Transformation und Vergleich von endlichen Automaten zur Analyse von Software-Protokollen

Gunther Vogel
Institut für Softwaretechnologie, Universität Stuttgart
Universitätsstraße 38, D-70569 Stuttgart

Abstract: Der Artikel beschreibt, wie endliche Automaten für Software-Protokolle aus dem Quelltext gewonnen und durch Transformationen für die Weiterverarbeitung und den Vergleich aufbereitet werden können. Die beschriebenen Techniken werden zur Prüfung von Protokollen oder zur Herleitung von Protokollspezifikationen eingesetzt. Messergebnisse zeigen die Praktikabilität des Verfahrens.

1 Einführung

Software-Protokolle geben Regeln und Konventionen für Programmabläufe vor. Ein wichtiger Bestandteil eines Protokolls sind Beschränkungen der Ausführungsreihenfolgen. Beispielsweise fordert das Protokoll für eine einzelne Variable, dass sie erst einen Wert zugewiesen bekommt, bevor sie ausgelesen wird. Ebenso schränken Protokolle von Software-Komponenten die erlaubten Reihenfolgen für die Aufrufe von Funktionen ein. Beispielsweise kann das Protokoll fordern, dass vor der Verwendung eine Initialisierung und am Ende eine Finalisierung stattfindet. Da solche Regeln nur selten ausreichend dokumentiert werden, findet in der letzten Zeit vermehrt der Versuch statt, Verwendungen von Protokollen durch dynamische und statische Analysen wiederzugewinnen [WML02, EKV05, QK06]. Spurgraphen sind dabei eine Möglichkeit, die zur Laufzeit auftretenden Abläufe bezüglich eines bestimmten Belangs zu repräsentieren. Sie erlauben jedoch nur wenige Operationen und lassen sich nicht direkt mit einer Spezifikation vergleichen.

Dieser Artikel beschreibt deshalb, wie ein Spurgraph in einen endlichen Automaten transformiert und dieser für weitergehende Operationen, wie Vereinigung, Differenz oder Vergleich aufbereitet werden kann.

1.1 Anwendungen

Bevor die eigentlichen Transformationen zur statischen Herleitung der endlichen Automaten aus dem Quelltext beschrieben werden, sollen zunächst zwei Anwendungsszenarien skizziert werden, bei denen ein Vergleich bzw. eine Differenzbildung zwischen Protokollen durchgeführt wird.

Prüfung einer Protokollverwendung Eine Protokollspezifikation beschreibt formal die Sequenzbeschränkungen, die für eine korrekte Verwendung einer Komponente gelten müssen. Ist die Spezifikation bekannt, so kann der aus dem Quelltext hergeleitete Automat A_V für die Verwendungen mit dem Automaten A_S für die Spezifikation verglichen werden. Ist der zugehörige Differenz-Automat $A_D = A_v - A_S$ nicht leer, so wurde eine Abweichung gegenüber der Spezifikation erkannt. Man erhält somit einen Hinweis auf einen potentiellen Fehler. Ob dieser während einer Programmausführung tatsächlich auftritt oder es sich um eine Überschätzung durch die statische Analyse handelt, muss vor einer Korrektur noch genauer untersucht werden.

Herleitung einer Protokollspezifikation Leider steht nur in seltenen Fällen eine ausreichende Spezifikation zur Protokollprüfung zur Verfügung. Abbildung 1 zeigt ein Anwendungsszenario zur Herleitung dieser Protokollspezifikation. Ausgangspunkt ist eine hypothetische Protokollspezifikation, die das bereits vorhandene Wissen über das Protokoll enthält. Diese ist aber möglicherweise unvollständig oder kann auch leer sein kann. Wird beim Vergleich der Protokollspezifikation mit einer (korrekten) Verwendungen des Protokolls eine Abweichung erkannt, so ist dies ein Hinweis auf eine unvollständige Spezifikation. Der Differenz-Automat $A_D = A_v - A_S$ beschreibt nun wieder die Abweichungen. Diese müssen zunächst noch auf ihre Korrektheit überprüft werden und können dann zur Spezifikation hinzugefügt werden. Diese manuelle Prüfung ist jedoch unbedingt notwendig, da keine fehlerhaften Ausführungen in die Spezifikation übernommen werden dürfen.

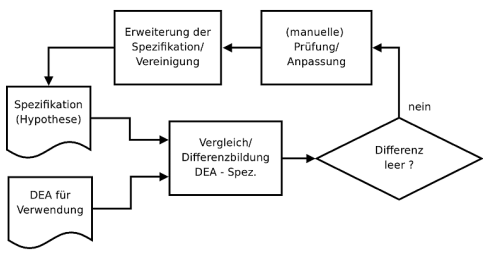

Abbildung 1: Anwendungsszenarien zur Herleitung einer Protokollspezifikation

1.2 Statische Extraktion von endlichen Automaten für Protokollverwendungen

Zur Extraktion eines endlichen Automaten werden die folgenden Schritte durchgeführt: (1) Extraktion eines Spurgraphen für das gewünschte Kriterium (bspw. ein Objekt oder eine Komponente), (2) Transformation des Spurgraphen in einen nicht-deterministischen endlichen Automaten (NEA), (3) Berechnung eines deterministischen endlichen Automaten (DEA), (4) Vergleich bzw. Weiterverarbeitung im Kontext der Anwendung. In diesem Prozess kommt es in Schritt (1) und (2) zu Überschätzungen durch die statische Analyse. In den weiteren Schritten tritt kein Informationsverlust auf – es sei denn, der Benutzer beschneidet manuell die Ergebnisse. Dies geschieht beispielsweise bei der Wiedergewinnung der Protokollspezifikation.

Die Determinierung des Automaten ist für Operationen wie Komplementbildung und Vergleich notwendig und erhöht zusätzlich die Verständlichkeit für den Benutzer. Im schlimmsten Fall werden jedoch exponentiell viele Zustände erzeugt. Dies tritt jedoch in der Praxis äußerst selten auf und konnte in den durchgeführten Tests nie beobachtet werden.

2 Spurgraphen

Die Analyse einer Protokollverwendung beginnt mit der Extraktion eines Spurgraphen. Bei diesem handelt es sich um einen interprozeduralen Kontrollflussgraphen, der die dynamischen Aspekte des verwendeten Protokolls repräsentiert. Spurgraphen enthalten die primitiven Aktionen des Protokolls, wie das Lesen und Schreiben von Variablen oder den Aufruf von primitiven Funktionen eines ADTs. Condition- und Join-Knoten stellen Verzweigungen und Zusammenfluss dar. Aufrufe werden durch Call-Knoten modelliert. Entry- und Exit-Knoten repräsentieren den Ein- und Austritt aus einem Unterprogramm. Die Ausführung beginnt bei einem Start- und endet mit einem Final-Knoten.

Kontrollfluss wird durch Kanten repräsentiert, wobei drei Arten unterschieden werden: lokale Kanten für den Kontrollfluss innerhalb einer Funktion, Call-Kanten zwischen Call-Knoten und Entry-Knoten für den Aufruf einer Funktion sowie Return-Kanten zwischen Exit- und Call-Knoten für die Rückkehr zum Aufrufer.

In [EKV05] wurde die Extraktion statischer Objektspurgraphen beschrieben, die alle Aktionen bzgl. eines bestimmen Objekts sowie alle weiteren Knoten des Kontrollflussgraphen, die für die Ausführung dieser Aktionen relevant sind, enthalten. Ein Spurgraph wird durch eine Transformation aus dem ursprünglichen Kontrollflussgraphen des Programms gewonnen, enthält jedoch weitaus weniger Knoten, da die irrelevanten Teile wegfallen.

Eine Spur des Spurgraphen ist ein Pfad vom Start- zu einem Final-Knoten bei dem die Paarung der Call- und Return-Kanten beachtet wird. Die Menge der Sequenzen von primitiven Aktionen, die in sämtlichen Spuren auftreten, wird im Folgenden als die Sprache des Spurgraphen bezeichnet.

3 Transformation des Spurgraphen in einen NEA

Ziel dieser Transformation ist die Erstellung eines endlichen Automaten, der mindestens die Sprache des Spurgraphen erkennt. Die Paarung von Call- und Return-Kanten kann jedoch nicht durch einen endlichen Automaten dargestellt werden, sodass auch nicht-realisierbare Spuren berücksichtigt werden. Ist dies in bestimmten Fällen nicht gewünscht, so kann dies durch Inlining von Funktionen im Spurgraphen (im nicht-rekursiven Fall) vermieden werden. Die eigentliche Transformation des Spurgraphen in einen NEA geschieht dann in zwei Schritten: Im ersten werden die Knoten des Spurgraphen durchlaufen und die korrespondierenden Zustände des endlichen Automaten erzeugt. Für primitive Aktionen werden gleich zwei Zustände angelegt, die über einen Übergang gemäß der Semantik

$Sem(a)$ der primitiven Aktion verbunden sind. Die Semantik-Funktion kann beispielsweise bestimmten primitiven Aktionen die selbe Semantik zuordnen oder nur-lesende Aktionen ganz ausblenden. Im letztem Fall wird ein ε-Übergang erzeugt. Für Call-Knoten werden ebenfalls zwei Zustände angelegt, die den Zustand vor und nach dem Aufruf repräsentieren. Für alle weiteren Knoten wird genau ein Zustand erzeugt. Im zweiten Schritt werden dann die Kanten des Spurgraphen durchlaufen und die Zustände des endlichen Automaten über ε-Übergänge gemäß des Kontrollflusses miteinander verbunden. Im Falle von Call-Kanten wird ein ε-Übergang vom Zustand vor dem Aufruf zu dem Zustand des aufgerufenen Entry-Knotens eingeführt. Umgekehrt wird für eine Return-Kante der Zustand für den Exit-Knoten mit dem Zustand nach dem Aufruf verbunden. Der Zustand für den Start-Knoten wird als initialer Zustand und jeder Zustand für einen Final-Knoten als Endzustand markiert.

Resultat der beiden Schritte ist somit ein nicht-deterministischer endlicher Automat, der ε-Übergänge enthält. Unter der Annahme, dass das Erzeugen und Einfügen neuer Zustände und Übergänge in konstanter Zeit durchgeführt werden kann, laufen die beiden Schritte in der Summe in $O(N+E)$, wobei N die Anzahl der Knoten und E die Anzahl der Kanten des Spurgraphen ist.

4 Transformation in einen DEA

Die erzeugten Automaten sind nicht-deterministisch und müssen zur Weiterverarbeitung in einen deterministischen Automaten überführt werden. Bei der Potenzautomatenkonstruktion wird für alle gleichzeitig erreichbaren Mengen von Zuständen die ε-Hülle berechnet, was auf Grund der Vielzahl an ε-Übergängen mit hohem Aufwand verbunden ist. Im Worst-Case werden exponentiell viele Hüllenberechnungen angestoßen. Ist ein Zustand in mehreren erreichbaren Teilmengen, so wird die ε-Hülle mehrfach berechnet. Die Idee besteht nun darin, den nicht-deterministischen Automaten durch eine Reihe von, in ihrer Komplexität ansteigenden, Transformationen in semantisch äquivalente Automaten ohne ε-Übergänge zu überführen und erst im Anschluss eine Potenzautomatenkonstruktion durchzuführen.

Lineare Vereinfachung: Zustände, von denen genau ein ε-Übergang und keine weiteren Übergänge ausgehen, werden mit ihrem Nachfolger vereinigt. Gleiches geschieht mit solchen Zuständen, die nur über einen ε-Übergang erreicht werden. Durch die Vereinigung der Zustände können bereits sehr viele ε-Übergänge entfernt werden. Die Prüfung für einen Zustand kann in konstanter Zeit durchgeführt werden. Damit liegt der Aufwand der beiden Transformationen in $O(s)$, wobei s die Anzahl der Zustände des Automaten ist. Abbildung 2 zeigt ein Beispiel für die lineare Vereinfachung.

Abbildung 2: Lineare Vereinfachung

Kubische Vereinfachung: Es wird nun ein äquivalenter nicht-deterministischer Automat konstruiert, der keine ε-Übergänge enthält. Die Komplexität des Algorithmus wird durch die Hüllenbildung für die ε-Übergänge dominiert und liegt in $O(s \cdot (s+e))$, wobei e die Anzahl der verbliebenen ε-Übergänge ist. Im schlimmsten Fall gibt es s^2 viele ε-Übergänge. Der Algorithmus läuft dann in $O(s^3)$.

Potenzautomatenkonstruktion: Der ε-freie NEA wird im letzten Schritt in einen deterministischen Automaten umgewandelt. Der Aufwand dafür liegt in $O(2^s)$. Bei größeren Automaten bietet sich weiterhin eine Minimierung an, die im Fall eines konstanten Alphabets in $O(s^2)$ [HMU06] durchgeführt werden kann.

5 Vergleich

Mit Hilfe der DEAs kann nun ein Vergleich auf Teilspracheigenschaften oder auf gleiche Sprache durchgeführt werden. Für die Prüfung der Teilspracheigenschaft zweier Automaten A_1 und A_2 wird ein Differenzautomat konstruiert, der die Sprache $\mathcal{L}(A_D) = \mathcal{L}(A_1) - \mathcal{L}(A_2)$ erkennt. Ist diese Sprache leer, so gilt: $\mathcal{L}(A_1) \subseteq \mathcal{L}(A_2)$. Aus $\mathcal{L}(A_1) - \mathcal{L}(A_2) = \emptyset$ und $\mathcal{L}(A_2) - \mathcal{L}(A_1) = \emptyset$ folgt die Äquivalenz der erkannten Sprachen.

6 Messungen

Um den beschriebenen Ansatz zu evaluieren, wurden Tests und Messungen mit 8 Programmen durchgeführt. Abbildung 3 zeigt die Kenngrößen der Testprogramme. Die Anzahl der Zeilen (SLoC) wurde mit dem Programm `sloccount`[1] ermittelt, die weiteren Spalten enthalten die Anzahl der Unterprogramme, der direkten und indirekten Aufrufe (über Zeiger) und der lesenden und schreibenden Speicherzugriffe.

Name	SLoC	Unterprog.	Dir. Aufrufe	Ind. Aufrufe	Writes	Reads
bash	88727	1433	7022	37	8821	31209
bison	25995	1164	3326	209	6143	24435
flex	16903	351	2374	1	2913	9346
gnuchess	12055	462	1506	4	2983	11025
sed	21739	299	1796	1	2741	12017
unzip	49127	272	1240	317	2967	6983
vsftpd	11923	557	2417	4	1330	5301
wget	25925	752	3994	23	4725	16599

Abbildung 3: Kenngrößen der Testprogramme

Für jedes Programm wurden die Objektspurgraphen für sämtliche globale Variablen und Heap-Objekte (eines pro `malloc`-Aufruf) extrahiert und in minimale endliche Automaten überführt. Als primitive Aktionen wurden die Schreib- und Lesezugriffe auf diesen Objekten gewählt. Für die konservative Abschätzung bei Zugriffen über Zeiger und die

[1] `http://www.dwheeler.com/sloccount/`

Berechnung eines Aufrufgraphen wurde eine Zeiger-Analyse nach Andersen [And94] eingesetzt. Von besonderem Interesse war die Frage, inwiefern sich die Vereinfachungen auf die Größe der Automaten auswirken und ob es zu einem exponentiellen Wachstum bei der Potenzautomatenkonstruktion kommt. Abbildung 4 zeigt die durchschnittliche (avg) und maximale (max) Anzahl der Zustände/Übergänge in den extrahierten Automaten. Bereits durch die lineare Vereinfachung (Lin-NEA) wurde eine starke Reduktion der Zahlen im Vergleich zum NEA erreicht. Durch die Kubische Vereinfachung (Kub-NEA) verringerten sich die Zahlen weiter. Der Worst-Case trat bei der Potenzautomatenkonstruktion niemals auf. Die zusätzliche Minimierung der DEAs hat sich speziell bei großen Automaten als sinnvoll erwiesen.

Name	Obj		Spurgraph	NEA	Lin-NEA	Kub-NEA	DEA	Min-DEA
bash	421	avg	1083/2139	1447/2155	156/468	16/120	11/21	7/11
		max	5812/12161	8283/12245	882/3003	296/5134	558/1583	425/1190
bison	262	avg	371/782	520/807	49/117	19/53	33/80	9/13
		max	42232/103953	63503/104609	2537/8680	392/2717	5214/15068	82/196
flex	257	avg	506/1030	629/1055	71/193	21/76	13/25	8/12
		max	34888/77959	43527/78266	3175/9865	519/1215	173/369	132/272
gnuchess	292	avg	147/277	194/294	39/80	17/81	26/63	7/8
		max	3732/8328	4673/8373	849/1340	728/4591	4709/13892	118/160
sed	83	avg	247/467	305/489	61/135	12/26	10/16	7/9
		max	1826/3661	2201/3767	515/1335	102/478	55/138	53/77
unzip	407	avg	104/210	141/222	28/63	10/20	35/89	5/5
		max	17664/42463	24910/42972	2954/9580	1977/6078	11962/34417	110/278
vsftpd	141	avg	812/1879	1247/1899	94/300	8/14	8/10	7/7
		max	54961/139499	87283/141270	5610/17211	26/118	16/32	16/29
wget	65	avg	2550/5959	3705/6024	618/1899	22/2040	10/18	6/8
		max	53460/131852	80172/133259	13599/42205	622/120756	106/289	26/44

Abbildung 4: Reduktion durch Transformationen

Für jedes Programm wurden die berechneten minimalen Automaten sämtlicher Objekte miteinander verglichen. Aus Platzgründen können nur die Ergebnisse für das Programm wget in Abbildung 5 gezeigt werden. Im linken Diagramm sind die Ergebnisse des Vergleichs auf die Teilspracheigenschaft dargestellt. Eine Markierung wurde im Punkt (x,y) gesetzt, falls der Automat x eine Teilsprache des Automaten y erkannte. Auf der Winkelhalbierenden gilt die Teilspracheigenschaft immer. Auffallend sind drei waagerechte Linien, die aufzeigen, dass das Protokoll sämtlicher Objekte ein Teilprotokoll dieser drei Objekte ist. Diese führen beliebige Zugriffsreihenfolgen aus. Die durchbrochenen senkrechten Linien deuten auf sehr einfache Protokolle hin, die von fast jedem Protokoll abgedeckt werden.

Das rechte Diagramm zeigt die Ergebnisse der Prüfung auf gleiche Sprache für ein Automatenpaar (x,y). Aus dem Bild lassen sich weitere Rückschlüsse auf die Qualität der Ergebnisse ziehen: Auf Grund von starken Überschätzungen könnten die Protokolle einzelner Objekte miteinander verschmelzen und fälschlicherweise als äquivalent erkannt werden. Dieser Fall trat jedoch äußerst selten auf. Die Zahl äquivalenter Automaten war relativ gering und konnte durch manuelle Prüfungen bestätigt werden. Darüber hinaus ist die Anordnung in einer Äquivalenzklasse auch ein Indiz für eine semantische Verwandschaft der Objekte. Bei Objekten, die das selbe Verwendungsmuster teilen, konnten oftmals logische Beziehungen festgestellt werden, beispielsweise `day` und `month` in wget.

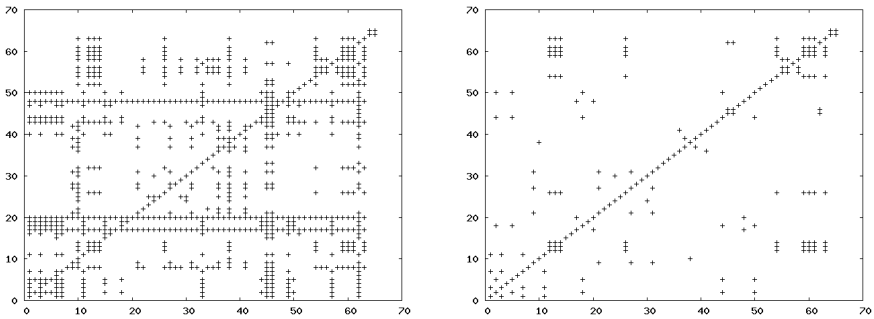

Abbildung 5: Vergleich auf Teilsprache (links) und gleiche Sprache (rechts)

7 Fazit

Es wurde ein Ansatz vorgestellt, Sequenzbeschränkungen von Software-Protokollen über Spurgraphen in endliche Automaten zu transformieren und sie für eine Weiterverarbeitung aufzubereiten. Die durchgeführten Messungen bestätigen, dass sich das Verfahren für reale Programme eignet. Die Präzision für globale Variablen war bereits zufrieden stellend, für Heap-Objekte sind bessere Verfahren in Aussicht. Die Modellierung von Protokollen durch endliche Automaten dürfte für die meisten Zwecke ausreichend sein. Zudem sind sie einfacher verständlich als ausdrucksstärkere (aber komplexere) Formalismen, wie Visibly Pushdown Languages [AM04] oder algebraische Spezifikationen [ZS06].

Literatur

[AM04] Rajeev Alur und P. Madhusudan. Visibly pushdown languages. In *Proceedings of the ACM symposium on Theory of computing*, Seiten 202–211. ACM Press, 2004.

[And94] L.O. Andersen. *Program Analysis and Specialization for the C Programming Language*. Dissertation, University of Copenhagen, 1994.

[EKV05] Thomas Eisenbarth, Rainer Koschke und Gunther Vogel. Static Object Trace Extraction for Programs with Pointers. *Journals of Systems and Software*, 2005.

[HMU06] John E. Hopcroft, Rajeev Motwani und Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison Wesley, Boston, MA, USA, 2006.

[QK06] Jochen Quante und Rainer Koschke. Dynamic Object Process Graphs. In *Proceedings of the Conference on Software Maintenance and Reengineering*. IEEE Computer Society, 2006.

[WML02] John Whaley, Michael C. Martin und Monica S. Lam. Automatic Extraction of Object-Oriented Component Interfaces. In *Proceedings of the International Symposium on Software Testing and Analysis*, July 2002.

[ZS06] Wolf Zimmermann und Michael Schaarschmidt. Automatic Checking of Component Protocols in Component-Based Systems. In *Software Composition*, Seiten 1–17, 2006.