Erfahrungsberichte zur Lokalisierung von Produktfunktionen im Code

Raimund Klein, Rainer Koschke, Jochen Quante
Arbeitsgruppe Softwaretechnik
Fachbereich 3, Universität Bremen
http://www.informatik.uni-bremen.de/st/
{ray, koschke, quante}@informatik.uni-bremen.de

23. März 2005

Zusammenfassung

Wir beschreiben unsere Erfahrungen mit unserer Technik zur Lokalisierung von Code-Teilen, die eine gegebene Produktfunktion implementieren. Wir untersuchen den Informationsgewinn, der sich erzielen lässt, wenn man die dynamische Information auf Grundblockebene durchführt statt auf Funktionsebene, und welchen Preis man in Bezug auf Speicher- und Rechenbedarf der Begriffsanalyse hierfür bezahlen muss. Außerdem untersuchen wir den Einfluss der Abdeckung der Produktfunktionen (vollständige versus partielle versus redundante) auf die Ergebnisse.

1 Einführung

In vielen Fällen ist es Programmierern nicht bekannt, welche Teile ihres Codes eine bestimmte Produktfunktion bzw. eine Menge verwandter Produktfunktionen implementieren. Dieses Wissen ist jedoch für Änderungen unabdingbar.

Im Folgenden sei eine Produktfunktion eine in der Anforderungsspezifikation beschriebene Funktionalität, die die Software erfüllt. Wir nehmen hier ohne Beschränkung der Allgemeinheit den selten anzutreffenden Idealfall an, dass alle Anforderungen spezifiziert und implementiert sind; bei Systemen, für die diese Annahme nicht gilt, könnte man sich leicht ein solches fiktives Dokument vorstellen. Für eine Produktfunktion zählt, dass das System ein Verhalten zeigt, das ein Benutzer wahrnehmen kann und für ihn relevant ist.

Die relevanten Code-Teile, d.h. solche, die eine Produktfunktion implementieren, nennen wir schlicht Code. Dazu zählen sowohl einzelne Anweisungen (deklarativer oder ausführbarer Art) als auch Funktionen oder noch gröbere Einheiten wie Klassen.

2 Lokalisierung von Produktfunktionen

Es existiert eine Reihe von Techniken, um Produktfunktionen im Code zu lokalisieren. Sie lassen sich grob kategorisieren in solche, die nur statische bzw. nur dynamische Informationen heranziehen, und solche, die beide Informationsquellen ausnutzen. Dynamische Techniken finden ausführbaren relevanten Code, während statische Techniken auch deklarativen Code mit einbeziehen.

Zu den rein statischen Methoden gehört die Technik von Chen und Rajlich [1], bei der ein globaler Programmabhängigkeitsgraph systematisch vom Menschen untersucht wird. Ein Werkzeug unterstützt bei der Navigation. Ohne jedes Vorwissen ist diese Technik jedoch kaum anzuwenden, weil nicht klar ist, wo man mit der Suche beginnen soll und wie man sie fortsetzt. Eine statische Lösung zu diesem Problem liefern Marcus und Maletic, die mit Hilfe von Information-Retrieval-Ansätzen Korrespondenzen zwischen Spezifikationsdokumenten und Bezeichnern im Quellcode herstellen [4]. Diese können als Einstiegspunkt für die Navigation dienen. Eine Kombination der beiden Verfahren wurde von Zhao et al. vorgestellt [7].

Die rein dynamischen Analysen protokollieren beim Programmlauf ausgeführte Code-Teile. Diese Daten werden aufbereitet. Wilde und Scully [5] kategorisieren die ausgeführten Code-Teile für alle Testfälle, die eine bestimmte Produktfunktion f verwenden, wie folgt:

- *code commonly involved*: Code, der stets für alle Testfälle unabhängig von f ausgeführt wird
- *code potentially involved*: Code, der in mindestens einem Testfall, der f benutzt, ausgeführt wird
- *code indispensably involved*; Code, der für alle Testfälle, die f benutzen, ausgeführt wird
- *code uniquely involved*: Code, der in allen Testfällen, die f benutzen, ausgeführt wird, und nur in diesen

Ein ähnlicher Ansatz stammt von Wong et al. [6], der den ausgeführten Code wie folgt vergleicht:

1. Die *invoking input set* I ist die Menge aller Testfälle, die die Produktfunktion f benutzen.
2. Die *excluding input set* E ist die Menge aller Testfälle, die f nicht benutzen.

3. Das Programm wird zwei Mal ausgeführt, einmal für I und einmal für E.

4. Durch die Differenz der resultierenden Mengen ausgeführten Codes erhält man den Code, der für f spezifisch ist.

Unser eigener Ansatz führt die Ideen der dynamischen Analysen fort und kombiniert sie mit der statischen Analyse [2]. Zunächst betrachten wir hierzu nicht nur eine einzige Produktfunktion, sondern eine Menge von Produktfunktionen. Die einfache Differenzmengenbildung der bisherigen dynamischen Techniken, die nur zwei Mengen verglich, wird dann ersetzt durch die formale Begriffsanalyse [3], die durch eine kombinierte Differenzmengenbildung gemeinsame Code-Teile für Testfälle herausfaktorisiert. Die vom Benutzer beigestellte Information darüber, welche Testfälle welche Produktfunktionen verwenden, erlaubt es dann, die Korrespondenz zwischen Produktfunktionen und Code herzustellen. Die Information im Begriffsverband wird dann benutzt, um eine Suche im statischen Abhängigkeitsgraphen informiert durchführen zu können.

3 Die Untersuchung

Die wesentliche Annahme aller dynamischen Analysen ist, dass man in der Lage ist, Testfälle zu finden, die einerseits möglichst nur die relevante Produktfunktion und andererseits alle Varianten der Produktfunktion abdecken. Ob dies für eine gegebene Menge von Testfällen gelingt, kann man jedoch ohne detaillierte Kenntnis des Systems nicht garantieren. Dieses Problem ist uns vom Software-Test bereits bekannt.

Anhand verschiedener Compiler-Implementierungen haben wir deshalb untersucht, welchen Einfluss die Abdeckung der Produktfunktionen auf die Ergebnisse hat. Hierzu gehen wir zunächst aus von einer vollständigen Abdeckung der Produktfunktionen mit einer möglichst minimalen Menge von Testfällen. Die Testfälle sind hierbei so gestaltet, dass man durch Differenzbildung jede Produktfunktion einmal isolieren kann, eine Grundvoraussetzung, um den spezifischen Code mit Hilfe der Begriffsanalyse finden zu können. Dann gehen wir über zu einer redundanten Menge von Testfällen, die aber immer noch alle Produktfunktionen abdecken. Man sollte erwarten, dass hierdurch keine wesentlichen Verbesserungen mehr stattfinden. Schließlich reduzieren wir in umgekehrter Richtung die Menge der Testfälle, so dass wir noch immer eine für uns interessante Menge von Produktfunktionen isolieren können.

Bei der Erweiterung der Testfälle erhalten wir einen so genannten Superkontext, d.h. Begriffe des Originalverbands können sich aufspalten. Bei der Verminderung der Testfälle erhalten wir einen Subkontext, bei dem mehrere Begriffe des Originalverbands zusammenfallen können. Der Zuwachs bzw. der Verlust an Information lässt sich im verkürzten Begriffsverband an der Anzahl nicht-leerer Begriffe messen.

Des Weiteren untersuchen wir den Informationsgewinn, der sich erzielen lässt, wenn man die dynamische Information auf Grundblockebene durchführt statt auf Funktionsebene, und welchen Preis man in Bezug auf Speicher- und Rechenbedarf der Begriffsanalyse hierfür bezahlen muss. Auch hier lässt sich der Gewinn messen durch die Verfeinerung des resultierenden Verbandes.

Literatur

[1] Kunrong Chen and Václav Rajlich. Case study of feature location using dependence graph. In *IWPC*. IEEE Press, 2000.

[2] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating features in source code. *IEEE TSE*, 29(3), 2003.

[3] Bernhard Ganter and Rudolf Wille. *Formale Begriffsanalyse – Mathematische Grundlagen*. Number ISBN: 3-540-60868-0. Springer, 1996.

[4] Andrian Marcus and Jonathan I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *ICSE*, pages 125–134. IEEE Press, May 2003.

[5] Norman Wilde and Michael Scully. Software reconnaissance: Mapping from features to code. *Journal on Software Maintenance and Evolution*, 7:49–62, January 1995.

[6] W. Eric Wong, Swapna S. Gokhale, Joseph R. Horgan, and Kishor S. Trivedi. Locating Program Features using Execution Slices. In *Proc. of the IEEE Symposium on Application-Specific Systems and Software Engineering & Technology*, pages 194–203, Richardson, TX, USA, March 1999. IEEE Press.

[7] Wei Zhao, Lu Zhang, Yin Liu, Jiasu Sun, and Fuqing Yang. Sniafl: Towards a static non-interactive approach to feature location. In *ICSE*, pages 293–303. IEEE Press, 2004.