Wiedergewinnung von Architekturinformationen: Ausblicke

Jörg Czeranski, Thomas Eisenbarth, Holger Kienle, Rainer Koschke und Daniel Simon
Universität Stuttgart, Breitwiesenstr. 20-22, D-70565 Stuttgart, Germany
bauhaus@droste.informatik.uni-stuttgart.de

Wiedergewinnen von Informationen über die Architektur einer Software ist eine Voraussetzung für viele Tätigkeiten des Reverse-Engineerings. Ein Aspekt der Software-Architektur ist die Aufteilung eines Softwareartefaktes in Komponenten, Konnektoren und Protokolle. Techniken zur halbautomatischen Komponentenerkennung sind inzwischen in vielfältigen Variationen verfügbar. Ziel des Bauhaus-Projektes ist es, eine Software-Architektur aus dem Quelltext eines Softwareartefaktes halbautomatisch abzuleiten. Die nächsten Ziele des Bauhaus-Projektes umfassen unter anderem die Erkennung von Traces und Konnektoren, die Klassifikation von Teilen einer Komponente, sowie die Unterstützung von Software-Prüfungen durch Datenflussanalysen.

1. Einführung

Im Rahmen des Bauhaus-Projektes wurde der bereits vorhandene Rigi-Editor [7] um Analysen zur Komponentenerkennung erweitert [3].

Unter einer atomaren Komponente (im folgenden einfach Komponente) verstehen wir einen *abstrakten Datentyp* (ADT), ein *abstraktes Datenobjekt* (ADO), eine Mischung aus beidem oder eine Sammlung zusammengehöriger Funktionen.

Als Basisinformation benutzt Bauhaus zwei Darstellungen des Programmcodes:

- IML (InterMediate Language) [2] ist eine detaillierte Zwischendarstellung, wie sie auch in einem Compiler Verwendung finden könnte. Sie wird vor allem für Datenflussanalysen genutzt.
- RG (Resource Graph) [2] ist eine Instanz eines Entity-Relationship-Modells, die den Quelltext des zu untersuchenden Systems in stark abstrahierter Form wiedergibt. Im RG sind nur globale Informationen wie Aufrufsrelationen, Typrelationen und Benutzungsrelationen enthalten.

Die Komponentenerkennung basiert auf dem RG und liefert ihr Ergebnis als Gruppierung von Knoten des RG ab. Der RG wird im Rigi-Editor dargestellt und kann dort direkt manipuliert werden. Im folgenden gehen wir von einer als gesichert geltenden Aufteilung des Systems in Komponenten aus, die durch halbautomatische Analysen mit Hilfe menschlicher Intervention gewonnen wurde.

2. Traces

Für jede Komponente wird eine Schnittstelle nach außen definiert. In der Regel sind mit der Verwendung einer Komponente Einschränkungen verbunden, die über die durch die Programmiersprache formulierbaren Einschränkungen hinausgehen. So sind zum Beispiel Unterprogramme in der Schnittstelle oft Einschränkungen über die Reihenfolge ihrer Aufrufe unterworfen. Die tatsächliche Reihenfolge von Unterprogrammaufrufen wird als *Trace*, die erwartete Reihenfolge als *Trace-Spezifikation* bezeichnet.

Trace-Spezifikationen sind durch die Programmiersprache üblicherweise nicht formulierbar und somit auch durch den Compiler nicht prüfbar, so dass sie in Software-Prüfungen durch den Menschen validiert werden müssen. Die Überprüfung von Hand ist aber insbesondere bei großen Systemen praktisch kaum durchführbar. Ein Beispiel für eine Trace-Spezifikation ist in Abbildung 1 dargestellt.

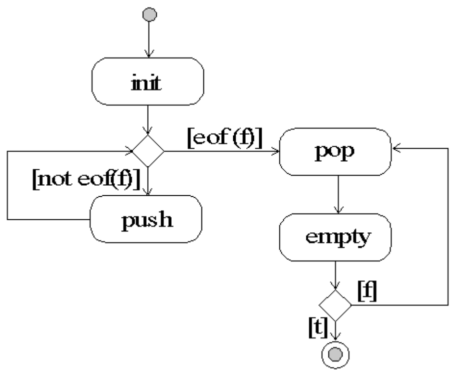

Abb. 1. Trace-Spezifikation für einen Stack

Traces lassen sich statisch herleiten, so dass auf Anfragen hin die tatsächliche Verwendungsweise von Komponenten ermittelt werden kann. Darüber hinaus evaluieren wir existierende Methoden zur Spezifikation von Traces, z.B. in UML. Hauptkriterien bei der Evaluation sind die praktische Verwendbarkeit und die Möglichkeit zum automatischen Abgleich von Trace-Spezifikationen einerseits und den tatsächlichen Traces andererseits.

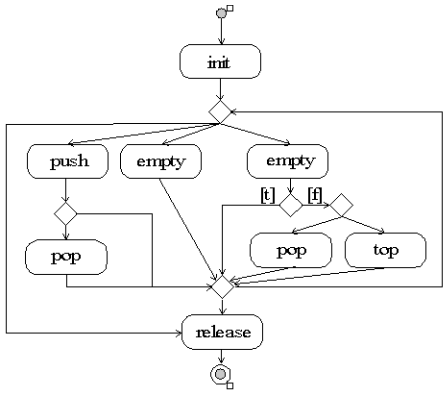

Abb. 2. Automatisch hergeleiteter Trace

Wir erwarten, dass von Hand erstellte explizite Trace-Spezifikationen es ermöglichen werden, einen Großteil der Prüfungen automatisch durchzuführen. Abweichungen der tatsächlichen Traces von ihrer Spezifikation können dann durch automatische Analysen identifiziert und dem Softwareengineer präsentiert werden, so dass dieser seine Arbeit effizient auf die Problembereiche lenken kann.

3. Konnektoren

Für die Beschreibung einer Architektur ist man, zusätzlich zu den Komponenten, auf Konnektoren angewiesen [5]. Aus der Sicht atomarer Komponenten liegen einzelne Prozeduraufrufe und Variablenzugriffe von ihrer Abstraktionsebene her zu tief [6], Interprozesskommunikation, beispielsweise über Pipelines, zu hoch. Allerdings lassen sich Prozeduraufrufe und Variablenzugriffe mit Komponenten so zusammenfassen, dass eine Form von Konnektoren auf der Ebene atomarer Komponenten entsteht. Wir nennen sie *Konnektoren auf mittlerem Abstraktionsniveau*.

Ein ADO selbst kann ein Mittel der Kommunikation zweier Komponenten A und B sein. Dies soll in Abbildung 3 und durch ein Beispiel illustriert werden. A schreibt mittels eines Modifikators, im Beispiel die Funktion Rein, einen Zustand fest, der von B durch Inspektoren, im Beispiel durch Raus angedeutet, abgerufen wird:

```
struct { item i; } Paket; /* das ADO */
void Rein (item x) {Paket.i = x;}
item Raus (void) {return Paket.i;}

void P (void) /* Teil von A */
  {Rein(5);}

void K (void) /* Teil von B */
  {int ergebnis;
   ergebnis = Raus();}
```

Abb. 3. Einfaches ADO in Benutzung

Instanzen eines ADTs lassen sich ebenfalls als Konnektoren auffassen – es sind jedoch komplexere Analysen notwendig, um die genauen Modalitäten solcher Konnektoren aufzudecken: wieviele Instanzen gibt es, was sind die verbundenen Partner usw. Hier spielen dynamische Erwägungen eine größere Rolle, da es zur Laufzeit mehrere Instanzen geben kann.

Wir haben einen einfachen Mustererkenner für RG-Strukturen entwickelt, der ein effizientes Erkennen auf relativ großen RGs zulässt. Allgemein lassen sich die verwendeten Muster durch eine Graphgrammatik beschreiben. Ein Beispiel für ein Muster zur Erkennung ADO-basierter Konnektoren findet sich in Abbildung 4a; A, B und C sind beliebige Komponenten, S_1,S_2 sind Unterprogramme, M ein Modifikator, I ein Inspektor und X die Variable, die als Zustandsspeicher dient. Abbildung 4b zeigt eine Instanz des Musters, die auf das Beispiel aus Abbildung 3 passt.

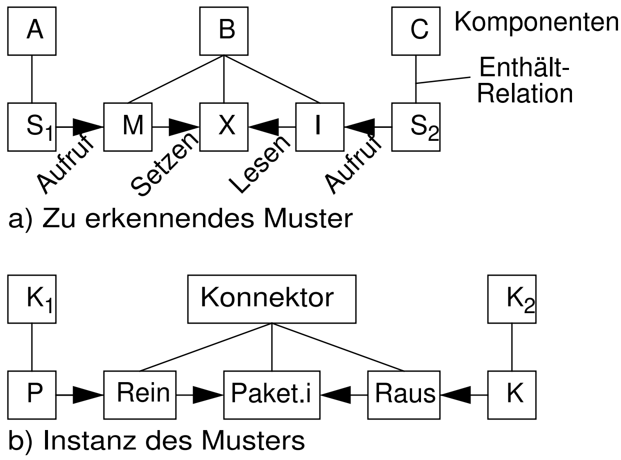

Abb. 4. Muster für ADO-basierte Konnektoren

Für die Konnektor-Analyse ist wichtig, dass die Unterprogramme einer Komponente einer der folgenden Klassen zugeordnet werden: Konstrukturen, Modifikatoren, Inspektoren, Destruktoren oder einer Mischform. Dadurch lässt sich die Richtung der durch den Konnektor hergestellten Verbindung angeben.

4. Statische Analysen

Zur Zeit befindet sich die IML-Darstellung im Umbruch. Die Modellierungsmöglichkeiten werden erweitert und die Integration einer globalen Alias-Analyse [8] und Static-Single-Assignment-Form [1] wird verbessert. Ein Front-End für C existiert bereits, ein weiteres für C++ befindet sich in Entwicklung.

Auf dieser erweiterten IML-Darstellung kann eine statische Seiteneffekt-Analyse [4] durchgeführt werden. Seiteneffekt-Freiheit ist ein wichtiges Merkmal eines Unterprogramms. Die Erkennung von Seiteneffekten ist vielseitig verwendbar. Für ADOs und ADTs lassen sich dadurch automatisch zumindest Konstruktoren und Modifikatoren von Inspektoren unterscheiden, da ein Schreibzugriff auf eine der ersten beiden Arten hinweist, ein lesender Zugriff jedoch auf einen Inspektor. Liegt beides vor, ist das Unterprogramm eine Mischform. Wenn die Klassifikation bekannt ist, so kann z.B. eine Richtung für Konnektoren angegeben werden, die den Informationsfluss anzeigt.

Mit Hilfe statischer Analysen können auch andere interessante Fakten über ein System gefunden werden, z.B. unerreichbare Funktionen, uninitialisierte Variablen oder die Verwendung eines Zeigerparameters als Referenzparameter.

Unerreichbare Funktionen können durch Programmierfehler bedingt sein, als Ausgaberoutinen für Debugzwecke gedacht sein oder einfach nicht mehr benötigte Funktionalität darstellen. Uninitialisierte Variablen können beispielsweise durch Wartungsarbeiten entstehen, wenn bestehende Aufrufreihenfolgen geändert werden und Seiteneffekte nicht bekannt sind. Einige Sprachen, wie C, besitzen keine Referenzparameter sondern nur allgemeine Zeiger als Ersatzdarstellung. Referenzparameter erlauben jedoch nur eine gegenüber Zeigern eingeschränkte Manipulation; es ist daher sinnvoll, dem Reengineer die semantische Information über die Verwendung eines Parameters zugänglich zu machen.

5. Fazit

Insgesamt bieten die Bauhaus-Werkzeuge eine einfache Möglichkeit, die Architektur eines bestehenden Software-Systems zu analysieren. Durch die neuen Techniken, die wir in nächster Zeit hinzufügen werden, wird das Werkzeug mächtiger, auch im Hinblick auf ergänzende Analysen, die die Sicherheit und Qualität der Software nach durchgeführten Wartungsarbeiten erhöhen werden. Insbesondere können Flüchtigkeitsfehler, wie Verstöße gegen ein implizites Protokoll, Einführen einer nicht immer initialisierten Variable oder Erzeugen toter Unterprogramme, weitgehend ausgeschlossen werden. In naher Zukunft möchten wir uns stärker mit der Analyse eines Systems nach Änderungen befassen. Dies soll dem Benutzer eine zeitraubende Neuanalyse des gesamten Systems ersparen und seine Aufmerksamkeit auf die wirklich geänderten Teile lenken.

Literatur

[1] Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K., ‘Efficiently Computing Static Single Assignment Form and the Control Dependence Graph’, ACM Transactions on Programming Languages and Systems, Vol. 13, No. 4, pp. 451-490, October 1991.

[2] Girard, J.G., Koschke, R., Würthner, M., ‘An Intermediate Representation for Reverse Engineering Analyses’, Proceedings of the Fifth Working Conference on Reverse Engineering, pp. 241-250, Honolulu, October 1998.

[3] Koschke, R., ‘Atomic Architectural Component Detection for Program Understanding and System Evolution’, Ph.D. thesis, University of Stuttgart, to appear in 2000.

[4] Landi, W., Ryder, B.G., Stocks, P., Zhang, S., Altucher, R., ‘A Schema for Interprocedural Side Effect Analysis with Pointer Aliasing’, Department of Computer Science, Rutgers University, Number DCS-TR-336, May, 1998.

[5] Shaw, M., Garlan, D., ‘Software Architecture: Perspectives on an Emerging Discipline’, Prentice Hall, 1996.

[6] Shaw, M., ‘Procedure Calls are the Assembly Language of System Interconnection: Connectors Deserve First-Class Status’, in D.A. Lamb (ed) Studies of Software Design, Proceedings of a 1993 Workshop, Lecture Notes in Computer Science No 1078, pp. 17-32, Springer-Verlag 1996.

[7] Tilley, S.R., Müller, H.A., Withney, M., Wong, K., ‘Domain-Retargetable Reverse Engineering’, Proceedings of the International Conference on Software Maintenance, IEEE Computer Society Press, 1993.

[8] Wilson, R.P., ‘Efficient Context-Sensitive Pointer Analysis for C Programs’, Ph.D. thesis, Stanford University, Computer Systems Laboratory, 1997.