basiert auf Angluins Lern-Algorithmis L* [5] und ist direkt für derartige maßgeschneiderte Modellkonstruktionen konzipiert worden.

Zu den Hauptaufgaben der Projektgruppe zählt jedoch die Entwicklung einer Ausführungsumgebung für regelbasierte Systeme. Dieser Simulator soll zusammen mit der LearnLib eingesetzt werden, um beobachtungsbasiert konsistente Automatenmodelle aus regelbasierten Systemen abzuleiten. In einer zweiten Phase soll das entstandene Framework an den vom Lehrstuhl entwickelten Modelcheckers GEAR zur Verifikation der extrapolierten Modelle angebunden werden.

Die von der Projektgruppe entwickelte Lösung soll anschließend anhand zweier strukturell möglichst unterschiedlicher praxisrelevanter Beispiele evaluiert werden. Hier bieten sich beispielsweise eine Anwendung aus dem Bereich der ‘Value-Added´ Telefonieservices und eine aus dem Bereich personalisierter Workflowsysteme an.

Im Einzelnen sind folgende Teilaufgaben vorgesehen:

1. Festlegung einer geeigneten Sprache zur Spezifikation von regelbasierten Systemen
2. Ausführungsumgebung für regelbasierte Systeme: Es wird ein Laufzeitsystem entwickelt, mit dem sich regelbasierte Systeme direkt ausführen lassen. Außerdem muss entschieden werden können, ob ein regelbasiertes System eine Anweisungsfolge zulässt oder nicht. Dabei ist zu beachten, dass die gewählten Anwendungs-Szenarien auch zu integrieren sind.

3. Automaten-Lern-Schnittstelle: Die vom Lehrstuhl entwickelte Bibliothek zur Analyse von Automaten-Lern-Algorithmen (LearnLib) [20] wird an die entwickelte Ausführungsumgebung angepasst und gegebenenfalls erweitert.

4. Modell-Checking: Der vom Lehrstuhl entwickelte Modellchecker wird integriert. Hierdurch ermöglicht es die Projektgruppe globale Eigenschaften von regelbasierten Systemen zu verifizieren. Die Ergebnisse des Modelcheckings werden mit bestehenden graphischen Visualisierungs-Tools dargestellt.

5. Evaluation: Hier sollen die ausgewählten Anwendungsszenarien mit Hilfe der entwickelten Technologie untersucht werden.

Kapitel 2

Grundlagen

Christian May

Dieses Kapitel soll die nötigen Grundlagen für das Verständnis der Arbeit der Projektgruppe schaffen. Die hier vorgestellten Themen geben einen groben Überblick über die Theorie, können aber keinen tieferen Einblick in diese verschaffen. Dazu muss der geneigte Leser auf die im Literaturverzeichnis angegebenen Arbeiten verwiesen werden.
Im Unterkapitel „ECA-Regelsysteme“ werden als erstes Eca-Regeln und aufbauend ECA-Regelsysteme definiert. Der Umgang mit ECA-Regelsystemen erfordert viel Strukturierung, was kurz dargestellt wird. Zum Schluss werden zwei wichtige Eigenschaften von ECA-Regelsystemen vorgestellt: Terminierung und Konfluenz.
Im Unterkapitel „ECA-Regelsysteme in der Praxis“ wird die Verwendung von ECA-Regelnsystemen in der Praxis anhand von einigen ausgewählten Beispielen vorgestellt.
Die im vorherigen Kapitel begonnene praxisbezogene Sicht auf ECA-Regelsysteme wird im Unterkapitel „Feature Interaction“ weitergeführt. Feature Interaction vielen modernen Systemen Verwendung, deshalb ist es angebracht, dieses Thema in einem eigenen Unterkapitel zu behandeln.
Ein Ziel der Arbeit der Projektgruppe LiVe ist das automatische Lernen von ECA-Regelsystemen. Im Unterkapitel “Learnlib“ werden die lerntheoretischen Grundlagen und die von der Projektgruppe verwendete Algorithmenbibliothek vorgestellt.
Ein wichtiger Teil der Arbeit der Projektgruppe war die Spezifikation einer geeigneten Sprache zur Beschreibung von ECA-Regelsystemen. Dazu istdas Verständnis der Arbeitsweise von Compilern unverzichtbar. Im letzten Unterkapitel „Compilerbau“ werden deshalb theoretische Grundlagen zum Compilerbau beschrieben.

2.1 ECA-Regelsysteme

Christian May

ECA-Regeln

Event-Condition-Action (ECA)-Regeln nehmen in vielen Anwendungen einen hohen Stellenwert ein, so sind aktive Datenbanksysteme ohne ECA-Regeln nicht zu realisieren. Diese Anwendungen sind charakterisiert durch ein reaktives Verhalten, weil diese auf bestimme Situationen

reagieren müssen. Zur Veranschaulichung werden zwei Beispiele für aktive Datenbanksysteme betrachtet:

Beispiel 2.1 *Lagerverwaltung*

In Versandhäusern werden Datenbanksysteme zur Verwaltung der bestellbaren Artikel eingesetzt. In diesen Systemen sind zahlreiche Daten gespeichert, wie z.B. Bezeichnungen, Größen, Farben, Preise, Bestände und Schwellenwerte. Um Verzögerungen bei der Bearbeitung von Kundenbestellungen zu vermeiden, müssen bei bestimmten Situationen die Artikel bei den Lieferanten neu bestellt werden. Dieses reaktive Verhalten lässt sich wie folgt beschreiben:

- *Sobald der Bestand eines Artikels den für ihn festgelegten Schwellenwert unterschreitet, ist für diesen Artikel eine Neubestellung auszulösen.*

Beispiel 2.2 *Patientenverwaltung*

In Krankenhäusern gibt es viele Situationen, auf die die Verwaltungssysteme reagieren müssen. Zwei Beispiele hierfür sind:

- *Wird ein Patient zu einer speziellen Untersuchung auf eine (andere) Station überwiesen, ist der leitende Arzt darüber zu informieren. Zudem muss ihm eine Zugriffsmöglichkeit auf die Krankheitsgeschichte des Patienten gewährt werden, damit er sich einen Überblick verschaffen kann.*
- *Jeden Morgen um 6:00 Uhr sind die Medikamentenverordnungen für die Patienten auszudrucken.*

Anhand der obigen Beispiele wird klar, dass Systeme, die ECA-Regeln realisieren, zwei Fähigkeiten haben müssen:

- Sie müssen bestimmte Situationen selbstständig erkennen
- Sie müssen bestimmte Reaktionen ausführen können

Diese Anforderungen an die Systeme werden durch ECA-Regeln verwirklicht. Die Struktur von ECA-Regeln erfordert, dass Situationen durch Ereignisse und Bedingungen repräsentiert werden. Die Reaktionen werden durch Aktionen beschrieben.
Die definierten Ereignisse werden durch die Systeme überwacht. Tritt ein Ereignis ein, werden die dadurch ausgelösten Regeln ermittelt und verarbeitet. Dabei umfasst die Bearbeitung nicht nur die durch die Regel bestimmte Aktion, sondern davor die Überprüfung der von der Regel geforderten Bedingung. Ist die Bedingung nicht erfüllt, wird die Regelverarbeitung sofort beendet.
Ein reaktives Verhalten von Systemen kann also durch ECA-Regeln verwirklicht werden. Dabei hat die Benutzung von ECA-Regeln den Vorteil,dass sie ein einheitliches Instrument zur Verwirklichung von Systemen mit reaktiven Verhalten darstellen.

ECA-Regeln lassen sich folgendermassen kurz beschreiben:

- Eine ECA-Regel umfasst eine Menge von Ereignissen, eine (optionale) Bedingung, und eine Aktion.

- Eine ECA-Regel wird genau dann ausgelöst, wenn ein in der Regel spezifiziertes Ereignis auftritt.
- Wenn die Regel eine bestimmte Bedingung beinhaltet, muss diese Bedingung für den aktuellen Zustand des Systems erfüllt sein, erst dann wird die Aktion der Regel ausgeführt.

2.1.1 Syntax und Semantik

Eine Ereignis-Bedingungs-Aktion Regel ist definiert als: $\{E\} : C \rightarrow A$, wobei:

- $\{E\}$ die Menge der auslösenden Ereignisse ist
- C ist eine zu prüfende Bedingung
- A ist die auszuführende Aktion

Ereignisse

Ereignisse lassen sich in zwei Arten gliedern, primitive und komplexe Ereignisse.

- **Primitive Ereignisse**
  Primitive Ereignisse können nicht weiter zerlegt werden. Sie werden als atomar bezeichnet.
  Beispiele für einige primitive Ereignisse:

  - Manipulationsereignisse
  Damit sind Ereignisse gemeint, die den Zustand eines Systems ändern (z.B. Löschbefehle in einem aktiven Datenbanksystem)
  - Zeit-Ereignisse
  Damit sind absolute Zeitpunkte gemeint. Zu einem bestimmten Zeitpunkt wird die Regel ausgeführt, z.B. um 12:00 Uhr am 31.12.2006.
  - Applikationsereignisse
  Dies sind Ereignsse, die auf der Applikationsebene auftreten. Diese müssen also vom Anwendungsprogramm überwacht und erkannt werden.

- **Komplexe Ereignisse**
  Komplexe Ereignisse müssen von der Regelsprache des Systems unterstützt werden. Komplexe Ereignisse setzen sich aus weiteren Ereignissen zusammen, häufig sind dies primitive Ereignisse, möglich sind aber auch weitere komplexe Ereignisse. Die Ereignisse sind durch Operatoren miteinander verknüpft (wobei durch diese Operatoren die Ereignisse immer weiter verschachtelt werden können).

  Einige Operatoren für komplexe Ereignisse:

  - Boolsche Operatoren
  z.B. $E_i \wedge E_j$ oder $E_i \vee \neg E_j$
  - Auswahl-Operatoren
  Mit Hilfe einer natürlichen Zahl und einer nichtleeren Menge von Ereignissen wird festgelegt, wie oft bestimmte Ereignisse erkannt werden müssen, damit das komplexe

Ereignis erkannt wird.
Z.B. tritt E_k ein, wenn E_i und E_j aus der spezifizierten Menge von Ereignissen eintreten und zwei Ereignisse aus dieser Menge reichen, damit das komplexe Ereignis eintritt.
- Sequenz-Operatoren
Mit Sequenz-Operatoren wird eine bestimmte Ereignisreihenfolge festgelegt, die, wenn sie auftritt, dann ein komplexes Ereignis auslöst.
Z.B. tritt E_k dann ein, wenn die Ereignisse E_1,E_2 und E_3 in dieser Reihenfolge aufgetreten sind.
- Zeit-Operatoren
Mit diesen Operatoren werden relative Zeit-Ereignisse (z.B. 5 Sekunden nach Aufhängen des Telefonhörers) und periodische Zeit-Ereignisse (z.B. alle 10 Sekunden) festgelegt.
- Intervall-Operatoren
Intervall-Operatoren werden für komplexe Ereignisse benötigt, die auftreten, wenn ein Ereignis innerhalb bestimmter Ereignisse erkannt werden soll.
Z.B. tritt E_k ein, wenn E_i innerhalb von E_h und E_j aufgetreten ist.
- Wiederholungs-Operatoren
Diese Operatoren werden benötigt, wenn ein komplexes Ereignis E_k dann eintreten soll, wenn ein Ereignis E_i x-mal eingetreten ist.
Z.B. E_k wird beim fünften, achten Eintreten usw. von E_i erkannt.

Bedingungen

Die Bedingung einer Regel wird ausgewertet, nachdem die Regel ausgelöst wurde. Dabei wird die Bedingung erfüllt, wenn sie den Wert TRUE annimmt; ansonsten ist sie verletzt. Es wird zwischen primitiven und komplexen Bedingungen unterschieden.

- *Primitive Bedingungen*
  Primitive Bedingungen können aus zwei Operanden zusammen gesetzt werden, die durch Vergleichsoperationen wie <, >, >=, <=, != verknüpft werden. Möglich ist auch der Aufruf von Prozeduren, die TRUE oder FALSE zurückliefern oder auch eine Eingabe des Benutzers erfordern können.

- *Komplexe Bedingungen*
  Komplexe Bedingungen setzen sich in der Regel aus primitiven Bedingungen zusammen, die durch Boolsche Operatoren AND, OR und/oder NOT verknüpft sind.

Aktionen

Durch die Aktion wird auf bestimmte Situationen reagiert. Die Aktion tritt ein, wenn das Ereignis erkannt wurde und die Bedingung erfüllt ist. Es kann zwischen primitiven und komplexen Aktionen unterschieden werden.

- *Primitive Aktionen*
  Primitive Aktionen sind atomar, bestehen also aus genau einer vom System zur Verfügung gestellten Aktion. Z.B. können Prozeduren aufgerufen, Meldungen ausgegeben oder der Zustand des System geändert werden.

- *Komplexe Aktionen*
  Komplexe Aktionen werden aus einer Folge von primitiven Aktionen gebildet.

Ausführungssemantiken

Nicht nur die Spezifikation der Regeln reicht allein aus, um ein ECA-Regelsystem aufzubauen. Weiterhin ist auch die Spezifikation der internen Verarbeitung von mehr als einer Regel notwendig, z.B durch die Vergabe von Prioritäten. Dies bekommt besondere Bedeutung im Zusammenhang mit Eigenschaften der Regelsysteme wie Terminierung oder Konfluenz.

Auslösungszeitpunkte

Für Regeln muss festgelegt werden, ob sie vor oder nach einem auslösenden Ereignis ausgeführt werden. Z.B. hat dies Bedeutung, wenn sie durch einen Befehl oder eine Prozedur ausgelöst wurden, durch den ein Zustand des System geändert wird und vorher eine Überprüfung der Benutzerrechte nötig ist. Mögliche Zeitpunkte der Regelausführung sind vor dem auslösenden Ereignis oder danach.

Fristen

Fristen werden dann notwendig, wenn z.B. Echtzeit-Systeme eine zeitlich bestimmte Reaktion erfordern.

Beispiel 2.3 *In Kernkraftwerken dürfen bestimmte, von Sensoren erfasste Temperaturen nicht überschritten werden. Werden sie überschritten, muss sofort eine Kühlung eingeleitet werden. Beginnt die Kühlung nicht innerhalb einer Minute, muss das gesamte System sofort abgeschaltet werden.*

Durch das Beispiel wird deutlich, dass auf bestimmte Situationen innerhalb des Systems reagiert werden muss. Ist das System überlastet und eine fristgerechte Verarbeitung nicht möglich, muss die Ersatzreaktion mit einer erhöhten Priorität ausgeführt werden.

Granularitäten

Mit Granularitäten wird für eine ECA-Regel festgelegt, wie oft diese auszuführen ist. Dies trifft insbesondere auf Regeln zu, die durch Manipulationsereignisse ausgelöst werden. Z.B. wird eine Regel in einer aktiven Datenbank in jeder Instanz ausgeführt, auf den der regelauslösende Befehl zugreift.

Prioritäten

In ECA-Regelsystemen können mehrere Regeln bestehen, die in der gleichen Situation ausgelöst werden. Dann besteht die Frage, welche Regel als erstes ausgewertet werden soll. Dafür werden an die Regeln Prioritäten vergeben, die dann bestimmen, welche Regel als erstes verarbeitet wird, welche Regel als zweites usw. Dabei wird zwischen absoluten und relativen Prioritäten unterschieden:

- **Absolute Prioritäten** Es wird ein numerischer Wertebereich festgelegt. In diesem Wertebereich liegen die Prioritäten als numerische Werte. Der maximale Wert hat die höchste Priorität, der minimale Wert die niedrigste Priorität.
- **Relative Prioritäten** Relative Prioritäten werden zwischen zwei Regeln festgesetzt. Dabei wird eine Regel mit höherer Priorität spezifiziert, diese wird dann vor der anderen Regel ausgewertet.

Ist ein Ereignis auslösend für mehrere Regeln, wird als erstes immer die Regel verarbeitet, die die höhere Priorität hat. Das bedeutet, die Regelverarbeitung folgt folgendem iterativem Algorithmus:

1. Eine ausgelöste Regel r wird für die weitere Regelverarbeitung so ausgewählt, dass keine gleichzeitig ausgelöste Regel eine höhere Priorität hat.
2. Die Bedingungen der Regel r werden geprüft.
3. Wenn die Bedingungen von r erfüllt sind, wird die Aktion von r ausgeführt.

Ist die Regel r abgearbeitet, beginnt der Algorithmus wieder bei 1., bis es keine zu verarbeitende Regel mehr gibt.
Werden drei Regeln r_i,r_j und r_k gleichzeitig ausgelöst, folgt daraus eine Transitivität der Regeln: wenn $r_i<r_j$ (d.h. r_j hat eine höhere Priorität als r_i) und $r_j<r_k$, dann auch $r_i<r_k$. Werden aber nur r_i und r_k ausgelöst, folgt nicht automatisch $r_i<r_k$!
Zwei Regeln r_i und r_j sind *geordnet* , wenn $r_i<r_j$ oder $r_j<r_i$. Ansonsten heißen sie *ungeordnet* .

Kopplungsmodi

Regeln werden im Normalfall ausgeführt, wenn ein spezifiziertes Ereignis eintritt, d.h. die Bedingung wird sofort geprüft und die Aktion gegebenenfalls verarbeitet. Dies muss aber nicht immer sinnvoll sein. So können z.B. bestimmte Bedingungen erst sinnvoll geprüft werden, wenn der Befehl, durch den die Regel ausgelöst wurde, verarbeitet wurde.

Parameterkontexte

Komplexe Ereignisse werden durch mehrere primitive Ereignisse ausgelöst. Dabei können primitive Ereignisse im Zeitablauf einmal oder mehrmals auftreten. Wenn ein komplexes Ereignis eintritt, muss entschieden werden, welche Parameter der primitiven Ereignisse, die mehrmals aufgetreten sind, für die Regelverarbeitung benutzt werden sollen. Das ist dann notwendig, wenn ECA-Regeln zur Überprüfung von transitionalen und dynamischen Integritätsbedingungen benutzt werden. Diese Bedingungen kontrollieren die Zulässigkeit von einem Zustandsübergang oder mehreren Zustandsübergängen. Für diese Kontrolle muss bei der Ereigniserkennung die erforderlichen Parameter protokolliert werden. Bei der Auswertung der Bedingungen und der Verarbeitung der Aktionen (z.B. geben die Parameter den Ort von zu löschenden Daten an) muss sich auf diese Ereignisparameter bezogen werden. Dann stellt sich das Problem, welche Parameter der primitiven Ereignisse, die mehrmals aufgetreten sein können, für die anschließende Regelverarbeitung, die durch das komplexe Ereignis ausgelöst wird, verwendet werden?
Als Lösung dieses Problems werden Parameterkontexte benutzt. Dazu werden die zu benutzenden Daten vorher spezifiziert. Es können vier Arten der Spezifikation unterschieden werden. Die vier Arten sollen anhand eines Beispiels erläutert werden:

Beispiel 2.4 *Parameterkontexte*
Die beiden komplexen Ereignisse E_{k1} und E_{k2} sind durch die primitiven Ereignisse E_1, E_2 und E_3 folgendermaßen spezifiziert:

- E_{k1} *= die primitiven Ereignisse aus der Menge* $\{E_1,E_2\}$ *und dann* E_3
- E_{k2} *= erst* E_1 *und dann die primitiven Ereignisse aus der Menge* $\{E_2,E_3\}$

E_{k1} tritt dann ein, wenn vor E_3 die beiden primitiven Ereignisse E_1 und E_2 eingetreten sind. Die Reihenfolge, in der E_1 und E_2 eintreten, ist beliebig. E_{k2} tritt ein, wenn nach E_1 die primitiven Ereignisse E_2 und E_3 eintreten. Die Reihenfolge von E_2 und E_3 ist wieder beliebig.
Um die komplexen Ereignisse richtig zu erkennen, muss die Reihenfolge der primitiven Ereignisse protokolliert werden. Für die folgenden Beispiele wird davon ausgegangen, dass die primitiven Ereignisse in der zeitlichen Reihenfolge $E_{2(1)}$:$E_{1(1)}$:$E_{1(2)}$:$E_{2(2)}$:$E_{1(3)}$:$E_{3(1)}$:$E_{3(2)}$ auftreten.

- **recent**
  Dadurch wird festgelegt, dass immer die Parameter der jüngsten Ereignisse benutzt werden. Für das Beispiel würde das bedeuteten, dass für E_{k1} die Parameter der Ereignisse $E_{1(3)}$,$E_{2(2)}$ und $E_{3(1)}$ benutzt werden.
  Dies eignet sich für Regeln, die immer nur die aktuellsten Parameter benötigen, wie beispielsweise in Überwachungssysteme in Krankenhäusern oder computergestützte Börsenprogramme.

- **chronicle**
  Für komplexe Ereignisse kann festgelegt sein, dass die Parameter der Teilereignisse in der chronologischen Reihenfolge genommen werden. Für das Beispiel heisst das, dass die beiden komplexen Ereignisse die Parameter der primitiven Ereignisse $E_{1(1)}$, $E_{2(1)}$ und $E_{3(1)}$ benutzen. Mit dem Eintritt von $E_{3(2)}$ wird E_{k1} ein zweites mal erkannt. Dann werden die Parameter der primitiven Ereignisse $E_{1(2)}$, $E_{2(2)}$ und $E_{3(2)}$ benutzt.
  Kausale Abhängigkeiten zwischen verschiedenen Ereignissen können durch diesen Parameterkontext überwacht und verwaltet werden, wie z.B. ein Rollback.

- **continuous**
  Die Parameter der primitiven Ereignisse können mehrmals verwendet werden. Dann werden die Parameter der Teilereignisse als potentielle Kandidaten betrachtet, die dann bei der Ereigniserkennung miteinander zu kombinieren sind. Daher können beim Eintreten des komplexen Ereignisses gewisse Regeln mehrmals ausgelöst werden. Auf das Beispiel bezogen bedeutet dies, das mit $E_{3(1)}$ die komplexen Ereignisse erkannt und mehrmals ausgelöst werden. E_{k1} wird insgesamt viermal ausgelöst, und zwar einmal mit den Parametern $E_{1(1)}$, $E_{2(1)}$ und $E_{3(1)}$, ein zweitesmal mit $E_{1(1)}$, $E_{2(2)}$ und $E_{3(1)}$, ein drittesmal mit $E_{1(2)}$, $E_{2(2)}$ und $E_{3(1)}$ und ein viertesmal mit $E_{1(3)}$, $E_{2(2)}$ und $E_{3(1)}$. E_{k2} wird zweimal ausgelöst, einmal mit den Parametern $E_{1(1)}$, $E_{2(2)}$ und $E_{3(1)}$ und ein zweitesmal mit den Parametern $E_{1(2)}$, $E_{2(2)}$ und $E_{3(1)}$.
  Dieser Parameterkontext eignet sich für Regeln, mit denen ein bewegliches Zeitfenster überwacht wird. Dies wird zum Beispiel bei Trendanalysen benutzt.

- **cumulative**
  Mit diesem Parameterkontext werden die Parameter aller primitiven Teilergebnisse berücksichtigt, die das komplexe Ereignisses auslösen. Für das Beispiel bedeutet dies, dass für E_{k1} und E_{k2} mit dem Eintritt von E_3 alle Parameter der primitiven Ereignisse folgendermaßen kombiniert werden: $E_{1(1)}$, $E_{1(2)}$, $E_{1(3)}$, $E_{2(1)}$, $E_{2(2)}$ und $E_{3(1)}$.

Zustände

Regeln können sich in zwei verschiedenen Zuständen befinden, im aktiven oder passiven Zustand.

- **aktiver Zustand**
  Regeln mit aktivem Zustand werden duch Ereignisse ausgelöst.
- **passiver Zustand**
  Regeln mit passivem Zustand werden nicht durch Ereignisse ausgelöst und also nicht verarbeitet. Diese Regeln bleiben aber trotzdem gespeichert.

Komponenten von Systemen zur Ausführung von ECA-Regeln

Für die Verarbeitung von ECA-Regeln sind Ereignisse zu erkennen, Bedingungen auszuwerten und Aktionen auszuführen. Systeme, die ECA-Regeln anwenden, müssen aus folgenden Komponenten bestehen:

1. **Ereigniserkennung**
   In der Ereigniskomponente wird spezifiziert, wodurch und wann die Regel ausgelöst wird. (Benutzer-)definierte Ereignisse müssen vom System überwacht werden, um ihr Eintreten richtig zu erkennen. Diese Aufgabe wird im System von der Ereigniskomponente erfüllt. Im Idealfall sollen Ereignisse selbständig vom System erkannt werden.
   Ereignisse können auch auf der Applikationsebene auftreten. Das Eintreten von diesen Ereignissen muss von Benutzern oder Anwendungsprogrammen ausgelöst werden.
2. **Bedingungsauswertung**
   In der Bedingungskomponente wird beschrieben, was nach der Regelauslösung zu überprüfen ist.
   Wird eine Situation als Ereignis erkannt, muss diese Komponente (benutzer-)definierte Bedingungen auswerten. Sie entscheidet, ob eine spezifizierte Situation oder nur eine ähnliche Situation existiert.
   Für die Auswertung in dieser Komponente können weitere Daten erforderlich sein, auf die die Komponete zugreifen können muss. Dies können also auch Abfragen sein, z.B. auf Datenbanken.
3. **Aktionsausführung**
   In der Aktionskomponente wird festgelegt, wie reagiert werden muss, wenn ein auslösendes Eregnis eintritt und sie die Bedingungen erfüllt.
   Die Ausführung von Aktionen geschieht in dieser Komponente. Hierzu kann es erforderlich sein, dass Daten benötigt werden, die in der Ereigniserkennung und Bedingungsauswertung verwendet wurden. Diese Daten müssen dieser Komponete also wieder zur Verfügung stehen.

2.1.2 ECA-Regelmengen

In diesem Abschnitt werden ECA-Regelmengen betrachtet. Als erstes werden zwei Darstellungsformen für ECA-Regelmengen vorgestellt. Zum Schluß werden Terminierung und Konfluenz, zwei besonders wichtige Eigenschaften von ECA-Regelmengen, beschrieben.

Triggering Graphs

Triggering Graphs werden nicht nur zur Visualisierung von ECA-Regelmengen genutzt, sondern auch von vielen Analysemethoden für die Analyse von Eigenschaften der Regelmenge. Ein Graph setzt sich dabei aus Knoten und gerichteten Kanten zusammen. Die Regeln werden durch die Knoten repräsentiert, die Regelabhängigkeiten unter den Regeln durch die Kanten. Z.B. gibt es eine Kante von dem Knoten r_i zum Knoten r_j, wenn durch die Verarbeitung der Regel r_i die Regel r_j ausgelöst wird. Dies passiert, wenn durch die Verarbeitung der Regel r_i das Ereignis der Regel r_j auftritt und die Bedingung der Regel r_j erfüllt ist.
Abb. 2 zeigt einen Triggering Graph.

Executions Graphs

Execution Graphs sind eine weitere Möglichkeit zur Visualisierung von Regelmengen. Execution Graphs bestehen aus Knoten und gerichteten Kanten. Ein Knoten repräsentiert dabei einen Zustand, in dem sich ein System befinden kann. Kanten repräsentieren eine Regel, durch deren Verarbeitung das System in den Zustand kommt, auf welchen die Kante zeigt.
Ein Execution Graph hat immer einen Startzustand und null oder mehr Endzustände. Startzustände stellen den Beginn der Regelverarbeitung dar. In Endzuständen terminiert die Regelverarbeitung, die entsprechenden Knoten haben also keine Kinder. Oder anders augedrückt: es werden keine weiteren Regel durch das Erreichen des (End-)Zustandes ausgelöst. Ein Weg im Execution Graph repräsentiert eine mögliche Ausführungsreihenfolge während der Regelverarbeitung. Wenn ein Knoten ein Kind oder mehr als ein Kind hat, gibt es genauso viele verschiedene Wege wie Kinder von diesem Knoten aus. Anders ausgedrückt, es gibt mehr als eine Regel, die untereinander keine Priorität haben und mit denen die Regelverarbeitung fortfährt. Sind für alle Regeln in der Regelmenge Prioritäten vergeben, gibt es nur einen möglichen Weg. Die Wege können unendlich lang sein (möglicherweise aber nicht auschließlich durch Kreise im Graphen).

Eigenschaften von ECA-Regelmengen

Für ECA-Regelsysteme muss sichergestellt sein, dass die Regelverarbeitung immer terminiert und zu einem eindeutigen Zustand führt. In den folgenden Abschnitten wird darauf eingegangen.

Terminierung

Es muss garantiert sein, dass die Verarbeitung von Regeln immer terminiert. Dies wird durch den Begriff Terminierung bezeichnet.

Definition 2.1 *Terminierung, termination [8]*

A rule set is guaranteed to terminate if, for any system state and initial modification, rule processing cannot continue forever (i.e., rules cannot activate each other idefinitely).

Dieses Problem kann durch Abhängigkeiten innerhalb der Menge der Regeln entstehen. Dadurch kann es z.B. passieren, dass durch die Verarbeitung einer Regel, eine oder mehrere weitere Regeln ausgelöst werden. Das passiert, indem durch die Verarbeitung der Regel die Ereignisse der

anderen Regeln erfüllt werden, deren Verarbeitung dann wieder andere Regeln auslöst. Dadurch entstehen sogenannte Regelkaskaden.
Im schlechtesten Fall entstehen durch Regelkaskaden ungewollte Zyklen, d.h. in einem Regelzyklus ist eine Regel enthalten, die sich immer wieder ausgelöst wird. Dies kann auf zwei Arten entstehen:

- *direkte Auslösung der Regel:* Die Regel löst sich selbst aus.
- *indirekte Auslösung der Regel:* Die Regel wird durch andere Regeln, die den Zyklus bilden, ausgelöst

Sind die Bedingungen immer erfüllt, wird die Verarbeitung dieser Regeln nie abbrechen.

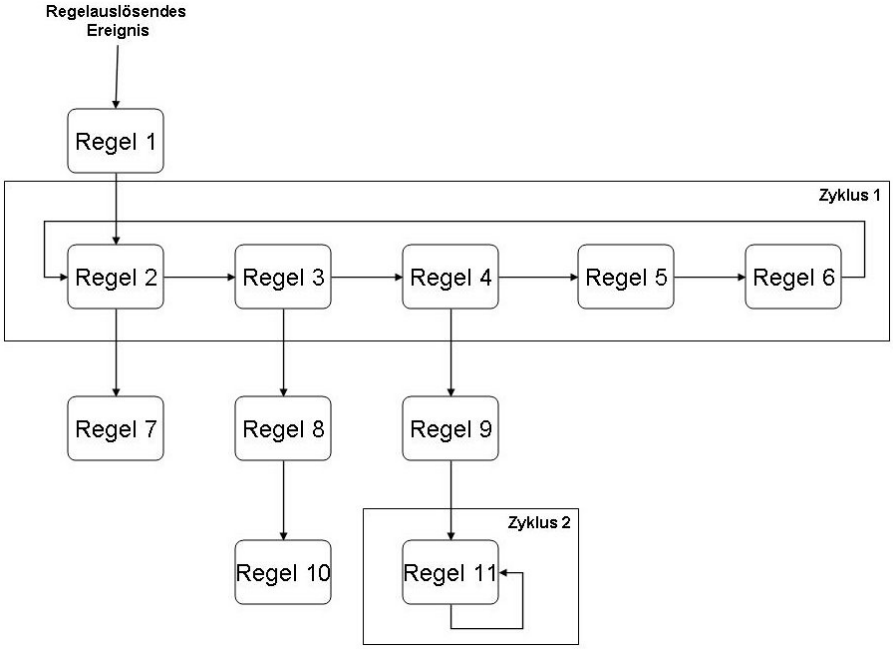

Abbildung 2.1: Regelkaskaden und Zyklen, [24].

Die Terminierung der Regelverarbeitung kann auf verschiedene Arten gewährleistet sein:

- Bei dem Entwurf eines Regelsystems müssen die Benutzer darauf achten, dass keine Zyklen in einem Triggering Graph entstehen bzw. es keine unendlichen großen Wege in Triggering Graphs gibt.
- Es wird ein Wert definiert, der die maximale Verarbeitungstiefe von Regelkaskaden begrenzt. Wird dieser Wert überschritten, wird die Verarbeitung von noch auszuführenden Regeln abgebrochen.
- Es gibt verschiedene Analysen um zu entscheiden, ob ein Regelsystem immer terminiert. Viele dieser Analysen (statische Analysen) basieren auf Triggering Graphs. Enthält dieser keinen Zyklus, ist die Terminierung der Regelmenge gesichert.
  Statische Analysen können nicht entscheiden, ob ein Zyklus nie abbricht. So könnte es der Fall sein, dass durch die Regel immer weiter Daten gelöscht werden, dabei aber keine

neuen Daten eingefügt werden. Dann terminiert die Regelverarbeitung, wenn es keine zu löschenden Daten mehr gibt.
In einem späteren Abschnitt wird eine statische Analysemethode vorgestellt.

Beispiel 2.5 *Terminierung*
In einem aktiven Datenbanksystem werden die Gehälter und der Rang der Angestellten verwaltet. Es gibt folgende Regeln in dieser Datenbank:

- ***Regel*** r_1***:*** *Wird das Gehalt eines Angestellten um mehr als 100 erhöht, wird sein Rang um 1 erhöht*
- ***Regel*** r_2***:*** *Wird der Rang eines Angestellten um 1 erhöht, wird auch sein Gehalt um das zehnfache seines neuen Ranges erhöht.*

Wird der Rang eines Angestellten größer als 10 oder das Gehalt eines Angestellten mit einem Rang von mindestens 10 um 100 erhöht, lösen sich die beiden Regeln immer wieder gegenseitig aus. Die Regelverarbeitung terminiert nie und läuft unendlich weiter.

Konfluenz

Mit dem Begriff Konfluenz (confluence) wird das Problem bezeichnet, dass ein System in einen eindeutigen Zustand beenden muss, unabhängig von der Ausführungsreihenfolge der Regeln. Konfluenz ist wie folgt definiert:

Definition 2.2 *Konfluenz, confluence [8]*

A rule set ist confluent if, for any system state and intitial modification, the final system state after rule processing is independent of the order in which activated rules are executed.

Um das Problem anschaulicher zu machen, wird ein Beispiel gegeben:

Beispiel 2.6 *Es wird ein aktives Datenbanksystem betrachtet. In diesem Datenbanksystem werden durch ein Ereignis die Regeln* r_1*,* r_2 *und* r_3 *ausgelöst.* r_1*,* r_2 *und* r_3 *haben alle die gleiche Priorität und die Bedingungen der Regeln sind alle erfüllt.*
Die Aktionen von r_1*,* r_2 *und* r_3 *sind Datenmanipulationsaktionen. So wird durch die Aktion von* r_1 *ein neuer Datensatz eingefügt, durch* r_2 *bestimmte Datensätze manipuliert und durch* r_3 *gewisse Datensätze gelöscht.*
Die Reihenfolge, in der die Regeln ausgeführt werden, ist durch die gleiche Priorität der Regeln nicht eindeutig festgeschrieben (vgl. Abb. 2). So ist es z.B. nicht ausgeschlossen, dass die von r_1 *eingefügten und/oder von* r_2 *veränderten Daten von* r_3 *gelöscht werden, wenn die Regeln in der Reihenfolge* r_1*,* r_2 *und* r_3 *ausgeführt werden. Werden die Regeln in der Reihenfolge* r_3 *und dann* r_2 *und* r_1 *verarbeitet, ist es ausgeschlossen, dass von* r_2 *manipulierte oder von* r_1 *neu eingefügte Datensätze gelöscht werden.*
Es ist offensichtlich, dass durch die Verarbeitungsreihenfolge der Regeln verschiedene Datenbankzustände erreicht werden können und Inkonsistenzen dadurch auftreten können. Um Inkonsistenzen zu vermeiden, muss garantiert sein, dass immer der gleiche Datenbankzustand erreicht wird, unabhängig von der Verarbeitungsreihenfolge von Regeln. Ist diese Bedingung erfüllt, ist das Regelsystem der Datenbank konfluent.

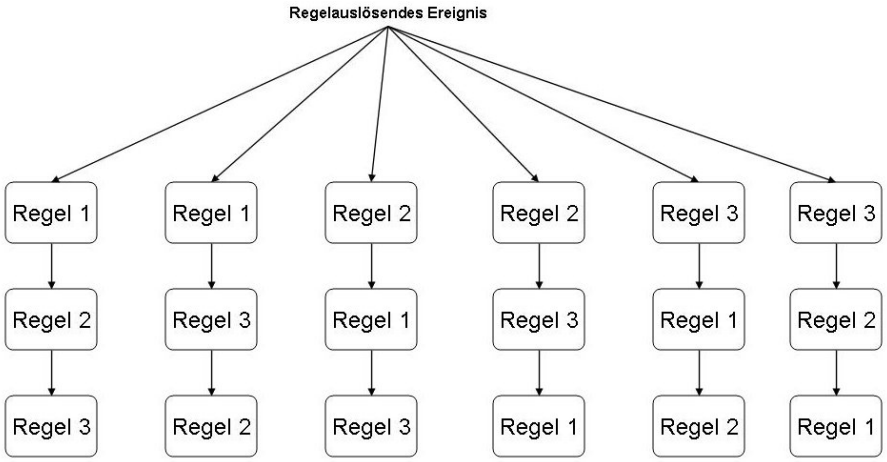

Abbildung 2.2: Ausführungsreihenfolgen der Regeln r_1,r_2 und r_3, [24] .

Wird eine Regelmenge R mit Hilfe von Execution Graphs dargestellt, ist sie genau dann konfluent, wenn jeder Execution Graph für R höchstens einen Endzustand hat.
Die Konfluenz eines Regelsystem kann auf verschiedene Arten erfüllt werden:

- Es wird garantiert, dass immer nur eine einzige Regel durch ein auslösendes Ereignis ausgelöst wird. Dies muss schon bei dem Entwurf des Regelsystem berücksichtigt werden.
- Durch Prioritäten wird garantiert, dass, werden mehrere Regeln durch ein Ereignis gleichzeitig ausgelöst, die Regeln immer in der gleichen Reihenfolge verarbeitet werden. Das kann z.B. dadurch erreicht werden, dass, wenn numerische Prioritäten vergeben werden, keine zwei oder mehr Regeln den gleichen numerischen Wert als Priorität erhalten.
- Die gegebene Definition von Konfluenz kann nicht dafür benutzt werden, um direkt die Konfluenz eines Regelsystems zu analysieren. Denn dazu müssten alle möglichen Ausführungsreihenfolgen für alle möglichen Startzustände des System geprüft werden.
  Deshalb wird die Regelmenge auf ihre Kommutativität untersucht.

Definition 2.3 *Kommutativität*

Zwei Regeln r_i und r_j sind kommutativ, wenn, egal in welchen Startzustand S begonnen wird, die Verarbeitung von r_i und dann von r_j in denselben Zustand S' führt wie die Verarbeitung von r_i und dann von r_j (2.3)).

Es ist offensichtlich, dass jede Regel zu sich selbst kommutativ ist. Zwei Regeln sind immer dann kommutativ, wenn sie sich nicht gegenseitig auslösen und/oder anderweitig beeinflussen können und ihre Verarbeitungsreihenfolge somit unwichtig ist.
Es gilt:

Eine Regelmenge R ist konfluent, wenn alle Regelpaare in R kommutativ sind.

Mit folgendem Beispiel wird gezeigt, dass die Konfluenz einer Regelmenge nicht immer leicht zu erkennen ist. Es werden drei Regeln spezifiziert, die auf jeden Fall terminieren. Den Regeln werden unterschiedliche Prioritäten vergeben, so dass es scheint, dass die Regeln konfluent sind. Dann wird gezeigt, dass die Regeln nicht konfluent sind.

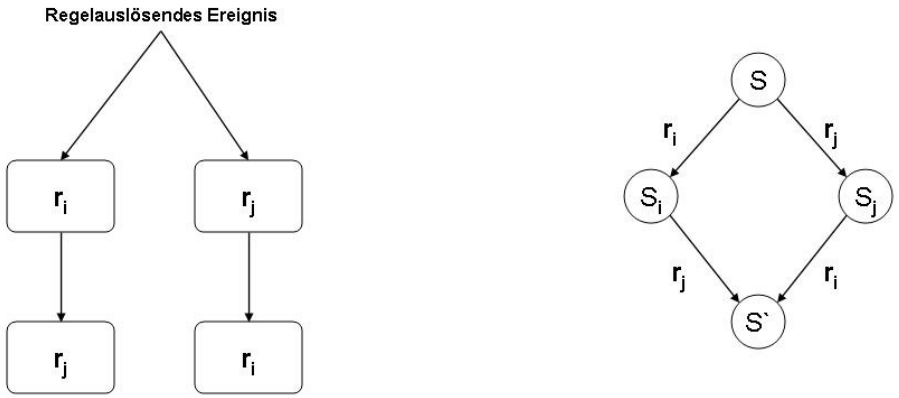

Abbildung 2.3: Kommutativität der Regel r_i und r_j, [24].

Beispiel 2.7 *Gegeben ist eine aktive Datenbank, die zur Verwaltung der Gehälter von Angestellten benutzt wird. Folgende Regeln wurden für die Datenbank spezifiziert:*

- ***Regel*** r_1***:*** *Erhöhe das Gehalt eines Angestellten um 10 immer dann, wenn er Waren für mehr als 50 in einem Monat verkauft.*
- ***Regel*** r_2***:*** *Erhöhe den Rang eines Angestellten um 1 immer dann, wenn der Angestellte Waren für mehr als 100 im Monat verkauft.*
- ***Regel*** r_3***:*** *Erhöhe das Gehalt eines Angestellten um 10% immer dann, wenn der Angestellte den 15ten Rang erreicht.*

Ein Ereignis kann gleichzeitig r_1 *und* r_2 *auslösen, aber die Aktion einer Regel hat nie eine Auswirkung auf die jeweils andere Regel. Es ist unwichtig, in welcher Reihenfolge* r_1 *und* r_2 *verarbeitet werden. Somit ist auch keine Priorisierung der Regeln nötig, um die Konfluenz der Regelmenge zu gewährleisten.*
Es kann vorkommen, dass r_3 *und* r_1 *gleichzeitig ausgelöst werden. Hier ist es offensichtlich, dass eine Priorisierung zwingend ist, um die Konfluenz für die Regelmenge zu gewährleisten.* r_3 *erhält eine höhere Priorität als* r_1*, so dass der Bonus von 10% unabhängig von dem Verkausfserfolg vergeben wird.*
Offensichtlich können auch r_3 *und* r_2 *gleichzeitig ausgelöst werden, so das auch hier eine Priorisierung notwendig wird. Wenn* r_2 *den Rang vor der Verarbeitung von* r_3 *erhöht, muss die Bedingung für* r_3 *nicht zwingend erfüllt sein (Rang = 15).*
r_3 *hat also nun eine höhere Priorität als* r_1 *und* r_2*.*
Trotzdem ist die Konfluenz der Regelmenge nicht gewährleistet. Angenommen r_1 *und* r_2 *werden zur selben Zeit für einen Angestellten ausgelöst. Der Angestellte erhält ein Gehalt von 60, hat im aktuellen Monat Waren im Wert von mehr als 100 verkauft und hat einen Rang von 14. Wenn* r_1 *als erstes verarbeitet wird, wird das Gehalt auf 70 erhöht, dann der Rang auf 15 und zum Schluss wird durch* r_3 *das Gehalt auf 77 ein weiteres mal erhöht.Nun wird davon ausgegangen, dass* r_2 *als erstes verarbeitet wird. Dann wird der Rang des Angestellten auf 15 erhöht, dadurch wird* r_3 *ausgelöst und erhöht das Gehalt auf 66. Zum Schluss wird das Gehalt nochmal auf 76 durch* r_1 *erhöht. Man erhält also zwei verschiedene Werte, die Regelmenge ist also nicht konfluent.*
Damit die Regelmenge konfluent wird, müssen die beiden Regeln r_1 *und* r_2*, die sonst nicht in Relation stehen, priorisiert werden. Dies ist für* r_3 *nicht mehr notwendig.*

2.2 ECA-Regelsysteme in der Praxis

Christian May, David Karla

Dieses Abschnitt soll das Verständnis für ECA-Regelsysteme durch Beispiele für deren praktischen Einsatz vertiefen. Anhand von Datenbanken und Workflow-Management-Systemen wird beschrieben, wie und warum ECA-Regeln in Systemen eingesetzt werden. Der letzte Teil befasst sich mit Sprachen für die Spezifikation von ECA-Regeln und in welchem Umfang man Regeln zwischen unterschiedlichen Systemen austauschen kann.

2.2.1 ECA-Regeln in aktiven Datenbanksystemen

Unterschied zu klassischen Datenbanksystemen

Ein Datenbanksystem besteht im Wesentlichen aus zwei Komponenten: Zum Einen aus der eigentlichen Datenbank, welche die Speicherung der Daten übernimmt und zum Anderen dem Datenbank-Management-System, welches die Schnittstelle für die Benutzer darstellt. Der Zweck eines Datenbanksystems besteht in der zentralen Datenhaltung, die einen Mehrbenutzerbetrieb erlaubt. Änderungen der Datensätze durch einzelne Benutzer sollen allen Anwendern direkt zur Verfügung stehen. Darüber hinaus ergeben sich weitere Vorteile durch die zentrale Datenhaltung, welche insbesondere eine zentrale Wartung und Pflege des Systems sind. Bei der klassischen Datenhaltung durch (auf mehrere Rechner) verteilte Dateien ergeben sich große Nachteile: Die Datenhaltung ist dabei exklusiv, also nur dem jeweiligen Benutzer zugänglich, was dazu führt, dass Daten mehrfach, nämlich auf jedem Rechner gespeichert werden müssen. Des Weiteren liegen die Daten in einem programmspezifischen Format vor und die Konsistenzprüfung muss durch dieses Programm zuverlässig gewährleistet werden. Insbesondere die verteilte Datenhaltung führt zu einem enormen Aufwand bei der Wartung, da beispielsweise von jedem Rechner ein Backup vorliegen muss und der Datenabgleich mit anderen Systemen ständig gewährleistet werden muss. Der Einsatz eines Datenbanksystems behebt diese Nachteile, wobei auch die zentrale Datenhaltung einen großen Nachteil hat: Fällt das Datenbank-Management-System aus, so kann kein Benutzer auf die Daten zugreifen. In der Praxis hat sich jedoch gezeigt, das diese Manko durch die Vorteile mehr als aufgewogen wird.

Aktive Datenbanken erweitern dieses Konzept um die Möglichkeit, auf bestimmte Ereignisse zu reagieren und somit Vorgänge zu automatisieren. Eine gängige Definition für ein aktives Datenbank System lautet nach Schlesinger [23]:

> Ein aktives Datenbanksystem ist ein System, welches Situationen von Interesse überwacht, und wenn diese auftreten, eine zeitgerechte adäquate Reaktion durchführt.

Daraus ergeben sich vielfältige Möglichkeiten, die ein passives System nicht bietet. Allen voran sind hier einfachste Mittel zur Integritätsprüfung der Daten zu nennen, wie es z.B. *NOT NULL*- oder *Schlüsselbedingungen* sind. Weitaus komplexere Integritätsprüfungen lassen sich durch so genannte *Trigger* realisieren, die später ausführlich beschrieben werden.

Konsistenz und Integrität

Unter *Konsistenz* versteht man nach Ritter die datenbankinterne korrekte Speicherung der Daten und der Verwaltungsinformationen, während *Integrität* die Korrektheit der Daten in Bezug ihres Kontextes ist [22].

Die reale (Geschäfts-) Welt wird auf die wesentlichen Daten reduziert. Ein Beispiel ist die Kontodatenbank einer Bank. In der Realität existiert zu jedem Konto ein Inhaber. Während es durchaus keine *Konsistenzverletzung* wäre in der Datenbank ein Konto ohne Inhaber zu speichern, stellt dies jedoch eine *Integritätsverletzung* dar, da die reale Welt nicht korrekt abgebildet ist. Es ist also festzuhalten, dass die *Integrität* verletzt sein kann, während die *Konsistenz* gewahrt bleibt.

Wenn ein aktives System die Wahrung beider Eigenschaften intern erfüllt, ist die Anwendungsentwicklung für Clientprogramme sehr viel einfacher, da entsprechende Prüfungen nicht redundant implementiert werden müssen, was letztendlich zu erhöhter Wirtschaftlichkeit führt.

Prinzipien für aktive Komponenten

Ein wesentlicher Punkt für die Benutzung von aktiven Systemen stellt also die Einhaltung der *Integrität* dar und es sollen nur Änderungen der Daten erlaubt sein, die diese Einhaltung sicherstellen. Dieser Grundsatz ist auch als *Golden Rule nach C. J. Date* bekannt [22]:

> Updatevorgänge werden nur erlaubt, wenn Sie das System in einen korrekten Zustand überführen und niemals, falls sie das System in einen inkorrekten Zustand überführen.

Integritätsbedingungen lassen sich nach Härder in mehrere Kategorien unterteilen [14]:

- Modellinhärente und sonstige Integritätsbedingungen

  Modellinhärente Bedingungen sind im Falle eines relationalen Datenmodells die referentielle Integrität der Fremdschlüssel und die Definition für Datenfelder (Datentyp, NOT NULL). Diese Bedingungen werden in der Regel bei der Erstellung der Datenbank berücksichtigt.

- Reichweite der Bedingung

  Hierbei erfolgt die Unterteilung aufgrund der Anzahl und Art der zur Integritätsprüfung herangezogenen Daten. *Attributwert-Bedingungen* ziehen statische Werte zur Prüfung in Betracht (z.B. Gehalt > 0), während *Satzbedingungen* zum Datensatz gehörende Felder heranziehen (z.B. neues Alter $>$ altes Alter). *Satzübergreifende Bedingungen* beziehen sich auf andere Datensätze der Datenbank (z.B. referentielle Integrität zwischen verschiedenen Tabellen)

- Statische und dynamische Bedingungen

  Statische Bedingungen garantieren den korrekten Zustand der Daten, indem sie inkorrekte ausschließen (z.B. Gehalt > 0 und Gehalt < 100000). Die Prüfung während eines Zustandsübergangs, also der Manipulation der Daten, übernehmen die dynamischen Bedingungen. Diese gewährleisten, dass der neue Zustand des Systems ein gültiger ist (z.B. neues Alter $>$ altes Alter). Dabei ist zu beachten, dass dynamische Bedingungen am Zustand der Datenbank nicht überprüfbar sind.

Die Datenqualität, worunter man die Übereinstimmung der Daten mit der abgebildeten „Miniwelt“ versteht, muss demnach durch die Bekanntmachung möglichst aller *Integritätsbedingungen* sichergestellt werden.

Trigger, Alerter und Regeln

Dieser Abschnitt befasst sich mit den technischen Möglichkeiten der Bekanntmachung aller für die *Integritätssicherung* relevanten Information in einem Datenbanksystem. Im Allgemeinen werden diese als *Stored procedures* bezeichnet, also Prozeduren, die in der Datenbank selbst gespeichert werden.
Trigger

Der Trigger (engl. Auslöser) ist eine Art der *stored procedures*. Ein Trigger dient in erster Linie der Korrektur des Datenbankzustands, nachdem ein Event in der Datenbank ausgelöst wurde. Ein Trigger selbst führt dabei zwingend nötige Änderungen an den Daten durch, um die Integrität zu bewahren. Dabei besteht die Reaktion auf ein Event aus einer Kette an Operationen.

```
AFTER <event> ON <table> DO <check -> operation_1, operation_2...>
```

Während in relationalen Datenbanken in der Regel nur Modifikationsoperationen der Daten als Event dienen, können in objektorientierten Datenbanken alle Methodenaufrufe ein Event darstellen. Ziel des aktiven Datenbanksystems ist die Erkennung der Events und die Durchführung aller nötiger Folgeoperationen. Ein Event kann dabei mehrere Triggerfunktionen auslösen, wobei die für regelbasierte Systeme typischen Probleme auftreten.

- Reihenfolge der Triggerausführung
- Terminierung
- Kontrolle:

  Änderungen lösen Trigger aus, die Daten ändern, wobei erneut Trigger ausgelöst werden.

Aufgrund dieser Probleme sollten Trigger mit Vorsicht eingesetzt werden und wenn möglich durch deklarative Integritätsbedingungen ersetzt werden (Datentypen, Wertebereiche).

Trigger werden etwa seit 1987 in auf dem Markt verfügbaren Datenbanksystemen eingesetzt. Mittlerweile existiert durch die SQL-Spezifikation von 1999 eine Standardisierung der Trigger. Demnach sind als *triggernde* Events die SQL-Operationen *INSERT, UPDATE und DELETE* zugelassen, was den Nachteil hat, dass einfache Überprüfungen, die bei allen Operationen ausgeführt werden sollen für jede Operation einzeln implementiert werden müssen. Der Zeitpunkt der Triggerausführung kann durch *BEFORE und AFTER* in Bezug auf das Event festgelegt werden. Eine verzögerte Ausführung, beispielsweise am Ende einer Transaktion, ist nicht möglich. Mit dem Schlüsselwort *WHEN* und der Beschreibung eines Zustandes kann der neue, durch die Datenänderung erreichte Zustand des Datensatzes auf Korrektheit geprüft werden und die Änderungen gegebenenfalls rückgängig gemacht werden. Des Weiteren werden die boolsche Operation *AND* und *OR* zur Bedingungsprüfung unterstützt.

Ein Beispiel aus Härder zeigt einen SQL-Trigger, der sicherstellt, dass das Gehalt einer Person nur erhöht werden kann [14]. Das auslösende *Event* ist dabei eine *UPDATE-Operation* der Spalte *GEHALT* in der Tabelle *PERS*.

```
CREATE TRIGGER GEHALTSTEST
  AFTER UPDATE ON GEHALT ON PERS
  REFERENCING OLD AS AltesGehalt
              NEW AS NeuesGehalt
  WHEN (NeuesGehalt < AltesGehalt)
    ROLLBACK;
```

Ein Nachteil der SQL-Trigger ist, dass sie nur durch Manipulation der Daten ausgelöst werden können. Ein Anwender der Datenbank kann die Trigger demnach nicht explizit aufrufen. In vielen Fällen ist es jedoch erwünscht, wenn die Datenbank auf Events von Außen reagieren kann, etwa durch Interaktion mit einem Benutzer oder einem anderen Programm. Solche von außen ausgelösten Events nennt man *Alerter*. Soll beispielsweise eine Nachbestellung von Rohstoffen zu einem bestimmten Zeitpunkt ausgeführt werden, so kann dies nicht durch einen Trigger gewährleistet werden. Wird eine *stored procedure* von Außen ausgelöst, so ist dem Datenbanksystem die Semantik seiner Reaktion völlig unbekannt, was ein Problem bei der Integration von Anwendungsprogrammen in das DBS darstellt.

Um eine Verallgemeinerung der Trigger-Funktionalität zu erreichen wird bei aktiven Datenbanksystemen zunehmend das Konzept der ECA-Regeln berücksichtigt. Trigger sind dabei als Teilmenge dieser Regeln zu verstehen. Bei den Regeln gibt es weitaus mehr auslösende Events. Diese können beispielsweise lesende Zugriffe auf die Datenbank, bestimmte Zeitpunkte, der Beginn oder das Ende einer Transaktion sein. Der *Condition-Teil* der Regeln ist mit den *WHEN-Klauseln* der Trigger vergleichbar. Neben den ECA-Regeln gibt es noch weitere Varianten, die eine ähnliche Struktur aufweisen. EA-Regeln werden dabei immer ausgeführt, ohne dass eine *Condition* geprüft wird. CA-Regeln verzichten auf ein spezifisches auslösendes Event (z.B. sie werden alle 5 Sekunden überprüft).

Im Gegensatz zu dem weitaus größeren Leistungsspektrum solcher Regelsysteme, steht der größere Verwaltungs- und Prüfungsaufwand. Insbesondere ist es viel aufwendiger, die regelauslösenden Events zu überwachen.

2.2.2 ECA-Regeln in Workflow Management Systemen

Was ist ein Workflow Management System ?

Mit Workflow Management bezeichnet man die computergestüzte Ausführung von strukturierten Arbeitsabläufen, in der Regel von Geschäftsprozessen. Ein Workflow-Management-System (WMS) dient der aktiven Steuerung der Prozesse. Darüber hinaus enthält ein solches System weitere Komponenten, mit deren Hilfe man Prozesse modellieren kann. Diese Komponenten bieten die Möglichkeit, Arbeitsabläufe auf einem hohen Abstraktionslevel zu beschreiben. Dazu wird eine geeignete Visualisierung der Prozesse zur Verfügung gestellt, etwa in Form von Ablaufdiagrammen. Da eine Diskrepanz zwischen High-Level-Design und der Ausführungsschicht des WMS besteht, müssen erstellte Workflows in ein entsprechendes Low-Level-Format umgewandelt werden, welches durch das WMS ausgeführt und überwacht werden kann.

Ein WMS Prototyp: TriGSflow

TriGSflow [16] ist ein Prototyp für ein Workflow-Management-System, an dem exemplarisch alle wichtigen Anforderungen aufgezeigt werden.
Designziele

Anpassungsfähigkeit: Arbeitsprozesse können sich stark voneinander unterscheiden. Es gibt beispielsweise wohlstrukturierte und weniger strukturierte Prozesse. Das System muss durch **ein** Prozessmodell alle Arten an unterschiedlichen Prozessen unterstützen. Um also eine adäquate Modellierung zu gewährleisten, müssen unterschiedliche Modellierungspradigmen unterstützte werden. Hier sind beispielsweise netzbasierte, rollenbasierte, proaktive und reaktive Ansätze zu nennen.

Flexibilität: Die Flexibilität eines WMFs wird durch zwei wesentliche Aspekte gekennzeichnet. Zum einen muss die Ausführungsschicht auf zukünftige Änderungen reagieren können, zum anderen müssen die Prozessmodelle leicht zu ändern sein.

Verteiltes System: Die verteilte Ausführung von Arbeitsprozessen ist eine wichtige Anforderung an ein WMS. Das System wird dadurch skalierbar und fehlertolleranter. Leider steht diese Anforderung teilweise im Widerspruch zur Flexibilität. Dieser Gegensatz muss durch ein WMS gelöst werden.

Wiedervewendbarkeit: Komplexere Prozesse setzen sich häufig aus weniger komplexen zusammen. Das System soll es ermöglichen, bereits spezifizierte Prozesse in andere integrieren zu können. In diesem Zusammenhang ist es sinnvoll, den Abstraktionsgrad der Modellierung verändern zu können.

Einfache Benutzbarkeit: Die Modellierungssprache soll es erlauben, auf einem hohen Abstraktionslevel Prozesse entwerfen zu können. Dazu müssen die Modellteile visualisiert werden, was bei einigen Modellierungspradigmen, wie den netzbasierten, von Natur aus gegeben ist.

Architektur: *TriGSflow* wurde auf Basis der objektorientierten Datenbank *GemStone* [6] entwickelt. Dazu wurde die Datenbank um weitere Module erweitert (siehe Abbildung 2.4), deren Funktionen nun im Detail erläutert werden.

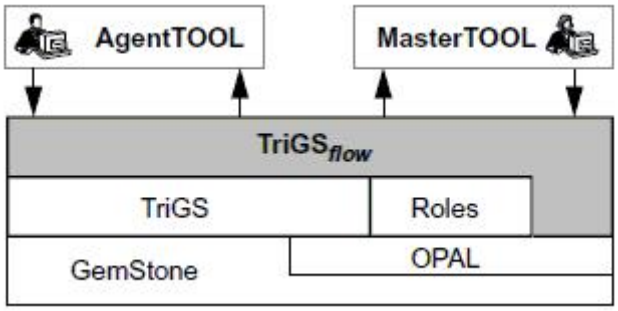

Abbildung 2.4: TriGSFlow: Architektur [16]

Datenbank *GemStone*: Dieses kommerzielle Datenbank-Management-System ist objektorientiert. Die Daten werden nicht relational in verschiedenen Tabellen verknüpft, sondern als eine Einheit, als Objekt, mit allen relevanten Informationen gespeichert. Durch die Ähnlichkeit zu dem objektorientierten Ansatz der modernen Programmiersprachen, lässt sich der Datenaustausch zwischen Applikationen und Datenbank für den Entwickler vereinfachen, da die Objekte seines Programms nahezu direkt in das DBMS übernehmen lassen.

OPAL: OPAL ist ein Derivat der Programmiersprache *Smalltalk80* und erweitert diese um eine einige Klassen für den Datenbankbetrieb.

Roles-Modul: In der Regel werden Objekte definiert, die genau von einer Klasse abstammen. Ist es nötig, ein Objekt zu verändern oder zu erweitern, so muss eine neue Instanz erzeugt und

die Attribute der alten Instanz müssen aufwendig übernommen werden. Das Konzept der *roles* soll es ermöglichen, während der Laufzeit die *Rolle* eines Objektes zu verändern oder sogar einem Objekt mehrere Rollen zuzuweisen. Hat man einmal einen Arbeitsablauf spezifiziert, kann es vorkommen, dass Aufgaben, die bisher von Menschen übernommen wurden, plötzlich durch Maschinen realisierbar sind. Durch das Rollenkonzept kann nun einfach die *Rolle* des Objekts geändert werden, ohne den Workflow neu modellieren zu müssen. Die Flexibilität und Wiederverwendbarkeit der Modelle wird dadurch erhöht.

TriGS: Dieses Modul stellt die ECA-Regeln inklusive ihrer Auswertung und Überwachung bereit. Es ist möglich, die Regeln in unterschiedlichen Granularitäten anzuwenden; Beispielsweise für alle Instanzen einer Klasse oder nur einige spezielle Instanzen, wobei es möglich ist dies während der Ausführung anzupassen. *TriGS* stellt ECA-Regeln für alle Phasen des Workflowmanagements bereit, was im Detail noch gezeigt wird.

AgentTOOL und MasterTOOL: Hierbei handelt es sich um zwei Applikationen, die ein grafisches Frontend zur Workflowmodellierung und -ausführung bieten.

Modellierung von Businessprozessen

Der Anwender eines WMS möchte Prozesse auf einem hohen Level (graphische Darstellung) modellieren, während ein Computerprogramm auf einem für Menschen weniger intuitiven niedrigeren Level arbeitet. Ein WMFS ist die Schnittstelle zwischen *high-level modelling* und *low-level enactment*. Während ein Computer sehr gut aus Basis von ECA-Regeln arbeitet, ist es für den Anwender nicht zumutbar, seine Modellierung ausschließlich durch Angabe solcher Regeln vorzunehmen.

Was macht einen Businessprozess aus?

Die Modellierung der Prozesse lässt sich bei der Anwendung von *TriGSflow* in mehrer Phasen unterteilen:

- Modellierung der organisatorischen Struktur, welche aus Abteilungen, Agenten und Rollen besteht
- (Zeitliche) Ordnung der Aktivitäten (*control flow)*
- Zuteilung von Agenten zu den Aktivitäten durch Spezifikation von Taktiken für die Agentenauswahl
- Festlegung der Kommunikation zwischen den Agenten (data flow)

Abbildung der organisatorischen Struktur: Für die Modellierung müssen grundlegende Objektklassen bereitgestellt werden. Diese orientieren sich an den Strukturen eines Unternehmens. Eine Organisation lässt sich in *Abteilungen* unterteilen, welche wieder Unterabteilungen haben können. *Agenten* übernehmen die Durchführung der Arbeitsprozesse, wobei *TriGSflow* hier zwischen Menschen (*HumanAgent)* und Software (*SofwareAgent*) unterscheidet. Aktivitäten, die durch Menschen bearbeitet werden, lassen sich interaktiv beeinflussen, während dies bei durch Software bearbeiteten Aktivitäten nicht der Fall ist. Software repräsentiert dabei real existierende Maschinen, wie Faxgeräte, Taschenrechner oder andere Computerprogramme. Jeder Agent wird einer *Rolle* zugeordnet, welche eine eindeutige kontextabhängige Zuordnung darstellt. Beispielsweise kann ein Faxgerät in unterschiedlichen Prozessen zum Einsatz kommen, wobei es stets der gleiche *Agent* ist, aber eine andere *Rolle* hat.

Aktivitäten ordnen: *Aktivitäten* müssen in einer bestimmten Reihenfolge ausgeführt werden, welche sich aus deren Beziehungen untereinander ergibt. Grundlegende Kontrollstrukturen wie die einfache Hintereinanderausführung (*sequencing*) von Aktivitäten, die Aufsplittung des Workflows (*branching*) in mehrere Folgeaktivitäten und die Zusammenführung (*joining*) werden zur Verfügung gestellt. Die Aufteilung, also das *branching*, lässt sich noch genauer kategorisieren, indem man *exOR-Branching* (es gibt mehrere mögliche Folgeaktivitäten, aber nur eine ist zulässig), *OR-Branching* (eine oder mehrere mögliche Folgeaktivitäten) und *AND-Branching* (alle möglichen Folgeaktivitäten werden ausgeführt) unterscheidet. Hat eine Aktivität mehr als einen Vorgänger, wir dies durch ein *AND-Join*, *exOR-Join* oder *inOR-Join* realisiert.

Diese grundlegenden Kontrollmechanismen lassen sich in ECA-Regeln überführen, welche die Reihenfolge der Aktivitäten abbilden. Das *Event* einer solchen Regel ist dabei stets die Beendigung der vorangegangenen Aktivität (bei *AND-Joining* Beendigung aller). Die *Condition* prüft dabei, ob die angestoßene Aktivität ausgeführt werden soll oder nicht, etwa indem geprüft wird, ob bestimmte Daten kommuniziert worden sind und somit zur Weiterverarbeitung vorliegen. Das Branching kann durch Regeln erfasst werden, indem man beim *AND-Branching* die *Conditions* aller Regeln identisch hält und beim *exOR-Branching* sich die Bedingungen gegenseitig ausschließen. Beim *OR-Branching* dürfen sich die Bedingungen überlappen. Die *Action* muss dafür sorgen, dass der *Agent* benachrichtigt wird, der die Aktivität ausführt. Dazu hat jede *Rolle*, die Methoden **perform** und **notifyAgent**.

Beispiel einer Regel aus Kappel [16]:

```
DEFINE RULE R_GetOffers
    ON POST (Activity, perform: aFolder) DO
    IF Offers notUpToDateFor: ((aFolder at: 'Order') positions) THEN
    EXECUTE a_get_offers notifyAgent
ACTIVATED FOR (a_create_order_form)
```

Die Parameterübergabe erfolgt in Zeile zwei. Hier wird angegeben, auf welchen Daten (**aFolder**) die Prüfung vorgenommen werden soll. Zeile drei führt die Prüfung durch und Zeile vier beschreibt die *Action*. In der letzten Zeile wird definiert, für welche *Rolle* die Regel aktiv ist. Die Aktivitäten lassen sich also durch Regeln dieser Art ordnen.

Agenten wählen/zuteilen: Während der Ausführung eines Prozesses ist es notwendig, für jede durchzuführende Aktion einen Agenten auszuwählen, der die Ausführung übernimmt. Die Auswahltaktik wird dabei durch ECA-Regeln realisiert. Ein Beispiel für eine Auswahltaktik ist, dass beurlaubte Mitarbeiter keine Tätigkeit übernehmen können. Zunächst wird für jede Aktion eine Menge an möglichen Agenten (*possible agents*) festgelegt. Aus dieser Menge muss zur Laufzeit ein Agent gewählt werden (*actual agent*). Um die Flexibilität zu erhöhen, soll der Anwender verschiedene Auswahltakiken definieren können und während der Laufzeit zwischen diesen wechseln können.

Ein Beispiel:

```
DEFINE RULE R_NoIllEmployee
    ON PRE (Activity, notifyAgent) DO
    IF (rule_trg0 possibleAgents) selNoIllEmployees THEN
    EXECUTE (rule_trg0 actAgRole) removeAll; add: QUERYRESULT
ACTIVATED FOR (a_select_supplier)
```

Die Prüfung wird vorgenommen bevor (**ON PRE**) die **notifyAgents**-Methode aufgerufen wird. In Zeile drei wird ein möglicher Agenten ausgewählt, indem die Methode **sellNoIllEmployees**

aufgerufen wird. Die Rückgabe dieser Methode, welche der Auswahl der Agenten entspricht, wird im **QUERYRESULT** gespeichert. Die *Action* in Zeile vier löscht zunächst die alte Menge der Agenten und ersetzt sie durch die neue Auswahl (**QUERYRESULT**).

Worklist Management: In *TriGSFlow* werden den Agenten so genannte *worklists* zugeordnet. In diesen Listen stehen alle Aktivitäten, die der Agent noch durchzuführen hat. Zu jeder Aktivität wird ein Ordner (*folder*) angegeben, welcher die zur Bearbeitung nötigen Informationen und Daten enthält. Jedes mal, wenn ein Prozess neu gestartet wird, wird ein Ordner für den Datenaustausch während des Prozesses angelegt. Dies kann durch einen Agenten realisiert werden, der berechtigt ist, den Prozess zu initiieren. Die Einträge in den Listen der Agenten enthalten dabei eine Referenz auf den neuen Ordner. Der gleichzeitige Zugriff auf die Ordner durch mehrere Agenten wird durch die Datenbank überwacht. Sollte es in besonderen Fällen nötig sein, dass Kopien der Ordner angelegt werden müssen, so muss eine weitere Kontrollinstanz bereitgestellt werden, die mögliche Intigritätsverletzungen unterbindet.

ECA-Regeln steuern die Bearbeitung der Ordner in den TODO-Listen der Agenten. Agenten sind während der Bearbeitung *aktiv* und ansonsten *inaktiv*. Wird eine Software durch einen Agenten repräsentiert so wird dieser inaktiv sobald ein Eintrag der Liste abgearbeitet und entfernt wurde. Dieses Ereignis kann als *Event* einer ECA-Regel verwendet werden:

```
DEFINE RULE R_StartOnRemove
    ON POST (Worklist, remove) DO
    IF rule_trg0 isEmpty isFalse THEN
    EXECUTE ((rule_trg0 nextWorklistItem) activity) perform:
            ((rule_trg0 nextworklistItem) folder)
ACTIVATED FOR (SoftwareAgent collect:[:x|x worklist]).
```

Nachdem die Regel ausgelöst wurde, wird geprüft, ob es noch weitere Einträge in der Arbeitsliste des Agenten gibt. Ist dies der Fall, so wird die nächste Aktivität aus der Liste gestartet (**perform**) und als Parameter der entsprechende Ordner (**folder**) übergeben. In der letzten Zeile wird festgelegt, dass diese Regel für Agenten des Typs **SoftwareAgent** gilt.

Durch den Entwurf des Workflow-Management-Systems *TriGSflow* wurde gezeigt, wie sich wesentliche Teile der Prozessausführung durch ECA-Regeln realisieren lassen. Dem Prototypen fehlen jedoch noch Komponenten zum verteilten Zugriff auf die Daten und der Kontrolle der Regeln während der Ausführung, wobei insbesondere die Terminierung zu nennen ist.

2.3 Feature Interaction und Beschreibung eines Telefonsystems als Regelsystem

Christian Frost

Das Problem Feature Interaction tritt in serviceorientierten Softwaresystemen auf. Vor allem softwarebasierte Telekommunikationssysteme werden durch ein Basissystem und verschiedene zusätzliche Dienste spezifiziert, die Features genannt werden.

Das Basissystem und die Features werden einerseits durch invariante Eigenschaften beschrieben, die immer erfüllt sein müssen. Andererseits definieren die Dienste Regeln, die angeben, wie das System von einem Zustand in einen neuen Zustand übergeht. Diese Zustandsübergänge werden durch Ereignisse ausgelöst.

Bei der serviceorientierten Spezifikation können aber unter anderem die folgenden Inkonsistenzen auftreten:

- Im System können Invarianten enthalten sein, die sich gegenseitig widersprechen.
- In einem bestimmten Zustand können zwei oder mehrere verschiedene Transitionsregeln anwendbar sein. In diesem Fall ist die Systemspezifikation nichtdeterministisch.
- Durch Zustandsübergangsregeln können Invarianten verletzt werden.

Die beschriebenen Inkonsistenzen bezeichnet man als unerwünschte Feature Interactions.
Das Problem von Feature Interactions tritt vor allem bei der Integration von neuen Diensten (Features) in ein bestehendes System auf. Diese Inkonsistenzen möchte man schon während der Entwurfsphase entdecken und beseitigen, weil der Aufwand zur Systemänderung in späteren Phasen viel größer ist.

Die in den folgenden Abschnitten vorgestellten Ergebnisse und Beispiele stammen aus der Arbeit von Aiguier, Berkani und Le Gall [4], deren Ziel die Entwicklung eines interaktiven Verfahrens zur Erkennung und Auflösung von Inkonsistenzen bei der Integration von neuen Diensten in ein Softwaresystem ist.
Die Entdeckung der Inkonsistenzen ist durch geeignete Algorithmen automatisch durchführbar, während die Entscheidungen über die Behebung der Inkonsistenzen von Experten getroffen werden. Die Experten müssen die Regeln bzw. Eigenschaften der Dienste so modifizieren, dass das gewünschte Systemverhalten erreicht wird.
Mögliche Änderungen der Dienste sind hierbei die Modifikation und die Entfernung von Regeln aus der Spezifikation, sowie die Hinzufügung neuer Regeln zur Konsistenzsicherung.

Zunächst wird im folgenden Abschnitt 2.3.1 eine Formulierung von Dienstspezifikationen in einer statischen logischen Syntax angegeben.
Der Abschnitt 2.3.2 beschreibt als Beispiel die Dienstspezifikationen eines Telefonsystems, das aus einem Basissystem und verschiedenen zusätzlichen Diensten besteht.
Beispiele für Feature Interactions, die durch die Integrationsmethode im Telefonsystem erkannt und behoben werden konnten, werden im abschließenden Abschnitt 2.3.3 vorgestellt.

Eine temporallogische Behandlung des Problems Feature Interaction mit Methoden des Model Checking wird bei Felty und Namjoshi [9] beschrieben.
Weitere Arbeiten zum Thema Feature Interaction findet man zum Beispiel im Buch von Gilmore und Ryan [10].

2.3.1 Syntax von Dienstspezifikationen

Zur Spezifikation von Diensten bzw. Features wird ein spezielles statisches logisches System mit der Bezeichnung STR eingesetzt, dessen Syntax ähnlich zur Definition von ECA-Regelsystemen ist.
Es werden also keine temporallogischen Formeln verwendet.

Die Signatur des Systems STR enthält im Falle von Telefonsystemen Symbole zur Beschreibung von Abonnements bestimmter Dienste durch Nutzer des Systems, von Zuständen der Telefonendgeräte und von Ereignissen, welche die Zustandsübergänge auslösen.

Definition 2.4 (Signatur) *Eine Signatur ist ein Tripel $\Sigma = (St, Sb, E)$, das aus den folgenden disjunkten Mengen besteht:*
St : Menge von Prädikatennamen für Zustände,
Sb : Menge von Prädikatennamen für Abonnements,
E : Menge von Ereignisnamen.
Eine Signatur Σ heißt endlich, wenn die drei Mengen St, Sb, E jeweils endlich sind.
Ein Element $p \in St \cup Sb \cup E$ mit der Stelligkeit $n \in \mathbf{N}$ wird als p^n bezeichnet.

Atomare Formeln in der Prädikatenlogik sind Prädikatensymbole, die auf Terme angewendet werden. Diese Terme werden aus Variablen und Funktionssymbolen aufgebaut.
Im hier verwendeten logischen System gibt es keine Funktionssymbole. Deshalb entsprechen die atomaren Formeln hier Prädikatensymbolen mit Variablen als Argumenten. Die Variablen aus der Menge X bezeichnen dabei jeweils Telefonendgeräte.
Die atomaren Formeln und die negierten atomaren Formeln werden nun formal definiert.

Notation 2.1 (Atomare Formeln) *Sei $\Sigma = (St, Sb, E)$ eine Signatur und sei X eine Variablenmenge. Mit $At_\Sigma(X)$ und $\overline{At}_\Sigma(X)$ werden die folgenden beiden Mengen bezeichnet:*

1. $At_\Sigma(X) = \{p(x_1, ..., x_n) | p^n \in St \cup Sb, x_i \in X, 1 \leq i \leq n\}$
2. $\overline{At}_\Sigma(X) = \{\neg p(x_1, ..., x_n) | p^n \in St \cup Sb, x_i \in X, 1 \leq i \leq n\}$

Mit $Sb_\Sigma(X)$ und $\overline{Sb}_\Sigma(X)$ (bzw. $St_\Sigma(X)$ und $\overline{St}_\Sigma(X)$) bezeichnet man die Teilmengen von $At_\Sigma(X)$ und $\overline{At}_\Sigma(X)$, die nur die Abonnementsprädikate Sb bzw. die Zustandsprädikate St enthalten.

Zur Spezifikation von Softwarediensten werden zwei verschiedene Arten von Formeln verwendet, nämlich Zustandsübergangsregeln und Invarianten.

Definition 2.5 (Formeln) *Seien $\Sigma = (St, Sb, E)$ eine Signatur und X eine Variablenmenge.*

1. *Eine Zustandsübergangsregel (STR-Formel) über Σ ist ein Satz der Form*
   $< ctr \mid subs : pre \overset{e(x_1,...,x_n)}{\longrightarrow} post >$ mit folgenden Komponenten:
   - *ctr : Menge von Ungleichungen $x \neq y$ mit $x, y \in X$,*
   - *$subs \subseteq Sb_\Sigma(x) \cup \overline{Sb}_\Sigma(X)$,*
   - *$pre, post \subseteq St_\Sigma(x) \cup \overline{St}_\Sigma(X)$ (endliche Zustandsmengen),*
   - *$e^n \in E$ und $x_i \in X$ für $1 \leq i \leq n$.*
2. *Eine Invariante über Σ ist ein Satz der Form $< ctr \mid \varphi >$ mit einer Ungleichungsmenge ctr wie bei den STR-Formeln und einer quantorfreien prädikatenlogischen Formel erster Ordnung φ über $St \cup Sb$.*

Die **STR-Formeln** beschreiben den Übergang von einem Zustand *pre* zu einem Systemzustand *post* und sind den ECA-Regeln sehr ähnlich. Die Mengen *pre* und *post* bestehen dabei jeweils aus verschiedenen Zustandsprädikaten.

Ein Zustandsübergang wird durch ein **Ereignis** $e(x_1,...,x_n)$ ausgelöst.
Die auslösenden Ereignisse sind im Falle von STR immer atomar und bestehen nur aus einem Namen und einer Liste von Parametern. Es werden keine Zeitaspekte berücksichtigt und jede Regel wird von genau einem Ereignis ausgelöst.

Die **Vorbedingungen** für die Durchführung einer Zustandstransition bestehen aus drei verschiedenen Arten. Die Teilbedingungen werden jeweils implizit durch Konjunktionen miteinander verknüpft.
Im ersten Teil der Vorbedingung einer Regel werden in der Menge *ctr* Ungleichungen von Werten der Variablensymbole formuliert. Als zweiten und dritten Teilausdruck enthält die Vorbedingung einer Regel immer die Mengen der Zustandsprädikate *pre* und der Abonnementsprädikate *subs*, die in einem Zustand gelten müssen, damit die Regel ausgeführt werden kann. Die Werte der Zustandsprädikate hängen vom konkreten Systemzustand ab, während die Abonnementsprädikate angeben, welche Benutzer (im Beispiel: Telefongeräte) einen Dienst abonnieren.

Als dritte Regelkomponente verwendet die STR-Formulierung von Regeln statt einer Aktionsmenge bei ECA-Regelsystemen eine Menge von **Nachbedingungen** *post*. Diese Nachbedingungen beschreiben den Zustand, der sich durch die Ausführung der Regel ergibt.
Es werden dabei implizit zwei Arten von Aktionen durchgeführt. Zuerst werden die als Nachbedingung formulierten Zustandsprädikate im neuen Zustand als wahr festgelegt.
Danach müssen alle Prädikate, die vor der Regelausführung gültig waren aber einem durch die Regel hinzugefügten Prädikat widersprechen, entfernt werden bzw. als falsch festgesetzt werden. Widersprüche zwischen Prädikaten können durch Invarianten formuliert werden, die im folgenden beschrieben werden.
Die übrigen Prädikate des alten Systemzustandes bleiben nach der Regelausführung unverändert.

In jedem Zustand des Systems soll höchstens eine Regel ausführbar sein, dadurch wird ein deterministischer Ablauf sichergestellt. Nichtdeterminismen gelten als unerwünschte Feature Interactions, die durch eine statische Analyse und die Modifikation der Regeln durch Experten beseitigt werden sollen.

Die **Invarianten** stellen eine zusätzliche Komponente der STR-Syntax dar, die in ECA-Regelsystemen normalerweise nicht enthalten ist. Ein Invariantenkonzept wurde aber auch in die Beschreibungssprache ECAL integriert, die in dieser Projektgruppe entwickelt wurde.
In der STR-Syntax werden Invarianten als Formeln ausgedrückt, die Eigenschaften des Dienstes beschreiben, die in jedem Systemzustand gelten müssen. Sie enthalten als Prädikate Abonnements und unveränderliche Zustände.

Wie bei den Vorbedingungen der Transitionsregeln bildet eine Menge von Ungleichheiten von Variablensymbolen die erste Komponente einer Invariante.
Darauf folgt als zweiter Bestandteil eine quantorfreie prädikatenlogische Formel aus Zustands- und Abonnementsprädikaten.

Widersprüche zwischen verschiedenen Invarianten und die Verletzung von Invarianten durch das Ergebnis einer Regel sind neben dem Nichtdeterminismus die weiteren Arten von Feature Interactions, die von Aiguier, Berkani und LeGall [4] untersucht wurden.

Nun kann man die Dienstspezifikation durch eine Signatur und eine Axiomenmenge (Formelmenge) formal definieren.
Die Signatur enthält alle Ereignis- und Prädikatensymbole, die für die Beschreibung des Dienstes notwendig sind. Die Axiomenmenge besteht aus den Formeln, die alle Zustandsübergänge und Invarianten des Systems beschreiben.

Definition 2.6 (Dienstspezifikation) *Eine Dienstspezifikation $\mathcal{F}$ ist ein 2-Tupel (Σ, Ax), das aus einer Signatur Σ und einer Axiomenmenge Ax besteht. Ax enthält STR-Formeln und Invarianten über Σ. $\mathcal{F}$ heißt endlich, wenn Σ eine endliche Signatur ist und Ax eine endliche Axiomenmenge ist. In der Folge wird Ax auch durch $STR \sqcup I$ bezeichnet, wobei STR alle STR-Formeln und I alle Invarianten aus Ax enthält.*

Die Syntax des logischen Systems zur Spezifikation von Softwarediensten ist nun vollständig definiert. Im nächsten Abschnitt werden Beispiele für Dienstspezifikationen in Telefonsystemen vorgestellt.

2.3.2 Beispiele für Dienstspezifikationen: Telefonsystem

Ein softwareorientiertes Telefonsystem kann durch ein Basissystem und verschiedene zusätzliche Dienste (Features) dargestellt werden.
Das Basissystem steht immer allen Nutzern des Telefonsystems zur Verfügung.
Die zusätzlichen Dienste können nur von den Nutzern verwendet werden, die den jeweiligen Dienst abonniert haben.
Die Dienstspezifikationen werden in diesem Zwischenbericht nicht vollständig definiert. Die beschriebenen Mengen bestehen meistens aus einer größeren Anzahl von Elementen als denen, die hier explizit angegeben werden.

Beispiel: Basistelefonsystem

Das Basistelefonsystem wird durch folgende Spezifikation beschrieben: $\mathcal{F}_T = (\Sigma_T, Ax_T)$ mit $\Sigma_T = (St_T, Sb_T, E_T)$ und $Ax_T = STR_T \sqcup I_T$.

Die Menge der **Zustandsprädikatsnamen** St_T enthält:

$Ruhe(x)$ („Gerät x befindet sich im Ruhezustand.“),
$warten(x)$ („x wartet auf die Wahl einer Telefonnummer.“),
$Anrufer(x,y)$ („x kommuniziert mit y als Anrufer.“),
$angerufen(x,y)$ („x kommuniziert mit y als Angerufener.“),
$läuten(x,y)$ („Gerät x läutet durch einen Anruf von y.“),
$Wahlton(x,y)$ („x hört den Wahlton des Anrufs an y.“),
$Besetztton(x)$ („x hört den Besetztton.“).

Die Menge der **Abonnementsprädikatsnamen** Sb_T ist leer, denn laut Voraussetzung abonnieren alle Telefone den Telefonbasisdienst. Dies muss nicht durch Prädikate formuliert werden.

Die Menge der **Ereignisnamen** E_T enthält:

$abheben(x)$ („Der Hörer von x wird abgehoben.“),
$auflegen(x)$ („Der Hörer von x wird aufgelegt.“),
$wählen(x,y)$ („x wählt die Telefonnummer von y.“).

Die Menge der **Zustandsübergangsregeln** STR_T enthält:

$\phi_1 :< |Ruhe(A) \overset{abheben(A)}{\longrightarrow} warten(A) >,$
$\phi_2 :< A \neq B | warten(A), Ruhe(B) \overset{wählen(A,B)}{\longrightarrow} Wahlton(A,B), läuten(B,A) >,$
$\phi_3 :< |warten(A), \overline{Ruhe(B)} \overset{wählen(A,B)}{\longrightarrow} Besetztton(A) >,$
$\phi_4 :< A \neq B | Wahlton(A,B), läuten(B,A) \overset{abheben(B)}{\longrightarrow} Anrufer(A,B), angerufen(B,A) >,$
$\phi_5 :< A \neq B | Anrufer(A,B), angerufen(B,A) \overset{auflegen(A)}{\longrightarrow} Ruhe(A), Besetztton(B) >,$
$\phi_6 :< A \neq B | Anrufer(A,B), angerufen(B,A) \overset{auflegen(B)}{\longrightarrow} Ruhe(B), Besetztton(A) >,$

$\phi_7 :< A \neq B | Wahlton(A,B), läuten(B,A) \overset{auflegen(A)}{\longrightarrow} Ruhe(A), Ruhe(B) >,$
$\phi_8 :< | Besetztton(A) \overset{auflegen(A)}{\longrightarrow} Ruhe(A) >,$
$\phi_9 :< | warten(A) \overset{auflegen(A)}{\longrightarrow} Ruhe(A) > .$

Die Regel ϕ_1 beschreibt das Abheben des Hörers von Telefongerät A im Ruhezustand. Das Gerät wartet danach auf die Wahl einer Telefonnummer.
Die STR-Formeln ϕ_2 und ϕ_3 beziehen sich auf die Wahl der Telefonnummer des Gerätes B durch den Nutzer A. Ist die Leitung von Gerät B frei, dann hört Nutzer A den Wahlton des Anrufes an B und das Telefongerät B läutet. Wenn die Leitung dagegen besetzt ist, so hört A den Besetztton.
ϕ_4 stellt den Beginn der Kommunikation durch das Abheben des Hörers des angerufenen Telefonendgerätes B dar. Nutzer A kommuniziert als Anrufer und Nutzer B als angerufener Teilnehmer.
Die Transitionsregeln ϕ_5 und ϕ_6 beschreiben die Beendigung eines Telefongespräches durch den Anrufer A bzw. den angerufenen Geprächsteilnehmer B.
Durch die Formeln ϕ_7 und ϕ_8 wird das Auflegen des Hörers nach der Wahl einer Telefonnummer durch den Anrufer A dargestellt, wenn der Hörer des angerufenen Telefongerätes bei freier Leitung noch nicht abgehoben wurde bzw. wenn die Leitung des Gerätes B besetzt war.
ϕ_9 beschreibt schließlich die Situation, in welcher der Anrufer A den Hörer vor der Wahl einer Telefonnummer wieder auflegt.

Die **Invariantenmenge** I_T enthält mehrere Invarianten, die ausdrücken, dass Zustandsprädikate, welche dieselben Variablen betreffen, sich gegenseitig ausschließen.
Die beiden folgenden Beispiele fordern, dass ein Nutzer A nicht gleichzeitig mit zwei verschiedenen Nutzern B und C sprechen kann bzw. dass sich ein Gerät nicht gleichzeitig im Ruhezustand und in Kommunikation mit einem anderen Telefongerät befinden kann.

$< B \neq C | \neg(sprechen(A,B) \wedge sprechen(A,C)) >$
$< | \neg(Ruhe(A) \wedge sprechen(A,B)) >$

Beispiel: Abschirmung eingehender Anrufe

Dieser Dienst schirmt eingehende Anrufe von Anrufern ab, die auf der schwarzen Liste des Abonnenten dieses Dienstes stehen. Der Abonnent kann also angeben, welche Anrufe er nicht annimmt.

Die Namensmenge der **Abonnementsprädikate** Sb_{Ab} enthält:

$Ab(y,x)$ („Anrufe von x an y wurden von y verboten.“).

Die **Invariantenmenge** I_{Ab} enthält:

$\psi_1 :< A \neq B | Ab(A,B) \Longrightarrow \neg Wahlton(B,A) >.$

Diese Invariante besagt, dass ein Nutzer B, der auf der schwarzen Liste des Abonnenten A steht, niemals den Wahlton eines Anrufes an A hören kann.

Die Menge der **STR-Formeln** STR_{Ab} enthält:

$\psi_2 :< | Ab(B,A) : warten(A), Ruhe(B) \overset{wählen(A,B)}{\longrightarrow} Besetztton(A) >.$

Wenn ein Gerät A auf der schwarzen Liste des Telefongerätes B steht, dann hört der Nutzer A bei einem Anruf an B wie im Fall einer besetzten Leitung den Besetztton.

Beispiel: Anrufweiterleitung

Der Dienst Anrufweiterleitung erlaubt es einem Abonnenten x, alle eingehenden Anrufe an ein festgelegtes Endgerät y weiterzuleiten, wenn das Endgerät des Abonnenten besetzt ist.

Die Menge der Namen der **Abonnementsprädikate** Sb_W enthält:

$W(x,y)$ („Wenn x nicht im Ruhezustand ist, dann werden eingehende Anrufe an y weitergeleitet.“).

Die **Invariantenmenge** I_W enthält:

$\chi_1 :< |\neg W(A,A) >,$
$\chi_2 :< B \neq C | W(A,B) \Longrightarrow \neg W(A,C) > .$

Die Invariante χ_1 sagt aus, dass ein Telefongerät A keine Anrufweiterleitung an Gerät A selbst abonnieren kann, während χ_2 fordert, dass ein Gerät A keine Anrufweiterleitung an zwei verschiedene Telefone B und C abonnieren kann.

Die Menge der **Zustandsübergangsregeln** STR_W enthält:

$\chi_3 :< B \neq C | W(B,C) : warten(A), \overline{Ruhe(B)}, Ruhe(C) \overset{wählen(A,B)}{\longrightarrow} Wahlton(A,C), läuten(C,A) >,$
$\chi_4 :< B \neq C | W(B,C) : warten(A), \overline{Ruhe(B)}, \overline{Ruhe(C)} \overset{wählen(A,B)}{\longrightarrow} Besetztton(A) > .$

In der Situation der Transitionsregeln χ_3 und χ_4 möchte ein Nutzer A einen Nutzer B anrufen, dessen Leitung besetzt ist. Da B eine Anrufweiterleitung an Gerät C abonniert hat, kann der Anruf bei freier Leitung von Gerät C weitergeleitet werden, während der Anrufer A bei besetzter Leitung von C den Besetztton hört.

Beispiel: Sperrung von ausgehenden Anrufen

Dieser Dienst erlaubt es einem Benutzer, die ausgehenden Anrufe während eines bestimmten Tagesabschnittes zu begrenzen. In diesem Zeitraum ist das Telefon für ausgehende Anrufe gesperrt. Diese Sperre kann nur durch die Eingabe einer PIN aufgehoben werden. Wenn die eingegebene PIN richtig ist, dann kann ein normaler Anruf durchgeführt werden, sonst wird der Benutzer aufgefordert, seinen Anruf zu beenden.

Die Menge der **Zustandsprädikatsnamen** St_{Sp} enthält St_T und zusätzlich spezifische Prädikate:

$Zeit(x)$ („Zeitraum, in dem der Benutzer x eine PIN eingeben muss, um ausgehende Anrufe durchführen zu können“),
$warten_{PIN}(x)$ („Das Telefongerät wartet auf die Eingabe der persönlichen PIN des Benutzers x.“) und
$ungültig(x)$ („Die eingegebene PIN ist ungültig.“).

Die Menge der **Abonnementsprädikate** Sb_{Sp} enthält:

$Sp(x)$ („x abonniert den Dienst Sp.“)

Die **Ereignismenge** E_{Sp} enthält zwei neue Ereignisse, die sich auf die Eingabe der PIN beziehen:

$richtig_{PIN}(x)$ („x gibt die erwartete korrekte PIN ein.“) und
$falsch_{PIN}(x)$ („x gibt eine falsche PIN ein.“).

Die **Invariantenmenge** I_{Sp} enthält neue Invarianten, die ausdrücken, dass sich die neuen Zustände $ungültig$ und $warten_{PIN}$ gegenseitig und mit den Zuständen aus St_T ausschließen.

Die Menge der **Transitionsregeln** STR_{Sp} enthält:

$\kappa_1 :< |Sp(A) : Zeit(A), Ruhe(A) \overset{abheben(A)}{\longrightarrow} warten_{PIN}(A), Zeit(A) >,$
$\kappa_2 :< |warten_{PIN}(A) \overset{richtig_{PIN}(A)}{\longrightarrow} warten(A) >,$
$\kappa_3 :< |warten_{PIN}(A) \overset{falsch_{PIN}(A)}{\longrightarrow} ungültig(A) >,$
$\kappa_4 :< |ungültig(A) \overset{auflegen(A)}{\longrightarrow} Ruhe(A) >,$
$\kappa_5 :< |warten_{PIN}(A) \overset{auflegen(A)}{\longrightarrow} Ruhe(A) > .$

Die STR-Formel κ_1 besagt, dass ein Telefonendgerät A während des Zeitraumes, in dem ausgehende Anrufe gesperrt werden, nach dem Abheben des Hörers auf die Eingabe einer PIN wartet.
Durch die Regeln κ_2 und κ_3 werden die Auswirkungen der Eingabe der richtigen bzw. einer falschen PIN beschrieben. Bei Eingabe der richtigen PIN wartet das Telefongerät auf die Wahl einer Telefonnummer, während es bei Eingabe einer falschen PIN in den Zustand *ungültig* übergeht.
Die Transitionsregeln κ_4 und κ_5 zeigen, dass ein Telefongerät durch das Auflegen des Hörers nach Eingabe einer ungültigen PIN bzw. während des Wartens auf die Eingabe einer PIN in den Ruhezustand wechselt.

2.3.3 Beispiele für Feature Interactions im Telefonsystem

Bei der Integration der im vorigen Abschnitt beschriebenen Dienste in das Basistelefonsystem werden durch die Integrationsmethode von Aiguier, Berkani und Le Gall [4] mehrere Feature Interactions erkannt, von denen drei hier beschrieben werden.
Die Dienste werden in folgender Reihenfolge in das Telefonbasissystem integriert:
$(((T +_{c_1} Ab) +_{c_2} W) +_{c_3} Sp)$.

- Integration des Dienstes **Abschirmung eingehender Anrufe** $(T +_{c_1} Ab)$:

  Ein Nichtdeterminismus zwischen ϕ_2 und ψ_2 wurde entdeckt:

  $\phi_2 :< A \neq B | warten(A), Ruhe(B) \overset{wählen(A,B)}{\longrightarrow} Wahlton(A,B), läuten(B,A) >,$
  $\psi_2 :< |Ab(B,A) : warten(A), Ruhe(B) \overset{wählen(A,B)}{\longrightarrow} Besetztton(A) >.$

  In der Situation $A \neq B | Ab(B,A) : warten(A), ruhe(B)$ sind die Vorbedingungen der beiden STR-Formeln erfüllt. Wenn das Ereigniss $wählen(A,B)$ eintritt, sind deshalb beide Regeln anwendbar. Die Regel ϕ_2 aus dem Basistelefonsystem sagt aus, dass Nutzer A nach der Wahl der Telefonnummer von Gerät B den Wahlton hört und das Telefongerät B läutet. Wenn aber A auf der schwarzen Liste von B steht, dann hört Nutzer A gemäß der STR-Formel ψ_2 den Besetztton.

  Der Konflikt kann durch eine Modifikation von ϕ_2 gelöst werden. Der beschriebene Zustandsübergang soll nur noch ausführbar sein, wenn der Anrufer A nicht auf der schwarzen Liste von B steht. Der Dienst zur Abschirmung eingehender Anrufe erhält dadurch Priorität gegenüber dem Basissystem.

- Zusätzliche Integration des Dienstes **Anrufweiterleitung** $((T +_{c_1} Ab) +_{c_2} W)$:

  Auch zwischen ϕ_3 und χ_3 wird ein Nichtdeterminismus erkannt:

  $\phi_3 :< |warten(A), \overline{Ruhe(B)} \overset{wählen(A,B)}{\longrightarrow} Besetztton(A) >,$
  $\chi_3 :< B \neq C | W(B,C) : warten(A), \overline{Ruhe(B)}, Ruhe(C) \overset{wählen(A,B)}{\longrightarrow} Wahlton(A,C), läuten(C,A) >.$

  Bei Erfülltheit der Vorbedingung von χ_3 ist insbesondere auch die Vorbedingung der Regel ϕ_3 erfüllt. Durch das Eintreten des Ereignisses $wählen(A,B)$ können deshalb zwei verschiedene Transitionen ausgelöst werden. Bei Anwendung der Regel ϕ_3 aus dem Basistelefonsystem hört

der Anrufer A den Besetztton, während durch den Zustandsübergang χ_3 eine Anrufweiterleitung an das Gerät C bewirkt wird.

Beim Abonnement der Anrufweiterleitung ist aber nur die durch χ_3 beschriebene Transition erwünscht. Zur Auflösung des Konfliktes wird darum ϕ_3 modifiziert, so dass diese Regel nur noch ausführbar ist, wenn keine Anrufweiterleitung abonniert wurde.

Nach dieser Modifikation stellt man fest, dass χ_3 die Invariante ψ_1 verletzt:

$\chi_3 :< B \neq C | W(B,C) : warten(A), \overline{Ruhe(B)}, Ruhe(C) \overset{wählen(A,B)}{\longrightarrow} Wahlton(A,C), läuten(C,A) >,$
$\psi_1 :< C \neq A | Ab(C,A) \Longrightarrow \neg Wahlton(A,C) >.$

Bemerkung:
Die Variablennamen in der Invariante ψ_1 wurden hier gegenüber der Definition in Abschnitt 2.3.2 geändert, damit die Verletzung der Invariante einfacher zu erkennen ist.

Die STR-Formel χ_3 beschreibt eine Situation, in der Nutzer A die Telefonnummer des Gerätes B wählt, dessen Leitung aber besetzt ist. Da B eine Anrufweiterleitung an C abonniert hat, hört A den Wahlton des Anrufes an C und das Telefongerät C läutet. Wenn aber gleichzeitig A auf der schwarzen Liste von C steht, so verletzt die Nachbedingung $wahlton(A,C)$ die Invariante ψ_1, die besagt, dass A niemals den Wahlton eines Anrufes an C hören kann.

Das Abonnement der Anrufweiterleitung soll nur wirksam sein, wenn der Anrufer nicht gleichzeitig auf der schwarzen Liste des Gerätes steht, an das der Anruf weitergeleitet werden soll. Dies kann durch Hinzufügung von $\neg Ab(C,A)$ zur Abonnementsmenge von χ_3 erreicht werden. Dadurch wird die Invariante nicht mehr verletzt, weil die Prämisse $Ab(C,A)$ nicht erfüllt ist.

Für den Fall, dass $Ab(C,A)$ gilt, muss man nun eine weitere STR-Formel zum System hinzufügen:

$$\chi_3' :< B \neq C | W(B,C), Ab(C,A) : warten(A), \overline{Ruhe(B)}, Ruhe(C) \overset{wählen(A,B)}{\longrightarrow} Besetztton(A), Ruhe(C) > .$$

Der Anrufer A hört den Besetztton, weil die Leitung von B besetzt ist und A auf der schwarzen Liste von C steht. Das Gerät C bleibt im Ruhezustand. Auch in diesem Fall wird die Invariante nicht verletzt.

Durch die getroffenen Expertenentscheidungen hat Ab Priorität gegenüber dem Dienst W und W hat Priorität gegenüber dem Basissystem T.

- Zusätzliche Integration des Dienstes **Sperrung ausgehender Anrufe**
  $(((T +_{c_1} Ab) +_{c_2} W) +_{c_3} Sp)$:

  Bei der Integration des Dienstes zur Sperrung ausgehender Anrufe tritt ein weiterer Nichtdeterminismus zwischen den Transitionsregeln ϕ_1 und κ_1 auf:

  $\phi_1 :< | Ruhe(A) \overset{abheben(A)}{\longrightarrow} warten(A) >,$
  $\kappa_1 :< | Sp(A) : Zeit(A), Ruhe(A) \overset{abheben(A)}{\longrightarrow} warten_{PIN}(A), Zeit(A) >.$

  In der Situation, in der die Vorbedingungen von κ_1 erfüllt sind, ist auch die Vorbedingung der Regel ϕ_1 erfüllt. Durch das Ereignis $abheben(A)$ können beide Transitionen ausgelöst werden.

  Das gewünschte Systemverhalten beim Vorliegen einer Sperrung ausgehender Anrufe wird aber nur durch die STR-Formel κ_1 beschrieben, weil in diesem Fall Anrufe nur nach Eingabe einer PIN möglich sein sollen. Dieser Konflikt wird daher durch die Modifikation von ϕ_1 aufgelöst. Dabei erhält der Dienst Sp Priorität gegenüber dem Telefonbasissystem T und den beiden Diensten Ab und W.

2.4 Lernen

Falk Howar

Um Lernen als algorithmische Methode umzusetzen, ist es nützlich den umgangssprachlichen Begriff des Lernens etwas zu formalisieren. Lernen soll ein Spiel zwischen zwei Spielern sein. Die beiden Spieler sind ungleich stark, der eine ist fast allwissend und verkörpert den Lehrer. Der Lehrer wird manchmal auch als Orakel bezeichnet und realisiert. Der andere Spieler ist der Schüler. Er beginnt das Spiel völlig wissenlos. Ziel des Spiels ist es für den Schüler, sein Wissen zu vergrößern; dazu stellt er dem Lehrer Fragen. In bestimmten Situationen, wenn sich für den Schüler durch das von ihm erlernte Wissen eine vollständige und konsistente Beschreibung der „Welt“ ergibt, stellt er dem Lehrer eine Frage, die offenlegt, ob er tatsächlich alles weis. Der Schüler versucht immer wieder sein „Weltbild“ zu vervollständigen und lernt so lange, bis sein dieses konsistent ist. Im Wesentlichen muss also beschrieben werden, wie der Schüler sein Wissen verwaltet, wie die verschiedenen Fragen an den Lehrer aussehen sollen und wie der Lehrer zu seinem Wissen kommt.

2.4.1 Angluins Algorithmus

Der von Dana Angluin entwicklete Algorithmus L^* [5] modelliert Lernen als Kommunikationsspiel. Es gibt zwei Spieler, den Lehrer und den Schüler. Der Schüler stellt im Laufe des Spiels Fragen an den Lehrer und erschließt sich so nach und nach das zu lernende Gebiet.

Er benutzt dazu zwei verschiedene Arten von Fragen: erstens stellt er Fragen, die sein Wissen vermehren und zweitens stellt er, jeweils wenn er glaubt, alles zu wissen, eine Frage, die man als einen freiwilligen Wissenstest beschreiben kann; er präsentiert dem Lehrer sein ganzes bisheriges Wissen.

Man kann über den Lehrer (bezüglich seiner Kompetenz) Verschiedenes annehmen. Hier sei der Einfachheit halber zunächst angenommen, dass der Lehrer beide Arten von Fragen vollständig und richtig beantworten kann. Der Lehrer muss also ein absoluter Experte auf seinem Gebiet sein, um die zweite Art von Fragen nach dem Wissenstand des Schülers im Vergleich zu dem zu erlernenden Stoff beurteilen zu können.

Der Algorithmus arbeitet auf regulären sprachen. Im Folgenden wird also davon ausgegangen, dass der Schüler eine reguläre Sprache L lernt, z.B $L_1 = \{$ Alle Worte mit gerade vielen a's und b's über dem Alphabet $\{a, b\}\}$. Für diese Sprache lässt sich sowohl ein deterministischer endlicher Autmonat (DFA), als auch ein regulärer Ausdruck angeben, welche sich auch ineinander überführen lassen. Eine reguläre Sprache zu lernen, ist also eine äquivalente Darstellung des Erlernens eines unbekannten Automaten oder regulären Ausdrucks.

Dabei bedient sich nun der Schüler folgender zwei Arten von Fragen: erstens erfragt er systematisch, ob bestimmte Worte in der Sprache sind oder nicht (Membership-Queries oder MEMBER(x)), zweitens konstruiert er aus seinem jeweiligen Wissensstand, wenn er ausreichend viel weis, einen DFA, welchen er vom Lehrer auf Äquivalenz mit dem zu der zu lernenden Sprache gehörenden Automaten prüfen lässt (conjecture oder EQUIV(M)).

Während des gesamten Spiels führt der Schüler Buch über sein bisheriges Wissen in Form einer Tabelle (Observation Table). Diese Tabelle kann man sich als ein zweidimensionales Array vorstellen, in welches der Schüler alle bisher gefragten Worte notiert, die in der Sprache enthaltenen genauso wie die nicht enthaltenen.

Die Zeilen teilen sich in zwei Bereiche S und $S \cdot A$. S und $S \cdot A$ sind Mengen von Worten, welche in den Bereichen Zeilenlabel darstellen. S ist eine nicht leere Präfix-vollständige Menge (prefix-closed set), d.h. alle Präfixe jedes enthaltenen Wortes sind ebenfalls enthalten. Bei Beginn des Spiels enthält S nur das leere Wort ε. In $S \cdot A$ sind alle Worte enthalten, die durch Konkatenation von Worten aus S mit Buchstaben aus dem Alphabet A der zu lernenden Sprache entstehen.

Die Spalten sind gelabelt durch Elemente aus E, einer nicht leeren Suffix-vollständigen Menge von Worten. Bei Beginn des Spiels gibt es nur eine einzige Spalte, welche mit dem leeren Wort ε gelabelt ist.

Die Einträge der Tabelle enthalten die Information, ob das an der jeweiligen Stelle repräsentierte Wort in der zu lernenden Sprache enthalten ist. Man stelle sich das in Form einer Funktion T vor, welche jedes Wort aus L auf 1 und jedes Wort nicht aus L auf 0 abbildet. An der Stelle [s/e] bzw. [s·a/e] steht nun das für das Wort s·a bzw. s·a·e der entsprechende Wert der Funktion T, aus $\{0,1\}$. Die „Observation Table“ wird kurz auch mit (S,E,T) beschrieben.

Der Schüler versucht stets, die Tabelle auf einen konsistenten und vollständigen Stand zu bringen. Dabei soll vollständig (closed) bedeuten, dass für jede Zeile aus S·A eine Zeile in S existiert, so dass $row(s \cdot a) = row(s)$, wobei die Funktion $row()$ nur die Werte aus einer Zeile liefert. Konsistent (consistent) soll die Tabelle immer dann sein, wenn folgende Bedingung gilt: $row(s_1) = row(s_2) \Rightarrow row(s_1 \cdot a) = row(s_2 \cdot a)$. Die beiden Bedingungen geben an, wann der Schüler versuchen kann, eine Äquivalenz-Anfrage zu stellen, nämlich genau dann, wenn beide Bedigungen gelten.

Wenn die Tabelle konsistent und vollständig ist, kann der Schüler daraus folgenden DFA konstruieren:

Definition 2.7 $M(S,E,T) = (Q, q_0, F, \delta, \Sigma)$

$$\begin{aligned} Q &= \{row(s); s \in S\} \\ q_0 &= row(\varepsilon) \\ F &= \{row(s); s \in S \text{ und } T(s) = 1\} \\ \delta(row(s), a) &= row(s \cdot a) \\ \Sigma &= A \end{aligned}$$

Der Automat ist wohldefiniert:

- q_0 existiert, da ε in S enthalten ist.
- F ist wohldefiniert, da ε in E enthalten sein muss, also gilt wenn $row(s_1) = row(s_2)$, dann $T(s_1) = T(s_1 \cdot \varepsilon) = T(s_2) = T(s_2 \cdot \varepsilon)$
- Dass δ ist wohldefiniert, folgt aus der Vollständigkeit der Tabelle: Es gibt zu jedem $s \cdot a$ aus $S \cdot A$ eine Reihe in S, so dass $row(s \cdot a) = row(s)$.

Wichtig für den Lernprozess ist folgende Erkenntnis:

Theorem 2.1 *Wenn (S,E,T) eine vollständige und konsistente Tabelle ist, dann verhält sich der DFA M(S,E,T) konsistent zu der der Funktion T. Jeder andere DFA, der sich T gegenüber konsistent verhält, aber nicht zu M(S,E,T) äquivalent ist, muss mehr Zustände haben als M(S,E,T).*

Die wichtige Erkenntnis aus dem Theorem ist folgende: Der Schüler stellt im Laufe des Lernprozesses verschiedene Automaten als Äquivalenzanfragen an den Lehrer, die gegen den zu lernenden Automaten konvergieren. D. h. die Automaten des Schülers wachsen monoton bis er einen Automaten konstruieren kann, der (maximal) gleich viele Zusände wie der zu lernende hat und diese beiden Automaten müssen dann äuivalent sein.

Das „Wachsen“ funkioniert folgendermaßen: Der Lehrer beantwortet Äuqivalenzanfragen ggf. mit einem Gegenbeispiel, welches der Schüler in seine Tabelle einarbeitet. Anschließend versucht er, die Tabelle vollständig und konsistent zu machen. Bei Beginn des Lernvorgangs besteht die Tabelle nur aus $S =$

$E = \varepsilon$. Der Schüler fragt dann für alle Buchstaben des Alphabets, ob diese Worte der zu lernenden Sprache darstellen.

Wenn zu einem bestimmten Zeitpunkt die Tabelle nicht konsistent ist, bedeutet das, dass in der Tabelle vier Zeilen zu finden sind mit $row(s_1) = row(s_2)$ aber $row(s_1 \cdot a \cdot e) \neq row(s_2 \cdot a \cdot e)$. Dies kann behoben werden, indem a·e in E aufgenommen wird. Die Tabelle wird entsprechend erweitert, die Werte für die neuen Zellen müssen vom Lehrer erfragt werden. Weil e bereits in E ist, bleibt die Suffix-vollständigkeit von E bei hinzufügen von a·e erhalten.

Wenn zu einem bestimmten Zeitpunkt die Tabelle nicht vollständig ist, bedeutet das, dass es eine Zeile $row(s \cdot a)$ gibt, die allen Zeilen in S ungleich ist. Vollständigkeit wird hergestellt, indem die betreffende Zeile nach S verschoben wird, d.h. S wird um s·a erweitert. Die Präfix-Vollständigkeit bleibt erhalten. Es entstehen durch die Operation neue Zeilen in S·A, die Werte für die Zellen werden vom Lehrer durch Membershipqueries erfragt.

Im Prinzip verwaltet der Schüler in S die Zustandsmenge des zu lernenden Automaten und in $S \cdot A$ die Menge der Transitionen. Wenn die Tabelle nicht vollständig ist, es in S·A also eine Zeile gibt, die in S nicht vorkommt, bedeutet das, dass es eine Transition gibt, die in einen nicht vorhandenen Zustand führt. Wenn die Tabelle nicht konsistent ist, bedeutet das, dass man beim Lesen des selben Buchstabens aus dem selben Zustand in zwei verschiedene Zustände gelangen kann. Da der konstruierte Automat allerdings deterministisch ist, müssen es statt einem zwei Startzustände sein. Deswegen muss die Menge der Unterscheidungsmerkmale E erweitert werden.

Es ergibt sich folgender Algorithmus für den Schüler:

Algorithmus 2.1 *Lerner*

```
REPEAT
  WHILE NOT (S,E,T) vollständig und konsistent
    IF NOT (S,E,T) konsistent
      SELECT row(s1) = row(s2) FROM S WHERE row(s1.a.e) != row(s2.a.e)
      // a aus dem Alphabet, s1,s2 aus S, e aus E
      ADD a.e TO E
      FORALL (S,E,T) IF T(s,e) == NULL THEN MEMBER(s.e) // s aus S und S.A, e aus E
    END IF
    IF NOT (S,E,T) vollständig
      SELECT row(s1.a) WHERE NOT row(s1.a) IN (row(S))
      // a aus dem Alphabet, s1,s2 aus S
      ADD s1.a TO S
      FORALL (S,E,T) IF T(s,e) == NULL THEN MEMBER(s.e) // s aus S und S.A, e aus E
    END IF
  END WHILE
  FAILURE = EQUIV((S,E,T))
  IF NOT FAILURE
    RETURN (S,E,T)
  ELSE
    ADD SPLIT_TO_PREFIXES(FAILURE) TO S
    FORALL (S,E,T) IF T(s,e) == NULL THEN MEMBER(s.e) // s aus S und S.A, e aus E
  END IF
UNTIL FALSE
```

Dana Angluin stellt weitere Überlegungen zur Korrektheit und Laufzeit von Alg 1 an. Hier seien nur die Ergebnisse erwähnt. Der Algorithmus ist korrekt und muss maximal n Äquivalenzanfragen

stellen, wenn der minimale DFA für die zu lernende Sprache n Zustände hat. Die Laufzeit und der Platzverbrauch hängen zusätzlich zu n von der Länge der Gegenbeispiele des Lehrers m ab. Die Größe der Tabelle kann durch $O(m^2n^2 + mn^3)$ nach oben abgeschätzt werden. Eine Untersuchung der vom Lerner durchzuführenden Operationen ergibt für die Laufzeit insgesamt ein Polynom in n und m. Kann man den Lehrer so umsetzen, dass er stets kürzeste Gegenbeispiele, d.h. welche mit maximaler Länge n angibt, ist die Rechenzeit durch ein Polynom in n bestimmbar.

2.4.2 Conformance Testing

Der vorgestellte Algorithmus L^* macht eine Voraussetzung, welche ihn in der vorgestellten Version zum Lernen von Ausdrücken oder Automaten, die nicht explizit beschrieben vorliegen, unbenutzbar macht. Es wird nämlich vorausgesetzt, dass der Lehrer das zu lernende System kennt. Ansonsten könnte er Äquivalenzanfragen nicht beantworten. In praktisch interessanten bzw. relevanten Szenarien ist diese Voraussetzung jedoch nicht haltbar. Man muss also anstelle der Äquivalenztests eine Methode benutzen, welche die Äquivalenz der beiden Automaten, des gelernten und des zu lernenden, ohne vollständige Kenntnis des zu lernenden Automaten testen kann. Solche Methoden findet man im modellbasierten Testen.

Es gibt verschiedene Ansätze im modellbasierten Testen, die sich dazu eignen, Äquivalenztests zu simulieren, z.B. „State-Cover“, „Switch-Cover“, „Branch-Cover“ und „Boundary-Interior-Cover“. Dabei bedeutet „Cover“ immer ein Testset, das auf bestimmte Weise den Programmgraphen abdeckt. Ein Testset ist eine Menge von Testfällen, theoretisch also von Worten, deren Zugehörigkeit zu der zu lernenden Sprache überprüft werden soll. Diese Ansätze sollen hier nur kurz vorgestellt werden.

„State-Cover“ Ein State-Cover ist ein Testset, das alle Zustände des vorliegenden Automaten bzw. des Programmgraphen überdeckt.

„Branch-Cover“ Ein Problem beim Finden und Überprüfen von Programm-Traces bei Programmen mit Schleifen ist, dass es unter Umständen unendlich viele mögliche Traces gibt. Ein Branch-Cover enthält zumindest für jeden Teil des Programms einen Trace der diesen überdeckt.

„Switch-Cover“ Ein Switch-Cover erweitert die Idee von Branch-Covern. Ein Switch-Cover enthält Traces, so dass jede mögliche Kombination von Ein- und Ausgangskanten in allen Knoten des Programmgraphen im Testset enthalten ist.

„Boundary-Interior-Cover“ Boundary-Interior-Cover bestehen aus Traces, die jede Schleife durchlaufen, sie aber gar nicht (Boundary) oder nur einmal (Interior) iterieren.

„Die W-Methode“ Die W-Methode [7], die Tsun S. Chow entwickelt, soll hier exemplarisch etwas genauer eingegangen werden.

Das Vorgehen bei der W-Methode lässt sich in drei Schritte gliedern. Dabei ist der erste Schritt nicht automatisierbar, sondern muss von einem Menschen durchgeführt werden.

Die Worte, die als Testfälle dienen sollen, werden durch Konkatenation von Worten einer Klasse P und Worten einer Klasse Z generiert. Worte aus P findet man auf die folgende Weise anhand des vorliegenden (bereits erlernten) Automaten: P soll eine Menge von Worten sein, so dass für je zwei Zusände q_i und q_j mit $\delta(q_i, a) = q_j$ ein Wort p mit $\delta(q_0, p) = q_i$ und das Wort pa in P sind. Solche Worte können gefunden werden, indem man den vorhandenen Automaten in einen Baum „entrollt“, so dass in dem Baum jeder Zustand des Automaten genau einmal als innerer Knoten vorkommt. Kommt ein

Knoten beim Entrollen zum zweiten Mal vor, d.h. der entsprechende Zustand des Automaten wurde auf dem Weg von der Wurzel bis zum aktuellen Knoten bereits durchlaufen, braucht man an dieser Stelle nicht weiterzusuchen; der Knoten wird ein Blatt. Die Menge P ist nicht eindrutig formuliert. Der Präfix-Abschluss über der Menge der Worte, welche durch die Äste bzw. durch deren konkatenierte Kantenlabel (jeweils von der Wurzel aus) enstehen. Abbildung 2.5 zeigt einen solchen Baum für die Sprache $L_1 = \{$ Alle Worte mit gerade vielen a's und b's über dem Alphabet $\{a,b\}\}$. Im dargestellten Fall wäre P = $\{a,b,aa,ab,ba,bb\}$

Q/Σ	a	b
10	01	00
01	10	00
00	00	10

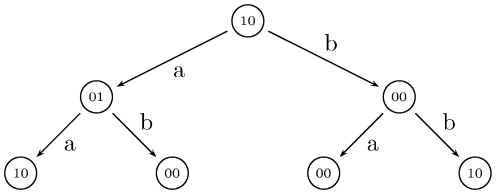

Abbildung 2.5: P-Baum (rechts), Charakteristische Auswahl (links)

Um Z zu beschreiben, definieren wir zunächst eine „charakteristische Auswahl" (characterization set). Dabei soll es sich um eine Auswahl von Buchstaben des Eingabe-Alphabets handeln, an Hand welcher jeder Zustand eindeutig identifizierbar ist (siehe Abbildung 2.4). Der triviale Fall ist das ganze Eingabe-Alphabet, allerdings wird man bei Alphabeten, die mehr als zwei Zeichen enthalten, versuchen, eine kleinere Auswahl zu bekommen, da sich die Größe der Auswahl auf die Anzahl der später zu testenden Worte auswirkt. Wie man eine solche Auswahl aufbauen kann, wird hier nicht beschrieben.

Sei nun m die geschätze maximale Anzahl von Zuständen des zu erlernenden Systems, n die Anzahl der Zustände des vorliegenden Automaten, X das Eingabe-Alphabet und W eine charakteristische Auswahl, dann ist Z definiert als $\{W \cup X \cdot W \cup \cdots \cup X^{m-n} \cdot W\}$. Dabei sei X^0 das leere Wort und $X^{i+1} = X \cdot X^i$.

Schätzen wir in unserem Beispiel m durch 4 ab und wählen als charakteristische Auswahl $\{a,b\}$, so ergibt sich $Z = \{a,b,aa,ab,ba,bb\}$. und

$$P \cdot Z = \{a,aa,ab,aaa,aab,aba,abb,b,ba,bb,baa,bab,bba,bbb,aaaa,aaab,aaba,$$
$$aabb,abaa,abab,abba,abbb,baaa,baab,baba,babb,bbaa,bbab,bbba,bbbb\}$$

Im allgemeinen Fall kann man bezüglich der Zeitkomplexität folgende Aussage machen: Die aufsummierte Länge aller Testeingaben ist kleiner als $n^2 \cdot m \cdot k^{m-n+1}$, mit m und n wie in den voherigen Absätzen und k = Anzahl der Zeichen im Eingabe-Alphabet. Das kann man durch $O(n^3)$ abschätzen, wenn man annimmt, das m-n maximal n ist und k wesentlich kleiner als n. Zumindest letzteres dürfte bei realen Systemen gegeben sein. Wenn man den Conformance Test in den L^* Algorithmus integriert, kann man die Menge $P \cdot Z$ natürlich noch einmal deutlich verkleinern, weil man die nötigen Membershipqueries für einen Teil der Worte aus $P \cdot Z$ bereits gestellt hat.

Es sei darauf hingewiesen, dass diese Verfahren an jeweils verschiedenen Problemen scheitern, so dass bei diesen Strategien eine Garantie für die Korrektheit der Aussage bezüglich der Äquivalenz bzw. Conformance nicht gegeben werden kann. Die Tabelle in Abbildung 2.6 zeigt zusammenfassend die Fähigkeiten der einzelnen Strategien.

Theoretisch gibt keines der Verfahren eine Garantie, dass ein Test korrekt beendet wird. Alle Verfahren haben einen einseitigen Fehler, die Verfahren können eine Eingabe fälschlicherweise als äquivalent akzeptieren. Allerdings ist es gerade zu Beginn des Lernprozesses in vielen Fällen nicht so schwer

Methode	erkennt Probleme
„State-Cover“	- Fehler in der Ausgabefunktion
„Branch-Cover“	- Fehler in der Ausgabefunktion
„Switch-Cover“	- Fehler in der Ausgabefunktion - manche Fehler in δ
„Boundary-Interior-Cover“	- Fehler in der Ausgabefunktion - manche Fehler in δ
„W-Methode“	- Fehler in der Ausgabefunktion - Fehler in δ - zuviele / fehlende Zustände (bis zu Anz. m-n)

Abbildung 2.6: Conformance Test Methoden

Gegenbeispiele zu finden und Nichtäquivalenzen aufzudecken. Generell kann man folgende Aussage machen: Je mehr Sicherheit ein Verfahren bietet, desto größer ist die benötigte Rechenzeit.

Es ist also für den praktischen Einsatz der Verfahren evtl. angemessen, zu Beginn des Lernprozesses ein Varfahren für die Äquivalenztests zu benutzen, das eine relativ hohe Fehlerquote hat und sobald dieses die Äquivalenz bestätigt, auf ein anderes Verfahren umzusteigen. Im Übrigen wird das Ziel des praktischen Einsatzes meist sein, möglichst schnell die meisten Fehler aufzudecken und nicht unbedingt, irgendwann alle Fehler aufzudecken.

2.4.3 LearnLib

Die LearnLib ist eine am Lehrstuhl 5 des Fachbereichs Informatik der Universität Dortmund in JAVA und C++ entwickelte Bibliothek, die im Wesentlichen den vorgestellten Algorithmus L^* umsetzt. Während L^* allerdings nur mit endlichen Automaten arbeitet, erweitert die LearnLib das Konzept und bietet als theoretische Grundlage auch Mealy-Automaten an. Mealy-Automaten produzieren im Unterschied zu DFAs bei jedem Zustandsübergang eine Ausgabe aus einem Ausgabe-Alphabet gemäß einer Ausgabefunktion. Eine Modellierung mit Hilfe von Mealy-Automaten reflektiert eine Reihe von realen Systemen mindestens besser als DFAs. Mealy-Automaten erlauben auch das Lernen von Systemen, die durch DFAs gar nicht sinnvoll dargestellt werden können, z.B. von Internetseiten.

Die LearnLib wurde entwickelt, um verschiedene Lernalgorithmen miteinander vergleichen zu können und um so Stärken und Schwächen einzelner Algorithmen bezüglich bestimmter Szenarien zu bestimmen. In der Projektgruppe „LiVe“ wird die Bibliothek zwar aller Voraussicht nach nicht primär zu diesem Zweck eingestezt, sie bietet jedoch ein lauffähiges und erprobtes Framework für das Lernen von Systemen, die sich durch reguläre Sprachen darstellen lassen.

Aufbau

Die Bibliothek verfügt über CORBA-Schnittstellen und kann so flexibel eingebunden werden, indem die Bibliothek als eigenständiger Server läuft. Als Frontend kann z.B. das ebenfalls am Lehrstuhl 5 entwickelte JavaABC dienen. Zusätzlich zum Lernalgorithmus stellt die Bibliothek eine Reihe von

Filtern bereit, die es z.B. ermöglichen die Anzahl der Queries an das zu lernende System zu reduzieren, indem im Vorhinein bekannte Struktureigenschaften des zu lernenden Systems ausgenutzt werden. Die Bibliothek kann automatisch oder interaktiv eingebunden werden, d.h. entweder die Bibliothek befindet sich zwischen der Anwendung und dem zu lernenden System, oder es werden nur die nötigen Anfragen verwaltet und bereitgestellt, die Kommunikation mit dem Zielsystem bleibt der Anwendung überlassen.

Die LearnLib ist modular aufgebaut, so dass sie einfach um zusätzliche Algorithmen und Filter erweitert werden kann. Abbildung 2.7 zeigt die Struktur der Bibliothek. Ein Modul enthält die implementierten Lernalgorithmen, ein weiteres alle Filter zur Reduzierung der Anfragen. Die dritte große Komponente bildet das Modul mit den Conformance Tests.

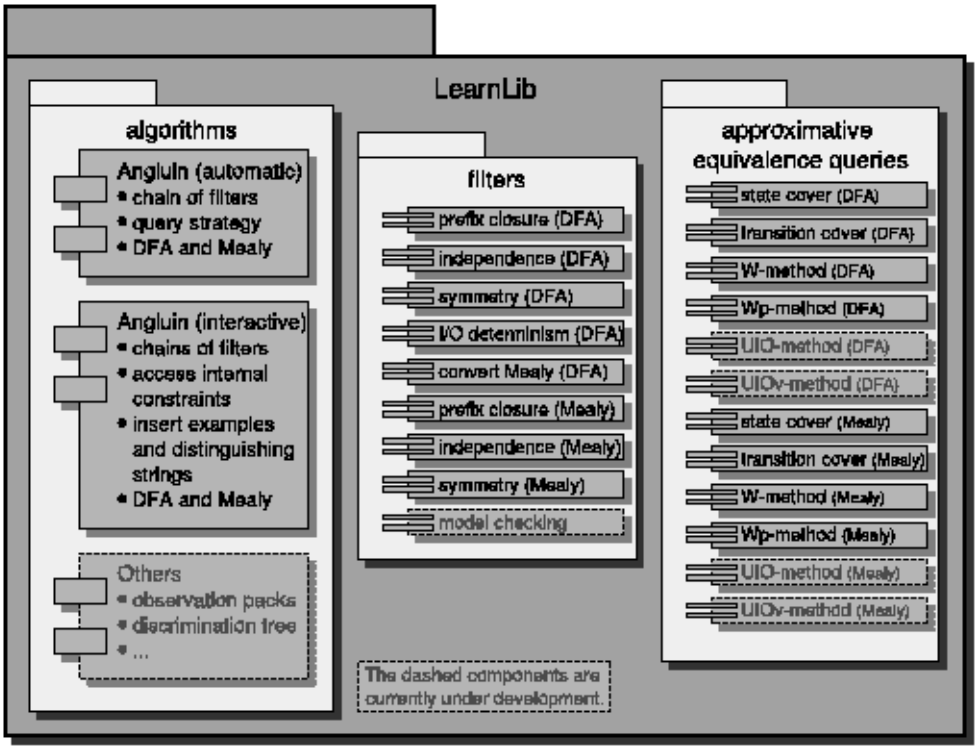

Abbildung 2.7: LearnLib - Aufbau [20]

Schnittstellen

Die LearnLib verfügt über zwei Schnittstellen, eine zur Kommunikation mit der Anwendung, welche die LearnLib einbindet und eine zur Kommunikation mit dem „System under Test (SUT)“. Dabei bleibt es dem Benutzer überlassen, die zweite Schnittstelle zu benutzen oder die Kommunikation mit dem Zielsystem selbst zu übernehmen. Um die SUT-Schnittstelle zu benutzen, betreibt man die LearnLib im automatischen Modus, wenn man die Kommunikation mit dem zu testenden System selbst übernimmt, betreibt man die LearnLib im interaktiven Modus.

Filter

Tatsächlich werden im Lernprozess nur Membershipqueries an ein zu lernendes System gestellt. Die Kosten bzw. die Rechenzeit des Lernprozesses können also an der Anzahl dieser Anfragen gemessen werden. So ist z.B. der Vergleich verschiedener Lernalgorithmen bezüglich eines bestimmten Szenarios

über die Anzahl der jeweils gebrauchten Anfragen möglich. Um das Lernen in praxisrelevanten Anwendungen benutzen zu können, ist es wünschenswert und nötig, die Anzahl der Membershipqueries so weit wie möglich zu reduzieren. Zu diesem Zweck stellt die LearnLib eine Reihe von Filtern bereit, die auf verschiedene Weise überflüssige Anfragen aussortieren. Im Folgenden werden einige der Filter kurz vorgestellt.

Es kann vorkommen, dass der Lernalgorithmus L^*, da man die Äquivalenzanfragen durch Conformance Tests auflöst, Anfragen doppelt erzeugt. Um diese Anfragen nicht tatsächlich doppelt an das Zielsystem zu stellen, implementiert die LearnLib einen Cache, in welchem gestellte Anfragen und die jeweilige Anwort des Testsystems gespeichert werden. Der Cache ist als Hash implementiert.

Ein enormes Einsparpotential bei den Membershipqueries lässt sich aus der Annahme entwickeln, dass das zu lernende System präfix-abgeschlossen ist. D. h. dass, wenn ein Wort in der akzeptierten Sprache enthalten ist, auch alle Präfixe in der Sprache enthalten sind. Weis man also, dass ein Wort w nicht in der Sprache ist, so kann auch kein anderes Wort in der Sprache sein, welches w als Präfix hat. Gerade wenn man also kurze Gegenbeispiele findet kann man mit dieser Annahme den Suchraum stark verkleinern.

Dass die Annahme in der Praxis relevant ist, zeigt folgende Überlegung: Reale Systeme reagieren normalerweise nur so lange sie sich in einem „gesunden“ Zusatand befinden auf Eingaben. Erreicht man einen Fehlerzustand, kann dieser zwar evtl. wieder verlassen werden, trotzdem müssen alle Worte, die das lernende System zuerst in diesen Zustand bringen nicht weiter betrachtet werden.

Der Unabhängigkeitsfilter basiert auf möglichem Expertenwissen über das zu lernende System. Ist bekannt, dass verschiedene Eingaben voneinander unabhängig sind, d.h. dass es nie eine Rolle spielt, in welcher Reihenfolge sie erfolgen bzw. dass das lernende System bei jeder Eingabe, in der die aufeinander folgenden unabhängigen Aktionen beliebig vertauscht sind, in den selben Zustand gelangen wird. Kann man also eine oder mehrere solcher unabhängigen Eingabemengen angeben, kann man entsprechende Teile des Suchraums auslassen, da eine Unteruschung dieser keine neuen Erkenntnisse bringen würde.

Der Symmetriefilter nutzt bekannte Redundanzen in dem zu lernenden System aus, indem er diese symbolisch behandelt. Weis man z.B. bei einem Telefonsystem, dass es unerheblich ist, welche zwei Teilnhemer genau miteinader interagieren, kann man die Teilnehmer symbolisch behandeln und etwa annehmen, dass nie mehr als drei symbolische Teilnehmer interagieren. Es ist unmittelbar einzusehen, dass es einen gewaltigen Unterschied macht, ob man ein Telefonsystem mit drei Akteuren oder eines mit mehreren hundert oder gar tausend Teilnehmern beschreiben möchte.

Abschließend lässt sich sagen, dass Lernen zur Zeit noch nicht unbedingt einen praxisrelevanten Beitrag zu der Lösung der beschriebenen Probleme darstellt. Jedoch wird sich dies mit andauerndernder Verbesserung der verwendeten Lernalgorithmen sowie der verwendeten Filtertechniken sicher in absehbarer Zeit ändern. An dieser Entwicklung wird das Projekt „LiVe“ vielleicht einen kleinen Anteil haben, indem wir zeigen können, dass das Lernen im Bereich der Modellierung bzw. des Testens und Verifizierens von Business-Prozess-Beschreibungen und anderen ECA-Regel-Systemen erfolgreich (im Sinne eines Proof of Concept) als Methode anwendbar ist.

2.5 Compilerentwicklung

Jochen Gerlach

Für ECA-Regelsysteme gibt es bislang noch keine brauchbare Sprache zum Automatenlernen. Eine der wichtigsten Aufgaben der Projektgruppe im ersten Semester lag, neben der Entwicklung der Runtime-Environment, auch in der Entwicklung einer geeigneten Programmiersprache für ECA-Regelsysteme. Diese ist zugleich das wichtigste Bestandteil des User-Interfaces. Dazu wurde Front-End konzipiert, um die Sprache eigene entwickelte Sprache(ECAL), der Runtime-Environment zugänglich zu machen. Dazu werden auf Technicken des Compilerbaus zurückgegriffen.

2.5.1 Syntaxanalyse

Aufgabe der Syntaxanalyse ist es nun, eine Datei zu zerlegen und einen Syntaxbaum zu erzeugen. Sie läßt sich in der Regel in die Lexikalische Analyse und den eigentlichen Parse Vorgang gliedern.

Lexikalische-Analyse

In der ersten Phase versucht ein Scanner, aus dem Datenstrom, die Tokens(Wörter) der Sprache zu erkennen. Unter Token werden im allgemeinen Schlüsselworte(„begin “, „end “etc.), Symbole (== ,!= ,+ ,- etc.), Bezeichner(Variabelnamen etc.) und Konstanten verstanden. Bei den Tokens handelt es sich um reguläre Ausdrücke. Damit kann der Scanner durch einen endlichen Automaten beschrieben werden.

Der Parse-Vorgang

Im Parse Vorgang versucht ein Parser aus dem eingehenden Tokenstream einen Syntaxbaum zu erzeugen. Die Regeln für die Arbeitsweise eines Parsers, werden formal durch Grammatiken beschrieben. Einem Syntaxbaum liegt, wie jedem Baum, eine kontextfreie Sprache zugrunde, weshalb der Parser durch einen Kellerautomaten beschrieben werden kann.

2.5.2 Kontextanalyse

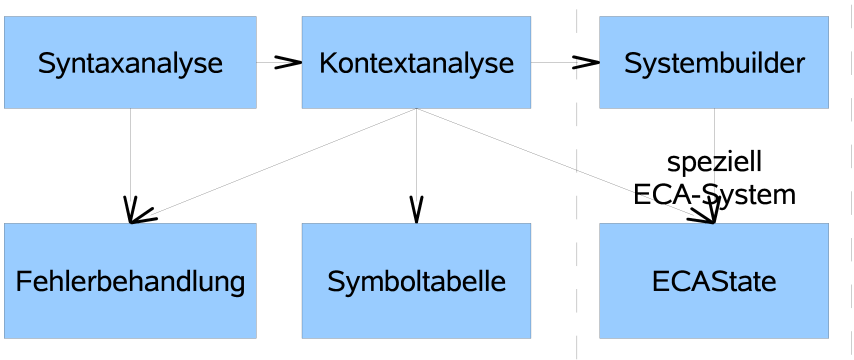

Abbildung 2.8: Die Schnittstellen der Kontextanalyse (speziell ECAX-System)

Regeln die weder von einem Lexer, noch von einem Parser geprüft werden können, werden auch Kontextbedingungen genannt. Die Kontextanalyse wird auch als semantische Analyse bezeichnet. Die Abbildung
2.5.2 zeigt die Schnittstellen der Kontextanalyse. Dabei werden die Knoten um die fehlenden Kontextinformationen in Form von Attributen erweitert. Der Wert der Attribute wird dabei je nach Eigenschaft bei einem Baumdurchlauf Bottom-up, oder Top-Bottom aus dem Kontext des Programmes berechnet. Für die formale Darstellung der Kontextbedingungen, werden Attribut-Grammatiken verwendet. Bei

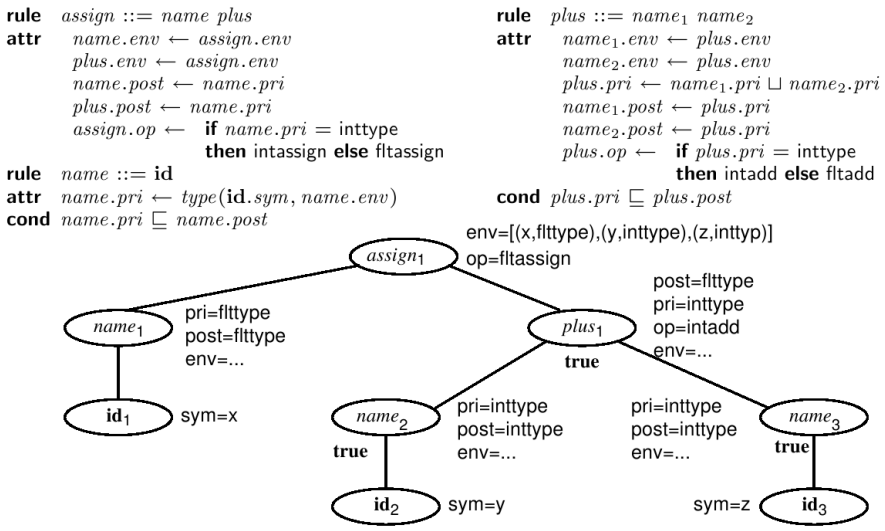

Abbildung 2.9: Beispiel einer Attribut-Grammatik für: x:=y+z

der Menge *G* handelt es sich um eine eingebettete Grammatik, welche die abstrakte Syntax aus der Syntaxanalyse repräsentiert. Eine Attributgrammatik kann den Baum der Kontextanalyse erkennen. Die Abbildung zeigt eine Attributgrammatik und ein Ergebnis. Da die Compilerentwicklung im Rahmen unserer PG zwar eine Menge Zeit in Anspruch genommen hat, jedoch thematisch nicht zum eigentlichen Thema gehört, werden die theoretischen Grundlagen hier nicht weiter erläutert. Für eine gute formale Beschreibung von Attributen und Attributgrammatiken sei auf die Folien von Zimmermann [31] verwiesen.

Realisierung der Deklaration und Typanalyse

In der Praxis werden Attributgrammatiken nicht als Baumdekorationen implementiert. Die Anzahl der Attribute ist meistens zu groß und deren Berechnung für jeden Knoten zu aufwendig. Alternativ bieten sich deshalb verschiedene weitere Realisierungen an:

Speicherung im Laufzeitkeller: Das Verfahren, Attributgrammatiken als Laufzeitkeller spart Platz gegenüber der kompletten Baumdeklaration. Je nach Sprache ist dies jedoch nicht immer möglich.

Speicherung im eigenem Attributkeller: Diese Realisierung spart noch weiteren Speicherplatz gegenüber der Realisierung im Laufzeitkeller. Auch diese Variante ist nicht immer realisierbar.

Speicherung in globaler Variable: Eine Speicherung der Attribute in einer globalen Variable ist zweifellos die sparsamste Variante. Diese Lösung kann jedoch nur in den seltensten Fällen vorgenommen werden.

Als Mischform der bisher vorgestellten Varianten hat sich das Speichern der Attribute in Form einer globalen Symboltabelle durchgesetzt. So kommt auch beim ECAX-System eine Symboltabelle zum Einsatz. Am Ende der Kontextanalyse kann das Programm als syntaktisch und semantisch korrekt interpretiert werden. Nun stehen auch genug Informationen zu Verfügung um Zustandsvariablen für die Laufzeitumgebung zu generieren und die Anweisungen zu verarbeiten.

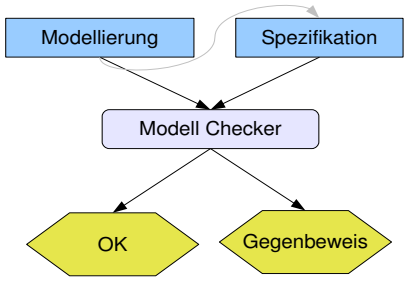

Abbildung 2.10: Vorgehensmodell beim Modell Checking

2.6 Model Checking

Jochen Gerlach, Svetla Nikolova

Nachdem ein Matrose, für seine Bestellungen, ein Formular abschickte, welches mit Nullen gefüllt war, löste ein „Division durch Null“-Fehler eine Kettenreaktion im Hauptcomputer des Schiffes aus und legte somit das ganze Schiff über mehrere Stunden lahm. Raketen mit teuren Satellitenanlagen stürzen ab oder explodieren am Boden, weil die Software logische Fehler enthält. Flugzeugabstürze, Autounfälle verursachen auf der Welt jährlich mehrere Millionen Euro Schaden durch Softwarefehler. Dennoch nimmt die Anzahl der Computer in unserer Umgebung ständig zu. Damit steigt jedoch auch das Risiko für weitere noch teurere Fehler. Deshalb ist es notwendig, neben den Methoden zur Softwareentwicklung auch Methoden zur Softwareverifikation zu finden. Moderne Software ist meistens jedoch sehr komplex und von einem einzelnen Menschen schwer zu durchschauen. Mathematische durch den Menschen durchgeführte Verifikation mit Hilfe formaler Methoden ist meistens sehr aufwendig durchzuführen. Aus diesem Grunde sucht die Wissenschaft nach Methoden, welche vollautomatisch dem Computer überlassen werden können.

Modell-Checking ist ein logikbasiertes vollautomatisches Verifikationsverfahren, um endliche und zunehmend auch unendliche, reaktive Systeme auf gewisse Eigenschaften hin zu überprüfen. Genauer gesagt handelt es sich beim Modell-Checking um eine Sammlung formaler Methoden und einem Vorgehensmodell. Im Idealfall soll Modell-Checking vollständig vom Computer übernommen werden können. Modell-Checker gehören in die Kategorie der vollautomatischen mathematischen Beweiser. Formal läßt sich das Modell-Checking in lokales und globales Modell-Checking einteilen und wie folgt definieren:

Definition 2.8 (Modell-Checking Problem) *Gegeben Sei ein Modell K auf einer Aussagenmenge Φ wobei*

- *Für das globale Modell-Checking gilt:*

$$\forall s \in S : M, S \models \Phi$$

- *Für das lokale Modell-Checking gilt:*
  Sei $s \in S$: Zeige

$$M, s \models \Phi$$

2.6.1 Vorgehensmodell des Modell Checkings

Die Arbeitsweise eines Modell-Checkers läßt sich in 3 Phasen gliedern. Jede dieser Phasen benötigt eigene formale Methoden. Um Software verifizieren zu können, muß in einem ersten Schritt das System modelliert werden. Derartige Modelle haben die Form von Automaten. Zustände werden durch

Knoten, ihre Übergänge durch Transitionen wiedergegeben. Über einen Zustand lassen sich Aussagen machen. Die Transitionen können durch ihre Aktion beschriftet werden. In der Regel muß nicht davon ausgegangen werden, dass Modelle von einem beliebigen Zustand aus gestartet werden können. Dies kann die Anzahl der möglichen Fehler deutlich eingrenzen, da Variablen gewisse theoretische Werte nie erreichen. In einer zweiten Phase wird das Modell als ganzes betrachtet. Nun sollen Aussagen darüber gemacht werden, welche für das ganze Modell gelten sollen. Derartige Aussagen sollen Aufschluss über mögliche Fehler in der Software geben, und fallen oft in die drei Kategorien: Sicherheit, Lebendigkeit und Blockierungsfreiheit. Sofern die ersten beiden Phasen noch vom Menschen kontrolliert und durchgeführt werden können, so ist wenigstens die dritte Phase vollautomatisch. Mit Hilfe eines Verifikationsalgortihmus werden nun die aufgestellten Eigenschaften am Modell überprüft. Wenn der Algorithmus terminiert, gibt er entweder im Erfolgsfall ein OK, oder er liefert im Misserfolgsfall einen Gegenbeweis für das Modell. Ein derartiger Gegenbeweis ist in der Regel die Menge von Beispielzuständen, für die die Eigenschaften auf dem Modell nicht zutreffend sind. Natürlich sind die formalen Methoden nicht unbedingt nur auf Software beschränkt. Modell-Checking kann und wird auch z.B. bei der Verifikation von Prozessoren und Platinen von neuer Hardware verwendet.

2.6.2 Probleme des Modell-Checkings

In der Theorie wird Modell-Checking durch formalen Methoden beschrieben. Diese Methoden liegen in einer derart abstrakten Form vor, dass sie sich im Idealfall auf allerlei Arten komplexer Systeme übertragen lassen. In der Praxis müssen diese Modelle jedoch in Form von Software implementiert werden. Die Fähigkeit der Software, und damit auch des Modell-Checkings, wird durch die knappen Rechenressourcen begrenzt. Manche Problemstellungen werden allein dadurch bereits unmöglich. Obwohl Hard- und Software in der Regel nicht die üblichen Probleme aufweisen, ein System zu modellieren, wie es zum Beispiel bei physikalischen, chemischen, ökonomischen, biologischen und gar psychologischen Systemen der Fall ist, kann sie dennoch ein nicht unerhebliches Maß an Komplexität annehmen. Ist ein Modell-Checker nun gezwungen alle möglichen Fälle für eine Eigenschaft zu verifizieren, so muß er eine gigantische Menge von Zuständen gleichzeitig im Speicher halten können. Derartige Probleme werden als Zustandsexplosion bezeichnet. Formale Methoden, um einer derartigen Explosion entgegen zu wirken, und die Ressourcen auf das nötigste zu reduzieren, sind derzeit immer noch Gegenstand der Forschung.

2.6.3 Formale Grundlagen in der Modellierungsphase

Ziel der Modellierungsphase ist es die Struktur des Systems in allen seinen Einzelheiten auf geeignete Weise zu repräsentieren. Da beim Modell Checking ausschließlich diskrete Systeme betrachtet werden, können deren Zustände und Transitionen als Mengen aufgefaßt werden. Transitionen und Mengen wiederum lassen sich in einem geeigneten Automaten wiedergeben. Automaten des Modell-Checkings dienen lediglich der Verifikation und brauchen somit kein Ein- und Ausgabealphabet. Sie unterscheiden sich hauptsächlich in der Art, wie die Zustände und deren Übergänge beschriftet werden sollen. Gleichzeitig bilden Automaten die Struktur der temporallogischen Aussagen, die in der Spezifikationsphase auf dem Modell gemacht werden sollen.

Kripke-Strukturen (KS)

Bei den Kripke-Strukturen handelt es sich um diejenigen Strukturen, die auch der Modallogik zugrunde liegen. Jeder Knoten wird als eine mögliche Welt angesehen, auf der atomare Aussagen gelten sollen. Auf diesen Welten können modale Aussagen gemacht werden.

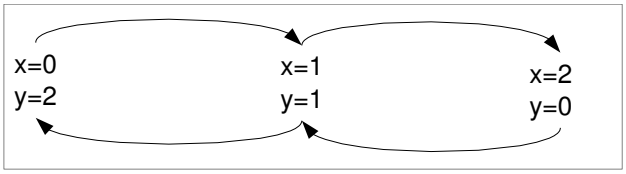

Abbildung 2.11: Kripke Struktur

Definition 2.9 (Definition: Kripke Struktur) *Gegeben sei ein Automat M. M ist eine Kripke Struktur*

$$M = (S, s_0, R, L)$$

auf AP wobei:

- S*: eine endliche Menge von Zuständen*
- $s_0 \in S$*: ist der Startzustand*
- $R \subseteq S \times S$*: ist die Zustandsübergangsrelation*
- $I : S \to 2^{AP}$*: Interpretationsfunktion auf der Menge atomarer Aussagen(AP) die in einem Zustand gelten*

Die Transition von Kripke-Modellen ist total. Das bedeutet, dass jeder Zustand $s \in S$ auf jedem Fall einen Nachfolgezustand s' besitzt. Sollte sich kein derartiger Zustand finden, so gilt, dass der Nachfolgezustand s'=s ist. Üblicherweise werden als atomare Aussagen(AP) meist Symbole verstanden. Sie charakterisieren den Zustand dadurch, ob sie in ihm gesetzt sind.
Kripke-Strukturen beschriften zwar lediglich die Zustände eines Systems, sind aber für viele Modell-Checking Probleme brauchbar. Einige der später betrachteten Temporallogiken benötigen genau diese Menge an Informationen um ein Ergebnis zu ermöglichen. Für ein umfassendes Modell-Checking, fehlen jedoch Informationen über die Art der Übergänge.

Labeled-Transitions-Systeme (LTS)

Während Kripke Strukturen die Dynamik eines Systems in Form atomarer Aussagen beschreibt, so versucht ein LTS die Statik eines Systems zu beschreiben. LTS sind Automaten, welche Kripke-Strukturen um die Beschriftung der Aktionen ergänzen. Auch derartige Beschriftungen können als atomare Aussagen aufgefasst werden. Für jede Transitionen darf allerdings nur jeweils eine Beschriftung verwendet werden.

Definition 2.10 (Definition: Labeled Transition Systems) *Ein LTS ist ein Triple:*

$$T = (S, Act, \to)$$

wobei folgendes gilt:

- S*: eine endliche Menge von Zuständen*
- Act*: eine endliche Menge von Aktionen*
- $\to \subseteq S \times Act \times S$*: eine Übergangsrelation.*

Die Elemente $(s, a, s') \in \to$ werden als „ a-Successors von s “ bezeichnet und zur besseren Lesbarkeit in der Form $s \xrightarrow{a} s'$ notiert.

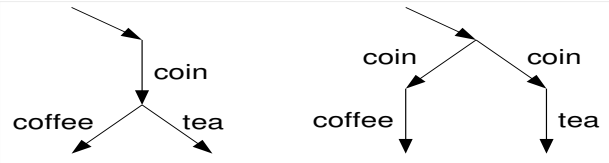

Abbildung 2.12: Labeled Transition Systems

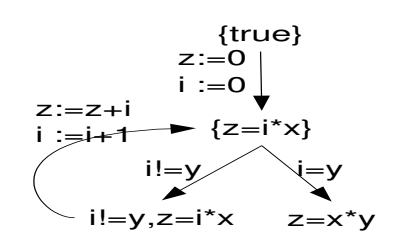

Abbildung 2.13: Kripke Transitions System

Kripke-Transitions-Systeme(KTS)

Kripke-Strukturen und Labeled Transition Systeme betrachten das System jeweils von zwei unterschiedlichen Betrachtungsweisen. Die Automaten können sich jederzeit ergänzen. Eine mögliche Ergänzung ist das Kripke-Transitions-System.

Definition 2.11 (Kripke-Transitions-System) *Ein KTS ist ein Triple:*

$$T = (S, Act, \rightarrow, I)$$

auf AP wobei folgendes gilt:

- S*: eine endliche Menge von Zuständen*
- Act*: eine Menge von Aktionen*
- $\rightarrow \subseteq S \times Act \times S$*: eine Übergangsrelation.*
- I*: Interpretation auf eine Menge von Aussagen*

Aus technischen Gründen sollen sich die Menge atomarer Aussagen nicht mit der Menge der Aktionen überdecken ($Act \cap Ap$).

Zusammenhang zwischen KS, LTS und KTS

KTS können sowohl als generalisierte KS, wie auch als generalisierte LTS aufgefaßt werden. Ein LTS ist ein KTS mit leerer Menge atomarer Aussagen ($AP = \emptyset$). Eine KS ist ein KTS mit einer trivialen Aktion. Obwohl KTS mit ihren verhältnismäßig komplexen Strukturen eine gute Modell-Checking Grundlage bilden, macht es trotzdem Sinn, die Komplexität der Struktur zu vereinfachen und ein gewöhnliches Kripke-Modell einzusetzen.

Abbildung 2.14: Linearzeit Operatoren

Grundlagen der Spezifikationsphase

In der Spezifikationsphase werden die Eigenschaften definiert, mit denen das Modell auf Fehler überprüft werden soll. Mit den bisher vorgestellten Modellen, kann man lediglich Aussagen direkt beantworten, die zu einem bestimmten Zeitpunkt gelten. Um die Korrektheit des gesamten Modells zu gewährleisten, benötigt man komplexere Sprachen, Eigenschaften zu formalisieren:

- Die Eigenschaft des Modells soll von einem einzelnen Zustand aus betrachtet, und auch Aussagen über zukünftige Zustände enthalten.
- Die Eigenschaft des Modells soll von einer Menge von Zuständen aus betrachtet, und auch Aussagen über zukünftige Zustände enthalten.
- Die Eigenschaft soll globale Aussagen, über das Verhalten des ganzen Modells enthalten.

Derartige Eigenschaften formalisieren zeitliche Bezüge. Um diese Bezüge formal korrekt darzustellen benötigt man temporallogische Sprachen.

Temporallogik

In der Logik haben sich zwei temporallogische Paradigmen entwickelt, welche sich hauptsächlich in der Definition des Zeitbegriffes unterscheiden.
Linearzeitlogiken sind dafür geeignet, allgemeine Eigenschaften über Pfade wiederzugeben. Sie ist nicht in der Lage, zwei unterschiedlichen Zustände, welche die gleiche Sprache erzeugen zu unterscheiden. Die Wahl der geeigneten Logik und der damit zugrundeliegenden Sprache hängt von den zu analysierenden Eigenschaften ab. Wegen ihrer besseren Selektionsfähigkeit, eignen sich Verzweigungslogiken eher dazu, reaktive Systeme zu analysieren. Linearzeitlogiken dagegen werden öfter in der Hardwareverifikation eingesetzt.

Linear-Time-Logic (LTL)

LTL, oder auch PLTL, findet oft Anwendung über Kripke Strukturen. Ihre Sprache läßt sich wie folgt definieren.

Definition 2.12 (LTL-Linearzeitlogik) *Gegeben sei eine Kripke Struktur K über AP. Sei p eine Menge atomarer Aussagen $p \subset P(AP)$*
So gilt folgende BNF:

$$\Phi ::= p \mid \neg\Phi \mid \Phi \vee \Phi \mid X\Phi \mid U(\Phi,\Phi) \mid F\Phi \mid G\Phi$$

Eine LTL Formel wird über den Pfaden der jeweiligen KS interpretiert und enthält diejenigen Aussagen p, die in der Menge AP festgelegt worden sind. LTL verwendet die modallogischen Operatoren X, F, G, U, welche umgangssprachlich folgende Bedeutung haben:

- $G\Phi$ (globally) „Φ gilt ab jetzt immer“

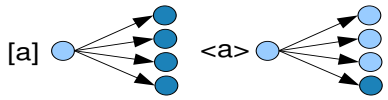

Abbildung 2.15: Verzweigungsoperatoren HML

- $X\Phi$ (next time) „Φ gilt zum nächsten Zeitpunkt“
- $F\Phi$ (finally) „Φ gilt irgendwann“
- $\Phi U \Psi$ (until) „Ψ wird gelten. bis dahin Φ“

Hennessy-Milner Logik(HML)

Eine einfache Logik unter den Verzweigungslogiken ist die HML. HML orientiert sich syntaktisch stark an der Modallogik und hat damit folgende BNF:

Definition 2.13 (HML-Hennessy-Milner-Logik) *Gegeben sei eine KS K über AP Sei p eine Menge atomarer Aussagen $p \subset P(AP)$*
So gilt folgende BNF:

$$\Phi ::= p \mid \neg\Phi \mid \Phi_1 \vee \Phi_2 \mid \Phi_1 \wedge \Phi_2 \mid [a]\Phi \mid \langle a\rangle\Phi$$

- $[a]\Phi$ *bedeutet:* Φ *gilt nach jeder Aktion a.*
- $\langle a\rangle\Phi$ *bedeutet: Es gibt eine Aktion a nach deren Ausführung* Φ *gilt.*

Damit ist die HML gebunden an ein LTS. Mit Hilfe des $[a]$ und des $\langle a\rangle$ Operators, ist es möglich auf die a-Successor Zustände zu verweisen. Manchmal macht es aber auch Sinn, auf Zustände mehrerer Operationen verweisen zu können. Zu diesem Zweck werden die Operatoren auf $[A]$ und $\langle A\rangle$ erweitert, wobei A eine Menge von Aktionen enthalten soll. Bezeichnet man mit A alle Elemente von Act, so notieren man [] und $\langle\rangle$ welche sich äquivalent zur Modallogik auf Kripke-Strukturen verhalten. Als Erweiterung der Modallogik, zeigt die HML, wie man auf LTS eine einfache Verzweigungslogik realisieren kann. Für viele Modell-Checking Anwendungen ist sie jedoch nicht mächtig genug. Deshalb wird in einem späteren Kapitel gezeigt, wie man die HML zu einer wesentlich mächtigeren Logik, dem μ-Kalkül, erweitern kann.

Computation-Tree-Logic (CTL)

Unter Verwendung von LTS oder KTS ist es nun möglich, Pfade durch die Aktionen voneinander zu trennen. Nun suchen wir nach einer Sprache, die auch ohne die Nennung von Aktionen, Verzweigungen erkennen kann. Eine weitere temporäre Verzweigungslogik ist die CTL. Die Syntax beinhaltet die Syntax der LTL. Die CTL erweitert jedoch die LTL durch die Pfadquantoren A und E, die dem modallogischen Pfadoperator vorangestellt werden müssen. Die Bedeutung der Pfadquantoren ist die folgende:

- $A\Phi$ (always $\forall$) entspricht: „Für alle Pfade gilt Φ “
- $E\Phi$ (exist $\exists$) entspricht: „ Es existiert ein Pfad auf dem Φ gilt“

Mit Hilfe der CTL ist es nun auch möglich Pfade auf Kripke-Strukturen voneinander zu unterscheiden. Die modallogischen Operatoren AX, AF, AG, EG können auch aus den übrigen Operatoren durch Äquivalenzumformungen erzeugt werden:

- $AX\Phi = \neg EX\neg\Phi$
- $EF\Phi = \neg E[\top U\Phi]$
- $AF\Phi = \neg A[\top U\Phi]$
- $EG\Phi = \neg AF\neg\Phi$
- $EG\Phi = \neg EF\neg\Phi$

Damit läßt sich die CTL auch in einer kürzeren Grammatik beschreiben. Sie läßt sich dann wie folgt definieren:

Definition 2.14 (CTL (verkürzte Grammatik)) *Gegeben sei eine KS K über AP Sei p eine Menge atomarer Aussagen $p \subset P(AP)$*
So gilt für die CTL folgende BNF:

$$\Phi ::= \bot \mid p \mid \neg\Phi \mid \Phi_1 \vee \Phi_2 \mid EX\Phi \mid E(\Phi U\Psi) \mid A(\Phi U\Psi)$$

Das μ-Kalkül

Keine der drei vorgestellten Temporallogiken ist in der Lage, jede Art von Anfrage auszudrücken. Mit dem μ-Kalkül gibt es allerdings einen Ansatz, eine übergreifende Logik für das Modell-Checking einzuführen. Dazu wird auf die HML zurückgegriffen. Die HML ist eine sehr einfache Logik, welche Schwächen in ihrer Ausdrucksfähigkeit aufweist. Die Ausdrucksfähigkeit ihrer Operatoren reicht lediglich bis zum nächsten Zustand. Möchte man über den nächsten Zustand hinaus Aussagen machen, so müssen die Operatoren geschachtelt werden. Angenommen man möchte, von einem Zustand s ausgehend, auf eine Eigenschaft p in einem Zustand s' hinweisen, so Kann man dies mit der HML nicht ausdrücken. Andere Logiken stellen dafür den Operator F zur Verfügung. Man benötigt also eine passende Erweiterung für die HML. Dazu greift man auf die Grundlagen der Fixpunkttheorie zurück. Angenommen man möchte Zustände S finden, in denen die Aussage p gilt. Dann kann man die Struktur mit einer rekursiven HML-Formel durchsuchen. Dabei ist es von entscheidender Bedeutung, ob man mit der kleinsten möglichen Menge $\emptyset$ oder der größten möglichen Menge S beginnt:

- Bei einer kleinsten Menge soll davon ausgegangen werden, dass die Formel im aktuellen Zustand gelten soll, oder dass für eine Aktion a die Formel rekursiv gelten soll:
  $S = p \vee (q \wedge \langle a\rangle S)$
- Bei der größten Menge wird davon ausgegangen, dass die Formel im aktuellem Zustand gelten soll, und zusätzlich nach Durchführung aller Aktionen a:
  $S = p \wedge [a]S$

Mit einer derartigen Definition wäre es in der HML möglich, einen Zustand s zu finden, auf dem die Aussage p gelten soll. Damit können die Operatoren U und F, auch in der HML repräsentiert werden. Die Erweiterung der HML wird als das μ-Kalkül bezeichnet. In ihrer Definition muß man lediglich die Fixpunktoperatoren hinzufügen:

Definition 2.15 (μ-Kalkül) *Gegeben sei eine KS K über AP. Sei p eine Menge atomarer Aussagen $p \subset P(AP)$. So gilt folgende BNF:*

$$\Phi ::= p \mid \neg\Phi \mid \Phi \vee \Phi \mid \Phi \wedge \Phi \mid [a]\Phi \mid \langle a \rangle \Phi \mid \mu X.\Phi \mid \nu X.\Phi$$

Der Fixpunktoperator $\mu X.\Phi$ steht für den kleinsten Fixpunkt für die Vorkommen X in Φ. Der Operator $\nu X.\Phi$ steht für den größten Fixpunkt. Auch das μ-Kalkül wird zunächst über LTS definiert. Eine beliebige μ-Kalkül Formel Φ auf einem LTS T wird dabei als eine Teilmenge von S interpretiert. Im Kapitel der HML haben wir aber bereits festgestellt, wie einfach sich die Logik auf eine KS anpassen läßt. Dies ist beim μ-Kalkül nicht anders. Damit wird das μ-Kalkül zu einer sehr mächtigen und logisch gut fundierten Sprache. Die folgenden Regeln zeigen, daß man sogar die CTL-Formeln ins μ-Kalkül übersetzen kann:

$$AU(\Phi, \Psi) = \mu X.(\Psi \vee (\Phi \wedge ([]X \wedge \langle\rangle \top))$$
$$EU(\Phi, \Psi) = \mu X.(\Psi \vee (\Phi \wedge \langle\rangle X))$$

Mit Hilfe von weiteren Äquivalenzumformungen, läßt sich jede CTL-Formel übersetzen. Die logische Schlichtheit des μ-Kalkül ist auf den ersten Blick nicht ersichtlich. Dennoch ist das μ-Kalkül bei den meisten Modell-Checkern die grundlegende Sprache. Moderne Modell-Checker, welche die Eingabe der Spezifikationseigenschaften auch in anderen Sprachen akzeptieren, übersetzen die jeweilige Formel zuerst ins μ-Kalkül, bevor sie mit ihrer Arbeit beginnen.

2.6.4 Grundlagen der Modell-Checking Phase

Modell-Checking Algorithmen setzen auf die Theorie der vorangehenden Phasen auf. Zusätzlich greifen sie auf weitere Theorien zurück, die aus anderen Wissenschaften stammen. Dazu gehören unter anderem BBDs und SAT-Solver. Modell Checking kann abhängig von verwendeter Automatenstruktur und temporaler Logik auf unterschiedliche Arten implementiert werden.

- Die *iterative Implementierung* arbeitet besonders gut mit CTL oder dem μ-Kalkül zusammen. Das Problem wird als iteratives finden des Fixpunktes angesehen.
- Die *automatentheoretische Implementierung* findet hauptsächlich bei LTL Anwendung. Die zugrundeliegende Struktur wird, genau wie die Verneinung der zu beweisenden Aussage, in einen Büchiautomaten umgeformt, und der Schnittautomat gebildet. Im Falle eines Fehlschlags liefert der Algorithmus den Gegenbeweis als Schnittmenge.
- Die *Implementierung auf Basis von Tableaus* versucht, mit Kalkülen, das Problem mit Hilfe von Beweisbäumen zu lösen.

2.6.5 Modell-Checking mit dem ECAX-System

Die eigentliche Entwicklung des ECAX-Systemes hat nicht viel mit dem eigentlichen Modell-Checking zu tun. Das ECAX-System ist jedoch für das Modell-Checking konzipiert worden. Um das Verhalten eines ECA-Systemes zu beschreiben wurde dazu, im Rahmen der PG, die Sprache ECAL entwickelt. Die Sprache ECAL verwendet jedoch keinerlei Informationen darüber, wie im zu untersuchenden System die Zustände definiert sind. Mit Hilfe eines Black-Box Ansatzes, testet die LearnLIB das System und versucht die Zustände herauszufinden. Der gefundene Zustandsgraph kann nun mit Hilfe der vorgestellten Modell-Checking Verfahren untersucht werden. Somit bietet die LearnLIB, zusammen mit dem ECAX-System eine sinnvolle Erweiterung für das Modell-Checking von technischen Systemen ohne Zustandsinformationen. Die Anwendung kann jedoch nur begrenzt über die Verifikation technischer Systeme hinausgehen, da diese trotz Black-Box-Ansatz formalisiert werden müssen.

Kapitel 3

Sprachbeschreibung

Falk Howar

Die ECA-Regelsysteme, auf die das Automatenlernen angewendet werden soll, werden als Programme in der Beschreibungssprache *ECAL* formuliert. Diese Sprache wurde in der Projektgruppe Live entwickelt, da keine der in der Seminarphase untersuchten existierenden Sprachen alle benötigten Elemente enthält. Die meisten untersuchten Sprachen basieren auf dem XML-Format, so dass diese zwar sehr gut strukturiert sind, Skripte in den Sprachen aber nicht einfach entwickelt werden können, da der Overhead des XML-Formats recht groß ist und somit die Erstellung der Skripte zeitaufwendig und unübersichtlich ist. *ECAL* bietet durch seine an gängige Programmiersprachen angelehnte Syntax die Möglichkeit ohne diesen Overhead auszukommen und sich an weithin bekannten Ausdrucksformen zu orientieren. Weiterhin sind die untersuchten Sprachen für spezielle Einsatzgebiete entwickelt, so dass deren Umfang über ein entsprechendes Gebiet nicht hinausgeht. *WS-ECA* [15] etwa ist für den Einsatz auf dem Gebiet der Webservices erstellt worden und verfügt nur über eine eingeschränkte Auswahl an Datentypen. *ECAL* bietet eine weitaus größere Ausdrucksstärke als die untersuchten Sprachen und eignet sich dadurch, Regelsysteme aus verschiedensten Gebieten darzustellen, was für die Ziele der Projektgruppe notwendig ist.

Dieses Kapitel beinhaltet eine Referenz für die Sprache *ECAL*. *ECAL* ist eine Event-basierte Programmiersprache, d.h. *ECAL* ist eine Sprache zur Beschreibung von reaktiven Systemen. In ECAL definiert man eine Menge von Events, die zur Ausführungszeit von außen als Eingaben an das System gemacht werden können und eine Menge von Regeln, die jeweils durch ein solches Event „angestoßen“ werden. Die Regeln bestehen aus einer Bedingung und einem Anweisungsteil. Ist die Bedingung erfüllt, werden die Anweisungen ausgeführt. Darüber hinaus können für ein System Invarianten angelegt werden, Bedingungen, die immer oder nie gelten sollen.

ECAL kann (eingeschränkt) auch zur strukturierten Programmierung benutzt werden; das ist aber vielmehr ein Nebenprodukt als eigentliches Entwicklungsziel.

3.1 Die Sprache ECAL

Falk Howar

3.1.1 Verwendete Notation

In dieser Referenz wird folgende Notation verwendet:

```
[ ... ]          Inhalt optional
[ ... ]*         Beliebige Wiederholung
[ ... ]+         Ein- oder mehrfaches Vorkommen
[ ... | ... ]    Alternativen

<name>           Wert / Bezeichner / Typ
```

3.1.2 Programmstruktur

Dieser Abschnitt beschreibt den generellen Aufbau eines *ECAL*-Scripts. *ECAL* ist keine so vergebende Sprache wie *PERL* oder *PHP* sondern lehnt sich mehr an Konzepte aus *PASCAL* oder *Java* an. *ECAL* unterscheidet streng zwischen Groß- und Kleinschreibung. Alle Schlüsselwörter und Termini, die zur Sprache gehören, werden klein geschrieben. Typennamen werden groß geschrieben. Variablennamen beginnen mit einem kleinen Buchstaben.

Per Konvention enden ECAL-Skripte mit der Dateinamenerweiterung „.eca“.

Leerzeichen und Zeilenumbrüche Leerzeichen und Zeilenumbrüche haben in *ECAL* nur bezüglich der Lesbarkeit eine Bedeutung, nicht aber bezüglich Syntax oder Semantik. Leerzeichen sind nur an Stellen erforderlich, an denen sich ansonsten Uneindeutigkeit ergeben würde, z.B. zwischen Schlüsselwörtern.

```
fori = 1 to10    // fault
for i=1to 10     // ok
```

Semikolon Generell wird jede Zeile mit einem Semikolon beendet. Ausnahmen bilden Zeilen, die mit einem Block enden, der von *BEGIN ... END* eingefasst wird, sowie Abschnittsangaben, z.B. (*Init:*).

Als eine einzige Zeile gelten Funktionendefinitionen, Regeln und Invarianten.

Kommentare Kommentare werden wie in Java gekennzeichnet.

```
// Einzeiler

/*
Mehrzeiler
*/
```

Abschnitte

Ein *ECAL*-Script ist in Abschnitte unterteilt. Die Abschnitte können nur in der dargestellten festen Reihenfolge in einem Script enthalten sein. Einige Abschnitte sind optional. Eine detaillierte Beschreibung der Funktionen der einzelnen Abschnitte folgt weiter unten.

```
[Options:]
[Typedef:]
Events:
Variables:
[Functions:]
[Init:]
Ruleset:
[Invariant:]
```

3.1.3 Datentypen und Variablen

In *ECAL* werden alle Variablen streng getypt. Jede Variable hat einen eindeutigen Typ. Implizites Casting wird nicht durchgeführt. Es gibt eine Reihe von in *ECAL* enthaltenen Typen (primitive und komplexe). Außerdem können beliebige zusätzliche Typen definiert werden.

Variablen werden immer durch Angabe des Typen, des Variablennamens und ggf. eines Initialwertes deklariert. Wird kein Initialwert angegeben, bekommt die Variable automatisch einen festgelegten Wert zugewiesen, z.B. *0* bei Zahlen und *false* bei Booleschen Variablen. Bei Aufzählungen wird automatisch das erste Element der Aufzählung zugewiesen.

Auch komplexe Variablen werden nie undefiniert erzeugt. Arrays werden nicht leer instanziiert sondern enthalten sofort alle mit Standardwerten initialisierten Elemente. Die Attribute eines Records werden ebenfalls initialisiert. Sets werden als leere Menge initialisiert.

```
<Type> <varname> [ = ... ];
```

Zuweisungen

Zuweisungen werden durch = vorgenommen und können an den Stellen im Script ausgeführt werden, an denen Anweisungen erlaubt sind.

```
<varname> = <wert>;
```

Zahlen

ECAL unterscheidet zwischen Ganzzahlen und Fließkommazahlen. Die beiden Arten von Zahlen können durch explizites Casten ineinander umgewandelt werden. Die entsprechenden Anweisungen *toInt()* und *toFloat()* werden in der Funktionenreferenz genau beschrieben.

```
Int twelf = 12;          // int
Float something = 12.7;  // float
```

Boolesche Werte

Boolesche Werte sind *true* und *false*. Es ist nicht möglich, Boolesche Werte direkt in Zahlen umzuwandeln oder umgekehrt.

```
Bool truth = true;
```

Strings

Strings sind Ketten von Zeichen. Strings werden durch Anführungszeichen eingefaßt. Für Strings werden keine Funktionen außer der Konkatenation angeboten.

```
String legal   = "Ein Text";     // legal string
String konkat =  "Ein Text" + "noch ein Text";
```

Arrays

Arrays enthalten Elemente eines bestimmten Typen. Dieser Typ wird bei der Instanziierung festgelegt, es wird immer ein Array eines bestimmten Typen erzeugt. Außerdem haben Arrays eine feste Größe, die ebenfalls bei der Instanziierung angegeben wird. Die Anzahl der Dimensionen eines Arrays kann zur Laufzeit mit *dimensions* erfragt werden. Ein bestimmte Dimensionsgröße wird mit der Funktion *dim* zurückgegeben.

Arrays können nicht in der Deklarationszeile initialisiert werden.

```
<Type> [ [<size>] ]+ <name>;

Int[3] numbers = [1,10,100];
Enum{_red, _yellow, _green}[4][7] crossing;
```

Sets

Sets sind Mengen von anderen Typen. In einer Menge kann jede Ausprägung eines Typen nur einmal vorkommen. Sets werden in Mengenschreibweise initialisiert; durch eine Liste von Werten zwischen geschweiften Klammern. Ein Spezialfall ist die Zuweisung einer leeren Menge, dies erfordert die Angabe des Mengentyps vor den geschweiften Klammern.

```
Set{Int} numberSet = {1,2,4,7};

Set{Int} set2 = Int{}; // special case!
```

Listen

Listen enthalten Elemente eines bestimmten Typen. Eine Liste wird leer initialisiert. Elemente können nur am Kopf und am Ende eingefügt, gelesen und entfernt werden. Dazu werden die Funktionen *prepend*, *append*, *getHead*, *getLast*, *removeHead* und *removeLast* benutzt.

```
List<String> myListofStrings;
```

3.1.4 Benutzerdefinierte Typen

Aufzählungen

ENUMs sind benutzerdefinierte Typen. Es wird eine Reihe von möglichen Werten festgelegt. Eine Instanz des Typen kann immer genau einen möglichen Wert aus der Liste annehmen. Enumwerte beginnen mit einem Unterstrich und müssen eindeutig sein. Es dürfen sich keine zwei ENUM-Typen einen Wert teilen.

```
Enum{_red, _yellow, _green} cols = _red;
```

Records

Records sind komplexe Strukturen. Sie enthalten als Attribute Variablen beliebiger anderer Typen. Ein Record darf (auch nicht indirekt) sich selbst als Attribut enthalten, da die erzeugte Datenstruktur unendlich groß wäre.

Records können nicht in der Deklarationszeile initialisiert werden.

```
Rec(
  String name;
  Int age;
) bilbo;

bilbo.name = "baggins";
bilbo.age  = 111;
```

Typdefinitionen

In dem Abschnitt *Typedef* eines *ECAL*-Programms können eigene Typen definiert werden. Als Typ kann alles deklariert werden, was als Variable instanziiert werden kann. Es kann als Typ keine Standardwerte bekommen. Der Typ bekommt einen Namen, über den der Typ später instanziiert werden kann. Typdeklarationen schaffen nicht tatsächlich mehr Typen sondern stellen eine abkürzende Schreibweise dar.

```
Typedef:

Person = Rec(
  String name;
  Int age;
);

Colors = Enum{_red, _yellow, _green};

Variables:

Set<Colors> trafficlight = {_red, _yellow};

Person[1000] largeCompany;
```

Sichtbarkeit

Variablen, die im *Variables*-Abschnitt definiert werden, sind global sichtbar.

Lokale Variablen sind nur innerhalb ihres Blocks, einer Regel, Invariante oder Funktion, sichtbar. Lokale Variablen verdecken bei Namensgleichheit globale.

3.1.5 Verzweigungen und Schleifen

if - then - else

Verzweigungen werden mit *if, then, else* realisiert, wobei der *else*-Block optional ist. Die auszuwertende Bedingung muss den Anforderungen an einen Ausdruck genügen und einen Booleschen Wert zurückgeben.

```
if (<expression>) then
  Anweisungen ...
[else
  Anweisungen ...]
endif
```

for-Schleifen

For-Schleifen zählen im Integer-Zahlbereich in Einerschritten auf- oder abwärts von einem Start- zu einem Endwert. Die Schleife wird auch für Start- und Endwert ausgeführt. Es ist möglich, die Zählvariable in der Schleife zu modifizieren.

```
for <name> = <start> [to|downto] <end> do
  Anweisungen ...
endfor
```

for each-Schleifen

For each-Schleifen können auf Arrays, Listen und auf Sets ausgeführt werden. Sie iterieren über alle in der jeweiligen Variable enthaltenen Elemente.

```
foreach <name> in <set|array> do
  Anweisungen ...
endforeach
```

3.1.6 Operatoren

Unäre Operatoren

Die zwei unären Operatoren sind die Negationen. Not (*!*) negiert einen Booleschen Wert oder Ausdruck, Minus (−) negiert den Betrag einer Zahl.

```
Bool x = false;
Bool notX = !x;  // true

int x = 10;
int y = -x; // y = -10
```

Arithmetische Operatoren

Auf Zahlen sind die vier Grundrechenarten, sowie modulo auf Integern, definiert. Die Operatoren haben verschieden starke Bindungskräfte. * und / binden stärker als + und -. % bindet am wenigsten stark. Der als Beispiel dargestellte Term würde - wie man das erwartet - zu -31 ausgewertet. Die Potenzierung ist mit dem Operator ^ möglich.

```
Int thrirtyOne = 4 + 20/2 + 3*(2-17)

Int powers = x^y;
```

Mengenoperatoren

Auf Mengen sind Vereinigung, Differenz, Schnitt und symmetrische Differenz definiert.

```
c = a [union|diff|inter|symmdiff] b;
```

Auf Mengen sind außerdem folgende Vergleiche definiert:

```
subset, superset, disjoint
```

Vergleichende Operatoren

Vergleichende Operatoren sind auf Sets und Zahlen definiert und liefern jeweils einen Booleschen Wert zurück.

Auf Zahlen sind definiert:

```
>, >=, <, <=
```

Gleichheit

Gleichheit (==) ist auf allen Datentypen definiert und liefert einen Booleschen Wert zurück.

Boolesche Operatoren

Für Booleans sind UND, ODER und NICHT definiert.

```
true && (true || false) == true
```

String-Konkatenation

Zwei Strings können mit + verbunden werden.

3.1.7 Ausdrücke

Ausdrücke können beliebig aus Teilausdrücken zusammengesetzt sein. Innerhalb von Ausdrücken haben verschiedene Operatoren verschieden starke Bindungen. Die schwächste Bindung haben Boolesche Operatoren. Stärker binden die Operatoren-Hierarchien spezieller Typen. Eingeklammerte Terme haben die höchste Bindung (gegenüber dem gesamten Ausdruck).

Funktionen

In *ECAL* können vom Benutzer Funktionen neu definiert werden, externe Funktionen können angestoßen werden und es gibt eine Reihe von zur Sprache gehörenden Funktionen. Funktionen, die zur Sprache gehören, können nicht überschrieben werden.

Benutzerdefinierte Funktionen

Funktionen werden im *ECAL*-Script im Block *Functions* definiert. Funktionen können alle Typen (auch selbst definierte) als Rückgabewert haben oder nichts zurückgeben. Eine Funktion besteht aus dem Funktionskopf, der sich aus Function-Keyword, Returntype, Name und Parameterliste zusammensetzt, und einem Rumpf, der eine Reihe von Anweisungen und ggf. ein „return ...;“ enthält. Funktionen können Elemente von Ausdrücken sein.

```
function [returnType]? <name>(<parameters>)
begin
  Anweisungen ...
  [return <value>]?;
end

// Zum Besipiel ...
function Int increment(Int i)
begin
  i = i + 1;
  return i;
end
```

Externe Funktionen

Externe Funktionen erlauben es, während der Regelverarbeitung Daten an externe Systeme zu übergeben. Externe Funktionen werden ebenfalls im *Function*-Block definiert. Zu jeder externen Funktion muss eine tatsächliche Funktion gleichen Namens und mit derselben Art und Anzahl von Parametern in der in die Ausführungsumgebung per *setUser* geladenen Klasse existieren (siehe auch Abschnitt 5.1.3). Externe Funktionen sind durch das Schlüsselwort extern gekennzeichnet und haben keinen Funktionsrumpf. Als Parameter sind nur primitive Datentypen (Int, Float, Bool und String) erlaubt

Externe Funktionen können nicht Teil von Ausdrücken sein. Externe Funktionen haben keinen Rückgabewert.

```
extern function <name>(<paramters>);
```

3.1.8 Der Init-Block

In einem ECAL-Script kann es einen durch *Init:* gekennzeichneten Init-Block geben. Innerhalb des Init-Blocks funktioniert *ECAL* wie eine strukturierte Programmiersprache. Der Block enthält eine beliebige Folge von Anweisungen.

Variablen, die erst im Init-Block deklariert werden, gelten nicht global sondern nur im Init-Block!

3.1.9 Eventdeklaration

Im Events-Block werden alle externen Events deklariert. Dazu wird ein Name und eine Parametertypenliste angegeben. Externe Events dürfen als Parameter nur Integer und Boolesche Werte haben. Für Integer ist es zudem möglich, den erlaubten Wertebereich einzuschränken.

An dieser Stelle werden nur Parametertypen festgelegt. Der Name, den ein Parameter haben soll, kann pro Regel, in der das Event vorkommt, neu vergeben werden.

```
Temperature(int[-20 to 40]);
```

Interne Events sind Events, welche durch das Verändern von globalen Variablen ausgelöst werden. Diese Events werden nicht explizit im Events-Block definiert sondern einfach im Regelkopf angegeben. Ein internes Event enthält genau einen Parameter, nämlich den Bezeichner der Variable, die beobachtet wird. Beobachtet man Arrays, so gibt man für die Indexe extra Bezeichnernamen an. Unter diesen Namen werden die geänderten Indexe als lokale Variablen verfügbar.

```
on change( myGlobal ) if ...

on change( |x,y,z| myRecs[x][y][z].name) if ...
```

3.1.10 Regeln

Regeln werden im Ruleset-Block angegeben. Regeln bestehen aus drei Teilen, einem Event, einer Bedingung und einem Anweisungsteil. Beim Auftreten des Events wird die Bedingung ausgewertet. Wenn die Bedingung gilt, d.h. als „true“ ausgewertet wird, dann werden die Anweisungen ausgeführt.

Der Eventteil enthält den Namen eines Events aus dem „Event:“-Block, für die erwarteten Parameter werden konrekte Namen vergeben. Die Bedingung ist ein beliebiger Boolescher Ausdruck, der Anweisungsteil kann eine beliebige Menge von Anweisungen (Verzweigungen, Schleifen, Ausgaben, Zuweisungen und Funktionsaufrufen) enthalten.

Regeln können explizit mit einer Priorität versehen werden. Prinzipiell werden höher eingestufte Regeln vor welchen mit niedrigeren Prioritäten ausgeführt. Ob Regeln ohne Angabe vorzuziehen oder hintenanzustellen sind, entscheidet die Auswahl des Regelcaches, den man in die Ausführungsumgebung lädt. Vergibt man gar keine Prioritäten, werden die Regeln in Aufschreibreihenfolge ausgeführt.

```
on <eventname>(<parameters>) [priority <number>]?
if (<expressions>)
do begin
  Anweisungen ...
end
```

3.1.11 Invarianten

In diesem Programmabschnitt werden Invarianten definiert. Dies sind Bedingungen, die in jedem Systemzustand gelten sollen. Welche Aktionen bei der Verletzung von Invarianten durchgeführt werden sollen, wird am Anfang des Programms im Block „Options:“ festgelegt. Außerdem wird dort spezifiziert, wann die Gültigkeit der Invarianten geprüft werden soll.

Die Definition einer Invarianten besteht aus einem Kopf (Schlüsselwort „invar“ und Name der Invarianten) und einem Rumpf, dem Anweisungsteil. Dieser Anweisungsteil enthält die algorithmische Überprüfung der Wahrheitsbedingungen der Invariante. Der Block enthält die Anweisung „exception “, welche das Feststellen einer Verletzung der Wahrheitsbedingungen markiert.

```
invar <name>
begin
  Anweisungen ...
  [exception("<text>");]?
end
```

3.1.12 Options

Es gibt zwei verschiedene Arten von Optionen, nämlich „errorHandling“ und „checkInvariantsAfter“.

Die Option „errorHandling“ spezifiziert die Aktionen, die bei der Verletzung einer Invariante ausgeführt werden sollen. Entweder kann ein Rollback zu dem Systemzustand vor Auslösung des letzten externen Events durchgeführt werden oder die Ausführungsumgebung kann vollständig heruntergefahren werden. Per Default wird „fail“ als Fehlerbehandlung angnommen.

Die Option „errorHandling“ bestimmt, zu welchem Zeitpunkt die Invarianten überprüft werden, nach Beendigung einer von einem externen Event ausgelösten Kaskade, nach jedem Event (intern und extern) oder nach jeder Zuweisung. Per Default wird „Sets“ als Zeitpunkt der Überprüfung angenommen.

Der Options-Block ist optional.

```
Options:

  errorHandling: [fail|rollback]

  checkInvariantsAfter: [externalEvents|internalEvents|sets]
```

3.2 Beschreibung der Standardfunktionen

Said Chihani, Christian Frost

In diesem Abschnitt werden die Standardfunktionen beschrieben, welche die Sprache ECAL dem Benutzer zur Verfügung stellt.

Funktionen zur Ausgabe und Ausnahmebehandlung

Funktionsname	Funktionsbeschreibung
print	**function print (String)** Mit der Funktion „print“ wird ein als Parameter übergebener Ausdruck vom Typ „String“ auf der Konsole ausgegeben. Außerdem wird die übergebene Zeichenkette beim Automatenlernen an die **LearnLib** gesendet, damit sie zur Konstruktion eines Automaten verwendet werden kann. Diese Funktion hat als Parameter einen Ausdruck, der auch als Konkatenation von Teilausdrücken des Typs „String“ definiert sein kann. Variablen und Werte anderer Typen müssen zuerst mit der Funktion „toString“ in Zeichenketten umgewandelt werden, damit sie als Parameter verwendet werden können.
exception	**function exception (String)** Durch die Funktion „exception“ wird eine Ausnahme bei der Überprüfung einer Invariante ausgelöst. Der als Parameter übergebene Ausdruck vom Typ „String“ wird wie bei der Funktion „print“ auf der Konsole angezeigt. Außerdem wird die Ausnahmebehandlung gestartet, deren Art im Optionsteil des Programms nach dem Schlüsselwort „errorHandling“ festgelegt wird. Die Definition der Invarianten und die Optionen werden in den entsprechenden Teilen des Abschnittes „Die Sprache ECAL“ erläutert.

Funktionen für Berechnungen bei Zahlentypen

Maximum, Minimum, Absolutbetrag, Quadrat, Quadratwurzel und Runden sind die allgemeinen Funktionen, die der Benutzer braucht, um Berechnungen mit ganzen Zahlen bzw. reellen Gleitkommazahlen auszuführen.
Alle verwendeten Parameter und der Rückgabewert einer dieser Funktionen müssen den gleichen Typen haben. Wenn die Parameter vom Typ „Int“ sind, so ist auch das Funktionsergebnis eine ganze Zahl. Analoges gilt bei Verwendung des Typen „Float“.

Funktionsname	Funktionsbeschreibung
maximum	**function Int maximum (Int, Int)** **function Float maximum (Float, Float)** Die Funktion „maximum“ berechnet die größere Zahl der zwei gegebenen Parameter und liefert diese als Ergebnis zurück.
minimum	**function Int minimum (Int, Int)** **function Float minimum (Float, Float)** Die Funktion „minimum“ berechnet auf analoge Weise die kleinere Zahl der beiden gegebenen Parameter und liefert diese zurück.
abs	**function Int abs (Int)** **function Float abs (Float)** Mit der Funktion „abs“ wird der Absolutbetrag einer als Parameter übergebenen ganzen oder reellen Zahl bestimmt.
square	**function Int square (Int)** **function Float square (Float)** Die Funktion „square“ berechnet das Quadrat der als Parameter gegebenen ganzen oder reellen Zahl und liefert es als Rückgabewert.
sqrt	**function Float sqrt (Float)** Mit der Funktion „sqrt“ kann man für reelle Gleitkommazahlen die Quadratwurzel bestimmen. Für ganze Zahlen steht diese Funktion nicht zur Verfügung. Im Falle ganzer Zahlen muss deshalb zuerst eine Typumwandlung mit der Funktion „toFloat“ durchgeführt werden
round	**function Float round (Float)** Die Funktion „round“ rundet eine als Parameter gegebene reelle Zahl auf die nächste ganze Zahl. Im Gegensatz zur Funktion „toInt“ wird aber keine Typumwandlung durchgeführt. Das Ergebnis hat zwar den Wert einer ganzen Zahl, es wird aber als „Float“-Variable zurückgeliefert.

Funktionen zur Typumwandlung

Die Sprache ECAL bietet die Möglichkeit, Variablen eines bestimmten Zahlentypen in einen anderen Zahlentypen umzuwandeln. Zu diesen Typumwandlungen dienen die folgenden beiden Funktionen. Außerdem kann mit der Funktion toString eine Darstellung eines Variablenwertes als Zeichenkette erzeugt werden.

Funktionsname	Funktionsbeschreibung
toFloat	**function Float toFloat (Int)** Die Funktion „toFloat“ wandelt eine als Parameter übergebene ganze Zahl in eine reelle Gleitkommazahl mit gleichem Wert um. Wegen der Darstellungsgenauigkeit für reelle Zahlen kann es bei dieser Umwandlung aber zu Verlusten kommen.
toInt	**function Int toInt (Float)** Durch die Funktion „toInt“ wird eine als Parameter übergebene reelle Gleitkommazahl in eine ganze Zahl umgewandelt. Dabei wird die Zahl jeweils zur nächsten ganzen Zahl auf- oder abgerundet.
toString	**function String toString (<type>)** Für alle Variablentypen kann mit der Funktion „toString“ eine Darstellung des Wertes einer Variablen als Zeichenkette erzeugt und zurückgegeben werden. Der Inhalt der erstellten Zeichenkette hängt dabei vom jeweiligen Variablentypen ab.

Gemeinsame Funktionen für Listen und Mengen

Einige Funktionen sind sowohl für Listen als auch für Mengen definiert.

Funktionsname	Funktionsbeschreibung
isEmpty	**function Bool isEmpty ([List< <type> > | Set{<type>}])** Die Funktion „isEmpty“ prüft, ob eine als Parameter übergebene Liste oder Menge leer ist. Der Boolesche Rückgabewert ist bei einer leeren Liste oder Menge „true“ und sonst „false“.
size	**function Int size ([List< <type> > | Set{<type>}])** Durch die Funktion „size“ wird die Anzahl der Elemente einer als Parameter übergebenen Liste bzw. Menge bestimmt. Die Rückgabe hat deshalb den Typen „Int“, weil die Elementanzahl stets ganzzahlig ist.
clear	**function clear ([List< <type> > | Set{<type>}])** Mit Hilfe der Funktion „clear“ können alle Elemente aus der als Parameter übergebenen Liste oder Menge entfernt werden. Das Ergebnis ist eine leere Liste bzw. Menge. Die Elementanzahl ist nach Ausführung der Funktion gleich null.

Funktionen für Listen

Die in der Sprache ECAL definierten Listen ermöglichen nur den Zugriff auf das erste und das letzte Listenelement. Damit sind insbesondere Warteschlangen (FIFO) und Stacks (LIFO) realisierbar.

Funktionsname	Funktionsbeschreibung
prepend	**function prepend (List< <type> >, <type>)** Mit der Funktion „prepend“ wird eine Kopie eines Elementes eines bestimmten Typs „type“ als Kopf in eine Liste mit dem entsprechenden Elementtypen eingefügt. Die übrigen Elemente werden um eine Position nach rechts verschoben und die Größe der Liste wird um eins erhöht. Der erste Parameter ist eine Liste mit dem Elementtypen „type“, der zweite Parameter ist das einzufügende Element vom Typen „type“.
append	**function append (List< <type> >, <type>)** Die Funktion „append“ arbeitet auf analoge Weise wie die Funktion „prepend“. Die Kopie des neuen Elementes wird aber am Ende der Liste angehängt, während die Positionen der alten Listenelemente nicht verändert werden. Die hier verwendeten Parameter haben die gleiche Bedeutung wie in der Funktion „prepend“.
getHead	**function <type> getHead (List< <type> >)** Um das erste Element einer als Parameter übergebenen Liste zu lesen, wird die Funktion „getHead“ aufgerufen. Bei einer Liste mit Elementtyp „type“ ist die Rückgabe eine Variable dieses Typen. Die Liste wird durch diese Funktion nicht verändert, es wird lediglich eine Kopie des Listenkopfes ausgegeben. Die Funktion arbeitet nur korrekt, wenn die Liste nicht leer ist. Dies gilt auch für die drei folgenden Funktionen „getLastElement“, „removeHead“ und „removeLastElement“.
getLast	**function <type> getLast (List< <type> >)** Auf analoge Weise kann durch die Funktion „getLast“ eine Kopie des letzten Elementes der Liste ausgegeben werden. Die Liste bleibt auch hier unverändert.
removeHead	**function <type> removeHead (List< <type> >)** Die Funktion „removeHead “ entfernt das erste Element einer als Parameter übergebenen Liste und gibt es als Rückgabewert aus. Die verbleibenden Listenlemente werden um eine Position nach links verschoben und die Größe der Liste wird um eins verringert.
removeLast	**function <type> removeLast (List< <type> >)** Durch die Funktion „removeLast“ kann man auf analoge Art wie bei der vorigen Funktion das letzte Element einer Liste entfernen und zurückgeben. Die Positionen der übrigen Elemente bleiben unverändert und die Größe der Liste wird um eins dekrementiert.

Funktionen für Mengen

Für Mengen gibt es Funktionen zum Suchen, Einfügen und Löschen von Elementen. Der erste Parameter gibt jeweils die Menge an, während der zweite Parameter eine Variable des Elementtypen dieser Menge ist.

Funktionsname	Funktionsbeschreibung
contains	**function Bool contains (Set{<type>}, <type>)** Die erste Funktion mit dem Namen „contains“ prüft, ob ein Element des bestimmten Typs „type“ in einer Menge mit entsprechendem Elementtypen enthalten ist. Dazu muss das Element in der Menge gesucht werden. Wenn das Element in der Menge gefunden wurde, dann wird der Boolesche Wert „true“ zurückgeliefert, sonst ist die Rückgabe „false“.
addItem	**function Bool addItem (Set{<type>}, <type>)** Mit Hilfe der Funktion „addItem“ kann die Kopie einer Variablen eines bestimmten Typen „type“ als Element in eine Menge dieses Typen eingefügt werden. Weil Mengen gleiche Elemente nicht mehrmals enthalten dürfen, wird vor der Einfügung immer überprüft, ob das einzufügende Element bereits vorhanden ist. Ist dies der Fall, so darf es nicht erneut hinzugefügt werden. Der Boolesche Rückgabewert kennzeichnet, ob das neue Element tatsächlich eingefügt werden konnte oder nicht. Bei erfolgreicher Hinzufügung hat die Rückgabe den Booleschen Wert „true“, während bei einem bereits vorhandenen Element die Rückgabe „false“ geliefert wird.
removeItem	**function Bool removeItem (Set{<type>}, <type>)** Durch die Funktion „removeItem“ wird ein Element vom spezifizierten Typen „type“ aus einer Menge mit entsprechendem Elementtypen entfernt, wenn es in dieser Menge vorhanden ist. Wenn dies der Fall ist, verringert sich die Anzahl der Mengenelemente um eins und der Boolesche Wert „true“ wird zurückgegeben. Ist die angegebene Variable dagegen nicht in der Menge enthalten, so bleibt diese unverändert und der Rückgabewert ist „false“.

Funktionen für Arrays

Der lesende oder schreibende Zugriff auf Elemente eines Arrays wird nicht durch Funktionsaufrufe sondern durch Angabe der Elementposition in eckigen Klammern durchgeführt. Der Wert des Elements kann hierbei durch Zuweisung eines neuen Wertes geändert werden.
Die angebotenen Funktionen dienen nur zur Ausgabe der Dimensionsgrößen von Arrays.

Funktionsname	Funktionsbeschreibung
dimensions	**function Int dimensions (<type>[x0][x1]...[xN])** Die erste Funktion „dimensions“ gibt die Anzahl der Dimensionen eines als Parameter übergebenen Arrays als ganze Zahl aus. Zum Beispiel ist das Ergebnis für einen Vektor aus ganzen Zahlen gleich eins und für eine Matrix aus ganzen Zahlen gleich zwei.
dim	**function Int dim (<type>[x0][x1]...[xN], Int)** Mit der Funktion „dim“ wird die Größe einer durch einen ganzzahligen Index bestimmten Dimension des als Parameter übergebenen Arrays als ganze Zahl zurückgegeben.

Der „return“-Befehl

Befehlsname	Befehlsbeschreibung
return	**return <type>;** In den Funktionen, die eine Rückgabe erzeugen, wird mit dem Schlüsselwort „return“ die Rückgabe der betreffenden Funktion festgelegt. Eine Rückgabe kann einen beliebigen Typen haben, der in der Funktionsdefinition spezifiziert wird. Durch den „return“-Befehl wird gleichzeitig die Funktionsausführung beendet.

Kapitel 4

Referenzbeispiel

4.1 Die Ausführungsumgebung

Falk Howar

Die Ausführungsumgebung für *ECAL* ist ein Set von Javaklassen (ecax). Die Klasse *EcaXSystem* stellt die Programmier-Schnittstelle dar. Das Einbinden der Ausführungsumgebung ist in Listing A.1 dargestellt.

4.1.1 Starten der Ausführungsumgebung

Zuerst wird ein neues *EcaXSystem* erzeugt. Diesem werden zwei Instanzen von Caches übergeben, ein Regelcache und ein Eventcache.

Eventcache Im Eventcache lagert das System Events, die zur Ausführung anstehen. Es wird dabei zwischen internen und externen Events unterschieden. Externe Events sind Events, die vom Benutzer ausgelöst werden, interne Events werden durch Monitore auf Zustandsvariablen ausgelöst. Die einbindbaren Eventcaches unterscheiden sich in der Reihenfolge der Abartbeitung der gespeicherten Events.

EcaEventCacheQueue bearbeitet die Events in der Reihenfolge des Eintreffens.

EcaEventCacheStack bearbeitet die Events so, dass die jeweils durch ein Event ausgelöste Kaskade vor allen weiteren anstehenden Events behandelt wird.

Regelcache Im Regelcache werden die zur Evaluierung anstehenden Regeln gesammelt. Diese Regeln können - müssen aber nicht - Prioritäten (Integerwerte) haben. Die Regeln werden zuerst nach Priorität und dann nach Ankunftszeit zur Ausführung sortiert.

EcaRuleCacheNonPriorityFirst bearbeitet Regeln ohne explizite Priorität so, als ob diese eine höhere Priorität hätten, als alle anderen gecachten Regeln.

EcaRuleCacheNonPriorityLast bearbeitet Regeln ohne explizite Priorität so, als ob diese eine niedrigere Priorität hätten, als alle anderen gecachten Regeln.

Externe Funktionen Nach dem Start der Ausführungsumgebung kann mit *setUser(object ext)* eine Klasse mit externen Funktionen geladen werden. Die in dieser Klasse enthaltenen Funktionen dürfen keine Rückgabewerte haben und können aus dem ECAL-Programm aufgerufen werden.

Laden eines ECAL-Scripts Schließlich kann man die Ausführungsumgebung durch Laden eines ECAL-Scripts konfiguriert werden: *loadFromFile(String filename)*.

4.1.2 Auslösen von Events

Die Ausführungsumgebung bietet einige Dienste an, z.B. das automatische Generieren des gültigen Eingabealphabets für ein konkretes System. *getAlphabet();* gibt einen *Vector<String>* mit allen gültigen Events zurück.

Das laufende System reagiert auf zugesandte Events. Die Events werden dem System als String übergeben. *runOnString(String events);* nimmt als Parameter eine Liste von Events in Form eines „# “getrennten Strings entgegen und gibt die Reaktion(en) des System als „# “getrennten String zurück.

Listing 4.1 *Einbindung der Ausführungsumgebung*

```
// start ecax system
EcaXSystem ecax = new EcaXSystem(
  new EcaEventCacheStack(),
  new EcaRuleCacheNonPriorityLast()
);

// load external functions
ecax.setUser(new EcaxExternalFunctions());

// load ruleset
ecax.loadFromFile(filename);

// get alphabet
Iterator<String> letters = ecax.getAlphabet().iterator();

// post an event to the system and get return
String ecax_out = ecax.runOnString(letters.next());
```

4.2 Die „Foodmachine“

Said Chihani, Christian Frost, Falk Howar

Im folgenden werden die einzelnen Programmteile eines *ECAL*-Skriptes am Beispiel eines Verkaufsautomaten für Lebensmittel beschrieben.
Das vollständige Listing des Beispiels findet sich in Anhang B.

4.2.1 Optionen

Die erste Komponente des Programms ist die Definition der Optionen für die Festlegung der Prüfzeitpunkte der Invarianten und für die Ausnahmebehandlung bei Verletzung von Invarianten.
Dieser Programmblock beginnt mit dem Schlüsselwort „Options:“. Ein Programm muss aber nicht unbedingt eine Festlegung der Optionen enthalten, weil die Voreinstellungen der Optionen verwendet werden können.

```
Options:

  errorHandling: rollback;
  checkInvariantsAfter: externalEvents;
```

Im konkreten Beispiel werden die Invarianten jeweils nach der vollständigen Bearbeitung einer, durch ein externes Ereignis ausgelösten, Regelkaskade überprüft. Dies wird durch den Befehl „checkInvariantsAfter: externalEvents;“ festgelegt.
Im Falle der Verletzung einer Invarianten wird ein Rollback zu dem Systemzustand vor der Auslösung des letzten externen Ereignisses durchgeführt (Befehl: „errorHandling: rollback;“).

4.2.2 Definition von Variablentypen

Im zweiten Block mit der Überschrift „Typedef:“ können Typen für Variablen definiert werden. Diese Typbezeichnungen dienen aber nur als Abkürzung für standardmäßig vorhandene Typen. Auch dieser Programmbestandteil ist optional.

Die verwendbaren Variablentypen lassen sich unterscheiden in einfache und komplexe Variablen.

Einfache Variablentypen sind ganze Zahlen („Int“), reelle Gleitkommazahlen („Float“), Boolesche Wahrheitswerte („Bool“), Zeichenketten („String“) und Aufzählungstypen („Enum“). Aufzählungstypen werden durch die Angabe der Menge der möglichen Variablenwerte definiert.

Bei den komplexen Variablentypen kann man zwischen Containertypen und Verbundtypen (Records) unterscheiden.

Die angebotenen Containertypen sind Arrays, Listen („List“) und Mengen („Set“). Diese Datentypen enthalten immer nur gleichartige Elemente. Als Elementtypen sind wiederum einfache und komplexe Datentypen erlaubt. Dabei sind Arrays nur gleichartig, wenn sie dieselbe Größe haben. Es können auch nur Records mit gleichen Attributnamen und Attributtypen in einem Containerobjekt gespeichert werden.

Der Verbundtyp („Rec“) bietet die Möglichkeit, komplexe Datentypen aus benannten Attributen unterschiedlicher Typen zu konstruieren.

```
Typedef:

  DigitElement = Enum{_top, _middle, _bottom, _left_upper,
                      _left_lower, _right_upper, _right_lower};

  DigitNumber = Set{DigitElement};

  Product = Rec(
    Int id;
    Int price;
    Int count;
    Lamp empty;
  );
```

Im Beispiel werden unter anderem die drei oben angegebenen benannten Variablentypen definiert.

Der Typ „DigitElement“ ist ein Aufzählungstyp. Die Aufzählungstypen werden durch das Schlüsselwort „Enum“ und die Angabe einer Liste der zulässigen Variablenwerte spezifiziert.

Der Typ „DigitElement“ ermöglicht die Definition von Variablen, die Komponenten von Ziffern in der Anzeige des Automaten darstellen. Die Ziffern setzen sich aus höchstens sieben verschiedenen Querbalken („_top“, „_middle“, „_bottom“) und Längsbalken („_left_upper“, „_left_lower“, „_right_upper“, „_right_lower“) zusammen.

Der Mengentyp „DigitNumber“ soll eine Ziffer einer Digitalanzeige beschreiben. Eine Variable dieses Typs ist eine Menge von „DigitElement“-Werten.

Zuletzt wird der Verbundtyp „Product“ festgelegt. Die Verbundtypen werden immer durch das Schlüsselwort „Rec“ und eine Liste von Attributtypen und Attributen definiert.

Der Typ „Product“ setzt sich aus den vier Attributen „id“, „price“, „count“ und „empty“ zusammen. Das Attribut „id“ ist eine Nummer, die das Produkt identifiziert. Mit dem ganzzahligen Attribut „price“ wird der Preis des Produktes in Cent festgesetzt. Die im Automaten enthaltene Menge des Produktes wird im ganzzahligen Attribut „count“ gespeichert. Die letzte Komponente „empty“ vom Typ „Lamp“ ist eine Lampe, die anzeigt, ob das Produkt noch im Automaten vorhanden ist.

4.2.3 Definition der Ereignisse eines ECA-Regelsystems

Mit der Komponente „Events:“ werden die Ereignisse des Regelsystems und ihre zugehörigen Parametertypen festgelegt. Die Regeln des Systems werden durch diese definierten Ereignisse ausgelöst.

Die Syntax eines Ereignisses besteht immer aus einem Ereignisnamen und einer Liste von Parametertypen, wobei auch parameterlose Ereignisse möglich sind.

```
Events:

  refillProduct(Int[0 to 2],Int [1 to 5]);

  giveReturn();

  insertCoin(Int[0 to 4]);

  buyProduct(Int[0 to 2]);
```

Im Beispiel werden im Block „Events:“ vier verschiedene Ereignisse definiert.

Das Ereignis „refillProduct“ ermöglicht das Nachfüllen von Produkten in den Automaten. Der erste Parameter ist eine ganze Zahl im Bereich von null bis zwei, die eines der drei angebotenen Produkte kennzeichnet. Mit dem zweiten Parameter wird die Menge des nachzufüllenden Produktes ausgewählt. Es sind ganze Zahlen aus dem Bereich eins bis fünf möglich, weil fünf als maximale Kapazität des Produktfachs erlaubt ist.

Durch das zweite Ereignis „giveReturn“ wird die Ausgabe des Wechselgeldes ausgelöst. Dieses Ereignis besitzt keine Parameter.

Die Eingabe einer Münze durch einen Kunden wird mit dem Ereignis „insertCoin“ dargestellt. Der Kunde hat die Möglichkeit, fünf verschiedene Münzen zu verwenden (10 Cent, 20 Cent, 50 Cent, 1 Euro, 2 Euro). Die Art der Münzen wird durch einen ganzzahligen Parameter mit dem Wertebereich der ganzen Zahlen von Null bis Vier festgelegt.

Das letzte Ereignis „buyProduct“ ermöglicht dem Kunden die Auswahl des Produktes, das er kaufen möchte. Mit dem Parameter aus dem Wertebereich der ganzen Zahlen von Null bis Zwei wird wie beim Ereignis „refillProduct“ das betreffende Produkt bezeichnet.

4.2.4 Definition der globalen Zustandsvariablen

Der Abschnitt „Variables:“ dient zur Deklaration der globalen Variablen des ECA-Regelsystems. Variablen von einfachen Typen können hier außerdem auch schon initialisiert werden.

Die Beschreibung der möglichen Typen ist in der Dokumentation des Blocks „Typedef“ zu finden.

Man kann bei der Deklaration einer Variablen entweder einen Standardtypen oder einen in der Typendefinition festgelegten Typennamen verwenden.

```
Variables:

  Product[3] slots;

  DigitNumber[10] digits;

  DigitNumber[3] creditDisplay;

  Rec(
    Int value;
    Int count;
  )[5] coins;

  Int credit = 0;
```

Im Referenzbeispiel werden unter anderem die fünf globalen Variablen „slots“, „digits“, „creditDisplay“, „coins“ und „credit“ deklariert.

Die ersten vier Variablen bezeichnen Arrays, die durch die Angabe eines Inhaltstypen und die darauffolgende Größenangabe in eckigen Klammern festgelegt werden.
Alle Arrayvariablen werden in diesem Programmabschnitt nur deklariert. Die Initialisierung muss später im Programmteil „Init“ durchgeführt werden.

Die Variable „slots“ wird als eindimensionales Array der Größe drei mit dem oben definierten Inhaltstypen „Product“ deklariert. Sie bezeichnet den Zustand der Produktfächer des Automaten.

Die Variable „digits“ ist ein Array der Größe zehn mit Komponenten, die Mengen mit Elementen vom Typ „DigitElement“ darstellen. Hierdurch werden für die zehn möglichen verschiedenen Ziffern Null bis Neun jeweils die Mengen der Segmente definiert, aus denen sie sich zusammensetzen.

Bei der Variable „creditDisplay“ handelt es sich wiederum um ein Array mit Inhaltstyp „DigitNumber = Set{DigitElement}“. Die Größe ist hier gleich drei, denn die Anzeige besteht aus drei Ziffern (Einer, Zehner, Hunderter).

Die Variable „coins“ ist ein Array der Größe fünf mit Elementen eines Verbundtyps. Für diesen Verbundtypen wurde im Gegensatz zur Deklaration der ersten Variable kein Name in der Typendefinition festgelegt. In dieser Zustandsvariable werden für die fünf Münzarten jeweils die im Automaten enthaltene Anzahl (ganzzahliges Attribut „count“) und der Münzwert (ganzzahliges Attribut „value“) gespeichert.

Zum Schluss folgt noch eine Variable eines einfachen Typs. Die ganzzahlige Variable „credit“ enthält den Münzbetrag, den ein Kunde in den Automaten eingeworfen hat, bzw. den Wechselgeldbetrag, den der Automat noch zurückgeben muss. Diese Variable wird mit dem Wert Null initialisiert.

4.2.5 Festlegung der im System verwendeten Funktionen

Im Programmteil „Functions:“ werden die Funktionen definiert, die in den Bedingungen und den Aktionen der ECA-Regeln aufgerufen werden können. Die Funktionen haben Einfluss auf den Systemzustand.

Die im Programm definierten Funktionen beginnen jeweils mit dem Schlüsselwort „function“, während externe Funktionen zusätzlich durch das Schlüsselwort „extern“ gekennzeichnet werden.

Darauf folgen ein Rückgabetyp und der Funktionsname. Wenn die Funktion keine Rückgabe erzeugt, so wird kein Rückgabetyp angegeben. Die Funktionsdeklaration endet mit einer Liste von Parametern und ihren entsprechenden Typen, wobei die Parameterliste auch leer sein kann.

Nach der Funktionsdeklaration schließt sich zwischen den Schlüsselwörtern „begin“ und „end“ ein Block von Anweisungen an.
Bei externen Funktionen entfällt dieser Anweisungsblock. Die Implementierung der externen Funktionen wird statt dessen in einer Javaklasse durchgeführt, die dem ECAX-System bekanntgemacht werden muss.

Als Anweisungen sind Wertzuweisungen, bedingte Anweisungen, Zählschleifen (Schlüsselwort „for“) und Funktionsaufrufe zulässig. Die Standardfunktion „print“ erzeugt eine Textausgabe auf der Konsole.

```
Functions:

  function updateDisplay(Int value)
  begin
    Int temp;
    temp = value;
    for i = 0 to 2 do
        creditDisplay[i] = digits[ temp / div[i] ];
        temp = temp - (temp / div[i]) * div[i];
    endfor
  end
```

Die Funktion „updateDisplay“ ist eine lokale Funktion, die den angezeigten Betrag ändert. Sie hat einen Parameter, nämlich den neuen Anzeigewert, und keinen Rückgabewert.

Aus dem Betrag („value“) werden die zugehörigen Ziffern berechnet. In der Variable „creditDisplay“ werden für die einzelnen Ziffern die Komponenten für ihre Darstellung in der Anzeige des Automaten neu festgelegt.

```
extern function dropOrder(Int slot);
```

Die zweite Funktion „dropOrder“ ist eine externe Methode mit einem einzigen ganzzahligen Parameter „slot“. Externe Funktionen besitzen keinen Anweisungsblock, sie müssen statt dessen in einer Javaklasse definiert werden.

Mit dieser Funktion wird eine Bestellung für ein bestimmtes Produkt ausgelöst, das durch den Parameterwert spezifiziert wird.

4.2.6 Initialisierung der globalen Variablen

Der folgende Block mit dem Namen „Init:“ enthält die Initialisierung der globalen Variablen. Zur Durchführung dieser Initialisierung werden Wertzuweisungen, Operationen und Funktionen verwendet. Auch diese Komponente ist optional, denn die Initialisierung von Variablen einfacher Typen ist auch im Abschnitt „Variables:“ möglich.

```
Init:

  for i = 0 to 4 do
    coins[i].count = 2;
  endfor

  coins[0].value = 10;
  coins[1].value = 20;
  coins[2].value = 50;
  coins[3].value = 100;
  coins[4].value = 200;

  for i = 0 to 2 do
    slots[i].count = 1;
    slots[i].empty = _lamp_off;
  endfor

  slots[0].id = 1;
  slots[1].id = 2;
  slots[2].id = 3;

  slots[0].price = 120;
  slots[1].price = 60;
  slots[2].price = 80;

  digits[1] = {_right_upper, _right_lower};
  digits[7] = digits[1] union {_top};
  ...
  digits[6] = digits[8] diff {_right_upper};
  ...

  creditDisplay[0] = digits[0];
  creditDisplay[1] = digits[0];
  creditDisplay[2] = digits[0];
```

Im Initialisierungsblock werden zunächst die Münzspeicher mit der Anzahl der enthaltenen Münzen initialisiert, dann wird jedem Münzspeicher ein Münzwert zugewiesen. Anschließend werden die Produktschächte initialisiert und mit Preisen versehen.

Schließlich werden die Ziffern der Anzeige mit Werten belegt. Als erste Ziffer wird die Eins durch eine Menge aus den Ziffernsegmenten „_right_upper“ und „_right_lower“ initialisiert.

Die Werte der weiteren Ziffern können dann durch Mengenvereinigung („union“) bzw. Mengendifferenz („diff“) aus den bereits initialisierten Ziffern und weiteren Segmentmengen erzeugt werden.

Das Display des Automaten wird so initialisiert, dass es drei Nullen anzeigt.

4.2.7 Festlegung der Regelmenge des ECA-Regelsystems

Der zentrale Teil des Programms hat die Bezeichnung „Ruleset:“. Hier werden alle Regeln des ECA-Regelsystems definiert.

Die einzelnen Regeln bestehen dabei immer aus drei Komponenten für die auslösenden Ereignisse, die Bedingungen und die ausgelösten Aktionen.

Jede ECA-Regel wird von genau einem Ereignis ausgelöst, das mit dem Schlüsselwort „on“ gekennzeichnet wird. Wenn das Ereignis bei der Programmausführung eintritt, dann wird die zugehörige Regel aktiviert.

Der zweite Teil der Regel beginnt mit dem Schlüsselwort „if“ und beinhaltet eine Bedingung, die sich aus mehreren mit logischen Konnektoren verbundenen Teilausdrücken zusammensetzen kann. Nach der Aktivierung der Regel wird die Gültigkeit dieser Bedingung überprüft. Im Falle der Gültigkeit der Bedingung werden die Aktionen im dritten Teil der Regel ausgeführt, sonst wird die Regelverarbeitung ohne Durchführung der Aktionen beendet.

Zuletzt folgen die Aktionen, die in einem Block zwischen den Schlüsselwörtern „do begin“ und „end“ angegeben werden. Alle diese Aktionen werden im Falle des Eintretens des Ereignisses und der gleichzeitigen Gültigkeit der Bedingung ausgeführt. Die Aktionen umfassen Zuweisungen, bedingte Anweisungen, Schleifen und Funktionsaufrufe.

```
Ruleset:

  on refillProduct(slot,refill)
  if (slots[slot].count + refill < 21)
  do begin
    slots[slot].count = slots[slot].count + refill;
    slots[slot].empty = _lamp_off;
  end
```

Die erste Regel im Referenzbeispiel wird durch das Ereignis „refillProduct“ mit den Parametern „slot“ und „refill “ ausgelöst. Dieses Ereignis und die Parametertypen wurden in der Komponente „Events“ deklariert.
Das Ereignis bedeutet, dass die Anzahl „refill“ eines Produktes in das entsprechende Fach „slot“ im Automaten nachgefüllt werden soll.
Beim Eintreten dieses Ereignisses wird die Bedingung der Regel überprüft. Diese ist genau dann wahr, wenn die Summe aus der aktuell im Automaten vorhandenen Menge des Produktes und der Anzahl „refill“ die maximale Kapazität des Faches (zwanzig) nicht überschreitet. Nur in diesem Fall kann das Nachfüllen auch durchgeführt werden, sonst wird die Regelbearbeitung ohne Aktionsausführung beendet.
Die Aktionen, die bei der Bearbeitung der Regel ausgeführt werden, sind zwei Zuweisungen. Zuerst wird die Menge des Produktes im ausgewählten Fach um die spezifizierte Anzahl erhöht. Danach wird der Status der Lampe auf „_lamp_off“ gesetzt, wodurch angezeigt wird, dass das Fach nicht leer ist.

```
  on insertCoin(nr)
  if (coins[nr].count < 5 && credit < 300)
  do begin
    credit = credit + coins[nr].value;
    coins[nr].count = coins[nr].count + 1;
    updateDisplay(credit);
  end
```

Für den Kauf eines Produktes muss der Kunde zuerst einen ausreichenden Geldbetrag in den Automaten einwerfen.

Das Ereignis „insertCoin“ beschreibt den Einwurf einer Münze der Art „nr“. Die Aktionen dieser Regel werden nur bearbeitet, wenn die Anzahl der im entsprechenden Münzfach enthaltenen Münzen die maximale Kapazität fünf noch nicht erreicht hat und der Gesamtkredit kleiner als 3 Euro ist.

Wenn diese Bedingungen erfüllt sind, wird zuerst der Zähler für den insgesamt eingeworfenen Betrag um den entsprechenden Münzwert erhöht. Die Anzahl der Münzen der ausgewählten Art wird um eins inkrementiert.

Die letzte Aktion ist der Aufruf der Funktion „updateDisplay“, durch den der neu berechnete Betrag „credit“ in die Anzeige des Automaten übernommen wird.

```
on buyProduct(slot)
if (credit - slots[slot].price >= 0
&& slots[slot].count > 0)
do begin
  credit = credit - slots[slot].price;
  updateDisplay(credit);
  slots[slot].count = slots[slot].count - 1;
end
```

Das Ereignis „buyProduct“ tritt ein, wenn ein Kunde ein bestimmtes Produkt kaufen möchte. Das Produkt wird durch den Parameter „slot“ bestimmt.

Die Bedingung besteht bei der obigen Regel aus der Konjunktion von zwei atomaren Bedingungen. Im ersten Ausdruck wird überprüft, ob der eingeworfene Geldbetrag mindestens so groß wie der Produktwert ist. Der zweite Ausdruck prüft, ob die vorhandene Menge des Produktes größer als 0 ist.

Nur wenn diese zusammengesetzte Bedingung wahr ist, kann der Kauf des Produktes tatsächlich erfolgen. Dabei werden die folgenden Aktionen durchgeführt.

Zuerst wird der eingeworfene Betrag um den Preis des gewählten Produktes dekrementiert und durch die Funktion „updateDisplay“ in die Anzeige des Automaten geschrieben. Dieser Betrag gibt nun das Wechselgeld an, das der Automat zurückgeben muss.

Durch den Kauf des Produktes verringert sich dessen Menge im entsprechenden Fach um eins.

```
on giveReturn()
if (true)
do begin
  Int retMoney;
  retMoney = credit;

  /* return the money */
  for i = 0 to 4 do
    coins[4-i].count = coins[4-i].count - (credit / coins[4-i].value);
    credit = credit - coins[4-i].value * (credit / coins[4-i].value);
  endfor

  /* refresh the display */
  creditDisplay[0] = digits[0];
  creditDisplay[1] = digits[0];
  creditDisplay[2] = digits[0];
end
```

Durch das Ereignis „giveReturn“ wird eine Regel ausgelöst, deren Aktionen immer ausgeführt werden. Dies wird durch die Bedingung „true“ ausgedrückt, die stets wahr ist.

In der Schleife werden die fünf Münzarten nach absteigenden Werten betrachtet. Für jede Münzart wird die maximal mögliche Anzahl von Münzen bestimmt, so dass der Restwert „credit“ nach der Subtraktion des Wertes dieser Münzen nichtnegativ bleibt. Die Anzahl der im Automaten enthaltenen Münzen einer bestimmten Art „i“ wird in der Variable „coins[i].count“ gespeichert.

Nach der Rückgabe des Wechselgeldes ist der Wert der Variable „credit“ gleich null. Die Ziffern der Anzeige werden deshalb jeweils als Null festgelegt.

```
on change( |slot| slots[slot].count)
if (slots[slot].count == 0)
do begin
  slots[slot].empty = _lamp_on;
  dropOrder(slot);
end
```

Die letzte Regel wird durch das Standardereignis „change“ ausgelöst. Dieses Ereignis hat stets genau eine globale Variable als Parameter. Es tritt immer dann ein, wenn der Variablenwert sich ändert.

In der konkreten Regel überwacht das Ereignis die Änderung der Zähler für die Produkte in den Fächern des Automaten. Wenn sich die Produktmenge in einem bestimmten Fach „slot“ ändert, wird die Regel aktiviert.

Die Bedingung legt fest, dass die Aktionen nur ausgeführt werden sollen, wenn der Zähler den Wert Null hat.

In diesem Fall ist das Produktfach leer, deshalb wird die Lampe für die Kennzeichnung des leeren Faches eingeschaltet.

Am Ende der Regel wird die externe Javamethode „dropOrder“ aufgerufen. Diese generiert eine Bestellung für das entsprechende Produkt.

Das Nachfüllen des bestellten Produktes wird durch die erste Regel im hier definierten ECA-Regelsystem modelliert.

4.2.8 Definition der Invarianten

Am Ende eines Programms kann noch die optionale Komponente „Invariants:“ auftreten. In diesem Programmabschnitt werden Invarianten definiert. Dies sind Bedingungen, die in jedem Systemzustand gelten sollen. Welche Aktionen bei der Verletzung von Invarianten durchgeführt werden sollen, wird am Anfang des Programms im Block „Options:“ festgelegt. Außerdem wird dort spezifiziert, wann die Gültigkeit der Invarianten geprüft werden soll.

```
Invariant:

  invar changeError
  begin
    for i = 0 to 4 do
      if (coins[i].count < 0) then
        exception("Coins empty");
      endif
    endfor
  end
```

Die einzige Invariante in diesem Beispiel hat den Namen „changeError“ und betrifft die Anzahl der Exemplare der verschiedenen Arten von Münzen im Automaten.

Wenn von einer Münzart, durch die Ausführung der Regel zur Ausgabe des Wechselgeldes, eine negative Anzahl im Automaten enthalten ist, so ist die Invariante verletzt. In diesem Fall wird eine Ausnahme (Exception) mit dem Text „Coins empty“ verursacht.

Die Art der Ausnahmebehandlung wird im Optionsblock festgelegt, was bereits erwähnt wurde.

Kapitel 5

Objektmodell

In diesem Abschnitt wird ein Überblick über das Objektmodell gegeben.
Im ersten Teil wird auf den Aufbau des Modells mit seinen Paketen eingegangen und diese kurz erläutert. Im zweiten Teil wird ein Einblick in den Parser gegeben sowie seine Arbeitsweise und Komponenten vorgestellt.

5.1 Laufzeitumgebung

David Karla, Svetla Nikolova

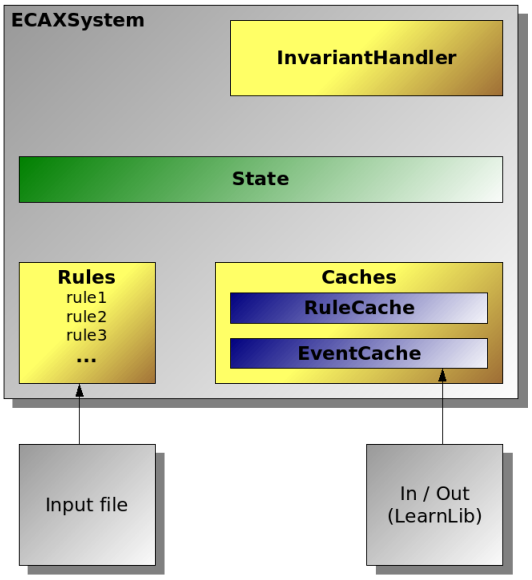

Abbildung 5.1: Schematische Darstellung der Laufzeitumgebung

Dieser Abschnitt gibt einen Überblick über die wesentlichen Teile des Quellcodes der Laufzeitumgebung (siehe Abbildung 5.1). Der Umfang beschränkt sich dabei auf die Beschreibung der Funktion eines Quellcodepaketes oder einer Klasse.

5.1.1 Beschreibung der Quellcodepakete

Das gesamte Javaprojekt wurde aufgrund des Umfangs und der Übersicht in mehrere Pakete aufgeteilt. Dabei sind logisch zusammengehörige Klassen im gleichen Paket zu finden. Im Folgenden werden alle Pakete detailliert beschrieben.

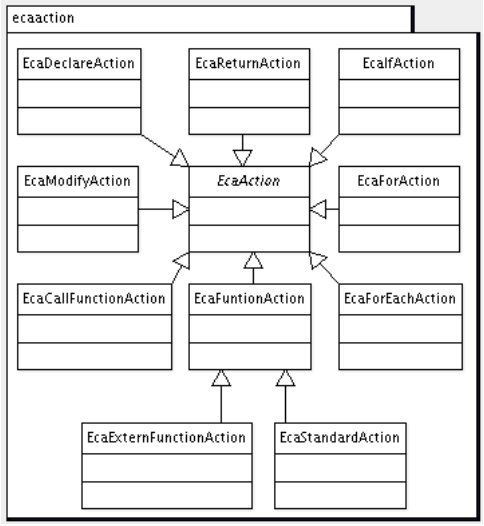

Abbildung 5.2: Hierarchie von Actions-Klassen

Package *ecaaction*

Diese Paket (siehe Abbildung 5.2) beinhaltet Klassen für die verschiedenen Aktionstypen einer Regel. Deshalb spiegeln alle hier definierten Klassen die Leistungsfähigkeit der Laufzeitumgebung wieder. Die Java-Quellcode-Datei „EcaAction.java“ beschreibt eine abstrakte Klasse, die Methoden bereit stellt, mit denen mehrere dieser Aktionen hintereinander ausführbar sind. Jedes Objekt einer Aktionsklasse enthält einen Zeiger auf eine Nachfolgeaktion. Die Nachfolgeaktionen lassen sich also so direkt ausführen, falls der Zeiger auf ein weiteres Objekt verweist.

Die wesentlichen Aktionstypen sind:

- *EcaDeclareAction* generiert zur Laufzeit eine Variable beliebigen Typs. Der Gültigkeitsbereich der Variablen ist auf die Regel beschränkt, in der die Definition vorgenommen wird.
- *EcaExternFunctionAction* dient der Benutzung externer Funktionen. Diese liegen als Java-Bytecode vor (.class-Datei) und können so in ein Eca-Skript eingebunden werden. Das bietet die Möglichkeit externe Geräte oder Programme zu steuern oder zu verwenden.
- *EcaForAction* repräsentiert einfache Schleifen, bei der eine Zählvariable verwendet wird. Es lassen sich andere Aktionstypen angeben, die während eines Schleifendurchlaufs ausgeführt werden sollen. Dabei kann auf die Zählvariable lesend und schreibend werden. (Anwendungsbeispiel: 3.1.5)
- *EcaForEachAction* dient zum Iterieren über abzählbare Mengen wie beispielsweise Sets oder Arrays. Dabei ist das jeweils aktuelle Element dieser Menge während der Iteration als lokale Variable verfügbar und kann so in verschiedenen weiteren Aktionen benutzt werden. (Anwendungsbeispiel: 3.1.5)

- *EcaFuncAction* ist äquivalent zu Funktionen einer Programmiersprache. Dadurch lassen sich mehrfach verwendete Aktionen zu einer wiederverwendbaren Funktion zusammen zu fassen. Diese Aktion stellt eine Schablone für eine Funktion dar, die genau beschreibt, was mit den Parametern und lokalen Variablen geschehen wird. Zur Laufzeit werden durch eine Instanz der EcaSetFuncParameterAction die tatsächlichen Werte der Parameter gesetzt.
- *EcaIfAction* erlaubt das Verhalten des Regelwerks aufgrund des Ergebnisses eines logischen Ausdrucks ([30]) zu verändern. Dieser Aktionstyp repräsentiert die „If-Then-Else“-Kontrollstruktur gängiger Programmiersprachen. Es lassen sich jeweils Folgeaktionen für den Fall, dass der logische Ausdruck wahr bzw. falsch ist angeben. (Anwendungsbeispiel: 3.1.5)
- *EcaModifyAction* ermöglicht die Manipulation des Systemzustands, indem einfach einer existierenden Zustandsvariablen ein neuer Wert zugewiesen wird. Will man den Systemzustand durch die Ausführung einer Regel verändern, so kann dies nur mit Hilfe dieser Klasse geschehen. (Anwendungsbeispiel: 3.1.3)
- *EcaReturnAction* erlaubt den Rückgabewert einer Funktion oder den eines logischen Ausdrucks in einer speziellen Variablen zu speichern.
- *EcaCallFunctionAction* ermöglicht das Verändern von Funktionsparameter zur Laufzeit. Wird eine Funktion mehrfach verwendet, so wird durch diese Aktion definiert, welche Werte die Parameter bei der nächsten Ausführung einer Funktion haben. (Ein Beispiel: erster Aufruf: sum(1,a), zweiter Aufruf: sum(a,b). Vor dem zweiten Ausführung der Funktion muss also sichergestellt werden, dass die Parameter nun die Werte von a und b annehmen und nicht mehr 1 und der Wert von a. Genau diese Zuweisung macht man mit Hilfe dieser Aktion). Diese Funktion wird nur intern verwendet, lässt sich also nicht durch die Sprache ECAL definieren.
- *EcaStandardFunctionAction* implementiert viele Standardfunktionen, die direkt zur Sprache gehören. Das beinhaltet mathematische Methoden (Min, Max, Sqrt usw.) als auch Funktionen für den Umgang mit komplexen Datentypen (Size, getFirstElement usw.).

Package *ecacondition*

Dieses Paket enthält nur die Klasse *EcaCondition*, welche einen booleschen Ausdruck repräsentiert, der sich zur Laufzeit unter Zugriff auf alle Systemvariablen auswerten lässt. Dies stellt also exakt den Conditionteil einer Eca-Regel dar. Ist die Bedingung erfüllt (also der Boolesche Ausdruck wahr), so wird der Actionteil der entsprechenden Regel ausgeführt.

Package *ecaevent*

Dieses Paket (siehe Abbildung 5.3) beinhaltet Klassen, die regelauslösende Ereignisse abbilden. Weiterhin sind Kontrollstrukturen zur Verwaltung der Ereignisse enthalten.

- *EcaSystemEvent*: Diese Klasse beschreibt bei der Zusammensetzung einer Regel den Eventteil. Das Ereignis kann durch einen Namen und die Angabe von Parametern weiter spezifiziert werden. Eine Regel wird aktiv, wenn zur Laufzeit ein Event mit diesen Eigenschaften auftritt.
- *EcaRuntimeEvent*: Diese Klasse dient der Repräsentation auftretender Events. Tritt ein EcaRuntimeEvent auf, so wird eine Regel mit äquivalentem EcaSystemEvent gesucht und ausgeführt. Diese „Suche“ ist durch EventListener implementiert, wodurch die Events mit Regeln verknüpft werden. Weiterhin kann unterschieden werden, ob ein Ereignis von extern oder intern (z.B. durch Änderung einer Zustandsvariablen) aufgetreten ist, was wichtig für die Bearbeitungsreihenfolge sein kann.

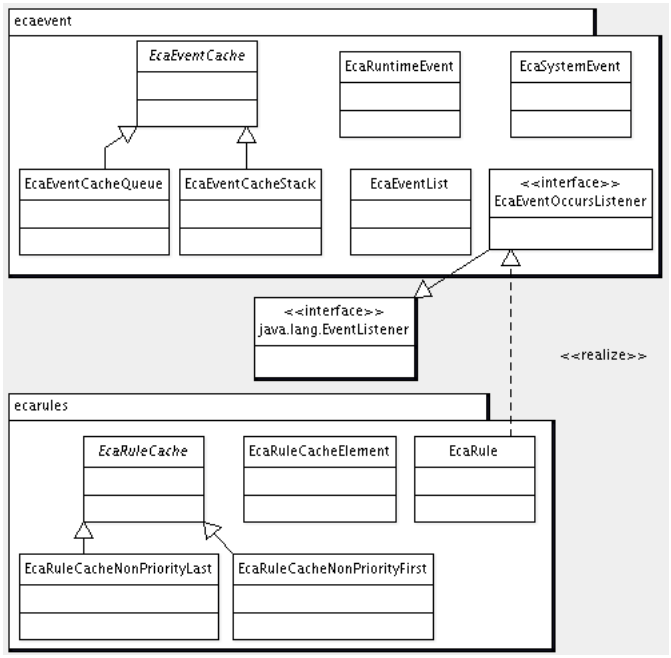

Abbildung 5.3: Hierarchie von Event-/Rules-Klassen

- *EcaEventList*: Alle in den Regeln definierten Eventobjekte werden in dieser Liste gespeichert.
- *EcaEventCache*: Diese abstrakte Klasse dient als Speicher für die Laufzeitereignisse. Dadurch können auftretende Ereignisse zwischengespeichert werden, falls das System noch mit der Regelbearbeitung anderer Ereignisse beschäftigt ist. Bisher sind hier als konkrete Klassen eine Queue und ein Stack implementiert. Andere Schedulingmethoden sind denkbar.

Package *ecaexpressions*

In diesem Paket (siehe Abbildung 5.4) befinden sich alle Klassen, die zur Auswertung aller Arten von Operationen nötig sind. Alle Klassen implementieren das Interface EcaEvaluable dieses Pakets, welches die Methode evaluate() definiert. Wird diese Methode für einen Ausdruck aufgerufen, so errechnet der Aufruf das Ergebnis des Ausdrucks. Weiterhin enthält diese Paket Hilfsklassen für die Namensauflösung der Variablen, so dass auch zu langen Namen (z.B. MyRecord.MyField1.Value[x].MyField2) komfortabel der eigentliche Wert abgefragt werden kann.

- *EcaBoolExpression*: Durch Objekte dieser Klasse lassen sich Boolesche Ausdrücke ([27]) darstellen und auswerten. Das Ergebnis der Evaluation eines Ausdrucks ist true oder false.
- *EcaExpressionDynamicTerminal*: Mit Hilfe dieser Klasse lassen sich dynamische Ausdrücke aufbauen. Das heißt, dass Objekte dieses Typs zur Laufzeit auf verschiedene Variablen zeigen können. Dies ist etwa bei der Erstellung von Funktionen wichtig, bei denen für jeden Aufruf andere Parameter verwendet werden sollen.
- *EcaExpressionStaticTerminal*: Dieser Ausdruck verweist direkt auf eine Zustandsvariable. Wertet man diesen Ausdruck durch die Methode „evaluate“ aus, so wird einfach die entsprechende Variable zurückgegeben.

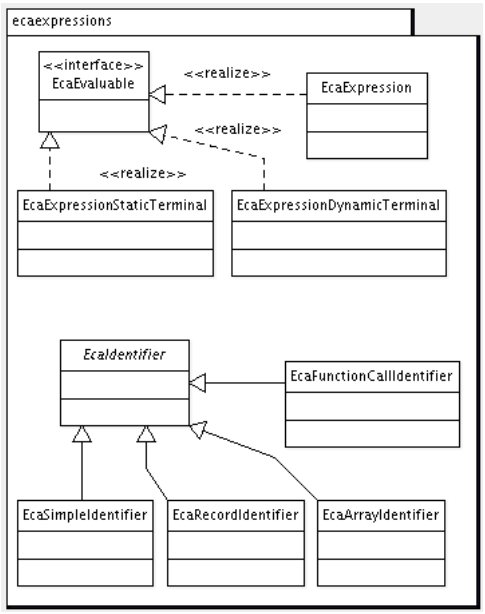

Abbildung 5.4: Hierarchie von Expressions-Klassen

- *EcaIntegerExpression*: Ausdrücke dieses Typs haben als Ergebnis eine Variable des Typs Integer. Natürlich lassen sich diese Ausdrücke mit integertypischen Operatoren (+, -, ...) verknüpfen, wahrend boolesche Ausdrücke entsprechend boolesche Operatoren verwenden.

Package *ecainiout*

Dieses Paket dient der Kommunikation mit anderen Programmen (etwa der LearnLib) oder einem interaktiven Benutzer. Die Kommunikation findet in drei Phasen statt, welche jeweils durch eine Klasse repräsentiert wird:

- *EcaAlphabetGenerator*: Um entsprechend sinnvolle Eingaben an die Laufzeitumgebung zu senden, kann diese Klasse einem Benutzer oder anderen Programm mitteilen, welche Eingaben gültig sind. Eingaben sind immer spezielle Events, oft mit Parameter. Ist der Parameter eines Events vom Typ Integer, so gäbe es Eingaben, die den gesamten Integerbereich abdecken werden (Event(1), Event(2), ...). Dadurch würde das Eingabealphabet sehr groß werden und der Lernvorgang der LearnLib würde entsprechend sehr viel Zeit in Anspruch nehmen. Deshalb ist es möglich, für Eingaben Wertebereiche zu definieren.

- *EcaInputGetter*: Die Klasse kann externe Eingaben in entsprechende Event-Objekte der Laufzeitumgebung umwandeln. Die Eingaben liegen als String vor. Während des Parsens dieser Eingaben wird auch die Korrektheit der Eventparameter usw. geprüft.

- *EcaOutputWriter*: Diese Klasse dient der Darstellung eines Ausgabepuffers. Dabei ist es wichtig, dass in der Ausgabe erkennbar ist, welche Ausgabe zu welchem Eingabeereignis gehört. Ansonsten kann die LearnLib die Ausgaben nicht korrekt zuordnen und ein Regelsystem könnte nicht richtig gelernt werden. Aus diesem Grund werden die Ausgaben für unterschiedliche Ereignisse durch ein Sonderzeichen „|“ getrennt.

Package *ecainvariant*

Invarianten beschreiben gültige Systemzustände bzw. die Grenzen zwischen gültigen und ungültigen Zuständen. Beispielsweise soll die Raumtemperatur nie über 25 Grad Celsius steigen. Geschieht dies, so ist eine Invariante verletzt und das System in einem ungültigen Zustand. Folgende Klassen spiegeln diese Umstände in der Laufzeitumgebung wieder:

- *EcaInvariant*: Hiermit lassen sich Aktionen beschreiben, die die Einhaltung der Invarianten testen. Sind Bedingungen für einen gültigen Systemzustand nicht eingehalten, so wird eine Exception geworfen.
- *EcaInvariantHandler*: Diese Klasse übernimmt die Ausführung der Invariantenprüfung. Dabei können verschiedene Prüfzeitpunkte gewählt werden: Es kann etwa festgelegt werden, ob die Invarianten zu jedem Zeitpunkt gelten müssen oder interne Ereignisse auch zwischenzeitlich in ungültige Zustände führen dürfen.

Package *ecarules*

Durch Regeln (siehe Abbildung 5.3) wird das gesamte Verhalten des Regelsystems beschrieben. Eine Regelobjekt besteht aus einem Event, einer Condition und einer Reihe von Actions. Tritt das Event auf, so wird die Regel ausgeführt. Sind alle Bedingungen erfüllt, werden die Aktionen gestartet. Das Laufzeitsystem unterstützt die Beeinflussung der Ausführungsreihenfolge der ausgelösten Regeln durch Angabe von Prioritäten.

- *EcaRuleCache*: Die Klasse stellt eine Warteschlange für die auszuführenden Regeln dar. Je höher eine Regel priorisiert ist, desto weiter vorne wird sie in die Schlange eingefügt. Hier sind mehrere erbende Klassen vorstellbar, die andere Schedulingmethoden implementieren. Prioritäten sind optional, so dass auch stets darauf geachtet werden muss, wie Regeln ohne Prioritätsangabe behandelt werden.

Package *ecastate*

Der Systemzustand der Laufzeitumgebung wird durch diese Klasse repräsentiert. Im Wesentlichen handelt es sich um einen Container mit Variabelobjekten. Die Klasse bietet noch einige Hilfsmethoden, um eine Kopie des Systemzustand zu erstellen oder diese Kopie wieder herzustellen, was etwa bei der Verletzung von Invarianten erforderlich sein kann.

Package *ecasystem*

Die einzige Klasse diese Pakets stellt die Hauptklasse einer Laufzeitumgebung dar. Wird ein Objekt instanziert, so muss ein Initialzustand angegeben werden. Dieser wird durch Parser und Treewalker aus der Eingabedatei erzeugt, ebenso wie alle Regelobjekte in das System eingefügt werden. Ist ein komplettes Laufzeitsystem aufgebaut, so kann es auf einer Eingabe, die eine Reihe auftretender Ereignisse darstellt, ausgeführt werden.

Package *ecavariables*

Alle Klassen dieses Pakets (siehe Abbildung 5.5) repräsentieren die Datentypen. Es stehen jeweils Methoden zur Generierung der Typen zur Verfügung, zum Manipulieren der Werte und zum Anlegen einer Kopie einer Objektinstanz. Es werden folgende gängigen Datentypen unterstützt:

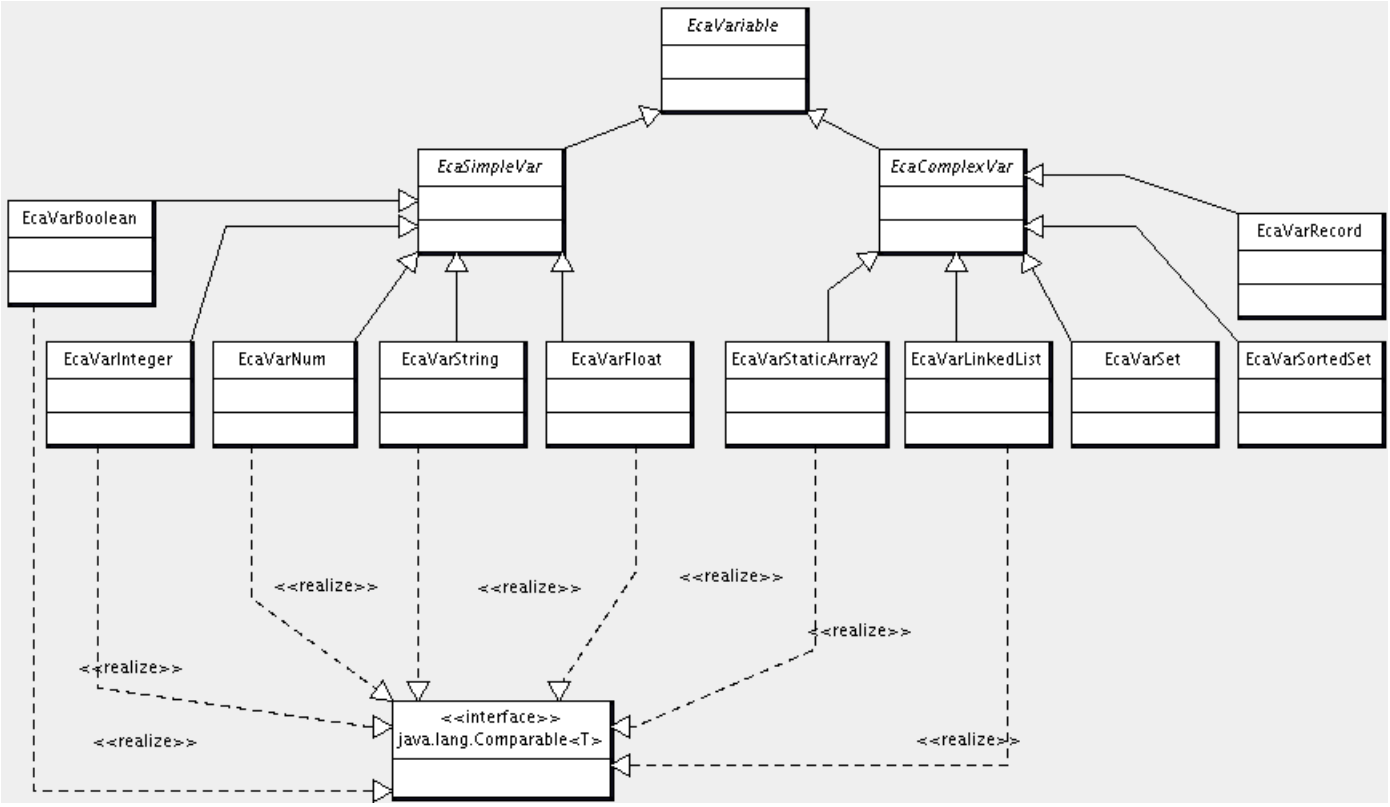

Abbildung 5.5: Hierarchie von Variablen-Klassen

- *Integer*: ganzzahlige Werte (-1,0,1,2)
- *Float*: Zahlen mit Nachkommaanteil (-0.75, 13.25)
- *Boolean*: True oder False
- *String*: Alphanumerische Zeichenfolgen
- *Array*: Indizierbare Mengen mit fester Größe
- *Sets*: Mengen eines bestimmten Typs
- *Records*: Feste Zusammenstellung anderer Datentypen
- *LinkedList*: Verkettet Listen

Details zu den einzelnen Datentypen sind der Sprachreferenz (3.1.3) zu entnehmen.

5.1.2 Ausführung einer Regel

Dieser Abschnitt wird kurz alle wesentlichen Schritte bei der Bearbeitung einer Regel durch die Laufzeitumgebung auflisten.

1. Eine Eingabe tritt auf. Eingaben sind stets Ereignisse (events).
2. Die Eingabe wird geparst und ein entsprechendes Laufzeitereignisobjekt wird generiert.

3. Das Laufzeitereignis wird in den Eventcache eingefügt.
4. Die Ereignisse (events) im Eventcache werden abgearbeitet. Diese Abarbeitung löst die zu den Ereignissen gehörenden Regeln aus.
5. Die Regeln, die den soeben ausgelösten Events entsprechen werden je nach Priorität in den Rulecache eingefügt.
6. Alle im Rulecache vorhandenen Regeln werden bearbeitet.
7. Die Bedingung (condition) einer Regel wird geprüft.
8. Ist die Bedingung erfüllt, so werden die Aktionen der Regel ausgeführt. Diese Ausführung lässt möglicherweise wieder neue Ereignisse (Events) aus, wodurch wieder andere Regeln aktiviert werden können. Das heißt, dass die Punkte 6-8 mehrfach durchgeführt werden. Bei der Aktionsausführung wird die Ausgabe des Systems produziert.
9. Nach Beendigung der Regelbearbeitung wird die Ausgabe an das externe Programm oder den Benutzer geschickt.
10. Die Invarianten werden geprüft. Wird eine Invariante verletzt, so wird die Bearbeitung abgebrochen.
11. Sind alle Regeln abgearbeitet, wartet das System auf neue Ereignisse.

5.1.3 Realisierung von externen Funktionen und externen Ereignissen

Svetla Nikolova, Christian Frost

Die Klassenstruktur für die Realisierung der externen Funktionen und der externen Ereignisse wird im Diagramm 5.6 dargestellt.

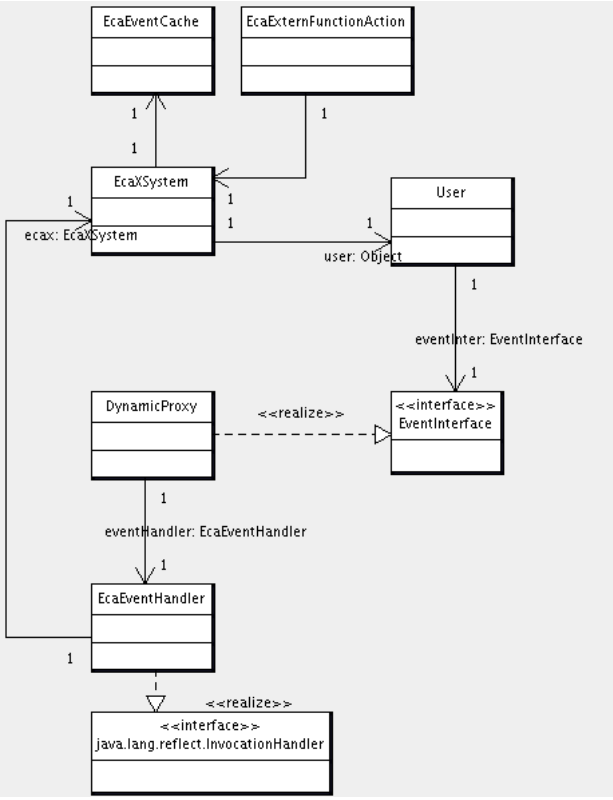

Abbildung 5.6: Realisierung von externen Funktionen und externen Ereignissen

Eine dynamische Proxy-Klasse ist eine Klasse, welche eine Liste von zur Laufzeit angegebenen Schnittstellen implementiert. Dabei wird ein Methodenaufruf durch eine der Schnittstellen auf einer Instanz der Proxy-Klasse kodiert und zu einem anderen Objekt durch ein einheitliches Interface geschickt. Methoden-Aufrufe auf einer Instanz einer dynamischen Proxy-Klasse werden zu einer einzigen Methode *invoke* im *InvocationHandler*-Objekt der Instanz abgeschickt und mit einem *java.lang.reflect.Method*-Objekt kodiert, welches die aufgerufene Methode und ein Array vom Typ *Object* mit den Argumenten ermittelt.

In Rahmen der PG LiVe dient die dynamische Proxy-Klasse als Adapter für den Aufruf von externen Ereignissen. Die Benutzer-Klasse ruft in der Proxyinstanz eine Methode auf. Die Aufrufe werden durch *EcaEventHandler* an *EcaXSystem* weitergeleitet. Durch die zugehörige *EcaEventHandler*-Instanz wird der Methodenaufruf in die Erzeugung eines Laufzeitereignisses umgewandelt, das in *EcaEventCache* geschrieben wird.

Die Typen von *User* und *EventInterface* sind für *EcaXSystem* unbekannt. *EcaXSystem* erhält ein Attribut *user* vom Typ *Object*.
Die *User*-Klasse enthält Methoden für externe Funktionen. Die Schnittstelle *EventInterface* beinhaltet Methoden für externe Ereignisse.

Bei der Registrierung von *User* bei *EcaXSystem* müssen eine dynamische Proxy-Instanz und ein zugehöriges *EcaEventHandler*-Objekt erzeugt werden.

Die Realisierung der Aufrufe der externen Funktionen erfolgt mit Hilfe der Reflection-API ([2, 3, 1]) , die Informationen über die Struktur von Java-Klassen zur Verfügung stellt. Unter dem Begriff Reflection versteht man, dass die Bestandteile von Klassen (Datenelemente, Konstruktoren und Methoden) dynamisch zur Laufzeit analysiert und benutzt werden können. Somit ermöglicht die Refection-API, zum Zeitpunkt des Kompilierens unbekannte Klassen und Interfaces (beispielsweise durch direkte Eingaben des Benutzers) zu benutzen. Folgende Funktionalitäten werden gewährleistet:

- Bestimmung der Klasse eines Objekts
- Bekommen von Informationen über Klassenmodifikatoren, Attribute, Methoden, Konstruktoren und Superklassen
- Erzeugung einer Instanz einer erst zur Laufzeit bekannten Klasse
- Erfragen und Aufrufen von Methoden
- Lesen und Setzen von Werten eines Members (Field) selbst wenn das Member namentlich erst zur Laufzeit bekannt wird
- Herausfinden welche Konstanten und Methodendeklarationen zu einem Interface gehören
- Erzeugung von Arrays, deren Größe und Typ bis zur Laufzeit nicht bekannt sind und Modifikationen an den einzelnen Array-Komponenten

Jedes mal wenn während der Laufzeit Informationen über bestimmte Klassen benötigt werden, muss immer als erstes das *Class*-Objekt für die Klasse ermittelt werden. Es gibt verschiedene Möglichkeiten, wie man an ein *Class*-Object kommt. Wenn bereits eine Instanz des unbekannten Objekts existiert kann man die Methode *getClass()* aus *java.lang.Object* aufrufen. Wenn der Klassenname zur Compilezeit zwar unbekannt ist aber zur Laufzeit sich ergibt (weil z.B. der Benutzer den Namen eingegeben hat) kann auch ein Class-Objekt erzeugt werden. Hierfür benutzt man die *forName(...)*-Methode aus *java.lang.Class*. Wenn die Klasse bekannt ist kann man hinten an den Klassennamen „.class“ anhängen, um das *Class*-Objekt zu bekommen. (Beispielsweise für einen String: *Class theClass = java.lang.String.class;*)
Mit diesem *Class*-Objekt ist es möglich alle erdenkliche Information über die Klasse zu erhalten.

Für den dynamischen Zugriff auf die Klassenbestandteile definiert die Reflection-API die Klassen *Constructor*, *Method* und *Field*. Jede dieser Klassen repräsentiert einen Klassenbestandteil der entsprechenden Art und verfügt über Methoden zum Aufrufen (*Constructor* und *Method*) bzw. zum Setzen und Auslesen (*Field*). Exemplare dieser Klassen werden nicht direkt erzeugt, sondern mit den Methoden *getConstructor()*, *getMethod()* und *getField()* der Klasse *Class* geholt. Es muss also zunächst immer das *Class*-Objekt für die Klasse ermittelt werden, auf deren Bestandteile dynamisch zugegriffen werden sollen.
Um die Modifikatoren zur Laufzeit eines Programms zu erfahren muss auf ein *Class*-Objekt die Methode *getModifiers()* aufgerufen werden

Man muss die folgenden Schritte durchführen um eine Methode mittels Reflection aufzurufen:

1. Erzeugen von einem *Class*-Objekt dessen Methode aufgerufen werden soll.
2. Erzeugen von einem *Method*-Objekt durch Aufruf von *getMethod* auf das *Class-Objekt*. Die *getMethod*-Methode besitzt zwei Parameter: Der erste Parameter vom Typ *String*, der den Namen der aufzurufenden Methode darstellt. Der zweite Parameter ist ein Array von *Class*-Objekten, welches die einzelnen Parameter der aufzurufenden Methode darstellt.

3. Die Methode wird via *invoke*-Methode aufgerufen.

Auch wenn die Reflection-API ein großes Maß an Flexibilität gewährleistet, sollte man sie nur dort einsetzen, wo es wirklich erforderlich ist. Denn dadurch bedingt, dass das Erzeugen von Exemplaren und die Aufrufe von Methoden vollig dynamisch erfolgen, kann der Compiler keine der sonst üblichen Prüfungen auf korrekte Datentypen durchführen.

Die externe Funktionen werden bei dem PG-LiVe-Projekt in beliebigen Javaklassen gespeichert, die sich beim System registriert haben. Durch die Verwendung von Reflection können die externen Funktionen in diesen Klassen gefunden und aufgerufen werden. Man kann verschiedene Klassen für externe Funktionen für die Ausgabeabstraktion beim Lernen der Beispiele verwenden.

5.2 Parsergenerierung mit ANTLR

Ahmed Alami, David Karla

Der Quellcode für ein ECA-Regelsystem liegt wie auch bei gängigen Programmiersprachen als Textdatei vor. Um das Regelsystem erfolgreich ausführen zu können, muss die Eingabedatei in einem korrekten Format vorliegen, aus dem sich problemlos Objekte der Laufzeitumgebung generieren lassen. Genauer gesagt muss die Eingabe auf syntaktische und semantische Korrektheit geprüft werden. Erst nach erfolgreicher Absolvierung dieser Prüfung können die Objekte fehlerfrei erzeugt werden. Enthält die Eingabe Fehler, so ist es wünschenswert aussagekräftige Fehlermeldungen zu generieren, die dem Benutzer möglichst große Hilfestellung zur Beseitigung der Fehler geben.

Um diese Aufgaben zu erfüllen, wird der Parsergenerator ANTLR eingesetzt [19]. Dabei handelt es sich um ein Javaprogramm, welches aus einer Beschreibung der Grammatik einer Sprache Rümpfe der benötigten Javaklassen zur syntaktischen und semantischen Prüfung generiert. Diese generierten Klassen lassen sich dann einfach durch individuellen Code ergänzen. In unserem Fall werden drei Klassen erzeugt, deren Generierung und Funktionen im Detail beschrieben werden.

5.2.1 Lexer

Die Aufgabe eines Lexers ist die Zusammensetzung der Eingabezeichen zu Token. Token entsprechen den Terminalsymbolen einer Grammatik und umfassen demnach Schlüsselwörter, Sonderzeichen (Operatoren, Semikolon, ...) und variable Zeichenketten. Damit sind etwa Ziffernfolgen gemeint, die als ein Token (etwa als Integerzahl) erkannt werden sollen. Der Parsergenerator ANTLR erhält als Eingabe eine Textdatei. In dieser werden zur Erzeugung der Lexerklasse alle Token angegeben. Ein Tokennamen wird in Großbuchstaben angegeben und der rechte Teil der Definition gibt an, welche Zeichenkette unter diesem Namen erkannt wird. Folgender Quellcodeauszug stammt aus der Lexerdefinition dieses Projektes:

```
class ECALexer extends Lexer;
options

tokens {
  EVENTS   = "Events";
  VARIABLES = "Variables";
  FUNCTIONS = "Functions";
  .
  BIGGEREQUAL = ">=";
  LESSEQUAL = "<=";
```

```
  .
  SET = "set";
  ADD = "+";
  .
  INTEGER    options {paraphrase = "integer";}    : ('-')?('0'..'9')+;
```

Die Tokennamen werden in den Beschreibungen der anderen Klassen (Parser, Treewalker) verwendet, so dass unter dem Tokennamen „ADD“ stets das „+“-Zeichen referenziert wird.

5.2.2 Parser

Aufgabe des Parsers ist es, den Tokenstream auf syntaktische Korrektheit hin zu prüfen und bei erfolgreicher Prüfung einen abstrakten Syntaxbaum der Eingabe zu generieren. Die Beschreibung einer syntaktisch korrekten Eingabe wird bei ANTLR durch Regeln angegeben, welche sehr starke Ähnlichkeit zu einer Grammatik in der erweiterten Backus-Naur-Form [29] hat. Folgender Code stammt aus der Beschreibung des Parsers dieses Projektes:

```
class ECAParser extends Parser;
.
g_vardec     : (BOOL|INT|IDENTIFIER) LBRACKET^ INTEGER RBRACKET! IDENTIFIER
             | BOOL^ IDENTIFIER
             | INT^  IDENTIFIER
             | IDENTIFIER^ IDENTIFIER
             ;
.
g_expression     : g_expr_level3 ( (AND^|OR^) g_expression)?
                 | NOT^ g_expression
                 ;
...
```

Der Code zeigt zwei Regeln der Grammatik. Die erste Regel „vardec“ beschreibt eine korrekte Variabeldeklaration für die Sprache ECAL dieses Projektes. Die groß geschriebenen Token entsprechen exakt den Token des Lexers. Des weiteren beinhalten die Regeln noch spezifische ANTLR-Informationen, die exakt angeben, wie der abstrakte Syntaxbaum aufgebaut werden soll. Das Zeichen „^“ bestimmt, dass ein Token zum Wurzelknoten eines Teilbaums des AST werden soll, das Ausrufezeichen „!“ besagt, dass ein Token kein Knoten des Syntaxbaumes werden soll. So können zum Beispiel überflüssige Information wie Klammern oder Semikolons während des Parsens vernachlässigt werden, um den Overhead möglichst gering zu halten. ANTLR generiert aus den Produktionsregeln der Grammatik eine Javaklasse für das Parsing, die für jede Regel genau eine Methode enthält, deren Rückgabewert ein abstrakter Syntaxbaum ist. Syntaxbäume von Regeln, die wiederum andere Regeln in der Ableitung verwenden, setzen sich also aus den Syntaxbäumen der verwendeten Regeln zusammen. Entsprechend repräsentiert der Baum der Startregel die gesamte Eingabe.

Der Syntaxbaum in Abbildung 5.7 entspricht folgender ECA-Regel:

```
on Temperatur (x)
if ( (x > 5) && true ) && ( MyBool == true )
do print ("Bigger");
```

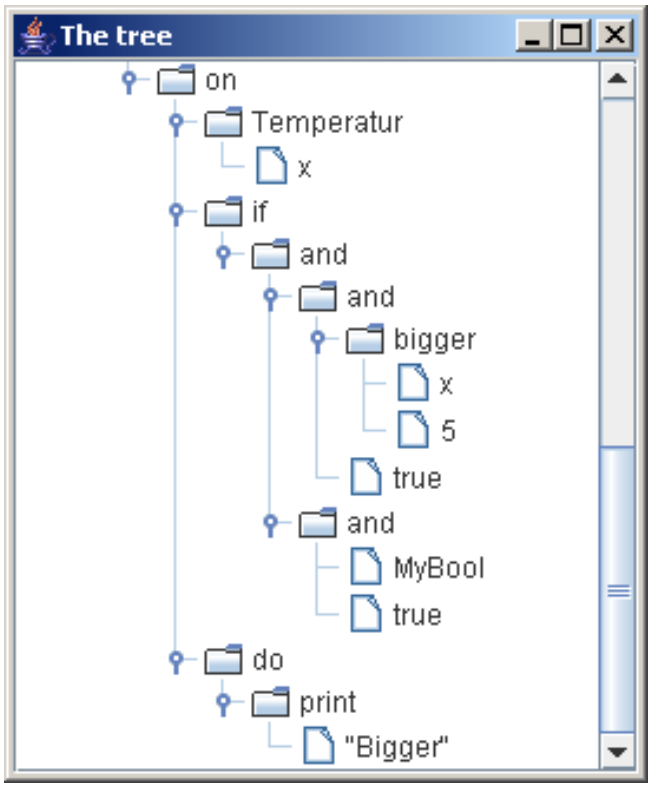

Abbildung 5.7: Teil eines abstrakten Syntaxbaumes, der eine Regeldeklaration beschreibt

Treten bei der Generierung des Syntaxbaumes Fehler auf, so kann der Parser genaue Beschreibungen des Fehlers ausgeben, da anhand der Grammatik klar ist, in welcher Regel die fehlerhafte Eingabe erkannt wurde. Weiterhin wird festgehalten, welche Stelle des Eingabedokumentes dem Fehler entspricht. Das ermöglicht aussagekräftige Meldungen wie *„File input.txt: line 10, column 13: ERROR - found „abc“ expecting integer value.“* ANTLR stellt für die Fehlerbehandlung bereits einige Exceptionklassen bereit, deren Fehlerausgabe nur noch mit Text gefüllt werden muss.

5.2.3 Treewalker

Der abstrakte Syntaxbaum, der als Ausgabe des Parsers entsteht, wird als Eingabe für eine Treewalkerklasse verwendet, deren Rumpf ebenfalls durch ANTLR aus einer entsprechenden Beschreibung generiert wird. Nachdem nun durch Lexer und Parser sichergestellt wurden, dass der Syntaxbaum korrekt ist, folgt die semantische Analyse der Eingabe. Die Sicherstellung der Typgleichheit bei Zuweisungen (x = y) ist ein typischen Beispiel für eine semantische Prüfung. Weiterhin wird geprüft, ob Variablen lokal oder global verwendet werden dürfen, Variabelnamen überhaupt oder gar mehrfach definiert werden und Operationen nur auf passenden Typen verwendet werden. Zusätzlich zur semantischen Analyse wird ein weiterer Treewalker dazu verwendet, aus dem Baum Objekte der Laufzeitumgebung zu generieren. Derzeit wird ein Treewalker verwendet, der beide Aufgaben parallel erledigt.

Die Beschreibung des Treewalker orientiert sich natürlich an der zu erwartenden Eingabe, nämlich dem abstrakten Syntaxbaum der Eingabe, welcher durch den Parser generiert wurde. Ein Regel beginnt stets mit dem in einem Baumknoten zu erwartenden Token und beschreibt dann, welche Kindknoten in diesem Teilbaum zu erwarten sind und was mit den dort enthaltenden Informationen geschehen soll. Als Beispiel wird hier die Regel für die Erkennung einer „Print“-Anweisung erläutert. Die Eingabedate enthält folgende Zeile

```
print ("Bigger");
```

Daraus generiert der Parser einen Teilbaum, dessen Wurzelknoten das Token „print“ enthält. Der einzige Kindknoten enthält den Text „Bigger“ zu der Printanweisung. Die semantische Prüfung wäre in diesem Fall, festzustellen, dass der Kindknoten eines „print“-Knoten einen String enthält. Solch einfache Prüfungen sind zusätzlich bereits durch den Parser abgedeckt. Ist die Prüfung abgeschlossen, wird aus diesen beiden Knoten ein Objekt der Klasse EcaPrintAction erzeugt, welche zur Ausgabe von Strings verwendet wird. Zusätzlich lässt sich Javacode direkt in die Treewalkerregeln integrieren. Die Regel zur Erstellung der Printaction sieht folgendermaßen aus:

```
sc_action returns [ EcaAction anAction = null ]
              :  #(PRINT  // Teilbaumwurzel -> Token PRINT
                  {
                  anAction = new EcaPrintAction ("","",); // Javacode
                  }
                  (output : STRING // Kindknoten -> enthält String
                  {
                  (  // String als Ausgabe der Action setzen - Javacode
                  (EcaPrintAction)anAction).setOutput(
                  output.getText().substring(1,output.getText().length()-1)
                  );
                  }
```

Komplexere Objekte lassen sich vom Prinzip her ähnlich erstellen. Der Treewalker ist so realisiert, dass nach einem Durchlauf durch den Syntaxbaum alle benötigten Objekte in Javalisten und Hashmaps erzeugt und unkompliziert durch die Eingabeklasse der Laufzeitumgebung entgegengenommen werden können. Zustandsvariablen, bzw. Instanzen der jeweilige Klasse werden beispielsweise durch den Treewalker in einer Hashmap gespeichert und können direkt als Startzustand der Laufzeitumgebung übernommen werden.

Abbildung 5.8 zeigt eine schamatische Darstellung der Parsergenerierung und des Parsingvorgangs.

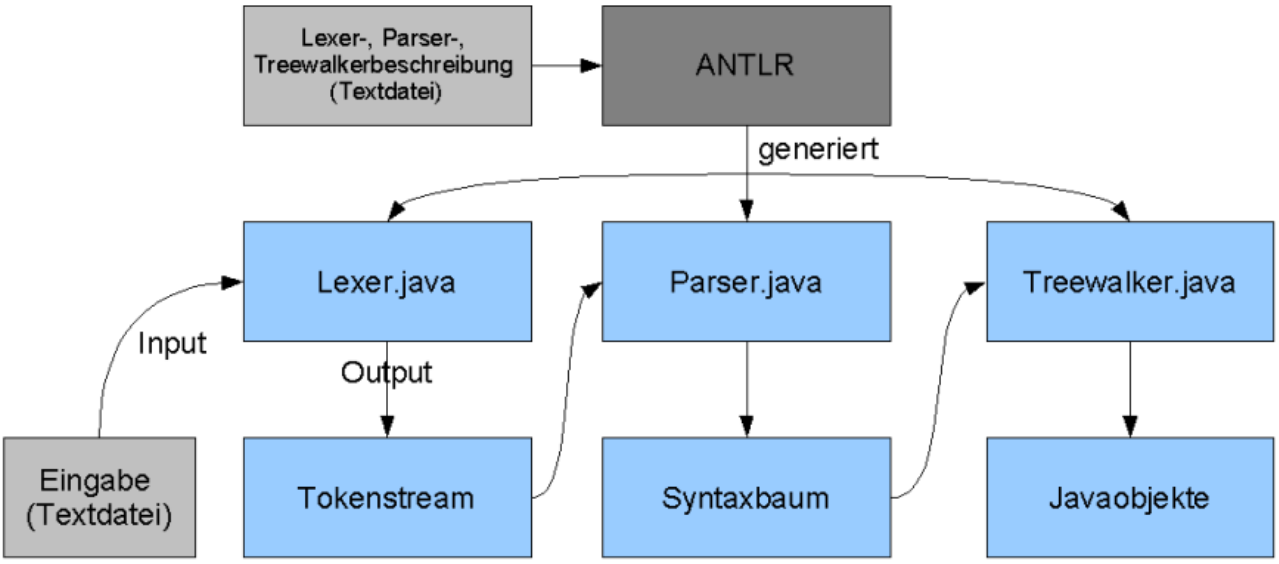

Abbildung 5.8: Lexer-, Parser-, Treewalkererzeugung und Einlesevorgang eines Regelsystems

Kapitel 6

jABC

6.1 jABC - Java Application Buildung Center

Marc Peschke

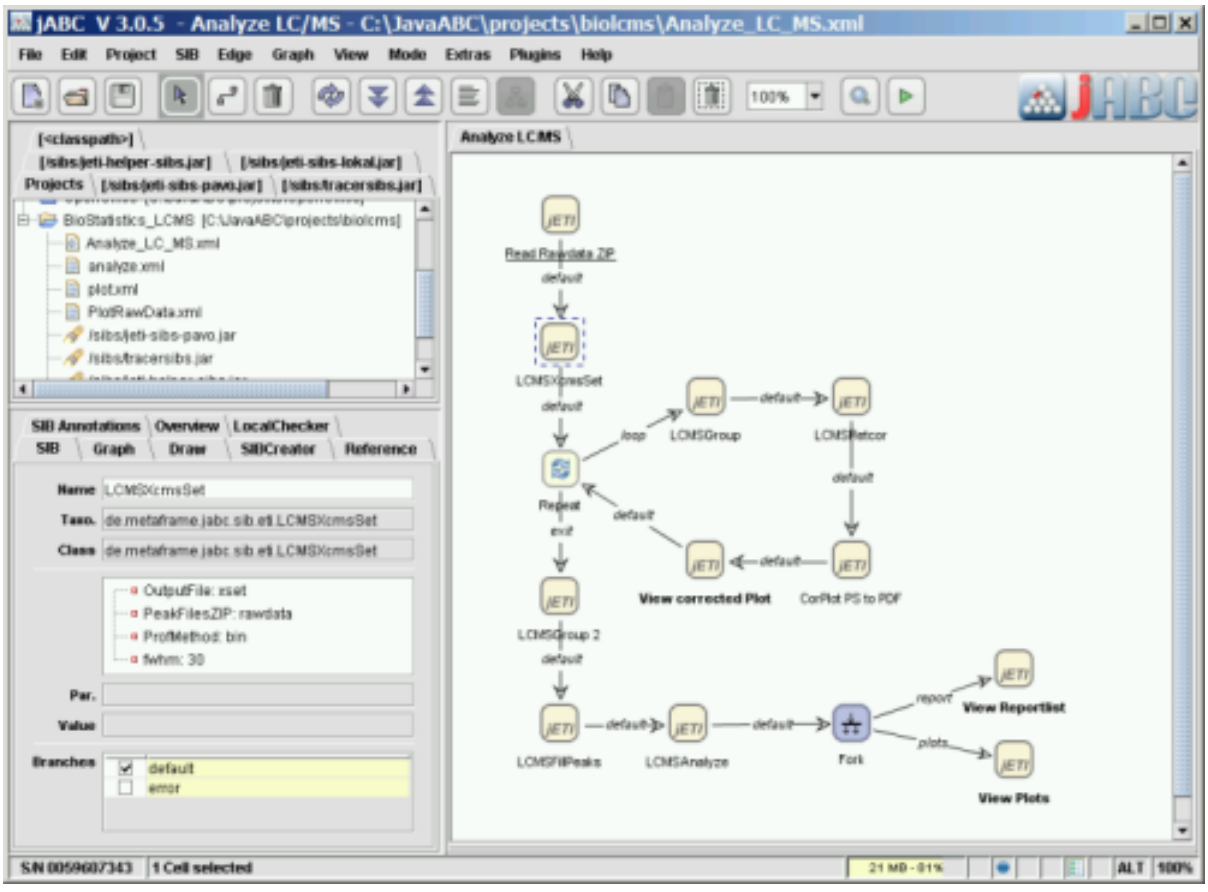

Abbildung 6.1: Das JavaABC

Das Ziel dieser Projektgruppe war es, die mit unserer Sprache umgesetzten und gelernten Automaten, mit Hilfe des jABC [18] weiter bearbeiten zu können. Darum hat man ein Plugin entwickelt, mit dem man die aus der LearnLib erhaltenen Ergebnisse, in das jABC übertragen kann. Dieses Plugin wird im nächsten Kapitel näher erläutert. Das jABC soll es jedem Anwender ermöglichen Software

zu entwickeln, auch ohne Ehrfahrungen im Programmieren zu haben. Abbildung 6.1 zeigt das jABC. Für eine genaue Erklärung der einzelnen Abschnitte und Funktionen wird auf das Manual des jABC verwiesen.

Mit dem jABC ist es möglich aus einzelnen Komponenten(SIBs) hierarchische Graphen zu entwerfen, die ein Programm repräsentieren. Diese Service Independent Building Blocks (SIBs) sind wiederverwendbare Komponenten im jABC und werden als Knoten in dem SIB-Graphen abgebildet.

Es wurde im Zusammenhang mit dem jABc ein neues Modellierungkonzept entworfen. Die Lightweight Process Coordination (LPC). Diese soll bereits bestehende Konzepte erweitern. Hauptidee dieses Konzeptes ist es, eine weitere Ebene in die bereits bekannte drei Schicht Architektur einzubauen. Diese neue Schicht ist das jABC. Letztendlich soll es dem Benutzer ermöglicht werden spielerisch Modelle und Programme aus SIBs (Knoten) und Branches (Kanten) zusammenzuklicken.

Die Arbeit mit diesem neuen Modell teilt sich in zwei Rollen, dem SIBexperte und dem Anwendungsexperten. Der Sibexperte hat genaue Kenntnis über die Entwicklung von SIBs und Plugins des JavaABC. Der Anwendungsexperte hat detailiertes Wissen über die zu implementierenden Prozesse und Anwendungen. In Abbildung 6.2 wird dieses verdeutlicht.

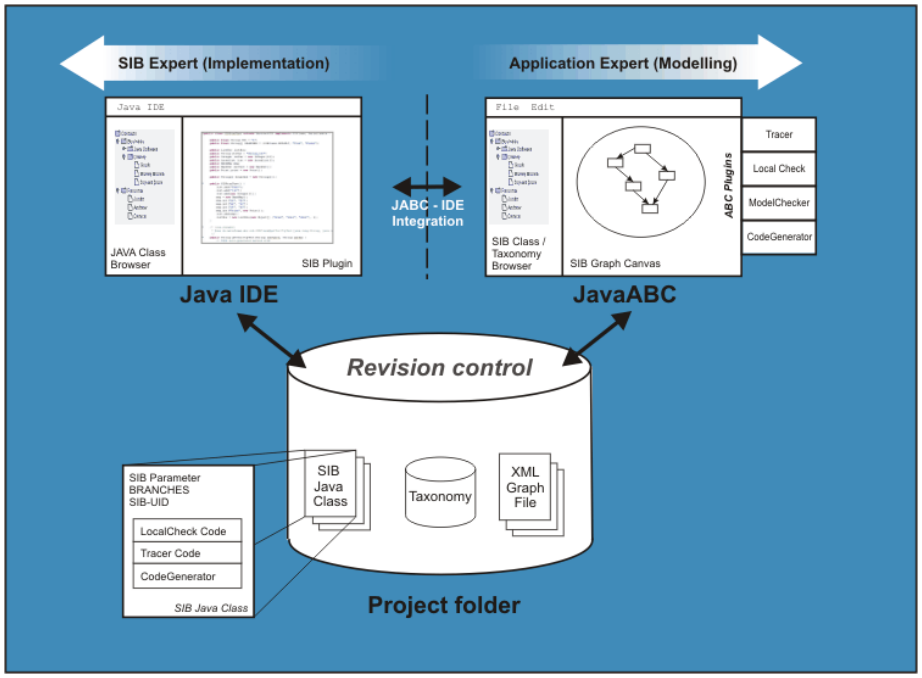

Abbildung 6.2: Das JavaABC Framework

Kapitel 7

Debugger und Plugin

7.1 Debugger

Mohsen Ahyaie

Der *ecaxDebugger* ist eine Entwicklungs- und Ausführungsumgebung für ecax-Skripte. Für den Entwurf der Skripte stellt das Programm einen Texteditor mit Syntaxhervorhebung zur Verfügung. Zusätzlich zu den Standardfunktionen wie dem Speichern oder Laden der Skripte bietet der ecaxDebugger die Möglichkeit die erstellten Skripte zu kompilieren. Werden während des Parsens oder der Kontrollflussanalyse der Skripte Fehler gefunden, so gibt das Programm aussagekräftige Fehlermeldungen mit Informationen über die Position des Fehlers im Quellcode und dessen Art.

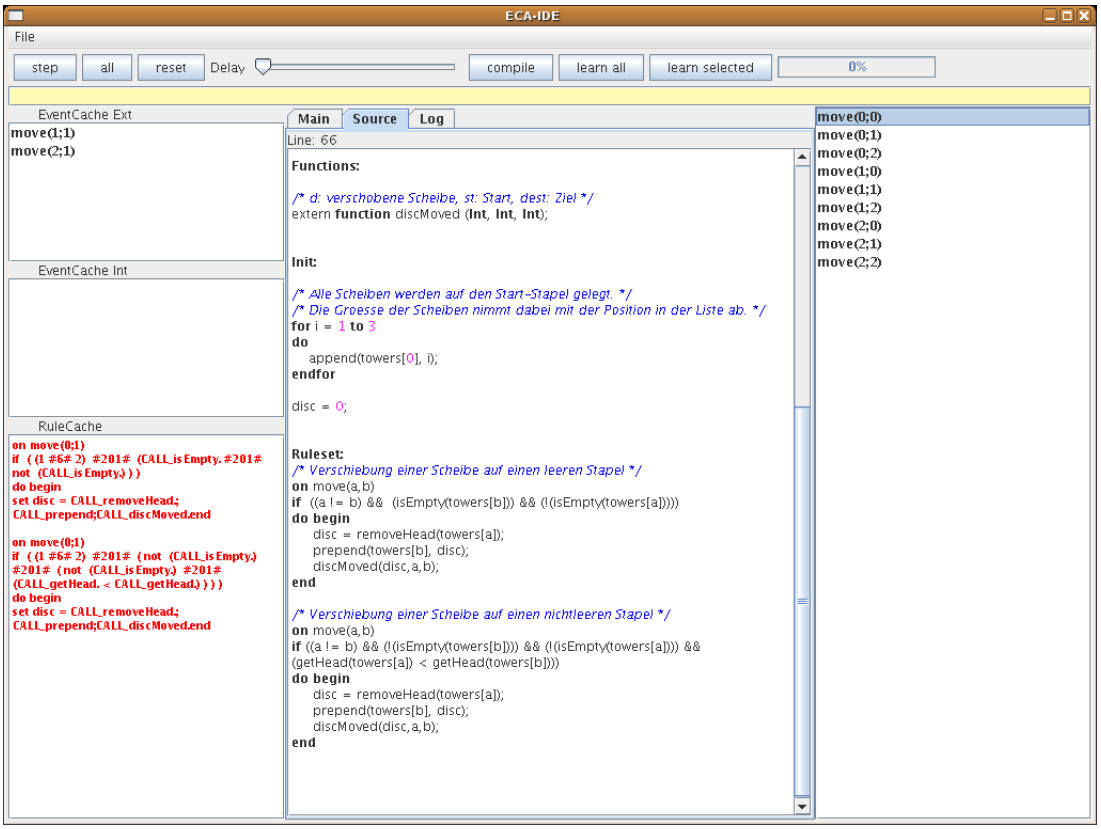

Abbildung 7.1: Der Debugger

Wird ein Skript erfolgreich kompiliert, so bietet der *ecaxDebugger* weitreichende Möglichkeiten, die Skriptausführung zu überwachen. Es werden sämtliche Bestandteile eines *ecaxSystems* dargestellt. Dadurch hat der Entwickler stets die Möglichkeit, den Systemzustand zu beobachten. Wird ein Skript kompiliert, so wird am rechten Rand des Frames eine Liste aller möglichen Events angezeigt. Hier kann der Entwickler auswählen, welche Events an das System geschickt werden sollen und die Reaktion des Systems auf diese überwachen. Es können einzelne oder mehrere Events ausgelöst werden. Diese können schrittweise oder ohne Pause ausgeführt werden. Während der Ausführung zeigt der Frame am linken Rand den Inhalt der Event- und Regelcaches an, so dass ersichtlich wird, welche Regeln ausgelöst und verarbeitet werden. Der gesamte Systemzustand, also die Belegung aller Variablen im System, kann in der Framemmitte auf einem Tabulator eingesehen werden.

Um das Verhalten des Systems als Zustandsautomaten abzubilden, kann der Entwickler eine Reihe von Events auswählen und das Systemverhalten durch die *LearnLib* lernen lassen. Der gelernte Automat wird anschließend textuell angezeigt.

7.2 Ziel des Plugins

Mohsen Ahyaie, David Karla

Das Plugin soll es dem Anwender ermöglichen, seine erstellten Regelsysteme in das *jABC* zu importieren, so dass diese dort mittels des integrierten Modelcheckers überprüft werden können. Dazu wird das Regelsystem durch die *LearnLib* als Zustandsautomat erlernt, indem die Reaktion des Systems auf das Auftreten von Events hin untersucht wird. Dieser erlernte Automat lässt sich abschließend als Graph in das *jABC* importieren, und dort zum Modelchecking verwenden. Dadurch können die gewünschten Eigenschaften des Regelsystems verifiziert bzw. widerlegt werden.

7.3 Verwendung des Plugins

Das Plugin kann aus dem *jABC* über das Menü *Plugins* gestartet werden. Die Benutzeroberfläche ist in Form eines Assistenten oder Wizards gehalten. Da für einen Lernvorgang stets die gleichen Informationen gesammelt werden müssen, wird der Anwender nacheinander zur Eingabe dieser aufgefordert. Diese Schritte werden nun im Details erläutert.

7.3.1 Laden des ECAL-Skriptes

Zunächst muss der Anwender angeben, welches Regelsystem erlernt und als Graph in das jABC exportiert werden soll (siehe Abbildung 7.2). Dazu muss ein existierendes Skript aus einer Datei geladen werden. Wurde das Skript noch nicht entworfen, so empfiehlt sich für die Erstellung die Verwendung des Debuggers. Wird das Skript korrekt geladen, so kann der nächste Schritt durchgeführt werden.

7.3.2 Auswahl der Events

Auf dem nächsten Bildschirm können die zu erlernenden Events ausgewählt werden (siehe Abbildung 7.3). D.h., dass die LearnLib das Verhalten des Systems genau auf die ausgewählten Events hin überprüft. Regelsysteme sind oftmals sehr komplex, so dass der Anwender die Möglichkeit hat, nur eine Teilmenge aller erlaubten Events lernen zu lassen. Der Bildschirm zeigt auf den linken Seite eine Liste der erlaubten Events. Diese können durch Angabe der Parameterwerte einzeln in die rechte Liste der zu verwendenden Events eingefügt werden. Dazu klickt der Benutzer doppelt auf ein Event der linken

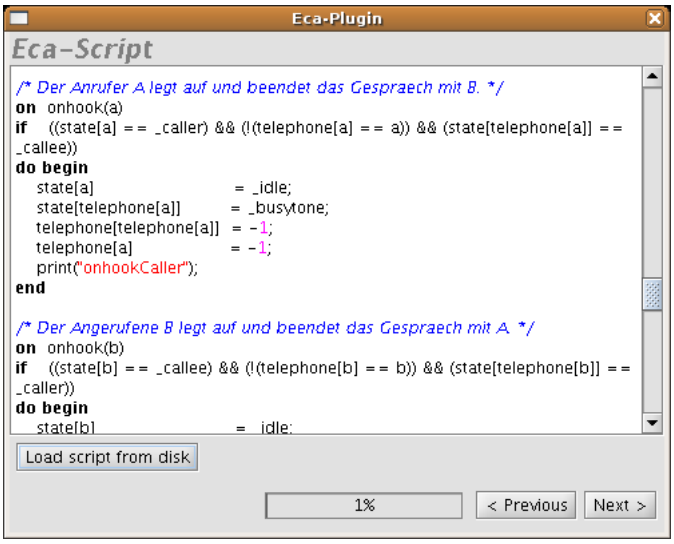

Abbildung 7.2: Laden eines ECAL-Skriptes

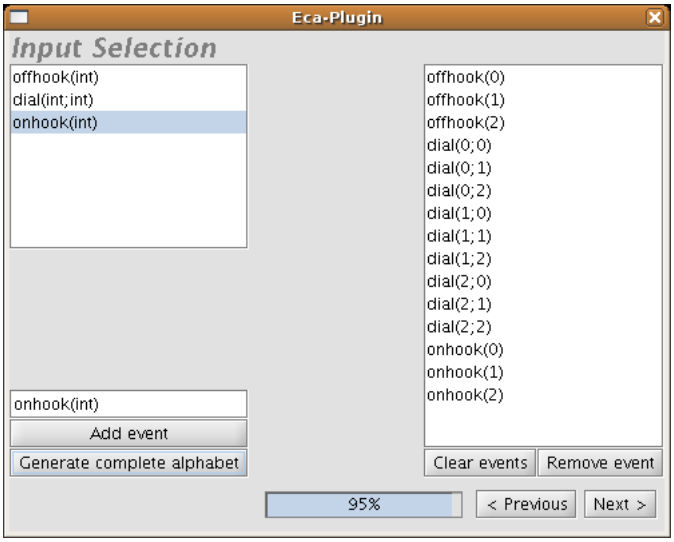

Abbildung 7.3: Auswahl der Events

Liste, gibt im darunterliegenden Textfeld die Parameterwerte an und fügt das Event mittels *Add Event* zur rechten Liste hinzu. Sollen alle erlaubten Events verwendet werden, so kann die Liste automatisch durch *Generate complete alphabet* erzeugt werden. Ist die Eventauswahl erfolgt, kann im nächsten Schritt bei der Verwendung externer Funktionen im Skript die entsprechende Java-Class-Datei ausgewählt werden, welche die Implementierung der externen Funktionen enthält.

7.3.3 Der Lernvorgang

Wird der Lernvorgang gestartet, so kann der Anwender diesen anhand von Log-Ausgaben verfolgen (siehe Abbildung 7.4). Da jeder erlernte Automat auf Vollständigkeit und Korrektheit hin untersucht werden muss, kann der gesamte Lernvorgang bei komplexen Systemen viel Zeit in Anspruch nehmen. Aus diesem Grund kann der Benutzer die Lernergebnisse auf dem nächsten Bildschirm speichern und bei späterer Verwendung des Plugins erneut laden.

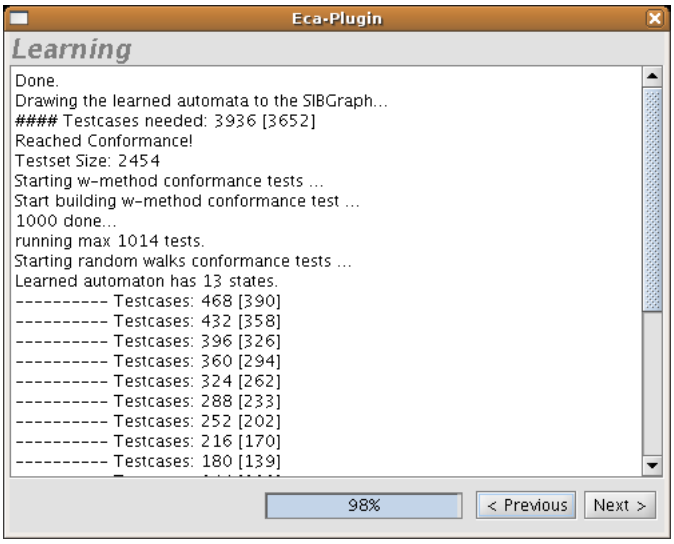

Abbildung 7.4: Log-Ausgabe während des Lernvorgangs

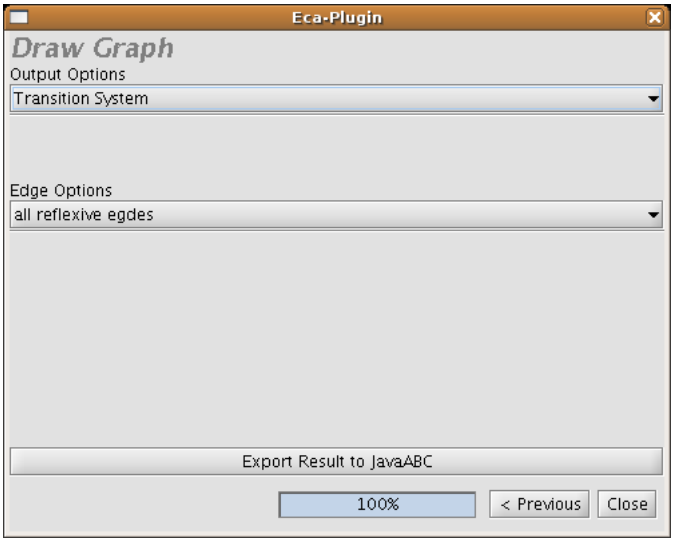

Abbildung 7.5: Export ins JavaABC

7.3.4 Erzeugung des Graphen

Abschließend bietet das Plugin die Möglichkeit, den erlernten Automaten als Graph in das JavaABC zu exportieren, damit dieser dort durch den Modelcheckers geprüft werden kann (siehe Abbildung 7.5). Der Export kann wahlweise als einfaches Transitionssystem oder auch als Kripkestruktur erzeugt werden. Weiterhin kann der Anwender wählen, ob reflexive Kanten erzeugt oder verworfen werden sollen. Der Graph wird mittels eines Layoutalgorithmus' formatiert, so dass er möglichst übersichtlich ist. Leider sind viele Systeme so komplex, dass deren Graphen trotzdem unübersichtlich erscheinen. Das beeinflusst die Ergebnisse des Modelcheckers natürlich nicht. Damit ist die Aufgabe des Plugins erledigt und die Weiterverarbeitung des Graphen im JavaABC kann beginnen.

7.4 Das Layouten des Graphen nach dem Fruchtermann Reingold Algorithmus

Jochen Gerlach

Nach dem erstellen des SIB-Graphen muß für die Darstellung auf der Arbeitsfläche das Layout festgelegt werden. Für das Layouten von Graphen besitzt das jABC bislang keine ausreichende Funktionalität. Deshalb wird für unsere Projektgruppe, das zusätzliche Einbinden eines Layouters notwendig. Eine umfangreiche Lösung zum Graphenzeichnen ist jedoch in Arbeit. Deshalb macht es in diesem Projekt wenig Sinn, eine endgültige Lösung zu entwerfen und bis ins Detail auszuarbeiten. Die vorliegende Lösung soll also nur für einen vorzeitigen Ersatz sorgen. Für das Layouten von Graphen gibt es eine Vielzahl möglicher implementierbarer Algorithmen. Für das Zeichnen von endlichen Automaten, liefert der Randomisierte-Fruchtermann-Reingold-Algorithmus(RFRA) außerordentlich gute Resultate.

7.4.1 Das Graphenproblem im jABC-Framework

SIB-Graphen sind gerichtete Graphen. Sie besitzen sowohl Beschriftungen an den Kanten wie auch an den Knoten. Für das Layouten müßte ein idealer Algorithmus folgende Eigenschaften unterstützen:

- Von einem speziellen Graphentyp(Baum, Planarer Graph, Listen, etc.) kann bei SIB-Graphen nicht ausgegangen werden. Deshalb sollte der Algorithmus möglichst allgemein sein.
- Knoten und Kanten eines SIB-Graphen können beliebig beschriftet werden. Je nach dem Layout können sich diese überlappen und machen den Graphen unästetisch und unleserlich. Gerade dieses Problem ist schwierig zu lösen. Derzeit sind uns keine Layouting Ansätze bekannt die dieses Problem lösen können.
- SIB-Graphen werden im jABC, insbesondere in diesem Projekt, auch für das Modell Checking eingesetzt. Derartige Graphen können sehr groß werden und besonders viele Kanten besitzen.

7.4.2 Der Randomisierte-Fruchtermann-Reingold-Algorithmus

Der RFRA ist ein Algorithmus, der für das Zeichnen von ungerichteten Graphen ausgelegt. Dabei arbeitet er nach einem Gravitationsmodell. Knoten mit vielen gemeinsamen Kanten ziehen sich an und liegen näher beieinander. Damit ist er jedoch eigentlich kein Algorithmus der für das Zeichnen von SIB-Graphen geschaffen ist. Zwei entgegengesetzte gerichtete Kanten mit den gleichen Knoten werden einfach übereinander gezeichnet und können bei der weiteren Arbeit gerne übersehen werden. Ein weiteres Problem des RFRA liegt darin, daß er als gewöhnlicher Graphenlayouter nicht in der Lage ist, die Beschriftungen des SIB-Graphen zu erkennen. Das bedeutet, daß die Lösung vom Benutzer nachgearbeitet werden muß. Dennoch liefern seine Resultate für die bestehende Aufgabe die besten Lösungen, wenn die Anzahl der Knoten und insbesondere der Kanten nicht zu groß ist. Der Algorithmus gibt dem Graphen durch seinen Zufallsgenerator eine ungeordnete nichtsymmetrische natürlich wirkende Form. Es kommt zu relativ wenigen überschneidungen, dadurch dass Knoten mit starken Zusammenhängen geclustert werden. Der RFRA ist bereits von einer anderen Projektgruppe Triple-A implementiert worden, und läßt sich über die Sourceforge.net Seiten beziehen. Um den Arbeitsaufwand möglichst gering zu halten wurde das Triple A Projekt als Ressource in das ECA-Plugin eingebunden, und mit geringfügigen Änderungen übernommen. Bei einer Erweiterung des jABC um Graphenzeichnern, kann das Projekt problemlos wieder entfernt werden.

Kapitel 8

Ergebnisse

8.1 Ergebnisse

Christian May, Marc Peschke

Dieses Kapitel beschreibt die erreichten Ziele der Projektgruppe.
Als dringenste Aufgabe bleibt die Implementierung aller Features, die geplant sind. Dazu gehört als erstes die Implemetierung einer lauffähigen Ausführungsumgebung für ECA-Regelsysteme. So versteht der Parser noch nicht vollständig die in Kapitel 3 beschriebene Sprachspezifikation. Variablen sind nicht vollständig implementiert so wie auch einige Actions. Dies liegt an der unvollständigen Implementierung der Treewalker bzw. des Symbolcheckers, für die im nächsten Semester viel Arbeitszeit investiert wird.
Weiterhin bleibt eine an die Sprachspezifikation angepasste Implementierung der Learnlib. Die Learnlib muss so angebunden werden, dass komplexe ECA-Regelsysteme gelernt werden können.
Als weiters Ziel ist die Kommunikation verschiedener ECA-Regelsystems miteinander zu ermöglichen. So soll ees möglich sein, daß ein ECA-Regelsystem nicht auschließlich String-Ausgaben erzeugt, sondern auch Events, die an andere Ausführungsumgebungen von weiteren ECA-Regelsysteme übergeben werden können. So könnten komplexe ECA-Regelsysteme aus einer Mengen von ECA-Regelsystemen simpel zusammengesetzt werden.
Als weiterer noch offener Punkt ist das 'Model Checking' zu nennen. Dieses Feature wurde bisher nicht behandelt. Hierzu muss eine komplette Implemtierungsidee und -strategie entwickelt werden.
Nicht so stark gewichtete Ziele, wie eine GUI, werden eingefügt, wenn noch genügend Arbeitskapazitäten zum Ende des zweiten Semesters erübrigt werden können.

8.2 Lernergebnisse

Zum Test des Automatenlernens für ECA-Regelsysteme wurden folgende Beispiele verwendet: mehrere Varianten des Telefonsystems, Blockwelt, Türme von Hanoi, Aufzug, Website und TicTacToe.

8.2.1 Telefonbeispiel

Christian Frost, Said Chihani

Die logische Beschreibung des Telefonsystems wurde im Abschnitt über Feature Interaction (2.3) vorgestellt. Im Anhang findet man die ECAL-Formulierung des Telefonbasissystems.

Es wurden vier verschiedene Versionen des Telefonbeispiels beim Automatenlernen untersucht, nämlich das Telefonbasissystem, die Integration des Dienstes Anrufweiterleitung (CFB), die Integration des Dienstes INTL (Sperrung ausgehender Anrufe) und die Integration des Dienstes TCS (Abschirmung eingehender Anrufe) in das Basissystem.

Die Beispiele wurden normalerweise für die Anzahlen von zwei und drei Telefongeräten untersucht. Bei der Integration des Dienstes CFB (Anrufweiterleitung) ist dagegen nur die Untersuchung des Problems mit drei Telefonendgeräten sinnvoll.

Außerdem wurden drei verschiedene Arten der Ausgabeabstraktion verwendet. Bei Verwendung der vollständigen Information werden der LearnLib für die ausgelösten Regeln sowohl eindeutige Regelnamen als auch die Indizes der beteiligten Telefongeräte als Parameter übergeben.

Eine weitere Möglichkeit ist es, der LearnLib nur eindeutige Regelnamen als Ausgaben mitzuteilen.

Bei den durchgeführten Tests unterscheiden sich die Zustandsanzahlen der gelernten Mealy-Automaten im Falle der beiden bereits vorgestellten Abstraktionsgrade nicht. Dies kann dadurch begründet sein, dass die Parameter der Regeln immer auch die Parameter der Ereignisse umfassen. Die Information über die Ereignisse wird von der LearnLib nämlich in jedem Fall berücksichtigt.

Der höchste Abstraktionsgrad wird erreicht, wenn der LearnLib bei jeder auftretenden Regel die gleiche Ausgabe übergeben wird. In diesem Fall reduziert sich die Zustandsanzahl im gelernten Automaten gegenüber den beiden vorherigen Abstraktionsgraden.

Telefonbasissystem

Für das Telefonbasissystem ergeben sich die folgenden Resultate.

Zustandsanzahlen der Modelle für das Basissystem mit zwei Telefonen:

- 13 Zustände (vollständige Information)
- 13 Zustände (nur Regelnamen als Ausgabe)
- 11 Zustände (größtmögliche Abstraktion)

Die Unterschiede in der Zustandsanzahl können bei diesem kleinen Modell noch gut erklärt werden. Wenn sich die Ausgaben für die unterschiedlichen Regeln nicht unterscheiden, können drei verschiedene Situationen nicht voneinander unterschieden werden. Deshalb verringert sich die Anzahl der Zustände von dreizehn auf elf.

Wenn die Benutzer der beiden Telefone miteinander sprechen, kann ohne Ausgabe eines eindeutigen Regelnamens nicht unterschieden werden, wer von beiden angerufen hat. Diese Situation kann in diesem Fall auch nicht von der Konstellation unterschieden werden, in der beide Teilnehmer das Besetztzeichen hören.

In allen drei Fällen ist es nämlich so, dass die einzigen Transitionen, die in diesem Zustand möglich sind, durch das Auflegen des Hörers durch einen Teilnehmer ausgelöst werden. Das zugehörige Telefon geht dann in den Ruhezustand über. Der andere Gesprächsteilnehmer hört das Besetztzeichen, weil das Gespräch beendet wurde oder weil sich das Telefongerät bereits im Zustand „besetzt“ befand.

Der Graph des gelernten Automaten für zwei Telefongeräte bei Verwendung vollständiger Information ist in der Graphik 8.1 abgebildet. Dabei ist q0 der Startzustand.

Zustandsanzahlen der Modelle für das Basissystem mit drei Telefonen:

- 63 Zustände (vollständige Information)
- 63 Zustände (nur Regelnamen als Ausgabe)
- 45 Zustände (größtmögliche Abstraktion)

Bei einer Anzahl von drei Telefongeräten erhöhen sich die Anzahlen der Zustände in den gelernten Mealy-Automaten. Gegenüber der Situation von zwei Telefongeräten hat sich die Zustandsanzahl im vollständigen Modell ungefähr verfünffacht, während bei größtmöglicher Abstraktion nur eine Vervierfachung auftritt. Die Anzahl der Zustände wächst also viel schneller als die Anzahl der Telefongeräte.

Telefonbasissystem mit Integration des Dienstes INTL

In diesem Abschnitt werden die Lernergebnisse für das Telefonbassissystem mit Integration des Dienstes INTL vorgestellt. Ein Abonnent des Dienstes INTL (Sperrung ausgehender Anrufe) kann nur telefonieren, wenn er zuvor eine gültige PIN eingegeben hat. Eine notwendige Bedingung für diese Systemeigenschaft konnte in jABC durch das Model Checking-Plugin GEAR überprüft werden.

Zustandsanzahlen der Modelle für das Basissystem mit Integration des Dienstes INTL im Falle von zwei Telefongeräten (ein Abonnent des Dienstes INTL):

- 19 Zustände (vollständige Information)
- 19 Zustände (nur Regelnamen als Ausgabe)
- 14 Zustände (größtmögliche Abstraktion)

Die Zustandsanzahlen in den gelernten Automaten sind größer als beim einfachen Telefonbasissystem. Dies ist auch einleuchtend, weil ein Telefongerät beim Abonnement des Dienstes INTL zusätzlich zu den Zuständen im Basissystem auf die Eingabe einer PIN warten kann und bei der Eingabe einer ungültigen PIN in den Status „ungültig“ gelangen kann.

Für das Model Checking wurde eine notwendige Bedingung für die folgende Anforderung als CTL-Formel formuliert:
Ein Abonnent des Dienstes INTL muss vor dem Wählen einer Telefonnummer eine gültige PIN eingeben. In unserem Fall ist Gerät 1 Abonnent dieses Dienstes.

Die entsprechende CTL-Formel hat in GEAR-Syntax die Form:
(IMP (SIBEXP name == q0) (AWU (NOT dial[1;0]) dialgoodpin[1])).

IMP bezeichnet hierbei die logische Implikation, während WU „weak until“ bedeutet. Die Formel besagt, dass ausgehend vom Startzustand q0 auf jedem Pfad in der Kripkestruktur des gelernten Modells dial[1;0] nie vor dem Eintreten des Ereignisses dialgoodpin[1] gelten kann.
Der Benutzer von Telefongerät 1 kann also erst die Telefonnummer wählen, wenn er eine gültige PIN eingegeben hat.

Ebenso wird eine Formel für das Ereignis dial[1;1] aufgestellt. Im Beispiel mit 3 Telefonen kommt darüber hinaus eine Formel für das Ereignis dial[1;2] hinzu, da es einen weiteren Teilnehmer im Telefonsystem gibt.

Die Formel wird nur vom Startzustand q0 ausgehend überprüft, weil es sonst sein könnte, dass vor dem Erreichen des untersuchten Zustandes schon eine gültige PIN eingegeben wurde. Der Model Checker würde dies nicht feststellen und melden, dass die Formel im angegebenen Zustand nicht gilt.

Die untersuchte Formel ist nur eine notwendige Bedingung für die Anforderung, weil z.B. nicht überprüft wird, ob das Telefon in der Zeit zwischen der Eingabe der PIN und dem Wählen der Telefonnummer in den Ruhezustand gewechselt ist. In diesem Fall müsste nämlich erneut eine PIN eingegeben werden, was hier aber nicht überprüft wird.

Die oben angegebene Formel gilt für die gelernten Automaten für alle drei untersuchten Abstraktionsgrade. Die Formel verwendet nämlich nur die regelauslösenden Ereignisse als atomare Propositionen, welche die LearnLib bei allen Abstraktionsgraden berücksichtigt. Trotzdem ist das Ergebnis nicht offensichtlich, weil die Zustandsanzahlen der Automaten unterschiedlich sind.

Zustandsanzahlen der Modelle für das Basissystem mit Integration des Dienstes INTL im Falle von drei Telefongeräten (ein Abonnent des Dienstes INTL):

- 89 Zustände (vollständige Information)
- 89 Zustände (nur Regelnamen als Ausgabe)
- 56 Zustände (größtmögliche Abstraktion)

Bei der Erhöhung der Anzahl der Telefongeräte auf drei verfünffacht sich die Zustandsanzahl bei vollständiger Information, während sie sich bei größtmöglicher Abstraktion nur vervierfacht. Dies stimmt mit den Beobachtungen für das Basissystem überein.

Die notwendige Bedingung für die gewünschte Systemeigenschaft, dass ein Abonnent des Dienstes INTL nur nach Eingabe einer gültigen PIN anrufen kann, kann auch hier für alle drei Abstraktionsgrade durch Model Checking nachgewiesen werden.

Telefonbasissystem mit Integration des Dienstes TCS

Das Telefonbasissystem wird nun um den Dienst TCS (Sperrung eingehender Anrufe) erweitert. Ein Abonnent dieses Dienstes kann andere Telefongeräte auf die schwarze Liste setzen. Die Anrufe von diesen Telefonen werden nicht angenommen sondern blockiert, die Anrufer hören deshalb das Besetztzeichen.

Im untersuchten Beispiel hat das Telefongerät 1 das Gerät 2 auf die schwarze Liste gesetzt. Im Fall von nur zwei Telefongeräten haben diese hier die Indizes 1 und 2, während im Fall von drei Telefonen die Indizes 0, 1 und 2 verwendet werden.

Zustandsanzahlen der Modelle für das Basissystem mit Integration des Dienstes TCS im Falle von zwei Telefongeräten (ein Abonnent des Dienstes TCS):

- 11 Zustände (vollständige Information)
- 11 Zustände (nur Regelnamen als Ausgabe)
- 10 Zustände (größtmögliche Abstraktion)

Gegenüber dem ursprünglichen Modell für das Telefonbasissystem verringert sich hier die Anzahl der Zustände, weil es nicht möglich ist, dass Teilnehmer 2 als Anrufer mit Teilnehmer 1 sprechen kann.

Die gewünschte Systemeigenschaft, dass ein Anruf von 2 nie vom Gerät 1 angenommen wird, kann durch relativ einfache CTL-Formeln ausgedrückt werden.

Im Fall der vollständigen Information ist dies besonders leicht. Es gibt dort die Regelnamen dialBusy für die Wahl einer Telefonnummer bei besetzter Leitung, dialBlock für einen blockierten Anruf und dial_Idle für einen Anruf bei freier Leitung, der angenommen wird.

Wenn Gerät 2 auf der schwarzen Liste von 1 steht, dann kann niemals dial_Idle[2,1] gelten. Die entsprechende einfache CTL-Formel lautet in GEAR-Syntax:
(NOT dial_Idle[2,1]).

Bei der Abstraktionsstufe, in der nur die Regelnamen ohne Parameter als Ausgabe verwendet werden, ist diese einfache Überprüfung nicht möglich.

Eine Formel, die in beiden Abstraktionsgraden überprüft werden kann, besagt, dass für jeden Zustand, in dem Telefon 2 bei Gerät 1 anruft (Ereignis dial[2;1]), die Leitung besetzt ist (Regel dialBusy) oder der Anruf abgeblockt wird (Regel dialBlock).

Die verwendete Operatorkombination AX sagt aus, dass eine Formel in allen direkten Nachfolgezuständen des untersuchten Zustands gelten muss.

Weil die Regeln, die durch ein Ereignis ausgelöst werden, in der Kripkestruktur im Nachfolgezustand des Ereignisses gespeichert werden, erhält man also die folgende CTL-Formel:

(IMP dial[2;1] (AX (OR dialBusy[2,1] dialBlock[2,1])))
(IMP dial[2;1] (AX (OR dialBusy dialBlock)))

Die erste Variante bezieht sich auf den Fall vollständiger Information, weil dort die Regelnamen nicht ohne Angabe der Parameter auftreten.
Die zweite Variante wird für den Fall verwendet, in dem nur die Regelnamen als Ausgabe an die LearnLib dienen.

Die gewünschte Systemeigenschaft konnte im Fall der vollständigen Information sowohl durch die einfache als auch durch die allgemeiner verwendbare Formel nachgewiesen werden.

Im Fall, in dem nur die Regelnamen als Ausgaben eingesetzt werden, konnte die Gültigkeit der allgemeineren Formel bewiesen werden.

Bei größtmöglicher Abstraktion kann die gewünschte Systemeigenschaft nicht als temporallogische Formel ausgedrückt werden. Die einzigen Informationen in der Kripkestruktur sind hier nämlich die Ereignisse (dial, offhook und onhook), weil die Ausgaben für die Transitionen alle gleich sind. Allein durch die auftretenden Ereignisse kann aber z.B. nicht festgestellt werden, ob ein Anruf abgeblockt wurde oder nicht.

Zustandsanzahlen der Modelle für das Basissystem mit Integration des Dienstes TCS im Falle von drei Telefongeräten (ein Abonnent des Dienstes TCS):

- 57 Zustände (vollständige Information)
- 57 Zustände (nur Regelnamen als Ausgabe)
- 42 Zustände (größtmögliche Abstraktion)

Bei einer Erhöhung der Anzahl der Telefongeräte auf drei vergrößert sich die Zustandsanzahl bei vollständiger Information wie schon in den anderen Varianten des Telefonbeispiels um den Faktor fünf, während sie sich bei maximal möglicher Abstraktion wiederum vervierfacht.

Auch in dieser Konstellation wurde die Bedingung überprüft, dass Anrufe von Gerät 2 an Telefon 1 vom letzteren nie angenommen werden. Der Model Checker hat die Gültigkeit dieser Bedingung auch für die Anzahl von drei Telefonen in den beiden Abstraktionsgraden „vollständige Information“ und „Regelnamen als Ausgabe“ nachgewiesen.

Telefonbasissystem mit Integration des Dienstes CFB

Der Dienst CFB stellt eine Anrufweiterleitung bei besetzter Leitung dar. Ein Abonnent dieses Dienstes kann genau ein Telefongerät angeben, an das die eingehenden Anrufe weitergeleitet werden, wenn die eigene Leitung besetzt ist.

Eine erfolgreiche Anrufweiterleitung ist nur in einem System mit mindestens drei Telefongeräten möglich. Ein Anruf von A an B kann bei besetzter Leitung von B an das Gerät C weitergeleitet werden. Hierbei müssen die Geräte A, B und C paarweise verschieden sein.

Im untersuchten Beispiel haben die Telefongeräte die Indizes 0, 1 und 2. Gerät 1 hat dabei eine Anrufweiterleitung an Telefongerät 2 abonniert.

Zustandsanzahlen der Modelle für das Basissystem mit Integration des Dienstes CFB im Falle von drei Telefongeräten (ein Abonnent des Dienstes CFB):

- 63 Zustände (vollständige Information)
- 63 Zustände (nur Regelnamen als Ausgabe)
- 45 Zustände (größtmögliche Abstraktion)

Man stellt fest, dass die Zustandsanzahlen der gelernten Mealy-Automaten den entsprechenden Anzahlen in den Modellen für das Telefonbasissystem entsprechen.

Durch die Integration des Dienstes Anrufweiterleitung ändert sich also die Anzahl der Systemzustände nicht. Der einzige Unterschied wird also in den möglichen Transitionen bestehen.

Im Fall der vollständigen Information als Ausgabe an die LearnLib kann man durch die beiden folgenden CTL-Formeln bestätigen, dass sich die Zustandsübergänge unterscheiden.

Im Telefonbasissystem gilt die Formel:
(IMP dial[0;1] (AX (OR dial_Idle[0,1] dialBusy[0,1])))

Im Telefonbasissystem gibt es nämlich bei einem Anruf von Telefon 0 an Telefongerät 1 (Ereignis dial[0;1]) die beiden Möglichkeiten, dass die Leitung von 1 frei (Regel dial_Idle[0,1]) oder besetzt ist (Regel dialBusy[0,1]). In jedem direkten Nachfolgerzustand des Ereignisses muss darum eine der beiden genannten Regeln gelten.

Im Falle des Abonnements der Anrufweiterleitung von Gerät 1 an Telefon 2 gilt:
(IMP dial[0;1] (AX (OR dial_Idle[0,1] dialCfbIdle[0,1,2] dialCfbBusy[0,1,2])))

Beim Abonnement der Anrufweiterleitung gibt es dagegen genau drei Möglichkeiten. Wie im Basissystem kann die Leitung von Gerät 1 frei sein (Regel dial_Idle[0,1]). Bei besetzter Leitung gibt es dagegen die beiden Situationen, dass die Leitung des Gerätes 2, an das der Anruf weitergeleitet werden soll, frei ist (Regel dialCfbIdle[0,1,2]) oder ebenfalls besetzt ist (Regel dialCfbBusy[0,1,2]).

Durch Model Checking mit GEAR konnte dieser Unterschied zwischen den beiden Systemen nachgewiesen werden.

8.2.2 Blockwelt

Christian Frost, Said Chihani

Das Beispiel Blockwelt wurde in der Forschung für Planungsprobleme untersucht. In der Projektgruppe Live wurde eine einfache Version dieses Problems verwendet, in der die Bausteine nicht nach Farbe und Form unterschieden werden.

Im Anfangszustand liegen immer alle Blöcke auf dem Tisch. Es gibt drei mögliche Operationen zur Verschiebung der Bausteine, die in der Sprache ECAL als Ereignisse formuliert werden. Die drei Ereignisse haben folgende Bedeutungen:

- stack(x,y): Wenn Baustein x allein auf dem Tisch liegt und über Block y kein weiterer Stein liegt, dann kann x durch die Operation stack(x,y) auf y gestapelt werden.
- unstack(x,y): Wenn Block x auf Block y liegt und der Raum über x frei ist, so kann x durch das Ereignis unstack(x,y) von y heruntergenommen und auf den Tisch gelegt werden.
- move(x,y,z): In der Situation, in der Baustein x auf Block y liegt und der Raum über Block x und über Block z frei ist, kann Stein x von Block y heruntergenommen werden und auf Baustein z gelegt werden.

Für den Test des Automatenlernens konnte das Beispiel nur mit einer Problemgröße von drei Blöcken gelernt werden.

In der Abbildung „Lernergebnis für das Beispiel Blockwelt“ (8.2) ist der Automat zu sehen, der durch die LearnLib bei geringster Abstraktion erlernt wurde.

Die Kantenbeschriftungen bestehen jeweils aus zwei Teilen, die durch Schrägstriche voneinander getrennt sind. Die erste Beschriftung gibt das Ereignis an, das den Zustandsübergang auslöst. Als zweite Beschriftung ist die Ausgabe des ECA-Regelsystems an die LearnLib zu sehen.

Im Fall der vollständigen Information wurde für jede Regel ein eindeutiger Name vergeben. Zusätzlich werden die Regeln auch noch durch ihre Parameter unterschieden, die hier den Parametern der Ereignisse entsprechen.

Die dreizehn Zustände des erlernten Automaten kann man in den Startzustand und zwei weitere Arten von Zuständen einteilen. Zu diesen beiden Zustandstypen gehören jeweils sechs Zustände.

Der Startzustand q0 ist in der Mitte der Graphik zu sehen. In diesem Zustand liegen alle Blöcke einzeln auf dem Tisch.

Durch das Stapeln eines Bausteines auf einen anderen (Operation: stack) können sechs verschiedene Zustände erreicht werden. Zwischen jeweils zwei dieser Zustände kann das System durch das Ereignis move wechseln, durch das der oberste Block von einem Stapel auf den anderen verschoben wird.

Stapelt man dagegen den einzigen noch einzeln auf dem Tisch liegenden Baustein auf die beiden übrigen, so entsteht ein Turm aus drei Zuständen. Diese Türme liegen in den sechs äußeren Zuständen vor. Nur durch Rückgängigmachen der Operation stack durch das Ereignis unstack mit den gleichen Parametern kann man diese Zustände wieder verlassen.

Die Richtung der Zustandsübergänge ist in der graphischen Darstellung nicht erkennbar, weil die jeweiligen Kanten übereinander liegen.

Andererseits erkennt man im Graphen drei gleichartige Gruppen aus jeweils vier Zuständen. Diese unterscheiden sich nur durch den Baustein, der zuerst auf einen der beiden anderen Blöcke gestapelt wurde.

Eine Möglichkeit der Abstraktion ist es, der LearnLib nur die Regelnamen ohne Parameter als Ausgabe mitzuteilen. Der gelernte Automat ändert sich aber dadurch nicht, weil die Regelparameter auch schon in der Eingabe als Ereignisparameter enthalten sind.

Selbst bei maximaler Abstraktion, bei der für jede Regel die gleiche Ausgabe erzeugt wird, kann ein Automat mit 13 Zuständen und gleichem Verhalten wie bei der vollständigen Information gelernt werden. Die Struktur des Problems Blockwelt ist also besonders einfach.

8.2.3 Beispiel: Die Türme von Hanoi

Said Chihani, Christian Frost

Bei den Türmen von Hanoi handelt es sich um ein klassisches Problem mit einem rekursiven Lösungsansatz.
Gegeben sind drei Stangen. Auf der ersten Stange ist eine Anzahl von Scheiben aufgereiht, nach Größe sortiert.
Eine Stange wird in der ECAL-Beschreibung des Hanoi-Beispiels als eine Liste aus ganzen Zahlen dargestellt. Die Scheiben werden gemäß ihrer Ordnungszahlen aufsteigend in der Liste gespeichert.
Das Beispiel hat ein einziges Ereignis:

- move(Int[0 to 2], Int[0 to 2]);

Das Ereignis „move(a,b)“ bedeutet, dass die kleinste Scheibe auf der Stange a, die dort am obersten Platz liegt, zur Stange b verschoben wird. Aufgabenstellung sei es, die Scheiben von der ersten auf die dritte Stange zu versetzen. Die zweite Stange wird als ein Hilfsablageplatz verwendet.
Um das Ganze nicht allzu einfach zu machen, sind einige Grundregeln für das Verschieben gegeben:

- Es darf pro Zug immer nur eine Scheibe bewegt werden.
- Eine größere Scheibe darf niemals auf eine kleinere Scheibe gelegt werden.

Die ideale Lösung von der Verschiebung von einer Stange zu einer anderen enthält sieben Schritte. Das System lernt aber alle Möglichkeiten, die für dieses Problem vorhanden sind. Zum Beispiel berechnet das System für nur drei Scheiben und drei Stangen genau siebenundzwanzig Zustände.

Am Anfang wird das Beispiel „Hanoi-Beispiel“ von dem System gelernt. Als Ergebnis wird der Graph in der Abbildung 8.3 mit genau siebenundzwanzig Zuständen erzeugt, die in Gruppen verteilt sind. Jede Gruppe enthält genau drei Zustände und innerhalb jeder Gruppe wird die kleinste Scheibe (Index 1) verschoben. Bei der Verschiebung einer größeren Scheibe wird der Pfeil zu einer anderen Gruppe verweisen. Die gleichen Aktionen werden jedes mal aufgerufen, bis alle möglichen Zustände durchsucht worden sind.
In diesem gelernten Graph kann man auch den Zielzustand (q24, ganz rechts) in sieben Schritten ereichen, indem man vom Startzustand (q0, links unten) den kürzesten Weg berechnet. Das wird auch erreicht, wenn man die sieben Kanten vom Start bis zum Ziel weiterverfolgt. Dieser Weg enthält die Zustände q0, q2, q4, q8, q10, q13, q17 und q24.
Die Zustandsanzahl siebenundzwanzig, die das System gelernt hat, kann man auch berechnen, wenn man die Permutationen aufzählt. In dem Beispiel sind dies genau 3^3 . Allgemein kann man die Anzahl der möglichen Zustände bestimmen, wenn man m^n berechnet, wobei m die Anzahl der Türme und n die Anzahl der Scheiben ist.
Zum Beispiel enthält der gelernte Automat für drei Stangen und vier Scheiben genau einundachtzig (3^4) Zustände. Der Lernaufwand für diese Beispielgröße war mit 11.574.772 Testfällen aber sehr viel größer als im Fall von nur drei Scheiben.

8.2.4 TicTacToe

Falk Howar

Das Beispiel beschreibt das TicTacToe-Spiel. TicTacToe ist ein Spiel auf einem 3x3-Brett für zwei Spieler. Die Spieler markieren abwechselnd ein unmarkiertes Feld. Gewonnen hat der Spieler, dem es gelingt, drei Felder in einer Reihe oder Diagonale zu markieren.

Prinzipiell beschreibt das Beispiel TicTacToe vollständig. Da der resultierende Automat allerdings sehr groß (mehrere hundert Zustände) wäre, wurde das Spielfeld, wie in Abbildung 8.4 gezeigt, vorbelegt. Spieler 1 (O) hat bereits drei Züge gemacht, Spieler 2 (X) hat zwei Züge gemacht und ist am Zug.

Das Beispiel hat zwei Ereignisse:

- einSetzeZeichen(Int [1 to 9])
- zweiSetzeZeichen(Int [1 to 9])

Das Beispiel wurde, um den Lernaufwand zu verringern nur mit den noch zültigen Zugmöglichkeiten gelernt. Das Lernen ohne Eingabenabstraktion würde das Ergebnis, bis auf reflexive Kanten, nicht verändern. Das TicTacToe-Script bildet ungültige Züge durch reflexive Kanten ab: es geschieht einfach keine Änderung des Zustands. Da es auch sonst keine reflexiven Kanten mit Bedeutug gibt, wurde das gelernte Ergebnis ohne die reflexiven Kanten ins JavaABC exportiert.

Für das Beispiel existiert eine Abstraktion der Ausgabe, die darin besteht, dass nur der Sieg eines der beiden Spieler oder ein Unentschieden angezeigt werden. Die nicht abstrahierte Version gibt nach jedem Zug aus, ob bzw. dass einer der Spieler ein Feld markiert. Da die nicht abstrahierte Version so leicht zu lernen ist, dass nur ein einziger Conformance-Test am Schluss durchgeführt wird, wurde mit der abstrakteren Version gearbeitet. Die gelernten Automaten unterscheiden sich jedoch nicht.

Auf dem gelernten Automaten wurde überprüft, welchen Spielzug Spieler 2 machen kann, um nicht zu verlieren. Schaut man sich Abbildung 8.4 an, wird klar, dass er unbedingt unten in der Mitte spielen muss. Im Beispiel ist das Feld Nr. 8. Die entsprechende Formel lautet:

(AND (NOT (OR eins (EX eins)))(BOXB start) (NOT start))

Die einzelnen Teile haben folgende Bedeutung:

- (NOT start) Der Zustand soll nicht der Startzustand sein
- (BOXB start) Der Zustand soll eine eingehende Kante haben, die im Startzustand beginnt
- (OR eins (EX eins)) Im Zusand selbst oder in einem Nachfolger soll “eins” gelten

Zusammen ergibt die Formel alle Nachfolgezustände des Startzustands, in denen Spieler 1 nicht im nächsten Zug gewinnen kann. Um die Formel überprüfen zu lassen, wurden nach dem Lernen in die entsprechenden Zustände die atomaren Propositionen “start”, “eins”, “zwei” und “unentschieden” eingetragen. Abbildung 8.5 zeigt den gelernten Automaten. Die atomaren Propositionen sind neben den jeweiligen Zuständen aufgetragen. Der Automat ist gemäß dem Ergebnis des Modelchecking markiert: Spieler zwei muss Feld 8, unten in der Mitte, spielen, um nicht unmittelbar zu verlieren.

Das Beispiel zeigt, dass bereits von Menschen als trivial empfundene Systeme, die sich in Regeln sehr einfach ausdrücken lassen, in ungünstigen Fällen eine viel größere Automatenrepräsentation haben. Eine weitere Besonderheit dieses Beispiels ist der fehlende Spielraum für Abstraktionen, welche es ermöglichen, dass Spiel auf das Wesentliche zu reduzieren: verkleinert man das Brett, gewinnt immer Spieler 1.

8.2.5 Website

Falk Howar

Das Besipiel “Website” modelliert eine Internetseite mit einem privaten Bereich, der erst nach einem Login verfügbar ist und nach einem Logout wieder unsichtbar wird. Das Modell besteht aus 11 Einzelseiten mit den IDs 0-11. Die Seiten ab ID 7 sind privat. Das Beispiel hat drei Ereignisse:

- login()
- logout()
- gotoPage(Int[0 to 10])

Das Beispiel wurde in zwei verschiedenen Ausgabevarianten gelernt. Im ersten Fall wurde für jede Seite ein individueller Text ausgegeben, wenn die Seite angezeigt wurde. Im zweiten Fall wurde vom Inhalt der einzelnen Seiten abstrahiert und nur ausgegeben, ob es sich bei der anzeiegten Seite um eine private oder um eine öffentliche Seite handelt.

Außerdem wurde an diesem Beispiel mit Hilfe der Eingabe-Abstraktion erfolgreich gelernt. Das Beispiel wurde einmal mit sämtlichen erlaubten Eingaben gelernt, einmal nur mit den relevanten Seite für “Login” und “Logout”, sowie ein drittes mal mit mehreren, aber nicht allen einzelnen Seiten.

In diesem Beispiel und in ähnlichen realen Fällen, kann mit der Eingabeabstraktion der Lernaufwand verringert werden, während das Ergenis aussagekräftig bleibt. Voraussetzung ist, dass die zu prüfenden Stellen z.B. einer Internetseite vorher bekannt sind bzw. dass von den nicht gelernten Seiten bekannt ist, dass von diesen Seiten aus keine kritischen Aktionen ausgelöst werden können. Das Ergebnis bleibt unter der genannten Bedingung aussagekräftig, da in dem gelernten Modell lediglich reflexive und doppelte Kanten fehlen werden.

Abbildung 8.6 zeigt die atomaren Propositionen des gelernten Modells mit abstrakter Eingabe für eine mittlere Anzahl von Seiten. Der Startzustand q0 ist oben links. Von oben links nach oben rechts gelangt man auf Seite Nr. 3. Auf dieser kann man sich einloggen und gelangt so nach rechts unten. Ein Logout ist nur auf Seite Nr. 8, unten links möglich. Nach der Abmeldung gelangt man zurück auf Seite Nr. 0, oben links.

An diesem Beispiel wurde überprüft, ob es möglich ist, eine private Seite zu sehen, ohne sich vorher angemeldet zu haben. Die entstprechende Formel hat folgende Syntax

(ABSENCE between logout[] login[] private)

Die Semantik ist, dass es auf allen Wegen zwischen einem Zustand mit der atomaren Proposition “logout[]” und einem Zustand mit der atomaren Prosopition “login[]” keinen Zustand mit der atomaren Proposition “private” geben darf. Da der gelernte Automat, wie dargestellt, im Wesentlichen ein Kreis ist, der durch den Startzustand verläuft war es nicht notwendig, eine zweite Bedingung für Seiten, die zwischen dem Startzustand und dem ersten Login liegen, zu formulieren.

Der Test hat ergeben, dass das Beispiel eine sichere Seite beschreibt. Die linke Seite von Abbildung 8.7 zeigt einen Ausschnitt des Graphen mit den grünen Markierungen, die anzeigen, dass die Bedingung in den markierten Knoten gilt. Die Bedingung gilt tatsächlich in jedem Zustand.

Beispielhaft wurde eine Unsicherheit eingebaut. Einen Ausschnitt des Ergebnisses zeigt die rechte Hälfte von Abbildung 8.7. Es gibt eine Seite, auf der private Informationen angezeigt werden, ohne dass man zuvor einen “Login[]”-Knoten passieren muss. Die Sicherheitsbedingung ist nicht erfüllt, die Knoten werden rot markiert.

Ein Nachteil der angewendeten Formel ist, dass sie immer auf dem gesamten Graphen gilt oder nicht gilt. Im Falle einer vorhandenen Sicherheitslücke wird zwar gezeigt, dass es eine solche gibt, nicht aber wo genau sie ist.

8.2.6 Die Foodmachine

Falk Howar

Das Beispiel Foodmachine konnte nicht erfolgreich gelernt werden. Das Problem dieses Beispiels ist die Komplexität, welche die Verschiedenen Münzarten und die möglichen Füllstände der einzelnen Produktfächer und Münzspeicher erzeugen. Mit fünf verschiednen Münzarten und zwei Produkten ergeben sich, wenn man von jeder Münzart fünf Stück und von jedem Produkt zwei Stück speichern kann, mindestens 180 Zustände.

Verkleinert man das Beispiel noch weiter, so dass z.B. nur eine Münzart und ein Produkt und je ein Speicher der Größe 1 existieren, wird das Resultat so trivial, dass man daran nichts mehr überprüfen kann. Es wäre z.B. sicher interessant gewesen, an diesem Beispiel zu überprüfen, ob der Automat tatsächlich nur verkauft, wenn danach das Wechselgeld passend ausgegeben werden kann, oder ob der Automat das Wechselgeld richtig ausrechnet.

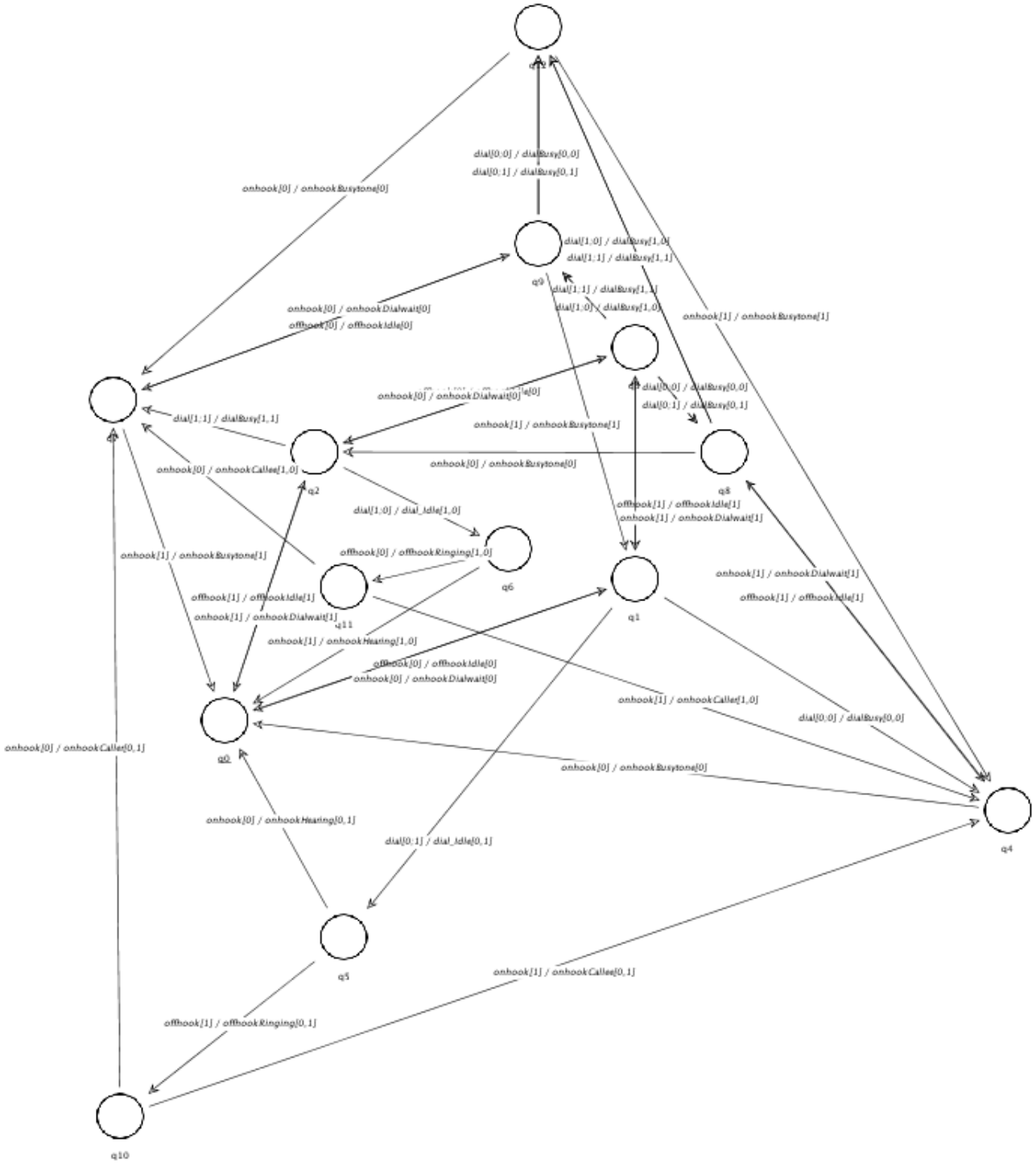

Abbildung 8.1: Lernergebnis für das Telefonbasissystem mit zwei Telefongeräten

Abbildung 8.2: Lernergebnis für das Beispiel Blockwelt

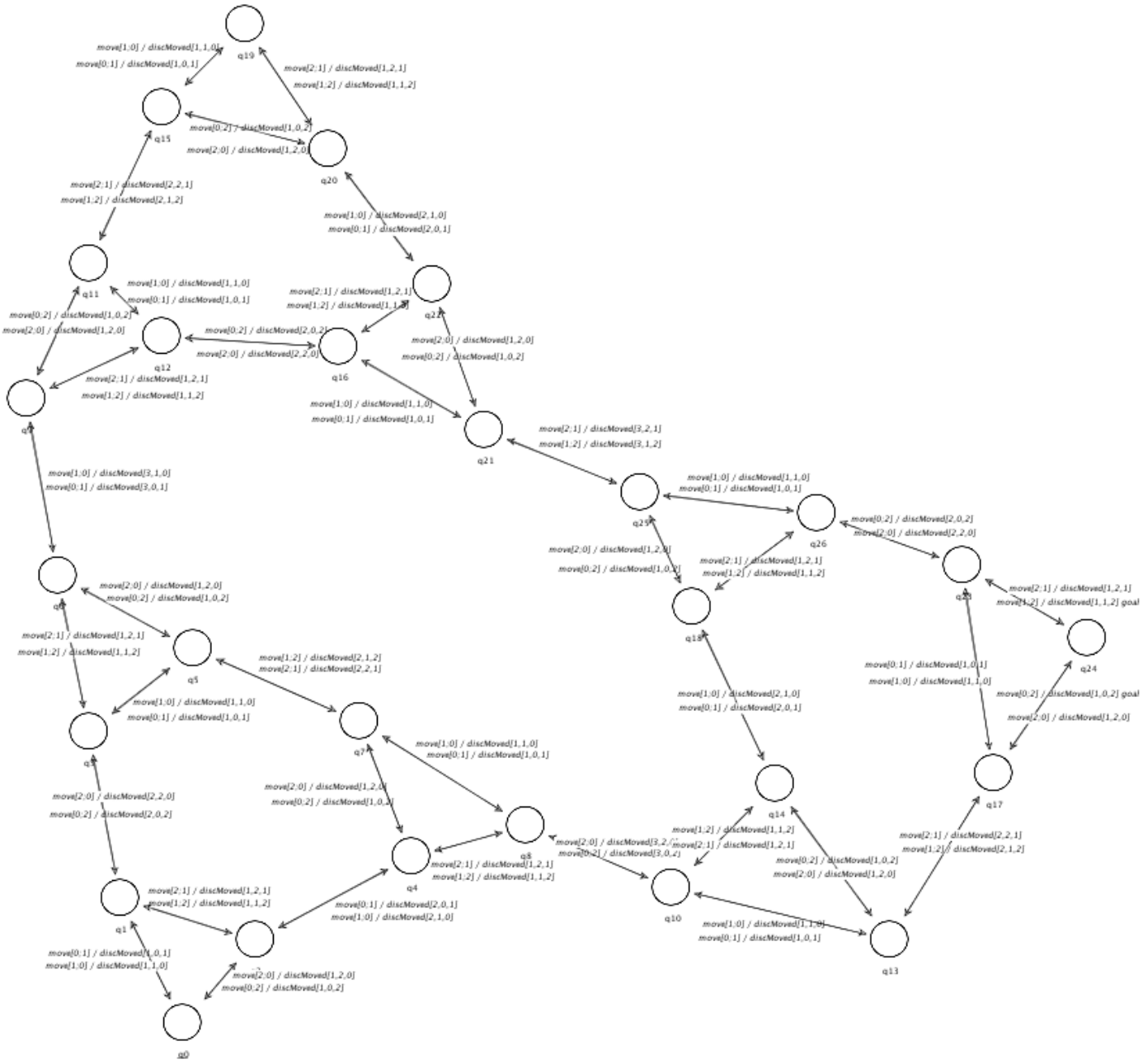

Abbildung 8.3: Lernergebnis für das Beispiel Türme von Hanoi

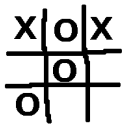

Abbildung 8.4: TicTacToe

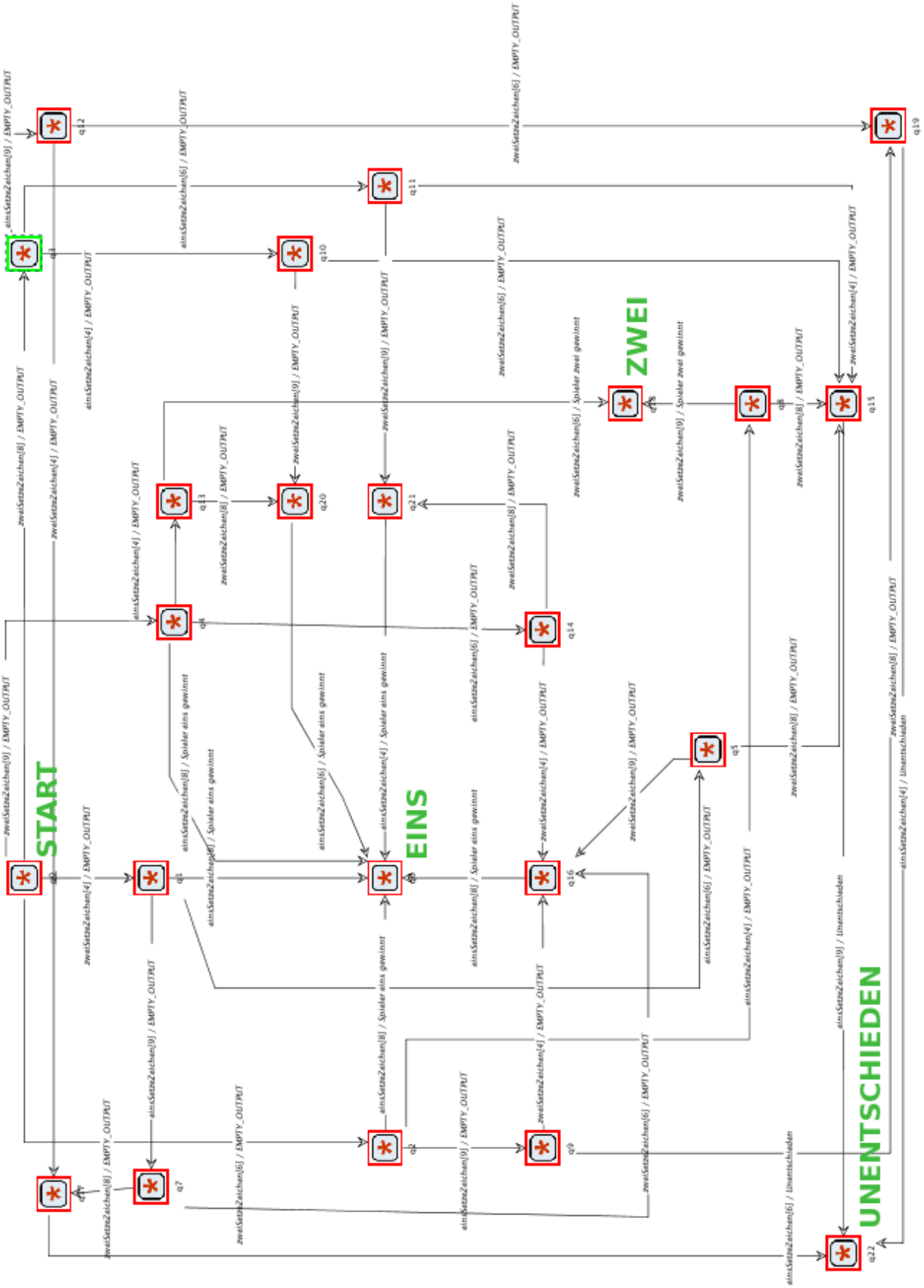

Abbildung 8.5: Gelernter Automat TicTacToe

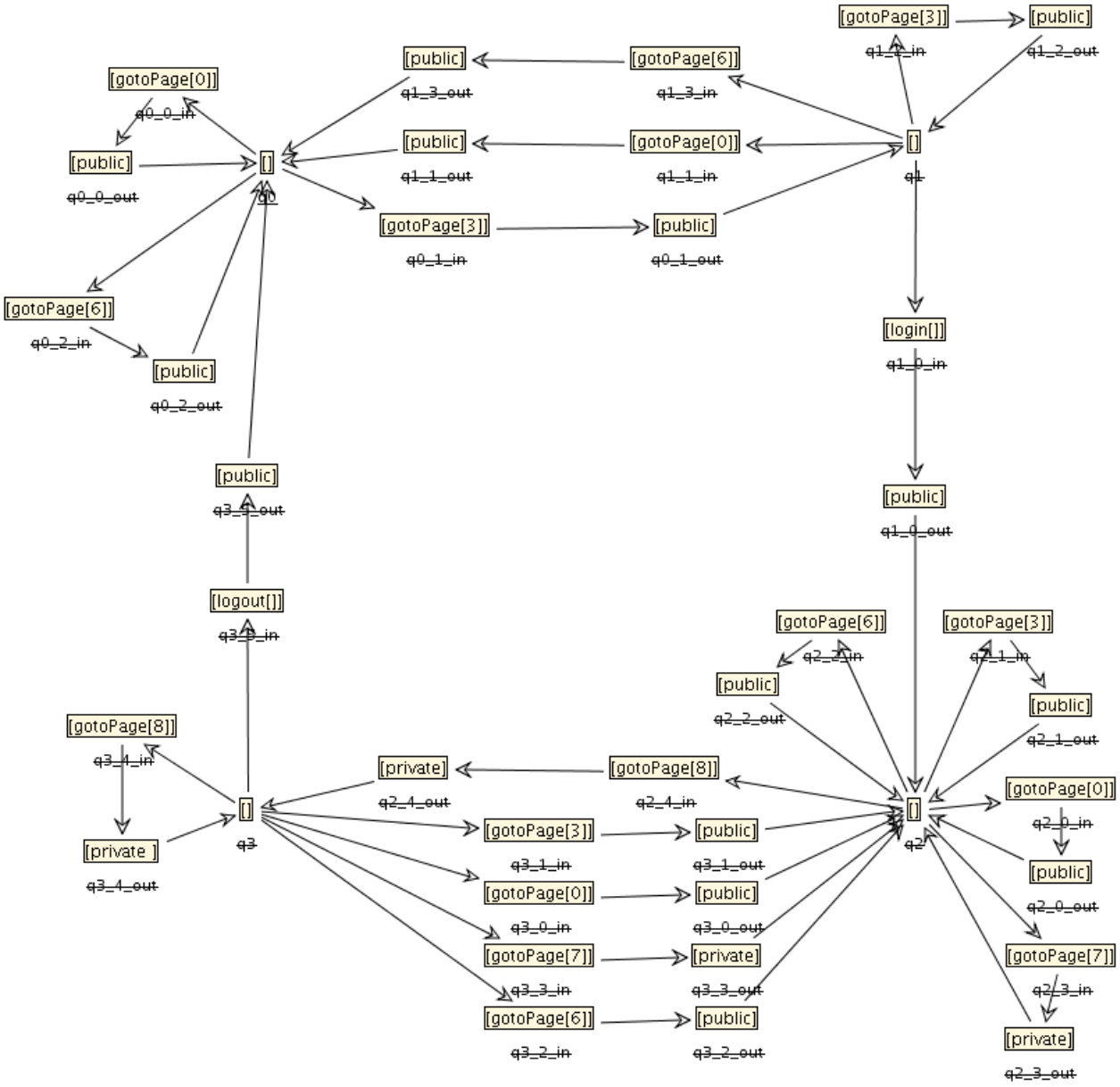

Abbildung 8.6: Propositionen Automat Website

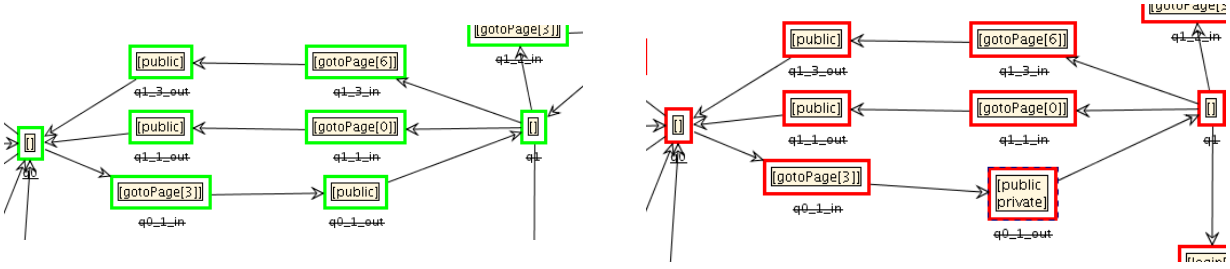

Abbildung 8.7: Modelcheking Website

Kapitel 9

Arbeitsprozess der Projektgruppe

9.1 Einleitung

Mohsen Ahyaie

Heutzutage wird der Bedarf an Computerprojekten immer größer und die Zeit von traditioneller Programmierung ist vorbei. Die Programme müssen so geschrieben werden, daß sie einfacher zu warten sind, d.h. man muss versuchen, weniger Zeit beim Debugging zu vergeuden und nach jeder Änderung feststellen können, ob das ganze Programm läuft, oder ob Konflikte durch die Änderung entstanden sind. Eine Testroutine zu schreiben, ist die beste Lösung für solche Fälle.

Man kann im Laufe der Zeit den Überblick über das gesamte Projekt verlieren, oder nicht mehr in der Lage sein, die Funktionalität des ganzen Programms zu überprüfen. Automatisierte Tests erleichtern die Änderungen und können das ganze Programm in Sekunden überprüfen. Diese Testroutinen sollten am Anfang jedes Projekts geschrieben und jeden Tag so oft wie möglich ausgeführt werden. Denn je größer der Zeitdruck, desto weniger Tests können durchgeführt werden. Je weniger Tests, desto unstabiler der Code. Je unstabiler der Code, desto mehr Fehlermeldungen und Debuggingzeit.

9.2 Extreme Programming

Extreme Programming (XP) ist ein agiler Softwareentwicklungsprozess für kleine Teams. Der Prozess ermöglicht es, langlebige Software zu erstellen und während der Entwicklung auf sich rasch ändernde Anforderungen zu reagieren.

Es wurde von Kent Beck, Ward Cunningham und Ron Jeffries während ihrer Arbeit im Projekt 'Comprehensive Compensation System' bei Chrysler als Vorgehen zur Erstellung von Software entwickelt.

Um ein Projekt nach XP Schritten zu realisieren, müssen bestimmte Techniken und Regeln vom Kunden, Entwickler und Manager beachtet und unterstützt werden. Aufgrund unseres Projektes möchten wir nur die Techniken für die Entwicklung kurz erwähnen und die Wichtigkeit der Rolle des Testens andeuten.

9.2.1 Techniken für die Entwicklung

- **Programmieren in Paaren :**

  Die Programmierer arbeiten stets zu zweit am Code und diskutieren während der Entwicklung intensiv über Entwurfsalternativen. Sie wechseln sich minütlich an der Tastatur ab und rotieren stündlich ihre Programmierpartner. Das Ergebnis ist eine höhere Codequalität, grössere Produktivität und bessere Wissensverbreitung.

- **Gemeinsame Verantwortlichkeit:**

  Der gesamte Code gehört dem Team. Jedes Paar soll jede Möglichkeit zur Codeverbesserung jederzeit wahrnehmen. Das ist kein Recht sondern eine Pflicht.

- **Erst Testen:**

  Gewöhnlich wird jede Zeile Code durch einen Testfall motiviert, der zunächst fehlschlägt. Die Unit Tests werden gesammelt, gepflegt und nach jedem Kompilieren ausgeführt.

- **Design für heute:**

  Jeder Testfall wird auf die einfachst denkbare Weise erfüllt. Es wird keine unnötig komplexe Funktionalität programmiert, die momentan nicht gefordert ist.

- **Refactoring:**

  Das Design des Systems wird fortlaufend in kleinen, funktionserhaltenden Schritten verbessert. Finden zwei Programmierer Codeteile, die schwer verständlich sind oder unnötig kompliziert erscheinen, verbessern und vereinfachen sie den Code. Sie tun dies in disziplinierter Weise und führen nach jedem Schritt die Unit Tests aus, um keine bestehende Funktion zu zerstören.

- **Fortlaufende Integration:**

  Das System wird mehrmals täglich durch einen automatisierten Build-Prozess neu gebaut. Der entwickelte Code wird in kleinen Inkrementen und spätestens am Ende des Tages in die Versionsverwaltung eingecheckt und ins bestehende System integriert. Die Unit Tests müssen zur erfolgreichen Integration zu 100% laufen.

9.3 JUnit

9.3.1 Schreiben von Tests

An dieser Stelle soll als Beispiel eine Methode der zum Projekt ”LIVE”gehörenden Klassen (EcaVarInteger) dem Test unterzogen werden. Die Klasse besteht wie gewöhnlich aus mehreren Methoden. Die Funktionalität der Methode (setVslue()) soll hier geprüft werden. Sie setzt den eingegebenen Intergerwert für ein Objekt der Klassse (EcaVarInteger) ein. Dies kann beispielsweise mit dem folgenden Abschnitt implementiert werden :

```
EcaVarInteger myInt = new EcaVarInteger();

// myInt muss jetzt leer sein

Object element = new Object();
```

```
myInt.setValue(4);

// myInt muss jetzt den Wert "4" haben
```

Der Inhalt des Objekts myInt wird durch das Aufrufen der Methode (getValue()) ausgegeben und anschließend im darauffolgenden Code-Abschnitt mit dem vorgegebenen Wert vergliechen. Die Ausgabe erfolgt auf die folgende Weise :

```
EcaVarInteger myInt = new EcaVarInteger();

myInt.addValue(4);

System.out.println(myInt.getValue());
```

Der Test kann unter anderem den automatisierten Vergleich beinhalten. Der Vergleich wird in dem Code-Abschnitt, der unten aufgeführt ist, realisiert :

```
EcaVarInteger myInt = new EcaVarInteger();

myInt.addValue(4);

System.out.println(myInt.getValue()==4);
```

Das manuelle Prüfen von Booleschen Werten ist in reellen Anwendungen aufgrund des hohen Zeit- und Kostenaufwand nicht effizient. Deshalb wird diese Aufgabe i.d.R dem Rechner überlassen. Die beispielfunktion ist dafür gedacht, eine Ausnahme auszulösen, falls der erzeugte Boolesche Wert ”falseïst :

```
void assertTrue(boolean condition) throws Exception

{

   if ( ! condition)

     throw new Exception

        ( Pruefung fehlgeschlagen );

}
```

Mit Hilfe der Methode (assertTrue())werden die Ausgaben erheblich übersichtlicher. Solange der Test erfolgreich durchläuft, wird nichts angezeigt. Das Erscheinen einer unbehandelten Ausnahme ist ein Indiz dafür, dass etwas schief gegangen ist. Der folgende Code stellt einen vollständig automatisierten Test dar :

```
EcaVarInteger myInt = new EcaVarInteger();

myInt.addValue(4);

assertTrue(fixture.size()==4);
```

9.3.2 Was ist JUnit?

JUnit ist ein Framework zum Testen von Java-Programmen. Junit automatisiert das Schreiben von Tests.

JUnit kann kostenlos von *www.junit.org* heruntergeladen werden. Alle Testklassen von JUnit sind in junit.jar enthalten. Man braucht nur diese Datei im Klassenpfad zu platzieren. JUnit kann auch in eine Reihe von Java-basierten Entwicklungsumgebungen, wie beispielsweise Eclipse, integriert werden.

9.3.3 Vorteile von JUnit

- Es führt die Tests automatisch aus.
- Es führt viele Tests gemeinsam aus und fasst die Ergebnisse zusammen.
- Es bietet einen bequemen Ort zum Sammeln von geschriebenen Tests.
- Es vergleicht die tatsächlichen Ergebnisse mit den erwarteten Ergebnissen und meldet Abweichungen.

9.3.4 Die Klassen-Struktur

In der Regel werden Sie es bei der Arbeit mit JUnit mit fünf Klassen beziehungsweise Interfaces zu tun haben :

- **Assert**
  Eine Sammlung von statischen Methoden, mit denen Ergebniswerte mit erwarteten Werten verglichen werden können (Zusicherungen).Die meisten für JUnit gescgriebenen Testfälle sind indirekt von Assert abgeleitet.
- **Test**
  Das Interface für alle als Tests dienenden Objekte. Das generische Interface Test wird von allen Objekten implementiert.
- **TestCase**
  Hier werden Testfall-Klassen von Junit.framework.TestCase abgeleitet. In der Regel müssen die Tests aus automatisch erzeugten Testreihen ausgeführt werden. TestCase implementiert Test.
- **TestSuite**
  Eine Testreihe aus Test-Objekten. Im einfachsten Fall enthält sie mehrere Testfälle, die alle ausgeführt werden, wenn auch die Testreihe ausgeführt wird.
- **TestResult**
  Eine Zusammenfassung der Ergebnisse aus der Ausführung von einem oder mehreren Tests.

  Die folgende Abbildung zeigt die Beziehungen zwischen den fünf grundlegenden JUnit-Klassen Assert, Test, Test Case, TestSuite und TestResult.

9.4 Arbeitsweise der Projektgruppe

Marc Peschke
Die Woche der Projektgruppe beginnt am Montag mit einer Gruppenarbeit am Vormittag und der Projektgruppensitzung am Nachmittag. Der Montagvormittag wurde festgelegt, um dort als ganze Projektgruppe gemeinsam zu arbeiten, da so Fragen und Anregungen direkt zusammen verwirklicht oder verworfen werden können.
Dies hat den Vorteil, dass diese dann in der Projektgruppensitzung den Projektgruppenleitern direkt vorgestellt werden können.

9.4.1 Zeiten

Am Anfang der Projektgruppe haben wir uns feste Zeiten gelegt an denen wir im Pool sein können. Jeder Projektgruppenteilnehmer hat seine Kernzeiten so gelegt, dass an jedem Tag immer jemand im Pool ist.
Ab Mitte des Semester wurde der Zugriff auf das CVS auch von Zuhause möglich. So wurden spontane Ideen, sowie aber auch flexiblere Arbeitszeiten möglich. Die Hauptarbeit wird aber weiterhin im Pool erledigt, da das Programmieren in Teams zur Arbeitsweise das „Extreme Programmings“[17] gehört.

9.4.2 Kommunikation

Als weitere Kommunikationmöglichkeit wurde ein Emailverteiler eingerichtet. Über diesen können die Teilnehmer und die Leiter der Gruppe Neuigkeiten posten, die für die Anderen wichtig sind. Desweiteren gibt es einen virtuellen Pool, in dem sich jeder der gerade an dem Projekt arbeitet, einloggt.
Die Absicht die mit diesem virtuellen Pool verfolgt wurde ist, dass man so sehen kann, ob jemand gerade eventuell von zuhause aus arbeitet. Zudem ist es möglich so einen geringeren Geräuschpegel im Pool zu ermöglichen, der die anderen Anwesenden weniger stört, indem man Fragen einfach dort klärt anstatt im Pool laut zu diskutieren.

9.4.3 CVS und Eclipse

CVS ist ein Software-System zur Versionsverwaltung von Dateien und unverzichtbar für die Umsetzung von größeren Softwareprojekten in einem Team. Wir setzen dieses ein, um gemeinsam an unseren Daten zu arbeiten. Desweiteren können wir mit dem CV-System auf ältere Versionen zurück greifen, falls mal etwas nicht funktioniert. [28]
Von der Projektgruppe wird das Open-Source-Framework Eclipse zur Entwicklung der Ausführungsumgebung genutzt. Eclipse ist für die Projektgruppe die optimale Entwicklungsumgebung, da es so möglich ist normale Plugins zu nutzen, wie auch vom Lehrstuhl entwickelte. [11]

9.4.4 Aufgabenverteilung

Am Ende jeder PG-Sitzung stellen wir eine ToDo-Liste auf. Jeder Aufgabe wird ein Team zugewiesen, das sich die folgende Woche mit der Realisierung befassen soll. In der nächsten Sitzung wird dann ein kurzer Überlick über die Fortschritte gegeben.

9.4.5 Webserver

Die PG hat vom Lehrstuhl 5 einen Webserver zur Verfügung gestellt bekommen. Auf diesem haben wir eine Groupware zur Arbeitsorganisation, sowie ein ANT zur Arbeitsfortschrittskontrolle installiert.

9.4.6 Organisation

In der Projektgruppenbesprechung werden wöchentlich Arbeitsaufgaben erstellt. Diese werden auf Zetteln in einem Ordner abgelegt und können dort eingesehen und die Aufgaben bearbeitet werden. Zudem besteht die Möglichkeit diese Arbeitsanweisungen in einer Groupware zufinden.
Desweiteren werden Zeitkarten in der Groupware angeboten, mit denen jeder Teilnehmer der Projektgruppe seine investierte Zeit zur Selbstkontrolle festhalten kann.

9.4.7 ANT

Als kleine Kontrolle der Arbeit und der Arbeitenden, wird stündlich ein ANT-Script ausgeführt. Dank diesem wird auf dem Webserver eine Webseite erstellt auf der man folgende Informationen abrufen kann: CVS Stats, Compile Log, JavaDoc, Doxygen Doku, Doxy (pdf) und JUnit Tests.

- *CVS Stats* : Hier findet man Informationen zum Projekt. Codezeilen, Top 10 der Programmierer sowie den Repository Tree.
- *Compile Log* : Hier werden Ausgaben des Compilers gezeigt. Man sieht auf einen Blick wo noch Fehler sind, oder ob das Projekt fehlerfrei kompiliert.
- *JavaDoc* : Die JavaDoc bietet einen Überblick über die Pakete und Klassen des Projektes. Zudem Beschreibungen der Klassen und der Methoden.
- *Doxygen Doku* : Doxygen bietet zusätzlich zu den vom JavaDoc angebotenen Informationen, Auflistungen von Abhängigkeiten der Pakete und Klassen untereinander.
- *Doxy (pdf)* : Hier kann man die Doxygen Doku als Pdf runterladen.
- *JUnit Tests* : Hier werden die Ergebnisse der JUnit Test angezeigt.

9.4.8 Die Realität

Die Projektgruppe hat erfolgreich Extreme Programming [17] in ihre Arbeitsweise eingebunden. Viele positive Eigenschaften von Extreme Programming haben sich während des Arbeitsprozesses eingestellt, wie zB, dass Spaß an der Arbeit erhalten bleibt, erstellte Software früh lauffähig und vorführbar war und Teamwork und Teamverständnis gefördert wurden.
Die Projektgruppe hat sich vorgenommen, nur laufähige Versonen unserer Ergebnisse in das CVS einzuspielen. Dies war notwendig, damit das System und somit auch das ANT einen kompletten Check durchführen kann.Dies ist notwendig, damit das ANT die Junit-Test regelmäßig durchführen kann.
Die Junit-Test sollten eine möglichst hohe Abdeckung haben. Die Projektgruppe hat zum jetzigen Zeitpunkt 236 Testfälle generiert. Es treten 20 Warnungen und 4 Fehler auf.

9.5 Code Quality Management und PG LiVe

Svetla Nikolova

Heutzutage sind mehr als 2/3 der Softwareprojekte Misserfolge: nur 29 % der Projekte werden erfolgreich abgeschlossen, von den restlichen 71 % verursachen 18 % der Projekte nur Kosten ohne produktive Ergebnisse und 53 % werden durch nicht eingehaltene Fristen und überschrittenes Budget und/oder Mängel an bestimmten geforderten Funktionen charakterisiert.
Bei der Projektgruppenarbeit der PG LiVe gab es kein Budget in dem Sinne wie in einem Unternehmen. Anforderungen und Zeitfristen mussten aber trotzdem eingehalten werden, was auch den PG-Teilnehmern gut gelungen ist. Damit gehört das PG-LiVe-Projekt zu den 29 % der erfolgreichen Projekte.

Qualität ist die Gesamtheit von Eigenschaften und Merkmalen eines Produktes oder einer Tätigkeit, die sich auf deren Eignung zur Erfüllung gegebener Erfordernisse bezieht. Unter Software-Qualität versteht man die Erfüllung von Qualitätszielen, d.h. die Umsetzung von Qualitätsmerkmalen.

Code Quality Management umfasst verschiedene Techniken, Prozesse und Werkzeuge, die die technische Qualität eines Softwaresystems dauerhaft sichern. Das System soll planbar sein, ökonomisch sinnvoll geändert und an neue Anforderungen angepasst werden können.

Es existieren verschiedene Ansätze für die Qualitätssicherung von großen Softwaresystemen: Verifikation, Validierung und Testen. Im folgenden werden die Qualitätssicherungs-Techniken erläutert und es wird erwähnt welche davon die PG-LiVe-Teilnehmer eingesetzt haben.

9.5.1 Verifikation

Die Programmverifikation ist ein formaler Nachweis von Eigenschaften von Programmen oder Programmteilen, der zum Beweis deren Korrektheit dient. Es muss eine vollständige und formale Spezifikation vorliegen, die der Ausgangspunkt des verifizierenden Verfahrens ist. Ein Programm (oder eine Anweisung) wird als **korrekt** gegenüber der Spezifikation bezeichnet, wenn es genau die vorgegebenen Spezifikationen erfüllt, also auf alle Eingaben mit den gewünschten Ausgaben reagiert. Der Beweis kann mittels formaler mathematisch-logischer Techniken, wie z.B. des Hoare-Kalküls, durchgeführt werden. Zuerst wird bewiesen, dass das Programm richtige Ergebnisse liefert, wenn es anhält. Für einen totalen Korrektheitsbeweis ist anschließend noch zu zeigen, dass das Programm immer terminiert. Der gesamte Beweis kann informell (per Hand) oder formal (mit einem automatischen Theorembeweiser) durchgeführt werden. Bei der Verifikation wird die Richtigkeit des Programms für alle Eingaben bewiesen und dabei wird stets eine präzise Semantik benötigt.

In der Praxis treten folgende Probleme auf:

- Auf Grund der riesigen Anzahl von möglichen Datenflüssen, Workflows und externen Datenquellen ist es schwer eine formale Spezifikation für umfangreiche Softwaresysteme zu erstellen, . Damit sind die heutigen formalen Verifikationstechniken überfordert und deswegen werden sie überwiegend im Embedded-Systems-Bereich angewendet.
- Dynamik der Spezifikation: Es gibt kaum ein Softwaresystem, dessen Anforderungen sich während des Einsatzes nicht verändern. Da das Modifizieren einer formalen Spezifikation und des damit zusammenhängenden Korrektheitsbeweises genau so große Probleme verursachen kann wie bei nicht verifizierten Systemen, wird die Verifikation in der Regel nur bei Systemen mit relativ konstanten Anforderungen (z.B. einer Kraftwerkssteuerung) verwendet.

Auf Grund der oben genannten Praxisprobleme wurde der Ansatz zur klassischen Verifikation in der Projektgruppe LiVe nicht verwendet .

9.5.2 Validierung

Mit Validierung bezeichnet man das Prüfen, ob die Entwicklungsergebnisse die ursprünglichen Anforderungen erfüllen, was in der jeweiligen Teststufe zu bewerten ist. Beim Validieren soll der Tester feststellen, ob das Produkt im Kontext der Produktnutzung sinnvoll ist, ob es eine bestimmte festgelegte Aufgabe tatsächlich löst. Man stellt bei der Validierung die Frage: „Ist es das richtige System?“, während die Frage bei der Verifikation lautet: „Ist das System richtig, sind die Spezifikationen korrekt umgesetzt?“

In der Praxis beinhaltet jeder Test beide Aspekte, wobei der Validierungsanteil mit steigender Teststufe zunimmt. Aspekte der Validierung wurden von der Projektgruppe LiVe eingesetzt.

Des weiteren ist Fehlerfreiheit durch Testen nicht erreichbar - sogar wenn alle ausgeführten Testfälle keine Fehler finden, kann mit Ausnahme von sehr kleinen Programmen nicht garantiert werden, dass es nicht zusätzliche Testfälle gibt, bei deren Ausführung weitere Fehler auftreten.

9.5.3 Testen

Unter dem **Testen von Software** versteht man eine stichprobenartige Ausführung eines Testobjektes, welche als Ziel besitzt, die Qualität von Softwaresystemen zu erhöhen und Fehler zu finden.

Der Testprozess beinhaltet fünf Phasen:

- **Testplanung**: Die Ressourcen (Mitarbeiter, Werkzeuge, Geräte) werden geplant und die Teststrategie wird festgelegt.
- **Testspezifikation**: Während dieser Phase werden Testfälle bestimmt, wobei als Grundlage alle Dokumente dienen, aus denen Anforderungen an das Testobjekt abgeleitet werden können. Die Testdaten umfassen nicht nur die Eingabedaten, sondern auch Vorbedingung, Nachbedingung, Randbedingungen und das Soll-Verhalten bzw. -Ergebnis, welches vor der Testdurchführung festzulegen ist. Ein Testfall soll so gewählt werden, dass eine bisher noch nicht bekannte Fehlerwirkung aufgedeckt werden kann.
- **Testdurchführung**
- **Testprotokollierung**: Aus dem Testprotokoll muss verständlich sein, welche Teile wann, von wem, wie intensiv und mit welchem Ergebnis getestet wurden.
- **Auswertung und Bewertung des Testers**: Beim Nichtübereinstimmen von Soll- und Ist-Verhalten kann eine Fehlerwirkung vorliegen, aber es ist auch möglich, dass das Soll-Ergebnis und/oder die Testumgebung und/oder die Testspezifikation falsch bzw. fehlerhaft sind.

Ein klassischer Testprozess in diesem Sinne fand in der PG LiVe nur in eingeschränkter Form statt. Man konzentrierte sich auf die Testdurchführung und Testauswertung. Dabei wurden die Tests mit JUnit durchgeführt. An dieser Stelle wird auf die entsprechende Ausarbeitung über JUnit und auf den ensprechenden Abschnitt des Endberichts hingewiesen.

Ein **vollständiger Test** ist allerdings praktisch nicht durchführbar. Ein vollständiger Test umfasst die Kombinationen aller möglichen Eingaben unter Betrachtung aller möglichen Randbedingungen. Durch die Vielzahl kombinatorischen Möglichkeiten ergibt sich eine nahezu unbegrenzte Anzahl an Tests, die durchzuführen wären. Infolgedessen ist ein solches Austesten nicht möglich.

Zur Zeit existiert **kein umfangreiches fehlerfreies Softwaresystem** und es wird wahrscheinlich in naher Zukunft auch keines geben. Sobald das System eine gewisse Komplexität und Anzahl an Programmzeilen erreicht, wird es unmöglich es fehlerfrei zu halten.

9.5.4 Statisches Testen und statische Analyse

Statisches Testen

Statische Untersuchungen, wie Reviews und wergzeugunterstützte Quellcode- und Dokumenten-Analysen, können bei der Qualitätssicherung sehr hilfreich sein.

Der statische Test (oft auch als **manueller Test**, **manuelles Prüfen** oder **statische Analyse** bezeichnet) ist eine Prüfmethode, bei der das Testobjekt (meistens ein für die Erstellung der Software wichtiges Dokument oder Quellcode) analysiert wird. Die Analyse kann durch intensive Betrachtung durch verschiedene Personen oder durch bestimmte Werkzeuge erfolgen, wobei es für die werkzeuggestützte Analyse notwendig ist, dass die Dokumente eine formale Struktur besitzen.

Review ist sowohl die Bezeichnung für ein bestimmtes Prüfverfahren von Dokumenten als auch ein Oberbegriff für die unterschiedlichen statischen Überprüfungen, die von Menschen durchgeführt werden.

Alle Dokumente, die während des Softwareentwicklungsprozesses erstellt oder verwendet werden, können Objekt eines Reviews sein (z.B. Anforderungsbeschreibung, Programmspezifikationen, Benutzerhandbuch, Vertrag usw.). In der Regel ist nur die manuelle Überprüfung der Semantik solcher Dokumente durch eine oder mehrere Personen möglich, daher sind die Reviews bestens dafür geeignet.
Weitere Arten der manuellen statischen Untersuchungen sind Inspektion, Walkthrough, technisches Review und informelles Review, welche in dem Buch *Spillner A., Linz T.: Basiswissen Softwaretest* [26] ausführlich behandelt werden.

Manuelle statische Untersuchungen fanden regelmäßig während des PG-LiVe-Arbeitsprozesses statt. Während der Projektgruppensitzungen wurden Anforderungen diskutiert und es wurde auf Probleme und Unklarheiten bezüglich dieser eingegangen.

Statische Analyse

Das Ziel der statischen Analyse ist das Finden von Fehlern oder fehlerträchtigen Stellen in einem Dokument. So kann man z.B. Rechtschreibprüfprogramme als eine Art statische Analysatoren betrachten. Die Bezeichnung „statische Analyse“ besagt, dass bei diesem Verfahren keine Ausführung der Prüfobjekte stattfindet. Ein weiteres Ziel der statischen Analyse ist die Ermittlung von Messgrößen oder Metriken, damit die Qualität bewertet, gemessen und nachgewiesen werden kann. Weiterführende Literatur zur Ermittlung von Messgrößen oder Metriken ist das Buch *Code Quality Management* [25].

Allerdings muss das zu analysierende Dokument eine formale Struktur besitzen, damit es durch ein Werkzeug überprüft werden kann. Beispiele für solche Dokumente, die nach einem vorgegebenen Formalismus aufgebaut werden, sind die technischen Anforderungen, die Softwarearchitektur oder der Softwareentwurf. In der Praxis ist oft der Programmcode das einzige formale Dokument, auf dem eine statische Analyse durchgeführt werden kann.

Mit der statischen Analyse lassen sich allerdings nicht alle Fehler nachweisen. Es existieren Fehler oder genauer gesagt Fehlerzustände, die erst bei der Ausführung, also zur Laufzeit, zur Wirkung kommen und sich vorher nicht ermitteln lassen. Zum Beispiel wird bei einer Division der Wert des Divisors in einer Variablen gehalten. Diese Variable kann nur zur Laufzeit den Wert Null annehmen, was zu einer Fehlerwirkung führt. Statisch ist dieser Fehlerzustand meistens nicht erkennbar, es sei denn, der Variablen wird eine Konstante mit dem Wert Null zugewiesen.

Alle Compiler führen eine statische Analyse des Programmtextes durch, indem sie überprüfen, ob die Syntax der jeweiligen Programmiersprache eingehalten wird. Zusätzlich zu dem Erkennen der Syntaxverletzung einer Programmiersprache werden auch weitere Informationen durch die meisten Compiler bereitgestellt, die durch die statische Analyse ermittelt werden können. Beispielsweise liefert der Java-Compiler die folgenden Informationen:

- Überprüfung der typgerechten Verwendung von Daten und Variablen
- Ermittlung von nicht deklarierten Variablen
- Nicht erreichbarer/verwendbarer Code
- Über- und Unterschreitung von Feldgrenzen bei statischer Adressierung

Diese Eigenschaften werden auch zusätzlich zu der Überprüfung der Syntaxverletzung von dem von der PG-LiVe entwickelten *EcaDebugger* gewährleistet, welcher zur Erstellung von ECAL-Scripts dient.

Prüfung der Einhaltung von Konventionen und Standards

Mit Hilfe von sprachspezifischen Analysatoren kann der Verstoß gegen Standards und weitere Konventionen ermittelt werden. Diese Art der Überprüfung benötigt wenig Zeit und so gut wie keine Personalressourcen. Es sollen in einem Projekt nur diejenigen Richtlinien zugelassen werden, zu denen ein geeignetes Werkzeug existiert, alle anderen Vorschriften erweisen sich in der Praxis als bürokratische Belastung. Des weiteren sind die Programmierer eher dazu bereit, die Richtlinien umzusetzen, wenn es ihnen bewusst ist, dass der Programmcode durch ein Werkzeug auf Einhaltung der Programmierkonventionen überprüft wird.

Die PG-LiVe-Teilnehmer haben als Laufzeitumgebung *Eclipse* benutzt, dadurch wurden beispielsweise jedes mal Warnungen ausgegeben, wenn die Java-Namenskonventionen für Attribute, Methoden, Klassen usw. nicht eingehalten wurden.

Datenflussanalyse

Eine weitere Möglichkeit zur Fehlerentdeckung ist die Datenflussanalyse. Bei der Datenflussanalyse wird die Verwendung von Daten auf Pfaden durch den Programmcode untersucht. Es ist nicht immer möglich, Fehlerzustände nachzuweisen, oft wird in diesem Zusammenhang von Anomalien oder Datenflussanomalien gesprochen. Unter einer Anomalie versteht man eine Unstimmigkeit, die zu einer Fehlerwirkung führen kann aber nicht unbedingt muss. Datenflussanomalien sind zum Beispiel die referenzierende Verwendung einer Variable ohne vorherige Initialisierung oder die Nichtverwendung eines Wertes einer Variablen. Solche Anomalien werden auch von dem Java-Compiler entdeckt. Zur Analyse wird die Benutzung der einzelnen Variablen untersucht.

Datenflussanalyse wird bei einer Reihe von Tools verwendet. Das von der Projektgruppe LiVe eingesetzte Eclipse-Plugin *Coverlipse* benutzt Datenflussanalyse (all-uses coverage) zur Visualisierung der Quellcode-Abdeckung im Zusammenhang mit JUnit-Tests.

9.5.5 Dynamisches Testen

Beim dynamischen Testen wird das Testobjekt innerhalb eines Testsystems auf einem Rechner ausgeführt. Ziel des Testens ist es, nachzuweisen, dass das bereits implementierte Programmstück die vorgegebenen Anforderungen erfüllt, und möglicherweise bestimmte Abweichungen und Fehlerwirkungen zu finden.

Beim Testen muss ein ablauffähiges Programm vorliegen.

Das Testobjekt ruft meistens weitere Programmteile auf. Wenn diese Programmteile noch nicht fertig sind, werden sie beim Testen durch Platzhalter ersetzt. Diese Platzhalter, oder auch stubs genannt, simulieren das Eingabe-/ Ausgabeverhalten der ersetzten Programmteile. Testtreiber dienen dazu,

Programmteile zu simulieren, die das Testobjekt mit Eingabewerten versorgen. Testtreiber und Platzhalter (oder auch Stellvertreter) bilden zusammen den Testrahmen und dienen dazu, die Testfälle auszuführen, auszuwerten und Testprotokolle aufzuzeichnen. Beim Testen können auch Testrahmengeneratoren eingesetzt werden. Der Testrahmen ist in der Regel vom Tester zu entwickeln.

Blackbox- and Whitebox-Verfahren

Es existieren verschiedene Verfahren zur systematischen Erstellung von Testfällen und damit zur Überprüfung von Testobjekten. Diese Verfahren können in zwei Kategorien eingeteilt werden: Blackbox- und Whitebox-Verfahren, präziser ausgedruckt: Blackbox- and Whitebox-Testfallentwurfsverfahren.

Bei einnem **Blackbox-Verfahren** ist das Testobjekt als schwarzer Kasten anzusehen, es sind keine Informationen über den Programmtext notwendig. Dabei ist das Testobjekt von außen zu betrachten, daher ist keine Steuerung des Ablaufs des Testobjekts außer durch die jeweiligen ausgewählten Eingaben möglich. Bei den Überlegungen zu den Testfällen konzentriert man sich auf die Spezifikation bzw. die Anforderungen.
Die Blackbox-Verfaren sind auch als *funktionale Testverfahren* bekannt, da die Prüfung des Ein-/Ausgabe-Verhaltens, also die Funktionalität des Testobjektes, die höchste Priorität besitzt.

Im Falle von einem **Whitebox-Verfahren** wird auf den Programmcode zugegriffen. Es wird der innere Ablauf im Testobjekt analysiert. Ein Eingriff in den Ablauf des Testobjekts ist nur in Ausnahmefälle möglich. Testfälle können aus der Programmstruktur abgeleitet werden.
Die Whitebox-Verfahren werden auch *srukturelle Testverfahren* genannt, da sie die Testobjekt-Struktur (Komponenten-Hierarchie, Kontrollfluss, Datenfluss des Testobjektes) berücksichtigen.

Die Whitebox-Verfahren sind für die unteren Teststufen geeignet, während Blackbox-Verfahren überwiegend für die höheren Teststufen angemessen sind.

In folgenden werden zwei Blackbox-Verfahren, *die Äquivalenzklassenbildung und die Grenzwertanalyse* und ein Whitebox-Verfahren *Anweisungsüberdeckung* vorgestellt.

Blackbox-Verfahren

Äquivalenzklassenbildung

Bei der Äquivalenzklassenbildung wird die Menge aller möglichen konkreten Eingabewerte für ein Eingabedatum in Äquivalenzklassen unterteilt. Zu einer Äquivalenzklasse gehören alle Eingabedaten, bei denen erwartet wird, dass die Eingabe eines beliebigen Datums aus der Äquivalenzmenge stammt. Der Test eines Repräsentanten der Äquivalenzklasse reicht aus. Neben den Äquivalenzklassen, welche die gültigen Eingaben beinhalten, müssen auch solche berücksichtigt werden, die die ungültigen umfassen.

Vorgehensweise zur systematischen Herleitung:

- Zuerst wird der Definitionsbereich für jede Eingabevariable bestimmt. Dadurch wird die Äquivalenzklasse der zulässigen Eingabewerte gebildet. Alle Eingaben außerhalb des Defitionsbereichs bilden die Äquivalenzklasse der unzulässigen Werte.
- Danach werden die Äquivalenzklassen verfeinert. Eingaben, die laut Spezifikation vom Testobjekt unterschiedlich verarbeitet werden, müssen in neue verschiedene Unteräquivalenzklassen eingefügt werden. Dies wird so lange fortgeführt, bis sich alle Anforderungen mit den Äquivalenzklassen decken.
- Aus jeder Äquivalenzklasse wird dann eine Eingabe als Testfall ausgewählt.
- Zusätzlich müssen die nötigen Vorbedingungen und das erwartete Ergebnis festgehalten werden.

- Das gleiche Prinzip kann auch für die Ausgabewerte verwendet werden. Es ist jedoch aufwändiger, da für jede Ausgabe die entsprechende Eingabe gefunden werden muss.

Grenzwertanalyse
Die Grenzwertanalyse stellt eine sehr wertvolle Ergänzung zu den Testfällen bereit, die durch die Äquivalenzklassenbildung bestimmt werden. Oft treten Fehlerzustände in Programmen an den Grenzbereichen der Äquivalenzklassen auf, weil hier fehlerträchtige Fallunterscheidungen in den Programmen entstehen können. Häufig werden die Fehlerwirkungen durch einen Test mit Grenzwerten entdeckt. Diese Technik ist nur anwendbar, wenn eine geordnete Menge von Daten einer Äquivalenzklasse vorliegt und sich deren Grenzen bestimmen lassen. Bei der Grenzwertanalyse werden die „Ränder"der Äquivalenzklasse untersucht. An jedem Rand werden der exakte Wert und die beiden, innerhalb und außerhalb der Äquivalenzklasse benachbarten, Werte getestet.

Die Kombination der Äquivalenzklassenbildung mit der Grenzwertanalyse ist sehr wirkungsvoll: durch die Äquivalenzklassen werden keine Tests, die gleiche Resultate liefern, doppelt durchgeführt. Andererseits sind die geordneten Äquivalenzklassen für die Grenzwertanalyse notwendig, um Grenzen zu finden. Fehler sind wahrscheinlicher an den Grenzen einer Äquivalenzklasse zu finden als irgendwo in der Mitte.

Die beiden oben vorgestellten Blackbox-Testfallentwurfsverfahren wurden von den PG-LiVe-Teilnehmern bei der Erstellung von JUnit-Tests berücksichtigt.

Blackbox-Verfahren sind sehr nützlich, um die Funktionalität des Softwaresystems zu prüfen.

Whitebox-Verfahren

Anweisungsüberdeckung *(statement coverage)*
Bei der Anweisungsüberdeckung werden einzelne Anweisungen untersucht. Es wird zunächst der Programmcode in einen Kontrollflussgraphen überführt. Die Anweisungen werden als Knoten dargestellt und der Kontrollfluss zwischen den Anweisungen wird durch die Kanten repräsentiert. Es wird vorausgesetzt, dass der Kontrollflussgraph nicht manuell sondern von einem Werkzeug erstellt wird, das eine Eins-zu-Eins-Abbildung zwischen dem Programmtext und dem Graphen sicherstellt. Beim Vorliegen von Sequenzen von Anweisungen werden diese als ein einziger Knoten gezeichnet, da bei der Ausführung der ersten Anweisung der Sequenz auch die nachfolgenden Anweisungen ausgeführt werden.
Mit dieser Methode kann nicht erreichbarer Code entdeckt werden. Allerdings ist es möglich, dass fehlende Anweissungen nicht erkannt werden. Das Anweisungsüberdeckungs-Verfahren wird bei dem Eclipse-Plugin *Coverlipse* verwendet, welches die PG-LiVe-Teilnehmer im Zusammenhang mit JUnit-Tests eingesetzt haben.

Whitebox-Verfahren eignen sich für die unteren Testebenen, da sie sich den Programmcode direkt anschauen.

Für zusätzliche Details zu den vorgestellten Methoden und für weitere Blackbox- und Whitebox-Verfahren wird auf die folgende Literatur verwiesen: *Spillner A., Linz T.: Basiswissen Softwaretest* [26], *Riedemann E. H.: Testmethoden für sequentielle und nebenläufige Software-Systeme*[13] und *Foliensatz zur Vorlesung Softwaretestmethoden II* [21].

9.5.6 Fazit

Heutzutage besitzt die Qualitätssicherung eine hohe Priorität. Von den vorgestellten und eingesetzten Techniken zur Qualitätssicherung hat das dynamische Testen, insbesondere die JUnit-Tests im Zusammenhang mit *Coverlipse*, sehr zur Entdeckung und Beseitigung von geholfen. Ein anderes Verfahren, das sich als hilfreich erwiesen hat, war die manuelle statische Untersuchung von Anforderungen.

Kapitel 10

Fazit

10.1 Fazit

Christian May

Die an die Projektgruppe gestellten Ziele wurden erfüllt. Ziel der Projekt war es, Systeme basierend auf ECA-Regeln zu lernen und als Zustandsmodelle abzubilden. Zu diesem Zweck wurde von der Projektgruppe als eine Laufzeitumgebung zur Ausführung von auf ECA-Regeln basierenden Systemen entwickelt. Diese musste dann an eine externe Lernbibliothek angebunden werden. Um die gelernten Zustandsmodelle abzubilden und um Systemeigenschaften mittels Model-Checking Tools verifizieren zu können, wurde ein Plugin für die Laufzeitumgebung in jABC implementiert jABC bietet ein mächtiges Tool für das Model-Checking.
Während der Arbeit wurden von der Projektgruppe weitere Ziele selbstständig weiterentwickelt und verwirklicht. Um Regelsysteme zu beschreiben wurde eine eigene Sprache zur Beschreibung dieser von den Teilnehmern der Projektgruppe entworfen. Um die so beschriebenen Regelsysteme in die Laufzeitumgebung einzugeben, war die Entwicklung eines eigenen Parsers und die Implementierung von diesen unabdingbar. Selbst ein eigener Editor zur Vereinfachung der Beschreibung bzw. Entwicklung von Regelsystemen wurde entworfen. Bei der Visualisierung der Zustandsmodelle ergab sich das Probleme, das die Darstellung schnell unübersichtlich wurden. Hier wurde von der Projektgruppe Layoutalgorithmen integriert, um die Darstellung lesbarer zu machen.
Als Nebenaufgabe an die Projektgruppe war die Erprobung eines ungewöhnlichen Entwicklungsprozesses *eXtreme Programming*. Dabei wurde soweit wie möglich auf eine genaue Planungsphase verzichtet und so früh wie möglich mit paarweiser Programmierung begonnen. So entwickelte sich das Projekt fast eigenständig in die richtige Richtung. Schnell zeigte sich dadurch, das vorherige Planungen nicht realisierbar waren. Durch den verringerten Planungsaufwand konnte schnell auf und in den Paaren individuell auf auftretende Probleme reagiert werden.
Bezüglich des Entwicklungsprozesses ist festzuhalten, das der geringe Planungsaufwand im Zuge des *eXtreme Programming* absolut berechtigt ist. Dies dürfte auch mit ein Grund dafür gewesen sein, daß die Ziele der Projektgruppe so gut verwirklicht wurden.
Die Überprüfung von Eigenschaften der gelernten Zustandsmodelle mittels der Model-Checking klappte unmittelbar mit Hilfe des sehr gut umgesetzten Model-Checking Plugins für jABC. Überprüft wurden von der Projektgruppe eigens erstellte Beispiele. Hier stellte sich schnell herraus, daß effizientere Algorithmen zur Darstellung von Zustandsmodellenwichtig sind. Einige Beispiele stellten sich für das Model-Checking als sehr undankbar herraus. So war es unmöglich, sinnvolle Eigenschaften für z.B. ein System für die Steuerung eines Aufzuges zu erarbeiten.
Die Arbeit und das Klima in der Projektgruppe waren sehr gut. Dies lässt sich natürlich auch darauf zurückführen, dass ein viele der Teilnehmer sich schon vorher kannte. Zu größeren Differenzen innerhalb der Projektgruppe kam es nicht. Entscheidend hierfür war aber wohl auch, dass das

Leistungsniveau der Teilnehmer ähnlich ist. Auf Grund der ähnlichen und guten Voraussetzungen gab es auch kaum Zeitverzug bei der Verwirklichung der Teilziele. Insgesamt hat die Zeitplanung also - vor allem auch wegen der sehr motivierten Teilnehmer - gut funktioniert.
Das Erlernen von Projektarbeit ist als Ziel erreicht worden. Dies lässt sich auch daran festmachen, dass viele Kleingruppen eigenständig entwickelt haben und die Zusammensetzung zu einem Gesamtprodukt keine Probleme machte. Des Weiteren wurden die gestellten Anforderungen ohne grossen Zeitverzug erreicht und selbstständig nicht geforderte Ziele verwirklicht. Auch wenn ein Projekt in der Wirtschaft andere Voraussetzungen und Dimensionen hat, ist das Konzept der Projektarbeit allen Teilnehmern näher gebracht worden.

Anhang A

Sprachreferenz

Falk Howar

Dieser Anhang enthält die EBNF im ANTLR- bzw. YACC-Format von *ECAL*.

Außerdem enthält die EBNF Steuerzeichen zur Erstellung des abstrakten Syntaxbaumes (^ , !).

Listing A.1 *Auszug aus ruleset.g*

```
//--------------------------------------
// Parserregeln
//--------------------------------------

g_test
: INIT^ (g_action)*
;

g_file
: (g_typedefs|g_options|g_events)
;

g_options
: OPTIONS^ DDOT! (g_option)* (g_typedefs|g_events)
;

g_typedefs
: TYPEDEF^ DDOT! (g_typedef)* g_events
;

g_events
: EVENTS^ DDOT! (g_eventdec SEMICOLON!)* g_variables
;

g_variables
: VARIABLES^ DDOT! (g_vardef SEMICOLON!)* (g_functions|g_init|g_rules)
;
```

```
g_functions
: FUNCTIONS^ DDOT! (g_function|g_ext_function)* (g_init|g_rules)
;

g_init
: INIT^ DDOT! (g_action)* g_rules
;

g_rules
: RULESET^   DDOT! (g_rule)* (g_invariants)?
;

g_invariants
: INVARIANTS^ DDOT! (g_invariant)*
;

//---------------------------------------
// Optionen
//---------------------------------------
g_option
: IDENTIFIER^ DDOT! IDENTIFIER SEMICOLON!
    ;

//---------------------------------------
// Invarianten
//---------------------------------------
g_invariant
: INVARIANT! IDENTIFIER^ BEGINBLOCK! (g_action)* ENDBLOCK!
    ;


g_exception
: EXCEPTION^ LPAREN! STRING RPAREN! SEMICOLON!
;


//---------------------------------------
// Funktionen
//---------------------------------------
g_function
: FUNCTION^ g_fct_call BEGINBLOCK (g_action)+ ENDBLOCK!
;

g_fct_call
: (g_vardec|IDENTIFIER) LPAREN (g_fct_param (COMMA! g_fct_param)* )? RPAREN!
;

g_fct_param
:  g_vardec
;
```

```
g_fct_return
: RETURN^ g_expression SEMICOLON!
;

g_ext_function
: EXTERNAL^ FUNCTION! IDENTIFIER LPAREN! (TYPIFIER (COMMA! TYPIFIER)* )? RPAREN! SEMICOLON!
;


//---------------------------------------
// Regeln
//---------------------------------------
g_rule
: ON^ g_event (g_priority)? g_condition g_actionblock
;

g_event
: CHANGE^ LPAREN! ( TOR IDENTIFIER ( COMMA! IDENTIFIER)* TOR)? g_identifier RPAREN!
| IDENTIFIER^ LPAREN! (IDENTIFIER (COMMA! IDENTIFIER)*)? RPAREN!
;

g_priority
: PRIO^ INTEGER;

g_condition
: IF^ g_expression
;

g_actionblock
    : DO! BEGINBLOCK! (g_action)* ENDBLOCK!
;

//---------------------------------------
// Actions
//---------------------------------------
g_action
: FOREACH^ IDENTIFIER IN! g_identifier DO! (g_action)* ENDFOREACH!
| FOR^ IDENTIFIER IS! g_expression (DOTDOT|DOWNTO) g_expression DO! (g_action)* ENDFOR!
| IF^ g_expression THEN! (g_action)* (ELSE (g_action)* )? ENDIF!
| g_vardef SEMICOLON!
| g_identifier (IS^ g_expression)? SEMICOLON!
| g_exception
| g_fct_return
;

//---------------------------------------
// Eventdecl.
//---------------------------------------
g_eventdec
: IDENTIFIER^ (LPAREN! (g_eventparam (COMMA! g_eventparam)* )? RPAREN!)
    ;
```

```
g_eventparam
: TYPIFIER^ (LBRACKET! INTEGER DOTDOT! INTEGER RBRACKET!)?
;

//---------------------------------------
// Typen & Variablen
//---------------------------------------
g_typedef
: TYPIFIER IS^ g_type SEMICOLON!
;

g_type
: g_orig_type (LBRACKET^ g_arrayType )?
;

g_orig_type
: ( RECORD^ LPAREN! (g_vardec SEMICOLON!)+ RPAREN!
| ENUM^ BEGIN! ENUMVALUE (COMMA! ENUMVALUE)* END!
| SET^ BEGIN! g_type END!
| LIST^ LESS! g_type BIGGER!
| TYPIFIER^
    );

g_arrayType
: INTEGER  RBRACKET! (LBRACKET! INTEGER  RBRACKET!)*
;

//---------------------------------------
// Variablen
//---------------------------------------
g_vardec
: g_type IDENTIFIER^
;

g_vardef
: g_vardec (IS^ g_varinit)?
;

g_varinit
: g_expression
;

//---------------------------------------
// Expressions
//---------------------------------------
g_expression
: g_expr_level3 ( (AND^
|OR^
|XOR^) g_expression)?
    ;
```

```
g_expr_level3
: g_expr_level2 ( (EQUALS^
|NOT_EQUALS^
|LESS^
|BIGGER^
|BIGGER_EQUAL^
|LESS_EQUAL^
|SUPERSETOF^
|SUBSETOF^
|DISJOINT^) g_expr_level3)?
                ;

g_expr_level2
: g_expr_level1 ( (ADD^
|SUB^) g_expr_level2)?
;

g_expr_level1
: g_expr_level0 ( (MUL^
|DIV^
|MOD^
|UNION^
|INTER^
|DIFF^
|SYMMDIFF^) g_expr_level1)?
;

g_expr_level0
: g_expr_unary_num ( (EXP^) g_expr_level0)?
;

g_expr_unary_num
: SUB^ g_expr_unary_num
| g_expr_unary_logical
;

g_expr_unary_logical
: NOT^ g_expr_unary_logical
| g_expr_term
;

g_expr_term
: g_identifier
| g_constant
| LPAREN! g_expression RPAREN!
    ;

//--------------------------------------
// Identifier
//--------------------------------------
```

```
g_identifier
: IDENTIFIER^ ( g_id_params )? (g_id_indexes | g_id_tail)?
;

g_id_params
: LPAREN^ (g_expression (COMMA! g_expression)* )? RPAREN!
;

g_id_tail
:  DOT^ g_id_tail_element
;

g_id_tail_element
: IDENTIFIER^ (g_id_indexes | g_id_tail)?
;

g_id_indexes
: LBRACKET^ g_expression RBRACKET! (LBRACKET! g_expression RBRACKET!)* (g_id_tail)?
;

g_constant
: INTEGER
| FLOAT
| TRUE
| FALSE
| ENUMVALUE
| STRING
| g_set_const
;

g_set_const
: BEGIN^ g_constant (COMMA! g_constant)* END!
| TYPIFIER^ BEGIN! END!
;
```

Anhang B

Listing Sprachbeispiel

Falk Howar

Listing B.1 *foodmachine.eca*

```
Options:

  /* only perform rollbacks on errors */
  errorHandling: rollback;
  checkInvariantsAfter: externalEvents;

Typedef:

  /* all possible elements of a digitnumber */
  DigitElement = Enum{_top, _middle, _bottom, _left_upper,
                      _left_lower, _right_upper, _right_lower};

  /* a digit number is a set of DigitElements */
  DigitNumber = Set{DigitElement};

  /* state of a lamp */
  Lamp = Enum{_lamp_on, _lamp_off};

  /* a product */
  Product = Rec(
    Int id;
    Int price;
    Int count;
    Lamp empty;
  );

Events:

  refillProduct(Int[0 to 2],Int [1 to 5]);

  giveReturn();
```

```
  insertCoin(Int[0 to 4]);

  buyProduct(Int[0 to 2]);


Variables:

  /* the slots holding the food */
  Product[3] slots;

  /* constant digit display information */
  DigitNumber[10] digits;

  /* the display showing the credit */
  DigitNumber[3] creditDisplay;

  /* the slots holding the coins */
  Rec(
    Int value;
    Int count;
  )[5] coins;

  /* credit */
  Int credit = 0;

  Int[3] div;


Functions:


  function updateDisplay(Int value)
  begin
    Int temp;
    temp = value;
    for i = 0 to 2 do
        creditDisplay[i] = digits[ temp / div[i] ];
        temp = temp - (temp / div[i]) * div[i];
    endfor
  end

  extern function dropOrder(Int slot);

Init:

  /* put two
   coins into each coin reservoire */

  for i = 0 to 4 do
    coins[i].count = 2;
  endfor
```

```
/* value in cents */
coins[0].value = 10;
coins[1].value = 20;
coins[2].value = 50;
coins[3].value = 100;
coins[4].value = 200;

/* put products into vending machine */
for i = 0 to 2 do
  slots[i].count = 1;
  slots[i].empty = _lamp_off;
endfor

slots[0].id = 1;
slots[1].id = 2;
slots[2].id = 3;

slots[0].price = 120;
slots[1].price = 60;
slots[2].price = 80;

/* define the ten digit numbers */
digits[1] = {_right_upper, _right_lower};
digits[7] = digits[1] union {_top};
digits[3] = digits[7] union {_middle, _bottom};
digits[9] = digits[3] union {_left_upper};
digits[8] = digits[9] union {_left_lower};
digits[6] = digits[8] diff {_right_upper};
digits[0] = digits[8] diff {_middle};
digits[5] = digits[6] diff {_left_lower};
digits[4] = digits[9] diff {_top};
digits[2] = digits[8] diff {_left_upper, _right_lower};

/* the credit display shows zero */
creditDisplay[0] = digits[0];
creditDisplay[1] = digits[0];
creditDisplay[2] = digits[0];


/* division consts */
div[0] = 100;
div[1] = 10;
div[2] = 1;
```

Ruleset:

```
  /* the product in slot is filled up by amount refill */
  on refillProduct(slot,refill)
  if (slots[slot].count + refill < 21)
  do begin
    slots[slot].count = slots[slot].count + refill;
    slots[slot].empty = _lamp_off;
  end

  /* insert a coin into vending machine */
  on insertCoin(nr)
  if (coins[nr].count < 5 && credit < 300)
  do begin
    credit = credit + coins[nr].value;
    coins[nr].count = coins[nr].count + 1;
    updateDisplay(credit);
  end

  /* buy a product */
  on buyProduct(slot)
  if (credit - slots[slot].price >= 0
  && slots[slot].count > 0)
  do begin
    credit = credit - slots[slot].price;

    /* refresh display */
    updateDisplay(credit);

    /* create output */
    slots[slot].count = slots[slot].count - 1;
  end

  /* return money */
  on giveReturn()
  if (true)
  do begin
    Int retMoney;
    retMoney = credit;

    /* return the money */
    for i = 0 to 4 do
      coins[4-i].count = coins[4-i].count - (credit / coins[4-i].value);
      credit = credit - coins[4-i].value * (credit / coins[4-i].value);
    endfor

    /* refresh the display */
    creditDisplay[0] = digits[0];
    creditDisplay[1] = digits[0];
    creditDisplay[2] = digits[0];
  end
```

```
  /* turn the light on, when a product is sold out */
  on change( |slot| slots[slot].count)
  if (slots[slot].count == 0)
  do begin
    slots[slot].empty = _lamp_on;
    dropOrder(slot);
  end

Invariant:

  /* track errors at return of coins and */
  /* perform a rollback in case of an error */
  invar changeError
  begin
    for i = 0 to 4 do
      if (coins[i].count < 0) then
        exception("Coins empty");
      endif
    endfor
  end
```

Listing B.2 *Telefon-Basissystem*

```
/* Telefon-Basissystem */
/* POTS (Plain Old Telephone Service) */

Typedef:

/* Typ fuer die Beschreibung des Zustandes eines Telefongeraetes */
StateType = Enum{_idle, _dialwait, _hearing, _ringing, _busytone, _caller, _callee};

/* _idle:      Ruhezustand*/
/* _dialwait:  Warten auf Wahl einer Telefonnummer */
/* _hearing:   A hoert Wahlton des Anrufs an B */
/* _ringing:   Geraet A laeutet nach einem Anruf von B */
/* _busytone:  A hoert das Besetztzeichen */
/* _caller:    A kommuniziert als Anrufer mit B */
/* _callee:    A kommuniziert als Angerufener mit B */

Events:

/* Die Ereignisse sind bisher fuer ein Netz aus
drei Telefongeraeten formuliert. */
offhook (Int[0 to 2]);                  /* abheben */
dial (Int[0 to 2],Int[0 to 2]);         /* waehlen */
onhook (Int[0 to 2]);                   /* auflegen */
```

```
Variables:

StateType[3] state;                /* Zustand der Telefongeraete */
Int[3]     telephone;              /* Nummer des Telefongeraetes, */
                                   /* mit dem A kommuniziert */
                                   /* Wert -1, wenn kein Telefon */
                                   /* ausgewaehlt wurde */

Functions:

extern function offhookIdle (Int);
extern function offhookRinging (Int, Int);
extern function dialIdle (Int, Int);
extern function dialBusy (Int, Int) ;
extern function onhookCaller (Int, Int);
extern function onhookCallee (Int, Int);
extern function onhookHearing (Int, Int);
extern function onhookBusytone (Int);
extern function onhookDialwait (Int);

Init:

/* Variableninitialisierung */
state[0] = _idle;
state[1] = _idle;
state[2] = _idle;
telephone[0] = -1;
telephone[1] = -1;
telephone[2] = -1;

Ruleset:

/* Durch Abheben des Hoerers geht das Telefongeraet vom Ruhezustand */
/* in den Wartezustand ueber. */
on offhook(a)
if   (state[a] == _idle)
do begin
    offhookIdle(a);
    state[a] = _dialwait;
end

/* Anruf von A an B bei freier Leitung */
on dial(a,b)
if   ((!(a == b)) && (state[a] == _dialwait) && (state[b] == _idle))
do begin
    dialIdle(a,b);
    state[a]        = _hearing;
    telephone[a]    = b;
    state[b]        = _ringing;
    telephone[b]    = a;
end
```

```
/*Anruf von A an B bei besetzter Leitung */
on dial(a,b)
if   ((state[a] == _dialwait) && (!(state[b] == _idle)))
do begin
    dialBusy(a,b);
    state[a] = _busytone;
end

/* B hebt den Hoerer nach einem Anruf von A ab. */
/* Beginn eines Gespraechs zwischen A und B */
on  offhook(b)
if    ((state[b] == _ringing) && (!(telephone[b] == b))
      && (state[telephone[b]] == _hearing))
do begin
    offhookRinging(telephone[b],b);
    state[telephone[b]]    = _caller;
    state[b]               = _callee;
end

/* Der Anrufer A legt auf und beendet das Gespraech mit B. */
on  onhook(a)
if    ((state[a] == _caller) && (!(telephone[a] == a))
      && (state[telephone[a]] == _callee))
do begin
    onhookCaller(a,telephone[a]);
    state[a]                = _idle;
    state[telephone[a]]     = _busytone;
    telephone[telephone[a]] = -1;
    telephone[a]            = -1;
end

/* Der Angerufene B legt auf und beendet das Gespraech mit A. */
on  onhook(b)
if    ((state[b] == _callee) && (!(telephone[b] == b))
      && (state[telephone[b]] == _caller))
do begin
    onhookCallee(telephone[b],b);
    state[b]                = _idle;
    state[telephone[b]]     = _busytone;
    telephone[telephone[b]] = -1;
    telephone[b]            = -1;
end
```

```
/* Der Anrufer A legt auf, bevor B den Hoerer abgehoben hat. */
on  onhook(a)
if    ((state[a] == _hearing) && (!(telephone[a] == a))
      && (state[telephone[a]] == _ringing))
do begin
    onhookHearing(a,telephone[a]);
    state[a]                = _idle;
    state[telephone[a]]     = _idle;
    telephone[telephone[a]] = -1;
    telephone[a]            = -1;
end

/* A legt nach Hoeren des Besetztzeichens auf. */
on  onhook(a)
if    (state[a] == _busytone)
do begin
    onhookBusytone(a);
    state[a] = _idle;
end

/* A legt vor der Wahl einer Telefonnummer auf. */
on  onhook(a)
if    (state[a] == _dialwait)
do begin
    onhookDialwait(a);
    state[a] = _idle;
end
```

Im Kapitel 2.3 werden die Begriffe zur Beschreibung des Telefonsystems gemäß der folgenden Tabelle übertragen:

POTS (Plain Old Telephone Service)	Basistelefonsystem
TCS (Terminating Call Screening)	Abschirmung eingehender Anrufe
CFB (Call Forward on Busy)	Anrufweiterleitung
INTL (IN Teen Line)	Sperrung ausgehender Anrufe
idle	Ruhe
dialwait	warten
caller	Anrufer
callee	angerufen
ringing	läuten
hearing	Wahlton
busytone	Besetztton
offhook	abheben
onhook	auflegen
dial	wählen
talking	sprechen
time	Zeit
waitpin	wartenPIN
invalid	ungültig
dialgoodpin	richtigPIN
dialbadpin	falschPIN

Listing B.3 *Die Türme von Hanoi*

```
/* Die Tuerme von Hanoi */

Typedef:

/* Ein Turm wird als Liste aus ganzen Zahlen dargestellt */
/* Die Scheibengroesse waechst mit ihrer Ordnungszahl */
Tower = List<Int>;

Events:

/* Verschiebung der obersten Scheibe von Stapel a (erster Parameter) */
/* auf Stapel b (zweiter Parameter) */
move(Int[0 to 2], Int[0 to 2]);

Variables:

/* Es gibt insgesamt drei Tuerme */
/* Turm 0 = start, Turm 2 = ziel, Turm 1 = Hilfsstapel */
Tower[3] towers;

/* Variable zum Speichern von Scheibenindizes */
Int disc;

Functions:

/* d: verschobene Scheibe, st: Start, dest: Ziel */
extern function discMoved (Int, Int, Int);
extern function goal(Int);

Init:

/* Alle Scheiben werden auf den Start-Stapel gelegt. */
/* Die Groesse der Scheiben nimmt dabei mit der Position in der Liste ab. */
for i = 1 to 3
do
    append(towers[0], i);
endfor

disc = 0;

Ruleset:
/* Verschiebung einer Scheibe auf einen leeren Stapel */
on move(a,b)
if  ((a != b) &&  (isEmpty(towers[b])) && (!(isEmpty(towers[a]))))
do begin
     disc = removeHead(towers[a]);
     prepend(towers[b], disc);
     discMoved(disc,a,b);
end
```

```
/* Verschiebung einer Scheibe auf einen nichtleeren Stapel */
on move(a,b)
if ((a != b) && (!(isEmpty(towers[b]))) && (!(isEmpty(towers[a])))
   && (getHead(towers[a]) < getHead(towers[b])))
do begin
     disc = removeHead(towers[a]);
     prepend(towers[b], disc);
     discMoved(disc,a,b);

     if (isEmpty(towers[0]) && isEmpty(towers[1]) && (size(towers[2])==3))
     then
         goal(2);
     endif
end
```

Listing B.4 *Blockwelt*

```
/* Beispiel Blockwelt */

Events:
    /* Stapeln von Block a auf Block b */
    stack(Int [0 to 2],Int [0 to 2]);

    /* Block a von Block b herunternehmen */
    unstack(Int [0 to 2],Int [0 to 2]);

    /* Block a von Block b auf Block c verschieben */
    move(Int [0 to 2],Int [0 to 2],Int [0 to 2]);

Variables:

    /* Block liegt auf dem Tisch */
    Bool[3] ontable;

    /* Block a liegt auf Block b */
    Bool[3][3] pr_on;

    /* Ueber dem Block liegt kein weiterer Block */
    Bool[3] pr_clear;


Functions:

extern function stackRule(Int, Int);
extern function unstackRule(Int, Int);
extern function moveRule(Int, Int, Int);
```

```
Init:

  /* Zu Beginn liegen alle Bloecke einzeln auf dem Tisch */

  for i = 0 to 2 do
    ontable[i] = true;
    pr_clear[i] = true;
  endfor


Ruleset:

   on stack(x,y)
   if ((x != y) && ontable[x] && pr_clear[x] && pr_clear[y])
   do
   begin
      pr_on[x][y] = true;
      ontable[x] = false;
      pr_clear[y] = false;
      stackRule(x,y);
   end

   on unstack(x,y)
   if ((x != y) && pr_on[x][y] && pr_clear[x])
   do
   begin
      ontable[x] = true;
      pr_clear[y] = true;
      pr_on[x][y] = false;
      unstackRule(x,y);
   end


   on move(x,y,z)
   if ((x != y) && (x != z) && (y != z) && pr_on[x][y]
       && pr_clear[x] && pr_clear[z])
   do
   begin
      pr_on[x][z] = true;
      pr_clear[y] = true;
      pr_on[x][y] = false;
      pr_clear[z] = false;
      moveRule(x,y,z);
   end
```

Listing B.5 *TicTacToe*

```
Events:

  einsSetzeZeichen(Int [1 to 9]);
  zweiSetzeZeichen(Int [1 to 9]);

Variables:

  /* Konstanten */
  Int spielerEinsAmZug = 0;
  Int spielerZweiAmZug = 1;

  Bool[2] amZug /* = [all = false] */;

  Bool spielNichtVorbei = true;

  Int[9] spielfeld /* = [all = 0] */;

Init:

amZug[spielerZweiAmZug] = true;

/**     2 | 1 | 2
  *    ---------
  *      _ | 1 | _
  *    ---------
  *      1 | _ | _
  */
spielfeld[0] = 2;
spielfeld[1] = 1;
spielfeld[2] = 2;
spielfeld[4] = 1;
spielfeld[6] = 1;

Ruleset:

    /*
    on einsSetzeZeichen(x)
    if (amZug[spielerZweiAmZug] && spielNichtVorbei)
    do begin
      spielNichtVorbei = false;
      print("Ungueltiger Zug");
    end

    on zweiSetzeZeichen(x)
    if (amZug[spielerEinsAmZug] && spielNichtVorbei)
    do begin
      spielNichtVorbei = false;
      print("Ungueltiger Zug");
    end
    */
```

```
on einsSetzeZeichen(x)
if (amZug[spielerEinsAmZug]
&& spielNichtVorbei
&& spielfeld[x-1] == 0)
do begin
  spielfeld[x-1] =1;
  amZug[spielerEinsAmZug] = false;
  amZug[spielerZweiAmZug] = true;
end

on zweiSetzeZeichen(x)
if (amZug[spielerZweiAmZug]
&& spielNichtVorbei
&& spielfeld[x-1] == 0)
do begin
  spielfeld[x-1] = 2;
  amZug[spielerZweiAmZug] = false;
  amZug[spielerEinsAmZug] = true;
end


on einsSetzeZeichen(x)
if ((spielfeld[0] == 1 && spielfeld[1] == 1 && spielfeld[2] == 1)
||  (spielfeld[3] == 1 && spielfeld[4] == 1 && spielfeld[5] == 1)
||  (spielfeld[6] == 1 && spielfeld[7] == 1 && spielfeld[8] == 1)
||  (spielfeld[0] == 1 && spielfeld[3] == 1 && spielfeld[6] == 1)
||  (spielfeld[1] == 1 && spielfeld[4] == 1 && spielfeld[7] == 1)
||  (spielfeld[2] == 1 && spielfeld[5] == 1 && spielfeld[8] == 1)
||  (spielfeld[0] == 1 && spielfeld[4] == 1 && spielfeld[8] == 1)
||  (spielfeld[2] == 1 && spielfeld[4] == 1 && spielfeld[6] == 1))
do begin
  spielNichtVorbei = false;
  print("Spieler eins gewinnt");
end


on zweiSetzeZeichen(x)
if ((spielfeld[0] == 2 && spielfeld[1] == 2 && spielfeld[2] == 2)
||  (spielfeld[3] == 2 && spielfeld[4] == 2 && spielfeld[5] == 2)
||  (spielfeld[6] == 2 && spielfeld[7] == 2 && spielfeld[8] == 2)
||  (spielfeld[0] == 2 && spielfeld[3] == 2 && spielfeld[6] == 2)
||  (spielfeld[1] == 2 && spielfeld[4] == 2 && spielfeld[7] == 2)
||  (spielfeld[2] == 2 && spielfeld[5] == 2 && spielfeld[8] == 2)
||  (spielfeld[0] == 2 && spielfeld[4] == 2 && spielfeld[8] == 2)
||  (spielfeld[2] == 2 && spielfeld[4] == 2 && spielfeld[6] == 2))
do begin
  spielNichtVorbei = false;
  print("Spieler zwei gewinnt");
end
```

```
    on einsSetzeZeichen(x)
    if (spielNichtVorbei
    && spielfeld[0] > 0 && spielfeld[1] > 0 && spielfeld[2] > 0
    && spielfeld[3] > 0 && spielfeld[4] > 0 && spielfeld[5] > 0
    && spielfeld[6] > 0 && spielfeld[7] > 0 && spielfeld[8] > 0)
    do begin
      spielNichtVorbei = false;
      print("Unentschieden");
    end
```

Listing B.6 *Website*

```
Events:

  login();
  logout();
  gotoPage(Int[0 to 10]);

Variables:

  Int pageId = 2;
  Bool loggedIn = false;

Functions:

  extern function displayPage(Int);
  extern function displayLogin(Bool);

  function showPage()
  begin
    displayPage(pageId);
    displayLogin(loggedIn);
  end

Ruleset:

  on login()
  if pageId == 3 && !loggedIn
  do begin
    loggedIn = true;
    showPage();
  end

  on logout()
  if loggedIn && pageId == 8
  do begin
    loggedIn = false;
    pageId = 0;
    showPage();
  end
```

```
on gotoPage(newPage)
if newPage < 7 || loggedIn
do begin
  pageId = newPage;
  showPage();
end
```

Literaturverzeichnis

[1] Overview (java 2 platform se 5.0). http://java.sun.com/j2se/1.5.0/docs/api/index.html.

[2] Reflection - die java reflection api. http://wiklet.javacore.de/index.php/Die_Java_Reflection_API.

[3] Trail: The reflection api. http://java.sun.com/docs/books/tutorial/reflect/.

[4] Marc Aiguier, Karim Berkani, and Pascale Le Gall. Feature specification and static analysis for interaction resolution. in FM 2006: Formal Methods, Seiten 346–379, 2006.

[5] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[6] P. Butterwoth, A. Otis, and J. Stein. The gemstone object database management system. *Comm. ACM*, 34:64–77, 1991.

[7] Tsun S. Chow. Testing software design modeled by finite-state machines. pages 391–400, 1995.

[8] Jennifer Widom Elena Baralis. Better static rule analysis for active database systems. Technical report, Department of Computer Science, Stanfod University, 1996.

[9] Amy Felty and Kedar Namjoshi. Feature specification and automated conflict detection. *ACM Transactions on Software Engineering and Methodology*, 12(1):3–27, 2003.

[10] S. Gilmore and M. Ryan, editors. *Language Constructs for Describing Features*. Springer–Verlag, 2000.

[11] Didier GIRARD. Eclipse/websphere studio application developer : Retour d'experience. application-servers.com, 2001.

[12] Standish Group. Third quarter research report chaos report. Technical report, The Standish Group International Inc., 2004.

[13] Riedemann E. H., editor. *Testmethoden für sequentielle und nebenläufige Software-Systeme*. Teubner, 1997.

[14] Theo Härder and Erhard Rahm. *Datenbanksysteme - Konzepte und Techniken der Implementierung*. Springer-Verlag, 2 edition, 1999.

[15] Jae-Yoon Jung, Seung-Kyun Han, Jonghun Park, and Kang Chan Lee. Ws-eca: An eca rule description language for ubiquitous services computing. http://www.research.att.com/~rjana/MobEA-IV/PPT/WS-ECA%20060523.ppt.

[16] Gerti Kappel, Peter Lang, S. Rausch-Schott, and Werner Retschitzegger. Workflow management based on objects, rules, and roles. *Data Engineering Bulletin*, 18(1):11–18, 1995.

[17] Cynthia Andres Kent Beck. *Extreme Programming Explained: Embrace Change*. 1999.

[18] Ralf Nagel. das javaabc.

[19] Terrence Parr. Antlr documentation. http://www.antlr.org, 01 2005.

[20] Harald Raffelt, Bernhard Steffen, and Therese Berg. Learnlib: a library for automata learning and experimentation. In *FMICS '05: Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, pages 62–71, New York, NY, USA, 2005. ACM Press.

[21] Eike Hagen Riedemann. Software-testmethoden 2. http://ls10-www.cs.uni-dortmund.de/~riedemann/Homepage/SWTestenII.html, 2005.

[22] Dr. Norbert Ritter. Begleitmaterial zur vorlesung: Datenbanken und informationssysteme. http://vsis-www.informatik.uni-hamburg.de/teaching/ss-03/dis/Kapitel9-6.pdf, Sommersemester 2003.

[23] Dr. Markus Schlesinger. Alfred - konzepte und prototyp einer aktiven sicht zur automatisierung von geschäftsregeln, kapitel 3 - aktive datenbanksysteme. http://www.digital-publications.ch/schlesin/kap03.pdf, 2000.

[24] Markus Schlesinger. *ALFRED - Konzepte und Prototyp einer aktiven Sicht zur Automatisierung von Geschäftsregeln, Kapitel 3 - Aktive Datenbanksysteme*. PhD thesis, Rechts- und Wirtschaftswissenschaftlichen Fakultät der Universität Bern, 1999.

[25] F. Simon and T. Mohaupt, editors. *Code Quality Management*. dpunkt.Verlag, 2006.

[26] A. Spillner and T. Linz, editors. *Basiswissen Softwaretest*. dpunkt.Verlag, 2004.

[27] Wikipedia. Boolesche algebra. http://de.wikipedia.org/wiki/Boolesche_Algebra.

[28] Wikipedia. Concurent versions system. http://de.wikipedia.org/wiki/Concurrent_Versions_System.

[29] Wikipedia. Erweiterte backus-naur-form. http://de.wikipedia.org/wiki/EBNF.

[30] Wikipedia. Logische ausdrücke. http://de.wikipedia.org/wiki/Logischer_Operator.

[31] Wolf Zimmermann. Vorlesung Übersetzerbau. http://swt.informatik.uni-halle.de/lehre/2006ws/uebersetzerbau_192587_192643/index.de.php, 01 2007.