T U M

I N S T I T U T F Ü R I N F O R M A T I K

Szenarien modellbasierten Testens

A. Pretschner, J. Philipps

TUM-I0205
Juli 02

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-07-I0205-0/0.-FI

Druck: Institut für Informatik der
Technischen Universität München

Szenarien modellbasierten Testens

Alexander Pretschner und Jan Philipps
Institut für Informatik, Technische Universität München
www4.in.tum.de/~{pretschn,philipps}

Zusammenfassung

Modellbasiertes Testen ist eine Technik zur Überprüfung des Verhaltens eines Systems, die auf dem Einsatz von Modellen, d.h. Abstraktionen, des Systems beruht. Verschiedene Einsatzmöglichkeiten von Modellen werden untersucht. Darunter fallen die Reihenfolge der Entwicklung von Modell und System (Modell-Code, Code-Modell oder unabhängige Entwicklung), die Rolle von Modellen bei der Testfallgenerierung sowie als Orakel bei der Testdurchführung und schließlich der Einsatz von Modellen bei Systementwicklung, -verifikation und -validierung.

1 Einleitung

Testen dient der Überprüfung der Übereinstimmung des Verhaltens eines Artefakts mit seinem erwünschten Verhalten. Ist das gewünschte Verhalten – grob: die funktionalen Anforderungen – in maschinell (potentiell) operationalisierbarer Form codifiziert, so spricht man von *Verifikation*: Die Übereinstimmung zweier formaler Dokumente in bezug auf das beschriebene Verhalten wird überprüft. Ist das gewünschte Verhalten hingegen nicht operationalisierbar, da es nur gedanklich gegeben oder in Form natürlichsprachiger Texte beschrieben ist, so spricht man von *Validierung*. Nach Definition kann eine Validierung nicht automatisch erfolgen. Abgesehen von der Unentscheidbarkeit des allgemeinen Äquivalenzproblems, ist das für Verifikation hingegen zumindest konzeptionell möglich. Der Unterschied besteht in der Existenz einer maschinellen Instanz (Orakel), die die Äquivalenz – zumindest konzeptionell – entscheiden kann. Für die Tätigkeit der Validierung ist diese Instanz stets eine menschliche Intelligenz.

Da das Äquivalenzproblem i.a. nicht oder aufgrund von Komplexitätsproblemen nicht effizient entscheidbar ist, stellt sich die Frage nach einer adäquaten Definition des eingangs erwähnten Terminus der *Übereinstimmung*. Für die Validierung ist ein solcher im Normalfall nicht präzise faßbar. Im Kontext der Verifikation wäre ein Beobachtungsbegriff wünschenswert, der zu einer entscheidbaren Variante der Äquivalenz führte, der aber zugleich das Vertrauen in die allgemeine Übereinstimmung zweier Verhalten

erhöht. Im allgemeinen ist ein solcher Beobachtungsbegriff wahrscheinlich nicht zu ermitteln. Ansätze umfassen u.a. endliche Repräsentationen von Verhaltensspezifikationen (Chow, 1978) oder strukturelle Überdeckungskriterien. Wissenschaftliche und dokumentierte empirische Evidenz für letztere ist im wesentlichen nicht erbracht; erstere ist nicht skalierbar. Tatsächlich ist es sogar so, daß Überdeckungskriterien in bezug auf ihre Fähigkeit, Fehler aufzudecken, nur dann effektiver als randomisiertes Testen sind, wenn die zugrundeliegenden Partitionierung des Eingabedatenraums a priori höhere Fehlerwahrscheinlichkeiten für manche der Partitionen bedingt (Duran und Ntafos, 1984; Hamlet und Taylor, 1990).

Obwohl es konzeptionell zunächst durchaus unbefriedigend ist, wird deshalb im folgenden die Übereinstimmung des Verhaltens zweier Artefakte in bezug auf eine existierende Menge von Eingabe-Ausgabepaaren verstanden. Die Charakterisierung einer solchen Menge wird als gegeben vorausgesetzt.

Das formale Dokument, das das gewünschte Verhalten codifiziert, wird i.a. als *Spezifikation* bezeichnet. Unter dem zweiten formalen Dokument, dem tatsächlichen System, wird i.a. eine *Implementierung* verstanden. Testen dient somit der Überprüfung der Übereinstimmung, der *Conformance*, von Spezifikation und Implementierung.

Spezifikationen sind i.a. weniger präzise als eine entsprechende Implementierung. Selbst wenn sie ausführbar sind, stellen sie somit *Abstraktionen* dar. Im folgenden werden solche ausführbaren Spezifikationen als *Modelle* bezeichnet. Allgemeine Aspekte der Problematik der Erstellung von Modellen werden von Schätz et al. (2002) im Kontext der modellbasierten Entwicklung diskutiert. Der Grund für die Wahl dieses Bezeichners liegt darin, daß Spezifikationen i.a. vor einer Implementierung entstehen. Zum Zweck des Testens können Modelle als Abstraktionen aber durchaus auch nach oder gleichzeitig mit einer Implementierung entstehen. Denkbar sind drei Szenarien: Code wird vor dem Modell erstellt, Code wird nach dem Modell erstellt, und Code und Modell werden unabhängig voneinander – im Idealfall sogar gleichzeitig – entwickelt.

Jede Form des Testens erfordert Redundanz in den Artefakten, die bei der Systementwicklung entstanden sind; ohne Redundanz gibt es keine Vergleichsmöglichkeit. Die Kernfrage des modellbasierten Testens ist somit, ob es vom Aufwand her günstiger ist, Modelle zu Generierung von Testsequenzen einzusetzen, oder Testsequenzen direkt zu erstellen, wie dies etwa im Extreme Programming (Beck, 1999) propagiert und auch von entsprechenden Bibliotheken (z.B. JUnit) unterstützt wird. Ohne empirische Untersuchungen läßt sich diese Frage nicht klären; Ziel dieses Beitrags ist es, die Konzepte modellbasierten Testens vorzustellen, um einen Grundstein für solche Untersuchungen zu legen.

Überblick. Dieser Beitrag diskutiert die drei Szenarien (Abschnitt 2). Die Diskussion beleuchtet auch Konsequenzen und Anforderungen für Entwicklungsprozeß und Werkzeugunterstützung. Da die schwierigste Frage beim Testen die ist, was getestet werden soll, wird in Abschnitt 3 die automatisierte Generierung von Testfällen und Testfallspezifikationen untersucht und in die drei Szenarien eingebettet. Abschnitt 4 beendet den Aufsatz mit einer Zusammenfassung.

Terminologie. Ein *Testziel* ist eine informelle Schilderung derjenigen Eigenschaft, die getestet werden soll. Es kann funktional oder strukturell sein. Eine *Testfallspezifikation* ist die Formalisierung eines Testziels. Ein *Testfall* ist eine Repräsentation einer Menge von Ein-/Ausgabepaaren, wobei der Ausgabeteil die erwartete Ausgabe darstellt. Eine *Testsequenz* ist eine voll instantiierte Instanz eines Testfalls. Im Kontext der Validierung ist ein *Orakel* ein Algorithmus oder eine menschliche Intelligenz, die entscheidet, ob die Ausgabe eines Systems sich mit der erwarteten, sich aus der Anforderung ergebenden Ausgabe deckt. Lötzbeyer und Pretschner (2000) geben eine Präzisierung dieser Definitionen.

2 Szenarien

Modelle als (ausführbare) Repräsentanten einer Spezifikation können zur Qualitätssicherung zweierlei Funktionen erfüllen. Zum einen können sie der manuellen oder automatisierten Testfallgenerierung dienen. Zum anderen können sie als Orakel dienen, wenn für eine Menge von Stimuli die entsprechende Menge von Ausgaben bestimmt werden muß. In diesem Abschnitt wird diskutiert, wie die Entwicklung von Modellen und Code miteinander verwoben werden kann; dabei wird besonderes Augenmerk auf Modelle als Grundlage für eine automatisierte Testfallgenerierung gelegt.

2.1 Modell entsteht vor generiertem Code

Modelle werden in diesem Zusammenhang als ausführbare Abbilder der Anforderungen bzw. der Spezifikation gesehen (Abbildung 1). Wenn diese Modelle als Grundlage für eine manuelle Codierung dienen, kann im wesentlichen von einer unabhängigen Entwicklung gesprochen werden; dieser Fall wird in Abschnitt 2.3 behandelt.

Wenn Modelle allerdings so präzise sind, daß sie sich als Grundlage der Überprüfung von Verhaltensübereinstimmungen einsetzen lassen, dann stellt sich aus ökonomischer Sicht die Frage, ob der Code nicht aus dem Modell generiert werden sollte. Das ist nicht zwingend für alle Aspekte eines Systems der Fall; denkbar ist beispielsweise ein Modell eines verteilten Systems, das von der Kommunikationsstruktur abstrahiert und somit zur Testfallgenerierung, nicht aber direkt zur Generierung des Systemcodes geeignet ist. Für

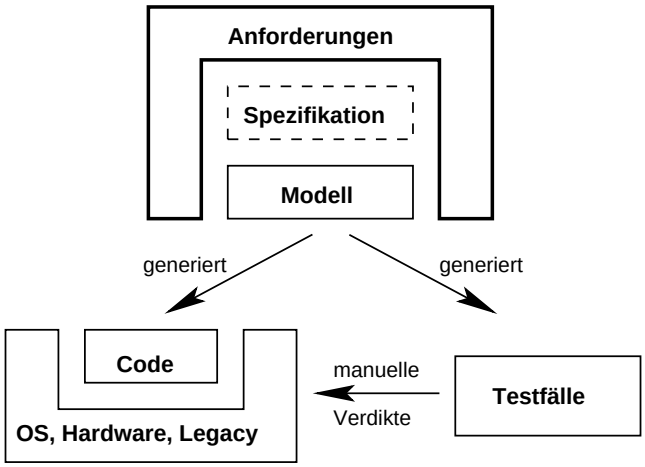

Abbildung 1: Code und Testfälle aus Modell generiert

beispielsweise isoliert betrachtete, gekapselte Steuergeräte hingegen ist das vorstellbar. Der Vorteil einer solchen *modellbasierten Entwicklung* liegt in dem höheren Abstraktionsniveau von Modellierungssprachen (Schätz et al., 2002) gegenüber den üblichen Programmiersprachen. Modelle, aus denen Produktionscode generiert werden kann, können als Programme in einer abstrakten domänenspezifischen Programmiersprache angesehen werden.

Codegeneratoren für Matlab-Modelle wie Beacon (ADI) oder TargetLink (dSpace) oder im Werkzeug ASCET-SD (ETAS) spiegeln diesen bereits heute partiell erfolgreichen Ansatz wieder. Schwierigkeiten ergeben sich im Bereich eingebetteter Systeme insbesondere aus der Integration verschiedener Abstraktionsebenen (Zusammenführen von Funktionalität auf Modellebene mit Hardware, Legacy-Systemen und Echtzeitbetriebssystemen; Deployment), die bis jetzt noch nicht befriedigend gelöst sind.

Wenn die Modelle zur Testfallgenerierung herangezogen werden, dann können sie nicht gleichzeitig als Orakel fungieren: Der generierte Code würde gewissermaßen gegen sich selbst getestet werden.

Die Verwendung von Modellen für die Testfallgenerierung in diesem Szenario ist trotzdem nicht ganz unsinnig. Zum einen ist es u.U. einfacher, für bereits existierende Stimuli die erwarteten Ausgaben manuell zu verifizieren, als diese Stimuli selbst zu finden. Vielversprechend ist in diesem Szenario also die Verwendung von Modellen zur Testfallgenerierung im Sinne der *Validierung* von Modellen. Durch den höheren Abstraktionsgrad von Modellen und die eingängigen Beschreibungstechniken erfordern Reviews eines Modells weniger Einarbeitung als in Code; Fragen zur Systemfunktionalität lassen sich (auch zusammen mit dem Auftraggeber) effizienter klären.

Zum anderen können die Testfälle zur Überprüfung von Umweltannah-

men und der Korrektheit von Codegeneratoren oder Übersetzern verwendet werden. Insbesondere im Kontext kontinuierlicher Systeme, in denen ein formaler Nachweis der Übereinstimmung von Simulations- und Produktionscodegeneratoren wegen auftretender numerischer Probleme i.a. sehr schwierig ist, wird der Sinn eines solchen Ansatzes deutlich. Hier ist das Problem, daß die Semantik der Codegeneratoren nicht exakt festgelegt oder festlegbar ist und zwei Generate (Produktions- und Simulationscode) u.U. sogar mit Werkzeugen zweier verschiedener Hersteller erstellt werden, wie das Beispiel der Erzeugung von Produktionscode aus Matlab-Modellen mit TargetLink oder dSpace illustriert. Ein weiteres Problem sind Umweltannahmen, die beispielsweise in Form einer zeitkritischen Einbettung in den Hardware- und Betriebssystemkontext Einfluß auf die korrekte Funktion des generierten Codes ausüben.

Zusammenfassung. Wenn Modelle so präzise sind, daß sie der automatisierten Generierung adäquater Testsuiten dienen können, muß aus ökonomischer Sicht auch ihr Einsatz zur Codegenerierung in Betracht gezogen werden. Vorteile dieses Ansatzes sind die fortgeschrittenen Techniken modellbasierter Entwicklung, die eine schnellere Entwicklung weniger fehlerhafter Produkte verheißt. Wenn aus den Modellen gleichzeitig Testfälle generiert werden sollen, können diese nicht als Orakel für funktionale Tests dienen. Allerdings ist es u.U. einfacher, erwartete Ausgaben von Testfällen manuell zu überprüfen, als die Testsuite vollständig manuell zu entwickeln. Außerdem können Codegeneratoren und Umweltannahmen überprüft werden.

2.2 Code ensteht vor Modell

Wenn die Wahl auf eine codezentrierte Entwicklung fällt, dann ist der Code oft zu kompliziert, als daß er direkt zur Verifikation herangezogen werden könnte. Im Bereich des Model Checking gibt es Ansätze, halbautomatisch Modelle aus Code zu erzeugen (Holzmann, 2001; Visser et al., 2000; Graf und Saïdi, 1997), für die dann Eigenschaften überprüft werden (Abbildung 2). Rückschlüsse auf den Code sollten natürlich möglich sein, die Abstraktion konservativ (z.B. Dams et al. (1997)). Testfälle sind u.U. zu konkretisieren (als inverse Abbildung der Codeabstraktionsabbildung). Wenn solche Modelle zur Testfallgenerierung herangezogen werden, können sie ebenfalls nicht direkt als Orakel Verwendung finden: der Code würde gegen sich selbst getestet. Eine manuelle Überprüfung der Ausgaben der Testfälle ist natürlich denkbar.

Da die Extraktion von Modellen aus Code eine nur teilweise zu automatisierende Tätigkeit darstellt (Abstraktionen existieren immer nur in bezug auf einen bestimmten Zweck), ist es wegen häufiger Änderungen nicht immer einfach, während der Entwicklung des Codes auch die Modelle zu extrahieren. Wenn der Code erst nach Fertigstellung abstrahiert wird, hat das den

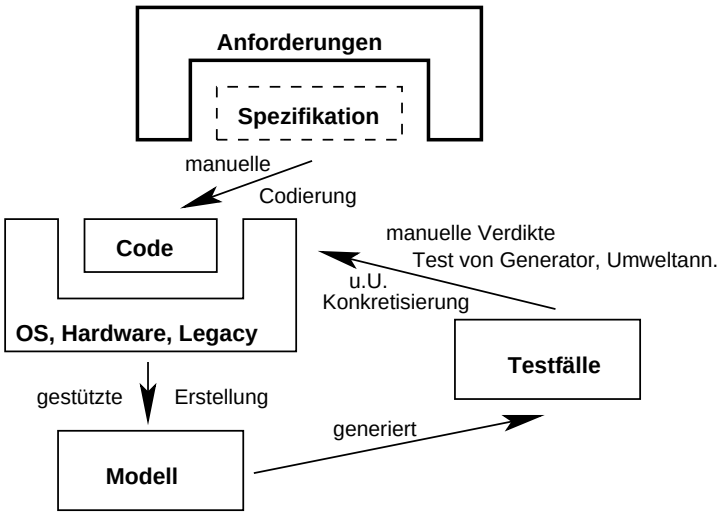

Abbildung 2: Modell entsteht nach Code

Nachteil, daß Testfälle ebenfalls erst nach der Codierung definiert werden. Wenn die aus dem abstrahierten Code erstellten Testfälle allerdings komplementär zu bereits erstellten Testfällen verwendet werden, ist dieser Nachteil trivialerweise behoben.

Dieses Szenario ist relevant auch im Zusammenhang mit der Validierung neuartiger Testfallgenerierungsverfahren. Dabei werden Testfälle für ein existierendes System erzeugt und mit einer existierenden Testsuite verglichen.

Zusammenfassung. Aus bestehendem Code können teilautomatisiert Abstraktionen errechnet werden, die nicht nur einer erschöpfenden Analyse durch Model Checking, sondern auch der Testfallgenerierung dienen können. Der Vorgang der Abstraktion ist aufwendig, was insbesondere bei sich ändernden Modulen Probleme bereitet. Vorteile einer modellbasierten Entwicklung kommen in einem solchen codezentrierten Ansatz nicht zur Geltung.

2.3 Code und Modell entstehen gleichzeitig

Wenn Codegeneratoren für eine bestimmte Zielsprache nicht existieren oder wenn Entwicklungs- und Qualitätssicherungsabteilung aus organisatorischen Gründen getrennt werden, dann können System und Modell unabhängig voneinander erstellt werden (Abbildung 3). Modelle der QS-Abteilung werden zum Test des Codes der Entwicklungsabteilung verwendet. Da Testen von Beginn an erfolgen sollte, ergibt sich der Koordinationsaufwand zwischen verschiedenen Modell- und Codeversionen als Schwierigkeit. Eine solche redundante Entwicklung ist außerdem teuer. Der große Vorteil dieses

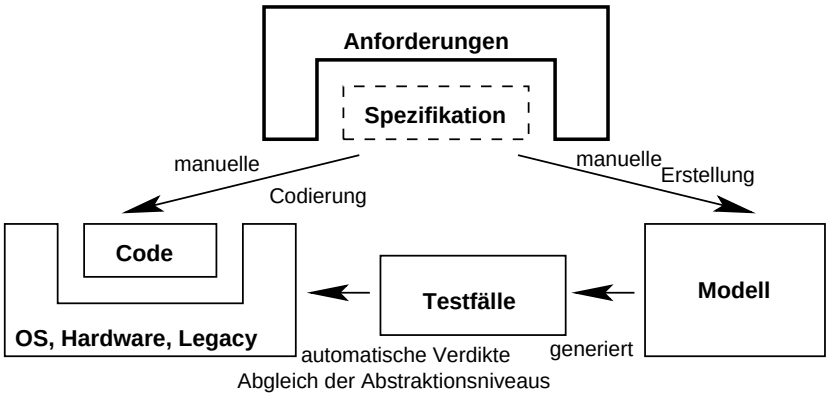

Abbildung 3: Modell und Code entstehen unabhängig

Szenarios liegt – bei Übereinstimmung der Schnittstellen und Abgleich der Abstraktionsniveaus – in der Möglichkeit, Modelle automatisch als Orakel zu verwenden. Wenn es Diskrepanzen zwischen Modell und Code gibt, muß geprüft werden, ob das Modell oder der Code fehlerhaft ist.

Dieses Szenario macht zunächst keine Aussage darüber, ob sich die Modellentwicklung auf Modul- oder auf Systemebene bewegt. Wenn die Spezifikation eines Moduls auszuprogrammieren ist und die Spezifikation nicht ausführbar ist, kann ein Modell erstellt werden, das der Erzeugung von Testfällen dient. Im Extremfall ist es denkbar, daß der Programmierer selbst ein zur Testfallgenerierung oder -durchführung geeignetes Modell erstellt. Problematisch dabei ist natürlich, daß in diesem Fall Mißverständnisse höchstwahrscheinlich nicht nur im Modell, sondern auch im Code anzutreffen sind, was bei getrennter Entwicklung weniger wahrscheinlich ist.

Als weiteres Szenario ist denkbar, daß zwei Modelle unabhängig erstellt werden, eins für die Entwicklung und eins für die Testfallgenerierung (Abbildung 4). Dieses Szenario entspricht im wesentlichen dem vorherigen mit dem Unterschied, daß hier auch die Erstellung des Systems von den Vorteilen einer modellbasierten Entwicklung profitieren kann.

Zusammenfassung. Wenn Modell und Code unabhängig voneinander entstehen, kann das Modell zur automatischen Verdiktbildung verwendet werden. Das setzt aber einen Abgleich der Abstraktionsniveaus voraus. Die redundante Entwicklung ist teuer, bietet aber wie bei jeder Entwicklungsredundanz den Vorteil, daß Mißverständnisse in den Anforderungen früh aufgedeckt werden.

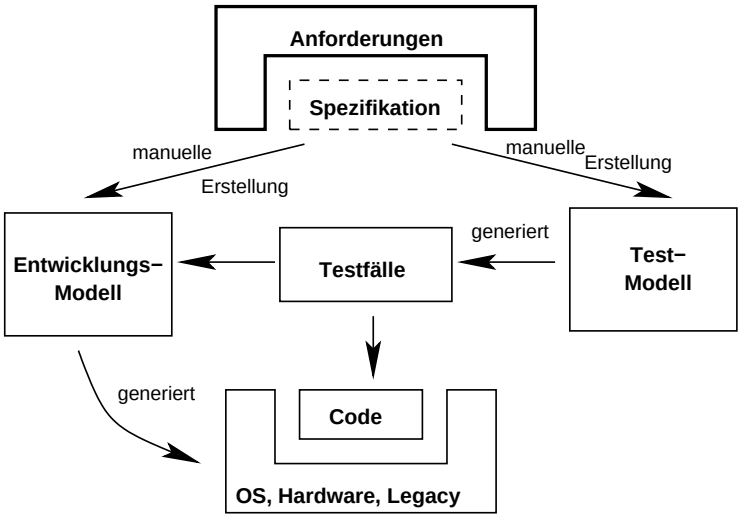

Abbildung 4: Modelle für Entwicklung und Testfallgenerierung

3 Testfallspezifikationen und Testfälle

Das Hauptproblem beim Testen ist die Frage, was gute Testfälle sind (Weyuker, 1986; Parrish und Zweben, 1991; Goodenough und Gerhart, 1975). Die Definition solcher Qualitätskriterien kann sowohl als Auswahlkriterium für die Generierung von Testfällen als auch als Testendekriterium interpretiert werden (Zhu et al., 1997). Eine verwandte Problematik tritt im Bereich des Model Checking oder deduktiven Beweisens auf, in dem die Frage, welche Eigenschaft nachgewiesen oder falsifiziert werden soll („Putativtheoreme“ im Sinne von Rushby (1995)), ebenfalls nicht immer einfach zu beantworten ist. Prinzipiell können Testfälle dem Aufdecken von Fehlern dienen (die eher akademische Sicht) oder der Abdeckung von Anforderungen (die eher praktische Sicht, in der momentan häufig erst am Ende der Entwicklung unter massivem Zeitdruck getestet wird). Im Zweifelsfall sollte natürlich die Fähigkeit, Fehler zu finden, die Abdeckung von Anforderungen einschließen.

Im folgenden wird nun zwischen der Generierung von Testfällen aus einer gegebenen Testfallspezifikation und der Generierung von Testfallspezifikationen unterschieden (Abbildung 5). Die Generierung von Testfallspezifikationen muß immer durch Testfallspezifikationen komplementiert werden, die sich aus der Aktivität der Anforderungsanalyse ableiten lassen.

Testfallspezifikationen. Testfallspezifikationen werden traditionell von Hand erstellt. Wenn sie dazu komplementär automatisiert erzeugt werden sollen, sind sie auf strukturelle Kriterien beschränkt. Spezielles Wissen über eine gegebene zu testende Anwendung wird also nicht herangezogen; dazu

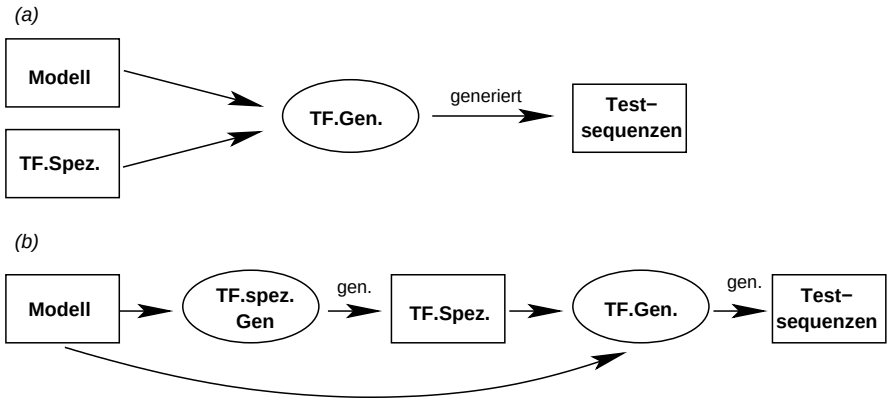

Abbildung 5: Testsequenzgenerierung ohne (a) und mit (b) automatisierter Generierung von Testfallspezifikationen

ist i.a. menschliche Intelligenz erforderlich.

- Eine *zufällige* Auswahl von Testfällen ist die einfachste Form der Testfallspezifikation (Duran und Ntafos, 1984; Hamlet und Taylor, 1990). Mit stochastischen Nutzungsmodellen wie im Cleanroom Reference Model (Prowell et al., 1999) kann die Auswahl hier noch gesteuert werden.
- Die Verwendung von *Coveragekriterien* (z.B. Ntafos (1988)) als Testfallspezifikation ist eine Möglichkeit. Vilkomir und Bowen (2001) formalisieren diverse strukturelle Coveragemaße, die als Testfallspezifikation verwendet werden könnten. Coveragekriterien sind aus der Verlegenheit entwickelt worden, daß keine besseren Mittel zu Bewertung der Güte einer Testsuite zur Verfügung stehen. Ihre Fähigkeit, Fehler zu finden, ist prinzipiell eingeschränkt (Hamlet und Taylor, 1990); empirische Evidenz (z.B. Dupuy und Leveson (2000)) liegt kaum vor.
- Unter Einschluß von Wissen über das Modell können *Testmuster* (Binder, 2001) Anwendung finden. Wünschenswert wären natürlich Sammlungen von Eigenschaften (und damit Testfallspezifikationen) für wiederkehrende Strukturen in der Entwicklung.

Testfallgenerierung. Wenn eine Testfallspezifikation vorhanden ist, müssen aus Testfallspezifikation und Modell noch Testfälle erzeugt werden. Dazu gibt es eine Reihe von Ansätzen; Pretschner et al. (2002) geben einen Überblick im Kontext reaktiver Systeme. Im wesentlichen läßt sich das Problem der Testfallgenerierung als Suchproblem sehen: Zustände oder Pfade, die eine spezielle Eigenschaft erfüllen oder nicht erfüllen, sind gemäß der Testfallspezifikation ,,interessante" Testfälle. So können Techniken des Model Checking

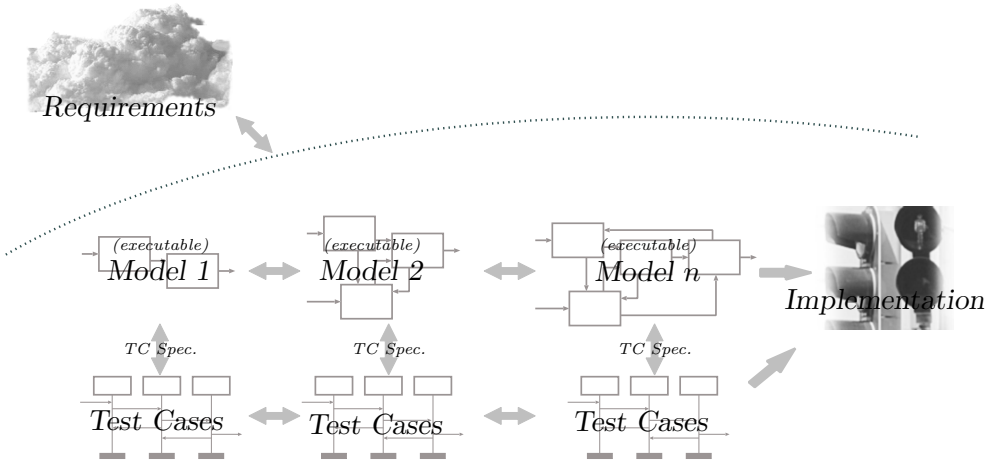

Abbildung 6: Modellieren und Testen in inkrementellem Design (Pretschner et al., 2002)

direkt zur Generierung von Testfällen verwendet werden (Ammann et al., 1998; Wimmel et al., 2000). Dedizierte Suchstrategien (Pretschner, 2001) und das Ausnutzen strukturellen Wissens um ein Modell (Bender et al., 2002) für kompositionale Testfallgenerierung sind Ansätze, das Problem der Zustandsexplosion in den Griff zu bekommen.

Ein bisher ungelöstes Problem ist die Generierung von Testfällen für pfaduniverselle Eigenschaften wie z.B. Invarianzen, weil hier nicht klar ist, nach welchem Zustand gesucht werden soll. Wenn das Modell korrekt ist, ist die negierte Invarianzeigenschaft als Testfallspezifikation von mäßigem Wert. Eine Möglichkeit besteht darin, den die Invariante verletzenden Zustand zu approximieren (Pretschner et al., 2002).

4 Zusamenfassung

In diesem Überblicksartikel wurden verschiedene Varianten des modellbasierten Testens beschrieben. Wenn Modelle so präzise sind, daß sie sich zur Ableitung von Testfällen eignen, dann werden sie im Normalfall auch für die Generierung von Produktionscode herangezogen werden. In diesem Fall bietet sich eine inkrementelle Entwicklung des Modells an, das, sobald es – unter Einbeziehung von halb- oder vollautomatisch erstellten Testsequenzen – validiert ist, zur Generierung des Codes verwendet wird (Abb. 6). Zeitgleich mit der Entwicklung des Modells können Testfälle generiert werden, die zum Regressionstest der Modellinkremente verwendet werden können. Da die Ausgaben der Testfälle in diesem Fall manuell überprüft werden müssen, ist die Testfallgenerierung hier als Debugginghilfe zu verstehen (s.

Abschnitt 2.1). Wenn das Modell als valide angesehen wird, kann daraus Code erzeugt werden. Zwei weitere Szenarien sind die Entwicklung von Modellen aus Code und die simultane sowie unabhängige Entwicklung von Modell und Code.

Obschon generell anwendbar, sind viele Aspekte in diesem Papier insbesondere im Zusammenhang mit eingebetteten Systemen von Relevanz. Wegen des i.a. bereits von der Programmiersprache und den dazugehörigen Ein-/Ausgabebibliotheken gekapselten Hardwareanteils ist der Unterschied in den Abstraktionsebenen von Modellen und Code bei Businessinformationssystemen wahrscheinlich weniger ausgeprägt. Die Umwelt ist zudem wahrscheinlich besser einschätzbar, auch wenn die Einbettung des Systems in einen Kontext von beispielsweise Datenbanksystemen und Webservern selbstverständlich getestet werden muß. Interaktions-, Kollaborations- und Zustandsdiagramme der UML werden bisweilen bereits auf Objektebene angefertigt und können dann direkt der für Testfälle verwendet werden. Allerdings spielt in solchen Systemen die Datenmodellierung bei solchen Systemen im Normallfall eine größere Rolle als bei eingebetteten Systemen, woraus zusätzliche Schwierigkeiten in bezug auf die Größe des Zustandsraums erwachsen.

Überdeckungskriterien sind üblicherweise auf Codeebene definiert. Für Modelle können sie dann entweder eigenständig definiert werden – z.B. W, UIO-Methoden oder Transitionstouren auf der Ebene beschrifteter Transitionssysteme (Ural, 1992) –, oder sie werden so definiert, daß ihre (oftmals kanonische; Ausnahmen sind z.B. die Generatoren von Esterel und Synchcharts (Berry und Gonthier, 1992; André, 1996)) Übersetzung dem Kriterium genügt. Dieser Ansatz ist z.B. für erweiterte endliche Zustandsmaschinen denkbar. In bezug auf Zertifizierung, die auf codezentrierter Überdeckung basiert, ist dieser Ansatz ebenfalls von Interesse, weil der Übergang von codebasierter zu modellbasierter Entwicklung bzgl. der Zertifizierung weniger abrupt ist als die direkte Definition von Coveragekriterien auf Modellebene.

Die Überprüfung eines Modells allein ist niemals ausreichend, weil Modelle nur Abstraktionen sind und Aspekte der Realität wie Betriebssysteme, direkte Hardwarezugriffe etc. ausblenden (Fetzer (1988) diskutiert diesen Aspekt generalisiert im Kontext der formalen Verifikation). Testen und formale Verifikation mit Model Checking oder Beweisen sind einander ergänzende Techniken. Wenn die Testfallgenerierung zur Validierung des Modells (mit einer menschlichen Instanz zur Verdiktbildung) verwendet wird, unterscheiden sich die Techniken in bezug auf ihren Wunsch nach Vollständigkeit.

Literatur

P. Ammann, P. Black, und W. Majurski. Using model checking to generate tests from specifications. In *Proc. 2nd IEEE Intl. Conf. on Formal*

Engineering Methods, pages 46–54, 1998.

C. André. Represenation and analysis of reactive behaviors: A synchronous approach. In *Proc. CESA'96*, July 1996.

K. Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley, 1999.

K. Bender, M. Broy, I. Péter, A. Pretschner, und T. Stauner. Model based development of hybrid systems: specification, simulation, test case generation. In *Modelling, Analysis and Design of Hybrid Systems*, LNCIS. Springer, 2002. To appear.

G. Berry und G. Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19:87–152, 1992.

R. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison Wesley, 2001.

T. Chow. Testing Software Design Modeled by Finite-State Machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–187, May 1978.

D. Dams, O. Grumberg, und R. Gerth. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19 (2):22–43, 1997.

A. Dupuy und N. Leveson. An Empirical Evaluation of the MC/DC Coverage Criterion on the HETE-2 Satellite Software. In *Proc. Digital Aviation Systems Conf.*, 2000.

J. Duran und S. Ntafos. An Evaluation of Random Testing. *IEEE Transactions on Software Engineering*, SE-10(4):438–444, July 1984.

J. Fetzer. Program Verification: The Very Idea. *CACM*, 37(9):1048–1063, September 1988.

J. Goodenough und S. Gerhart. Toward a Theory of Test Data Selection. *IEEE Transactions on Software Engineering*, SE-1(2):156–173, June 1975.

S. Graf und H. Saïdi. Construction of abstract state graphs with PVS. In *Proc. 9th Conf. on Computer-Aided Verification*, pages 72–83, 1997.

D. Hamlet und R. Taylor. Partition Test Does Not Inspire Confidence. *IEEE Transactions on Software Engineering*, 16(12):1402–1411, December 1990.

G. Holzmann. From Code to Models. In *Proc. 2nd Int. Conf. on Applications of Concurrency to System Design*, pages 3–10, June 2001.

H. Lötzbeyer und A. Pretschner. Testing Concurrent Reactive Systems with Constraint Logic Programming. In *Proc. 2nd workshop on Rule-Based Constraint Reasoning and Programming*, Singapore, 2000.

S. Ntafos. A Comparison of Some Structural Testing Strategies. *IEEE Transactions on Software Engineering*, 14(6):868–874, June 1988.

A. Parrish und S. Zweben. Analysis and Refinement of Software Test Data Adequacy Properties. *IEEE Transactions on Software Engineering*, 17 (6):565–581, June 1991.

A. Pretschner. Classical search strategies for test case generation with Constraint Logic Programming. In *Proc. Formal Approaches to Testing of Software*, pages 47–60, August 2001.

A. Pretschner, H. Lötzbeyer, und J. Philipps. Model Based Testing in Incremental Software Development. To appear in the Journal of Systems and Software, 2002.

S. Prowell, C. Trammell, R. Linger, und J. Poore. *Cleanroom Software Engineering*. Addison Wesley, 1999.

J. Rushby. Formal methods and their role in the certification of critical systems. Technical Report CSL-95-1, Computer Science Laboratory, SRI, 1995.

B. Schätz, A. Pretschner, F. Huber, und J. Philipps. Model-based development. Technical Report TUM-I0204, Institut für Informatik, Technische Universität München, 2002.

H. Ural. Formal methods for test sequence generation. *Computer Communications*, 15(5):311–325, June 1992.

S. Vilkomir und J. Bowen. Formalization of control-flow criteria of software testing. Technical Report SBU-CISM-01-01, South Bank University, 2001.

W. Visser, S. Park, und J. Penix. Using predicate abstraction to reduce object-oriented programs for model checking. In *Proc. 3rd Workshop on Formal Methods in Software Practice*, August 2000.

E. Weyuker. Axiomatizing Software Test Data Adequacy. *IEEE Transactions on Software Engineering*, SE-12(12):1128–1138, December 1986.

G. Wimmel, H. Lötzbeyer, A. Pretschner, und O. Slotosch. Specification Based Test Sequence Generation with Propositional Logic. *J. Software Testing, Validation, and Reliability*, 10(4):229–248, 2000.

H. Zhu, P. Hall, und J. May. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.