Automatisierte Erzeugung von TTCN-3 Testfällen aus UML-Modellen

Jens R. Calamé, Natalia Ioustinova, Jaco van de Pol
CWI, P.O. Box 94079, NL-1090 GB Amsterdam
{jens.calame, natalia.ioustinova, vdpol}@cwi.nl

Abstract: Der Test von Software ist ein notwendiges, jedoch ressourcenintensives Unterfangen. Aus diesem Grund wurden verschiedene Ansätze entwickelt, die einzelnen Aspekte des Softwaretests zu automatisieren. In diesem Paper stellen wir einen Ansatz zur automatischen Testfallerzeugung für den modellbasierten Test vor. Dabei werden aus *UML*-Modellen eines Softwaresystems und der Beschreibung von Testszenarien parametrisierbare Testfälle in *TTCN-3* erzeugt. Die Parametrisierung erhöht dabei die Wiederverwendbarkeit der Testfälle, unterstützt die gezielte Auswahl geeigneter Testdaten und wirkt so dem Phänomen der Testfallexplosion entgegen.

1 Einleitung

Softwarefehler können schwerwiegende, teils verheerende Folgen haben, weshalb Qualitätssicherung auf dem Gebiet des Software Engineering einen hohen Stellenwert genießt. Eines der am häufigsten industriell angewandten Verfahren zur Software-Qualitätsanalyse ist der Softwaretest, der dem Nachweis von Fehlern dient.

Obwohl ausgereifte Verfahren zur Erzeugung von Testfällen und der Durchführung von Tests existieren [Lig02], hat die Erfahrung gezeigt, dass das Testen von Software ein sehr ressourcenintensives Unterfangen ist. Dies ist unter anderem der Fall, da ein manueller Test Personal bindet und so hohe Kosten verursacht. Aus diesem Grund wurden in den vergangenen Jahren verschiedene Ansätze entwickelt, den Softwaretest zu automatisieren. Zum einen konzentriert man sich dabei auf die Automatisierung der Testausführung, zum anderen auf die automatische Testfallerzeugung.

In diesem Paper diskutieren wir die automatische modellbasierte Erzeugung von Testfällen für den Blackbox-Konformitätstest, bei dem das Testsystem und das zu testende System (das *System Under Test*, SUT) nur über die von außen zugänglichen Schnittstellen des SUT interagieren. Die Testfälle für den Blackbox-Test werden aus den Modellen des SUT erzeugt, die bspw. als *UML*-Spezifikation vorliegen. Sie sind parametrisierbar, was zum einen ihre Wiederverwendbarkeit erhöht, zum anderen aber auch der Testfallexplosion sowie der Explosion des zu betrachtenden Zustandsraums eines SUT entgegenwirkt.

Der Konformitätstest stellt eine Möglichkeit dar, zu validieren, ob sich eine Anwendung getreu ihrer Spezifikation verhält. Nach der $ioco$-Theorie (*Input-Output Conformance*, [Tre96]) wird eine Implementation als genau dann spezifikationskonform definiert, wenn

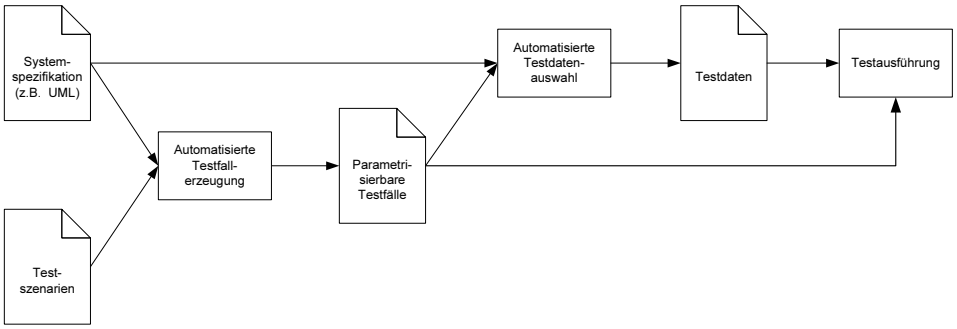

Abbildung 1: Der Testfallerzeugungsprozess

sie in allen spezifizierten Programmpfaden alle laut Spezifikation möglichen Eingaben akzeptiert und dabei höchstens die spezifizierten Ausgaben liefert. Das Testfallerzeugungs-Framework, das wir in diesem Paper vorstellen, dient der automatischen Testfallerzeugung aus *UML*-Spezifikationen für den Konformitätstest.

In Abbildung 1 haben wir die Vorgehensweise in unserem Testfallerzeugung-Framework dargestellt. Am Beginn des Generierungsprozesses stehen die Systemspezifikation in *UML* sowie eine Reihe von Testszenarien, die ebenfalls in *UML* formuliert sein können. Letztere enthalten jedoch nicht vollständige Testfälle, sondern skizzieren lediglich die Szenarien, für die später Testfälle generiert werden sollen.

Aus der Systemspezifikation und den Testszenarien werden nun parametrisierbare Testfälle generiert, wobei formale Spezifikationen als Zwischenformat verwendet werden [CIPS05, CIP06]. Bei der Parametrisierung von Testfällen werden Kontrollfluss und Daten eines Tests separat betrachtet. So ist es nicht mehr notwendig, eine große Anzahl beinahe identischer Testfälle für unterschiedliche Testdatensätze zu erzeugen. Vielmehr wird der Testfall als solcher nur einmal generiert und erst bei der Ausführung mit entsprechenden Daten instanziiert. Gleichzeitig kann durch eine getrennte Betrachtung von Kontrollfluss und Daten eines Systems die Explosion seines Zustandsraums verhindert werden.

Bei der Testfallerzeugung wird eine formale Spezifikation als Zwischenformat verwendet. Die Testfälle geben im Gegensatz zu den Testszenarien *vollständige* Programmpfade an, die später ausgeführt werden. Für diese Pfade werden nun Testdaten selektiert, wobei neben den Informationen über den Kontrollfluss des Testfalls auch alle Bedingungen auf den betreffenden Pfaden herangezogen werden. Hierfür ist wieder die Systemspezifikation nötig. Abschließend werden die Testfälle mit den Testdaten parametrisiert und ausgeführt.

Der vorgestellte Testgenerierungsprozess beginnt mit der Analyse einer Systemspezifikation in *UML* und hat als Zielsprache die Testbeschreibungssprache *TTCN-3* [GHR+03]. In Abschnitt 2 diskutieren wir, aus welchen Diagrammtypen Testfälle automatisch erzeugt werden können. Die Transformation der erzeugten Testfälle nach *TTCN-3* ist Gegenstand des Abschnitts 3, bevor wir mit Abschnitt 4 schließen.

2 *UML*-Diagrammtypen für die Testfallerzeugung

Die *Unified Modeling Language* (*UML* 2) bietet eine Vielzahl von Diagrammtypen zur Modellierung von Softwaresystemen. Nicht alle sind jedoch gleichermaßen als Input für die automatische Testfallgenerierung geeignet, da sie häufig die modellierte Software auf einem hohen Abstraktionsniveau darstellen, das mit den hier vorgestellten Methoden nur schwer zu verarbeiten ist. Einen ausreichenden Detailgrad vorausgesetzt, können einige Diagrammtypen jedoch durchaus den Ausgangspunkt für die automatische Testfallerzeugung bilden. Diese Diagrammtypen werden wir in diesem Abschnitt identifizieren und ihre Anwendung kurz erläutern. Dabei betrachten wir gesondert die Spezifikation von Daten und Verhalten des SUT sowie die der Testszenarien.

Für die Modellierung der *Datentypen* in der Systemspezifikation bietet sich die Verwendung statischer Klassendiagramme an, die die Definition von Datenstrukturen und Enumerationen erlauben. Primitive Datentypen wie *Boolean* und *Integer* sind bereits in der *UML* selbst definiert und können direkt verwendet werden.

Das *Verhalten* eines Systems lässt sich in *UML* sowohl als Zustandsdiagramm, als auch als Aktivitätendiagramm darstellen, wobei für unseren Ansatz das Zustandsdiagramm am geeignetsten ist. Das Verhalten eines Systems wird dabei als eine Reihe von Zuständen und Zustandsübergängen definiert. Jeder Zustandsübergang kann ein Ereignis sowie eine Aktion enthalten. Das Eintreten des Ereignisses löst den Zustandsübergang und somit die Aktion aus. Die Auswahl eines Zustandsübergangs kann dabei durch eine Bedingung (*Guard*) erfolgen.

Zustandsdiagramme bieten jedoch auf struktureller Ebene weitere Features. So müssen Zustandsübergänge nicht unbedingt eine Aktion ausführen. Da jedoch in den von uns verwendeten formalen Spezifikationen ein Zustandsübergang mit einer Aktion bewertet sein muss, führen wir für diesen Fall eine interne Aktion τ ein. Zustandsdiagramme und einzelne Zustände können weiterhin komposit sein, d.h. wiederum Zustandsautomaten enthalten (zusammengesetzte Zustände bzw. Zustände mit Unterautomaten). Diese Hierarchien werden bei der Umsetzung in die formale Spezifikation aufgelöst; orthogonale Zustände werden gesondert übersetzt.

Aufgrund ihres Szenariencharakters ist es naheliegend, zur Modellierung von *Testszenarien* Sequenzdiagramme heranzuziehen. Diagramme dieses Typs erlauben es, das SUT sowie das Testsystem als kommunizierende Komponenten darzustellen. Auch die Kommunikationsrichtung, d.h. der Empfang von Eingabesignalen durch das SUT und der Versand von Ausgabesignalen, wird in einem Sequenzdiagramm expliziert.

Die Beschreibung von Testszenarien, wie wir sie hier verwenden, wurde vom Konzept der *Test Purposes* [LdBB$^+$01, JJ05] inspiriert. Diese stellen Zustandsautomaten dar, bei denen verschiedene mögliche zu testende Programmpfade definiert werden können. Hierbei kann auch explizit unterschieden werden, ob ein bestimmtes Programmverhalten für den Test interessant ist, oder ob dieses Verhalten im Test nicht auftreten soll.

3 Erzeugung von Testfällen in *TTCN-3*

Aus den Modellen und Testszenarien werden nun unter Verwendung formaler Methoden Testfälle erzeugt. Wir abstrakieren dazu die originale Systemspezifikation [Dam96] und bilden das synchrone Produkt aus der abstrahierten Spezifikation und den Testszenarien [CIPS05, CIP06]. Aus diesem Produkt erhalten wir abstrakte Testfälle, die in parametrisierbare *TTCN-3* Testfälle transformiert werden. Des weiteren wird aus der originalen Spezifikation ein Testorakel zur Testdatenermittlung generiert.

Die *TTCN-3* Testfälle sind modular aufgebaut, wobei wir eine strikte Unterteilung in Testdaten und den Kontrollfluss des Testfalls vorgenommen haben. Wir unterscheiden folgende Module: Im Modul *DataTypes* sind Datentypdefinitionen abgelegt, im Modul *Variables* die Definition der Testfallparameter, *Signatures* enthält die Definition von Prozedur- bzw. Nachrichtensignaturen sowie *TestCase* den eigentlichen Testcode. Ferner existiert ein Modul *GenericComponents*, das die für die Anbindung des Testfalls an das SUT notwendigen Informationen über die beim Test verwendeten Nachrichtenkanäle bereithält.

Die Transformation der *TTCN-3* Testfälle verläuft im einzelnen folgendermaßen: Zunächst werden die Datentypen der originalen Systemspezifikation in *TTCN-3* definiert. Es handelt sich hierbei zumeist um die üblichen Datentypen wie Strukturen, Enumerationen und Arrays sowie um Mappings auf Basisdatentypen wie *Boolean* oder *Integer*. In den von uns verwendeten formalen Spezifikationen werden algebraische Datentypen definiert, die mehrere solcher Typen in einer Definition vereinen können. So kann bspw. eine Liste als eine Datenstruktur definiert sein, die ein ausgezeichnetes Element als Endemarkierung enthält. Einen solchen Datentyp definieren wir dann mithilfe des *union*-Konstrukts von *TTCN-3* als Mischform zwischen Datenstruktur und Enumeration.

Als nächstes werden alle im Testfall vorkommenden Systemein- und -ausgaben extrahiert und im Modul *Variables* als eine eindeutig benannte Variable definiert, die bei der Testausführung als Parameter für den Testfall dienen.

In einem dritten Schritt werden die modellierten Aktionen des SUT im Modul *Signature* deklariert. Dies kann sowohl nachrichten-, als auch prozedurorientiert geschehen, d.h. es werden Datenstrukturen definiert, die bei der Testausführung zwischen Testsystem und SUT hin- und hergeschickt werden, bzw. die Schnittstellen des SUT nachgebildet.

Im vierten Schritt wird der Testfall selbst in *TTCN-3* realisiert. Er wird als eine ausgezeichnete Funktion definiert, deren Implementation in weitere Funktionen unterteilt werden kann. Wir definieren bei der Erzeugung des *TTCN-3* Codes jeweils eine Funktion für jeden Kontrollflusszustand des Testfalls. Innerhalb dieser Funktionen erfolgt die Kommunikation mit dem SUT, wobei Aus- und Eingaben des Testsystems zu unterscheiden sind.

Funktionen, die Ausgaben des Testsystems an das SUT senden, bestehen nur aus dem Aufruf zum Versenden dieser Nachricht sowie dem Aufruf der dem Folgezustand entsprechenden Funktion im Modul *TestCase*. Funktionen, die eine Eingabe vom SUT entgegennehmen, müssen hingegen eine Entscheidung über die Bewertung dieser Eingabe bzw. den weiteren Testverlauf treffen. Darum sind diese Funktionen als *TTCN-3 alt*-Konstrukt realisert, in dem die verschiedenen Alternativen für den Testverlauf bzw. das Setzen eines Testurteils notiert sind.

4 Zusammenfassung

In diesem Paper haben wir ein Framework zur automatisierten Erzeugung von Testfällen vorgestellt. In diesem Framework werden aus einer *UML*-Spezifikation eines SUT sowie aus Testszenarien Testfälle in *TTCN-3* generiert. Um die Wiederverwendbarkeit der Testfälle zu erhöhen und die Testfallexplosion zu vermeiden, werden diese erst zum Zeitpunkt der Testausführung mit konkreten Testdaten parametrisiert.

Eine Alternative zur Testfallerzeugung von einer abstrahierten Spezifikation ist die *symbolische Testfallgenerierung* [JJRZ05] am nächsten. Wir verwenden Abstraktionstechniken jedoch nicht zur Erreichbarkeitsanalyse, sondern zur Reduzierung externer Daten auf endliche Intervalle, um enumerative Werkzeuge anwenden zu können. Die Verwendbarkeit der beiden Ansätze für bestimmte Systemklassen muss durch weitere Fallstudien geklärt werden. Ferner ist die Erweiterung des Ansatzes um automatische Verfahren zur Auswahl konkreter Testdaten, bspw. durch Äquivalenzklassenanalyse, geplant.

Literatur

[CIP06] Jens R. Calamé, Natalia Ioustinova und Jaco van de Pol. Towards Automatic Generation of Parameterized Test Cases from Abstractions. Technical Report SEN-E0602, Centrum voor Wiskunde en Informatica, March 2006. To appear in ENTCS.

[CIPS05] Jens R. Calamé, Natalia Ioustinova, Jaco van de Pol und Natalia Sidorova. Data Abstraction and Constraint Solving for Conformance Testing. In *12th Asia-Pacific Software Engineering Conference (APSEC 2005)*, Seiten 541–548. IEEE, 2005.

[Dam96] Dennis Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD dissertation, Eindhoven University of Technology, Juli 1996.

[GHR+03] Jens Grabowski, Dieter Hogrefe, György Réthy, Ina Schieferdecker, Anthony Wiles und Colin Willcock. An introduction to the testing and test control notation (TTCN-3). *Computer Networks*, 42(3):375–403, 2003.

[JJ05] Claude Jard und Thierry Jéron. TGV: Theory, Principles and Algorithms. *International Journal on Software Tools for Technology Transfer*, 7(4):297–315, 2005.

[JJRZ05] B. Jeannet, T. Jéron, V. Rusu und E. Zinovieva. Symbolic Test Selection based on Approximate Analysis. In *11th Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05), LNCS Vol. 3440*, Edinburgh, April 2005.

[LdBB+01] Y. Ledru, L. du Bousquet, P. Bontron, O. Maury, C. Oriat und M.-L. Potet. Test Purposes: Adapting the Notion of Specification to Testing. In *16th IEEE International Conference on Automated Software Engineering (ASE'01)*, San Diego, California, 2001.

[Lig02] Peter Liggesmeyer. *Software-Qualität*. Spektrum Akademischer Verlag, 2002.

[Tre96] Jan Tretmans. Test generation with inputs, outputs, and repetitive quiescence. *Software - Concepts & Tools*, 17(3):103–120, 1996.