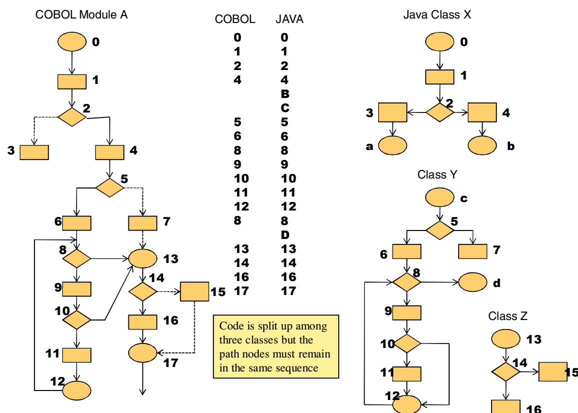
Validierung der funktionalen Äquivalenz konvertierter JAVA Programme durch einen dynamischen Source-Abgleich

Harry M. Sneed
ANECON GmbH, Wien

1. Die zugrunde liegende Theorie

Die Theorie, die diesem Testansatz zugrunde liegt besagt, unabhängig davon, wie die untersten Codebausteine umstrukturiert werden und statisch gegliedert sind, müssen sie dynamisch immer in der gleichen Reihenfolge ausgeführt werden, um funktional äquivalent zu sein. D.h. die COBOL Paragraphen können zerlegt und als Methoden auf diverse Java Klassen verteilt werden. Dennoch müssen die Java Methoden exakt in die gleichen Reihenfolge ausgeführt werden wie die ursprünglichen COBOL Paragraphen bzw. elementare Operationen. Darüber hinaus müssen auch die einzelnen Anweisungen in der gleichen Reihenfolge vorkommen. Es können zwar zusätzliche Methoden und Anweisungen dazwischen kommen, aber die aus dem COBOL Code übernommene Java Methoden müssen in der gleichen Folge vorkommen wie die alten COBOL Bausteine. Die Bedingungen müssen in der gleichen Weise erfüllt oder nicht erfüllt werden und die Schleifen müssen dieselbe Anzahl Wiederholungen haben. (*siehe Abb. 1: Reihung der elementaren Operationen)*



Jegliche Abweichung in der Ausführungsfolge der Grundbausteine deutet auf einen Fehler in der Transformation hin. Demzufolge muss versucht werden, die beiden Komponente - das COBOL Programm und das Java Paket – parallel zu einander ohne Daten auszuführen und die Ablaufpfade miteinander zu vergleichen. Die Methode hierfür heißt symbolische Analyse und geht zurück auf die Forschung von W. Howden in den 70-er Jahren. Danach wird das Programm mit künstlichen Daten ausgeführt. Der Tester steuert den Ablauf durch den Code in dem er bei allen Verzweigungen bestimmt, wie es weitergeht - z.B. wenn er zu einer Alternativanweisung kommt, bestimmt der Tester ob die Bedingung erfüllt oder nicht e4rfüllt wird und wenn es zu einer Schleife kommt, bestimmt er, wann die Schleife beendet wird. Auf dieser Weise werden die unterschiedlichen Pfade durch den Code aufgezeichnet. In der ursprünglichen Forschung von Howden wurden diese Pfade mit dem spezifizierten Pfade abgeglichen um die Korrektheit der Programmlogik zu verifizieren.

2. Aufbau der Anweisungstabellen

Bei der Verifikation konvertierter Komponente wird nicht mit einer Spezifikation, sondern mit den Ablaufpfaden des ursprünglichen Codes verglichen. Der Tester prüft die symbolische Ausführung der beiden Codeversionen – von der COBOL Version wie auch von der Java Version – eine interne Anweisungstabelle generiert. Die Anweisungen sind darin mit ihrer Zeilennummern gekennzeichnet. Neben die Zeilennummer der jeweiligen Anweisung deutet eine zweite Anweisungsnummer auf die folgende Anweisung. Im Falle einer Alternativanweisung gibt es zwei potentielle Nachfolgeranweisungen. Im Falle einer Schleife gibt es die erste Anweisung in der Schleife und die erste Anweisung hinter der Schleife. Im Falle einer Fallanweisung kann es *n* potentielle Nachfolgeranweisungen geben.
Eine dritte Zeilennummer verbindet die Anweisung in dieser Version mit der entsprechenden Anweisung in der anderen. In der Java Tabelle wird auf die COBOL Zeile oder Zeilen hingewiesen, aus der die Java Anweisung stammt. In der COBOL Tabelle wird umgekehrt auf die Java Zeile bzw. Zeilen hingewiesen, die aus dieser COBOL Anweisung abgeleitet wurden. So sind die beiden Tabellen rückwärts und vorwärts miteinander gekettet. (siehe Sample 1: Statement Table)

Neben den zwei Anweisungstabellen wird eine Aufrufstabelle erzeugt, in der in COBOL alle PERFORM und CALL Aufrufe und in JAVA alle Methodenaufrufe eingetragen sind. Sollte in Java die gleiche Methode in mehreren Klassen vorkommen – polymorphy – kann der Tester die gewünschte Klasse auswählen. (Siehe Sample2: Method Table)

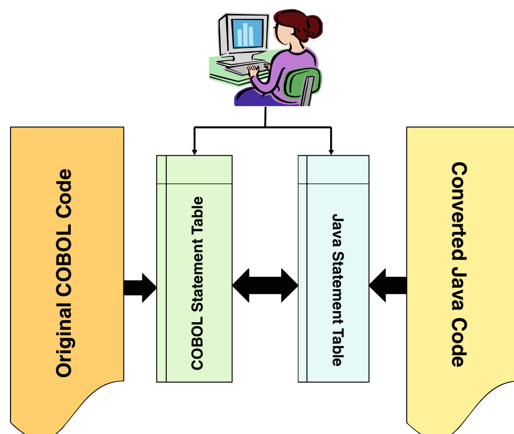
Außer den zwei Anweisungstabellen werden auch zwei Datentabellen generiert, in denen alle globalen und lokalen Variabel in COBOL wie auch in Java registriert sind. Zu jeder Variabel kommt eine Liste der Zeilennummer der Anweisungen, in denen diese Variabel referenziert wird, zusammen mit einem

Kennzeichen für die Art der Verwendung – als Argument, Ergebnis oder als Prädikat bzw. Bedingungsoperanden. Diese Referenzen werden bei der Ausführung der symbolischen Analyse benutzt im Datenflusspfade zu bilden. (seine Sample 3: Data Table)

3. Ausführung der symbolische Analyse

Nach dem Aufbau der Anweisungs- und Datentabellen beginnt der Tester mit der Nach dem Aufbau der Anweisungs- und Datentabellen beginnt der Tester mit der symbolischen Analyse. Als erstes steuert er ausgewählte Pfade durch den Java Code hindurch. Dabei wird ein Trace-Protokoll der durchlaufenen Java Anweisungen sowie ein Trace-Protokoll der referenzierten Java Daten erstellt. Das erste Protokoll dokumentiert den Steuerungsfluss, das zweite Dokument den Datenfluss.

Anschließend wiederholt der Tester die Ausführung der gleichen Pfade durch den COBOL Code. Dabei werden die gleichen zwei Protokolle für die COBOL Version erstellt. (siehe Abb. 2: Symbolische Analyse der beiden Source-Versionen)



Zum Schluss werden die Java Trace-Protokolle mit dem COBOL Trace-Protokollen automatisch abgeglichen. Die Anweisungen sollten in beiden Fällen in der gleichen Reihenfolge kommen. Wenn nicht, oder wenn eine Anweisung fehlt, erfolgt eine Fehlermeldung. Bei den Daten-Trace-Protokollen wird geprüft, ob die Datenreferenzen in der gleichen Reihenfolge erscheinen und ob sie von der gleichen Verwendungsart sind. Wenn nicht, erfolgt auch hier eine Fehlermeldung. Es werden auf diese Weise zweierlei Fehlerarten in der Konversion aufgezeigt:

- Fehler bei der Ablaufsteuerung und
- Fehler beim Datenfluss.

4. Erfahrung und Weiterführung

Dieser Testansatz den Vorteil, dass er die Auflösung des Programmcodes simuliert, ohne den Overhead einer aufwendigen Testumgebung – und das bei einem relativ geringen Aufwand. Der einzige manuelle Aufwand ist die Steuerung der Ablaufpfade durch den Tester, bzw. der Reengineer. Dies hat aber auch ein nützliches Seiteneffekt. Es zwingt den Reengineer sich mit dem Code auseinander zu setzen. dadurch stößt er unweigerlich auf weitere Anomalien im Programmverhalten.

Die Fehler die hier auftauchen, sind fasst alle auf Fehler in dem Konversionswerkzeug zurückzuführen. Meistens sind es ungewöhnliche Konstrukte in COBOL, wie der SEARCH Befehl, die vom Werkzeug nicht richtig behandelt wurden. Wenn diese Konstrukte mehrmals vorkommen, muss die gesamte Codetransformation wiederholt werden. Wenn sie nur in einem oder zwei Sourcen vorkommen, ist es besser sie in dem Java Code manuell nachzubessern.

Der Test des konvertierten Systems bleibt weiterhin der Hauptkostentreiber in allen Migrationsprojekten. Alles, was dazu beitragen kann, den Testaufwand zu reduzieren, ist vom Nutzen. Dieses Verfahren verspricht mindestens einen Teil der Fehler zu entfernen, eher es zum großen Regressionstest kommt. Damit wird der System entlastet, da der Aufwand, den vielen gewöhnlichen Umsetzungsfehlern nachzugehen, fällt weg. Es bleibt genug zu tun mit den Kompatibilitätsfehlern.

Literaturhinweise:

[1] Sneed, H., Wolf, E., Heilmann, H.: “Software Migration in der Praxis”, dpunkt.verlag, Heidelberg, 2010.

[2] Sneed, H.: “ Migrating PL/I Code to Java”, IEEE Proc. of 15th CSMR, IEEE Computer Society Press, Oldenburg, p. 287

[3] Broy, M.: “Zur Spezifikation von Programmen für die Textverarbeitung”, in Wossido, P. (ed.): Textverarbeitung und Informatik, Informatik Fachberichte 30, Springer Verlag, Heidelberg, 1980, p. 75

[4] Howden, W.: "Symbolic Testing with the DISSECT Symbolic Evaluation System" in IEEE Trans. on S.E. Vol. 1, No. 4, July 1977, p. 266

[5] Parrish,A./Borie,R./Cordes,D.: “Automated Flow Graph-based Testing of Object-oriented Software Modules” Journal of Systems and Software, Vol. 23, No. 1,1993, pp. 95-109

[6] Sneed, H.: “Validating Functional Equivalence of reengineered Programs via Control Path, Result and Data Flow Comparison”, Software Testing, Verification, & Reliability, Vol. 4, No. 1, March 1994, p. 33

[7] Howden, W.: "Reliability of the Path Analysis Testing Strategy”, IEEE Trans. on S.E. Vol. 1, No. 4, Sept. 1976, p. 208

[8] Peters, D., Parnas, D.: “Using Test Oracles generated from Program Documentation”, IEEE trans. On S.E., Vol. 24, No. 3, March 1998, p. 161

[9] Sneed, H.: “Source Animation as a means of Program Comprehension for object-oriented Systems” Proc. of 8th IEEE International Workshop on Program Comprehension, Computer Society Press, Limerick, June 2000, p. 179

[10] Cornelissen, B., Zaidman, A., van Deursen, A.: „A controlled Experiment for Program Comprehension through Trace Visualization“, IEEE Trans. on S.E., Vol. 37, No. 3, p. 341