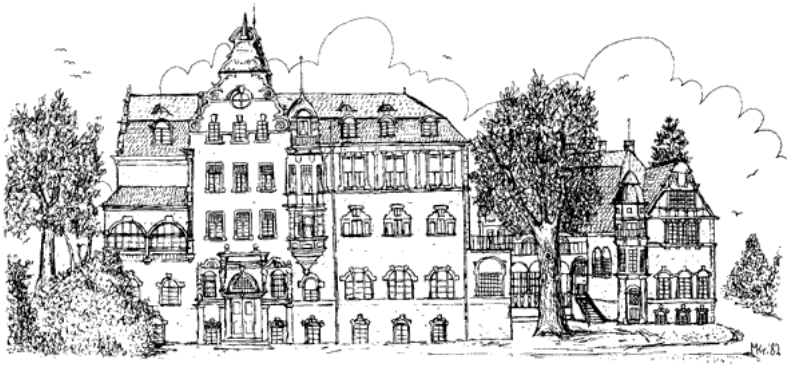
5. Workshop Software Reengineering (WSR 2003)

Bad Honnef, 7.-9. Mai 2003

Jürgen Ebert, Volker Riediger, Andreas Winter
(Universität Koblenz-Landau)
Franz Lehner (Universität Regensburg)



Vorwort

Die *Workshops Software-Reengineering* (WSR) (`http://www.uni-koblenz.de/ist/wsr`) im Physikzentrum Bad Honnef wurde vor vier Jahren von Jürgen Ebert und Franz Lehner ins Leben gerufen, um neben den erfolgreichen internationalen Tagungen im Bereich Reengineering (CSMR, ICSM, IWPC, WCRE, etc.) auch ein deutsch-sprachiges Diskussionsforum zu schaffen. Hier sollten die in dem Bereich des Software-Reengineering tätigen Arbeitsgruppen aus Softwaretechnik und Wirtschaftsinformatik zusammenkommen können.

Ziel der Treffen ist es, einander kennen zu lernen und auf diesem Wege auch eine Basis der Kooperation zu schaffen, so dass das Themengebiet weiteren Fortschritt und Konsolidierung erfährt. Dies geschieht in einem offenen "low-cost"-Workshop ohne eigenes Budget.

Das 5. WSR-Treffen vom 7. - 9. Mai 2003 wurde erneut von den Arbeitgruppen in Koblenz und Regensburg vorbereitet und durchgeführt. Hans Becker sei an dieser Stelle für die administrative Hilfe gedankt. Die GI-Fachgruppen *2.1.1 Softwaretechnik* und *5.1.3 Reengineering und Wartung betrieblicher Anwendungssysteme* haben dieses Vorhaben wieder mitgetragen und über ihre Verteiler bekannt gemacht.

Mittlerweile haben sich in diesen Treffen schon ortsübergreifende Kontakte gebildet und manche Teilnehmer waren schon mehrfach dabei, um in einer diskussions-intensiven Atmosphäre im Physikzentrum in Bad Honnef drei Tage mit Vorträgen, Demos und kleineren gemeinsamen Unternehmungen (übrigens erneut bei sehr gutem Wetter) zu genießen. Die Unterbringung und Versorgung im Physikzentrum war auch in diesem Jahr perfekt, das Essen und die Bereitstellung aller Ressourcen lief reibungslos und die Betreuung war von Entgegenkommen und Freundlichkeit geprägt. Stellvertretend für das gesamte Personal sei hierfür Herrn Gomer und Frau Viehöfer gedankt.

Das Treffen fand zeitgleich mit dem 20. Workshop der GI-Fachgruppe *2.1.4 Programmiersprachen und Rechenkonzepte* statt. Durch eine gemeinsamen Sitzung, die gemeinsame Unterbringung und eine gemeinsame Wanderung zum Drachenfels wurde auch hier der Kontakt miteinander vertieft. Beide Treffen werden voraussichtlich auch in den nächsten Jahren zeitgleich stattfinden.

In diesem Jahr waren 40 Teilnehmer (davon 15 aus der Wirtschaft) anwesen. Es gab 24 Vorträge die das gesamte Spektrum des Reengineerings abdecken. Diese umfassten aktuelle Themen der Forschung wie *Refactoring, Recovery, Aspekte, Kontrollfluss-Analysen, Werkzeugbau und -integration, Qualitätssicherung, Dynamische Programmanalyse, Aufwandsschätzung von Reengineering Projekten, Application Wrapping* und *XML-Anfragetechniken*. Aber auch *Erfahrungsberichte* aus der industriellen Anwendung (insgesamt sieben Beiträge) und eine Beleuchtung der *Lehre* zum Reengineering fehlte nicht. Acht Tool-Vorführungen in einer für Donnerstagabend angesetzten Demo-Session ergänzten das Programm.

Die Vorträge des Workshops und Kurzbeschreibungen der vorgestellten Werkzeuge sind im folgenden zusammengestellt. Sie sind auch - wie die ausführlicheren Vortragszusammenstellungen der Vorjahre - unter `http://www.uni-koblenz.de/ist/wsr` erhältlich.

Software-Reengineering ist nach unserem Eindruck ein auch im deutsch-sprachigem Raum prosperierendes Forschungsgebiet. Ein zunehmendes Industrieinteresse auch an diesem Workshop zeigt, dass die Relevanz dieses Themas immer mehr gesehen wird. Die Informationssites des Reengineering Wiki bei `http://www.program-transformation.org/` und das Reengineering Forum `http://www.reengineer.org/` helfen Interessierten einen Einstieg in die "Szene" zu finden. Im deutsch-sprachigem Raum bietet die Reengineering-Mailingliste ein Forum, dem man aktuelle Ereignisse entnehmen und in das man eigene Informationen einstellen kann (Anmeldung unter `http://mailhost.uni-koblenz.de/mailman/listinfo/reengineering`).

Die nächsten Workshops sind für 3.-5. Mai 2004 und 2.-4.Mai 2005 geplant.

Mai 2003 Jürgen Ebert, Franz Lehner, Volker Riediger, Andreas Winter.

Programm

1. **R. Laemmel**
   (VU Amsterdam)
   Generic Refactoring
2. **V. Kuttruff, T. Genßler, M. Bauer, O. Seng**
   (FZI Karlsruhe)
   Werkzeuggestützte Problemidentifikation und -behebung
3. **S. Bellon, D. Simon**
   (Universität Stuttgart)
   Vergleich von Klonerkennungstechniken
4. **R. Gimnich**
   (IBM Global Services, Frankfurt)
   Reengineering-Schwerpunkte im Transaction Banking
5. **W. Teppe**
   (START Amadeus GmbH)
   Redesign der START Amadeus Anwendungssoftware
6. **J. Genz, M. Franke**
   (VR Kreditwerk AG, Hamburg),
   K. Zelmer (Bausparkasse Schwäbisch Hall)
   Re-Architecting von Legacy Systemen
7. **J. Knodel**
   (IESE Kaiserslautern)
   Reconstruction of Architectural Views by Design Hypothesis
8. **I. Philippow, I. Pashov, M. Riebisch**
   (TU Ilmenau)
   Application of Feature Modeling for Architecture Recovery
9. **K. Schützler, T. Thiel**
   (Humboldt-Universität Berlin)
   Automatisierte Ermittlung von Subsystemschnittstellen
10. **A. Winter**
    (Universität Koblenz-Landau)
    Referenzschemata im Reverse Engineering
11. **S. Breu, J. Krinke**
    (Universität Passau)
    Aspect Mining Using Dynamic Analysis
12. **T. Rötschke, A. Schürr**
    (TU Darmstadt)
    Software Engineering II für Ingenieure: Wartung, Reengineering und Evolution
13. **T. Eisenbarth, R. Koschke, G. Vogel**
    (Universität Stuttgart)
    Extraktion statischer Objekt Prozess Graphen
14. **S. Gossens, M. Dal Chin**
    (Universität Erlangen-Nürnberg)
    Strukturelle Analyse explizit fehlertoleranter Programme
15. **T. Haase**
    (RWTH Aachen)
    A-posteriori-Integration verfahrenstechnischer Entwicklungswerkzeuge
16. **M. Bauer, O. Seng**
    (FZI Karlsruhe)
    Werkzeuggestützte Qualitätssicherung: Ein Erfahrungsbericht Programmanalyse
17. **M. Müller-Olm**
    (Universität Dortmund),
    H. Seidl
    (Universität Trier)
    (Linear) Algebra for Program Analysis Dynamische Programmanalyse
18. **Ch. Steigner, J. Wilke**
    (Universität Koblenz-Landau)
    Verstehen dynamischer Programmaspekte mittels Software-Instrumentierung
19. **U. Kaiser**
    (pro et con Innovative Informatikanwendungen GmbH, Chemnitz)
    Erfahrungen bei der Entwicklung von Werkzeugen zum Reverse Engineering
20. **U. Erdmenger**
    (pro et con Innovative Informatikanwendungen GmbH, Chemnitz))
    BTRACC- Ein Parsergenerator auf der Basis des Back Tracking Verfahrens
21. **H. Sneed**
    (Universität Regensburg)
    Aufwandschätzung von Reengineering Projekten
22. **C. Synwoldt**
    (Fogelberg & Partner GmbH, Frankfurt)
    Ist Web-to-Host bereits alles?
23. **G. Fischer, J. Wolff von Gudenberg**
    (Universität Würzburg)
    Simplifying Source Code Analysis by an XML Representation
24. **M. Hopfner**
    (Universität Tübingen),
    D. Seipel, J. Wolff von Gudenberg, G. Fischer
    (Universität Würzburg)
    Reasoning about Source Code in XML-Representation

Pattern/Refactoring

1 Towards Generic Refactoring

Ralf Lämmel
Vrije Universiteit, De Boelelaan 1081a, NL-1081 HV Amsterdam, CWI, Kruislaan 413, NL-1098 SJ Amsterdam, Ralf.Laemmel@cwi.nl

We define a challenging and meaningful benchmark for genericity in language processing, namely the notion of generic program refactoring. We provide the first implementation of the benchmark based on functional strategic programming in Haskell. We use the basic refactoring of abstraction extraction as the running example. Our implementation comes as a functional programming framework with hot spots for the language-specific ingredients for refactoring, e.g., means for abstraction construction and destruction, and recognisers for name analysis. The language-parametric framework can be instantiated for various, rather different languages, e.g., Java, Prolog, Haskell, or XML schema.
The full paper appeared in [1].

Bibliography

[1] R. Lämmel. Towards Generic Refactoring. In *Proc. of Third ACM SIGPLAN Workshop on Rule-Based Programming RULE'02*, pages 15–28, Pittsburgh, USA, 5 Oct. 2002. ACM Press. Paper obtainable from the ACM Digital Library.

2 Werkzeuggestützte Problemidentifikation und -behebung

Volker Kuttruff, Thomas Genßler, Markus Bauer, Olaf Seng
Forschungszentrum Informatik Karlsruhe (FZI), Haid-und-Neu-Straße 10-14, 76131 Karlsruhe,
{kuttruff|genssler|bauer|seng}@fzi.de

2.1 Einleitung und Ziel

Zur Evolution von Software werden mit wachsender Größe der Software in zunehmenden Maße unterstützende Werkzeuge verwendet. Die Anwendungsgebiete solcher Werkzeuge umfassen die Analyse und Visualisierung von Software, die Identifikation von Problemstellen und die automatische Reorganisation von Software durch Refaktorisierungsoperationen. Während für jedes dieser Gebiete mehr oder weniger brauchbare Insellösungen existieren, fehlt derzeit jedoch eine geschlossene Methoden- und Werkzeugunterstützung von der automatisierten Problemerkennung über die Auswahl (oder Unterstützung bei der Auswahl) von Refaktorisierungsoperationen bis hin zur automatischen Durchführung dieser Refaktorisierungen. In [BGKS02] wurden erste Ansätze zur Verzahnung der oben genannten Techniken vorgestellt. Dieser Beitrag vertieft dieses Thema und berichtet über erste Fortschritte. Das Ziel unserer Arbeit ist die Entwicklung einer geschlossenen Methoden- und Werkzeugkette für die Evolution von Software. Dies umfasst:

- Die Verzahnung von Problemerkennung und Problembehebung auf technischer Ebene zur Beschreibung von Problemmustern und Lösungen mit lokaler Kenntnis des Systems.
- Die Integration der technischen Realisierung der Problemidentifikation und -behebung mit Qualitätsmodellen und einem Expertensystem zur teilautomatischen Auswahl von Problemmustern und möglicher Lösungsstrategien anhand vom Nutzer zu bestimmender und zu gewichtender Qualitätsmerkmale.

In diesem Beitrag konzentrieren wir uns auf den ersten Problembereich.

2.2 Ansatz

Metamodell Um die werkzeuggestützte Problemidentifikation und -behebung geschlossen durchführen zu können, benötigen die daran beteiligten Werkzeuge ein gemeinsames Metamodell. Ausgehend von den am FZI entwickelten Werkzeugen zur Analyse (jGoose [BSLM03])

und Transformation (Inject/J [GK03], Recoder [Rec02]) vereinheitlichten wir daher deren bisher unabhängig voneinander entstandenen Metamodelle. Dieses vereinheitlichte Metamodell besitzt im Wesentlichen zwei Stoßrichtungen: Zum einen ist dies Sprachunabhängigkeit, zum anderen ein dem jeweiligen Anwendungszweck angepasster Detaillierungsgrad. Sprachunabhängigkeit bezieht sich in unserem Kontext auf die Eignung des Modells für ausdrucksbasierte, statisch typisierte, objektorientierte Sprachen. Dies wird durch einen sprachunabhängigen Kern (Klassen, Methoden, Attribute etc.) des Metamodells mit jeweils sprachabhängigen Erweiterungen sichergestellt. Zur Zeit existiert eine detaillierte Beschreibung einer solchen Erweiterung nur für die Sprache Java. Die Abbildung auf C++, C# und Delphi werden derzeit nachgezogen. Der Kern stellt den niedrigsten Detaillierungsgrad des Metamodells dar. Die im Kern vorhandenen Informationen sind im Allgemeinen ausreichend, um Problemstellen in einem System zu identifizieren. So stellt der Kern zum Beispiel bereits eine Reihe von Basismetriken (z.B. McCabe-Komplexität von Methoden) zur Verfügung. Weiterhin existiert eine Erweiterung des Kerns mit höherem Detaillierungsgrad, welche feingranularere Informationen sowie Basistransformationen zur Verfügung stellt.

Aufbauend auf diesem Metamodell existiert eine Skriptsprache [GK01], die sowohl für die Beschreibung der Problemstellen als auch für die Spezifikation der das Problem lösenden Transformationen geeignet ist.

Erkennungsmuster Die geschlossene Beschreibung der problematischen Stellen eines Systems erfolgt in unserer Sprache mit Hilfe so genannter *Erkennungsmuster (detection patterns)*. Dies sind im Wesentlichen deklarative Beschreibungen von nicht notwendigerweise zusammenhängenden Graphmustern, welche die Problemstellen eines Systems charakterisieren. Erkennungsmuster beschränken sich dabei nicht auf die Beschreibung rein struktureller Eigenschaften, sondern erlauben auch die Nutzung darüber hinausgehender Kontextinformationen wie zum Beispiel Metriken. Um diese über strukturelle Muster hinausgehenden Kontextinformationen zu nutzen, haben Erkennungsmuster Zugriff auf die gesamten Informationen, welche das Metamodell zur Verfügung stellt, also insbesondere auch auf Basismetriken, Typ- und Querverweisinformationen etc. Wie bereits erwähnt, geschieht die Spezifikation der zu suchenden Problemmuster deklarativ. Werden von einem Erkennungsmuster allerdings Informationen benötigt, die komplexere Berechnungen zur Folge haben (z.B. komplexe Metriken), so lassen sich diese – im Gegensatz zu reinen musterbasierten Analyse- und Transformationssystemen – imperativ angeben. Dieses Vorgehen wurde gewählt, da sich Berechnungen im Allgemeinen einfacher imperativ als deklarativ beschreiben lassen, insbesondere dann, wenn eine große Menge an Kontextinformationen benötigt wird. Die Suche nach potentiellen Problemstellen, die den in den Erkennungsmustern angegebenen Bedingungen genügen, kann nun mit dem entsprechenden Werkzeug Inject/J automatisiert werden.
Die den Bedingungen der Erkennungsmuster genügenden Graphmuster werden als Instanzen eines Erkennungsmusters bezeichnet. Diese können im Weiteren als Einheit betrachtet werden, auch wenn die zum erkannten Graphmuster beitragenden Strukturelemente über weite Teile des Systems verteilt sind.

Transformation Auf Basis der gefundenen Erkennungsmuster-Instanzen können im nächsten Schritt die notwendigen Transformationen angegeben werden. Dies geschieht ebenfalls mit der durch Inject/J bereitgestellten Skriptsprache. Die Spezifikation der Transformationen erfolgt dabei imperativ, da dies meist einfacher anzugeben ist, insbesondere falls zahlreiche Kontextinformationen beachtet werden müssen. Die teilweise komplexen und umfangreichen Transformationen werden in Termen sogenannter *Basistransformationen* ausgedrückt, welche durch die entsprechenden Erweiterungen unseres Metamodells bereitgestellt werden. Die Basistransformationen lassen sich dabei mit nur lokaler Kenntnis der zu transformierenden Stellen durchführen. Dies bedeutet, dass eine Basistransformation in eine primäre Transformation und weitere sekundäre bzw. abhängige Transformationen aufgeteilt werden kann. Die primäre Transformation führt die eigentlich gewünschte Änderung durch (zum Beispiel das Umbenennen eines Attributs), die sekundären Transformation ändern automatisch alle abhängigen Stellen (zum Beispiel die Benutzungsstellen des Attributs) oder führen notwendige Kontextanpassungen durch (zum Beispiel das Ausrollen von Ausdrücken). Zusammen mit einer Reihe von Vorbedingungen, welche für jede Basistransformation durch das Transformationsmodell gegeben sind, wird die Übersetzbarkeit des transformierten Systems durch unser Werkzeug Inject/J garantiert. Darüberhinaus lassen sich in Inject/J noch nutzerspezifische Vorbedingungen für komplexe Transformationen angegeben, um weitere Eigenschaften wie zum Beispiel Verhaltensbewahrung zuzusichern.

Werkzeug Das Werkzeug Inject/J erlaubt es nun, automatisch nach möglichen Problemstellen im System zu suchen. Ist eine solche Problemstelle durch ein Erkennungsmuster gefunden und eventuell durch den Benutzer bestätigt worden, so lassen sich unter Zuhilfenahme der in der Erkennungsmuster-Instanz gekapselten Informationen entsprechende Transformationen durchführen.

2.3 Zusammenfassung

Im vorliegenden Beitrag haben wir ein Verfahren zur Integration von Techniken der Problemerkennung und automatisierten Softwaretransformation skizziert. Die Hauptidee besteht darin, die Transformationen direkt mit der Beschreibung von Probleminstanzen zu verbinden und dadurch zielgerichtet durchzuführen. Die Struktur von Probleminstanzen wird mit Hilfe von Erkennungsmustern spezifiziert. Erkennungsmuster fassen Strukturelemente,

die im Strukturgraphen des Systems möglicherweise verteilt sind, anhand ihrer Eigenschaften (Beziehungen im Strukturgraphen, Basismetriken etc.) zu neuen, 'virtuellen' Einheiten zusammen und erlauben die geschlossene Transformation dieser Einheiten. Die Transformation selbst wird durch eine Kombination von Basistransformationen beschrieben, welche die Eigenschaft haben, notwendige Sekundäroperationen zur Behebung nicht-lokaler Effekte der Primärtransformation automatisch durchzuführen. Dadurch wird die aktuelle Lücke zwischen Problemerkennung und Behebung zumindest auf technischer Ebene teilweise geschlossen.

Literaturverzeichnis

[BGKS02] M. Bauer, T. Genßler, V. Kuttruff, and O. Seng. Werkzeugunterstützung für evolutionär Softwareentwicklung. In *Proceedings of the 4th German Workshop on Software-Reengineering*, July 2002.

[BSLM03] M. Bauer, O. Seng, S. Luzar, and T. Marz. jGoose Echidna WWW Page. `http://jgoose.sf.net/`, 2003.

[GK01] T. Genßler and V. Kuttruff. Werkzeugunterstützte Softwareadaption mit Inject/J. In *Proceedings of the 3th German Workshop on Software-Reengineering*, July 2001.

[GK03] T. Genssler and V. Kuttruff. Inject/J WWW Page. `http://injectj.sf.net/`, 2003.

[Rec02] The RECODER/Java homepage. `http://recoder.sf.net`, 2002.

---

3 Vergleich von Klonerkennungstechniken

Stefan Bellon, Daniel Simon
Universität Stuttgart, Institut für Softwaretechnologie, Universitätsstraße 38, D-70569 Stuttgart,
{bellon,simon}@informatik.uni-stuttgart.de

3.1 Einleitung

„Copy & Paste“ ist noch immer das vorherrschende Programmierparadigma, wenn es um Wiederverwendung von Code geht. Durch häufiges „Copy & Paste“ leidet jedoch die Wartbarkeit des Systems, da ein kopierter Fehler eventuell an vielen Stellen korrigiert werden muss. Allerdings ist in den seltensten Fällen dokumentiert, wohin ein Stück Code kopiert wurde. In der Literatur wurden eine Reihe von Techniken zur Entdeckung so genannter Klone (also Code-Stücke, die sich aus „Copy & Paste“ ergaben) vorgeschlagen. Jedoch ist bis dato unklar, welche der Techniken unter welchen Umständen die bessere ist.

Um dies herauszufinden, haben sich die Wissenschaftler Baker [1], Baxter [2], Kamiya [5], Krinke [7], Merlo [6] und Rieger [4] zusammengetan, um die verschiedenen Techniken quantitativ und qualitativ zu vergleichen.

Es wurde ein Vorgehen erarbeitet, welches den quantitativen Vergleich der Ergebnisse ermöglicht. Die Techniken wurden auf Java- und C-Systeme angewendet, die mit versteckten Klonen präpariert wurden, um die Aussagekraft der Ergebnisse einordnen zu können. Die Analyse und Auswertung der Ergebnisse soll vorgestellt und die Stärken und Schwächen der einzelnen Techniken herausgearbeitet werden.

3.2 Vergleichsmethode

Um die Techniken der sechs genannten Wissenschaftler miteinander vergleichen zu können, musste zuerst ein gemeinsames Verständnis von „Klon“ gefunden werden. Die Teilnehmer konnten sich alle darauf einigen, dass ein Paar von aufeinander folgenden, ununterbrochenen Sourcecode-Zeilen, so genannten „Codefragmenten“, ein „Klonpaar“ ergibt und dies als Vergleichsgröße aller Werkzeuge und Techniken dient. Techniken, die mit „Klonklassen“ arbeiten, können diese als Menge von Klonpaaren darstellen und können somit auch eingebunden werden.

Drei Typen von Klonen wurden definiert:

Typ 1: Exakte Kopie (keinerlei Veränderung bis auf Whitespace und Kommentare, z. B. Inlining von Hand)

Typ 2: Parametrisierte Übereinstimmung (Bezeichner werden in der Kopie umbenannt, z. B. „Wiederverwendung“ einer Funktion)

Typ 3: Kopie mit weiteren Modifikationen (Code der Kopie wird abgeändert, nicht nur Bezeichner, z. B. „Erweiterung“ einer Funktion)

Nach Test-Läufen mit kleineren Systemen wurden im Hauptteil des Experiments insgesamt acht Systeme ausgewählt, darunter vier in C und vier in Java. Die Größen der Systeme variierten von 30 KLOC bis über 200 KLOC, um eventuell Aussagen über das Auftreten von Klonen im Verhältnis zur Systemgröße treffen zu können.

Um die von den Teilnehmern des Experiments eingesen-

deten Klon-Kandidaten miteinander vergleichen zu können, wurde eine Referenzmenge von Klonpaaren erstellt. Dazu wurden zufällig ausgewählte Klon-Kandidaten der Teilnehmer betrachtet und entweder in die Referenzmenge übernommen (eventuell mit leichten Veränderungen) oder verworfen. Anhand dieser Referenzmenge werden Werte wie Recall (Anzahl gefundener Kandidaten im Verhältnis zur Referenzmenge) und Precision (Anzahl gefundene Kandidaten im Verhältnis zur eingesandten Menge der Kandidaten) angegeben. Um diese Werte angeben zu können, ist es notwendig, ein Maß für die Überdeckung von Klon-Kandidaten mit Klonen aus der Referenzmenge einzuführen. Damit zu kleine Überdeckungen nicht als korrekte Klone gewertet werden, sind die folgenden Maße für die Überdeckung ausgewählt worden:

- OK-Match(p) gdw. Anteil des Schnitts von Kandidat und Referenz ist zu mindestens p % im Kandidat oder in der Referenz enthalten.
- Good-Match(p) gdw. Anteil des Schnitts von Kandidat und Referenz ist zu mindestens p % in der Vereinigung von Kandidat und Referenz enthalten.

Good-Match(p) impliziert also OK-Match(p) und ist daher ein stärkeres Kriterium.

3.3 Ergebnisse

Stellvertretend für die acht untersuchten Systeme sollen hier repräsentativ die Ergebnisse des Systems SNNS (Stuttgart Neuronal Network Simulator, 115 KLOC) vorgestellt werden. Es handelt sich hierbei um das größte System, zu dem alle Teilnehmer eine Einsendung machen konnten.

In Abbildung 3.1 sind die eingesendeten Klon-Kandidaten aller Teilnehmer und die Referenzmenge (sog. Orakel) abgebildet. In Abbildung 3.2 ist die Anzahl der von den eingesendeten Kandidaten überdeckten Referenzen dargestellt. Die bereits angesprochenen Werte für Recall und Precision sind in Abbildung 3.3 zu sehen. Die Referenzmenge ist allerdings nicht vollständig, sondern durch Bewertung von nur 2 % aller Kandidaten entstanden.

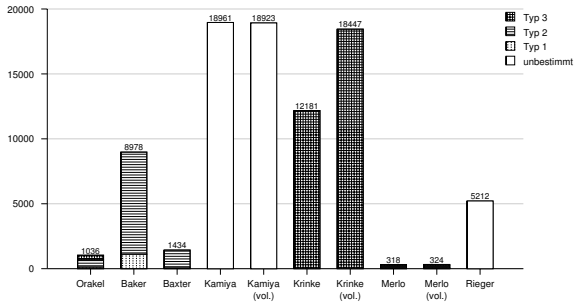


Abbildung 3.1. Kandidaten aller Teilnehmer für das System SNNS

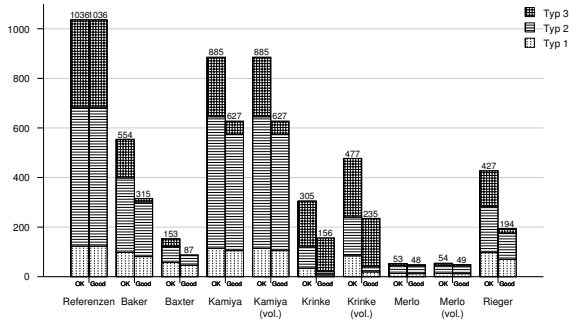


Abbildung 3.2. Überdeckte Referenzen aller Teilnehmer für das System SNNS

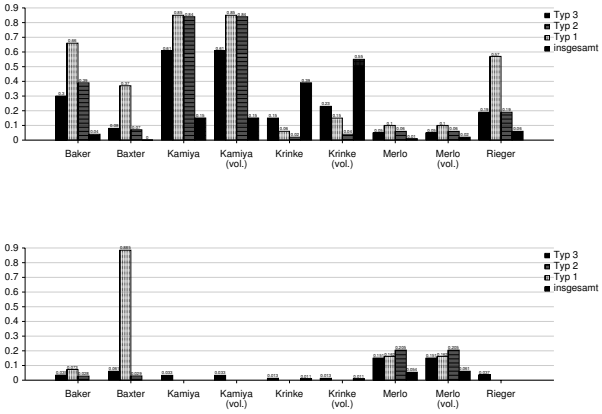


Abbildung 3.3. Recall (oben) und Precision (unten) für das Projekt SNNS

Man erkennt deutlich, dass Werkzeuge, die sehr viele Kandidaten zurück liefern, einen hohen Recall aber auch eine niedrige Precision haben. Bei Werkzeugen, die eher weniger Kandidaten melden, ist dies genau umgekehrt. In unserem Experiment wurden noch weit mehr Daten erhoben, die in [3] veröffentlicht sind.

3.4 Zusammenfassung

Der Vergleich hat gezeigt, dass es nicht *das* Werkzeug oder *die* Technik zur Erkennung von Klonen gibt. Die Wahl des geeigneten Werkzeuges wird daher je nach Anwendungsgebiet unterschiedlich ausfallen. Will man möglichst viele Klone erkennen und ist bereit, die Kandidaten manuell zu betrachten, so bieten sich Werkzeuge mit hohem Recall an. Dazu zählen die von Baker, Kamiya und Rieger. Will man jedoch die Klone automatisiert erkennen und eventuell sogar beseitigen, so muss die Precision möglichst hoch sein, wie bei Baxter und Merlo.

Alle Techniken jedoch erkennen noch zu wenig Klone: Es wurden nicht einmal die Hälfte der von uns in den Systemen versteckten Klone entdeckt.

Interessanterweise bezieht keine der am Vergleich beteiligten Techniken Kommentare im Quellcode in die Erken-

nung mit ein. Dies könnte insbesondere bei „kleinen“ Klonen die Erkennungsrate erhöhen. Derartige Ansätze verfolgen wir zur Zeit und erwarten in Kürze erste Ergebnisse.

Literaturverzeichnis

[1] BAKER, BRENDA S.: *Parameterized Pattern Matching: Algorithms and Applications*. Journal Computer System Science, 52(1), 1996.

[2] BAXTER, IRA D., ANDREW YAHIN, LEONARDO MOURA, MARCELO SANT'ANNA und LORRAINE BIER: *Clone Detection Using Abstract Syntax Trees*. in *Proc. ICSM*, 1998.

[3] BELLON, STEFAN: *Vergleich von Techniken zur Erkennung duplizierten Quellcodes*. Diplomarbeit, Universität Stuttgart, 2002.

[4] DUCASSE, STÉPHANE, MATTHIAS RIEGER und SERGE DEMEYER: *A Language Independent Approach for Detecting Duplicated Code*. in *Proc. ICSM*, 1999.

[5] KAMIYA, TOSHIHIRO, SHINJI KUSUMOTO und KATSURO INOUE: *CCFinder: A Multi-Linguistic Token-based Code Clone Detection System for Large Scale Source Code*. TSE (to appear).

[6] KONTOGIANNIS, K., R. DEMORI, M. BERNSTEIN, M. GALLER und ETTORE MERLO: *Pattern matching for design concept localization*. in *Proc. WCRE*, 1995.

[7] KRINKE, JENS: *Identifying Similar Code with Program Dependence Graphs*. in *Proc. WCRE*, 2001.

Praxisberichte

4 Reengineering-Schwerpunkte im Transaction Banking

Rainer Gimnich
IBM Global Services, BCS Financial Services, Frankfurt/Main, gimnich@de.ibm.com

Ein großer Teil der Projekte im Bereich Finanzdienstleistungen, insbesondere bei Transaktionsbanken, basiert auf existierenden, gewachsenen Anwendungssystemen. Der Vortrag beschreibt überblicksartig aktuelle Reengineering-Projektansätze und stellt dabei die technischen und organisatorischen Kernprobleme vor.

4.1 Reengineering-Projektkategorien

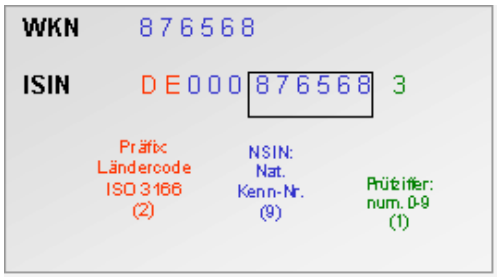
Aufgrund der Globalisierung und des anhaltenden Kostendrucks verfolgen viele Banken das Ziel, Backoffice-Prozesse (z.B. Order-Management, Wertpapierabwicklung, Verwaltung, Bewertung) organisatorisch zusammenzufassen. Als weiterer Schritt kann die Ausgründung dieser Dienstleistungen in eigenen Tochterunternehmen oder in Joint Ventures mit anderen Banken folgen. Außerdem können Backoffice-Dienstleistungen bereits heute von Transaktionsbanken am Markt bezogen werden. Die Kosten dafür berechnen sich nach Volumen (d.h. Anzahl Transaktionen) und nicht mehr als Fixkosten des eigenen Hauses. Transaktionsbanken können für viele Auftraggeber arbeiten und damit hohe Transaktionsvolumina erreichen. Umso eher lohnen sich Investitionen in neue Produkte (d.h. neue bzw. erweiterte Anwendungsfunktionen) und neue IT-Infrastrukturen. Die meisten deutschen Transaktionsbanken arbeiten heute mit umfangreichen Abwicklungssystemen, die von ihnen selbst (oder ihrer Muttergesellschaft vor der Ausgründung) entwickelt wurden: Legacy-Systeme im besten Sinne. Die Anwendungen haben mit ihrer ‚Peripherie' (u.a. Buchungsschnittstelle, Börsenanbindung) häufig Größenordnungen von mehreren Millionen LOC und sind überwiegend in den zur Entwicklungszeit gängigen Sprachen (COBOL, PL/I, Assembler) realisiert, wobei häufig auch 4GL-'Aufsätze' (wie Delta oder Telon) verwendet werden. Ausgehend von den generellen Projektzielsetzungen lassen sich die Vorhaben in mindestens vier Kategorien einordnen:

- Umstellungen aufgrund gesetzlicher Vorgaben oder Standardisierung
- Funktionale Erweiterung und Flexibilisierung
- Einführung von Standardsoftware für zentrale Systemkomponenten
- Outsourcing

4.2 Umstellungen aufgrund gesetzlicher Vorgaben und Standardisierung

Hier handelt es sich um sog. ‚Muss-Projekte', die in der Gesamtplanung gegenüber den vielen anderen Projekten im Unternehmen entsprechend abgestimmt und priorisiert werden müssen. Typische Beispiele für gesetzlich notwendige Projekte sind Änderungen der Handelsbedingungen für Wertpapiere oder steuerliche Änderungen. Ein aktuelles Beispiel für Umstellungsprojekte aufgrund von

Standardisierungsvorgaben bilden die gerade abgeschlossenen Projekte zur ISIN-Einführung: Die Bankenverbände mehrerer europäischer Länder, darunter Deutschland und Österreich, haben beschlossen, statt der nationalen Wertpapierkennnummern (WKN) in Zukunft den internationalen Nummerierungsstandard ISIN (International Securities Identification Number) zu verwenden. Einführungstermin: 22.04.2003 (‚Osterdienstag'). Die heutige 6-stellige numerische WKN wird durch die 12-stellige alphanumerische ISIN als führende Kennung ersetzt. Hintergrund ist die Ausschöpfung der WKN-Nummernkreise, bei heute bereits über 250.000 von WM (Wertpapier-Mitteilungen) gepflegten WKN. Außerdem wird durch die ISIN-Einführung eine Vereinheitlichung von nationalen und internationalen Handelskonventionen erreicht, was langfristig vorteilhaft ist.



Die sogenannte “alte WKN” kann übergangsweise bankintern noch verwendet werden, jedoch ist Vorsicht angebracht, da neue Nummernausgaben in Zukunft alphanumerisch sein können. Intern ist es für diesen Zweck mit einer Transformation `PIC 9(6) -> PIC X(6)` im Datentyp der WKN-Variablen nicht getan: WKN werden häufig als Schlüssel für Datenbankzugriffe verwendet, und die Datenbankfelder haben noch den alten Typ. Außerdem sind WKN häufig gepackt repräsentiert: `PIC 9(6) PACKED-DECIMAL`, und die 4 Byte der internen Repräsentation können nur mit binärer Interpretation und jeweiliger “Umrechnung” auf die alphanumerische WKN abgebildet werden, und das bei jedem Zugriff. Dieses Vorgehen birgt eine Reihe neuer Fehlerquellen und stellt insgesamt nur eine Zwischenlösung beim Übergang zur ISIN dar. Dass solche Verfahren dennoch eingesetzt werden, zeigt, wie groß der Zeitdruck und der Gesamtumstellungsaufwand in den Unternehmen ist.

4.3 Funktionale Erweiterung und Flexibilisierung

Diese Reengineering-Projekte zielen auf die Überarbeitung der existierenden Systeme im Hinblick auf neue Kunden- und Marktanforderungen ab, einschließlich technologischer Anforderungen (z.B. WAP-Order). Beispiele für diese Art Reengineering-Projekte sind: Einführung der Mandantenfähigkeit, Optimierung der mandantenspezifischen Verarbeitung, Automatisierung der Mandantenmigration, Standardisierung der externen Schnittstellen, ‚Bereinigung' und Modularisierung der gewachsenen Systemlandschaft (Programmcode, Datenhaltung, Ablaufsteuerung), Realisierung von Skalierbarkeitsanforderungen, Unterstützung der Portierung in andere Umgebungen (Ermittlung der IT-Infrastruktur-Abhängigkeiten) u.v.m. Viele dieser Reengineering-Projekte haben starken Einfluss auf die Anwendungsarchitektur, sowohl im Hinblick auf die Strukturierung der Anwendungskomponenten als auch auf die nicht-funktionalen Anforderungen und die Infrastruktur. Daher ist die Analyse und Beschreibung der Ist-Architektur des Gesamtsystems häufig der erste Reengineering-Schritt. Nach der Entwicklung der Zielarchitektur lassen sich interne Anforderungen in Bezug auf das Configuration Management und den Reengineering-Werkzeugeinsatz ableiten. Erfahrungsgemäß stützt sich die Lösung konkreter Reengineering-Probleme vielfach auf die Eigenschaften der im Gesamtsystem ermittelten Feldtypen ab:

- Kennungen bzw. Nummernkreise:
  - Ordernummer
  - Depotnummer
  - Mandantennummer
  - Vertriebsbereichs-/Agenturnummer
  - Kunden-/Vertragsnummer
  - Wertpapierkennnummer (WKN), soweit nicht in Standardisierungsprojekt bearbeitet (s.o.)
- Berechnungsgrößen:
  - Betragsfelder, insbesondere Summenfelder
  - (programminterne) Tabellen, insbesondere Staffelungen

In einem kürzlich abgeschlossenen Projekt basierte die Lösung für skalierbare Wertpapierabwicklung inhaltlich auf 2 Feldtypen, die in allen ihren Ausprägungen über die gesamte Anwendungslandschaft ermittelt und angepasst wurden: Mandantennummern und Summenfelder.

4.4 Einführung von Standardsoftware

Kern dieser Reengineering-Projekte ist die Entscheidung, Teile der Legacy-Anwendungen durch am Markt verfügbare Software-Produkte zu ersetzen, z.B. das Order-Management, die ‚reine' Abwicklung oder die bankfachlichen Funktionen. Der Hauptgrund für dieses Vorgehen ist Kostensenkung durch vereinfachte Wartung in der Zukunft. Ein Problem ist jedoch die Integration der heutigen, im existierenden System hinterlegten Prozesse mit denen, die die Standardsoftware ‚erwartet'. Daher werden in der Regel die Standardfunktionen angepasst (‚customizing') und fehlende Funktionalität aus Legacy-Komponenten übernommen, angepasst oder neu entwickelt. Das Reengineering-Problem liegt in der Ermittlung und Realisierung der Schnittstellen von vorhandener und neuer (Standard-)Funktionalität.

4.5 Outsourcing

Hiermit ist die Verlagerung der Anwendungssysteme an einen IT-Anbieter gemeint, der von der Infrastruktur bis zu Anwendungsbetrieb, Wartung und Weiterentwicklung verantwortlich ist. Kern der Vereinbarungen zwischen Kunden und Anbieter sind Service Level Agreements (SLAs). Der Reengineering-Aufwand liegt in diesen Projekten nicht bei der Transaktionsbank, sondern beim Outsourcing-Anbieter. Dieser hat aufgrund seiner bereits optimierten Infrastruktur und der ‚economies of scale' in der Regel bessere Möglichkeiten, die technologische Weiterentwicklung der Anwendungen voranzutreiben. Outsourcing-Projekte haben damit andere Grundlagen und Ausrichtungen als die zuvor beschriebenen Projektkategorien. Dennoch werden beim Anbieter intern in der Regel die gleichen Reengineering-Methoden und -Werkzeuge verwendet wie bei den anderen Projekttypen.

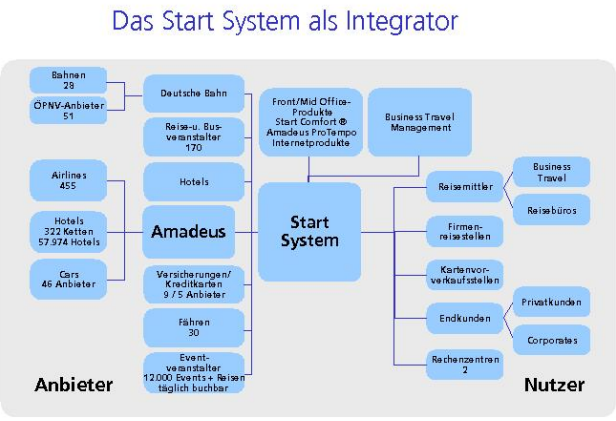
---

5 Redesign der START Amadeus Anwendungssoftware, Ein Erfahrungsbericht

Werner Teppe
Start Amadeus GmbH, Marienbader Platz 1, D 61341 Bad Homburg,
Werner.Teppe@start.de

5.1 Einleitung

Das START System fungiert im deutschen Reisemarkt als Integrator zwischen Anbieter- und Nutzerseite. Anbieter sind: Airlinesys- teme, Touristik- und Reiseveranstalter, Deutsche Bahn und Regionalverbundsyste- me, Versicherungsgesellschaften, Fähren, Eventveranstalter, Hotels sowie Autovermie- ter. Auf der Nutzerseite befinden sich Reisevermittler (Reisebüros), Firmenreise- stellen, Kartenvorverkaufsstellen und Endkunden.



Start Amadeus entwickelt und betreut die Endgerätesoftware, ist für das Netzwerk zwischen den Partnern zuständig, entwickelt und betreibt das zentrale Anwendungssystem. Um das Redesign dieser zentralen Anwendung geht es im folgenden Beitrag.

5.2 Ausgangssituation

Der Kern der zentralen Anwendungssoftware von START Amadeus (ASW) wurde vor rund 25 Jahren entwickelt.

Die ASW bestand damals aus ca. 300.000 Lines of Code (SPL und Assembler). Es wurde ein dafür speziell entwickeltes Dateihandlingsystem eingesetzt. Zu Beginn wurden rund 1000 (dumme) Terminals bedient, die einstellige Transaktionsraten in der Spitzenzeit verursachten. Mit drei Rechnerkopplungen zu weiteren Informationsanbietern wurde begonnen. Die ASW wurde stark auf Performance getrimmt, da Großrechnerressourcen teuer waren (und auch heute noch sind). In der Zwischenzeit wurde die ASW massiv weiter entwickelt und mehr als 170 Informationsanbieter mit ihren Reservierungssystemen wurden angeschlossen. So besteht die ASW heute aus mehr als 3,5 Millionen Lines of Code (MLoCs).

Die Spitzentransaktionsrate beträgt 655 Benutzertransaktionen pro Sekunde. Angeschlossen an das START-System sind heute ca. 45.000 PC und 27.000 Drucker. Aktuell wird alle 5 Wochen ein neues Anwendungsrelease eingeführt, das neue Funktionen zur Verfügung stellt oder einen weiteren Informationsanbieter anschließt, Tendenz steigend. Zu Beginn gab es jede Woche eine neue Version. Diese Frequenz wird auch heute von unseren Kunden gefordert.

Da die ASW immer komplexer und damit Erweiterungen und Änderungen immer zeitaufwendiger wurden, einzelne Fehler großflächigere Ausfälle zur Folge haben können und nur noch schwer Personal für diese Art der Mainframeanwendungen zu finden ist, starteten wir vor ca. 2 Jahren das Projekt “Redesign ASW”.

5.3 Vorgehen

Dabei wurden mehrere Ansätze verfolgt und untersucht:

- Zerlegung der ASW in Komponenten
- Flexibilisierung bei der Datenhaltung (Einsatz von Datenbanksystemen)
- Unabhängigkeit vom Betriebssystem
- Wechsel der Programmiersprache (von SPL nach C / C++)
- Reduzierung der Komplexität

Bei allen geplanten Änderungen darf der laufende Betrieb natürlich nicht gefährdet werden. Der Benutzer im Reisebüro darf die Änderungen nicht bemerken. Es handelt sich so zu sagen um eine “Operation am offenen Herzen”.

Über die Vorgehensweise im Projekt, die verschiedenen Ansätze habe ich auf der Tagung im letzten Jahr berichtet (WSR2002).

In der Zwischenzeit wurden einige dieser Ansätze in die Praxis umgesetzt und laufen bereits in der Produktion. So wurde eine Business Logic Komponente aus der ASW ausgegliedert und kann nun über CORBA-Technologie aufgerufen werden. Weiterhin wurde ein XML-Zugang zur ASW geschaffen, der in Kürze in Produktion geht, um neue Partneranwendungen mit moderner Technologie anzuschließen. Komponenten wurden aus der Mainframeumgebung auf Unixsysteme verlagert.

Über die bisher gemachten Erfahrungen, den aktuellen Projektstand und das weitere geplante Vorgehen berichte ich in meinem Vortrag und im Folgenden.

5.4 Komponententechnologie

Wie aus Bild 5.1 hervorgeht, steigt die Zeit zwischen zwei ASW-Versionen, die dem Markt zur Verfügung gestellt werden können, stärker als linear mit der Anzahl der Lines of Code. So konnte Start Amadeus im Jahr 1990 bei rund 1 MLoCs ca. jede Woche eine neue Version in Betrieb nehmen und den Kunden zur Verfügung stellen, heute bei mehr als 3,5 MLoCs liegen 5-6 Wochen zwischen den Versionen.

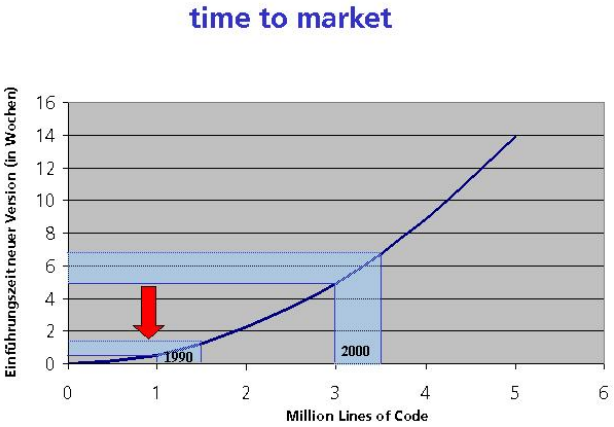


Abbildung 5.1. time to market

Daher war ein Hauptansatzpunkt im Projekt die Zerlegung der ASW in Komponenten. Außerdem kann man, wenn Komponenten vorliegen, auch die übrigen Maßnahmen an einzelnen Komponenten durchführen (und nicht an der kompletten Anwendung).

Zunächst wurden die statischen und dynamischen Beziehungen der mehr als 1750 Sourcen (Übersetzungseinheiten) sowie die Verwendung der ca. 3000 Headerfiles untersucht. Dazu mussten eigene Werkzeuge entwickelt werden, da sowohl die kommerziellen als auch die in der Forschung dafür verfügbaren Tools die Sprache SPL nicht unterstützen. Meist fehlt ein entsprechendes Frontend eines geeigneten Tools. SPL (ein Subset von PL/I) steht für System Programming Language und ist die Implementierungssprache des Betriebssystems BS2000 von Fujitsu Siemens.

Nachdem die Beziehungen zwischen den Modulen analysiert waren, wurden Cluster identifiziert, die sich für eine Zerlegung eignen. Es wurde ein Cluster als Pilot ausgewählt, der einerseits nicht zu umfangreich ist, andererseits alle wesentlichen Eigenschaften großer Cluster beinhaltet, damit die gewonnenen Erfahrungen auch später übertragen werden können.

Die Kommunikation der Komponenten sollte über Corba Technologie erfolgen, u.a. um am Markt verfügbare Anwendungen leichter anschließen zu können (buy statt make). Für die bei Start Amadeus vorhandenen Systemplattformen (BS2000, verschiedene Unix-Derivate, Windows) war kein geeigneter Object Request Broker (ORB) am Markt verfügbar. Eine Portierung kommerzieller Systeme schied aus Kostengründen aus. Daher wurde das Open Source Produkt omniORB (bester der untersuchten ORBs) ins BS2000 portiert. Die Portierung wurde der Open- Source-Gemeinde zur Verfügung gestellt, sodaß andere BS2000 Anwender den omniORB einsetzen können. Zusätzlich wurde für die Transaktionssteuerung über Komponentengrenzen hinaus die Funktion “Verteilte Transaktionen (VTA)” entwickelt. Sie stellt sicher, dass Datenänderunegn vollständig oder gar nicht über Komponentengrenzen hinweg durchgeführt werden (Sicherstellung der Datenkonsistenz).

5.5 Proof in Production

Wichtig von Projektbeginn an war, die Projektergebnisse in Produktionsumgebung zu verifizieren. So sollten nicht nur Konzepte entwickelt werden, die Prototypen sollten auch im harten Produktionsalltag ihre Tauglichkeit beweisen (Performance, Produktionsabläufe, ...). Aktuell hat die Komponente FAHR alle Teststufen durchlaufen und wird gerade in die Produktion eingeführt.

5.6 Ausblick

Anschließen werden sich Performancemessungen, Tuningmaßnahmen und Optimierungen der Programme und der Produktionsabläufe. Danach kann die Herauslösung weiterer Komponenten in Angriff genommen werden.

6 Re-Architecting von Legacy Systemen

Marc Franke, Jürgen Genz
VR Kreditwerk AG, Hamburg {Marc.Franke,JürgenGenz}@kreditwerk.de

Klaus Zelmer
Bausparkasse Schwäbisch Hall Klaus.Zelmer@Schwaebisch-Hall.de

Die Bausparkasse Schwäbisch Hall betreibt seit mehr als 40 Jahren eine umfangreiche IT-Landschaft, die im Bereich der Kerngeschäftsanwendungen überwiegend selbst entwickelt wurde. Während dieser langen Zeitstrecke hat sich die Bausparkasse Schwäbisch Hall zum Benchmarktführer in der Produktivität, innerhalb der Verwaltung von Bausparverträgen, entwickelt. Durch die technologischen und methodischen Entwicklungen hat sich hierbei eine umfangreiche und äußerst komplexe Gesamtlandschaft ergeben.

Die alleine im Host-Umfeld über 5 Millionen Lines of Code wurden in verschiedenen Programmiersprachen entwickelt, wobei heute der Schwerpunkt auf Cobol- und Assembler-Programmen liegt. Ebenso sind in dem Programmbestand sowohl strukturierte als auch objektorientierte und komponentenorientierte Programmierparadigmen zu finden. Im Laufe der Zeit wurden bei einem stetig steigenden Grad der Integration immer mehr Fachgebiete IT-technisch abgebildet. Der Einsatz neuer Technologien führte fast immer zu einem zusätzlichen Bedarf (Beispiel PC-Einsatz).

All diese Entwicklungen hatten und haben eine stetige Steigerung der Komplexität und Heterogenität der Gesamtlandschaft sowie vielfältige Abhängigkeiten zwischen einzelnen Komponenten und Systemteilen bzw. Anwendungsbereichen zur Folge. Letztendlich leidet darunter die Beherrschbarkeit der Systemlandschaft. Aufwände für Pflege und Wartung sowie für die Integration neuer gesetzlicher Anforderungen benötigen einen immer größer werdenden Teil der verfügbaren Ressourcen. Die Möglichkeiten zur Umsetzung und Einführung innovativer und die Produktivität signifikant steigernder Funktionen oder Neuerungen, die aus Marktsicht schnell benötigt werden, können immer schwerer realisiert werden.

Ziel des Reengineerings ist es deshalb, die Systemlandschaft unter Abbau der Komplexität und Heterogenität neu zu strukturieren. Dadurch soll die heute schon hohe Funktionalität und Stabilität weiter ausgebaut werden und für die Zukunft mit weniger Aufwand wartbar sowie einfacher erweiterbar sein.

Recovery

---

7 Reconstruction of Architectural Views by Design Hypothesis

Jens Knodel
Fraunhofer Institute for Experimental Software Engineering (IESE) Sauerwiesen 6, D-67661 Kaiserslautern, Germany, knodel@iese.fraunhofer.de

Abstract

The literature proposes many techniques for reconstructing software architectures. However, there are limited guidelines on when and how to apply these techniques. There is even less information on how to combine them depending on the objectives and the boundary conditions of the reconstruction. This paper presents an approach to create an up-to-date high-level design model of a system and reports on the experience gained by applying this approach in an industrial case study.

Keywords: software architecture, architecture recovery process and tools, reverse engineering, reflexion model

7.1 Introduction

A successful software system evolves over time. In order to manage the effects of increasing complexity and continuous change (Lehman´s laws, see [7]), a software legacy system has to be maintained. [3] reports that more than 50% of time spent on maintenance is devoted to comprehension activities. For this reason, [10] proposes to facilitate the process of comprehending programs as an approach to improve software maintenance. Reverse engineering provides a direct attack on the program comprehension problem, but it is difficult, because it must bridge the gap between high-level descriptions and solutions on

source code level and the gap between the documented and the actual structure of the software architecture, which may have diverged over time. Architecture recovery supports the process of program understanding for software maintenance and system evolution. Ideally, the documentation of the architecture should be complete and kept up-to-date, but this is rarely the case because of undocumented changes, system evolution, or employee turnover. The description of the software architecture has to be recovered. An approach is presented in the following sections to attain these goals and towards the bridging of the gaps. It is called “Design Hypothesis” and aims at the achievement of an up-to-date high-level design model.

7.2 Context

This approach was developed in a diploma thesis ([7]) and validated in a case study performed in close cooperation with an industrial partner. Feedback and comments of the developers at the industrial partner have been integrated into the second version of this approach, which is presented here. The source code of a firmware controlling automated integrated circuit tester was analyzed to construct the module view (see [4]) of the software system. The firmware of the industrial partner has been evolved over approximately 15 years. Since then the firmware was ported to new hardware requirements, adjusted to new versions of the operating system, and changed several times due to enhancements and enlargements of the functionality. These maintenance activities became very complex and quite expertise-based, and hence very expensive. Up to the present the firmware consists of about 500 KLOC, more than 1,2 million LOC with comments.

7.3 Approach

The main idea of this top-down approach is that the reverse architect starts with a quite inaccurate view on the software architecture. His impression is then iteratively validated and refined with several reverse engineering methods, until a stable state of the module view is reached. Figure 7.1 shows an overview about the “Design Hypothesis” approach in IDEFØ notation [5]. The reverse architect makes an assumption of the structure of the software system under investigation. The assumption involves of several logical modules and the relations among them representing the software, mapped to source code entities, e.g. files, routines, variables. This high-level design model of the software is compared to the source code with the reflexion model technique [9]. To support the mapping activities, a tool was developed, that allowed the reverse architect to facilitate the assignments. For computing the reflexion model, the jRMTool [6] is used. During several iterations the high-level design model and the mappings to source code entities are refined further until the model is in a stable state. The refinement employs several tools: The Bauhaus tool [1] provides the reverse architect with information about the source code, e.g. the call graph, the declaration of function and data types, the dominance tree, part type relations, etc. CodeSurfer [2], a tool using program analysis techniques for providing the user with intra- and interprocedural slicing and pointer analysis can also be helpful when revising the mapping. Xrefactory [12] is used for fast navigation in the source code. The reverse architect can find out about the declaration and all other reference in the source code of mapped entities. Based on the structural information gained through the analysis the reverse architect successively adjusted his impressions of the system, so it became more and more precise. The purpose of this approach is the construction of an up-to-date module view on the software system. Based on this model, further steps in the maintenance, refactoring or reengineering process can be executed or the documentation can be improved or validated. This approach allows the decomposition of a software system into a hierarchy of subsystems. However, this is not yet supported by the jRMTool. It can only visualize one level of a hierarchy at a time. Therefore an enhancement of this tool or another form of visualization is imaginable and should be addressed in the future.

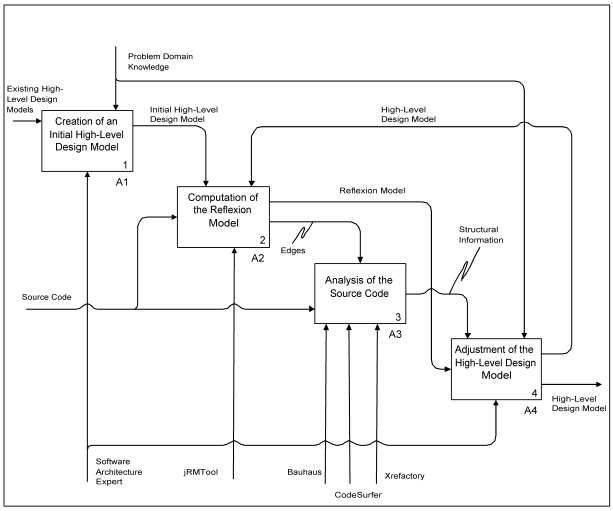


Figure 7.1. The “Design Hypothesis” Approach

7.4 Lessons Learned

The approach helped to identify subsystems and the relations among them. For this reason, the reverse architect can see, which of them have to be handled with caution due to a high coupling. If a developer wants to change the source code in a subsystem, he is now aware of possible side effects. The awareness among the developers to take preventive actions was raised. It became obvious that the restructuring activities should start with a pilot project concerning only a small part of the system, because restructuring the whole system at once is a nearly impossible task due to the high amount of dependencies between

the source code elements. The bigger the subsystems are, the more imprecise and less accurate the mappings become due to the inherent complexity of the system. So the restructuring has to be done step by step. The results for analyses in the small lead to very useful information about the software architecture. Insights can be gained, that ease the work of the developers. Several analyses of smaller parts can be combined to reconstruct a view on the whole system.

The role of the expert is very important. His knowledge is needed, because the reverse architect normally cannot decide, whether a routine or a variable is to be mapped to one module or another. It has become clear, when working with a system as large as the firmware, the reverse architect has to cooperate with several experts since it is impossible for one person to have in-depth knowledge of all parts of the software system.

The mapping done manually with editors or with shell scripts was not sufficient and took quite a long time, therefore, a tool was developed to accelerate the mapping procedure, and hence, to improve the efficiency of the approach. The combination of manual mappings for regular expressions, the mapping tool for single source code elements like (directories, file, routines, etc.) and the shell-scripts for the automatic repetition of mappings with slight changes lead to a noticeable faster performance of the mapping activity than before.

The developer of the industrial partner could apply the approach almost alone just after a short time of introduction. It is nearly no training needed, neither for using the tools, nor for performing the activities of the approach. This leads to a fast production of results.

7.5 Conclusion

The outcomes of this work were a couple of views on specific pars of the firmware and the whole system. These results showed that the “Design Hypothesis” approach allows a company to make sound statements about the software architecture of their software. Subsystems have been detected, modules have been identified, and the relations among them have been revealed. The developers at the industrial partner stated that the case studies provided them with useful, insightful, interesting results.

Bibliography

[1] Projekt Bauhaus: `http://www.bauhaus-stuttgart.de/`

[2] CodeSurfer, `http://www.grammatech.com/products/codesurfer/`

[3] R. K. Fjelstad, W. T. Hamlen: Application Program Maintenance Study: Report To Our Respondents, Tutorial on Software Maintenance, IEEE Computer Society, April 1983

[4] Christine Hofmeister, Robert Nord, Dilip Soni: Applied Software Architecture, Addison-Wesley, Reading MA, 2000

[5] IDEF0 Notation: `http://www.idef.com/idef0.html`

[6] jRMTool: `http://www.cs.ubc.ca/~murphy/jRMTool/doc/`

[7] J. Knodel: Process Models for the Reconstruction of Software Architecture Views, Diplomarbeit, Universität Stuttgart, Juli 2003

[8] M.M. Lehman, L. Belady: Program Evolution, Processes of Software Change, Academic Press, London, 1985

[9] Gail C. Murphy, David Notkin, Kevin Sullivan: Software Reflexion Models: Bridging the Gap between Source and High-Level Models, Proceedings of SIGSOFT´95, ACM Press, New York, 1995, pp. 18-28

[10] Spencer Rugaber: White Paper on Reverse Engineering, Software Engineering Research Center, Georgia Institute of Technology, Atlanta, March 1994, `http://www.cc.gatech.edu/reverse/repository/white_paper.ps`

[11] Ian Sommerville: Software Engineering, Third Edition, Addison-Wesley, 1989

[12] Xrefactory, `http://www.xref-tech.com/`

8 Application of Feature Modeling for Architecture Recovery

Ilka Philippow, Ilian Pashov, Matthias Riebisch
Technical University of Ilmenau, Max-Plank-Ring 14, P.O. Box 100565, 98693 Ilmenau,
{IlkaPhilippow,IlianPashov,MatthiasRiebisch}@tu-ilmenau.de

8.1 Introduction

The available evidence in a legacy software system often is not sufficient for its understanding and recovery. In most cases the software documentation is outdated and poor. It is possible to argue that the most reliable information is in the source code. Nevertheless a significant knowledge about the problem domain is required to improve the facility for extraction of useful architectural information. In this paper is proposed an approach for applying system domain knowledge for program understanding. This approach determines an architecture recovery process initiated at the level of domain knowledge and supported with feature models.

8.2 State of the Art

Several methods and approaches from the field of forward and reverse engineering have to be integrated into an architecture recovery process.

Feature Modeling

Originally feature modeling was introduced by the FODA methodology [Kang et al. 1990] for structuring domain properties from the view of custumers. [Czarnecki et al. 2000] has extended feature models by constrains and relations. In [Riebisch et al. 2002] these relations are extended with formal described constrains. The method FORM [Kang et al. 1998] describes how to use feature models for domain architectures and reusable components. But reverse engineering needs a more general separation of feature spaces than the one offered by FORM (see 8.3).

Architecture Recovery

Architecture recovery requires a well defined process and tool support. There are several promising approaches. Considering the integration of domain knowledge the Rigi [Storey et al. 1997] system has to be mentioned. Rigi provides two approaches for presenting software structures in its graph editor: firstly, to display the structure trough multiple, individual windows and secondly, (simple hierarchical multi-perspective view) fisheye views of nested graphs. It offers an open environment and can be extended for a tool based feature oriented recovery. In [Riva et al. 2002] this idea is discussed but up to now there is no concept for analyzing and presenting of features.

Program Comprehension

In most cases program comprehension is based on source code analysis. [Brooks et al. 1983] connect for the first time program understanding with domain knowledge. In [Letovsky et al. 1986] Brooks ideas are elaborated. The authors in [Rajlich et al. 1994] take the top-down-program understanding techniques one step further: programmers create a chain of hypotheses along with subsidiary hypotheses. These are verified in the code. Similar hypotheses are used in the here proposed approach (see 8.3). A recent discussion of domain knowledge and program understanding in [Rugaber et al. 2000] describes a number of ways for presenting domain knowledge. In this paper they are extended by feature modeling as a bridge to later architecture development.

8.3 Feature Model Based Architecture Recovery

In the software life cycle design objectives and design decisions are involved, that can be used for architecture recovery. This approach achieves this through splitting the features into two spaces (problem space and solution space) and establishing the corresponding feature model. The architecture recovery process consists of four major activities: Requirements and Domain Analysis, Legacy Architecture Analysis, Architecture Recovery by Hypotheses and Scenario Driven Dynamic Analysis.

Requirements and Domain analysis

Functional requirements represent design objectives. They have to be elaborated with domain experts and represented in a design objectives feature model.

Requirements have to be refined by case descriptions and scenarios that are later used during hypotheses assessment step and for verification and dynamic descriptions.

Legacy architecture analysis

The legacy architecture analysis is based on existing documents that are studied with comparison to requirements form the previous step. Knowledge about the domains reference architecture and pattern is included as a later source of hypotheses. The results are collected in a design decisions feature model. The nodes in the hierarchy represent solutions (structures, pattern and others). The design decision model also serves as a hypotheses stock.

Architecture recovery by hypotheses determination and verification (static architecture description)

Both, design objectives feature model and design decisions feature model are created iterative and serve as stock for hypotheses that must be formulated and verified. A hypothesis describes a supposed relationship between a feature and an architectural element (e.g. interface, class, component, method and other). Hypotheses are collected in cross-reference-tables with references "Features to Architectural elements" and "Features to Source code".

The legacy source code is analyzed in a conventional way. The results of the analysis refine hypotheses through architectural diagrams. Hypothesis can lead to new one. A hypothesis verification can fail due to an invalid feature-architectural element relation or a non-present feature. If there is no feature corresponding to a hypothesis then the hypothesis is considered to be wrong. A feature without any corresponding architectural element could be obsolete, invalid or optional. If none of this is true than obviously there is a new missing hypothesis. All verified hypothesis are stored in the cross reference table and serve for static architecture description.

Scenario driven dynamic analysis

The scenarios from step 1 are used for analyzing the dynamic aspects by common coverage analyzers and profilers for the description of the dynamic behavior. Behavior information can be used for the construction of test cases.

8.4 Conclusions

The presented architecture recovery approach shows that problem domain knowledge plays a significant role for program understanding and architecture recovery. Feature models act as bridge between requirements and architecture. Finally, they serve as a mean of generation and verification of architecture hypothesis.

Currently the application of the approach is supported by a configuration of several tools. Relational databases are used for maintaining cross-reference lists. They are connected to visualisation tools and editors using XML. The introduced recovery method was applied for an industrial project with the Postal Automation Division of the Siemens Dematic AG. Up to now the approach mainly covers the analysis and recovery of static software architectures. The dynamic aspects are considered as future work. Furthermore, research has to be directed to:

1. Elaboration and stricter representation of feature models
2. Scaling the process for large systems by decision support by metrics
3. Managing the cases of exiguous domain knowledge
4. Recovering the architecture execution view

Bibliography

[Brooks et al. 1983] Brooks, R.: Towards a Theory of the Comprehension of Computer Programs. Intl. J. Man-Machine Studies 18, 6 (June 1983)

[Czarnecki et al. 2000] Czarnecki, K., Eisenecker, U.W.: Generative Programming. Addison Wesley, Reading, MA, 2000

[Kang et al. 1990] Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-021, SEI Institute, Carnegie Mellon University, Pittsburgh, 1990

[Kang et al. 1998] Kang, K. C., Kim, S., Lee, J., Kim, K., Shin, E., Huh. M.: FORM: A feature-oriented reuse method with domain-specific reference architectures. Annals of Software Engineering, 5:143–168, 1998

[Letovsky et al. 1986] Letovsky, S., Soloway E.: Delocalized Plans and Program Comprehension. IEEE Software 3, May 1986

[Rajlich et al. 1994] Rajlich, V., J. Doran, Gudla R.: Layered Explanations of Software: A Methodology for Program Comprehension, 3d Workshop on Program Comprehension, WPC'93, Washington, D.C., pp. 46-52, Nov. 1994

[Riebisch et al. 2002] Riebisch, M., Böllert, K., Streitferdt, D., Philippow, I.: Extending Feature Diagrams with UML Multiplicities. 6th Conference on Integrated Design & Process Technology, Pasadena, California, USA. June 23 - 30, 2002

[Riva et al. 2002] Riva, C., Rodriguez, J. V.: Combining Static and Dynamic Views for architecture reconstruction. Proc. of the Sixth European Conference on Software Maintenance and Reengineering, Budapest, March 2002

[Rugaber et al. 2000] Rugaber, S.: The use of domain knowledge in program understanding. Annals of Software Engineering, 2000

[Storey et al. 1997] Storey, M. D., Fracchia F.D., Müller H. A.: Rigi: A Visualization Environment for Reverse Engineering. Proc. of International Conference on Software Engineering (ICSE'97), Boston, U.S.A., May 17-23, 1997

9 Automatisierte Ermittlung von Subsystemschnittstellen

Kay Schützler, Tobias Thiel
Humboldt-Universität zu Berlin, {schuetzl,thiel}@informatik.hu-berlin.de

9.1 Einleitung

Am Institut für Informatik der Humboldt-Universität zu Berlin wird im Rahmen eines studentischen Projekts (Projekt "Softwaresanierung": [1, 4]) ein Reengineering an einer Steuerungssoftware durchgeführt. Die in C++ implementierte Software wird am Institut für Physik der Humboldt-Universität zu Berlin zur Steuerung der Hardware verschiedener Messplätze bei Langzeitexperimenten eingesetzt.

Die Beschäftigung mit den Quellen der Software offenbarte große Defizite seitens des Originalentwicklers bei der Definition von Subsystemen und ihren Schnittstellen. Nach der Aufteilung der Quellen in einzelne Subsysteme [2] wurde klar, dass nicht alle über Header-Dateien veröffentlichten Teile eines Subsystems auch zu dessen Schnittstelle gehören. Nicht klar ist jedoch, welche der Elemente tatsächlich nicht dazugehören und somit im Subsystem gekapselt werden können. Gesucht sind also die tatsächlich benutzten Schnittstellen der Subsysteme.

Da im Projekt das Reengineering-Tool SNiFF+ [3] bereits erfolgreich eingesetzt wurde, entstand der Wunsch, die von SNiFF+ gewonnenen Strukturinformationen über das System für die oben beschriebene Aufgabe weiter zu verwenden. Für den Zugriff auf Strukturinformationen - und speziell auf gegenseitige Referenzen von Sprachelementen - bietet SNiFF+ ein gut dokumentiertes Java-API.

Ein am Institut entwickeltes Tool berechnet nun die Subsystemschnittstellen aus den erhaltenen Informationen. Konkret werden dazu die Referenzinformationen genutzt, welche darüber Aufschluß geben, welches Symbol an welchen anderen Stellen benutzt wird. Die Symbole eines Subsystems, die in einem anderen Subsystem referenziert werden, bilden die tatsächlich genutzte Subsystemschnittstelle.

9.2 Vorstellung des entwickelten Tools

Funktionalität

Der Einsatz des Tools setzt ein vorhandenes SNiFF+-Projekt mit entsprechender Projektdatei voraus.

Der Anwender spezifiziert bei dem anschließenden Tool-Einsatz zunächst das Verzeichnis, in dem sich die SNiFF+-Projektdatei befindet.

Im nächsten Dialog erfolgt die Angabe der Subsystemeinteilung. Dabei wird jede Quelldatei genau einem Subsystem zugeordnet. Das Tool schlägt als mögliche Aufteilung zunächst die in der Projektdatei vorgefundene physische Verteilung der Dateien auf die vorhandenen Unterverzeichnisse (die als mögliche Subsysteme betrachtet werden) vor.

Im Anschluss daran stellt das Tool eine Verbindung zu SNiFF+ her und nutzt diese zur Übertragung der Informationen. Alle zur späteren Analyse benötigten Informationen werden zwischengespeichert.

In der nun folgenden Auswertungsphase wird durch das Tool zuerst jedes Symbol genau dem Subsystem zugeordnet, in dem sich die Datei mit der entsprechenden Symboldefinition befindet. Es folgt die Auswertung der Referenzinformationen und darauf aufbauend die Ermittlung der Subsystemschnittstellen.

Die Arbeit mit dem Tool wird mit der Ausgabe der Subsystemschnittstellen abgeschlossen. Die Ausgabe erfolgt zum einen in Tabellenform auf der Benutzeroberfläche und zum anderen in Form einer Liste, die in einer Datei gesichert wird.

Architektur

Die Architektur des Tools besteht im Wesentlichen aus drei Schichten: einer Datenschicht, einer Logikschicht und einer Anwendungsschicht. Die einzelnen Komponenten des Tools können Abbildung 9.1 entnommen werden. Grau hinterlegt ist die externe SNiFF+-Anwendung, deren Funktionen über das erwähnte Java-API genutzt werden.

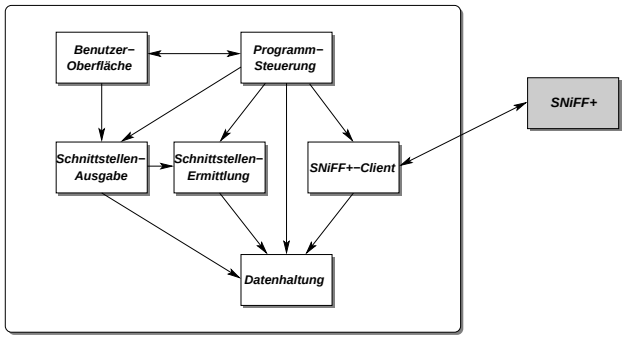


Abbildung 9.1. Architektur des entwickelten Tools

Stand der Entwicklung

Derzeit liegt bereits eine erste nutzbare Version des Tools vor. Abbildung 9.2 zeigt die aktuell implementierte Benutzeroberfläche, die sich prinzipiell bewährt hat und nur noch einiger weniger Ergänzungen bedarf.

Auch die gewünschte Funktionalität ist größtenteils bereits implementiert. Dabei sind unter Anderem die nachfolgend beschriebenen Probleme entdeckt worden, die teilweise noch zu behandeln sind.

Die von SNiFF+ gelieferten Strukturinformationen sind nicht in allen Fällen ausreichend, so liefert SNiFF+ zwar

Informationen zu Makrodefinitionen (#define), aber keine Referenzinformationen für diese Makros. Daher muss die Verwendung der Makros selbst ermittelt werden.

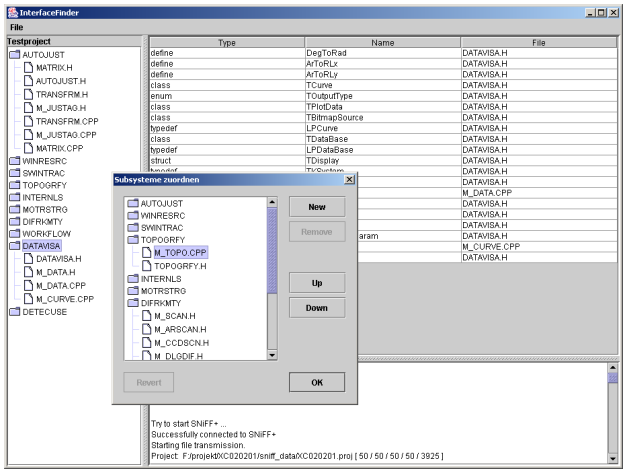


Abbildung 9.2. Screenshot des Hauptfensters mit Dialog zum Bearbeiten der Subsystemaufteilung

Dazu werden zunächst vom Tool die Makrodefinitionen gespeichert. Im folgenden Schritt werden alle Quelltextdateien gescannt und nach den Makronamen gesucht. Wird auf diese Weise eine Makrobenutzung gefunden, so wird diese wie eine normale Referenz behandelt. Bei dieser Art der Makrobehandlung besteht allerdings noch das Problem, dass zwei verschiedene Makros mit dem selben Namen, die in verschiedenen Übersetzungseinheiten definiert wurden, nicht unterschieden werden können.

Eine weitere Schwierigkeit stellt das Schlüsselwort `extern` dar, das bei Variablen beachtet werden muss. Da SNiFF+ `extern` ignoriert, muss auch bei der Behandlung von Variablen auf den Quelltext zurückgegriffen werden. Ist die Variable im Quelltext als `extern` bezeichnet, so werden alle Referenzen auf diese Variable als Referenz auf die an anderer Stelle stehende Definition behandelt.

Da das Tool vorrangig für die Bedürfnisse im Projekt ”Softwaresanierung” entwickelt wurde, richtete sich die Aufmerksamkeit zunächst hauptsächlich auf die in den vorliegenden Quellen auch tatsächlich vorzufindenden C++-Elemente. Daher erfolgt durch das Tool in seiner jetzigen Form auch noch keine Behandlung von Namespaces und damit verbundenen Aspekten der Sichtbarkeit von Schnittstellen-Elementen.

9.3 Ausblick

Die Arbeit mit dem Tool in seiner derzeit vorliegenden Form hat bereits Erweiterungsmöglichkeiten aufgezeigt.

Es wäre in jedem Fall wünschenswert, auch Namespaces zu behandeln, um die Anwendungsmöglichkeiten bezüglich der zu untersuchenden Quelltexte zu erweitern.

Derzeit erfolgt über das SNiFF+-API die Auswertung von statischen Analyse-Informationen, was beispielsweise beim Vorhandensein von Vererbungshierarchien mit polymorphen Zugriffen zu konservativen Strategien bei der Schnittstellenermittlung zwingt. Um hier genauere Aussagen über die tatsächlich benutzten Schnittstellenelemente treffen zu können, wäre die Einbeziehung dynamischer Analyse-Informationen wünschenswert.

Im Falle des XCTL-Projekts steht eine andere Erweiterung des Tools im Vordergrund. Die Funktion zur freien Festlegung der Zuordnung von Quelldateien zu Subsystemen soll durch eine Möglichkeit zur physischen Speicherung der vorgenommenen Aufteilung ergänzt werden. Damit würden die Möglichkeiten zur Untersuchung der Vor- und Nachteile verschiedener Subsystemaufteilungen sinnvoll erweitert.

Literaturverzeichnis

[1] Bothe, Klaus, *Reverse Engineering: the Challenge of Large-Scale Real-World Educational Projects*, Proc. of 14th Conference on Software Engineering Education and Training, Charlotte, USA, 2001

[2] Schützler, Kay, *Manuelle Subsystemextraktion anhand von Use-Cases*, 4. Workshop Software Reengineering, Bad Honnef, 2002

[3] SNiFF+: Reengineering-Tool, *Kurzdarstellung des Herstellers (Wind River International)*, http://www.windriver.com/products/sniff_plus/

[4] *Web-Site des Projekts ”Softwaresanierung” am Institut für Informatik*, http://www.informatik.hu-berlin.de/swt/projekt98/

10 Referenzschemata im Reverse Engineering

Andreas Winter
Universität Koblenz-Landau, Institut für Softwaretechnik, D-56016 Koblenz,
winter@uni-koblenz.de

10.1 Motivation

Die Entwicklung und Weiterentwicklung von Softwaresystemen erfordert umfangreiche und detaillierte Kenntnis unterschiedlichster Aspekte des Systems. Unabhängig, ob Softwaresysteme nach klassischen oder agilen Vorgehensmodellen entwickelt werden, wird das Verstehen vorhandener Softwaresysteme zur immer zentraleren Aktivität der Software-Entwicklung. Zur Unterstützung dieser Aktivität wurden in den letzten Jahren vielfältige Reverse-Engineering-Techniken und Werkzeuge entwickelt. Reverse-Engineering Werkzeuge basieren in der Regel auf einer Repository-basierten Architektur [3]. *Extaktoren* übertragen die Problem-relevanten Fakten aus Quelltexten in ein *Repository*. Analysewerkzeuge (*Abstraktoren*) untersuchen, verändern oder ergänzen diese Daten. Analyseergebnisse werden mit Hilfe geeigneter Visualisierungswerkzeuge (*Visualisierer*) präsentiert.

Die Synchronisierung dieser Einzelkomponenten erfolgt über das gemeinsame Repository. Dessen Struktur legt fest, welche Fakten über das Softwaresystem bereitgestellt werden und welche Analysen hierauf ausgeführt werden können. Jeder Problemkontext, der durch das jeweilige Software-Entwicklungsproblem, die hierbei zu betrachtenden Programmier- und Modellierungssprachen, sowie die jeweils eingesetzten Reverse-Engineering-Techniken geprägt ist, erfordert hierbei jeweils *individuell definierte Repository-Strukturen*.

Die Definition solcher Repository-Strukturen durch *Schemata* und der hierauf definierten Dienste ist eine wesentliche Aufgabe zur Entwicklung von Reverse-Engineering-Werkzeugen. Aufgrund der vielfältigen Aspekte, die bei der Entwicklung dieser Schemata zu beachten sind, ist sowohl die Entwicklung solcher *Problem-angemessener Schemata* als auch die Entwicklung passender Extraktoren zeitaufwändig und teuer.

Abhilfe schafft eine Sammlung geeigneter *Referenz-Schemata*, die jeweils für einzelne Problembereiche etablierte und bewährte Schemata anbieten.

10.2 Standard- und Referenz-Schemata

Standards beschreiben allgemein akzeptierte und festgeschriebene Lösungen für bestimmte Probleme. Eine Sammlung von *Standard-Schemata* im Reverse Engineering bietet eine fest definierte Menge vollständig definierter Schemata für *alle* Reverse-Engineering Probleme. Eine solche Sammlung muss hierbei die unterschiedlichesten Programmiersprachen aus verschiedenen Programmierkulturen auf diversen Granularitäts-Ebenen und unterschiedlichste Notationen zur Visualisierung berücksichtigen. Aufgrund der Heterogenität der Reverse-Engineering Probleme und der fehlenden Flexibilität von Standards scheint die Entwicklung einer solchen allumfassenden Standard-Schema-Sammlung ein unlösbares Unterfangen.

Referenz-Schemata bieten einerseits die benötigte Stabilität, indem sie die charakteristischen Eigenschaften einer Klasse von Schemata definieren. Andererseits bieten sie auch die nötig Flexibilität, indem sie durch wenige, einfache Anpassungen (Löschen, Spezialisieren und Ergänzen von Konzepten) an konkrete Anforderungen eines Anwendungsbereichs angepasst werden können. Referenz-Schemata sind folglich genereller aber auch unvollständiger als Standard-Schemata.

Zentrale Dokumente der Software-Entwicklung sind der Programm-Quelltext und diverse visuelle Modelle. Diese Artefakte bestimmen auch die grundlegende Struktur der *Referenz-Schemata im Reengineering*. Abstraktoren basieren i. Allg. auf Repräsentationen des Quelltexts in unterschiedlicher Granularität. Visualisierer setzen i. Allg. auf Strukturen auf, die durch die zur Visualisierung verwendeten Notationen determiniert ist. In beiden Bereichen lassen sich diverse *elementare Aspekte* identifizieren, die in Schemata für unterschiedliche Problembereiche in leicht modifizierter Form berücksichtigt werden. Die charakteristischen Eigenschaften dieser elementaren Aspekte werden durch *Schema-Moleküle* beschrieben. Referenz-Schemata für einzelne Problembereiche sind aus diesen Schema-Molekülen und vorhandenen Referenzschemata zusammengesetzt.

10.3 Programmiersprachen-Sicht

Quelltext ist oft die einzige verlässliche Informationsquelle zu einem bestehenden Softwaresystem. Im Reverse-Engineering sind die Strukturen dieser Softwaresysteme zu erkennen und zu visualisieren.

Reverse-Engineering-Werkzeuge benötigen Repräsentationen der unterschiedlichsten Programmiersprachen aus verschiedenen Programmierkulturen auf unterschiedlichen Abstraktionsebenen. Schemata dieser Repräsentationen können entlang wesentlicher Sprachmerkmale klassifiziert werden. Je nach verwendeter Programmiersprache sind hierzu u. A. *Aufrufstrukturen*, *Ausdrücke*, *Enthaltenseinsstrukturen (Include)*, *Funktions/Methoden-Deklarationen*, *Klassen-Definitionen*, *Kontrollstrukturen und Anweisungen*, *Nebenläufigkeit*, *Objekt/Variablen-Deklaration*, *Präprozessor-Anweisungen*, *Templates*, *Typ-Definitionen* zu berücksichtigen. Abhängig von der, der Programmiersprache zugrunde liegende Programmierkultur, sind diese Kernaspekte durch Schema-Moleküle in unterschiedlichen Ausprägungen zu beschreiben. Beispielsweise erfordert die Modellierung der Kontrollstrukturen in Cobol, Lisp, oder Pascal unterschiedliche Ausprägungen. Referenzschemata zur Darstellung dieses Aspekts für C++ oder Java sind dagegen weitgehend ähnlich.

Referenz-Schemata für einzelne Problembereiche setzen sich dann aus geeignet angepassten Schema-Molekülen für die jeweils genutzten Aspekte in ihren jeweiligen Ausprägungen zusammen. So besteht z. B. das Columbus C++ Schema [2], das als Ausgangspunkt für ein C++ Referenz-Schema genutzt werden kann, aus Komponenten zur Modellierung von Ausdrücken (expr), Klassendefinitionen (struc), Kontrollstrukturen und Anweisungen (stmt), Templates (templ), und Typdefinitionen (Type).

10.4 Modellierungssprachen-Sicht

Informationen über Softwaresysteme werden sowohl zur initialen Erstellung als auch während der Weiterentwicklung durch visuelle Modellierungsmitteln beschrieben.

Ausgehend von geeigneten Repräsentationen der vorliegenden Quelltexte, werden im Reverse Engineering Softwarezusammenhänge z. B. durch Klassendiagramme, Datenfluss-Diagramme, Aktivitätsdiagramme, oder Sequenzdiagramme visualisiert. Diese visuellen Beschreibungsmittel stellen unterschiedliche dynamische und statische Aspekte von Software-Systemen in den Mittelpunkt: u. A. *Datenflüsse*, *Klassendefinitionen*, *Klasseninteraktionen*, *Kontrollflüsse*, *Nachrichtenaustausche*, *Objektinteraktionen*, *Systemzustände und Übergänge*. Diese Kernaspekte visueller Modellierungsmittel sind ebenfalls durch Schema-Moleküle zu beschreiben. Geeignetes Zusammenfassen, evtl. angepasster Schema-Moleküle der jeweils benötigten Modellierungsaspekte, liefert Referenz-Schemata konkreter Modellierungssprachen (also *Referenz-Metaschemata*). Entlang gemeinsam dargestellter Konzepte bzw. Aspekte lassen sich diese Referenz-Metaschemata zu integrierten Referenz-Metaschemata gruppieren [4]. Je nach Implementierungsnähe überlappen sich diese Schema-Moleküle mit den Schema-Molekülen aus Sicht der Programmiersprachen (vgl. den Aspekt *Klassendefinition*).

10.5 Abgeleitete Referenz-Schemata

Referenz-Schemata bieten einerseits mit Schema-Molekülen aus Programmier- und Modellierungsprachen-Sicht bzw. mit jeweils sprach-spezifischen Schemata eine Sammlung bewährter Schemata zur Lösung klassischer Reverse-Engineering-Probleme. Sie bieten andererseits aber auch eine Bausteinsammlung, aus der, durch geeignetes Anpassen und Zusammenfassen einzelner Bausteine, weitere Schemata abgeleitet werden können.

So lassen sich beispielsweise Schemata zur Beschreibung von *Software-Architekturen* entlang bekannter Architekturstile [1] durch Variantenbildung und Komposition aus Schema-Molekülen und Referenz-Schemata ableiten: Schemata zur Beschreibung von Software-Architekturen nach dem *Pipe-Filer-Style* erfordern Konzepte zur Modellierung von Prozessen (Filter), die Daten verändern, und Datenflüssen (Pipe), die den Datenaustausch zwischen Prozessen beschreiben. Ein solches Schema entspricht einem stark vereinfachen Metaschema zur Modellierung mit Datenfluss-Diagrammen, und lässt sich leicht aus einem Schema-Molekül zur Beschreibung von Datenflüssen ableiten. Ist eine breite Verwendbarkeit eines solchen Schemas erwiesen, kann dieses auch in die Sammlung der Referenz-Schemata übernommen werden.

10.6 Zusammenfassung

Referenz-Schemata im Reverse Engineering liefern vorgefertigte Standardlösungen für wiederkehrende Reverse-Engineering-Probleme, bestehend aus Datenstrukturen und den hierzu passenden Extraktor- und Analysediensten. Referenz-Schemata beschreiben charakteristische Eigenschaften dieser Lösungen und sind durch wenige Änderungen an weitere Reverse-Engineering-Probleme anpassbar.
Die Bereitstellung eines solchen Referenz-Schema-Werkzeugkastens erfordert neben der weiteren Entwicklung einzelner Schema-Moleküle und Referenz-Schemata, auch die Entwicklung einer Methodik zur Verwendung dieser Schemata. Diese umfasst u. A. die Bereitstellung von Operationen zur Anpassung und Integration von Schemata und deren Übertragung auf entsprechende Instanz-Daten.

Literaturverzeichnis

[1] L. Bras, P. Clements, R. Kazman. *Software Architecture in Practice*. Addison-Wesley, Boston, 1998.

[2] R. Ferenc, F. Magyar, Á. Beszédes, Á. Kiss, M. Tarkiainen. Columbus - Tool for Reverse Engineering Large Object Oriented Software Systems. In *Proceedings SPLST 2001, Szeged, Hungary*, pp 16–27. June 2001.

[3] C. Lange, H. Sneed, A. Winter. Comparing Graph-based Program Comprehension Tools to Relational Database-based Tools. in 9th International Workshop on Program comprehension. IEEE, Los Alamitos, pp 209–218. 2001.

[4] A. Winter. *Referenz-Metaschemata für visuelle Modellierungssprachen*. DUV, Wiesbaden, 2000.

11 Aspect Mining Using Dynamic Analysis

Silvia Breu, Jens Krinke
Lehrstuhl Softwaresysteme, Universität Passau, breu@infosun.fmi.uni-passau.de

11.1 Motivation

Concerns express a specific interest in some topic regarding a particular system of interest. *Separation of concerns* (originally invented by Dijkstra) is essential in the software development process: It is an important paradigm in software engineering to cope with the increasing number of special purpose concerns in today's applications. To deal with that increasing complexity, several new approaches like Composition Filters, Hyperslices and last but not least *Aspect-Oriented Programming* [3] (including programming languages like AspectJ) have been proposed. But what about legacy systems, where separation of concerns could only be applied in a restricted way within the object-oriented paradigm? It is possible to find *aspects* and to encapsulate them without changing software behavior, improving maintainability and re-usability, reducing tangled and scattered code. This is illustrated in the following sections.

11.2 Dynamic Analysis

This short report will present a first approach to dynamically analyze existing Java software to find aspects and crosscutting concerns. For this purpose, different general classes of aspects have been defined:

1. *Outside-Aspects* are patterns where a call of method `a` is always followed by a call of method `b`.
2. *Inside-Aspects* are patterns where a call of method `b` is always inside a call of method `a`.

These two main classes of aspects have been split further, each in two more subclasses:

Outside-Aspect can either be *before* or *after* a specific method call.

Inside-Aspect can either be *first-* or *last-in* a specific method call.

Realize that before- and after-set of aspects in a certain software system need not obligatory be equal, based on the assumption that we are looking for exhaustive sets. This holds for first-in- and last-in-set as well.
However, defining classes of aspects and their structure is not enough to find real aspects. To identify aspects in existing software systems, a possible approach is to generate program traces. The obtained traces are analyzed, using a tool written in Java. It searches for outside and inside aspect candidates with respect to a certain limitation:

```
1  B.a() {
2    C.d() {
3      G.h() {...}
4      H.g() {...}
5    }
6  }
7  A.b() {...}
8  B.a() {
9    C.d() {...}
10 }
11 A.b() {...}
12 B.a() {
13   C.d() {
14     G.h() {...}
15     H.g() {...}
16   }
17   C.c() {...}
18 }
19 F.f() {
20   H.i() {...}
21 }
22 C.c() {...}
23 G.h() {...}
24 H.g() {...}
25 A.b() {...}
26 B.a() {
27   C.d() {...}
 :    ...
30   F.f() {
31     H.k() {...}
32     H.i() {...}
33   }
34 }
35 D.e() {
36   C.d() {...}
 :    ...
40   A.b() {...}
41   B.a() {
42     C.d() {...}
43   }
 :    ...
50   G.h() {...}
51   E.f() {...}
52 }
```

Abbildung 11.1. Example trace

The pattern has to exist *always in the same composition*.

This constraint results from the following: A method call to `a` which is identified as being an outside aspect before a method call to `b` has always to be immediately before method calls to `b`, otherwise it is not a recurring pattern (aspect) in the system. The argumentation for outside after and inside first-/last-in aspects is analogous.

Figure 11.1 shows an example trace which shall be the basis to explain the dynamic analysis procedure and its result in more detail. Analyzing the given trace leads to the following aspect candidates:

- `G.h()` before `H.g()` (3x)
- `G.h()` before `E.f()` (1x)
- `B.a()` after `A.b()` (4x)
- `C.d()` first-in `B.a()` (5x)
- `C.d()` first-in `D.e()` (1x)
- `H.i()` last-in `F.f()` (2x)

The example shows that reversing 'before aspects' [1] does not necessarily lead to 'after aspects': `A.b()` is always followed by `B.a()` but it is not correct to say that before `B.a()` `A.b()` is always called—`B.a()` in line *1* is not preceded by `A.b()`. However, in general, we can say that iff both 'method `a` before method `b`' and 'method `b` after method `a`' applies, the number of occurrences of both

[1] In the remaining of the report only 'before', 'after', 'first-in' and 'last-in' are used to characterize the aspects as the nomenclature is clear.

patterns is the same—the aspects are *symmetric*.

Additionally, we can see that `C.d()` has no first-in or last-in aspect: Not every appearance of `C.d()` in the example trace has the same structure. In lines *2-5* as well as in lines *13-16* it would have `G.h()` as first-in and `H.g()` as last-in aspect, but in lines *9, 27, 36* and *42* it has no first-in and/or last-in aspect. Therefore, neither `G.h()` first-in `C.d()` nor `H.g()` last-in `C.d()` are added to the corresponding result sets. For similar reasons, `G.h()` after `C.c()` has not been detected because this pattern only holds for lines *22/23* but not for line *17*.

Applying a further analysis to the before-/after-/first-in- and last-in-set can now find crosscutting code candidates. Therefore, an additional constraint has to be checked:

> The found pattern has to occur in
> *more than one calling context*.

This means, that a method `a` is a first-in/last-in aspect only if it has this predicate concerning several different methods whereas a method `b` is a before/after aspect if the pattern appears more than once in the trace. Following this definition we can find several candidates for crosscutting concerns in the example:

- `G.h()` is a before aspect of `H.g()` and `E.f()`
- `B.a()` is an after aspect of `A.b()`
- `C.d()` is a first-in aspect of `B.a()` and `D.e()`

11.3 First Evaluation

For a first evaluation of the presented technique, a simple visualization tool for chopping and slicing (AnChoVis), written in Java, has been traced. The presented analysis of traces of AnChoVis runs revealed that there are method calls for a certain functionality: logging. Those calls always appear with the same pattern at identical places (as first-in and last-in aspect).

Another version of AnChoVis without the logging functionality scattered throughout the code has been created. The logging functionality has been implemented as simple aspect, written in AspectJ, and woven to the program. That resulted in a program version with the same behavior as before but with better understandability and maintainability of the code.

11.4 Future Work

Up to now, the analysis presented in this short report is not fully assessed and explored but first evaluations are promising.

An open question is whether and how to merge identical first-in and last-in aspects. The current analysis does not provide any information if the aspect is one and the same call inside another method or if there are two calls of one and the same method with anything (statements, other method calls) in between.

More precise information could be gained by not just tracing method calls but also statements. However, this would make the analysis more complex—a tradeoff between higher gain and precision on the one hand, reduced speed on the other hand.

Another question is whether it is worthwhile finding more complex aspects: for example summarizing chains of aspects like `a` before `b`, `b` before `c`, and `c` before `d` to one single aspect `abc` before `d`. Solutions for these points may give additional information to the question how simple it is to encapsulate located aspects properly and correctly.

Anyhow, there are even more open questions which have to be explored:

- Do case studies with bigger software systems confirm the results we have so far?
- Are there other aspects except the already well-known aspects like logging etc.?

11.5 Related Work

There are two other approaches known to the author which present techniques to identify aspects. The work of Hannemann and Kiczales [2] is a query-based Aspect Mining Tool (AMT): The query specifies a type or a regular expression; every line of an analyzed system is checked against the query and highlighted if it matches. In the second approach, Aspect Browser [1], crosscuts are identified with textual-pattern matching and highlighted in the program files' windows with a specific color.

Literaturverzeichnis

[1] William G. Griswold, Y. Kato, and J. J. Yuan. Aspect Browser: Tool Support for Managing Dispersed Aspects. Technical Report CS99-0640, Department of Computer Science and Engineering, University of California, San Diego, Dezember 1999.

[2] Jan Hannemann and Gregor Kiczales. Overcoming the Prevalent Decomposition of Legacy Code. In *Workshop on Advanced Separation of Concerns*, 2001.

[3] Georg Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *European Conference on Object-Oriented Programming (ECOOP)*, 1997.

12 Software Engineering II für Ingenieure: Wartung, Reengineering und Evolution

Tobias Rötschke, Andy Schürr
TU Darmstadt, Merckstraße 25, 64283 Darmstadt,
{tobias.roetschke,andy.schuerr}@es.tu-darmstadt.de

Abstract

In diesem Artikel stellen wir unser Konzept für eine Vorlesung „Software Engineering II“ vor, die ab dem WS2003/2004 jährlich für Studenten der Ingenieurstudiengänge „Elektrotechnik und Informationstechnik“ sowie „Informations- und Kommunikationstechnik“ gehalten wird. In dieser Hauptstudiumsvorlesung sollen die Themen „Softwarewartung, -reengineering und -evolution“ in einer den Studiengängen angemessenen Weise behandelt werden. Der Schwerpunkt soll dabei auf der Lösung von praxisnahen Problemen unter Verwendung gängiger Werkzeuge liegen.

12.1 Einleitung

Angesichts von immer komplexer werdenden Systemen, zunehmender Globalisierung und der kontinuierlichen Weiterentwicklung von Betriebssystemen, Programmiersprachen und Komponentenrahmenwerken spielen Softwarewartung, -reengineering und -evolution eine immer größere Rolle in der Softwareentwicklung, die auch vor eingebetteten Systemen nicht Halt macht.

Ziel der Vorlesung „Software Engineering II“ ist es, Studenten der Studiengänge „Elektrotechnik und Informationstechnik“ sowie „Informations- und Kommunikationstechnik“ – also keine Informatikstudenten – auf die erfolgreiche Mitarbeit in Projekten zur Weiterentwicklung von bestehenden software-intensiven eingebetteten Systemen vorzubereiten. Dabei soll unsere Vorlesung nicht speziell auf eines der folgenden Gebiete ausgerichtet sein: Übersetzerbau, Reengineering [7], Softwarequalitäts-Management [4] oder Prozessmodelle [3]. Wichtige Themen, wie in [2, 11] vorgeschlagen, sollen vielmehr möglichst breit abgedeckt werden, ohne diese jedoch im einzelnen zu vertiefen. Begleitend zur Vorlesung bieten wir eine Übung an, in welcher die Studenten anhand eines durchgängigen Beispiels Erfahrungen mit geeigneten CASE-Werkzeugen sammeln können.

12.2 Randbedingungen

Eine besondere Herausforderung ist dabei, den Stoff so zu gestalten, dass er optimal auf den Kenntnissen der Studenten aufbaut. Die meisten Studenten haben nur eine Programmiersprache gelernt (in der Regel Java) und kennen nur Microsoft Windows als Betriebssystem. Damit scheidet eine Übung basierend auf elementaren Unix-Werkzeugen wie in [13] aus. In der Vorlesung „Software Engineering I“ haben die Studenten bereits erste Erfahrungen in der Modellierung mit UML gesammelt. Die Programmierkenntnisse beschränken sich in der Regel auf Übungsaufgaben aus den Grundstudiumsvorlesungen und das Softwarepraktikum, bei welchem vor allem das Programmieren im Kleinen im Vordergrund stehen.

12.3 Gliederung der Vorlesung

Die Vorlesung gliedert sich in fünf Kapitel. Zunächst werden in der Einleitung Vorgehensweisen zur iterativen Softwareenwicklung mit dem Schwerpunkt auf Wartung und Evolution vorgestellt. Anschließend werden im Kapitel „Konfigurationsmanagement“ die Bereiche Releasemanagement, Versions- und Variantenmanagement sowie Änderungs- und Fehlermanagement behandelt. Im Kapitel „Statische Programmanalyse“ besprechen wir Softwarearchitekturen, Softwarevisualisierung, kontrollflussorientierte Analysen, datenflussorientierte Analysen und Metriken. Das Kapitel „Dynamische Programmanalyse“ deckt die Bereiche Laufzeit- und Speicherplatzverbrauch, funktionsorientierte, kontrollflussorientierte und datenflussorientierte Testverfahren sowie Testen objektorientierter Programme ab. Abschließend werden im Kapitel „Programmtransformationen“ Refactoring, reguläre Ausdrücke und Texttransformationen sowie optional kontextfreie Grammatiken und Baumtransformationen behandelt.

12.4 Beispiele

Unser Ziel bei der Suche nach geeigneten Beispielprogrammen war es, jeweils ein durchgängiges Beispiel für die Vorlesung und die Übung zu finden. Die Anwendungsgebiete sollten verschieden und potentiell interessant für unsere Studenten sein. Das Produkt sollte grundsätzlich lauffähig sein, aber genügend offensichtliche Fehler beinhalten, um daran etwas lernen zu können. Der Quelltext musste zugänglich sein und das Projekt als Ganzen nicht

zu groß, aber auch nicht trivial sein. Als Programmiersprache soll C++ verwendet werden, damit die Studenten lernen, neben Java noch einer weitere relevante Sprache zu erlernen.

Durch Verwenden der vorhandenen Filter von SourceForge.net, der wichtigsten Werkzeugplatform für Open Source Software-Entwicklung [8], gelang es uns, ein Open-Source-Projekt zu finden, welches uns für die Übung geeignet scheint: TinyCAD [9] ist ein einfaches CAD-Programm für das Zeichnen von Schaltplänen. Das Programm verwendet benutzerdefinierte Symbolbibliotheken, kann Teilelisten erstellen und erlaubt es, einfache Analysen auf dem Diagramm auszuführen. Es ist mit Visual C++ unter Verwendung des Microsoft-Foundation-Classes Rahmenwerkes geschrieben.

Neben einigen offensichtlichen Bugs enthält das Programm Speicherlöcher, ist nur oberflächlich dokumentiert und bietet Ansatzpunkte für Refactoring: Dateien enthalten mehrere öffentliche Klassen, es gibt keine sichtbare Pakethierarchie, einige Klassen sind sehr groß und haben zu viele Abhängigkeiten zu anderen Klassen.

Für die Vorlesung sind wir noch auf der Suche nach einem geeigneten Beispiel. Dieses sollte aus einem anderen Anwendungsgebiet kommen, damit die Studenten sich daran gewöhnen, sich in verschiedene Gebiete hineindenken zu müssen.

12.5 Werkzeuge

Bei der Auswahl von Werkzeugen haben wir auf eine gute Mischung von kommerziellen und frei verfügbaren Produkten geachtet, wobei deren Anzahl möglichst klein sein sollte, ohne unsere Vorlesung vollständig auf ein einziges Produkt zu stützen. Das Arbeiten mit den Werkzeugen sollte ohne großen Einarbeitungsaufwand zu Ergebnissen führen, damit die Studenten motiviert bleiben, die angegebenen Werkzeuge auch tatsächlich zu benutzen.

Als Entwicklungsumgebung verwenden wir Visual Studio, welches die Studenten bereits aus dem vorher stattfindenden C++-Kurs kennen. Für Versions,- Änderungs- und Fehlermanagement verwenden wir SourceForge.net, welches auf dem Concurrent Versions System [6] basiert ist. Wie in [5] vorgeschlagen, werden wir dieses Werkzeug auch dazu verwenden, um die Leistung der Studenten individuell zu beurteilen.

Für die Konfigurationsverwaltung verwenden wir die make-Variante „nmake“, welche zu Visual Studio gehört. Für Reverse Engineering, Visualisierung, Metriken, Dokumentation und Audits verwenden wir sowohl Doxygen [12] als auch Together [1], welches die Studenten bereits in der Übung zur Vorlesung „Software Engineering I“ und im Softwarepraktikum für Entwicklung und Modellierung benutzt haben.

Für die dynamische Analyse verwenden wir Purify, PureCoverage und Quantify, aus der Rational Suite Enterprise [10]. Obwohl dieses Produkt oft auch für andere Aufgaben wie Versionsverwaltung, Bugtracking und Reverse Engineering eingesetzt werden, hat sich die Bedienung als zu kompliziert herausgestellt, um im Rahmen einer einstündigen Übung eingesetzt zu werden.

12.6 Ausblick

Wir möchten auf dem Workshop zum einen unser Konzept zur Diskussion stellen, wobei sowohl die Interessen der Lehre als auch der Industrie deutlich werden dürften. Zum anderen wollen wir Erfahrungen über geeignete Fallbeispiele sowie Visualisierungs- und Metrikwerkzeuge austauschen.

Literaturverzeichnis

[1] Borland Software Corporation. Borland Together ControlCenter. Produktinformation, 2003. `www.togethersoft.com/products/controlcenter/`.

[2] P. Bourque and R. Dupuis. Guide to the Software Engineering Body of Knowledge (SWEBOK). `www.swebok.org/documents/Trial_1_00/Trial_Version1_00_SWEBOK_Guide.pdf`, May 2001.

[3] M. Broy. Projektorganisation und Management in der Softwareentwicklung. Vorlesung TU München, Apr. 2001.

[4] K.-P. Fähnrich. Software-Qualitätsmanagement. Vorlesung Universität Leipzig, 2002.

[5] M. Gehrke, H. Giese, U. A. Nickel, J. Niere, M. Tichy, J. P. Wadsack, and A. Zündorf. Reporting on Industrial Strength Software Engineering Courses for Undergraduates. In *Proc. 24th International Conference on Software Engineering (ICSE)*, pages 395–405. ACM Press, 2002.

[6] D. Grune. Concurrent Versions System, a Method for Independent Cooperation. Technical Report IR 113, Vrije Universiteit Amsterdam, 1986.

[7] R. Koschke. Vorlesungen zum Thema Software-Reengineering. In J. Ebert and F. Lehner, editors, *Proc. 2. Workshop Software-Reengineering*, Bad Honnef, 2000. Universität Koblenz-Landau.

[8] Open Source Development Network. *An Overview of the SourceForge.net User Community*, 2003. `sourceforge.net/docman/display_doc.php?docid=9331&group=1`.

[9] M. Pyne. TinyCAD. Open Source Project, 2002. `tinycad.sourceforge.net`.

[10] Rational Software Corporation. Rational Suite Enterprise. Produktinformation, 2002. `www.rational.com/products/entstudio/`.

[11] A. E. K. Sobel. Computing Curricula – Software Engineering Volume. `sites.computer.org/ccse/know/SecondDraft.pdf`, Dec. 2002.

[12] D. van Heesch. *Doxygen Manual*, Apr. 2003. `www.doxygen.org/manual.html`.

[13] A. Zeller and J. Krinke. *Programmierwerkzeuge: Versionskontrolle - Konstruktion - Testen - Fehlersuche unter Linux*. dpunkt Verlag, Heidelberg, 2000.

13 Extraktion statischer Objekt-Prozess-Graphen

Thomas Eisenbarth, Rainer Koschke, Gunther Vogel
Universität Stuttgart, Institut für Softwaretechnologie, Universitätsstraße 38, D-70569 Stuttgart,
{eisenbarth,koschke,vogel}@informatik.uni-stuttgart.de

13.1 Statische Prozess-Graphen

Eine *dynamische Spur (Trace)* ist eine Aufzeichnung der während einer Programmausführung durchgeführten Aktionen. Dynamische Spuren sind abhängig von der Eingabe. *Statische Spuren* hingegen enthalten alle potentiell durchgeführten Aktionen einer beliebigen Ausführung, unabhängig von Eingaben. Eine *statische Objektspur* ist eine statische Spur, die nur solche Anweisungen enthält, die sich auf ein bestimmtes Objekt beziehen. Da ein Programm prinzipiell unendlich viele *statische Objektspuren* besitzen kann, werden Mengen von *statischen Objekspuren* durch *statische Objekt-Prozesse* zusammengefasst.

Die Extraktion statischer Objekt-Prozesse ist eine Basistechnologie, die eine Reihe von neuen Anwendungen ermöglicht, wie zum Beispiel die Unterstützung von Programmverstehen durch die Reduktion des zu untersuchenden Quellcodes auf die wesentlichen Anweisungen (ähnlich wie Program Slicing), die Erkennung und Validierung von Protokollen, das Auffinden von Interaktionen zwischen Objekten (wie sie z.B. durch UML-Interaktionsdiagramme beschrieben werden) und die Bestimmung der Testabdeckung von Testfällen hinsichtlich der verschiedenen Aufrufsequenzen von Schnittstellenelementen einer Komponente.

Objekte werden zur Laufzeit entweder implizit als Teil eines Aktivierungsrecords einer Funktion oder explizit durch Belegung von Speicher auf der Halde erzeugt. Aufgrund von Rekursion und Schleifen lässt sich die Menge der Objekte im Allgemeinen nicht statisch bestimmen. Deshalb wird jedem Objekt ein statisches Objekt zugeordnet, das gleichzeitig eine potentiell unendliche Menge von Objekten repräsentiert. Für jede Deklaration einer Variablen und für jeden Aufruf einer Routine zur Speicherbelegung existiert ein statisches Objekt, für das ein *Objekt-Prozess-Graph* aus dem Quelltext extrahiert werden kann. Abbildung 13.1 zeigt die verschiedenen Knoten und Kanten von Objekt-Prozess-Graphen. Jeder Objekt-Prozess-Graph enthält genau einen Start- und Final-Knoten für den Programmstart und das Ende der Ausführung. Die Lebensdauer von lokalen und globalen Variablen wird durch Begin-of-Lifetime und End-of-Lifetime Knoten angezeigt. Lesende und schreibende Zugriffe auf Variablen und Haldenobjekte werden durch Read- und Write-Knoten repräsentiert.

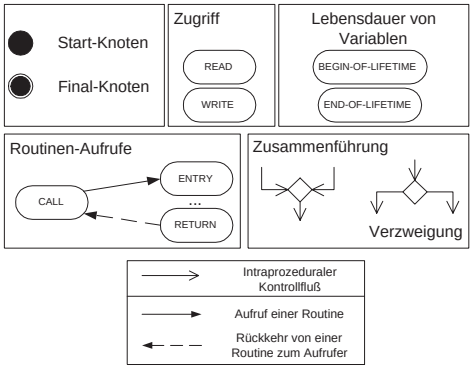


Abbildung 13.1. Objekt-Prozess-Graphen.

Für Verzweigungen und Zusammenführungen von Kontrollfluss existieren entsprechende Knoten. Call-Knoten repräsentieren Aufrufe von Routinen. Entry- und Return-Knoten zeigen den Eintritt in eine Routine oder den Austritt aus einer Routine an.

Kanten in Objekt-Prozess-Graphen repräsentieren intraprozeduralen und interprozeduralen Kontrollfluss.

Abbildung 13.2 zeigt ein kleines Beispiel eines *Objekt-Prozess-Graphen* für ein Objekt vom Typ `Stack`, welches durch den Aufruf von `init` erzeugt wird. Die Funktion `reverse` legt neue Elemente auf dem Stack ab, die dann in `main` entnommen und gezählt werden.

```
int main () {
  int i = 0;
  Stack *s1 = init ();
  Stack *s2 = read ();
  reverse (s2, s1);
  do {
    pop (s1);
    i = i + 1; }
  while (!is_empty (s1));
}
void reverse
 (Stack *from,
  Stack *to) {
  while (!is_empty (from))
    push (to,
          pop (from));
}
```

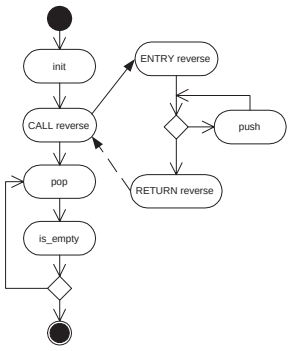


Abbildung 13.2. Objekt-Prozess-Graphen für das Objekt, das durch den Aufruf von `init` erzeugt wird.

Die primitiven Zugriffsfunktionen des Stacks werden als atomar angesehen. Die Zugriffe innerhalb dieser Funktionen sind deshalb nicht dargestellt.Würden weitere Zugriffe auf andere Objekte in den Quelltext eingebaut werden, so würde sich an dem Objekt-Prozess-Graph nichts ändern.

13.2 Aufrufpfade

Für die automatische Erkennung von Protokollen reicht die Unterscheidung von statischen Haldenobjekten durch den entsprechenden Aufruf einer Speicher belegenden Funktion allein noch nicht aus. Falls der Aufruf innerhalb einer Wrapper-Funktion eingebettet ist, können die erzeugten Objekte unterschiedliche Protokolle besitzen.

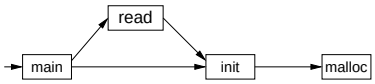


Abbildung 13.3. Aufrufgraph

Abbildung 13.3 zeigt ein kleines Beispiel eines Aufrufgraphen. Hier existieren zwei Pfade, die von `main` zu dem Aufruf der Speicherbelegungsroutine `malloc` führen. Auf den Objekten, die durch die beiden unterschiedlichen Pfade angelegt werden, werden unterschiedliche Aktionen ausgeführt. Die Aktionen der Routine `read` werden zum Beispiel nur dann ausgeführt, wenn auch der Aufrufpfad über `read` führt.

Eine Verfeinerung des ursprünglichen Objektbegriffs besteht daraus, statische Objekte durch ihre Aufrufpfade zu unterscheiden. Hierzu wird in [EKV02] eine Methode vorgestellt, die es dem Benutzer erlaubt, beliebige Sequenzen von rekursiven Funktionsaufrufen durch reguläre Ausdrücke zu beschreiben.

Eine weitere Methode, mit der es möglich ist, eine für die Protokollerkennung interessante Menge von Haldenobjekten automatisch zu identifizieren, basiert auf dem Konzept des expandierten Aufrufgraphen. Dieser enthält Knoten für sämtliche nicht-zyklischen Pfade durch den Aufrufgraphen, die mit dem initialen Aufruf an die Routine `main` beginnen. Jeder Knoten des expandierten Aufrufgraphen, dessen Pfad zu einer Speicherbelegung führt, wird zur Identifikation eines Objekts herangezogen. Weitere Einzelheiten zu dieser Methode werden in Bälde veröffentlicht.

13.3 Extraktion von Objekt-Prozess-Graphen

Für Programme, die Zeiger verwenden, ist zunächst eine Zeiger-Analyse notwendig, um alle potentiellen Referenzen auf das relevante Objekt zu bestimmen. Zeiger-Analysen werden in flusssensitive/-insensitive und monovariante/polyvariante Analysen eingeteilt.

Unsere Technik der Extraktion von statischen Spuren ist unabhängig von der zu Grunde liegenden Zeiger-Analyse. Allerdings werden die extrahierten Objekt-Prozess-Graphen umso genauer, je genauere Zeigeranalysen verwendet werden. Die besten Ergebnisse werden mit einer flusssensitiven, polyvarianten Analyse erzielt.

Sobald alle Referenzen auf das relevante Objekt bestimmt sind, kann eine Projektion des Kontrollflusses auf die relevanten Anweisungen (jene, die sich auf das gegebene Objekt beziehen) ermittelt werden. Die Projektion wird in zwei Phasen ermittelt. In der ersten Phase wird der Kontrollflussgraph traversiert und Knoten für die Aktionen auf dem relevanten Objekt erzeugt. Die zweite Phase verbindet die Knoten durch Kontrollflusskanten. In dieser Phase werden zusätzlich noch weitere Knoten für Verzweigungen eingefügt, falls bestehende Knoten von einer Bedingung kontrollabhängig sind.

13.4 Fallstudien

Erste Ergebnisse, die mit einem Werkzeug auf der Basis einer flusssensitiven und polyvarianten Zeiger-Analyse nach Wilson [Wil97] ermittelt wurden, zeigen, dass die extrahierten Graphen im Vergleich zum ursprünglichen Kontrollflussgraphen des Programms deutlich kleiner sind. So wurde beispielsweise *Concepts* [Lin94], ein Programm zur formalen Begriffsanalyse, das aus etwa 8000 Zeilen C-Quellcode mit 143 Routinen besteht, analysiert. Der Kontrollflussgraph enthält 21.410 Knoten mit 597 Funktionsaufrufen. Es existieren 74 Aufrufe von „`malloc`“ zur Belegung von Haldenspeicher.

Für die 549 lokalen und globalen Variablen wurden Objekt-Prozess-Graphen extrahiert. Die mittlere Anzahl von Knoten pro Graph betrug 22 (min: 5, max: 590). Es wurden insgesamt 770 unterscheidbare Halden-Objekte gefunden. Die Prozess-Graphen der Haldenobjekte hatten eine mittlere Größe von 320 Knoten (min: 12, max:789).

Auf einem PC mit Athlon XP 1700+ Prozessor und 1GB RAM unter Linux dauerte die Extraktion der Prozess-Graphen für die lokalen und globalen Variablen lediglich 13 Sekunden. Für die Haldenobjekte wurden 197 Sekunden benötigt, da oftmals der gesamte Kontrollflussgraph durchsucht werden musste.

Details über weitere Untersuchungen, bei denen auch eine flussinsensitive Analyse nach Steensgaard [Ste96] eingesetzt wird, werden in Kürze veröffentlicht.

Literaturverzeichnis

[EKV02] Thomas Eisenbarth, Rainer Koschke, and Gunther Vogel. Static trace extraction. pages 128–138, 2002.

[Lin94] Christian Lindig. Concepts 0.3e. Available at URL http://www.eecs.harvard.edu/ lindig/src/concepts.html, 1994.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[Wil97] R.P. Wilson. *Efficient, Context-Sensitive Pointer Analysis*. PhD thesis, Department of Electrical Engineering, Stanford University, December 1997.

14 Strukturelle Analyse explizit fehlertoleranter Programme

Stefan Gossens, Mario Dal Cin
Universät Erlangen-Nürnberg, Germany
Institut für Informatik, Lehrstuhl 3 (Rechnerarchitektur)
{gossens,dalcin}@informatik.uni-erlangen.de

Zusammenfassung

Explizit fehlertolerante Programme zeichnen sich durch proaktive Massnahmen aus, die deren Robustheit und Korrekturfähigkeit gegenüber Fehlern sichern sollen. In einer gewollt fehlertoleranten Anwendung wird meistens eine von mehreren Standardachitekturen realisiert. Durch graphbasierte Methoden kann man aus bestehenden Anwendungen eine Kontrollflussabstraktion bezüglich der Architektur gewinnen. Diese kann dann zur strukturellen Analyse der implementierten Fehlertoleranzarchitektur oder zur Bestimmung numerischer Zuverlässigkeitsgrössen genutzt werden. Dieser Artikel skizziert die zugrundeliegenden Ideen und Ansätze.

Stichworte:
Fehlertolerante Programmierung, Analyse fehlertoleranter Eigenschaften, explizit fehlertolerante Programmierung, Kontrollflussabstraktion

Eine wichtige Eigenschaft von Systemen in sicherheitskritischen Anwendungsbereichen ist die Robustheit und Fehlertoleranz gegenüber Störungen. Im wesentlichen werden diese Eigenschaften durch Replikation der berechnenden Komponenten gewährleistet (Redundanz), deren unabhängige Ergebnisse zum Vergleich und gegebenenfalls zur Korrektur genutzt werden.

Man unterscheidet *explizite Fehlertoleranz*, bei der die Software die Fehlertoleranz individuell nach den genannten Prinzipien aktiv absichert und *transparenter Fehlertoleranz*, wo die gewünschten Eigenschaften durch die Infrastruktur, etwa ein Betriebssystem, gewährleistet werden [1]. Im Gegensatz zur transparenten Fehlertoleranz sind bei der expliziten Fehlertoleranz die fehlertoleranten Eigenschaften eines Programmes in dessen Struktur festgelegt und so der Analyse zugänglich.

Fehlertolerante Eigenschaften schlagen sich in charakteristischen Mustern des Kontrollflusses nieder. So sollte beispielsweise in einem System, das einen Einzelfehler erkennen kann, stets nicht nur eine doppelte Werteberechnung stattfinden, sondern auch in jedem möglichen nachfolgenden Ablaufpfad ein Vergleich der Ergebnisse stattfinden. Diese Strukturen bieten die Basis fuer ein Reengineering vorhandener Software mit dem Ziel, Robustheits- und Fehlertoleranzeigenschaften zu bestimmen und zu klassifizieren.

Die fehlertolerante Struktur ist in Unterstrukturen des Kontrollflussgraphen definiert, die sich aus der Reduktion des Graphen auf eine spezielle Menge von Knotentypen und deren Erreichbarkeit untereinander ergeben. Diese Unterstrukturen kann man durch Kontrollflussabstraktion [2] gewinnen. Dabei wird der Kontrollflussgraph eines Systems anhand einer bestimmten Klasse von Knoten abstrahiert. Das Ergebnis dieses Arbeitsschritts ist ein Graph, der die transitive Erreichbarkeit der gewählten Knoten repräsentiert. Für die Architekturanalyse sind Knotentypen wie Bedingungsberechnungen, Bedingungsprüfungen, Abstimmungen (Votingfunktionen) etc. interessant.

Anhand dieser Unterstrukturen kann man die untersuchte Software bezüglich bekannter Muster [1] des fehlertoleranten Entwurfs (Multi Version Programming, Recovery Block Scheme, Conversations, ...) kategorisieren, die üblicherweise ein Redundanzkonzept zur Sicherstellung der fehlertolerierenden Eigenschaften einsetzen. Diese Kategoriesierung führt auf Grapherkennungsprobleme, wenn sie direkt durchgeführt werden soll. Ist es allerdings möglich, eine Architektur durch Pfadbedingungen zu beschreiben, ist eine effiziente Analyse möglich. In diesem Fall kann die Erkennung durch Pfadverfolgung oder Model Checking realisiert werden.

Weiterhin lassen sich numerische Kenngrössen der untersuchten Programme bestimmen. Sind etwa für die einzelnen Elemente des Kontrollflusses Ausfallwahrscheinlichkeiten bekannt, so lässt sich eine Gesamtwahrscheinlichkeit für das Funktionieren des Systems aus den ermittelten Unterstrukturen berechnen. Diese Analysemethode funktioniert analog zur Fehlerbaumanalyse (Überblick z.B. in [3]) und eignet sich zur Berechnung verschiedener Zuverlässigkeitsparameter. Je nach gewählter Knotenmenge, anhand derer die Abstraktion stattfindet, lassen sich differenziert Zuverlässigkeitsparameter berechnen, etwa auch für ausgewählte Berechnungspfade.

Ein dritter Analyseansatz ist der Nachweis gewünschter Sicherheitseigenschaften des Systems durch Analyse der Fehlertoleranzstruktur. Will man beispielsweise absichern, dass ein Ergebnis nie ohne eine vorherige Prüfung durch Abstimmung (Voting) freigegeben werden kann, so ist diese Eigenschaft durch das Nichtvorhandensein eines entsprechenden Kontrollflusspfades bewiesen. Hier bietet sich neben einer direkten Analyse durch Traversierung auch das aus anderen Bereichen bekannte Model Checking als Methode an, die durch keine bestimmte Traversierungsstrategie festgelegt ist. Die Methode dieses Anwendungsgebietes ähnelt der Architekturerkennung, ist aber in der Regel effizienter zu realisieren, da üblicherweise nur einzelne Eigenschaften geprüft werden müssen statt ein definierender Satz von Regeln.

Die Analyse bestehender Software nach den vorgestellten Konzepten verspricht nicht nur eine nachgeschaltete Klassifizierung und Bewertung bereits fertiggestellter

Software, sondern auch ein Mittel zur Vermeidung von Entwurfs- und Implementationsfehlern. Da die Analyse auf Sicherheitseigenschaften im wesentlichen automatisiert durchgeführt werden kann, ist der Prüfaufwand gering und kann Gewissheit bringen, dass gewisse Sicherheitsprinzipien nicht durch Fehler bei Implementation oder Entwurf unterlaufen werden.

Literaturverzeichnis

[1] M. Dal Cin: Zur explizit fehlertoleranten Programmierung von Parallelrechnern, in *Proc. PARS Workshop*, Gesellschaft für Informatik, S. 47-55, München, April 1989.

[2] S. Gossens: Enhancing Validation with Behavioural Types, in *Proc. High-Assurance System Engineering Symposium HASE 2002*. IEEE, S. 201-208, Tokio, Japan, Oktober 2002.

[3] W. G. Schneeweiss: *Die Fehlerbaum-Methode*, LiLoLe-Verlag, Hagen, 1999.

Toolintegration

15 A-posteriori-Integration verfahrenstechnischer Entwicklungswerkzeuge

Thomas Haase
RWTH Aachen, Lehrstuhl für Informatik III, Ahornstr. 55, D-52074 Aachen,
thaase@i3.informatik.rwth-aachen.de

15.1 Einführung

Der *SFB 476 IMPROVE* beschäftigt sich mit der informatischen Unterstützung verfahrenstechnischer Entwicklungsprozesse zur Konstruktion einer verfahrenstechnischen Anlage auf Ebene des Basic Engineering [NW99]. Hierbei verfolgt der SFB den Ansatz, *existierende verfahrenstechnische Entwicklungswerkzeuge* (z.B. Fließbildwerkzeuge für den Grobentwurf der zu entwickelnden Anlage, 1D- und 3D-Simulatoren zur Simulation von aus dem Fließbildwerkzeug abgeleiteten mathematischen Modellen der Anlage, Tabellenkalkulation zur Kostenschätzung der Anlage, ...) *a-posteriori* zu *integrieren* und miteinander zu *verschalten*.

Auf den technischen Werkzeugen aufbauend werden *neue informatische Unterstützungskonzepte* (feingranulare Prozessunterstützung; feingranulare Integratoren; informelle, multimediale Kommunikation; mittelgranulares Prozess-, Produkt- und Ressourcen-Management) entwickelt, die die Funktionalität der gegebenen technischen Werkzeuge erweitern [NSW03]. Die Werkzeuge zur Realisierung der neuen Unterstützungsfunktionalitäten werden als *erweiterte Werkzeuge* bezeichnet und werden ebenfalls miteinander verschaltet. Diese Verschaltung wird mit dem Terminus der *synergetischen Verschränkung* beschrieben.

Bei der synergetischen Verschränkung der erweiterten Werkzeuge liegt zum Teil wiederum eine a-posteriori-Integrationsthematik vor, da die Werkzeuge an verschiedenen Lehrstühlen mit unterschiedlichen Implementierungs- resp. Generierungsmechanismen entstanden. Eine Angleichung dieser Mechanismen wäre aus Aufwandsgründen unrealistisch.

Die Gesamtheit der technischen und erweiterten Werkzeuge und deren synergetischer Verschränkung ergibt dann die *erweiterte Arbeitsumgebung* für einen verfahrenstechnischen Entwickler.

15.2 Wrapper-Klassifikation

Zur Einbindung der angesprochen technischen und erweiterten Werkzeuge in die Arbeitsumgebung werden *Wrapper* eingesetzt. Langfristiges Ziel des SFB bzgl. solcher Wrapper ist es, die Konstruktion der Wrapper durch geeignete Werkzeuge zu unterstützen und sie letztendlich (semi-)automatisch zu generieren.

Dazu wurden in einem ersten Schritt die im SFB-Szenario betrachteten Werkzeuge hinsichtlich ihrer Integrationsfähigkeit klassifiziert. Beispiele für *Klassifikationsmerkmale* sind: (a) die Systemstruktur (interaktives Werkzeug oder Batch-Werkzeug), (b) die Verfügbarkeit (Quelltext vorhanden oder nicht) oder (c) die statischen und/oder dynamischen Zugriffsmöglichkeiten auf das Werkzeug (Kommandozeilenparameter vs. Programmierschnittstelle).

Anschließend wurde ein Wrapper in verschiedene *Komponenten* unterteilt (siehe Abbildung 15.2), d.h. bei einem Wrapper handelt es sich nicht um eine monolithische Komponente, sondern vielmehr um ein softwaretech-

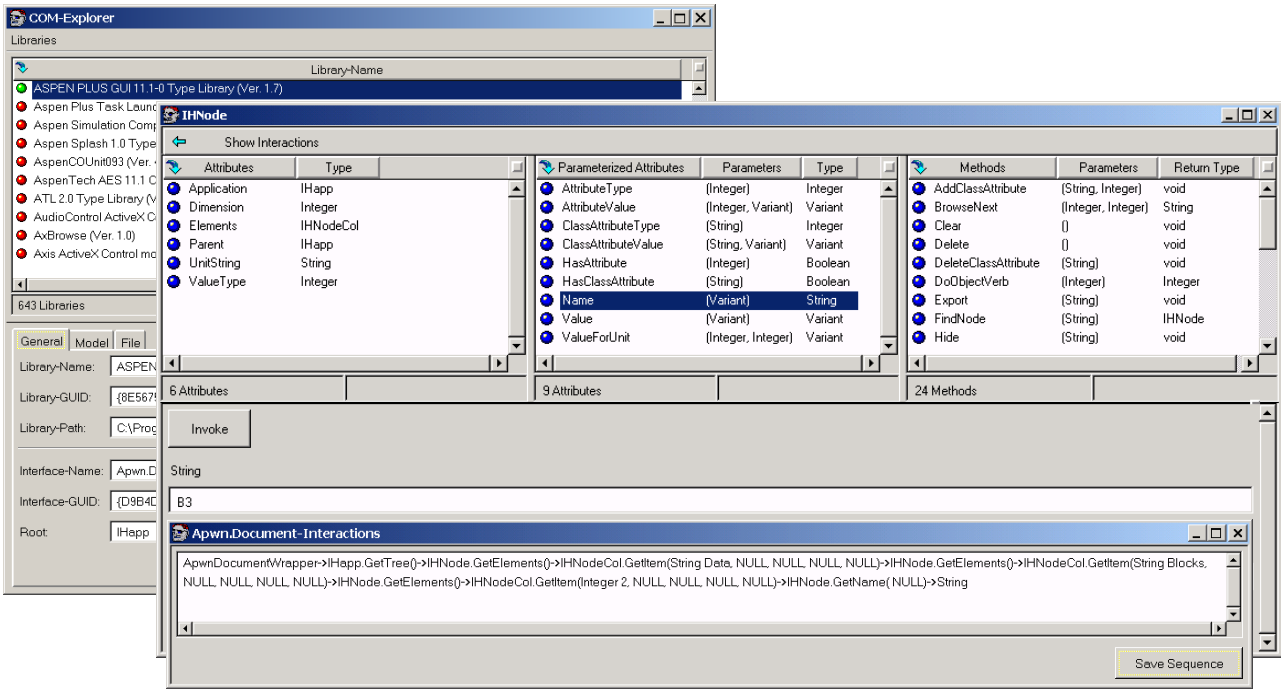


Abbildung 15.1. Werkzeugunterstützung zur interaktiven Exploration von COM-Schnittstellen

nisches Teilsystem, dessen Bestandteilen verschiedenartige Aufgaben zugeordnet sind. Die Gesamtfunktionalität des Wrappers ergibt sich aus dem Zusammenspiel dieser Bestandteile.

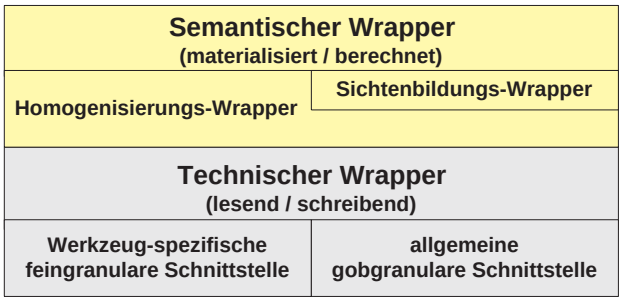


Abbildung 15.2. Wrapper-Teilsystem

Auf einer untersten Schicht lassen sich die beiden folgenden *Wrapper-Typen* unterscheiden: (1) *grobgranulare technische Wrapper* (Black-Box-Sicht auf das Werkzeug) und (2) *feingranulare technische Wrapper* (White-Box-Sicht auf das Werkzeug). Ein weiteres, orthogonales Differenzierungsmerkmal ist, ob ein lesender und/oder schreibender Zugriff auf das Werkzeug gegeben ist. Die Schnittstellen der technischen Wrapper ermöglichen einen orts- und implementierungstransparenten Zugriff auf das Werkzeug.

Die Schnittstellen der technischen Wrappers werden von den *semantischen Wrappern* genutzt. Hier gibt es die folgende Differenzierung: (1) (*Funktions-* und/oder *Daten-*)*Homogenisierungs-Wrapper* und (2) *Sichtenbildungs-Wrapper*. Man kann weiterhin unterscheiden, ob die homogene Sicht auf das Werkzeug (i) nur berechnet, (ii) berechnet und anschließend materialisiert oder (iii) eine Mischform aus (i) und (ii) realisiert wird. Semantische Wrapper offerieren einen Zugriff auf die Funktionen und Daten eines Werkzeuges auf einem einheitlichen semantischen Niveau.

15.3 Werkzeugunterstützung

Für Anwendungen, die über eine COM-Schnittstelle verfügen, wurde als eine erste Werkzeugunterstützung zur Konstruktion der Wrapper ein Werkzeug zur *interaktiven Exploration von COM-Schnittstellen* implementiert (siehe Abbildung 15.1).

Die Komponenten einer COM-Schnittstelle werden durch eine sogenannte *Type-Library* textuell beschrieben. Auf Basis einer solchen Type-Library erfasst das Werkzeug zunächst die *statische Syntax und Semantik* der betrachteten Programmierschnittstelle durch das Parsen der entsprechenden Type-Library. Anschließend können diese Informationen u.a. graphisch, in Form von *UML-Klassendiagrammen*, visualisiert werden.

Des weiteren bietet das Werkzeug die Möglichkeit, die Programmierschnittstelle interaktiv zur Laufzeit der Anwendung zu explorieren. Dazu wird die Anwendung über eine generische Operation der Schnittstelle gestartet, d.h. die initiale Komponente der Schnittstelle wird instantiiert und eine Referenz auf das erzeugte COM-Objekt wird zurückgegeben. Auf Grundlage der vorhandenen statischen Informationen generiert das Werkzeug eine graphische Benutzungsoberfläche zum Abfragen und Ändern

der Attributwerte und zum Aufrufen der Methoden des Objektes. Attributwerte oder die Rückgabewerte von Methodenaufrufen können wiederum eine Referenz auf eine Instanz einer Komponente sein, welche wiederum über eine entsprechend generierte Oberfläche inspiziert werden kann. Die sich ergebenen Sequenzen von Benutzerinteraktionen werden durch das Werkzeug aufgezeichnet und können anschließend ebenfalls, in Form von *UML-Sequenzdiagrammen*, graphisch visualisiert werden.

15.4 Ausblick

Die auf die oben beschriebene Art gewonnenen statischen und dynamischen Informationen bilden die Basis für die Schnittstellen der technischen Wrapper. Zukünftige Arbeiten sollen klären, wie diese Informationen genutzt werden können, die Schnittstellen der semantischen Wrapper zu spezifizieren, z.B. die Komposition der originären Operationen der Schnittstelle zu semantisch höherwertigen Operationen.

Literaturverzeichnis

[NSW03] M. Nagl, R. Schneider, and B. Westfechtel. Synergetische Verschränkung bei der A-posteriori-Integration von Werkzeugen. In M. Nagl and B. Westfechtel, editors, *Modelle, Werkzeuge und Infrastrukturen zur Unterstützung von Entwicklungsprozessen*, pages 137–154. Wiley-VCH, 2003.

[NW99] M. Nagl and B. Westfechtel, editors. *Integration von Entwicklungssystemen in Ingenieuranwendungen: substantielle Verbesserungen der Entwicklungsprozesse*. Springer, 1999.

gemeinsame Sitzung mit dem Workshop der GI-Fachgruppe 2.1.4 Programmiersprachen und Rechenkonzepte

16 Werkzeuggestützte Qualitätsuntersuchungen: Ein Erfahrungsbericht

Markus Bauer, Olaf Seng
Forschungszentrum Informatik Karlsruhe (FZI), Haid-und-Neu-Straße 10-14, 76131 Karlsruhe,
{bauer|seng}@fzi.de

16.1 Einführung

Qualitätsuntersuchungen (Softwareassessments) werden durchgeführt, um unter Einsatz von ReverseEngineering Techniken die innere Qualität eines Systems zu untersuchen. Die innere Qualität eines Systems wird durch diejenigen Eigenschaften bestimmt, die der Entwickler wahrnimmt, wie z.B. Erweiterbarkeit, Wiederverwendbarkeit, etc. Für Softwareunternehmen ist es von großem Interesse, eine quantitative Aussage über die innere Qualität ihrer Systeme machen zu können, da diese Entwicklungsaufwand und Wartbarkeit der Systeme beeinflusst und somit die anfallenden Kosten bestimmt. Zudem können kritische Stellen identifiziert werden. Solche Stellen können für Restrukturierungsmaßnahmen, Testfälle und Dokumentation bilden.

16.2 Softwareuntersuchungen – Techniken und Vorgehensweise

Zur Durchführung der Qualitätsuntersuchung werden aus dem Bereich Software-Reengineering bekannte Techniken eingesetzt [FAM99] [BC01]. Dazu gehört an erster Stelle die sogenannte *Faktenextraktion*. Hierbei wir mit Hilfe von Compilerbautechnolgien und graphtheoretischen Konzepten ein Designdatenbank des Systems erstellt, auf der weitere Analysetechniken aufsetzen. *Visualisierungstechniken* können das Verständnis von Architekturen und Strukturen erleichtern und erlauben z.B. eine einfache Überprüfung geschichteter Architekturen. Mit Hilfe von *Abstraktionstechniken* kann die Information in der Designdatenbank verfeinert werden, in dem z.B. bei prozeduralen System Methoden mit Datentypen zu einer Einheit gruppiert werden. Mit Hilfe von *Software-Heuristiken und -Metriken* können z.B. Schwachstellen gesucht und Entwurfsrichtlinien überprüft werden [Ciu99].

Der Ablauf werkzeuggestützter Qualitätsuntersuchungen gliedert sich in die folgenden Schritte, für die insgesamt mindestens fünf Arbeitstage veranschlagt werden müssen:

1. *Zieldefinition:* Zuerst muss mit den Entwicklern zusammen geklärt werden, welche Ziele das Assessment verfolgen soll und welche Analysen überhaupt durchgeführt werden sollen.
2. *Faktenextraktion und Analyse:* Zu Beginn der Analyse steht die (oft von Problemen begleitete) Faktenextraktion, zudem können einfach und schnell durchzuführende Analysen erste Einschätzungen

über das System vermitteln.

3. *Zwischenergebnisse:* Die ersten Analyseergebnisse sollten möglichst bald, idealerweise am zweiten Tag des Assessments gemeinsam mit einem oder mehreren Entwicklern diskutiert werden.
4. *Verfeinerte Faktenextraktion und Analyse:* Die Faktenextraktion kann verfeinert und auf weitere Systemteile ausgeweitet werden.
5. *Abschlussworkshop:* Die Ergebnisse sollten möglichst vielen Entwicklern z.B. in Form einer Präsentation gezeigt werden, um sie zu diskutieren und zu bewerten.

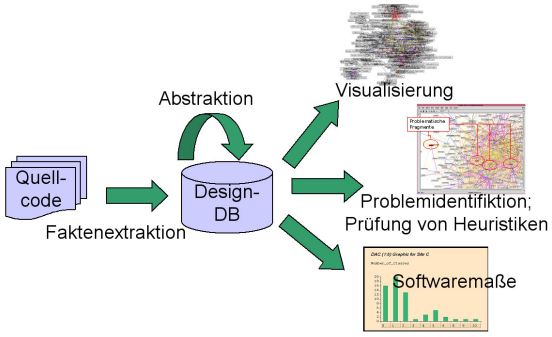


Abbildung 16.1. Techniken und Werkzeuge

16.3 Fallstudie und Ergebnisse

Bei der von uns untersuchten Fallstudie handelt es sich um ein komplett in C geschriebenes Datenbanksystem. Wir konzentrierten unsere Analysen auf den Server-Teil des Systems (Umfang: 1 Million LOC, 28 Subsystemen, 1347 Dateien).

Zuerst wurden die grobgranularen Einheiten des Systems – in diesem Fall durch Verzeichnisse gegebene Subsysteme – analysiert. Auf dieser Ebene wurden Architekturrichtlinien überprüft. Weiterhin konnten durch Kombination von Metriken wie LOC und McCabe die komplexesten Subsysteme identifiziert werden, die im weiteren Verlauf der Analysen besonders intensiv untersucht werden sollten. Einige der Subsysteme zeichnen sich durch eine besonders hohe Komplexität aus. Diese Subsysteme waren jedoch gut gekapselt (Analyse der Kopplung), so dass die hohe Komlexität kein Problem darstellt.

Auf der darunterliegenden Abstraktionsebene wurden in den besonders komplexen Subsystemen die vorhandenen Datentypen untersucht, indem Daten und zugehörige Funktionen zu abstrakten Datentypen gruppiert wurden. Dies war in diesem System über Heuristiken (Namenskonventionen, Parametertypen der Funktionen) möglich, da die Programmierer einen objektorientierten Programmierstil verwendet hatten. Auf diesen abstrakten Datentypen konnten dann ebenfalls Komplexitäts- und Kopplungsanalysen angewandt werden. In Abbildung 16.2 sind die Kopplungswerte für Paare von abstrakten Datentypen zu sehen.

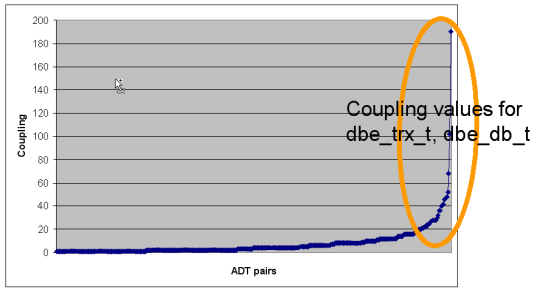


Abbildung 16.2. Kopplung abstrakter Datentypen

Manche der Datentypen sind äußerst komplex und zeichnen sich durch eine hohe Kopplung zu anderen Datentypen aus. Da sie aber zentrale und komplexe Konzepte wie Transaktionen repräsentieren, kann dies wahrscheinlich nicht vermieden werden.

Auf der untersten Abstraktionsebene können Analysen auf Methodenebene ausgeführt werden. Hierzu gehören z.B. die Suche nach unerwünschten Rekursionen oder Komplexitätsberechnungen. In dieser Fallstudie konnten – im Gegensatz zu Mozilla – keine langen Rekursionsketten gefunden werden.

16.4 Praxiserfahrungen

Im Verlaufe zahlreicher durchgeführter Softwareassessments konnten wir einige Erfahrungen sammeln, über die wir im Folgenden berichten wollen.

Hindernisse

Die Durchführung von Softwareassessments kann durch eine Reihe von Hindernissen erschwert werden.

- Zunächst behindert die Skepsis (“Hilfe, meine Leistung wird begutachtet!”) der Entwickler die Durchführung der Softwareassessments. Deren Know-How ist für erfolgreiche Assessments jedoch von großer Bedeutung (Zieldefinition, Interpretation der Analyseergebnisse). Abhilfe schaffen hier Kurzpräsentationen von Zwischenergebnissen, in denen den Entwicklern Potenzial und Nutzen der Untersuchungen demonstriert werden.
- Defizite auf Werkzeugebene: Werkzeuge zur Faktenextraktion kommen oft mit industriellem Code nicht zurecht. Sie scheitern häufig an unvollständigem Code, Makros und speziellen Programmiersprachendialekten. Zudem bereitet die große Menge an Rohdaten (Fakten) Probleme. Die meisten Analysewerkezeuge verarbeiten große Datenmengen nur unzuverlässig, zudem bieten nur wenige Werkzeuge Abstraktionsmechanismen zur Reduktion der Datenflut, die die Analysten auswerten müssen.

- Verfahren zur automatisierten Identifikation von Qualitätsmängeln können nur Hinweise auf potenzielle Schwachstellen liefern. Diese müssen dann von Analysten anhand des Quelltextes des Systems untersucht und bewertet werden. Schwierigkeiten bereiten in diesem Zusammenhang der Zeitmangel während der Assessments und ggf. die mangelnde Erfahrung der Analysten in der Anwendungsdomäne des Systems.

Nutzen

Insbesondere die Workshops zur Präsentation und Diskussion der Analyseergebnisse mit den Entwicklern zeigen jedoch, dass Softwareassessments von Auftraggebern und Entwicklern überwiegend positiv bewertet werden. Neben dem eigentlichen Ziel, objektive Aussagen über die innere Qualität der Software zu bekommen, lassen sich einige weitere positive Effekte identifizieren:

- Schon im Verlaufe der Abschlussworkshops beginnt in der Regel eine ausführliche Diskussion der Entwickler über Softwarequalität und über möglicherweise verbesserungsbedürftige Teile des Systems. Häufig entsteht so, motiviert durch die Analyseergebnisse, ein verbessertes Qualitätsbewusstsein.
- Entwickler interessiert oft der Vergleich mit anderen Softwaresystemen. Externe Analysten können hierzu leichter Aussagen machen, obgleich die Vergleichbarkeit von Softwaremaßen nur schwer zu gewährleisten ist: Anwendungsdomäne, Programmiersprache und -stil beeinflussen die Messwerte maßgeblich.

16.5 Schlussbemerkung

In diesem Papier haben wir gezeigt, dass mit werkzeuggestützten Qualitätsuntersuchungen quantitive Aussagen über die innere Qualität eines Softwaresystems gemacht werden können. Wir haben in diesem Papier Nutzen und Hindernisse solcher Assessments aufgezeigt. Hinsichtlich der Hindernisse erhoffen wir uns insbesondere für den Bereich der Werkzeugunterstützung in Zukunft einige Verbesserungen. Das FZI arbeitet mit verschiedenen Partnern im Rahmen der Forschungsprojekte COMPOBENCH und QBENCH an diesen Themen.

Literaturverzeichnis

[BC01] BAUER, MARKUS und OLIVER CIUPKE: *Objektorientierte Systeme unter der Lupe*. in *Proceedings of the third German Workshop on Software-Reengineering*. Universität Koblenz-Landau, Institut für Informatik, 2001.

[Ciu99] CIUPKE, OLIVER: *Automatic Detection of Design Problems in Object-Oriented Reengineering*. in *Technology of Object-Oriented Languages and Systems - TOOLS 30*. IEEE Computer Society, 1999.

[FAM99] FAMOOS PROJEKTTEAM: *The FAMOOS Object-Oriented Reengineering Handbook*. Technischer Bericht, Forschungszentrum Informatik, Karlsruhe, Software Composition Group, University of Berne, 1999.

---

17 (Linear) Algebra for Program Analysis

Markus Müller-Olm
FernUniversität Hagen, LG Praktische Informatik V, 58084 Hagen, Germany
On leave from Universität Dortmund, mmo@ls5.cs.uni-dortmund.de

Helmut Seidl
Universität Trier, FB 4-Informatik, 54286 Trier, Germany, seidl@uni-trier.de

This talk reported on onging research in which we use techniques from algebra and linear algebra to construct highly precise analysis routines for imperative programs or program fragments. We are interested in programs that work on variables taking values in some fixed ring or field $\mathbb{F}$, e.g., the integers or the rationals. Our analyses precisely interpret assignment statements with affine or polynomial right hand side and treat other assignment statements as well as guarded branching statements conservatively as non-deterministic statements. More specifically, we have constructed

1. an interprocedural analysis that determines for every program point of an affine program all valid affine relations[1];

[1] An *affine relation* is a condition of the form $a_0 + \sum_{i=1}^{n} a_i x_i = 0$, where $a_0, \ldots, a_n \in \mathbb{F}$ are constants from the underlying field and $x_1, \ldots, x_n$ are the program variables. A relation is *valid* at a program point, if it holds whenever control reaches that program point.

2. a generalization of this analysis to polynomial relations of bounded degree[2] and to affine programs with local variables; and
3. an intraprocedural analysis that decides validity of polynomial relations in polynomial programs.

The running time of analysis 1 and 2 is linear in the size of the program and polynomial in the number of program variables. For analysis 3 we have a termination guarantee but do not know an upper complexity bound. These analyses have many potential applications, because analysis questions can often be coded as affine or polynomial relations easily. Some obvious examples are:

- x is a constant of value $a \in \mathbb{F}$ at a program point p iff the affine relation $x - a = 0$ is valid at p;
- x and y have always the same value at p iff the affine relation $x - y = 0$ is valid at p; and
- x only takes values in the set $\{a_1, \ldots, a_k\} \subseteq \mathbb{F}$ iff the polynomial relation $(x - a_1) \cdot \ldots \cdot (x - a_k)$ is valid.

Preliminary results of this line of research can be found in the references.

Bibliography

[1] M. Müller-Olm. Variations on Constants. Habilitationsschrift, Fachbereich Informatik, Universität Dortmund, 2002.

[2] M. Müller-Olm and H. Seidl. Polynomial constants are decidable. In M. Hermenegildo and G. Puebla, editors, *SAS 2002 (Static Analysis of Systems)*, volume 2477 of *Lecture Notes in Computer Science*, pages 4–19. Springer, 2002.

[3] M. Müller-Olm and H. Seidl. Computing interprocedurally valid relations in affine programs. Technical report, Universität Trier, Fachbereich 4-Informatik, 2003.

---

18 Verstehen dynamischer Programmaspekte mittels Software-Instrumentierung[1]

Christoph Steigner, Jürgen Wilke
Universität Koblenz-Landau, Postfach 201 602, D-56016 Koblenz
{steigner,wilke}@uni-koblenz.de

Abstract

Die Performance-Analyse und die dynamische Analyse zum Zwecke des Programmverstehens verfolgen im Kern ein gemeinsames Ziel: Beide Disziplinen möchten Einsicht in das dynamische Verhalten von Programmen gewinnen. Laufzeitdaten, die im Rahmen der Performance-Analyse erhoben werden, können daher auch für das Verstehen dynamischer Programmaspekte von Interesse sein. Da die Performance-Analyse gegenüber der dynamischen Programmanalyse auf eine wesentlich längere Tradition zurückblickt, bietet es sich an, etablierte Techniken und Werkzeuge der Performance-Analyse auf ihre Verwendbarkeit hinsichtlich des Verstehens dynamischer Programmaspekte zu untersuchen. Dieser Beitrag stellt das Performance-Analyse-Tool CoSMoS *vor und diskutiert, inwieweit die von* CoSMoS *aufgezeichneten Daten für das Verstehen dynamischer Programmaspekte nutzbar sind.*

18.1 Einführung

Um ein Programm zu verstehen, kommen – abgesehen vom Studium der Programmdokumentation – hauptsächlich zwei Vorgehensweisen in Betracht: die Analyse des Quelltexts (statische Analyse) und die Analyse von Laufzeitdaten, die bei Programmausführungen gewonnen wurden (dynamische Analyse). Die Forschung im Bereich des Software Reengineering hat sich lange auf die statische Analyse konzentriert. Erst in jüngerer Vergangenheit wurde erkannt, dass das Studium des dynamischen Verhaltens von Programmen wertvolle Erkenntnisse liefern kann, die aus der Quelltextanalyse nicht zu erschließen sind [1].

Eine systematische Gewinnung von Laufzeitdaten ist ohne geeignete Werkzeugunterstützung nicht möglich. Da solche Werkzeuge auch für die Performance-Analyse von Programmen essentiell sind und dort schon seit langer Zeit eingesetzt werden, liegt es nahe, deren Verwendbarkeit für die Zwecke der dynamischen Programmanalyse zu untersuchen. Im Folgenden stellen wir den Applikationsmonitor

[2] A *polynomial relation* is a condition of the form $p(x_1, \ldots, x_n) = 0$ where $p(x_1, \ldots, x_n)$ is a multi-variate polynomial in $x_1, \ldots, x_n$ over $\mathbb{F}$. A polynomial relations is bounded by $d \in \mathbb{N}$ if the sum of exponents of every monomial in p is less than or equal d.

[1] Die in diesem Papier dargestellten Forschungsarbeiten wurden teilweise durch die *Stiftung Rheinland-Pfalz für Innovation* gefördert

des Performance-Analyse-Werkzeugs *CoSMoS* vor. Wir werden zeigen, dass die Laufzeitdaten, die von dem Monitor aufgezeichnet werden, direkt für die Rekonstruktion des Aufrufgraphen der vermessenen Programmausführung verwendet werden können.

18.2 Der CoSMoS-Applikationsmonitor

Das Performance-Analyse-System *CoSMoS*[2] besteht im Kern aus drei eigenständigen Monitoring-Komponenten, die Daten auf unterschiedlichen Systemebenen aufzeichnen [2]. Die für die Datenerfassung auf Applikationsebene zuständige Komponente ist der *Applikationsmonitor*. Um Daten mit diesem Werkzeug gewinnen zu können, müssen in einem ersten Schritt zusätzliche Anweisungen (*Messsensoren*) in den Quelltext der Applikation eingebracht werden, die der Erhebung und Aufzeichnung von Messdaten dienen. Dieser Schritt wird *Quelltext-Instrumentierung* genannt. Er kann wahlweise automatisch (Vollinstrumentierung) oder interaktiv (selektive Instrumentierung) erfolgen.

Die eingefügten Messsensoren bestehen im Falle von *CoSMoS* aus Aufrufen einer Messmethode. Jeder dieser Aufrufe wird mit einer Nummer versehen, die die Quelltextstelle, an der der Sensor eingefügt wurde, eindeutig identifiziert. Bei jedem Aufruf zur Laufzeit erhebt der Sensor einen Zeitstempel und speichert diesen zusammen mit seiner Sensornummer in einem Ereignisdatensatz ab. Die Folge der Ereignisdatensätze (*Ereignisspur*) wird für jede sequentielle Ausführungseinheit (Prozess bzw. Thread) getrennt gespeichert, so dass die Ereignisfolge jeder Ereignisspur streng sequentiell ist.

Meist werden Sensoren paarweise eingesetzt, wobei ein Sensor den Beginn und ein Sensor das Ende einer bestimmten Programmaktivität markiert. Um etwa einen Methodenaufruf [3] zu instrumentieren, wird je ein Sensor vor und hinter den Aufruf gesetzt, so dass der Eintritt und die Rückkehr aus der Methode registriert werden.

Eine Alternative zur Instrumentierung des Aufrufs besteht darin, die Methode selbst zu instrumentieren. In diesem Fall wird im Rumpf der Methode ein Sensor vor der ersten Anweisung eingefügt und je ein weiterer Sensor vor jedem Ausgang der Methode. Die Instrumentierung des Methodenrumpfes ist insbesondere dann sinnvoll, wenn *alle* Aufrufe der Methode erfasst werden sollen.

Nach der Einbringung der Sensoren in den Quelltext muss die Applikation neu kompiliert und mit einer Bibliothek (*Messbibliohek*) gebunden werden, die den Objektcode der Messmethode bereitstellt. Die so präparierte Applikation erzeugt beim nächsten Programmlauf die gewünschten Messdaten.

18.3 Erstellung des Aufrufgraphen

Ein wesentlicher Mehrwert, der von der dynamischen Programmanalyse erbracht werden kann, besteht darin, Aufrufbeziehungen zwischen den Methoden einer Applikation nachzuweisen, die durch statische Analyse nicht ermittelbar sind. Die Verwendung von Polymorphismus in objektorientierten Sprachen, aber auch der Einsatz von Funktionszeigern in C-Programmen führen dazu, dass sich unter Umständen erst zur Laufzeit entscheidet, welche Methode an einer bestimmten Programmstelle tatsächlich aufgerufen wird.

Die von den *CoSMoS*-Sensoren aufgezeichneten Ereignisspuren erlauben eine Rekonstruktion des Aufrufgraphen für den vermessenen Programmlauf. Voraussetzung ist eine Instrumentierung aller Methodenrümpfe. Die Datenaufzeichnung erfolgt dadurch nicht beim Aufrufer, sondern beim Aufgerufenen, so dass die Identifikation des Aufgerufenen grundsätzlich gewährleistet ist.

Wurden sowohl der Aufrufer als auch der Aufgerufene instrumentiert, sind beide aus der Ereignisspur an Hand der Zeitstempel eindeutig identifizierbar: seien t_{A_1} und t_{A_2} der Eintritts- bzw. der Austrittszeitstempel eines Aufrufs der Methode A und seien t_{B_1} und t_{B_2} die entsprechenden Zeitstempel eines Aufrufs der Methode B und sei weiterhin B von A aufgerufen worden, so muss gelten: $t_{A_1} < t_{B_1} < t_{B_2} < t_{A_2}$. Wegen der Sequentialität der Ereignisse in einer Ereignisspur gilt aber auch die Umkehrung, d.h. aus der zeitlichen Ordnung der Methodenein- und -austrittsereignisse lässt sich die Aufrufbeziehung herleiten. Allerdings ist zu beachten, dass die Zeitinformation keinen Rückschluss erlaubt, ob B von A direkt aufgerufen wurde. Sei C eine Methode einer (nicht instrumentierten) Fremdklasse bzw. -bibliothek, so ist der indirekte Aufruf $A \rightarrow C \rightarrow B$ vom direkten Aufruf $A \rightarrow B$ an Hand der Ereignisspur nicht zu unterscheiden. Die Erfassung von Aufrufen solcher Fremdmethoden ist allerdings prinzipiell möglich, indem neben den Methodenrümpfen zusätzlich alle Methodenaufrufe instrumentiert werden. [4]

Zu beachten ist ferner, dass es bei Programmiersprachen und -umgebungen, die eine asynchrone Unterbrechung der Programmausführung erlauben, zur Herleitung falscher Aufrufbeziehungen kommen kann. Trifft beispielsweise in einem C-Programm während der Bearbeitung einer Methode A ein Signal ein, welches zur Ausführung eines (instrumentierten) Signal-Handlers B führt, legen die Zeitinformationen der Ereignisspur fälschlicherweise nahe, dass B von A aufgerufen wurde.

Für die Berechnung des Aufrufgraphen aus den Zeitdaten einer Ereignisspur implementierten wir das C-Programm `mkcallgraph`. Abbildung 18.1 zeigt einen Ausschnitt des Aufrufgraphen, der aus der Ereignisspur einer Ausführung des Programms `mkcallgraph` selbst gewonnen wurde. Der Visualisierung des Graphen wurde mit dem

[2] Coblenz Software Monitoring System

[3] Mit dem Begriff *Methode* bezeichnen wir im Folgenden sowohl Methoden objektorientierter Sprachen als auch Prozeduren und Funktionen prozeduraler Sprachen.

[4] Die automatische Instrumentierung aller Methodenaufrufe einer Applikation wird derzeit von *CoSMoS* allerdings noch nicht unterstützt.

Programm `dot` aus dem `graphviz`-Paket [3] erzeugt. Neben der reinen Aufrufinformation sind durch farbliche Unterlegungen diejenigen Knoten markiert, die den am häufigsten aufgerufenen Methoden entsprechen (gelb bzw. hellgrau), bzw. denjenigen Methoden, die die meiste Berechnungszeit verbraucht haben (rot bzw. dunkelgrau).

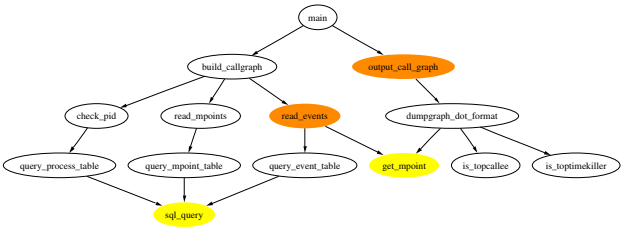


Abbildung 18.1. Aufrufgraph von `mkcallgraph`

18.4 Fazit

Die Technik der Software-Instrumentierung zur Gewinnung von Laufzeitdaten auf Applikationsebene ist für die dynamische Programmanalyse gewinnbringend nutzbar und wurde vereinzelt auch schon eingesetzt [4]. Wie am Beispiel des Applikationsmonitors des *CoSMoS*-Systems gezeigt wurde, sind aus dem Bereich der Performance-Analyse bereits fertige Werkzeuge verfügbar, die diese Technik implementieren und Laufzeitdaten liefern, die für die Erzeugung von Aufrufgraphen direkt verwendbar sind. Aufgrund der hochgradig laufzeitoptimierten Sensoren, die im Bereich des Performance-Monitoring unabdingbar sind, gewährleisten solche Werkzeuge zudem, dass das Ablaufverhalten der instrumentierten Applikation geringstmöglich verändert wird. Desweiteren werden ohne zusätzlichen Aufwand Zeitinformationen verfügbar, die die Lokalisierung wichtiger (weil häufig genutzter) Methoden unterstützen.

Literaturverzeichnis

[1] T.A. Corbi, “Program Understanding: Challenge for the 1990’s,” IBM Research Report, 1989

[2] Ch. Steigner und J. Wilke, “Performance Tuning of Distributed Applications with CoSMoS,” in *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-21)*, Phoenix, Arizona, USA, April 2001.

[3] Stephen North et al., *GraphViz* – Graph Visualization Project, `http://www.graphviz.org`.

[4] T. Richner, S. Ducasse, “Recovering High-Level Views of Object-Oriented Applications from Static and Dynamic Information,” in *Proceedings of the International Conference of Software Maintenance (ICSM)*, Oxford, England, 1999.

Werkzeugbau

19 Erfahrungen bei der Entwicklung von Werkzeugen zum Reverse Engineering

Uwe Kaiser
pro et con, Innovative Informatikanwendungen GmbH, Annaberger Straße 240, 09125 Chemnitz
`Uwe.Kaiser@proetcon.de`

Seit 1994 entwickelt ”pro et con” Software-Werkzeuge für das Reverse Engineering und die Migration von kommerziellen Programmsourcen. Dieser Beitrag vermittelt einen Überblick über die Werkzeuge und stellt resultierende Erfahrungen ihrer Entwicklung zur Diskussion.

19.1 Compiler front-end’s

Für eine Reihe von Programmiersprachen wurden front-end’s entwickelt. Der Begriff ”front-end” besagt, daß der jeweilige Preprozessor der Programmiersprache sowie die syntaktische und die statisch semantische Analyse realisiert wurden, und daß die Abbildung in einen internen, abstrakten Syntaxbaum erfolgt. Die Liste existierender front-end’s besteht aus:

- PL/I,
- COBOL,
- NATURAL,
- TAL (Tandem Application Language),
- SQL,
- C,
- Java.

Im Beitrag kann nur ein Überblick vermittelt werden, Diskussionen von Details sprengen seinen Rahmen. Aus diesem Grund folgen zu den einzelnen front-end’s nur kurze

Anmerkungen:

- Sie laufen auf den Betriebssystemen Windows (NT, 2000, XP) und UNIX (Linux, Solaris).
- Verschiedene front-end's laufen ebenso auf den Host-Rechnern, auf denen die jeweilige Sprache implementiert ist, z.B. PL/I und COBOL auf MVS, COBOL und TAL auf NonStop Servern von HP.
- Für COBOL und PL/I auf MVS werden Panvalet und Librarian unterstützt, embedded SQL, CICS und IMS werden analysiert.
- Die COBOL-Analyse läßt sich für verschiedene Dialekte skalieren (COBOL II, OS/VS COBOL, NonStopCOBOL 85, ScreenCOBOL).
- Die Entwicklung eines front-end's für SQL war notwendig, weil verschiedene Sprachen (PL/I, COBOL, NATURAL,..) eine Schnittstelle zu SQL besitzen (embedded SQL) und die SQL-Abschnitte in den Sourcen ebenfalls analysiert werden.
- Wenn in den Aufzählungen realisierter front-end's C aufgeführt ist, C++ aber fehlt, dann liegt die Ursache darin, daß bisher der hohe Entwicklungsaufwand für C++ gescheut wurde (z.B. bei der Realisierung von Templates).
- Das TAL front-end entstand im Zuge eines Projektes, bei welchem ein kommerzielles Softwarepaket von TAL nach C migriert wurde.
- Das Java front-end unterstützt Java, Version 1.4. Das Werkzeug ist UNICODE-fähig.

19.2 Entwicklungserfahrungen

Aus der Entwicklung der front-end's resultieren eine Reihe von Erfahrungen, welche nachfolgend zusammengefaßt werden. Insbesondere interessieren Unterschiede zwischen der Analyse bei Compilern und bei Reengineering-Werkzeugen.

1. Für Reengineering und Migration ist es wesentlich, daß keine Informationen der Source während des Analyseprozesses verlorengehen. Dazu gehören unter anderem Informationen zu Preprozessoranweisungen und Kommentaren. Die Realisierung erfolgt durch die Aufnahme dieser Informationen in die Tokenliste, d.h., Preprozessoranweisungen und Kommentare bilden eigene Token im Tokenstrom.
2. Für Reverse Engineering und Migration ist das genaue Führen der Sourcecodeposition von Objekten wichtig. Die Sourcecodeposition wird zur Komponente des Wertestroms des Scanners. Eine konkrete Anwendung besteht z.B. darin, beim COBOL-Reengineering die Deklarationsstelle eines Datenobjektes zu dokumentieren (In welcher COPY-Strecke und dort in welcher Zeile und Spalte ist das Datenobjekt lokalisiert ?).
3. Reengineering von komplexen Programmen geht über die Analyse von Sourcecode hinaus. Bekanntes Beispiel sind Java-Programme, für deren Analyse neben verschiedenen in der Import-Liste aufgeführten Java-Sourcen ebenso die Analyse von ".class"-Files bzw. "rt.jar"-Files notwendig ist. Ein alternatives Beispiel ist NATURAL, hier werden Datendefinitionen in spezieller Notation in sogenannten "DATA-AREA"-Files abgelegt, auf die im Sourceprogramm Bezug genommen wird. Als Konsequenz für die Programmanalyse sind spezielle Analysatoren zu entwickeln, z.B. bei Java für das Lesen von ".class"-, bzw. "rt.jar"-Files, bei NATURAL für das Aggregieren der Informationen aus den "DATA-AREA"-Files.
4. front-end's müssen eine gewisse Robustheit gegenüber fehlerhaften bzw. unvollständigen Sourcen besitzen. Sie müssen auch dann noch Informationen liefern. Praktische Anwendungen sind fehlende COPY-Books bei der Analyse oder die Kundenanforderung, separate COPY-Books von COBOL analysieren zu können.
5. Der Arbeitsaufwand für die Entwicklung o.g. front-end's wurde quantifiziert. Für den Entwicklungsprozeß ausgehend von der Spezifikation bis zur GA-Version und nach Durchlaufen des gesamten QA-Prozesses ergibt sich ein durchschnittlicher Zeitrahmen von ca. 18 Monaten und ein durchschnittlicher Aufwand von ca. 3.5 Entwicklungsjahren je front-end.

19.3 Tools und Metatools

Parallel zur Entwicklung der front-end's entstanden eine Reihe von Meta-Tools, welche der Produktion von Reengineering-Werkzeugen dienen. Alle o.g. front-end's wurden mit einem eigenentwickelten Parsergenerator realisiert. Sein Name "Backtracking Compiler Compiler" (BTRACC) verweist auf das zugrundeliegende Grammatikanalyseverfahren der generierten Parser. In der Theorie wird auf die schlechtere Performance des Backtracking-Verfahrens gegenüber anderen Analyseverfahren verwiesen. Praktische Erfahrungen zeigen jedoch, daß diese zu vernachlässigen ist. Durchgeführte Tests ergaben, daß auf einem mit 1.2 GHz getakteten Pentium z.B. eine C-Source von 13.000 LOC in 1.2 Sekunden mit diesem Verfahren analysiert wird. Das Backtracking-Verfahren besitzt den Vorteil, daß Grammatikbeschreibungen von Programmiersprachen, welche in Dokumentationen vorkommen und überwiegend pragmatisch entstanden, keiner aufwendigen Umstellung von Grammatikregeln bzw. Konfliktbereinigung bedürfen, um von BTRACC akzeptiert zu werden.

Die front-end's sind Bestandteil von Werkzeugen für das Reverse Engineering oder sie werden in Form von Service bei Migrationsprojekten eingesetzt. Ein kommerzielles Tool für das Reverse Engineering und die Redokumentation von Programmsourcen ist "Flow Graph Manipulator´´ (FGM).

19.4 Aktuelle Entwicklungen

Aktuelle Entwicklungen betreffen die Weiterentwicklung des Tools FGM:

- Es werden Steuerflußinformationen in Form von Programmablaufplänen entwickelt.
- Ein NATURAL front-end wird in das Werkzeug integriert.

Über die feingranulare Analyse auf der Ebene eines Programmes hinaus soll Applikationswissen aufgebaut werden. Ein Sourceprogramm existiert nicht unabhängig von seiner Umwelt, sondern es existieren Schnittstellen (Datenbanken, Files, Transaktionsmonitore). In FGM soll aus diesem Grund eine SQL-Datenbasis eingebunden werden, welche dieses Applikationswissen zukünftig verwaltet und dem Nutzer durch Anfragen zur Verfügung stellt.

---

20 BTRACC-Ein Parsergenerator auf der Basis eines Backtracking-Verfahrens

Uwe Erdmenger
pro et con, Innovative Informatikanwendungen GmbH, Annaberger Straße 240, 09125 Chemnitz
Uwe.Erdmenger@proetcon.de

Einige Besonderheiten der syntaktischen Analyse von Quellcode im Reengineering-Bereich sind mit den bekannten Parsergeneratoren nur schwer realisierbar. Insbesondere der Wunsch, sehr schnell brauchbare Analysewerkzeuge erstellen zu können, war der Anlaß für die Entwicklung des neuen Parsergenerators *BTRACC*.

Der folgende Beitrag beschreibt die grundlegende Arbeitsweise des Werkzeugs. Ausgehend von der bekannten Erweiterten Backus-Naur-Form (EBNF) wird die Generierung des Parsers und seine Funktionalität beschrieben, wobei auch auf die Frage der Attributierung eingegangen wird.

20.1 Einleitung und Motivation

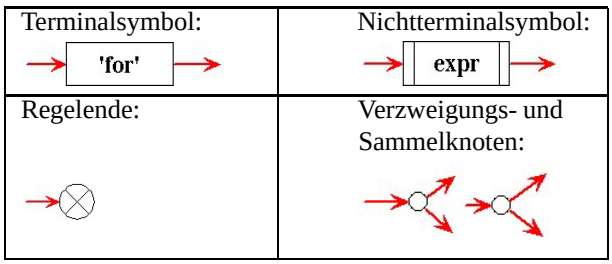
Bei der Erstellung von Analysewerkzeugen für Reengineering-Zwecke liegt die Grammatik meist in einer Form vor, welche für bekannte Parsergeneratoren (YACC, PCCTS, COCO/R; ...) nicht optimal geeignet ist. Insbesondere der große Sprachumfang und der Wunsch, mehrere Dialekte mit einem Parser bearbeiten zu können, sind die Ursache dafür. Außerdem stehen hier häufig Korrekturen und Erweiterungen an, welche bei oben genannten Werkzeugen zu größeren Umstellungen der vorhandenen Grammatik führen.

Um vorliegende Grammatiken ohne große Umstellung in Parsern verwenden und diese auch einfach pflegen zu können, wird bei der Erstellung von Parsern in der Firma *pro et con GmbH* ein eigener Parsergenerator **BTRACC** (BackTRAcking Compiler Compiler) verwendet, dessen Funktionsweise im Weiteren beschrieben werden soll. Mit diesem Werkzeug wurden bereits ein SQL-, ein Natural- ein Java- und ein C-Parser realisiert, die auch in kommerziellen Produkten Verwendung finden.

20.2 Interne Darstellung der Grammatik

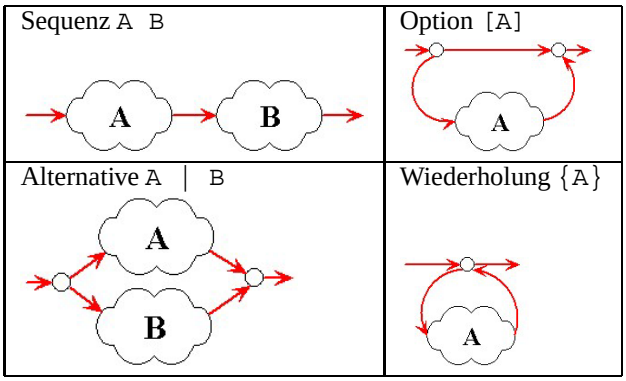
Ausgangspunkt ist die attributierte Grammatik der zu analysierenden Sprache in EBNF. Diese liest der Parsergenerator BTRACC, prüft ihre syntaktische Korrektheit und baut eine interne Darstellung auf.

Dabei handelt es sich um eine Datenstruktur, welche als Graph angesehen werden kann. Weil BTRACC in C++ geschrieben wurde, lag die Verwendung von dynamisch allokierten Strukturen als Knoten des Graphen und Zeigern als Kanten nahe. Dabei gibt es folgende Arten von Knoten:



Aus diesen Grundbausteinen kann die interne Repräsentation der Regeln zusammengesetzt werden. Dabei wird zu jeder Regel ein Graph aufgebaut, welcher die rechte Regelseite wiederspiegelt. Alle diese Regeln werden, sortiert nach ihrem Namen, in einer Hash-Liste verwaltet, welche einen schnellen Zugriff über den Regelnamen (Nichtterminalsymbol) gestattet.

Die Strukturen der Erweiterten Backus-Naur-Form werden wie folgt zusammengesetzt:



Dabei stehen `A` und `B` für Terminalsymbol-Knoten, Nichtterminalsymbol-Knoten oder wiederum komplette Strukturen (rekursive Definition).

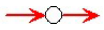
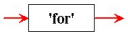
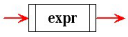
20.3 Generierung der Parser-Quellcodes

Die Graphen dürfen keine Zyklen ohne mindestens ein Terminalsymbol enthalten. Das führt zu einer Endlosschleife im Algorithmus. Dies wird vor der Generierung getestet. Praktisch bedeutet das, daß die Eingabegrammatik nicht linksrekursiv sein und keine Verschachtelung optionaler Konstrukte der Art `{ [ A ] }` enthalten darf.

Der zu generierende Parser-Quellcode besteht in der Hauptsache aus einem Feld von C-Strukturen, welche als Maschinenbefehle einer virtuellen Maschine angesehen werden können. Dabei wird zu jedem Knoten im Graphen ein solcher Befehl erzeugt. Er enthält auch die Kante zum nächsten/alternativen Knoten als Feldindex des ihm zugeordneten Befehls. Das ist möglich, da nach obiger Vorschrift ein Knoten maximal 2 Nachfolger haben kann. Folgende C-Struktur wird dabei verwendet:

```
struct BEFEHLE {
   int mnemonic; // Art des Befehls
   int next;     // naechster Befehl
   int alt;      // alternativer Befehl
   int token;    // bei Terminalen
};
```

Die folgenden Befehle werden generiert:

Befehl	Knoten	Befehl	Knoten
BRANCH		GOTO	
TEST		CALL	
RETURN			

20.4 Arbeitsweise der virtuellen Maschine

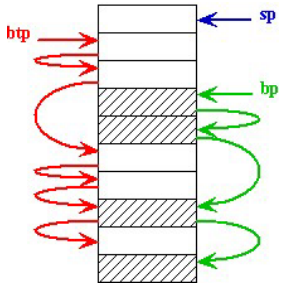
Die virtuelle Maschine verwaltet einen Stack mit zwei verschiedenen Arten von Stackframes:

1. **Entscheidungspunkte**
   Bei Alternativen (Befehl BRANCH) wird der erste Weg (Komponente `next` der Struktur `BEFEHLE`) weiter verfolgt, wohingegen der 2. Weg in diesen Entscheidungspunkten gemerkt wird. Sie beinhalten daher die folgenden Komponenten:
   - aktuelle Position des Scanners
   - Index des Stackframes des vorhergehenden Entscheidungspunktes
   - Index des beim Anlegen aktuellen Call-Frames
   - Index des ersten Befehls des alternativen Weges
2. **Call-Frames**
   Beim Auftreten von Nichtterminalsymbolen (Befehl CALL) müssen Informationen zum Weg nach dem Aufruf gespeichert werden (ähnlich wie beim Funktionsaufruf in Programmiersprachen). Dies sind:
   - Index des beim Aufruf aktuellen Call-Frames
   - Index des nachfolgenden Befehls im Feld der Befehle

Dazu werden 5 Variablen mit folgender Bedeutung verwaltet:

1. `ip` Index des aktuell abzuarbeitenden Befehls
2. `sp` Index des 1. freien Stackframes
3. `btp` Index des zuletzt angelegten Entscheidungspunktes
4. `bp` Index des zuletzt angelegten Call-Frames
5. `tok` Position des aktuellen Tokens als Index in eine lineare Liste aller Token

Die nebenstehende Skizze verdeutlicht diesen Sachverhalt.



Die Abarbeitung beginnt mit dem 1. Befehl der Regel, welche zum Startsymbol gehört. Dabei ist der Stack leer und `sp`, `btp`, `bp` und `tok` haben den Wert 0.

Die einzelnen Befehle bewirken folgendes:

1. **BRANCH**: Es wird ein neuer Entscheidungspunkt auf dem Stack abgelegt und dabei `btp` und `sp` aktualisiert. Es wird beim Befehl, auf den `next` verweist, weitergearbeitet (`ip=next`).
2. **GOTO**: Die Variablen und der Stack werden nicht geändert. Es wird beim Befehl, auf den `next` verweist, weitergearbeitet.
3. **CALL**: Es wird ein neuer Call-Frame auf dem Stack angelegt und dabei `sp` und `bp` aktualisiert. Dann wird beim 1. Befehl der gerufenen Regel weitergearbeitet, der in der `next`-Komponente gespeichert ist.
4. **RETURN**: Ist `bp!=0`, so wird `bp` aus dem aktuellen Call-frame zurückgestellt und beim dort angegebenen Befehl weitergearbeitet. Wichtig ist hier, daß dabei `sp` und `btp` nicht verändert werden. Der Stack wird also nicht verkleinert, damit die nach dem Aufruf der Regel eingerichteten Entscheidungspunkte im Backtracking-Fall auch erreicht werden können.
   Ist `bp==0`, so ist die Regel zum Startsymbol vollständig abgearbeitet und das Parsen erfolgreich beendet.
5. **TEST**: Falls das aktuelle Token gleich dem getesteten ist, so wird `tok` auf die Position des nächsten

Tokens gestellt. Dann wird bei `next` weitergearbeitet.
Sind beide Token nicht gleich und ist `btp!=0`, so wird Backtracking ausgelöst. Das heißt, `bp`, `btp` und `tok` werden aus dem stackobersten Entscheidungspunkt wieder hergestellt und `sp` wird auf `btp` gesetzt. Damit wird der Stack reduziert. Weitergearbeitet wird mit dem Befehl, welcher im stackobersten Entscheidungspunkt angegeben ist.
Ist `btp==0`, so gibt es keine offene Alternative mehr und das Parsen wird ohne Erfolg beendet.

20.5 Berechnung der Attribute

Die Berechnung der Attribute (meist Syntaxbäume) ist im Backtracking-Fall nur schwer rückgängig zu machen. Daher erfolgt sie in einem separaten Durchlauf nach erfolgreichem Parsen. Die Besonderheit dabei ist, daß in diesem 2. Durchlauf noch der Stackinhalt zur Verfügung steht, der im 1. Durchlauf aufgebaut wurde. Dieser wird nun als eine Art „roter Faden“ verwendet, so daß der Durchlauf deterministisch erfolgen kann.
Zum Zwecke der Attributierung generiert BTRACC zu jeder Regel eine Funktion. Die Regelparameter sind die Parameter der Funktion und die Berechnung der Attribute geschieht innerhalb des Funktionsblocks. Diese Funktionen rufen sich gegenseitig auf. Dies ist analog zum Verfahren des rekursiven Abstiegs, welches auch von LL(1)-Parsergeneratoren, wie z.B. COCO/R verwendet wird.
Die Funktionen ändern die Variablen wie oben beschrieben, manipulieren aber den Stack nicht. Im Falle einer Verzweigung (wenn also ein alternativer Weg existiert), wird verglichen, ob dieser Weg im stackobersten Entscheidungspunkt steht und die dort angegebene Position in der Tokenliste der aktuellen Position entspricht. Ist dies der Fall, so führte in der Parsing-Phase der 1. Weg zum Ziel. Dieser ist nun auch hier zu wählen. Ist es nicht der Fall, so ist der alternative Weg der richtige.
Auf diese Weise kann in der Attributierungsphase der korrekte Weg deterministisch (ohne Backtracking) gefunden werden und eine aufwendige Rücknahme der Attributberechnung entfallen.

Aufwandsschätzung

21 Aufwandsschätzung von Software-Reengineering-Projekten

Harry M. Sneed
Institut für Wirtschaftsinformatik Universität Regensburg

21.1 Rechtfertigung für Reengineering

Reengineering bestehender Anwendungssysteme ist eine preiswerte Alternative zur kompletten Neuentwicklung. Neuentwicklungen sind bekanntlich teuer und risikoreich. Auch mit den neuen agilen Entwicklungsansätzen sind die Kosten eines ganzen Anwendungssystems schwer abschätzbar. Bei den besten Schätzmethoden bleibt immer noch ein großer Unsicherheitsfaktor mit Schwankungen bis zu $\pm$ 80%. Ohne jedoch genau zu wissen, was ein Ersatzsystem kostet, ist es kaum möglich, eine Kosten/Nutzen-Rechnung zu erstellen. Also bleibt der Return on Investment für eine Neuentwicklung weitestgehend im Dunklen. Der ROI $= (\text{Nutzen} - \text{Kosten})/\text{Kosten}$ läßt sich nur errechnen, wenn die Kosten bekannt sind.

Der Hauptvorteil von Reengineering bleibt nach wie vor die Berechenbarkeit. Der Aufwand für ein Reengineering-Projekt läßt sich im Verhältnis zu einer Neuentwicklung relativ leicht und genau abschätzen. Andererseits kann der Nutzen einer ”reengineered” Anwendung nie so hoch sein wie der einer neuen Anwendung. Man kann zwar die Software modernisieren, die Oberflächen austauschen, die Daten migrieren und das System sogar webfähig gestalten. Dennoch bleibt die Fachlichkeit unverändert. Es handelt sich um eine neue Verpackung eines alten Inhaltes.
Dennoch kann ein Reengineering-Projekt auch bei beschränktem Nutzen ein höheres Return on Investment aufweisen als eine Neuentwicklung mit hohem Nutzen, aber ebenso hohen Kosten und Risiken. Vor allem sind es die überschaubaren Risiken, die ein Reengineering-Projekt zu einer attraktiven Alternative machen. Es wird weniger riskiert und weniger ausgegeben.

21.2 Unterschiedliche Zielsetzung

Das Ziel eines Reengineering-Projektes ist es, den bestehenden Code zu verändern und eventuell in eine andere Sprache zu transformieren. Eigentlich gibt es drei verschiedene potentielle Ziele die hier als

- Kapselung,
- Sanierung und
- Konvertierung

bezeichnet werden.

Das Ziel **Kapselung** bedeutet, daß der Code nur so weit geändert wird, daß man ihn in eine neue Umgebung oder Architektur einbinden kann. Beispielhaft für diese Projekte sind Integrationsprojekte, die darauf zielen, alte Hostprogramme in eine Webanwendung einzubinden. Hier wird nicht auf die Qualität und auch nicht auf die Sprache des Codes geachtet. Es geht ausschließlich darum, den Code in einer anderen Umgebung wiederverwendbar zu machen. Lediglich die Schnittstellen der Module werden angepaßt. Dies ist die billigste Art des Reeingineering, denn sie hat die geringste Auswirkung auf den Code, unter 20%.

Das Ziel **Sanierung** bedeutet, daß der Code im Hinblick auf eine Steigerung der internen Qualität verändert wird. Es soll möglich werden, den Code mit geringerem Aufwand zu warten. Restrukturierung und Refakturierung sind Beispiele für diese Art Reengineering. Der Code wird zwischen von 20-40% verändert.

Das Ziel **Konvertierung** bedeutet die Transformation des kompletten Codes in eine andere Programmiersprache, z.B. Assembler in COBOL oder COBOL in Java. In diesem Falle ist der Änderungsgrad mehr als 90%. Fast jede einzelne Anweisung wird übersetzt. Oft ist eine Konvertierung mit einer Sanierung gebunden, denn die neue Sprache wird nur strukturierte und modulare Programme verkraften. Je größer die Kluft zwischen der alten und der neuen Sprache, um so näher kommt ein Konvertierungsprojekt an ein Entwicklungsprojekt heran. Im Grunde genommen bewahrt man nur die fachliche Funktionalität und sonst nichts.

21.3 Der Weg zu einer systematischen Schätzung

In dem hier vorgestellten Verfahren zur Aufwandsschätzung und Risikoanalyse eines Software-Reengineering-Projektes werden acht Schritte benötigt:

- Schritt 1 ist die Erfassung der Reengineering-Anforderungen,
- Schritt 2 ist die Analyse der vorhandenen Programme und Daten,
- Schritt 3 ist die Auswertung der Analyseergebnisse,
- Schritt 4 ist die Übernahme der Quantitäts- und Komplexitätsdaten,
- Schritt 5 ist die Einstellung der Qualitätsziele und der Projekteinflußfaktoren,
- Schritt 6 ist die Kalibrierung der Produktivitätsdaten aufgrund der bisherigen Erfahrungen,
- Schritt 7 ist die Risikoanalyse und Errechnung des Risikofaktors und
- Schritt 8 ist die Schätzung der erforderlichen Aufwände und der Mindestdauer des Projektes.

Erfassung der Reengineering-Anforderungen

Im ersten Schritt bekommt der Besitzer des Legacy-Anwendungssystems einen Fragebogen, den er mit Hilfe eines erfahrenen Reengineering-Beraters auszufüllen hat. Es werden im Fragebogen Fragen zum Istzustand und zum gewünschten Sollzustand der betreffenden Software gestellt. Im Fragebogen werden drei Anforderungskategorien angesprochen:

- Komplexitätsanforderungen,
- Quantitätsanforderungen und
- Qualitätsanforderungen.

Bei den Komplexitätsanforderungen wird nach der gegenwärtigen und geplanten technischen Umgebung, nach den gegenwärtigen und geplanten Datenbanken, nach den gegenwärtigen und geplanten Programmiersprachen sowie nach der gegenwärtigen und geplanten Entwicklungsumgebung gefragt, um hier den Abstand zwischen Ist und Soll zu messen. Ein Reengineering-Projekt soll schließlich die Kluft zwischen Ist und Soll überbrücken, und es empfiehlt sich, gleich von Anfang an die Breite der Kluft zu kennen. Außerdem soll der Besitzer des Anwendungssystems die Komplexität seines jetzigen Systems und die des künftigen Systems einstufen und zwar auf einer ordinalen Skala von sehr niedrig, niedrig, mittel, hoch und sehr hoch. Bei den Quantitätsanforderungen geht es um das Mengengerüst der vorhandenen Software. Der Systembesitzer muß angeben

- die Anzahl der Dialoge,
- die Anzahl der Batchläufe,
- die Anzahl der Programme nach Sprache,
- die Anzahl der Datenbanken nach Typ,
- die Anzahl der Masken und Berichte und
- die Anzahl der JCL-Prozeduren bzw. -Skripte.

Damit hat man wenigstens eine Ahnung von der Größe des Zielsystems. Bei den Qualitätsanforderungen soll der Systembesitzer sein System selbst beurteilen, und zwar

- nach dem Grad der Benutzerzufriedenheit,
- nach der Performance,
- nach der Bedienbarkeit und
- nach der Wartungsfreundlichkeit.

Dazu verwendet er die gleiche ordinale Skala wie bei der Komplexität. Hinzu kommt die jährliche Fehlerrate, um die Zuverlässigkeit des Systems beurteilen zu können. Wenn er dies für das vorhandene System gemacht hat, soll er das gleiche auf das geplante System projektieren. Damit läßt sich der Grad der Qualitätssteigerung grob bemessen.

Schließlich soll der Benutzer aus seiner Sicht die wesentlichen Risiken für ein Reengineering-Projekt auflisten, ihre Wahrscheinlichkeit angeben und den Grad ihrer Auswirkung auf die Projektkosten schätzen. Diese Information soll der späteren Risikoanalyse zugute kommen.

Analyse der bestehenden Software

Im zweiten Schritt werden sämtliche Source-Programme, Masken, Datenbankschemen und JCL-Prozeduren des Zielsystems einer statischen Analyse unterzogen. Für jeden Sourcetyp muss es einen eigenen Analysator geben. Dennoch produziert jeder Analysator in etwa die gleichen Berichte.

Zum Ersten entsteht ein Mängelbericht, in dem Schwachstellen und Normabweichungen im Code registriert werden. Dieser zeugt von der Qualität der jeweiligen Sourcen. Die Mängel werden in schwere, mittlere und leichte Mängel kategorisiert und eine Konformitätsrate relativ zur Sourcegröße errechnet.

Zum Zweiten entsteht ein Metrikbericht mit Quantitäts-, Komplexitäts- und Qualitätsmetriken. Die Quantitätsmetriken belegen die Größe der Software z.B. in Codezeilen, Anweisungen, Data-Points und Function-Points. Es werden auch jede Menge andere Eigenschaften gezählt, wie die Anzahl Attribute, die Anzahl Referenzen, die Anzahl Schnittstellen und die Anzahl Bedingungen. Für das Reengineering besonders relevant ist die Anzahl wiederverwendbarer Datenstrukturen und Programmbausteine sowie die Anzahl konvertierbarer Datenstrukturen und Codeabschnitte.

Zu den Komplexitätsmetriken gehören die Datenkomplexität, die Datenverwendungskomplexität, die Datenzugriffskomplexität, die Schnittstellenkomplexität, die Ablaufkomplexität, die Entscheidungskomplexität, die Verzweigungskomplexität und die Sprachkomplexität.

Zu den Qualitätsmetriken gehören die Konformität, Modularität, Flexibilität, Portierbarkeit, Testbarkeit, Wiederverwendbarkeit, Konvertierbarkeit und allgemeine Wartbarkeit.

Zur Nachdokumentation gehören Strukturdiagramme, Datenflußtabellen, Datenbäume und Pseudocodes bzw. Struktogramme. Mustermasken werden aus den Maskenbeschreibungen und Datenbäume aus den Datenbanken abgeleitet. Diese Dokumente gewähren Einblick in die Logik der Programme sowie in die Zusammensetzung der Datenstrukturen. Sie werden benutzt, um die Risiken der Codesanierung und -konvertierung abschätzen zu können.

Zuletzt wird eine Exportschnittstelle mit den wichtigsten Metriken für das Schätzwerkzeug - SoftCalc - generiert. In dieser Schnittstelle werden die Anweisungen, Datenelemente, Ein- und Ausgaben und die Prozeduren gezählt und die Komplexität des Codes als hoch, mittel oder niedrig eingestuft. Von dieser Schnittstelle werden dreierlei Arten erstellt - eine für die Programme bzw. die Prozesse, eine für die Dateien und Datenbanken und eine für die Systemschnittstellen. Damit ist die Brücke zum 4. Schritt geschaffen - zur Übertragung der Quantitäts- und Komplexitätsdaten in das Schätzwerkzeug.

Auswertung der Analyseergebnisse

Im dritten Schritt werden die Ergebnisse aus der Systemanalyse ausgewertet. Ausgewertet werden

- die Metriken,
- die Mängelberichte und
- die Programmdokumente,

um zu einer Gesamtbewertung des Systems zu kommen. Aus den Metriken ist zu entnehmen, wie es mit der Größe, Komplexität und Qualität der Software im Ganzen steht. Die Größe drückt sich in verschiedenen Mengen aus wie z.B. Modulen, Codezeilen, Anweisungen, Bedingungen, Verzweigungen, benutzte Daten, Data-Points, Function-Points und Object-Points. Kein einziges Maß ist für sich genügend, denn jedes Maß mißt eine andere Eigenschaft der Software. Die Modulanzahl gibt an wie viele Einheiten zu verarbeiten sind. Die Anzahl Codezeilen erfaßt das rein physikalische Volumen des Codes, während die Anzahl Anweisungen das logische Volumen ausdrückt. Die Anzahl Bedingungen ist ein Indikator für die Komplexität der Entscheidungslogik und die Anzahl Verzweigungen ein Maß für die Komplexität der Ablauflogik. Die Anzahl benutzter Daten verweist auf die Größe der Datendomains. Data- Points sind ein aggregiertes Maß für die Größe des Datenmodells in Bezug auf die Entitäten, Beziehungen, Attribute und Sichten in den E/R-Diagrammen. Function-Points hingegen sind ein aggregiertes Maß für die Größe des Funktionsmodells in Bezug auf die Kanten und Knoten in einem Datenflußgraph. Object-Points sind schließlich ein aggregiertes Maß für die Größe eines Objektmodells an Hand der Anzahl Klassen, Schnittstellen, Attribute und Methoden. Die wahre Größe eines Softwaresystems läßt sich nur über die Kombination dieser vielen Einzelgrößen messen. Software ist letztendlich ein multidimensionales Phänomen, das mit einem einzigen Meßwert nicht zu fassen ist.

Übernahme der Quantitäts- und Komplexitätsdaten

Im vierten Schritt handelt es sich im Wesentlichen um einen technischen Vorgang. Die Exportschnittstellen, die aus der Sourceanalyse erzeugt wurden, werden mit einem Eintrag pro Programm, Schnittstelle und Datei bzw. Datenbankentität in die Datenbank des Schätzwerkzeugs übertragen. Danach sind die Prozeß-, Daten- und Kommunikationstabellen des Schätzwerkzeugs mit Daten aus dem vorhandenen System gefüllt. So bleibt es dem Schätzer nur übrig, die Änderungsrate einzustellen. Die Änderungsrate ist der Prozentsatz, zu dem jedes Element - Prozeß, Schnittstelle und Datentabelle - zu ändern ist.

Einstellung der Qualitätsziele und der Projekteinflußfaktoren

Im fünften Schritt obliegt es dem Schätzer, die Qualitätsziele zu definieren und die Projekteinflußfaktoren zu bestimmen. Bei den Qualitätszielen hat er die Möglichkeit, den Grad der Steigerung bzw. Senkung der Qualität gegenüber der Qualität des bestehenden Systems zu setzen. Die bestehende Qualität ergibt sich aus der Metrikbewertung des alten Codes. Z.B. hat das alte System eine Modularität von $0,45$. Wenn das neue System den gleichen Grad an Modularität haben soll, ist der Qualitätsjustierungsfaktor bzw. Quality Adjustment Factor $= 1$. Also wird das keinen Einfluß auf den Aufwand haben. Wenn aber die Modularität steigen soll von $0,45$ auf $0,60$, dann ergibt sich aus der Formel $\frac{\text{Soll-Qualität}}{\text{Ist-Qualität}}$ ein Multiplikationsfaktor von $1,33$ oder 33% mehr Aufwand. Andererseits, falls der Anwen-

der bereit sein sollte, einen Verlust an Modularität in Kauf zu nehmen, z.B. von $0,45$ auf $0,30$, dann ist der Multiplikationsfaktor $0,67$ oder 33% weniger Aufwand.

Die Kalibrierung der Produktivitätsdaten

Produktivität wird an Hand der Anzahl Anweisungen, Function-Points, Data-Points, Object-Points und Dokumente pro Personenmonat gezählt. Da dies von Betrieb zu Betrieb stark variieren kann, bleibt es jedem Betrieb überlassen, seine eigene Produktivitätskurve zu plotten. Man kann zwar Produktivitätsdaten aus der einschlägigen Literatur oder von anderen Betrieben übernehmen, aber dies dürfte nur als Ausgangsbasis dienen, denn Studien belegen Produktivitätsunterschiede von 1:5 zwischen Firmen. Für Reengineeringprojekte sind diese Schwankungen nicht ganz so stark, können aber bis zu 1:2 schwanken.

Sanierungsprojekte lassen sich am besten durch Anweisungen berechnen. Konvertierungsprojekte werden in der Regel über Codezeilen abgerechnet. Kapselungsprojekte beschränken sich auf ein Reengineering der Schnittstellen. Also spielt die Anzahl Anweisungen keine Rolle. Hier sind die Function-Points ausschlaggebend.

Die Berrechnung des Risikofaktors

Der letzte Schritt vor der eigentlichen Aufwandsschätzung ist die Analyse der Risiken und die Errechnung des Risikofaktors. Bereits im ersten Schritt hat der Besitzer des Legacy-Systems die Risiken für das Reengineering- Projekt aus seiner Sicht identifiziert und deren Wahrscheinlichkeit des Auftretens geschätzt. Jetzt werden diese Risiken im Schätzwerzeug erfaßt und durch andere Risiken ergänzt, die sich im Laufe der Analyse ergeben haben.

Für jedes Risiko wird nicht nur die Wahrscheinlichkeit des Auftretens als Prozentsatz von 0 bis 99% angegeben. Es wird auch die Risikoauswirkung als Multiplikationsfaktor von $0,1$ bis 2 angegeben. Dieser Multiplikationsfaktor drückt aus, welche zusätzlichen Kosten durch das Auftreten des Risikos verursacht werden. Der Risikofaktor ist das Produkt aus der Wahrscheinlichkeit und der Auswirkung plus 1

$$\text{Rf} = 1 + (\text{Wahrscheinlichkeit} \times \text{Auswirkung})$$

Um die Auswirkung der Risiken zu mildern, können Gegenmaßnahmen eingesetzt werden. Die Gegenmaßnahmen können ein Risiko um n% (1 bis 99%) neutralisieren. Somit ist der endgültige Risikofaktor

$$\text{Rf} = 1 + (\text{Wahrscheinlichkeit} \times \text{Auswirkung}) \times \text{Gegenwirkung}$$

Da es aber nicht ein Risiko, sondern mehrere gibt und die Risiken sich gegenseitig potenzieren, wird der Gesamtrisikofaktor als das Produkt aller einzelnen Risikofaktoren errechnet. Bei vier Risikofaktoren von $1,125$ ist dann das Gesamtrisko $1,6$. D.h., zum Abfangen dieser Risiken werden 60% mehr Kosten kalkuliert.

Schätzung des Aufwands

In diesem letzten Schritt wird nach jeder Methode, die der Benutzer auswählt, eine Kalkulation durchgeführt. Je nachdem welche Methode gewählt wird, wird was anders gezählt — Anweisungen, Komponenten, Function-Points, Data-Points, Object-Points, u.s.w. — um zur Systemgröße zu kommen. Diese Größe wird zunächst mit dem Qualitätsfaktor und anschließend mit dem jeweiligen Projekteinflußfaktor justiert. Der Qualitätsfaktor ist für alle Methoden gleich. Der Einflußfaktor ist jedoch, wie aus dem 5. Schritt hervorging, für jede Methode anders. Die Methoden unterscheiden sich also dadurch, wie sie die Systemgröße ermitteln und welche Einflußfaktoren sie hinzuziehen. Um so wichtiger ist es, nach verschiedenen Methoden vorzugehen, um zu erkennen, wo sie voneinander abweichen.

Mit der justierten Größe wird jetzt in die Produktivitätstabelle eingegangen und der zu dieser Größe passende Aufwand in Personenmonaten herausgeholt. Dies ist der entscheidende Schritt im Schätzverfahren, bei dem Größe in Aufwand umgesetzt wird. Wenn hier die Produktivitätsdaten nicht stimmen, wird auch das genaueste Größenmaß zu irreführenden Ergebnissen führen.

Nun erfolgt der Einfluß des Risikofaktors, der mit dem Aufwand multipliziert wird. Ein Risikofaktor von $1,6$ wird den Aufwand um 60% erhöhen. Dieser justierte Aufwand fließt dann in das Endergebnis ein. Mit der COCOMO-Formel $2,5$ Aufwand ^{0,35} wird aus dem Projektaufwand die Mindestprojektdauer errechnet. Allerdings wird diese Formel für Reengineering-Projekte verändert, da im Reengineering-Projekt eine viel größere Parallelität in der Arbeit möglich ist. Reengineering-Projekte dauern in der Regel nur halb so lange wie Entwicklungsprojekte der gleichen Größenordnung. Dies ist neben den geringeren Kosten und den reduzierten Risiken ein Hauptmotiv für Reengineering.

22 Ist Web-to-Host bereits alles? Die Kunst der Integration

Christian Synwoldt
Fogelberg & Partner GmbH, Frankfurt christian.synwoldt@netphantom.de

Der weitaus größte Teil aller geschäftskritischen Anwendungen wird nach wie vor auf Mainframe-Systemen abgewickelt - und deren Ablösung ist ferner denn je. Neben der traditionellen Rolle in der Unternehmens-IT steht inzwischen die Nutzung über das Internet immer mehr im Vordergrund. Und es drängt sich die Frage auf: Ist Web-to-Host bereits alles?

Schon auf den ersten Blick handelt es sich bei dieser Wortschöpfung um einen Irrtum: Nicht das World wide Web kommt zum Host, sondern der Mainframe soll ein (neues) Tor zum Internet erhalten! Waren dies ursprünglichen 3270-Terminals, die rasch durch PCs mit Terminal-Emulationen abgelöst wurden, so stehen heute Browser-basierte Clients im Mittelpunkt des Interesses - das Fat Client Paradigma scheint überholt. Wie aber werden die Mainframe-Applikationen über das Internet genutzt?

Können komplexe und in der Bedienung wenig komfortable Host-Anwendungen tatsächlich einfach ins Web gebracht werden (= Host-to-Web)? Wer sind die Benutzer - gut geschulte Sachbearbeiter oder ungeübte Gelegenheitsanwender? Was sind ihre Aufgaben? Selbst wenn - auf Grund der Art der Nutzung - ein umfassendes Re-Design zunächst nicht notwendig erscheint, so ist doch gerade im Unternehmensumfeld das Zusammenspiel mit anderen Applikationen zunehmend wichtiger. Folglich kann ein unreflektiertes Web-Enabling auch nur sehr kurzfristig Erfolg versprechen.

Von der Migration zur Integration

Aber noch ein ganz anderer Aspekt wird erkannt: Dominierten bislang Ideen wie die Ablösung des Mainframes durch Client-/ Server-Systeme, so ist inzwischen klar geworden, dass trotz großer Migrationsanstrengungen immer mehr der technischen Entwicklung hinterher gelaufen wird. Infolgedessen wird auch eher an die Integration der verschiedenen IT-Welten gedacht. Das führt dazu, dass inzwischen bis zu 40% aller IT-Aufwändungen im Zusammenhang mit der Integration der unternehmensweit verteilten Applikationen stehen. - Letztlich wird dieses Vorgehen als deutlich weniger Risiko behaftet angesehen, da die wohl erprobten Backend-Systeme praktisch unangetastet bleiben. Doch auch Integration ist nicht gleich Integration: Mittlerer Weile geht der Trend zu standardisierten Konnektoren und Lösungen die eine unternehmensweite Integration unterstützen.

Hohe Kosten ...

Da die Aufwändungen für die Integration unternehmensweit verteilter Applikationen so enorm hoch sind, stellt sich unwillkürlich die Frage, warum nicht an einer zentralen Applikations- und Systeminfrastruktur festgehalten wird? Das Thema Enterprise Application Integration (EAI) wäre unmittelbar von der Tagesordnung! Doch es gibt treibende Kräfte:

- Neue Anforderungen der Endanwender machen die Integration der Benutzerschnittstelle mit Back-end-Systemen notwendig.
- Unternehmenszusammenschlüsse bedingen die Notwendigkeit schneller Integration extrem heterogener IT-Landschaften
- Integration von performanten und zuverlässigen Altanwendungen mit modernen Softwarepaketen

...führen zu verschiedenen Ansätzen

In die Integration von Geschäftsprozessen sind regelmäßig heterogenste Applikationen und Systeme involviert - die Spanne reicht vom Mainframe bis zum Java-Applikationsserver mit EJB-Containern. Entsprechend hoch ist die Anzahl der Freiheitsgrade, die von einer Integrationslösung erwartet werden: Von der Präsentationsebene bis hin zur Integration von Funktionen und Daten.

Integration auf Präsentationsebene

Mittels eines Java Frameworks lässt sich ein integrierter GUI Arbeitsplatz für alle Geschäftsprozesse erstellen. Prinzipiell ist die Funktionalität durch die zu integrierenden Anwendungen determiniert. Darüber hinaus ist hier besonderes Augenmerk auf die Performance zu legen, da meist zusätzliche Softwareschichten einbezogen werden müssen. Erst wenn das gewählte Tool seinerseits entsprechend leistungsfähige Schnittstellen besitzt, kann über die

gemeinsame Oberfläche sogar weitere Funktionalität eingebunden werden.

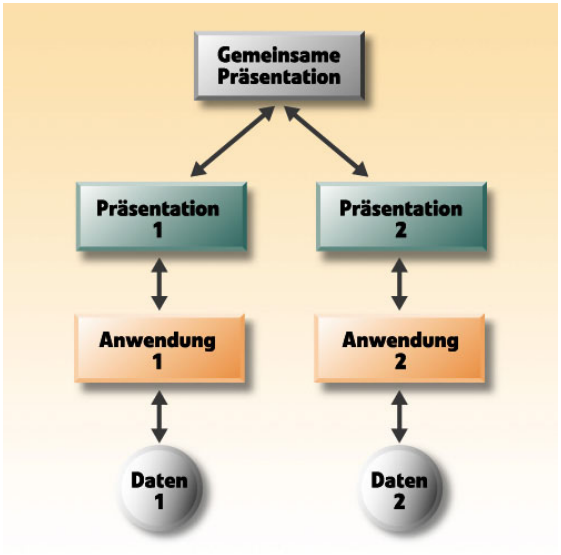


Abbildung 22.1. Erst durch eine Kapsel auf Präsentationsebene wird für den Endanwender eine durchgängige Bedienung erzielt.

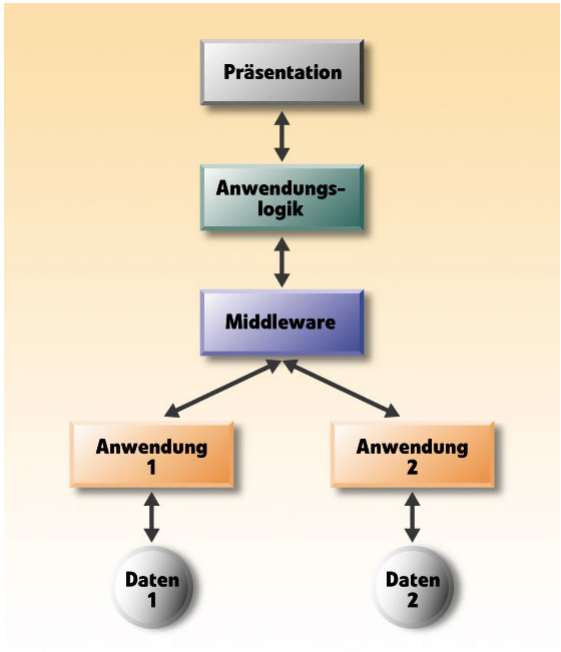


Abbildung 22.2. Die Middleware ermöglicht eine gemeinsame Sicht auf die Geschäftsprozesse.

Integration auf Datenebene

Mittels neu programmierter Logik werden die notwendigen Geschäftsprozesse abgebildet, die auf die Datenbestände vorhandener Applikationen zugreifen. Dabei sind regelmäßig Fragen der Integration durch die Umgebung der bisherigen Applikationen, die ebenfalls auf den Datenbeständen arbeitet, zu beachten. Auf der anderen Seite kann beim Einsatz geeigneter Tools und Paradigmen (Use Cases) ein hohes Maß an Widerverwendbarkeit der neuen Komponenten erreicht werden, obwohl diese zunächst neu erstellt werden müssen.

Funktionale Integration

Über eine Middleware-Schicht wird versucht, die in abgeschlossenen Modulen hinterlegten Funktionen zusammen zu führen. Bei diesen Modulen kann es sich um EJBs oder Mainframe Transaktionen handeln - der Bandbreite ist praktisch unbegrenzt.

Eine wesentliche Schwierigkeit bei dieser Form der Integration liegt meist in den technisch sehr unterschiedlich ausgeführten Interfaces, die erst mittels Konnektoren an die Middleware angefügt werden können. Auf der Basis eines Aufbaus aus Middleware plus Konnektoren lassen sich Softwarekomponenten mit hoher Widerverwendbarkeit gestalten. Gleichzeitig ist bei diesem Ansatz zu bedenken, dass Modifikationen in den zu integrierenden Anwendungen vorgenommen werden müssen.

Das Tool: NetPhantom

Über zahlreiche so genannte User Exits kann mittels gängiger Java-Standards (J2EE) gegen den Serverprozess programmiert werden. Diese Schnittstellen stellen nicht nur Client- wie auch Host-seitig Interfaces zur Verfügung, sondern erlauben ebenso Eingriffe in die Zugriffssteuerung oder die Kontrolle des Serverprozesses als solchem.

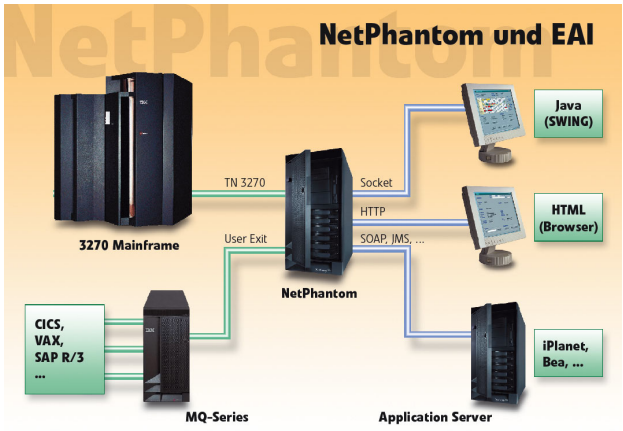


Abbildung 22.3. NetPhantom als zentrale Middleware: Für eine gemeinsame Sicht auf die Geschäftsprozesse und die funktionale Integration von Applikationen.

Interfaces für Menschen ...

Je nach gewünschter Funktionalität der Benutzungsoberfläche und der technischen Randbedingungen können verschiedene Client-Interfaces genutzt werden:

- Java-SWING
- HTML-Interface

... und Maschinen

Moderne IT-Architekturen sind auf die direkte Integration unterschiedlichster Systeme angewiesen! Neben den traditionellen Mainframe-Anwendungen spielen hier regelmäßig auch Applikationsserver (EJB, JSP, JMS) eine wichtige Rolle. Für die - ansonsten meist schwierige - Integration beider Welten wird über ein Remote Application Protocol Interface (RAPP-API) der Weg zu neuen Standards (u. a. SOAP, Web Services, ...) geöffnet.

XML/Query

---

23 Simplifying Source Code Analysis by an XML Representation

G. Fischer, J. Wolff v. Gudenberg
Universität Würzburg, Institute for Computer Science, Am Hubland, D-97074 Würzburg,
Germany, {fischer,wolff}@informatik.uni-wuerzburg.de

Abstract

JAML is an XML notation for Java source code. It retains all formatting information and comments, hence the original source is obtained by stripping all XML–tags and –attributes. Simple queries allow to retrieve information about the source that considerably improves the understanding of the program.

Keywords: XML, source code representation, source code analysis, tool

23.1 XML Representation of Source Code

The source code of a program is the most important means of labour for a software developer. Even in times where source is partially generated automatically by case tools or other integrated programming environments, it is essential that the source code is readable and comprehensible. Hence, source is usually formatted to support structural information. Linebreaks, empty lines, spaces and indentation group together or seperate code in order to enhance the readability, properly chosen names and comments help to understand the meaning. A lot of programming conventions therefore impose specific rules how to write the source code. Some tools even process the contents of (structured) comments.

On the other hand, a source code document is a structured document rather than a free-form text. It obeys certain rules given by the programming language. With these rules information from the code can be extracted. The more information we can get out of the source, the better.

Our goal is to retrieve as much information as possible out of the source text. We want to performe different kinds of analyses with different kinds of data in different degrees of detail.

Usually, the compiler or parser extracts (and derives) all the information from the source code that is needed to compile the program. This includes information about parameter types of a method or operation, points of variable definitions, etc. It does not include, however, comments or the meaning of names. The parser stores its information in internal data structures like a parse tree (abstract syntax tree) and a symbol table. These structures usually are not accessible, and, even if they are, they are hard to read and comprehend without deeper knowledge of the compilation process.

23.2 JAML

Hence we propose to store all the data needed for various checks in an augmented version of the source code document. As we already stated, source code is a structured document. The best way to represent this structured document, is to choose an appropriate XML application [2] where all the structural information is represented as tags. Two rules are mandatory:

- The original source code is obtained by removing all the tags.
- Comments and white spaces are not changed.

It is essential for program comprehension to have all information available.

The developer will not recognize his source code or accept

any modifications or transformations on it, if these rules do not hold. Also exact localisation of problems would be wearisome in a changed/reformatted document.
The rules are not fulfilled by JavaML [1], a XML representation for Java Source. The second rule is even failed by most parsers.
In contrast to srcML [3] we restrict ourselves to Java, since we are interested in a working and applicable tool. The ideas certainly can be applied to other programming languages.
Our XML (JAML) representation of the source document contains all the information the parser detected. This includes detailed information about types of variables or calls of methods. All additional information about the program is considered as meta-information and stored in XML-elements and -attributes. The JAML document further contains all the comments and text formatting properties of the original source code. Hence, the original source code may be regained by just stripping the tags.

23.3 Applications and Examples

Based on JAML many applications from querying the source code for certain constructs to automated source code quality checks, from improving program understanding to generating different views of the code can be realized. This section demonstrates some of the possibilities.

Querying

Information from the JAML -data can be queried using different techniques. Applicable methods depend on the environment but range from SQL or Prolog to more native XML languages like XQuery an XPath.
We will now demonstrate some examples that retrieve information from the JAML -document:

1. This expression selects the names of all publicly available attributes(fields):
   //class-definition/field-declaration
   [@visibility='public']/ @name
2. This expression selects the signatures of all native methods:
   //method-declaration[@native='true']/@signature
3. This expression selects all attributes in class `Sort` that are of a type that implements `Comparable`:
   //class-definition[@name='Sort']/field-declaration
   [@type-ref=//class-definition[all-interfaces/interface/
   @name= 'java.lang.Comparable']/@name]/@name

Views

Given the highly annotated source document, now different views of the source can be generated.

1. Obvoiusly the unchanged source can be reextracted from the JAML -document. Furthermore, the contained structural information can be used to automatically highlight the code and even add e.g. type information or documentation (from comments) at the point of usage, that can be displayed on demand.
2. The class hierarchy of all classes used directly or indirectly in the code can be easily derived from the document.
3. The signatures of methods and associated comments can be as easily retrieved from the document. This can facilitate e.g. the automatic documentation generation.

Navigation

JAML can also support navigation of the source code. A view can be extended to include not only additional information but also to provide links, that enable the user to navigate to other locations of interest for a certain code fragment. Class-, method- or attribute-definitions can be linked from where they are used. More such links, like from interface-methods to their implementations, can be thought of.

23.4 Conclusion and Future Work

In this paper we focused on simple applications of the JAML approach collected during the development of a Java program examination tool. These applications ensure some quality features by enforcing given programming conventions, e.g. Clearly, all these tasks may be performed by an appropriate Java compiler, but we have shown that the XML approach is easier and more flexible. This ease of use and the flexibility proof that it is worth to assemble a comprehensive document containing all the information.
Being defined by a DTD, JAML provides a solid foundation for tools to be developed. It also is used to ensure the validity of JAML documents before further processing as well as the integrity of the JAML generator itself.

Bibliography

[1] *G. Badros:* JavaML: A Markup Language for Java Source Code Proceedings of WWW9, Amsterdam, 2000 http://www9.org/w9cdrom/342/342.html

[2] *G. Fischer, J. Wolff v. Gudenberg* JAML - An XML Representation of Java Source Code, Würzburg University, Technical Report, 2003

[3] *J. Maletic, M. Collard, A. Marcus* Source Code Files as Structured Documents, Proceedings of IWPC 2002, p.289-292

24 Reasoning about Source Code in XML–Representation

Marbod Hopfner
University of Tübingen, Wilhelm–Schickard Institute for Computer Science, Sand 13, D – 72076 Tübingen, Germany, hopfner@informatik.uni-tuebingen.de

Dietmar Seipel, Jürgen Wolff von Gudenberg, Gregor Fischer
University of Würzburg, Institute for Computer Science, Am Hubland, D – 97074 Würzburg, Germany, {seipel,wolff,fischer}@informatik.uni-wuerzburg.de

Overview of VISUR/RAR

We have implemented a PROLOG–tool VISUR/RAR for reasoning about various types of source code, such as PROLOG–rules or JAVA–programs. RAR provides *retrieval*, *update* and *inference* operations for a deductive database storing XML–representations of the investigated code, rules for analysing PROLOG–code based on suitable *dependency graphs* and rules for recovering the design of JAVA–software; it uses a query language FNQUERY that is based on path expressions. VISUR/RAR can be applied for improving the design of rule–based systems, for computing certain *software metrics*, and for supporting *refactoring techniques*. The obtained results are visualised using graphs or tables in VISUR.

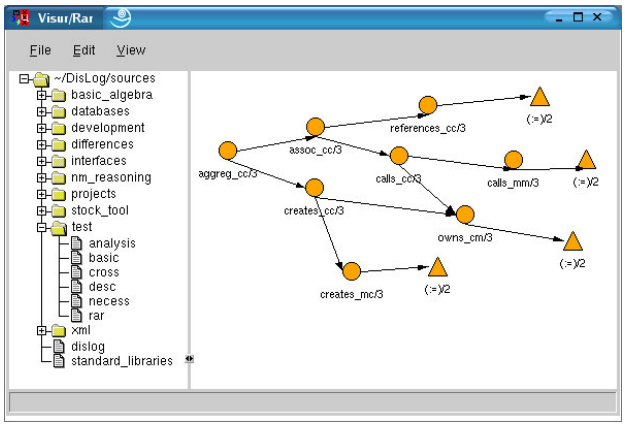


Figure 24.1. Graphical User Interface of VISUR

VISUR/RAR is part of the toolbox DISLOG [7], which is also developed under XPCE/SWI–PROLOG; the functionality of DISLOG ranges from reasoning in disjunctive deductive databases to applications such as the management and visualisation of stock information.

Representing Source Code in XML

The following JAVA–method implements the well–known sorting algorithm *merge sort* on a "globally defined" array. The method for merging two sorted sub–arrays is not shown here.

```
public void sort (int l, int r) {
   if (l<r) {
      int m = (l+r)/2;
      sort(l,m);
      sort(m+1,r);
      merge(l,m,r); } }
```

In [3] an XML–language JAML is introduced for representing the parse trees of JAVA–code – we could also use other XML–representations such as JavaML and srcML. In the following we show an abbreviated JAML –representation for a small portion of the JAVA–code, namely the method invocation `sort(l,m)`:

```
<method-invocation
   name="sort" qualifier=""
   argument-0="l" argument-1="m"
   signature= "sort(int, int)"
   return-type-ref="void">
   <le literal="sort" type-ref=""/>
   <symbol kind="left-parenthesis"/>
   <le literal="l" type-ref="int"/>
   <symbol kind="comma"/>
   <le literal="m" type-ref="int"/>
   <symbol kind="right-parenthesis"/>
</method-invocation>
```

In VISUR/RAR complex structured objects – i.e., JAVA–programs and PROLOG–programs – are conceptionally represented in XML–notation, which itself is physically stored in a PROLOG–term representation called *field notation*. We *query* and *manipulate* this field notation using a library FNQUERY developed in [7]. FNQUERY also contains additional, more *advanced methods*, such as the selection/deletion of all elements/attributes of a certain pattern, the transformation of sub–components according to substitution rules in the style of XSLT, and the manipulation of path or tree expressions.

Reasoning about Source Code

The following rules define some basic relations between methods and classes using path expressions in FNQUERY. For example, `references_cc/3` describes that the class `C1` has an attribute of the type `C2`. This holds, if at arbitrary depth in the JAML –representation `Code` of the source code there exists a `class-definition`–element `U` which itself contains a `field-declaration` `V` (at arbitrary depth) with an `identifier`–subelement. In that case `C1` is taken to be the `name`–attribute of `U` and `C2` is the `type`–attribute of `V`.

```
references_cc(Code,C1,C2) :-
   U := Code^_^class-definition,
   V := U^_^field-declaration,
   C1 := U@name,
   C2 := V@type,
   _ := V^identifier.

calls_mm(Code,M1,M2) :-
   U := Code^_^method-declaration,
   V := U^_^method-invocation,
   M1 := U@signature,
   M2 := V@signature.

owns_cm(Code,C,M) :-
   U := Code^_^class-definition,
   V := U^_^method-declaration,
   C := U@name,
   M := V@signature.
```

For visualising the call structure of rule–based systems we use the concept of *dependency graphs*, which is well–known from deductive databases [2]; the *call graph* of our example is shown in Figure 24.1.

The second layer of rules is based on the basic predicates. These rules allow for a more complex analysis of the JAVA–source code. They have been proposed in [6] for the pattern–based design recovery of JAVA–software. For example, *aggregations* and *associations* provide a higher–level view of the original *design* in contrast to the implementational view of the source code:

```
calls_cc(Code,C1,C2) :-
   calls_mm(Code,M1,M2),
   owns_cm(Code,C1,M1),
   owns_cm(Code,C2,M2).

assoc_cc(Code,C1,C2) :-
   calls_cc(Code,C1,C2),
   references_cc(Code,C1,C2).
```

Conclusions

The integration of XML–processing, visualisation, and reasoning in the logic programming environment XPCE–PROLOG has created a powerful and flexible tool. VISUR/RAR allows for analysing various types of complex structures including PROLOG–rules, JAVA–statements, and even ER–diagrams.

In the future, we will gradually extend VISUR/RAR with additional features. E.g., it is conceivable to integrate a possibility to move predicates from one location in a file to another location in another file (or the same file) using drag–and–drop operations; heuristic rules in RAR might be used for ensuring that the moves do not change the meaning of the program.

Bibliography

[1] *S. Abiteboul, P. Bunemann, D. Suciu:* Data on the Web – From Relations to Semi–Structured Data and XML, Morgan Kaufmann, 2000.

[2] *S. Ceri, G. Gottlob, L. Tanca:* Logic Programming and Databases, Springer, 1990.

[3] *G. Fischer, J. Wolff von Gudenberg:* JAML – An XML–Representation of JAVA–Source Code, Technical Report, University of Würzburg, 2003.

[4] *M. Fowler:* Refactoring – Improving the Design of Existing Code, Addison–Wesley, 1999.

[5] *M. Hopfner, D. Seipel:* Comprehending and Visualising Software based on XML–Representations and Call Graphs. Proc. 11th Intl. Workshop on Program Comprehension, IEEE Press, 2003.

[6] *J. Seemann, J. Wolff von Gudenberg:* Pattern–Based Design Recovery of JAVA Software, Proc. Intl. Symposium on the Foundations of Software Engineering 1998.

[7] *D. Seipel:* Processing XML–Documents in PROLOG, Proc. 17th Workshop on Logic Programming WLP 2002.

Tool-Präsentation

Bauhaus

Referent: R. Koschke (koschke@informatik.uni-stuttgart.de)

Anbieter/Hersteller: Universität Stuttgart, Institut für Softwaretechnologie

Kurzbeschreibung: Das Forschungsprojekt Bauhaus in Stuttgart erforscht Methoden und Techniken der Programmanalyse und Architekturrekonstruktion.

Im Bereich der Programmanalysen unterstützt Bauhaus die Erkennung duplizierten Quellcodes, die Suche nach Verwendungen potentiell uninitialisierten Variablen (globale Analyse) und totenRoutinen, verschiedene Zeigeranalysen, Ermittlung von Seiteneffekten sowie Program Slicing.

Auf Architekturebene bietet Bauhaus Methoden zur Komponentenerkennung, Herleitung und Validierung von Protokollen, Objektinteraktionen, Validierung von Software-Architekturen sowie Lokalisierung funktionaler Eigenschaften im Quellcode.

Gegenwärtig ist das System für Programme in C anwendbar. Eine Erweiterung auf objektorientierte Systeme ist in Arbeit.
Techniken im Bauhaus wurden bereits mehrfach im industriellen Umfeld erprobt. (`http://www.bauhaus-stuttgart.de`)

CCWB/1 CollCare Workbench

Referent: S. Knöpfler (`knoepfler@collogia.de`)
Anbieter/Hersteller: Collogia Unternehmensberatung GmbH (im Rahmen des BMBF-geförderten Verbundprojektes DARE (1996-1998) mit Verbundpartner FZI (Karlsruhe) und TWS (Stuttgart))
Kurzbeschreibung: CCWB/1 dient der interaktiven Analyse ("Exploration") des Kontroll- und besonders des Datenflusses von Programmen zum Zwecke der Code-Anpassung, etwa bei

- Migrationsprojekten (Wechsel von HW/Betriebssystem, Übergang zu RDBMS etc),
- Wrapping von Legacy-SW (z.B. zwecks Einpassung in OO-Architektur)
- und sonstigen begrenzten Umbauten (Einsatz-Beispiel: Y2k-Umstellung).

NetPhantom

Referent: C. Synwoldt (`christian.synwoldt@netphantom.de`)
Anbieter/Hersteller: TietoEnator Financial Solutions, Stockholm, Schweden
Kurzbeschreibung: Kapselung bestehender Mainframe-Applikationen, Integration mit modernen Architekturen und Ergänzung der Funktionalität
Dazu stehen die beiden Komponenten "Editor"(für die Gestaltung des grafischen Front-Ends) und "NetPhantom- Server" (als Ablaufumgebung und Integrations-Hub) zur Verfügung.

CMAD

Referent: T. Rötschke (`Tobias.Roetschke@es.tu-darmstadt.de`)
Anbieter/Hersteller: Philips Medical Systems, Eindhoven; TU Darmstadt
Kurzbeschreibung: CMAD erlaubt es, Trends von Metriken auf Architekturniveau darzustellen und darin zu navigieren.

FGM Flow Graph Manipulator

Referent: U. Kaiser (`Uwe.Kaiser@proetcon.de`)
Anbieter/Hersteller: pro et con GmbH, Chemnitz
Kurzbeschreibung: Reverse Engineering Tool

GUPRO - Generische Umgebung zum Programmverstehen

Referent: V. Riediger (`riediger@uni-koblenz.de`)
Anbieter/Hersteller: Universität Koblenz-Landan, Institut für Softwaretechnik
Kurzbeschreibung: GUPRO is an integrated workbench to support program understanding of heterogenous software systems on arbitrary levels of granularity. GUPRO can be adapted to specific needs by an appropriate conceptual model of the target software.
GUPRO is based on graph-technology. It heavily relies on graph querying and graph algorithms. Source code is extracted into a graph repository which can be viewed by an integrated querying and browsing facility. For C-like languages *GUPRO* browsing includes a complete treatment of preprocessor facilities. (`http://www.gupro.de`)

PapView

Referent: U. Kaiser (`Uwe.Kaiser@proetcon.de`)
Anbieter/Hersteller: pro et con GmbH, Chemnitz
Kurzbeschreibung: Kontrollflußgraph-Darstellung von C-Programmen

Strafunski

Referent: Ralf Laemmel (`Ralf.Laemmel@cwi.nl`)
Anbieter/Hersteller: R. Laemmel (VU & CWI, Amsterdam), J. Visser (SIG & CWI, Amsterdam)
Kurzbeschreibung: Strafunski is a Haskell-centered software bundle for generic programming and language processing. Strafunski provides programming support for generic traversal as useful for the implementation of program analyses and transformation components of language processors. The demo will illustrate Strafunski's use for Cobol reverse engineering. (`http://www.cs.vu.nl/Strafunski/`)