Extrapolation von Prozessmodellen aus Black-Box-Systemen mittels Automatenlernverfahren

Dissertation
zu Erlangung des Grades eines
Doktors der Naturwissenschaften
der technischen Universität Dortmund
an der Fakultät für Informatik

von
Harald Raffelt

Dortmund
2009

Tag der mündlichen Prüfung: 26.06.2009

Dekan: Prof. Dr. Peter Buchholz

Gutachter: Prof. Dr. Bernhard Steffen
Prof. Dr. Jakob Rehof

Danksagung

Wissenschaftliche Forschung ist ohne den intensiven Austausch mit Fachkollegen nicht denkbar. Häufig tragen viele Köpfe dazu bei, dass wissenschaftliche Resultate entstehen und im richtigen Licht dargestellt werden. Aus dem Bestreben, in der Theorie gut verstandene Automatenlernverfahren für praktische Anwendungen zugänglich zu machen, ist die LearnLib mit ihren Anwendungen entstanden. Auf dem Weg dorthin sind Ideen und Konzepte von vielen Wissenschaftlern eingeflossen.

Besonders bedanken möchte ich mich bei Prof. Dr. Bernhard Steffen und Prof. Dr. Ing. Tiziana Margaria-Steffen für die vielen Diskussionen und Anregungen. Ihre ansteckende Begeisterungsfähigkeit war immer eine Motivation für mich. Sie haben mich bei wichtigen Entscheidungen immer gut beraten.

Danken möchte ich auch für die produktive Zusammenarbeit mit Prof. Dr. Bengt Jonsson und Therese Bohlin (geb. Berg).

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

1.1 Hintergrund und Motivation

Informationsverarbeitende Systeme umgeben uns mittlerweile in fast allen Lebensbereichen. Durch die zunehmende Verbreitung von komponentenbasierter Softwareentwicklung, serviceorientierten Architekturen und der Allgegenwärtigkeit des Internets, werden diese Anwendungen immer leistungsfähiger und komplexer. Auch die einsetzende Verwendung modellbasierter Softwareentwicklung trägt dazu bei, dass immer größere und umfangreichere Softwaresysteme entwickelt werden können.

Diese Entwicklung bleibt nicht ohne Auswirkung auf die Qualitätssicherung der Projekte. Die Qualitätssicherung wird auf Grund der gestiegenen Anforderungen und zunehmender Komplexität immer anspruchsvoller: Ein Heer von Wissenschaftlern beschäftigt sich mit der Fragestellung, wie dieser Bereich der Informatik verbessert werden kann. Beim derzeitigen Stand der Entwicklung wird vorerst „Angemessen Testen!" die allgemein akzeptierte, praxistaugliche Antwort auf diese Fragestellung bleiben. Es existiert eine große Vielfalt von Testmethoden, die sich beispielsweise nach den zugrundeliegenden Prüfverfahren klassifizieren lassen [83]. Man unterscheidet statisches Testen, dynamisches Testen, kontrollfluss- und datenfluss-orientiertes Testen, funktionsorientiertes Testen und Regressionstests. Alternativ lassen sich Testmethoden auch nach dem Informationsstand über das zu testende System unterteilen. Black-Box-Tests, auch Funktionstests genannt, setzen keine Kenntnisse über den inneren Aufbau des zu testenden Systems vorraus. Sie werden in der Praxis häufig von speziellen Test-Abteilungen oder Test-Teams entwickelt. White-Box-Tests, auch Strukturtests genannt, nutzen die Struktur einer Implementierung aus und haben somit Kenntnisse über das zu testende System. White-Box-Tests sind Black-Box-Tests in dem Sinne überlegen, dass sie systematischer mögliche Fehler ausschließen können. Im Allgemeinen lässt sich festhalten, dass Beschreibungen und Spezifikationen von Systemen sehr hilfreich eingesetzt werden können, um effiziente Tests zu entwickeln.

Zahlreichen heutzutage verfügbaren informationsverarbeitenden Systemen fehlt eine angemessene Spezifikation. Dies liegt zum einen daran, dass sie aus vielen Kom-

ponenten bestehen, die zum Teil nicht genau spezifiziert sind. Zum anderen finden aus Zeitgründen Fehlerkorrekturen und andere Änderungen in letzter Minute kaum den Weg zurück in eine vorhandene Spezifikation.

Im Telekommunikationsbereich wird dies deutlich: Spezifikationen von Telekommunikationsprotokollen werden traditionell als natürlichsprachliche Textdokumente entwickelt. Es gibt keine formale Verbindung zur Implementierung. Diese Vorgehensweise schließt die Anwendung von Validierungstechniken wie modellbasiertem Testen und formaler Verifikation im Sinne von *Model Checking* aus. Darüber hinaus ist es schwierig, diese Art von Spezifikationen auf einem aktuellen Stand zu halten. Auch der gerade entstehende, modellgetriebene Softwareentwicklungsansatz mildert diese Problematik nur leicht, da typischerweise nur die statischen Eigenschaften eines Systems modelliert werden.

Das Fehlen von formalen Spezifikationen ist der wesentliche Grund dafür, dass die in der Wissenschaft gut verstandenen formalen Methoden aus den Bereichen modellbasiertes Testen und formale Verifikation nur sporadisch in der Praxis angewendet werden.

Techniken aus dem Bereich Automatenlernen, können dazu beitragen, diese Problematik zu überwinden. Diese Verfahren erlauben es im Kontext „Testen", Modelle, die das Verhalten einer Anwendung abbilden, automatisch zu generieren [67, 82]. Automatenlernalgorithmen erzeugen auf Basis von Beobachtungen endliche Automaten, die das Ein-/Ausgabeverhalten von Systemen wiedergeben oder zumindest approximieren. Bei reaktiven Systemen lassen sich diese Techniken immer dann gut anwenden, wenn die Systeme ein deterministisches Ein-/Ausgabeverhalten zeigen und sich als reguläre Sprachen abbilden lassen. In diesem Sinne kann Automatenlernen, in der theoretischen Informatik reguläre Inferenz genannt, als Extrapolationstechnik verstanden werden, die versucht, ein möglichst präzises reguläres Modell zu prognostizieren, welches mit den beobachteten Ausführungen des Systems konsistent ist und so das Verhalten einer Applikation abbildet.

Die negativen Resultate in diesem Gebiet lassen den praktischen Einsatz dieser Methoden unrealistisch erscheinen. Das Problem, aus einer Menge von positiven und negativen Beispielen einen entsprechenden minimalen Automaten zu generieren, ist NP-vollständig [8, 62]. Methoden, die durch Anfragen die Menge der Beispiele beeinflussen können, haben eine bessere, polynomielle Komplexität. Die enorme Menge der benötigten Beispiele sprengt aber auch bei diesen Ansätzen normalerweise praxistaugliche Dimensionen.

Diese Ausschlusskriterien für die Entwicklung eines praxistauglichen Lernsystems lassen sich mit Hilfe von Optimierungen, die zusätzliches Expertenwissen über die Struktur der Systeme ausnutzen, relativieren.

Kern dieser kumulativen Dissertation ist die LearnLib, eine Plattform zum aktiven Lernen von Automatenmodellen. Die LearnLib ist darauf ausgelegt, auf systematische Weise endliche Automatenmodelle von realen Systemen durch aktives Lernen zu extrapolieren. Mit ihrer Hilfe sollen theoretische Forschungsergebnisse aus dem Gebiet Automatenlernen für praktische Anwendungen zugänglich gemacht werden. Darüberhinaus stellt sie eine umfangreiche Plattform dar, die es erlaubt, mit unterschiedlichen Lernalgorithmen zu experimentieren und ihre Eigenheiten im Sinne

von Lernaufwand, Laufzeit und Speicherverbrauch statistisch zu analysieren. Dies gilt insbesondere für die in der LearnLib realisierten Erweiterungen und Optimierungen von Lernalgorithmen.

1.2 Überblick

Im folgenden Kapitel 2 wird der wissenschaftliche Beitrag dieser kumulativen Dissertation vorgestellt und die Veröffentlichungen in den Gesamtkontext eingeordnet.

Das Kapitel 3 stellt einige aktive und passive Automatenlernverfahren vor und diskutiert ihre Vor- und Nachteile.

Die LearnLib, eine Plattform zur Entwicklung und Evaluation von Anwendungen des Automatenlernens wird in Kapitel 4 vorgestellt. In Abschnitt 4.1 wird die Architektur der LearnLib beschrieben und in Abschnitt 4.2 die zugrundeliegenden aktiven Automatenlernverfahren. Darüber hinaus stellt das Kapitel in Abschnitt 4.3 anwendungsspezifische Optimierungen und die Art und Weise vor, wie Äquivalenz-Anfragen in der LearnLib approximiert werden.

In Kapitel 5 werden einige Anwendungen des Automatenlernens beschrieben, die auf Basis der LearnLib entwickelt wurden.

Das Kapitel 6 enthält eine kurze Zusammenfassung und stellt Weiterentwicklungsmöglichkeiten der LearnLib vor.

Im Anhang Abschnitt A finden sich einige formale Definitionen, auf die im Laufe der Arbeit zurückgegriffen wird.

Die Veröffentlichungen, dieser kumulativen Dissertation befinden sich im Anhang B und sind dort zusätzlich stichpunktartig zusammengefasst.

Kapitel 2

Wissenschaftlicher Beitrag

Die Qualitätssicherung ist ein wesentlicher Baustein im Softwareentwicklungsprozess. In der wissenschaftlichen Forschung gibt es eine große Bandbreite von gut untersuchten formalen Validierungstechniken und Verifikationstechniken. Real existierende Softwaresysteme verfügen aber typischerweise nicht über formale Spezifikationen, so dass sich ein Großteil dieser Methoden nicht direkt anwenden lässt. Im Bereich der theoretischen Informatik sind reguläre Inferenz-Techniken entwickelt worden, mit denen sich durch Beobachtung von außen Automatenmodelle zur Beschreibung von Black-Box-Systemen extrapolieren lassen.

Der wissenschaftliche Beitrag dieser Dissertation besteht darin, theoretisch fundierte Automatenlernverfahren für praktische Anwendungen zugänglich zu machen. Im Fokus liegt hierbei die Extrapolation von Automatenmodellen aus Black-Box-Systemen mit dem Ziel, unspezifizierte Systeme für die Anwendung von formalen, modellbasierten Verifikations- und Validierungstechniken zugänglich zu machen.

2.1 Ausgangslage

2.1.1 Verifikations- und Validierungstechniken

Formale Verifikations- und Validierungstechniken sind in der Forschung von hohem Interesse.

Im Bereich der Verifikationstechniken stellt *Model Checking* [12, 39, 101] den Status Quo dar. *Model Checking* ist ein Verfahren zur vollautomatischen Verifikation einer als Automatenmodell gegebenen Systembeschreibung gegen eine Spezifikation in Form einer Formel. Mit Hilfe von temporal-logischen Formeln lassen sich auf abstrakter Ebene Anforderungen an das Systems beschreiben. Das gegebene Modell wird daraufhin überprüft, ob es diese Anforderungen und spezifizierten Eigenschaften erfüllt.

Im Bereich der Validierung werden insbesondere Verfahren zum modellbasierten Testen [28] vorgeschlagen. Diese Methoden generieren Testfälle für Systeme auf

Basis von Automatenmodellen, welche die Systeme spezifizieren. Diese Vorgehensweise bietet eine Reihe von Vorteilen:

Zum einen ist es auf Grund des Spezifikationsmodells möglich, wohldefinierte Test-Überdeckungskriterien zu definieren. Zum anderen lassen sich durch Anwendung von formalen Verifikationsverfahren (*Model Checking*) auf einfache Weise Heuristiken realisieren, die nach bestimmten „typischen" Fehlern suchen [59, 60, 116, 115]. Darüber hinaus werden im Bereich *Conformance Testing* Verfahren entwickelt, welche die Konformität einer Implementierung zu einer als Modell gegebenen Spezifikation prüfen [37, 57, 126, 133].

2.1.2 Maschinelles Lernen

Maschinelles Lernen ist bereits seit längerer Zeit Gegenstand überwiegend theoretischer Forschung. Im Allgemeinen geht es in diesem Forschungsgebiet um die Fragestellung, wie sich Beschreibungen und Modelle aus Systemen erzeugen lassen. Im Bereich der Hard- und Software-Entwicklung liegt der Schwerpunkt zum einen auf der Synthese von statischen Eigenschaften, wie beispielsweise Invarianten [29, 55, 103], zum anderen auf der Gewinnung von Automaten-Modellen zur Beschreibungen des dynamischen Verhaltens eines Systems [42, 43, 44, 95, 108, 134].

Vereinfacht ausgedrückt stellt Modelllernen einen Prozess dar, der aus den Ein- und Ausgaben eines Systems ein Modell extrapoliert, welches die (zuvor unbekannten) internen Abläufe des untersuchten Systems abbildet.

Es gibt eine Reihe von negativen Resultaten die belegen, dass Automatenlernen ein schwieriges Problem ist. Bereits 1956 wurde von Moore [99] gezeigt, dass sich der Lernaufwand im Sinne von Ein- und Ausgabe im *Worst Case* exponentiell zu der Anzahl der internen Zustände verhält. Weiterhin ist bekannt, dass das Problem, aus einer Menge von positiven und negativen Beispielen einen entsprechenden minimalen Automaten zu generieren, NP-vollständig ist [8, 62]. Dies gilt auch für Annäherungen [109]. Dennoch lassen sich in der Praxis viele Modelle mit akzeptablem Aufwand erlernen. Einen guten Überblick über das Forschungsgebiet bieten die Arbeiten von Higuera [50] sowie von Parekh und Honavar [106].

Automatenlernverfahren wurden bereits zur Verifikation und Validierung von unspezifizierten Systemen vorgeschlagen [108, 65]. Diese Ansätze und seine theoretischen Grundlagen werden in der Veröffentlichung „Model-Based Testing of Reactive Systems: Model Checking" [20] vorgestellt. Bei der Arbeit an dieser Veröffentlichung fiel auf, dass die Praxistauglichkeit der entwickelten Methoden nur selten untersucht wurde. Die Publikationen in diesem Gebiet enthalten nur vereinzelt Fallstudien und die dabei simulierten, typischerweise randomisierten Automaten bestehen höchstens aus mehreren hundert Zuständen [17, 41, 119]. Fallstudien mit praxisrelevanten Modellen bzw. Kommunikationsprotokollen oder gar realen Systemen liegen in der Größenordnung noch deutlich darunter: zu wenig für einen industriellen Einsatz.

Es gibt einen Ansatz, Automatenlernverfahren auf reale Systeme anzuwenden, der neben einem Automatenlernenverfahren zusätzliches anwendungsspezifisches Ex-

pertenwissen ausnutzt [67, 72]. Die erzielten Ergebnisse vermitteln den Eindruck, dass auf diesem Weg die Praxistauglichkeit von Automatenlernverfahren erreicht werden kann.

2.2 Die Experimentier- und Anwendungsplattform

Mit dem Ziel, die Qualitätssicherung im Bereich der *Computer Telephony Integration* Systeme durch den Einsatz von Automatenlernverfahren zu verbessern, wurde ausgehend unseren bisherigen Ansätzen [68, 67, 71, 72, 102] ein Verfahren zur direkten Extrapolation von Mealy-Automaten entwickelt. Dieser Ansatz wird in der Veröffentlichung „Efficient Test-based Model Generation of Legacy Systems" [129] vorgestellt. Durch den Übergang von deterministischen endlichen Automaten zu Mealy-Automaten, ließen sich die schon zuvor betrachteten Telefonsysteme deutlich schneller lernen. Durch die Kooperation mit der Siemens AG konnte dieser Ansatz an einer realen Telefonanlage durchgeführt werden. Auf Grund von Managemententscheidungen beendete Siemens kurz darauf das Engagement, so dass das bereitgestellte Testequipment zurückgeben werden musste und die Arbeiten in diesem Bereich nicht fortgesetzt werden konnten.

Daher entschieden wir uns, von der Fokussierung auf ein spezielles Anwendungsgebiet Abstand zu nehmen und eine universell verwendbare Plattform für Anwendungen des Automatenlernens zu entwickeln: die LearnLib. Das Ziel bestand darin, eine modulare Bibliothek von Lernalgorithmen und Optimierungen zu entwickeln, mit der es möglich ist, Modelle von realen Systemen zu extrapolieren. Es sollte möglich sein, die Bausteine der Bibliothek in den unterschiedlichsten Szenarien zu kombinieren, um so auf experimentellem Wege die Eigenheiten der Lernalgorithmen und Optimierungen im Sinne von Lernaufwand, Laufzeit und Speicherverbrauch statistisch zu analysieren. Darüber hinaus sollte durch eine serviceorientierte Architektur gewährleistet werden, dass die Module der LearnLib durch andere Wissenschaftler als Dienst genutzt werden können. Die LearnLib wird erstmalig in der Veröffentlichung „LearnLib: A Library for Automata Learning and Experimentation" [113] vorgestellt. Die LearnLib umfasst die drei Bereiche Lernalgorithmen, anwendungsspezifische Optimierungen und approximative Äquivalenz.

2.2.1 Lernalgorithmen

Im Bereich der Lernalgorithmen sind in der LearnLib verschiedene Variationen von Angluins Algorithmus [10] für deterministische endliche Automaten (DFA) und Mealy-Automaten realisiert. Die Mealy-Lernalgorithmen basieren auf Nieses Algorithmus für „*input/output deterministic finite automata (IODFA)*" [102]. Der wesentliche Unterschied zwischen Mealy-Automaten und IODFAs besteht darin, dass Mealy-Automaten (siehe Definition A.2) bei Verarbeitung eines Eingabesymbols genau ein Ausgabesymbol erzeugen, IODFAs können jedoch nach jedem Eingabesymbol eine endliche Sequenz von Ausgabesymbolen erzeugen. Die LearnLib realisiert in gewissem Sinne also ein einfacheres, direkteres und allgemeingültigeres

Verfahren. Auf Grund unserer guten Erfahrungen durch den Übergang von deterministischen endlichen Automaten zu Mealy-Automaten, wurden im Rahmen eines gemeinsamen Forschungsprojekts mit der *Testing of Reactive Systems Group*, der *Uppsala University* zwei weitere Lernalgorithmen entwickelt. Diese werden in den Veröffentlichungen „Regular Inference for State Machines with Parameters" [18] und „Regular Inference for State Machines using Domains with Equality Tests" [19] detailliert vorgestellt.

2.2.2 Anwendungsspezifische Optimierungen

Klassische aktive Automatenlernverfahren sind wenig geeignet, Modelle von realen Systemen zu extrapolieren. Dies liegt zu einem großen Teil an der enormen Menge von Anfragen, welche beim Extrapolieren realer Systeme durch Ausführung von Testfällen beantwortet werden müssen. Um dieses Problem in den Griff zu bekommen, bietet die LearnLib eine Reihe von Filtertechniken an, die anwendungsspezifisches Expertenwissen ausnutzen, wodurch die Menge der Anfragen drastisch verringert werden kann. Diese Filtertechnologien basieren auf der Arbeit von Hungar et al. und Niese [72, 102]. Sie wurde bei der Entwicklung der LearnLib grundlegend überarbeitet und generalisiert, insbesondere im Bereich der partiellen Ordnung (unabhängige Aktionen) und der Symmetrie (siehe Abschnitt 4.3). Die ursprünglichen Verfahren generierten und speicherten alle, bezüglich partieller Ordnung und Symmetrie äquivalenten Anfragen in einer Datenstruktur. Dies führte zu einem immensen Speicherverbrauch, was die Extrapolation größerer Systeme nahezu unmöglich machte. In der LearnLib sind die Filter für partielle Ordnung und Symmetrie anders realisiert worden: Sie berechnen zu einer Anfrage die zugehörige lexikographisch kleinste äquivalente Anfrage, so dass für jede Äquivalenzklasse von Anfragen nur noch eine Anfrage gespeichert werden muss. Bei der Entwicklung dieser optimierenden Filter stellte sich die Frage, wie sich die Filter gegenseitig ergänzen und ob es eine „optimale Kombination" der Filter gibt. Die Wechselwirkungen der Filter untereinander wurden gründlich untersucht und in der Veröffentlichung „Analyzing Second-Order Effects Between Optimizations for System-Level Test-Based Model Generation" [90] vorgestellt. Einen umfassenden Gesamtüberblick über die anwendungsspezifischen Optimierungen der LearnLib und ihre Anwendung im Bereich *Computer Telephony Integration* gibt die Veröffentlichung „Knowledge-Based Relevance Filtering for Efficient System-Level Test-Based Model Generation" *(Journal)* [91].

2.2.3 Approximative Äquivalenz

Bei den in der LearnLib realisierten Lernalgorithmen handelt es sich um aktive Lernalgorithmen. Diese Verfahren benötigen ein allwissendes Orakel, mit dem sie abschließend ihre Hypothesen auf Äquivalenz zum unbekannten System überprüfen können. Bei den Äquivalenz-Anfragen handelt es sich aber um ein (im allgemeinen) unentscheidbares Problem. Durch reines Testen ist es nicht möglich, die Äquivalenz von zwei endlichen Automaten zu zeigen. Aus dem Bereich *Conformance Tes-*

ting sind Methoden bekannt, mit denen sich die Äquivalenz von Hypothesemodell und Implementierung approximativ entscheiden lässt. Die LearnLib implementiert zwei dieser Verfahren jeweils für DFAs und Mealy-Automaten: die W-Methode von Chow [37] und die Wp-Methode von Fujiwara et al. [57]. Bei der Realisierung dieser Algorithmen sind große Gemeinsamkeiten zwischen den Forschungsgebieten *Conformance Testing* und (aktives) Automatenlernen aufgefallen. Unsere Erkenntnisse in diesem Bereich haben wir in der Veröffentlichung „On the Correspondence Between Conformance Testing and Regular Inference" [16] zusammengefasst. Die Ähnlichkeiten zwischen den Gebieten gehen soweit, dass die Testfälle für die W-Methode mittlerweile direkt aus den Datenstrukturen der Lernalgorithmen erzeugt werden.

2.3 Anwendungen des Automatenlernens

Die LearnLib hat sich im Laufe der Zeit zu einer umfangreichen Experimentier- und Entwicklungsplattform für Anwendungen des Automatenlernens weiterentwickelt. Insbesondere ihre serviceorientierte, offene Architektur ermöglichte es, eine Reihe von Anwendungen zu realisieren.

2.3.1 *Requirements Engineering*

Das Tool Smyle stellt einen fortschrittlichen Ansatz zum *Requirements Engineering* [104] dar. Diese Applikation stellt die erste, nicht von uns entwickelte Anwendung auf Basis der LearnLib dar. Smyle [22] steht für „**S**ynthesizing **M**odels b**y** **L**earning from **E**xamples" und ist ein Werkzeug zur Synthese asynchroner, verteilter Implementierungsmodelle. Smyle ist in Java entwickelt und bindet die LearnLib als entfernten CORBA-Dienst an. Dies hat den Nachteil, das Smyle auf eine Internet-Verbindung zur LearnLib angewiesen ist, bietet aber den Vorteil, dass Smyle unmittelbar von Verbesserungen der LearnLib profitiert und entlastet gleichzeitig lokale Rechner vom Lernprozess, insbesondere beim Lernen großer Systeme. Die Anbindung von Smyle an die LearnLib auf Basis der jETI-Plattform wird in der Veröffentlichung „The LearnLib in FMICS-jETI" [92] beschrieben. Eine weitere Anwendung im Bereich *Requirements Engineering* ist der von uns in Zusammenarbeit mit der NASA weiterentwickelte R2D2C-Ansatz [69]. *R2D2C* steht für „Requirements to Design to Code" und stellt ein Verfahren dar, welches Systemanforderungen maschinell in formale Modelle transformieren soll, die möglichst genau die gewünschte Applikation reflektieren. Unsere Erweiterung dieses Ansatzes um ein automatisches Lernverfahren und seine Anwendung im Bereich biologischer Prozesse wird in der Veröffentlichung „Completing and Adapting Models of Biological Processes" [88] vorgestellt.

2.3.2 Testen von Web-Applikationen

Den größten Erfolg verzeichnet die LearnLib aber im Bereich Testen von Web-Applikationen. Die Idee besteht darin, Automatenlernverfahren zum Extrapolieren von Verhaltensmodellen realer Web-Applikationen einzusetzen, und dadurch Modelle und Testfälle für Applikation zu gewinnen. Dieser Ansatz wird in der Veröffentlichung „Dynamic Testing via Automata Learning" [114] vorgestellt. Web-Applikationen werden bei diesem Ansatz durch Mealy-Automaten abgebildet, die beim Verarbeiten einer Eingabe bzw. einer Benutzeraktion genau eine Ausgabe bzw. Web-Seite erzeugen. Der Ansatz erlaubt es, parallel zum Lernprozess interaktiv weitere Benutzeraktionen in das Alphabet des Lernalgorithmus hinzuzufügen. Im Gegensatz zu herkömmlichen Web-Crawlern und Link-Checkern, welche nur die statische Seitenstruktur einer Website erfassen, bildet der lernbasierte Ansatz das dynamische Verhalten einer Web-Applikation ab. Bei diesem Ansatz wurden die Ausgaben der Web-Applikationen nur unzureichend behandelt. Die Ausgaben der Web-Applikation definierten sich über den gesamten Inhalt der zurückgelieferten Web-Seiten. Nur über Konfigurationsdateien können bestimmte Teile der Web-Seiten, wie beispielsweise das aktuelle Datum, ausgeblendet werden. Diese Vorgehensweise ist zu kompliziert und zu unflexibel für einen produktiven Einsatz des Tools. Daher entschieden wir uns, unseren Ansatz zum dynamischen Testen mit klassischem Record- und Replay-Testen zu kombinieren. Dieser Ansatz wird in der Veröffentlichung „Hybrid Test of Web Applications with Webtest" [111] vorgestellt und ermöglicht einen fließenden Übergang von Record- und Replay-Testen zu dynamischem Testen: Die Testingenieure entwickeln wie bisher Testfälle mit Aktions-Schritten und Kontroll-Schritten, in die ihr Expertenwissen über die Anwendung einfließt. Die einzelnen Schritte der Testfälle bilden direkt das Alphabet für den Lernalgorithmus und die gewonnen Modelle stellen direkt ausführbare, randomisierte Testfälle für die Applikation dar.

Das, derzeit im Reviewprozess befindliche *invited Jounal-Paper* „Dynamic Testing via Automata Learning" *(Journal)* [112] stellt dass von uns entwickelte Werkzeug zum Testen von Web-Applikationen detailliert vor. Es wird auch im Abschnitt 5.7 skizziert.

2.4 Eigener Beitrag

Aus dem Bestreben, in der Theorie gut verstandene Automatenlernverfahren für praktische Anwendungen zugänglich zu machen, ist die LearnLib mit ihren Anwendungen entstanden.

Die LearnLib ist fast vollständig vom Autor der vorliegenden Arbeit entwickelt und realisiert worden. Dies umfasst:

- die serviceorientierte Architektur der LearnLib, die sich in der Aufteilung in LearnLib-Plugin (Abschnitt 4.1.1) und LearnLib-Kern (Abschnitt 4.1.2) niederschlägt,

- die zwei sich ergänzenden Modellierungs-Modi der LearnLib, dargestellt in Abschnitt 4.1.1.1 und 4.1.1.2,
- die Auswahl, Optimierung und Realisierung der Lernverfahren, welche in Abschnitt 4.1.1 vorgestellt werden. Darüber hinaus war der Autor maßgeblich an der Realisierung und Entwicklung des Algorithmus zum Lernen von Automaten mit Booleschen Parametern [18] beteiligt.
- die Entwicklung und Implementierung der in Abschnitt 4.3 vorgestellten anwendungsspezifischen Optimierungen. Hervorzuheben ist die vom Autor erzielte Generalisierung dieser optimierenden Filter und deren Übertragung auf den Mealy-Fall sowie ihre Speicherplatz-effiziente Realisierung durch Normalisierung.
- das Desing der in Abschnitt 4.4 beschriebenen Äquivalenz-Approximation.

Bei der Implementierung der LearnLib wurde stets auf Effizienz geachtet. Die Implementierung der LearnLib benötigt zu simulierten Lernen eines DFAs mit 100 Zuständen und 25 Symbolen heute im Durchschnitt 191 Millisekunden. Eine ähnliche Arbeit aus dem Jahre 2003 benötigte für diese Form von Modellen ca. 1 Stunde [17]. Das größte bisher mit der LearnLib gelernte Modell besteht aus 39974 Zuständen bei 51 Symbolen. Hierbei wurden in etwas mehr als 3 Stunden 696 Millionen Anfragen von einem Modell des *Plain Old Telephone Systems* aus der OPEN/Caesar Tool-Suite [58] simuliert ausgeführt.

Darüber hinaus wurde vom Autor ein Großteil der experimentellen Fallstudien geplant, durchgeführt und ausgewertet.

Die Webtest-Umgebung wurde darüber hinaus vom Autor entworfen und in großen Teilen realisiert. Dies umfasst die Anforderungsanalyse für die Testbausteine, ihr Design und ihre Realisierung einschließlich der graphischen Benutzeroberfläche, sowie die Umsetzung des Record- und Replay-Modus und des lernbasierten dynamischem Testens, einschließlich der Möglichkeit, Testfälle verteilt auszuführen. Die Webtest-Umgebung wird in Abschnitt 5.7 zusammengefasst vorgestellt.

Kapitel 3

Passive und aktive Lernverfahren

Bei der Entwicklung einer Experimentier- und Entwicklungsplattform für Anwendungen des Automatenlernens stellt sich zunächst die Frage, welche Automatenlernverfahren sich für das anvisierte Anwendungsgebiet, die Extrapolation von Modellen realer Black-Box-Systeme, eignen. Dieses Kapitel begründet die Entscheidung, nur Variationen eines Automatenlernverfahrens in die LearnLib zu integrieren, welche in Kapitel 4 vorgestellt wird.

Automatenlernverfahren lassen sich ganz grob in zwei Arten aufteilen. Man unterscheidet zwischen passivem Lernen, bei dem lediglich Beobachtungen analysiert, aber keine Aktionen veranlasst werden, und aktivem Lernen, bei dem gezielt Beobachtungen durch Anfragen vorgenommen werden. Das Konzept des aktiven Lernens wurde 1981 von Angluin eingeführt [9]. Im Falle von deterministischen endlichen Automaten lassen sich mit einer polynomiellen Anzahl von *Membership* und *Equivalence Queries* Automatenmodelle gewinnen [10]. *Membership Queries* prüfen, ob Worte von dem zu lernenden DFA akzeptiert werden. *Equivalence Queries* prüfen, ob die Hypothese des Lernalgorithmus mit dem zu lernenden DFA äquivalent ist und liefern gegebenenfalls ein Gegenbeispiel. Bei der Betrachtung des *Trade-Off*, zwischen *Membership* und *Equivalence Queries* [15] lässt sich im wesentlichen feststellen, dass eine exponentielle Anzahl von *Membership Queries* benötigt wird, um eine *Equivalence Query* zu ersetzen.

3.1 Passive Automatenlernverfahren

Im Bereich der theoretischen Forschung wurden eine Reihe von passiven Automatenlernverfahren entwickelt. Diesen Ansätzen ist gemein, dass sie ein gegebenes Trainingsset von positiven Beispielen und häufig auch negativen Beispielen analysieren und daraus endliche Automatenmodelle ableiten. Viele dieser Ansätze arbeiten mit einem statischen, vorgegebenen Trainingsset, es gibt aber auch eine Reihe inkrementeller Methoden.

Bei den passiven Ansätzen haben sich drei unterschiedliche Richtungen herausgebildet, die im Folgenden vorgestellt werden.

3.1.1 Neuronale Netze

Es gibt eine Reihe von Arbeiten [40, 48, 136, u. a.] die basierend auf rekurrenten, neuronalen Netzen[1] reguläre Automatenstrukturen lernen können. Diese Verfahren definieren zunächst eine neuronale Netzwerkarchitektur und analysieren dann, nach der passiven Trainingsphase, die internen Aktivitäten und Neuronen, um daraus Zustände und Transitionen des gesuchten Automaten abzuleiten. Die Ansätze unterscheiden sich im wesentlichen nur darin, wie die verborgenen internen Neuronen analysiert werden, um daraus Zustandsinformationen abzuleiten. Die Analyse der internen Strukturen von neuronalen Netzen ist in der neuronalen Netzwerk-Community umstritten [44], da typischweise nur dem Ein-/Ausgabeverhalten der Netze Relevanz zugestanden wird und nicht dem inneren Aufbau.

3.1.2 Stochastische Ansätze

Stochastische Ansätze basieren häufig auf *Hidden Markov* Modellen. Diese Modelle entsprechen Mealy-Maschinen, bei denen die Transitionsfunktion und die Ausgabefunktion durch Wahrscheinlichkeitsverteilungen ersetzt sind. In diesen Modellen gibt es also Wahrscheinlichkeiten für die Transition von einem Zustand in einen anderen und für die dabei erzeugten Ausgaben.

Es gibt eine Reihe von Ansätzen, die *Hidden Markov* Modelle lernen. Die grundlegende Idee des Ansatzes von Cook et al. [43] besteht darin, die Wahrscheinlichkeit von Teilsequenzen in dem zugrundeliegenden Trainingsset auszunutzen. Bei diesem Ansatz wird berechnet, welche Teilsequenzen mit welcher Wahrscheinlichkeit aufeinander folgen. Auf Basis der so gewonnen Daten lässt sich ein endlicher Automat erstellen, der nur Sequenzen mit einer Wahrscheinlichkeitsverteilung akzeptiert, die bis auf einen vorgegebenen Schwellenwert mit der gemessenen Wahrscheinlichkeitsverteilung übereinstimmt. Andere Ansätze wie beispielsweise der *Alergia* Algorithmus von Carrasco und Oncina [35] für einfachere stochastische Prozesse oder der Ansatz von Stolcke und Omohundro [130], basieren auf ähnlichen Prinzipien und setzen darüber hinaus noch weitere Methoden zur Vereinigung und Identifikation von Zuständen der Modelle ein.

3.1.3 Automatentheoretische Ansätze

Es gibt eine ganze Reihe von passiven automatentheoretischen Ansätzen zum Lernen von Automatenmodellen. Zwei der bekanntesten werden im Folgenden vorgestellt.

Der Ktail-Algorithmus [44] ist ein rein algorithmischer Ansatz und basiert auf den Arbeiten von Feldman und Bierman [56]. Als Eingabe verwendet der Algorithmus eine vorgegebene Menge von (positiven) Beispielsequenzen S, die vom unbekannten Automaten akzeptiert werden. Die zentrale Idee des Ktail-Algorithmus

[1] Rekurrente neuronale Netze erlauben Rückkopplungen zwischen den Neuronen.

besteht darin, dass sich jeder Zustand des Systems durch sein zukünftiges Verhalten charakterisieren lässt. Ein Zustand wird hierbei festgelegt durch ein Präfix p aus der Menge P aller Präfixe von Sequenzen aus S und den bezüglich S möglichen „Zukünften" von p. Die Zukünfte definieren sich hierbei durch alle Verlängerungen von p mit einer Maximallänge k, die in P enthalten sind. Der Algorithmus erzeugt einen nichtdeterministischen endlichen Automaten, dessen Zustände den Äquivalenzklassen einer entsprechenden Relation auf P entsprechen. Zwei Präfixe $p, p' \in P$ sind hierbei genau dann äquivalent, wenn für alle Sequenzen $t \in \Sigma^*$ mit Länge $|t| \leq k$ die Worte pt und $p't$ entweder beide in P enthalten sind oder beide nicht enthalten sind.

Der Parameter k des Algorithmus steuert den Abstraktionsgrad des Algorithmus. Ein großes k führt zu einer geringen Abstraktion, es werden fast ausschließlich die Beispielsequenzen aus S als positiv erkannt. Ein kleines k führt zu starker Abstraktion und der resultierende Automat kann fast beliebig vom zu lernenden System abweichen. Typische Werte für k liegen zwischen 2 und 4 [44, 85, 84, 96, 118].

Der RPNI-Algorithmus [105] von Oncina and García und seine inkrementelle Weiterentwicklung von Dupond [52] verwenden eine Menge von positiven und negativen Beispielen S^+ und S^-. Diese Algorithmen erzeugen in polynomieller Zeit einen deterministischen endlichen Automaten, der zu den vorgegebenen Beispielen S^+ und S^- konsistent ist.

Die Algorithmen konstruieren zunächst einen *Prefix Tree* Automaten, der genau die Sequenzen aus S^+ akzeptiert. Die Zustände des *Prefix Tree* Automaten werden aufsteigend bezüglich einer lexikographischen Ordnung ihrer Zugangssequenzen durchnummeriert. Dieser Automat stellt die aktuelle Hypothese des Algorithmus dar, seine Zustände werden auf systematische Weise verschmolzen, um so eine Lösung zu finden, die auch die negativen Beispiele aus S^- berücksichtigt. Hierzu wird in aufsteigender Reihenfolge versucht, einen Zustand q_i mit einem anderen Zustand q_j zu verschmelzen. Ist der hierbei entstehende Automat konsistent mit den negativen Beispielen S^-, so wird dieser Automat zur neuen Hypothese und der Algorithmus versucht den nächsten Zustand zu verschmelzen.

Unter der Voraussetzung, dass die Beispielmenge S eine sogenannte charakteristische Teilmenge von Sequenzen (*Characteristic Set*) für den zu lernenden Automaten enthält, erzeugt der RPNI-Algorithmus ein kanonisches Automatenmodell des Systems [105]. Ein *Characteristic Set* $S = S^+ \cup S^-$ hat die Eigenschaften, dass

1. die Menge der positiven Beispiele S^+ strukturell vollständig ist, was bedeutet, dass es jede Transition des unbekannten Systems mit einer Sequenz überdeckt wird und

2. die Menge der negativen Beispiele S^- in der Lage ist, je zwei Zustände des *Prefix Tree* Automaten zu trennen, die nicht äquivalent sind.

3.1.4 Passive Lernverfahren in der Praxis

Mit passiven Lernverfahren ist es im Allgemeinen nicht möglich, ein exaktes Modell eines unbekannten endlichen Automaten zu lernen. Dies gelingt nur, wenn den Lernverfahren hinreichend große, gut strukturierte Mengen von Beispielen präsentiert werden, wie beispielsweise einem *Characteristic Set*. In der Praxis lassen sich diese Arten von Trainigsmengen aber nur schwer zusammenstellen. Im Sinne der Qualitätssicherung scheint es ein aussichtsloses Unterfangen zu sein, durch reines Beobachten und Protokollieren von Ausführungen ein gut strukturiertes Trainingsset gewinnen zu wollen. Es ist zwar leicht, große Mengen von Trainingsdaten aufzuzeichnen aber es ist sehr unwahrscheinlich, dass hierbei gut strukturierte Beispiele entstehen, die möglichst alle alternativen Ausführungen eines Systems abdecken.

Passive Lernverfahren werden hauptsächlich in Bereichen eingesetzt, wo exakte Modelle nicht zwingend erforderlich sind. Beispielsweise zur Mustererkennung [32], im Bereich der Computerlinguistik [2, 98], der Spracherkennung [7] und in der Bioinformatik [27].

Es gibt aber auch Ansätze, passive Lernverfahren trotz ihrer gravierenden Einschränkungen in anderen Anwendungsgebieten einzusetzen: beispielsweise zur Gewinnung von Verhaltensmodellen von Softwaresystemen [43, 85]. Der Grund ist, dass passive Verfahren schon mit wenig Trainingsdaten brauchbare Ergebnisse erzielen können. Aktive Verfahren, die im Folgenden vorgestellt werden, erlauben die exakte Abbildung eines Modells, benötigen aber recht umfangreiche Trainingsdaten.

3.2 Aktive Automatenlernverfahren

Das Konzept des aktiven Lernens ist relativ weit verbreitet und es gibt eine Vielzahl von Arbeiten, die mit unterschiedlichen Arten von Anfragen Lernalgorithmen beschreiben. Büchi-Automaten lassen sich mit Hilfe von Anfragen über unendliche Worte lernen [87]. Mealy-Automaten lassen sich auf ähnliche Weise lernen wie DFAs [102, 129]. Zwei-Band-Automaten lassen sich ebenso mit einer polynomiellen Anzahl von Anfragen lernen [135]. Unter Zuhilfenahme von strukturellen Informationen ist es sogar möglich, kontextfreie Sprachen zu lernen [123, 124]. Die meisten dieser Arbeiten basieren auf dem Algorithmus von Angluin, welcher im Folgenden vorgestellt wird.

3.2.1 Angluins Algorithmus

Angluins Algorithmus geht davon aus, dass das zu lernende System als deterministischer endlicher Automat $\mathsf{M} = (S, s_0, \Sigma, \delta, F)$ (Def. A.1) modelliert werden kann. Der Algorithmus stellt zwei Arten von Anfragen an ein Orakel (auch Lehrer genannt) :

- ***Membership Queries*** bestehen darin, das Orakel zu Fragen, ob Worte $w \in \Sigma^*$ von dem unbekannten Automaten M akzeptiert werden: $MQ(w) \Leftrightarrow \sigma(w) \in F$

- ***Equivalence Queries*** fragen das Orakel, ob ein extrapoliertes Hypothesemodell M' äquivalent zum unbekannten Automaten M ist. Das Orakel antwortet mit „Ja" falls die Modelle übereinstimmen, andernfalls liefert das Orakel ein Gegenbeispiel u aus der symmetrischen Differenz der von M und M' erkannten Sprachen: $u \in (\mathscr{L}(\mathsf{M}) \backslash \mathscr{L}(\mathsf{M}') \cup \mathscr{L}(\mathsf{M}') \backslash \mathscr{L}(\mathsf{M}))$.

Angluins Algorithmus stellt solange *Membership Queries*, bis er ein Hypothesemodell bilden kann. Das Hypothesemodell wird dann mit einer *Equivalence Query* überprüft. Liefert das Orakel kein Gegenbeispiel, terminiert der Algorithmus. Liefert das Orakel ein Gegenbeispiel, dann verwendet es der Algorithmus, um mit weiteren *Membership Queries* seine Hypothese zu überarbeiten.

Angluins Algorithmus verwaltet seine Beobachtungen in einer *Observation Table* T. Die Spalten der Tabelle sind mit Elementen aus dem sogenannten *Distinguishing Set* E beschriftet. Das *Distinguishing Set* ist eine suffixabgeschlossene (Def. A.4) Menge von Worten $e \in E \subset \Sigma^*$ und dient dazu, Zustände des Hypothesemodells zu trennen. Das *Distinguishing Set* enthält initial nur das leere Wort ε, es wird vom Algorithmus erweitert. Die Zeilen der Tabelle teilen sich in einen sogenannten oberen und unteren Teil auf. Die Zeilen des oberen Teils der *Observation Table* sind mit Elementen aus dem sogenannten *Access Set* S beschriftet. Das *Access Set* ist eine präfixabgeschlossene (Def. A.3) Menge von Worten $s \in S \subset \Sigma^*$. Es enthält Zugangssequenzen zu den identifizierten Zuständen des Hypothesemodells. Initial enthält das *Access Set* S nur das leere Wort ε, im Laufe des Algorithmus wird es entsprechend erweitert. Der untere Teil der Tabelle ist mit Elementen aus $S.\Sigma$ beschriftet, hier werden die Transitionen des Automaten abgebildet. Für jede Sequenz $s \in S \cup S.\Sigma$ und jeden Suffix $e \in E$, gibt der Tabelleneintrag $T(s,e)$ an, ob die Sequenz se akzeptiert wird oder nicht. Die unterschiedlichen Zeilen der Tabelle charakterisieren die Zustände des Hypothesemodells, mit ihrer Hilfe lässt sich eine Äquivalenzrelation $\cong$ über den Sequenzen $s \in S \cup S.\Sigma$ definieren. Für zwei Sequenzen $s, s' \in S \cup S.\Sigma$ sei $s \cong s' \Leftrightarrow \forall e \in E.\, T(s,e) = T(s',e)$. Die Äquivalenzklassen von $\cong$ stellen die Zustände des Hypotheseautomaten dar. Um auf Basis von S und $\cong$ einen vollständigen und deterministischen Automaten erzeugen zu können, setzt Angluin zwei Bedingungen vorraus:

$$\begin{array}{ll} \text{Vollständigkeit:} & \forall s \in S \wedge \forall a \in \Sigma \wedge \exists s' \in S.\, sa \cong s' \\ \text{Konsistenz:} & \forall s, s' \in S \wedge \forall a \in \Sigma.\, s \cong s' \Rightarrow sa \cong s'a \end{array}$$

Die Vollständigkeit stellt sicher, dass sich für jede Äquivalenzkasse von $\cong$ und jedes Symbol $a \in \Sigma$ eine entsprechende Transition definieren lässt. Die Konsistenz stellt sicher, dass diese Transitionen einen eindeutig bestimmten Nachfolger besitzen.

Angluins Algorithmus versucht die Vollständigkeit und Konsistenz seiner Beobachtungen zu gewährleisten. Hierzu muss er die *Observation Table* mit Hilfe von *Membership Queries* $MQ(se)$ für jede Sequenz $s \in S \cup S.\Sigma$ und jedes Suffix $e \in E$

komplettieren. Jedes Mal, wenn die *Observation Table* nicht vollständig ist, wird das *Access Set S* um entsprechende Sequenzen, welche die fehlenden Äquivalenzklassen repräsentieren, erweitert. Jedes Mal, wenn die Tabelle nicht konsistent ist, wird das *Distinguishing Set* mit einem Suffix erweitert, welches dazu führt, dass die inkonsistente Äquivalenzklasse in zwei neue konsistente Äquivalenzklassen aufgespalten wird. Immer, wenn der Algorithmus die Vollständigkeit und Konsistenz der Beobachtungen hergestellt hat, wird ein entsprechendes Hypothesemodell $\mathsf{M}' = \left(S', s'_0, \Sigma', \delta', F'\right)$ gebildet und mit Hilfe einer *Equivalence Query* überprüft. Liefert das Orakel auf diese Anfrage ein Gegenbeispiel w, so wird dieses mit allen seinen Präfixen in das *Access Set S* eingefügt, was dazu führt, dass die Beobachtungen nicht mehr konsistent sind.

Angluins Algorithmus benötigt zu Lernen eines Systems mit n Zuständen und k Symbolen $O\left(kn^2m\right)$ *Membership Queries* und $O(n)$ *Equivalence Queries*, unter der Annahme, dass m die Länge des größten Gegenbeispiels ist.

3.2.2 Weitere aktive Lernalgorithmen

Es gibt noch andere weitere Lernalgorithmen, die mit *Membership Queries* und *Equivalence Queries* arbeiten. Beispielsweise der Algorithmus von Kearn und Vazirani [76], welcher auf *Discrimination Trees* basiert oder der *Reduced Observation Table* genannte Ansatz von Rivest und Schapire [120]. Wie Balcázar et al. zusammenfassend darstellen, benötigen beide Algorithmen weniger *Membership Queries* als Angluins Algorithmus [14]. Bei einer Alphabetgröße von $k = |\Sigma|$, einer Automatengröße von n und einer Maximallänge m für Gegenbeispiele benötigen diese Algorithmen nur $O\left(kn^2 + n\log m\right)$ Anfragen in Gegensatz zu $O\left(kn^2m\right)$ bei Angluins Algorithmus. Die *Worst Case* Abschätzung für die Anzahl der *Equivalence Queries* ist bei allen drei Algorithmen $O(n)$. Der Algorithmus von Kearn und Vazirani benötigt genau n *Equivalence Queries,* Angluins Algorithmus und der Algorithmus von Rivest und Schapire kommen in der Praxis häufig mit weniger *Equivalence Queries* aus.

Da sich beim Lernen von realen System *Equivalence Queries* prinzipiell nicht beantworten lassen, wurde darauf verzichtet den Algorithmus von Kearn und Vazirani in der LearnLib umzusetzen. Die Ideen von Rivest und Schapire wurden hingegen aufgegriffen, ihre Umsetzung ist im Abschnitt 4.2.1 dargestellt.

Es gibt noch einen weiteren interessanten Algorithmus von Rivest und Schapire. Bei diesem Ansatz [119] werden sogenannte *Homing Sequences* eingesetzt, mit deren Hilfe es möglich ist, dynamisch eine sehr lange Me*mbership Query* zu erzeugen, die es erlaubt, mit Wahrscheinlichkeit $1-\varepsilon$ einen endlichen Automaten exakt zu lernen. Für den praktischen Einsatz ist dieser Algorithmus aber ungeeignet:

- Die Gesamtlänge der Me*mbership Query* ist beschränkt durch $O\left(n^3\left(n+m\right)\left(n\log\left(n/\varepsilon\right)\right)+kn+\log m\right)$.
- Die Anzahl der *Equivalence Queries* ist beschränkt durch $O\left(n^2\right)$.

3.2.3 Aktive Lernverfahren in der Praxis

Aktives Lernen ist fast ausschließlich Gegenstand theoretischer Forschung. Hinweise auf die Praxistauglichkeit der entwickelten Methoden gibt es nur selten. Veröffentlichungen enthalten nur vereinzelt Fallstudien und die hier simulierten, typischerweise randomisierten Automaten bestehen höchstens aus mehreren hundert Zuständen [17, 41, 119]. Fallstudien mit praxisrelevanten Modellen, wie bzw. Kommunikationsprotokollen oder gar realen Systemen, liegen in der Größenordnung noch deutlich darunter. Das ist zu wenig für einen industriellen Einsatz. Dies liegt zu einem großen Teil an dem immensen Lernaufwand. Aus theoretischer Sicht handelt es sich bei Angluin's Algorithmus L^* mit Komplexität $O(kn^3)$ *Membership Queries* und $O(n)$ *Equivalence Queries* [10] um einen effizienten Algorithmus. Bei dieser Betrachtungsweise wird aber verschwiegen, dass sich im Allgemeinen *Equivalence Queries* nicht realisieren lassen. Die Äquivalenz von Spezifikations-Modell und Implementierung lässt sich nur approximieren. Aus dem Bereich *Conformance Testing* sind hierfür Methoden bekannt [37, 57], die bei Kenntnis der Anzahl der zusätzlichen Zustände z einer Implementierung mit exponentiellem Aufwand $O(k^z)$ die Äquivalenz zeigen können. Kleine praktische Probleme mit $n \geq 100$ Zuständen und $k \geq 10$ Symbolen im Eingabealphabet lassen sich damit aber oft nicht in angemessener Zeit lernen.

Im Gegensatz zu passiven Automatenlernverfahren, siehe Abschnitt 3.1, haben aktive Automatenlernverfahren die Möglichkeit, mit Hilfe von Anfragen an einen sogenannten Lehrer, aktiv die Ausgestaltung der „Trainingdaten" zu beeinflussen. Die meisten aktiven Lernverfahren basieren auf dem Ansatz von Angluin [10] und sind in der Lage, exakte Modelle von Systemen zu extrapolieren.

Kapitel 4

Die LearnLib

Die LearnLib ist ein Client-Server-System zum aktiven Lernen von Automatenmodellen. Ursprünglich war die LearnLib darauf ausgelegt, auf systematische Weise endliche Automatenmodelle von realen Systemen durch aktives Lernen zu extrapolieren. Mittlerweile hat sie sich zu einer umfangreichen Plattform weiterentwickelt, die es auch erlaubt, mit unterschiedlichen Lernalgorithmen zu experimentieren und ihre Eigenheiten im Sinne von Lernaufwand, Laufzeit und Speicherverbrauch statistisch zu analysieren. Dies gilt insbesondere für die in der LearnLib realisierten Erweiterungen und Optimierungen von Lernalgorithmen. Die LearnLib wird in der Veröffentlichung „LearnLib: A Library for Automata Learning and Experimentation" [113] detailliert vorgestellt.

4.1 Der Client-Server Ansatz

Die LearnLib besteht aus einem Server-Kern und dem LearnLib-Plugin, einer graphischen Benutzeroberfläche, die als Plugin für das jABC realisiert ist. Im im folgenden Abschnitt 4.1.1 wird das LearnLib-Plugin vorgestellt. Anschließend folgt eine Beschreibung des LearnLib Kerns in Abschnitt 4.1.2, welcher auch die realisierten Algorithmen skizziert.

4.1.1 Das LearnLib-Plugin

Das LearnLib-Plugin ist die graphische Benutzeroberfläche der LearnLib. Es erweitert das jABC, welches im Folgenden vorgestellt wird. Das LearnLib-Plugin nutzt das jABC als Entwicklungs- und Experimentierplattform.

Das jABC als Entwicklungsplattform ermöglicht es, die Bausteinen der LearnLib, mit anderen Diensten zu kombinieren und so Anwendungen, die auf Automatenlernen basieren, zu Entwickeln. Ein Vertreter dieser Art von Applikationen ist neben Smyle [21], welches in Abschnitt 5.3 vorgestellt wird, die Webtest-Umgebung, welche in Abschnitt 5.7 beschrieben wird.

Das jABC als Experimentierplattform dient dazu, mit den angebotenen Bausteinen der LearnLib und ihren Einstellungen zu experimentieren und ihre Eigenschaften im Sinne von Lernaufwand, Laufzeit und Speicherverbrauch statistisch zu analysieren. In den Abschnitten 4.2.4 und 4.2.3 werden Beispiele dieser Auswertungen vorgestellt.

Das jABC ist ein Framework zur Modellierung, Entwicklung und Ausführung von Applikationen und Diensten [128, 94]. Es unterstützt *Lightweight Process Coordination* [93] und wurde seit 1995 unter anderem zum Design von Telekommunikationsdiensten [127], Web-basierten Entscheidungsunterstützungssystemen (*Decision Support Systems*) [75] und als Testautomatisierungsumgebung für *Computer Telephony Integrated Systems* [68] eingesetzt.

Das jABC erlaubt es Benutzern, auf einfache Weise Applikationen und Dienste durch die Zusammenstellung wiederverwendbarer Bausteine zu entwickeln. Der Entwicklungsprozess wird durch eine beständig wachsende Menge von Plugins unterstützt, die den gesamten Entwicklungszyklus unterstützen. Der jABC-Ansatz ist gedacht als Erweiterung [73] und nicht als Ersatz anderer Modellierungsansätze wie beispielsweise dem UML-Basierten *Rational Unified Process* [79].

Die Integration der LearnLib in das jABC bietet eine Reihe von Vorteilen, die sich direkt aus den Features dieser Entwicklungsumgebung ergeben:

Einfachheit Das jABC ist für Anwendungsexperten gedacht, die typischerweise keine Programmierer sind.

Agilität Der *Lightweight Process Coordination* Ansatz unterstellt, dass Anforderungen, Modelle und andere Artefakte mit der Zeit häufig angepasst werden müssen.

Anpassbarkeit Die Bausteine, aus denen im jABC Anwendungen zusammengesetzt werden, können umbenannt, umgeordnet, und mit eigenen Symbolen versehen werden, um den Gewohnheiten der Anwendungsexperten zu entsprechen.

Konsistenz Dem gesamten Entwicklungsprozess unterliegt dasselbe Modellierungsparadigma, so dass die semantische Konsistenz vom ersten Prototyp bis zum fertigen Produkt gewahrt bleibt.

Verifikation Der Anwendungsexperte kann lokale und globale Eigenschaften angeben, die ein Modell erfüllen muss. Diese werden unter anderem mit einem *Model Checker* automatisch geprüft, so dass Modellierungsfehler direkt grafisch angezeigt werden können.

Service-Orientierung Existierende externe Bibliotheken, Applikationen und Dienste können auf einfache Weise in Modellierungsbausteine des jABC gekapselt werden.

Ausführbarkeit Die Modelle des jABC sind auf unterschiedliche Weise ausführbar. Sie können direkt in der Entwicklungsumgebung (auch Schrittweise) ausgeführt werden, oder zu eigenständigen Applikationen übersetzt werden.

Universalität Auf Grund von Java [11] als plattformunabhängiger, objektorientierter Implementierungssprache, kann das jABC leicht an eine Vielzahl von technischen Rahmenbedingungen und Anwendungsgebieten angepasst werden.

Zusätzlich zu diesen „gratis" Features des jABC, bietet das LearnLib-Plugin zwei unterschiedliche Arten der Modellierung an: Im Vorkonfigurations-Modus, definiert der Benutzer ein Experimentierszenario, dass zur statistischen Auswertung des Lernprozesses verwendet wird. Im Prozess-Modellierungs-Modus hat der Benutzer die vollständige Kontrolle über den Lernprozess und kann kontextspezifisch Optimierungen und Suchstrategien auswählen sowie eigene Lernanwendungen orchestrieren.

Die beiden Modellierungsmodi der LearnLib werden in der Veröffentlichung „LearnLib: A Library for Automata Learning and Experimentation" [113] detailliert vorgestellt.

4.1.1.1 Der Vorkonfigurations-Modus

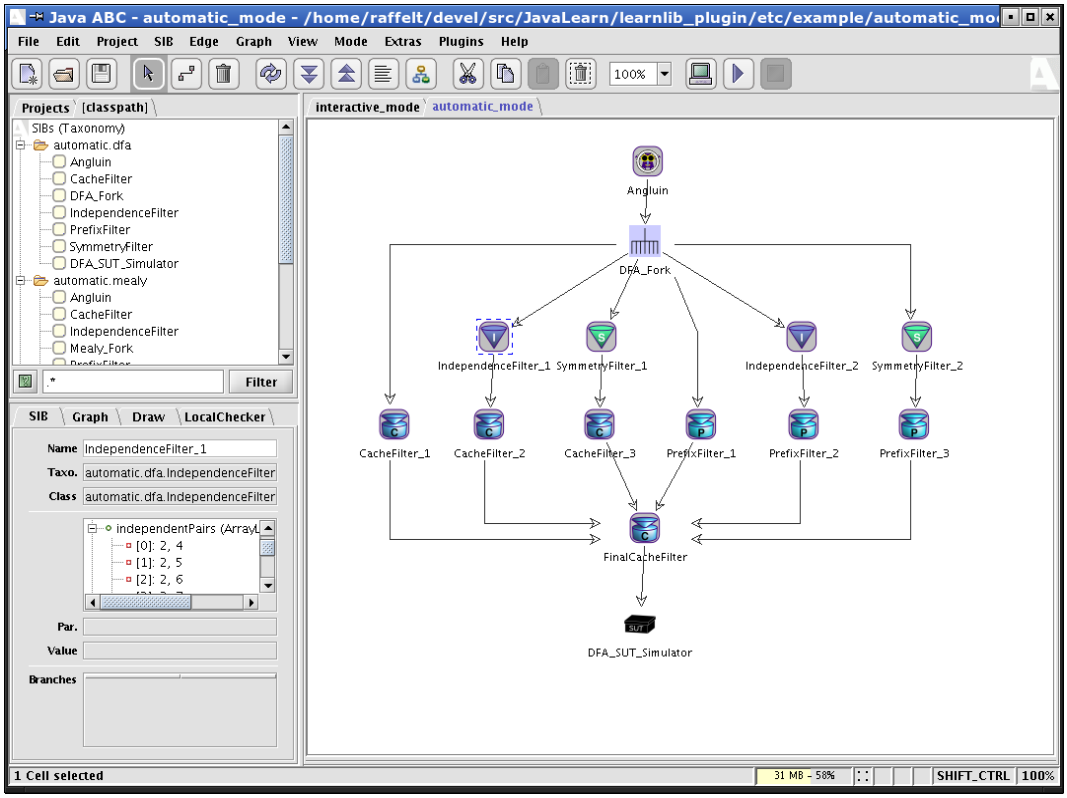

Abbildung 4.1: Vorkonfigurations-Modus: Entwurf einer Filter-Evaluation

Im Vorkonfigurations-Modus modelliert der Benutzer ein experimentelles Szenario. Mit Hilfe eines Datenflussgraphen legt der Benutzer fest, wie die *Membership Queries* und *Equivalence Queries* die Bausteine der LearnLib durchlaufen. Eine solche Konfiguration ist in Abbildung 4.1 dargestellt. Im Vorkonfigurations-Modus erstellt der Benutzer einen gerichteten azyklischen Graphen bestehend aus

- einem Lernalgorithmus,
- einer Menge von Filtern und
- einer Schnittstelle zu einem Back-Box-System.

Der Graph wird im jABC-Stiel editiert: Der Benutzer wählt die LearnLib-Bausteine in einer Taxonomie (Abb. oben links) aus, zieht sie auf die Zeichenfläche und verbindet die Bausteine mit Kanten, welche festlegen wie die Bausteine miteinander interagieren. Jeder Baustein kann darüberhinaus über eine Menge von Parametern eingestellt und angepasst werden.

4.1.1.2 Prozess-Modellierungs-Modus

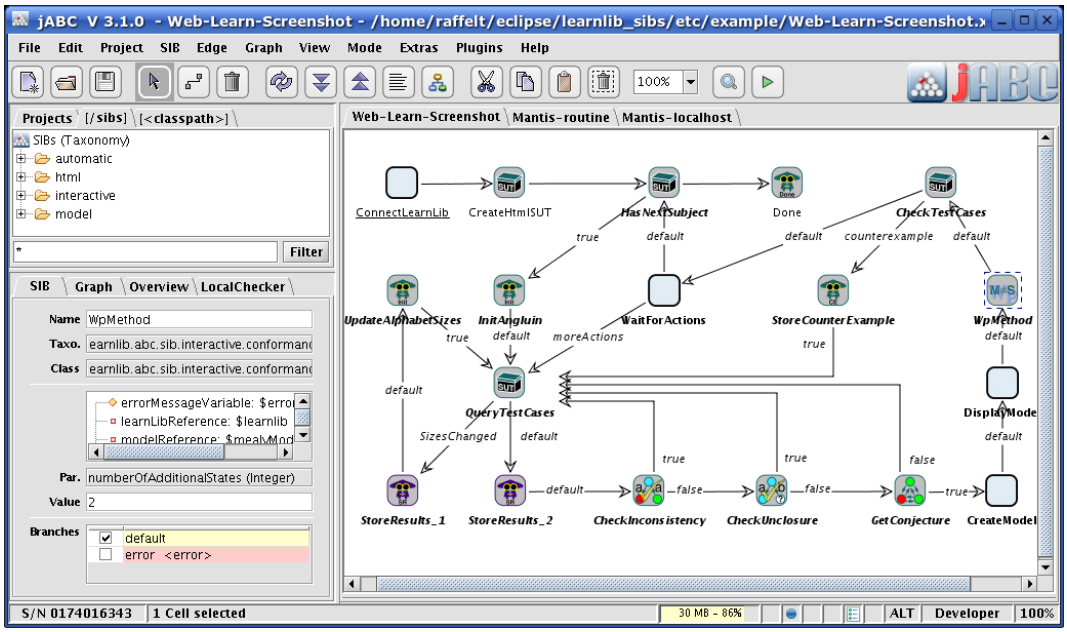

Abbildung 4.2: Prozess-Modellierungs-Modus: Entwurf einer Lernapplikation

Die Modelle im Prozess-Modellierungs-Modus werden auf dieselbe Weise bearbeitet wie im Vorkonfigurations-Modus. Der Graph entspricht jetzt aber einem Kontrollflussgraphen und die Knoten des Graphen repräsentieren atomare Funktionen des LearnLib-Kerns. Die Kanten steuern den Kontrollfluss und geben an, wie und unter welchen Bedingungen der Graph ausgeführt werden soll. In Abbildung 4.2 ist der Kontrollfussgraph einer einfachen Variante von Angluins Algorithmus dargestellt [112].

Die Ausführung des Graphen kann interaktiv mit den *Tracer* des jABC gesteuert werden. Dieser ist in der Lage, den Graphen schrittweise auszuführen und ermöglicht es, die zwischen den Bausteinen ausgetauschten Daten zur Fehlerbeseitigung einzusehen. Des Weiteren ist es auch möglich, die spezifizierte Lernanwendung in eine eigenständige Java-Applikation zu übersetzten.

4.1.2 Der LearnLib-Kern

Der LearnLib-Kern ist zum einen eine Bibliothek von Lernalgorithmen und zum anderen ein Server, welcher die Komponenten der LearnLib-Bibliothek als Dienst anbietet. Der LearnLib-Kern ist in C++ implementiert und wird als 64-Bit- und 32-Bit-Version unter Linux und Solaris entwickelt. Die aktuelle Codebasis des Kerns umfasst mehr als 600 Klassen und insgesamt mehr als 250.000 Zeilen Code.

Der LearnLib-Server basiert auf CORBA [1] und stellt keinen SOAP basierten Web-Service [78] dar. Der Grund ist, dass CORBA gegenüber Web-Services in technischer und konzeptioneller Hinsicht, im Sinne von Performance, Skalierbarkeit, Sicherheit, Interoperabilität und Portabilität überlegen ist [13, 54, 61]. Die einzigen relevanten Nachteile von CORBA sind die Firewallproblematik und das bestehende Vorurteil CORBA sei kompliziert.

Der LearnLib-Kern unterteilt sich in drei Bereiche: Lernalgorithmen, optimierende Filter und approximative Äquivalenz. Im folgenden Abschnitt 4.2 werden die, in der LearnLib realisierten Algorithmen vorgestellt. Der Abschnitt 4.3 beschreibt anwendungsspezifische Optimierungsfilter der LearnLib, mit deren Hilfe die Menge der Anfragen reduziert werden kann. Die Behandlung von Äquivalenzanfragen wird in Abschnitt 4.4 kurz skizziert.

4.2 Lernalgorithmen

Die LearnLib basiert im wesentlichen auf Angluins Algorithmus [10]. Neben mehreren Variationen dieses Algorithmus, sind auch verwandte Algorithmen zum direkten Lernen von Mealy-Automaten realisiert. Die Mealy-Lernalgorithmen basieren auf Nieses Algorithmus für „*input/output deterministic finite automata (IODFA)*" [102]. Der Unterschied zwischen Mealy-Automaten und IODFAs besteht darin, dass Mealy-Automaten (siehe Definition A.2) bei Verarbeitung eines Eingabesymbols genau ein Ausgabesymbol erzeugen, IODFAs können jedoch nach jedem Eingabesymbol eine endliche Sequenz von Ausgabesymbolen erzeugen. Die LearnLib realisiert in gewissem Sinne also ein einfacheres, direkteres und allgemeingültigeres Verfahren. Darüber hinaus sind noch zwei weitere Algorithmen zum Lernen von parametrisierten Systemen entwickelt worden. Sie werden in Abschnitt 4.2.5 skizziert,.

Die LearnLib implementiert Angluins Algorithmus für deterministische endliche Automaten und leicht abgewandelt für Mealy-Automaten. Der konventionelle Ansatz [10] ist aber neben dem Einsatz von effizienten Datenstrukturen um vier Opti-

mierungen erweitert worden. Zum einen wurden die Ideen von Rivest und Schapire aufgegriffen, ihre Umsetzung ist im folgenden Abschnitt 4.2.1 dargestellt. Zum anderen ermöglicht die LearnLib eine besondere Behandlung von langen Gegenbeispielen, vorgestellt in Abschnitt 4.2.2. Des Weiteren lassen sich mit Hilfe von Heuristiken die Algorithmen an die Struktur des Black-Box-Systems anpassen. Die resultierenden Strategien werden in Abschnitt 4.2.4 vorgestellt.

Die vierte marginale Verbesserung von Angluins Algorithmus ist Folgende (vergleiche auch Abschnitt 3.2.1): Angluins Algorithmus speichert Gegenbeispiele vom Äquivalenzorakel vollständig mit allen Präfixen im *Access Set S*. Dies führt zu einer Inkonsistenz in der *Observation Table* und der Algorithmus hat einen Anhaltspunkt sein Hypothesemodell zu überarbeiten. Wenn das Gegenbeispiel mit einem Suffix aus dem *Distinguishing Set E* endet, was trivialerweise immer der Fall ist, kann man das Gegenbeispiel um dieses Suffix gekürzt zum *Access Set S* hinzufügen. Die dem Gegenbeispiel zugrundeliegende, abweichende Beobachtung führt dann an einer andern Stelle der *Observation Table* zu einer vergleichbaren Inkonsistenz, die Menge S ist aber kleiner, so dass *Membership Queries* eingespart werden können.

4.2.1 Präfixaustausch

Der Algorithmus von Rivest und Schapire [120] optimiert Angluins Algorithmus in zweierlei Hinsicht. Zum einen wird dafür Sorge getragen, dass das *Access Set S* für jede Äquivalenzklasse von $\cong$ nur genau eine Sequenz enthält. Zum anderen wird darauf verzichtet, dass das *Distinguishing Set E* suffixabgeschlossen ist. Jedes Mal, wenn bei diesem Algorithmus ein Hypothesemodell mit Hilfe einer *Equivalence Query* auf Äquivalenz mit dem unbekannten System überprüft wird und das Orakel ein Gegenbeispiel w liefert, wird das Gegenbeispiel $w = uv$ wie folgt in ein Präfix u und ein Suffix v aufgeteilt. Die Idee besteht darin, einen möglichst langes Präfix u des Gegenbeispiels w durch ein möglichst kurzes bezüglich $\cong$ äquivalentes Präfix $u' \in S$ zu ersetzen, so dass die Sequenz $u'v$ weiterhin ein Gegenbeispiel ist. Beim Algorithmus von Rivest und Schapire wird nun ausschließlich das Suffix v zum *Distinguishing Set E* hinzugefügt, was dazuführt, dass die *Observation Table* nicht mehr vollständig ist.

In der Implementierung der LearnLib lässt sich diese, „Präfixaustausch" getaufte Optimierung, beim Lernen von deterministischen endlichen Automaten und Mealy-Automaten optional aktivieren. Im Gegensatz zum Algorithmus von Rivest und Schapire wird aber nicht das Suffix v zum *Distinguishing Set E* hinzugefügt, es wird wie gehabt das gesamte Gegenbeispiel $u'v$ zum *Access Set S* hinzugefügt. Der Grund ist, dass das Suffix v unter Umständen sehr lang ist, was sich negativ beim Lernen realer Systeme auswirken kann, da häufig nicht nur die Anzahl der Anfragen sondern auch ihre Größe relevant ist.

4.2.2 Suffixreduktion

Die Länge der Anfragen ist ein generelles Problem. Angluins Algorithmus behandelt lange Gegenbeispiele nicht effizient. Die Größe der *Observation Table* und so-

mit die Anzahl der *Membership Queries* verhält sich quadratisch zur Länge der Gegenbeispiele [10]. Dieses Problem ist Ziel der „Suffixreduktion" genannten Optimierung. Lange Gegenbeispiele lassen sich nicht verhindern. Sie entstehen beispielsweise, wenn man zur Überprüfung der Äquivalenz das Hypothesemodell und das System parallel ausführt und mit zufälligen Eingaben beschickt, bis eine Abweichung auftritt oder entschieden wird, dass die Äquivalenz hinreichend getestet wurde. Es ist sehr wahrscheinlich, dass Gegenbeispiele dieser Art einzelne Zustände des unbekannten Systems mehrfach durchlaufen. Dies gilt zwingend, wenn ihre Länge die Anzahl der Zustände des Systems übersteigt. Auch Gegenbeispiele, die auf *Unique Input/Output Sequences* basierenden *Conformance Test* Methoden, wie Beispielsweise der UIO-Methode von Shen, Lombardi und Dahbura [126] oder der UIOv-Methode von Vuong, Chan und Ito [133], gewonnen werden, haben diese Eigenschaft. Grund ist, dass diese Verfahren die Äquivalenz von Spezifikation und Implementierung mit nur einer sehr langen Testsequenz zeigen.

Ziel der Suffixreduktion ist es, nur das interessante Suffix eines Gegenbeispiels zu bestimmen und eventuelle Schleifen, die das Gegenbeispiel auf dem System durchläuft, zu entfernen. Hierzu trennt die Suffixreduktion ein Gegenbeispiel $w = uv$ in ein Präfix u und ein Suffix v auf. Dasr Suffix v wird so gewählt, dass es zunächst dem kürzesten Suffix, das nicht bereits im *Distinguishing Set E* enthalten ist, entspricht. Jedes kürzere Suffix ist nicht in der Lage, Zustände des Hypothesemodells aufzuspalten, da es bereits zum Trennen von Zuständen verwendet wird. Es wird angenommen, dass die Schleife des Gegenbeispiels direkt vor dem Suffix v endet. Nun wird das kürzeste Präfix u' von u gesucht, so dass $w' = u'v$ weiterhin ein Gegenbeispiel ist. Hierzu sind weitere *Membership Queries* nötig. Existiert ein solches u', so wird das, nun schleifenfreie, Gegenbeispiel w' wie gehabt mit all seinen Präfixen zum *Access Set S* hinzugefügt. Existiert ein solches u' nicht, so gibt es keine Schleife, die direkt vor dem Suffix v endet. Nun wird das Suffix v um ein Symbol verlängert und die Suche nach einem geeigneten Präfix u' beginnt von vorne. Wird auf diese Weise keine Schleife bzw. kein entsprechendes Präfix $'u$ für alle Suffixe v von w gefunden, dann ist das Gegenbeispiel w schleifenfrei und muss ungekürzt mit allen Präfixen zum *Access Set S* hinzugefügt werden.

4.2.3 Auswertung von Präfixaustausch und Suffixreduktion

In Abbildung 4.3 ist die Wirkung des Präfixaustausches und der Suffixreduktion dargestellt. Die Effektivität des Präfixaustausches und der Suffixreduktion liegt in allen vier dargestellten Fällen sehr nahe beieinander. Die Kombination von Präfixaustausch und Suffixreduktion bringt nur eine sehr geringere Verbesserung gegenüber einer einzelnen Optimierung. Bei ausgewogenen, zufälligen DFAs mit 50 Prozent akzeptierenden Zuständen reduzieren diese Optimierungen die Anzahl der Anfragen um ca. 14 Prozent, bei unausgewogenen, realistischeren DFAs mit nur 1 Prozent akzeptierender Zuständen werden über 22 Prozent erreicht. Betrachtet man das Volumen der Anfragen, also die Gesamtanzahl aller Symbole in den Anfragen, erreichen diese Optimierungen noch bessere Werte. Bei ausgewogenen DFAs mit 50 Prozent akzeptierenden Zuständen konnte eine Reduktion des Anfragevolumens

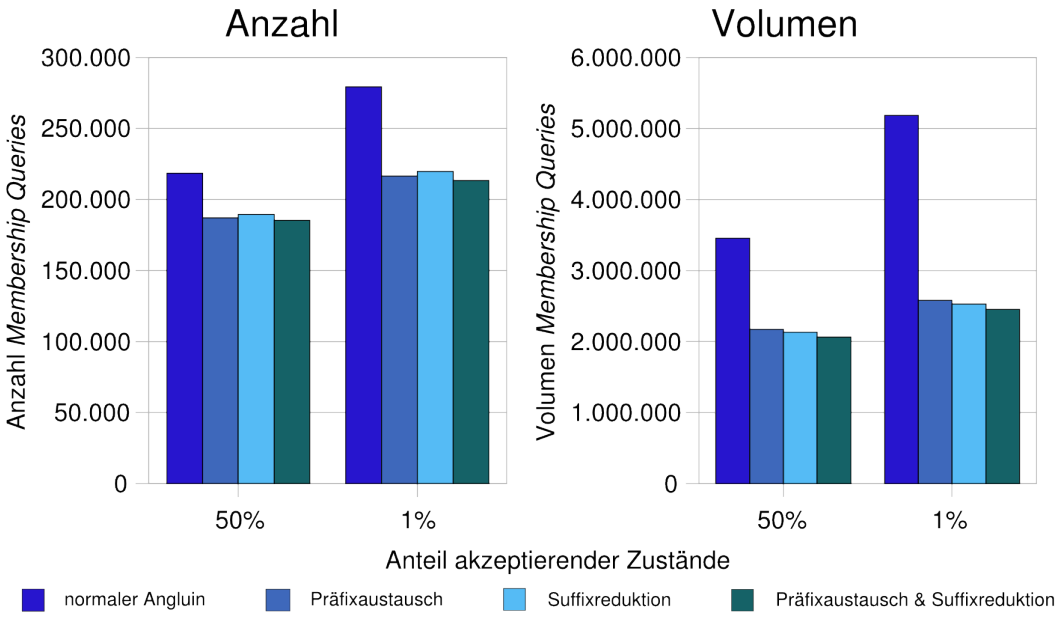

Abbildung 4.3: Anfragereduktion durch Präfixaustausch und Suffixreduktion

um 38 Prozent erzielt werden, bei den unausgewogenen DFAs liegt die Reduktionsrate sogar bei über 51 Prozent. In diesem experimentellen Szenario wurden für jeden der acht Fälle 1000 DFAs mit je 1000 Zuständen und 10 Aktionen gelernt. Die einzelnen Balken der Abbildung stellen jeweils den Durchschnitt von je 1000 Einzelwerten dar. Von den im nachfolgenden Abschnitt 4.3 vorgestellten Optimierungsfiltern wurde nur der Redundanz-Cache (Abschnitt 4.3.1) eingesetzt. Die Äquivalenz von den gelernten Hypothesemodellen und den simulierten „unbekannten" Systemen wurden approximativ mit Hilfe von zufälligen *Membership Queries* mit jeweils 100 Aktionen überprüft. Die in der Abbildung dargestellten Messwerte stellen die insgesamt vom System beantworteten Anfragen einschließlich dieser zufälligen Anfragen dar. Der Anteil dieser zufälligen Anfragen ist sehr gering, da im Durchschnitt weniger als 200 zufällige Anfragen benötigt wurden.

4.2.4 Strategien für Vollständigkeit und Konsistenz

Zentraler Punkt von Angluins Algorithmus ist das Sicherstellen von Vollständigkeit und Konsistenz der *Observation Table*. Beim Überprüfen der Tabelle werden typischerweise immer mehrere Stellen gefunden, an denen die Vollständigkeit und Konsistenz verletzt ist. Die Wahl der nächsten aufzulösenden Stelle, hat Auswirkungen auf alle weiteren Anfragen des Algorithmus. Die LearnLib bietet daher eine Reihe von Strategien an, die festlegen, wie der nächste Schritt zur Herstellung von Vollständigkeit und Konsistenz ausgewählt wird. Hierbei werden zum Teil auch die im folgenden Abschnitt 4.3 dargestellten optimierenden Filter befragt.

Die Strategie kann unabhängig für Vollständigkeit und Konsistenz ausgewählt werden. Die derzeit verfügbaren Strategien sind:

- **random:** Wähle zufällig aus.
- **fast:** Nimm die erste gefundene Möglichkeit und suche nicht nach weiteren Alternativen.
- **long:** Bevorzuge Alternativen, deren Auflösung zu langen *Membership Queries* führt.
- **short:** Bevorzuge Alternativen, deren Auflösung zu kurzen *Membership Queries* führt.
- **cheap:** Bevorzuge Alternativen, für deren Auflösung *Membership Queries* erzeugt werden, die von den nachgeschalteten Filtern beantwortet werden können.
- **expensive:** Bevorzuge Alternativen, für deren Auflösung *Membership Queries* erzeugt werden, die von den nachgeschalteten Filtern nicht beantwortet werden können.

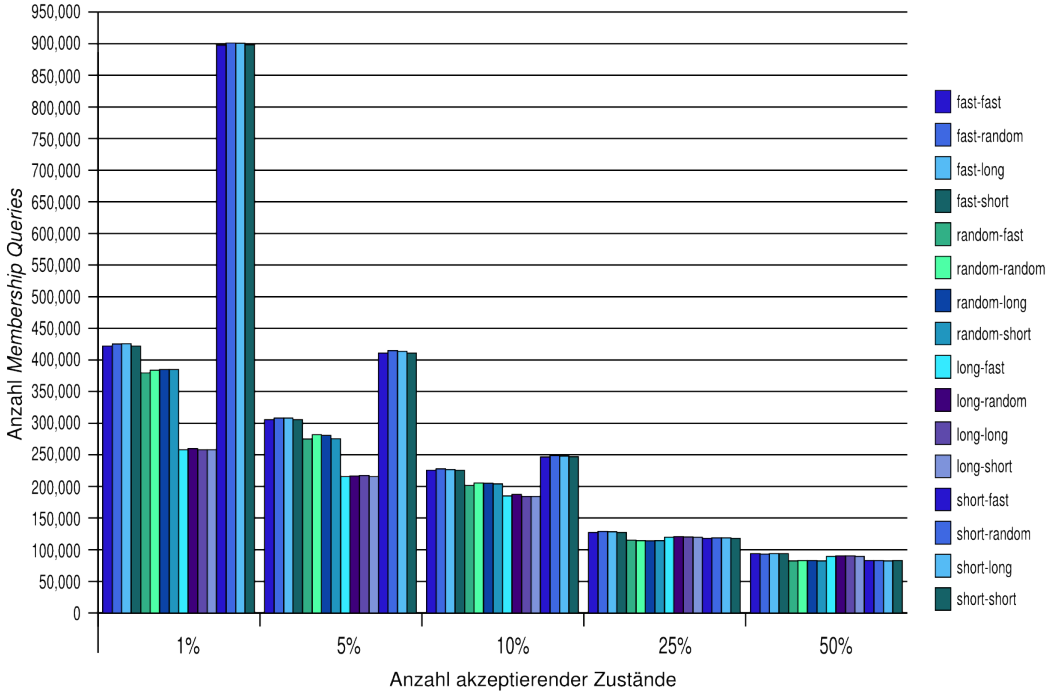

Abbildung 4.4: Strategie bei zufällig erzeugten DFAs

Das Diagramm in Abbildung 4.4 zeigt den Einfluss einiger Strategien beim Lernen zufällig erzeugter DFAs. Jeder Balken in der Abbildung stellt die durchschnittliche Anzahl von *Membership Queries* bei 1000 Stichproben dar. Die Modelle dieser Simulation haben je 100 Zustände und 64 Aktionen. Der Anteil akzeptierender Zustände variiert schrittweise zwischen 1 Prozent und 50 Prozent. Da akzeptierende und nicht akzeptierende Zustände symmetrisch behandelt werden, ist es nicht notwendig den Bereich von 50 Prozent bis 99 Prozent akzeptierender Zustände

zu untersuchen. Die Ergebnisse zeigen, dass zumindest die Strategien im Bereich der Konsistenz einen großen Einfluss auf die Anzahl der notwendigen *Membership Queries* haben. Es zeigt sich aber auch, dass es keine optimale Strategie für alle Situationen gibt. Die Auswahl der richtigen Strategie ist abhängig von der Struktur des Modells. Beispielsweise führt die Bevorzugung kurzer *Membership Queries* bei DFAs mit einem ausgewogenen Anteil von akzeptierenden Zuständen zu recht guten Ergebnissen, bei unausgewogenen Modellen sind aber lange *Membership Queries* zu präferieren.

4.2.5 Weitere Lernalgorithmen

Die LearnLib implementiert Angluins Algorithmus für deterministische endliche Automaten und leicht abgewandelt für Mealy-Automaten. Daneben ist in der LearnLib ein weiterer Algorithmus integriert. Basis sind hier Automaten, deren Aktionen mit Booleschen Parametern versehen sind. Dieser Algorithmus ist in einem gemeinsamen Forschungsprojekt mit Therese Bohlin (geb. Berg) aus der *Testing of Reactive Systems Group*, der *Uppsala University* entstanden. Dieser Algorithmus basiert auch auf Angluins Algorithmus und übertrifft diesen hinsichtlich der Anzahl von *Membership Queries* dann, wenn die Anzahl der „relevanten" Parameter der Aktionen gering ist. Der Algorithmus wird in der Veröffentlichung „Regular Inference for State Machines with Parameters" [18] vorgestellt.

Aus der Kooperation mit der *Uppsala University* ist noch ein weiterer Algorithmus entstanden. Dieser ist Thema der Veröffentlichung „Regular Inference for State Machines using Domains with Equality Tests" [19]. Der Algorithmus basiert auch auf Angluins Algorithmus. Er ermöglicht das Lernen von Mealy-Automaten, die mit Zustandsvariablen über einem (potentiell) nicht endlichen Wertebereichen erweitert sind. Dieser Algorithmus, der vermutlich erste für unendliche Systeme, ist leider noch nicht implementiert.

Die Idee, Automaten mit parametrisierten Aktionen direkt zu lernen, wird derzeit von Howar im Rahmen einer Diplomarbeit [70] ausgebaut. Bei diesem erweiterten Ansatz bestehen die Wertebereiche der Parameter aus Intervallen über ganzen Zahlen. Die Parameter der Aktionen spannen einen mehrdimensionalen Raum auf, welcher sich in eine Menge von Regionen bzw. Quader unterteilen lässt. Die Transitionen eines Zustandes lassen sich zu Äquivalenzklassen von Transitionen zusammenfassen, die zum selben Nachgfolgezustand führen. Die Parameter äquivalenter Transitionen liegen bei den hier betrachteten Automaten alle in derselben Region des ausgespannten Raumes. Durch die Kombination eines Automatenlernverfahrens mit einem Lernalgorithmus für geometrische Konzepte [30, 31], wird gezielt versucht, die „Ränder" dieser Regionen zu lernen. Das Alphabet des Lerners muss nicht mehr Symbole für alle möglichen Kombinationen der Parameter enthalten.

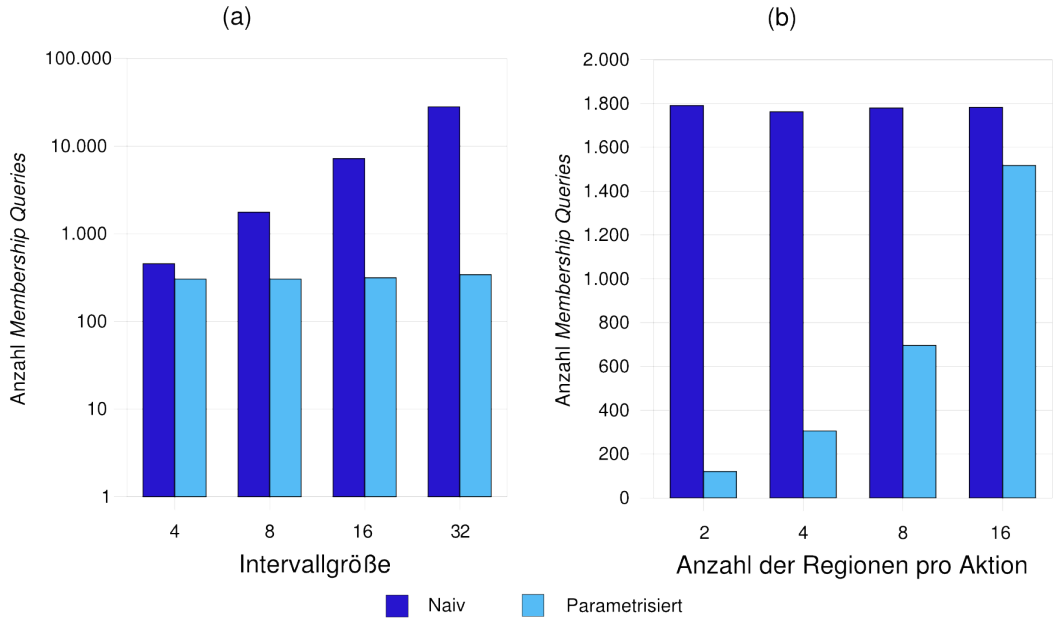

Abbildung 4.5: Anfragereduktion durch Verwendung parametrisierter Modelle

Die ersten Ergebnisse, dargestellt in Abbildung 4.5, entsprechen unseren Erwartungen. Mit zunehmender Intervallgröße[1] erhöht sich der Vorteil diese Ansatzes (a). Mit zunehmender Anzahl von Regionen[2] pro Aktion relativiert sich der Vorteil (b).

4.3 Anwendungsspezifische Optimierungen

Klassische aktive Automatenlernverfahren sind nicht geeignet, Modelle von realen Systemen zu extrapolieren. Dies liegt an der enormen Menge von *Membership Queries*, welche durch Ausführung von Testfällen beantwortet werden müssen. Um diesem Problem zu entgegnen, bietet die LearnLib eine Reihe von Filtertechniken an, die anwendungsspezifisches Expertenwissen ausnutzen, um die Menge der Anfragen drastisch zu verringern. Diese Filtertechnologie basiert auf der Arbeit von Hungar et al. [72]. Sie wird in der Veröffentlichung „Efficient Test-based Model Generation of Legacy Systems" [129] thematisiert und anhand von Beispielen aus dem Telekommunikationsbereich evaluiert. Die Wechselwirkungen der Filter untereinander sind Thema der Veröffentlichung „Analyzing Second-Order Effects Between Optimizations for System-Level Test-Based Model Generation" [90]. Die, in den Veröffentlichungen dargestellten experimentellen Resultate, zeigen in eindrucksvoller Weise die Effektivität der Filter. Es gibt eine vergleichbare Arbeit von Rivest

[1] Abbildung 4.5 (a) stellt die Anzahl der *Membership Queries* beim Lernen eines Moore-Automaten mit sechs Zuständen und einer Aktion mit zwei Parametern dar, wobei sich die Transitionen eines Zustands jeweils in vier äquivalente Regionen aufteilen lassen. Die Intervallgröße variiert zwischen 4 und 32.

[2] Abbildung 4.5 (b) stellt die Anzahl der *Membership Queries* beim Lernen eines Moore-Automaten mit sechs Zuständen und einer Aktion mit zwei Parametern dar, wobei die Wertebereiche Intervallen aus 8 Werten entsprechen. Die Anzahl der Regionen pro Aktion variiert zwischen 2 und 16.

und Schapire [121], die noch deutlich bessere Resultate erzielt. Dieser Ansatz setzt die wenig praxisrelevante Eigenschaft voraus, dass die Zustandstransformationsfunktion (vergl. Def. A.1) für jedes Symbol eine Permutation der Zustände ist.

Die Filtertechnologie wird im Folgenden detailliert vorgestellt, da die Mealy-Version der Filter bisher in keiner Veröffentlichung genau beschrieben ist.

In Abbildung 4.7 ist das Modell eines Mealy-Automaten dargestellt. Anhand dieses abstrakten Beispiels wird im Folgenden die Charakteristik der Filter vorgestellt. Der Automat stellt eine stark entstellte Turingmaschine dar. Sie besteht aus einem Schreib-/Lese-Kopf, der sich nach rechts und links bewegen kann und einem Band aus Speicherzellen, wie in Abbildung 4.6 dargestellt.

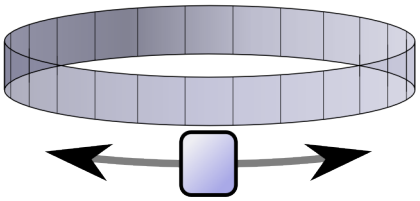

Abbildung 4.6: Einfache Turing-ähnliche Bandmaschine

Jede Zelle des Bandes kann ein Symbol aus $\{empty, 0, 1\}$ enthalten. Im Gegensatz zu Turingmaschinen [131] hat das Band eine begrenzte Länge und seine Enden sind zu einem Ring verbunden. Initial sind alle Zellen des Bandes leer (*empty*). Der Kopf kann nur die Symbole 0 und 1 schreiben, es ist nicht möglich eine Zelle wieder zu löschen. Die Lese-Operation gibt, wenn die Zelle nicht leer ist, das Symbol der aktuellen Zelle zurück. Andernfalls wir das Symbol *error* zurückgegeben: Der Automat versagt vollständig und liefert von diesem Zeitpunkt an nur noch *error* Symbole. Das Modell in Abbildung 4.7 beschreibt einen Automaten mit Bandlänge 2. Auf Grund der eigenwilligen Konstruktion lassen sich die im Folgenden vorgestellten Filter sehr gut an diesem Beispiel verdeutlichen.

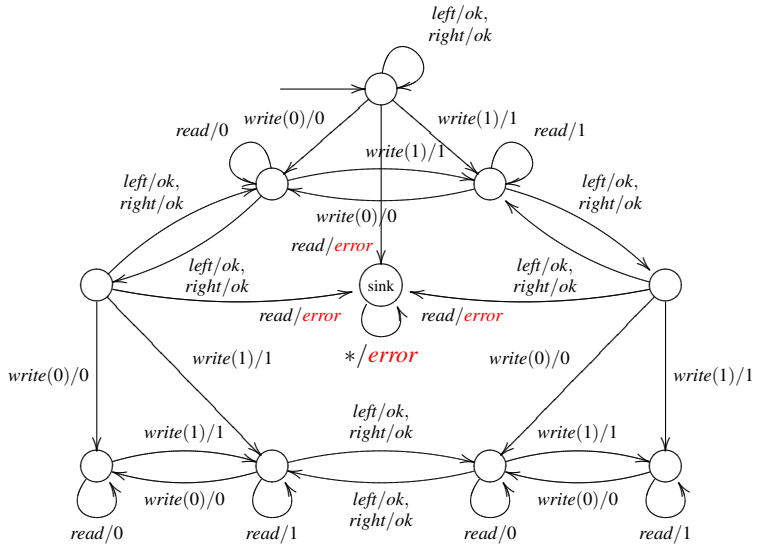

Abbildung 4.7: Präfixabgeschlossener Mealy-Automat mit unabhängigen Aktionen und Symmetrie

4.3.1 Redundanz

Der klassische Algorithmus von Angluin generiert redundante *Membership Queries*. Das liegt daran, dass in seiner zentralen Datenstruktur, der *Observation Table*, die Ergebnisse ein und derselben Anfrage in mehreren Tabellenzellen gespeichert werden, und daher auch mehrfach abgefragt werden. Die Ergebnisse von Anfragen der Form uvw mit $u,v,w \in \Sigma^*$ können in der *Observation Table* in Zeile uv und Spalte w und gleichzeitig in Zeile u und Spalte vw benötigt werden.

Ein einfacher Cache kann verhindern, dass *Membership Queries* mehrfach vom zu lernenden System beantwortet werden. Der Cache $T : \Sigma^* \rightarrow \Gamma \cup \{\text{unknown}\}$ ist als Baum organisiert und speichert die Ergebnisse aller Anfragen. Nur wenn, eine Antwort nicht bereits im Cache befindet, muss die Anfrage wirklich ausgeführt werden. Für gegebene *Membership Queries* $MQ(q)$ folgen direkt die entsprechenden Filterregeln:

$$\begin{aligned} T(q) = \text{true} &\quad\Rightarrow\quad MQ(q) = \text{true} \\ T(q) = \text{false} &\quad\Rightarrow\quad MQ(q) = \text{false} \end{aligned}$$

4.3.2 Präfixabgeschlossenheit

Häufig ist es ausreichend nur eine Beschreibung möglicher Ausführungsreihenfolgen eines Systems darzustellen. Im Vordergrund steht die Frage, welche Aktionen in welcher Reihenfolge durchgeführt werden können. Diese Betrachtungsweise führt zu einer sehr mächtigen Optimierung, da der Lernalgorithmus Fortsetzungen von Sequenzen nicht beachten muss, von denen bereits bekannt ist, dass der Automat sie nicht akzeptiert. Umgekehrt müssen Präfixe von Sequenzen, die bereits als akzeptiert gekennzeichnet sind, auch vom zu lernenden Automaten akzeptiert werden.

Definition 4.1 *Ein deterministischer endlicher Automat* $\mathsf{M} = (S, s_0, \Sigma, \delta, F)$ *heißt genau dann* präfixabgeschlossen, *wenn die Menge der nicht akzeptierenden Zustände* $S \backslash F$ *bezüglich der Transitionsfunktion* δ *abgeschlossen ist.*

$$\forall a \in \Sigma.\ \forall s \in (S \backslash F)\,.\,\delta(s,a) \in (S \backslash F)$$

Im Falle von Mealy-Automaten kann man Präfixabgeschlossenheit einführen, indem man ein spezielles Fehlerausgabesymbol definiert, welches anzeigt, dass der Automat in einem speziellen Fehlerzustand fest sitzt, und diesen nicht mehr verlassen wird. Im Beispiel aus Abbildung 4.7 ist dieser Zustand mit „*sink*" bezeichnet, das Fehlerausgabesymbol ist *error*.

Definition 4.2 *Ein Mealy-Automat heißt genau dann* präfixabgeschlossen bezüglich f, *wenn es ein Fehlerausgabesymbol* $f \in \Gamma$ *gibt, so dass der Automat, nachdem er das Fehlerausgabesymbol* f *ausgegeben hat, unabhängig von der Eingabe nur Fehlersymbole produzieren kann.*

$$\forall a \in \Sigma, \forall s, s' \in S.\ s \xrightarrow{a/f} s' \Longrightarrow \forall b \in \Sigma.\ \gamma(s', b) = f$$

Der Filter für Präfixabgeschlossenheit ist als „optimierter" Cache implementiert. Die entsprechenden Filterregeln für *Membership Queries* $MQ(q)$ sind:

$$\begin{array}{lcl} \exists v \in \Sigma^*.\ T(qv) = \text{true} & \Rightarrow & MQ(q) = \text{true} \\ \exists v, w \in \Sigma^*. q = vw \wedge\ T(v) = \text{false} & \Rightarrow & MQ(q) = \text{false} \end{array}$$

4.3.3 Unabhängige Aktionen

Beobachtbare Aktionen eines Systems können voneinander unabhängig sein. Sie können in beliebiger Reihenfolge ausgeführt werden und führen doch immer zum selben Systemzustand. Sobald eine Ausführungsreihenfolge bekannt ist, lassen sich alle anderen äquivalenten Ausführungsreihenfolgen und die Ergebnisse der entsprechenden *Membership Queries* ableiten. Im DFA-Fall ist dies besonders einfach, da alle, bezüglich dieser Eigenschaft äquivalenten Ausführungsreihenfolgen, entweder akzeptierend oder nicht akzeptierend sind. Im Mealy-Fall gibt es für zwei äquivalente Eingaben $w, v \in \Sigma^n$ eine Permutation $\pi : [1,n] \to [1,n]$ mit $\forall i \in [1,n]. v_i = w_{\pi(i)}$. Die Permutation beschreibt, wie die Symbole einer Eingabe w vertauscht werden müssen, um daraus die äquivalente Eingabe v zu erzeugen. Um aus der Antwort $\gamma(w)$ einer *Membership Query* $MQ(w)$ das Ergebnis $\gamma(v)$ für die Anfrage $MQ(v)$ zu berechnen, braucht man die Ausgabesequenz $\gamma(w)$ nur in derselben Weise vertauschen, wie die Eingabesequenz: $\gamma(v) = \Pi(\gamma(w))$ mit $\Pi(w) = w_{\pi(1)} w_{\pi(2)} \ldots w_{\pi(n)}$.

Im Gegensatz zum Präfixfilter (4.3.2) ist der Unabhängigkeitsfilter auf die Eingabe eines Anwendungsexperten angewiesen, der in Form einer Unabhängigkeitsrelation definiert, welche Aktionen in unabhängiger Reihenfolge ausgeführt werden können.

Der Mealy-Automat aus Abbildung 4.7 enthält genau ein Paar von unabhängigen Aktionen: (*left*, *right*). In jedem Zustand überführen die Transitionssequenzen *left right* und *right left* den Automaten in denselben Zielzustand[3]. Die Vertauschbarkeit gilt nur, wenn die Aktionen *left* und *right* direkt aufeinander folgen. Eine dazwischen eingefügte *write*- oder *read*-Aktion kann das System in unterschiedliche Zielzustände überführen oder trägt dazu bei, dass unterschiedliche Ausgaben produziert werden.

Formal ist die Unabhängigkeit von Aktionen eine homogene, irreflexive, symmetrische Relation auf Aktionen.

Definition 4.3 *Zwei Aktionen $a, b \in \Sigma$ heißen genau dann unabhängig bezüglich eines minimalen DFA $\mathsf{M} = (S, s_0, \Sigma, \delta, F)$, wenn in jedem Zustand des Automaten die Sequenzen ab und ba zu demselben Nachfolgezustand führen.*

$$\forall a, b \in \Sigma \wedge \forall s \in S.\ \delta(s, ab) = \delta(s, ba)$$

[3] Die beiden Aktionen *left* und *right* sind in dem Beispiel mit Bandlänge 2 sogar exakt identisch. Die Eigenschaft gilt jedoch für beliebige Bandlängen.

Definition 4.4 *Zwei Aktionen $a,b \in \Sigma$ heißen genau dann unabhängig bezüglich eines minimalen Mealy-Automaten, wenn in jedem Zustand des Automaten die Sequenzen ab und ba zu demselben Nachfolgezustand führen und die Ausgaben entsprechend vertauscht werden können.*

$$\forall a,b \in \Sigma \wedge \forall s \in S. \begin{cases} \delta(s,ab) = \delta(s,ba) \wedge \\ \gamma(s,a) = \gamma(\delta(s,b),a) \wedge \\ \gamma(s,b) = \gamma(\delta(s,a),b) \end{cases}$$

Die Menge aller unabhängigen Aktionen bildet die Unabhängigkeitsrelation $I : \Sigma \times \Sigma$ und induziert eine Äquivalenzrelation $\equiv_I \subseteq \Sigma^* \times \Sigma^*$ auf Anfragen. Hierbei sind zwei Anfragen q und q' genau dann äquivalent, wenn es eine Umordnung Π der Aktionen gibt, die Anfrage q in q' transformiert, und die der Unabhängigkeitsrelation I entspricht. Die Anzahl möglicher Umordnungen einer Sequenz bezüglich der Unabhängigkeitsrelation ist exponentiell in der Anzahl vertauschbarer Paare in I. Aus diesem Grund berechnet der Unabhängigkeitsfilter nicht alle möglichen Umordnungen sondern normalisiert die Anfragen bezüglich der Äquivalenzrelation $\equiv_I$. Es wird die lexikographisch kleinste äquivalente Anfrage q_{min} zu einer Anfrage q und die zugehörige Umordnung Π berechnet. Für jede Äquivalenzklasse wird nur die Anfrage q_{min} an einen nachgeschalteten Redundanz- oder Präfixfilter (4.3.1) weitergeleitet. Im Mealy-Fall kann mit Hilfe der Umordnung Π die gesuchte Antwort aus der Antwort für die normalisierte Anfrage zurückberechnet werden.

4.3.4 Symmetrie

Hardware und insbesondere Telekommunikationssysteme enthalten oft eine Vielzahl gleichartiger Komponenten. Von außen betrachtet lassen sich diese häufig nur durch eine Identifikationsnummer oder Ähnliches unterscheiden. Man denke an eine Telefonanlage mit mehreren Telefonen. Für eine Modellierung, die das Verhalten dieses Systems abbilden soll, spielt es keine Rolle, welches individuelle Endgerät eine Aktion initiiert, solange sich alle Telefone gleich verhalten. Die Art und Weise, wie eine Telefonverbindung von Telefon A zu Telefon B aufgebaut wird, ist vollständig symmetrisch in der Gegenrichtung. Diese Art von Symmetrie bietet ein großes Potential für Optimierungen, welches insbesondere mit der Zahl gleicher Systemkomponenten wächst. Wie auch der Unabhängigkeitsfilter, ist der Symmetriefilter auf die Eingabe eines Anwendungsexperten angewiesen, der die Symmetrie des zu lernenden Modells vorgibt.

Eingabe Permutation	Ausgabe Permutation
ID	*ID*
$(write(0),\ write(1))$	$(0,\ 1)$
$(left,\ right)$	*ID*
$(write(0),\ write(1))\,(left,\ right)$	$(0,\ 1)$

Tabelle 4.1: Zulässige Permutationen für den Mealy-Automaten aus Abbildung 4.7

Formal lässt sich Symmetrie als Permutationsgruppe über dem Alphabet beschreiben. Jede Permutation beschreibt einen zulässigen Austausch von Aktionen, bei dem sich das System symmetrisch verhält. Im DFA-Fall ist es ausreichend zu beschreiben, wie die (Eingabe-) Aktionen permutiert werden können. Im Mealy-Fall dagegen, zieht eine Permutation der Eingabeaktionen typischerweise auch eine Permutation der Ausgabeaktionen nach sich. Im Beispiel aus Abbildung 4.7 lassen sich die Aktionen $write(0)$ und $write(1)$ symmetrisch austauschen. Um ein äquivalentes Modell zu erhalten, müssen aber auch die Ausgabesymbole 0 und 1 in gleicher Weise ausgetauscht werden. In Tabelle 4.1 sind alle zulässigen Permutationen des Beispiels aufgeführt. In der linken Spalte sind die Permutationen der Eingabesymbole in Zyklenschreibweise aufgeführt und rechts daneben jeweils die zugehörige Permutation der Ausgabesymbole.

Definition 4.5 *Eine Permutation π über dem Alphabet Σ heißt genau dann zulässig für einen deterministischen endlichen Automaten $\mathsf{M} = (S, s_0, \Sigma, \delta, F)$, wenn π das Alphabet so abbildet, dass jedes Wort $w \in \Sigma^*$ genau dann vom DFA akzeptiert wird, wenn auch das zugehörige permutierte Wort $w' = \Pi(w)$ mit $\Pi(v) = \pi(v_1)\,\pi(v_2)\ldots\pi\left(v_{|v|}\right)$ vom DFA akzeptiert wird.*

$$\forall w \in \Sigma^*.\, \delta(w) \in F \Leftrightarrow \delta(\Pi(w)) \in F$$

Definition 4.6 *Ein Paar von Eingabe- und Ausgabe-Permutation $(\pi_\Sigma,\ \pi_\Gamma)$ über den Eingabe- und Ausgabe-Alphabeten Σ und Γ heißt genau dann zulässig für einen Mealy-Automaten $\mathsf{M} = (S, s_0, \Sigma, \Gamma, \delta, \gamma)$, wenn gilt:*

$$\forall w \in \Sigma^*.\, \Pi_\Gamma(\gamma(w)) = \gamma(\Pi_\Sigma(w))$$

Hierbei sind die Permutationen $\pi_\Sigma : \Sigma \to \Sigma$ und $\pi_\Gamma : \Gamma \to \Gamma$ über Symbolen auf gleiche Weise zu Permutationen $\Pi_\Sigma : \Sigma^* \to \Sigma^*$ und $\Pi_\Gamma : \Gamma^* \to \Gamma^*$ über Worte erweitert worden, wie im DFA-Fall: $\Pi_\Sigma(v) = \pi_\Sigma(v_1)\,\pi_\Sigma(v_2)\ldots\pi_\Sigma\left(v_{|v|}\right)$ und $\Pi_\Gamma(v) = \pi_\Gamma(v_1)\,\pi_\Gamma(v_2)\ldots\pi_\Gamma\left(v_{|v|}\right)$.

$$\begin{array}{ccc} \Sigma^* & \xrightarrow{\gamma} & \Gamma^* \\ {\scriptstyle \Pi_\Sigma}\downarrow & & \downarrow{\scriptstyle \Pi_\Gamma} \\ \Sigma^* & \xrightarrow{\gamma} & \Gamma^* \end{array}$$

Abbildung 4.8: Kommutatives Diagramm der Mealy-Symmetrie

Mit anderen Worten: Die erweiterte Eingabepermutation Π_Σ, die Ausgabepermutation Π_Γ und die Ausgabefunktion γ kommutieren wie in Abbildung 4.8 dargestellt.

Die Implementierung des Symmetriefilters normalisiert die Anfragen und leitet sie, wie auch der Unabhängigkeitsfilter, an einen nachgeschalteten Redundanz- oder

Präfixfilter weiter. Die Symmetrie-Normalisierung wählt aus der Permutationsgruppe die Permutation, welche eine Anfrage auf ihre lexikographisch kleinste äquivalente Anfrage abbildet. Im Gegensatz zum Unabhängigkeitsfilter, welcher auf lokale Weise nur die Reihenfolge der Aktionen einer Anfrage vertauscht, verhält sich der Symmetriefilter globaler: Er kann die Symbole einer Anfrage vollständig austauschen.

4.3.5 Wechselwirkungen der Filter

Wie bereits in Abschnitt 4.1.1 dargestellt, lassen sich die Filter in beliebiger Reihenfolge kombinieren. Die Filter beeinflussen sich gegenseitig. Die Wechselwirkungen der Filter untereinander werden im Rahmen der Veröffentlichung „Analyzing Second-Order Effects Between Optimizations for System-Level Test-Based Model Generation" [90] detailliert analysiert. Ein interessantes Ergebnis dieser Arbeit ist, dass die Reihenfolge von Symmetriefilter und Unabhängigkeitsfilter einen Einfluss auf die Menge der herausgefilterten Anfragen hat. Bei den untersuchten Beispielen zeigte es sich, dass manche Anfragen von einem Symmetriefilter mit nachgeschaltetem Unabhängigkeitsfilter beantwortet werden konnten, von einem Unabhängigkeitsfilter mit nachgeschaltetem Symmetriefilter aber nicht (und umgekehrt). Beiden Filtern liegt eine Äquivalenzrelation zu Grunde, so dass die Ursache für diese Auffälligkeit nicht ganz offensichtlich ist.

Der Grund für dieses Verhalten der Filter ist die Tatsache, dass beide Filter mit Normalisierung arbeiten. In einem deterministischen endlichen Automaten mit Alphabet $\Sigma = \{a, b, c\}$ seien die Aktionen a und b unabhängig und folgende Permutation zulässig $(a\,b\,c)$. Gegeben sei eine Anfrage $w = cba$.

- Der Unabhängigkeitsfilter mit nachgeschaltetem Symmetriefilter normalisiert die Anfrage *cba* zunächst zu *cab*, der Symmetriefilter normalisiert dann weiter zu *abc*.
- Der Symmetriefilter mit nachgeschaltetem Unabhängigkeitsfilter normalisiert die Anfrage *cba* zunächst zu *acb*, der Unabhängigkeitsfilter kann hier nichts weiter ausrichten, da die Symbole *a* und *b* nicht direkt hintereinander auftreten: es bleibt bei *acb*.

Je nach dem, ob die Anfragen *abc* oder *acb* bereits zu einem früheren Zeitpunkt gestellt wurden, weist sich die eine oder andere Kombination der Filter als vorteilhaft aus.

4.4 Äquivalenz Approximation

Wie bereits zuvor erwähnt, ist sind *Equivalence Queries* für endliche Automaten durch reines Testen generell unentscheidbar. Die Äquivalenz von Hypotesemodell und Implementierung lässt sich nur approximativ entscheiden. Die LearnLib bietet

neben der Erzeugung von zufälligen Testsequenzen noch zwei Methoden aus dem Bereich *Conformance Testing* an.

Die Problemstellung im Bereich Conformance Testing lässt sich wie folgt kurz beschreiben [82]. Bei einem gegebenen

- endlichen Automaten M_S, der Spezifikation und
- einer Black-Box-Implementierung M_I, die nur in der Lage ist Testfälle auszuführen,

soll durch Testen festgestellt werden, ob die Implementierung M_I die Spezifikation M_S korrekt (bzw. konform) implementiert. Dieses Problem ist im allgemeinen unentscheidbar. Es gibt aber ein ganzes Forschungsgebiet von Ansätzen, von denen manche unter bestimmten Voraussetzungen die Äquivalenz von Spezifikation und Implementierung zeigen können. Eine typische Voraussetzungen ist beispielsweise eine begrenzte Anzahl von Zuständen in der Implementierung. Die LearnLib implementiert zwei dieser Verfahren jeweils für DFAs und Mealy-Automaten: die W-Methode von Chow [37] und die Wp-Methode von Fujiwara et al. [57].

Beide Methoden setzen vorraus, dass die Implementierung höchstens k zusätzliche Zustände besitzt als die Spezifikation, welche im Rahmen von Automatenlernen als Hypothesemodell gegeben ist. Unter der Annahme, dass die Implementierung nicht mehr Zustände hat, als die Spezifikation, sind diese Verfahren nicht in der Lage, Hypothesen des Lernalgorithmus zu widerlegen. Der Implementierung müssen $k > 1$ zusätzliche Zustände zugestanden werden, damit es dann mit exponentiellem Aufwand $O\left(|\Sigma|^k\right)$ möglich ist, die Äquivalenz zu widerlegen. Dies liegt daran, dass die Methoden in den Gebieten Automatenlernen und *Conformance Testing* eng verwandt sind. Die Beziehungen zwischen diesen beiden Gebieten ist Thema der Veröffentlichung „On the Correspondence Between Conformance Testing and Regular Inference" [16].

Bei den experimentell untersuchten, praxisnahen Systemen, und insbesondere auch bei simulierten, zufälligen Systemen, hat es sich als effizient herausgestellt, die Äquivalenz zunächst mit zufälligen Testsequenzen zu überprüfen, da in der Anfangsphase die Differenzen zwischen Hypothese und Implementierung noch sehr groß sind und sich Abweichungen auf diese Weise schnell aufdecken lassen. Erst wenn die schnelle, randomisierte Methode nicht mehr wirkt, lassen sich die *Conformance Test* Methoden sinnvoll (d.h mit kleinem k) anwenden. Hierdurch wird abschließend eine gewisse Sicherheit bezüglich der Äquivalenz von Hypothese und Implementierung erreicht.

Kapitel 5

Anwendungen

Ziel der LearnLib ist es, die praktische Anwendung von Automatenlernverfahren zu ermöglichen. In diesem Kapitel werden einige Ansätze und Anwendungen vorgestellt, die auf Basis der LearnLib entwickelt wurden. Die Ansätze und Anwendungen lassen sich grob in zwei Arten von Ansätzen unterteilen: *Requirements Engineering* und *Modelbased Regression Testing*. Darüber hinaus wird in Abschnitt 5.4 eine Anwendung der LearnLib vorgestellt, die nicht in dieses Raster passt. Bei diesem Ansatz werden durch Lernverfahren iterativ Hypothesemodelle aus *Black-Box*-Systemen gewonnen, die mit Hilfe eines *Model Checkers* auf die Einhaltung globaler Eigenschaften hin geprüft werden.

Requirements Engineering ist ein wesentlicher Teil des Anforderungsmanagements im Software- und Systementwicklungsprozess. Ziel ist es, die Anforderungen an das zu entwickelnde System möglichst genau zu erfassen.

Die in Abschnitt 5.1, 5.2 und 5.3 vorgestellten Arbeiten stellen neuartige Ansätze zum *Requirements Engineering* [104] dar. Ziel dieser Ansätze ist es, Spezifikationsmodelle lernbasiert zu erfassen. Bei diesen Ansätzen nimmt ein Anwendungsexperte die Rolle des Orakels ein (siehe auch Abschnitt 3.2.1).

Die Aufgabe des Anwendungsexperten ist es, die generierten Beispiele in gute und schlechte Fälle zu klassifizieren und anschließend die erzeugten Hypothesemodelle zu begutachten. Bei diesen Ansätzen hat es sich als hilfreich herausgestellt, den Anwendungsexperten, mit Hilfe von temporallogischen Formeln, größere Mengen von Anfragen auf einen Schlag klassifizieren zu lassen und dadurch das ermüdende manuelle Beantworten vieler Anfragen zu vermeiden.

Im Gegensatz zu anderen modellbasierten *Requirements Engineering* Ansätzen wie beispielsweise *CREWS* [86], *KAOS* [47] oder *Inquiry Cycle* [110], steuert bei den hier vorgestellten Arbeiten ein (aktiver) Lernalgorithmus die Erzeugung von Beispielen und trägt so dazu bei, dass die Spezifikation vollständig überdacht wird. Es gibt auch eine lernbasierte Erweiterung von *KAOS* [45]. Bei diesem Ansatz wird aber ein passiver Lernalgorithmus eingesetzt, so das keine zusätzlichen Beispiele abgefragt werden können.

Modelbased Regression Testing stellt eine Erweiterung vom herkömmlichen Regressionstest dar. Die wesentliche Idee von Regressionstestes besteht darin, durch Wiederholung von Testfällen, Nebenwirkungen von Modifikationen in bereits getesteten Teilen einer Anwendung festzustellen. Unter modellbasiertem Regressionstest verstehen wir einen Regressionstest, der Automatenmodelle zur Spezifikation von Testsuiten verwendet und es so ermöglicht, modellbasierte Testenverfahren zum validieren von Systemen einzusetzen.

In den Abschnitten 5.5, 5.6 und 5.7 werden diese Ansätze vorgestellt.

5.1 Anforderungs-Vervollständigung am Beispiel biologischer Prozesse

Eine Idealvorstellung der Softwaretechnik ist es, Anforderungen an ein komplexes System formal erfassen zu können und daraus direkt einen entsprechenden Code generieren zu können. Ein vielversprechendes Konzept, um dieses Ziel zu erreichen, bietet der *R2D2C*-Ansatz [69]. *R2D2C* steht für „Requirements to Design to Code" und stellt ein generelles Verfahren dar, um Systemanforderungen maschinell in formale Modelle zu transformieren, die möglichst genau die gewünschte Applikation reflektieren. *R2D2C* ist ein Forschungsprojekt der NASA und wird bei der Entwicklung hoch zuverlässiger, autonomer, eingebetteter Systeme eingesetzt, beispielsweise der Steuerungssoftware von Robotern zur Ausführung von komplexen Montage- und Reparaturarbeiten, wie dem Austausch der Kameras am Hubble Space Teleskop.

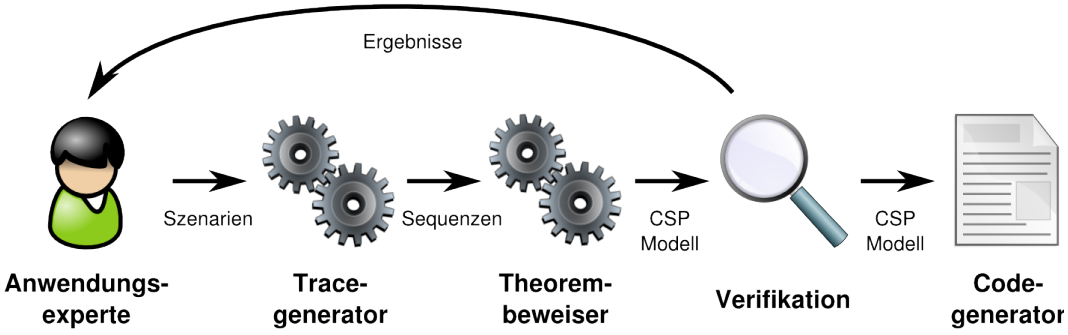

Abbildung 5.1: Der R2D2C-Ansatz

Der *R2D2C*-Ansatz ist in Abbildung 5.1 dargestellt und arbeitet in folgenden fünf Phasen.

1. Entwickler, Endbenutzer und Andere spezifizieren textuell oder graphisch Szenarien, die das geplante Verhalten des Systems abbilden (Anforderungsanalyse).

2. Die Szenarien werden zu Sequenzen von atomaren Ereignissen aufbereitet (Sequenzbildung).

3. Mit Hilfe eines Theorembeweisers wird aus den Sequenzen ein formales Modell bzw. eine formale Spezifikation abgeleitet. Die Spezifikation wird als *Communicating Sequential Processes* (CSP) [122] dargestellt (Modellextrapolation).

4. Basierend auf der formalen Spezifikation können diverse Analysen durchgeführt werden. Aufgedeckte Fehler dienen zur Verbesserung der Anforderungsanalyse (Verifikation).

5. Die formale Spezifikation wird in ausführbaren Code übersetzt (Codegenerierung).

Spezifikationen im Sinne von individuellen Sequenzen können das Gesamtsystem immer nur sehr partiell beschreiben. Oft werden nur die wichtigsten und offensichtlichen Szenarien beschrieben. Diese Unvollständigkeit ist ein wesentliches Problem bei der Anforderungsanalyse.

In Zusammenarbeit mit der NASA ist daher ein Ansatz entwickelt worden, der dieser Problematik entgegenwirkt. Der *R2D2C*-Ansatz wurde um ein Automatenlernverfahren zur Vervollständigung der Anforderungsanalyse ergänzt. Der *R2D2C*-Prozess bietet zwei Angriffspunkte für den Einsatz des Automatenlerners.

- Der Lernalgorithmus wird zwischen Phase 2 und 3 eingesetzt und mit den Sequenzen aus Phase 2 initialisiert. Die zur Bildung einer konsistenten, abgeschlossenen Hypothese notwendigen *Membership Queries* stellen mögliches, unspezifiziertes Verhalten dar. Sie müssen von Anwendungsexperten und Entwicklern klassifiziert werden und tragen so zur Vervollständigung der Anforderungsanalyse bei.

- Der Lernalgorithmus ersetzt den Theorembeweiser zur Modellextrapolation aus Phase 3 vollständig. Wie zuvor bilden die Sequenzen aus Phase 2 eine initiale Menge von (Gegen-) Beispielen zur Initialisierung des Lerners. Die Hypothese des Lerners stellt direkt die Spezifikation dar. Diese Vorgehensweise schränkt den *R2D2C*-Ansatz auf reguläre Systeme ein.

Der Ansatz wird in der Veröffentlichung „Completing and Adapting Models of Biological Processes" [88] vorgestellt und auf die Erfassung von biologischen Prozessen angewendet. Das generelle Konzept dieses Ansatzes wurde im Rahmen einer Diplomarbeit auf die Erfassung von Arbeitsabläufen und Geschäftsprozessen übertragen und untersucht. Die Ergebnisse sind im nachfolgendem Abschnitt 5.2 dargestellt.

5.2 *Automatic Business Process Reengineering*

Bei der Entwicklung von Softwaresystemen ist die Erfassung der Anforderungen an das zu erstellende System der Kernbestandteil in den ersten Entwicklungsphasen. Eine exakte und ausführliche Erfassung dieser Anforderungen ist essenziell

für jedes größere Software Projekt, da es sonst Gefahr läuft, die ihm zugedachten Aufgaben nicht zu erfüllen. Insbesondere wenn außenstehende Entwickler für eine ihnen fremde Organisation ein Softwaresystem entwerfen und entwickeln sollen, muss die Analyse der Anforderungen mit Sorgfalt durchgeführt werden. Eventuell vorhandene Spezifikationslücken können von den Entwicklern auf Grund von Unkenntnis der unausgesprochenen, firmeninternen, impliziten Vorgehensmodelle und Strukturen nicht aufgedeckt werden. Daher besteht immer die Möglichkeit, dass Mängel dieser Art erst zur Einführung des Softwaresystems aufgedeckt werden.

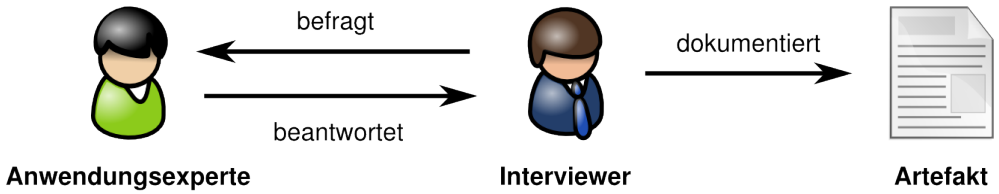

Abbildung 5.2: Klassisches Softwareengineering-Interview [97]

Eine weitverbreitete Methode zur Anforderungserfassung ist das klassische Interview, dargestellt in Abbildung 5.2. In der klassischen Interviewsituation befragt ein Entwickler üblicherweise in mehreren Sitzungen einen Anwendungsexperten, der mit den Anforderungen und den betrieblichen Abläufen vertraut ist. Der Entwickler dokumentiert die Anforderungen an die Anwendung, beispielsweise in UML- Anwendungsfalldiagrammen. Am Ende der Interviewphase sollen alle notwendigen Anforderungen erfasst und hinreichend genau dokumentiert sein. Eine Garantie für eine vollständige Erfassung gibt es nicht. Vielmehr ist es wahrscheinlich, dass Spezifikationslücken unentdeckt bleiben.

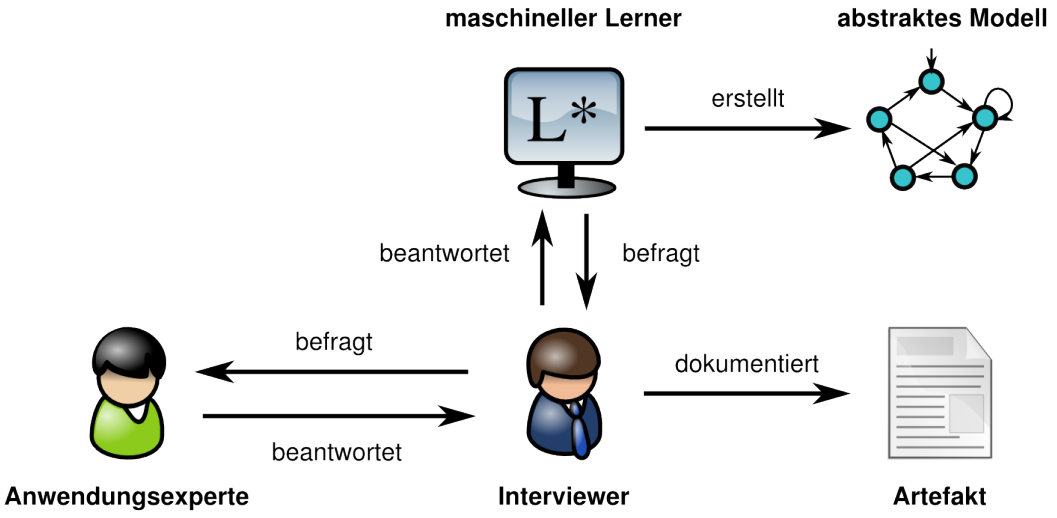

Abbildung 5.3: Interview mit lernbasierter Anforderungsvervollständigung [97]

Im Rahmen der Diplomarbeit „Automatic Business Process Reengineering" [97] wurde, um diesem Problem entgegenzutreten, ein lernbasierter Ansatz zur formalen Erfassung von Arbeitsabläufen realisiert. Auf Basis der LearnLib entstand ein

Softwaresystem, dass den Analysten bei der Erfassung von Anforderungen während eines Interviews mit einem Anwendungsexperten unterstützt, mit dem Ziel, die in einem Unternehmen vorhandenen Prozesse möglichst vollständig zu erfassen. Bei diesem Ansatz steuert der Automatenlernalgorithmus das Interview und sorgt dafür, dass mögliche alternative Arbeitsabläufe hinterfragt werden. Diese Vorgehensweise ist in Abbildung 5.3 dargestellt.

Da aktive Lernverfahren eine Vielzahl von Beispielen erzeugen, die bei einer naiven Implementierung von dem Analysten manuell in gültige und ungültige Arbeitsabläufe klassifiziert werden müssen, erlaubt es die *BusinessLearner* getaufte Software, Beziehungen zwischen einzelnen Arbeitsschritten anzugeben, mit deren Hilfe sich ein Großteil der vom Lernalgorithmus generierten Anfragen automatisch beantworten und somit herausfiltern lässt. Bei den Beziehungen zwischen einzelnen Arbeitsschritten handelt es sich um temporal-logische Bedingungen. Um die Benutzerschnittstelle möglichst einfach zu halten, bietet der *BusinessLearner* anstelle der freien Definition von temporal-logischen Eigenschaften in z.B. PLTL, eine Menge von einfachen Beziehungsschablonen an, mit denen Abhängigkeiten von Arbeitsschritten beschrieben werden können.

In der Diplomarbeit [97] werden insgesamt acht unterschiedliche Fallstudien betrachtet, die ein breites Spektrum von Anwendungsgebieten umfassen. Ein Modell zum wechselseitigen Ausschluss, ein Kaffeeautomat, das Gesetzgebungsverfahren der Bundesrepublik (für Gesetze, die nicht in die Verwaltungshoheit der Länder eingreifen) und der Prozess der Reisekostenabrechnung am Lehrstuhl 5 (TU Dortmund / Fakultät für Informatik), seien hier exemplarisch genannt. Die von dem automatischen Lernverfahren erzeugte Anfragemenge konnte in allen untersuchten Fällen durch den Einsatz von temporal-logischen Regeln auf ein für Benutzer erträgliches Maß reduziert werden.

5.3 *Message-Passing Automata*

Die Synthese verteilter Systeme aus benutzerdefinierten Szenarien ist ein aktuelles Forschungsthema. Es gibt bereits eine Reihe von Ansätzen in diesem Themengebiet. Die Arbeiten umfassen Algorithmen zur Erstellung von Zustandsmodellen aus *Message Sequence Charts* (MSC) [80, 6], Modellsynthese basierend auf der Basis von *Live Sequence Charts* [25, 46] und ähnliche Varianten [125, 132]. Die Gemeinsamkeit aller dieser Ansätze ist, dass sie auf weitgehend vollständige, detaillierte Spezifikationen des Systems angewiesen sind. Derzeit ist nur ein weiterer lernbasierter Ansatz bekannt [100], der sich aber auf die Abbildung des globalen Systems und auf synchrone Protokolle beschränkt.

Auf Basis der LearnLib wurde das Tool Smyle entwickelt. Smyle [22] steht für „**S**ynthesizing **M**odels b**y** **L**earning from **E**xamples" und ist ein Werkzeug zur Synthese asynchroner, verteilter Implementierungsmodelle. Grundlage sind hierbei Szenarien in Form von *Message Sequence Charts*, die in positive und negative Beispiele klassifiziert werden. Smyle verwendet den DFA-Lernalgorithmus der LearnLib und propositionale dynamische Logik (PDL) über *Message Sequence Charts* zur

Erzeugung von Systembeschreibungen in Form von *Message-Passing* Automaten, welche die vorliegenden Beispiele reflektieren. *Message-Passing Automata* (MPA) [26] bestehen aus einer Menge von endlichen Automaten (Prozessen), mit einem gemeinsamen globalen Initialzustand und einer gemeinsamen Menge von akzeptierenden Zuständen. Die Kommunikation unter den Prozessen geschieht mittels *First-In First-Out*-Warteschlangen. Die Transitionen lassen sich in Sende- und Empfangsaktionen unterteilen. Sendeaktionen schreiben ein Symbol in eine Warteschlange, Empfangsaktionen lesen ein Symbol aus einer Warteschlange. Empfangsaktionen sind nur möglich, wenn sich entsprechende Symbole am Kopf einer Warteschlange befinden. Die theoretischen Grundlagen von Smyle sind in [21] detailliert beschrieben.

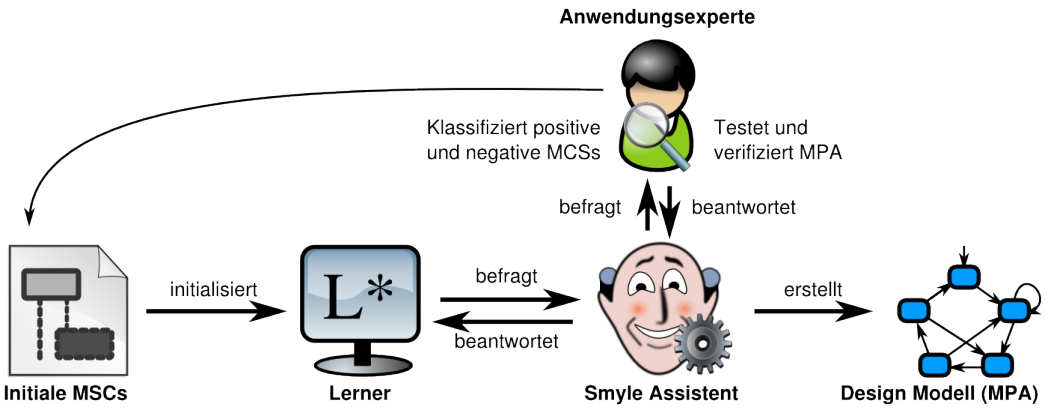

Abbildung 5.4: Der Smyle-Prozess

Der Smyle-Prozess ist in Abbildung 5.4 dargestellt und beginnt mit dem sogenannten „learning setup". Zunächst müssen Randparameter festgelegt werden, wie beispielsweise die maximalen Kapazitäten der Warteschlangen. Danach wird der Lernalgorithmus mit einer initalen Menge von MSCs initialisiert. Die *Membership Queries* des Lernalgorithmus werden vom sogenannten *Smyle Assistent* aufbereitet. Auf Grund struktureller Eigenschaften des Übersetzungsprozess von Anfragen in MSCs kann eine Vielzahl von Anfragen direkt vom *Smyle Assistent* beantwortet werden [21]. Die verbleibenden Anfragen werden dem Anwendungsexperten zur Klassifikation in gültige und ungültige Beispiele in Form von *Message Sequence Charts* vorgelegt. Der Anwendungsexperte klassifiziert die Beispiele und kann darüberhinaus Formeln in propositionaler dynamischer Logik (PDL) über *Message Sequence Charts* angeben, mit denen sich gewünschtes und ungewünschtes Verhalten lose spezifizieren lässt. Der *Smyle Assistent* verwendet diese Formeln zur Filterung von Anfragen. Wenn der Lernprozess eine Hypothese bilden kann und der *Smyle Assistent* ein vollständiges und konsistentes MPA-System bilden kann, wird dieses dem Anwendungsexperten zur Begutachtung und zur Simulation vorgelegt. Entdeckt der Anwendungsexperte falsches oder fehlendes Verhalten, kann er die entsprechenden MSCs als Gegenbeispiel an den Lerner zurückgeben und der Lernprozess fährt fort bis zur nächsten konsistenten Hypothese.

Smyle ist in Java entwickelt und bindet die LearnLib als entfernten CORBA-Dienst

an. Dies hat den Nachteil, dass Smyle auf eine Internet-Verbindung zur LearnLib angewiesen ist, bietet aber den Vorteil, dass Smyle unmittelbar von Verbesserungen der LearnLib profitiert und entlastet gleichzeitig lokale Rechner vom Lernprozess, insbesondere beim Lernen großer Systeme. Die Anbindung von Smyle an die LearnLib auf Basis der jETI-Plattform wird in der Veröffentlichung „The LearnLib in FMICS-jETI" [92] beschrieben.

5.4 ECA-Regelsysteme

Event-Condition-Action-Regelsysteme (ECA-Regelsysteme) nehmen in vielen Anwendungen einen hohen Stellenwert ein. Vor allem Anwendungen, die sich durch reaktives Verhalten charakterisieren lassen, d.h. bestimmte Situationen erkennen und darauf angemessen reagieren müssen, werden häufig mit Hilfe von ECA-Regelsystemen realisiert.

ECA-Regelsysteme sind im Bereich aktiver Datenbanksysteme [49, 51, 107] entwickelt worden. Mittlerweile unterstützen fast alle relationalen Datenbanken aktive Regeln [36]. Das Konzept ist aber auch für viele andere Anwendungsgebiete geeignet. ECA-Regelsysteme werden verwendet, um ereignisorientierte Web-Services zu realisieren [23, 24], sie unterstützen Workflow-Management-Systeme [81, 77] und finden auch im Bereich der Telefonie Anwendung [117].

ECA-Regeln folgen folgendem Paradigma:
Bei Eintritt eines Ereignisses (*Event*),
wird eine Bedingung überprüft (*Condition*)
und dann gegebenenfalls eine Aktionen ausgelöst (*Action*).

Beispiele: In einem Warenwirtschaftssystem soll, bei Änderung des Bestandes (*Event*), wenn ein Mindestbestand unterschritten wird (*Condition*), der Artikel automatisch beim Lieferanten nachbestellt werden (*Action*).
In einem Krankenhaus soll, bei Verlegung eines Patienten auf eine andere Station (*Event*), wenn sich der Gesundheitszustand des Patienten verschlechtert hat (*Condition*), der leitende Arzt darüber informiert werden (*Action*).
Bei einem Telefonsystem soll, Teilnehmer A ein besetzt Zeichen erhalten (*Action*), wenn Teilnehmer A die Nummer von Teilnehmer B wählt (*Event*) und Teilnehmer A auf der Blacklist von Teilnehmer B eingetragen ist (*Condition*).

Event-Condition-Action-Regeln bestehen aus:

Ereignissen, welche von dem Regelsystem erkannt werden müssen. Man unterscheidet zwischen primitiven Ereignissen, die nicht weiter zerlegt werden können (Änderung von Daten, Timer, ...), und komplexen Ereignissen, welche durch Kombination von primitiven Ereignissen und Ereignisoperatoren gebildet werden.

Bedingungen, mit denen sich prüfen lässt, in welchen Fällen der Aktionsteil der Regel ausgeführt werden soll. Bedingungen bestehen typischerweise aus logischen und auch mathematischen Ausdrücken über dem aktuellen Zustand des Systems.

Aktionen, die festlegen, wie auf eine bestimmte Situation reagiert werden soll. Aktionen von ECA-Regelsystemen sind naturgemäß anwendungsspezifisch und bieten häufig die Mächtigkeit imperativer Programmiersprachen.

Die Definition einzelner Regeln ist einfach und intuitiv. Ein ECA-Regelsystem besteht aber normalerweise aus sehr vielen einfachen Regeln. Der globale Einfluss einer einzelnen Regel auf den durch das Regelsystem definierten Gesamtdienst ist aber nicht offensichtlich und im Allgemeinen nicht einmal berechenbar. In diesem Sinne können ECA-Regelsysteme als Black-Box-Systeme aufgefasst werden. Oft ist es kaum möglich nachzuvollziehen, was passiert, wenn nur eine einzige Regel hinzugefügt, verändert oder gelöscht wird. Kleinste Variantenbildung an thematisch entfernten Modulen kann dazu führen, dass geforderte Eigenschaften des Gesamtsystems nicht mehr erfüllt werden können. Diese Problematik tritt insbesondere bei der Entwicklung von Telefonsystemen auf und wird im Forschungsgebiet *Feature Interaction* untersucht [33].

Beispiel: Eine Autofirma hatte ein neues Sicherheitssystem (also ein komplexes Security-Feature) für ihre Luxusklasse vorgestellt und einen Preis für den geboten, dem es gelingt, den Wagen ohne Schlüssel zu öffnen. Ein „Experte" nahm daraufhin einen Gummihammer und schlug auf die vordere Stoßstange, worauf alle Türen aufsprangen — eine Maßnahme zur Bergung von Verletzten nach einem (hier vermeintlichen) Unfall (ein Personenschutz-Feature). Das wegen der Wichtigkeit des Personenschutzes vorrangige Feature hatte den Security-Mechanismus außer Kraft gesetzt. Crash-Sensoren, die einen Unfall feststellen und automatisch die Türverriegelung lösen, gehören mittlerweile auch bei Kleinwagen zur Serienausstattung (Mini Cooper, Smart, ...). Es ist offensichtlich, wie derartige Effekte Dienste jeder Art bedrohen können. Aufgedeckt werden können sie nur auf Basis globaler Modelle, die das Gesamtsystem abdecken.

Da einzelne Regeln immer nur sehr lokale Verhaltensweisen und Eigenschaften eines Systems beschreiben, ist es nicht möglich globale Eigenschaften des Gesamtsystems anhand der Regeln abzulesen oder gar formal nachzuweisen. Globale Eigenschaften lassen sich andererseits leicht mit Hilfe temporal-logischer Formeln formal beschreiben. Tatsächlich ist es möglich, mit Hilfe formaler Verifikationstechniken *Feature Interactions*, beispielsweise durch *Model Checking*-Techniken, unmittelbar zu erkennen [34]. Diese Methoden setzen aber in der Regel formale Modelle in Form erweiterter (endlicher) Automaten (Transitionssysteme) voraus. Eine direkte Umwandlung von regelbasierten Systemen in Transitionssysteme gemäß einer operationalen Semantik scheitert aber typischerweise daran, dass die entstehenden Transitionssysteme unendlich groß oder zumindest so groß würden, dass sie nicht mehr beherrscht werden können.

Im Rahmen der Projektgruppe „LiVe: Lernen Impliziter Verhaltensmodelle” [3], einer Lehrveranstaltung der TU Dortmund, sollten daher zwölf Studenten in einem Softwareprojekt ein Framework entwickeln, das es erlaubt, mit Hilfe von Automatenlernverfahren, aus ECA-Regelsystemen globale Modelle zu generieren. Die gelernten Modelle und Hypothesen sollten dann mittels *Model Checking* untersucht werden. Da existierende ECA-Regelsystembibliotheken nicht hundertprozentig zum Anforderungsprofil dieses Projektes passten und insbesondere die, für den Einsatz von aktiven Lernverfahren wichtige „Reset”-Operation, nicht verfügbar war, entwickelte die Projektgruppe ein eigenes Regelsystemframework und evaluierte den Ansatz an mehreren Beispielen, von denen im Folgenden eines exemplarisch vorgestellt wird.

System	Zustände (2 Teilnehmer)		Zustände (3 Teilnehmer)	
	abstrakt	vollständig	abstrakt	vollständig
Basissystem	11	13	45	63
Basissystem + INTL	14	19	56	89
Basissystem + TCS	10	11	42	57
Basissystem + CFB	–	–	45	63

Tabelle 5.1: Telefonsysteme als ECA-Regelsysteme

Die Projektgruppe untersuchte vier verschiedene Versionen des von Aiguier, Berkani und Le Gall als Regelsystem modellierten Telefonsystems [4]: das Basis-Telefonsystem, die Integration des Dienstes „Anrufweiterleitung” (CFB), die Integration des Dienstes „Sperrung ausgehender Anrufe” (INTL) und die Integration des Dienstes „Abschirmung eingehender Anrufe” (TCS) in das Basissystem. Die Beispiele wurden bis auf die Anrufweiterschaltung für zwei und drei Teilnehmer mit unterschiedlichen Abstraktionsgraden der Ausgaben untersucht. Die Größen der gelernten Modelle sind in Tabelle 5.1 dargestellt. Die globalen Eigenschaften der Features CF, INTL, und TCS konnten mit Hilfe des *Model Checkers* GEAR anhand der abstrakten und der vollständigen Modelle nachgewiesen werden.

Im Nachhinein betrachtet ist der von der Projektgruppe realisierte Ansatz vergleichbar mit dem *Black-Box-Checking*-Ansatz von Peled, Vardi und Yannakakis [108]. Hierbei werden durch Lernverfahren iterativ Hypothesemodelle aus *Black-Box*-Systemen gewonnen, die dann mit Hilfe eines *Model Checkers* auf die Einhaltung globaler Eigenschaften hin geprüft werden. Die Projektgruppe konnte erfolgreich zeigen, das sich die Kombination von Automatenlernen und *Model Checking* vielfach zur Verifikation von ECA-Regelsystemen einsetzen lässt.

5.5 Programmiersprachen

Im Rahmen der Projektgruppe „TTCP: Test and Testing Control Plattform” [5], einer Lehrveranstaltung der Universität Dortmund, entstand die Idee, Automatenlernverfahren zur Testfallgewinnung für den Compilerbau einzusetzen. Die zwölf

an dem Projekt beteiligten Studenten hatten unter anderem zur Aufgabe, einen frei verfügbaren Compiler für die Programmiersprache *TTCN-3* zu entwickeln. *TTCN-3* steht für Testing and Test Control Notation (Version 3) und ist eine Sprache zur Beschreibung von ausführbaren Prüfvorschriften und Testspezifikationen [63, 64]. *TTCN-3* stellt eine einheitliche Testnotation zur Beschreibung von Testverhalten und Testverfahren dar. In *TTCN-3* ist es möglich, Testeingaben und -ausgaben, einfache und komplexe Testverhalten im Sinne von Sequenzen, Alternativen und Schleifen, zu beschreiben. Mit *TTCN-3* lassen sich verschiedene Arten von Tests durchführen, wie z.B. Robustheits-, Leistungs-, Performanz-, System- und Integrationstests. Anwendungsbereiche von TTCN-3 sind Protokolle, APIs und Software-Module. *TTCN-3* ist eine komplexe Programmiersprache: Ihre Spezifikation in erweiterter Backus-Naur-Form *(EBNF)* [74] besteht aus über 600 Nichtterminalen. Im Vergleich dazu bestehen die in *EBNF* dargestellten Grammatiken von *Java 1.5* und *ANSI C* nur aus 100 bis 200 Nichtterminalen.

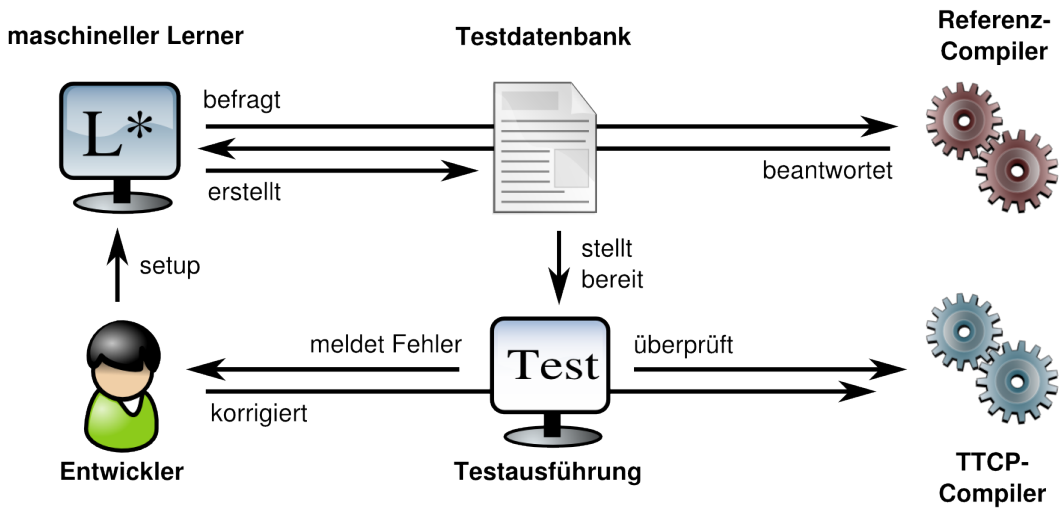

Abbildung 5.5: TTCP-Prozess

Der Projektgruppe stand der kommerzielle TTCN-3 Compiler aus TTWorkbench der Firma Testing Technologies als Referenzimplementierung zur Verfügung. Neben der Verwendung selbstentwickelter Testsuiten und frei verfügbarer TTCN-3-Beispiele zum Testen, lag es nahe, die LearnLib zum vollautomatischen Erzeugen und Klassifizieren von Beispielcode zu verwenden. Der Ansatz ist in Abbildung 5.5 dargestellt. Das Eingabealphabet und initiale (Gegen-) Beispiele wurden hierbei aus bestehendem TTCN-3-Quellcode extrahiert und zum Initialisieren des Lernalgorithmus verwendet. Der Lernalgorithmus sollte versuchen, die vom Referenzcompiler akzeptierte Sprache zu lernen und dabei eine Testdatenbank generieren, die einer Menge von zufälligen Eingaben überlegen ist. Die Testdatenbank sollte zur Validierung des neu entwickelten Compilers verwendet werden.

Programmiersprachen sind in der Regel nicht kontextfrei, so dass der Lernprozess, welcher reguläre Sprachen abbildet, von vornherein zum Scheitern verurteilt war. Es bestand aber immernoch die begründete Hoffnung, auf diesem Wege eine brauchbare Menge von positiven und negativen Testfällen zu erhalten.

Zusammenfassend lässt sich sagen, dass dieser Ansatz nicht erfolgreich war. Auch in den verschiedensten Konstellationen brach der Lernalgorithmus nach wenigen Stunden Auf Grund fehlender Hauptspeicherressourcen ab[1]. Bis dahin wurden zwar mehrere hunderttausend ungültige Eingaben erzeugt, aber nur sehr wenige Gültige. Da sich auf diesem Wege keine sinnvollen Testsuiten erzeugen ließen, wurde der Ansatz nicht weiter verfolgt.

5.6 *Computer Telephony Integration*

Computer Telephony Integration (CTI) ist die Verknüpfung von Telekommunikation mit elektronischer Datenverarbeitung. Sie wird in vielen Anwendungsbereichen eingesetzt, Schwerpunkt sind aber Call-Center-Anwendungen. Die CTI ermöglicht, aus Computerprogrammen heraus den automatischen Aufbau, die Annahme und Beendigung von Telefongesprächen, den Aufbau von Telefonkonferenzen, das Senden und Empfangen von Faxnachrichten, Telefonbuchdienste, die Weitervermittlung von Gesprächen, und vieles mehr. Diese Systeme sind sehr komplex und werden aus vielen heterogenen Modulen aufgebaut. Auf diese Art von Systemen lassen sich etablierte Methoden zur Qualitätssicherung, insbesondere modellbasiertes Testen und formale Verifikation nicht anwenden. Grund ist, dass diese Applikationen vielfach nicht formal spezifiziert sind und das Gesamtverhalten auf Systemebene — wenn überhaupt — in unterschiedlichen Abstraktionsgraden und typischerweise nicht formal dokumentiert sind.

Ein erfolgreicher Ansatz zum Testen dieser Systeme besteht darin, wiederverwendbare Bausteine für einzelnen Testschritte, wie beispielsweise das Signalisieren eines eingehenden Anrufs, das Entgegennehmen des Anrufs oder auch einfache Aktionen wie das Auflegen eines Telefonhörers, zu erzeugen. Mit Hilfe dieses Baukastens werden dann Testfälle spezifiziert, welche die vorgegebenen Anwendungsfälle abdecken. Ein Vertreter dieser Art von Testwerkzeugen ist das ITE (*Integrated Test Environment*) [68, 89].

In diesem Szenario können die ausführbaren Testschritte direkt als Alphabet für Automatenlernverfahren verwendet werden. Die Testumgebung liefert die Ausführungsplattform für die Anfragen des Lernalgorithmus und und trägt so dazu bei, durch Beobachtung aussagekräftige Hypothesemodelle für das Gesamtsystem zum Zwecke der Verifikation und Validierung zu gewinnen [67, 71]. Die synthetisierten Hypothesemodelle eignen sich bei diesem Vorgehen in idealer Weise zum Ableiten von Testsuiten, da sich die Transitionen dieser Modelle wieder Eins-zu-Eins auf die Testbausteine des ITE abbilden lassen.

Auf Basis des ITE wurde eine Reihe von Modellen aus realen Telefonsystemen extrapoliert. Die von Hungar et al. gewonnenen Ergebnisse [72], auszugsweise dargestellt in Abbildung 5.6, zeigen in aller Deutlichkeit die Wirksamkeit der in Abschnitt 4.3 beschriebenen Filtertechnologie bei der Extrapolation realer, komplexer Systeme.

[1] Zu diesem Zeitpunkt war nur die 32Bit-Version der LearnLib verfügbar.

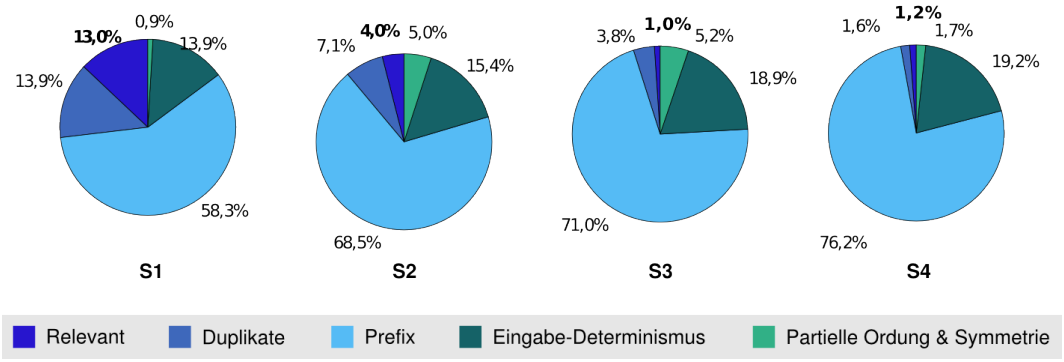

Abbildung 5.6: Filtereffektivität bei CTI-Systemen [91]

In der Abbildung ist die Effektivität der Filter in unterschiedlichen Szenarien dargestellt. In den Szenarien $S1$ bis $S3$ variiert die Anzahl der Telefone (von 1 bis 3) und die einzig möglichen Aktionen sind das Abnehmen und das Auflegen der Telefonhörer. Das Szenario $S4$ erweitert $S3$ um die Aktion, von einem Telefon ein anderes anzuwählen. In allen Szenarien werden die Reaktionen der Telefonanlage auf diese Eingaben als Ausgaben betrachtet. Da bei diesem Ansatz der ursprüngliche L^* Algorithmus zum Lernen von DFAs eingesetzt wurde, bilden Eingaben und Ausgaben zusammen das Alphabet für den Lernalgorithmus und das System wird umständlich auf einen deterministischen endlichen Automaten abgebildet. Auf Grund dieser Modellierung konnte ein weiterer Filter (Eingabe-Determinismus) eingesetzt werden, welcher der Erkenntnis Rechnung trägt, dass die Eingabe- und Ausgabe- Symbole nur alternierend auftreten können [72].

Szenario	L^*	L^*_{opt}	L^*_{Mealy}	$\frac{L^*}{L^*_{Mealy}}$	$\frac{L^*_{opt}}{L^*_{Mealy}}$
	(Anzahl von Anfragen)				
$S1$	108	14	10	10,8	1,4
$S2$	2421	97	26	93,5	3,7
$S3$	19426	206	42	462,5	4,9
$S4$	132300	1606	314	388,0	4,7

Tabelle 5.2: Vergleich von DFA- und Mealy- basiertem Lernen von CTI Systemen

Der direkte Einsatz des Mealylernalgorithmus L^*_{Mealy} führt, selbst ohne den Einsatz von Filtern, beim Lernen dieser Systeme nochmal zu einer deutlichen Verringerung der Anfragemenge, wie sich leicht aus Tabelle 5.2 ablesen lässt. Diese Resultate sind in der Veröffentlichung „Efficient Test-based Model Generation of Legacy Systems" [129] ausführlich diskutiert. Der gesamte Ansatz und eine genauere Betrachtung der Filtereffektivität und der Abhängigkeiten der Filter untereinander ist in der Veröffentlichung „Knowledge-Based Relevance Filtering for Efficient System-Level Test-Based Model Generation" *(Journal)* [91] beschrieben.

5.7 Dynamisches Testen von Web-Applikationen

Testen ist ein zentraler Bestandteil des Entwicklungsprozesses und der Qualitätssicherung. Zur Entwicklungszeit stellt man im Allgemeinen durch geeignete Tests sicher, dass neue Funktionalitäten wie gewünscht funktionieren. Im Rahmen von Regressionstests wird, bei fortschreitender Entwicklung, durch die Wiederholung aller oder einer Teilmenge aller Testfälle versucht, Nebenwirkungen von Modifikationen in bereits getesteten Teilen der Software aufzuspüren. Tests des Gesamtsystems aus der Benutzerperspektive werden hierbei häufig mit Record- und Replay-Techniken durchgeführt. Hierbei bedient ein Testingenieur, unterstützt durch ein entsprechendes Testautomatisierungswerkzeug, wie ein normaler Benutzer die Applikation und bildet die geforderten Anwendungsfälle nach. Das Testautomatisierungswerkzeug zeichnet die Aktionen des Testingenieurs auf und erstellt typischerweise ausführbare Skripten. Diese Skript enthalten sämtliche zur Testausführung benötigten Informationen und können angepasst, erweitert und wiederverwendet werden. Prominente Vertreter dieser Art von Testautomatisierungswerkzeugen sind das HP Quality Center oder der Rational Functional Tester, wobei sich beide Plattformen als „Rundum-Sorglos"-Lösungen verstehen, die den gesamten Lebenszyklus des Qualitätssicherungsprozesses abdecken.

5.7.1 Record und Replay

Record und Replay ist eine weit verbreitete Methode, um auf Benutzerlevel automatisiert abzusichern, dass sich eine Applikation entsprechend ihrer Spezifikation verhält. Mit diesem Ansatz können aber nur Funktionalitäten getestet werden, für die Testsequenzen aufgezeichnet oder manuell erstellt wurden. Modellbasierte Testverfahren [28] haben das Potential, diese Einschränkung zu überwinden, da sie Testfälle direkt aus einer (formalen) Spezifikation der Applikation erzeugen können. Modellbasierte Testverfahren stellen eine deutliche qualitative Verbesserung im Bereich Testen dar. Sie ermöglichen es, unter anderem, fundierte Angaben zur Testüberdeckung einer Testsuite zu machen und erlauben es, für einzelne Anwendungsaspekte gezielt Testsuiten zu generieren. Voraussetzung dieser, von der Wissenschaft gut verstandenen, Methoden ist aber die Existenz einer formalen Spezifikation des zu testenden Systems. Formale Spezifikationen für real existierende Systeme sind aber in der Praxis eine Seltenheit. Sie lassen sich aber post mortem mit Hilfe von Automatenlernverfahren erzeugen.

In Zusammenarbeit mit einem in Europa führenden IT-Dienstleister für Informations- und Kommunikationstechnologie, wurde eine flexibele Testumgebung für Web-Applikationen entwickelt, welche etabliertes Record- und Replay-Testen mit den Vorteilen von modellbasiertem Testen vereint. Sie trägt bisher nur den schlichten Arbeitstitel „Webtest" und ist in Abbildung 5.7 dargestellt. Bei diesem Ansatz entwickelt der Testingenieur mit einem klassischen Record- und Replay-Verfahren eine Testdatenbank, die als Basis für den Lernalgorithmus dient. Der Lernalgorithmus analysiert die Web-Applikation und erstellt ein formales Modell der Anwendung. Webtest ist eine Weiterentwicklung des in der Veröffentlichung „Dy-

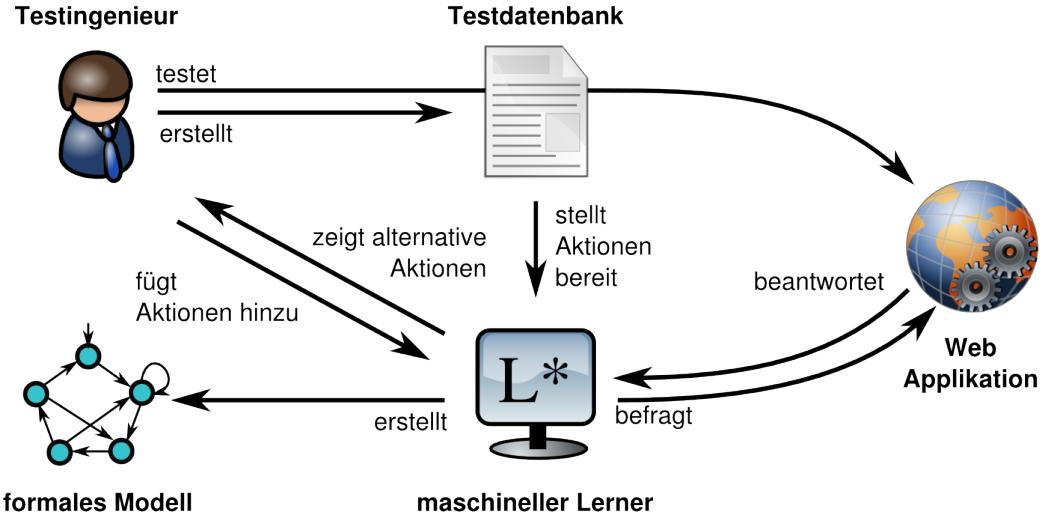

Abbildung 5.7: Der Webtest-Ansatz

namic Testing via Automata Learning" [114] vorgestellten Ansatzes. Im Fokus dieser Arbeit steht die Gewinnung von Spezifikationsmodellen aus realen Web-Applikationen mit Hilfe der LearnLib. Die Kombination dieses Ansatzes mit Record und Replay wird in der Veröffentlichung „Hybrid Test of Web Applications with Webtest" [111] erstmalig vorgestellt. Einen detaillierten Überblick über die gesamte Testumgebung gibt das derzeit im Reviewprozess befindliche *invited Jounal-Paper* „Dynamic Testing via Automata Learning" *(Journal)* [112].

5.7.2 Die Webtest-Testumgebung

Die Webtest-Testumgebung ist als Plugin für das jABC realisiert, um, wie auch die LearnLib, die in Abschnitt 4.1.1 beschriebenen Vorteile des jABC zu nutzen. Das Webtest-Plugin ist als komfortable, skalierbare Umgebung zum Aufzeichen, Ausführen und Verwalten von Testfällen konzipiert. Es wurde direkt im industriellen Kontext entwickelt, wo auch Randbedingungen wie Effizienz und Benutzerfreundlichkeit eine Rolle spielen, damit das Werkzeug intensiv genutzt werden kann. Es bietet darüber hinaus mit Hilfe der LearnLib an, umfangreiche Testsuiten und formale Modelle von realen Web-Applikationen zu erzeugen.

Das Webtest-Plugin bietet, neben der lernbasierten Methode, dem Testingenieur zwei Möglichkeiten an, Testfälle zu spezifizieren. Zum einen kann er, ganz nach jABC-Manier, aus einer Taxonomie einzelne Testschritte auswählen und zu komplexen Testfällen zusammenstellen. Diese Testfälle können neben Kontrollstrukturen herkömmlicher Programmiersprachen auch anwendungsfremde Bausteine aus der gesamten Vielfalt des jABC enthalten. Zum anderen erlaubt es die Firefox-Extension des Webtest-Plugins, Testsequenzen komfortabel durch simples Bedienen der Applikation im Firefox-Browser aufzuzeichnen.

5.7.3 Testschritte für Web-Applikationen

Die Testschritte in den aufgezeichneten und modellierten Testfällen sind identisch. Sie repräsentieren atomare Aktionen, die ein Benutzer beim Bedienen der Anwendung ausführt, wie beispielsweise das Auswählen eines Hyperlinks oder das Ausfüllen und Absenden eines Formulars. Darüber hinaus gibt es noch Verifikationsschritte, mit denen sich der Inhalt einer Webseite auslesen und überprüfen lässt. Mit Hilfe von XPath-Ausdrücken können hierbei gezielt bestimmte Bereiche der Webseiten überprüft werden.

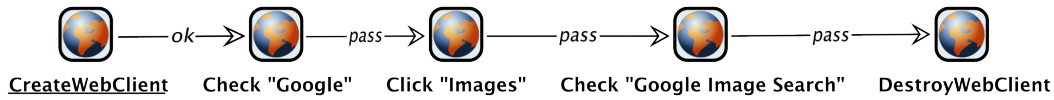

Abbildung 5.8: Graphische Modellierung des „Google" -Testfalls

In Abbildung 5.8 ist ein exemplarischer Testfall für die Google Web-Suche dargestellt. Er beginnt mit der Instanziierung eines Test-Browsers, dessen Startseite über die Parameter des Testbausteins auf „www.google.com" eingestellt ist. Nachdem mit dem nächsten Testbaustein der Titel der Startseite überprüft wurde, wird der Link zur Bildersuche ausgewählt. Der Testfall prüft nun über den Titel der Webseite, ob die Webseite für Bildersuche auch angezeigt wird. Abschließend werden der Test-Browser geschlossen und seine Ressourcen freigegeben. In der Abbildung ist nur der erfolgreiche Teil des Testfalls sichtbar. Alle Testbausteine verfügen aber außer der dargestellten „pass"-Verzweigung noch über eine „fail"-Verzweigung, mit der sie anzeigen können, dass ein Test fehl schlug und über eine „error"-Verzweigung, mit der schwerwiegende Probleme der Testumgebung angezeigt werden, wie beispielweise ein Verbindungsabbruch.

Die „Anwendungsschnittstelle" einer Web-Applikation wird innerhalb des Webtest-Plugins implizit über die atomaren Bausteine der Testfälle abgebildet. Insbesondere können Formulare durch Sequenzen von Testschritten, die einzelne Formularfelder füllen und abschließend den „Submit"-Button drücken, beschrieben werden. Gerade für HTTP-Formulare ist es naheliegend, diese „Schnittstellen" über das HTML-Binding der standardisierten *Web Service Description Language* (WSDL) [38] oder mit Hilfe der *Web Application Description Language* (WADL) [66] zu spezifizieren. Webtest verwendet diese Standards nicht, da sie Web-Applikationen nur auf HTTP-Protokollebene beschreiben und daher völlig ungeeignet sind, diese Anwendungen aus Benutzersicht zu beschreiben. Darüberhinaus ist es auf Basis von protokollbasierten Spezifikationen wie WSDL oder WADL nicht möglich, Testfälle für Client-Seitige, vom Browser ausgeführte Skripten zu definieren. Ein zu testendes Formular des Kooperationspartners hatte beispielsweise die Eigenschaft, nach dem Ausfüllen eines Formularfeldes, in Abhängigkeit des eingegebenen Wertes, dynamisch mit Javascript ein zusätzliches Formularfeld anzuzeigen.

5.7.4 Eingabe- und Ausgabeabstraktion

Automatenlernverfahren erzeugen Spezifikationen in Form von endlichen Automaten. Web-Applikationen können auf einfache Weise als Mealy-Automaten aufgefasst werden. Die Benutzerinteraktionen bilden das Eingabealphabet des Automaten und die erzeugten HTML-Seiten stellen die Ausgaben dar. Für die Anwendung von aktiven Lernverfahren ist es aber in der Praxis undurchführbar, alle möglichen Eingaben und Ausgaben einer Web-Applikation zu betrachten. Aktive Automatenlernverfahren sind eine Größenordnung aufwändiger als erschöpfendes Testen im Sinne von *Conformance Test* Methoden [16]. Darüberhinaus ist das Eingabealphabet typischerweise nicht einmal von endlicher Größe, da in Formularfeldern prinzipiell jede Eingabe möglich ist.

Es müssen Abstraktionen angewendet werden, um die Komplexität des Ansatzes zu beherrschen. Diese Abstraktionen bestehen typischerweise darin, Eingaben und Ausgaben zu äquivalenten Gruppen zusammenzufassen. Es ist beispielsweise häufig ausreichend, Testfälle für nur einen Account einer bestimmten Rolle zu erstellen und nicht für alle Accounts dieser Rolle.

Diese Art von Abstraktion beherrschen Testingenieure von jeher. Mit Hilfe ihres Expertenwissens über die Anwendung führen sie Anwendungsfälle aus, um die Anforderungen zu überprüfen. Selbstverständlich wählen sie dabei relevante Beispieldaten für die Eingaben aus und überprüfen die zurückgelieferten Ausgaben bzw. Web-Seiten auf die wesentlichen Details. Die Einzelschritte der so entwickelten Testfälle bilden daher eine angemessene Menge von Aktionen und Kontrollen, welche direkt als Eingabealphabet für den Lernalgorithmus verwendet werden kann. Auf diese Weise entspricht das Ausgabealphabet nicht mehr direkt den ausgelieferten Web-Seiten der Anwendung. Es bildet sich aus den Rückgabewerten der Aktionsbausteine und den Testurteilen der Kontrollschritte.

Neben der Möglichkeit, vordefinierte Aktionen aus existierenden Testfällen als Alphabet zum Lernen der Anwendung einzusetzten, gibt es auch die Möglichkeit, während des Lernen, weitere Aktionen zum Alphabet hinzuzufügen. Hierbei wird jede Web-Seite, die von der Web-Applikation an das Webtest-Plugin ausgeliefert wird, auf neue, alternative Benutzeraktionen hin durchsucht. Diese werden dann dem Testingenieur in einem Dialog angeboten, und können jederzeit in das Alphabet des Lernalgorithmus eingefügt werden. Darüberhinaus ist das Alphabet des Lernansatzes nicht auf die speziellen Webtest-Bausteine beschränkt.

Grundsätzlich lassen sich, bis auf wenige Ausnahmen, wie beispielsweise die Kontrollstrukturen Fork und Join, alle Bausteine des jABC zum Lernen in das Eingabealphabet aufnehmen. Dies gilt insbesondere für das sogenannte „Graph-SIB”, mit dem sich komplexe Teilprozesse zu einem einzigen Baustein zusammenfassen lassen. Einschränkend muss aber hinzugefügt werden, dass sich nur die Webtest-Bausteine dynamisch zum Alphabet hinzufügen lassen, da nur für diese Bausteine beim Lernen geprüft wird, ob sie sinnvoll angewendet werden können.

5.7.5 Das gelernte Modell

Das gelernte Modell einer Web-Applikation kann im jABC als Kontrollflussgraph dargestellt werden. Das Modell in Abbildung 5.9 ist das Ergebnis des Lernprozesses unter Verwendung der Aktionen des „Google"-Testfalls aus Abbildung 5.8. Die Zustände des Mealy-Automaten werden durch die (Test)-Schritte mit dem „Würfelsymbol" abgebildet. Diese Würfel-Bausteine wählen bei ihrer Ausführung einen ihrer Nachfolger zufällig aus. Die Aktionen der Transitionen des Automaten hängen als Testbausteine zwischen den Würfel-Bausteinen. Die Ausgaben der Transitionen finden sich in den Beschriftungen der ausgehenden Kanten ihrer zugehörigen Aktions-Bausteine wieder. Dieses Modell ist direkt im jABC ausführbar und stellt einen randomsierten Testfall für die Anwendung dar.

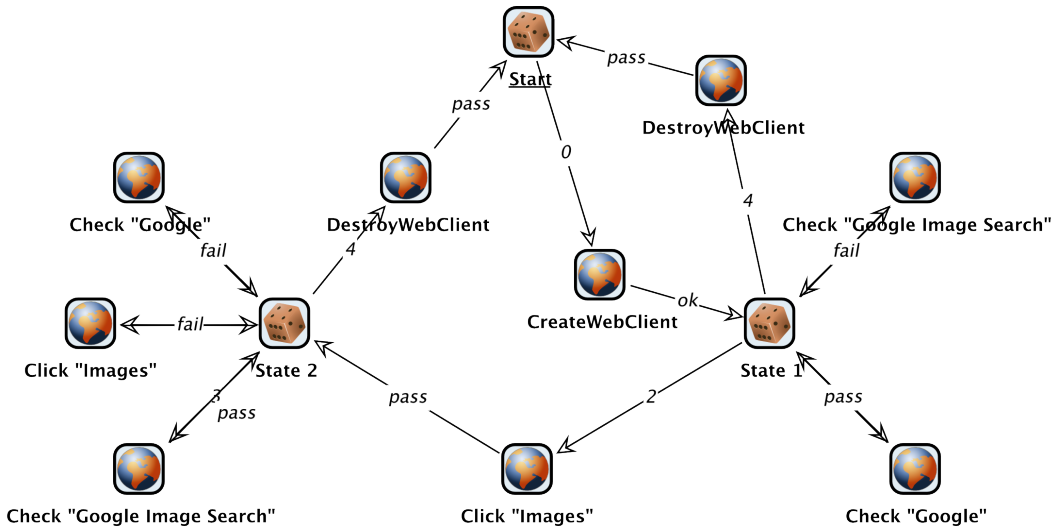

Abbildung 5.9: Gelerntes Modell einer Web-Applikation

Das Webtest-Plugin realisiert einen neuartigen Ansatz der systematisch Black-Box-System testet. Ausgehend von einfachen Benutzerinteraktionen erkundet der Lernalgorithmus nach und nach eine Web-Applikation und extrapoliert gleichzeitig ein formales Modell der Anwendung. Auf diese Weise können Testsuiten gebildet werden, die sich in idealer Weise für Regressionstests eignen. Die gewonnenen Modelle können als Basis für eine breite Palette von formalen Methoden, wie beispielsweise *Conformance Testing* oder *Model Checking,* verwendet werden.

Im Gegensatz zu herkömmlichen *Web-Crawlern* und *Link-Checkern*, welche rekursiv alle erreichbaren Seiten einer Web-Applikation abrufen und die statische Seitenstruktur einer Website erfassen, bildet der lernbasierte Webtest-Ansatz das dynamische Verhalten einer Web-Applikation ab. Insbesondere werden in den gelernten Modellen auch interne Zustände der Applikationen abgebildet. Erste Erfahrungen mit realen Systemen, dargestellt in der Veröffentlichung „Hybrid Test of Web Applications with Webtest" [111] zeigen, dass sich der Ansatz auch auf größere Projekte als das „Google"-Beispiel anwenden lässt.

Kapitel 6

Ausblick

6.1 Zusammenfassung

Wesentlicher Bestandteil dieser Dissertation ist die LearnLib, eine Plattform zur Extrapolation von Prozessmodellen aus Black-Box-Systemen mittels Automatenlernverfahren. Das Ziel der LearnLib ist es zum einen, eine Experimentierplattform anzubieten, mit deren Hilfe sich Automatenlernverfahren und deren Optimierungen auf experimentellem Wege statistisch analysieren lassen. Zum anderen will die LearnLib, Anwendungen des Automatenlernens von der Theorie in die Praxis übertragen. Für die Praxis stellt der Learnaufwand klassischer, nicht optimierter Lernverfahren ein Ausschlusskriterium dar. Real existierende Systeme lassen sich nur effizient extrapolieren, wenn mit Hilfe von Expertenwissen über das System die Menge der Anfragen an das System reduziert werden kann. Die LearnLib stellt hierzu neben Optimierungen der zu Grunde liegenden Lernalgorithmen eine Reihe von optimierenden Filtern bereit, die ihre Wirksamkeit bereits in vielen Anwendungsgebieten unter Beweis gestellt haben.

6.2 Ausblick

Die LearnLib stellt bereits eine optimierte, praxistaugliche Plattform für Anwendungen des Automatenlernens dar. Es gibt aber noch ein enormes Entwicklungspotential für Optimierungen, dass sich in mehrere Teilbereiche aufteilten lässt:

- Filterung der Anfragen,
- Verwendung geeigneterer Modelle,
- gerichtete Suche nach Gegenbeispielen,
- Nutzung hochparalleler Hardware.

6.2.1 Filterung von Anfragen

Die LearnLib enthält bereits eine Reihe von optimierenden Filtern. Diese haben zum Ziel, aus jeder Anfrage so viele Informationen wie möglich abzuleiten, um die Gesamtzahl an Anfragen, die tatsächlich zu bearbeiten sind, so klein wie möglich zu halten.

Diese Optimierungsdimension kann durch erweiterte Verfahren zur Nutzung von Anwendungswissen ausgestaltet werden. Zum einen können formale Verifikationsmethoden (*Model Checking*) eingesetzt werden, um Expertenwissen zu erfassen und somit Anfragen des Lernalgorithmus a priori beantworten zu können. Erste Ergebnisse in dieser Richtung liefert der in Abschnitt 5.2 beschriebene Ansatz zum Erfassen von Geschäftsprozessen. Zum anderen kann das Konzept der Äquivalenzklassen auf Anfragen, welches bisher nur für Symmetrie, unabhängige Aktionen und Präfixabgeschlossenheit realisiert ist, verallgemeinert werden. Idealerweise sollten, alle praxisrelevanten Äquivalenzen erfassbar und zur Filterung von Anfragen nutzbar gemacht werden. Darüber hinaus scheint es einfach möglich zu sein, nicht nur zuvor bekanntes Expertenwissen zu verwenden, sondern hypothetisches Expertenwissen aus den gelernten Modellen abzuleiten und in weiteren Lernphasen wiederzuverwenden. Im Falle unabhängiger Aktionen könnte man sogar zunächst alle Paare von Aktionen als unabhängig definierten und, immer wenn ein Gegenbeispiel zu dieser Annahme auftritt, die Unabhängigkeitsrelation sukzessive anpassen. Der Lernalgorithmus ist somit in der Lage, auch das „Expertenwissen“ zu lernen.

6.2.2 Verwendung erweiterter Modelle

Optimierungen durch Verwendung erweiterter Modelle tragen der Erkenntnis Rechnung, dass die Wahl der Modellierung einen großen Einfluss auf die Anzahl der Anfragen hat. Wie in Abschnitt 5.6 dargestellt, führte der Übergang von einem DFA-Lernalgorithmus zu einem Mealy-Lernalgorithmus zu einer massiven Einsparung von Anfragen [129]. Reale Systeme lassen sich häufig gut durch Modelle beschreiben, bei denen die Aktionen mit komplexen Parametern versehen sind. Es könnte also vorteilhaft sein, einen Lernalgorithmus zu entwickeln, der direkt auf parametrischen Systemen arbeitet. Hierbei könnte man Automatenlernverfahren mit Lernalgorithmen, für geometrische Konzepte z.B. [30, 31], kombinieren. Die geometrischen Konzepte dienen bei diesem Ansatz dazu, Äquivalenzklassen auf den Parametern der Aktionen zu erfassen und zu lernen.

6.2.3 Gerichtete Suche nach Gegenbeispielen

Das größte Problem aktiver Lernverfahren sind die *Equivalence Queries*. Bisher werden *Conformance Test* -Methoden eingesetzt, um die gelernte Hypothese mit dem realen System auf Äquivalenz zu prüfen. Diese Vorgehensweise hat sich als ungeeignet herausgestellt, da die zur Bildung der Hypothese gestellten Anfragen,

wie in der Veröffentlichung „On the Correspondence Between Conformance Testing and Regular Inference" [16] beschrieben, selbst einen Äquivalenztest darstellen und Abweichungen nur durch bisher unentdeckte System-Zustände aufgedeckt können. Hinzu kommt, dass die verwendeten Methoden systematisch die Äquivalenz von Hypothese- und System-Zuständen prüfen und so Inäquivalenzen bzw. Gegenbeispiele ausschließen. Für Automatenlernverfahren ist es aber wünschenswert, möglichst schnell irgendein Gegenbeispiel zu finden. Jedes Gegenbeispiel zeigt auf, dass das bereits eingeholte Wissen unvollständig ist und ermöglicht es erst, diese Wissenslücken zu schließen. Darüber hinaus ist es nicht notwendig, bereits als inäquivalent gekennzeichnete Zustände nocheinmal auf Inäquivalenz zu überprüfen.

Um möglichst schnell Gegenbeispiele zu finden, bieten sich zum einen heuristische Suchverfahren [53] an. Zum anderen lassen sich aus dem gesammelten Wissen des Lerners Heuristiken ableiten, mit denen sich die Suche nach Gegenbeispielen steuern lässt.

6.2.4 Nutzung hochparalleler Hardware

Die zunehmende Verbreitung von Rechnerarchitekturen, die mehrere Programmflüsse gleichzeitig unterstützen, lässt sich auszunutzen, um den Lernprozess zu beschleunigen. Insbesondere Multicoresysteme, die pro Kern mehrere Threads bearbeiten können, bieten sich an, um beispielsweise mehrere *Membership Queries* des Lernsystems gleichzeitig zu bearbeiten. Hierbei haben *Membership Queries* die vorteilhafte Eigenschaft, per Definition voneinander unabhängig zu sein. Die experimentellen Resultate des Webtest-Ansatzes aus Abschnitt 5.7, vorgestellt in dem, derzeit im Reviewprozess befindlichen *invited Jounal-Paper* „Dynamic Testing via Automata Learning" *(Journal)* [112], sind bereits auf einem Multicoresystem entstanden. Durch Einsatz einer Sun UltraSPARC T1 konnte durch Ausnutzung von 32 parallelen Threads die Gesamtausführungszeit des Lernprozesses drastisch gesenkt werden. Die derzeitige Implementierung der LearnLib erlaubt es aber nur abwechselnd,

- gemachte Beobachtungen auf Vollständigkeit und Konsistenz zu prüfen und
- Mengen von *Membership Queries* parallel auszuführen.

Um eine noch bessere Auslastung der zugrundeliegenden Hardwareplattform zu erreichen, ist es wünschenswert das beide Berechnungen gleichzeitig ausgeführt werden können.

Es ist zu erwarten, dass durch den massiven Einsatz von Parallelität (im Zusammenspiel mit den anderen Dimensionen des Optimierens) die Anwendbarkeit von Verfahren zum Automatenlernen so deutlich erweiterbar ist, dass reelle Produktivsysteme anvisiert werden können.

Literaturverzeichnis

[1] Common object request broker architecture (corba/iiop). http://www.omg.org/spec/CORBA.

[2] Pieter W. Adriaans and Marco Vervoort. The EMILE 4.1 grammar induction toolbox. In *Proc. of the* 6^{th} *Int. Colloquium on Grammatical Inference (ICGI '02)*, pages 293–295, London, UK, 2002. Springer-Verlag.

[3] Mohsen Ahyaie, Ahmed Alami, Said Chihani, Christian Frost, Jochen Gerlach, Falk Howar, David Karla, Christian May, Svetla Nikolova, and Marc Peschke. PG LiVe: Lernen impliziter verhaltensmodelle. Projektgruppenendbericht, Universität Dortmund, Fakultät für Informatik, Lehrstuhl für Programmiersysteme, 2007.

[4] Marc Aiguier, Karim Berkani, and Pascale Le Gall. Feature specification and static analysis for interaction resolution. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *Proc. of* 14^{th} *Int. Symposium on Formal Methods (FM '06), Hamilton, Canada, August 21-27,2006*, volume 4085 of *Lecture Notes in Computer Science*, pages 364–379. Springer, 2006.

[5] Adem Altinata, Jorge Carrillo de Albornoz, Marguerite Djomkouo Simo, Edward Fondis, Alberto Garcia, Christian Holle, Alexander Kout, German Martinez, Maik Merten, Aboubakr Mkhdramine, Dominik Opolony, and Murat Zabun. PG TTCP: Test and testing controll plattform. Projektgruppenendbericht, Universität Dortmund, Fakultät für Informatik, Lehrstuhl für Programmiersysteme, 2006.

[6] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. Inference of message sequence charts. volume 29, pages 623–633, Piscataway, NJ, USA, 2003. IEEE Press.

[7] Juan-Carlos Amengual, Alberto Sanchis, Enrique Vidal, and José-Miguel Benedí. Language simplification through error-correcting and grammatical inference techniques. *Mach. Learn.*, 44(1-2):143–159, 2001.

[8] Dana Angluin. On the complexity of minimum inference of regular sets. *Information and Control*, 39(3):337–350, 1978.

[9] Dana Angluin. A note on the number of queries needed to identify regular languages. *Information and Control*, 51(1):76–87, 1981.

[10] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 2(75):87–106, 1987.

[11] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. The Java series ... from the source. Addison-Wesley, 3 edition, 2000.

[12] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, May 2008.

[13] Seán Baker. Web services and corba. In *On the Move to Meaningful Internet Systems, Proc. of Confederated Int. Conf. DOA, CoopIS and ODBASE 2002*, volume 2519 of *Lecture Notes in Computer Science*, pages 618–632, London, UK, 2008. Springer-Verlag.

[14] José L. Balcázar, Josep Dìaz, and Ricard Gavaldà. Algorithms for learning finite automata from queries: A unified view. In *Advances in Algorithms, Languages, and Complexity*, pages 53–72, 1997.

[15] José L. Balcázar, Josep Dìaz, Ricard Gavaldà, and Osamu Watanabe. The query complexity of learning DFA. *New Generation Comput.*, 12(4):337–358, 1994.

[16] Therese Berg, Olga Grinchtein, Bengt Jonsson, Martin Leucker, Harald Raffelt, and Bernhard Steffen. On the correspondence between conformance testing and regular inference. In Maura Cerioli, editor, *Proc. of* 8^{th} *Int. Conf. on Fundamental Approaches to Software Engineering (FASE '05)*, volume 3442 of *Lecture Notes in Computer Science*, pages 175–189. Springer Verlag, April 4-8 2005.

[17] Therese Berg, Bengt Jonsson, Martin Leucker, and Mayank Saksena. Insights to Angluin's learning. Technical Report 2003-039, Department of Information Technology, Uppsala University, August 2003.

[18] Therese Berg, Bengt Jonsson, and Harald Raffelt. Regular inference for state machines with parameters. In Luciano Baresi and Reiko Heckel, editors, *Proc. of* 9^{th} *Int. Conf. on Fundamental Approaches to Software Engineering (FASE '06)*, volume 3922 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2006.

[19] Therese Berg, Bengt Jonsson, and Harald Raffelt. Regular inference for state machines using domains with equality tests. In José Luiz Fiadeiro and Paola Inverardi, editors, *Proc. of* 11^{th} *Int. Conf. on Fundamental Approaches to Software Engineering (FASE '08), Held as Part of the Joint European Conferences on Theory and Practice of Software, (ETAPS '08), Budapest, Hungary, March 29-April 6, 2008*, volume 4961 of *Lecture Notes in Computer Science*, pages 317–331. Springer, 2008.

[20] Therese Berg and Harald Raffelt. *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, chapter Model Checking, pages 557–603. Springer, 2005.

[21] Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, and Martin Leucker. Replaying play in and play out: Synthesis of design models from scenarios by learning. In Orna Grumberg and Michael Huth, editors, *Proc. of* 13^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '07), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS '07) Braga, Portugal*, volume 4424 of *Lecture Notes in Computer Science*, pages 435–450. Springer, 2007.

[22] Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, and Martin Leucker. Smyle: A tool for synthesizing distributed models from scenarios by learning. In Franck van Breugel and Marsha Chechik, editors, *Proc. of* 19^{th} *Int. Conf. on Concurrency Theory (CONCUR '08), Toronto, Canada, August 19-22, 2008*, volume 5201 of *Lecture Notes in Computer Science*, pages 162–166. Springer, 2008.

[23] Angela Bonifati, Stefano Ceri, and Stefano Paraboschi. Active rules for xml: A new paradigm for e-services. *The VLDB Journal*, 10(1):39–47, 2001.

[24] Angela Bonifati, Stefano Ceri, and Stefano Paraboschi. Pushing reactive services to xml repositories using active rules. In *Proc. of the* 10^{th} *Int. Conf. on World Wide Web (WWW '01)*, pages 633–641, New York, NY, USA, 2001. ACM.

[25] Y. Bontemps, P. Heymans, and P.-Y. Schobbens. From live sequence charts to state machines and back: a guided tour. *Software Engineering, IEEE Transactions on*, 31(12):999–1014, Dec. 2005.

[26] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *Journal of the Association of Computing Machinery*, 30(2):323–342, 1983.

[27] Alvis Brazma, Inge Jonassen, Jaak Vilo, and Esko Ukkonen. Pattern discovery in biosequences. In *Proc. of the* 4^{th} *Int. Colloquium on Grammatical Inference (ICGI '98)*, pages 257–270, London, UK, 1998. Springer-Verlag.

[28] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner. *Model-Based Testing of Reactive Systems:*, volume 3472 of *Lecture Notes in Computer Science*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[29] Yuriy Brun and Michael D. Ernst. Finding latent code errors via machine learning over program executions. In *Proc. of the* 26^{th} *Int. Conf. on Software Engineering (ICSE '04)*, pages 480–490, Edinburgh, Scotland, May 2004.

[30] Nader H. Bshouty and Lynn Burroughs. On the proper learning of axis-parallel concepts. *J. Mach. Learn. Res.*, 4:157–176, 2003.

[31] Nader H. Bshouty, Paul W. Goldberg, Sally A. Goldman, and H. David Mathias. Exact learning of discretized geometric concepts. *SIAM J. Comput.*, 28(2):674–699, 1999.

[32] Horst Bunke and Alberto Sanfeliu, editors. *Syntactic and Structural Pattern Recognition: Theory and Applications*. World Scientific,, 1990.

[33] Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Computer Networks*, 41(1):115–141, 2003.

[34] Muffy Calder and Alice Miller. Using spin for feature interaction analysis—a case study. In *Proc. of the* 8^{th} *Int. SPIN Workshop on Model Checking of Software (SPIN '01)*, pages 143–162, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[35] Rafael C. Carrasco and José Oncina. Learning stochastic regular grammars by means of a state merging method. In *Proc. of the* 2^{nd} *Int. Colloquium on Grammatical Inference and Applications (ICGI'94)*, volume 862 of *Lecture Notes in Computer Science*, pages 139–152, London, UK, 1994. Springer-Verlag.

[36] Stefano Ceri and Raghu Ramakrishnan. Rules in database systems. *ACM Comput. Surv.*, 28(1):109–111, 1996.

[37] Tsun S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, May 1978.

[38] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (wsdl) 1.1. `http://www.w3.org/TR/wsdl`, March 2001.

[39] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, January 2000.

[40] Axel Cleeremans, David Servan-Schreiber, and James L. McClelland. Finite state automata and simple recurrent networks. *Neural Comput.*, 1(3):372–381, 1989.

[41] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Pasareanu. Learning assumptions for compositional verification. In *Proc. of the* 9^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 'Damas03)*, volume 2619 of *Lecture Notes in Computer Science*, pages 331–346. Springer Berlin / Heidelberg, 2003.

[42] Jonathan E. Cook, Zhidian Du, Chongbing Liu, and Alexander L. Wolf. Discovering models of behavior for concurrent systems. Technical report, New Mexico State University, Deppartment of Computer Science, August 2002. NMSU-CS-2002-010.

[43] Jonathan E. Cook, Zhidian Du, Chongbing Liu, and Alexander L. Wolf. Discovering models of behavior for concurrent workflows. *Computation Industry*, 53(3):297–319, 2004.

[44] Jonathan E. Cook and Alexander L. Wolf. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 7(3):215–249, 1998.

[45] Christophe Damas, Bernard Lambeau, and Axel van Lamsweerde. Scenarios, goals, and state machines: a win-win partnership for model synthesis. In *Proc. of the* 14^{th} *ACM SIGSOFT Int. Symposium on Foundations of Software Engineering (SIGSOFT '06/FSE-14)*, pages 197–207, New York, NY, USA, 2006. ACM.

[46] Werner Damm and David Harel. Lscs: Breathing life into message sequence charts. *Form. Methods Syst. Des.*, 19(1):45–80, 2001.

[47] R. Darimont, E. Delor, P. Massonet, and A. van Lamsweerde. Grail/kaos: an environment for goal-driven requirements engineering. In *Proc. of the* 19^{th} *Int. Conf. on Software Engineering (ICSE '97)*, pages 612–613, New York, NY, USA, 1997. ACM.

[48] Sreerupa Das and Michael Mozer. A unified gradient-descent/clustering architecture for finite state machine induction. In Jack D. Cowan, Gerald Tesauro, and Joshua Alspector, editors, *Proc. of* 7^{th} *Annual Conf. on Advances in Neural Information Processing Systems 6 (NIPS '93), Denver, Colorado, USA, 1993*, pages 19–26. Morgan Kaufmann, 1994.

[49] U. Dayal. Active database management systems. In *Proc. of* 3^{rd} *Int. Conf. on Data and Knowledge bases - Improving usability and Responsiveness*, 1988.

[50] Colin de la Higuera. A bibliographical study of grammatical inference. *Pattern Recognition*, 38:1332–1348, September 2005.

[51] Klaus R. Dittrich, Stella Gatziu, and Andreas Geppert. The active database management system manifesto: A rulebase of ADBMS features. *SIGMOD Rec.*, 25(3):40–49, 1996.

[52] Pierre Dupont. Incremental regular inference. In *Proc. of* 3^{rd} *Int. Colloquium on Grammatical Inference: Learning Syntax from Sentences*, volume 1147 of *Lecture Notes in Computer Science*, pages 222–237. Springer, 1996.

[53] Stefan Edelkamp, Shahid Jabbar, and Alberto Lluch-Lafuente. Heuristic search for the analysis of graph transition systems. In Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg, editors, *Proc. of* 3^{rd} *Int. Conf. on Graph Transformations (ICGT '06), Natal, Rio Grande do Norte, Brazil, September 17-23, 2006*, volume 4178 of *Lecture Notes in Computer Science*, pages 414–429. Springer, 2006.

[54] Robert Elfwing, Ulf Paulsson, and Lars Lundberg. Performance of soap in web service environment compared to corba. In *Proc. of the* 9^{th} *Asia-Pacific Software Engineering Conf. (APSEC '02)*, page 84, Los Alamitos, CA, USA, 2002. IEEE Computer Society.

[55] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *Proc. of* 22^{nd} *Int. Conf. on Software Engineering (ICSE '00)*, pages 449–458, June 2000.

[56] J.A. Feldman and A. Biermann. On the synthesis of finite-state machines from samples of their behavior. *IEEE Transactions on Computers*, C-21:592–596, 1972.

[57] Susumu Fujiwara, Gregor von Bochmann, Ferhat Khendek, Mokhtar Amalou, and Abderrazak Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, 1991.

[58] Hubert Garavel. Open/caesar: An open software architecture for verification, simulation, and testing. In Bernhard Steffen, editor, *Proc. of the* 1^{st} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '98)*, volume 1384 of *Lecture Notes in Computer Science*, pages 68–84. Springer Verlag, 1998.

[59] Angelo Gargantini and Constance Heitmeyer. Using model checking to generate tests from requirements specifications. *SIGSOFT Softw. Eng. Notes*, 24(6):146–162, 1999.

[60] Angelo Gargantini, Elvinia Riccobene, and Salvatore Rinzivillo. Using spin to generate testsfrom asm specifications. In Egon Börger, Angelo Gargantini, and Elvinia Riccobene, editors, *Proc. of* 10^{th} *Int. Workshop on Abstract State Machines, Advances in Theory and Practice (ASM '03), Taormina, Italy, March 3-7, 2003*, volume 2589 of *Lecture Notes in Computer Science*, pages 263–277. Springer, 2003.

[61] Aniruddha Gokhale, Bharat Kumar, and Arnaud Sahuguet. Reinventing the wheel? corba vs. web services. In *Proc. of the* 11^{th} *Int. World Wide Web Conference,Sheraton Waikiki Hotel, Honolulu, Hawaii, USA ,7-11 May 2002*, 2002.

[62] E. Mark Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302–320, 1978.

[63] Jens Grabowski, Dieter Hogrefe, György Réthy, Ina Schieferdecker, Anthony Wiles, and Colin Willcock. An introduction to the testing and test control notation (ttcn-3). *Comput. Netw.*, 42(3):375–403, 2003.

[64] Jens Grabowski, Anthony Wiles, Colin Willcock, and Dieter Hogrefe. On the design of the new testing language ttcn-3. In *Proc. of the IFIP TC6/WG6.1* 13^{th} *Int. Conf. on Testing Communicating Systems (TestCom '00)*, pages 161–176, Deventer, The Netherlands, The Netherlands, 2000. Kluwer, B.V.

[65] Alex Groce, Doron Peled, and Mihalis Yannakakis. Adaptive model checking. In J.-P. Katoen and P. Stevens, editors, *Proc. of the* 8^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 357–370. Springer Verlag, 2002.

[66] Marc J. Hadley. Web application description language (wadl). Technical report, Sun Microsystems Inc., 2006.

[67] Andreas Hagerer, Hardi Hungar, Oliver Niese, and Bernhard Steffen. Model generation by moderated regular extrapolation. In H. Weber R. Kutsche, editor, *Proc. of the* 5^{th} *Int. Conf. on Fundamental Approaches to Software Engineering (FASE'02)*, volume 2306 of *Lecture Notes in Computer Science*, pages 80–95, Heidelberg, Germany, April 2002. Springer-Verlag.

[68] Andreas Hagerer, Tiziana Margaria, Oliver Niese, Bernhard Steffen, Georg Brune, and Hans-Dieter Ide. Efficient regression testing of cti-systems: Testing a complex call-center solution. *Annual Review of Communication, Int. Engineering Consortium (IEC), Chicago (USA)*, 55:1033–1040, 2001.

[69] Michael G. Hinchey, James L. Rash, and Christopher A. Rouff. A formal approach to requirements-based programming. In *Proc. of the* 12^{th} *IEEE Int. Conf. and Workshops on Engineering of Computer-Based Systems (ECBS '05)*, pages 339–345, Washington, DC, USA, 2005. IEEE Computer Society.

[70] Falk Howar. Inferenz parametrisierter moore-automaten. Diplomarbeit, Technische Universität Dortmund, Fakultät für Informatik, Lehrstuhl für Programmiersysteme, to be published in 2009.

[71] Hardi Hungar, Tiziana Margaria, and Bernhard Steffen. Test-based model generation for legacy systems. In *Proc. of 2003 International Test Conference (ITC 2003)*, pages 971–980, Charlotte, NC, September 2003. IEEE Computer Society.

[72] Hardi Hungar, Oliver Niese, and Bernhard Steffen. Domain-specific optimization in automata learning. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Proc. of the* 15^{th} *Int. Conf. on Computer Aided Verification (CAV '03)*, volume 2725 of *Lecture Notes in Computer Science*, pages 315–327. Springer Verlag, July 2003.

[73] Martina Hörmann, Tiziana Margaria, Thomas Mender, Ralf Nagel, Michael Schuster, Bernhard Steffen, and Hong Trinh. The jabc approach to collaborative development of embedded applications. In *Proc. of Int. Workshop on Challenges in Collaborative Engineering - State of the Art and Future Challenges on Collaborative Design (CCE '06)*, April 2006. (Industry Day), Prag (CZ).

[74] ISO/IEC. EBNF syntaxt specification standard, EBNF: ISO/IEC 14977, 1996.

[75] Martin Karusseit and Tiziana Margaria. Feature-based modelling of a complex, online-reconfigurable decision support service. *Electr. Notes Theor. Comput. Sci.*, 157(2):101–118, 2006.

[76] Michael J. Kearns and Umesh V. Vazirani. *An Introduction to Computational Learning Theory*, chapter Learning Finite Automata by Experimentation, pages 155–184. MIT Press, 1994.

[77] Gerhard Knolmayer, Rainer Endl, and Marcel Pfahrer. Modeling processes and workflows by business rules. In Wil M. P. van der Aalst, Jörg Desel, and Andreas Oberweis, editors, *Business Process Management, Models, Techniques, and Empirical Studies*, volume 1806 of *Lecture Notes in Computer Science*, pages 16–29. Springer, 2000.

[78] Donald Kossmann and Frank Leymann. Web services. *Informatik-Spektrum*, 27(2):117–128, April 2004.

[79] Philippe Kruchten. *The Rational Unified Process: An Introduction*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[80] Ingolf Krüger, Radu Grosu, Peter Scholz, and Manfred Broy. From MSCs to statecharts. In Franz J. Rammig, editor, *Distributed and Parallel Embedded Systems, IFIP WG10.3/WG10.5 International Workshop on Distributed and Parallel Embedded Systems (DIPES '98), October 5-6. 1998, Schloß Eringerfeld, Germany*, volume 155 of *IFIP Conference Proceedings*, pages 61–72. Kluwer, 1998.

[81] P. Lang, S. Rausch-schott, and W. Retschitzegger. Workflow management based on objects, rules, and roles. *IEEE Bulletin of the Technical Committee on Data Engineering*, 18:11–18, 1995.

[82] David Lee and Mihalis Yannakakis. Principles and methods of testing finite state machines — a survey. *Proc. IEEE*, 84(8):1090–1126, 1996.

[83] Peter Liggesmeyer. *Software-Qualität. Testen, Analysieren und Verifizieren von Software.* Spektrum Akademischer Verlag, August 2002.

[84] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. Inferring state-based behavior models. In *Proc. of the 2006 Int. Workshop on Dynamic Systems Analysis (WODA '06)*, pages 25–32, New York, NY, USA, 2006. ACM.

[85] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. Automatic generation of software behavioral models. In *Proceedings of the* 30^{th} *International Conference on Software engineering (ICSE '08)*, pages 501–510, New York, NY, USA, 2008. ACM.

[86] N. A. M. Maiden. CREWS-SAVRE: Scenarios for acquiring and validating requirements. *Automated Software Engg.*, 5(4):419–446, 1998.

[87] Oded Maler and Amir Pnueli. On the learnability of infinitary regular sets. In *Proc. of the* 4^{th} *annual workshop on Computational learning theory (COLT '91)*, pages 128–138, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[88] Tiziana Margaria, Mike G. Hinchey, Harald Raffelt, J.L. Rash, C. A. Rouff, and Bernhard Steffen. Completing and adapting models of biological processes. In *Proc. of IFIP Conf. on Biologically Inspired Cooperative Computing (BiCC '06), Santiago (Chile)*. Springer Verlag, Aug. 2006.

[89] Tiziana Margaria, Oliver Niese, Bernhard Steffen, and Andrei Erochok. System level testing of virtual switch (re-)configuration over ip. In *Proc. of the IEEE European Test Workshop (ETW'02)*, pages 67–74. IEEE Computer Society Press, 2002. ETW2002.

[90] Tiziana Margaria, Harald Raffelt, and Bernhard Steffen. Analyzing second-order effects between optimizations for system-level test-based model generation. In *Proc. of IEEE International Test Conference (ITC'05)*. IEEE Computer Society, November 2005.

[91] Tiziana Margaria, Harald Raffelt, and Bernhard Steffen. Knowledge-based relevance filtering for efficient system-level test-based model generation. *Innovations in Systems and Software Engineering*, 1(2):147–156, July 2005.

[92] Tiziana Margaria, Harald Raffelt, Bernhard Steffen, and Martin Leucker. The learnlib in fmics-jeti. In *Proc. of* 12^{th} *Int. Conf. on Engineering of Complex Computer Systems (ICECCS '07), 10-14 July 2007, Auckland, New Zealand*, pages 340–352. IEEE Computer Society, 2007.

[93] Tiziana Margaria and Bernhard Steffen. Lightweight coarse-grained coordination: A scalable system-level approach. *Int. J. Softw. Tools Technol. Transf.*, 5(2):107–123, 2004.

[94] Tiziana Margaria, Bernhard Steffen, and Manfred Reitenspieß. Service-oriented design: The jabc approach. In *Proc. of Service Oriented Computing ,(SOC 2005)*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.

[95] Leonardo Mariani and Mauro Pezzè. A technique for verifying component-based software. In *Proc. of Int. Workshop on Test and Analysis of Component Based Systems (TACoS '04)*, pages 17–30, March 2004.

[96] Leonardo Mariani and Mauro Pezzè. Dynamic detection of cots component incompatibility. *IEEE Softw.*, 24(5):76–85, 2007.

[97] Maik Merten. Automatic business process reengineering. Diplomarbeit, Universität Dortmund, Fachbereich Informatik, Lehrstuhl für Programmiersysteme, Juni 2007.

[98] Mehryar Mohri. Finite-state transducers in language and speech processing. *Comput. Linguist.*, 23(2):269–311, 1997.

[99] Edward F. Moore. Gedanken-experiments on sequential machines. *Annals of Mathematical Studies*, 34:129–153, 1956.

[100] Erkki Mäkinen and Tarja Systä. Mas — an interactive synthesizer to support behavioral modelling in uml. In *Proc. of the* 23^{rd} *Int. Conf. on Software Engineering (ICSE '01)*, pages 15–24, Washington, DC, USA, 2001. IEEE Computer Society.

[101] Markus Müller-Olm, David A. Schmidt, and Benhard Steffen. Model-checking: A tutorial introduction. In G. File A. Cortesi, editor, *Proc. of Static Analysis Symposium (SAS'99), Venice, Italy*, volume 1694 of *Lecture Notes in Computer Science (LNCS)*, pages 330–354, Heidelberg, Germany, September 1999. Springer-Verlag.

[102] Oliver Niese. *An Integrated Approach to Testing Complex Systems*. PhD thesis, University of Dortmund, Germany, 2003.

[103] Jeremy W. Nimmer and Michael D. Ernst. Automatic generation of program specifications. In *Proc. of the 2002 Int. Symposium on Software Testing and Analysis (ISSTA '02)*, pages 229–239, Rome, Italy, July 22–24, 2002.

[104] Bashar Nuseibeh and Steve Easterbrook. Requirements engineering: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering (ICSE '00)*, pages 35–46, New York, NY, USA, 2000. ACM.

[105] José Oncina and Pedro García. Identifying regular languages in polynomial time. In *Advances in Structural and Syntactic Pattern Recognition, volume 5 of Series in Machine Perception and Artificial Intelligence*, pages 99–108. World Scientific, 1992.

[106] Rajesh Parekh and Vasant Honavar. *Handbook of Natural Language Processing*, chapter Grammar Inference, Automata Induction, and Language Acquisition, pages 727–764. Marcel Dekker, 1998.

[107] Norman W. Paton, editor. *Active Rules in Database Systems*. Springer, New York, 1999.

[108] Doron Peled, Moshe Y. Vardi, and Mihalis Yannakakis. Black box checking. In Jianping Wu, Samuel T. Chanson, and Qiang Gao, editors, *Proc. of the Joint Int. Conference on Formal Description Techniques for Distributed System and Communication/Protocols and Protocol Specification, Testing and Verification FORTE/PSTV '99:*, pages 225–240. Kluwer Academic Publishers, 1999.

[109] Leonard Pitt and Manfred K. Warmuth. The minimum consistent dfa problem cannot be approximated within any polynomial. *J. ACM*, 40(1):95–142, 1993.

[110] Colin Potts, Kenji Takahashi, and Annie I. Antón. Inquiry-based requirements analysis. *IEEE Softw.*, 11(2):21–32, 1994.

[111] Harald Raffelt, Tiziana Margaria, Bernhard Steffen, and Maik Merten. Hybrid test of web applications with webtest. In *Proc. of the 2008 Workshop on Testing, analysis, and verification of web services and applications (TAV-WEB '08)*, pages 1–7, New York, NY, USA, 2008. ACM.

[112] Harald Raffelt, Maik Merten, Bernhard Steffen, and Tiziana Margaria. Dynamic testing via automata learning. *Int. Journal on Software Tools for Technology Transfer (STTT)*, invited Paper currently under review.

[113] Harald Raffelt, Bernhard Steffen, and Therese Berg. LearnLib: A library for automata learning and experimentation. In *Proc. of the* 10^{th} *Int. Workshop on Formal Methods for Industrial Critical Systems (FMICS '05)*, pages 62–71, Lisbon, Portugal, 2005. ACM Press. Lisbon, Portugal.

[114] Harald Raffelt, Bernhard Steffen, and Tiziana Margaria. Dynamic testing via automata learning. In *Proc. of the Haifa Verification Conference 2007 (HVC '07)*, volume 4899 of *Lecture Notes in Computer Science*, pages 136–152. Springer-Verlag Berlin Heidelberg, 2008.

[115] Sanjai Rayadurgam and Mats Per Erik Heimdahl. Coverage based test-case generation using model checkers. In *Proc. of* 8^{th} *IEEE Int. Conf. on Engineering of Computer-Based Systems (ECBS '01), 17-20 April 2001, Washington, DC, USA*, pages 83–93. IEEE Computer Society, 2001.

[116] Sanjai Rayadurgam and Mats Per Erik Heimdahl. Test-sequence generation from formal requirement models. In *Proc. of the* 6^{th} *IEEE Int. Symposium on High-Assurance Systems Engineering (HASE '01), Special Topic: Impact of Networking, 24-26 October 2001, Albuquerque, NM, USA*, pages 23–31. IEEE Computer Society, 2001.

[117] Stephan Reiff-Marganiec and Kenneth J. Turner. Feature interaction in policies. *Comput. Netw.*, 45(5):569–584, 2004.

[118] Steven P. Reiss and Manos Renieris. Encoding program executions. In *Proc. of the* 23^{rd} *Int. Conf. on Software Engineering (ICSE '01)*, pages 221–230, Washington, DC, USA, 2001. IEEE Computer Society.

[119] Ronald L. Rivest and Robert E. Schapire. Inference of finite automata using homing sequences. In *Proc. of the* 21^{st} *Annual ACM Symposium on Theory of Computing (STOC '89)*, pages 411–420. MIT Laboratory for Computer Science, ACM Press, May 1989.

[120] Ronald L. Rivest and Robert E. Schapire. Inference of finite automata using homing sequences. *Inf. Comput.*, 103(2):299–347, 1993.

[121] Ronald L. Rivest and Robert E. Schapire. Diversity-based inference of finite automata. *Journal of the Association for Computing Machinery*, 41(3):555–589, 1994.

[122] A. W. Roscoe, C. A. R. Hoare, and Richard Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[123] Yasubumi Sakakibara. Learning context-free grammars from structural data in polynomial time. *Theor. Comput. Sci.*, 76(2-3):223–242, 1990.

[124] Yasubumi Sakakibara and Hidenori Muramatsu. Learning context-free grammars from partially structured examples. In Arlindo L. Oliveira, editor, *Grammatical Inference: Algorithms and Applications, 5^{th} Int. Colloquium, (ICGI 2000), Lisbon, Portugal, September 11-13, 2000*, volume 1891 of *Lecture Notes in Computer Science*, pages 229–240. Springer, 2000.

[125] Bikram Sengupta and Rance Cleaveland. Triggered message sequence charts. *SIGSOFT Softw. Eng. Notes*, 27(6):167–176, 2002.

[126] Yinan N. Shen, Fabrizio Lombardi, and Anton T. Dahbura. Protocol conformance testing using multiple uio sequences. In *Proc. of the 9^{th} Int. Symposium on Protocol Specification, Testing and Verification*, pages 131–143. North-Holland, 1990.

[127] Bernhard Steffen and Tiziana Margaria. METAFrame in practice: Design of intelligent network services. In *Correct System Design, Recent Insight and Advances*, Lecture Notes in Computer Science, pages 390–415, London, UK, 1999. Springer-Verlag.

[128] Bernhard Steffen, Tiziana Margaria, Ralf Nagel, Sven Jörges, and Christian Kubczak. Model-driven development with the jabc. In Eyal Bin, Avi Ziv, and Shmuel Ur, editors, *Hardware and Software, Verification and Testing, Second International Haifa Verification Conference, HVC 2006, Haifa, Israel, October 23-26, 2006. Revised Selected Papers*, volume 4383 of *Lecture Notes in Computer Science*, pages 92–108. Springer Verlag, 2006.

[129] Bernhard Steffen, Tiziana Margaria, Harald Raffelt, and Oliver Niese. Efficient test-based model generation of legacy systems. In *Proc. of the 9^{th} IEEE Int. Workshop on High Level Design Validation and Test (HLDVT '04)*, pages 95–100, Sonoma (CA), USA, November 2004. IEEE Computer Society Press.

[130] Andreas Stolcke and Stephen M. Omohundro. Inducing probabilistic grammars by bayesian model merging. In *Proc. of the 2^{nd} Int. Colloquium on Grammatical Inference and Applications (ICGI'94)*, volume 862 of *Lecture Notes in Computer Science*, pages 106–118, London, UK, 1994. Springer-Verlag.

[131] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society. Second Series*, 42:230–265, 1936.

[132] Sebastian Uchitel, Jeff Kramer, and Jeff Magee. Synthesis of behavioral models from scenarios. *IEEE Trans. Softw. Eng.*, 29(2):99–115, 2003.

[133] S.T. Vuong, W.W.L. Chan, and M.R. Ito. The UIOv-method for protocol test sequence generation. In Jan de Meer, Lothar Machert, and Wolfgang Effelsberg, editors, *Proc. of 2nd Int. Workshop on Protocol Testing Systems (IWPTS'89)*, pages 161–175. North-Holland, 1990.

[134] Tao Xie and David Notkin. Mutually enhancing test generation and specification inference. In Alexandre Petrenk and Andreas Ulrich, editors, *Proc. of 3rd Int. Workshop on Formal Approaches to Testing of Software (FATES '03)*, volume 2931 of *Lecture Notes in Computer Science*, pages 60–69. Springer Verag, 2004.

[135] Takashi Yokomori. Learning two-tape automata from queries and counterexamples. In *Mathematical Systems Theory*, pages 259–270, 1996.

[136] Z. Zeng, R.M. Goodman, and P. Smyth. Learning finite state machines with self-clustering recurrent networks. *Neural Computation*, 5(6):976–990, 1993.

Anhang A

Definitionen

Für das Verständnis der Arbeit sind einige formale Grundlagen notwendig, die in diesem Anhang erläutert werden.

Definition A.1 *Ein deterministischer endlicher Automat (DFA:* Deterministic Finite Automaton*) ist ein Tupel* $\mathsf{M} = (S, s_0, \Sigma, \delta, F)$*, bestehend aus*

S	*einer nichtleeren endlichen Menge von Zuständen,*
$s_0 \in S$	*genau einem Startzustand,*
Σ	*einer endlichen Menge von Symbolen: dem Alphabet,*
$\delta : S \times \Sigma \to S$	*einer Transitionsfunktion und*
$F \subseteq S$	*einer Menge akzeptierender Zustände.*

Ausgehend vom Startzustand s_0 wechselt ein deterministischer endlicher Automat beim Verarbeiten eines Symbols (oder einer Aktion) $a \in \Sigma$ von einem Zustand $s \in S$ gemäß der Transitionsfunktion (oder Zustandsübergangsfunktion) σ in einen Nachfolgezustand $s' = \sigma(s, a)$.

Ein Wort (oder eine Sequenz) $w \in \Sigma^*$ heißt genau dann von einem DFA *akzeptiert*, wenn sich der DFA nach Verarbeitung des Wortes w ausgehend vom Startzustand in einem akzeptierenden Zustand befindet.

Im Folgenden beschreibt $s \xrightarrow{a} s'$, dass ein DFA beim Verarbeiten des Symbols $a \in \Sigma$ vom Zustand $s \in S$ in Zustand $s' \in S$ wechselt. Des Weiteren wird die Transitionsfunktion $\delta : S \times \Sigma \to S$ auf Worte über dem Alphabet Σ erweitert zu $\delta : S \times \Sigma^* \to S$. Für alle Zustände $s \in \Sigma$, Symbole $a \in \Sigma$ und Worte $w \in \Sigma^*$ gilt hierbei $\delta(s, \varepsilon) = s$ und $\delta(s, aw) = \delta(\delta(s, a), w))$. Darüber hinaus sei $\sigma : \Sigma^* \to S$ definiert als $\sigma(w) := \sigma(s_0, w)$. Für DFAs sein auch eine Ausgabefunktion $\gamma \colon \Sigma^* \to \{true, false\}$ definiert als $\gamma(w) := \sigma(w) \notin F$.

Die Sprache $\mathscr{L}(\mathsf{M})$, die ein deterministischer Automat erkennt, ist definiert als die Menge aller akzeptierenden Worte: $\mathscr{L}(\mathsf{M}) := \{ w \in \Sigma^* | \, \sigma(w) \in F \}$.

Definition A.2 *Ein Mealy-Automat ist ein Tupel* $\mathsf{M} = (S, s_0, \Sigma, \Gamma, \delta, \gamma)$*, bestehend aus*

S	*einer nichtleeren endlichen Menge von Zuständen,*
$s_0 \in S$	*genau einem Startzustand,*
Σ	*einer endlichen Menge von Eingabesymbolen (Eingabealphabet),*
Γ	*einer endlichen Menge von Ausgabesymbolen (Ausgabealphabet),*
$\delta : S \times \Sigma \to S$	*einer Transitionsfunktion und*
$\gamma : S \times \Sigma \to \Gamma$	*einer Ausgabefunktion.*

Mealy-Automaten arbeiten ähnlich wie deterministische endliche Automaten. Ausgehend vom Startzustand s_0 wechselt ein Mealy-Automat beim Verarbeiten eines Symbols $a \in \Sigma$ von einem Zustand $s \in S$ gemäß der Transitionsfunktion σ in einen Nachfolgezustand $s' = \sigma(s,a)$. In jedem Schritt wird aber genau ein Ausgabesymbol $x \in \Gamma$ gemäß der Ausgabefunktion γ erzeugt.

Im Folgenden beschreibt $s \xrightarrow{a/x} s'$, dass ein Mealy-Automat beim Verarbeiten des Symbols $a \in \Sigma$ vom Zustand $s \in S$ in Zustand $s' \in S$ wechselt und dabei das Ausgabesymbol x erzeugt. Die Transitionsfunktion $\delta : S \times \Sigma \to S$ wird analog zum DFA-Fall zu $\delta : S \times \Sigma^* \to S$ bzw. $\delta : \Sigma^* \to S$ auf Worte über dem Alphabet Σ erweitert. Darüberhinaus wird auch die Ausgabefunktion $\gamma : S \times \Sigma \to \Gamma$ so erweitert, dass sie auf Worten arbeitet $\gamma : S \times \Sigma^* \to \Gamma^*$. Für alle Zustände $s \in \Sigma$, Symbole $a \in \Sigma$ und Worte $w \in \Sigma^*$ gilt hierbei $\gamma(s,\varepsilon) = \varepsilon$ und $\gamma(s,aw) = \gamma(s,a)\gamma(\delta(s,a),w))$. Zusätzlich sein $\gamma : \Sigma^* \to \Gamma^*$ definiert als $\gamma(w) = \gamma(s_0,w)$.

Definition A.3 *Eine Menge von Worten $P \subseteq \Sigma^*$ heißt genau dann* präfixabgeschlossen*, wenn zu jedem Wort w aus P ein Präfix $p \in \Sigma^*$ von w existiert, der auch in P enthalten ist. Ein Präfix eines Wortes w ist ein Wort $p \in \Sigma^*$ zu dem ein $u \in \Sigma^*$ existiert, so das die Konkatenation von p und u dem Wort w entspricht.*

Definition A.4 *Eine Menge von Worten $S \subseteq \Sigma^*$ heißt genau dann* suffixabgeschlossen*, wenn zu jedem Wort w aus S ein Suffix $s \in \Sigma^*$ von w existiert, der auch in S enthalten ist. Ein Suffix eines Wortes w ist ein Wort $s \in \Sigma^*$ zu dem ein $u \in \Sigma^*$ existiert, so das die Konkatenation von u und s dem Wort w entspricht.*

Anhang B

Veröffentlichungen

Dieser kumulativen Dissertation liegen folgende Veröffentlichungen zu Grunde:

1. Analyzing Second-Order Effects Between Optimizations for System-Level Test-Based Model Generation [90]
   Automatenlernen lässt sich ohne weitere Optimierungen nur schwer in der Praxis anwenden. Die Wechselwirkungen dieser Optimierungen, vorgestellt in Abschnitt 4.3 werden in dieser Veröffentlichung analysiert.

2. Completing and Adapting Models of Biological Processes [88]
   In Abschnitt 5.1 wird ein Ansatz skizziert, der mit Hilfe von Automatenlernverfahren die Anforderungsanalyse im Softwareentwicklungsprozess unterstützt. Die Veröffentlichung stellt diesen Ansatz vor, und beschreibt seine Anwendung auf biologische Prozesse.

3. Dynamic Testing via Automata Learning [114]
   Diese Veröffentlichung beschreibt den Ansatz, Automatenlernverfahren zum Testen von Web-Applikationen einzusetzen.

4. Dynamic Testing via Automata Learning *(Journal)* [112]
   Dieses *invited Jounal-Paper* befindet sich derzeit im Reviewprozess. Es vereinigt die Publikation 3 und 6. Es stellt dass in Abschnitt 5.7 skizzierte Werkzeug zum Testen von Web-Applikationen detailliert vor.

5. Efficient Test-based Model Generation of Legacy Systems [129]
   Diese Veröffentlichung beschreibt ein Werkzeug zum Testen von komplexen Telefonie-Anwendungen (siehe auch Abschnitt 5.6) und vergleicht die DFA und Mealy-Version von Angluins Algorithmus anhand eines praxisnahen Beispiels.

6. Hybrid Test of Web Applications with Webtest [111]
   Diese Veröffentlichung beschreibt die Vereinigung von dynamischem Testen (Publikation 3) mit traditionellem Record und Replay Testen.

7. Knowledge-Based Relevance Filtering for Efficient System-Level Test-Based Model Generation *(Journal)* [91]
   Dieses *Jounal-Paper* basiert auf Publikation 1 und 5. Es beschreibt die Optimierungen der LearnLib mit ihren Wechselwirkungen bei der Anwendung im Bereich *Computer Telephony Integration*.

8. LearnLib: A Library for Automata Learning and Experimentation [113]
   Diese Veröffentlichung stellt die in Kapitel 4 zusammengefasste LearnLib, eine Plattform zur Entwicklung und Evaluation von Anwendungen des Automatenlernens vor.

9. Model-Based Testing of Reactive Systems: Model Checking [20]
   Ein Buchkapitel in dem automatenbasiertes *Model Checking*, Automatenlernverfahren, und die kombinierte Anwendung beider Verfahren zum Testen von reaktiven Systemen beschrieben wird.

10. On the Correspondence Between Conformance Testing and Regular Inference [16]
    Diese Veröffentlichung diskutiert die Beziehungen zwischen *Conformance Testing* und Automatenlernverfahren.

11. Regular Inference for State Machines using Domains with Equality Tests [19]
    Ein Ansatz zum Lernen von (potentiell) unendlichen Automaten wird in dieser Veröffentlichung vorgestellt. (Siehe auch Abschnitt 4.2.5)

12. Regular Inference for State Machines with Parameters [18]
    Diese Veröffentlichung beschreibt einen Algorithmus zu Lernen von endlichen Automaten, deren Aktionen mit Booleschen Parametern erweitert sind. (Siehe auch Abschnitt 4.2.5)

13. The LearnLib in FMICS-jETI [92]
    Diese Veröffentlichung beschreibt die Realisierung einer Anwendung zur lernunterstützten Synthese von *Message Passing Automata* auf Basis der LearnLib im Rahmen der FMICS-jETI Plattform. Diese Anwendung wird in Abschnitt 5.3 vorgestellt.