Projekt Bauhaus:
Interaktive und inkrementelle Wiedergewinnung von SW-Architekturen

www.informatik.uni-stuttgart.de/ifi/ps/bauhaus

Thomas Eisenbarth, Rainer Koschke,
Erhard Plödereder

Universität Stuttgart
Breitwiesenstr. 20-22
D-70565 Stuttgart, Germany

{eisenbts, koschke, ploedere}@informatik.uni-stuttgart.de

Jean-François Girard, Martin Würthner

Fraunhofer Einrichtung für Experimentelles Software Engineering (IESE)
Sauerwiesen 6
D-67661 Kaiserslautern, Germany

{girard, wuerthne}@iese.fhg.de

Das Projekt Bauhaus befaßt sich mit Techniken, das Verstehen eines umfassenden Programmsystems durch einen Wartungsingenieur zu vereinfachen. Es werden Techniken erforscht, wie eine Systemarchitektur bestehend aus Komponenten, Konnektoren und Subsystemen ausschließlich aus dem Quelltext eines Programmsystems halbautomatisch abgeleitet werden kann. Dabei steht der Wartungsingenieur im Mittelpunkt; sein Wissen wird benutzt, um einen interaktiven Prozeß zur Architekturerkennung zu lenken.

Bei Änderungen am Quelltext sollen diese gegen die abgeleitete Architektur geprüft werden können. Die gewonnene Architektur dient so dazu, mögliche Fehlersituationen während typischer Wartungsaufgaben aufzudecken. Darüberhinaus soll ein inkrementelles Vorgehen unterstützt werden, wobei Änderungen am Quelltext in der Architekturbeschreibung nachgeführt werden können.

1. Das Projekt Bauhaus

Das Projekt Bauhaus ist eine Kooperation zwischen dem Institut für Informatik der Universität Stuttgart, Abteilung Programmiersprachen und Compiler, und der Fraunhofer Einrichtung für Experimentelles Software-Engineering (FhG IESE), Kaiserslautern.

Generelles Ziel des Projektes Bauhaus ist es, die Arbeit eines Wartungsingenieurs zu unterstützen und zwar insbesondere auf der Ebene der Software-Architektur. Für viele in der Wartung anfallenden Arbeiten ist es für den Wartungsingenieur notwendig, sowohl die Soll- als auch die Ist-Architektur des ihm vorliegenden Systems zu kennen, um seine Arbeit effektiv durchführen zu können. Beispielsweise wird die tatsächliche Rolle einzelner Teile des Programms im Gesamtsystem oft erst auf der Ebene der Ist-Architektur klar, so daß sich der Wartungsingenieur nicht allein auf die Soll-Architektur verlassen kann. Vielfach liegt dem Wartungsingenieur jedoch weder Ist- noch Soll-Architektur vor. Unkenntnis der Architektur bei Änderungen am System führt nach und nach zu einem architektonischen Verfall, da beispielsweise Abstraktionsbarrieren durchbrochen oder Code unnötig dupliziert wird (Code Clones).

Besonders folgende Aufgaben sollen deshalb durch Bauhaus-Werkzeuge unterstützt werden: das Herleiten von Soll- und Ist-Architektur eines Altsystems, das Nachführen der Architektur bei Änderungen am System und schließlich die Abschätzung, wie sich eine geplante Programmänderung auf die Architektur auswirken wird. Erforscht werden Beschreibungsmittel sowie Analysemethoden und -werkzeuge, die es ermöglichen, Architekturen zu extrahieren, darzustellen und zu manipulieren. Dabei soll so weit wie möglich auf existierende Methoden und Werkzeuge zurückgegriffen werden.

Für die Basisanalysen des Quelltextes werden bekannte Techniken aus dem Compilerbau, insbesondere Daten- und Kontrollflußanalysen, angepaßt und eingesetzt.

Durch Experimente, teilweise innerhalb der Universität, teilweise in der Wirtschaft, versprechen wir uns Aufschluß über geeignete Darstellungs- und Interaktionsformen und über weitere Informationen, die der Wartungsingenieur für seine Arbeit benötigt.

In Abschnitt 2 dieses Artikels wird die Infrastruktur des Bauhauses vorgestellt. Danach werden in Abschnitt 3 die von uns bereits geleisteten sowie angestrebten Beiträge in den einzelnen Teilbereichen der Architekturerkennung beschrieben. 3.1 geht auf den iterativen Prozeß der AC-Erkennung ein, 3.2 beschreibt die hierbei eingesetzten Analyse-Techniken. Abschnitt 3.3 beschreibt die Erkennungsmechanismen für funktionale Komponenten. Abschnitt 3.4 befaßt sich mit der Erkennung von Konnektoren und Protokollen und 3.5 schließlich mit der abgeleiteten Architektur.

Abschließend werden in Abschnitt 4 die bislang erreichten Ziele und zukünftigen Forschungsgebiete des Bauhauses zusammengefaßt.

2. Infrastruktur im Bauhaus

Die Bauhaus-Werkzeuge umfassen derzeit ein Frontend für ANSI C (C++ und Ada 95 sind in Planung), das universellen Zwischencode (IML) erzeugt [2]. Der Zwischencode wurde mit dem Ziel entworfen, weitgehend programmiersprachenunabhängige Darstellungen zu ermöglichen, um so alle weiteren Analysen zu vereinfachen und deren Wiederverwendbarkeit zu erhöhen. Gleichzeitig enthält der Zwischencode aber auch ausreichende Information, die Analyseergebnisse in Form der ursprünglichen Gegebenheiten im Quellcode wiederzugeben. Beim Entwurf von IML wurde deshalb besonderer

Wert auf größtmögliche Quellennähe bei gleichzeitiger hoher Abstraktion gelegt.

In Abbildung 1 ist der Zwischencode dargestellt, wie er für die Schleife while (P) A; in C erzeugt wird (b), zusammen mit der von while unabhängigen Grundstruktur (a). Analysen können ignorieren, daß es sich hier um eine while-Schleife handelt und ein allgemeines Verfahren zur Schleifenbehandlung anstoßen, da die Semantik der while-Schleife durch einfachere IML-Konstrukte ausgedrückt wird, und diese muß die Analyse auf jeden Fall behandeln.

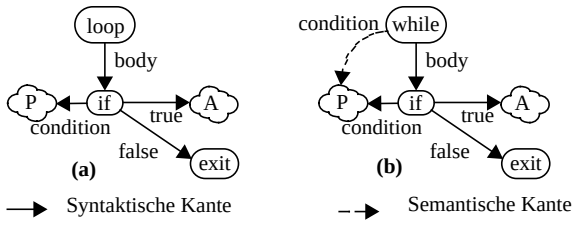

Abbildung 1: Beispiel für IML

Allerdings kann eine Analyse das Faktum, daß es sich hier eben um eine while-Schleife handelt, ausnutzen. Rückmeldungen an den Benutzer können in einer ihn nicht verwirrenden Form erzeugt werden.

Die hier dargestellte Situation der Spezialisierung wird durch Vererbung modelliert, d.h. der Schleifenknoten gehört zur Klasse while, die eine Unterklasse von loop ist. Die Semantik der Schleife kann vollständig erfaßt werden, ohne die Unterklasse des Knotens zu betrachten. Das hat den Vorteil, daß andere Sprachen mit z.B. anderen Schleifensemantiken ohne Änderungen an bestehenden Analysealgorithmen unterstützt werden können, da die Spezialitäten durch bereits bestehende Konstrukte ausgedrückt werden.

Derzeit wird der Zwischencode mit Datenflußinformationen angereichert. Dazu wird ein Generator implementiert, der eine spezielle Form der Static Single Assignment-Form (SSA) aufbaut [6], die eine kompakte Darstellung von Def-Use-Informationen sowie effiziente Datenabhängigkeitsanalysen erlaubt. Eine erste Version dieses Generators, der zunächst auf pessimistischen Annahmen über potentielle Datenabhängigkeiten beruht, wird in Kürze abgeschlossen werden. Analysen zur Verfeinerung der SSA-Form durch Points-To-Analysen zur Präzisierung des Aufrufgraphen bezüglich Funktionszeigern und Bestimmung der potentiellen Alias-Mengen schließen sich an.

Auf dem Zwischencode setzen alle folgenden Analysen auf. Außerdem wird aus dem Zwischencode eine stark abstrahierende Zwischendarstellung, der Resource Flow Graph (RFG) [2], abgeleitet. Dieser enthält nur noch die globalen Unterprogramme, Typen und Variablen als Basis-Knoten. Kanten zwischen den Basis-Knoten drücken die Beziehungen der Einheiten untereinander im Quellprogramm aus, siehe Abbildung 2. Der RFG dient, wie im folgenden beschrieben wird, als Ausgangsbasis für die Erkennung und Repräsentation von architektonischen Komponenten.

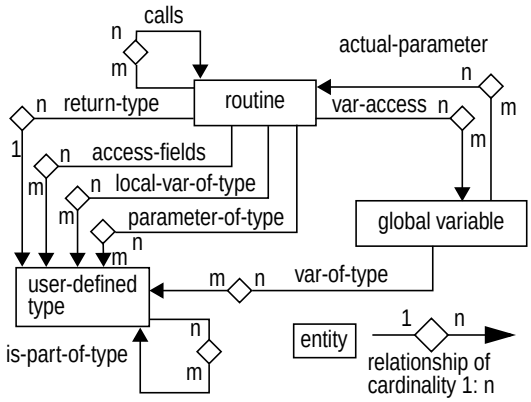

Abbildung 2: Entity-Relationship Diagramm für RFGs

Die Ergebnisse der Bauhaus-Analysen werden in einer modifizierten Version des Rigi-Editors visualisiert und lassen sich dort direkt manipulieren [4], siehe Abbildung 3. Aus dieser Abbildung ist sofort ersichtlich, daß automatische Analysen zur Architekturerkennung notwendig sind, denn das dargestellte Chaos an Knoten und Kanten läßt sich von Hand nicht mehr adäquat analysieren.

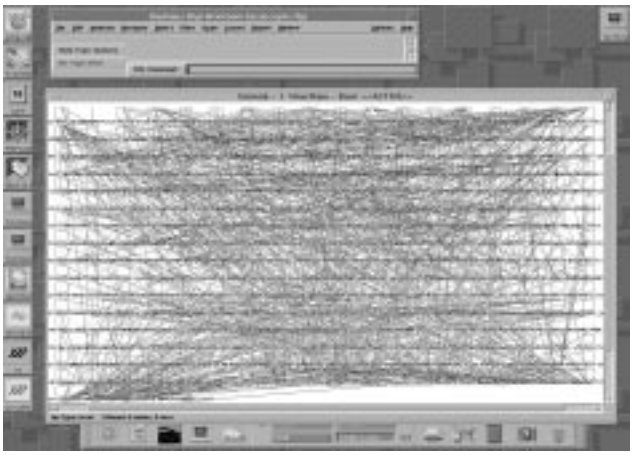

Abbildung 3: Oberfläche von Rigi

In Abbildung 4 ist der Ablauf der Programmanalysen – bis schließlich der RFG vorliegt – skizziert.

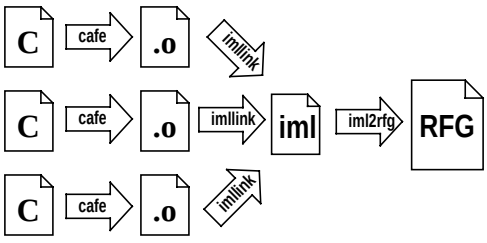

Abbildung 4: Analyse eines C-Systems

3. Vom Quellcode zur Architektur

Eine Architektur besteht aus Komponenten und Konnektoren [8]. Komponenten sind Berechnungseinheiten mit oder ohne eigenen Kontrollfluß. Konnektoren sind die Kommunikationsformen zwischen Komponenten (z.B. Unterprogrammaufrufe, Remote Procedure Calls, Dateien, Pipes).

Komponenten lassen sich weiter gliedern in dynamische Komponenten, d.h. solche, die erst zur Laufzeit generiert werden, wie z.B. Prozesse, und statische Komponenten, die bereits zur Übersetzungszeit festliegen. Letztere lassen sich in atomare Komponenten (ACs), d.h. kohäsive logische Module, und hierarchische Subsysteme gliedern. ACs sind atomar, d.h. sie lassen sich nicht weiter in ihrerseits strukturierte Untereinheiten gliedern. Sie stellen also z.B. reine abstrakte Datentypen und -objekte dar, können aber auch hybride Zwischenformen oder Sammlungen von Funktionen (z.B. mathematische Bibliotheken) sein.

Konnektoren können auf verschiedenen Ebenen auftreten und sind entsprechend vielfältig: Zwischen Systemen (z.B. als Dateiformate), zwischen Tasks (z.B. als Pipes), zwischen ACs (z.B. durch andere ACs). Es hängt von der gewählten Abstraktionsebene ab, ob eine AC oder ein Subsystem als Konnektor zwischen anderen Teilen des Systems gesehen wird.

Subsysteme fassen mehrere ACs, ihre Konnektoren und eventuell weitere Subsysteme zu größeren, hierarchisch gegliederten Einheiten zusammen.

3.1. Der iterative Erkennungsprozeß für ACs

Der Quelltext ist oft der einzige Informationsträger, der ein Altsystem wirklich verläßlich beschreibt. Deshalb versucht das Bauhaus eine Architektur direkt aus dem Quelltext abzuleiten.

Unsere Evaluation publizierter Techniken zur Erkennung von ACs hat ergeben, daß keine der Techniken die menschliche Erkennungsqualität erreicht [3]. Damit ist eine gänzlich vollautomatische Architekturerkennung praktisch nicht machbar. Deshalb haben wir einen Ansatz gewählt, der den Wartungsingenieur – und damit sein Wissen – bereits früh in den Prozeß der Architekturerkennung einbezieht und ihn ins Zentrum der Aktivitäten stellt.

Der generelle Prozeß zur Erkennung von ACs ist in Abbildung 5 dargestellt. Er setzt auf der RFG-Darstellung auf. Der Wartungsingenieur kann über eine modifizierte Version des Graphen editors Rigi [4] den RFG betrachten und interaktiv verschiedene Analysen zur AC-Erkennung anstoßen. Wir haben bei den Analysen zur AC-Erkennung sowohl eigene Ideen verwirklicht als auch bestehende Techniken adaptiert [7]. Beispiele für Analysetechniken zur automatischen AC-Erkennung werden in Abschnitt 3.2 erläutert.

Die von den Analysen vorgeschlagenen Kandidaten muß der Wartungsingenieur anschließend sichten, d.h. bestätigen (und so in die sogenannte Benutzersicht überführen) oder verwerfen. Die Benutzersicht stellt den gegenwärtigen Stand der Erkennung dar. Dabei stehen dem Wartungsingenieur eine Reihe von Basisinformationsdiensten (einschließlich eines direkten Bezugs zum Quellcode), automatische Layout-Möglichkeiten und grundlegende Verknüpfungsoperationen zur Verfügung. Außerdem kann er sich die einzelnen Kandidaten durch Metriken vorsortieren lassen. Er kann darüber hinaus selber atomare Komponenten zusammenstellen und bereits Bestätigtes direkt manipulieren.

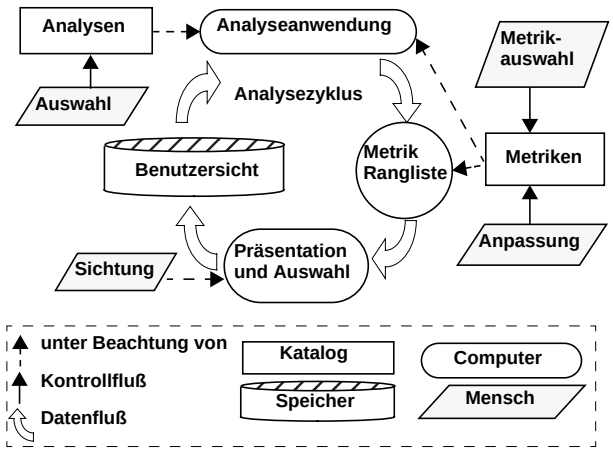

Abbildung 5: Der Analysezyklus

Durch inkrementelle Analysen lassen sich bezugnehmend auf die bestätigten Ergebnisse weitere ACs auffinden bzw. bereits existierende ergänzen. Es ergibt sich also ein inkrementeller Analyse-Zyklus. Durch mehrere Iterationen dieses Zyklus findet der Wartungsingenieur die atomaren Komponenten der Ist-Architektur.

Bisweilen werden jedoch sehr große oder sehr ungenaue Kandidaten vorgeschlagen. In diesen Fällen haben sich einige Techniken bewährt, um die Kandidaten zu verkleinern bzw. um das Vertrauen, das man einem Kandidaten entgegenbringen kann, zu steigern. Wirkungsvoll ist die Verknüpfung verschiedener Analysen durch primitive Operationen wie Schnitt und Vereinigung. Im Schnitt zweier Analyse-Ergebnisse liegen nur ACs deren Teile von beiden Analysen jeweils genau einer AC zugeordnet worden sind. Dadurch ist das Vertrauen in den Vorschlag gestiegen, da er zweimal bestätigt worden ist. Analysen können auch auf dem Inhalt einer AC aus einem vorherigen Analyselauf gestartet werden, um diese weiter zu zerteilen.

Eine weitere Methode, gute von schlechten Kandidaten zu trennen, ist der Einsatz von Metriken. Die Kandidaten werden durch Metriken, die unabhängig von der eingesetzten Erkennungstechnik ist, bewertet und dem Wartungsingenieur entsprechend präsentiert.

Ein Ansatz, der in dieselbe Richtung weist, ist eine Technik, die wir als *voting approach* bezeichnen. Dabei werden die einzelnen im weiteren hier skizzierten Erkennungstechniken benutzt, um die durch eine Technik gefundenen Kandidaten zu bewerten. Für jeden Basisknoten einer AC darf jede Technik abstimmen, ob sie ihn zu dem

Rest der AC hinzugefügt hätte. Daraus ergibt sich eine Bewertung der AC als ganzes.

3.2. Techniken zur AC-Erkennung

Atomare Komponenten bestehen aus einer Teilmenge der Basis-Einheiten eines Quellprogramms, also aus einigen seiner Unterprogrammen, globalen Variablen und Typen.

Man kann ACs nochmals in abstrakte Datentypen (ADTs), abstrakte Datenobjekte (ADOs) und hybride Komponenten unterteilen: ADTs enthalten nur Unterprogramme und Typen, ADOs nur Unterprogramme und Variablen. Hybride können alle drei Typen von Basis-Einheiten enthalten. Die Aufgabe der AC-Erkennung ist es also, aus der Menge aller Basis-Einheiten diejenigen zu identifizieren, die (mit hoher Wahrscheinlichkeit) zusammen eine AC bilden.

Wir haben die Techniken zur AC-Erkennung, die wir untersucht und beschrieben haben, in drei Klassen eingeteilt: verbindungsbasiert, metrikbasiert und graphbasiert.

3.2.1. Verbindungsbasierte Techniken

Verbindungsbasierte Techniken benutzen die direkten Beziehungen zwischen den Basis-Einheiten, um nahe zusammengehörige Einheiten zusammenzufassen. Diesem heuristischen Vorgehen liegt die Idee zugrunde, daß eng verwandte Basis-Einheiten auch in direkter Beziehung zueinander stehen. Zum Beispiel stehen alle Zugriffsfunktionen eines ADTs mit dem Typ, der die Daten des ADTs kapselt, durch parameter-of-type-Kanten in Beziehung, siehe Abbildung 6.

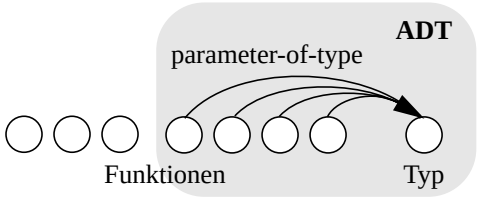

Abbildung 6: Beispiel für die ADT-Erkennung

Ein Beispiel für die verbindungsorientierten Techniken ist die Internal Access-Heuristik. Dieser Heuristik liegt die Vorstellung zugrunde, daß alle Unterprogramme, die auf die interne Darstellung eines Typs zugreifen, zusammen mit diesem Typ einen ADT bilden (analoges gilt, bezogen auf Variablen, für ADOs). In Abbildung 7 greift die Funktion *f* auf die Interna des Typs T zu. Als Zugriff auf die interne Darstellung gelten indizierte Zugriffe auf Arrays, Feld-Zugriffe auf Records und Dereferenzierungen von Zeigern. Die Tatsache, daß ein interner Zugriff vorliegt, wird als Attribut an den entsprechenden Kanten (use-edge, set-edge) des RFG notiert, da sich sonst interne von aggregierten Zugriffen nicht unterscheiden ließen.

```
typedef struct {int a; ...} T;
void f (T *pt){
  pT t = *pt;
  t.a = 5;   /* interner Zugriff */
  *pt = t;}
```

Abbildung 7: Beispielprogramm für Internal Access

Manchmal wird die Annahme dieser Heuristik nicht zutreffen, da z.B. aus Effizienzüberlegungen direkt auf eine Variable zugegriffen wird, anstatt eine Selektionsfunktion des zugehörigen ADT aufzurufen.

3.2.2. Metrikbasierte Techniken

Metrikbasierte Techniken nutzen Ähnlichkeitsmaße, um ein Clustering der Basis-Knoten des RFG zu ACs durchzuführen. Ein Beispiel für eine solche Technik ist *similarity clustering*. Als Ähnlichkeitsmaße dienen hier direkte, indirekte und informale Maße, jeweils auf zwei Knoten bezogen. Die Ähnlichkeit zweier Mengen von Knoten wird durch die paarweisen Ähnlichkeiten ausgedrückt.

Direkte Maße drücken strukturelle Ähnlichkeiten im RFG aus. Setzt ein Unterprogramm F eine Variable V, dann sind sich F und V ähnlich. *Indirekte Maße* drücken Ähnlichkeiten bezogen auf Nachbarn aus. Werden z.B. zwei Unterprogramme F, G vom selben Unterprogramm aus aufgerufen, dann sind F und G ähnlich. Ein *informales Maß* ist die Namensähnlichkeit zweier Bezeichner.

Da einzelne Metriken immer fehl gehen können, lassen sich Kombinationsmetriken definieren, die die Vorteile verschiedener Metriken vereinen und die Nachteile ausschließen sollen. Allerdings ergeben sich sehr viele Parameter, die eingestellt werden müssen. Z.B. ist die direkte Ähnlichkeit zweier Knoten A und B im RFG definiert als:

$$Sim_{direct}(A, B) = W(link(A, B))$$

W berechnet die gewichtete Summe über alle möglichen Kantentypen im RFG; zur Zeit sind dies 15 Typen. Es lassen sich also bereits bei dieser Teilmetrik 15 verschiedene Gewichte einstellen und von den gewählten Werten hängen die Qualität und die Art der gefundenen ACs ab. Werden Kanten, die zwischen Variablen und Unterprogrammen bestehen, mit dem Gewicht 0 belegt, werden beispielsweise keine ADOs gefunden.

Metrikbasierte Techniken sind – wie die verbindungsorientierten Techniken – Heuristiken, d.h. falls die zugrundeliegenden Annahmen verletzt sind, werden die Ergebnisse keine hohe Qualität aufweisen. Aus diesem Grund werden verschiedene Möglichkeiten erprobt, die Parameter mittels Optimierung für ein vorgegebenes System zu einzustellen. Die Idee hierbei ist, daß zunächst ein kleiner Ausschnitt des Systems durch andere Techniken oder von Hand analysiert wird, um Referenzkomponenten aufzustellen. Anhand dieser Referenzen werden die Parameter dann eingestellt. Dies kann beispielsweise durch Evolutionsstrategien oder numerische Optimierungsverfahren geschehen. Unsere Hoffnung ist, daß sich der Rest des Systems durch dieselben Parameterwerte sinnvoll gruppieren läßt.

3.2.3. Graphbasierte Techniken

Graphbasierte Techniken arbeiten auf den Strukturen des RFG. Eine solche Methode ist die Dominanzanalyse, die es z.B. ermöglicht, Unterprogramme als lokal zu einem einzelnen Unterprogramm oder einer AC zu erkennen. Solche Unterprogramme lassen sich in C nicht lokal

deklarieren, trotzdem ist die Information über diese Form der Lokalität sehr nützlich.

Die Dominanzanalyse wird auch in der Komponentenerkennung benutzt, siehe Abschnitt 3.3.

3.2.4. Evaluation des inkrementellen Ansatzes

Um den inkrementellen Ansatz, den wir im Bauhaus-Projekt verfolgen, zu evaluieren, wurde ein Experiment durchgeführt. Als Versuchspersonen dienten dabei zehn Studenten. Um zu große Abweichungen unter den Teilnehmern zu vermeiden, wurden die Techniken und Bauhaus-Tools zur AC-Erkennung im Vorfeld allen Teilnehmern vorgeführt. Eine Fünfergruppe analysierte dann mit Hilfe des Bauhaus-Rigis. Die andere durfte zwar den Rigieditor verwenden, aber ohne automatische AC-Erkennung; d.h. es standen die grundlegenden Informationen des RFGs mittels der Browsingfähigkeiten des Rigi-Editors zur Verfügung. Allerdings mußten diese nicht verwendet werden, beliebige Editoren und Suchtools waren erlaubt (wurden aber, außer Unix-grep und emacs, nicht benutzt).

Es wurde das Kernsystem von Mosaic (33KLoc), einem in C geschriebener Web-Browser, auf ACs hin untersucht. Dazu standen sechs Stunden Zeit zur Verfügung. Unsere Hypothese ist, daß in derselben Zeit mehr ACs (mit besserer Qualität) gefunden werden, wenn der Bauhaus-Rigi mit automatischen Analysen eingesetzt wird.

Diese Hypothese konnte bestätigt werden: Mit 90%-iger Wahrscheinlichkeit verbessert der Einsatz der automatischen Analysen die Zahl der in derselben Zeit gefundenen ACs. Im durchgeführten Versuch belief sich diese Verbesserung auf 35%.

3.3. Erkennung von funktionalen Komponenten

Der erste Schritt bei der Erkennung von Subsystemen besteht darin, Gruppen von Unterprogrammen zu identifizieren, die gemeinsam eine bestimmte Funktionalität für den Rest des Systems implementieren. Das Ziel ist es also, funktional zusammenhängende Gruppen von Unterprogrammen (sogenannte *functionally cohesive components*, oder kurz FCCs) zu finden, die tatsächlichen Systemfunktionalitäten zugeordnet werden können.

3.3.1. Dominanzanalyse

Ein vielversprechender Ansatz, um solche Komponenten zu finden, ist Dominanzanalyse [9]. Dazu wird der Aufrufgraph des Programms zunächst in einen Dominanzbaum überführt. Dieser stellt die Dominanzrelation dar (unter Ausschluß der transitiven Beziehungen): Ein Unterprogramm X dominiert ein Unterprogramm Y genau dann, wenn Y nur in Aufrufketten auftreten kann, in denen X enthalten ist (wobei Aufrufketten von einem bestimmen Wurzelknoten ausgehen).

Im Beispiel in Abbildung 8 ist A die Wurzel des Aufrufgraphs. Unterprogramm G kann von A aus nur aufgerufen werden, wenn Unterprogramm B in der Aufrufkette enthalten ist. Deshalb wird G von B dominiert, was im Dominanzbaum in Abbildung 9 daran zu erkennen ist, daß G in einem Unterbaum von B enthalten ist. Unterprogramm F hingegen wird von B nicht dominiert, da eine Aufrufkette von der Wurzel aus auch über C gehen kann.

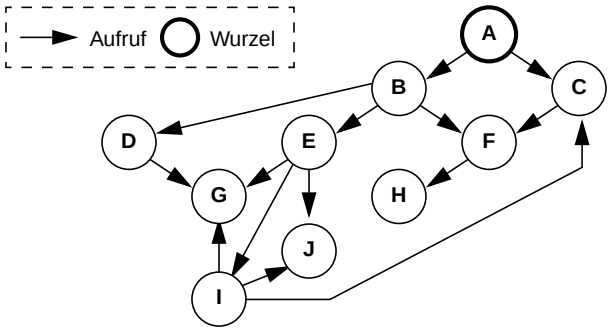

Abbildung 8: Beispiel eines Aufrufgraphs

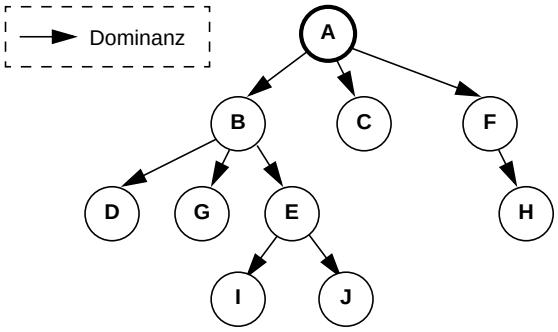

Abbildung 9: Zugehöriger Dominanzbaum

3.3.2. Hierarchische Komponentensicht

Da ein dominiertes Unterprogramm nur im Kontext seines Dominators aufgerufen werden kann, liegt es nahe, dominierte Unterprogramme mit ihren Dominatoren zusammenzufassen. Jeder Unterbaum des Dominanzbaums kann so als eine Komponente aufgefaßt werden.

Auf diese Weise entsteht eine Hierarchie von Komponenten. Im obigen Beispiel kann {E, I, J} als eine Komponente angesehen werden, das seinerseits zu einer übergreifenden Komponente {B, D, G, E, I, J} gehört.

Jede Komponente hat einen eindeutigen Eintrittspunkt (nämlich die Wurzel des Unterbaumes), der als Schnittstelle zum Rest des Systems aufgefaßt werden kann. Es ist aber zu beachten, daß die Komponenten nicht unbedingt in sich abgeschlossen sind. So ruft z.B. I außer J auch noch C und G auf, obwohl diese Unterprogramme nicht zur Komponente von I gehören. Allerdings gilt die Garantie, daß jede Komponente nur durch ihre Wurzel “betreten” wird.

3.3.3. Unterteilung großer Komponenten

Eine hierarchische Aufschlüsselung der Komponenten ist für viele Anwendungen hilfreich. Allerdings wird oft keine Hierarchie von Komponenten, sondern eine Menge unabhängiger Komponenten benötigt, die bestimmte Systemfunktionalitäten erfüllen.

Dies gilt zum Beispiel im Falle einer Wiederverwendung von Komponenten eines Altsystems in einem neuentworfenen System. Hier geht es darum, eine Menge von

Komponenten gezielt auf ihre Wiederverwendbarkeit zu untersuchen und gegebenenfalls anzupassen, z.B. durch eine zusätzliche Schnittstelle, die den wiederverwendeten Code in das restliche System integriert (Wrapping).

Für eine solche Verwendung ist die obige Hierarchie nicht direkt nützlich, denn sie enthält Komponenten auf vielen Granularitätsstufen. Spezielle große Unterbäume enthalten oft verschachtelte Unterbäume, die separat wiederverwendbar sind, da sie eine eigenständige Systemfunktionalität erfüllen.

Um große Komponenten aufzuteilen, kann man gezielt Unterbäume aus dem Dominanzbaum auswählen und von der übergeordneten Komponente abtrennen. Die Unterteilung eines Unterbaumes (*subtree promotion*) ist schematisch in Abbildung 10 dargestellt. Wenn z.B. im obigen Beispiel die Komponente {E, I, J} ausgewählt wird, bleibt von der übergeordneten Komponente {B, D, G, E, I, J} noch der Teil {B, D, G} übrig.

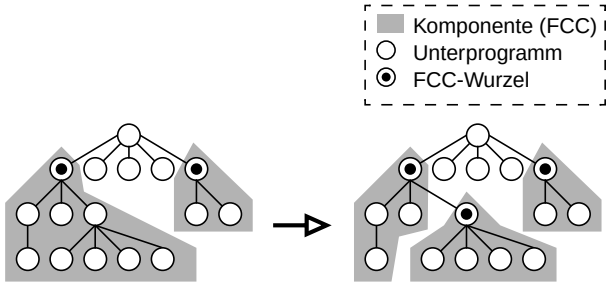

Abbildung 10: Unterteilung von Unterbäumen

3.3.4. Heuristiken zur Auswahl von Unterbäumen

In einem großen System ist die Auswahl der Unterbäume, die als eigenständige Komponenten betrachtet werden sollen, nicht trivial, insbesondere wenn der Dominanzbaum eine große Tiefe hat. Abbildung 11 zeigt einen Ausschnitt aus einem Dominanzbaum, der im Rahmen eines Industrieprojektes [1] erstellt und mit Rigi visualisiert wurde.

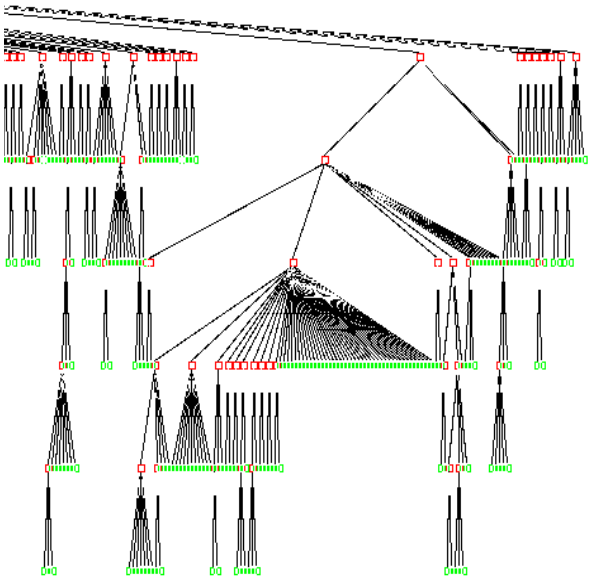

Abbildung 11: Ausschnitt aus einem Dominanzbaum für ein großes FORTRAN-System (RAMSIS)

Auswahl nach Größe. Eine einfache Heuristik, um diese Auswahl zu treffen, kann die Größe des Unterbaumes sein, d.h. die Anzahl der Unterprogramme im Unterbaum. Man kann dazu den Baum bottom-up traversieren, die Anzahl der Knoten mitzählen und ab einer gewissen Grenze einen Unterbaum als Komponente betrachten und abtrennen.

Es ist allerdings nicht gewährleistet, daß die auf diese Weise postulierten Grenzen zwischen den Komponenten auch den tatsächlichen logischen Grenzen im Sinne der Funktionalität des Systems entsprechen.

Unähnlichkeitsbasierte Auswahl. Eine verbesserte Heuristik beruht auf der Annahme, daß eine Unterkomponente *U*, die funktional einen anderen Bereich abdeckt als die übergeordnete Komponente *K*, sich von dieser wahrscheinlich im Datenzugriffsverhalten unterscheidet. Da *U* einen Dienst versieht, der nur im Kontext der übergeordneten Komponente *K* benötigt wird, ist es wahrscheinlich, daß *U* ebenfalls auf die Daten zugreift, auf die *K* - *U* (also die übergeordnete Komponente *K* ohne die Unterkomponente *U*) zugreift. Wenn aber *U* wirklich einen anderen Funktionalitätsbereich abdeckt, dann benötigt *U* wahrscheinlich auch noch zusätzliche Daten.

Die unähnlichkeitsbasierte Auswahl legt die Grenzen zwischen den Komponenten so, daß die Unähnlichkeit zwischen einer ausgewählten Komponente *U* und ihrer übergeordneten Komponente *K* maximiert wird. Die Unähnlichkeit von *U* zu *K* wird dabei folgendermaßen definiert:

$$dissim(U,K) = \frac{|C(U) - C(K-U)|}{|C(K)|}$$

C(*X*) ist dabei für eine Unterprogrammenge *X* die Menge der von *X* benutzten Variablen (der *Kontext* von *X*). Die Unähnlichkeit *dissim*(*U*, *K*) mißt also das Verhältnis der von *U* zusätzlich benutzten Variablen zur Gesamtmenge der von *K* benutzten Variablen.

Die Lösung dieser Maximierungsaufgabe kann effizient durch einen inkrementellen Algorithmus erfolgen, der die Kontexte der Komponenten speichert und Änderungen durch die hierarchische Struktur propagiert.

3.3.5. Verfeinerung der Komponenten

Die mit Hilfe der oben beschriebenen Algorithmen gefundenen FCCs können als Ausgangsbasis für verschiedene weitere Vorgehensweisen dienen: So kann zum Beispiel jedes der restlichen Unterprogramme des Systems der jeweils passendsten Komponente zugeordnet werden, um den Anteil der erkannten Komponenten am gesamten System zu erhöhen. Alternativ können die Komponenten selbst als Ausgangsbasis für bekannte Clustering-Algorithmen dienen, um deren Ergebnisse zu verbessen.

FCC-Kontextsicht. Eine weitere Möglichkeit besteht darin, gemeinsam benutzte Unterprogramme (*shared routines*) zu identifizieren und diejenigen Unterprogramme zu gruppieren, die die gleichen FCCs unterstützen. Diese Sicht wird *FCC-Kontextsicht genannt*. Zur Erstellung dieser Sicht wird für jedes Unterprogramm, das von mehr als

einer Komponente benutzt wird, festgestellt, von welcher Menge von Komponenten es benutzt wird. Die Unterprogramme, die von genau derselben Menge von Komponenten benutzt werden, werden zusammengefaßt und stellen einen *shared cluster* dar. Derselbe Vorgang kann mit den Variablen des Systems durchgeführt werden. Die Erstellung der FCC-Kontextsicht ist schematisch in Abbildung 12 dargestellt.

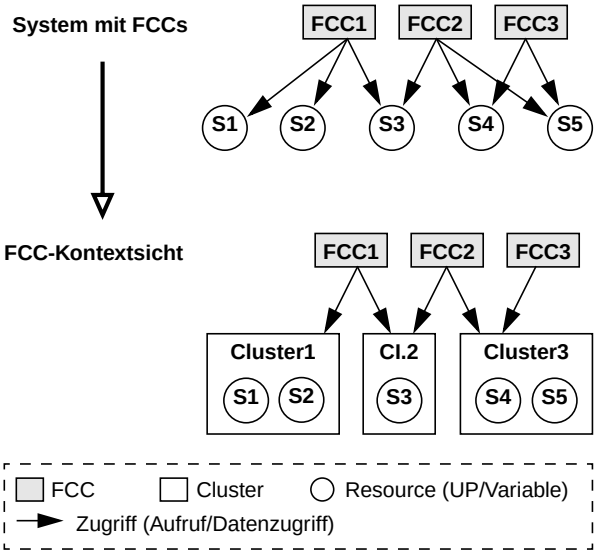

Abbildung 12: FCC-Kontextsicht

Die auf diese Weise gewonnenen Komponenten stellen einen ersten Schritt hin zu einer vollständigen Beschreibung der Ist-Architektur des Systems dar.

3.4. Erkennung von Konnektoren und ihrer Protokolle

Wer an Konnektoren denkt, hat oft Beziehungen zwischen Tasks oder ganzen Programmen, die evtl. in einem Stapel-Betrieb laufen, im Sinn [8]. Wir wollen im Bauhaus Konnektoren auf verschiedenen Abstraktionsebenen finden.

Wir sind der Meinung, daß Konzepte wie Pipes und RPCs zwischen ganzen Tasks zu grobgranular sind, um ein Programm in seiner Feinstruktur zu verstehen. Man könnte diese Art der Konnektoren als Interprogramm- oder Interprozeß-Konnektoren klassifizieren. Die Beziehungen der Feinstrukturen innerhalb eines Programmes sind mindestens genauso wichtig: Konnektoren auf einer Intraprogramm-Ebene drücken die Verbindungen der ACs untereinander aus.

Das schließt natürlich nicht aus, daß es auch innerhalb einer AC Konnektoren zwischen den Teilen einer AC gibt.

Bislang wurden die ACs einer Architektur identifiziert, nun wollen wir uns im Bauhaus auch den Beziehungen dieser ACs untereinander zuwenden. Die Ergebnisse der Datenflußanalysen ermöglichen eine Aussage über Existenz und Art der Konnektoren zwischen den einzelnen Komponenten eines Systems. Datenflußanalysen sind hierfür essentiell, da sie es ermöglichen, nicht-triviale Zusammenhänge zwischen Teilen eines Systems zu analysieren, z.B. ob eine Komponente in eine Variable schreibt, aus der eine andere liest.

Aus den Kontroll- und Datenflußanalysen lassen sich Protokolle, also Verwendungsmuster, für Komponenten und Konnektoren ableiten. Ein Vergleich durch interaktive Ableitung spezifizierter Protokolle mit den tatsächlich realisierten liefert Hinweise auf korrekte oder fehlerhafte Verwendung einer Komponente sowohl vor als auch nach Änderungen, z.B. ob an einer Stelle eine Initialisierung vergessen wurde.

Der Wartungsingenieur muß in die Konnektor- und Protokollerkennung interaktiv eingreifen und die Vorschläge der Analysen sichten. So entstehen zu den Komponenten die Konnektoren der Ist-Architektur und ihre Protokolle.

3.5. Ableitung einer Architekturbeschreibung

3.5.1. Ableiten einer Ist-Architektur

Der Wartungsingenieur kann die verschiedenen Techniken zur AC-Erkennung auf das Programmsystem, dessen Architektur er analysieren möchte, anwenden. Als Ergebnis werden ihm Kandidaten für ACs geliefert. Er kann die Ergebnisse der Analysen durchgehen und einzelne Kandidaten modifizieren, akzeptieren oder verwerfen.

Sind hinreichend viele ACs erkannt worden, so kann mit der Subsystemerkennung begonnen werden. Dabei werden die gefundenen ACs zu immer größeren Einheiten zusammengefaßt, die jeweils einen umfassenderen Zweck erfüllen, als jede einzelne AC für sich gesehen.

Die Konnektoren und Protokolle des Systems lassen sich – zumindest auf der Ebene der ACs und Subsysteme – nun ebenfalls erkennen. Dabei werden Techniken angewendet werden, die zur Zeit noch Gegenstand unserer Forschung sind.

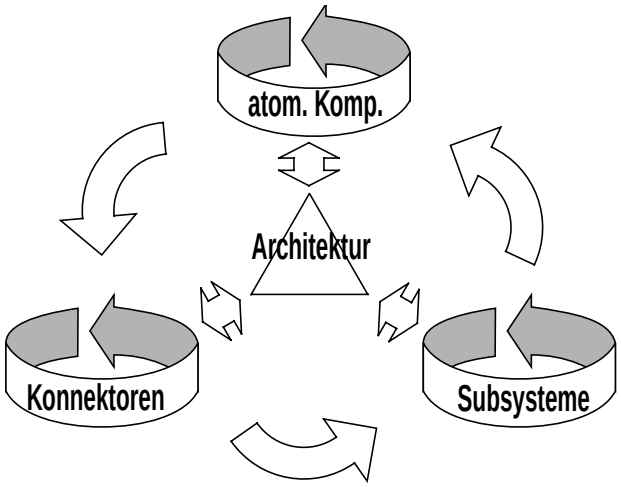

Abbildung 13: Übergeordneter Analysezyklus

Natürlich lassen sich alle drei Tätigkeiten, AC-Erkennung, Subsystem-Erkennung und Konnektor-Erkennung, überlagernd ausführen, womit ein interaktiver und inkrementeller Zyklus auf einer höheren Ebene entsteht, siehe Abbildung 13.

Durch den hier beschriebenen Prozeß kann der Wartungsingenieur eine Ist-Architektur des analysierten Systems ableiten.

3.5.2. Soll-Architektur und inkrementelle Konsistenzprüfungen

Mit Hilfe seines Wissens über anzustrebende Architektureigenschaften, die die antizipierte Evolution des Systems gewährleisten, einerseits und der Betrachtung der Ist-Architektur andererseits wird der Wartungsingenieur eine Soll-Architektur spezifizieren können. Damit kann die eigentlich beabsichtigte gegenüber der tatsächlich vorhandenen Architektur kontrastiert werden.

Bei Änderungen am System, die der Soll- oder Ist-Architektur widersprechen, können später Abweichungen diagnostiziert und gegebenfalls durch den Benutzer toleriert werden. Außerdem bekommt die präventive Wartung durch die Soll-Architektur eine Zielvorgabe.

Das Ergebnis aller Analysen wird in eine geeignete Darstellung für Architekturen überführt [5] und dauerhaft gespeichert. Auf diese Weise bildet sich ein graduell entstehendes "Corporate Memory" für die Wartung. Jede Änderung am Quellcode eines Systems muß dann in dieser Datenbasis nachgeführt werden. Wegen der Arbeit, die der Wartungsingenieur in die interaktiven Analyseschritte bereits investiert hat, muß dabei in jedem Falle vermieden werden, daß Änderungen am Quellcode eine vollständige von Null an beginnende Neuanalyse notwendig machen. Daher suchen wir nach einem Ansatz, der es erlaubt, die Inkremente der Änderungen in das bestehende Analyseergebnis zu integrieren.

Darüberhinaus soll es möglich sein, bei Verletzungen der Architektur während der Wartungstätigkeit (z.B. beim Durchbrechen einer Abstraktionsbarriere) Warnhinweise zu geben und die Stellen zu identifizieren, an denen die Verletzung stattfindet. Dann ist es natürlich auch möglich, Stellen im System zu finden, die momentan gegen die Soll-Architektur verstoßen um so Ansatzpunkte für die präventive Wartung zu erhalten.

4. Unsere Forschung

Unsere Evaluation publizierter Techniken zur Erkennung atomarer Komponenten hat ergeben, daß keine vollständig automatische Technik zur AC-Erkennung und auch keine nicht triviale Kombination verschiedener Techniken ausreicht, um die menschliche Erkennungsrate zu erreichen. Die Quote der gefundenen ACs liegt bei maximal 60% der nach Ansicht von Software-Ingenieuren tatsächlich vorhandenen ACs, meistens jedoch weit darunter. Zudem kann die Zahl der *False Positives* sehr hoch sein [3]. Dies rechtfertigt unseren interaktiven Ansatz.

Zur Evaluierung des interaktiven Ansatzes zur AC-Erkennung wurden verschiedene Experimente mit Studenten durchgeführt. Die statistische Auswertung der Experimente zeigt, daß sich tatsächlich ein positiver Effekt einstellt.

Die Generierung der SSA-Form zur Darstellung von Datenflußinformation auf der Basis pessimistischer Annahmen ist nahezu abgeschlossen, so daß erste Ansätze zur Konnektor- und Protokollerkennung entwickelt und erprobt werden können. Dabei werden zunächst vereinfachte Protokolle als reguläre Ausdrücke über die Aufrufe von Unterprogrammen aus den Schnittstellen von ACs aus dem Aufrufgraphen extrahiert.

Die genauen Modalitäten der inkrementellen Analyse von Architekturen nach Änderung des Quelltextes sind Gegenstand angehender Forschung. Die Forschung zu Analysen der Diskrepanz der Ist- zur Soll-Architektur und eventuell die Bewertung einer Architektur an sich wird mittelfristig begonnen.

4.1. Industrieller Einsatz

Die in 3.3 beschriebenen Vorgehensweisen zur Erkennung von Komponenten wurden in einem Industrieprojekt benutzt, um die Migration eines FORTRAN-Systems nach C++ unter Wiederverwendung mathematisch komplexer FORTRAN-Komponenten zu unterstützen [1].

Dafür wurden wiederverwendbare Komponenten identifiziert, ein objektorientierter Systementwurf erstellt und Wrapping-Mechanismen entwickelt, die es erlauben, die wiederverwendeten FORTRAN-Komponenten auf transparente Weise in das C++-System zu integrieren. Mit Hilfe von Wartungs- und Migrationsrichtlinien können die FORTRAN-Komponenten mit der Zeit ebenfalls nach C++ migriert werden.

In einem in kurzer Zeit anlaufenden Projekt sollen die Erfahrungen bei der Erkennung von Komponenten auch auf objektorientierte Systeme (C++) ausgedehnt werden.

Teile der in 2., 3.1 und 3.2 beschriebenen Bauhaus-Werkzeuge sind im Einsatz in der Industrie; es besteht Interesse an weiteren Kooperationen.

5. Referenzen

[1] J. Bayer, J.-F. Girard, M. Würthner, J.-M. DeBaud and M. Apel. Transitioning Legacy Assets to a Product Line Architecture. To be published at ESEC/FSE ‘99.

[2] J.-F. Girard, R. Koschke, M. Würthner. An Intermediate Representation for Reverse Engineering Analyses. Proceedings of the Working Conference on Reverse Engineering - WCRE’98, 1998

[3] J.-F. Girard, R. Koschke, G. Schied. A Metric-based Approach to Detect Abstract Data Types and State Encapsulations. Journal of Automated Software Engineering, to appear.

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

[5] N. Medvidovic. A Classification and Comparison Framework for Software Architecture Description Languages. Technical Report UCI-ICS-97-02, Dept. of Information and Computer Science, University of California, Irvine, 1996.

[6] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems, 13(4):451-490, Oct. 1991.

[7] R. Koschke. Atomic Architectural Component Recovery for Program Understanding and Evolution. Ph.D. thesis. University of Stuttgart. To be published.

[8] D. Garlan, M. Shaw. An Introduction to Software Architecture. Advances in Software Engineering and Knowledge Engineering, Volume 1, World Scientific Publishing Company, New Jersey, 1993

[9] J.-F. Girard and R. Koschke. Finding components in a hierarchy of modules: a step towards architectural understanding. In International Conference on Software Maintenance, pages 66–75, Bari, September 1997.