Modellbasiertes Testen mit AutoFocus/Quest

A. Pretschner und B. Schätz
Institut für Informatik, Technische Universität München
`www4.in.tum.de/~{pretschn,schaetz}`

Zusammenfasung. Das CASE-Tool AutoFocus/Quest erleichtert Spezifikation und Validierung sicherheitskritischer eingebetteter Systeme. Wir präsentieren die dem Werkzeug zugrundeliegenden Ideen, umreißen einen modellbasierten inkrementellen Entwicklungsprozeß und skizzieren ein Verfahren zur Generierung von Testfällen auf der Basis von Constraint-Logikprogrammierung.

1 Einleitung

Evolutionäre SW-Entwicklungsprozesse haben mit der Diskussion, die das Extreme Programming [1] angestoßen hat, erneut an Popularität gewonnen. In diesen – seit langem propagierten – Prozessen wird dem Kunden am Ende definierter kurzer Phasen ein ausführbares System zur Validierung seiner Requirements vorgelegt, was dazu führt, daß Mißverständnisse früh ausgeräumt und intelektuelle Kontrolle über den Prozeß bewahrt werden können.

Viele solcher Prozesse (mit der Ausnahme von z.B. [2, 9]) verwenden Programmiersprachen als Spezifikations- und Implementierungssprache. Diese Programmiersprachen sind aufgrund ihrer komplexen Semantik und des niedrigen Abstraktionsniveaus sicherlich nicht immer die ideale Wahl.

In diesem Beitrag beschreiben wir den Werkzeugverbund AutoFocus/Quest, mit dem reaktive parallele Systeme graphisch spezifiziert, simuliert und wegen der einfachen Semantik auch validiert bzw. verifiziert werden können. AutoFocus-*Modelle* sind Beschreibungen solcher Systeme, die wegen des hohen Abstraktionsgrads inkrementell schnell erstellt werden können. Die einzelnen Inkremente, also ausführbare Teilsysteme, können dann vom Kunden validiert werden. Der erzeugte Code kann, sofern er genügend effizient ist, im späteren System direkt Verwendung finden; ist er es nicht, kann das Modell als Spezifikation für handgeschriebenen, effizienten Code herangezogen werden.

Als entscheidenden Vorteil eines solchen modellbasierten inkrementellen Entwicklungsprozesses sehen wir die Möglichkeit der automatisierten Berechnung von Testfällen aus solchen Modellen. Wir stellen deshalb zunächst die Beschreibungstechniken von AutoFocus und angeschlossene Werkzeuge vor; danach umreißen wir kurz die Erzeugung von Testfällen auf der Basis von AutoFocus-Modellen.

2 AutoFocus/Quest

Bei vielen kommerziell verfügbaren Werkzeuge werden pragmatische Aspekte wie komfortable Codegenerierung und einfache Bedienbarkeit betont; konzeptuelle Grundlagen und semantische Aspekte werden kaum behandelt. Daneben existiert eine Reihe von Werkzeugen, bei denen besonderer Wert auf die Umsetzung mathematischer Techniken gelegt wird, die jedoch für Systementwickler in der Praxis *zu sehr* an mathematischen Konzepten orientiert sind. Der Werkzeugprototyp AutoFOCUS/QUEST [5] unterstützt Kernkonzepte der FOCUS-Methodik [4] zur Entwicklung verteilter Systeme und stellt darauf aufbauend fundierte und anwendungsorientierte Notationen und Methoden zur Verfügung.

Unabhängig davon, wie Spezifikationen eingebetteter Systeme notationell dargestellt werden, läßt sich eine Reihe von grundlegenden Konzepten identifizieren, mit denen solche Systeme beschrieben werden:

Komponenten sind die Grundbausteine eines eingebetteten Systems. Sie stellen Einheiten dar, die Daten, Struktur (in Form von Schnittstellen und Unterkomponenten) und Verhalten umfassen und mit ihrer Umwelt kommunizieren.

Datentypen definieren die Datenstrukturen, die in einem eingebetteten System gespeichert bzw. verarbeitet werden.

Daten sind in Komponenten gekapselt und erlauben die Speicherung von Zustandsinformation. Daten werden durch einfache getypte Zustandsvariablen realisiert.

Kanäle sind gerichtete, mit Namen und Datentyp versehene Kommunikationsverbindungen zwischen Komponenten. Sie sind mit den Komponenten über Ports verbunden.

Verhalten beschreibt, wie eine Komponente auf Eingaben aus ihrer Umwelt reagiert. Diese Reaktion ist ein Ergebnis der Eingaben und des internen Zustands der Komponente. Das Verhalten wird zustandsorientiert durch Kontrollzustände, Transitionen und Datenelemente beschrieben.

Diese elementaren Konzepte der FOCUS-Theorie stellen die Grundbausteine des konzeptuellen Modells und damit auch des formalen Modells als auch der konkreten Notationen dar. Dieses konzeptuelle Modell - vergleichbar dem Metamodell in der UML - erlaubt es, die einzelnen Beschreibungsformen auf ein gemeinsames Modell abzubilden, und so einer Systembeschreibung ein konsistentes abstraktes syntaktisches Modell zuzuordnen. Dies ermöglicht auch den diagrammübergreifenden Einsatz von Konsistenzbedingungen, beispielsweise in einer OCL-Form, um Anforderungen an das Modell (z.B. Determinismus von Transitionen) für den Entwicklungsprozess zu definieren [3].

Um eine umfassende und strukturierte Spezifikation eines Systems erstellen zu können, werden zur Darstellung des Systems Sichten verwendet, die konsequent hierarchisch strukturiert sind und methodische Konzepte wie Dekomposition und Verfeinerung bzw. Verbergen von Detailinformation und Abstraktion explizit unterstützen:

Systemstrukturdiagramme (SSDs) beschreiben den statischen Aufbau eines Systems bestehend aus seinen Komponenten sowie den Kommunikationskanälen, die zwischen ihnen verlaufen. Da jede Komponente selbst als verteiltes System gesehen werden kann, ist es möglich, hierarchische Systemspezifikationen zu erstellen. In AutoFOCUS/QUEST kann eine Komponente mit einer hierarchischen Unterstruktur, gegeben durch ein SSD, mit einer Verhaltensbeschreibung durch Zustandsdiagramme (STDs) oder Event Traces (EETs) und mit Komponentendatendeklarationen (CDDs) versehen werden.

Datentypdefinitionen (DTDs) beschreiben diejenigen Typen von Daten, die verwendet werden, um lokale Komponentenvariablen zu deklarieren, Datentypen von Kanälen und Ports festzulegen, sowie lokale Variablen in Transitionen zu deklarieren.

Zustandsübergangsdiagramme (STDs) beschreiben das Verhalten eines Systems oder einer Komponente in Form der Reaktionen auf Eingaben aus der Umgebung und der daraus resultierenden Ausgaben in die Umgebung. Die Reaktionen hängen vom aktuellen Zustand der Komponente ab. Da ein STD eine erweiterte endliche Zustandsmaschine mit lokalen Datenzuständen der zugehörigen Komponente beschreibt, wird der Gesamtzustand einer Komponente durch den Kontroll- und Datenzustand charakterisiert. Ähnlich wie bei SSDs kann jeder Zustand eines STDs wiederum in ein weiteres STD dekomponiert werden. Transitionen sind dabei mit Vor- und Nachbedingungen annotiert, die durch Prädikate über dem lokalen Datenzustand der zugehörigen Komponente gegeben sind. Außerdem enthalten sie eine Menge von Ein- und Ausgabemustern, welche die Nachrichten, die von den Eingabeports gelesen bzw. auf den Ausgabeports ausgegeben werden, beschreiben.

Im AutoFOCUS/QUEST-Ansatz wird auf allen Eingabeports simultan gelesen und auf alle Ausgabeports simultan geschrieben. Als Konsequenz dieser ungepufferten Semantik ist das Verhalten von AutoFOCUS/QUEST-Komponenten näher an hardwareorientierten Beschreibungstechniken, wie beispielsweise StateCharts, als an abstrakten Systembeschreibungssprachen wie SDL (mit Eingabepuffern für jede Komponente).

Ereignisdiagramme (EETs) beschreiben Verhalten aus einem komponenten- und kommunikationsorientierten Blickwinkel durch exemplarische Kommunikationshistorien ähnlich dem ITU-Standard für Message Sequence Charts (MSCs).

Die Integration dieser Sichten, insbesondere auf der Grundlage eines gemeinsamen mathematischen Modells, führt zu einer integrierten und formalen Spezifikation des Systems. Dabei wurde die oben beschriebene, hardwareorientierte Semantik aus zweierlei Gründen bewusst gewählt: einerseits ist das getaktete Verarbeitungskonzept sehr ähnlich zu dem in SPS-Systemen verwendeten (z.B. AWL) und daher im Bereich eingebetteter Systeme gut akzeptiert. Andererseits erlaubt die vergleichsweise einfache Semantik verschiedene effektive Verfahren, um einen weitreichenden Prozeß auf Basis des Konzeptmodells in AutoFOCUS/QUEST zu unterstützen. Beispielsweise gestattet die einfache Anbindung von Model Checkern in Kombination mit einem aufgrund der taktsynchronen Kommunikation reduzierten Zustandsraum verschiedene automatische Verifikationsverfahren in der Designphase: die Überprüfung der Verfeinerungsbeziehung zwischen STDs und EETs (realisiert mit dem relationalen μ-Kalkül-Checker μcke) oder die Verifkation temporaler Eigenschaften von STDs (realisiert mit dem Model Checker SMV), jeweils mit Gegenbeispielgenerierung. Für die späte Designphase steht beispielsweise die Simulation einer Spezifikation auf Diagrammebene zur Verfügung. Die Umsetzungsphase wird durch eine parametrierte Codegenerierungsschnittstelle (z.B. C, Java, ADA) unterstützt. Als Ergänzung zu den konstruktiven und analytischen Verfahren der Analyse-, Design- und Umsetzungphase stehen auch Testverfahren zur Verfügung, wie im Folgenden beschrieben.

3 Modellbasiertes Testen

Durch die Einbettung in einen inkrementellen Entwicklungsprozeßist die ständige Berechnung von Testfällen aus Modellen nicht nur zum Zweck des Regressionstesten des Modells an sich wünschenswert, sondern auch für den Test der späteren Implementierung. Das ist der Fall, wenn letzere durch Hardware realisiert wird oder wenn die Codegeneratoren z.B. in bezug auf Performanz inadäquaten Code erzeugen.

Modellbasiertes Testen eignet sich also nicht nur für den Conformance-Test der Implementierung, sondern auch als Debugginghilfe zur Fehlerlokalisation beim Design der ausführbaren Spezifikation. Dabei spielen zwei Arten von Testfällen eine Rolle: funktionale und strukturelle Tests. Funktionale Tests dienen der (black box) Validierung einer spezifischen Funktionalität (im Extreme Programming [1] stellen sie gar die Spezifikation dar). Strukturelle (white box-) Tests haben üblicherweise den Zweck, ein bestimmtes Überdeckungskriterium, z.B. Transitions- oder Bedingungsabdeckung, zu erfüllen.

Beide Arten von Testfällen müssen in geeigneten Sprachen formuliert werden (z.B. TTCN, Message Sequence Charts oder Automaten [10]) Als diesbzüglich erfreulich praktikabel haben sich mit Hilfe der Constraint Handling Rules (CHR, [6]) definierbare Testfälle erwiesen, die wir als backend für noch festzulegende graphische Eingabesprachen favorisieren. CHR erlauben die Definition von Constraint-Handlern, die z.B. aus einer Constraint-Logiksprache (CLP) heraus aufgerufen werden können. Letztere unterscheiden sich von klassischen Logiksprachen insofern, als sie im Gegensatz zum dort üblichen Ansatz des "generate and test" ein a-priori-Beschneiden des Suchbaums und damit ggf. signifikante Suchraumeinschränkungen gestatten (vgl. on-the-fly-Model Checking).

Ein CLP-basierter Ansatz zur Erzeugung von Testfällen weist aber noch weitere Vorteile auf. Durch die Flexibilität bei der Definition von Testfällen wird eine explizite Kontrolle über die "Gegenbeispiel"-Erzeugung möglich, wie sie bei der Verwendung von Model Checkern oder propositionalen Lösern wie SATO [10] zum selben Zweck nur schwer vorstellbar sind. Erste industrielle Fallstudien weisen darauf hin, daß diese Flexibilität der Schlüssel zur Skalierbarkeit unseres Ansatzes sein könnte – das Slicing von Modellen durch Weglassen von Zuständen oder Transitionen beispielsweise gestaltet sich in der Formulierung durch Constraints sehr einfach.

Im wesentlichen funktioniert die Testfallberechnung wie folgt: Aus einem AutoFocus-Modell (SSDs, DTDs, STDs) wird vollautomatisch CLP-Code für Automaten, Funktions- und Datentypdefinitionen generiert [7]. Der Testfall (funktional oder strukturell) wird mit Hilfe vordefinierter oder selbstgeschriebener CHR-Handler formuliert; unter diesen Constraints wird das Modell dann symbolisch für eine maximale (endliche) Anzahl von Schritten ausgeführt. Diese Ausführung ist üblicherweise nichtdeterministisch, weil z.B. nur wenige konkrete Ein- oder Ausgaben vorliegen. Ist die maximale Schrittzahl erreicht oder sind alle Constraints erfüllt, müssen im ersten Fall verbleibende Constraints (z.B. Bedingungen an Variablen) so aufgelöst werden, daß eine möglichst "gute", kleine ausführbare Testfallmenge entsteht (Grenzwertinstantiierungen). Im zweiten Fall muß dasselbe ggf. für ungebundene - weil für den Systemlauf irrelevante - Variablen geschehen; diese werden ihrem Typ gemäß instantiiert. Backtracking liefert dann weitere Testfälle.

Bei der symbolischen Ausführung helfen verschiedene Strategien bei der Vermeidung von Endlosschleifen, die durch Prologs Tiefensuche induziert werden: (1) Für jeden Zustand wird memoriert, welche Transition als letztes aus diesem genommen wurde; wenn der Zustand erneut erreicht wird, kann diese Information entsprechend genutzt werden. (2) Die Definition von Transitionswahrscheinlichkeiten zum Erzielen guter Abdeckungsraten ist ebenfalls möglich.

Experimente [8] anhand industrieller Fallstudien haben gezeigt, daß die vorzugebende maximale Länge der Testsequenz entscheidenden Einfluß auf die (zeitliche) Performanz hat aufgrund von Prologs Tiefensuch- und Backtrackingmechanismus. Ein Hauptziel unserer aktuellen Arbeiten ist daher die Definition von Fitneß- bzw. Entfernungsfunktionen, die eine "intelligentere" Auswahl von Transitionen z.B. im A*-Algorithmus ermöglichen.

4 Ausblick

In diesem Beitrag haben wir den Werkzeugverbund AutoFocus/Quest zur Entwicklung reaktiver Systeme vorgestellt. Die Einbettung in einen inkrementellen Entwicklungsprozeß stellt die Grundlage unserer constraintbasierter Strategien zur Berechnung von Testfällen dar. In bezug auf AutoFocus/Quest finden momentan Arbeiten zum Anschluß weiterer Werkzeuge (z.B. DOORS, ATTOLL Unit-Test/Coverage) statt; darüberhinaus wird ein Algorithmus zur automatischen Generierung von Automaten aus Sequenzdiagrammen implementiert.

Die Testfallgenerierung wird, wie bereits erwähnt, durch eine "intelligentere", A*-artigen Suchstrategie für die Auswahl von Transitionen (und Dateninstantiierungen) erweitert. Das Ziel in bezug auf Zertifizierung ist, eine Coverage auf *Modellebene* automatisch auf eine Coverage auf *Codeebene* zu liften. Voraussetzung dafür ist eine einfache Semantik, wie sie AutoFocus zugrundeliegt; die Techniken zur Codeerzeugung aus Modellen könnten denen ähnlich sein, mit denen Testfälle für Modelle auf Testfällen auf Codeebene abgebildet werden.

Nicht erwähnt wurden Arbeiten zur Kompositonalität von Testfällen, die Problematik des Testens nichtdeterministischer Systeme sowie die Ähnlichkeit einer CLP-basierten Simulation mit Abstrakter Interpretation, die dazu führen kann, Abstraktionen automatisch zu berechnen. Abstraktionen spielen auch eine Rolle in unseren Arbeiten zum Test gemischt diskret-kontinuierlicher Systeme. Darüberhinaus bietet die Fragestellung nach einem geeigneten theoretischen Konzept für den Zusammenhang von Automaten mit Testfällen viel Raum für zukünftige Arbeiten. Schließlich validieren wir unseren Ansatz in mehreren industriellen Kooperationen.

Danksagung. P. Braun, F. Huber, H. Lötzbeyer, O. Slotosch und G. Wimmel sind maßgeblich an der Entwicklung von AutoFocus/Quest beteiligt; die Arbeiten zur CLP-basierten Testfallgenerierung entstehen in Zusammenarbeit mit H. Lötzbeyer.

Literatur

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

[2] M. Boger, T. Baier, F. Wienberg, and W. Lamersdorf. Extreme modeling. In *Proc. Extreme Programming and Flexible Processes in SW Engineering (XP'00)*, 2000.

[3] P. Braun, H. Lötzbeyer, B. Schätz, and O. Slotosch. Consistent integration of formal methods. In S. Graf and M. Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of System 2000*, 2000.

[4] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. Gritzner, and R. Weber. The Design of Distributed Systems - An Introduction to FOCUS. TUM-I 9202-2, Technische Universität München, Jan. 1993. SFB-Bericht Nr.342/2-2/92 A.

[5] M. Broy, F. Huber, and B. Schätz. Autofocus – ein werkzeugprototyp zur entwicklung eingebetteter systeme. *Informatik Forschung und Entwicklung*, 14(3):121–134, 1999.

[6] T. Frühwirth. Theory and practice of constraint handling rules. *J. Logic Prog.*, 37(1-3):95–138, October 1998.

[7] H. Lötzbeyer and A. Pretschner. AutoFocus on Constraint Logic Programming. In *LPSE'00*, July 2000.

[8] A. Pretschner, H. Lötzbeyer, and J. Philipps. Model Based Testing in Evolutionary SW development, 2001. Submitted to RSP'01.

[9] S. Prowell, C. Trammell, R. Linger, and J. Poore. *Cleanroom Software Engineering*. Addison Wesley, 1999.

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