Modelle als Mittel zur Verstärkung von strukturellen Tests sicherheitskritischer Software

Dirk Wischermann, Wolfgang Schröder-Preikschat
Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl für Verteilte Systeme und Betriebssysteme
Erlangen, Deutschland
{dw,wosch}@cs.fau.de

Abstract:
Software kann durch eine gezielte Überdeckung des Programmtextes mit Tests systematisch auf Fehler abgesucht werden. Viele Fehler treten aber nur bei der Ausführung ganz *bestimmter Pfade* durch den Programmtext auf. Da die Zahl dieser Pfade mit der größe des Programms schnell ansteigt, sinkt die Wahrscheinlichkeit der Fehleraufdeckung. Mit diesem Aufsatz stellen wir eine Methode vor (Arbeitstitel PERSPECTIVE TESTING), um mit Verhaltensmodellen eine Auswahl besonders relevanter Testfälle zu treffen und diese effektiv zu beurteilen. Die Modelle werden dabei in zweierlei Hinsicht genutzt: Zum gezielten Abtesten der modellierten Verhaltensgesichtspunkte mit starken Überdeckungskriterien und zur Generierung von *Kontrollflussorakeln*, das sind im Code verwobene Prüfpunkte, die die Modellkonformität des Kontrollflusses überwachen. Dadurch unterscheidet sich unser Ansatz erheblich von den meisten modellbasierten Testansätzen, die vor allem einen hohen Automatisierungsgrad oder die Wiederverwendung von Modellen aus dem Entwicklungsprozess zum Ziel haben.

1 Einleitung

Software, die für formale Verifikation oder vollständiges Testen zu komplex ist, wird längst auch schon in sicherheitskritischen Bereichen angewendet. Umfassende funktionale Tests gegen eine formale Spezifikation können dabei sinnvoll von strukturellen Tests[1] ergänzt werden, mit denen eine gewisse Evidenz für die *Abwesenheit* von Fehlern ermittelt wird. Allerdings ist dabei gerade die Aufdeckung von Fehlern mit Kontrollflussbezug nur mit geringer Wahrscheinlichkeit gewährleistet, da die Zahl der zu testenden Pfade einfach zu groß wird. Es gilt also, möglichst repräsentative Pfade für die Ausführung vorzusehen.

Modelle können beim Testen in vielerlei Hinsicht genutzt werden [UPL06]. Oft wird mit Modellen von der komplexen Struktur des Programmes abstrahiert und Testfallauswahl und Testfortschritt anhand von struktureller Überdeckung *des Modells* gemessen und dargelegt ("Grey-Box Testen", z.B. in [PSN09]). Damit lässt sich aber nur bedingt Vertrauen in die Richtigkeit *des Codes* gewinnen, denn oft korrespondiert eine Modellentität zu mehreren Stellen im Code. Genau hier setzt unsere Methode an, indem Verhaltensmo-

[1] Für die Grundbegriffe strukturellen Testens verweisen wir auf einschlägige Literatur, exemplarisch [Bei90].

delle auf den Code projiziert werden und der dadurch ausgezeichnete Code mit starken Überdeckungskriterien getestet wird (reines "White-Box Testen"). So können systematisch Fehler mit Kontrollflussbezug aufgedeckt werden. Der Ansatz eignet sich für Integrationstesten (Test auf die richtige Verwendung von Schnittstellen) sowie allgemein für den Test von Komponenten auf die Einhaltung modellierter Verhalten oder Protokolle.

Übersicht Der Hauptteil des Aufsatzes ist zweigliedrig: In Abschnitt 2.1 werden die von uns verwendeten Modelle vorgestellt und ihre Bedeutung am Beispiel erläutert. Im Abschnitt 2.2 wird ausgeführt, wie daraus Überdeckungsmetriken und Kontrollflussorakel gewonnen werden. Da es sich um eine laufende Arbeit handelt, präsentieren wir noch keine Ergebnisse. Eine kurze Zusammenfassung rundet die Arbeit ab.

2 Testen nach Gesichtspunkten

Bei PERSPECTIVE TESTING setzen wir endliche Automaten ein, die einzelne zu testende Verhaltensaspekte (abgekürzt CUT, Concern under Test) einer Komponente näher beschreiben. Unser Ansatz gewinnt aus den Modellen zwei Mittel, um einen solchen CUT besonders gut nach Fehlern mit Kontrollflussbezug (zu denen mehrere Programmzeilen beitragen) abzusuchen:

1. Eine gezielte Verstärkung der Überdeckungskriterien für den CUT.
2. Automatisch generierte *Kontrollflussorakel*, die solche Fehler durch eine dem Automaten entsprechende Instrumentierung des Codes aufdecken.

Insbesondere decken die Kontrollflussorakel auch Fehler auf, die sich (noch) nicht in Softwareversagen niedergeschlagen haben. Ein einfaches Beispiel eines CUT, der von Unit-Tests häufig nicht ausreichend erfasst wird, ist die Freigabe einer wiederverwendbaren Resource (Speicher, eine Datei oder eine Sperrvariable) nach deren Verwendung. Die funktionale Bewertung der Testläufe sowie die eigentliche Erzeugung der Testfälle sind nicht Thema dieses Aufsatzes.

Wir begleiten unsere Ausführungen mit einem realistischen Beispiel. Das Szenario folgt einem typischen Aufbau sicherheitskritischer Software, gegeben durch ein Bibliotheksbetriebssystem, das (von einem Systemintegrator) zusammen mit Treibern und Anwendungskomponenten übersetzt und auf ein eingebettetes System mit Sensoren und Aktoren verbracht wird. Als CUT betrachten wir die differenzierte Synchronisation von Aktivitätsträgern beim Zugriff auf Sensoren, Aktoren und gemeinsamen Speicher.

2.1 Gestalt und Bedeutung von CUT-Modellen

CUT-Modelle sind endliche Automaten, die eine *gerichtete* Abstraktion vom Code darstellen: Bezüglich der betrachteten Zuständigkeit bleiben sie praktisch auf Ebene des Codes,

andere Belange werden dagegen völlig ausgeblendet. Den direkten Code-Bezug stellt die Menge der *relevanten* Operationen her (ggf. mit Jokerzeichen[2], die von dem Automaten berücksichtigt werden sollen. Wir betrachten solche Modelle in zwei verschiedenen Rollen: Als *Spezifikationsmodelle* geben sie ein grundsätzlich erlaubtes Verhalten an und als *Designmodelle* stellen sie das vom Entwickler intendierte Verhalten der Software dar.[3]

In unserem Beispiel wird ein globales Regelwerk für die Verwendung von Sperrvariablen vorgegeben (siehe Abbildung 1). Das Protokoll verhindert Verklemmungen durch die Vermeidung zirkulärer Wartebedingungen. Dies kann leicht (zumindest für eine feste Zahl an Instanzen) mit einem Modelchecker überprüft werden. Ein Systemintegrator kann auf dieser Grundlage sämtliche Komponenten (Treiber, Anwendungen), die in das System eingebracht werden sollen, überprüfen und Verklemmungen zuverlässig ausschließen.

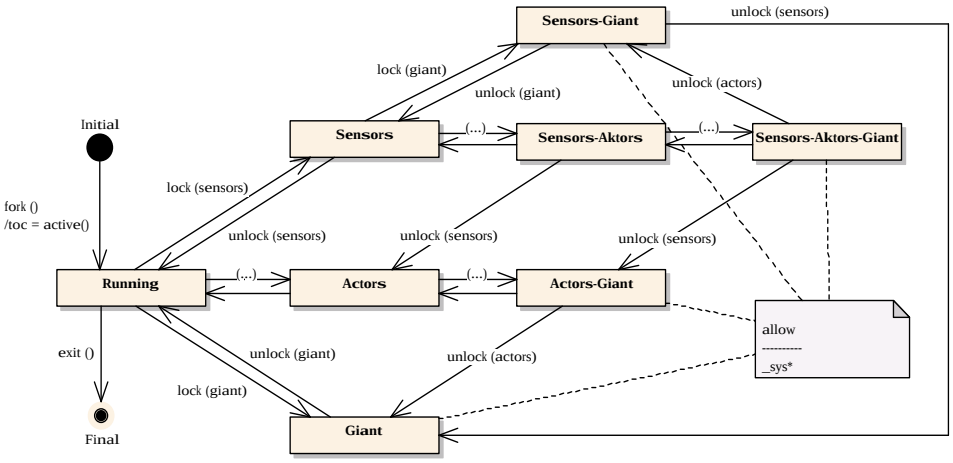

Abbildung 1: Allgemeiner Reihenfolgezwang für Sperrvariablen

In unserem Szenario spezialisiert der Entwickler diese Vorgabe dahingehend, dass je ein Faden[4] für das Auslesen der Sensoren und für die Bedienung von Aktoren verwendet wird. Ausserdem modelliert er für den Datenaustausch zwischen beiden die Verwendung eines beschränkten Puffers mit Semaphoren. Das Design der beiden Fäden (Abbildung 2) schränkt weiter ein, dass jeder Faden nur die tatsächlich benötigten Sperrvariablen auch belegen darf.

[2]Das für PERSPECTIVE TESTING gewählte Mittel ist derzeit die AspectC++ Pointcut Sprache [SGSP02]. Allgemein ist eine *Traceability* zwischen Code und Modell notwendig.

[3]Design Modelle spezialisieren Spezifikationsmodelle, siehe [WSP10]. Bei Design-Modellen kann *zusätzlich* zur Code-Überdeckung auch die Überdeckung von Modell-Entitäten betrachtet werden. Das zeigt, ob das Modell vollständig implementiert wurde.

[4]Faden: Engl. Thread of Control (toc)

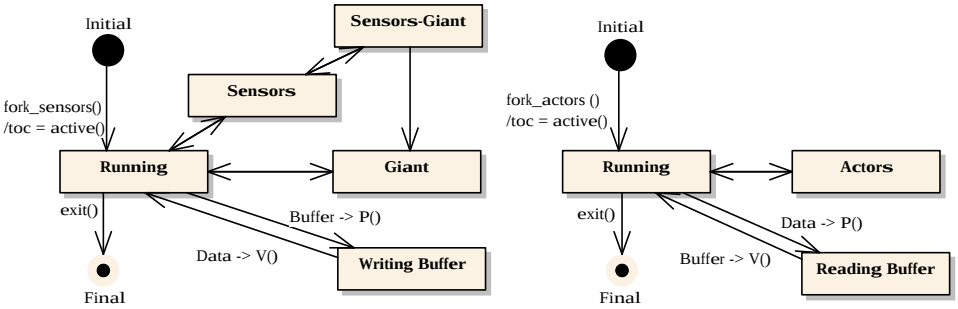

Abbildung 2: Modelle für Aktor- und Sensorthread mit Semaphorenbenutzung

2.2 Anwendung der Modelle beim Testen

Die Modelle werden auf zwei Arten genutzt: Zum einen zeichnen sie Teile des gesamten Programmtextes als *relevant* für einen Belang aus, was eine gezielte Verstärkung der Überdeckungskriterien erlaubt. Zum anderen werden aus ihnen *Kontrollflussorakel* extrahiert, die für jeden Testfall bewerten, ob der beschrittene Pfad dem Modell genügt.

Modellbasierte Überdeckungskriterien Voraussetzung für die Definition der verstärkten Überdeckungskriterien ist die Festlegung, wann Ausführungspfade durch den Programmtext als gleichwertig für einen CUT angesehen werden sollen:

Definition 1 *Zwei Pfade p und p' heissen **äquivalent bezüglich eines CUT-Modelles** $\mathcal{M}_D$ ($p \equiv|_{\mathcal{M}_D} p'$), wenn sie die selben relevanten Operationen von $\mathcal{M}_D$ in gleicher Reihenfolge durchlaufen.*[5]

Dies impliziert auf natürliche Weise eine Vereinfachung des Kontrollflussgraphen, sodass dieser nur noch die relevanten Pfade repräsentiert. Das erlaubt es uns, *beliebige* konventionelle Überdeckungskriterien auf die beschnittene Version des Kontrollflussgraphen zu übertragen. Für die Aufdeckung kontrollflussbezogener Fehler sind vor allem die Pfadkriterien (mit und ohne Schleifenbegrenzung) von Bedeutung; es können aber auch Kriterien der Bedingungsüberdeckung übertragen werden, indem sie nur auf Verzweigungen im beschnittenen Kontrollflussgraphen angewendet werden.

Die Modelle des Beispiels verlangen, die wechselseitige Puffer-Synchronisation mit Semaphoren *gemeinsam* mit der übergreifenden Synchronisation mit Sperrvariablen zu testen. Ein Pfadkriterium würde daher auch die Pfade enthalten, auf denen während der Pufferverwendung noch das `Actors`-Lock gehalten wird. Um einen solchen Fehler (der sich kaum manifestiert) tatsächlich aufzudecken, eignen sich *Kontrollflussorakel*.

[5] Eine abschwächende Definition würde auch Pfade identifizieren, die Abfolgen gleich lautender relevanter Operationen enthalten.

Kontrollflussorakel als zustandsbasierte Selbsttests Kontrollflussorakel sind das Mittel zur tatsächlichen Aufdeckung kontrollflussbezogener Fehler (beispielsweise fehlende Methodenaufrufe, falsche Methodenreihenfolge oder Aufrufe am falschen Objekt). Sie entscheiden für jede relevante Operation,

- ob eine neue Automateninstanz (s.u.) benötigt wird (bei daten- oder threadbezogenen Verhalten),
- ob eine Operation gerade erlaubt ist
- und ob die Operation einen Zustandsübergang auslöst.

Die CUTs im Beispiel werden fadenweise beobachtet, es wird also beim Starten eines neuen Thread eine den Automaten repräsentierende Datenstruktur angelegt und bei relevanten Operationen aktualisiert. Kommt es zu einer Verletzung des Synchronisationsprotokolls, kann eine Fehlermeldung mit der Historie von relevanten Codestellen zur Unterstützung der Fehlersuche ausgegeben werden. Der besondere Vorteil liegt darin, dass die Fehler oft früher und unabhängig von ihrer Manifestation aufgedeckt werden als durch ein konventionelles Orakel. Der Rückschluss auf die eigentliche Fehlerstelle wird dadurch vereinfacht.

Faktorisierung von Testobliegenheiten und Gestaltung für Testbarkeit Modelle dürfen sich ausdrücklich in Zuständen und Kanten überschneiden. Das ist ein Vorteil des Testers gegenüber dem Entwickler, der es ihm erlaubt, Testobliegenheiten entlang von Belangen zu *faktorisieren*, anstatt sie entlang der Programmstruktur zu spalten. Mit der Möglichkeit, beliebige Kriterien für verschiedene CUTs anzugeben, kann eine Komponente *adäquat* in dem Sinne getestet werden, dass wichtige Eigenschaften besser überprüft werden als andere. Bemerkenswert ist, dass der Testaufwand durch die Trennung der Belange erheblich eingedämmt werden kann: Die Anwendung von Pfadkriterien auf mehrere Belange *einzeln* verhält sich diesbezüglich nur additiv (statt multiplikativ).

In der gezielten Verstärkung von Überdeckungskriterien liegt ein Mittel, um die Gestaltung der Software zu Gunsten der *Testbarkeit* sicherheitskritischer Eigenschaften zu gewährleisten. Idealerweise wird eine solche Eigenschaft als Automat modelliert, mit einem Modelchecker formal überprüft und dann mit möglichst starken Überdeckungskriterien, die nur mit entsprechend testbarem Design zu erreichen sind, getestet. Beispielsweise verbietet (im Extremfall) die Forderung nach Pfadüberdeckung, dass relevante Operationen in Schleifen einbezogen werden. Andere Aspekte der Softwaregestaltung (Wiederverwendung, Variabilität) müssen sich der Testbarkeit dann nötigenfalls unterordnen.[6]

Verwandte Arbeiten und Ausblick Es gibt ausgesprochen viele verwandte Arbeiten, beispielsweise aus den Bereichen von modellbasiertem Testen, Integrationstesten, AOP, Code Slicing, Modelchecking oder Policy Enforcement. Um den Rahmen nicht zu sprengen, verweisen wir nur kurz auf zwei Arbeiten: Das SLAM Projekt [BR02] (enthalten

[6] Brückenpfeiler werden auch in erster Linie dick genug gebaut – auch wenn das mehr kostet und vielleicht nicht so gut aussieht.

im *Static Driver Verifier* von Microsoft) arbeitet zwar statisch, ist aber dennoch eng mit unserem Ansatz verwandt. Dort werden Modelle aus dem Code extrahiert und auf (insgesamt 65) temporallogische Eigenschaften untersucht. Die Vorgehensweise ist allerdings auf kleine Protokolle beschränkt. Als Ausblick sei die Idee erwähnt, mit symbolischer Ausführung [CDE08], *gezielt* nach Testfällen zu suchen, die eine Protokollverletzung nachweisen. Symbolische Ausführung könnte sich auch eignen, nicht überdeckte Pfade als nicht ausführbar nachzuweisen, indem die symbolisch aus den Fallunterscheidungen akkumulierten Bedingungen als widersprüchlich offenbart werden.

3 Fazit

In diesem Aufsatz wurden zwei Mittel des modellbasierten strukturellen Testens vorgestellt, die sich zur gezielten Aufdeckung von Fehlern mit Kontrollflussbezug eignen. Ausgehend von endlichen Automaten (idealerweise mit Modelchecking geprüft), wurden (Pfad-)Überdeckungskriterien *auf den Code* projiziert. Sie können aufzeigen, ob eine Zuständigkeit ausgiebig genug getestet wurde. Kontrollflussorakel stellen dabei sicher, dass der Kontrollfluss einem intendierten Muster entspricht. Damit stellt PERSPECTIVE TESTING Mittel zur Verfügung, die dynamischen Abläufe möglichst strikt an die Vorgaben zu binden.

Literatur

[Bei90] Boris Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[BR02] Thomas Ball und Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. *SIGPLAN Not.*, 37(1):1–3, 2002.

[CDE08] Cristian Cadar, Daniel Dunbar und Dawson R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, Seiten 209–224, 2008.

[PSN09] Florin Pinte, Francesca Saglietti und Achim Neubauer. "Visualisierung überdeckter sowie zu überdeckender Modellelemente im modellbasierten Test". *Beiträge der 39. Jahrestagung der Gesellschaft für Informatik e.V. (GI)/ Workshop Modellbasiertes Testen (MOTES) Vol. P-154 of Lecture Notes in Informatics (LNI)*, 2009.

[SGSP02] Olaf Spinczyk, Andreas Gal und Wolfgang Schröder-Preikschat. AspectC++: An Aspect-Oriented Extension to C++. In *40th (TOOLS Pacific '02)*, Seiten 53–60, Sydney, Australia, Februar 2002.

[UPL06] Mark Utting, Alexander Pretschner und Bruno Legeard. A taxonomy of model-based testing. Bericht, University of Waikato, April 2006.

[WSP10] Dirk Wischermann und Wolfgang Schröder-Preikschat. Separating Testing Concerns by Means of Models. In *1st ECOOP Workshop on Testing Object-Oriented Systems (ETOOS)*, June 2010. to appear.