KONTINUIERLICHE QUALITÄTSKONTROLLE VON WEBANWENDUNGEN AUF BASIS MASCHINENGELERNTER MODELLE

Dissertation

zur Erlangung des Grades eines

D o k t o r s d e r I n g e n i e u r w i s s e n s c h a f t e n

der Technischen Universität Dortmund
an der Fakultät für Informatik

von

STEPHAN WINDMÜLLER

Dortmund

2014

Tag der mündlichen Prüfung: 21.07.2014
Dekan: Prof. Dr.-Ing. Gernot A. Fink

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

Danksagungen

Bedanken möchte ich mich vor allem bei Prof. Dr. Bernhard Steffen. Er hat eine Atmosphäre geschaffen, die mir eine freie Entfaltung ermöglichte. Zusammen mit dem Umstand, dass er für Probleme und Fragen stets ein offenes Ohr hatte, sorgte er dafür, dass diese Arbeit erfolgreich abgeschlossen werden konnte.

Weiterhin möchte ich mich bei meinem zweiten Gutachter, Prof. Dr. Jakob Rehof, bedanken, sowie bei Prof. Dr. Gabriele Kern-Isberner für die Übernahme des Vorsitzes der Prüfungskommission und bei Prof. Dr. Günter Rudolph als weiterem Mitglied der Kommission.

Daneben gilt mein Dank meiner Familie und meiner Frau, die mich während meiner gesamten Laufbahn in vielerlei Hinsicht unterstützt haben.

Besonderer Dank gilt meinem Bürokollegen Dr. Johannes Neubauer, dessen kreative Ader und Fachkompetenz dabei halfen, aus vermeintlichen Sackgassen neue Möglichkeiten zu schaffen. In schwierigen Phasen fand er stets die richtigen Worte, um neue Motivation zu schöpfen. Sowohl von der gemeinsamen Entwicklung des OCS als auch der Arbeit an wissenschaftlichen Ausarbeitungen habe ich sehr profitiert.

Ebenfalls möchte ich mich bei Malte Isberner bedanken, der nicht nur meine Ausarbeitung korrekturgelesen hat, sondern auch alle meine Fragen zur Verwendung der LearnLib geduldig beantwortete.

Insgesamt möchte ich all meinen Kollegen und Mitarbeitern am Lehrstuhl für die entspannte Arbeitsatmosphäre und die gute Zusammenarbeit danken.

Für
Rolf Max Hermann Kayser

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Bereits seit mehreren Jahren ist das Internet sowohl im beruflichen als auch privaten Umfeld nicht mehr wegzudenken. Mit der steigenden Verbreitung von mobilen Endgeräten wie Smartphones und Tablets verlagern sich jedoch nicht nur vormals analoge Tätigkeiten in die digitale Welt, auch klassische PC-Anwendungen werden mehr und mehr ausschließlich online verwendet. Zu Beginn hauptsächlich ein Informationsmedium, hat sich das World Wide Web, vor allem mit dem Aufkommen des sogenannten Web 2.0, weiterentwickelt. Termine werden in Kalendern verwaltet, die auf allen Geräten synchronisiert sind und das bequeme Einladen weiterer Teilnehmer erlauben. Filehoster versprechen die sichere Aufbewahrung persönlicher Daten in der Cloud. Der aktuelle Gipfel dieser Entwicklung sind komplette Office-Suiten, welche die Bearbeitung von komplexen Dokumenten mit vielen Personen gleichzeitig erlauben.

Die Entwicklung von Webanwendungen ist daher ein sehr lukrativer Markt mit mehreren Vorteilen. Zum Einen sind für jedes aktuelle Betriebssystem Internet-Browser verfügbar, mit denen sich die Anwendung bedienen lässt. Auf diese Weise muss die Implementierung nicht für verschiedene Plattformen erfolgen, sondern nur die Eigenheiten der entsprechenden Geräte – beispielsweise die Displaygröße bei Smartphones und Tablets – berücksichtigen. Zum Anderen gibt es faktisch keine Produktpiraterie, da das Produkt nicht im herkömmlichen Sinn an den Kunden ausgeliefert wird. Zusammen mit der Gewissheit, dass der Betreiber die Aufgabe der Datensicherung übernimmt, steigert dies die Zahlungsmoral für Webanwendungen und somit die Lukrativität der Entwicklung.

Wie bei konventionellen Anwendungen ist jedoch auch hier eine funktionierende Qualitätskontrolle von großer Bedeutung. Zwar sind im Falle eines gefundenen und behobenen Fehlers automatisch alle Anwender mit der korrigierten Version versorgt, im Gegenzug ist es ihnen jedoch nicht wie bei klassischen Anwendungen möglich, bei Problemen weiterhin eine ältere und als funktionierend bekannte Version einzusetzen. Ein unentdeckter Fehler betrifft auf diese Weise den gesamten Nutzerkreis und sollte daher noch vor der Veröffentlichung gefunden und beseitigt werden.

Bei jeder Veröffentlichung einer neuen Version für eine Webanwendung stellen sich daher zwei Fragen:

- Sind alle Funktionen fehlerfrei bedienbar?
- Verhält sich die Anwendung vergleichbar zur vorherigen Version?

Die erste Frage lässt sich bei komplexen Systemen nie mit Gewissheit beantworten, da hierzu alle möglichen Eingaben untersucht werden müssten. Allerdings helfen diverse automatische Testverfahren und die gründliche Prüfung durch Personen mit ausreichender Kenntnis der Anwendung, dem Optimum hier nahe zu kommen. Schwieriger ist die Frage, ob sich eine Anwendung anders verhält als die Vorgängerversion. Selbst wenn beide fehlerfrei sind, so können sie sich dennoch in Details unterscheiden. Solche Änderungen sind aufgrund der Weiterentwicklung von Software häufig gewünscht, können jedoch auch existierende Arbeitsabläufe stören oder gar komplett blockieren.

Ein weiterer wichtiger Aspekt bei diesen Überlegungen ist der Sicherheitsgedanke. Konventionelle Anwendungen funktionieren meist ohne jeglichen Netzwerkzugriff und lagern den Datenbestand lokal. Für Webanwendungen ist hingegen nicht nur ein höherer Zugangsschutz sicherzustellen, sondern auch eine saubere Trennung der Daten einzelner Nutzer. Entwickler müssen daher sehr genau darauf achten, durch Aktualisierungen nicht versehentlich Zugriffsrechte auszuweiten.

1.1 Beitrag dieser Dissertation

Die vorliegende Dissertation behandelt die oben genannten Probleme bei der Entwicklung von Webanwendungen mit Hilfe einer kontinuierlichen Qualitätskontrolle, welche auf Ergebnissen des maschinellen Automatenlernens basiert. Hierzu werden vier Herangehensweisen verwendet, die in der Praxis auch kombinierbar sind.

Kontinuierliches Lernen

Mit dieser Methode wird die aktuelle Entwicklungsversion dauerhaft durch einen Lernalgorithmus untersucht und aus den Ergebnissen ein Modell des Systems erzeugt. Zwar stellt dieses Modell nur eine Approximation des tatsächlichen Systemverhaltens dar, die kontinuierliche Komponente des gezeigten Ansatzes sorgt jedoch für eine stetig verbesserte Genauigkeit.

Ist eine neue Version verfügbar, so wird zunächst überprüft, ob Veränderungen gegenüber dem Vorgänger vorhanden sind. Treten Unterschiede auf, so lässt sich die Ursache auf diese Weise schnell eingrenzen. Andernfalls können die zuvor gelernten Informationen weiterverwendet werden, um das Modell weiter zu verbessern.

Das Verfahren wurde erstmals in [WNS+13] vorgestellt. Für diese Veröffentlichung bin ich Erstautor und war hauptverantwortlich für Konzeption und Umsetzung. Neben der Implementierung des Grundgerüsts, der Testtreiber und der notwendigen Alphabetsymbole führte ich die Testläufe der Fallstudie sowie deren Auswertung durch.

Automatisiertes Aufspüren des Ursprungs eines Fehlers

Häufig kommt es vor, dass ein Fehler keine direkt sichtbaren Auswirkungen hat und daher erst nach einiger Zeit entdeckt wird. Ist dann die Ursache nicht sofort klar, ist es essenziell, die spezifische Änderung zu finden, welche das Problem verursacht hat. Die für das kontinuierliche Lernen notwendigen Anpassungen des getesteten Systems erlauben in diesem Fall eine effiziente Rückwärtssuche in früheren Systemversionen. In Kombination mit einem Versionskontrollsystem lässt sich dann ermitteln, wer die Änderung durchgeführt hat und was der Grund für diese war.

Vergleich des Verhaltens von Präsentations- und Steuerungsschicht

Webanwendungen sind häufig in mehreren Schichten aufgebaut. Eine Aktion in der Präsentationsschicht durch den Nutzer wird durch eine Aktion in der darunterliegenden Steuerungsschicht ausgeführt. Durch die Vielzahl an Endgeräten kann es jedoch geschehen, dass mehrere Präsentationsschichten gleichzeitig auf dieselbe Steuerungsschicht zugreifen, weshalb vor allem Schutzmaßnahmen auf dieser unteren Schicht eingebaut sein sollten. Ein Vergleich der Schichten ermöglicht es, Unterschiede im Verhalten festzustellen.

Verifikation von Systemmigrationen

Im Laufe der Entwicklung einer Webanwendung werden häufig neue Funktionen hinzugefügt, die eine Anpassung der Persistenzschicht benötigen. An dieser Stelle müssen bereits im

System existierende Daten in das neue Schema überführt werden, sodass sich das Verhalten gegenüber neu hinzugefügten Daten nicht unterscheidet. Eine automatische Untersuchung, die Zustände von der alten Version übernimmt und nach der Migration weiterverarbeitet, kann dabei helfen, Probleme bei einer solchen Migration aufzudecken.

Zusätzlich zu diesen vier Verfahren wurde die Konstruktion von automatisierten Testfällen vereinfacht. Durch die Kombination der kontinuierlichen Qualitätskontrolle mit einem Werkzeug zur graphischen Prozessmodellierung ist eine Spezifikation von Testfällen auch für Personen möglich, die über wenig oder keine Programmierkenntnisse verfügen. Dieses Verfahren wird in [NSB+12] beschrieben, wo ich als Koautor unter anderem für die Alphabeterstellung, die Einbindung in den ACQC-Ansatz sowie für Durchführung und Auswertung der Fallstudie verantwortlich war.

1.2 Aufbau des Dokuments

Der Aufbau der vorliegenden Dissertation gliedert sich wie folgt: In Kapitel 2 werden zuerst die Grundlagen der in dieser Arbeit verwendeten Techniken erläutert und verschiedene existierende Arten von Anwendungstests vorgestellt. Die Eigenschaften und Vorteile der kontinuierlichen Qualitätskontrolle werden in Kapitel 3 aufgezeigt. Anschließend folgt in den Kapiteln 4 und 5 die Beschreibung zweier Fallstudien, bei denen der Ansatz erfolgreich angewendet wurde. Ein wichtiger Teil dieser Arbeit ist die Erstellung von Alphabetmodellen, für den eine alternative Herangehensweise in Kapitel 6 gezeigt wird. Verwandte Arbeiten werden in Kapitel 7 vorgestellt, bevor abschließend in Kapitel 8 ein Fazit gezogen und mögliche künftige Entwicklungen ausgelotet werden.

2 Grundlagen

In diesem Kapitel werden die nötigen Grundlagen, welche in der Dissertation verwendet werden, erläutert. Dies beinhaltet sowohl die zu testenden Webapplikationen und die dafür existierenden Testmethoden als auch automatische Lernverfahren. Anschließend folgt eine Beschreibung der in dieser Arbeit eingesetzten Softwarekomponenten.

2.1 Reaktive Systeme

Der Begriff des *reaktiven Systems* wurde bereits 1985 eingeführt [HP85] und bezeichnet ein konstant laufendes System, das auf äußere Einflüsse reagiert. Diese Definition ist nicht auf die Informatik beschränkt, auch biologische Systeme sind damit erfasst. Die im Rahmen dieser Dissertation betrachteten Systeme beschränken sich jedoch auf informationstechnische Varianten, die auf gestellte Eingaben reagieren und nach [Hal98] mehrere Anforderungen erfüllen:

Determinismus Das Verhalten auf die gestellten Eingaben muss auch bei mehreren Ausführungen dasselbe Ergebnis liefern.

Nebenläufigkeit Das System ist permanent aktiv und kann auch mehrere parallele Anfragen aus seiner Umgebung verarbeiten.

Zuverlässigkeit Fehler in kritischen Komponenten, die als reaktive Systeme gestaltet wurden, können fatale Konsequenzen haben. Reaktive Systeme müssen daher zuverlässig und nach Möglichkeit auch innerhalb einer gewissen Zeitspanne auf Eingaben reagieren.

Die Forderung des Determinismus ist schwierig zu erfüllen, wenn das System aus mehreren Quellen der Umgebung Eingaben erhält. So kann die Interaktion eines Klienten mit dem System den internen Zustand derart ändern, dass Reaktionen auf andere Klienten fortan anders verarbeitet werden. Für eine deterministische Betrachtung ist es daher wichtig, die gesamte Umgebung des Systems kontrollieren zu können.

Ein Beispiel für ein reaktives System stellt etwa ein Personenaufzug dar. Sein interner Zustand ist das aktuelle Stockwerk, welches durch eine Anzeige jedem (potenziellen) Fahrgast dargestellt wird. Eingaben erfolgen durch Personen, welche über den Rufknopf eine Fahrt anfordern. Befindet sich ein Fahrgast beispielsweise im vierten Stock und der Aufzug im ersten, so kann normalerweise davon ausgegangen werden, dass sich der Fahrstuhl nach oben in Bewegung setzt. Betätigt jedoch eine andere Person im Erdgeschoss ebenfalls den Rufknopf, so bewegt sich der Aufzug aus Sicht der Person im vierten Stock aus unbekannten Gründen in die falsche Richtung.

Die in dieser Dissertation betrachteten reaktiven Systeme reagieren, zusätzlich zu den oben genannten Eigenschaften, auf gestellte Eingaben mit entsprechenden Ausgaben. Bei diesen Ausgaben kann es sich sowohl um berechnete Werte als auch um eine Information über den internen Systemzustand handeln. Hierbei ist jedoch ebenfalls wichtig, dass die generierten Ausgaben deterministisch nachvollziehbar sind. Auf diese Weise ist es möglich, das Verhalten

in Automatenmodellen (siehe Abschnitt 2.4) abzubilden. Eine weit verbreitete Art solcher Systeme sind die im nächsten Abschnitt beschriebenen Webapplikationen.

2.2 Webapplikationen

Unter einer Webapplikation versteht man eine Anwendung, die vom Nutzer über einen Webbrowser bedient wird, der wiederum mit einem Webserver kommuniziert. Dabei ist es unerheblich, ob der Webserver im Internet, einem Rechner im firmeneigenen Intranet oder auf dem lokalen Rechner des Nutzers läuft. Anfragen an die Anwendung sowie deren Beantwortung werden über standardisierte Protokolle wie HTTP verschickt und das Ergebnis daraufhin im Browser des Nutzers dargestellt. Bekannte Beispiele für solche Anwendungen sind Webmailer, Kalender, Bugtracker oder Wikis.

2.2.1 Aufbau von Webapplikationen

Webapplikationen lassen sich auf eine Vielzahl von Arten realisieren. Die gebräuchlichste Form ist die der Drei-Schichten-Architektur [Eck95], bestehend aus einem über ein Netzwerk erreichbaren Webserver, der meist statischen Inhalt liefert (Präsentationsschicht), einem vom Webserver angesprochenen Applikationsserver (Anwendungslogik) sowie einer Datenbank (Datenschicht). Der Ablauf der Kommunikation, welche durch einen Nutzer über einen Browser initiiert wird, ist schematisch in Abb. 2.1 dargestellt. Sobald der Nutzer eine Anfrage an die Anwendung stellt, wird diese vom Webserver empfangen und im Falle einer notwendigen Auswertung vom Applikationsserver weiterverarbeitet. Letzterer bietet eine Laufzeitumgebung für spezielle Hochsprachen wie Java oder .NET und greift eventuell auf eine Datenbank oder eine andere Persistenzschicht zu, bevor er das Ergebnis über den Webserver zurück an den Browser liefert. Bei einer Anfrage kann es sich dabei etwa um das Anklicken eines Links oder das Absenden eines Formulars, z.B. dem Suchfeld einer Suchmaschine, handeln. Die empfangene Antwort wiederum besteht im einfachsten Fall aus einem HTML-Dokument, das im Browser des Nutzers dargestellt wird.

Der gezeigte Aufbau variiert je nach den Anforderungen des angebotenen Dienstes. Ist nur ein einziger Webserver für die Kommunikation verantwortlich, so ist die gesamte Anwendung nicht benutzbar, sobald der Server wegen Wartungsarbeiten deaktiviert ist. Viele Anbieter setzen daher auf eine größere Anzahl von Servern, die über ein Lastverteilungssystem automatisch aufgerufen werden.

Ähnliche Überlegungen gelten für den Einsatz des Applikationsservers. Teilweise verfügen solche Server bereits über Techniken, um Ausfällen entgegenzuwirken, beispielsweise durch die Vernetzung mehrerer Instanzen. Zusätzlich ist es möglich, im Hintergrund mehrere Datenbankserver parallel zu betreiben, um sowohl den Datendurchsatz als auch die Ausfallsicherheit zu erhöhen.

2.2.2 Vor- und Nachteile

Der große Vorteil von Webapplikationen liegt darin, dass der interne Aufbau für den aufrufenden Nutzer völlig transparent ist. Weder die eingesetzte Programmiersprache noch die Struktur der dahinterliegenden Datenbankserver sind für ihn relevant. Weiterhin ist für die Benutzung meist keine spezielle Software erforderlich, ein aktueller Internetbrowser ist ausreichend und auf jedem System im Regelfall vorinstalliert. Ausnahmen bilden hier Anwendungen, die spezielle Plug-Ins wie *Adobe Flash* oder eine Java-Laufzeitumgebung voraussetzen.

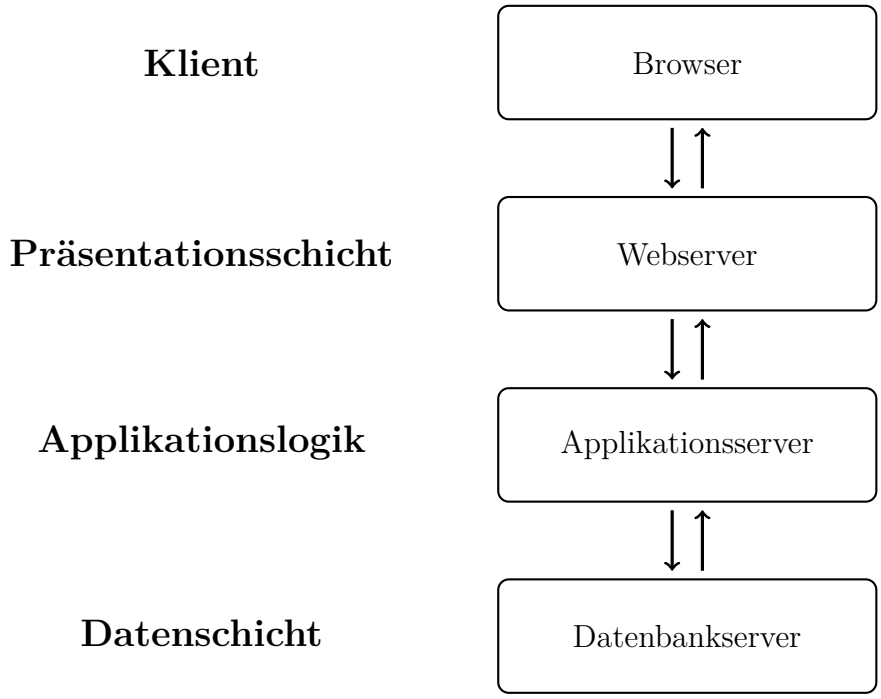

Abbildung 2.1: Kommunikation eines Webbrowsers mit einer Webapplikation

Grundsätzlich sollte sogar jede der genannten Schichten austauschbar sein, ohne dass dies einen Einfluss auf den Anwender hat. Angefangen beim durch den Nutzer verwendeten Browser, der durch Verwendung standardisierter Techniken nicht auf ein Produkt festgelegt ist, bis hin zur installierten Datenbank, deren Austausch dank eingesetzter Abstraktionsschichten möglich ist, sollte sich die gesamte Anwendung kontinuierlich identisch bedienen lassen.

Weiterhin braucht der Nutzer sich nicht um Details wie Datensicherungen oder Softwareaktualisierungen zu kümmern. Der Betreiber einer Webapplikation fertigt im Regelfall ein zentrales Backup der Daten aller Nutzer an und kann dies im Notfall ohne deren Zutun wieder rekonstruieren. Fehler durch veraltete Anwendungsversionen gehören ebenfalls der Vergangenheit an, da der Betreiber die neue Version auf seinen Servern installieren kann und die Nutzer ab sofort mit genau dieser arbeiten. Bei größeren Diensten wie beispielsweise dem sozialen Netzwerk *Facebook* kommt es jedoch durchaus vor, dass neue Funktionen erst nach und nach für bestimmte Nutzerkreise, etwa abhängig vom Land des Nutzers, freigeschaltet werden.

Vom ökonomischen Standpunkt eines Entwicklers aus liegt ein weiterer Vorteil in der niedrigen Produktpiraterie. Die meisten kommerziellen Betriebssysteme und Applikationen sind illegal im Internet verfügbar und werden häufig eingesetzt, ohne die entsprechenden Lizenzkosten zu begleichen. Bei Webanwendungen ist dies nur schwer möglich, da der Nutzer keine installierbare Komponente erhält, die sich illegal verbreiten ließe, sondern nur den Zugriff per Browser auf eine in vielen Fällen öffentlich zugängliche Webseite. Da kein Zugang zu der auf dem Server betriebenen Komponente besteht, wäre hier wesentlich mehr kriminelle Energie nötig, um die Software zu erhalten und zu verbreiten. Selbst wenn dieser Fall eintritt, müsste der Nutzer einen eigenen Server bereitstellen, was deutlich komplexer als die Installation einer einzelnen Applikation auf dem PC ist und die Hürde für die Nutzung weiter nach oben legt.

Allerdings können sich genau diese Vorteile auch als Nachteile herausstellen. Ist ein Dienst nicht erreichbar, hat der Nutzer in der Regel keinen Zugriff auf seine Daten. Darüber hinaus muss er dem Betreiber ein gewisses Vertrauen entgegenbringen, dass dieser mit dem ihm anvertrauten Daten korrekt umgeht. Dazu gehören nicht nur die Durchführung einer regelmäßigen Sicherung sondern auch der Schutz vor Datendieben oder die Unterlassung einer Weitergabe der Daten an Dritte. Fehlt eine Export-Funktion der Daten, kann es bei der Schließung eines Dienstes zudem passieren, dass die in der Anwendung hinterlegten Daten für die Nutzer nicht mehr zugänglich sind.

2.3 Qualitätskontrolle durch Anwendungstests

Vor der Veröffentlichung bzw. Abgabe eines Produkts oder der Veröffentlichung einer aktualisierten Version sind Tests unabdingbar, welche die Anwendung auf Funktionsfähigkeit prüfen. Abhängig von Art der Anwendung sowie dem Ablauf der Entwicklung lassen sich dazu verschiedene Arten von Tests unterteilen. Man unterscheidet dabei sowohl den Umfang der einzelnen Tests als auch das Wissen der Tester über das System.

2.3.1 Statische Code-Analyse

Moderne Entwicklungsumgebungen für Programmiersprachen erleichtern einem Entwickler seine Arbeit auf unterschiedliche Weise. Nicht nur die Dokumentation ist direkt im Editor verfügbar, auch Syntaxfehler werden schon während der Eingabe erkannt und Vorschläge zur Beseitigung angeboten. Auf diese Weise erübrigt sich ein separater Aufruf des entsprechenden Compilers und der Entwickler kann dank der direkten Rückmeldung Fehler frühzeitig erkennen.

Diese Meldungen beschränken sich jedoch nicht auf Fehler, die spätestens beim Kompilieren eines Programms sowieso aufgefallen wären, sondern zeigen auch andere häufig gemachte Programmierfehler auf. Ein Beispiel ist die Durchführung einer Datenflussanalyse [Ste91, Ste93, LMS06] und die anschließende Hervorhebung unbenutzter Variablen, deren Verwendung zwar vorgesehen war, aber nie umgesetzt wurde. Hier kann es sein, dass die notwendige Verarbeitung nicht implementiert wurde und das fertige Programm sich daher nicht wie erwartet verhalten wird. Eine andere Möglichkeit ist, dass die in den Variablen gespeicherten Werte im Laufe des Entwicklungsprozesses wirklich nicht mehr benötigt werden. In diesem Fall kann sich eine Entfernung positiv auf die Performance des fertigen Produkts auswirken.

Während das Erkennen von unbenutzten Variablen gerade in lokalen Methoden nur einfache Prüfungen durch die Software erfordert, beherrschen viele Produkte zur statischen Code-Analyse auch deutlich ausgefeiltere Mechanismen. So lassen sich nicht nur unbenutzte Variablen, sondern auch nicht verwendete Methoden aufspüren. Endlose Rekursionen können ebenso identifiziert werden wie Verzweigungen, die nie durchlaufen werden, weil die überprüfte Bedingung einen konstanten Wert hat. Zusätzlich geben Metriken wie etwa die zyklomatische Komplexität [McC76] Aufschluss darüber, ob Klassen oder Methoden bestimmte Größen oder Verschachtelungstiefen überschreiten.

Da der Funktionsumfang der auf dem Markt befindlichen Entwicklungsumgebungen sich in dieser Hinsicht stark unterscheidet, wird die statische Code-Analyse häufig durch externe Werkzeuge durchgeführt und in den Build-Prozess ausgelagert. Für Java existiert etwa die

Software FindBugs [HP04], die über ein Maven-Plugin[1] eingebunden werden kann. Ein weiterer Vorteil dieser Lösung ist, dass die Einrichtung der Software an zentraler Stelle erfolgt, was den Aufwand für jeden einzelnen Entwickler verringert und dafür sorgt, dass alle mit der aktuellen Konfiguration arbeiten.

Die statische Code-Analyse kann zwar dabei helfen, Fehler im Vorfeld zu vermeiden, sie ist jedoch kein Mittel der Wahl, um die Funktionsfähigkeit einer Software zu testen. Der Grund dafür ist, dass sie in keiner Weise das Verhalten des laufenden Systems betrachtet oder gar die Kombination diverser Komponenten berücksichtigt. Dazu existieren verschiedene Testverfahren, die im Folgenden vorgestellt werden.

2.3.2 Modultests

Komplexe Softwaresysteme bestehen aus einer Vielzahl von Modulen, die sich einzeln, also ohne Wechselbeziehung, testen lassen. Im einfachsten Fall existieren für jede Methode einer Klasse eine oder mehrere Funktionstests, welche die Reaktion auf erlaubte oder auch unspezifizierte Eingabeparameter prüfen.

Modultests sollten schnell ausführbar sein und keine Abhängigkeiten zu externen Diensten, wie etwa Dateisystemen oder Datenbanken, besitzen. Sind die Module auf solche externen Dienste angewiesen, so lässt sich deren Verhalten über sogenanntes *Mocken* [Lin05] vorgeben. Auf diese Weise können die Tests von den Programmierern bereits während der Softwareentwicklung aufgerufen werden, um einen schnellen Überblick der Auswirkungen von Änderungen zu erhalten.

Aufgrund ihrer Nähe zum zu testenden Quellcode und dem Umstand, dass die Testfälle meist in derselben Programmiersprache geschrieben sind wie das eigentliche Programm, eignen sich Modultests ebenfalls zur Dokumentation der Nutzung einzelner Methoden sowie zur Beschreibung erwarteten Verhaltens. Dass ihre Ausführung kein vollständig laufendes System mit allen Abhängigkeiten voraussetzt, erhöht zudem die Motivation der Mitarbeiter, solche Modultests anzulegen und häufig auszuführen.

2.3.3 Integrationstests

Nachdem die Funktionsweise der einzelnen Komponenten anhand der Modultests überprüft wurde, ist eine Betrachtung des Zusammenspiels sinnvoll. Dies wird durch einen Integrationstest umgesetzt, der „eine Aktivität zwischen dem Ende des Komponententests und dem Beginn des Systemtests“ [WEMS+12] darstellt und durch die Modularisierung aktueller Software begünstigt wird. Hierbei werden die Module kombiniert und anhand von Schnittstellentests geprüft. Dies kann, je nach Umfang der Tests, wie beim Modultest direkt vom Entwickler durchgeführt werden oder auch automatisiert geschehen.

Die Ausführung von Integrationstests kann flexibel auf das jeweilige System angepasst werden. Ein Ansatz ist zum Beispiel, existierende Komponenten paarweise miteinander zu verbinden und so ohne Störung durch andere Faktoren zu prüfen. Auch ist es denkbar, inkrementell weitere Module hinzuzufügen und so die Komplexität des Testsystems nach und nach zu steigern.

In der Praxis werden Integrationstests jedoch wesentlich seltener durchgeführt als etwa Modul- oder Systemtests. Ein Grund dafür können die Zuständigkeiten sein: Entwickler fühlen sich zwar für das von ihnen entwickelte Modul verantwortlich und schreiben entsprechende Modultests, vernachlässigen aber die Kombination mit anderen Modulen. Zudem wird oft

[1] `http://mojo.codehaus.org/findbugs-maven-plugin/`

ein vollständiger Systemtest als großer Integrationstest angesehen. Eine isolierte Betrachtung der Schnittstellen kann jedoch im Rahmen eines Integrationstests deutlich gründlicher in kürzerer Zeit erfolgen, sodass diese Art des Anwendungstests nicht vernachlässigt werden sollte.

2.3.4 Systemtests

Liegt der Fokus bei Modul- und Integrationstests noch auf dem Zusammenspiel ausgewählter Module, so wird bei einem sogenannten Systemtest die gesamte Anwendung untersucht. Dazu wird meist eine Umgebung simuliert oder erstellt, welcher der Produktivumgebung beim Kunden entspricht.

Systemtests können sehr aufwändig sein und auch schon mal mehrere Stunden andauern. Eine häufige Ausführung durch die Entwickler ist hier nicht sinnvoll, weshalb sich automatisierte Testumgebungen anbieten, welche diese Arbeit etwa nachts durchführen. Stehen keine automatisierten Tests bereit oder sollen diese ergänzt werden, kann ein Systemtest auch zu bestimmten Zeitpunkten, etwa vor der Veröffentlichung einer neuen Version, von menschlichen Testern oder gar vom Kunden selbst durchgeführt werden.

Beide Varianten haben ihre Vor- und Nachteile. Während automatisiert laufende Testverfahren meist wenig flexibel in der Auswahl der durchzuführenden Testfälle sind, erfordert die manuelle Durchführung einen deutlich höheren Zeit- und Personalaufwand. Abhilfe bietet hier eine Kombination der beiden Verfahren, bei der Testfälle in einer einfach zu bedienenden graphischen Umgebung zusammengestellt und dann automatisiert ausgeführt werden [NSM+01, MS02].

2.3.5 Black-Box-Tests

Wird ein Systemtest von Personen durchgeführt, die kein Wissen über den inneren Aufbau des Systems haben oder dieses bewusst nicht verwenden, so spricht man von Black-Box-Tests [Bei95]. Ein Beispiel hierfür ist der Konformitätstest eines Produktes, für das aus Gründen des Schutzes geistigen Eigentums kein Quellcode zur Verfügung steht.

Während eines solchen Tests werden Anfragen an das zu testende System gestellt. Da die interne Verarbeitung dieser Anfragen nicht sichtbar ist, können nur Schlüsse anhand des äußerlich sichtbaren Verhaltens gefolgert werden. Der Test gilt als bestanden, wenn das Verhalten mit der Spezifikation des Systems übereinstimmt.

2.3.6 White-Box-Tests

Im Gegensatz zum Black-Box-Test ist beim White-Box-Test (auch Glass-Box-Test genannt) der innere Aufbau und insbesondere der Quellcode des Systems bekannt. Das Wissen um den Quellcode kann hierbei etwa dazu genutzt werden, um zu prüfen, welcher Anteil des Codes von den Tests aufgerufen wird. Diese Art des Testens wird daher vorrangig bei Modul- und Integrationstests verwendet. Ziel hierbei sollte es sein, eine möglichst vollständige Abdeckung zu erzielen. Die tiefergehende Analyse des Sourcecodes ermöglicht darüber hinaus das Aufdecken von versteckten Fehlern, etwa wenn ein Problem nur unter bestimmten Umständen auftreten kann, die mit anderen Testverfahren möglicherweise nicht abgedeckt wären.

Das Problem von White-Box-Tests besteht in der engen Verknüpfung der Tests mit dem Code der Anwendung. Auf diese Weise wird vorrangig die korrekte Funktionsweise des Systems und weniger die Übereinstimmung mit dessen Spezifikation geprüft.

2.3.7 Grey-Box-Tests

Während White-Box-Tests nur von Entwicklern mit Zugriff auf die Systeminterna durchgeführt werden können, ist ein Black-Box-Test auch von Anwendern durchführbar, welche nur die Spezifikation oder die Programmschnittstellen (API) des Systems kennen. Eine Vereinigung der Vorteile dieser beiden Testarten verfolgt das sogenannte Grey-Box-Testing [KGTB07]. Hierbei benötigt der Tester keinen Zugriff auf den Quellcode des Systems, sondern nur Kenntnis über für den Test relevante Interna. Dabei kann es sich etwa um interne Dokumentation oder das Wissen über die verwendeten Algorithmen handeln.

2.3.8 Fuzzing

Eine weiteres Testverfahren, das sich sehr gut für Webanwendungen eignet, ist das sogenannte Fuzzing [SGA07]. Dabei werden automatisiert Eingaben erzeugt und die Reaktion des Systems darauf getestet. Bei Webanwendungen lassen sich so etwa URL-Parameter und Eingabefelder daraufhin überprüfen, ob beispielsweise bei Überschreitung einer gewissen Länge einer Zeichenkette die Sicherheitsfunktionen umgangen werden können. Dies geschieht normalerweise durch Black-Box-Testing, die innere Struktur der Software ist also nicht bekannt. Erst wenn ein möglicher Fehler gefunden wurde, kann Fuzzing dort mit Kenntnis des Quellcodes helfen, die genaue Ursache aufzudecken.

2.3.9 Formale Verifikation

Im Gegensatz zu Testverfahren, welche direkt die lauffähige Anwendung (oder auch nur Teile davon) auf korrektes Verhalten prüfen, soll die formale Verifikation einen Beweis dafür liefern, ob ein System sich konform zu einer gegebenen Spezifikation verhält. Eine Möglichkeit dazu wäre ein mathematischer Beweis, der für ein Programm zeigt, dass es auf alle möglichen Eingaben korrekt reagiert. Dies ist etwa mit Kalkülen wie dem Hoare-Kalkül [Hoa69] möglich, jedoch für komplexe Systeme nur unter hohem Aufwand anwendbar [Cla77] und daher nur für kritische Systeme, bei denen etwa Menschenleben gefährdet sind, sinnvoll.

2.3.10 Modellbasiertes Testen

Modellbasiertes Testen [UL07, BDG$^+$08] ist eine Variante der formalen Verifikation und erlaubt eine automatisierte Erstellung von Testfällen, sofern für ein zu testendes System bereits ein abstraktes Modell existiert, welches das erwünschte Systemverhalten abbildet. Ein solches Modell kann schon zu Beginn der Entwicklung aus einer Formalisierung der Anforderungen entstehen (zum Beispiel aus UML-Diagrammen [OA99]) oder aber auch nach erfolgter Fertigstellung des Systems durch aktives Automatenlernen.

Um aus dem Modell auf algorithmischem Wege Testfälle zu erzeugen, kann es beispielsweise in einen endlichen, deterministischen Automaten umgewandelt werden. Auf diese Weise ist es möglich, durch Suche nach ausführbaren Pfaden auf dem Automaten Aktionsfolgen zu finden, die auch auf dem zu testenden System (System Under Test, SUT) möglich sein sollten. Da diese Menge bei komplexen Systemen schnell sehr groß werden kann, sind Filterkriterien zur Auswahl der durchzuführenden Tests notwendig.

Das Modell stellt ein abstraktes Abbild des SUT dar, daher sind die daraus erzeugten Testfälle ebenfalls abstrakt und können nicht direkt ausgeführt werden. Ein sogenannter *Mapper* ist an dieser Stelle dafür verantwortlich, die gewählten Testfälle ausführbar zu machen,

beispielsweise durch eine Umwandlung in konkreten Quellcode der verwendeten Programmiersprache. Aus diesem Grund wird modellbasiertes Testen auch die „Automatisierung des Designs von Black-Box-Tests“ [UL07] genannt.

Bei der Ausführung des modellbasierten Testens unterscheidet man zwischen den Varianten *Online* und *Offline*. Während Online-Tests direkt auf dem SUT ausgeführt werden, generiert die Offline-Variante ausführbare Tests, die zu einem späteren Zeitpunkt aufgerufen werden können. Hierbei muss es sich jedoch nicht einmal um ausführbaren Quellcode handeln, auch eine textuelle, menschenlesbare Beschreibung ist möglich. Auf diese Weise lassen sich beispielsweise automatisch Prüflisten für Abnahmetests generieren.

2.3.11 Lernbasiertes Testen

Das lernbasierte Testen von reaktiven Systemen [HMS03, MNRS04, RMSM09] setzt voraus, dass das zu testende System von einem Lernalgorithmus angesprochen werden kann. Während des Testablaufs werden dann Testfälle in Form von Anfragen erzeugt, an das System gestellt und die Ausgabewerte aufgezeichnet. Anhand der Kombination von Ein- und Ausgabewerten kann dann ein abstraktes Modell des SUT synthetisiert werden, welches mit einer vorhandenen formalen Spezifikation abgeglichen werden kann.

Existieren Unterschiede zwischen Modell und Spezifikation, so ist eine Prüfung darauf erforderlich, ob die Anzahl und Auswahl der Testfälle nicht ausreichend war und das Modell verfeinert werden muss, oder ob sich das SUT tatsächlich nicht konform zur Spezifikation verhält [PVY99]. Im ersten Fall kommt ein Algorithmus zur Anwendung, der das Modell mit Hilfe der zusätzlichen Informationen iterativ erweitert.

Ein großer Vorteil dieses Ansatzes liegt darin, dass er den Testaufwand deutlich vereinfacht [SN11]: Die manuelle Erstellung von Testfällen entfällt, zusätzlich werden leicht verständliche Modelle erzeugt, welche das Systemverhalten abbilden.

Selbst wenn initial keine formale Spezifikation des Systems vorliegt, kann das lernbasierte Testen genutzt werden, um Veränderungen zu entdecken. Hierzu erfolgt ein Vergleich des aktuell gelernten Modells mit dem Ergebnis früherer Iterationen. Weiterhin ist ein Vergleich nicht nur mit vollständig gelernten Modellen möglich, auch eine Kombination mit Kontrollsystemen ist denkbar, welche die Kommunikation eines laufenden Systems überwachen [BCMS12]. Tritt eine unerwartete Reaktion auf, so kann dies entweder auf einen Fehler im System hinweisen oder Informationen zur weiteren Verfeinerung des gelernten Modells liefern.

2.3.12 Risiko-orientiertes Testen

Die gründliche Suche nach Fehlern in einem Softwareprodukt ist eine wichtige Komponente bei der Erstellung qualitativ hochwertiger Programme, doch häufig fehlt es hier an den notwendigen Ressourcen. Vor allem kurz vor der Veröffentlichung einer neuen Version ist häufig Zeit ein knapper Faktor, und gerade manuelle Prüfungen durch menschliche Tester können für ein Unternehmen im Vergleich zu automatisierten Testverfahren wesentlich höhere Ausgaben bedeuten.

Je nach Art des Produkts ist jedoch eine vollständige Auswertung aller existierenden Tests gar nicht notwendig, sodass eine sorgfältige Auswahl der durchzuführenden Tests hier Zeit und Geld sparen kann. Wurde etwa nur eine einzelne Funktion überarbeitet, so kann es ausreichen, diese separat zu testen und eventuell zusätzlich andere zu betrachten, welche mit dieser im Zusammenhang stehen. Eine andere Möglichkeit besteht darin, Kernfunktionen

auf jeden Fall immer zu prüfen, um im Falle eines übersehenen Fehlers dem Kunden zumindest keine vollkommen unbenutzbare Version zu präsentieren. Eine solche Priorisierung von Testfällen wird *Risiko-orientiertes Testen* (Risk-based Testing, RBT) genannt [FR14].

2.4 Maschinelles Automatenlernen

Beim maschinellen Lernen von Systemen, deren innere Funktionsweise nicht oder nur teilweise bekannt ist, wird aus Reaktionen des Systems auf gestellte Eingaben ein Modell des Systemverhaltens erstellt. Dieses Modell weist im Idealfall dasselbe Ein- und Ausgabeverhalten wie das reale System auf und kann beispielsweise dazu verwendet werden, die Erfüllung einer Spezifikation zu überprüfen. Aufgrund der Einschränkungen einer Black-Box kann jedoch nie mit Sicherheit festgestellt werden, ob das reale System korrekt und vollständig vom gelernten Modell abgebildet wird.

Zur Erstellung des Modells sind Beobachtungen des Systemverhaltens notwendig. Hat die aktiv lernende Komponente – im Folgenden Lerner genannt – keine Möglichkeit, eigene Anfragen an das SUL (System Under Learning) zu stellen, so muss er sich auf frühere Ausgaben des Systems, die etwa in Form von Logfiles vorliegen, beschränken. Diese Art des Lernens wird passives Lernen genannt. Demgegenüber steht das aktive Lernen, bei welchem der Lerner die Anfragen selbst generiert.

Das aktive Automatenlernen wird auch als testbasierte Modellierung bezeichnet, da auf Basis von heuristisch ausgewählten Tests ein Modell erzeugt wird. Dies darf jedoch nicht mit dem modellbasierten Testen verwechselt werden, welches in Abschnitt 2.3.10 beschrieben wurde.

2.4.1 Passives Lernen durch Logfile-Analyse

Die meisten Software-Produkte protokollieren – automatisch oder auf ausdrücklichen Wunsch des Nutzers – ihr Verhalten in Logdateien. In erster Linie dient dies der Fehlersuche: Tritt im laufenden Betrieb ein Problem auf, geben die Dateien häufig einen guten Hinweis, wo das Problem liegen könnte. Ist dies nicht der Fall, lassen sich oft zusätzliche Meldungen archivieren, die eigentlich für die Entwickler der Software gedacht sind. Werden diese Informationen zusammen mit einem Fehlerbericht an die Entwickler geschickt, sind diese eher in der Lage, das Problem aufzuspüren und zu beheben.

Eine weitere Anwendung für Logdateien ist die Archivierung von wichtigen Ereignissen, auch wenn diese nicht durch einen Fehler verursacht werden. Als Beispiel sei hier ein Intrusion-Detection-System genannt, welches ungewöhnliches Verhalten an Netzwerkdiensten bemerkt und an entsprechende Personen weiterleitet. Zwar findet hier auch eine Benachrichtigung per E-Mail oder SMS statt, ein Eintrag in den System-Logs ist jedoch auch später noch von anderen berechtigten Personen einsehbar. Ein anderes Beispiel ist die Aufzeichnung von Zugriffen auf einen Webserver. Mit Hilfe geeigneter Werkzeuge wie etwa Webalizer[2] können diese grafisch aufbereitet werden, um Informationen über die Nutzer einer Webpräsenz, beispielsweise deren geographische Herkunft oder der verwendete Browser, zu erhalten.

Zusätzlich können Logdateien genutzt werden, um Veränderungen am System nachzuweisen. Wird in einer Webanwendung jede schreibende Änderung festgehalten, so lässt sich im Nachhinein leicht der verantwortliche Nutzer ermitteln, wenn etwa Daten – ob versehentlich oder vorsätzlich – gelöscht oder verändert wurden. Dies setzt allerdings voraus, dass die

[2] `http://www.webalizer.org/`

Logdateien möglichst gegen Veränderungen durch Administrator-Accounts geschützt sind, beispielsweise durch elektronische Signaturen.

Logdateien lassen sich jedoch nicht nur zur Überwachung von Systemen im Produktiveinsatz verwenden, auch während der Entwicklung können sie hilfreiche Dienste leisten. Mit Hilfe der in [And98] vorgestellten Sprache LFAL lassen sich Analysierer für Logdateien spezifizieren, die nicht nur für vollständige Systemtests, sondern auch zielgerichtet für die Durchführung von Modultests verwendet werden können.

Eines der Hauptprobleme bei der Protokollierung des Systemverhaltens ist die Auswahl der relevanten Einträge. Werden zu wenige Ereignisse gespeichert, besteht die Gefahr, wichtige Veränderungen zu übersehen. Das Gleiche kann jedoch auch bei einer zu großen Anzahl passieren, die manuell nicht mehr sinnvoll ausgewertet werden kann. Programme wie das Unix-Tool logcheck[3] erlauben hier, anhand von regulären Ausdrücken nur ungewöhnliche Vorkommnisse herauszufiltern. Dies setzt jedoch eine entsprechende Konfiguration voraus, die meist auch laufend an neue Gegebenheiten angepasst werden muss. In [Mem08] wird hierfür ein automatisierter Ansatz vorgestellt, der Logeinträge durch Inferenz einer entsprechenden Grammatik automatisch kategorisieren und Verhaltensauffälligkeiten erkennen kann.

Eine solche abgeleitete Grammatik kann verwendet werden, um ein Modell des untersuchten Systems zu erstellen [CW98], was selbst für nebenläufige Prozesse möglich ist [CDLW04]. Allerdings basieren die so erzeugten Resultate nur auf den – möglicherweise unzureichenden – Logdateien. Je nach Nutzungsintensität des Systems und Menge der protokollierten Daten kann es durchaus vorkommen, dass eine sinnvoll auswertbare Menge erst nach einigen Wochen bereitsteht. Dies mag für die nachträgliche Untersuchung von Produktivsystemen ausreichend sein, eignet sich jedoch nicht für den Vergleich von in der Entwicklung befindlichen Versionen. Zwar wäre es hier möglich, die von automatisierten Systemtests erzeugten Logdateien auszuwerten, allerdings würde dies das Ergebnis auf die vorher festgelegten Tests einschränken. Um jedoch auch unvorhergesehene Fehler aufzudecken, sind aktive Verfahren in der Lage, ein System selbstständig zu untersuchen.

2.4.2 Aktives Lernen von Black-Box-Systemen

Im Gegensatz zu passiven Lernverfahren basiert das Ergebnis bei der aktiven Variante nicht nur auf bereits existierenden Daten, sondern der Lernalgorithmus ist in der Lage, das SUL zu beeinflussen und mit den daraus resultierenden Reaktionen das gelernte Modell zu verbessern. Bereits seit fast 30 Jahren wird an diesen Verfahren geforscht, an der Verwendung im Zusammenhang mit realen Systemen seit über 10 Jahren [IHS13b]: Neben der Unterstützung zusätzlicher Modelltypen wie Mealy-Maschinen [Mea55] oder Registerautomaten [HSJC12] wurden unter anderem auch Erweiterungen entwickelt, welche die Lernvorgänge immens beschleunigen.

Wie in Abschnitt 2.3.5 beschrieben, verfügt der Lerner eines Black-Box-Systems über kein internes Wissen des SUL. Er muss jedoch voraussetzen können, dass das System deterministisch und nur auf solche Eingaben reagiert, die vom Lerner beobachtet werden können oder von diesem erzeugt wurden. Dies ist für Webapplikationen in einer kontrollierten Umgebung, also ohne Zugriff durch Dritte auf das System, gegeben.

Zur Darstellung des gelernten Modells wird ein Mealy-Automat verwendet.

[3] `http://sourceforge.net/projects/logcheck/`

Definition 1 (*Mealy-Automat*)

Ein Mealy-Automat ist definiert als ein Tupel $M = (Q, \Sigma, \Omega, \delta, \lambda, q_0)$ *mit*

- Q *als eine endliche Menge von Zuständen*
- Σ *als dem endlichen Eingabealphabet*
- Ω *als dem endlichen Ausgabealphabet*
- $\delta : Q \times \Sigma \to Q$ *als der Zustandsübergangsfunktion*
- $\lambda : Q \times \Sigma \to \Omega$ *als der Ausgabefunktion*
- $q_0 \in Q$ *als dem Startzustand*

Übergibt man ein beliebiges Eingabewort $w_i \in \Sigma^*$ an einen solchen Automaten, so erhält man ein Ausgabewort $w_o \in \Omega^*$ derselben Länge. Auf das Modell einer Webapplikation übertragen bedeutet dies, dass eine bestimmte Abfolge von Aktionen (w_i) eine Ausgabe (w_o) erzeugt, anhand derer beispielsweise erkennbar ist, ob die Aktionen erfolgreich durchgeführt wurden oder ob sich eventuell der interne Systemzustand geändert hat. Für die Umwandlung von Aktionen in Eingabesymbole und die Auswertung der Ausgabesymbole ist ein sogenannter Testtreiber (siehe Abschnitt 3.2.2) verantwortlich.

Ein Lernalgorithmus erzeugt mit Hilfe solcher Anfragen im ersten Schritt ein Hypothesemodell in Form eines Mealy-Automaten. Dieses wird mittels einer sogenannten *Equivalence Query* daraufhin untersucht, ob das gelernte Modell mit dem eigentlichen System übereinstimmt. Ist dies nicht der Fall, liefert die Equivalence Query ein Gegenbeispiel in Form eines Eingabeworts, welches vom Hypothesenautomaten nicht korrekt behandelt wird. Anhand dieses Gegenbeispiels kann der Algorithmus nun die Hypothese verbessern. Die Gegenbeispielbehandlung wird so häufig ausgeführt, bis kein Gegenbeispiel mehr gefunden werden und der Automat als vollständig angesehen werden kann.

2.4.3 Der Algorithmus L^*_M

Als Lernalgorithmus kommt im Rahmen dieser Dissertation eine angepasste Variante des L^*-Algorithmus von Dana Angluin [Ang87] zum Einsatz. Dieser wurde ursprünglich für deterministische, endliche Automaten (kurz DEA oder DFA) entwickelt und zeichnet sich unter anderem dadurch aus, dass sich die verwendete Datenstruktur auf einfachem Wege speichern und vergleichen lässt. Die in [MNRS04] beschriebene Erweiterung L^*_M ermöglicht zusätzlich die Inferenz von Mealy-Automaten.

An einem Lernvorgang sind drei Parteien beteiligt. Der aktive Teil wird vom *Lerner* ausgefüllt, der initial nichts über das Verhalten des zu lernenden Systems weiß, daraus jedoch ein Modell erstellen soll. Dazu greift er auf zwei *Orakel* zu, die Anfragen über das System wahrheitsgemäß und deterministisch beantworten. Zum Einen ist dies das *Membership-Query-Orakel*, welches Anfragen in Form von Eingabeworten mit einer Sequenz von Ausgabesymbolen beantwortet. Diese werden als *Membership Querys* (MQ) bezeichnet und mit Hilfe eines Data-Mappers in für das SUL passende Ausführungsschritte konkretisiert. Zum Anderen wird ein *Equivalence-Query-Orakel* benötigt, das Unterschiede zwischen dem gelernten Modell und dem realen System auffindet. Existiert ein Unterschied zwischen diesen beiden, so

Tabelle 2.1: Beispiel einer Observation Table mit den Ausgabesymbolen ✓und ✗

		E		
		A	B	C
S	ϵ	✓	✓	✗
	A	✓	✗	✗
	B	✓	✓	✗
$S\Sigma$	C	✓	✓	✗
	AA	✓	✓	✗
	AB	✗	✗	✓
	BC	✗	✗	✗

liefert das Orakel ein Eingabewort als Gegenbeispiel, mit welchem sich das Hypothesemodell dann verfeinern lässt.

Beide Orakeltypen unterscheiden sich deutlich in ihrer Komplexität. Ein MQ-Orakel leitet im einfachsten Fall die Anfrage an das SUL weiter, wertet die Antwort aus und gibt eine entsprechende Rückmeldung an den Lerner. Anders sieht es beim EQ-Orakel aus. Equivalence Querys lassen sich zwar effizient durchführen, wenn das zu lernende System bereits als Modell vorliegt. Dies ist bei einer Black-Box jedoch nicht der Fall, und selbst die Anzahl der internen Zustände ist im Allgemeinen nicht bekannt. Der Equivalenztest ist daher für solche Systeme nicht durchführbar und kann durch eine Approximation mittels Membership Querys erfolgen [SHM11].

Sind alle drei Komponenten einsatzbereit, stellt der Algorithmus zu Anfang systematisch Membership Querys an das MQ-Orakel. Jede dieser Anfragen besteht aus einem Wort $w_i \in \Sigma^*$ von Eingabesymbolen. Als Reaktion darauf erhält der Algorithmus eine Antwort, welche aus einer identisch langen Folge von Ausgabesymbolen besteht. Anhand dieser Kombinationen von Ein- und Ausgaben lässt sich eine Hypothese über das Systemverhalten in Form eines Mealy-Automaten erzeugen.

Da jede Anfrage einer Membership Query wieder im Startzustand q_0 beginnen soll, muss das SUL zuvor vom MQ-Orakel mittels eines sogenannten *Reset* zurückgesetzt werden. Die Forderung nach Determinismus besagt hier, dass ein System nach einem Reset für ein Eingabewort immer dasselbe Ausgabewort erzeugt. Es existieren jedoch auch Ansätze, um mittels der Abstraktion von Alphabeten auch bei anderen Systemen ein deterministisches Verhalten zu erzeugen [HSM11, IHS13a].

Die verwendete Datenstruktur ist eine sogenannte *Observation Table*. Sie besteht aus den beiden Wortmengen S und $S\Sigma$ sowie einer Suffixmenge E. Zu jedem Wort se, wobei $s \in S \cup S\Sigma$ und $e \in E$, wird in dieser Tabelle das letzte Ausgabesymbol der Antwort gespeichert. Tabelle 2.1 zeigt exemplarisch den Aufbau einer solchen.

2.4.4 Optimierung durch Einsatz von Filtern

In erster Linie hängt die benötigte Dauer eines aktiven Lernverfahrens von der Anzahl der notwendigen Anfragen, einer möglichen Parallelisierbarkeit sowie der Antwortgeschwindigkeit des SUL ab. Da letztere kaum beeinflusst werden kann, ist ein sinnvoller Ansatz zur Beschleunigung, die gestellten Anfragen mit Hilfe von domänenspezifischem Wissen zu minimieren oder aber ohne Rückfrage an das SUL direkt zu beantworten. Einige dieser so-

genannten *Filter* wurden bereits in [HNS03] vorgestellt. In dieser Dissertation kommt der Reuse-Filter [BNSH12] zum Einsatz, der neun Jahre später entworfen wurde und seine Stärken vor allem bei den in dieser Arbeit betrachteten Webanwendungen ausspielen kann.

Die hauptsächliche Idee dieses Filters ist, ein Zurücksetzen des Systems möglichst oft zu vermeiden. Solch ein Reset ist jedoch notwendig, damit ein System deterministisch gelernt werden kann, da es zu Beginn jeder Evaluierung eines Wortes in einen vordefinierten Zustand zurückgesetzt werden muss. Abhängig von der Art des zu lernenden Systems kann ein solcher Reset sehr aufwändig sein oder sogar einen Großteil der Gesamtdauer einer Anfrage ausmachen. Bei Webanwendungen ist ein Zurücksetzen etwa durch eine erneute Initialisierung des Dienstes realisierbar, beispielsweise durch Löschen der Datenbank. Dies kann jedoch dazu führen, dass der Reset das parallele Lernen eines Systems verhindert, etwa wenn das Löschen der Datenbank alle derzeit laufenden Anfragen beeinflusst. Weiterhin ist zu bedenken, dass eine vollständige Initialisierung einige Zeit in Anspruch nehmen kann, beispielsweise für das Erstellen von Datenbanktabellen und der Eintragung initialer Werte. In noch schwierigeren Fällen kann vor der ersten Verwendung auch eine automatisierte Aktualisierung notwendig sein, welche abhängig von der Verbindungsgeschwindigkeit zum Internet ist.

Der Reuse-Filter stellt eine deutliche Optimierung für diesen Fall dar. Er ermöglicht die Wiederverwendung bereits existierender Systemzustände, indem er diese speichert und bei späteren Anfragen fortführt. Wurde beispielsweise an ein System, welches die Eingabesymbole A und B erkennt, die Eingabe AB gestellt, so kann dieser Zustand bei einer zukünftigen Abfrage, etwa ABA, wieder aufgegriffen werden. Ohne den Reuse-Filter wäre für beide Anfragen ein Reset notwendig, durch die Wiederverwendung sinkt der Zeitaufwand deutlich.

Mittels weiterer Annahmen, die der Reuse-Filter über reaktive Systeme macht, lässt sich die Effizienz der Lernvorgänge noch weiter steigern. So ist es etwa häufig der Fall, dass fehlgeschlagene Aktionen den Zustand eines Systems nicht verändern. Ist in einer Webanwendung beispielsweise eine Funktion zu einem bestimmten Zeitpunkt (noch) nicht erlaubt, so erhält der ausführende Nutzer eine entsprechende Fehlermeldung. Da die Aktion nicht durchgeführt wurde, hat dies keine Auswirkung auf andere Funktionen. Auf den Reuse-Filter übertragen bedeutet dies, dass eine Anfrage AB, bei der nur das Symbol A erfolgreich ausgeführt werden konnte, so verwendet werden kann, als wäre B nie gestellt worden, beispielsweise für das Wort AA. Zusätzlich lassen sich vor Beginn des Lernvorgangs manuell Symbole festlegen, von denen bekannt ist, dass sie nur lesenden Zugriff auf das SUL ausüben.

Das Problem der Parallelität bei komplett neu initialisierten Systemen kann der Reuse-Filter auf diese Weise jedoch nicht lösen, da die gespeicherten Systemzustände natürlich nur dann verwendet werden können, wenn das zum Zustand gehörende System noch existiert. An dieser Stelle ist es dann notwendig, parallel mehrere Systeme (zum Beispiel auf verschiedenen Rechnern) zu installieren und diese zeitgleich zu lernen. Sofern der vom Reuse-Filter gespeicherte Systemzustand einen Eintrag über das verwendete System erhält, wäre so eine parallele Nutzung möglich, bei welcher der Filter zusätzlich unnötige Resets verhindert.

2.5 Verwendete Softwarekomponenten

Für die Implementierung der kontinuierlichen Qualitätskontrolle kamen mehrere Softwarekomponenten zum Einsatz, die zum Teil am Lehrstuhl für Programmiersysteme entwickelt wurden.

2.5.1 LearnLib

Die LearnLib [RS06, RSBM09, MSHM11] ist ein in Java geschriebenes Framework für maschinelles Automatenlernen, das Implementierungen sowohl von diversen Lernalgorithmen als auch von Äquivalenztests und Filtern bereitstellt, darunter der Algorithmus L_M^* oder der Reuse-Filter. Während die ursprüngliche Version der LearnLib nicht quelloffen angeboten wurde, entstand im Jahr 2013 eine Neuimplementierung, die unter der *GNU Lesser General Public License* (LGPL) veröffentlicht wurde. Sie basiert auf der ebenfalls unter der LGPL veröffentlichten AutomataLib, welche Datenstrukturen für Automaten, Graphen und Transitionssysteme bereitstellt. Alle Experimente in dieser Dissertation wurden mit Hilfe der aktuellen LearnLib entwickelt und durchgeführt.

2.5.2 Selenium

Bei Selenium[4] handelt es sich um ein Test-Framework für die Überprüfung von Webanwendungen. Tests können entweder mit einer Browser-Erweiterung manuell aufgezeichnet und dann beliebig oft abgespielt oder aber programmatisch spezifiziert werden. Dabei werden verschiedene Sprachen unterstützt, unter anderem Java, C# und Python.

Selenium ist nicht nur in der Lage, eine Anwendung mit verschiedenen Browsern wie Mozilla Firefox oder dem Internet Explorer zu verarbeiten, sondern unterstützt zudem HtmlUnit, einen in Java geschriebenen Browser ohne grafische Oberfläche. Mit Hilfe von HtmlUnit ist es möglich, Webseiten aufzurufen und Aktionen durchzuführen, etwa die Auswahl von Bedienelementen oder dem Folgen von Links. Gleichzeitig bietet es eine relativ gute Performance im Gegensatz zu grafischen Browsern, die für das Darstellen einzelner Seiten einige Zeit benötigen. Da bei den hier verwendeten Aktionen das Ergebnis und nicht die Darstellungsqualität der Webseite im Vordergrund steht, ist HtmlUnit die logische Wahl.

2.5.3 Das Java Application Building Center

Das Java Application Building Center (jABC) ist ein Werkzeug zur grafischen Erstellung von Prozessmodellen [SMN+07], das auf eine über 20-jährige Geschichte zurückblicken kann [Nag09]. Der Arbeitsablauf mit dem jABC basiert auf dem Entwicklungsmodell des *Extreme Model-driven Development* (XMDD) [MS12], das die Ansätze der Serviceorientierung sowie des modellgetriebenen Designs vereint und sich dabei auf Anwender ohne Programmierkenntnisse konzentriert [Sul09].

Zusätzlich zum grafischen Editor besteht das jABC aus einem Framework-Modul, welches durch Plug-ins funktional erweitert werden kann [NNL+13]. So prüft etwa der *Local-Checker* [Neu07] die lokalen Eigenschaften einzelner Elemente eines Modells, während der Model-Checker GEAR [BMRS09] eine Analyse des gesamten Modells anhand temporallogischer Formeln erlaubt. Weiterhin existieren Plug-ins, welche domänenspezifische Modelle im jABC ermöglichen. Dazu gehören zum Beispiel der Code-Generator Genesys [Jö13], das Synthesewerkzeug PROPHETS [NLS12] oder auch die Validierung von Firewall-Regeln mit Hilfe simulierten Netzwerkverkehrs [Win11].

Die aktuell in der Entwicklung befindliche Version des Java Application Building Center mit dem Namen *jABC4* erweitert das Framework um *Higher Order*-Prozessmodellierung [NSM13, Neu14], die den flexiblen Austausch von Diensten innerhalb der Modelle ermöglicht (in der Literatur *Laufzeitvariablilität* genannt) und die dynamische Einbindung von

[4] `http://seleniumhq.org`

Diensten. Eine hierarchische Strukturierung der Elemente versetzt dabei jeden Anwender dazu in die Lage, mit genau den Graphen zu arbeiten, die für sein Arbeitsgebiet von Interesse sind.

Auf Ebene der Entwickler sind dies einzelne API-Methoden von Java-Klassen, welche dynamisch als *Service-Independent Building Blocks* (SIBs) in *technische Service-Logik-Graphen (SLGs)* eingebunden werden. Jeder Graph verfügt dabei über eine Schnittstellenbeschreibung, bestehend aus möglichen Ein- und Ausgabewerten, sowie einen Kontext, der den Inhalt der lokalen Variablen speichert. Ein SLG, welcher einen Ganzzahlwert als einelementige Liste zurückgibt, kann beispielsweise mit zwei solcher Variablen erstellt werden: Zuerst wird der gewünschte Wert vom aufrufenden Graphen in die erste gesichert, die zweite übernimmt den Rückgabewert des Konstruktoraufrufs einer Implementierung des Java-Interfaces *List*. Mit einem Aufruf der Methode `add(int)` kann nun der Wert der Liste hinzugefügt und diese anschließend an den aufrufenden Graphen zurückgegeben werden.

Eine mächtige Eigenschaft des Kontextes ist zudem die Möglichkeit, externe Dienste dynamisch zu injizieren. Die Belegung einer Variablen wird dadurch abhängig von den zur Laufzeit verfügbaren Implementierungen, was einen einfachen Austausch von Funktionalität zulässt.

In der Hierarchieebene oberhalb der technischen SLGs arbeiten die Applikationsexperten im Sinne der Serviceorientierung mit den applikationsspezifischen Graphen. Sie bilden diese vorwiegend aus den von den Entwicklern bereitgestellten und einfach zu verwendenden SLGs, haben aber bei Bedarf dennoch Zugriff auf einzelne API-Methoden. Auch diese SLGs verfügen über einen Kontext für lokale Variablen, die verwendet werden können, um den Datenfluss zwischen den einzelnen Komponenten zu modellieren.

3 Kontinuierliches Lernen

In diesem Kapitel wird die Technik einer kontinuierlichen Qualitätskontrolle vorgestellt, die auf dem aktiven Lernen von Automatenmodellen basiert.

3.1 Funktionsweise

Die Entwicklung eines Softwareprojekts ist ein komplexer Prozess, bei dem anfallende Fehler möglichst vor Veröffentlichung des Produkts gefunden werden sollten. Dabei können die in Abschnitt 2.3 beschriebenen Testmethoden helfen, indem bereits während der Entwicklungsphase Testfälle automatisiert ausgeführt werden oder vor der Veröffentlichung einer neuen Version ein umfangreicher, manueller Systemtest anhand eines vorher definierten Kriterienkatalogs durchgeführt wird.

Diese Vorgehensweise ist allgemein anerkannt, birgt jedoch diverse Nachteile. Zum Einen weisen viele Entwickler dem Schreiben von Testfällen eine eher niedrige Priorität zu. Dies kann diverse Gründe haben, etwa permanenter Zeitdruck in der Entwicklung oder die Ansicht, dass ein gerade behobener Fehler in Zukunft nie wieder auftreten wird. Auch ist es möglich, dass der Entwicklungsleiter oder andere Personen weiter oben in der Unternehmenshierarchie die Notwendigkeit für Tests nicht sehen und diese nicht in ihre Zeitplanung aufnehmen. Entwicklungsmodelle wie die testgetriebene Entwicklung (Test-driven Development, TDD) leisten hier Abhilfe, indem beispielsweise *vor* dem Schreiben von neuem Code ein Test geschrieben wird, der das gewünschte Ergebnis abbildet [Bec03].

Zum Anderen spielt der Zeitfaktor eine große Rolle. Testfälle auf Modul-Ebene sind häufig schnell geschrieben und können auf lange Sicht sogar Zeit sparen. Das Ändern eines automatisierten Systemtests, welcher zudem eine hohe Laufzeit von mehreren Stunden aufweisen kann, erfordert aber je nach Design der Testumgebung einen deutlich höheren Aufwand. Ausführliche manuelle Tests vor einer Veröffentlichung können zudem viele personelle Ressourcen binden, die dann an anderer Stelle fehlen. Eine Möglichkeit, diesen Aufwand zu reduzieren, liefert das in Abschnitt 2.3.12 beschriebene *risikorientierte Testen.*

Ein wichtige Funktion einer funktionierenden Qualitätskontrolle ist die Gewissheit für den Nutzer einer Anwendung, eine neuere Version genauso einsetzen zu können wie deren Vorgänger. Hiermit sind jedoch weniger Änderungen an der Oberfläche, wie zum Beispiel eine Umstellung der Menüstruktur, gemeint. Anpassungen in der Bedienung können zwar anfänglich verwirrend sein oder gar die Produktivität der Anwender in der Umgewöhnungszeit senken, wichtiger sind jedoch ein gleichbleibender Funktionsumfang sowie eine vergleichbare Reaktion auf dieselben Eingaben. Sofern die Entwickler einer Anwendung deren Verhalten nicht bewusst geändert haben, sollten dieselben Aktionen eines Nutzers sowohl mit der alten als auch der neuen Version dasselbe Ergebnis liefern.

Die Reaktion eines deterministisch arbeitenden Systems auf äußere Einflüsse lässt sich mit einem Mealy-Automaten (siehe Abschnitt 2.4.2) abbilden. Zu jeder Zeit befindet sich dieser in einem klar definierten Zustand und reagiert auf jede Eingabe mit einer eventuellen Zustandsänderung und der Rückgabe eines Ausgabesymbols. Ein mögliches Ausgabealphabet könnte

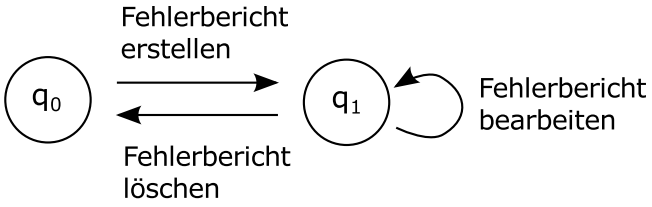

Abbildung 3.1: Modell einer Bug-Tracking-Software mit den drei Funktionen „Erstellen“, „Bearbeiten“ und „Löschen“

in diesem Zusammenhang lediglich aus zwei Symbolen bestehen, welche die Unterscheidung ermöglichen, ob die Aktion erfolgreich durchgeführt wurde oder eine Fehlermeldung produziert hat.

Mit der hier vorgestellten Technik des *Active Continuous Quality Control* [WNS+13] existiert eine Lösung, die (Web-)Anwendungen automatisiert in die oben beschriebenen Modelle überführt und dazu geeignet ist, Änderungen im Systemverhalten zwischen verschiedenen Versionen aufzudecken. Ein Lernalgorithmus erzeugt dazu automatisierte Testfälle, die im Idealfall von allen Systemen mit derselben Ausgabe beantwortet werden. Sollte dies nicht der Fall sein, so lässt sich im Modellvergleich schnell erkennen, an welcher Stelle der Unterschied aufgetreten ist.

3.2 Notwendige Voraussetzungen

Um kontinuierliche Lernvorgänge durchführen zu können, werden für eine zu testende Software eine stabile Abstraktionsebene sowie ein Testtreiber zur Durchführung der Aufrufe benötigt. Damit der Lernvorgang möglichst effizient abläuft, ist zusätzlich eine starke Wiederverwendung der Ergebnisse erforderlich.

3.2.1 Stabile Abstraktionsebene

Ein Softwareprodukt, das aktiv weiterentwickelt wird, verändert sich ständig: Neue Funktionen werden hinzugefügt, alte überarbeitet oder ersetzt. Basiert eine Anwendung auf Bibliotheken oder Frameworks von Dritten, so stehen hier zusätzlich Aktualisierungen an. Keine dieser Änderungen sollte jedoch verhindern, dass der Nutzer Arbeitsschritte, die bisher möglich waren, mit einer neuen Version des Produkts nicht mehr durchführen kann.

Eine Bug-Tracking-Software kann zum Beispiel das Formular zum Erstellen eines neuen Fehlerberichts optisch überarbeiten oder in mehrere Schritte aufteilen. Gleichzeitig kann sich die zur Speicherung der Daten genutzte Datenbankversion geändert haben. All dies sollte jedoch auf den Nutzer keinen Einfluss haben, sofern er weiterhin die Aktion „Einen neuen Fehlerbericht verfassen“ durchführen kann.

Betrachtet man für das Beispiel der Bug-Tracking-Software nur die drei Anwendungsfälle „Erstellen“, „Bearbeiten“ und „Löschen“, so lässt sich ein gewünschtes Verhalten mit dem Modell in Abbildung 3.1 darstellen: Ein einmal erstellter Fehlerbericht lässt sich beliebig oft verändern, bis er schlussendlich gelöscht wird. Auf welche Weise dies geschieht, ist auf dieser Ebene der Modellierung nebensächlich. Wichtig ist, dass jede der genannten Aktionen in jeder zu testenden Version des Softwareprodukts vorhanden ist. Da davon ausgegangen werden kann, dass sich die (abstrakte) Grundfunktionalität einer Software im Laufe ihrer Entwicklung wenig verändert, ist diese Forderung leicht zu erfüllen.

Diese Art der Abstraktion versteckt zudem die konkret transportierten Inhalte vor dem Lernalgorithmus. Für das im Endeffekt erstellte Modell ist es nämlich irrelevant, mit welcher Beschreibung oder welchem Datum ein Fehlerbericht erzeugt wurde [HS04] – wichtig für solch ein verhaltensbasiertes Modell ist, *dass* er erzeugt werden konnte.

3.2.2 Entwicklung eines generischen Testtreibers

Ein generischer Lernalgorithmus stellt selten Anfragen direkt an das zu lernende System, sondern erstellt Worte aus den ihm übergebenen Alphabetsymbolen, welche dann an das SUL gestellt werden. Die Ausführung dieser Symbole sowie die Auswertung der Rückgaben ist Aufgabe eines sogenannten Testtreibers, der somit Lernalgorithmus und SUL verbindet.

Sobald die im letzten Abschnitt beschriebene Abstraktionsebene definiert wurde, muss diese daher für jede zu testende Version des Softwareprodukts an dessen Application Programming Interface (API) angeschlossen werden. Dazu ist ein Adapter notwendig, der die abstrakten Aktionen in konkrete API-Aufrufe im laufenden System umsetzt. Dies ist notwendig, da sich eine einzelnes Symbol aus mehreren einzelnen Aktionen zusammensetzen kann. Im Falle der Erstellung eines Fehlerberichts könnte dies etwa so aussehen:

1. Anmelden am System: `login(username, password);`
2. Wechsel zum gewünschten Unterprojekt: `switchToProject(projectName);`
3. Erstellen des Berichts: `newBugReport(title, version, comment);`
4. Abmelden vom System: `logout();`

Zwischen den einzelnen Versionen eines Produkts kann sich zusätzlich auch die API selbst ändern. So wäre es zum Beispiel denkbar, dass der Aufruf von `newBugReport` ab einer bestimmten Version zusätzlich die Angabe einer Deadline erlaubt:

```
newBugReport(title, version, comment, deadline);
```

Sofern die API aus Stabilitätsgründen die alten Aufrufe noch unterstützt, stellt dies kein Problem dar. Spätestens jedoch, wenn Parameter vom selben Typ eine andere Bedeutung erhalten, muss der Aufruf angepasst werden. Existierende Adapter lassen sich also für mehrere Versionen eines Produkts einsetzen, sofern sich die zugrundeliegende API nicht geändert hat und die verwendete Programmiersprache dies erlaubt.

Eine alternative Herangehensweise an dieses Problem wäre die automatische Erzeugung von Testtreibern durch Analyse der verfügbaren Schnittstellen [MIH+12]. Dieser Ansatz setzt jedoch eine Beschreibung im WSDL-Format voraus, welche das in Kapitel 5 untersuchte OCS nicht bietet. Zudem ist das Verfahren nur für die frühere Version der LearnLib verfügbar, daher kommen im Rahmen dieser Dissertation ausschließlich manuell implementierte Treiber zum Einsatz.

3.2.3 Wiederverwendung gelernter Informationen

Lernvorgänge, die auf reale Systeme zugreifen, sind in der Ausführungsgeschwindigkeit stark begrenzt durch die Reaktionszeit der zu lernenden Systeme. Selbst bei Beseitigung von eventuellen Verzögerungen durch Netzwerkzugriffe – etwa indem man Lerner und zu lernendes System auf demselben Rechner startet – können komplexe Anfragen durchaus länger als

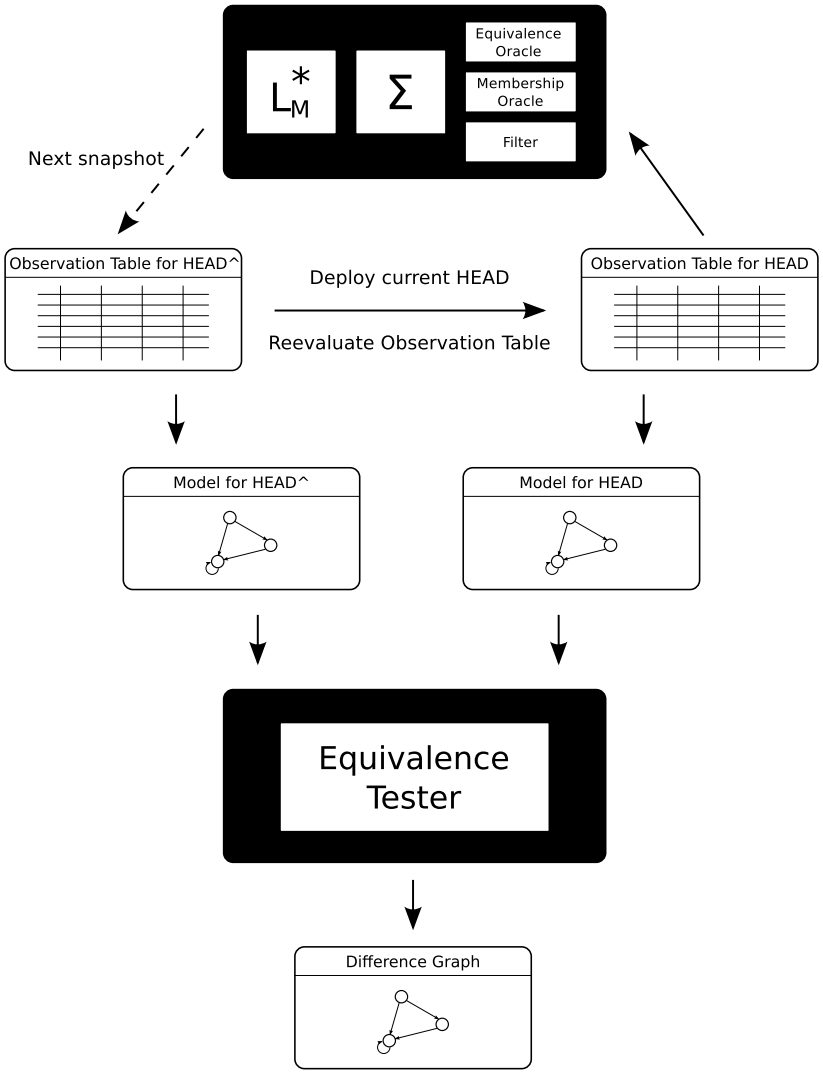

Abbildung 3.2: Wiederverwendung der Observation Table. Quelle: [WNS+13]

eine Sekunde pro Alphabetsymbol dauern. Lösungen wie etwa der Reuse-Filter (siehe Abschnitt 2.4.4) können zwar die Anzahl der Anfragen drastisch reduzieren, dennoch kann ein Lernvorgang mehrere Tage benötigen, um ein erstes Hypothesenmodell zu erzeugen.

Aus diesem Grund ist es wünschenswert, möglichst viel Information aus früheren Lernvorgängen wiederzuverwenden. An dieser Stelle zeigt sich ein Vorteil des gewählten L_M^*-Algorithmus: Die von ihm verwendete Datenstruktur speichert sowohl alle an das zu lernende System gestellten Anfragen als auch deren Ergebnis. Verwendet man die zuvor erzeugte Tabelle als Vorlage, ohne jedoch die evaluierten Ergebnisse zu übernehmen, und stellt die Anfragen anschließend an die neue Version des Systems, erhält man mit vergleichsweise wenig Aufwand einen stabilen und deterministischen Ausgangspunkt für den weiteren Lernvorgang.

Abbildung 3.2.3 zeigt den schematischen Ablauf dieser Arbeitsweise. Durch Anfragen an die aktuelle Version des zu lernenden Systems, hier mit `HEAD` bezeichnet, wird vom Algorithmus L_M^* mit Hilfe des Alphabets Σ ein Modell erstellt. Hierzu wird die Observation Table der zuletzt untersuchten Version (`HEAD^`) erneut ausgewertet und das entstehende Modell mit dem der Vorgängerversion verglichen. Sind die Modelle unterschiedlich, so kann dies ein

Indiz dafür sein, dass sich mit den durchgeführten Änderungen die Nutzung des Systems auf eine Weise verändert hat, die für dessen Anwender problematisch ist.

Durch diese Wiederverwendung ist zudem das wichtige Kriterium des Determinismus erfüllt: Würde das Modell für jede Version von Grund auf neu gelernt, so wäre es möglich, dass durch nichtdeterministische Implementierungen des L^*_M-Algorithmus Hypothesen-Modelle in unterschiedlicher Reihenfolge generiert werden, obwohl sich die untersuchten Systeme in Wahrheit identisch verhalten. Eine erneute Auswertung einer Observation Table sorgt stattdessen dafür, dass exakt die gleichen Anfragen bei beiden Systemen gestellt werden.

Stellt sich nach erneuter Auswertung der Observation Table heraus, dass sich die Systeme identisch verhalten, so ist der Lernvorgang damit noch nicht beendet. Vielmehr können weitere Anfragen an das System gestellt werden, um das Modell weiter zu verfeinern. Findet diese Art des Lernens etwa einmal pro Tag statt, so kann auch nach einer mehrstündigen Neuauswertung der Rest der Zeit darauf verwendet werden, weitere Zustände im Modell zu finden. Auf diese Weise kann das Wissen um das Modell kontinuierlich jeden Tag wachsen. Benötigt der Lernvorgang mehr Zeit, so sind natürlich längere Zeiträume zwischen den einzelnen Durchläufen notwendig.

3.3 Lernen von Aspekten

Die Basis für jegliche an ein zu lernendes System gestellten Anfragen ist das zugrundeliegende Alphabet Σ. Es besteht aus einzelnen Alphabetsymbolen, die vom Lernalgorithmus zu Worten zusammengesetzt werden. Gemeinsam mit den möglichen Reaktionen stellen diese Symbole die feingranularste Ebene dar, die das fertige Modell repräsentieren kann.

Würde man die in Abschnitt 3.2.2 gezeigten API-Aufrufe für einen Bugtracker vervollständigen, könnte man sich folgende Methoden vorstellen:

- Anmelden am System: `login(username, password);`
- Wechsel zum gewünschten Unterprojekt: `switchToProject(projectName);`
- Erstellen des Berichts: `newBugReport(title, version, comment);`
- Bearbeiten eines Berichts: `editBugReport(id, newTitle, newComment);`
- Löschen eines Berichts: `deleteBugReport(id);`
- Abmelden vom System: `logout();`

Mit einem Alphabet, welches für jede dieser Methoden ein eigenes Symbol erhält, ergibt sich ein Modell mit relativ vielen Zuständen. Damit lässt sich dann durch eine visuelle Kontrolle einfach feststellen, ob etwa das Bearbeiten eines Fehlerberichts nur nach erfolgreichem Anmelden möglich ist. Sollen jedoch Fragen wie „Ist das Löschen eines Berichts nach dem Bearbeiten noch möglich?“ beantwortet werden, ist ein Modell wie in Abbildung 3.1 hilfreicher, da es stattdessen abstrakte Anwendungsfälle darstellt.

Aus diesem Grund ist es sinnvoll, mehrere Aktionen zu sogenannten Aspekten zusammenzufassen. Das obige Beispiel bestünde dann nur aus den drei Aspekten *Erstellen*, *Bearbeiten* und *Löschen*. Jeder Aspekt stellt ein Alphabetsymbol dar und fasst einen oder mehrere API-Aufrufe zusammen. Im Falle der Erstellung eines Fehlerberichts wäre dies etwa die Abfolge `login` → `newBugReport` → `logout`. Schlägt hier das Anmelden am System fehl, so gilt die gesamte Ausführung des Symbols als nicht möglich. Ein Test, ob sich bestimmte Aktionen

nur nach Authentifizierung am System durchführen lassen, kann jedoch auch sehr effizient über herkömmliche Integrationstests durchgeführt werden.

Ein weiterer Vorteil liegt in der Zusammenfassung von mehreren Aspekten zu einem. Sollen etwa nur bereits bearbeitete Berichte beachtet werden, so lassen sich Erstellung und Bearbeitung in einem einzelnen Aspekt zusammenfassen. Auch ist es möglich, die Aspekte bereits im Reset des Systems unterzubringen. Auf diese Weise würde der Lernvorgang schon zu Beginn einen erstellten Bericht vorfinden, sodass die späteren Zustände fokussiert betrachtet werden können. In diesem recht kurzen Beispiel kann der Nutzen dieses Vorgehens jedoch nur skizziert werden, eine detaillierte Betrachtung ist den Kapiteln 4 und 5 vorbehalten.

3.4 Inkrementelle Formalisierung

Das automatisierte Lernen mit Hilfe von Aspekten erzeugt für jede untersuchte Version einer Software ein entsprechendes Automatenmodell. Diese Automaten können effizient darauf geprüft werden, ob sie das gleiche Systemverhalten abbilden. Unter der Annahme, dass eines der Modelle fehlerfrei ist, kann auf diese Weise ein Vergleich ohne weitere Voraussetzungen durchgeführt werden.

In vielen Fällen fehlt jedoch ein solches als fehlerfrei und vollständig bekanntes Modell, sodass in der ersten Iteration eine manuelle Überprüfung des Ergebnisses erforderlich ist. Auch sind wiederkehrende Fehler denkbar, die zwar beim ersten Auftreten korrekt erkannt und behoben wurden, im weiteren Entwicklungsverlauf jedoch erneut auftreten und dieses Mal fälschlicherweise als korrektes Verhalten interpretiert werden.

Verhindern lässt sich dies durch eine *inkrementelle Formalisierung* der Anforderungen, die an ein korrektes Modell gestellt werden. Der Begriff wurde bereits im Rahmen der METAFrame-Umgebung verwendet [SMCB96], wo er das schrittweise Hinzufügen von formellen Anforderungen zu einem Entwicklungsprozess beschreibt. Auch dort ist das Modellieren ohne formale Vorbedingungen möglich, deren kontinuierliche Spezifizierung verbessert jedoch die vom System bereitgestellte automatische Unterstützung bei der Anwendungsmodellierung.

Im Kontext der kontinuierlichen Qualitätskontrolle lassen sich bereits bekannte Fehler durch die Prüfung temporallogischer Formeln auffinden, welche auf den Modellen angewendet werden. Wenn es etwa einen Bug gegeben hat, der erlaubte, dass ein gelöschter Bericht weiterhin bearbeitet werden kann, so lässt sich die Abwesenheit dieses Fehlers anhand folgender Formel testen, die in der *Linear temporal logic (LTL)* verfasst ist:

$$\mathbf{G}(Delete \Rightarrow (\neg Edit \ \mathbf{W} \ Create))$$

Ein Grundstock solcher Formeln lässt sich meist aus der Spezifikation der Anwendung ableiten. Existieren beispielsweise in der *Unified Modeling Language (UML)* verfasste Sequenzdiagramme, so können die darin enthaltenen Abläufe auf Ebene der abstrakten Anwendungsfälle in entsprechenden Formalisierungen festgehalten werden. Das kontinuierliche Vorgehen erlaubt es zudem, im weiteren Verlauf entdeckte Fehler beziehungsweise deren entsprechende Formeln inkrementell hinzuzufügen. Hierdurch ist sichergestellt, dass Abweichungen von der Spezifikation in jeder getesteten Version bemerkt werden.

3.5 Ursachenbestimmung

Wird ein Fehler nicht bereits während der Entwicklung durch Modultests aufgedeckt, sondern beispielsweise erst im Produktivbetrieb oder durch die in dieser Arbeit beschriebenen Lernverfahren, so kann eine Lokalisierung der fehlerhaften Stelle im Code recht aufwändig werden. Dies trifft umso mehr auf Fehler zu, bei denen nicht klar ist, welcher Teil des Programms für das Fehlverhalten verantwortlich ist. Ein Beispiel ist etwa eine ungenügende Prüfung von Eingabeparametern, die zu einem inkonsistenten Systemzustand führt. Die Stelle, an welcher dann der Fehler ausgelöst wird, kann sich in einem völlig anderen Teil des Programms befinden.

Um die verantwortliche Stelle einzugrenzen, kann ein Entwickler eine Version im Entwicklungszweig suchen, welche den Fehler nicht aufweist. Von dieser ausgehend wird dann eine Suche gestartet, welche für jede zu testende Version überprüft, ob der Fehler vorliegt. Je nach Komplexität der duchzuführenden Tests kann dies umfangreiche manuelle Arbeit bedeuten, weshalb der Vorgang automatisiert werden sollte. In den seltensten Fällen existieren jedoch die Voraussetzungen für einen solchen Suchvorgang.

An dieser Stelle können die Techniken, welche zum Einsatz der kontinuierlichen Qualitätskontrolle implementiert wurden, wiederverwendet werden, da sie sowohl das automatisierten Erstellen von zu testenden Instanzen des SUL als auch gezielte Funktionstests erlauben. Auf diese Weise ist es nicht nur möglich, effiziente Fehlerüberprüfungen durchzuführen, sondern diese beispielsweise auch auf eine bestimmte Art des Zugriffs (etwa über einen Webbrowser) zu beschränken.

3.6 Validierung von Systemmigrationen

Selbst wenn mit den zuvor genannten Methoden geprüft wurde, dass sich eine neue Version bezüglich der automatisch erzeugten Testfälle im Verhalten nicht von ihrem Vorgänger unterscheidet, so wurde dennoch eine häufige Fehlerquelle nicht untersucht. Es handelt sich um Probleme, die bei der Migration einer existierenden Installation auf eine neue Version entstehen. Häufig sind in einem solchen Fall Änderungen in der Persistenzschicht notwendig, um die Datenbasis an neue oder zukünftige Bedürfnisse anzupassen.

Sollte während der Migration ein Fehler auftreten, so können die Auswirkungen gravierend sein. Ein Komplettverlust wichtiger Daten ist hier der schlimmste anzunehmende Fall, sodass viele Softwareprodukte selbst bei kleinen Änderungen zu einer vorherigen Sicherung der Datenbank raten. Es ist jedoch auch möglich, dass die Migration scheinbar erfolgreich abgeschlossen wurde, Fehler aber erst deutlich später sichtbar werden, und zwar erst dann, wenn wieder auf die zuvor migrierten Daten zugegriffen wird. Je nach Art des Systems können zwischen den beiden Zeitpunkten Wochen oder gar Monate liegen, sodass der Bezug zur Anpassung der Datenbank nicht sofort ersichtlich ist. Spätestens jedoch, wenn sich die Probleme mit neu angelegten Datensätzen nicht reproduzieren lassen, ist eine genaue Überprüfung der Migrationen angeraten.

Grundsätzlich sollte davon ausgegangen werden, dass eine Migration das Systemverhalten nicht beeinflusst. Dies gilt umso mehr, wenn mit den Methoden der kontinuierlichen Qualitätskontrolle gezeigt werden konnte, dass sich beide Versionen bei einer „frischen“ Installation identisch verhalten. Mit dieser Voraussetzung ist es unerheblich, ob eine Aktion vor oder nach der Anpassung durchgeführt wurde: Folgeaktionen sollten in beiden Fällen dasselbe Ergebnis liefern.

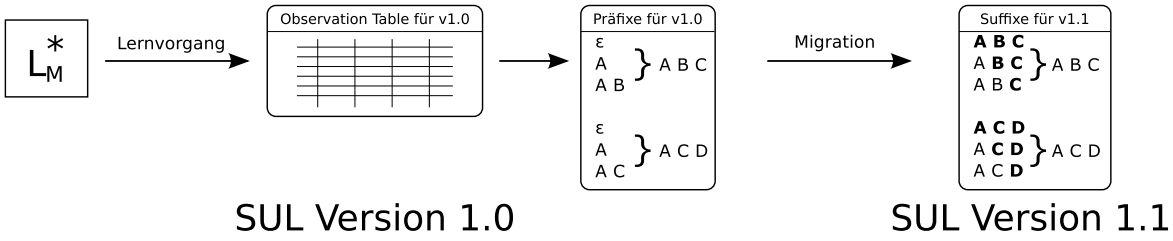

Abbildung 3.3: Vor der Migration eines Systems werden alle möglichen Präfixe evaluiert und das Ergebnis der vollständigen Ausführung durch Ergänzung der Suffixe nach der Migration geprüft.

Um eine solche Art von Test durchzuführen, wird zunächst ein Lernvorgang der zu migrierenden Version mit dem Algorithmus L_M^* durchgeführt, welcher eine Observation-Table erzeugt (siehe Abbildung 3.3). Mit Hilfe der Zustandsmenge S sowie der Suffixmenge E ließe sich nun ein Automatenmodell erzeugen, welches an dieser Stelle jedoch nicht benötigt wird. Stattdessen wird die Menge U mit $\forall s \in S, \forall e \in E : se \in U$ erzeugt. Diese enthält alle im Automatenmodell verwendeten Worte, für welche auch bereits die evaluierten Ausgabesymbole bekannt sind.

Die bisher genannten Methoden vergleichen nur das Ergebnis der vollständigen Worte aus U. Für eine Prüfung aller Migrationen müssen jedoch Teilworte betrachtet werden, deren erster Teil vor und der zweite Teil nach der Migration ausgewertet wird. Dazu wird eine Menge P mit allen Präfixen aus U erzeugt und für jedes $p \in P$ eine Anfrage an das SUL vor der Migration gestellt. Wichtig ist, dass an dieser Stelle der in Abschnitt 2.4.4 beschriebene Reuse-Filter nicht verwendet wird, da zur späteren Kontrolle eine Vielzahl von gleichen Präfixen benötigt wird.

Zusätzlich zum Ausgabewert für jeden Präfix müssen Informationen zum aktuellen Systemzustand gespeichert werden. Diese werden normalerweise vom Lernvorgang verwendet, um wichtige Kontextinformationen zu speichern, um das Ausführen späterer Symbole zu ermöglichen. Im Beispiel des Bugtrackers kann ein solcher Zustand beispielsweise die Nummer eines zuvor erzeugten Fehlerberichts enthalten, damit spätere Symbole auf diesem arbeiten können. Auch ein Reset, der das System auf einen vordefinierten Zustand zurücksetzt, kann initiale Informationen zu einem solchen Zustand beisteuern. Wenn er etwa das System nicht komplett zurücksetzt, sondern ein isoliertes Projekt mitsamt den notwendigen Benutzerzugängen im laufenden System anlegt, werden diese von den meisten folgenden Symbolen zur Ausführung benötigt.

Wurden alle Elemente der Menge P evaluiert und die Ausgaben mitsamt den Systemzuständen gesichert, kann das System heruntergefahren und migriert werden. Alle zuvor angelegten Zustände sollten dabei weiterhin ihre Gültigkeit behalten. Anschließend wird das System mit der neuen Version gestartet und die zweite Phase des Migrationstests kann beginnen. Hierbei werden alle Präfixe soweit vervollständigt, bis jeweils das gesamte Wort aus U als Eingabe an das System gestellt wurde.

Enthält die Menge S beispielsweise das Wort (A, B, C), so wurden in der ersten Phase die Präfixe $(\epsilon), (A), (A, B)$ gebildet und ausgeführt. Der erste Präfix (ϵ) wurde dabei nur durch den Reset initialisiert, sodass nach der Migration das vollständige Wort (A, B, C), jedoch ohne Reset ausgeführt wird. Dementsprechend wird der zweite Präfix mit (B, C) und der dritte nur mit dem einzelnen Symbol (C) vervollständigt, jeweils mit dem zuvor generierten Systemzustand. Auf diese Weise wird das Wort durch drei verschiedene Trennungen

dahingehend geprüft, dass eine Migration keinen Einfluss auf das Verhalten hat: Alle drei Ausführungen sollten dasselbe Ergebnis liefern, also dieselbe Abfolge von Ausgabesymbolen wie die zuvorige Ausführung des gesamten Wortes auf dem nicht migrierten System. Tritt an einer Stelle eine Inkonsistenz auf, lässt sich der Fehler anhand der vorliegenden Daten schnell ausfindig machen.

4 Erste Fallstudie: Mantis Bug Tracker

Der Ansatz der kontinuierlichen Qualitätskontrolle wurde im Rahmen dieser Dissertation mit zwei Produkten evaluiert. Ein großer Unterschied zwischen beiden ist der Zugriff auf den Sourcecode: Während der in diesem Kapitel vorgestellte Bugtracker als Open-Source-Software verfügbar ist, verfügt der Autor dieser Dissertation für das in Kapitel 5 verwendete OCS zusätzlich über die Möglichkeit, das System dauerhaft an die Erfordernisse des Verfahrens anzupassen. Im Folgenden liegt der Fokus daher auf der Einbindung eines „fremden“ Systems, das nur in Maßen modifiziert werden sollte.

4.1 Der Mantis Bug Tracker

In den meisten komplexen Softwareprojekten ist irgendwann der Punkt erreicht, an dem anfallende Fehler oder neue Funktionswünsche übersichtlich aufbereitet werden müssen. Mag es für ein kleines, eventuell von nur einer Person entwickelten Projekts noch möglich sein, dies als kurze Notizen im Quellcode zu vermerken, so ist gerade für größere Teams ein dediziertes Werkzeug dafür verantwortlich, Ordnung in bestehende Aufgaben zu bringen oder auch eine Diskussion über diese zu ermöglichen. Auch wenn der Anwender der Software selbst Fehler melden und über den Fortschritt auf dem Laufenden gehalten werden soll, ist der Einsatz einer Bug-Tracking-Software (auch *Bugtracker* genannt) sinnvoll.

Aktuelle Bugtracker leisten dabei mehr, als nur eine durchsuchbare Datenbank für bestehende Fehler oder Funktionswünsche anzubieten. Sie dienen zum Beispiel der Planung der Releasezyklen, indem Bugs in Schweregrade einsortiert und einem zukünftigen Release zugeordnet werden. So ist auf einen Blick zu erkennen, welche Fehler als nächstes zu beheben sind oder auch in welcher Version ein Fehler behoben wurde. Projektleiter sind so in der Lage, die Bugs einzelnen Entwicklern gezielt zuzuweisen und so für eine ausgewogene Arbeitsverteilung zu sorgen.

Eine Volltextsuche ermöglicht das Finden von hilfreichen Informationen, etwa wenn ein bereits behobener Fehler erneut auftritt. Diese Suche schließt auch die Diskussionen der Entwickler ein, die Entscheidungsprozesse selbst nach Jahren noch klar abbilden können. Benachrichtigungen per E-Mail weisen die Entwickler automatisch auf Änderungen in den ihnen zugewiesenen Fehlern hin, sodass regelmäßiges Durchsehen aller Fehlerberichte unnötig ist.

Die englische Wikipedia listet insgesamt über 50 verschiedene Produkte für diese Aufgabe,[1] jedoch werden nicht alle davon aktiv weiterentwickelt. Als bekannteste Vertreter dürften die Systeme *BugZilla* (Mozilla), *JIRA* (Atlassian), *Launchpad* (Canonical), *Mantis*, *Redmine*, der *Team Foundation Server* (Microsoft) und *YouTrack* (JetBrains) gelten. Stellvertretend für die Klasse von Bugtrackern wurde in dieser Dissertation die Software *Mantis* untersucht, da sie sowohl quelloffen als auch plattformunabhängig ist. Zusätzlich existieren für dieses System bereits Erfahrungen im Zusammenhang mit aktivem Automatenlernen [RSM08, RMSM08]. Mit entsprechenden Modifikationen lässt sich das Verfahren jedoch auch auf die meisten

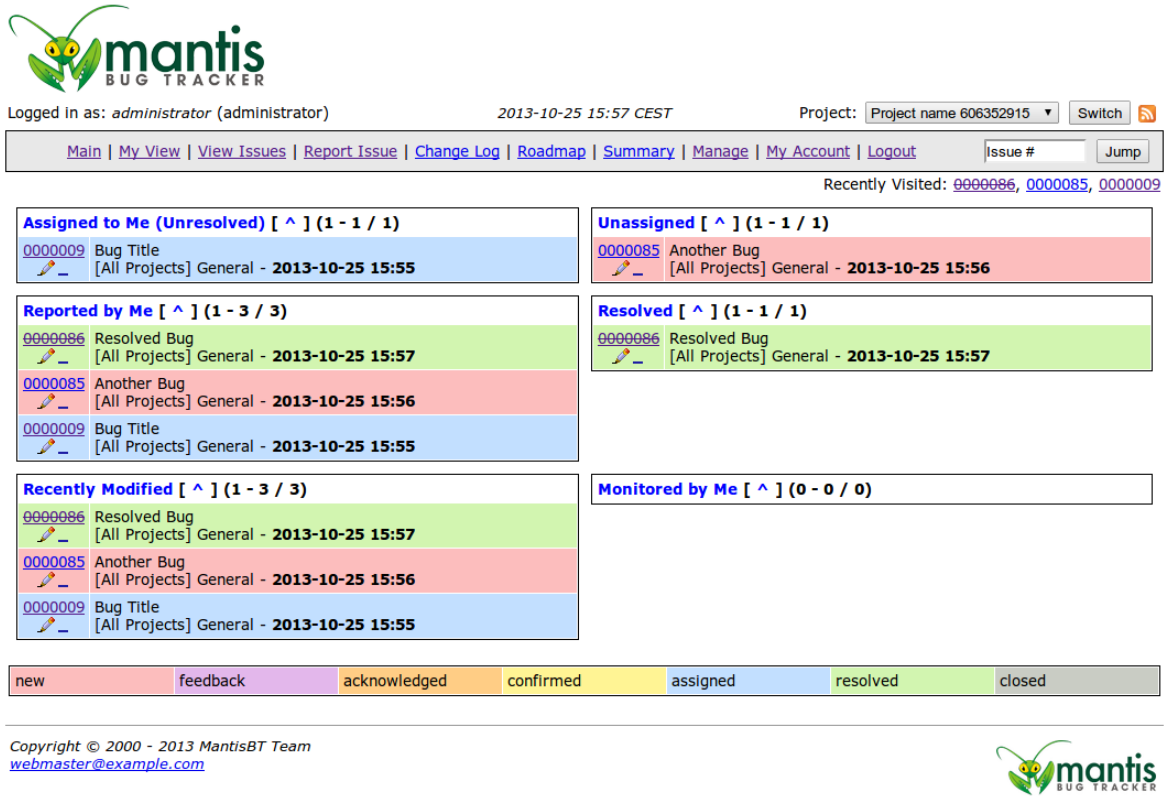

Abbildung 4.1: Weboberfläche des Bugtrackers *Mantis*

anderen Systeme anwenden.

Mantis ist ein in der Programmiersprache PHP geschriebener Bugtracker. Sowohl Name als auch Logo der Software weisen auf die Fangschrecke *Gottesanbeterin* hin, die wohl stellvertretend für alle Arten von Insekten (englisch: *Bug*) stehen soll. Zur Installation sind lediglich ein Web-Server und PHP notwendig, was es erlaubt, die Software auf vielen unterschiedlichen Plattformen zu betreiben. Der Datenbestand wird in einer SQL-Datenbank (wahlweise MySQL, MS SQL oder PostgreSQL) vorgehalten. Abbildung 4.1 zeigt einen Screenshot der Browser-Oberfläche, die mit HTML und CSS realisiert wurde und auch einige Komfortfunktionen mit JavaScript realisiert.

Eine Installation kann mehrere Projekte verwalten, für welche Bugs und Feature-Requests getrennt angelegt werden können. Diese Projekte wiederum können in Unterprojekte aufgeteilt werden. Auf diese Weise ist es möglich, eine einzige Mantis-Installation auch bei einer großen Anzahl von Projekten zu verwenden. Über die Nutzerverwaltung lässt sich jedem Nutzer eine globale sowie für jedes Projekt jeweils eine projektspezifische Rolle (etwa Reporter, Entwickler oder Manager) zuweisen. Die Benachrichtigung der Nutzer läuft primär per E-Mail, es gibt jedoch auch Plug-ins für die direkte Integration in die Entwicklungsumgebung des Entwicklers oder abonnierbare RSS-Feeds.

Jeder eingestellte Fehlerbericht durchläuft einen konfigurierbaren Arbeitsablauf. Neu eingestellte Bugs können bestätigt und an den zuständigen Entwickler zugewiesen werden. Sind

[1] `https://en.wikipedia.org/wiki/Comparison_of_issue-tracking_systems`

Rückfragen an den Ersteller notwendig, so lässt sich dieses mit einer entsprechenden Funktion einfordern, andernfalls kann der Fehler als reproduzierbar bestätigt werden. Wurde der Fehler behoben, kann dies mit einem Kommentar, beispielsweise der nächsten Version, welche den Fehler nicht mehr enthält, als *gelöst* markiert werden. Sollte der Bug bereits in anderer Form im System auftauchen, kann er nicht reproduziert werden oder soll er aus anderen Gründen nicht weiter bearbeitet werden, ist auch ein vorzeitiges Schließen möglich.

Für Projektleiter bietet Mantis eine Reihe von Funktionen, um den Fortschritt einzelner Versionen im Auge zu behalten. So können einzelne Bugs der jeweiligen Veröffentlichung zugewiesen werden und eine Roadmap zeigt übersichtlich die noch ausstehenden Bugfixes. Aus dieser lässt sich dann nach der Herausgabe einer neuen Version bequem ein sogenanntes *Changelog* erstellen, welches die Änderungen gegenüber der vorigen Version dokumentiert. Zusätzlich kann eine von drei möglichen Wiki-Engines benutzt werden, um weiterführende Dokumentation für das Projekt zu erstellen.

Über Plug-ins ist es zudem möglich, den Bugtracker um zusätzliche Funktionen zu erweitern. Beispielhaft sei hier *SourceIntegration* genannt, welches das verwendete Versionskontrollsystem direkt in Mantis einbindet. Auf diese Weise lassen sich Verbindungen zwischen Fehlerbericht und der verursachenden Stelle im Quellcode einfacher herstellen.

Mantis ist unter der Open-Source-Lizenz GPL veröffentlicht, der Sourcecode ist auf GitHub verfügbar.[2] Zusätzlich wird der Hosting-Dienst *MantisHub* angeboten, der gegen eine monatliche Gebühr eine Mantis-Installation bereitstellt.

4.2 Stabile Abstraktionsebene

Um das Verhalten mehrerer Mantis-Versionen vergleichen zu können, wird eine Abstraktion benötigt, die verschiedene Vorgänge immer gleich abbilden kann. Daraus ergeben sich eine Reihe von Symbolen, die nicht nur in Mantis möglich sind, sondern die Funktionalität der meisten auf dem Markt befindlichen Bugtracker abdecken. Dies sind im Einzelnen:

NB – New Bugreport Erstellt einen neuen Fehlerbericht.

EB – Edit Bugreport Bearbeitet die Beschreibung eines Fehlerberichts.

ACK – Acknowledge Bugreport Rückmeldung, dass der Fehlerbericht gesehen wurde.

AB – Assign Bugreport Weist einen Fehlerbericht einem Entwickler zu.

UB – Unassign Bugreport Hebt die Zuweisung für einen Fehlerbericht wieder auf.

RF – Request Feedback Der Entwickler erfragt weitere Informationen.

CFG – Confirm Bugreport Der Entwickler bestätigt das Problem.

RB – Resolve Bugreport Der Fehlerbericht wird als erledigt markiert.

CB – Close Bugreport Der Fehlerbericht wird geschlossen.

Jedes dieser Symbole ruft mindestens eine Methode des im nächsten Abschnitt beschriebenen `MantisApiAdapter` auf, der die Zustandsänderung im Bugtracker durchführt.

[2] `https://github.com/mantisbt`

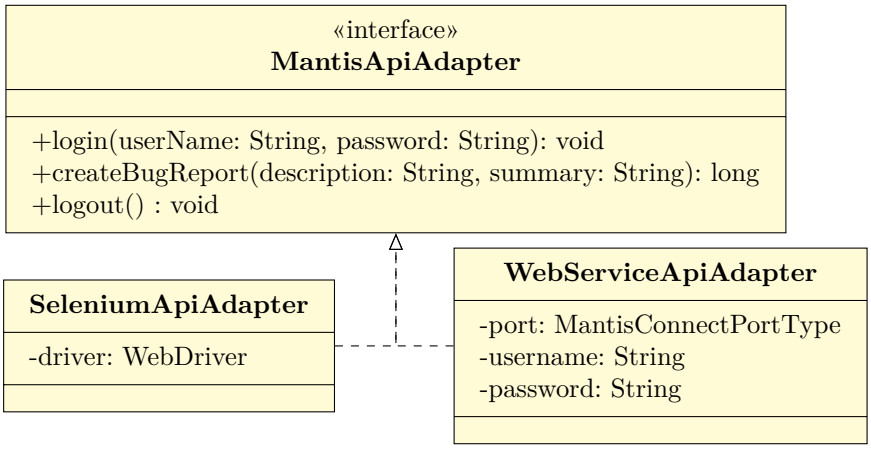

Abbildung 4.2: Mantis-Installationen können sowohl durch einen Selenium-Testtreiber gelernt werden, als auch über Webservices.

4.3 Entwicklung eines generischen Testtreibers

Mantis kann in der Standardkonfiguration nicht nur über eine Weboberfläche gesteuert werden, sondern verfügt auch über eine Webservice-Schnittstelle. Es existieren daher zwei mögliche Wege, das Verhalten einer Mantis-Installation über maschinelles Lernen zu ermitteln. Um diese vergleichbar zu halten, wurde ein Interface `MantisApiAdapter` definiert. Dieses wird, wie in Abbildung 4.2 dargestellt, von den beiden Klassen `SeleniumApiAdapter` und `WebServiceApiAdapter` implementiert, welche im Folgenden vorgestellt werden.

4.3.1 Selenium-Adapter

Da es sich bei Mantis um eine Webanwendung handelt, liegt es nahe, die einzelnen Aktionen direkt in der Weboberfläche vorzunehmen. Für diesen Zweck kam das in Abschnitt 2.5.2 beschriebene Test-Framework *Selenium* zum Einsatz.

Wie in Abbildung 4.2 ersichtlich ist, verfügt der `SeleniumApiAdapter` über einen internen Zustand in Form eines Objekts vom Typ `WebDriver`. Diesen kann man als Browser-Sitzung verstehen: Nachdem sich ein Nutzer über die Funktion `login` des Adapters angemeldet hat, verfügt der interne Zustand über alle notwendigen Informationen, etwa gespeicherte Cookies mit Anmeldedaten, um nachfolgende Aktionen in derselben HTTP-Sitzung ausführen zu können.

Listing 4.1 zeigt exemplarisch das Vorgehen beim Zuweisen eines Entwicklers zu einem Fehlerbericht. Nachdem die Seite des Fehlerberichts geladen ist, wird ein Auswahlfeld, welches die bekannten Nutzer des Bugtrackers auflistet und den Namen `handler_id` besitzt, gesucht und dort der Name des Entwicklers ausgewählt. Nach einem Klick auf den Button mit der Beschriftung „Assign to:“ ist die Methode beendet und der Entwickler zugewiesen.

4.3.2 Webservice-Adapter

Das Projekt MantisConnect[3] stellt einen Webservice für Mantis bereit, über den externe Programme auf den Bugtracker zugreifen können. Weiterhin bietet das Projekt Bibliotheken

[3] `http://www.futureware.biz/mantisconnect/`

Listing 4.1: Zuweisen eines Entwicklers über die Weboberfläche mit Selenium

```
@Override
public void assignDeveloperToBugreport(String userName, long reportId) {
        openBugPage(reportId);

        Select assignField =
                new Select(driver.findElement(By.name("handler_id")));
        assignField.selectByVisibleText(userName);

        By byXpath = By.xpath("//form/input[@value='Assign To:']");
        driver.findElement(byXpath).click();
}
```

Listing 4.2: Editieren eines Fehlerberichts über den Webservice

```
@Override
public void editBugReport(long id, String summary, String description) {
        BigInteger bugId = BigInteger.valueOf(id);
        IssueData issue = port.mc_issue_get(username, password, bugId);
        issue.setSummary(summary);
        issue.setDescription(description);
        port.mc_issue_update(username, password, bugId, issue);
}
```

in mehreren Programmiersprachen an, darunter C# und Java. Zwar ist der Einsatz dieser Bibliotheken an kostenpflichtige Lizenzen geknüpft, der Webservice an sich ist jedoch seit Version 1.1.0a4 in Mantis integriert und steht unter einer Open-Source-Lizenz. Er kann über das standardisierte Netzwerkprotokoll SOAP angesprochen werden, was einen Einsatz auch ohne die Bibliotheken von MantisConnect ermöglicht.

Der Webservice wird derzeit unter anderem von Plug-Ins für Entwicklungsumgebungen wie Eclipse oder IntelliJ IDEA verwendet, um Entwicklern die Möglichkeit zu geben, Bugs bearbeiten zu können, ohne zwischen Programmen wechseln zu müssen. Auch existiert ein Task für das .NET-Build-Tool NAnt, welcher automatisiert Bugs erstellen kann, falls bei der Kompilierung eines Programms ein Fehler auftritt.

In Analogie zum `WebDriver` des Selenium-Adapters wird der interne Zustand durch ein Objekt vom Typ `MantisConnectPortType` dargestellt. Dieser enthält jedoch keine Zugangsdaten zur aktuellen Sitzung, da der Mantis-Webservice zustandslos ist und sowohl Nutzername als auch Passwort bei jeder Anfrage übertragen werden. Dementsprechend werden diese beiden Informationen beim Aufruf der `login`-Methode im Adapter gespeichert.

Listing 4.2 zeigt, wie mit Hilfe des Webservice ein Fehlerbericht editiert wird. Der `port` verfügt über die Methode `mc_issue_get`, welche vom Webservice mit genau diesem Namen bereitgestellt wird und ein Objekt vom Typ `IssueData` zurückliefert. Ein solches Objekt enthält alle relevanten Daten eines Fehlerberichts, etwa die Beschreibung, das Datum der letzten Änderung oder den zugewiesenen Entwickler. Nach Anpassung der Werte kann dieses Objekt mit der Methode `mc_issue_update` wieder an den Bugtracker gesendet werden.

4.4 Ablauf eines Lernvorgangs

Damit der Lernalgorithmus von einer bestimmten Mantis-Version ein Modell erzeugen kann, wird eine funktionierende Instanz benötigt. Dieser Abschnitt beschreibt zunächst, wie eine solche mittels Modifikation des Quellcodes sowie Installation von Datenbank und Webserver erzeugt wird. Weiterhin wird der Ablauf eines Lernvorgangs, vor allem in Hinblick auf den Datenfluss, erläutert.

Zugriff auf spezifische Versionen

Um eine Mantis-Instanz zu installieren, sind Installations-Dateien in der jeweiligen Version notwendig. Dafür wäre es beispielsweise möglich, die von den Entwicklern bereitgestellten Release-Archive[4] zu verwenden, allerdings sind diese auf Releases beschränkt und erlauben keinen Zugriff auf Zwischenversionen. Eine Alternative stellen die *Nightly Builds*[5] dar, welche den Entwicklungsstand tagesaktuell abbilden. Hier existiert jedoch auch keine Möglichkeit, gezielt einzelne Commits, die am selben Tag entstanden sind, herunterzuladen. Zudem scheint es nicht vorgesehen zu sein, auf ältere Stände zugreifen zu können.

Die einfachste Lösung für dieses Problem ist der direkte Zugriff auf das öffentliche Repository der Entwickler. Da es sich bei Mantis um eine Open-Source-Anwendung handelt, ist der Quellcode öffentlich einsehbar, in diesem Fall über den Hosting-Dienst GitHub. Eine Kompilierung des Quellcodes ist zudem dank der Wahl von PHP als Programmiersprache ebenfalls nicht notwendig, sodass die zuvor genannten Release-Archive kaum vom Stand im Repository abweichen sollten.

Aus diesem Grund lässt sich der Quellcode aller Versionen über das Versionskontrollsystem Git [Cha09] herunterladen. Bei Git handelt es sich um ein dezentrales Versionskontrollsystem (Distributed Version Control System, DVCS), sodass dabei eine lokale Kopie des gesamten Repositorys erstellt wird [AS09]. Ein weiterer Zugriff auf das offizielle Repository ist danach nicht mehr notwendig, was die Abhängigkeit von Netzwerkverbindungen während der Lernvorgänge senkt.

Anpassung einer Version an den Lernvorgang

Die Standard-Installation von Mantis eignet sich eher schlecht für automatische Lernverfahren. Einige Hindernisse lassen sich zwar durch entsprechende Konfiguration beseitigen, allerdings existieren auch solche, die eine Anpassung des Quelltextes erfordern. Im Einzelnen muss bei jeder Version folgendes eingerichtet werden:

- Standardmäßig wird eine Wartezeit auf der Weboberfläche verwendet, die eine Weiterleitung nach einer Aktion verzögert. Zum schnelleren Testen mit Selenium wurde diese über die Konfiguration entfernt.
- Beim Anlegen eines Benutzers erhält dieser eine E-Mail mit seinem Passwort. Damit an dieser Stelle keine Verzögerung auftritt, wurde das Senden per E-Mail deaktiviert.
- Aufgrund der deaktivierten Registrierungs-E-Mails erhalten neue Nutzer jetzt automatisch ein leeres Passwort. Da die SOAP-Schnittstelle im Gegensatz zur Weboberfläche keine leeren Passwörter erlaubt, wurde diese Prüfung deaktiviert.
- Neuere PHP-Versionen verursachten Probleme mit älteren Mantis-Versionen.[6] Damit

[4] `http://sourceforge.net/projects/mantisbt/files/mantis-stable/`
[5] `http://mantisbt.org/builds.php`
[6] `http://www.mantisbt.org/bugs/view.php?id=14157`

dennoch alle Version mit derselben PHP-Installation getestet werden können, wird die entsprechende Lösung für alle zu lernenden Versionen eingebunden.

- Die SOAP-Schnittstelle wurde um diverse Funktionen erweitert, die voraussichtlich auch in zukünftigen Versionen von Mantis vorhanden sein werden.[7] Dazu gehören etwa das Registrieren neuer Benutzer oder die Abfrage aller verfügbaren Projekte.

Datenbank-Einrichtung

Zur Installation der jeweils zu lernenden Mantis-Version ist zudem eine Datenbank notwendig. Da Mantis offiziell mehrere SQL-Dialekte unterstützt, fiel die Wahl auf PostgreSQL, da die hierfür entwickelte Funktionalität ebenfalls für die in Kapitel 5 beschriebene zweite Fallstudie verwendet werden konnte. Eine `PostgresManager` genannte Klasse übernimmt dabei sowohl die Erstellung und Initialisierung als auch Starten und Stoppen der Datenbank.

Dynamisch gestarteter Webserver

Zur Verarbeitung der PHP-Dateien wird ein Webserver benötigt, der über eine Unterstützung dieser Programmiersprache verfügt. Häufig eingesetzte Server wie etwa Apache[8] oder der schlankere lighttpd[9] erfordern jedoch eine einmalige Konfiguration auf dem verwendeten System. Um den Einrichtungsaufwand zum Ausführen der Tests möglichst klein zu halten, wurde stattdessen der in PHP enthaltene[10] Webserver verwendet, welcher seit PHP 5.4.0 verfügbar ist. Ein Nachteil dieser Lösung ist jedoch, dass dieser Server keine parallelen Anfragen verarbeiten kann.

Initialisierung der Mantis-Installation

Sobald Datenbank und Webserver gestartet sind, kann die Mantis-Installation initialisiert werden. Dieser Vorgang besteht nur aus der Angabe der Verbindungsdetails für die gewählte Datenbank, kann jedoch nicht über den Webservice geschehen. Unabhängig vom gewählten `ApiAdapter` füllt daher eine mit Selenium realisierte Methode die entsprechende vom Webserver gelieferte Seite aus und erzeugt damit eine neue Mantis-Instanz, die noch über keinerlei Projekte verfügt.

Auswertung durch den Lernalgorithmus

Nachdem eine funktionsfähige Mantis-Version installiert und gestartet ist, kann der Lernalgorithmus mit dem Stellen von Membership Querys beginnen. Für jede einzelne Abfrage wird ein Wort erzeugt, das aus einer Auswahl der in Abschnitt 4.2 vorgegebenen Alphabetsymbolen besteht. So wird etwa mit der in Abb. 4.3 angegebenen Abfolge zu Beginn ein neuer Bug erst erstellt, dann bearbeitet und zugewiesen. Da die Symbole von der gewählten Art des Zugriffs abstrahieren, sorgt der jeweils verwendete `ApiAdapter` für eine Ausführung des Symbols, indem er den Zugriff auf den Bugtracker durchführt und das Ergebnis zurückgibt.

Wichtig hierbei ist, dass für jedes Wort ein Reset durchgeführt wird, damit das Verhalten deterministisch ist. Da ein Zurücksetzen der Datenbank zu aufwändig wäre, erzeugt jede Anfrage daher automatisch ein neues Projekt im Bugtracker mit entsprechenden Nutzern.

[7] `http://www.mantisbt.org/bugs/view.php?id=10349`
[8] `http://httpd.apache.org/`
[9] `http://www.lighttpd.net/`
[10] `http://www.php.net/manual/de/features.commandline.webserver.php`

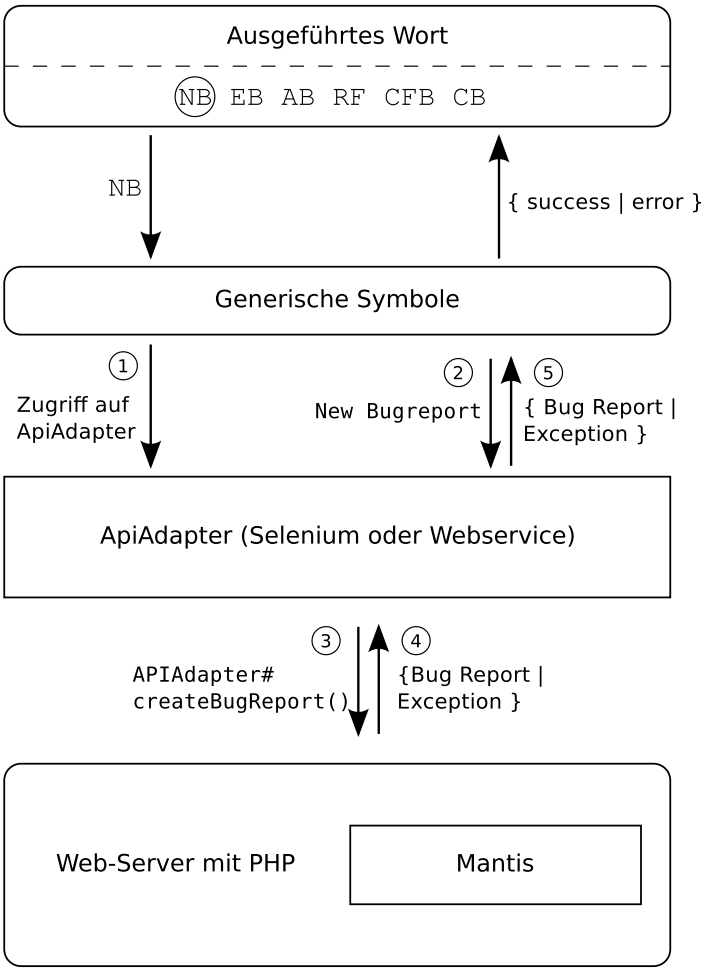

Abbildung 4.3: Auswertung eines vom Lernalgorithmus an Mantis gestellten Wortes mittels eines `ApiAdapter`s

Listing 4.3: Überprüfung der SOAP-API auf leere Passwörter. Quelle: Mantis 1.2.15

```
if( is_blank( $p_password ) ) {
    # require password for authenticated access
    return false;
}
```

4.5 Verifikation von Systemmigrationen

Weitreichende Änderungen an der Struktur der von Mantis verwendeten Datenbank geschehen nur bei größeren Versionssprüngen, also beispielsweise von Version 1.1 auf 1.2. Dennoch sieht der Aktualisierungsprozess vor, dass auch bei Minor-Releases, etwa beim Wechsel von 1.2.16 auf 1.2.17 das Werkzeug zur Migration aufgerufen wird.

Das Aktualisieren auf eine aktuellere Mantis-Version verläuft ähnlich wie die Installation einer neuen Instanz. Nachdem der Bugtracker in einen Offline-Modus geschaltet wurde, der jegliche Interaktion durch andere Nutzer blockiert, kann der Administrator die neuen Dateien auf Dateisystemebene kopieren. Anschließend ruft er dieselbe Seite auf, welche auch zur Installation verwendet wird, und startet hier die Migration.

Zum automatischen Testen der Aktualisierung, wie sie in Abschnitt 3.6 beschrieben wurde, ist es daher nach dem Erzeugen der Präfixe ausreichend, die neuen Dateien aus der Versionskontrolle zu kopieren und die Installations-Seite mittels eines Selenium-Kommandos aufzurufen und dort die Migrationsprozedur zu starten. Die weitere Auswertung der Suffixe auf der aktualisierten Version verwendet dann die zuvor generierten Systemzustände. Diese enthalten unter anderem das vom Reset erstellte Projekt, auf dem die weiteren Symbole ausgeführt werden, sowie Informationen über einen eventuell erstellten Bugreport.

4.6 Ergebnisse

Im Idealfall sollte jedes erzeugte Modell einer Mantis-Version mit der Vorgängerversion übereinstimmen. Zusätzlich ist ein identisches Verhalten von Weboberfläche und Webservice-API wünschenswert. Jedoch wurden bereits bei der Entwicklung der notwendigen Adapter Unterschiede aufgedeckt, die in diesem Abschnitt erläutert werden.

4.6.1 Erkenntnisse während der Entwicklung

Als erstes Hindernis stellte sich die Erzeugung von Nutzerkonten heraus, die für den Reset des Lernalgoithmus benötigt wird. Bei jedem Aufruf des Reset wird ein neues Projekt mit dazugehörigem Konto für einen Entwickler erzeugt. Allerdings erlaubt es Mantis dem Administrator nicht, ein Passwort für neue Nutzer festzulegen. Stattdessen werden Bestätigungs-E-Mails versendet, mit denen der Nutzer die Möglichkeit hat, ein eigenes Passwort zu vergeben. Was aus sicherheitstechnischer Sicht eine sinnvolle Vorgehensweise ist, eignet sich jedoch leider nur schlecht für automatisierte Tests. Aus diesem Grund wurde das Versenden von E-Mails per Konfigurationsparameter unterbunden.

Mit dieser Änderung erzeugt Mantis automatisch Nutzerkonten, deren Passwort leer ist. Einem neuen Nutzer ist es danach also möglich, sich an der Weboberfläche anzumelden und sein Passwort zu ändern. Diese Vorgehensweise funktioniert auch mit dem Selenium-Adapter auf Anhieb, allerdings weigert sich die SOAP-Schnittstelle, leere Passwörter anzuerkennen. Dass dies kein Versehen ist, zeigt das in Listing 4.3 gezeigte Codefragment.

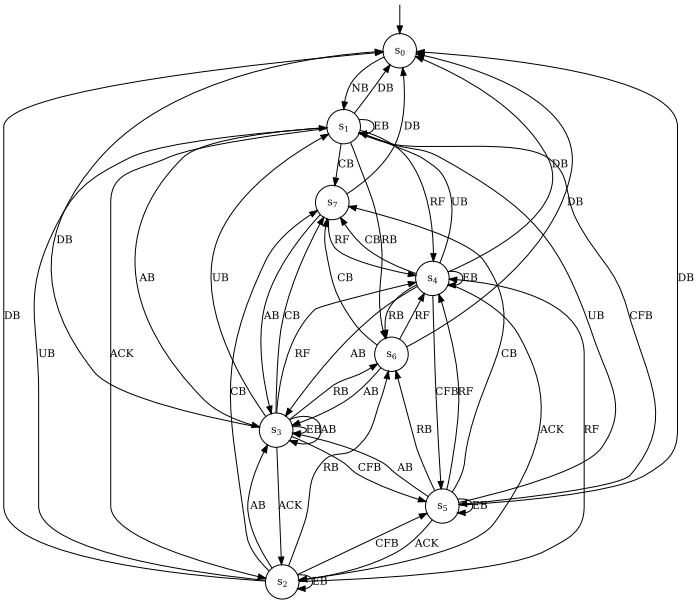

Abbildung 4.4: Gelerntes Modell für Mantis 1.2.17 mit Zugriff über das Browser-Interface

4.6.2 Vergleich zwischen Weboberfläche und Webservice

Die erste Untersuchung bezog sich auf die aktuelle Mantis-Version (derzeit 1.2.17) und prüfte, ob Unterschiede zwischen den Zugriffsarten bestehen, also ob die Webservice-API etwa Aktionen erlaubt, die im Browser-Interface nicht möglich sind oder umgekehrt. Für beide Versionen wurde daher jeweils ein Modell erzeugt. Abbildung 4.4 zeigt die Version der Browseroberfläche, Abbildung 4.5 die des Webservice.

Dass sich die Modelle unterscheiden, ist schon aufgrund der Anzahl der Zustände offensichtlich. Das Modell, welches den Webservice repräsentiert, verfügt mit sieben Zuständen über einen weniger als sein Pendant. Der Grund hierfür wird ersichtlich, wenn man sich den Übergang vom Zustand s_5 zu s_6 in Abbildung 4.5 ansieht: Es gibt keine Unterscheidung, ob ein Bug nur gelöst (RB) oder auch geschlossen (CB) ist.

Eine Analyse des Quelltexts ergab dazu, dass der Webservice Berechtigungen anders berechnet als die Browser-Oberfläche. Für letztere ist es möglich, zu konfigurieren, für welche Rollen (z.B. Entwickler, Manager, Administrator) es erlaubt ist, einen Bug nach dessen Änderung auf den Status „Gelöst“ noch ändern zu können. Standardmäßig ist dies für Entwickler erlaubt, allerdings wird diese Einstellung von der SOAP-API nicht ausgewertet. Anscheinend handelt es sich hierbei jedoch nicht um die einzige Unterscheidung zwischen den beiden Schnittstellen, was ein bestehender Fehlerbericht [Man10] im Bugtracker für Mantis deutlich macht. Da der im Rahmen dieser Dissertation gefundene Fehler noch nicht bekannt war, wurde er von mir an die Entwickler von Mantis weitergeleitet[11].

4.6.3 Leistungsunterschiede

Aufgrund der Eigenschaften der beiden unterschiedlichen Zugriffsarten auf Mantis ist es nicht verwunderlich, dass die Selenium-Variante deutlich mehr Zeit benötigt, um ein Modell zu

[11] `http://www.mantisbt.org/bugs/view.php?id=16579`

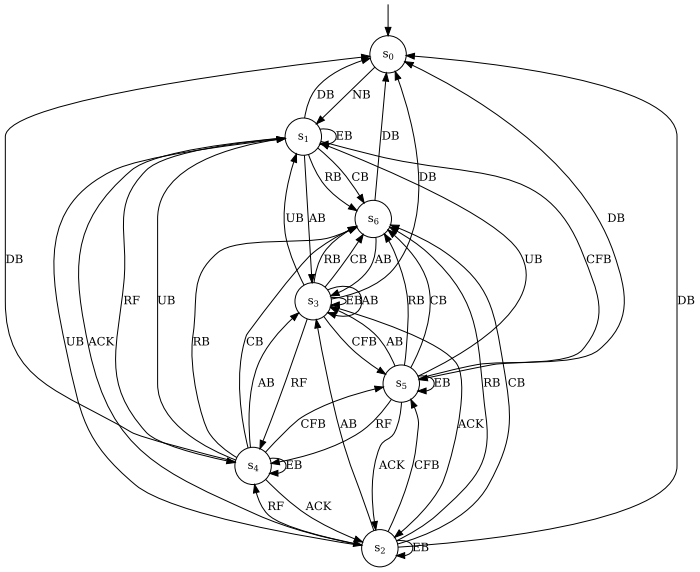

Abbildung 4.5: Gelerntes Modell für Mantis 1.2.17 mit Zugriff über das SOAP-Protokoll

erstellen. Für einen genauen Vergleich wurde die Version 1.2.17 von Mantis mit beiden Adaptern untersucht. Zu beachten ist hierbei, dass für die Selenium-Variante ausschließlich die HtmlUnit-Implementierung verwendet wurde. Diese simuliert einen Browser und lässt sich somit auch automatisiert auf Systemen ohne grafische Oberfläche ausführen. Die Einbindung eines realen Browsers wäre ebenfalls möglich gewesen, jedoch läge die benötigte Dauer eines Lernvorgangs noch weitaus höher.

Als erstes zeigt sich die deutlich höhere Laufzeit des Selenium-Adapters (84 Minuten), welcher mehr als die siebenfache Zeit benötigt wie der Webservice-Adapter (11 Minuten). Dabei sollte jedoch beachtet werden, dass die Lernvorgänge nicht identisch sind, da sich die gelernten Modelle wie in Abschnitt 4.6.2 beschrieben unterscheiden und daher auch die Anzahl der durchgeführten Membership Querys nicht identisch ist. In der Tat ist es so, dass die Webservice-Variante mit 482 Aufrufen 9% weniger Anfragen stellt als der Selenium-Adapter mit 531 Aufrufen. Daraus ergibt sich eine mittlere Geschwindigkeit von 0,73 MQ/s für den Webservice und 0,11 MQ/s für den Zugriff über die Weboberfläche.

Eine detailliertere Aufteilung erhält man durch die Betrachtung der einzeln ausgeführten Symbole. Zwar unterscheidet sich auch hier die Anzahl der Aufrufe, jedoch liefert die mittlere Laufzeit eines Symbols einen guten Näherungswert. Abbildung 4.6 zeigt diese Laufzeiten, welche beim Lernen eines Modells der Version 1.2.17 benötigt wurden. Bei der verwendeten Hardware handelt es sich um einen Intel Core i7 920 mit 2,67 GHz sowie 8 GB Hauptspeicher.

4.6.4 Verbesserung des Qualitätsmanagements

Während sich die oben genannten Fehler auf die aktuelle Version 1.2.17 von Mantis beziehen, erlaubt der Ansatz der kontinuierlichen Qualitätskontrolle auch den Vergleich unterschiedlicher Versionen. In diesem Zusammenhang wurden alle Versionen von Mantis seit Erscheinen der Version 1.2.0 im Februar 2010 durch das System gelernt und die entstandenen Modelle verglichen.

Die auffälligste Änderung zeigte sich für die Version 1.2.11, da hier der Lernvorgang gar

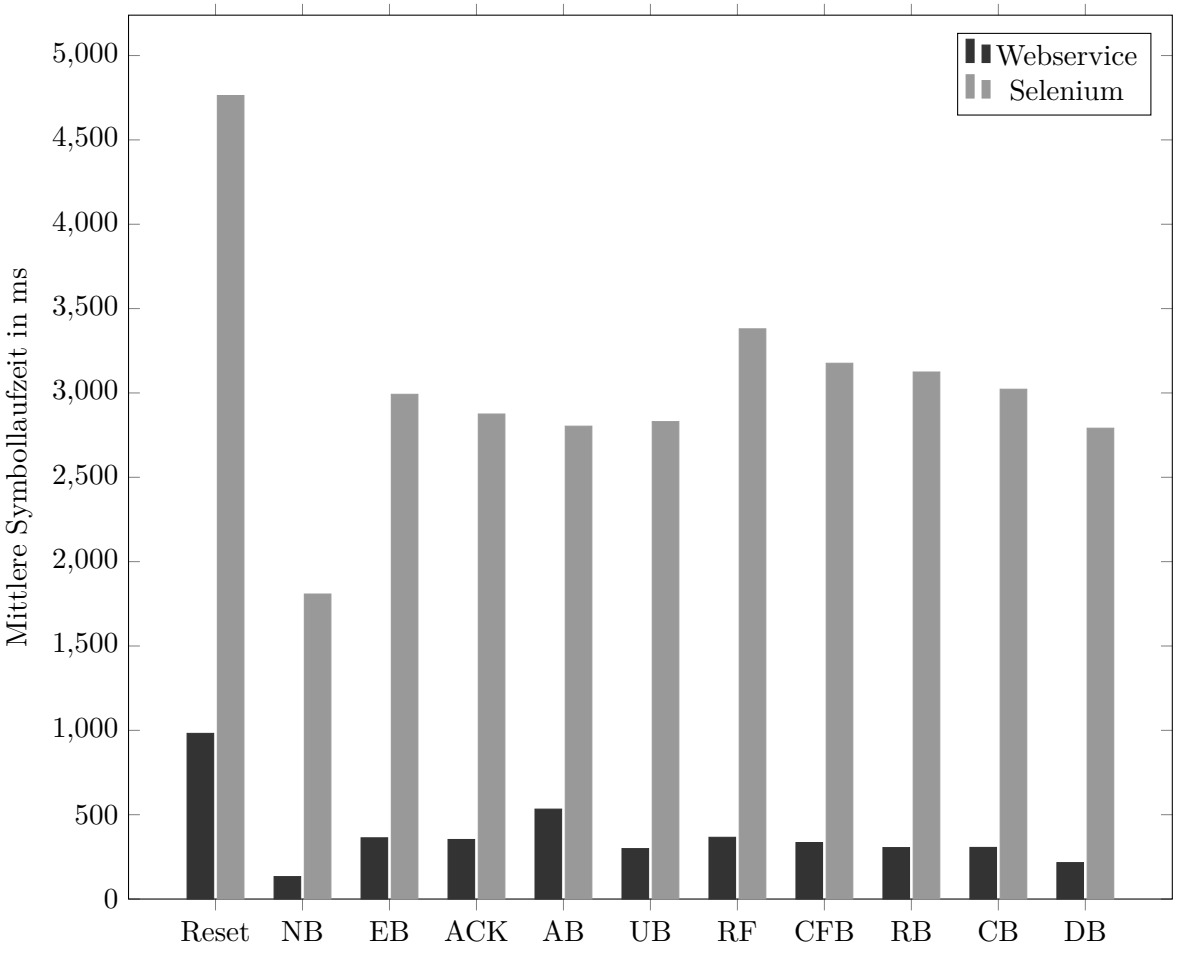

Abbildung 4.6: Durchschnittliche Laufzeit eines Alphabetsymbols je nach ApiAdapter beim Lernen von Mantis 1.2.17

nicht erst starten konnte. Der Grund hierfür liegt an einem Fehler während der Ausführung des Reset-Symbols, der verhindert, dass neue Projekte im Bugtracker erstellt werden können. Dass dieser Fehler für eine veröffentlichte Version unentdeckt blieb, lässt sich über die Voraussetzungen für dessen Auftreten erklären. Zum Einen sind ausschließlich neue Installationen betroffen, ein Upgrade von 1.2.10 auf 1.2.11 verläuft problemlos. Zum Anderen, und hier liegt wohl der eigentliche Grund, wurde für die im Rahmen dieser Dissertation durchgeführten Tests die Datenbank PostgreSQL verwendet. Diese wird von Mantis offiziell unterstützt und ist seit Version 1.2.6 nicht mehr als experimentell gekennzeichnet, während der Installation wird aber MySQL empfohlen. Die Vermutung liegt daher nahe, dass die Tests vor einem Release hauptsächlich mit dieser Datenbank durchgeführt werden.

Der Fehler wurde vier Tage nach Veröffentlichung der Version 1.2.11 im Bugtracker von Mantis gemeldet [Man12], jedoch erst fünf Monate später mit der Nachfolgeversion 1.2.12 behoben. In der Zwischenzeit waren Nutzer gezwungen, über den Upgradeprozess von Version 1.2.10 ausgehend zu arbeiten. Ein Qualitätsmanagement, welches alle unterstützten Datenbanken umfasst, hätte ein solches Problem verhindern können.

4.6.5 Lokalisierung der Fehlerursache

Der in Abschnitt 4.6.4 beschriebene Fehler wurde während der Entwicklung der Version 1.2.11 eingeführt. Anhand der gelernten Modelle lässt sich feststellen, dass die vorherige Veröffentlichung 1.2.10 nicht betroffen war, allerdings ist nicht klar, welcher Commit genau für das Problem verantwortlich ist. Diese Information wäre jedoch hilfreich, um die fehlerhafte Stelle im Code schneller zu finden oder den Autor zu kontaktieren.

Werkzeuge wie das Kommando `git blame` können hier hilfreich sein, indem sie anzeigen, welche Zeile einer Datei von welchem Autor zuletzt bearbeitet wurde und gleich den entsprechenden Commit nennen. Hierzu muss allerdings erst einmal bekannt sein, an welcher Stelle der Fehler ausgelöst wird. Auch ist diese Art der Information bei gelöschten Zeilen nicht verfügbar.

Während der Entwicklung der Version 1.2.11 wurden 67 Commits von neun Autoren in das zentrale Repository hochgeladen. Diese Versionen einzeln zu prüfen wäre ein unverhältnismäßig großer manueller Aufwand. Git bietet auch hier eine Hilfe in Form des Kommandos `git bisect`, welches dem Entwickler mittels einer Binärsuche einzelne Commits präsentiert, welcher dieser dann als *good* oder *bad* markieren kann. Dies erleichtert zwar die Suche, bietet aber keine Lösung für den manuellen Testaufwand jeder einzelnen Version.

Eine Abhilfe für dieses Problem ist durch den existierenden Aufbau für die kontinuierliche Qualitätskontrolle gegeben. So ist es nicht nur möglich, einzelne Versionen von Mantis zu installieren, sondern dort auch automatisierte Tests durchzuführen. Ein minimales Alphabet, bestehend aus nur einem Symbol, ist in diesem Fall völlig ausreichend, da das Problem bereits während des Reset auftritt. Die algorithmisch einfachste Suche besteht dann darin, ausgehend von der als funktionierend bekannten Version, in diesem Fall die veröffentlichte Version 1.2.10, jeden einzelnen Commit zu prüfen.

Nach Beginn des Suchvorgangs lieferte der Algorithmus bereits nach wenigen Minuten das Ergebnis: Der Commit mit der ID `b8d4b50`[12] wurde am 24. Mai 2012, also gut zwei Wochen vor der Veröffentlichung von 1.2.11, erzeugt. Der Autor behebt damit einen Bug, der im Zusammenhang mit der PostgreSQL-Datenbank steht. Zusätzlich führt er auch einen „code cleanup“ durch. Welche dieser Änderungen für den Fehler genau verantwortlich ist, geht aus

[12] `https://github.com/mantisbt/mantisbt/commit/b8d4b5039598248d0b0c78619450c51d4dc98df2`

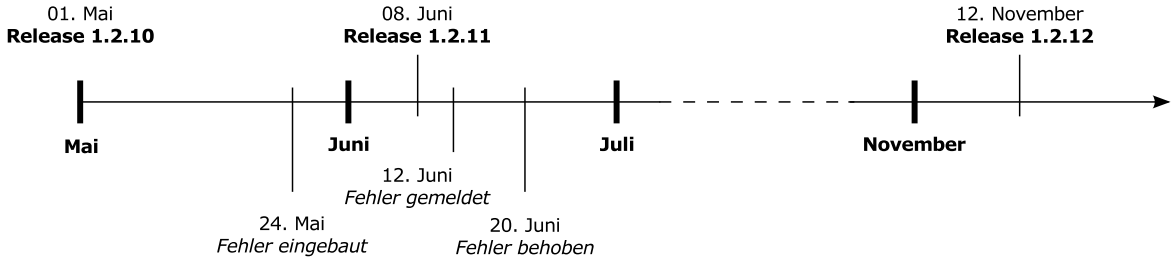

Abbildung 4.7: Zeitlicher Ablauf des Bugs in Mantis, welcher das Erstellen von Projekten mit PostgreSQL-Datenbanken verhinderte

dem Ergebnis zwar nicht hervor, jedoch beschränkt sich der Commit auf effektiv 14 Zeilen geänderten Code an vier Stellen, welche leicht zu überprüfen sind.

Behoben wurde der Fehler schließlich mit Commit 0dfd3f0[13] am 20. Juni 2012, also nur acht Tage nach Bekanntwerden der Probleme (siehe Abb. 4.6.5).

[13] `https://github.com/mantisbt/mantisbt/commit/0dfd3f068d0dee0c95bb35a86b9cfcdebd3cfc47`

5 Zweite Fallstudie: Online Conference Service

Nachdem in Kapitel 4 eine Fallstudie anhand eines öffentlich verfügbaren Open-Source-Projekts vorgestellt wurde, rückt in diesem Kapitel eine deutlich komplexere Anwendung in den Fokus, für die jedoch nicht nur der Quellcode verfügbar war, sondern auch die Anwendung an die Bedürfnisse des Lernvorgangs angepasst werden konnte.

5.1 Der Online Conference Service

Beim Online Conference Service (OCS) handelt es sich um eine Webanwendung, welche die Begutachtung von Manuskripten für wissenschaftliche Konferenzen unterstützt. Nach Erstellung einer Konferenz kann der verantwortliche Nutzer (im System *PC Chair* genannt) sein Programmkommittee (bestehend aus mehreren *PC Membern* oder weiteren PC Chairs) einladen und den Dienst nach seinen Vorstellungen konfigurieren. Wurde der Dienst aktiviert, so ist es Autoren möglich, ihre wissenschaftlichen Arbeiten (*Paper*) ins System zur Begutachtung hochzuladen. Der OCS unterstützt den PC Chair unter anderem bei der Zuweisung der Paper zu einzelnen Personen des Programmkommittees, der Verwaltung eingegangener Begutachtungen oder der Planung des resultierenden Konferenzbandes.

Eine einzelne Instanz des OCS kann mehrere Konferenzen gleichzeitig verwalten, auf einem im Produktivbetrieb befindlichen System sind es derzeit mehr als 110. Dies hat den Vorteil, dass Nutzer des Systems mit einem einzigen Benutzerkonto auf mehreren Konferenzen arbeiten und zwischen diesen wechseln können. Da viele Konferenzen vertrauliche bzw. noch unveröffentlichte Daten verarbeiten, ist hierbei ein ausgeklügeltes Rollen- und Rechte-System notwendig.

Der OCS wird seit 1999 vom Lehrstuhl für Programmiersysteme in Zusammenarbeit mit dem Springer-Verlag entwickelt [KM06]. Im Jahre 2009 wurde die Codebasis unter Verwendung eines modernen objektrelationalen Mappers sowie eines modernen Web-Frameworks für JavaEE 5.0 von Grund auf neu entwickelt. Version 1.0 wurde im April 2010 veröffentlicht und befindet sich seitdem im produktiven Einsatz. Zahlreiche Aktualisierungen haben das System in den letzten Jahren um Funktionen erweitert, verwendete Module wurden aktualisiert oder durch Eigenentwicklungen ersetzt (siehe Tabelle 5.1). So wurde nicht nur das verwendete Web-Framework Tapestry aktualisiert, sondern auch die ursprüngliche Authentifikation über LDAP entfernt, um die Abhängigkeit von externen Diensten zu senken und so auch den Installationsaufwand für neue Instanzen gering zu halten. Weiterhin wurde auch der verwendete Applikationsserver JBoss mit der Zeit durch neue Versionen ersetzt, was ebenfalls größere Änderungen an der Anwendung erforderlich machte.

Trotz all dieser Änderungen hat sich der wesentliche Ablauf einer Konferenz seit der ersten Veröffentlichung nur in Details geändert. So existiert beispielsweise eine Funktion, mit welcher potentielle Reviewer ihr Interesse an einer Begutachtung für bestimmte Paper angeben können. Ursprünglich war dies erst möglich, nachdem ein Paper eingereicht und das entsprechende Dokument hochgeladen wurde. Da es jedoch der Wunsch von Konferenzveranstaltern

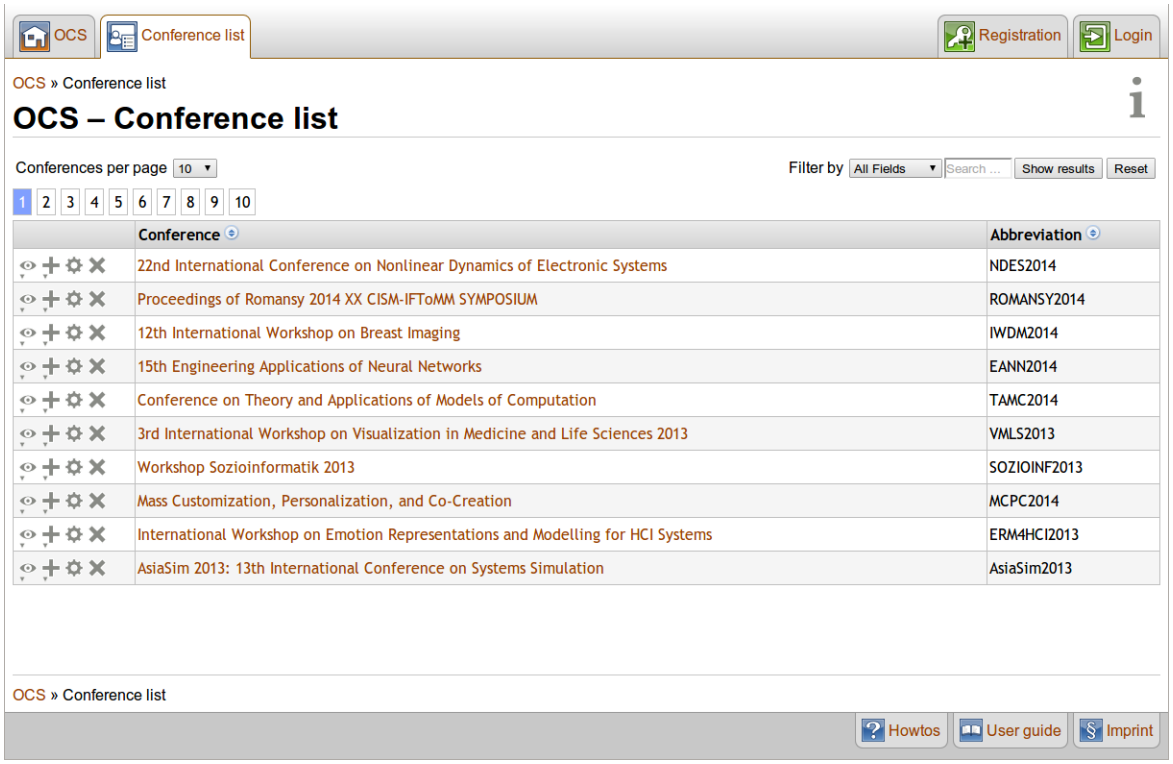

Abbildung 5.1: Liste der Konferenzen im Online Conference Service (OCS)

Tabelle 5.1: Versionsgeschichte des OCS

OCS	Veröffentlichung	JBoss	Tapestry	LDAP	JForum	Cache	LP Solve
1.0.0	April 2010	5.1.0-GA	5.1.0.5	Ja	Ja	Nein	Ja
1.1.0	September 2010	5.1.0-GA	5.1.0.5	Ja	Nein	Nein	Ja
1.2.0	Dezember 2010	5.1.0-GA	5.1.0.5	Ja	Nein	Nein	Ja
1.3.0	Mai 2011	5.1.0-GA	5.2.5	Nein	Nein	Nein	Ja
1.4.0	Juni 2011	5.1.0-GA	5.2.5	Nein	Nein	Nein	Ja
1.5.0	August 2011	5.1.0-GA	5.2.6	Nein	Nein	Nein	Ja
1.6.0	Oktober 2011	5.1.0-GA	5.2.6	Nein	Nein	Nein	Ja
1.7.0	Mai 2012	7.1.1	5.3.3	Nein	Nein	Nein	Ja
1.8.0	August 2012	7.1.1	5.3.4	Nein	Nein	Ja	Nein
1.9.0	Januar 2013	7.1.3	5.3.4	Nein	Nein	Ja	Nein
1.10.0	Feburuar 2014	7.1.3	5.3.4	Nein	Nein	Ja	Nein

war, dieses sogenannte *Bidding* auch anhand der Zusammenfasung (*Abstract*) eines Papers zu ermöglichen, ist ein hochgeladenes Dokument jetzt keine zwingende Voraussetzung mehr.

5.2 Stabile Abstraktionsebene

Während der mehrjährigen Entwicklung des OCS wurde die Codebasis an vielen Stellen an neue Anforderungen angepasst, was natürlich Änderungen in den verwendeten Programmierschnittstellen (APIs) nach sich zog. Da sich jedoch viele Aktionen, etwa das Einreichen eines Papers oder das Zuweisen eines Reviewers, über alle Versionen hin gleich verhalten, ist es möglich, diese als zu testende *Aspekte* zu verwenden, welche unabhängig von der API die gleichen Vorbedingungen haben und das gleiche Ergebnis erzeugen. Um beispielsweise ein Paper einzureichen, muss ein Autor existieren, der die Aktion durchführt und dabei die notwendigen Daten eines Papers (etwa Titel, Stichworte, Abstract und Co-Autoren) angibt. Als Folge davon wird das Paper im System angelegt und kann von anderen Benutzern gesehen werden.

Basierend auf diesen Aspekten wurde im Rahmen der ersten Vorstellung des ACQC-Ansatzes [WNS+13] ein Alphabet erstellt, welches aus Symbolen besteht, die den Ablauf vom Einreichen eines Papers bis zum Einreichen eines Reports durch einen Reviewer darstellen. Mit der Zeit wurde dieses Alphabet so erweitert, dass es den gesamten Ablauf bis hin zum Abschluss einer Konferenz abbildet. Dabei handelt es sich im Einzelnen um folgende Symbole:

`SP` *Submit Paper*: Ein Autor erstellt ein Paper in einer Konferenz. Dabei gibt er den Titel und die Zusammenfassung des Papers, Schlagworte zur Kategorisierung sowie eventuelle Koautoren an. Nach Ausführung des Symbols ist das Paper in der Konferenz für andere Konferenzteilnehmer mit den entsprechenden Rechten sichtbar. Obwohl es möglich ist, mehrere Paper hochzuladen, beziehen sich alle folgenden Symbole nur auf das erste eingereichte Paper.

`ES` *End Submission*: Der PC Chair beendet die Einreichungsphase. Ab jetzt ist es nicht mehr möglich, weitere Paper im System anzulegen. Als Folge wird die Bidding-Phase gestartet, was PC Member in die Lage versetzt, Wünsche für Begutachtungen abzugeben.

`UD` *Upload Document*: Der Autor eines Papers lädt ein Dokument (üblicherweise eine PDF-Datei) für das Paper hoch, welches andere Konferenzteilnehmer dann herunterladen können. Dieser Vorgang ist mehrfach möglich, um das Dokument nach Änderungen zu aktualisieren.

`DD` *Download Document*: Der PC Chair lädt das Dokument für ein Paper herunter, falls es existiert. Hierbei handelt es sich um das einzige Symbol, welches keine Änderungen am Status der Konferenz vornimmt.

`EU` *End Upload*: Der PC Chair beendet die Upload-Phase. Danach ist es nicht mehr möglich, Dokumente zu eingereichten Papern hochzuladen oder diese nachträglich zu ändern.

`BD` *Bidding*: Ein PC Member gibt für ein existierendes Paper an, dass er es begutachten möchte. Der PC Chair kann diese Wünsche einsehen und verwenden, um Entscheidungen für die spätere Zuweisung zu treffen.

EB *End Bidding*: Der PC Chair beendet die Bidding-Phase, sodass es Konferenzteilnehmern nicht mehr möglich ist, Wünsche über mögliche Begutachtungen abzugeben.

SA *Special Assignment*: Der PC Chair weist dem Paper einen Reviewer zu, sodass er in der entsprechenden Phase eine Begutachtung des Papers vornehmen kann. Der Zusatz *Special* im Namen des Symbols soll andeuten, dass eine Zuweisung in diesem Fall voraussetzt, dass der entsprechende PC Member ein Bidding für das Paper abgegeben hat.

EA *End Assignment*: Der PC Chair beendet die Zuweisungsphase. Es ist ihm und anderen PC Chairs danach nicht mehr möglich, Reviewer zu Papern zuzuweisen. Das Beenden der Phase startet automatisch die Review-Phase, in welcher die Reviewer ihre Begutachtungen einreichen können, sowie die Decision-Phase, in der die abschließende Bewertung eines Papers festgelegt werden kann.

SR *Submit Report*: Der zu einem Paper zugewiesene Reviewer übermittelt seine Begutachtung, die daraufhin für den PC Chair und andere zugewiesene Reviewer sichtbar wird.

ER *End Review*: Der PC Chair beendet die Reviewphase, die Einreichung weiterer Reviews ist jetzt nicht mehr möglich.

PA *Paper Accept*: Der PC Chair stuft ein Paper als *akzeptiert* ein, sodass es Teil des Konferenzbandes wird.

PR *Paper Reject*: Der PC Chair lehnt ein Paper für die Aufnahme in den Konferenzband ab.

SF *Submit Final Report*: Der PC Chair sendet einen finalen Report an den Autor des Papers mit der Bitte, die finale Version hochzuladen.

ED *End Decision*: Der PC Chair beendet die Decision-Phase, eine Änderung an der Einstufung der Paper ist jetzt nicht mehr möglich. Das Beenden der Phase startet die finale Phase, in welcher die notwendigen Daten für akzeptierte Paper übermittelt werden.

UF *Upload Final Document*: Der Autor lädt die finale Version seines Papers ins System hoch.

EF *End Final* Der PC Chair beendet die finale Phase, die Konferenz ist damit abgeschlossen.

Der große Vorteil in der Wahl dieser abstrakten Symbole liegt darin, dass die durchgeführten Aktionen in jeder Version des OCS möglich sind. Zwar können sich die Vor- und Nachbedingungen ändern, beispielsweise in den automatisch gestarteten Phasen, dies führt jedoch wie gewünscht zu einem veränderten Modell des Systems und kann so dabei helfen, gewünschte Verhaltensänderungen zu bestätigen oder aber bisher unbekannte Fehler aufzudecken.

5.3 Entwicklung eines generischen Testtreibers

Im Gegensatz zum im Kapitel 4 getesteten Bugtracker Mantis hat sich die Programmierschnittstelle des OCS über die Zeit relativ häufig geändert. Dies betrifft nicht nur die aus mehreren Controller-Interfaces bestehende Geschäftslogik, sondern auch die grundlegenden Datenobjekte, etwa für Benutzer oder Paper. Als Folge davon ist es nicht möglich, einen einzigen Testtreiber über alle Versionen hinweg zu verwenden.

Eine weitere Hürde stellt der Klassenlader von Java dar. Die Paketnamen des OCS beinhalten keine Versionsnummer, sodass beispielsweise die Klasse für Konferenzen in jeder Version als `de.ls5.ocs.backend.data.Conference` verfügbar ist. Liegen nun mehrere Bibliotheken der API auf dem Klassenpfad, so müsste ein eigener Klassenlader verwendet werden, der abhängig von der aktuell zu lernenden Version ausschließlich die passende Bibliothek lädt.

Beide Probleme wurden durch den Einsatz von *Apache Maven*, einem weit verbreiteten Build-Management-Tool, behoben. Sollen etwa verschiedene Versionen des OCS gelernt und verglichen werden, so wird der Lernvorgang über einen eigenen Maven-Prozess gestartet. Dieser läuft nun in einer eigenen *Java Virtual Machine* (JVM) mit einem eigenen Klassenlader. Aufgrund der Fähigkeit von Maven, Profile zu verwalten, lassen sich die notwendigen Abhängigkeiten für jede Version des OCS in einem entsprechenden Profil zusammenfassen, welches beim Start des Prozesses angegeben wird.

Damit der Lernvorgang unabhängig von der getesteten Version Aktionen ausführen kann, wurde ein Interface namens `ApiAdapter` erstellt, über welches jede Kommunikation mit dem aktuell betrachteten OCS abläuft. Für jede zu testende Version existiert eine eigene Implementierung dieses Adapters, die mit Hilfe der *Contexts and Dependency Injection for Java EE* (CDI)[1] in die Lernsymbole injiziert wird. Während der Adapter auf diese Weise eine Abstraktion der Geschäftslogik bietet, leisten generische Datenobjekte wie etwa die `CommonConference` das gleiche für die ausgetauschten Daten, indem sie eine gemeinsame Datenbasis für alle Versionen des OCS bereitstellen.

Mit dem Begriff *Version* sind in diesem Zusammenhang größere Major-Releases (etwa 1.1.0 oder 1.8.0) gemeint, nicht einzelne Commits während der Entwicklungsphase. Während nämlich Erstere mit an Sicherheit grenzender Wahrscheinlichkeit Änderungen in der Programmierschnittelle aufweisen, ist dies für Letztere nicht zwingend erfüllt. Sofern sich die vom `ApiAdapter` verwendete Schnittstelle nicht ändert, kann der Adapter auch für andere Versionen verwendet werden. Erst diese Flexibilität erlaubt das kontinuierliche Lernen des aktuellen Entwicklungszweiges, da eine tägliche (manuelle) Implementierung zu aufwändig wäre.

Der Implementierungsaufwand eines `ApiAdapter` für eine spezifische Version ist jedoch eher gering. Da die meisten Aufrufe gar keine eigene Logik enthalten sollen, werden sie direkt an die entsprechenden Controller-Implementierungen des jeweiligen OCS weitergereicht. Auf diese Weise kann der Quellcode eines Adapters meist ohne größere Veränderungen von der Vorgängerversion kopiert werden.

Analog zu den in Abschnitt 4.3 beschriebenen Testtreibern für Mantis wurden auch für den OCS im Rahmen dieser Dissertation zwei unterschiedliche Wege gewählt, ein laufendes System anzusprechen: Mit direktem Zugriff auf die Geschäftslogik oder über die Weboberfläche des OCS. Das Klassendigramm in Abbildung 5.2 zeigt, dass der `ApiAdapter` in beiden Fällen derselbe ist, jedoch abhängig von der aktuell gewählten Konfiguration entweder auf das `ConferenceControllerBean` der Geschäftslogik oder die Schnittstelle `ConferenceControllerClient` zum Web-Frontend zugreift.

5.3.1 Backend-Adapter

In Abbildung 5.3 ist die Verwendung des generischen Testtreibers mit Zugriff auf die Geschäftslogik schematisch dargestellt. Auf der obersten Ebene stellt der Lernalgorithmus ein Anfragewort, bestehend aus einem oder mehreren generischen Symbolen. Innerhalb der Sym-

[1] `http://docs.oracle.com/javaee/7/tutorial/doc/cdi-basic.htm`

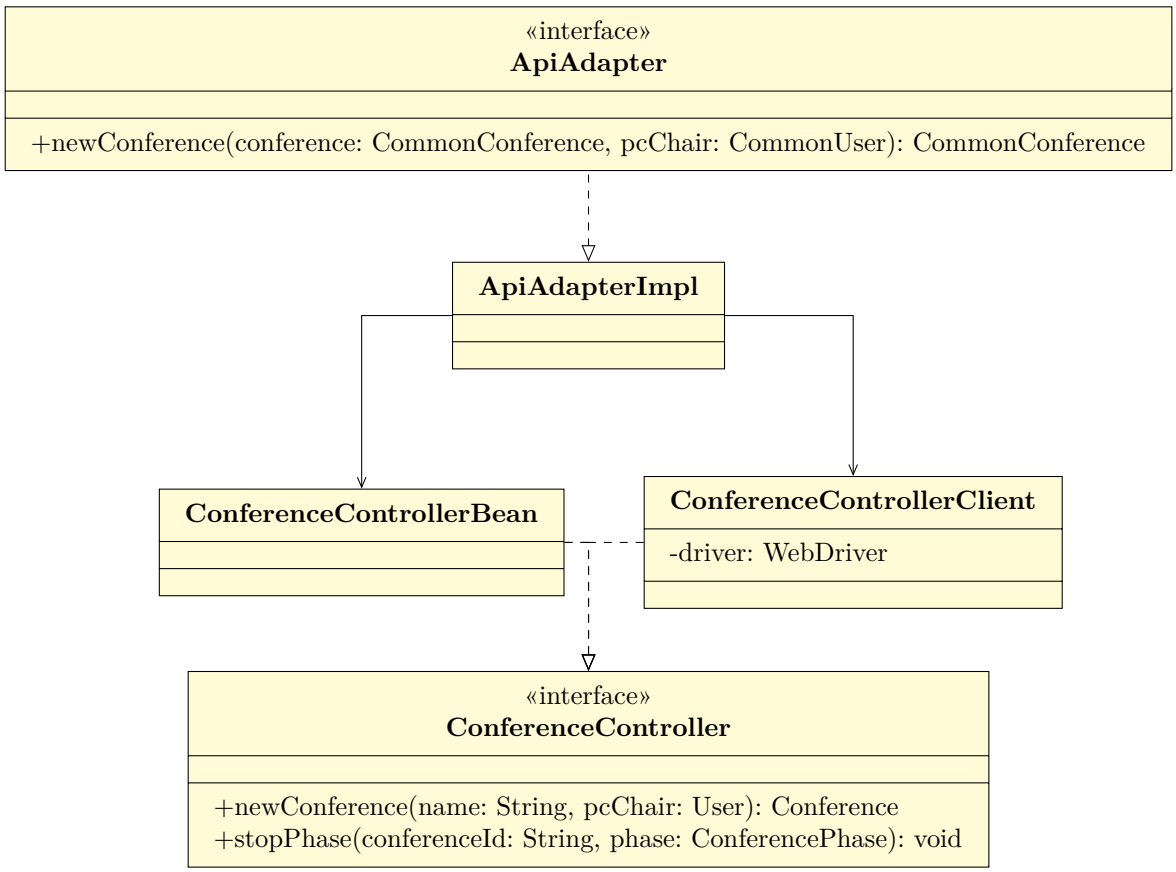

Abbildung 5.2: Klassendiagramm der Verwendung der OCS-Controller über einen generischen ApiAdapter am Beispiel des `ConferenceControllers`

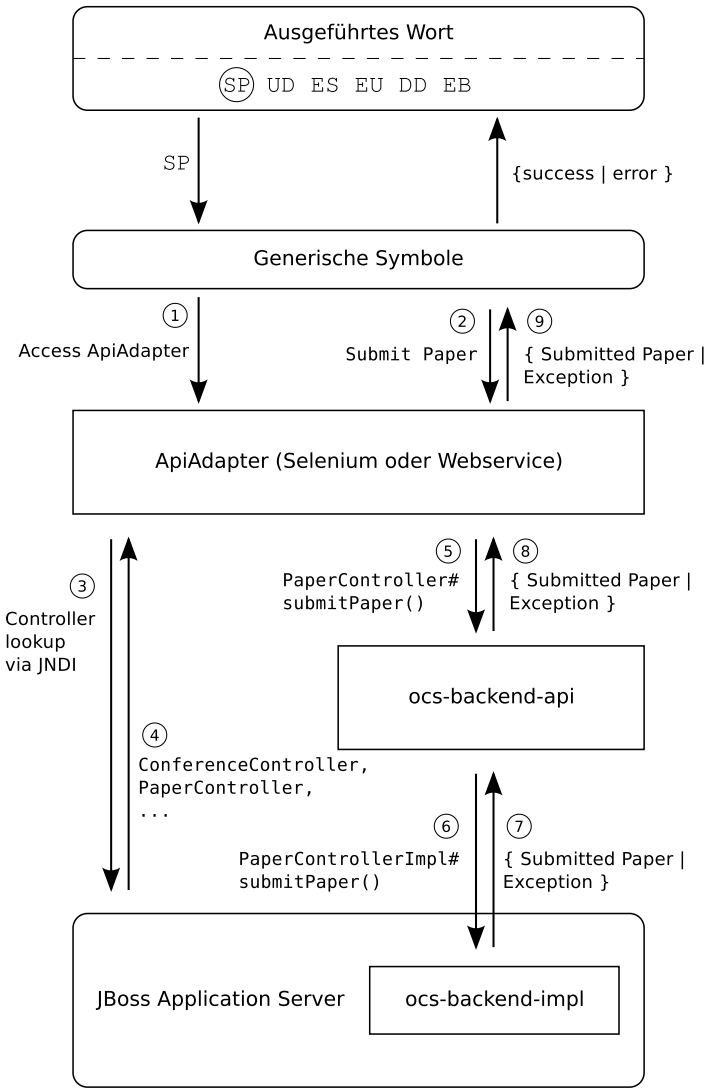

Abbildung 5.3: Der generische Testtreiber ermöglicht den Zugriff des Lernalgorithmus auf das zu lernende System, hier mit direktem Zugriff auf die Geschäftslogik (basiert auf [WNS+13]).

bole wird der `ApiAdapter`, passend zur aktuell gelernten Version, injiziert. Sobald auf diesem Adapter die Authentifikation eines Nutzers erfolgt, werden über das *Java Naming and Directory Interface* (JNDI) Remote-Proxy-Objekte der eigentlichen Controller der Geschäftslogik (etwa `ConferenceController` oder `PaperController`) erstellt. Jeder Aufruf auf einem solchen Objekt erfolgt mit dem authentifizierten Benutzer und verhält sich so, als würde der Controller in derselben Anwendung laufen, auch wenn damit eine JVM auf einem über das Internet erreichbaren Rechner angesprochen wird.

Soll nun, wie in Abbildung 5.3 exemplarisch dargestellt, innerhalb des Symbols `SP` ein Paper eingereicht werden, so wird zunächst ein generisches Objekt vom Typ `CommonPaper` erstellt. Dieses enthält die zum Erstellen notwendigen Daten, wie etwa den Titel oder die beteiligten Autoren, welche wiederum als `CommonUser` referenziert sind. Die Implementierung des `ApiAdapter`s, welche mit dem zur Version passenden Maven-Profil gestartet wurde und folglich auf die konkreten Klassen des Pakets `ocs-backend-api` der OCS-Version zugreifen kann, konvertiert diese Objekte in `Paper` und `User`. Nun wird das Proxy-Objekt des `PaperController`s verwendet, um die Methode `submitPaper` aufzurufen. Das momentan aktive OCS beantwortet diese Anfrage entweder mit der persistierten Version des übergebenen Papers oder einer Exception im Fehlerfall.

Konnte das Paper erfolgreich angelegt werden, konvertiert der `ApiAdapter` es wieder in ein Objekt vom Typ `CommonPaper` und liefert es an das ausführende Symbol. Im Fehlerfall wird die geworfene Exception mit einer generischen Variante, etwa `CommonSecurityException`, umhüllt und weitergeworfen. Im letzten Schritt des Testtreibers wertet dann das aufgerufene Symbol die Rückgabe aus, speichert das eventuell erhaltene Paper im Systemzustand des Reuse-Filters und liefert eine Aussage über den Erfolg oder Misserfolg an den Lernalgorithmus.

5.3.2 Frontend-Adapter

Ähnlich wie der in Abschnitt 4.3.1 vorgestellte Frontend-Adapter für Mantis basiert auch die Variante für den OCS auf dem Test-Framework *Selenium*. Da sich jedoch die Oberfläche des OCS im Vergleich zu Mantis über die Zeit hinweg wesentlich stärker geändert hat, wäre ein manuelles Erstellen eines solchen Adapters sehr aufwändig.

Erfreulicherweise verfügt der OCS jedoch bereits über mit Selenium realisierte Frontend-Tests, die automatisiert ausgeführt werden können. Diese sind so realisiert, dass sie die Schnittstelle der Geschäftslogik implementieren und können daher als Ersatz für die Controller der Geschäftslogik fungieren (siehe Abbildung 5.2). Aus diesem Grund reicht es aus, wenn die existierenden Frontend-Tests als Abhängigkeit der `ApiAdapter`-Implementierung hinzugefügt werden. Auch das Holen einer spezifischen Implementierung über einen JNDI-Aufruf wie in Abbildung 5.3 gezeigt, entfällt ersatzlos: Da die einzige Version des `ConferenceControllerClient` auf dem Klassenpfad liegt, wird automatisch die korrekte Klasse verwendet.

5.4 Ablauf eines Lernvorgangs

Die Erstellung eines Systemmodells für eine bestimmte Version des OCS erfordert die Einrichtung einer entsprechenden Instanz sowie einen anschließenden Zugriff des Lernalgorithmus auf diese. Im Folgenden wird der erforderliche Ablauf beschrieben.

Zugriff auf spezifische Versionen

Zu jeder veröffentlichten Version des OCS existieren fertig kompilierte EAR-Archive[2], welche zusammen mit der jeweiligen JBoss-Version ausgeführt werden können. Dies gilt allerdings nicht für einzelne Entwicklungsstände zwischen den veröffentlichten Versionen. Eine Anpassung der Pakete an die Anforderungen des Lernvorgangs gestaltet sich zudem ebenso schwierig wie die Einrichtung der jeweils passenden Softwareumgebung, weshalb ein Zugriff auf den Quellcode erforderlich ist.

Da es sich beim OCS nicht um eine Open-Source-Anwendung handelt, ist der Quellcode nicht öffentlich verfügbar. Für Entwickler existiert jedoch ein privates Repository, das genau wie bei Mantis über Git verwaltet wird und alle Versionen bis hin zum Beginn der Entwicklung im Jahr 2009 beinhaltet. Auch hier macht sich die Eigenschaft von Git, den gesamten Stand eines Repositorys vorzuhalten, positiv bemerkbar, da der Zugriff auf einzelne Versionen ohne Netzanbindung funktioniert.

Anpassung einer Version an den Lernvorgang

Damit Installation und Lernvorgang problemlos ablaufen, sind abhängig von der Version des OCS bis zu zwei Anpassungen am Quellcode notwendig:

- Veröffentlichte Versionen des OCS gehen davon aus, dass sie auf einem Server mit bereits existierenden Datenbanktabellen installiert werden. Auf diese Weise wird bereits beim ersten Start nach einem Upgrade deutlich, ob eine eventuell notwendige Migration der Datenbank nicht durchgeführt wurde. Diese Überprüfung wird deaktiviert, um eine Erstellung einer Instanz ohne Daten zu ermöglichen.

- Webanwendungen leiden häufig unter dem Problem des automatischen Schließens von User-Sessions nach einer gewissen Zeit von Inaktivität. Hat der Nutzer in dieser Zeit jedoch in der Webanwendung einen längeren Text verfasst, etwa den Abstract zu einem Paper, so kann dieser im schlimmsten Fall verlorengehen. Aus diesem Grund wurde die Zeitbeschränkung im OCS auf 90 Minuten erhöht. Dieser Wert kann sich während eines Lernvorgangs jedoch negativ auswirken, da die maximale Anzahl von Sessions beschränkt ist. Der Wert wird daher vor der Installation auf fünf Minuten gesenkt.

Kompilierung und Initialisierung einer OCS-Version

Nachdem die im letzten Abschnitt beschriebenen Anpassungen vorgenommen wurden, muss der Quellcode neu kompiliert werden. Um den Entwicklungsprozess möglichst einfach zu halten, verfügt der OCS bereits seit sehr frühen Versionen über ein Kommandozeilenscript, welches diese Funktion mit einem einzigen Kommando erledigt. Weiterhin führt dieses Script auch die Initialisierung der Datenbank und eines eventuellen LDAP-Servers durch.

Ein großer Vorteil der Verwendung dieses Scripts liegt darin, dass die durchzuführenden Schritte sich mit der Weiterentwicklung des OCS geändert haben, das Script jedoch immer zum entsprechenden Entwicklungsstand passt und das System somit korrekt initialisiert.

Zum Starten der neuen Instanz kommt der bereits in Abschnitt 4.4 beschriebene `PostgresManager` sowie analog ein `LdapManager` für den LDAP-Server zum Einsatz. Zusätzlich sorgt die Klasse `JbossManager` für ein korrektes Starten des JBoss-Servers unabhängig von der verwendeten Version.

[2] EAR steht für *Enterprise Archive* oder *Enterprise Application Archive* und bezeichnet ein im Java-Umfeld gebräuchliches Paketformat für Webandwendungen.

Auswertung durch den Lernalgorithmus

Sobald die Instanz des OCS initialisiert und gestartet wurde, kann der Lernalgorithmus mit dem Stellen von Membership Querys beginnen. Genau wie bei Mantis werden dazu Worte aus generischen Symbolen erstellt, die ihre Aktionen wiederum auf dem gewählten `ApiAdapter` durchführen. Der Ablauf für den auf die internen Controller zugreifenden Adapter wurde bereits in Abschnitt 5.3.1 erläutert. Zugriffe durch den Selenium-Adapter erfolgen analog, sind jedoch in der Ausführung wesentlich langsamer.

5.5 Verifikation von Systemmigrationen

Im Laufe der Entwicklung des OCS wurde das System um diverse Funktionen erweitert, die eine Anpassung der verwendeten Datenbank notwendig machten. Zu diesen Funktionen gehören beispielsweise das mehrmalige Durchlaufen von Reviewzyklen für einzelne Paper oder auch nur das Speichern zusätzlicher Informationen wie den Autor eines Reports. In früheren Versionen wurden zudem die Daten einzelner Nutzer nicht in der Datenbank, sondern von einem LDAP-Server verwaltet. Im Gegenzug erfolgte die Ablage von hochgeladenen Dateien in der Datenbank, statt wie bei aktuellen Versionen im Dateisystem, was die Sicherung der Tabellen sehr verlangsamte.

Bei den ersten notwendigen Änderungen handelte es sich um simple Operationen wie das Hinzufügen neuer Datenbankspalten oder die Anpassung von Wertebereichen. Diese wurden dann manuell während der Installation einer neuen Version durchgeführt, was zum Einen sehr fehleranfällig war und zum Anderen Nachteile in Hinblick auf Nachvollziehbarkeit und Dokumentation besitzt. Spätestens jedoch als mit Veröffentlichung der Version 1.3.0 die Informationen der LDAP-Server in die Datenbank integriert werden mussten, war eine robustere Lösung notwendig.

Zu diesem Zweck wurde eine Migrator-Software entwickelt, die als ausführbares Programm selbstständig eine laufende OCS-Datenbank migrieren kann, vorausgesetzt, die OCS-Installation ist in dem Moment inaktiv. Die Software enthält dazu für jeden Versionssprung, der eine Migration erfordert, ein entsprechendes Modul, das manuell aufgerufen werden kann, etwa die Migration von 1.2.8 nach 1.3.0 (Übernahme der LDAP-Daten) oder die von 1.3.0 nach 1.4.0 (Speichern von Dokumenten im Dateisystem).

Für die in Abschnitt 3.6 beschriebene Verifikation einer Migration ist es also notwendig, dass nach dem Auswerten der Präfixe das System heruntergefahren und der Migrator bei laufender Datenbank ausgeführt wird. Wichtig ist hierbei, eine aktuelle Version der Migrator-Software zu verwenden, da auch hier nachträglich Fehler erkannt wurden. Solche fehlerhaften Migrationen wurden meist manuell korrigiert und der Migrator im Nachhinein entsprechend angepasst, sodass zukünftige Aufrufe reibungslos funktionieren.

Zum nachträglichen Auswerten der Suffixe werden die zuvor erzeugten Systemzustände benötigt. Diese enthalten im Falle des OCS unter anderem

- die vom Reset erzeugte Konferenz, auf der die Symbole ausgeführt werden
- die vom Reset erzeugten Nutzer (etwa PC Chair oder Autor), welche zur Durchführung der Symbole verwendet werden
- das Paper, welches eventuell im Laufe der Ausführung in der Konferenz erzeugt wurde

5.6 Ergebnisse

Mit Hilfe der kontinuierlichen Qualitätskontrolle wurden im OCS mehrere Fehler entdeckt, die mit den bisherigen Testverfahren unentdeckt blieben. Während einer der Fehler eine fehlende Sicherheitsabfrage aufdeckte, die ein unterschiedliches Verhalten von Geschäfts- und Präsentationsschicht verursachte, trat ein anderer nach bereits erfolgter Behebung erneut auf. Andere entdeckte Probleme führten dazu, dass etwa Fehlkonfigurationen der verwendeten Server erkannt und beseitigt werden konnten.

In diesem Abschnitt werden die erhaltenen Ergebnisse vorgestellt. Zusätzlich erfolgt eine Auswertung der Leistung der verwendeten Adapter-Implementierungen.

5.6.1 Vergleich veröffentlichter Versionen

Damit sichergestellt werden kann, dass sich der aktuelle Entwicklungsstand in der Funktionsweise nicht von der derzeit im Produktivbetrieb eingesetzten Version unterscheidet, ist ein Modell dieser Version als Referenz hilfreich. Um zusätzlich einen Vergleich über die Entwicklung der letzten Jahre zu erhalten, wurde für alle Versionen aus Tabelle 5.1 beginnend mit 1.1.0 ein entsprechendes Modell erzeugt.

Das Modell für die aktuelle Version 1.10.0 des OCS verfügt über 253 Zustände. Der entsprechende Graph wäre daher zu groß, um ihn in diesem Dokument abzudrucken. Aus diesem Grund zeigt Abbildung 5.4 einen Ausschnitt des Modells mit insgesamt 24 Zuständen. Aus Gründen der Übersichtlichkeit wurden nur erfolgreiche Aktionen dargestellt und dabei möglichst auf überschneidende Kanten verzichtet.

Auch wenn die Abbildung nur etwa ein Zehntel der Zustände zeigt, ist die Struktur der Abläufe im OCS recht gut zu erkennen. Auf der linken Seite erfolgt in jedem Fall das Einreichen mindestens eines Papers, genauer in allen Sequenzen von Aktionen, welche die Zustände q_1 oder q_7 passieren. Folgt man diesen Pfaden weiter nach links, und zwar über die Zustände q_4, q_{12} oder q_{19}, so ist hier der gewöhnliche Konferenzablauf sichtbar, bei dem für eingereichte Paper auch ein Dokument zur Begutachtung hochgeladen wird. Dieser Status spiegelt sich um folgenden im Aufruf des invarianten Symbols für den Dokument-Download (DD) wider.

Deutlich wird beim manuellen Verfolgen der Pfade auch die Bedeutung der einzelnen Phasen im OCS. Gewisse Aktionen wie das Bidding (BD) sind erst in einer bestimmten Phase möglich, hier beispielsweise nach Einreichen eines Papers und dem Beenden der Submission-Phase. Den weiteren Verlauf deutet die Mitte des Graphen an, wo nach erfolgtem Bidding und dem Schließen der entsprechenden Phasen das Assignment (SA) vorgenommen werden kann.

Einen Sonderfall, der zwar möglich aber weniger getestet ist, zeigt der rechte Teil des Graphen. In allen Zuständen, die auf q_3 oder q_8 folgen, wird niemals ein Paper eingereicht werden, da die Submission-Phase bereits beendet wurde. Ohne existierendes Paper lassen sich jedoch die meisten Symbole nicht ausführen, da für das Hochladen eines Dokuments, das Bidding oder die Zuweisung von Reviewern immer ein Paper benötigt wird. Aus diesem Grund beschränkt sich dieser Teil des Graphen ausschließlich auf das Beenden von Phasen und eignet sich daher gut, um deren Abhängigkeiten übersichtlich darzustellen.

5.6.2 Fehlererkennung durch visuelle Überprüfung

Die einfachste Vorgehensweise der Fehlersuche ist eine visuelle Verifikation der gelernten Modelle. Werden die Automaten nicht allzu groß, lassen sich Ausführungspfade nachvollziehen

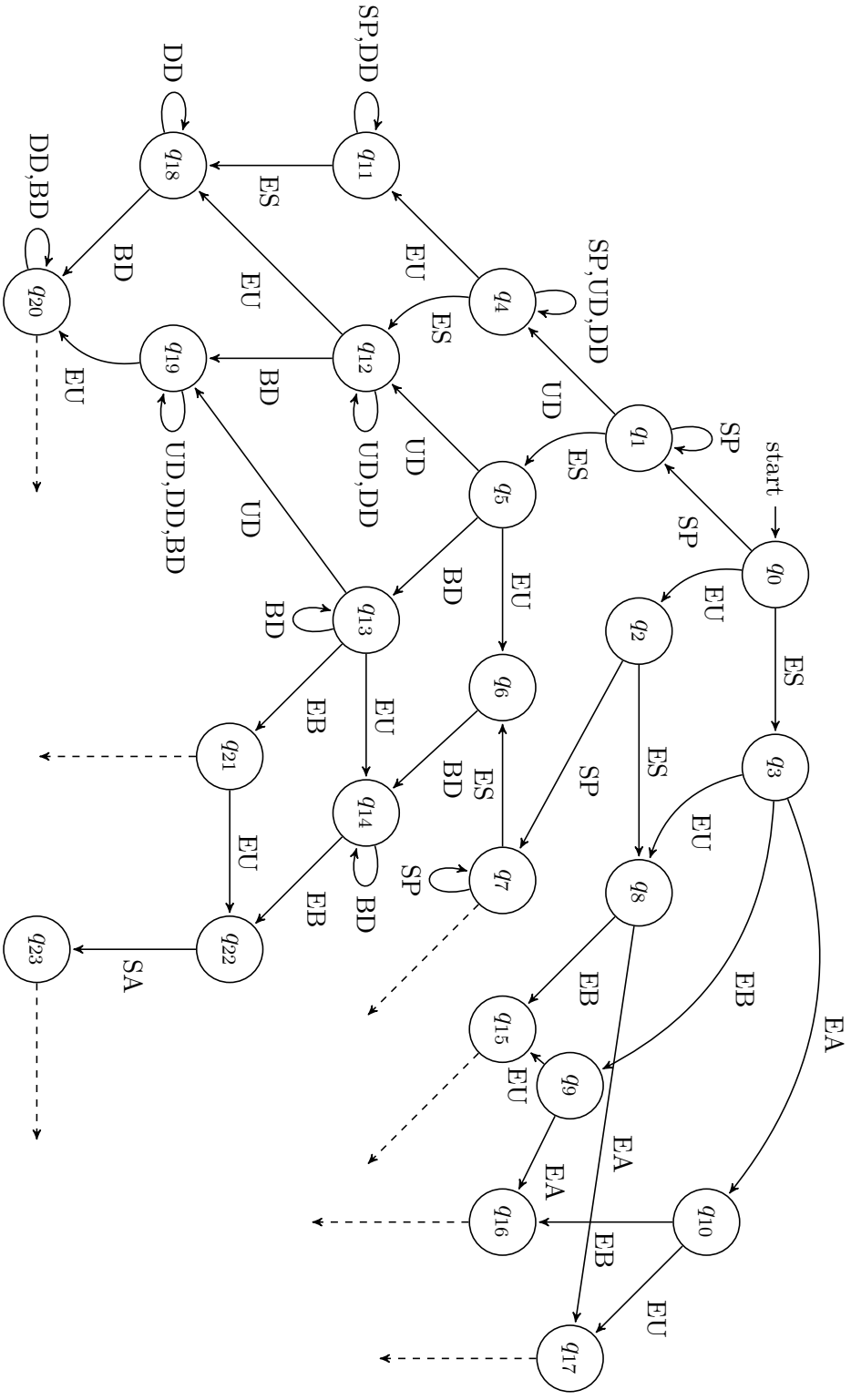

Abbildung 5.4: Ausschnitt des automatisiert gelernten Modells des OCS in Version 1.10.0

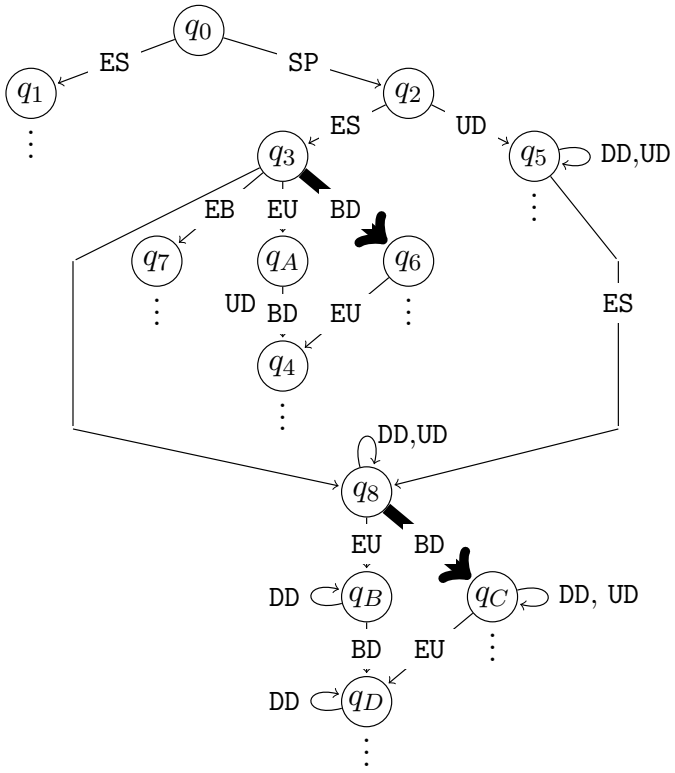

Abbildung 5.5: Unterschied zwischen zwei Modellen, nachdem die Übergangslogik der Phasen geändert worden war. Quelle: [WNS+13]

und es leicht möglich, unerlaubte Aktionen aufzuspüren oder auch das Fehlen von eigentlich möglichen Vorgängen zu entdecken.

Weitere Fehler fallen schnell auf, wenn Modelle von aufeinanderfolgenden Versionen unterschiedlich sind. Die Prüfung auf Äquivalenz kann zuvor effizient automatisiert erledigt werden, etwa mit dem Algorithmus von Hopcroft [Hop71]. Wurde eine Abweichung festgestellt, so kann die Ursache durch eine eine synchronisierte Breitensuche über beide Modelle ermittelt werden. Das Resultat ist ein Ausführungspfad, der für jeden der beiden Automaten ein unterschiedliches Ergebnis liefert. Mit diesem nun bekannten Unterschied kann ein Entwickler daraufhin beginnen, den Fehler einzuordnen.

Die einfachste Erklärung liefern Unterschiede, welche durch eine gewünschte Änderung im Systemverhalten hervorgerufen werden. Hier eignet sich dieser Ansatz sehr gut dazu, zu überprüfen, ob die Anpassung wirklich die vorgesehene Wirkung erzielt hat.

Ein Beispiel für eine so erkannte Änderung wurde beim Wechsel der OCS-Version von 1.4.0 auf 1.5.0 deutlich. Während in der vorhergehenden Version das Bidding für ein Paper nur dann möglich war, wenn zuvor auch ein Dokument hochgeladen wurde, so wurde diese Einschränkung mit Version 1.5.0 aufgehoben. Der Grund für diese Änderung war, dass einige Konferenzveranstalter ein Bidding nur anhand des Paper-Abstract durchführen wollten.

Abbildung 5.5 zeigt den Unterschied zwischen beiden Versionen. Die hier fett dargestellten Transitionen zeigen Zustandsübergänge, die nur im neuen Modell zu finden sind. Hier ist ein Bidding (BD) sowohl ohne ($q_0 \rightarrow q_2 \rightarrow q_3$) als auch mit hochgeladenem Dokument ($q_0 \rightarrow q_2 \rightarrow q_5 \rightarrow q_6 \rightarrow q_8$) möglich. In keinem der beiden Fälle ist jedoch zuvor ein Beenden der Upload-Phase (EU) notwendig.

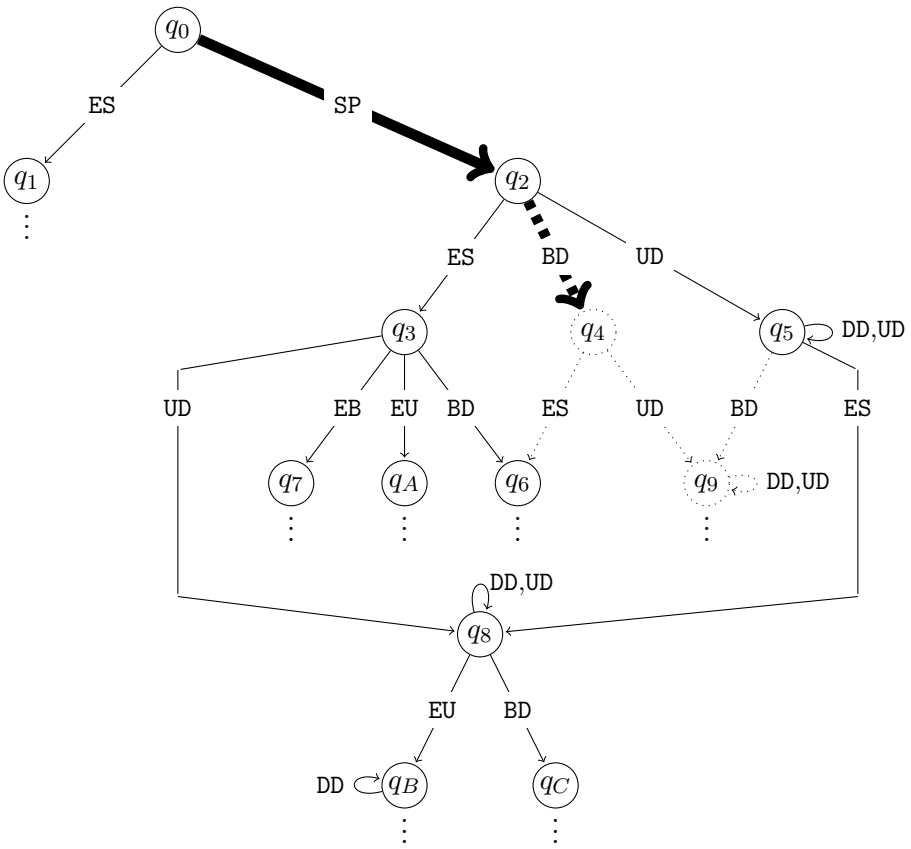

Abbildung 5.6: Modell des OCS, bei dem fälschlicherweise direkt nach dem Einreichen eines Papers ein Bidding für dieses abgegeben werden konnte, obwohl die entsprechende Phase noch nicht aktiv war. Quelle: [WNS+13]

Da es sich um eine erwünschte und somit erwartete Änderung handelt, waren in diesem Fall keine weiteren Aktionen erforderlich. Das gelernte Modell belegt die Wirksamkeit der Anpassung und ein Vergleich mit den Modellen zukünftiger Versionen stellt sicher, dass dieses erwünschte Verhalten beibehalten wird.

Doch nicht alle entdeckten Fehler lassen sich auf gewünschte Änderungen zurückführen. Ein Beispiel für die Entdeckung eines realen Fehlers zeigt das Modell in Abb. 5.6. Aufgrund einer fehlerhaften Rechteüberprüfung wäre es hier einem Nutzer möglich gewesen, ein Bidding für ein eingereichtes Paper abzugeben (BD), obwohl die entsprechende Phase noch gar nicht (als Folge von ES) gestartet war. Da die entsprechende Abfolge von Aktionen nicht Teil der automatisiert ausgeführten Tests war, wurde sie erst durch visuelle Überprüfung des gelernten Modells entdeckt.

Ein weiterer Fehler, der mittels visueller Überprüfung der Modelle erkannt werden konnte, wird in Abb. 5.7 deutlich. Hier war ein PC Chair in der Lage, direkt nach Erstellung der Konferenz die Upload-Phase zu beenden (EU), wodurch es nicht möglich gewesen wäre, Dokumente zu eingereichten Papern hochzuladen. Zwar lassen sich Phasen im OCS bei Bedarf reaktivieren, jedoch könnte eine solche Fehlbedienung zu Irritationen bei den Nutzern führen. Der Fehler wurde daraufhin behoben, sodass der vorgegebene Ablauf einer Konferenz wieder eingehalten wird.

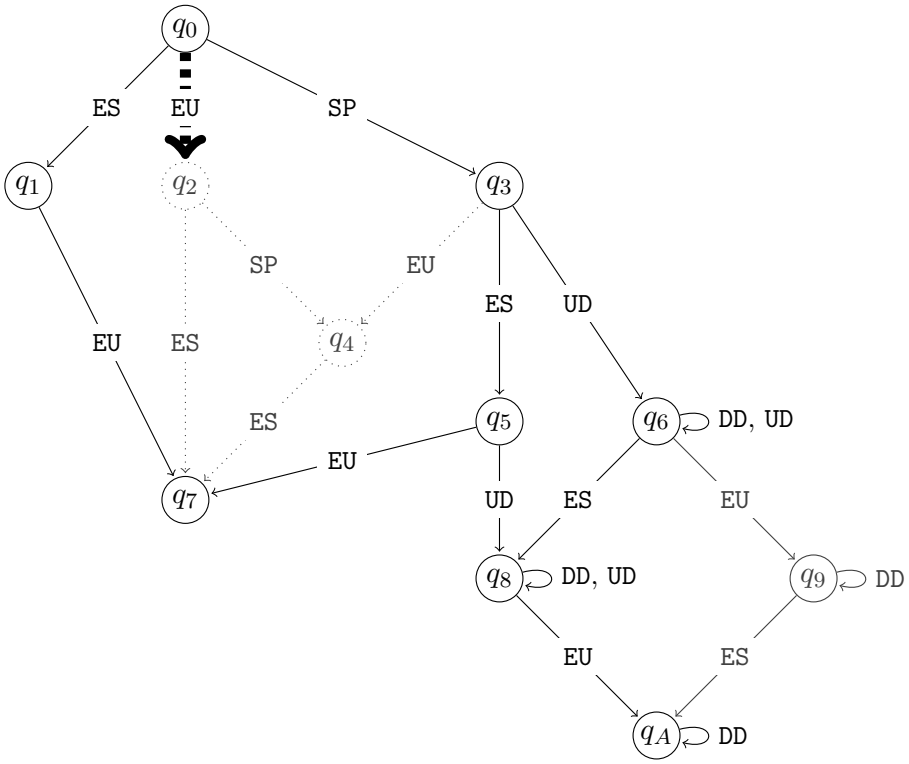

Abbildung 5.7: Mit dem in diesem Modell gezeigten Fehler war es möglich, die Upload-Phase zu beenden (EU), bevor die Submission-Phase beendet wurde (ES). Quelle: [WNS+13]

5.6.3 Fehlererkennung durch Model Checking

Die visuelle Überprüfung von Modellen ist ein einfaches und schnelles Hilfsmittel, um offensichtliche Fehler schnell aufdecken zu können. Werden die Modelle jedoch größer und tritt der Fehler nicht wie in den vorangegangenen Beispielen in der Nähe des Startzustandes auf, so ist es selbst für den geübten Beobachter schwierig, Probleme direkt zu erkennen. An dieser Stelle setzt das *Model Checking* an, welches eine effiziente Verifikation endlicher Zustandssysteme erlaubt [CGP99]. Mittels temporallogischer Formeln, etwa in der Linear Temporal Logic (LTL), lassen sich Modelle daraufhin überprüfen, ob sie bestimmte Eigenschaften erfüllen oder verletzen. Die in den Abbildungen 5.6 und 5.7 gezeigten Problemfälle lassen sich beispielsweise durch die Formel

$$(\neg(BD \vee EU)) \mathbf{W} ES$$

abdecken. Sie besagt, dass weder ein Bidding (BD) noch ein Beenden der Upload-Phase (EU) geschehen dürfen, bevor nicht die Submission-Phase beendet wurde (ES). Wird diese Formel für alle zukünftig gelernten Modelle überprüft, lassen sich die genannten Fehler dauerhaft ausschließen.

Doch nicht nur bereits bekannte Fehler können auf diese Weise entdeckt werden, auch die Einhaltung gewisser Rahmenbedingungen über den gesamten Konferenzablauf kann auf diese Weise geprüft werden. Eine zentrale Aufgabe des OCS ist beispielsweise die Verwaltung

von Paper-Reviews, die auf den von Autoren hochgeladenen Dokumenten basieren. Damit alle Reviewer dasselbe Dokument sehen, sollten sie nur dasjenige verwenden, welches vom Autor ins System hochgeladen wurde. Fehlt dieses Dokument jedoch und der Autor nennt etwa in der Kurzfassung des Papers einen eigenen Download-Link für das Paper, kann dies nicht mehr sichergestellt werden.[3] Daher sollte ein Review (SR) immer nur dann möglich sein, wenn ein Dokument vorliegt (UD), was mit der Formel

$$\neg SR \, \mathbf{W} \, UD$$

geprüft werden kann. Weiterhin sollte es für den Autoren unmöglich sein, zu Beginn der Review-Phase sein Paper nach wie vor zu aktualisieren. Da die Review-Phase automatisch mit Beenden der Assignment-Phase (EA) gestartet wird, ergibt sich daraus die Formel

$$\mathbf{G}(EA \Rightarrow \mathbf{G}\neg UD)$$

5.6.4 Vergleich zwischen Weboberfläche und Kontrollschicht

Bereits eine einfache visuelle Überprüfung, wie sie in Abschnitt 5.6.2 beschrieben wurde, konnte den in Abbildung 5.6 gezeigten Fehler aufdecken. Dieser erlaubte es, dass ein PC Member ein Bidding für ein eingereichtes Paper abgeben konnte, obwohl die entsprechende Phase noch gar nicht gestartet war. Eine interessante Eigenschaft dieses Fehlers ist, dass er auch durch ausführliche Integrationstests nicht vor der Veröffentlichung gefunden wurde. Daher stellte sich die Frage, ob es eventuell einen Unterschied gibt zwischen den durch das Backend erlaubten Aktionen und denen, welche ein Nutzer auf der Weboberfläche ausführen kann.

Aus diesem Grund wurde ein vergleichender Lernvorgang gestartet, der das System mittels Selenium so anspricht, als würde ein Benutzer die Seite im Browser aufrufen. Erfreulicherweise verfügt der OCS bereits seit mehreren Jahren über Selenium-Tests, welche dasselbe Interface wie die Backend-Controller implementieren (siehe Abbildung 5.2). Mit einigen moderaten Anpassungen war es auf diese Weise möglich, ein Modell der Version 1.5.0 zu erstellen, welches fast ausschließlich aus Aufrufen besteht, die im Browser vorgenommen werden können. Eine vollständige Abdeckung über Selenium lässt sich in diesem Fall nicht erreichen, da einige der Methoden interne Informationen, die vom Browser nicht dargestellt werden, benötigen. Insgesamt dauerte dieser Vorgang für ein aus zehn Symbolen bestehendes Alphabet knapp 55 Minuten, während das Vergleichsmodell mit Backend-Aufrufen in 15 Minuten gelernt wurde.

Der in Abbildung 5.8 gezeigte Modellausschnitt stellt das Problem deutlich dar: Nachdem die erste Einreichung eines Papers geschehen ist ($q_0 \to q_1$), ist es einem PC Chair möglich, ein Bidding für dieses Paper abzugeben ($q_1 \to q_4$). Dies sollte jedoch nicht erlaubt sein, da die Bidding-Phase noch gar nicht gestartet wurde, was daran erkennbar ist, dass ein Beenden der Phase (`BD`) in den Zuständen q_0 und q_1 gar nicht möglich ist. Vielmehr wird diese Phase erst gestartet, nachdem die Upload-Phase beendet wurde (`EU`), was beispielsweise gut an den Pfaden $q_0 \to q_2 \to q_6$ oder $q_0 \to q_1 \to q_5 \to q_8$ erkennbar ist. Das Modell lässt vermuten, dass eine aktive Bidding-Phase keine Voraussetzung für ein Bidding ist, was einen ernsten Fehler in der Systemlogik darstellt.

[3] Eine weitere Problematik wäre, dass der Autor durch die Zugriffe auf ein Dokument erkennen könnte, von welcher IP ein Dokument aufgerufen wurde und so Informationen über die eigentlich anonymen Reviewer erhält.

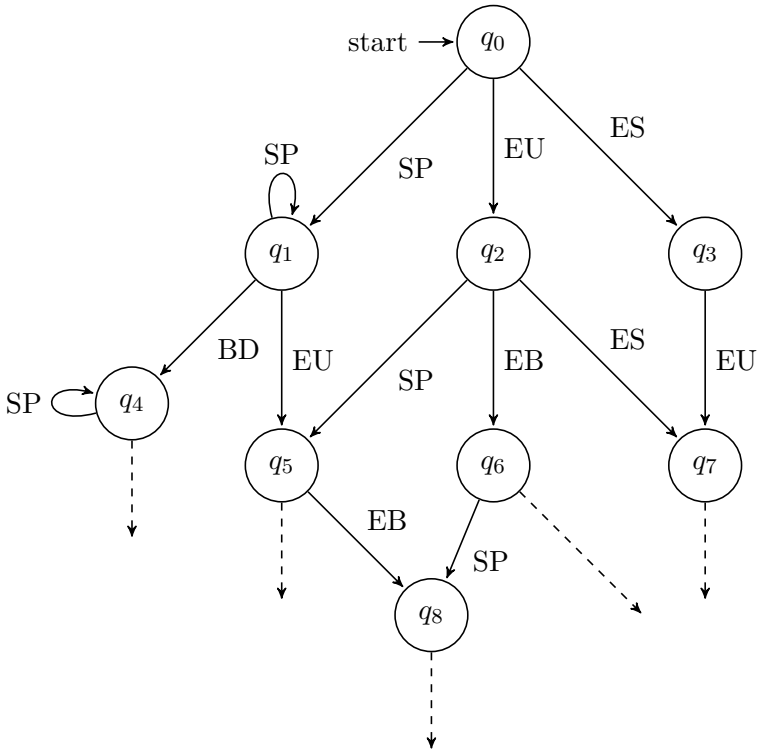

Abbildung 5.8: Ausschnitt des Modells des OCS in Version 1.5.0, gelernt mit dem Backend-Adapter

Im direkten Vergleich zeigt der in Abbildung 5.9 dargestellte Ausschnitt ein grundsätzlich anderes Verhalten. Ein Bidding ist hier nur möglich, wenn sowohl ein Paper eingereicht als auch die Upload-Phase beendet wurde ($q_0 \rightarrow q_1 \rightarrow q_4 \rightarrow q_7 \rightarrow q_9$). Der Grund für das korrekte Verhalten in diesem Modell liegt darin, dass die Präsentationsschicht des OCS in vielen Fällen eigene Überprüfungen der Eingaben des Nutzers durchführt, bevor sie an die Geschäftslogik weitergereicht werden. Auf diese Weise entfallen unnötige Anfragen, welche das System eventuell zusätzlich auslasten würden. Im günstigsten Fall geschieht eine solche Überprüfung direkt per JavaScript im Browser des Benutzers, sodass überhaupt gar keine Anfragedaten an den Server transportiert werden.

Eine weitere Auffälligkeit, die beim Vergleich der beiden Modelle auffällt, ist die Reihenfolge der Phasenübergänge, die schon ausgehend vom Zustand q_0 unterschiedlich sind. So erlaubt das Backend als erste Aktion das Beenden der Upload-Phase ($q_0 \rightarrow q_2$), während dazu im Browser zuerst die Submission-Phase beendet werden muss ($q_0 \rightarrow q_2 \rightarrow q_5$). Auch hier führt die Logik der Präsentationsschicht einen eigenen Abgleich der Daten durch und verhindert so problematische Konferenzzustände.

Diese Art der doppelten Überprüfung hat in beiden Fällen verhindert, dass Aktionen nicht durchgeführt werden konnten, die in der Geschäftslogik aber möglich gewesen wäre. Ein manueller Integrationstest im Browser hat diese Fehler nicht entdeckt, da sie von der Logik der Präsentationsschicht verdeckt wurden. Nichtsdestotrotz ist eine Korrektur des fehlerhaften Codes dringend angeraten, etwa für den Fall, dass in Zukunft die Präsentationsschicht angepasst oder Drittanbieter Zugriff auf die Geschäftslogik erhalten sollen.

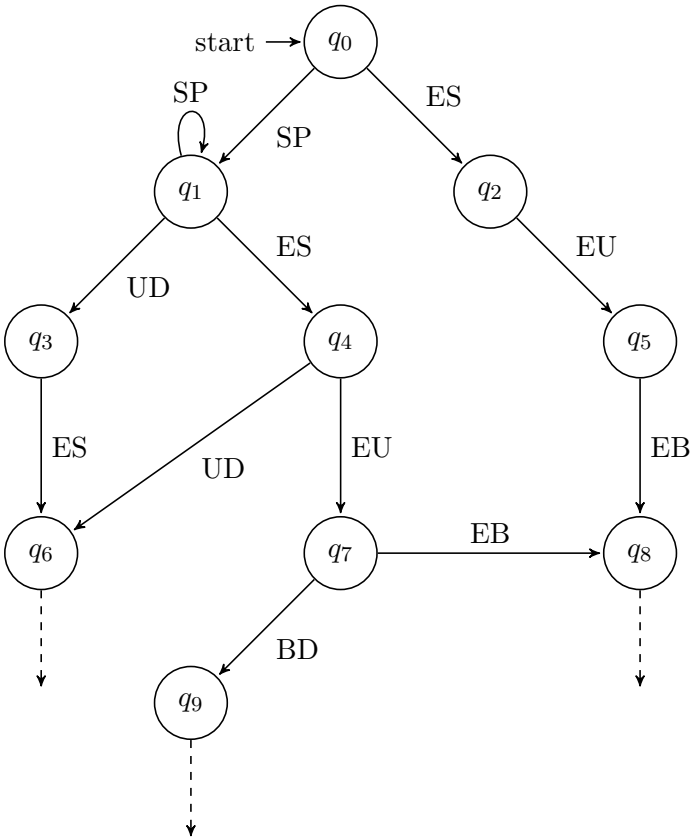

Abbildung 5.9: Ausschnitt des Modells des OCS in Version 1.5.0, gelernt mit dem Selenium-Adapter

5.6.5 Leistungsunterschiede

Wie in Abschnitt 5.6.4 gezeigt, ist ein Vergleich des Verhaltens von Geschäftslogik und Präsentationsschicht hilfreich, um Unterschiede aufzudecken, die in einem anderen Kontext Probleme verursachen könnten. Wünschenswert wäre daher eine dauerhafte oder zumindest regelmäßige Prüfung, allerdings erfordert die Erstellung eines Modells mit Hilfe des Selenium-Adapters deutlich mehr Zeit.

Für den Bugtracker Mantis wurde in Abschnitt 4.6.3 gezeigt, dass der Unterschied zwischen beiden Adaptern ungefähr bei der siebenfachen Laufzeit liegt. Während sich dort allerdings die gelernten Modelle schon in der Anzahl der Zustände unterschieden und ein genauer Laufzeitvergleich schwierig war, gilt dies nicht für die aktuelle Version des OCS. Die mit beiden Adapter-Varianten gelernte Version lieferte jedes Mal nach etwa 20.300 Membership Querys dasselbe Modell mit 253 Zuständen.

Um eine Vergleichbarkeit der einzelnen Symbollaufzeiten zu gewährleisten, wurden beide Lernvorgänge mit nur einem Thread ausgeführt. Das führt zwar für den `ApiAdapter`, welcher die Geschäftslogik direkt anspricht, zu deutlichen Einbußen in der Laufzeit (siehe Abschnitt 5.6.9), andernfalls würden sich jedoch mehrere gleichzeitig ausgeführte Symbole gegenseitig blockieren und somit das Ergebnis verfälschen.

Die Gesamtlaufzeit für den Backend-Adapter liegt in diesem Fall bei 79 Minuten, was einer Rate von 257 Membership Querys in der Minute entspricht. Beim Zugriff auf den

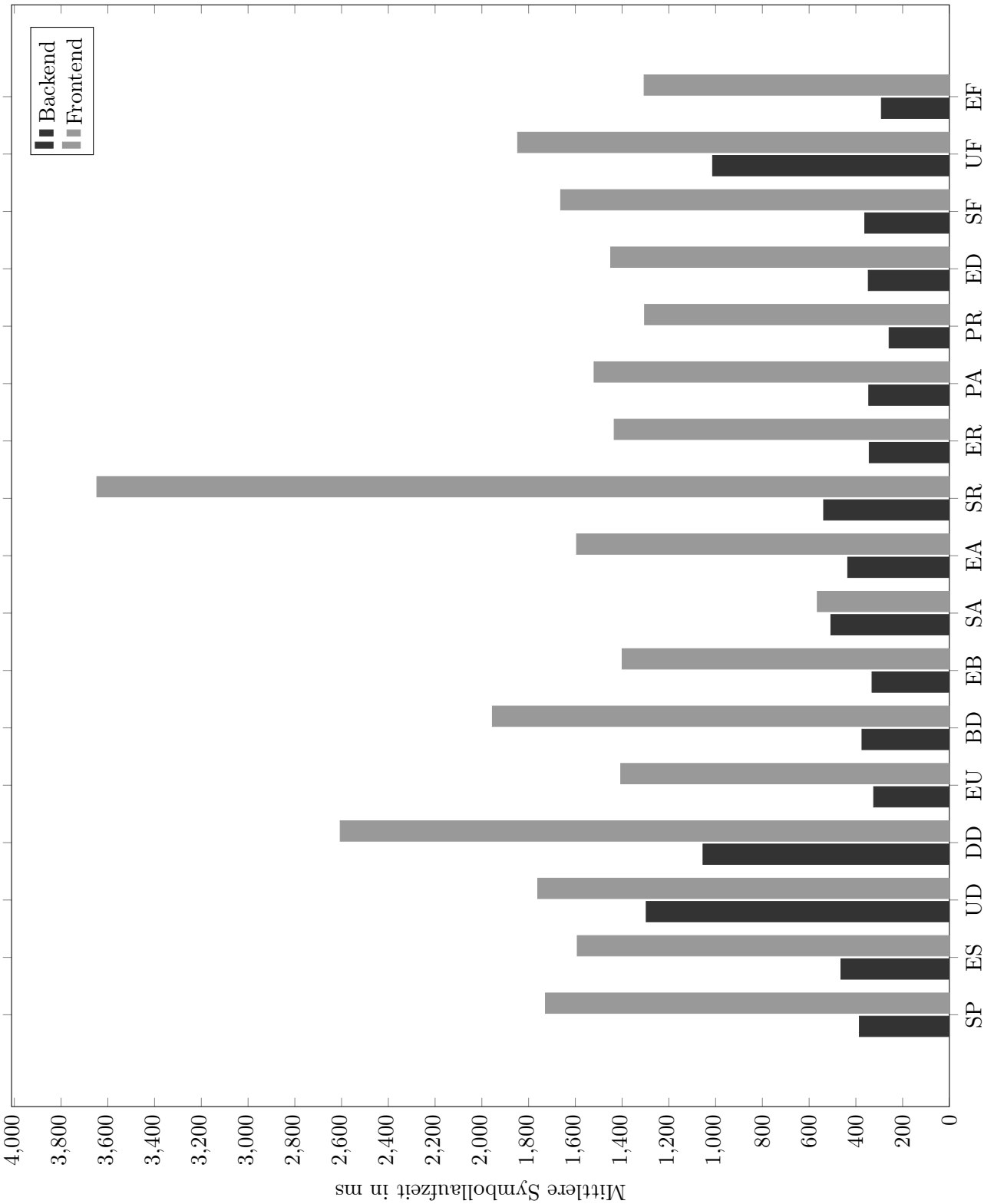

Abbildung 5.10: Durchschnittliche Laufzeit eines Alphabetsymbols je nach ApiAdapter beim Lernen von OCS 1.10.4

Selenium-Adapter bricht dieser Wert drastisch ein, nur etwa 8-9 Membership Querys werden pro Minute verarbeitet. Dies entspricht grob gerechnet einem Geschwidigkeitsunterschied um den Faktor 30. Die Gesamtdauer liegt hier bei über 39 Stunden und ist damit selbst auf aktueller Hardware für einen täglichen Test nicht geeignet.

Eine genaue Auflistung der einzelnen Symbole ist in Abbildung 5.10 zu sehen. Auffällig ist, dass der Unterschied für jedes Symbol anders ausfällt, also kein konstanter Faktor zu beobachten ist. Der Grund hierfür liegt darin, dass sich einige Funktionen der Controller-Interfaces nur schwer oder auch gar nicht mit Zugriff auf das Browser-Interface implementieren lassen. Ein Beispiel ist die Erstellung einer neuen Konferenz, bei der im Erfolgsfall ein Objekt zurückgegeben wird, welches die Daten der erstellten Konferenz repräsentiert und somit auch eine von der Datenbank vergebene ID enthält. Diese interne ID ist im Browser nicht sichtbar, wird jedoch für einige andere Methoden benötigt. Aus diesem Grund ist im Selenium-Code ein Aufruf der Methode `getConference` des Backend-Adapters notwendig. Häufen sich diese Art von Abfragen, so nähert sich die Laufzeit der Symbole von Frontend- und Backend-Adapter. Gleichzeitig sinkt aber die Aussagekraft des gewünschten Vergleichs, wenn zu viele Abfragen nicht mehr über Selenium getestet werden.

5.6.6 Lokalisierung einer Fehlerursache

Die in Abschnitt 5.6.4 beschriebene Diskrepanz zwischen Verhalten der Geschäftslogik und Anzeige im Webclient bedeutet ein potenzielles Sicherheitsrisiko, da es einem Nutzer mit Zugriff auf das Backend möglich wird, Aktionen durchzuführen, für die er keine Rechte besitzt. Aus diesem Grund ist es hilfreich, die entsprechende Änderung im Quellcode zu finden. Mit Hilfe dieser Information ist nicht nur eine schnelle Behebung des Fehlers möglich, sie erlaubt zudem eine bessere Einschätzung des Gefährdungspotenzials auf andere Funktionen im OCS.

Da sich das korrekte Verhalten effizient mit einer einzelnen Membership Query testen lässt, kann bei der Suche nach der Fehlerursache auf die Erstellung eines vollständigen Modells verzichtet werden, was die zum Auffinden benötigte Dauer stark verkürzt. Es genügt daher, das Git-Repository auf die zu untersuchende OCS-Revision zurückzusetzen, diese Version zu starten und die entsprechende Query an das System zu stellen. In diesem Fall besteht sie aus den beiden Symbolen `SP` (Submit Paper) und `BD` (Bidding), die in dieser Reihenfolge an das laufende OCS gesendet werden. Handelt es sich um eine fehlerhafte Version, so können beide Symbole erfolgreich ausgeführt werden. Bei einer OCS-Version ohne diesen Fehler ist das Symbol `SP` erfolgreich, das Bidding schlägt jedoch fehl.

Ein mögliches Problem bei der Art, wie der `ApiAdapter` mit dem OCS kommuniziert, ist in diesem Fall die Inkompatibilität der einzelnen Java-Klassen. Für die in den vorhergehenden Abschnitten beschriebenen Verfahren existierte für jede größere Release-Version des OCS eine eigene Variante des entsprechenden Adapters. Dies ist notwendig, da sich der OCS in konstanter Entwicklung befindet und einige Schnittstellen mit der Zeit an neue Funktionswünsche angepasst werden müssen. Da bei einem Aufruf per JNDI Server- und Client-Code zusammenpassen müssen, verwendet jeder Adapter eigens definierte Abhängigkeiten zu den entsprechenden Bibliotheken des OCS.

Ändert sich die API des OCS zwischen zwei Versionen nicht, so sind auch die Adapter weiter funktionsfähig. Aus diesem Grund ist es möglich, auch einzelne Revisionen des Entwicklungszweigs mit Hilfe der existierenden Adapter zu testen. Treten Inkompatibilitäten auf, kann testweise der Adapter der nachfolgenden bzw. vorherigen Release-Version verwendet werden. Hilft auch das nicht, muss ein zusätzlicher `ApiAdapter` erzeugt werden. Der

notwendige Aufwand dafür liegt meist bei wenigen Minuten, da sich die meisten Funktionen nicht von denen eines ähnlichen Adapters unterscheiden und die notwendigen Dateien daher größtenteils fast unverändert übernommen werden können.

Eine weitere Möglichkeit ist der Einsatz des in Abschnitt 5.6.4 beschriebenen Selenium-Adapters. Unterschiede in der API liegen hier nicht vor, da jeweils die für Integrationstests geschrieben Klassen der jeweiligen OCS-Revision verwendet werden können. Natürlich eignet sich dieser Ansatz aber nicht für Fehler, bei denen wie in diesem Fall ein Unterschied zum Verhalten der Geschäftslogik besteht. Weiterhin ist diese Art von Adapter wie in Abschnitt 5.6.5 beschrieben deutlich langsamer, was jedoch bei einer einzelnen Membership Query nicht groß ins Gewicht fällt.

Da ein Einsatz des Selenium-Adapters aufgrund der Art des Fehlers nicht in Frage kommt, wurde die Suche mit dem `ApiAdapter` der OCS-Version 1.5.0 durchgeführt. Im Gegensatz zu dem in Abschnitt 4.6.5 beschriebenen Vorgehen evaluiert das Verfahren die einzelnen Revisionen in diesem Fall rückwärts bis zur Version 1.4.0, welche den Fehler nicht aufweist.

Bei der Durchführung der Suche traten zwei Probleme auf. Das erste bestand aus einer Revision, welche durch eine Zusammenführung verschiedener Entwicklungszweige entstand. Durch diese Zusammenführung enthielt die Revision mehrere tausend Zeilen veränderten Code, darunter auch jene Änderung, welche den Fehler verursachte. Dem Resultat konnte jedoch auf diese Weise nicht entnommen werden, welche der vielen Änderungen schlussendlich verantwortlich war. Aus diesem Grund wurde diese zusammengeführte Revision in einem zweiten Durchlauf schlichtweg ignoriert und die Suche mit der nächsten Revision fortgesetzt.

Das zweite Problem trat exakt zu dem Zeitpunkt auf, als die gesuchte Revision gefunden wurde. An dieser Stelle erzeugte das fehlgeschlagene Bidding einen Ausnahmefehler, genauer gesagt eine OCS-eigene `SecurityException`. Da sich diese Klasse jedoch zwischen den Versionen 1.4.0 und 1.5.0 geändert hatte, schlug eine Serialisierung fehl und der Testvorgang brach ab. Dies konnte einfach behoben werden, indem ab der Revision mit der geänderten Klasse der Adapter der Version 1.4.0 verwendet wurde, welcher an dieser Stelle problemlos funktionierte.

Insgesamt überprüfte das Verfahren 36 Revisionen, bis es auf diejenige stieß, welche den Fehler aus Version 1.5.0 nicht aufweist. Jeder Test benötigte 3–4 Minuten, insgesamt dauerte der Vorgang weniger als zwei Stunden. Als Ergebnis wurde eine Revision vom 19. August 2011 ausfindig gemacht, die Änderungen an fünf verschiedenen Klassen enthält. Die Anzahl angepasster Zeilen ist diesmal mit 20 sehr überschaubar, sodass der Grund für den Fehler jetzt schnell entdeckt wurde und behoben werden konnte.

5.6.7 Testen von Änderungen der Systemumgebung

Mit wachsenden Anforderungen an eine Webanwendung kann sich die gewählte Plattform mit der Zeit als leistungstechnisch unterdimensioniert herausstellen, sodass die Anwendung auf ein schnelleres System migriert werden muss. Dies kann auch bei ausreichender Hardwareleistung notwendig sein, etwa wenn eine Neuinstallation oder ein Upgrade des Betriebssystems erforderlich ist, um weiterhin mit Sicherheitsupdates versorgt zu werden. An dieser Stelle ist es hilfreich zu wissen, ob sich die Anwendung noch wie erwartet auf dem neuen System verhält, da sich sowohl die installierte Software als auch der betreuende Administrator ändern können.

Der ACQC-Ansatz leistet hier wertvolle Dienste, da die erzeugten Modelle in den meisten Fällen keine Unterschiede zwischen dem alten und dem neuen System aufweisen sollten. Weiterhin kann ein kontinuierlicher Lernvorgang als Stresstest verwendet werden, um das

Tabelle 5.2: Auszug aus dem Server-Logfile des OCS, welcher den Nichtdeterminismus aufzeigte

Nr	Zeitpunkt	Dauer	Benutzer	Methode	Erfolg
88733	00:32:00,930	371	PC Member	`bidForPaper`	true
88734	00:32:01,247	45	PC Chair	`assignReviewer`	true
88735	00:31:55,442	-5838	PC Chair	`stopAssignmentPhase`	true
88736	00:31:58,287	12	PC Member	`submitReviewReport`	false

Systemverhalten unter hoher Last zu überwachen.

Beim OCS handelt es sich um eine Java-Enterprise-Anwendung, die nur einen installierten Applikationsserver, eine PostgreSQL-Datenbank, ein aktuelles Java Development Kit (JDK) sowie ein LaTeX-System voraussetzt. Dennoch konnte beim Erzeugen der Modelle auf einem Rechner, der von einem anderen Administrator als die restlichen betrieben wurde, ein Fehler aufgedeckt werden, der unabhängig von diesen Komponenten ist.

Das Problem fiel auf, als ein lang laufender Lernvorgang mit der Meldung abbrach, dass eine Membership-Query beim erneuten Auswerten ein anderes Ergebnis zurücklieferte als zuvor. Da sich der OCS deterministisch verhält, was eine Grundvoraussetzung für das aktive Automatenlernen ist, wurde diese Meldung genauer untersucht. Dazu wurde die genannte Query mehrere tausend Mal aufgerufen, lieferte nun jedoch immer dasselbe Ergebnis. Der Verdacht lag also nahe, dass ein spezielles Ereignis auf dem Testsystem zu dem Fehler geführt hat. Der OCS verfügt zu diesem Zweck über ein präzises Logging aller durchgeführten Aktionen, sodass die Abfolge zum Zeitpunkt der Meldung rekonstruiert werden konnte.

Tabelle 5.6.7 zeigt den entsprechenden Aktionsablauf. Beginnend mit Aktion Nr. 88733 gibt der PC Member ein Bidding ab und wird dann diesem Paper zugewiesen (88734). Der PC Chair beendet im Abschluss die Assignment-Phase (88735), was einen automatischen Start der Review-Phase zur Folge hat. Nun sollte der PC Member in der Lage sein, den Report einzureichen, allerdings schlägt diese Aktion fehl (88736).

Auffällig an diesem Auszug ist der Zeitstempel der jeweiligen Aktionen: Obwohl jeder genannte Methodenaufruf erst dann durchgeführt wurde, nachdem der vorherige abgearbeitet ist, liegen die Zeitpunkte der ersten beiden Zeilen zeitlich hinter denen der letzten beiden Zeilen. Eine weitere Auffälligkeit ist die Dauer der Methode `stopAssignmentPhase`, die einen negativen Wert aufweist. Da negative Ausführungszeiten unmöglich sind, stand fest, dass hier eine Fehlkonfiguration der Systemuhr vorlag.

Tatsächlich wies der Server bei erster Inbetriebnahme eine starke Abweichung von 36 Minuten auf. Der Administrator wurde daraufhin gebeten, dies dauerhaft zu korrigieren. Er richtete daraufhin einen periodischen Mechanismus ein, welcher die Systemuhr einmal pro Tag auf den korrekten Stand setzt. Da jedoch die interne Uhr des Servers anscheinend eine starke Abweichung aufweist, bewegen sich die entstehenden Sprünge im Bereich von mehreren Sekunden.

Mit diesem Wissen ließ sich nun der Ablauf leicht rekonstruieren: Die Assignment-Phase wurde nach 00:32 Uhr beendet. Ab diesem Zeitpunkt ist es dem PC Member möglich, einen Report einzureichen. Da jedoch vor dieser Aktion die Uhr auf 00:31 Uhr zurückgestellt wurde, war die Aktion noch nicht gültig und führte zu einem Fehlschlag.

Die Abhilfe für diesen Fehler ist einfach umzusetzen: Ein Dienst wie der *Network Time Protocol daemon* (ntpd) ist in der Lage, die Systemuhr dauerhaft auf der korrekten Zeit zu halten, ohne große Sprünge zu verursachen. Dies wird erreicht, indem etwa die interne Uhr

künstlich verlangsamt wird, um einem Vorauseilen entgegenzuwirken.

Auf den ersten Blick mag es so aussehen, als könnte eine solche Fehlkonfiguration keine Auswirkungen auf den realen Produktivbetrieb haben, schließlich wird ein Report nicht innerhalb weniger Sekunden verfasst und übermittelt. Ausgehend von der initialen Abweichung von 36 Minuten ist dieses Szenario jedoch wieder wahrscheinlicher und ein Nutzer könnte trotz aktiver Review-Phase daran gehindert worden sein, einen Report einzureichen.

5.6.8 Erkennung von seltenen Fehlern durch kontinuierliches Lernen

Ein erwünschter Nebeneffekt eines kontinuierlichen Lernvorgangs ist die Tatsache, dass auch selten auftretende Fehler, die nur in einer bestimmten Datenkonstellation auftreten können, entdeckt werden. In diesem Abschnitt wird ein solcher Fehler beschrieben, der im OCS aufgefallen ist.

Nutzer im OCS können in mehreren Konferenzen gleichzeitig agieren, ein Rechtesystem schränkt die möglichen Aktionen je nach Rolle des Nutzers in der Konferenz ein. Die Berechnung dieser Rechte ist eine komplexe Aufgabe und erfordert Rechenleistung sowie Zeit, sodass das Ergebnis in einem Cache festgehalten wird. Es handelt sich dabei um eine Optimierung, welche die Bedienung des OCS wesentlich beschleunigt, ohne die Funktionalität einzuschränken. Da sich die Rechte nur selten ändern, werden sie nur nach speziellen Aktionen oder einer gewissen Zeit neu berechnet. Dies funktioniert jedoch nur korrekt, wenn wirklich alle Aktionen, welche eine Rechteänderung hervorrufen können, beachtet wurden.

Dass diese Vorbedingung verletzt wurde, zeigte folgende Abfolge von Aktionen während des Lernvorgangs:

SP ES EU BD SA EB EA SR → success
SP ES EU BD SA EB EA SR → error

Die beiden Anfragen sind auf den ersten Blick identisch, liefern jedoch unterschiedliche Ergebnisse. Ein hundertfacher Durchlauf dieser einzelnen Membership Query ergab jedoch in allen Fällen ein *success*.

Eine weitere Untersuchung zeigte, dass die zweite Zeile vom Reuse-Filter auf einer Konferenz ausgeführt wurde, die zuvor die ersten vier Symbole evaluierte und dann die Aktion *Submit Report* (SR) aufrief. Dies kann jedoch an dieser Stelle nicht ausgeführt werden, da die Review-Phase zu diesem Zeitpunkt noch nicht gestartet ist. Das Einreichen eines Reviews ist also erst möglich, wenn die Assignment-Phase beendet wurde (EA), wie der erfolgreiche Ablauf in der ersten Zeile zeigt. Damit ergibt sich eine leicht veränderte Sicht auf die gestellten Membership Querys:

SP ES EU BD SA EB EA SR → success
SP ES EU BD SR SA EB EA SR → error

Der Reuse-Filter geht davon aus, dass fehlgeschlagene Aktionen keine Auswirkung auf den weiteren Ablauf der Konferenz haben, da ja keine Änderung vorgenommen wurde. Infolgedessen kann die Konferenz so verwendet werden, als wäre das Symbol nie ausgeführt wurden. In oben gezeigten Fall war diese Annahme jedoch nicht korrekt, da das Recht (oder auch das Fehlen desselben) im Cache verbleibt, nachdem ein Versuch unternommen wurde, ein Review einzureichen. Da der Cache nicht zurückgesetzt wird, wenn die Phase Assignment-Phase beendet wird, erlaubt der OCS auch beim zweiten Versuch nicht, die Aktion durchzuführen.

Aufgrund der automatischen Invalidierung des Caches nach einiger Zeit hat dieses Pro-

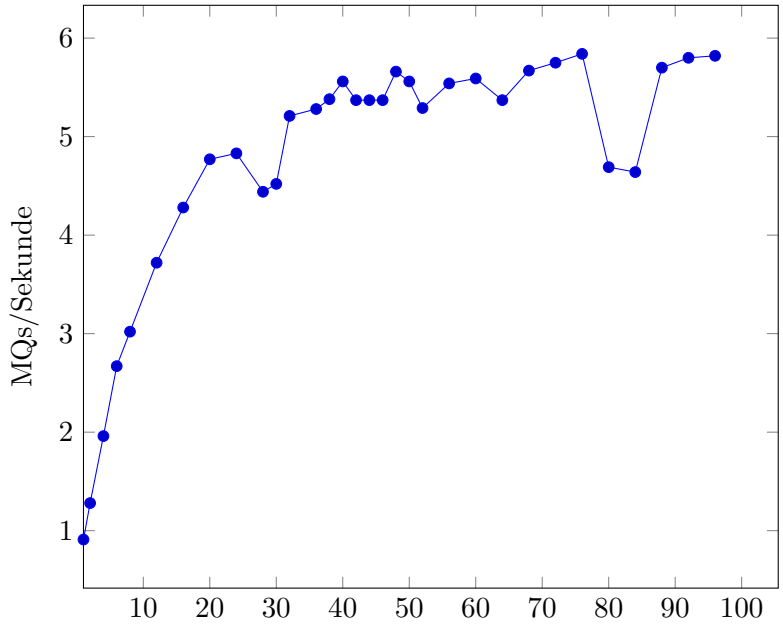

Abbildung 5.11: Geschwindigkeit eines Lernvorgangs des OCS in MQs/Sekunde auf einem Rechner mit 16 Kernen, abhängig von der Anzahl der genutzten Threads

blem allerdings nur eine untergeordnete Relevanz für den Produktivbetrieb des OCS. Eine einfache und doch effektive Lösung für zukünftige Lernvorgänge bestand darin, den Cache beim Abmelden des Nutzers zu löschen. Wird eine Konferenz vom Reuse-Filter dann erneut verwendet, werden die Rechte neu ausgewertet und das Problem kann nicht mehr auftreten.

5.6.9 Einfluss paralleler Lernvorgänge

Beim automatisierten Lernen von komplexen Systemen wie dem OCS ist die benötigte Zeit der dominierende Faktor. Je schneller ein Modell erstellt werden kann, desto eher kann es für einen Vergleich mit der zuletzt als funktionierend bekannten Version verwendet werden. Gerade in kritischen Entwicklungsphasen, etwa kurz vor einer bevorstehenden Veröffentlichung, kann sich daher eine Optimierung der Lerngeschwindigkeit auszahlen. Auch ein kontinuierlicher Lernvorgang profitiert von einer höheren Geschwindigkeit, lassen sich doch auf diese Weise eventuell mehr Modellzustände im aktuellen Testzyklus entdecken.

Eine naheliegende Art der Leistungssteigerung ist vor dem Hintergrund der aktuellen Mehrprozessorarchitekturen das parallele Lernen eines Systems [HBM+12]. Dabei greifen mehrere Threads gleichzeitig auf das SUL zu und liefern das Ergebnis an den Lernalgorithmus. Voraussetzung hierfür ist jedoch, dass das SUL parallele Anfragen verarbeiten kann. Im Falle des OCS trifft dies nur für die Remote-Beans zu, der Adapter zur Weboberfläche beherrscht diese Art des Zugriffs nicht, auch wenn die Webanwendung natürlich zeitgleich von mehreren Personen verwendet werden kann.

Abbildung 5.11 zeigt die Geschwindigkeit der Lernvorgänge für den OCS in Version 1.10.4. Jede Iteration benötigt etwa 20.130 Membership Querys und, je nach Anzahl der verwendeten Threads, zwischen einer und neun Stunden. Die verwendeten Prozessoren sind zwei AMD Opteron 6136 mit jeweils acht physikalischen Kernen, welche mit bis zu 2.400 MHz getaktet werden können.

Wenig überraschend steigt die Geschwindigkeit zu Beginn rasant an, eine Verdoppelung der Threads bewirkt teilweise eine 50%ige Beschleunigung. Sobald die Anzahl der parallelen Abläufe doppelt so groß ist wie die Anzahl der verfügbaren Prozessoren, flacht die Kurve jedoch schnell ab und bewegt sich zwischen fünf und sechs Membership Querys pro Minute. Eine weitere Erhöhung bringt demnach keine größeren Zeitersparnisse.

Die Schwankungen in Abbildung 5.11 entstehen dadurch, dass bei einer parallelen Ausführung nicht vorhergesagt werden kann, in welcher Reihenfolge die Schritte abgearbeitet werden. Dadurch verändert sich die Trefferwahrscheinlichkeit des Caches des Reuse-Filters, sodass unter Umständen mehr Membership Querys an das SUL gestellt werden. Unter Umständen ist es zudem möglich, dass periodische Arbeiten auf dem Test-Server das System zusätzlich belastet haben und so für eine Senkung der Leistung führten.

6 Alphabetmodellierung mit dem Java Application Building Center

Damit das in den vorigen Kapiteln besprochene Lernverfahren durchgeführt werden kann, sind Alphabete aus Symbolen notwendig, welche die Kommunikation mit dem zu lernenden System ermöglichen. Diese Symbole werden im einfachsten Fall manuell entwickelt, jedoch hat dieser Ansatz einige Nachteile. Im Folgenden wird daher die Modellierung von Alphabeten mit einem Prozessmodellierungswerkzeug als Alternative vorgestellt.

6.1 Nachteile manuell erstellter Alphabete

Insbesondere zu Beginn der Erstellung eines neuen Testtreibers für ein zu lernendes System wird diese Entwicklung primär direkt in der jeweiligen Programmiersprache durchgeführt. Einzelne Symbole können auf diese Weise schnell darauf überprüft werden, ob sie ihre Aufgabe korrekt erfüllen. Automatisierte Modultests, bestehend aus mehreren Anfrageworten samt erwarteter Rückgabe, ermöglichen währenddessen eine Kontrolle der Kombination mehrerer Symbole.

Problematisch wird diese Art der Entwicklung spätestens, wenn Symbole in einem anderen Kontext verwendet oder auf eine andere Art kombiniert werden sollen. Mit geschickt gewählten Vererbungshierarchien in objektorientierten Sprachen lässt sich zwar der Anteil von dupliziertem und somit schwer wartbarem Quellcode reduzieren. Der programmierte Code wird jedoch spätestens dann unübersichtlich, wenn dasselbe Symbol in unterschiedlichen Alphabeten unterschiedliches Verhalten zeigen soll. An dieser Stelle ist eine Parametrisierbarkeit der Symbole erforderlich, sodass bei Definition des Alphabets das gewünschte Verhalten spezifiziert werden kann.

Da nicht nur die Symbole, sondern auch die Alphabete in der jeweiligen Programmiersprache erstellt werden, kann für jede Änderung ein erneutes Kompilieren notwendig sein. Sind mehrere Entwickler oder Tester involviert, steigt auch das Risiko für Konflikte im Quellcode. So kann es passieren, dass eine sehr hohe Parametrisierbarkeit notwendig ist, obwohl die meisten Verwender eines Symbols nur einen Bruchteil des Funktionsumfangs benötigen.

Ein weiteres Problem liegt in den technischen Fähigkeiten der mit der Qualitätskontrolle beauftragten Personen. Diese müssen nicht zwingend über Zugriff auf den Quellcode oder gar Kenntnisse der entsprechenden Programmiersprache verfügen. Vielmehr sind sie als sogenannte *Applikationsexperten* mit der detaillierten Arbeitsweise des Systems vertraut und stellen daher die bestmögliche Wahl für die Erstellung von Testfällen dar. Im oben beschriebenen Fall sind diese Experten jedoch von der Arbeit der Entwickler zu abhängig: Nicht nur die Erstellung anpassbarer Symbole, sondern auch die Anpassung von Alphabeten muss bei diesen in Auftrag gegeben werden, was zu deutlichen Verzögerungen führen kann. Wünschenswert wäre daher, wenn die Applikationsexperten möglichst unabhängig von den Entwicklern Alphabete erstellen und diese auch konfigurieren könnten.

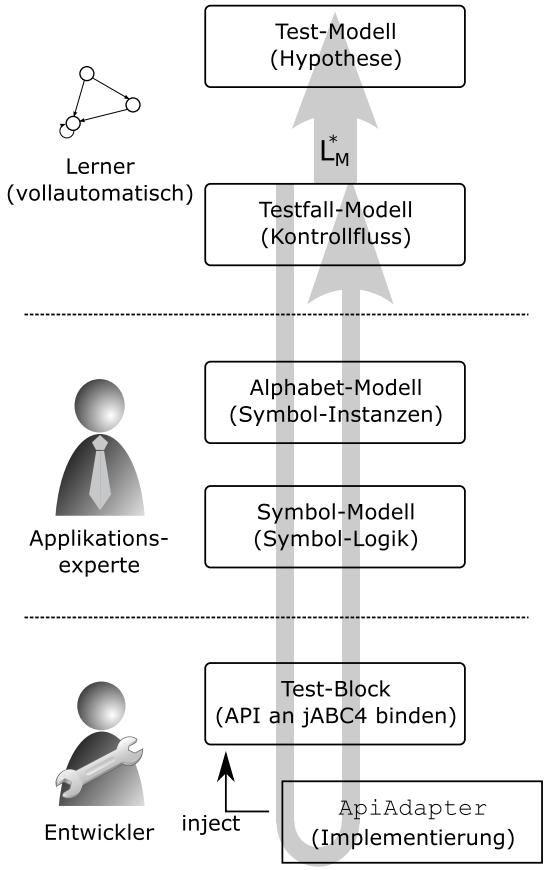

Abbildung 6.1: Modellschichten für die Modellierung von Alphabeten durch einen Applikationsexperten (basiert auf [NWS14])

6.2 Modellierung durch Applikationsexperten

Um die im vorigen Abschnitt genannten Probleme zu entschärfen, bietet sich eine serviceorientierte [MSR05] Lösung an, die einem Applikationsexperten erlaubt, Alphabetmodelle ohne die Hilfe eines Entwicklers anzupassen oder gar komplett neu zu erstellen. Dies kann erreicht werden, indem die Programmierung der Symbole von der Erzeugung der Alphabete abgekoppelt wird. Während die Applikationsexperten so in der Lage sind, Alphabetmodelle in einer grafischen Umgebung zu erstellen oder zu konfigurieren, liefern die Entwickler weiterhin die dafür benötigten Symbole. Ein solches Vorgehen steigert die Einfachheit für die erstgenannte Gruppe, was zur Folge hat, dass ein größerer Personenkreis mit dieser Aufgabe betraut werden kann. Dies führt im Endeffekt zu einer besseren Verteilung der zur Verfügung stehenden Ressourcen [MFS11] und kann positive Auswirkungen auf die Agilität der Produktentwicklung haben [MS09, MS10].

Abbildung 6.1 zeigt die Einbindung der verschiedenen Personengruppen. Wie gehabt wird auf der untersten Ebene von einem `ApiAdapter` Gebrauch gemacht, welcher für die Kommunikation zum SUL verantwortlich ist. Da sich an dieser Stelle keine Änderung ergibt, kann derselbe Adapter verwendet werden, der auch bei den manuellen Symbolen zum Einsatz kommt – ein zusätzlicher Implementierungsaufwand ist hier also nicht notwendig.

Damit der Applikationsexperte möglichst wenig mit der API direkt arbeiten muss, erstellt der Entwickler zusätzlich *Test-Blöcke*, welche auf den `ApiAdapter` zugreifen und eine stabile Abstraktion der API darstellen. Diese Blöcke werden jedoch schon nicht mehr im Quellcode entwickelt, sondern in einer grafischen Software zur Prozessmodellierung zusammengestellt. Auf diese Weise ist es auch Applikationsexperten, die über wenig oder keine Kenntnisse der API-Funktionalität verfügen, möglich, eigene Test-Blöcke zu entwerfen.

Basierend auf den Test-Blöcken ist der Applikationsexperte nun in der Lage, Symbole nach eigenen Vorstellungen zu entwerfen. Existieren beispielsweise für den in Kapitel 5 beschriebenen OCS zwei Blöcke zum Erstellen von Konferenzen und dem Einreichen eines Papers, so lassen sich diese zu einem Symbol zusammenfassen, welches als Reset verwendet werden kann. Auf diese Weise lassen sich in kürzerer Zeit nur die Konferenzen testen, welche über mindestens ein eingereichtes Paper verfügen.

An dieser Stelle wird ein weiterer Vorteile dieses Ansatzes deutlich: Der Datenfluss zwischen den einzelnen Komponenten ist nicht im Code spezifiziert, sondern kann auf Modellebene angepasst werden. Auf diese Weise ist es im obigen Beispiel möglich, das Paper genau in der gerade erstellten Konferenz einzureichen, obwohl beide Blöcke so gestaltet wurden, dass sie möglichst unabhängig voneinander sind.

Sind alle notwendigen Symbolmodelle erstellt, kann der Applikationsexperte daraus im nächsten Schritt Alphabetmodelle erzeugen. Dazu kombiniert er mehrere Symbole in einem Alphabetmodell, ohne jedoch den Kontrollfluss, also die Ausführungsreihenfolge, festzulegen. Stattdessen spezifiziert er ausschließlich den Datenfluss, etwa die von den Symbolen erzeugten Objektinstanzen. Um bei dem Beispiel des Resets zu bleiben, würde dieser sowohl eine Konferenz als auch ein Paper erzeugen, welche beide von einem Symbol *Report für ein Paper einreichen* benötigt werden.

Wird nun anhand eines solchen Alphabetmodells ein Lernvorgang durchgeführt, so stellt der Lernalgorithmus wie gehabt aus Membership Querys bestehende Anfragen an das System. Anhand des im Alphabetmodell spezifizierten Datenflusses wird dafür zu Beginn automatisch Code generiert, der bei jeder Anfrage verwendet werden kann. Aus dem Ergebnis dieser Anfragen kann dann schlussendlich ein Hypothesenmodell generiert werden, welches das Verhalten des SUL möglichst genau abbildet.

6.3 Verwendung des Java Application Building Centers

Für die Alphabeterstellung kommt das in Abschnitt 2.5.3 beschriebene jABC zur Anwendung. Der generelle Aufbau des Editors ist in Abbildung 6.2 gezeigt. Auf der linken Seite sind in der oberen Hälfte der Projekt-Browser, die verfügbaren Graphen sowie die projektspezifischen Dienste (in diesem Fall für den OCS) angeordnet. Im Screenshot sind mehrere für den OCS verwendete Graphen, sogenannte *Service Logic Graphs* (SLGs), zu sehen, etwa der markierte mit dem Namen *EndPhase* zum Beenden von Phasen einer Konferenz. Der Abschnitt im linken unteren Teil zeigt die verfügbaren *Inspektoren*, mit deren Hilfe sich die Eigenschaften des gerade ausgewählten SLGs oder dessen Komponenten anpassen lassen. In der Abbildung sind derzeit die Attribute des genannten SLG sichtbar, etwa der Name, das Java-Package oder das verwendete Icon. Darunter folgt eine Liste der Variablen im Kontext des Graphen wie beispielsweise die übergebene Konferenz oder in diesem Fall die zu beendende Konferenzphase.

Das Hauptfenster auf der rechten Seite stellt die Struktur des SLG graphisch dar und zeigt anschaulich den festgelegten Kontrollfluss. Mit einem Klick auf die Variablen im Inspektor

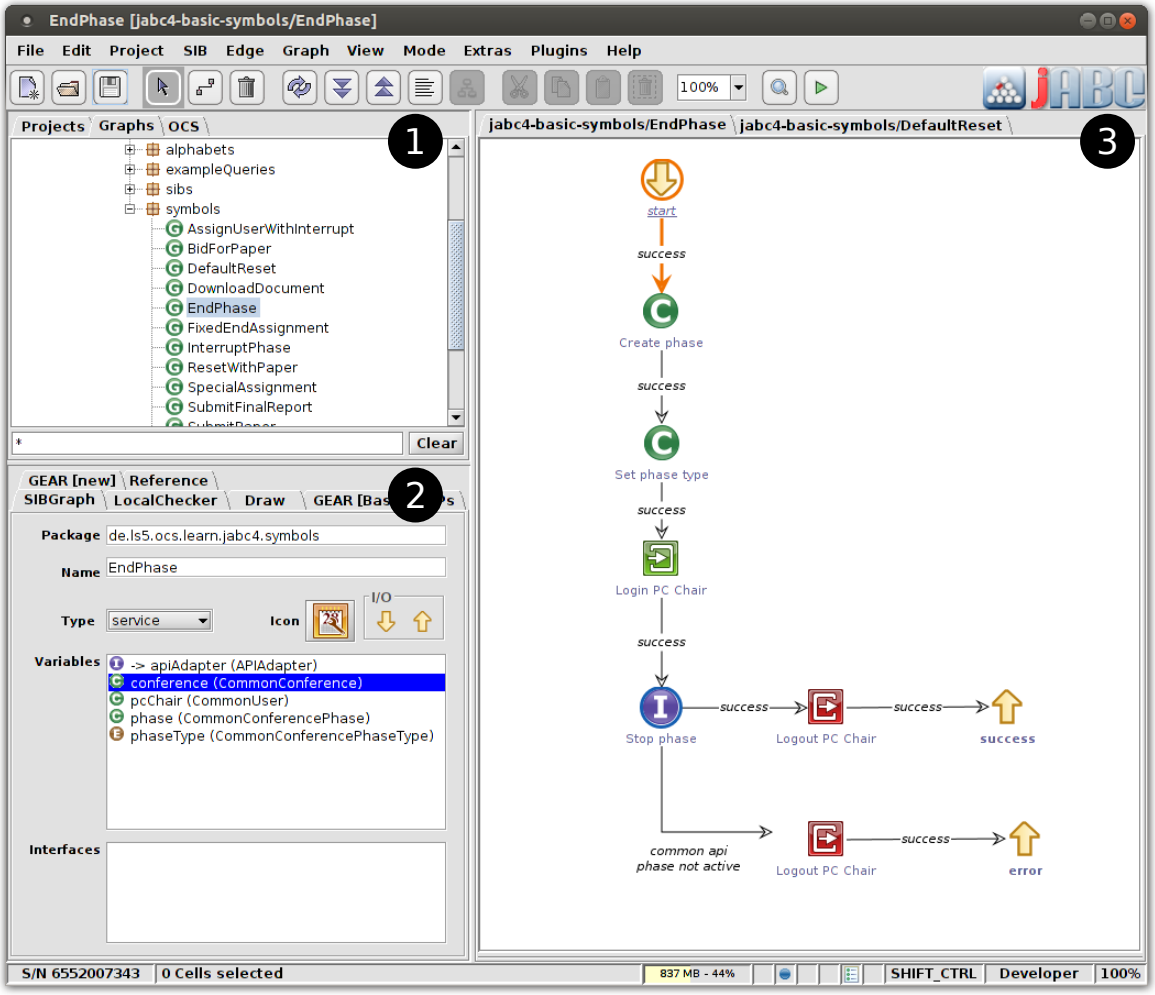

Abbildung 6.2: Screenshot des jABC4. Quelle: [NWS14]

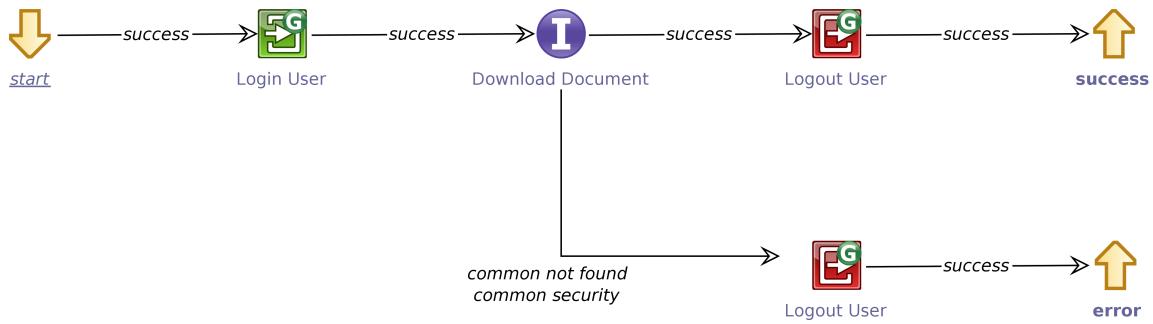

Abbildung 6.3: SLG zum Herunterladen eines Dokuments als ein bestimmter Nutzer. Quelle: [NWS14]

lässt sich zudem der Datenfluss visualisieren, im gezeigten Beispiel wird etwa der Wert von `conference` zu Beginn des Graphen geschrieben (orange) und bei Ausführung des SIBs `Stop phase` gelesen (blau). Ein weiteres wichtiges Detail hierbei ist, dass die Variable `apiAdapter` erst zur Laufzeit durch Dependency Injection mit einer Implementierung belegt wird, was einen Austausch der verwendeten Komponente ohne Änderung des Graphen ermöglicht.

6.4 Entwicklung von technischen SLGs

Um die API eines Systems mit Hilfe wiederverwendbarer SLGs abzubilden, kombiniert ein Entwickler diese API-Aufrufe in technischen SLGs. Ein Beispiel für einen solchen Graphen ist in Abbildung 6.3 dargestellt.

Hierbei ist zu beachten, dass der Graph aus verschiedenartigen Komponenten besteht. Sowohl der Beginn als auch das Ende des Graphen sind mit Input- respektive Output-Knoten gekennzeichnet, die jeweils Daten entgegennehmen beziehungsweise zurückgeben können. Während die Knoten zum An- und Abmelden des Nutzers aus bereits existierenden SLGs bestehen, ruft das SIB *Download Document* eine Methode des in Abschnitt 5.3 beschriebenen `ApiAdapter`s auf und stellt so eine Verbindung zum zu testenden System her.

Bei Ausführung des Graphen wird der Nutzer, als welcher das Dokument heruntergeladen werden soll, vom Startknoten im Kontext gespeichert. Da es sich hierbei um Datenfluss handelt, ist diese Aktion im dargestellten Kontrollfluss nicht sichtbar. Dafür erlaubt diese Konfigurierbarkeit des Datenflusses, den SLG mit verschiedenen Nutzern aufzurufen, etwa um zu testen, ob ein Dokument sowohl als Autor als auch als PC Member verfügbar ist. Abhängig vom Erfolg der Methode gibt der Graph anschließend den Wert `success` oder `error` zurück, nachdem der Nutzer ordnungsgemäß abgemeldet wurde.

Zu unterscheiden sind im Fehlerfall die erwarteten Exceptions und solche, die auf schwerwiegende Probleme hinweisen. Wie an der vom *Download Document*-SIB ausgehenden Kante zu sehen, wird diese bei zwei Exceptions weiter verfolgt: Tritt eine `CommonNotFoundException` auf, so ist davon auszugehen, dass bisher kein Dokument hochgeladen wurde. Im Falle einer `CommonSecurityException` verfügt der aufrufende Nutzer nicht über das Recht, ein existierendes Dokument herunterzuladen. Beide Situationen führen dazu, dass der SLG den Nutzer ordnungsgemäß abmeldet und den Wert `error` zurückgibt.

Sollte allerdings ein anderer Fehler auftreten, so ist davon auszugehen, dass ein Problem mit dem SUL besteht, etwa ein Abbruch der Datenverbindung. In diesem Fall wird die Exception an den Lernalgorithmus weitergereicht, der den gesamten Lernvorgang mit einer

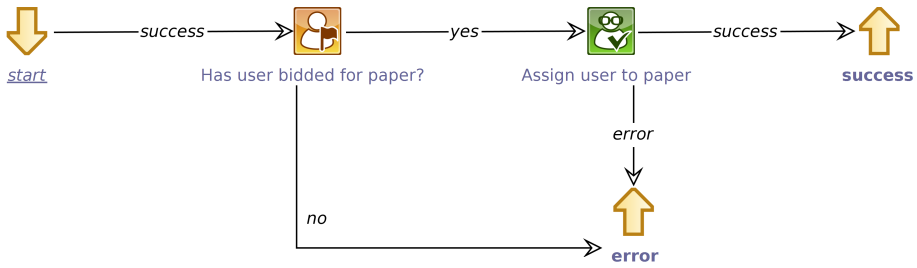

Abbildung 6.4: Applikationsspezifischer Graph für das *Special Assignment*, welches nur bei vorhandenem Bidding einen Nutzer zuweist. Quelle: [NWS14]

entsprechenden Fehlermeldung abbricht. Würde stattdessen der Graph in jedem Fehlerfall den Wert `error` zurückgeben, würden selbst kurzzeitige Aussetzer im System nicht entdeckt und das Lernergebnis auf diese Weise verfälscht werden.

6.5 Kombination in applikationsspezifischen SLGs

Mit den zuvor erstellten technischen SLGs ist der Applikationsexperte nun in der Lage, ohne Wissen über die API des zu lernenden Systems applikationsspezifische Graphen zu erstellen. Dazu kombiniert er die existierenden Bausteine, welche wiederum auch aus von ihm selbst erstellten Graphen bestehen können, und spezifiziert den notwendigen Kontroll- und Datenfluss im jABC4.

Ein Beispiel für einen solchen applikationsspezifischen Graphen ist in Abbildung 6.4 gezeigt. Es handelt sich um das in Abschnitt 5.2 beschriebene Symbol `SA` (Special Assignment), welches dafür sorgt, dass ein PC Member nur dann an ein Paper zugewiesen wird, wenn dieser zuvor ein entsprechendes Bidding abgegeben hat.

Im Gegensatz zu den in Abschnitt 6.4 beschriebenen technischen SLGs ist das hier verwendete Abstraktionsniveau wesentlich höher. Weder sind Zugriffe auf den `ApiAdapter` noch ein manuelles An- und Abmelden des betreffenden Nutzers notwendig. Zu beachten ist auch die Wahl der Kantenbeschriftungen: Zwar erwartet der zur Kompilierung des Alphabets verwendete Codegenerator die Rückgabewerte `success` und `error` bei allen Alphabetsymbolen, allerdings gilt diese Einschränkung nicht für Hilfsgraphen wie der in der Abbildung gezeigte SLG mit der Bezeichnung *Has user bidded for paper?*. Nicht nur die Beschriftung des Symbols macht den Graphen einfacher verständlich, auch die Rückgabewerte `yes` und `no` helfen dabei.

6.6 Auswahl eines Lernalphabets

Um das vom Lernalgorithmus verwendete Alphabet zusammenzustellen, legt der Applikationsexperte einen neuen Graphen an, in dem er die ausgewählten Symbole in Form von zuvor spezifizierten SLGs platziert. Abbildung 6.5 zeigt ein solches Alphabet, welches den Lernvorgang auf die Aktionen beschränkt, welche bis zum Einreichen eines Reports durch einen PC Member durchgeführt werden. Auffallend sind hier die vier identischen Symbole für das Beenden der Phase, anhand derer erkennbar ist, dass derselbe Graph mehrfach verwendet wurde und sich die Alphabetsymbole nur durch die Konfiguration unterscheiden.

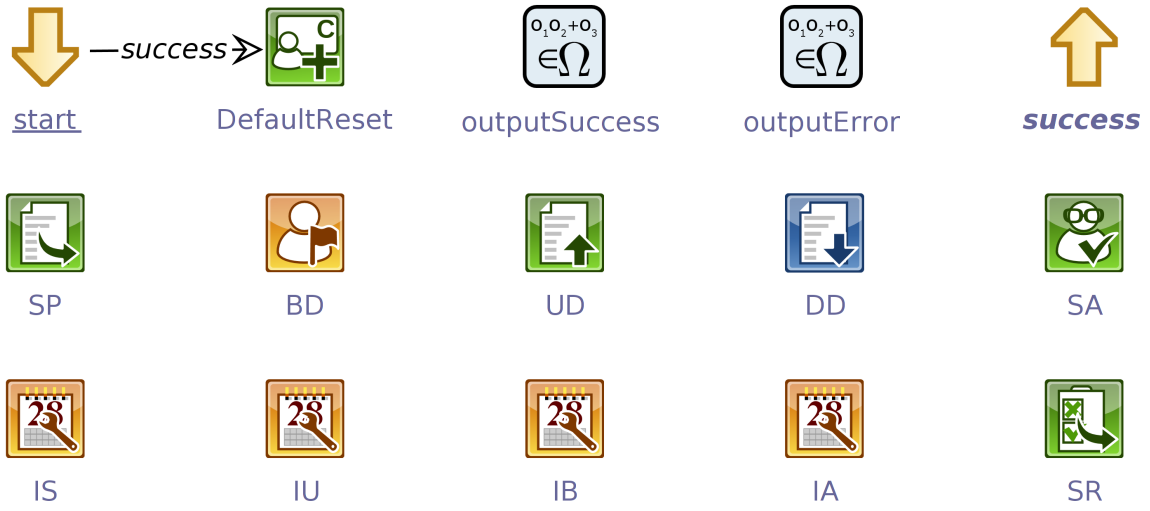

Abbildung 6.5: Auswahl von Alphabetsymbolen, welche den Konferenzablauf bis zum Einreichen eines Reports abdecken. Quelle: [NWS14]

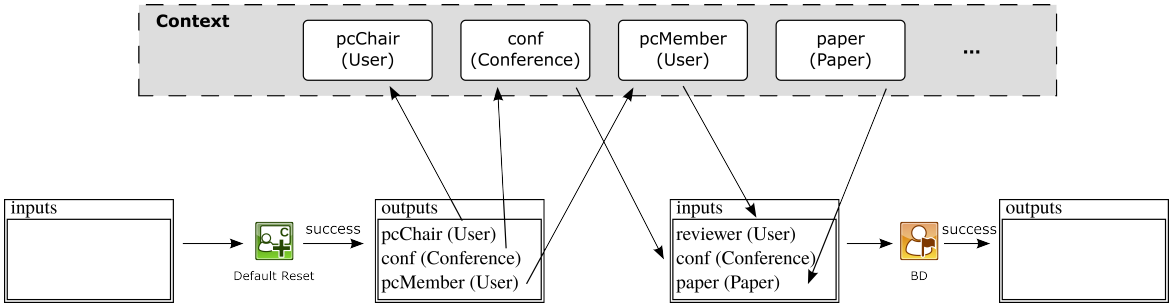

Abbildung 6.6: Exemplarische Parametrisierung für zwei Symbole des in Abbildung 6.5 gezeigten Alphabets. Quelle: [NWS14]

Im Gegensatz zu den in den vorigen Abschnitten gezeigten Graphen ist in Abbildung 6.5 kein Kontrollfluss spezifiziert, die einzige Kante ausgehend vom Startknoten definiert den zu verwendenden Reset. Der Grund hierfür liegt darin, dass die Reihenfolge der auszuführenden Symbole erst während des Lernvorgangs dynamisch vom Algorithmus bestimmt wird.

Von größerer Bedeutung an dieser Stelle ist daher der Datenfluss. Der Reset in diesem Beispiel erzeugt sowohl die Konferenz als auch die agierenden Nutzer mit ihren jeweiligen Rollen, die in jedem anderen Graph benötigt werden. Auch das Symbol *SP* (Submit Paper) erzeugt ein Objekt, welches an späterer Stelle verwendet wird. Im jABC4 ist der Datenfluss für jede Variable sichtbar, sobald diese angewählt wird. Leider ist diese Art der Darstellung für eine schriftliche Ausarbeitung wenig geeignet, sodass der Datenfluss exemplarisch in Abbildung 6.6 gezeigt ist.

Wird vom Lernalgorithmus eine Membership Query mit Reset[1] erzeugt, so werden im anzunehmenden Erfolgsfall die drei Variablen `conf`, `pcChair` und `pcMember` mit entsprechenden Objekten initialisiert. Der Wert der Variablen `paper` wird wie beschrieben nicht durch den Reset gesetzt, kann aber durch ein anderes Symbol geändert worden sein. Wird

nun im späteren Verlauf das Symbol *BD* (Bidding) ausgeführt, so greift es auf die Werte von `pcMember`, `conf` und `paper` zu. Die Namen der Variablen müssen nicht übereinstimmen, schließlich ist es möglich, dass das Symbol mit einem anderen Nutzer ausgeführt werden soll.

6.7 Automatisiertes Erzeugen von Membership Querys

Sind die manuellen Vorbereitungen von Entwicklern und Applikationsexperten abgeschlossen, geschieht der Rest des Lernvorgangs vollautomatisch, von der Generierung des notwendigen Java-Codes über die Erzeugung von Membership Querys bis hin zum fertig gelernten Modell.

Der erste Schritt auf diesem Weg besteht in der Generierung von Java-Code basierend auf den zuvor definierten Graphen. Dazu wird ein von Johannes Neubauer neu entwickelter Codegenerator verwendet, der eine Weiterentwicklung des Genesys-Plugins [Jö13] darstellt und dessen Vorgehensweise in [NWS14] im Detail beschrieben wird. Dieser erzeugt nicht nur typsicheren Quellcode, der Ein- und Ausgabewerte der Graphen korrekt einbindet, sondern unterstützt auch Java-Interfaces innerhalb der SLGs. Auf diese Weise ist es möglich, die verwendete Implementierung (in diesem Fall die zu lernende OCS-Version) erst zur Laufzeit festzulegen.

Damit die generierten Java-Klassen vom Lernalgorithmus ebenfalls typsicher verwendet werden können, lässt sich für das modellierte Alphabet ein sogenannter Interface-Graph angeben, der das gewünschte Verhalten spezifiziert. Im Falle der Alphabetmodelle ist dies zum Beispiel das Interface `LearnAlphabet`, welches unter anderem die Methode `execute()` bereitstellt. Diese Methode wird jedes Mal aufgerufen, wenn eine Membership Query auf dem Alphabet durchgeführt werden soll.

Wie in Abschnitt 6.6 beschrieben, verfügen die vom Applikationsexperten definierten Alphabetmodelle – abgesehen vom Reset, der vom Startknoten aus erreichbar ist – über keinerlei Kontrollfluss. Dieser wird stattdessen während des Lernvorgangs vom Algorithmus anhand der gestellten Membership Querys bestimmt. Während der Ausführung einer Query transformiert der sogenannte *Membership Query Builder*, welcher ebenfalls in Form eines SLG spezifiziert wurde, die vom Algorithmus generierten Anfragen in ausführbare Modelle mit dem entsprechenden Kontrollfluss.

Wie bereits erwähnt, können in den SLGs Interfaces verwendet werden, um die Implementierung zur Laufzeit austauschen zu können. Dies erlaubt Lernvorgänge mit verschiedenen Versionen des OCS, wobei jedoch immer derselbe generierte Code verwendet wird. Möglich wird diese Flexibilität durch Einsatz eines *Inversion of Control*-Containers (IoC), in diesem Fall die mit JavaEE 6 eingeführte *Contexts and Dependency Injection* (CDI).

Eine weitere durch das Interface `LearnAlphabet` definierte Methode ist `getSuccessReturn()`, welche das Ergebnis einer einzelnen ausgeführten Membership Query liefert. Anhand des Rückgabewerts kann das verwendete Membership-Orakel unterscheiden, welche Symbole des zu testenden Wortes erfolgreich ausgeführt wurden. Das Ergebnis landet im letzten Schritt beim Lernalgorithmus, der mit den erhaltenen Informationen das aktuelle Hypothesenmodell verbessern kann.

[1] Dies ist aufgrund der Verwendung des Reuse-Filters nicht zwingend notwendig. Bei einer Membership Query ohne Reset wurde dieser jedoch schon vorher ausgeführt und der Kontext entsprechend initialisiert.

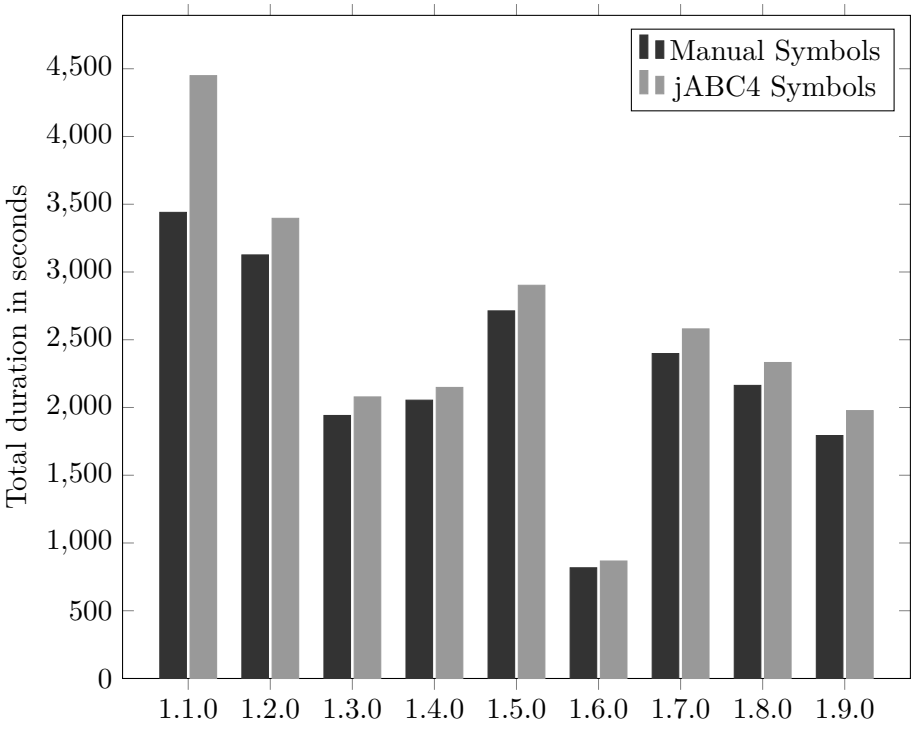

Abbildung 6.7: Laufzeit des Lernalgorithmus für jede Version des OCS, jeweils für manuell erstellte Symbole und solche, die mit dem jABC4 generiert wurden. Quelle: [NWS14]

6.8 Leistungsvergleich zu manuell erstellten Alphabeten

Der in diesem Kapitel beschriebene Zwischenschritt, der einen Applikationsexperten einbindet und dafür den für den Lernvorgang notwendigen Code automatisch generiert, wird in vielen Fällen einen negativen Einfluss auf die für eine Modellerstellung notwendige Dauer haben. Dies liegt in erster Linie daran, dass generierter Code in den meisten Fällen weniger optimiert ist als der eines erfahrenen Entwicklers. Die unkomplizierte Bedienung durch einen Nutzer ohne Programmiererfahrung wird daher durch gewisse Leistungseinbußen „erkauft“.

Uns interessierte die Frage, wie hoch diese Einbußen sind. Aus diesem Grund führten wir einen Lernvorgang mit dem in Abbildung 6.5 gezeigten Alphabet sowohl mit den manuell erstellten als auch den aus SLGs generierten Symbolen durch. Das Ergebnis zeigt der Graph in Abbildung 6.7.

Insgesamt liegt der zusätzlich notwendige Zeitaufwand demnach bei etwa 5-10% bei einer durchschnittlichen Gesamtdauer von ca. 40 Minuten. Eine genauere Betrachtung zeigt jedoch, dass der generierte Code nur etwa für 2% der Gesamtzeit verantwortlich ist – der Rest wird vom Lernalgorithmus sowie dem verwendeten `ApiAdapter` verbraucht. Der Hauptgrund für die längere Laufzeit liegt demnach in der stärkeren Modularisierung bzw. Wiederverwendbarkeit der erstellten Graphen. So erfordert etwa die in Abbildung 6.4 gezeigte Kombination zweier SLGs für jeden der Schritte ein An- und Abmelden des PC Chairs. Ein manuell erstelltes Symbol mit demselben Funktionsumfang könnte beide Aktionen innerhalb derselben Sitzung durchführen und ist daher schneller.

Die durch eine erhöhte Modularisierung verschlechterte Laufzeit ließe sich jedoch bereits

auf Graph-Ebene verbessern. So wäre es denkbar, häufig verwendete SLGs von einem Entwickler als optimierte technische SLGs erstellen zu lassen. Dieser hat dann die Möglichkeit, API-Aufrufe wie das An- und Abmelden eines Nutzers entsprechend zu verbessern und so den Performancenachteil auszugleichen.

6.9 Einsatzmöglichkeit: Risikoorientiertes Testen

Gerade bei umfangreichen Systemen wird die Qualitätskontrolle eines Produktes nicht selten von Personen durchgeführt, die über wenig oder keine Programmiererfahrung verfügen und nicht einmal Zugriff auf den Quellcode der Software haben. Ein Beispiel, wo der Ansatz der mit dem jABC4 generierten Modelle hilfreich sein kann, ist das in Abschnitt 2.3.12 beschriebene risiko-orientierte Testen. Es wird dazu benutzt, um Testfälle zu priorisieren, wenn aufgrund von Ressourcenbeschränkungen eine Durchführung aller Tests nicht effizient oder gar nicht möglich wäre. Der oben genannte Applikationsexperte wird in diesem Fall durch einen Risikoanalysten verkörpert, der entscheidet, welche Testfälle ausgewählt werden. Diese Entscheidung kann etwa aus dem Wissen um geänderte Funktionen, den Anforderungen des Kunden oder auch persönlicher Erfahrung getroffen werden.

Wichtig ist hierbei die Unterscheidung, dass der jABC4-basierte Ansatz nicht dafür eingesetzt wird, Testfälle zu priorisieren. Die dafür notwendigen Entscheidungen werden weiterhin vorab vom Risikoanalysten getroffen, der nun jedoch in der Lage ist, diese Ergebnisse in übersichtlichen Modellen festzuhalten und ohne weitere Hilfe der Entwickler die gewünschten Testfälle durchzuführen. Da die Größe eines Lernalphabets eine wesentliche Rolle in der für einen Lernvorgang notwendigen Dauer spielt, liegt die primäre Aufgabe weniger in der Erstellung von applikationsspezifischen Symbolen, sondern vielmehr in der Auswahl passender Alphabete.

Ist beispielsweise bekannt, dass sich für die aktuell zu testende Version nur bestimmte Funktionen geändert haben, so kann ein entsprechendes Alphabet, welches eben diese Symbole enthält, in einem Bruchteil der Zeit gelernt werden. Tauchen diese Funktionen erst spät im Verlauf der Applikation auf, so verhindert ein speziell hierfür entworfenes Reset-Symbol das unnötige Testen des vorherigen Ablaufs. Um also beispielsweise im OCS die Funktion des finalen Reports zu testen, wäre ein Reset-Symbol geeignet, welches eine Konferenz erzeugt und den gesamten Ablauf bis zur zu testenden Stelle vorbereitet.

Zusätzlich zu bekannten Funktionsänderungen ist es häufig erwünscht, unmodifizierte Teile eines Systems dennoch zu prüfen. Zum Einen wäre es möglich, dass als unabhägig angesehene Änderungen einen unvorhergesehenen Effekt auf andere Funktionen haben (bekannt als *Feature interaction problem* [WNS+13]). Zum Anderen gibt es in den meisten Produkten gewisse Hauptfunktionen, die in jedem Fall funktionieren müssen und für die daher ein erneutes Testen in jeder Version notwendig ist.

Um die Arbeitsweise beim Erstellen von spezifischen Alphabeten zu verdeutlichen, wurden im Rahmen von [NWS14] zwei mögliche Fehlerquellen im OCS betrachtet, welche bereits in Abschnitt 5.6.3 erläutert wurden. Bei der ersten handelt es sich um die Forderung, dass es Autoren von Papern nicht erlaubt sein darf, das hochgeladene Dokument zu ändern, nachdem die Review-Phase begonnen hat. Um dies zu testen, ist eine Betrachtung von Konferenzen ohne eingereichtes Paper unnötig, da der Vorgang des Hochladens eines Dokuments ein Paper voraussetzt. Aus diesem Grund wurde der Reset des Alphabets so gewählt, dass er nicht nur eine Konferenz erstellt, sondern zusätzlich ein Paper einreicht und die Einreichungsphase beendet. Die Erstellung eines solchen Reset im jABC ist in diesem Fall schnell erledigt, da

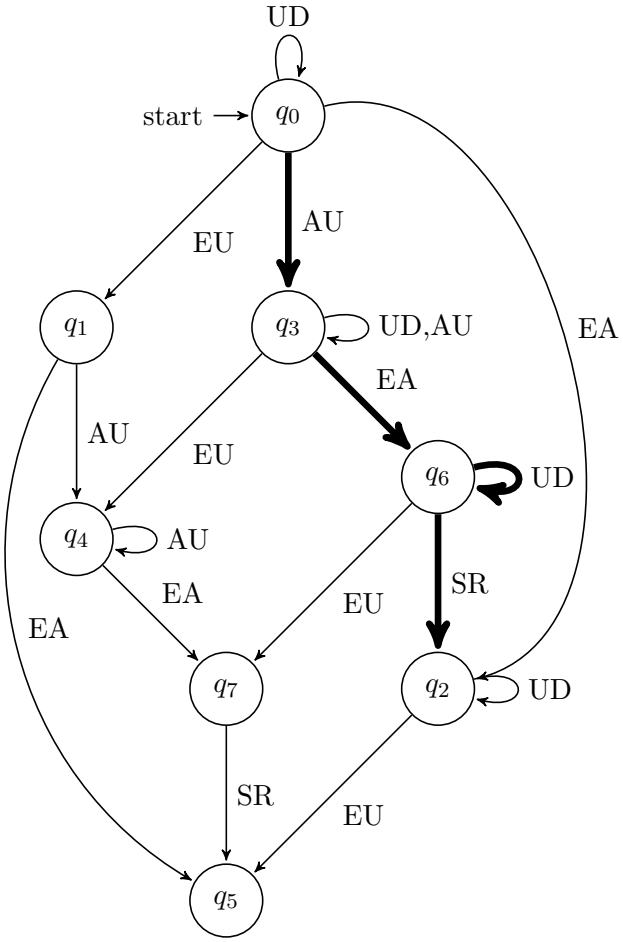

Abbildung 6.8: Modell, das mittels eines speziell zusammengestellten Alphabets in kurzer Zeit gelernt wurde und belegt, dass es möglich ist, ein Dokument für ein Paper hochzuladen, nachdem die Review-Phase begonnen hat

die entsprechenden Symbole `SP` und `ES` bereits existieren und nur dem neuen Reset-Symbol hinzugefügt werden müssen.

Zusätzlich zum angepassten Reset wurden fünf weitere Symbole ausgewählt:

UD Der Autor lädt ein Dokument für das im Reset eingereichte Paper hoch.

AU Weist dem im Reset erstellten PC Member das Paper zu. Im Gegensatz zu dem bisher verwendeten Symbol `SA` wird hier kein Bidding des Nutzers überprüft.

EU Beendet die Upload-Phase, was ein automatisches Starten der Assignment-Phase bewirkt.

EA Beendet die Assignment-Phase, was ein automatisches Starten der Review-Phase bewirkt.

SR Der PC Member reicht seinen Report für das Paper ein.

Im Gegensatz zu dem in Abschnitt 6.8 verwendeten Alphabet, welches im Durchschnitt 40 Minuten für einen vollständigen Lernvorgang benötigt, lieferte die Kombination der oben

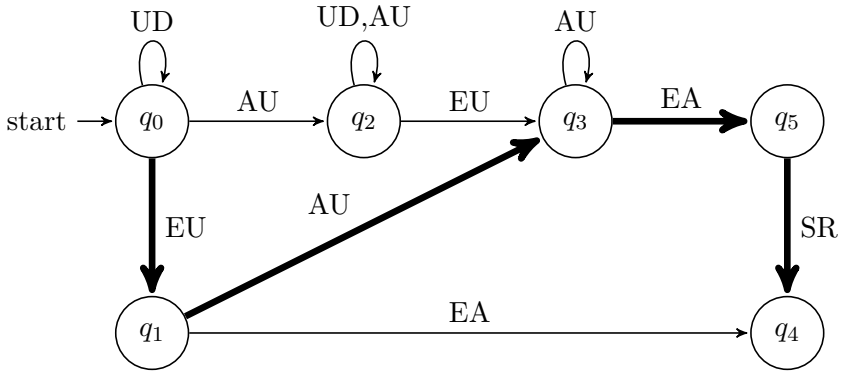

Abbildung 6.9: Der in Abbildung 6.8 gezeigte Fehler ist behoben. Allerdings ist es jetzt möglich, einen Report einzureichen, ohne dass ein Dokument existiert.

genannten Symbole zusammen mit dem angepassten Reset bereits nach etwa vier Minuten mit 381 benötigten Membership Querys das erste zutreffende Modell, welches in Abbildung 6.8 dargestellt ist.

Bei dem Zustand, der an dieser Stelle zentral von Bedeutung ist, handelt es sich um q_6. Im Gegensatz zu q_7 wurde bei ihm nicht die Upload-Phase beendet, sodass sowohl das mehrmalige Hochladen des Dokuments (`UD`) als auch das Einreichen eines Reports (`SR`) möglich ist, da bei dem Übergang $q_3 \rightarrow q_6$ die Review-Phase automatisch gestartet wurde. Noch deutlicher würde das Problem, wenn das Alphabet Reports von mehreren PC Membern oder aber das Editieren von Reports erlauben würde. Diese Anpassung würde jedoch den Lernaufwand wieder deutlich erhöhen und das gezeigte Modell deutet bereits den Fehler im System an.

Da das Modell jedoch nur eine Approximation darstellt, musste im nächsten Schritt der Fehler verifiziert werden. Dazu wird eine einzelne Membership Query direkt auf dem SUT ausgeführt. Es handelt sich dabei um den in der Abbildung markierten Pfad mit folgendem Ergebnis:

$$AU \quad EA \quad UD \quad SR \quad \rightarrow \quad success$$

Nachdem geprüft wurde, dass ein Dokument nach Ende der Assignment-Phase hochgeladen werden kann und der Fehler somit im realen System auftritt, begann die Suche nach der Ursache. Diese konnte schnell gefunden werden, da sie Teil des Standardablaufs einer Konferenz war. Um diesen nämlich möglichst kompakt zu halten, wurde ab Version 1.7.0 die Bidding-Phase nicht mehr nach der Upload-, sondern nach der Submission-Phase gestartet. Der Grund dafür ist, dass es auf diese Weise möglich ist, die Zuweisung der Reviewer vorzunehmen, während parallel die Dokumente der Paper aktualisiert werden können. Vergisst der PC Chair jedoch das Beenden der Upload-Phase, bevor er das Review startet, kommt es zu oben genanntem Fehler.

Nachdem das Problem im OCS behoben wurde, konnte ein erneuter Lernvorgang mit demselben Alphabet gestartet werden. Abbildung 6.9 zeigt den resultierenden Automaten, der den Fehler nicht mehr enthält, dafür aber einen weiteren aufweist, der bei genauerer Betrachtung bereits im ersten Modell aufgetreten ist. In diesem Fall ist ein Reviewer in der Lage, einen Report einzureichen, obwohl für das Paper gar kein Dokument hochgeladen wurde. Analog zum ersten Fehler konnte dies wieder direkt am SUL mit einer einzelnen Query bestätigt werden:

$$EU \quad AU \quad EA \quad SR \quad \rightarrow \quad success$$

Im Gegensatz zum erlaubten Hochladen während der Review-Phase wurde der OCS diesmal nicht angepasst, da dies eine Funktionsweise ist, die im Verantwortungsbereich des PC Chair liegt. Beendet dieser die Upload-Phase, ohne dass für jedes Paper ein Dokument existiert, so wird eine prominente Warnung angezeigt, welche auf die Folgen dieser Aktion hinweist. Ignoriert der PC Chair wissentlich diese Warnung, so kann das Konferenzsystem dennoch in vollem Umfang genutzt werden. Die Alternative wäre eine Sperre, die verhindert, dass die Upload-Phase bei fehlenden Papern gestoppt werden kann. Dies könnte aber auch bei nur einem einzigen Paper ohne Dokumente geschehen und so den Ablauf für hunderte andere Paper blockieren.

Eine weitere Möglichkeit wäre, das Alphabetsymbol so zu modifizieren, dass es die Anzeige der Warnung überprüft und in diesem Fall ein Stoppen der Upload-Phase verhindert. Das resultierende Modell würde dann genau den Fall repräsentieren, dass sich der verantwortliche PC Chair an den vom OCS vorgegebenen Ablauf hält und Warnungen nicht ignoriert.

Abseits von Warnungen, welche den PC Chair auf Unregelmäßigkeiten im Konferenzablauf aufmerksam machen, gibt es jedoch eine weitere Möglichkeit, die oben genannten Probleme auszulösen. Um möglichst flexibel auf die Wünsche der PC Chairs reagieren zu können, bietet der OCS die Funktion, einzelne Phasen nachträglich wieder zu aktivieren. Dies geschieht relativ häufig, um zum Beispiel ein verspätetes Paper trotz Submission-Deadline noch in die Konferenz aufzunehmen. Je nach Fortschritt der Konferenz müssen in diesem Fall gleich mehrere Phasen erneut gestartet werden, etwa die Submission-, Upload-, und Assignment-Phase.

Da der in Abbildung 6.9 gezeigte Fehler zeigte, welche Auswirkungen ein Abweichen vom ursprünglich vorgesehenen Ablauf haben kann, wurde das eigentlich als bereits behoben angesehene Problem mit einem zusätzlichen Symbol getestet: Im Gegensatz zu `EU`, welches die Upload-Phase beendet, startet das Symbol `RU` diese Phase erneut.

Aufgrund der erhöhten Anzahl der Symbole stieg die benötigte Zeit des Lernalgorithmus auf etwa sieben Minuten und die Anzahl der benötigten Membership Querys auf 641. Abbildung 6.10 zeigt das dadurch erzeugte Modell, welches erneut das eigentlich als behoben geglaubte Problem zeigt. Nach dem Beenden der Assignment-Phase ($q_3 \rightarrow q_5$) ist zwar kein direkter Upload eines Dokuments mehr möglich, der Neustart der entsprechenden Phase ($q_5 \rightarrow q_7$) ermöglicht ihn jedoch wieder ($q_7 \rightarrow q_7$).

Auch an dieser Stelle wäre eine Deaktivierung der Funktionalität im OCS, eine Phase neu zu starten, eher hinderlich. In den meisten Fällen hat der PC Chair gute Gründe für eine solche Aktion, sodass eine Einschränkung an dieser Stelle störend wäre. Wie bereits im vorigen Beispiel erläuert ist daher auch in diesem Fall eine entsprechend platzierte Warnung hilfreicher.

6.10 Fazit

Mit Hilfe der Alphabetmodellierung in einem Werkzeug zur graphischen Prozessmodellierung ist es auch Nutzern ohne Programmierkenntnisse möglich, Symbole ohne Hilfe der Systementwickler zu erstellen und zu kombinieren. Dies wurde am Beispiel des risiko-orientierten Testens gezeigt, bei dem ein Risikoanalyst nach bestimmten Kriterien Testfälle selektiert und ausführt.

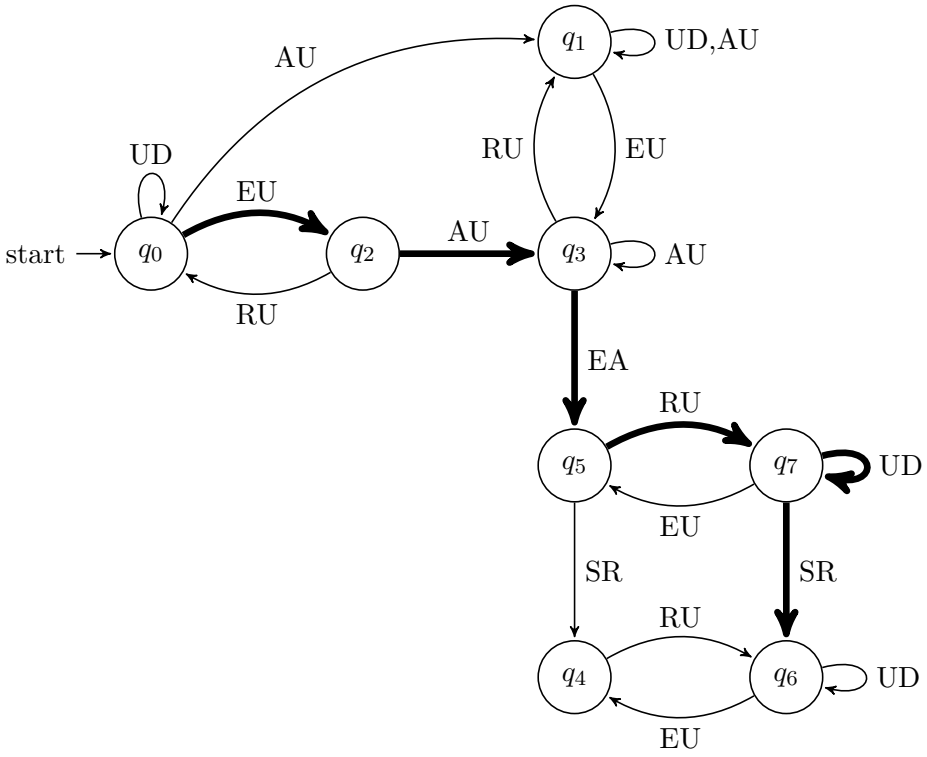

Abbildung 6.10: Nach Aufnahme des neuen Symbols `RU` zum erneuten Starten der Upload-Phase taucht der zuvor als behoben geglaubte Fehler wieder auf.

Im OCS konnte auf diese Weise in wenigen Minuten ein Fehler entdeckt werden, der es ermöglicht hätte, hochgeladene Dokumente während des Reviews zu verändern. Zwei weitere Tests zeigten in ähnlich kurzer Zeit weitere mögliche Schwachstellen, welche von den Entwicklern jedoch bewusst gewählt wurden, um dem PC Chair einer Konferenz möglichst effektives und effizientes Arbeiten zu ermöglichen.

7 Verwandte Arbeiten

Die vorliegende Dissertation leistet einen Beitrag zu den Forschungsgebieten der automatisierten Testverfahren, des aktiven Automatenlernens sowie der modellgetriebenen Softwareentwicklung. In diesem Kapitel werden diverse Arbeiten aufgeführt, welche die Entwicklung beeinflusst haben oder vergleichbare Ansätze präsentieren.

7.1 Automatische Testgenerierung

In der Softwareentwicklung sind Entwurf, Implementierung, Durchführung und Auswertung von Tests notwendig, um die Qualität eines Produktes zu gewährleisten. Dabei benötigen sie jedoch einen nicht zu unterschätzenden Anteil am gesamten Arbeitsaufwand. Gesucht sind daher Verfahren, welche diese Vorgänge automatisiert ausführen können. Der Teil mit dem größten Optimierungspotenzial stellt hierbei die Suche nach und Umsetzung von geeigneten Testfällen dar, weshalb es an dieser Stelle verschiedene Ansätze gibt, den Vorgang zu optimieren.

Liegt der Quellcode der zu testenden Software vor, was in vielen Fällen zutreffend sein dürfte, so lassen sich mit White-Box-Tests gezielt einzelne Methoden auf Korrektheit prüfen. Ein Ansatz, dies zu automatisieren, wird bereits 1998 von Korel und Al-Yami in [KAY98] beschrieben und verwendet den Quellcode einer in der Programmiersprache Pascal geschriebenen Anwendung, um aus den Funktionsaufrufen Testfälle ableiten zu können. Einen ähnlichen Ansatz verfolgen Taneja und Xie in [TX08], wobei die Programme, welche in diesem Fall in Java geschrieben sind, zuvor durch Instrumentierung des Quellcodes angepasst werden. Auf diese Weise stellen die Autoren sicher, dass die Ausführung von Code, der sich zwischen zwei Versionen geändert hat, durch generierte Testfälle garantiert werden kann.

Während solche Quellcode-basierten Ansätze für kleine Programme oder Bibliotheken gute Ergebnisse liefern können, stoßen sie bei komplexeren Anwendungen schnell an ihre Grenzen. Gerade mehrschichtige Webanwendungen bestehen aus einer Vielzahl von Komponenten, deren Zusammenspiel auf diese Weise nur schwer untersucht werden kann. Soll das System als Einheit getestet werden, bieten sich daher Zugriffe auf die bereitgestellten Schnittstellen an. Bei Webanwendungen sind Web Services eine verbreitete Möglichkeit für solch einen Zugriff, wie es in Kapitel 4 am Beispiel von Mantis gezeigt wurde. In [XDTC05] verwenden Bai et al. daher die von Web Services bereitgestellten Schnittstelleninformationen, welche in Form einer WSDL-Spezifikation vorliegen, um automatisch Testfälle zu generieren.

Die Überprüfung einzelner Methoden, ob nun auf Ebene des Quellcodes oder durch Aufruf von Web Services, ist sehr feingranular und eignet sich daher nicht dazu, Systeme auf Ebene von abstrakten Anwendungsfällen zu vergleichen. Auch die für ein System zuvor definierten Spezifikationen werden nicht berücksichtigt, obwohl diese bereits das erwünschte Systemverhalten repräsentieren. Tahat et al. nutzen daher in [TVKB01] solche Spezifikationen, die textuell oder im SDL-Format vorliegen können, um daraus Systemmodelle und im zweiten Schritt Testfälle generieren zu können. Einen ähnlichen Ansatz, der zudem abstrakte Anwendungsfälle fokussiert, verfolgen Nebut et al. in [NFTJ06]. Hier werden UML-Diagramme

zusammen mit einer eigens definierten Anforderungssprache kombiniert, um mit Hilfe von Sequenzdiagrammen Anwendungsfälle ableiten und in Testfälle umwandeln zu können.

Bei der in dieser Arbeit vorgestellten kontinuierlichen Qualitätskontrolle ist eine Herangehensweise aus beiden Richtungen möglich. Zuvor festgelegte Spezifikationen sind nicht notwendig, da das System als Black Box behandelt und automatisch gelernt wird. Dennoch lassen sich die daraus resultierenden Modelle bei Bedarf mit Hilfe von Model-Checking-Formeln daraufhin verifizieren, ob das System das gewünschte Verhalten aufweist. Mit der Fokussierung auf abstrakte Anwendungsfälle bleiben die Modelle zudem über mehrere Versionen einfach vergleichbar. Dabei ist die Lösung flexibel genug, um Systeme mit Zugriff auf diverse Programmierschnittstellen anzusprechen und so auch Vergleiche etwa von Geschäftslogik- und Präsentationsschicht zu ermöglichen.

7.2 Testen durch aktives Automatenlernen

Die Inferenz von Modellen existierender Systeme wurde bereits zur Jahrtausendwende am Lehrstuhl für Programmiersysteme durchgeführt. Zu Beginn von Hagerer et al. als *reguläre Extrapolation* [HHNS02] bezeichnet, findet hier noch keine Verwendung des Algorithmus von Angluin statt. In der Dissertation von Oliver Niese mit dem Titel „An Integrated Approach to Testing Complex Systems“ [Nie03] kommt dann eine Testumgebung mit der Bezeichnung *Integrated Test Environment* (ITE) zum Einsatz, die das globale Verhalten eines Systems anhand vordefinierter Tests prüfen kann. Analog zur Vorgehensweise bei der kontinuierlichen Qualitätskontrolle sind hierbei zuerst sogenannte *Testblöcke* von einem Entwickler bereitzustellen, die anschließend von einem Applikationsexperten zu Testfällen kombiniert werden können. Letzteres geschieht mit Hilfe des Vorgängers des jABC, dem seit 1993 entwickelten *Agent Building Center* (ABC).

Auch das Testen von Webapplikationen ist mit der ITE möglich. Hierzu wurde eine Reihe von generischen Testblöcken entworfen, die etwa einen Link anklicken oder Textfelder ausfüllen können. Mit deren Hilfe wurden anschließend zwei Web-Anwendungen getestet, die denen recht ähnlich sind, welche in der vorliegenden Dissertation verwendet wurden: der Vorgänger des Online Conference Service sowie der Bugtracker *Bug Tracking System*.

Eine wesentliche Neuerung in der Arbeit von Niese ist zudem die Anpassung des Algorithmus von Angluin auf Mealy-Maschinen. Dies ermöglicht es, auch reaktive Systeme zu untersuchen und aus den Ergebnissen a posteriori entsprechende Automatenmodelle zu erstellen. Die Kluft zwischen Lernalgorithmus und zu lernendem System überbrücken dabei die existierenden Testblöcke des ITE.

Im Jahr 2009 wurde die ITE von Raffelt et al. in [RMSM09] zum dynamischen Testen von reaktiven Systemen verwendet, wobei auch der für Mealy-Maschinen angepasste Algorithmus von Angluin zum Einsatz kam. Nicht nur die Erstellung von Testfällen fand diesmal bereits im jABC statt, auch der Ablauf eines Lernvorgangs konnte auf diesem Weg spezifiziert werden. Mit diesem Aufbau wurden Modelle sowohl des Bugtrackers Mantis als auch eines Hardware-Routers erzeugt.

Die Idee des Einsatzes eines Lernalgorithmus zur Generierung von Testfällen für reaktive Systeme wurde weiterhin in der Dissertation von Muddassar Azam Sindhu mit dem Titel „Algorithms and Tools for Learning-based Testing of Reactive Systems“ [Sin13] weiter ausgeführt. Sindhu stellt dabei verschiedene Lernalgorithmen und passende Werkzeuge vor, etwa das von ihm selbst entwickelte LBTest [MS13], welches unter anderem einen aktiven Lernalgorithmus mit einem Model-Checker kombiniert.

Der Ansatz der kontinuierlichen Qualitätskontrolle greift dieses Vorgehen auf und führt es konsequent weiter. Den oben genannten Arbeiten ist gemein, dass ein Vergleich der gelernten Modelle nicht vorgesehen war, was unter anderem daran liegt, dass die verwendeten Testblöcke zu generisch waren. Ist beispielsweise eine Weboberfläche in einer Version so gestaltet, dass gegenüber dem Vorgänger eine Bestätigung zur Sicherheit notwendig ist, würde dies eine Veränderung im Modell erzeugen, obwohl sich das Verhalten bzw. die Funktionalität des Systems nicht geändert hat. Eine Betrachtung des Systemverhaltens über den gesamten Entwicklungszyklus hinweg ist auf diese Weise nur schwer durchführbar. Durch die jetzt eingeführte konsequente Abstraktion der Anwendungsfälle ist nicht nur ein Vergleich verschiedener Versionen möglich, auch Unterschiede im Verhalten von Präsentations- und Geschäftslogik bei mehrschichtigen Systemen lassen sich so aufdecken. Als positiver Nebeneffekt sinkt die Anzahl der Zustände eines Modells, was eine manuelle Auswertung vereinfacht.

Gleichzeitig wurden mit der kontinuierlichen Qualitätskontrolle die verwendeten Konzepte und Komponenten auf einen aktuellen Stand gebracht. Während die ITE an vielen Stellen das kommerzielle Automatisierungswerkzeug *Rational Robot*[1] verwendet, welches ausschließlich für Systeme mit Microsoft Windows konzipiert wurde, basiert die in der vorliegenden Arbeit vorgestellte Lösung primär auf aktuellen, plattformübergreifenden Softwareprodukten unter einer Open-Source-Lizenz, wie dem in Java implementierten Lernalgorithmus oder dem Selenium-Framework. Verbesserungen wie der vor einigen Jahren entwickelte Reuse-Filter helfen zudem dabei, den Lernvorgang weiter zu beschleunigen. Webanwendungen können nicht nur mit Hilfe von gut getesteten Bibliotheken wie dem Selenium-Framework, welche einen Browser emulieren, getestet werden, auch der Zugriff auf alternative Schnittstellen wie Webservices oder RMI (Remote Method Invocation) ist möglich. Die Modellierung im aktuellen jABC erlaubt zudem eine automatisierte Codegenerierung der Testfälle, wobei als zugrundeliegende Programmiersprache das weitverbreitete Java eingesetzt wird.

7.3 Einbindung von Anwendungsexperten

Die in Kapitel 6 beschriebene Art zur Erstellung von Alphabetsymbolen durch Anwendungsexperten setzt das von Margaria und Steffen in [MS09] und [MS12] beschriebene Paradigma des *Extreme Model Driven Design (XMDD)* um. Hierbei werden Personen, welche die Funktionsweise einer Software im Detail kennen, jedoch über weniger oder gar keine technischen Kenntnisse verfügen, in den Entwicklungsprozess eingebunden. Dies ist durch eine Fokussierung auf eine modellgetriebene Softwareentwicklung möglich, bei welcher die Arbeitsabläufe in einer graphischen Umgebung anhand von Prozessdiagrammen beschrieben und im Anschluss mit Hilfe eines Codegenerators in eine lauffähige Anwendung überführt werden.

In welcher Art die Einbindung von Anwendungsexperten möglich ist, zeigten bereits Lamprecht et al. in [LMS08] mit ihrer Plattform Bio-jETI. Mit dieser ist es möglich, dass Biologen komplexe Arbeits- und Analyseabläufe nicht nur spezifizieren sondern auch direkt ausführen können.

Eine mögliche Gefahr bei der modellgetriebenen Entwicklung besteht in einer zu starken Spezifizierung. Sind die in den Modellen verwendeten Komponenten zu unflexibel und wenig vielseitig, kann es schnell geschehen, dass der eigentlich gewünschte Vorteil durch diese Art der Entwicklung sich ins Gegenteil verkehrt. Anwendungsexperten stehen dann vor dem Problem, dass ihnen nicht klar ist, auf welche Weise die ihnen bereitgestellten Bausteine zu

[1] `http://www-03.ibm.com/software/products/de/robot`

kombinieren sind, um das gewünschte Verhalten zu modellieren. Um dem entgegenzutreten, führten Lamprecht et al. in [LNMS10] das sogenannte *Loose Programming*, welches auf der Synthese von Software basiert. Hiermit können Komponenten mit einer „Unterspezifizierung“ kombiniert werden, damit im Anschluss ein Synthesealgorithmus mit Hilfe einer Datenflussanalyse die notwendigen Dienste zu einer lauffähigen Anwendung verbindet. In Kombination mit dem sogenannten *variability modeling* lassen sich auf diese Weise auch Produktlinien agil auf der Modellierungsebene umsetzen [LMSS12].

In der kontinuierlichen Qualitätskontrolle wurde zu diesem Zweck ein anderer Ansatz gewählt, bei dem Entwickler der Anwendung die notwendigen Test-Blöcke bereitstellen, welche von den Anwendungsexperten dann zu Modellen kombiniert werden können. Bei der Verbindung der einzelnen Komponenten sind jedoch die ausgetauschten Daten eingeschränkt, etwa auf eingereichte Paper oder auszuführende Nutzer, um die Übersichtlichkeit zu gewährleisten. Weiterhin muss der Anwendungsexperte bei der Modellierung nicht die Art des Zugriffs oder die zu testende OCS-Version beachten. Dies ist möglich, da die technischen Voraussetzungen durch eine stabile Abstraktion für alle Versionen des OCS existiert. Eine Datenflussanalyse ähnlich dem des Synthesealgorithmus findet auf Basis der definierten Alphabete nicht statt, da die Symbole vom Lernalgorithmus entsprechend zu Anfrageworten zusammengestellt werden.

Für die Spezifizierung von Prozessmodellen existieren noch weitere Standards, etwa die *Business Process Model and Notation (BPMN)* [WM08] zur graphischen Modellierung von Geschäftsprozessen. Die aktuelle Version BPMN 2.0 [All09] wurde 2011 verabschiedet und spezifiziert ein auf XML basierendes Dateiformat, welches den Datenaustausch zwischen verschiedenen Werkzeugen erlaubt. Diese Standardisierung zusammen mit der Möglichkeit, sowohl fachliche als auch technische Abläufe mitsamt einer Ausführungssemantik zu beschreiben, sorgten für eine weite Verbreitung der Spezifikation. Jedoch fehlt es BPMN an Flexibilität, wenn es darum geht, einzelne mit den Modellen verknüpften Diensten auszutauschen [NS13]. Dies und die Möglichkeit, nativen Java-Code [Neu14] benutzen zu können, führten daher unter anderem zum Einsatz des Java Application Building Center (siehe Abschnitt 2.5.3).

8 Zusammenfassung und Ausblick

In der vorliegenden Diplomarbeit wurden Konzepte zur kontinuierlichen Qualitätskontrolle von Webanwendungen vorgestellt und anhand von zwei Beispielsystemen demonstriert. Im Folgenden werden die gesammelten Erkenntnisse zusammengefasst und ein Ausblick auf zukünftige Erweiterungen gegeben.

8.1 Qualitätskontrolle durch Modellvergleiche

Damit verschiedene Versionen einer Software daraufhin geprüft werden können, ob sich das vom Benutzer beobachtbare Verhalten ändert, ist zuerst eine Abstraktion der möglichen Aktionen notwendig gewesen. Für die beiden getesteten Systeme, den Bugtracker Mantis und den OCS, wurden dazu Alphabetsymbole erstellt, die abstrakte Anwendungsfälle im jeweiligen System repräsentieren. Mit Hilfe dieser Symbole konnten unter Verwendung eines Lernalgorithmus Modelle der Systeme erzeugt werden, die auf unterschiedliche Weise auf Fehler oder andere Veränderungen geprüft werden können. Ein auf diese Weise gefundenes Problem im OCS hätte es ermöglicht, Aktionen durchzuführen, für welche die Nutzer die eigentlich notwendigen Rechte nicht besitzen. Ist ein solcher Fehler erst einmal bekannt, kann er durch Model Checking mit entsprechenden temporallogischen Formeln ohne Vergleich mit früheren Versionen auch in zukünftigen Modellen schnell entdeckt werden.

Wenn die gewählten Alphabetsymbole abstrakt genug sind, wäre an dieser Stelle eine Erweiterung auf thematisch ähnliche Produkte interessant. So könnten die für Mantis erstellten Alphabetsymbole – die entsprechenden Testtreiber vorausgesetzt – auch mit anderen Bugtracker-Produkten in entsprechende Modelle überführt werden. Diese könnten verwendet werden, um unterschiedliche Arbeitsabläufe aufzuzeigen, damit etwa Abweichungen in den Rechten einzelner Nutzer verdeutlicht oder die verschiedenen Lebenszyklen eines Fehlerberichts nachverfolgt werden können.

Die zum Ansprechen der Systeme notwendigen Testtreiber stellen derzeit noch eine Schwierigkeit beim Einsatz der genannten Technik dar. Nach einmaligem, initialem Aufwand ist der Großteil der notwendigen Arbeit zwar meist schon geleistet, beispielsweise konnten die beiden für Mantis entwickelten Adapter für alle Versionen der letzten vier Jahre verwendet werden. Häufige Änderungen in den zugrundeliegenden Schnittstellen, wie sie im OCS öfter geschehen, erhöhen jedoch die Gefahr, dass ein bisher funktionierender Treiber inkompatibel wird. Um dem zu begegnen, könnte bei Produkten mit entsprechendem Zugriff bereits während der Entwicklung auf eine stabile API für Testzwecke geachtet werden. Wo das nicht möglich ist, können automatisierte Tests prüfen, ob sich Änderungen auf die Kompatibilität des Treibers auswirken. Einen anderen Ansatz stellt die selbstständige Erzeugung von Testtreibern dar, wie sie etwa in [MIH+12] beschrieben wird.

Als Lernalgorithmus kam in dieser Dissertation ausschließlich eine modifizierte Version des Algorithmus L^* von Dana Angluin zur Anwendung. Dieser stellt jedoch nicht die einzige Möglichkeit zum Lernen von Mealy-Automaten dar. Der in [How12] beschriebene Observation-Pack-Algorithmus verfügt beispielsweise über eine effizientere Datenstruktur, welche zu einer Minimierung der notwendigen Membership Querys beiträgt. Zur weiteren Verbesserung der

Modellerstellung wäre es daher hilfreich, die Laufzeit durch einen eventuellen Austausch dieser Komponente weiter zu verbessern.

8.2 Vergleich verschiedener Zugriffsarten

Unabhängig von der Schnittstelle, mit der eine Anwendung aufgerufen wird, sollte das beobachtete Verhalten identisch sein. Dass diese Annahme nicht immer zutrifft, konnte im Rahmen dieser Dissertation an beiden untersuchten Softwareprodukten gezeigt werden, indem zusätzliche Testtreiber entwickelt wurden, welche auf das Test-Framework Selenium zurückgreifen. Dieser benötigte zwar in beiden Fällen eine deutlich längere Zeit zum Erstellen der notwendigen Modelle, konnte jedoch Unterschiede im Systemverhalten aufdecken. So erlaubte Mantis gewisse Aktionen nicht für Entwickler, wenn diese nicht über die Weboberfläche, sondern mit Hilfe des SOAP-Protokolls Änderungen vornehmen wollten. Dies stellt einen Fehler dar, der an das Entwicklungsteam des Bugtrackers gemeldet wurde.

Für den OCS existierten bereits Adapter, welche per Selenium auf die Weboberfläche zugreifen, da diese auch für die automatisierten Systemtests verwendet werden. Mit deren Hilfe war es möglich, zu klären, weshalb manuelle Tests vor der Veröffentlichung das oben genannte Problem nicht aufgedeckt haben. An dieser Stelle lag der Grund in zusätzlichen Sicherungsmaßnahmen der Weboberfläche, diese konnten aber nach Bekanntwerden des Unterschieds auch für die Geschäftslogik umgesetzt werden.

Selenium wurde im Rahmen dieser Dissertation ausschließlich mit der HTMLUnit-Komponente betrieben, welche einen Browser nur simuliert und daher Anfragen weitaus schneller verarbeiten kann als ein realer Browser. Für weitere Untersuchungen wären jedoch auch Lernvorgänge hilfreich, die aktuelle Produkte wie Chrome oder Firefox verwenden. Auf diese Weise ließen sich Unterschiede im Systemverhalten feststellen, die allein von der Wahl des eingesetzten Browsers abhängen. Eine Modifikation der Testtreiber sollte an dieser Stelle nicht notwendig sein, allerdings würde der notwendige Zeitaufwand aufgrund der durchgeführten Darstellung von Webseiten noch weiter steigen. Interessant ist an dieser Stelle auch der Einsatz von mobilen Browsern, um beispielsweise die Funktionsgleichheit der mobilen Variante einer Webanwendung zu verifizieren.

8.3 Automatisierte Suche nach Fehlerursachen

Zur Analyse und Behebung eines Fehlers kann es hilfreich sein, den Zeitpunkt zu kennen, an dem das Problem das erste Mal im Code aufgetreten ist. Auch für einen Fehler, der bereits als behoben galt und nun erneut auftritt, kann eine solche Information die Behebung unterstützen.

Am Beispiel von Mantis konnte mit den Mitteln der kontinuierlichen Qualitätskontrolle die gesamte Geschichte eines schwerwiegenden Fehlers von der Entstehung über die Lösung bis hin zur Veröffentlichung verfolgt werden, was in diesem Fall einen Zeitraum von über vier Monaten umfasst. Eine ähnliche Suche für einen Fehler im OCS lieferte nach einer Suchdauer von zwei Stunden die Änderung im Code, welche für einen zuvor entdeckten Fehler verantwortlich war. Selbst wenn ein menschlicher Entwickler dies schneller erreichen könnte, so steht er dennoch während dieser Zeit nicht für andere Aufgaben zur Verfügung. Aus diesem Grund kann sich die automatische Suche schon allein aus kostentechnischen Gründen lohnen.

Optimieren ließe sich dieses Verfahren, indem man zusätzliche Informationen in den Suchvorgang einfließen lässt. Sind bestimmte Revisionen der Anwendung garantiert fehlerfrei oder weisen aber den Fehler auf jeden Fall auf, so kann ein eingegrenzter Suchbereich die benötigte Dauer stark reduzieren. Je nach Art der verfügbaren Informationen bieten sich auch andere Suchstrategien, wie etwa eine Binärsuche, an.

8.4 Verifikation von Systemmigrationen

Einer der mit der kontinuierlichen Qualitätskontrolle in Mantis gefundenen Fehler trat exakt dann auf, wenn eine neue Mantis-Instanz angelegt wurde, während migrierte Installationen problemlos weiter funktionierten. Das Gegenteil hiervon, also Fehler nach der scheinbar erfolgreichen Migration einer Software, ist wesentlich schwerer zu entdecken, da die Konsequenzen erst nach einiger Zeit sichtbar werden können. Um diese Art von Fehler ebenfalls abzudecken, wurde eine Methode zur Verifikation von Systemmigrationen vorgestellt, die erfolgreich in den beiden genutzten Softwareprodukten eingesetzt werden konnte.

Bei dem gezeigten Ansatz werden jedoch ausschließlich zwei isolierte, aufeinanderfolgende Versionen betrachtet. Eine Erweiterung könnte den Migrationspfad zwischen den Versionen verfolgen, indem mehrere Versionssprünge nacheinander verglichen werden. Dies würde beispielsweise Fehler aufdecken, die erst im Zusammenspiel mit zwei Migratoren auftauchen.

8.5 Graphische Alphabetmodellierung

Eine Schwierigkeit bei Einrichtung und Einsatz der kontinuierlichen Qualitätskontrolle ist das notwendige technische Verständnis, welches für die Erstellung der Testtreiber und Alphabetsymbole benötigt wird. Um dies teilweise zu entkoppeln, wurde die Alphabetmodellierung mit Hilfe eines Werkzeugs zur graphischen Prozessmodellierung gezeigt. An dieser Stelle können Personen, die zwar umfangreiches Wissen über die Applikation, nicht jedoch über deren technischen Aufbau besitzen, Testfälle graphisch modellieren und ausführen. Durch die Verwendung hierarchischer Graphen lassen sich auf diese Weise wiederverwendbare Komponenten erzeugen, welche auch in zukünftigen Testfällen eingesetzt werden können.

Demonstriert wurde dieses Vorgehen am Beispiel eines Risikoanalysten, der Testfälle aufgrund von gegebenen Beschränkungen priorisieren muss. Durch die Beschränkung auf wenige Alphabetsymbole ist er so in der Lage, in kurzer Zeit übersichtliche Modelle zu erzeugen, welche Anwendungsfälle mit einem hohen Fehlerrisiko abdecken. Zwei solcher Risiken konnten daraufhin im OCS entdeckt werden.

Dieser Ansatz konzentriert sich derzeit auf die Erstellung von Systemmodellen, ließe sich in Zukunft aber auch auf weitere in dieser Dissertation genannten Aspekte ausweiten. So wäre es etwa möglich, Datenbankmigratoren ebenfalls aus Prozessmodellen heraus anzusteuern, um auf diese Weise gezielt Aktualisierungen einzelner Versionen zu testen, gegebenfalls mit einer entsprechenden Auswahl von Alphabetsymbolen.

Literaturverzeichnis

[All09] Allweyer, Thomas: *BPMN 2.0 - Business Process Model and Notation.* Books on Demand, 2009. – ISBN 978–3839121344

[And98] Andrews, James H.: Testing using Log File Analysis: Tools, Methods, and Issues. In: *13th IEEE International Conference on Automated Software Engineering*, 1998, S. 157–166

[Ang87] Angluin, Dana: Learning Regular Sets from Queries and Counterexamples. In: *Information and Computation* 75 (1987), Nr. 2, S. 87–106

[AS09] Alwis, Brian de ; Sillito, Jonathan: Why are software projects moving from centralized to decentralized version control systems? In: *Cooperative and Human Aspects on Software Engineering, 2009. CHASE '09. ICSE Workshop on*, 2009, S. 36–39

[BCMS12] Bertolino, Antonia ; Calabrò, Antonello ; Merten, Maik ; Steffen, Bernhard: Never-stop Learning: Continuous Validation of Learned Models for Evolving Systems through Monitoring. In: *ERCIM News* 2012 (2012), Nr. 88, 28–29.

[BDG+08] Baker, Paul ; Dai, Zhen R. ; Grabowski, Jens ; Schieferdecker, Ina ; Williams, Clay: *Model-Driven Testing: Using the UML Testing Profile.* Springer, 2008. ISBN 978–3–540–72562–6

[Bec03] Beck, Kent: *Test-Driven Development: By Example.* Addison-Wesley Professional, 2003. – ISBN 978–0321146533

[Bei95] Beizer, Boris: *Black-Box Testing: Techniques for Functional Testing of Software and Systems.* New York, NY, USA : Wiley, 1995. – ISBN 0–471–12094–4

[BMRS09] Bakera, Marco ; Margaria, Tiziana ; Renner, Clemens ; Steffen, Bernhard: Tool-supported enhancement of diagnosis in model-driven verification. In: *Innovations in Systems and Software Engineering* 5 (2009), 211-228. ISSN 1614–5046

[BNSH12] Bauer, Oliver ; Neubauer, Johannes ; Steffen, Bernhard ; Howar, Falk: Reusing System States by Active Learning Algorithms. Version: 2012. In: Moschitti, Alessandro (Hrsg.) ; Scandariato, Riccardo (Hrsg.): *Eternal Systems* Bd. 255. Springer-Verlag, 2012.

[CDLW04] Cook, Jonathan E. ; Du, Zhidian ; Liu, Chongbing ; Wolf, Alexander L.: Discovering Models of Behavior for Concurrent Workflows. In: *Computers in Industry* 53 (2004), Apr, Nr. 3, S. 297–319.

[CGP99] Clarke, Edmund M. ; Grumberg, Orna ; Peled, Doron A.: *Model Checking.* The MIT Press, Cambridge, MA, USA, 1999

[Cha09] Chacon, Scott ; Parkes, Duncan (Hrsg.): *Pro Git*. 1. Apress, 2009 `http://git-scm.com/book`. – ISBN 978–1430218333

[Cla77] Clarke, Edmund M. Jr.: Programming Language Constructs for Which It is Impossible to Obtain Good Hoare-like Axiom Systems. In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. New York, NY, USA : ACM, 1977 (POPL '77), S. 10–20

[CW98] Cook, Jonathan E. ; Wolf, Alexander L.: Discovering Models of Software Processes from Event-based Data. In: *ACM Trans. Softw. Eng. Methodol.* 7 (1998), Jul, Nr. 3, S. 215–249.

[Eck95] Eckerson, Wayne W.: Three Tier Client/Server Architecture: Achieving Scalability, Performance, and Efficiency in Client Server Applications. In: *Open Information Systems* 10 (1995)

[FR14] Felderer, Michael ; Ramler, Rudolf: A multiple case study on risk-based testing in industry. In: *International Journal on Software Tools for Technology Transfer* (2014), S. 1–17.

[Hal98] Halbwachs, Nicolas: Synchronous Programming of Reactive Systems. Version: 1998. In: Hu, AlanJ. (Hrsg.) ; Vardi, MosheY. (Hrsg.): *Computer Aided Verification* Bd. 1427. Springer Berlin / Heidelberg, 1998. – ISBN 978–3–540–64608–2, S. 1–16

[HBM+12] Howar, Falk ; Bauer, Oliver ; Merten, Maik ; Steffen, Bernhard ; Margaria, Tiziana: The Teachers' Crowd: The Impact of Distributed Oracles on Active Automata Learning. Version: 2012. In: Hähnle, Reiner (Hrsg.) ; Knoop, Jens (Hrsg.) ; Margaria, Tiziana (Hrsg.) ; Schreiner, Dietmar (Hrsg.) ; Steffen, Bernhard (Hrsg.): *Leveraging Applications of Formal Methods, Verification, and Validation*. Springer Berlin Heidelberg, 2012 (Communications in Computer and Information Science).

[HHNS02] Hagerer, Andreas ; Hungar, Hardi ; Niese, Oliver ; Steffen, Bernhard: Model Generation by Moderated Regular Extrapolation. Version: 2002. In: Kutsche, Ralf-Detlef (Hrsg.) ; Weber, Herbert (Hrsg.): *Fundamental Approaches to Software Engineering* Bd. 2306. Springer Berlin / Heidelberg, 2002. – ISBN 978–3–540–43353–8, S. 80–95

[HMS03] Hungar, H. ; Margaria, T. ; Steffen, B.: Test-based model generation for legacy systems. In: *Test Conference, 2003. Proceedings. ITC 2003. International* Bd. 1, 2003. – ISSN 1089–3539, S. 971–980

[HNS03] Hungar, Hardi ; Niese, Oliver ; Steffen, Bernhard: Domain-Specific Optimization in Automata Learning. In: Jr., Warren A. H. (Hrsg.) ; Somenzi, Fabio (Hrsg.): *Proc. 15^{th} Int. Conf. on Computer Aided Verification* Bd. 2725, Springer Verlag, July 2003 (Lecture Notes in Computer Science), S. 315–327

[Hoa69] Hoare, C. A. R.: An axiomatic basis for computer programming. In: *Communications of the ACM* 12 (1969), Nr. 10, S. 576–580. ISSN 0001–0782

[Hop71] Hopcroft, John E.: An N Log N Algorithm for Minimizing States in a Finite Automaton / Stanford University. Stanford, CA, USA : Stanford University, January 1971. – Forschungsbericht

[How12] Howar, Falk: *Active Learning of Interface Programs*, TU Dortmund University, Diss., 2012. `https://eldorado.tu-dortmund.de/bitstream/2003/29486/1/Dissertation.pdf`

[HP85] Harel, D. ; Pnueli, A.: On the Development of Reactive Systems. Version: 1985. In: Apt, Krzysztof R. (Hrsg.): *Logics and Models of Concurrent Systems* Bd. 13. Springer Berlin / Heidelberg, 1985. – ISBN 0–387–15181–8, Kapitel On the Development of Reactive Systems, S. 477–498

[HP04] Hovemeyer, David ; Pugh, William: Finding Bugs is Easy. In: *ACM SIGPLAN Notices* 39 (2004), Dec, Nr. 12, S. 92–106.

[HS04] Hungar, Hardi ; Steffen, Bernhard: Behavior-based model construction. In: *Int. J. Softw. Tools Technol. Transf.* 6 (2004), Nr. 1, S. 4–14

[HSJC12] Howar, Falk ; Steffen, Bernhard ; Jonsson, Bengt ; Cassel, Sofia: Inferring Canonical Register Automata. Version: 2012. In: Koncak, Viktor (Hrsg.) ; Rybalchenko, Andrey (Hrsg.): *Verification, Model Checking, and Abstract Interpretation* Bd. 7148. Springer Berlin / Heidelberg, 2012.

[HSM11] Howar, Falk ; Steffen, Bernhard ; Merten, Maik: Automata Learning with Automated Alphabet Abstraction Refinement. Version: 2011. In: Jhala, Ranjit (Hrsg.) ; Schmidt, David (Hrsg.): *Verification, Model Checking, and Abstract Interpretation* Bd. 6538. Springer Berlin / Heidelberg, 2011.

[IHS13a] Isberner, Malte ; Howar, Falk ; Steffen, Bernhard: Inferring Automata with State-Local Alphabet Abstractions. In: Brat, Guillaume (Hrsg.) ; Rungta, Neha (Hrsg.) ; Venet, Arnaud (Hrsg.): *NASA Formal Methods* Bd. 7871, 2013 (LNCS), S. 124–138

[IHS13b] Isberner, Malte ; Howar, Falk ; Steffen, Bernhard: Learning register automata: from languages to program structures. In: *Machine Learning* (2013), S. 1–34.

[Jö13] Jörges, Sven: *Lecture Notes in Computer Science*. Bd. 7747: *Construction and Evolution of Code Generators - A Model-Driven and Service-Oriented Approach*. Springer Berlin Heidelberg, Germany, 2013.

[KAY98] Korel, Bogdan ; Al-Yami, Ali M.: Automated Regression Test Generation. In: *Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysi*. New York, NY, USA : ACM, 1998 (ISSTA '98), S. 143–152

[KGTB07] Kicillof, Nicolas ; Grieskamp, Wolfgang ; Tillmann, Nikolai ; Braberman, Victor: Achieving Both Model and Code Coverage with Automated Gray-Box Testing. In: *A-MOST*, 2007, S. 1–11

[KM06] Karusseit, Martin ; Margaria, Tiziana: Feature-based Modelling of a Complex, Online-Reconfigurable Decision Support Service. In: *Electronic Notes in Theoretical Computer Science* 157 (2006), Nr. 2, 101 - 118. ISSN 1571–0661

[Lin05] Link, Johannes: *Softwaretests mit JUnit : Techniken der testgetriebenen Entwicklung*. dpunkt-Verlag, 2005. – 97 – 135 S. – ISBN 3–89864–325–5

[LMS06] Lamprecht, Anna-Lena ; Margaria, Tiziana ; Steffen, Bernhard: Data-Flow Analysis as Model Checking Within the jABC. In: Mycroft, Alan (Hrsg.) ; Zeller, Andreas (Hrsg.): *Compiler Construction* Bd. 3923, Springer Berlin Heidelberg, 2006 (Lecture Notes in Computer Science). – ISBN 978–3–540–33050–9, 101-104

[LMS08] Lamprecht, Anna-Lena ; Margaria, Tiziana ; Steffen, Bernhard: Seven Variations of an Alignment Workflow - An Illustration of Agile Process Design and Management in Bio-jETI. In: *Bioinformatics Research and Applications* Bd. 4983. Atlanta, Georgia : Springer, 2008 (Lecture Notes in Bioinformatics), 445–456

[LMSS12] Lamprecht, Anna-Lena ; Margaria, Tiziana ; Schaefer, Ina ; Steffen, Bernhard: Synthesis-Based Variability Control: Correctness by Construction. In: *10th International Symposium, FMCO 2011, Turin, Italy, October 3-5, 2011, Revised Selected Papers* Bd. 7542, Springer Berlin Heidelberg, 2012 (Lecture Notes in Computer Science), S. 69–88

[LNMS10] Lamprecht, Anna-Lena ; Naujokat, Stefan ; Margaria, Tiziana ; Steffen, Bernhard: Synthesis-Based Loose Programming. In: *Proc. of the 7th Int. Conf. on the Quality of Information and Communications Technology (QUATIC 2010), Porto, Portugal*, 2010, S. 262–267

[Man10] Mantis Bugtracker: *Fehlerbericht: Normalise access checks between the web interface and the SOAP API*. `http://www.mantisbt.org/bugs/view.php?id=12328`. Version: September 2010

[Man12] Mantis Bugtracker: *Fehlerbericht: Impossible to create a new project with fresh install on PostgreSQL*. `http://www.mantisbt.org/bugs/view.php?id=14385`. Version: Juni 2012

[McC76] McCabe, Thomas J.: A Complexity Measure. In: *IEEE Transactions on Software Engineering* SE-2 (1976), Dec, Nr. 4, S. 308–320.

[Mea55] Mealy, George H.: A Method for Synthesizing Sequential Circuits. In: *Bell System Technical Journal* 34 (1955), Nr. 5, S. 1045–1079

[Mem08] Memon, Ahmed U.: *Log File Categorization and Anomaly Analysis Using Grammar Inference*, Queen's University, Diss., 2008. `http://hdl.handle.net/1974/1217`

[MFS11] Margaria, Tiziana ; Floyd, Barry D. ; Steffen, Bernhard: IT Simply Works: Simplicity and Embedded Systems Design. In: *IEEE 35th Annual Computer Software and Applications Conference Workshops (COMPSACW), 2011*, 2011, S. 194–199

[MIH+12] Merten, Maik ; Isberner, Malte ; Howar, Falk ; Steffen, Bernhard ; Margaria, Tiziana: Automated Learning Setups in Automata Learning. Version: 2012. In: Margaria, Tiziana (Hrsg.) ; Steffen, Bernhard (Hrsg.): *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change* Bd. 7609. Springer Berlin Heidelberg, 2012.

[MNRS04] Margaria, Tiziana ; Niese, Oliver ; Raffelt, Harald ; Steffen, Bernhard: Efficient test-based model generation for legacy reactive systems. In: *HLDVT '04: Proceedings of the High-Level Design Validation and Test Workshop, 2004. Ninth IEEE International.* Washington, DC, USA : IEEE Computer Society, 2004. – ISBN 0–7803–8714–7, S. 95–100

[MS02] Margaria, Tiziana ; Steffen, Bernhard: Scalable System-level CTI Testing through Lightweight Coarse-grained Coordination. In: *Electronic Notes in Theoretical Computer Science* 66 (2002), Nr. 2, S. 66–83.

[MS09] Margaria, Tiziana ; Steffen, Bernhard: Agile IT: Thinking in User-Centric Models. In: Margaria, Tiziana (Hrsg.) ; Steffen, Bernhard (Hrsg.): *Leveraging Applications of Formal Methods, Verification and Validation* Bd. 17, Springer Berlin / Heidelberg, 2009 (Communications in Computer and Information Science). – ISBN 978–3–540–88479–8, S. 490–502

[MS10] Margaria, Tiziana ; Steffen, Bernhard: Simplicity as a Driver for Agile Innovation. In: *Computer* 43 (2010), Nr. 6, S. 90–92. ISSN 0018–9162

[MS12] Margaria, Tiziana ; Steffen, Bernhard: Service-Orientation: Conquering Complexity with XMDD. Version: 2012. In: Hinchey, Mike (Hrsg.) ; Coyle, Lorcan (Hrsg.): *Conquering Complexity.* Springer London, 2012. – ISBN 978–1–4471–2296–8, 217-236

[MS13] Meinke, Karl ; Sindhu, Muddassar A.: LBTest: A Learning-Based Testing Tool for Reactive Systems. In: *IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST), 2013*, 2013, S. 447–454

[MSHM11] Merten, Maik ; Steffen, Bernhard ; Howar, Falk ; Margaria, Tiziana: Next Generation LearnLib. In: Parosh Aziz Abdulla, K. Rustan M. L. (Hrsg.): *Tools and Algorithms for the Construction and Analysis of Systems* Bd. 6605, Springer Berlin / Heidelberg, 2011 (Lecture Notes in Computer Science), S. 220–223

[MSR05] Margaria, Tiziana ; Steffen, Bernhard ; Reitenspiess, Manfred: Service-Oriented Design: The Roots. Version: 2005. In: *Proc. of the 3rd Int. Conf. on Service-Oriented Computing (ICSOC 2005), Amsterdam, The Netherlands* Bd. 3826. Springer, 2005.

[Nag09] Nagel, Ralf: *Technische Herausforderungen modellgetriebener Beherrschung von Prozesslebenszyklen aus der Fachperspektive von der Anforderungsanalyse zur Realisierung*, Technische Universität Dortmund, Dissertation, Juli 2009

[Neu07] Neubauer, Johannes: *LocalChecker plugin for the jABC*, Technische Universität Dortmund, Studienarbeit, July 2007. `http://hdl.handle.net/2003/30118`

[Neu14] Neubauer, Johannes: Higher-Order Process Engineering: The Technical Background / Technische Universität Dortmund. Version: April 2014. `http://hdl.handle.net/2003/33102`. 2014. – Forschungsbericht

[NFTJ06] Nebut, Clémentine ; Fleurey, Franck ; Traon, Yves L. ; Jézéquel, Jean-Marc: Automatic Test Generation: A Use Case Driven Approach. In: *IEEE Transactions on Software Engineering* 32 (2006), Mar, Nr. 3, S. 140–155.

[Nie03] Niese, Oliver: *An Integrated Approach to Testing Complex Systems*, University of Dortmund, Germany, Diss., 2003

[NLS12] Naujokat, Stefan ; Lamprecht, Anna-Lena ; Steffen, Bernhard: Loose Programming with PROPHETS. In: Lara, Juan de (Hrsg.) ; Zisman, Andrea (Hrsg.): *Proc. of the 15th Int. Conf. on Fundamental Approaches to Software Engineering (FASE 2012), Tallinn, Estonia* Bd. 7212, Springer Heidelberg, 2012 (LNCS), S. 94–98

[NNL+13] Naujokat, Stefan ; Neubauer, Johannes ; Lamprecht, Anna-Lena ; Steffen, Bernhard ; Jörges, Sven ; Margaria, Tiziana: Simplicity-First Model-Based Plug-In Development. In: *Software: Practice and Experience* 44 (2013), December, Nr. 3, S. 277–297. first published online

[NS13] Neubauer, Johannes ; Steffen, Bernhard: Plug-and-Play Higher-Order Process Integration. In: *IEEE Computer* 46 (2013), August, Nr. 11, S. 56–62. ISSN 0018–9162

[NSB+12] Neubauer, Johannes ; Steffen, Bernhard ; Bauer, Oliver ; Windmüller, Stephan ; Merten, Maik ; Margaria, Tiziana ; Howar, Falk: Automated continuous quality assurance. In: *Formal Methods in Software Engineering: Rigorous and Agile Approaches (FormSERA), 2012*, 2012, 37–43

[NSM+01] Niese, Oliver ; Steffen, Bernhard ; Margaria, Tiziana ; Hagerer, Andreas ; Brune, Georg ; Ide, Hans-Dieter: Library-Based Design and Consistency Checking of System-Level Industrial Test Cases. Version: 2001. In: Hussmann, Heinrich (Hrsg.): *Fundamental Approaches to Software Engineering* Bd. 2029. Springer Berlin / Heidelberg, 2001. – ISBN 978–3–540–41863–4, S. 233–248

[NSM13] Neubauer, Johannes ; Steffen, Bernhard ; Margaria, Tiziana: Higher-Order Process Modeling: Product-Lining, Variability Modeling and Beyond. In: *Electronic Proceedings in Theoretical Computer Science* 129 (2013), S. 259–283.

[NWS14] Neubauer, Johannes ; Windmüller, Stephan ; Steffen, Bernhard: Risk-Based Testing via Active Continuous Quality Control. In: *International Journal on Software Tools for Technology Transfer* (2014), S. 1–23.

[OA99] Offutt, Jeff ; Abdurazik, Aynur: Generating Tests from UML Specifications. In: France, Robert (Hrsg.) ; Rumpe, Bernhard (Hrsg.): *Proceedings of the 2Nd International Conference on The Unified Modeling Language: Beyond the Standard*, Springer Berlin / Heidelberg, 1999, S. 416–429

[PVY99] Peled, Doron ; Vardi, Moshe Y. ; Yannakakis, Mihalis: Black Box Checking. In: Wu, Jianping (Hrsg.) ; Chanson, Samuel T. (Hrsg.) ; Gao, Qiang (Hrsg.): *Proc. FORTE '99*, Kluwer Academic, 1999, S. 225–240

[RMSM08] Raffelt, Harald ; Margaria, Tiziana ; Steffen, Bernhard ; Merten, Maik: Hybrid Test of Web Applications with Webtest. In: *TAV-WEB '08: Proceedings of the 2008 workshop on Testing, analysis, and verification of web services and applications*. New York, NY, USA : ACM, 2008. – ISBN 978–1–60558–053–1, S. 1–7

[RMSM09] Raffelt, Harald ; Merten, Maik ; Steffen, Bernhard ; Margaria, Tiziana: Dynamic testing via automata learning. In: *International Journal on Software Tools for Technology Transfer (STTT)* 11 (2009), Nr. 4, S. 307–324. – ISSN 1433–2779

[RS06] Raffelt, Harald ; Steffen, Bernhard: LearnLib: A Library for Automata Learning and Experimentation. Version: 2006. In: Baresi, Luciano (Hrsg.) ; Heckel, Reiko (Hrsg.): *Fundamental Approaches to Software Engineering* Bd. 3922. Springer Berlin / Heidelberg, 2006. – ISBN 978–3–540–33093–6, S. 377–380

[RSBM09] Raffelt, Harald ; Steffen, Bernhard ; Berg, Therese ; Margaria, Tiziana: LearnLib: a framework for extrapolating behavioral models. In: *International Journal on Software Tools for Technology Transfer (STTT)* 11 (2009), Nr. 5, S. 393–407. – ISSN 1433–2779

[RSM08] Raffelt, Harald ; Steffen, Bernhard ; Margaria, Tiziana: Dynamic Testing Via Automata Learning. In: Yorav, Karen (Hrsg.): *Hardware and Software: Verification and Testing* Bd. 4899, Springer Berlin / Heidelberg, 2008 (Lecture Notes in Computer Science), S. 136–152

[SGA07] Sutton, Michael ; Greene, Adam ; Amini, Pedram: *Fuzzing: Brute Force Vulnerability Discovery*. Pearson Education, 2007. – ISBN 978–0321446114

[SHM11] Steffen, Bernhard ; Howar, Falk ; Merten, Maik: Introduction to Active Automata Learning from a Practical Perspective. Version: 2011. In: Bernardo, Marco (Hrsg.) ; Issarny, Valérie (Hrsg.): *Formal Methods for Eternal Networked Software Systems* Bd. 6659. Springer Berlin Heidelberg, 2011.

[Sin13] Sindhu, Muddassar A.: *Algorithms and Tools for Learning-based Testing of Reactive Systems*, School of Computer Science and Communication, KTH Royal Institute of Technology, Diss., 2013

[SMCB96] Steffen, Bernhard ; Margaria, Tiziana ; Classen, Andreas ; Braun, Volker: Incremental Formalization: A Key to Industrial Success. In: *Software - Concepts and Tools* 17 (1996), Nr. 2, S. 78–95

[SMN+07] Steffen, Bernhard ; Margaria, Tiziana ; Nagel, Ralf ; Jörges, Sven ; Kubczak, Christian: Model-Driven Development with the jABC. Version: 2007. In: Bin, Eyal (Hrsg.) ; Ziv, Avi (Hrsg.) ; Ur, Shmuel (Hrsg.): *Hardware and Software, Verification and Testing* Bd. 4383. Springer Berlin / Heidelberg, 2007. – ISBN 978–3–540–70888–9, 92-108

[SN11] Steffen, Bernhard ; Neubauer, Johannes: Simplified Validation of Emergent Systems through Automata Learning-Based Testing. In: *34th IEEE Software Engineering Workshop (SEW), 2011*, 2011, S. 84–91

[Ste91] Steffen, Bernhard: Data Flow Analysis as Model Checking. In: *Proceedings of the International Conference on Theoretical Aspects of Computer Software*, Springer-Verlag, 1991. – ISBN 3–540–54415–1, 346–365

[Ste93] Steffen, Bernhard: Generating Data Flow Analysis Algorithms from Modal Specifications. In: *Science of Computer Programming* 21 (1993), Nr. 2, S. 115–139

[Sul09] Sultanow, Eldar: Softwareprojekte managen mit EPF und xMDD. In: *iX* (2009), Nr. 4/2008, 98-102. `http://heise.de/-227154`

[TVKB01] Tahat, Luay H. ; Vaysburg, Boris ; Korel, Bogdan ; Bader, Atef J.: Requirement-Based Automated Black-Box Test Generation. In: *25th Annual International Computer Software and Applications Conference, 2001. COMPSAC 2001.*, 2001, S. 489–495

[TX08] Taneja, Kunal ; Xie, Tao: DiffGen: Automated Regression Unit-Test Generation. In: *23rd IEEE/ACM International Conference on Automated Software Engineering, 2008. ASE 2008.*, 2008, S. 407–410

[UL07] Utting, Mark ; Legeard, Bruno: *Practical Model-Based Testing: A Tools Approach*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2007 `http://www.cs.waikato.ac.nz/research/mbt/`. – ISBN 9780080466484

[WEMS+12] Winter, Mario ; Ekssir-Monfared, Mohsen ; Sneed, Harry M. ; Seidl, Richard ; Borner, Lars ; Dubau, Jürgen (Hrsg.): *Der Integrationstest: Von Entwurf und Architektur zur Komponenten- und Systemintegration*. Carl Hanser Verlag, 2012. – ISBN 978–3446425644

[Win11] Windmüller, Stephan: Offline Validation of Firewalls. In: *Proceedings of the 2011 IEEE 34th Software Engineering Workshop*. Washington, DC, USA : IEEE Computer Society, 2011 (SEW '11), S. 36–41

[WM08] White, Stephen A. ; Miers, Derek ; Fischer, Layna (Hrsg.): *BPMN Modeling and Reference Guide*. Future Strategies Inc., 2008

[WNS+13] Windmüller, Stephan ; Neubauer, Johannes ; Steffen, Bernhard ; Howar, Falk ; Bauer, Oliver: Active Continuous Quality Control. Version: 2013. In: *16th International ACM SIGSOFT Symposium on Component-Based Software Engineering*. New York, NY, USA : ACM SIGSOFT, 2013 (CBSE '13).

[XDTC05] Xiaoying, Bai ; Dong, Wenli ; Tsai, Wei-Tek ; Chen, Yinong: WSDL-Based Automatic Test Case Generation for Web Services Testing. In: *Service-Oriented System Engineering, 2005. SOSE 2005. IEEE International Workshop*, 2005, S. 207–212