Reengineering-Verfahren und Software-Test

Matthias Hamburg, Sogeti, Düsseldorf
Timea Illes, Universität Heidelberg
Stefan Jungmayr
Rainer Koschke, Universität Bremen

Abstract: Der dynamische Test von Software hat einen hohen Ressourcenbedarf. Daher ist es sinnvoll, durch automatisierte statische und dynamische Analysen bereits vor dem dynamischen Test möglichst viele Fehler zu identifizieren und zu entfernen. In diesem Bericht wird dargestellt, welchen Beitrag Reengineering-Verfahren für die Fehlerentdeckung leisten können. Gleichzeitig wird diskutiert, welche Test-Artefakte für das Reengineering von Interesse sind.

1 Einleitung

Der Arbeitskreis „Testen objektorientierter Programme“ hat im Herbst 2005 eine Umfrage zu Fehlerhäufigkeiten in objektorientierter Software durchgeführt und ausgewertet [1]. Im Anschluss stellte sich die Frage, welche der Fehlerkategorien automatisch durch Werkzeuge gefunden werden können.

Im Rahmen eines gemeinsamen Treffens der GI-Fachgruppen Reengineering und TAV im Mai 2006 in Bad-Honnef wurde u.a. diskutiert, inwieweit Reengineering-Verfahren zur Fehlersuche im Test-Kontext verwendet werden können. Diskussionsinhalt waren sowohl etablierte Reengineering[1]-Verfahren, die bereits in kommerziellen Produkten verfügbar sind, als auch Reengineering-Verfahren, die derzeit noch erforscht werden.

Unter den etablierten Verfahren im Reengineering-Bereich sind die Datenfluss-Analyse und die Klonerkennung (d.h. die Aufdeckung von Code-Duplikaten) besonders gut geeignet, um potentiell fehlerhaften Code zu finden. Noch in der Erforschung stehen Verfahren zur statischen und dynamischen Rekonstruktion von Protokollen buw. Zustandsautomaten. Die genannten Verfahren werden in den folgenden Abschnitten kurz dargestellt und die Motivation im Kontext Reengineering und Test erläutert. Abschließend werden offene Fragen diskutiert und die Ergebnisse zusammengefasst.

2 Datenfluss-Analyse

In der Datenfluss-Analyse werden für eine Variable alle Lese- und Schreibzugriffe aus dem Source-Code extrahiert und als Graph dargestellt. Für die Erstellung des Graphen wird die Repräsentation als „Static-Single-Assignment-Form“ verwendet, bei der jede Variable genau einmal definiert wird. Für die Überführung des ursprünglichen Quelltextes in eine Static-Single-Assignment-Form existieren effiziente Algorithmen [2].

Folgende Herausforderungen stellen sich bei der Datenfluss-Analyse:

- Aliases[2]: Um jeden Zugriff auf eine Variable zu kennen, müssen alle ihre Aliase bekannt sein. Diese können, z.B. bei dynamischer Berechnung von Zeigern, eine hohe Komplexität des Datenflusses verursachen.
- Seiteneffekte: hier ist eine globale und damit teure Analyse des Source-Codes notwendig.
- Aufrufe von Bibliotheksklassen wie z.B. Swing: hier könnte potentiell jede Variable von Seiteneffekten betroffen sein. Pessimistische Ansätze würden den Wert jeder Variable ungültig machen. Aktuell wird am Lehrgebiet von Prof. Koschke an der Universität Bremen untersucht, mit welchen Annahmen man die besten Resultate erzielen kann.

2.1 Motivation im Kontext Reengineering

Im Reengineering bildet die Static-Single-Assignment-Form u.a. die Grundlage des Program-Slicing, einer Technik zum Zerlegen großer Komponenten in kohäsive, lose miteinander gekoppelte Teile, sowie zur Überprüfung von Vorbedingungen automatischer Code-Transformationen.

2.2 Motivation im Kontext Testen

Mit Hilfe der Datenfluss-Analyse können Datenfluss-Anomalien aufgedeckt werden wie z.B. eine Verwendung einer Variablen ohne vorhergehende Initialisierung (welche zur Laufzeit z.B. zu einer Nullpointer-Exception führen kann). Eine verbesserte Kohäsion der Komponenten durch eine Bündelung der Variablenzugriffe führt außerdem zu besserer Testbarkeit.

3 Aufdeckung von redundantem Code

Der Source-Code von Modulen bzw. Klassen eines Systems wird auf redundante Teile (sog. Code-Klone) unter-

[1] Reengineering umfasst die Untersuchung und Änderung eines bestehenden Systems, um es in neuer Form zu implementieren und zu restrukturieren. Ziel ist dabei z.B. die Verbesserung der Änderbarkeit oder eine Migration auf eine andere Technologie.

[2] Aliases sind unterschiedliche Namen für überlappende Speicherbereiche. Aliase entstehen z.B. durch Übergabe der selben Variable an zwei formale Referenzparameter derselben Funktion, durch indizierte Feldzugriffe oder durch Zeiger

sucht. Dabei können token-basierte und syntax-basierte Verfahren eingesetzt werden:

- Token-basierte Verfahren sind von linearer Komplexität, liefern aber Code-Fragmente, die nicht notwendigerweise einer syntaktischen Einheit entsprechen. Folglich sind die gefundenen Fragmente (z.B. gleiche Folgen von „end if"-Anweisungen) nicht immer sinnvolle Kandidaten für eine Faktorisierung.
- Syntaxbasierte Verfahren liefern genauere Ergebnisse, pro zu unterstützender Programmiersprache ist aber ein relativ hoher Aufwand zur Erstellung des Parser-Frontends notwendig.

3.1 Motivation im Kontext Reengineering

Redundanter Code, der durch Copy und Paste entstanden ist, soll wieder herausfaktorisiert und entfernt werden.

3.2 Motivation im Kontext Testen

Redundanter Code ist zwar nicht notwendig ein Fehler, er erhöht aber die Fehlerwahrscheinlichkeit bei Änderungen [3] und verursacht Mehraufwand beim Testen:

- Zum Erreichen der code-bezogenen (white-box) Testabdeckung muss aufgrund der größeren Code-Menge mehr getestet werden (insbesondere im Regressions-Test nach Änderungen), sodass der Zeitaufwand steigt.
- Die Fehlerisolation und -Beseitigung ist ebenfalls redundant notwendig.
- Der Testcode spiegelt die Redundanzen im eigentlichen Programm wieder und ist somit meist gleich stark oder stärker redundant als das zu testende System. Bei Änderungen oder Wartungsarbeiten am Code muss ggf. auch der Testcode an mehreren Stellen redundant geändert bzw. gewartet werden.

Das Reengineering-Verfahren zur Reduktion von redundantem Code kann dazu verwendet werden, den Testcode bei Beibehaltung der Abdeckung entsprechend zu reduzieren, und dadurch die Änderbarkeit des Testcodes zu verbessern.

4 Statische Rekonstruktion von Protokollen (Zustandsautomaten)

Für eine Instanz einer bestimmten Klasse wird durch statische Generierung von Objekt-Traces (Spuren) der Zustandsautomat generiert. Schwierig ist die Anwendung des Verfahrens u.A. bei dynamischen Heap-Strukturen.

4.1 Motivation im Kontext Reengineering

Die Bildung eines Zustandsmodells dient zum Verständnis der Programm-Funktionalität. Diese Technik ist insbesondere bei fehlender Programmspezifikation nützlich, und gehört deshalb zum Reverse-Engineering. Das Verständnis des Zustandsmodells ist aber auch für nicht-triviale Restrukturierungen des Codes wichtig.

4.2 Motivation im Kontext Testen

Dieses Verfahren kann beim zustandsbasierten Testen genutzt werden, um die Abdeckung des Zustandautomats durch die Testfälle zu ermitteln: Dazu wird für den Testcode ebenfalls der Zustandsautomat generiert und die Differenz betrachtet.

Überdeckungskennzahlen auf Modell-Ebene sind ebenso möglich, indem die Zustandsautomaten auf Modell- anstatt der Code-Ebene gebildet werden.

5 Dynamische Rekonstruktion von Protokollen

Der Code wird instrumentiert und durch Ausführung des Programms mit Testfällen ein Zustandsautomat generiert (dieses Verfahren wird *dynamische Generierung von Objekt-Traces bzw. Spuren* genannt).

5.1 Motivation im Kontext Reengineering

Durch statische Analyse können insbesondere objektorientierte Programme nicht vollständig statisch analysiert werden. Die „Aufzeichnung" von Zustandsautomaten, die den Lebenszyklus von mehreren Instanzen einer Klasse abbilden und deren anschließendes Zusammenfügen erleichtert aber die Rekonstruktion eines nicht dokumentierten Protokolls erheblich.

5.2 Motivation im Kontext Testen

Analog zum zustandsbasierten Komponententest kann dieses Verfahren zur Messung der Testüberdeckung von Protokollen im Integrationstest verwendet werden.

6 Offene Fragen

Die folgenden offenen Fragen wurden in der Diskussion identifiziert:

- Analyse von objektorientierten Systemen, die z.B. mit Java entwickelt wurden: Durch dynamisches Laden von Klassen und Reflection kann die statische Analyse wesentlich erschwert werden.
- Im Kontext „Rekonstruktion von Protokollen": Um Zustandsautomaten aus der Aufzeichnung unterschiedlicher „Objektlebensläufe" abzuleiten, müssen Szenarien ausgewählt und ausgeführt werden. Um eine bestimmte Funktionalität zu testen, müssen ebenfalls repräsentative Szenarien ausgewählt werden. Inwieweit unterscheiden sich die Szenarien, wo gibt es Überschneidungspunkte? Welche Unterschiede gibt es bei der Auswahl von „Test"-Szenarien?
- Qualität des Testcodes: Ist es sinnvoll und praktikabel, die Änderbarkeit des Testcodes (die i.d.R. viel schlechter ist als die des Anwendungscodes) durch Reengineering Verfahren zu verbessern?

7 Zusammenfassung

Das Reengineering hat von Testtechniken wie z.B. den ersten Ansätzen der Datenflussanalyse profitiert und diese weiterentwickelt – mehr noch: das Reengineering

setzt diese Techniken sogar erfolgreicher in der Praxis ein, als die Tester selbst. Die Tester können von dieser Weiterentwicklung profitieren, insbesondere bei der statischen Codeanalyse als immer wichtiger werdende Qualitätssicherungsmaßnahme. Klassische Fehlerrisiken, wie z.B. uninitialisierte Variablen, können heute mit hohem praktischem Nutzen automatisch ermittelt werden.

Um die Effizienz der Tester voranzubringen, ist die Nutzung der Reengineering-Techniken auch in neuen, zusätzlichen Teilbereichen, wie der automatischen Protokoll-Rekonstruktion, ein viel versprechender Ansatz.

Die eingangs erwähnte Umfrage des Arbeitskreises „Testen objektorientierter Programme“ hat beispielsweise bestätigt, dass die fehlerhafte Interpretation der Anforderungen das größte Fehlerrisiko darstellt. Eine statische bzw. dynamische Rekonstruktion von Protokollen aus dem Testgegenstand und ihrer Gegenüberstellung zum Sollverhalten könnte die Arbeit der Tester gerade in diesem Risikobereich gut unterstützen.

8 Acknowledgements

Vielen Dank an Lars Borner und Mario Winter für Review-Kommentare zu diesem Artikel.

9 Referenzen

[1] L. Borner, M. Hamburg und S. Jungmayr. Fehlerhäufigkeiten in objektorientierten Systemen: Ergebnisse einer Online-Umfrage. Softwaretechnik-Trends, Band 26, Nr. 1, Feb. 2006, S. 43-45.

[2] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. In ACM Transactions on Programming Languages and Systems, vol. 13, no. 4, Okt. 1991, S. 451—490.

[3] D. Nakae, T. Kamiya, A. Monden, H. Kato, S. Sato and K. Inoue. Quantitative Analysis of Cloned Code on Legacy Software. In Proceedings of IEICE SS2000-49, Vol. 100, No. 570, 2001, S. 57—64.