Automatisierte Erzeugung konkreter Testfälle für Webanwendungen aus einem textbasierten Modell

Arne-Michael Törsel
Arne-Michael.Toersel@fh-stralsund.de

Gerold Blakowski
Gerold.Blakowski@fh-stralsund.de

Abstract: Anwendungsmodelle als Basis für die Testfallableitung in der Domäne der Webanwendungen sind ein aktuelles Forschungsthema. Ein textbasiertes Anwendungsmodell als Grundlage der Ableitung von funktionalen Testfällen wird beschrieben. Es integriert Variablen zur Modellierung von Zustandstransformationen und Bedingungsausdrücken, die die Formulierung und Konkretisierung einfacher Testorakel als auch die Bestimmung einer Ausführungsreihenfolge der Testfälle erlaubt. Ein Prototyp wird auf der openArchitectureWare Plattform umgesetzt und erlaubt die Ableitung von abstrakten Testfällen unter Einbeziehung von Testdaten und die Testfalltransformation zu konkreten, ausführbaren Testskripten für Testautomatisierungswerkzeuge mit minimal notwendigem Aufwand für den Tester.

1 Einleitung

Webanwendungen sind allgegenwärtig und werden zunehmend komplexer. Die Folgen von Softwarefehlern in den Anwendungen sind mitunter schwerwiegend. Dementsprechend werden an das funktionale wie nichtfunktionale Testen von Webanwendungen gewöhnlich hohe Anforderungen mit Automatisierungen im Testprozess gestellt. Die Wartung von (automatisierten) Testfallsuiten für den funktionalen Test ist in einem Umfeld, das von iterativen Entwicklungsprozessen mit häufigen Erweiterungen und Änderungen geprägt ist, jedoch aufwändig. Während einfache Änderungen am Nutzeroberflächenlayout gewöhnlich durch Abstraktionsmechanismen von Testwerkzeugen handhabbar sind, wirken sich in der Regel auch fachliche Änderungen der Anwendung auf der Nutzeroberfläche aus und erfordern entsprechende Anpassungen der automatisierten Testfälle. Mitunter ist die Testfallwartung aufwändiger als die eigentliche Umsetzung geänderter oder neuer Anforderungen! Kaner et al. [KPB01] benennen wiederholten Wartungsaufwand sogar als den hauptsächlichen Grund, dass Testautomatisierung auf der Ebene von grafischen Nutzeroberflächen fehlschlägt.

Modellbasiertes Testen verspricht Effizienzgewinne durch eine Automatisierung des Testfalldesigns in Szenarien, in denen als Artefakt des Entwicklungsprozesses bereits eine umfassende formale Spezifikation vorliegt oder aber der Wartungsaufwand einer Testfallsuite durch häufige Änderungen der Anwendung so groß ist, dass der Aufwand für die Erstellung und Pflege einer redundanten Spezifikation des Systems zur Testfallableitung gerechtfertigt ist [Pre06]. Es wird dabei angenommen, dass die Ableitung der Testfälle aus dem Modell und ihre Konkretisierung quasi „kostenlos“ möglich sind. Die Frage nach der Anwendbarkeit und Effizienz solcher Verfahren für den Test von Webanwendungen

ist speziell unter dem Gesichtspunkt der Testfallwartung also berechtigt. Nach der Einordnung modellbasierter Testverfahren in [UL07] stellt dabei die Ableitung von Testfällen mit Testorakeln aus einem Verhaltensmodell der Anwendung die potenziell gewinnbringendste aber auch konzeptionell und technisch herausfordernste Interpretation modellbasierten Testens dar. Geeignete Modelltypen und Verfahren in dieser Anwendungsdomäne sind deshalb Gegenstand der Forschung. Neben der erhöhten Kohärenz der Testfälle zur Anwendungsspezifikation und Kontrolle der Testabdeckung soll eine Entkopplung der Testfälle von ihrer Implementation in konkreten Testautomatisierungswerkzeugen zur Verringerung des Wartungsaufwands bewirkt werden.

Erste Ansätze wie in [RT01], in dem die UML zur Modellierung von Webanwendungen genutzt wird, bilden strukturelle Aspekte der Anwendung ab. Solche Ansätze sind nur eingeschränkt für die Ableitung von Testfällen nutzbar. Zur Modellierung des dynamischen Verhaltens, zum Beispiel von Nutzereingaben und erwarteten Reaktionen der getesteten Anwendung, existieren Arbeiten, die auf Graphenstrukturen beruhen [AOA05], [BLZH07]. In [AOA05] wird ein hierarchisches Zustandsübergangssystem zur Modellierung genutzt. Die Zustandsübergänge im Modell sind mit Eingabeconstraints einer eigenen Grammatik annotiert. Sie definieren den Typ der Eingaben, ihre Eingabereihenfolge und Kardinalität sowie die Propagation der Daten im Testverlauf. Die erzeugten Testfälle müssen manuell mit konkreten Testdaten angereichert werden. Die von Belli et al. [BLZH07] vorgestellte Methode beschreibt die Nutzung sogenannter Ereignissequenzgraphen zur hierarchischen Modellierung der Anwendung. Eingaben für die Testfälle und Randbedingungen werden mithilfe von Entscheidungstabellen im Modell integriert. Konkrete Testdaten sind direkt in den Entscheidungstabellen enthalten. Durch Traversierung des Graphen werden Testsequenzen mit den erforderlichen Eingaben ermittelt. Eine Inversion der vorhandenen Kanten erlaubt zudem die Erzeugung von Negativtestsequenzen, also Testfälle für unerwünschtes Anwendungsverhalten. Mit beiden Ansätzen sind aufgrund der hierarchischen Modellierung auch größere Anwendungen strukturiert abbildbar. Die Übertragung und Ergänzung der erzeugten abstrakten Testfälle in konkrete Testskripte für ein Testwerkzeug erfolgt semiautomatisch beziehungsweise manuell, was speziell in Wartungsszenarien einen nicht unerheblichen Aufwand bedeutet.

In beiden Ansätzen ist als Testorakel der erwarteten Endzustand im Modell ableitbar. Die konkrete Implementation des Testorakels obliegt jedoch dem Testentwickler. Beim Blackboxtest von Webanwendungen ist das gewöhnlich nur durch eine Auswertung des HTML-Quelltextes der Nutzeroberfläche möglich. Dabei tritt häufig das Problem der in [AOA05] beschriebenen geringen Beobachtbarkeit der Testauswirkung auf. In einer Webanwendung, die typischerweise in einer Mehrschichtenarchitektur implementiert ist, wirkt sich eine Interaktion des Nutzers oft bis zur Datenhaltungsschicht aus, wobei nur Teile dieser Auswirkung unmittelbar in der Nutzeroberfläche beobachtbar sind.

Zur partiellen Lösung dieses Problems wird im nachfolgend vorgeschlagenen textbasierten Ansatz ein Graphmodell um die Abbildung von Anwendungszustandsinformationen mittels globaler Variablen und Datenflüssen sowie Bedingungsausdrücken auf den Variablen erweitert. Der vorgeschlagene Modelltyp erlaubt die Verfolgung von relevanten testfallübergreifenden Zustandsänderungen im System zur Ableitung konkreter HTML-textmatchingbasierter Testorakel auf Basis von Testdaten und Zustandsvariablen. Damit

ist zum Beispiel auch die verzögerte Beobachtung der Reaktion auf einen Teststimulus möglich. Ein wichtiges Ziel der Entwicklung des Prototyps war es zudem, die erforderliche Intervention des Testentwicklers bei der Testfallgenerierung bis zur Ausführung zu minimieren. Mittels eines Testtreibers ausführbare Testskripte sollen ohne weitere manuelle Arbeitsschritte aus dem Modell erzeugt werden.

2 Modell und Testfallableitung

Die Webanwendung wird ähnlich zu den Ansätzen in [AOA05] und [BLZH07] als zusammenhängende Graphstruktur modelliert. Die grundlegenden Bestandteile sind Sichten, das heißt Webseiten beziehungsweise Seitenfragmente der Anwendung (Knoten im Graphen) und Übergänge zu anderen Sichten (gerichtete Kanten im Graphen). Der Übergang zu einer anderen Sicht wird in der Regel durch eine Nutzerinteraktion als Stimulus ausgelöst. Für Webanwendungen bedeutet das in der Regel, dass der Nutzer entweder einen Link wählt oder ein Formular mit Eingabedaten verschickt. Denkbar sind jedoch auch andere Stimuli, wie etwa zeitgesteuerte Übergänge. Schleifenkanten bilden den Verbleib auf einer Sicht als Reaktion auf den Stimulus ab.

Beim funktionalen Testen von Webanwendungen ist zu beachten, dass zwischen Nutzer und Anwendung in der Regel ein Sitzungszustand (Session) besteht. In der Sitzung werden Daten des Anwendungszustands wie zum Beispiel der Loginstatus oder ein Warenkorb gespeichert. Ein Testtreiber kann im Testverlauf diesen impliziten Zustand zurücksetzen und so einen neuen Client simulieren. Zur Abbildung dieses Anwendungszustands können im Modell Variablen (Zeichenketten, Zahlen und Boolesche Werte) in drei Gültigkeitsbereichen - *Global*, *Session* und *Lokal* - deklariert werden:

- Global: Variablen sind gültig über den Ablauf aller Testfälle
- Session: Variablen sind gültig über den Ablauf aller Testfälle, werden aber nach einer simulierten Sitzung auf den im Modell definierten Ausgangswert zurückgesetzt
- Lokal: Variablen sind gültig in einem Sichtübergang

Modellvariablen werden zur Formulierung von mathematischen, logischen und Zeichenkettenausdrücken verwendet:

- Als Guardbedingung für Übergänge im Modell: Übergänge können bei der Testfallerzeugung nur ausgewählt werden, wenn alle Guardbedingungen zu „wahr“ ausgewertet werden.
- Für Zeichenketten, die bei der Erzeugung von Testfällen zur Formulierung von Testorakeln dienen und über Textmatchingmechanismen auf dem Webseitenquelltext operieren. Im Prototyp wird im gesamten Seitenquelltext nach dem Zeichenkettenausdruck gesucht. Denkbar wären Erweiterungen wie reguläre Ausdrücke oder eine Vorbehandlung des Quelltexts wie in [SPE+07] beschrieben, zum Beispiel eine

vorhergehende Ausfilterung aller HTML-Tags, um Fehltreffer beim Matching zu verringern.

Testdaten werden als Tupel gruppiert zum Beispiel über eine Exceltabelle bereitgestellt und als strukturierte Variable (assoziatives Feld) in das Modell integriert. Damit sind sie analog zu den einfachen Datentypen ebenso in Ausdrücken verwendbar, können jedoch nicht als Ziel von Zuweisungen verwendet werden.

Im folgenden Beispiel ist ein Ausschnitt aus einem Anwendungsmodell bestehend aus zwei Sichten abgebildet. Die Darstellung erfolgt in der domänenspezifischen Sprache des Prototyps (Erläuterung der Elemente im Anschluss an das Modell):

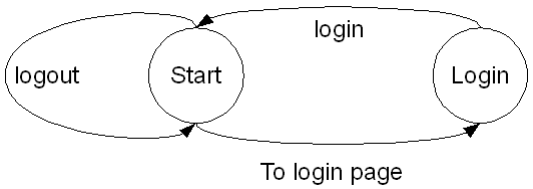
Listing 1: Webanwendungsmodell

```
1   Session-State:
2   Tuple user = "User"
3   Boolean loggedIn = "false"
4
5   Views:
6   View Start {
7     Text "Welcome |user.username|" When "loggedIn == true"
8     Text "Please log in" When "loggedIn == false"
9
10    Transition Login {
11      When "loggedIn == false"
12      Link "login"
13    }
14
15    Transition Start {
16      When "loggedIn == true"
17      Button "logout"
18      Assign loggedIn = "false"
19    }
20  }
21
22  View Login {
23    Text "Please enter login information"
24
25    Transition Start {
26      Required "username" = "|user.username|"
27      Required "password" = "|user.password|"
28      Button "login.submit"
29      Assign loggedIn = "true"
30    }
31  }
```

Das Modell enthält zwei mit der Direktive *View* deklarierte Sichten („Start“ in Zeile 6, „Login“ in Zeile 22). Es existieren drei Übergänge (Direktive *Transition*):

- Von Start zu Login in Zeile 10 über einen Link (Direktive *Link*)

Abbildung 1: Webanwendungsmodell als Graph



- Von Start zu Start in Zeile 15 über einen Button (Direktive *Button*)
- Von Login zu Start in Zeile 25 über eine Formulareingabe/Button (Direktive *Required* kennzeichnet benötigte Eingaben)

In Zeile 3 wird eine Boolesche Variable „loggedIn“ deklariert. Die Variable wird in einer Bedingung (Direktive *When*) als Guard für den umschließenden Übergang verwendet (Zeile 16). Mit der Direktive *Tuple* in Zeile 2 wird ein Testdatensatz aus der Menge „User“ in der Feldvariable „user“ abgelegt. Die Felder „username“ und „password“ dieses Testdatensatzes werden in den Zeilen 26 und 27 den benötigten Eingaben (Direktive *Required*) zugewiesen. Die Zeile 7, 8 und Zeile 23 enthalten in der jeweiligen Sicht als vorhanden erwartete Textausdrücke. In den Zeilen 7 und 8 sind die Ausdrücke an eine Bedingung (Direktive *When*) gekoppelt. Die entsprechenden Ausdrücke sollen also nur dann in der Sicht enthalten sein, wenn die Variable „loggedIn“ den entsprechenden Wert besitzt. Die Textausdrücke werden bei der Testfallerzeugung für die Generierung textmatchingbasierter Testorakel genutzt. Zuweisungen an Variablen erfolgen durch die Direktive *Assign* im Modell in den Zeilen 18 und 29. Das Modell ist als Graph in Abbildung 1 dargestellt.

Zur Testfallerzeugung wird der gerichtete Graph des Modells zunächst mit einer Tiefensuche (Depth First Search) vom Startknoten aus traversiert. Der Startknoten ist eine speziell markierte Sicht, für die der URL zum direkten Aufruf durch den Testtreiber in einer externen Konfigurationsdatei hinterlegt ist. In von der Tiefensuche erreichten Knoten werden mehrere Pfade aggregiert: die Baumkanten der Tiefensuche sowie weitere Baumpfade von deren Knoten aus der aktuelle Knoten über Vorwärtskanten erreichbar ist. Die aggregierten Pfade dienen als Alternativen bei der Pfadauswahl zur Testfallerzeugung. Das Selektionskriterium für die Testfallerzeugung ist der angestrebte Besuch aller Sichten und aller Übergänge, die mit Nutzereingaben verbunden sind. Die erzeugten Testfälle werden in einem abstrakten Testfallformat gespeichert.

Generell ist für die Wartbarkeit einer Testfallbibliothek anzustreben, dass Testfälle unabhängig voneinander sind. Dies ist in der Praxis jedoch oft nicht umsetzbar und die Ausführbarkeit eines Testfalls hängt von der vorigen Ausführung eines oder mehrerer anderer Testfälle ab. Zum Beispiel muss ein Nutzer sich zunächst einloggen, um dann einen Beitrag in einem Webforum schreiben zu können. Aufgrund der integrierten Bedingungsausdrücke an den Übergängen im vorgeschlagenen Modelltyp ist die Ableitung von Abhängigkeiten der Testfälle und die Ermittlung einer Ausführungsreihenfolge möglich.

Zur Sortierung der Testfälle zu einer möglichen Ausführungsreihenfolge wird ein Permutationsalgorithmus genutzt, der die Ausführbarkeit eines Pfades über die Auswertung der Guardbedingungen und Simulation der Zuweisungen überprüft und bei Nichterfüllung die Pfadreihenfolge umsortiert. Eine Heuristik, die die „Kosten“ eines Pfades abschätzt, sortiert die Pfade vor, um den Permutationsaufwand zu verringern. Auf diese Weise werden Pfade bevorzugt ausgeführt, die keine Auswirkungen auf den Anwendungszustand in Form von Variablenzuweisungen oder Nutzereingaben verursachen. Prinzipiell nutzt der Algorithmus einen brute-force-Ansatz, dessen Rechenaufwand bei größeren Modellen entsprechend schlecht skaliert. Bei der Erweiterung des Prototyps soll ein effizienterer Algorithmus zur Pfadauswahl umgesetzt werden.

Beim Ablauf eines Pfades zur Testfallerzeugung durch den Generator finden bei der Verarbeitung einer Sicht folgende grundlegende Schritte statt:

1. Textmatchingbasierte Orakel gemäß den in der Sicht deklarierten Textausdrücken werden erzeugt und als Prüfungstestschritte im abstrakten Testfall eingefügt.
2. Zur Aktivierung des Übergangs notwendige Eingaben werden über die Auswertung der Ausdrücke im Modell erzeugt und in Form von Nutzereingaben oder der Auswahl eines Links als Testschritte eingefügt. Über Zuordnungsdateien werden dabei die logischen Eingabefeld- und Linknamen auf die implementationsspezifischen Bezeichner und URLs abgebildet, um die Abstraktion im Modell zu wahren. Damit ist im Änderungsfall die Zuordnung nur an einer einzigen, zentralen Stelle anzupassen.
3. Zuweisungen an Variablen im Übergang (Direktive *Assign*, siehe Beispiel) werden ausgeführt
4. 1-3 werden für die nun aktuelle Sicht wiederholt

3 Architektur und Evaluation des Prototyps

Bei der Umsetzung des Konzepts in einem Prototyp wurde mehreren quelloffenen Frameworks genutzt. Das System besteht aus mehreren Komponenten, die bei der Testfallgenerierung und -konkretisierung aufeinander aufbauen. Die Komponenten kommunizieren ausschließlich über die jeweiligen Ein- und Ausgabedaten und sind damit relativ unabhängig voneinander, was die Weiterentwicklung einzelner Komponenten begünstigt.

Metamodell für die Modellierung von Webanwendungen Für die Definition des Webanwendungsmetamodells wird das openArchitectureWare-Framework[1] („oAW“) genutzt. Das Framework setzt auf dem Eclipse Modelling Framework[2] und dessen Meta-Metamodell „Ecore“ auf. Mittels oAW-Xtext wurde die domänenspezifische Sprache definiert und ein Parser für die Programmiersprache Java generiert. Das

[1] http://www.openarchitectureware.org
[2] http://www.eclipse.org/modeling/emf

Meta-Metamodell ist sehr kompakt und besteht aus insgesamt 16 Regeln zur Definition von Modellelementen zur Beschreibung von Struktur, Zustand und Verhalten der Anwendung (siehe Listing des Metamodells im Anhang).

Editor für Webanwendungsmodelle Der für die Eclipse IDE von openArchitectureWare aus dem Anwendungsmetamodell generierte Editor bietet zum Beispiel eine automatische Vervollständigungsfunktion für Schlüsselwörter und Variablennamen sowie Syntaxhighlighting und -prüfung. Die Bearbeitung eines textbasierten Modells ist damit für geübte Anwender gewöhnlich effizienter als die Bedienung eines grafischen Editors. Eine grafische Darstellung und Bearbeitung der Modelle ist zum Beispiel unter Nutzung des Eclipse Graphical Modelling Framework Projektes[3] jedoch weiterhin möglich.

Testfallgenerator Der Testfallgenerator ist in Java implementiert. Zur internen Verarbeitung des Strukturmodells, also des gerichteten Graphen, wird das JUNG Framework[4] benutzt, das Datenstrukturen und grundlegende Graphalgorithmen zur Verfügung stellt. Für die Verwaltung der Modellvariablen in den Modellinstanzen ist im Hintergrund die Mozilla JavaScript-Engine Rhino[5] verantwortlich. Das bedeutet, die Modellvariablen werden während der Testfallerzeugung in einem Ausführungskontext / Gültigkeitsbereich (Context / Scope) der Engine verwaltet. Die Engine wird dann zur Auswertung von Ausdrücken in Bedingungen und Zuweisungen an Variablen verwendet. Ausdrücke sind im Modell also in Javascript-Syntax zu formulieren.

Metamodell für abstrakte Webanwendungstestfälle Das mittels Ecore definierte Metamodell dient der Definition eines Metamodells für die abstrakte Repräsentation von Webanwendungstestfällen. Modellinstanzen enthalten abstrakte Testschritte wie Nutzereingaben und die Prüfung von Ausgaben. Die Instanzen werden bei der Testfallableitung durch den Generator erzeugt.

Transformation abstrakter Testfälle in ausführbare Testfälle Mittels des oAW-Xpand Frameworks werden aus Instanzen des abstrakten Webanwendungstestfallmodells konkrete, ausführbare Testfälle erzeugt. Xpand ist auf die Erzeugung von Textdateien spezialisiert und bietet eine Templatesprache zur Verarbeitung der Eingabemodelle. Im Prototyp werden XML-basierte Testfallskripte für das Testwerkzeug Canoo Webtest[6] erzeugt. Xpand ist sehr flexibel, so können aus den abstrakten Testfällen auch Steueranweisungen für weitere Testwerkzeuge erzeugt werden. In einer studentischen Arbeit wurde zum Beispiel ein Xpand-Template für das HTML-Unit-Testframework[7] erstellt.

Eine erste Evaluation fand zur Testgenerierung für ein Onlineforumsystem[8] statt. Dazu wurden 17 Sichten und 41 Übergänge modelliert sowie entsprechende Testdaten bereitge-

[3] http://www.eclipse.org/modeling/gmf
[4] http://jung.sourceforge.net
[5] http://www.mozilla.org/rhino
[6] http://webtest.canoo.com
[7] http://htmlunit.sourceforge.net
[8] http://www.jforum.net

stellt. Das Anwendungsmodell hat einen Umfang von 193 Modelldirektiven[9] mit 11 Variablen. Es konnten damit typische Bestandteile des Forums wie Registrierung, Login/Logout, Nutzerprofil, Beiträge und private Nachrichten abgebildet werden. Die generierten Testfälle bestehen aus insgesamt 404 Testschritten. Solche Testschritte sind zum Beispiel: Betätigen eines Links, Eintragen eines Testdatums in ein Eingabefeld, Prüfen ob ein bestimmter Text im HTML-Quelltext gefunden werden kann etc..

Generell sind Anwendungsfälle mit einer hohen „Sichtbarkeit“ der Auswirkung von Nutzerinteraktionen auf der Nutzeroberfläche für die Testfallableitung mit Testorakel mit dem vorgestellten Ansatz geeignet, insbesondere Szenarien die aus Dateneingaben und korrespondierenden Ausgaben bestehen. Dementsprechend sind Anwendungsfälle gänzlich ohne beobachtbare Auswirkung auf der Nutzeroberfläche nicht testbar - das ist jedoch eine generelle Einschränkung von Blackbox-Tests. Hier ist der Test mit Whitebox-Verfahren zu ergänzen. Aus den mit dem Testfallgenerator erzeugten abstrakten Positiv-Testfällen wurden ohne zusätzlich nötigen manuellen Aufwand ausführbare Testskripte erzeugt, was für den praktischen Einsatz ein wichtiges Kriterium darstellt. Verglichen mit einer manuellen Implementierung der generierten Testfälle ist die Modellierung des Evaluationsbeispiels zunächst aufwändiger. Die Modellierung bietet jedoch zunächst den Vorteil, ein systematisches Vorgehen zu unterstützen. Ein Vorteil in Bezug auf den erforderlichen Arbeitsaufwand entsteht bei späteren Änderungen des Modells und einer einfachen Neugenerierung der kompletten Testsuite. Zur weiteren Evaluation des Ansatzes ist die Modellierung von Anwendungsfällen anderer Webanwendungen geplant.

Ein aktueller Trend bei der Entwicklung von Webanwendungen ist der Einsatz partieller Updates der Nutzeroberfläche („Ajax“ [Gar05]). Beim Einsatz des aktuellen Prototyps würde eine Berücksichtigung partieller Updates zu einer höheren Anzahl zu modellierender eigenständiger Sichten und damit zu einer stärkeren Fragmentierung des Modells führen. Ein vermutlich effizienterer Ansatz ist die Modellierung von Teilsichten und ihre Aggregation zu logischen Gesamtsichten. Das ist mit dem gegenwärtigen Prototyp noch nicht umsetzbar. Die Übertragung auf andere Anwendungsdomänen wie zum Beispiel den Test allgemeiner Systeme mit grafischen Nutzeroberflächen ist nicht einfach möglich, da im Ansatz zwei spezielle Eigenschaften der Sprache HTML genutzt werden. Erstens beschränken die durch HTML vergleichsweise begrenzten Darstellungs- und Interaktionsmöglichkeiten die Komplexität des Metamodells. Ein generischer Ansatz zur Modellierung aller denkbaren Systeme mit grafischen Nutzeroberflächen würde vermutlich zu einer vom Nutzer nicht mehr gewinnbringend handhabbaren Komplexität des Metamodells führen. Zweitens vereinfacht die maschinelle Verarbeitbarkeit des HTML-Quelltextes der Nutzeroberfläche erheblich die Formulierung konkreter Testorakel und die Konkretisierung von Testfällen für Testwerkzeuge.

Die Einsatzfähigkeit des beschriebenen Testverfahrens hängt erheblich von der Testbarkeit der unterliegenden Anwendung ab. Nur wenn alle Eingabeelemente und Links eindeutig identifizierbar sind, ist zum Beispiel die Transformation des abstrakten Testfalls zum konkreten Testskript über Zuordnungsdateien ohne Nutzerintervention möglich. Zur grundlegenden Verbesserung der Testbarkeit ist zudem denkbar, einen speziellen Testmodus in der Anwendung bereitzustellen, der im Testverlauf anwendungsinterne Zustandsdaten zur

[9] Das gesamte Metamodell mit allen Direktiven ist im Anhang dargestellt

Beobachtung über die Nutzeroberfläche offenlegt. Solche Testbarkeitsanforderungen sind bei der Planung und Umsetzung der Anwendung zu berücksichtigen.

4 Fazit und Ausblick

Es wurde ein textbasierter Modelltyp für die Ableitung von Testfällen für Webanwendungen vorgestellt. Mit dem Prototyp ist die Ableitung von konkreten, ausführbaren Testfällen inklusive auf dem Quelltext der Nutzeroberfläche operierender Testorakel möglich. Nach dem gegenwärtigen Stand ist das System gut geeignet, eine Testsuite für Smoketests mit kontrollierter Testabdeckung zu generieren. Besonders in Wartungsszenarien ist die Möglichkeit, nach einer Anpassung des Modells ohne weitere manuelle Intervention die Testsuite neu zu generieren, ein Vorteil.

Um die für eine Regressionstestsuite erforderliche Testtiefe zu erreichen, sollen in der zukünftigen Weiterentwicklung das Anwendungsmetamodell und der Testgeneratoralgorithmus erweitert werden. Hier ist das für den Anwender optimale Verhältnis zwischen der Mächtigkeit des Modells und dessen Komplexität und Benutzbarkeit zu ermitteln. Außerdem ist die Laufzeiteffizienz des Algorithmus für den praktischen Einsatz zu verbessern. Zur besseren Handhabung komplexer Systeme ist es in der fortführenden Entwicklung erforderlich, das Modell in mehrere Teilmodelle zerlegen zu können. Dazu soll ähnlich den in [AOA05] und [BLZH07] beschriebenen Ansätzen über Komposition und Hierarchisierung die Modellierung größerer Anwendungen ermöglicht werden.

Literatur

[AOA05] Anneliese Amschler Andrews, Jeff Offutt und Roger T. Alexander. Testing Web applications by modeling with FSMs. *Software and System Modeling*, 4(3):326–345, 2005.

[BLZH07] Fevzi Belli, Michael Linschulte, Ralf Zirnsak und Günter Hofmann. 'Negativ'-Tests interaktiver Systeme und ihre Automatisierung. In Wolf-Gideon Bleek, Henning Schwentner und Heinz Züllighoven, Hrsg., *Software Engineering (Workshops)*, Jgg. 106 of *LNI*, Seiten 35–44. GI, 2007.

[Gar05] Jesse James Garret. Ajax: A New Approach to Web Applications. http://adaptivepath.com/ideas/essays/archives/000385.php, 2005.

[KPB01] Cem Kaner, Bret Pettichord und James Bach. *Lessons Learned in Software Testing*. Wiley, 2001.

[Pre06] Alexander Pretschner. Zur Kosteneffektivität des modellbasierten Testens. In *Proc. Dagstuhl-Workshop MBEES 2006: Modellbasierte Entwicklung eingebetteter Systeme*, 2006.

[RT01] Filippo Ricca und Paolo Tonella. Analysis and Testing of Web Applications. In *ICSE*, Seiten 25–34. IEEE Computer Society, 2001.

[SPE+07] Sara Sprenkle, Lori Pollock, Holly Esquivel, Barbara Hazelwood und Stacey Ecott. Automated Oracle Comparators for Testing Web Applications. In *ISSRE '07: Proceedings*

of the The 18th IEEE International Symposium on Software Reliability, Seiten 117–126, Washington, DC, USA, 2007. IEEE Computer Society.

[UL07] Mark Utting und Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan-Kaufmann, 2007.

5 Anhang: Metamodell

Das nachfolgende Listing zeigt das komplette Metamodell für die Anwendungsmodelle. Das openArchitectureWare-Framework Xtext generiert aus diesem Metamodell Modellklassen, einen Parser und einen Editor.

```
WebApplication:
  "Global-State:"
  (globals+=Variable)*
  "Session-State:"
  (session+=Variable)*
  "Views:"
  (views+=View)*;

View:
  "View" name=ID "{"
    (title=Title)?
    (texts+=Text)*
    (transitions+=Transition)*
  "}";

Transition:
  "Transition" target=[View|ID] "{"
    (guards+=When)*
    (variables+=Variable)*
    (transferAction=TransferAction)
    (assigns+=Assign)*
  "}";

TransferAction:
  Link | Input;

Link:
  "Link" refName=STRING;

Input:
  (inputData+=InputDataDirective)*
  (button=Button)+;

InputDataDirective:
  Required | Optional;
```

```
Required:
  "Required" refName=STRING "=" varName=STRING;

Optional:
  "Optional" refName=STRING "=" varName=STRING;

Button:
  "Button" refName=STRING;

Text:
  "Text" text=STRING (condition = When)?;

Title:
  "Title" text=STRING;

Variable:
  type=DataType name=ID ("=" value=STRING)?;

Enum DataType:
  string="String" | number="Number" |
  boolean="Boolean" | tuple="Tuple";

Assign:
  "Assign" var=[Variable|ID] "=" expression=STRING;

When:
  "When" expression=STRING;
```

5.1 Kurzbeschreibung der Regeln

WebApplication Bildet die Webanwendung auf oberster Ebene ab. Eine Webanwendung besteht aus Sichten, globalen Variablen und Variablen mit Sitzungsgültigkeit.

View Bildet eine Sicht der Anwendung ab. Von einer Sicht gibt es Übergänge zu anderen Sichten. Sie enthält Textausdrücke und einen Titel.

Transition Bildet den Übergang zu einer anderen Sicht ab. Ein Übergang besteht aus Deklaration lokaler Variablen, einer Aktion die den Übergang auslöst (*TransferAction*), Bedingungsausdrücken für den Übergang sowie Zuweisungen an Variablen (aller Gültigkeitsbereiche), die bei erfolgtem Übergang ausgeführt werden.

TransferAction Aktion, die den Übergang zu einer anderen Sicht auslöst - entweder ein Link oder eine Formulareingabe.

Link Repräsentiert einen Link und wird bei der Testfallkonkretisierung dem entsprechenden HTML-Element zugeordnet.

Input Repräsentiert Formulareingaben.

InputDataDirective Repräsentiert Nutzereingaben in Formularelemente (zum Beispiel Texteingabefelder), die Eingaben werden bei der Testfallkonkretisierung dem entsprechenden HTML-Element zugeordnet.

Required Repräsentiert eine erforderliche Eingabe.

Optional Repräsentiert eine optionale Eingabe.

Button Ein Button - in HTML oft Bestandteil eines Formulars, kann jedoch auch einzeln Auslöser für einen Übergang zu einer anderen Sicht sein.

Text Textausdruck, der in der Sicht gefunden werden soll. Optional ist es möglich, eine Bedingung anzugeben, unter der dieser Ausdruck vorhanden sein soll.

Title Textausdruck, der im Titel der Sicht gefunden werden soll.

Variable Deklariert eine Variable einer der möglichen Typen und belegt die Variable optional mit einem Ausdruck.

Enum DataType Beschreibt die möglichen Datentypen. *String*, *Number* und *Boolean* sind elementare Datentypen, *Tuple* ist ein strukturierter Datentyp, der für die Aufnahme von Datentupeln aus dem Testdatenpool genutzt wird.

Assign Zuweisen eines Ausdrucks an eine Variable.

When Bedingungsausdruck für Übergange („Guard“) und bedingt in Sichten dargestellten Textausdrücken.