Birgit Hofer
(bhofer@ist.tugraz.at)
ist Dissertantin am Institut für Softwaretechnologie der Technischen Universität Graz. Zu ihren Forschungsschwerpunkten zählen Testautomatisierung und die automatische Lokalisierung von Fehlern in Software.

Prof. Dr. Franz Wotawa
(wotawa@ist.tugraz.at)
forscht und lehrt am Institut für Softwaretechnologie der Technischen Universität Graz in den Bereichen Softwaretechnologie, Testen, Diagnose, Künstliche Intelligenz und Robotik. Er ist derzeit Dekan der Fakultät für Informatik und Gründer sowie wissenschaftlicher Leiter von Softnet Austria.

Fallstudien zum Einsatz modellbasierter Testtechniken in der industriellen Praxis

Das Testen von Software zur Steigerung der Qualität gewinnt in der Praxis immer mehr an Bedeutung. Gründe dafür sind zum einen die Kosten, die bei Systemausfällen anfallen, und zum anderen das sinkende Vertrauen in Produkte und dadurch bedingt negative Auswirkungen auf die Reputation von Unternehmen. Im Jahr 2006 wurden für Österreich Kosten in Höhe von 3 bis 5 Milliarden Euro pro Jahr durch Softwarefehler geschätzt (siehe www.news.at/articles/0605/547/132184/software-fehler-oesterreich-mrd-euro-probleme-ausbildung, 2. Februar 2006). Um die direkten Kosten, die durch Systemausfälle entstehen, sowie andere negative Auswirkungen von Softwarefehlern reduzieren zu können, ist ein rigoroser Softwaretest unumgänglich. In diesem Artikel konzentrieren wir uns auf das modellbasierte Testen von Software und die Anwendung im industriellen Kontext.

Unter modellbasiertem Testen verstehen wir die automatische Erzeugung von Testfällen aus formalen Spezifikationen, welche das Systemverhalten in einer abstrakten Form beschreiben. Das Haupteinsatzgebiet des modellbasierten Testens ist der System- und Integrationstest. Mehr Informationen hinsichtlich des modellbasierten Testens und dessen Anwendung in der Praxis sind in [Bak08] zu finden. Die Anwendbarkeit des modellbasierten Testens wird unter anderem auch durch die Aktivitäten von großen Softwarehäusern, wie zum Beispiel Microsoft [vgl. Vea08], nachgewiesen. Wir werden in diesem Artikel zwei Beispiele für die Anwendung modellbasierter Testmethoden im industriellen Kontext vorstellen und uns anschließend den offenen Problemstellungen widmen. Die Anwendungen unterstreichen die Wichtigkeit des modellbasierten Testens in der Praxis und stammen von Projektaktivitäten des Instituts für Softwaretechnologie der Technischen Universität Graz in Zusammenarbeit mit Softnet Austria. Softnet Austria (siehe www.soft-net.at) ist ein privater Verein, der sich zum Ziel gesetzt hat, Ergebnisse der Forschung im Bereich Softwaretesten und -entwicklung direkt in die Praxis zu bringen. Neben wissenschaftlichen Partnern sind vor allem Unternehmen an Softnet Austria beteiligt.

Abbildung 1 gibt einen Überblick über modellbasiertes Testen. Ausgehend von einem System unter Test (SUT) wird ein formales Modell erstellt. Dieses Testmodell beschreibt eine Abstraktion des Systemverhaltens und wird direkt zur Testfallerzeugung verwendet. Diese meist abstrakten Testfälle werden durch Hinzufügen von Testdaten konkretisiert. Die konkreten Testfälle werden zur Ausführung des SUT verwendet. Im Fall einer Abweichung vom erwarteten Systemverhalten wird ein *Fail* zurückgeliefert. Ist es nicht möglich, eindeutig zu entscheiden, ob ein Systemverhalten korrekt ist oder nicht, liefert die Testausführung *Inconclusive* und andernfalls *Pass* als Ergebnis zurück.

Ein Vorteil des modellbasierten Testens ist, dass das erwartete Systemverhalten zumindest in abstrakter Form im Modell

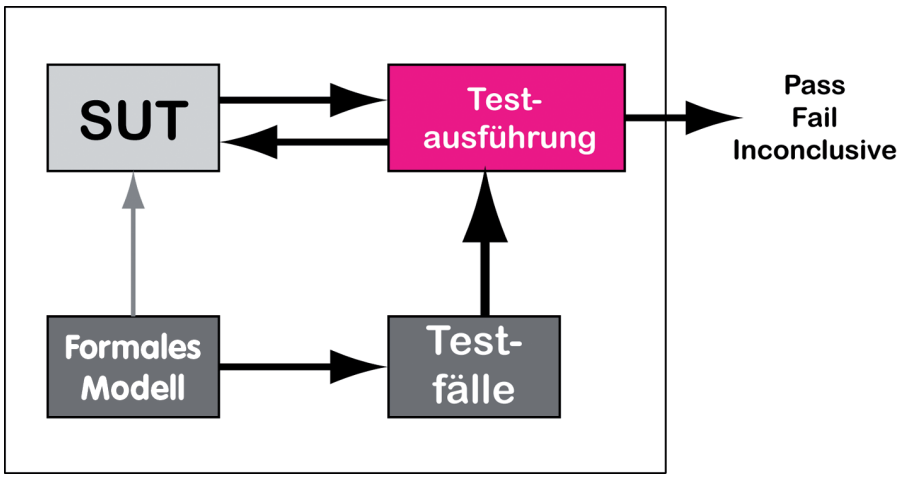

Abb. 1: Überblick über modellbasiertes Testen.

abgebildet wird. Damit kann das Orakelproblem gelöst werden und eine vollautomatisierte Testfallerzeugung durch Bereitstellung der Eingabewerte für ein SUT und der erwarteten Ausgabewerte ist möglich. Probleme des modellbasierten Testens betreffen unter anderem die Modellierung und die Testfallgenerierung. Zum einen ist es oft nicht einfach, geeignete Modelle zu erstellen, zum anderen ist die Modellerstellung mit einem Mehraufwand und zusätzlichen Ressourcen verbunden. Aufgrund der Komplexität der Testberechnung ist die Erzeugung von Testfällen aus größeren Testmodellen nicht immer in einem angemessenen Zeitrahmen möglich. Somit sind Abstraktionen und Heuristiken erforderlich.

Protokolltesten

Ein wichtiges Anwendungsgebiet des modellbasierten Testens ist die Überprüfung von Protokollimplementierungen. Als Testmodell für die Erzeugung von Tests dient eine Formalisierung des Protokolls. In diesem Zusammenhang werden endliche Automaten oder auch Labeled Transition Systems (LTS) für die Modellierung verwendet. Für LTS stehen geeignete Sprachen wie LOTOS [ISO8807] und Werkzeuge für die Testfallgenerierung, wie zum Beispiel Test Generation with Verification technology (TGV) [vgl. Jard05], zur Verfügung. Trotz der Verfügbarkeit dieser Werkzeuge und Modellierungssprachen wird modellbasiertes Testen in der Praxis noch nicht großflächig eingesetzt. Gründe hierfür sind sicherlich die Notwendigkeit der Bereitstellung von Modellen und die Berechnungskomplexität der Testfallgenerierung, die den Einsatz der Techniken bei großen Modellen schwierig macht. Aufgrund der vermehrten Verwendung von Modellen in der Softwareentwicklung (Stichwort „Model-Driven Design (MDD)“) ist damit zu rechnen, dass die Modellierungsproblematik entschärft wird. Die Berechnung von Testfällen komplexerer Modelle ist hier ein weit größeres Problem, das durch den Einsatz von Mechanismen zur Reduktion des Suchraums gelöst werden kann. Eine Möglichkeit besteht in der Verwendung eines Testzwecks (engl. test purpose), der angibt, welche Teile des Testmodells für einen Testfall von Interesse sind. Die Generierung von Testfällen wird dann auf das reduzierte Testmodell angewendet. Die Testfallerzeugung unter Angabe von Testzwecken wird beispielsweise von TGV unterstützt.

Von Softnet Austria wurde gezeigt, dass der Einsatz des modellbasierten Testens in der Praxis aufgrund der verfügbaren Modellierungssprachen und Werkzeuge bereits machbar ist und auch zu Verbesserungen im Softwaretest führt. Als Teil des Projekts zur Überführung des modellbasierten Testens in die Praxis sollte eine kommerzielle Implementierung des Voice Over IP-Protokolls SIP (Session Initiation Protocol) einer Partnerfirma getestet werden. Zur Verfügung standen die Definition von SIP [vgl. Ros02] und die kommerzielle SIP-Implementierung sowie eine Open Source-SIP-Implementierung. Die SIP-Implementierungen lagen bereits in getesteter Form vor.

Die Modellierung des SIP-Registrars, der für die Lokalisierung von SIP-Benutzern zuständig ist, wurde mit Hilfe von LOTOS durchgeführt. Im Anschluss wurde TGV zur Erzeugung von Testfällen verwendet. Dabei stellte sich heraus, dass die Definition von geeigneten Testzwecken komplex ist und eine Hürde bei der Verwendung modellbasierter Testtechniken darstellen kann. Aus diesem Grund wurde von Weiglhofer und Kollegen [vgl. Wei09] vorgeschlagen, Testzwecke automatisch aus Abdeckungsbedingungen (engl. coverage) abzuleiten. Die Idee hierbei war, Abdeckungen für LOTOS-Modelle zu definieren, die sich aus Programmabdeckungen, wie zum Beispiel Befehls- oder Pfadabdeckungen, ableiten lassen. Im Speziellen wurden folgende Abdeckungen für LOTOS-Modelle definiert: Prozessabdeckung (process coverage), Aktionsabdeckung (action coverage), Bedingungsabdeckung (condition coverage), Entscheidungsabdeckung (decision coverage), Bedingungs-/Entscheidungsabdeckung (C/D coverage) und die modifizierte Bedingungs-/Entscheidungsabdeckung (MC/DC coverage).

Die Testfallgenerierung mit Hilfe von TGV unter Verwendung von Abdeckungskriterien hat sehr gute Ergebnisse geliefert, die im Detail in [Wei09] nachzulesen sind. Unter anderem wurden Testfälle generiert, die es erlauben, bis zu drei zusätzliche Fehler in den SIP-Registrar-Implementierungen zu finden. Dabei hat sich herausgestellt, dass die korrespondierenden Testfälle der Prozessabdeckung weniger dazu beitragen, Fehler zu finden als die Testfälle der komplexeren Abdeckungsmetriken. Dies entspricht im Wesentlichen den Erwartungen. Die Testfallgenerierung dauerte insgesamt über zwei Tage und wurde unter Verwendung eines Athlon-64 X2 Dual Core-Prozessors 4200+ und 2 GB RAM durchgeführt. Eine interessante Beobachtung war, dass manuell konstruierte Testzwecke zu Testfällen führten, die nur einen Fehler finden konnten. Obwohl beide SIP-Registrar-Implementierungen bereits ausführlich getestet wurden, hat modellbasiertes Testen das Potenzial zusätzliche Fehler zu finden.

Die Beobachtung, dass modellbasiertes Testen Fehler in bereits getesteter Software findet, wird auch von anderen Arbeiten bestätigt. In der Diplomarbeit von Herrn Haiden [vgl. Hai11], die sich mit dem Einsatz des Microsoft-Werkzeugs Spec Explorer in der industriellen Software-Entwicklung beschäftigt, wird auf Seite 78 angeführt, dass mit Hilfe der von Spec Explorer gelieferten Testfälle Fehler in ausführlich getesteten Programmteilen gefunden wurden. Für die Programmteile standen bereits über 100 manuell erstellte Testfälle zur Verfügung. Ebenso wurden die Programmteile bereits in einer Testanlage mehrere Wochen verwendet. Trotzdem konnte der modellbasierte Test sieben Fehler finden, wobei ein Fehler als kritisch und ein weiterer Fehler als schwer eingestuft wurde.

Zusammenfassend kann festgestellt werden, dass der modellbasierte Test das Potenzial hat Fehler zu finden, die im Fall von manuell geschriebenen Testfällen nicht gefunden werden. Darüber hinaus gibt es bereits eine Reihe von Werkzeugen zur Erstellung von Testmodellen und zur automatisierten Testfallgenerierung, die praxistauglich sind. Die Erzeugung von Testfällen basierend auf Abdeckungsmetriken der Modelle ist möglich und liefert bessere Resultate als die Verwendung von manuellen Testzwecken zur Einschränkung der Suche bei der Testfallgenerierung.

Smart Monkeys

Ein anderes Projekt beschäftigt sich mit der Generierung von randomisierten Testsequenzen für Programme mit grafischen Benutzeroberflächen – oder genauer gesagt mit Smart Monkey Testing, einer Unterart des Fuzz Testing [vgl. Sut07].

Unter einem Monkey-Test verstehen wir einen randomisierten Test einer Applikation über die (grafische) Benutzerschnittstelle. Generell werden zwei Arten

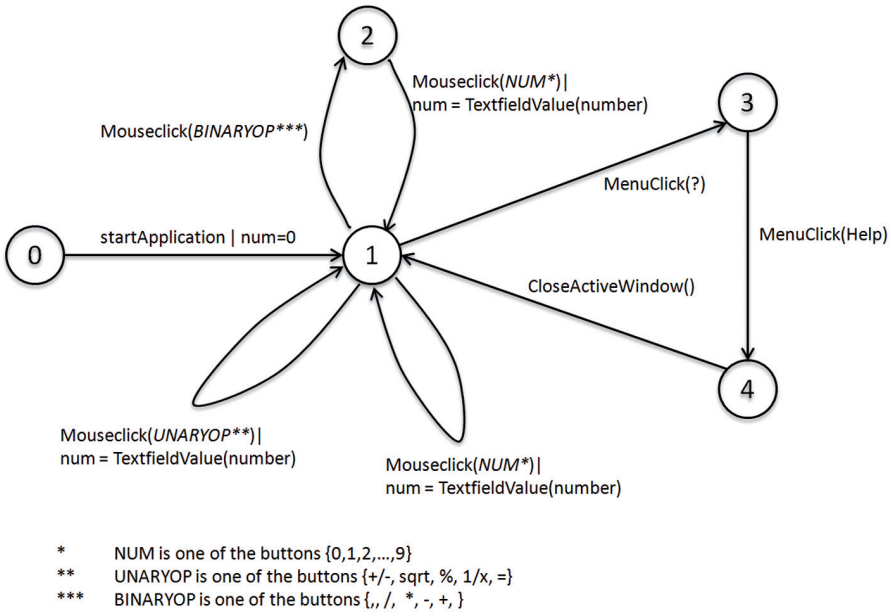

Abb. 2: Vereinfachtes Modell des Windows-Taschenrechners.

von Test-Monkeys unterschieden: Dumb Monkeys und Smart Monkeys [vgl. Nym00]. Dumb Monkeys verfügen über kein applikationsspezifisches Wissen, sondern lediglich über generelles Wissen, wie über die Benutzerschnittstelle mit dem SUT interagiert werden kann. Dumb Monkeys können daher nur primitive Fehler wie Programm- oder Systemabstürze entdecken. Smart Monkeys hingegen verfügen über ein vereinfachtes Modell des SUT. Dadurch können sie auch andere Fehlerarten, wie beispielsweise falsche Berechnungen, erkennen.

Ein Modell für einen Smart Monkey besteht aus Zuständen, in denen bestimmte Eigenschaften erfüllt sein müssen, sowie Zustandsübergängen, in denen der Monkey eine bestimmte Benutzeraktion über die Benutzerschnittstelle des SUT ausführt. Nach jeder Aktion überprüft der Monkey, ob der nach außen sichtbare Zustand der Applikation mit den im Modell definierten Eigenschaften übereinstimmt. Als Eigenschaft können beispielsweise Werte eines Textfelds oder das momentan aktive Fenster festgelegt werden.

Abbildung 2 zeigt ein vereinfachtes Modell des Windows-Taschenrechners [vgl. Hof09]. Mit Hilfe dieses vereinfachten Modells war es möglich, einen Fehler im Taschenrechner in den Versionen Windows XP und Windows Vista zu entdecken. Der Fehler wird offensichtlich, wenn der Benutzer eine Division durch Null durchführt (beispielsweise 5/0) und danach das Hilfe-Menü (Hilfe > Hilfe anzeigen) öffnet. Dadurch wird als Ergebnis im Textfeld statt der Fehlermeldung „Division durch 0 nicht möglich“ eine Zahl angezeigt. Dieser Fehler wurde mit traditionellen, strategischen Testmethoden nicht entdeckt.

Die Vorteile eines Smart Monkey-Tests sind vielfältig:

- Während in einem strukturierten Test – sofern Kombinationen von Benutzeraktionen überhaupt getestet werden – immer wieder dieselben Kombinationen getestet werden, generiert ein Smart Monkey für jeden Testlauf neue Kombinationen von Benutzeraktionen. Dadurch können Fehler gefunden werden, die erst durch das Zusammenspiel bestimmter Module entstehen.
- Ein Smart Monkey benötigt kein detailliertes Modell des SUT. Ein vereinfachtes Modell für einen Monkey-Test kann semi-automatisch aus der grafischen Benutzeroberfläche des SUT generiert werden. Dadurch verringert sich der Modellierungsaufwand enorm. Evaluierungen haben gezeigt, dass bereits mit einem solchen vereinfachten Modell Fehler in Programmen gefunden werden können, wie das Beispiel des Windows-Taschenrechners zeigt.
- Oft wird aus Zeitmangel oder Budgetgründen nur ein eingeschränkter Systemtest durchgeführt. Ein Monkey-Test kann als kostengünstiger End-to-End-Test, der sowohl die Funktionalität als auch die grafische Benutzeroberfläche testet, verwendet werden. Eine Fallstudie am FTP-Programm FileZilla zeigt, dass annähernd 40 % der Funktionen und 18 % der Quellcodebedingungen getestet werden, wenn der Monkey eine Stunde lang zufällig Benutzeraktionen auswählt.

Monkey Testing stellt eine leichtgewichtige Variante des modellbasierten Testens dar. Während andere Varianten des modellbasierten Testens ein exaktes Modell des SUT benötigen, kann der Grad der Genauigkeit des Modells bei Monkey-Tests frei gewählt werden. Im Extremfall (Dumb Monkeys) wird nur überprüft, ob das SUT in der Lage ist, beliebige Eingaben ohne Systemabsturz zu verarbeiten. Des Weiteren wählt ein Test-Monkey die Benutzeraktionen zufällig (aus den möglichen Aktionen) aus, d. h., es wird keine Teststrategie, wie beispielsweise Code Coverage, verfolgt. Dadurch kann nicht garantiert werden, dass alle Teile des SUT getestet werden. Das zufällige Aneinanderreihen von Benutzeraktionen kann jedoch Fehler aufzeigen, die in systematischen Tests unentdeckt geblieben wären. Monkey Testing ist somit kein Ersatz für strategische Testmethoden, jedoch eine wertvolle Ergänzung.

Offene Problemstellungen

Auch wenn der Einsatz modellbasierter Testmethoden weiter steigt, gibt es noch eine Reihe von Problemstellungen, die nicht nur von wissenschaftlichem Interesse sind, sondern auch Bedeutung für die Praxis haben. Einen Problembereich stellt die Testfallgenerierung dar. Bei einer möglichst effizienten Berechnung der Testfälle darf auch der Zweck des Testens nicht aus den Augen verloren werden. Neben der Erkennung möglichst vieler unterschiedlicher Fehler ist es auch von Interesse, die Ursache der Fehler effizient zu finden. Eine Testfallerzeugung, die auch die Fehlerlokalisierung unterstützt und es erlaubt, unterschiedliche Ursachen zu unterscheiden, kann die Effizienz der Gesamtentwicklung von Software steigern.

Ein anderer Problembereich ist die Bereitstellung von Testmodellen für nicht funktionale Tests. Ein Anwendungsfall ist die Informationssicherheit, wo es um

Vertraulichkeit, Verfügbarkeit und Integrität von Informationen geht. In dieser Anwendungsdomäne kann der modellbasierte Test helfen, zumindest bekannte Fehler und Systemschwächen zu vermeiden, da diese durch die aus speziellen Testmodellen erzeugten Testfälle abgedeckt werden. Diese Idee wird derzeit unter anderem im ITEA2-Projekt DIAMONDS (siehe http://http://www.itea2-diamonds.org) mit dem Fokus auf Anwendbarkeit im industriellen Umfeld vorangetrieben.

Fazit und Dank

Der modellbasierte Test ist ein wichtiger Bestandteil des gesamten Testprozesses. Er ergänzt bestehende Testverfahren. Die in diesem Artikel zusammengefassten Fallstudien zeigen, dass der modellbasierte Test Fehler findet, die durch manuelle Tests nicht aufgedeckt wurden. Die Erzeugung von Tests kann durch die Auswahl von Modellabdeckungen wie MC/DC Coverage weiter automatisiert werden und ersetzt die manuelle Bereitstellung von Testzwecken zur effizienten Berechnung der Testfälle. Eine Kombination von Modellen mit einer zufälligen Auswahl von Aktionsfolgen kann effizient für Systemtests basierend auf grafischen Benutzerschnittstellen angewendet werden. Im Gegensatz zu randomisierten Ansätzen ohne Modelle können auch Fehler gefunden werden, die nicht zu Systemabstürzen oder dergleichen führen.

Die den beschriebenen Arbeiten zugrunde liegende Forschung wurde von Softnet Austria durchgeführt. Softnet Austria wird vom Österreichischen Bundesministerium für Wirtschaft, Familie und Jugend (BMWFJ), der Österreichischen Forschungsförderungsgesellschaft (FFG), der Steirischen Wirtschaftsforschungsförderungsgesellschaft (SFG) und der Technologieagentur der Stadt Wien (ZIT) finanziell gefördert. Wir möchten uns an dieser Stelle auch bei Gordon Fraser, Bernhard Peischl und Martin Weiglhofer für deren Beiträge bei der Durchführung der Fallstudien bedanken.

Referenzen

[Bak08] Baker, P.; Dai, Z. R.; Grabowski, J.; Haugen, Ø.; Schieferdecker, I.; Williams, C.: Model-Driven Testing – Using the UML Testing Profile. Springer-Verlag, Berlin, 2008, ISBN 3-642-09159-8

[Gül10] B. Güldali, S. Jungmayr, M. Mlynarski, S. Neumann, M. Winter, Starthilfe für modellbasiertes Testen: Entscheidungsunterstützung für Projekt- und Testmanager in OBJEKTspektrum 03/2010, 2010

[Hai11] Philipp Haiden, Qualitätssicherung in verteilten Systemen – Testautomatisierung mittels modellbasierten Test, Diplomarbeit, Technische Universität Graz, 2011

[Hof09] Hofer, B.; Peischl, B.; Wotawa, F: GUI savvy end-to-end testing with smart monkeys. ICSE Workshop on Automation of Software Test (AST '09), pp. 130-137, 2009

[ISO8807] ISO, 8807: information processing systems – open systems interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour, 1989

[Jard05] C. Jard, T. Jéron, TGV: theory, principles and algorithms. International Journal on Software Tools for Technology Transfer, 7(4): 297-315, 2005

[Nym00] Nyman, N.: Using monkey test tools. Software Testing & Quality Engineering Magazine, January/February 2000

[Ros02] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, E. Schooler, SIP: Session Initiation Protocol, RFC 3261, IETF, 2002. Siehe http://www.jdrosen.net/papers/rfc3261.txt.

[Sch07] Schieferdecker, I.: Modellbasiertes Testen. OBJEKTspektrum 3/2007, S. 39-45, 2007

[Sut07] Michael Sutton, Adam Greene and Pedram Amini (2007). Fuzzing: Brute Force Vulnerability Discovery. Addison-Wesley. ISBN 0-321-44611-9

[Vea08] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson, Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer, in Formal Methods and Testing, vol. 4949, pp. 39-76, Springer Verlag, 2008

[Wei09] M. Weiglhofer, G. Fraser, F. Wotawa, Using coverage to automate and improve test purpose based testing, Information and Software Technology 51:1601-1617, 2009