Die Ableitung des Verhaltens einzelner Objekte aus Quellcode

Dominik Rauner-Reithmayer

Institut für Informatik-Systeme, Universität Klagenfurt, Österreich
E-mail:dominik@isys.uni-klu.ac.at

Zusammenfassung

Wird im Reverse Engineering oder auch im Forward Engineering die Dynamik eines Systems betrachtet geschieht dies meist durch die Beschreibung des Verhaltens von Klassen. Verwendet man jedoch diese Modelle zur Wartung einzelner Objekte, so ist eine Beschreibung auf Klassen-Niveau in vielen Fällen zu ungenau. In dieser Arbeit wird nun ein Ansatz vorgestellt, der aus der Verhaltens-Beschreibung einer Klasse und aus im Quellcode enthaltenen Informationen jenes Verhaltens-Modell ableitet, das dem Verhalten des betrachteten Objekts entspricht.

1. Einleitung

Im Software Reverse Engineering findet man eine Fülle von Ansätzen, die es ermöglichen, aus prozeduralem Quellcode durch Betrachtung von strukturellen- ([2, 3, 5, 9, …]) , funktionalen ([1, 4, 7, 10, …]) und Kontroll-Aspekten [8] funktional äquivalenten objekt-orientierten Quellcode abzuleiten. Die dabei entstehenden Abstraktionen seien es Statik- und/oder Dynamik-Modelle oder die funktionale Beschreibung einzelner Methoden sind sehr hilfreich, wenn die abgeleiten Klassen in einem gleichen oder auch in unterschiedlichem Kontext wiederverwendet werden. Betrachtet man jedoch speziell den Kontroll-Aspekt aus der Sicht der Wartung, so sind die erzeugten Dynamik-Modelle nur dann sinnvoll, wenn die Wartungstätigkeiten alle Objekte dieser Klasse betreffen sollen. Liegt jedoch der Betrachtungspunkt der Wartung nicht auf dem Verhalten aller Objekte dieser Klasse sondern auf einzelnen Objekten dieser Klasse, dies ist z.B. dann der Fall, wenn einzelne Objekte auf Ereignisse scheinbar nicht korrekt reagieren und sich somit in einem falschen Zustand befinden oder wenn das Verhaltens-Repertoire der Klasse zu umfangreich ist. Dann wird es notwendig, neben den Verhaltens-Modell der Klasse, das alle möglichen Zustände, Ereignisse und Zustandübergänge repräsentiert, ein Verhaltens-Modell zu verwenden, das nur mehr das Verhalten der betrachteten Objekte beinhaltet.

Zu diesem Zweck ist es notwendig, das Verhaltens-Modell der Klasse, auf jene Einreignisse und die dadurch eingeleiteten Zustandsübergänge und Zustände einzuschränken, die das betrachtete Objekt direkt betreffen. Um dies zu ermöglichen wird im folgenden eine Ereignisfolge definiert und gezeigt, wie mittels dieser Ereignisfolge das Verhaltens-Modell der Klasse auf das Verhaltens-Modell des Objekts eingeschränkt werden kann.

2. Dynamik einzelner Objekte auf Quellcode Niveau

2.1. Ereignisfolgen

Betrachtet man Methodenaufrufe als Ereignisse für ein Objekt, so ist es möglich, alle auf ein Objekt wirkende Methodenaufrufe in ihrer zeitlichen Reihenfolge darszustellen. Diese Folge von Aufrufen, die im folgenden Ereignisfolge genannt wird, stellt die an ein Objekt gesendeten Ereignisse dar und beschreibt somit einen mehr oder weniger typischen Lebenszyklus für genau dieses eine Objekt.

Für das Bestimmen von Ereignisfolgen stehen prinzipiell zwei Methoden zur Verfügung:

1. Die Ereignisfolge wird mit Information aus der Programmverfolgung erstellt (event tracing).

2. Die Ereignisfolge wird mittels statischer Analyse aus dem Quellcode abgeleitet.

Die erste Variante ist sicher die genaueste, da sie für ein Objekt genau eine gültige Ereignisfolge liefert, hat aber den Nachteil, dass man für eine umfassende Beschreibung des Objektes eine Unzahl von Ereignisfolgen und dafür eine ebenso hohe Anzahl von Programmdurchläufen benötigt. Daher kann davon ausgegangen werden, dass diese Variante zur Bestimmung von Ereignisfolgen sehr teuer kommt und daher die Einstzbarkeit nur in sehr wenigen speziellen Fällen gerechtfertigt sein wird.

Die zweite Variante hat den Vorteil, dass man eine umfassende Beschreibung des Objektes relativ einfach aus dem Quellcode ableiten kann, allerdings mit dem Nachteil etwas an Genauigkeit zu verlieren.

Als Darstellung für alle möglichen Ereignisfolge bietet sich daher ein Kontrollflussgraph an, in dem sowohl die Ereignisse als auch die Kontrollabhängigkeit und damit ihre Reihenfolge dargestellt werden kann.

> **Definition 1 (Kontrollflussgraph):** Ein Kontrollflussgraph CFG für ein Programm P ist ein vier-Tupel (V, A, En, Ex), wobei V die Menge der Knoten, A die Menge der gerichteten Kanten zwischen zwei Knoten und En und Ex den Eintritts-und Austritts-Knoten des Graphen darstellen. Eine Kante von einem Knoten v_i zu einem Knoten v_j (mit $v_i \in V \wedge v_j \in V \wedge (v_i, v_j) \in A$) bedeutet, dass während der Programmausführung die Kontrolle vom Knoten v_i auf den Knoten v_j übergehen kann.

In einem normalen CFG repräsentieren die einzelnen Knoten beliebige Anweisungen oder Bedingungen. Für den hier betrachteten Zweck ist es ausreichend, wenn die Knoten entweder Methodenaufrufe des untersuchten Objekts der Klasse oder Bedingungen darstellen.

2.2. Die Einschränkung eines Verhaltens-Modells mittels Ereignisfolgen

Um ein Verhaltens-Modell mittels einer Ereignisfolge einschränken zu können, ist es notwendig es etwas formaler zu definieren. Aufgrund der Ähnlichkeit zwischen Dynamik-Modellen (UML Zustandsdiagramme) und endlichen Automaten bietet sich dafür, ein auf der Definition von endlichen Automaten basierender Zustand-Ereignis Automat an.

Definition 2 (Zustand-Ereignis Automat): Ein Zustand-Ereignis Automat (ZEA) für eine Klasse K ist ein Quintupel (Z, E, δ, z_0, Z_e), wobei Z eine endliche Menge von disjunkten Zuständen, E die endliche Menge der auf ein Objekt der Klasse K einwirkenden Ereignisse (Methoden), δ die Zustandsübergangsfunktion, die von $Z \times E \times B$ auf eine Menge von Zuständen abbildet, wobei B die Menge der auf den Attributen von K definierbaren Bedingungen ist, z_0 ($z_0 \in Z$) der Anfangszustand und Z_e ($Z_e \subseteq Z$) die Menge der Endzustände darstellt.

Mit dem oben definierten Zustand-Ereignis Automat ist es möglich, das Verhalten einer Klasse formal zu beschreiben, in Bezug auf UML Zustandsdiagramme betehen jedoch einige Unterschiede die bei der Betrachtung von Zustand-Ereignis Automten berücksichtigt werden müssen:

1. Die Zustände im Zustand-Ereignis Automat sind alle disjunkt, daher können die aus UML bekannten Konzepte des sequential composite states und des concurrent composite state nicht direkt abgebildet werden.

2. Das Gleichsetzen von Methoden und Ereignissen kann auch problematisch sein, da mit einem Ereignis üblicherweise eine zeitlose Ein-Weg Übermittlung von Information von einem Objekt zu einem anderen assoziert wird, und dies bei Methodenaufrufen nicht garantiert ist.

Trotz dieser Einschränkungen ist es sinnvoll, das Verhalten von Objekten einer Klasse mit Zustand-Ereignis Automaten zu beschreiben, da diese Abstraktion einerseits aus dem Quellcode ableitbar ist und andererseits dem SW-Entwickler die Möglichkeit gegeben wird, mit einer Darstellung zu arbeiten, die ihm aus der Modellierung bekannt ist.

Ausgehend von der formalen Beschreibung des Verhaltens-Modells einer Klasse K mittels eines Zustand-Ereignis Automaten und einer Ereignisfolge für ein Objekt dieser Klasse kann man jenen Zustand-Ereignis Automaten erzeugen, der genau das Verhalten des betrachteten Objekts beschreibt. Diese Generierung basiert im wesentlichem auf dem Gedanken, dass der genau ein Objekt beschreibende Zustand-Ereignis Automat ein Teilautomat des eine Klasse beschreibenden Zustand-Ereignis Automats sein muss. Das bedeutet, dass in dem entstehendem Automaten nur Zustände und Zustandsübergänge vorhanden sein dürfen, die schon im Zustand-Ereignis Automaten der Klasse vorhanden waren. In Abbildung 1 ist ein iterativer Algorithmus angegeben, der aus einem Zustand-Ereignis Automat einer Klasse K ($ZEA_K = (S_K, E_K, \delta_K, q_{K_0}, q_{K_{final}})$) und des Kontrollflussgraphen für ein Objekt o dieser Klasse ($CFG_o = (V_{CFG}, A_{VFG}, En, Ex)$) jenen Zustand-Ereignis Automaten ($ZEA_o = (S_o, E_o, \delta_o, q_{o_0}, q_{o_{final)}}$) ableitet, der alle aus dem CFG_o generierbaren und mit dem ZEA_K akzeptierbaren Ereignisse, akzeptiert. Der Algorithmus setzt dazu voraus, dass für jeden Knoten im CFG_o die zwei boolschen Funktionen *Event* und *Condition* und das Attribut *name* definiert sind. Die zwei Funktionen liefern dabei abhängig davon ob der Knoten einen das Ereignis betreffenden Methodenaufruf (Event) oder eine Bedingung (Condition) repräsentiert den entsprechenden Wahrheitswert als Ergebnis und das Attribut *name* beinhaltet den Namen der aufgerufenen Methode, der gleichzeitig ein Ereignis im ZEA_K darstellt.

Die Idee hinter dem Algorithmus, ist die Abarbeitung der aus dem Kontrollflussgraphen ableitbaren Ereignisfolgen im ZEA_K zu simulieren und jene Zustandübergänge in den ZEA_o aufzunehmen, die bei der Simulation verwendet wurden. Dazu wird versucht, ausgehend vom Startknoten

function *restrict* $((S_K, E_K, \delta_K, q_{K_0}, q_{K_{final}}), (V_{CFG}, A_{VFG}, En, Ex)) :$
$(S_o, E_o, \delta_o, q_{o_0}, q_{o_{final}})$

$\delta_o = \emptyset$

$q_{o_0} = q_{K_0}$

$V = \emptyset$

$toV = \{(En, q_{K_0})\}$

repeat

$V = toV \cup V$

$toV' = \{(n_t, q_t) \mid$

$\exists (n_f, q_f) \in toV \bullet (n_f, n_t) \in A_{CFG} \wedge$

$(Event(n_t) \wedge (\exists ((q, e, b), ss) \in \delta_K \mid q_f = q \wedge e = n_t.name \bullet q_t \in ss))$

$\vee (Condition(n_t) \wedge q_f = q_t)\}$

$\delta_o = \delta_o \cup \{((q_o, e_o, b_o), ss_o) \in \delta_K \mid$

$\exists (n_f, q_f) \in toV \mid q_o = q_f \bullet$

$\forall (n_t, q_t) \in toV' \mid (n_f, n_t) \in A_{CFG} \wedge Event(n_t) \wedge e_o = n_t.name \bullet q_t \in ss_o\}$

$toV = toV'$

until $(toV \subseteq V)$

$S_o = \{q \mid \exists ((q_f, e, b), ss) \in \delta_o \bullet q = q_f \vee q \in ss\}$

$E_o = \{event \mid \exists ((q_f, e, b), ss) \in \delta_o \bullet e = event$

$q_{o_{final}} = S_o \cap q_{K_{final}}$

end function

Abbildung 1. Algoithmus zur Einschränkung eins ZEA mittels eines CFGs

En des CFG_o und dem Startzustand q_{K_0} des ZEA_K jeden Knoten des CFG_o mit einem Zustand aus dem ZEA_K, abhängig davon ob es einen entsprechenden Übergang im ZEA_K und im CFG_o gibt, zu assozieren und die entsprechenden Übergänge in der Übergangsfunktion des ZEA_o zu sammeln. Der Algorithmus endet, wenn keine neuen Zustand-Knoten Paare mehr erzeugt werden können.

Dies kann, neben dem erwarteten Ergebnis zu folgenden Fällen führen:

- In dem ZEA_o sind Zustände und Zustandsübergänge enthalten, die nicht zu einem Endzustand führen.
  Dies geschieht dadurch, dass bei einem nicht-deterministischen ZEA_K bei der Simulation der Ereignis-Folgen alle möglichen Folgezustände betrachtet werden und die entsprechenden Zustände und Zustandsübergänge in den ZEA_o aufgenommen werden.

Diese Zustände und die ihnen zugeordneten Zustandsübergänge können, entsprechend der z.B. in [6] angegebenen Methode zur Minimierung endlicher Automaten, entfernt werden ohne die Menge der akzeptierten Ereignis-Folgen zu verändern. Die Minimierung kann jedoch bei der Interpretation Auswirkungen haben, da einerseits bei der Minimierung nur auf das Akzeptieren von Ereignis-Folgen Wert gelegt wird und die jeweilige Semantik der Zustände nicht berücksichtigt wird. Andererseits können die nicht notwendigen Zustände und Zustandsübergänge bei der Fehlersuche oder bei dem Versuch das Objektverhalten zu verstehen sehr hilfreich sein.

- Der entstehende Zustand-Ereignis Automat ZEA_o enthält keine Endzustände.
  Dies kann im wesentlichen zwei Ursachen haben:

  1. Im CFG_o ist kein entsprechendes Zerstörungsereignis enthalten
     Vor allem in Software, bei der nachträglich Objekte und Methoden identifiziert worden sind, ist es möglich, dass keine Methoden für die Objekt-Zerstörung berücksichtigt wurden. In diesem Fall ist es notwendig eben jene Ereignisse sowohl im Kontrollflussgraphen des konkreten Objekts als auch im Zustand-Ereignis Automaten zu berücksichtigen, bzw. hinzuzufügen.

  2. Es existieren Zustandsänderungen außerhalb von Methoden.
     Dieser Umstand sollte in objektorientierter Software nicht vorhanden sein, trotzdem lässt es sich nicht ausschließen, dass Werte von Zustands-Indikatoren unabhängig von Methoden des Objekts geändert werden und es dadurch zu Veränderungen des Zustands ohne einen Methoden-Aufruf kommt.

Die so erzeugten Automaten können eingesetzt werden, um ein besseres Verständnis und dadurch eine leichtere Wartbarkeit des Objekts zu erreichen. Wesentlich bei allen Interpretationen und der Verwendung des Automaten sind die oben angeführten Spezialfälle, da meist genau in diesen Spezialfällen die Ursache für ein Fehlverhalten des Objekts zu finden ist.

Neben dem Nutzen für Wartung und Verständnis können die für Objekte einer Klasse erzeugten Verhaltens-Modell auch Rückschlüsse auf die Klasse selbst zulassen. Eine Bewertung und genauere Untersuchung der Klasse wird vor allem dann notwendig sein, wenn Verhaltens-Modelle von mehreren Objekten dieser Klasse nur sehr wenige Gemeinsamkeiten haben, seien dies nun Zustände, Zustandsübergänge oder Ereignisse. In diesen Fällen kann davon ausgegangen werden kann, dass von dieser Klasse mehr als ein Verhaltens-Modell implementiert wird und daher eine nähere Betrachtung dieser Klasse hinreichend begründet scheint.

3. Zusammenfassung

Der oben kurz skizzierte Ansatz zeigt, dass es möglich ist von detailliertem Quellcode und einem Verhaltens-Modell unter Zuhilfenahme von statischer Programmanalyse (Kontrollflussgraph), ein Zustand-Ereignis Diagramm für ein Objekt einer speziellen Klasse abzuleiten, das sowohl als Dokumentation als auch als Basis für Wartungstätigkeiten, die nur das betrachtete Objekt betreffen, dienen kann.

Literaturverzeichnis

[1] Ilene Burnstein, Abdul Mirza, Katherine Roberson, Floyd Saner and Abdallah Roberson. KNOWLEDGE ENGINEERING FOR AUTOMATED PROGRAM RECOGNITION AND FAULT LOCALIZATION. In *Proceedings of the 8th International Conference on Software Engineering and Knowledge Engineering SEKE 96*, Seiten 85–91, Lake Tahoe (Nevada), June 1996.

[2] G. Canfora and A.Cimitile. An Improved Algorithm for Identifying Objects in Code. *Software Practice and Experience* **26** (1), 25–48 (January 1996).

[3] Doris L. Carver. Reverse engineering Procedural Code for Object Recovery. In *Proceedings of the 8th International Conference on Software Engineering and Knowledge Engineering SEKE 96*, Seiten 442–449, Lake Tahoe (Nevada), June 1996.

[4] Anielo Cimitile, Andrea De Lucia and Malcom Munro. A Specification Driven Slicing Process for Identifying Reusable Functions. *Software Maintenance: Research and Practice* **8**, 145–178 (1996).

[5] Harald Gall and René Klösch. *Objektorientiertes Reverse Engineering.* Springer–Verlag, Berlin, Heidelberg, 1995.

[6] John E. Hopcroft und Jeffrey Ullman. *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie.* Addison-Wesley, Bonn, Paris, Mass.. 3. Auflage, 1994.

[7] K. A. Kontogiannis, R. Demori, M. Galler and M. Bernstein. Pattern Matching for Clone and Concept Detection. *Automated Software Engineering* **3**, 77–108 (1996).

[8] Dominik Rauner-Reithmayer and Roland Mittermeir. Behavior Abstraction to support Reverse Engineering. In *Proceedings of the 10th International Conference on Software Engineering and Knowledge Engineering SEKE 98*, San Francisco, USA, June 1998.

[9] Harry M. Sneed. Migration of Procedural Oriented COBOL Programs in an Object-Oriented Architecture. In *Proceedings of the International Conference on Software Maintenance 1992*, Seiten 105–112, Orlando, Florida, November 1992.

[10] Mark Weiser. Program Slicing. *IEEE Transactions on Software Engineering* **10** (4), 352–357 (July 1984).