

Studiengang: Softwaretechnik

Prüfer: Prof. Dr. Erhard Plödereder
Betreuer: Dr. Rainer Koschke,
Dipl.-Inf. Stefan Bellon

begonnen am: 8. August 2003
beendet am: 8. Februar 2004

CR-Klassifikation: D.2.4, D.2.7, D.3.3, F.3.1, F.3.2

Diplomarbeit Nr. 2135

Werkzeuggestützte Herleitung von Protokollen

Dietrich Haak

Institut für Softwaretechnologie
Universität Stuttgart
Universitätsstr. 38
70569 Stuttgart

Zusammenfassung

Im Rahmen von Wartungstätigkeiten stehen Software-Entwickler oftmals vor der Aufgabe, Architekturinformationen aus Quellcode wiederzugewinnen. Eine konkrete Problemstellung ist dabei die Herleitung von Protokollen, d.h. von sequenziellen Abhängigkeiten der primitiven Operationen einer Komponente. Protokolle können dazu verwendet werden, neuen oder veränderten Code zu validieren. Diese Arbeit beschreibt zwei mögliche Ansätze, um Protokolle aus Spuren (das sind diejenigen Teile im Quellcode, die die untersuchte Komponente benutzen) herzuleiten, wobei die Spuren in Graphform durch ein Werkzeug zur Spurextraktion gegeben sind. Beide Ansätze werden einander gegenübergestellt und einer davon wird umgesetzt. Dabei wird zunächst ein Verfahren zur Ersetzung von inter-prozeduralen Zyklen, die in Spurgraphen auftreten, vorgestellt. Zuguterletzt werden Techniken zur Validierung von Protokollen gegen Spuren und umgekehrt beschrieben.

Inhaltsverzeichnis

1	Einleitung	7
1.1	Motivation	7
1.2	Ausschreibung	12
1.3	Gliederung dieses Dokuments	16
2	Grundlegendes	19
2.1	Bauhaus und Reengineering	19
2.2	Komponentenerkennung	20
2.3	Spurextraktion	21
2.4	Extrahierte Spurgraphen – Beschreibung	22
2.4.1	Sequenz-Kanten	22
2.4.2	Anfang und Ende	23
2.4.3	Direkte Zugriffe	24
2.4.4	Routinen	24
2.4.5	Routinenaufrufe	24
2.4.6	Bedingungen	30
2.4.7	Relevante Knoten und Kanten	32
2.5	Extrahierte Spurgraphen – Definition	33
2.5.1	Syntax des Spurgraphen	33
2.5.2	Semantik des Spurgraphen	37
2.6	Gravis	42
2.7	Verwandte Forschung	43
2.7.1	Herleitung von Protokollen	43
2.7.2	Validierung mit Protokollen	44
3	Ansätze zur Herleitung von Protokollen	47
3.1	Induktiver Ansatz	47
3.1.1	Übertragen von Heibers Ergebnissen	48
3.1.2	Finden von Vereinfachungsmustern	51
3.1.3	Graphtransformationssysteme	51
3.2	Hypothesengetriebener Ansatz	57
3.2.1	Korrekte und falsche Spuren	58

3.2.2	Überdeckung	61
3.2.3	Obligatorische und verbotene Vorgänger und Nachfolger	62
3.2.4	Prozess	67
3.3	Auswahl des Ansatzes	70
3.3.1	Größe und Lesbarkeit des Protokollgraphen	70
3.3.2	Unterstützung des Wartungs-Ingenieurs	71
3.3.3	Benötigtes Vorwissen	71
3.3.4	Lernprozess	72
3.3.5	Weitere Anwendungsbereiche	73
3.3.6	Risiken in der Durchführung	73
3.3.7	Auswahl	74
4	Spurbereinigung	75
4.1	Anfang und Ende	75
4.2	Routinen und Routinenaufrufe	75
4.2.1	Fall 1	76
4.2.2	Fall 2	77
4.2.3	Fall 3	78
4.3	Bedingungen	78
4.4	Bereinigte Spurgraphen – Definition	79
4.4.1	Syntax	79
4.4.2	Semantik	80
5	Rekursionsauflösung	81
5.1	Motivation	81
5.2	Änderung der Semantik	82
5.3	Ersetzungsmuster	86
5.4	Algorithmus zur Rekursionsauflösung	86
5.4.1	Überblick	87
5.4.2	Code-Auszug und Beschreibung	89
5.4.3	Veranschaulichung	93
5.4.4	Vervollständigung und Optimierungen	96
6	Validierung	99
6.1	Protokollgraphen – Definition	99
6.2	Umwandlung in endlichen Automaten	100
6.3	Umwandlung in deterministischen Automaten	101
6.4	Validierung gegen korrekte Spuren	103
6.5	Validierung gegen falsche Spuren	105

6.6	Zweigüberdeckung	105
6.7	Obligatorische und verbotene Vorgänger und Nachfolger .	108
6.7.1	Dominanz und Postdominanz	108
6.7.2	Verbotene Vorgänger und Nachfolger	112
7	Schlüsse und Schluss	113
7.1	Ergebnisse dieser Arbeit	113
7.2	Offene Punkte	114
7.3	Rückblick und Erfahrungen	115
7.4	... und Schluss.	115
A	Glossar	117
B	Kurzeinführung in IDEF0	121
	Literaturverzeichnis	123

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Der Begriff des Software-Engineering existiert seit 1967[Bau93]. Software-Engineering ist daher ein sehr junges Forschungsgebiet, das noch viel unerschlossenes Neuland bietet. Es lässt sich in zwei große Bereiche unterteilen: Forward-Engineering und (Software-)Reengineering, wobei ersteres untersucht, wie man neue Software entwickelt, letzteres, wie man vorhandene Software wartet. Diese Diplomarbeit versucht, ein kleines Stück Land aus dem zuletztgenannten Gebiet zu erschließen, dem Reengineering.

Eine typische Situation für Entwickler ist die folgende: Eine Software, die schon seit Jahrzehnten im Einsatz ist, soll so erweitert werden, dass sie von nun an via Internet benutzt werden kann. Um die notwendigen Änderungen und Erweiterungen vornehmen zu können, muss der Entwickler zunächst die Software, genauer: den Quellcode, verstehen. Dabei gibt es jedoch zwei Dinge zu beachten: Zum Einen ist jede nicht-triviale Software ein komplexes System. Zum Anderen hat das menschliche Arbeitsgedächtnis nur begrenzte kognitive Ressourcen, weswegen Menschen mit komplexen Systemen meist überfordert sind. Es erscheinen zwei mögliche Auswege aus der Situation: Entweder konsultiert der Entwickler ein Mitglied des ursprünglichen Entwickler-Teams oder er zieht die Dokumentation zu Rate. Allerdings ist in den meisten Fällen weder das eine noch das andere verfügbar.

Dem Wartungs-Ingenieur – nachdem der Entwickler ja eine Wartungstätigkeit auszuführen hat und ähnliche Rahmenbedingungen herrschen bei wie Ingenieuren aus anderen Gebieten, ist dies die korrekte Bezeichnung – bleibt nur die Möglichkeit, den Quellcode manuell zu analysieren und zu verstehen.

Eine mögliche Vorgehensweise ist dabei die folgende:

1. Identifizieren der Stellen, an denen der Wartungs-Ingenieur neuen Code hinzufügen muss, um die neuen Anforderungen zu erfüllen.

1 Einleitung

2. Analysieren, ob er den vorhandenen Code zunächst verändern muss, bevor er neuen Code hinzufügen kann, z.B. um vom chaotisch gewachsenen Zustand wieder in einen geordneten zu gelangen (Restrukturierung).
3. Bedingungen und Abhängigkeiten herausfinden, gegen die bei den Änderungen nicht verstoßen werden darf.
4. Restrukturierung vornehmen und Erweiterungen implementieren.
5. Prüfen, ob die neu entstandene Version oder Variante des Systems die neuen Anforderungen erfüllt und gegen keine der aufgestellten Bedingungen und Abhängigkeiten verstößt.

Bei einem Top-Down-Ansatz wird der Wartungs-Ingenieur zunächst die Architektur der Software herausarbeiten. Dazu werden *Komponenten*¹ identifiziert und deren Zusammenspiel untersucht. Wichtig sind dabei insbesondere die Schnittstellen.

Schnittstellen beschreiben zunächst auf syntaktischer Ebene im Quellcode, wie eine Komponente benutzt werden kann. Dabei werden exportierte Typen, globale Variablen und primitive Operationen dieser Komponente definiert. Primitive Operationen sind Routinen, die zur Komponente gehören und Zugriffe auf Teile der exportierten Typen oder globalen Variablen. Es gibt jedoch über die syntaktisch definierbaren Aspekte einer Schnittstelle hinaus noch weitere Aspekte, die in den meisten Programmiersprachen nicht explizit beschrieben werden können. Einer dieser Aspekte sind die sequenziellen Abhängigkeiten der primitiven Operationen. Diese nennt man *Protokoll*.

Protokolle

Betrachten wir kurz ein Beispiel eines Protokolls. In Abbildung 1.1 auf der nächsten Seite ist die Definition eines Abstrakten Datentyps (ADT) zu sehen, der ein dynamisches Arrays repräsentiert.

Das Protokoll dieser Komponente lässt sich wie folgt beschreiben: „Vor allen anderen Operationen muss *create()* aufgerufen werden. Anschließend darf an beliebiger Stelle *destroy()* aufgerufen werden. Wird zwischen den beiden Aufrufen eine andere primitive Operation ausgeführt, dann muss die erste *put()* sein. Danach darf beliebig oft und in beliebiger Reihen-

¹Im Sinne dieser Arbeit sind das ADTs und ADOs, siehe auch Glossar auf Seite 117

```

typedef struct {...} DynArray;

DynArray *create();
void      destroy(DynArray *array);

void     put(DynArray *array, int index, ElemType *element);
ElemType *get(DynArray *array, int index);

```

Abbildung 1.1: C-Code-Beispiel – Dynamisches Array

folge *get()* und *put()* aufgerufen werden. Nach *destroy()* darf nichts mehr aufgerufen werden.“

Beachten Sie, dass dabei lediglich sequenzielle Abhängigkeiten der Routinen untersucht werden, davon ausgehend, dass alle auf dem gleichen Objekt arbeiten. Es wird z.B. nicht berücksichtigt, auf welchen Index bei *get()*- und *put()*-Aufrufen zugegriffen wird.

Dieses Protokoll kann zur besseren Übersicht als Diagramm dargestellt werden (siehe Abbildung 1.2). Dabei geben die Knoten die primitiven Operationen an, die Kanten die sequenziellen Abhängigkeiten.

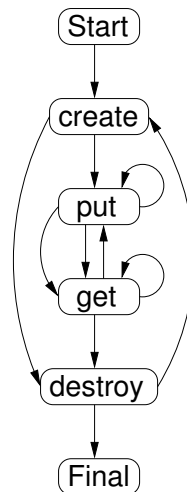


Abbildung 1.2: Protokoll eines dynamischen Arrays

Anwendungsgebiete

Bringen wir nun Protokolle in Zusammenhang mit der Aufgabe des Wartungs-Ingenieurs. Versucht er, das Protokoll einer Komponente herzuleiten, dann hat er dazu zwei Möglichkeiten. Zum Einen kann er sich den Quellcode der Komponenten ansehen und versuchen, über die inneren Zusammenhänge Schlüsse auf die sequenziellen Abhängigkeiten der primitiven Operationen zu ziehen. Dieser Ansatz wird Glass-Box-Understanding genannt [Kos02]. Im Gegensatz dazu steht das Black-Box-Understanding, bei dem der Wartungsingenieur Quellcode betrachtet, der die betroffene Komponente benutzt. Solchen Code nennen wir von nun an *Benutzercode*.

Hat der Wartungs-Ingenieur aus diesen Analysen ein Protokoll hergeleitet, dann kann er nun entscheiden, ob die Komponente restrukturiert werden muss oder ob sie so belassen werden kann. Will er die Komponente ändern, dann hilft ihm das Protokoll dabei. Zunächst ändert er das Protokoll, so dass es den Zielzustand beschreibt. Dann gleicht er den Benutzercode gegen das Protokoll ab und findet damit heraus, welche Teile des Benutzercodes nach einer Neustrukturierung der Komponente verändert werden müssen.

Auch beim Hinzufügen von neuem Code ist das Protokoll nützlich. Benutzt der neue Code nämlich die untersuchte Komponente, dann kann er wiederum gegen das Protokoll auf Korrektheit validiert werden.

Über das hier beschriebene Beispiel hinaus gibt es weitere Verwendungsmöglichkeiten für Protokolle. So kann z.B. untersucht werden, wie gut Testcode einer Komponente deren Protokoll überdeckt. Bleiben Lücken in den überdeckten Zweigen des Protokolls, dann ist das ein sicherer Hinweis darauf, dass weitere Testfälle hinzugefügt werden müssen. Des Weiteren kann vorhandener Code auf Korrektheit getestet werden, indem er gegen das Protokoll validiert wird.

Unterstützung durch Werkzeuge

Wie eingangs schon erwähnt steht der Wartungs-Ingenieur bei den skizzierten Aufgaben einer Fülle komplexer Informationen gegenüber. Um Fehler zu vermeiden und die Effizienz zu erhöhen, erscheint es sinnvoll, Prozesse und Werkzeuge zu definieren und zu erstellen, die den Wartungs-Ingenieur bei seiner Arbeit unterstützen und entlasten.

Die vorliegende Diplomarbeit beschäftigt sich mit der Entwicklung von Werkzeugen und Prozessen für die Herleitung von Protokollen aus Benutzercode. Es existieren schon Werkzeuge zur Identifizierung von Komponenten und zur Extraktion von *Spuren*. Spuren sind genau diejenigen Teile vom Benutzercode, die auf der Komponente arbeiten, Spurgraphen sind eine graphische Repräsentation, ähnlich dem oben vorgestellten Beispiel eines Protokollgraphen.

Einschränkungen

Der hier beschriebene Ansatz ist verschiedenen Einschränkungen unterworfen. Zunächst werden, wie schon im Beispiel erwähnt, lediglich sequenzielle Abhängigkeiten der primitiven Operationen selbst berücksichtigt, der Datenfluss wird dabei außer Acht gelassen. Darüber hinaus beschränkt sich die Mächtigkeit von Protokollen auf die Menge der regulären Sprachen. Der Grund hierfür liegt in der automatischen Validierung von Spuren gegen Protokolle. Da sowohl Spuren als auch Protokolle Sprachen spezifizieren, geschieht bei der Validierung nichts anderes, als dass eine Sprache darauf untersucht wird, ob sie Teilmenge einer anderen Sprache ist. Dieses Problem ist jedoch nur für reguläre Sprachen entscheidbar.

Vorarbeiten

Diese Arbeit baut an vielen Stellen auf der Diplomarbeit von Timo Heiber [Hei00] auf, die sich mit der Herleitung von Protokollen beschäftigt. Darin sind im Wesentlichen vier Schritte beschrieben:

1. Zunächst wird aus den extrahierten Spuren ein Protokollgraph erstellt, indem die Spuren gemeinsame Start- und Endknoten erhalten.
2. Dann werden bekannte Verfahren aus der Automatentheorie verwendet, um den erstellten Protokollgraph zu vereinfachen. Ziel ist dabei zum Einen, den Graphen zu verkleinern, zum Anderen, unnötige Restriktionen aus dem Graph zu entfernen.
3. In einem weiteren Schritt werden Vereinfachungsmuster angewendet, deren Vorbedingungen der Wartungs-Ingenieur bestätigen muss, weil sie die Semantik des Protokollgraphen verändern und dies nur unter bestimmten, nicht automatisch prüfbareren Bedingungen erlaubt ist. Da diese Muster nicht nur für eine spezielle Komponente gelten, werden sie von nun an *komponentenunabhängige Muster* genannt.

1 Einleitung

4. Zuguterletzt spezifiziert der Wartungs-Ingenieur Vereinfachungsmuster, die nur speziell für die untersuchte Komponente gelten. Diese Muster nennen wir ab jetzt *komponentenabhängige Muster*.

Die vorliegende Arbeit sucht nach Möglichkeiten der Herleitung von Protokollen, die die Arbeit von Heiber erweitern und komplementieren.

Im nächsten Abschnitt ist die Ausschreibung dieser Arbeit wiedergegeben, woran sich eine Übersicht über die Gliederung der Arbeit anschließt.

1.2 Ausschreibung

Hintergrund

Komponenten (Klassen, Module und Subsysteme) verfügen über eine Schnittstelle, die ihre zulässige Verwendungsweise definiert. Der Begriff Schnittstelle wird oft in einem sehr engen Sinne aufgefasst: als die Menge exportierter (bzw. auch importierter) Routinen, Typen und globaler Variablen (bei Modulen) bzw. Methoden und Attribute (bei Klassen). Diese Art von Schnittstelle bezeichnen wir als die syntaktische Schnittstelle. Sie ist mit einfachen Mitteln zu erkennen. Gemäß [Par72] ist der Begriff Schnittstelle jedoch viel weiter zu fassen: als die Menge aller zulässigen Annahmen über eine Komponente. Hierzu zählen alle Vor- und Nachbedingungen von Routinen der Komponente, alle Konsistenzbedingungen der Typen und Variablen bzw. Attribute sowie auch Annahmen über das Laufzeitverhalten und den Speicherbedarf.

Eine spezielle Form der Annahmen über Komponenten sind sequentielle Beschränkungen der Ausführungsreihenfolge ihrer Operationen (d.h. Aufrufe der Routinen und Zugriffe auf Variablen und Attribute ihrer Schnittstelle). Beispielsweise muss vor Verwendung eines abstrakten Datentyps Stack zuerst die Initialisierungsroutine und zuletzt die Freigaberoutine aufgerufen werden. Außerdem darf zu keinem Zeitpunkt die Anzahl der Pop-Operationen die der Push-Operationen überschreiten.

Solche Sequenzbeschränkungen können mit herkömmlichen Programmiersprachen nicht formuliert werden und werden leider, selbst wenn diese Möglichkeit besteht, nur in seltenen Fällen von Programmierern tatsächlich ausreichend dokumentiert. Die Verletzung solcher Sequenzbeschränkungen hat in der Regel ein Fehlverhalten während der Laufzeit zur Folge.

Unit-Tests können diese Fehler zum Teil aufdecken, geben allerdings keine Garantie, dass wirklich alle Fehler gefunden werden. Außerdem liefern Unit-Tests keine Erklärung, wie es zum Fehler kommt. Durch statische Prüfung können solche Fehler vermieden und erläutert werden. Solche statische Prüfungen sind leider nicht vollständig automatisierbar, da viele daran geknüpfte Fragen unentscheidbar sind. Sie müssen somit zu einem großen Teil vom Menschen übernommen werden. Der Aufwand statischer Prüfungen durch den menschlichen Analytiker ist jedoch ohne Werkzeugunterstützung sehr hoch. Außerdem unterlaufen menschlichen Analytikern Fehler, insbesondere wenn Information mühsam von Hand gesammelt werden muss.

Olender und Osterweil [Ole92] haben für die statische Prüfung von Sequenzbeschränkungen ein Verfahren entwickelt, das Sequenzbeschränkungen prüft, wenn diese in Form einer regulären Sprache spezifiziert werden können (denn dann ist diese Frage entscheidbar). Verwendet wird hierzu ein Datenflussrahmenwerk. Butkevich und Kollegen [But00] schlagen ebenfalls reguläre Sprachen zur Spezifikation von Sequenzbeschränkungen von Klassen und Schnittstellen vor. Diese Spezifikationen werden im Falle von Vererbungen auf Konformität geprüft. Somit ist zwar eine Konformität zwischen Spezifikationen sichergestellt, nicht jedoch die Konformität des Quelltextes bezüglich der Spezifikation. Model-Checking-Methoden können für die Prüfung von Sequenzbeschränkungen ebenfalls herangezogen werden; allerdings bedarf es hierfür zunächst der Extraktion des Modells aus dem Quelltext [Cor00].

Sowohl Olender und Osterweil als auch Butkevich und Kollegen setzen die Existenz einer Spezifikation voraus, die für viele reale Systeme nicht vorliegt. Cook und Wolf [Coo98] schlagen deshalb vor, Modelle des Verhaltens durch eine Markov-Analyse der tatsächlichen Ausführung herzuleiten. Diese Informationen können Rückschlüsse auf Sequenzbeschränkungen zulassen. Wie beim Testen besteht auch hier das für nichttriviale Systeme unlösbare Problem, eine vollständige Abdeckung aller möglichen Eingaben zu erreichen, um jedes prinzipiell mögliche Verhalten zu erkennen.

Eigene Vorarbeiten

Unser prinzipielles Vorgehen zur statischen Herleitung von Protokollen wurde bereits ausgearbeitet [Kos02], [Hei00]. Das semi-automatische Verfahren zur Herleitung von Sequenzbeschränkungen stützt sich auf folgende verfügbare Informationsquellen:

1 Einleitung

- die tatsächliche Verwendungsweise des Moduls oder der Klasse in Form statischer Spuren aller Instanzen des Moduls bzw. der Klasse; die statische Spur einer Instanz (eines Objekts) enthält alle und nur jene Anweisungen, die potentielle Operationen auf diesem Objekt darstellen [Eis02]; statische Objektspuren werden durch statische Kontroll- und Datenflussanalysen ermittelt [Eis02]
- statische Informationen über die inneren Abhängigkeiten des Moduls oder der Klasse (Kontroll- und Datenabhängigkeiten)
- Domänenwissen unter Einbeziehung des menschlichen Experten und möglicherweise verfügbarer Dokumentation

Unser Prozess der Protokollherleitung lässt sich wie folgt beschreiben: Zunächst werden die statischen Spuren aller Instanzen eines Moduls oder einer Klasse extrahiert. Die gefundenen statischen Spuren stellen beispielhafte Verwendungen dar und bilden somit das Rohmaterial für den Herleitungsprozess. Diese statischen Spuren können dann durch den Benutzer validiert werden. Die validierten statischen Spuren können vollautomatisch vereinheitlicht werden, soweit dies durch bekannte Techniken aus der Automatentheorie möglich ist; z.B. können durch bekannte Verfahren zur Umwandlung von nichtdeterministischen endlichen Automaten in deterministische Automaten gemeinsame Präfixe - und in umgekehrter Form auch gemeinsame Suffixe - gefunden werden [Hop79].

Mittels Graphmustern kann der Benutzer darüber hinaus häufig wiederkehrende Verwendungsmuster aufspüren, die sich einer konservativen vollautomatischen Vereinheitlichung entziehen. Ein Beispiel hierfür sind Operationen, die teils durch Prädikate bedingt und teils unbedingt ausgeführt werden. Dieses Muster liefert einen Hinweis, dass das Prädikat optional sein könnte. Dank der Unterstützung von Program-Slicing kann der Benutzer seine Hypothesen über Sequenzbeschränkungen rasch verifizieren. Der Benutzer wird durch unsere Datenflussanalysen dabei auf Seiteneffekte der Operationen aufmerksam gemacht. Hat eine Operation keine Seiteneffekte und geht von ihr auch keine Kontrollabhängigkeit für andere Operationen des Moduls bzw. der Klasse aus, dann ist diese Operation möglicherweise optional in einer Sequenz.

Der Benutzer selbst kann zu jeder Zeit eigene Sequenzbeschränkungen bzw. -lockerungen hinzufügen. Auf diese Weise können möglicherweise wieder neue vereinfachende automatische Transformationen angewandt werden. Der Herleitungsprozess ist somit iterativ und inkrementell.

Ein erster rudimentärer Prototyp zur Unterstützung dieses Prozesses wurde im Rahmen einer Diplomarbeit für C-Komponenten implementiert. Der Prototyp dient als Konzeptbeweis, genügt jedoch nicht für echte Fallstudien. Realistische Fallstudien müssen durchgeführt werden, um die Tragfähigkeit des Verfahrens zu ermitteln. Der Prototyp berücksichtigt außerdem auch keine inneren Kontroll- und Datenabhängigkeiten der Komponente selbst und bietet dem Benutzer nicht die Möglichkeit, eigene Graphmuster bzw. Graphtransformationen anzugeben.

Aufgabenstellung

Die Aufgabe dieser Diplomarbeit ist es, den vorgeschlagenen Prozess durchgängig durch Werkzeuge zu unterstützen. Hierzu kann eine Reihe existierender Bausteine übernommen werden (eventuell mit Anpassungen):

- Extraktion statischer Spuren, die bereits die extrahierten Spuren in einem Resource-Flow-Graph (RFG) ablegt
- Visualisierung der Objektspuren im Grapheneditor Gravis für RFGs
- vollautomatische Vereinheitlichung der Objektspuren mit Hilfe von Techniken aus der Automatentheorie
- rudimentäre Implementierung zur Suche nach Graphmustern in RFGs

Im Einzelnen umfasst die Diplomarbeit folgende Aufgaben:

- Anpassung der vollautomatischen Vereinheitlichung an die Änderungen, die sich für die Modellierung von Objektspuren seit der Arbeit von Timo Heiber ergeben haben
- semiautomatische Vereinheitlichung der Objektspuren mit Hilfe von benutzerdefinierten Graphmustern; hierzu muss ein Formalismus zur Spezifikation von Graphmustern für die Zwecke der Vereinheitlichung beschrieben und umgesetzt werden („generische“ Muster); die Graphmuster sollen vom Benutzer in Gravis graphisch spezifiziert werden können; eine entsprechende textuelle Repräsentation der Graphmuster ist ebenso zu spezifizieren und zu unterstützen
- Spezifikation (durch den Benutzer) und Berücksichtigung (durch die Analysen) äquivalenter Operationen: beispielsweise soll für einen Datentyp Stack ausgedrückt werden können, dass sich ein Push und ein unmittelbar darauf folgendes Pop gegenseitig aufheben; diese Äqui-

1 Einleitung

valenzen können mit Hilfe der Graphmuster spezifiziert werden, sie unterscheiden sich jedoch darin, dass hierzu zusätzlich der Graph transformiert wird; außerdem sind diese Muster spezifisch für eine Komponente

- Integration der vollautomatischen und semi-automatischen Vereinheitlichung im Grapheneditor Gravis
- optional, sofern die Zeit reicht: Durchführung einer Fallstudie an einem realen System; als ein Kandidat hierfür kommt das System *concepts* in Frage

Die Graphmuster und Graphtransformationen sollen möglichst allgemein für den RFG implementiert werden, so dass sie auch für andere Zwecke wiederverwendet werden können.

Die Ansprüche an die Qualität der Implementierung sind hoch. Es soll ein tatsächlich verwendbares Werkzeug entwickelt werden. Vollständige Dokumentation und ein umfangreicher systematischer Test sind deshalb Teil der Aufgabe.

Implementierungswerkzeuge

Der Ada-Compiler Gnat; Bauhaus-Werkzeuge.

Voraussetzungen

Kenntnisse in Ada 95.

Literatur

Das Literaturverzeichnis der Ausschreibung wurde ins Literaturverzeichnis dieser Arbeit integriert.

Betreuer

Dr. Rainer Koschke, Stefan Bellon

1.3 Gliederung dieses Dokuments

Kapitel 2 ordnet die Arbeit in den Kontext von Bauhaus und von anderen Forschungsprojekten auf dem Gebiet des Reengineering ein. Dabei

1.3 Gliederung dieses Dokuments

werden insbesondere Werkzeuge aus Bauhaus beschrieben, die eng mit der vorliegenden Arbeit verbunden sind. In Kapitel 3 wird der induktive und der hypothesengetriebene Ansatz untersucht. Die beiden Ansätze werden einander gegenübergestellt und der hypothesengetriebene Ansatz wird als Gegenstand der Untersuchung dieser Arbeit ausgewählt. Kapitel 4 beschreibt, wie die extrahierten Spuren von irrelevanten Informationen bereinigt und auf eine einheitliche Form mit Protokollgraphen gebracht werden. In Kapitel 5 wird ein Verfahren zur Ersetzung von rekursiven Routinenaufrufen in Spurgraphen durch Schleifenkonstrukte vorgestellt, um die Sprache der Spurgraphen aus der Menge der kontext-freien in die Menge der regulären Sprachen zu verschieben. Kapitel 6 beschreibt Techniken, die zur Validierung beim hypothesengetriebenen Ansatz verwendet werden. In Kapitel 7 werden schließlich die Ergebnisse der Arbeit zusammengefasst.

1 Einleitung

2 Grundlegendes

Dieses Kapitel gibt zunächst einen Überblick über Bauhaus und Reengineering, anschließend werden die Bauhaus-Werkzeuge zur Komponentenerkennung und Spurextraktion erläutert. Der Hauptteil des Kapitels ist die Beschreibung und Definition der Spurgraphen, die von der Spurextraktion ausgegeben werden. Danach wird Gravis, der Grapheneditor von Bauhaus, kurz beschrieben, worauf anschließend ein Überblick über Forschungsprojekte folgt, die mit dem Thema dieser Arbeit verwandt sind.

2.1 Bauhaus und Reengineering

Ziel des Bauhaus-Projekts [Bau04] ist es, Prozesse und Werkzeuge zu definieren und zu erstellen, die einem Wartungsingenieur helfen, verschiedene Schritte beim Reengineering von Softwaresystemen durchzuführen.

Der Schwerpunkt liegt dabei auf der Wiedergewinnung von Informationen über die Architektur der Software und der dauerhaften Speicherung dieser Informationen. Damit kann der Wartungsingenieur z.B. Veränderungen am Quellcode vornehmen, um die ursprüngliche Architektur wiederherzustellen, die im Lauf der Zeit durch unkontrollierte Wartung verloren ging. Oder er kann Änderungen, die er am Quellcode vornehmen muss, gegen die vorhandenen Informationen überprüfen, um Fehler zu vermeiden.

In der Regel wird als erster Schritt vorhandener Code in die Zwischendarstellung IML übersetzt. Nun sind verschiedene Informationen vorhanden, u.a. Kontrollfluss- und Datenflussgraphen. Diese Informationen werden in Form eines Resource Flow Graph (RFG) gespeichert, der neben der Speicherung auch zur visuellen Darstellung der gewonnenen Informationen dient. Es ist auch möglich, Quellcode direkt in einen RFG zu übersetzen.

Momentan kann mit der Bauhaus-Suite Quellcode folgender Programmiersprachen untersucht werden:

2 Grundlegendes

- C durchgängig
- C++ mit Einschränkungen: späte Typbindung (Dispatching) wird noch nicht unterstützt, daher können auch keine Spuren extrahiert werden
- Cobol – nur Übersetzung in RFG, nicht in IML
- Java – Übersetzung in veraltete IML-Version, Aktualisierung ist geplant
- Ada95 – Übersetzung in RFG ist geplant

Im Folgenden werden diejenigen Werkzeuge und Erkenntnisse aus dem Bauhaus-Projekt vorgestellt, die für diese Arbeit von Bedeutung sind.

2.2 Komponentenerkennung

Wie eingangs erwähnt, werden in dieser Arbeit sequenzielle Abhängigkeiten von Komponentenoperationen untersucht. Dazu müssen zunächst die Komponenten des betroffenen Softwaresystems identifiziert werden.

[Kos00] beschäftigt sich umfassend mit der Erkennung von Komponenten. Es werden verschiedene Techniken einander gegenübergestellt und ihre Stärken und Schwächen herausgearbeitet. Die beschriebenen Techniken stehen als Bauhaus-Analysewerkzeuge zur Verfügung. Um eine solche Analyse anzustoßen, muss zunächst ein RFG aus der Zwischendarstellung IML erzeugt werden. Die Werkzeuge zur Komponentenerkennung erzeugen logisch zusammengehörende Komponenten, das sind Typen, globale Variablen und primitive Operationen, im RFG-Format.

Dabei ist anzumerken, dass direkte Zugriffe (d.h. Zugriffe, die nicht durch Routinen gekapselt sind) auf globale Variablen einer Komponente oder auf Felder eines Typs, der in einer Komponente definiert ist, nicht zur identifizierten Komponente hinzugefügt werden.

Wie schon in [Hei00] gesagt wird, hängt die Qualität des wiedergewonnenen Protokolls einer Komponente maßgeblich davon ab, dass die Komponente korrekt identifiziert wurde. Der Wartungsingenieur muss also vor der Wiedergewinnung des Protokolls zunächst mit oben erwähnten Bauhaus-Werkzeugen die gewünschte Komponente identifizieren und weitestgehend sicherstellen, dass dies fehlerfrei geschehen ist.

2.3 Spurextraktion

In Abschnitt 1.2 auf Seite 12 wurde beschrieben, dass Komponentenprotokolle aus Spuren hergeleitet werden sollen. Dieser Abschnitt beschreibt die Extraktion von Spuren.

Ziel der Spurextraktion ist es, durch Analyse des Kontrollflusses diejenigen Operationen aus gegebenem Quellcode zu extrahieren, die auf einem statisch nachweisbaren Objekt arbeiten. Als Objekt wird dabei die Instanz einer Komponente bezeichnet, also eines ADTs oder ADOs. Die hier vorliegende Analyse arbeitet rein statisch. In Abgrenzung dazu gibt es dynamische Verfahren (siehe Abschnitt 2.7 auf Seite 43), die Spuren erfassen, welche von Objekten zur Laufzeit erzeugt werden.

[Eis04] gibt Auskunft über den neuesten Stand der Spurextraktion im Bauhaus-Projekt.

Gegenstand der Untersuchung sind sowohl Objekte, die auf dem Stapel, als auch Objekte, die auf der Halde abgelegt werden. Zunächst werden diejenigen Stellen im Kontrollflussgraph identifiziert, an denen Objekte erzeugt werden. Bei Stapelobjekten (d.h. lokalen und globalen Variablen) ist dies die Deklaration, bei Haldenobjekten der Aufruf zur Speicherallokierung. Jede dieser Stellen repräsentiert eine Äquivalenzklasse von Objekten.

Als Nächstes werden alle Pfade des Kontrollflussgraphen durchlaufen, die zu einer Stelle führen, an der Objekte zerstört werden. Bei Stapelobjekten ist dies das Ende des Gültigkeitsbereichs, bei Haldenobjekten der Aufruf zur Freigabe des Speichers.

Entlang dieser Pfade werden alle Operationen und Bedingungen identifiziert, die auf der momentan untersuchten Komponente arbeiten. Operationen sind Routinenaufrufe, Anweisungen, die Felder eines exportierten Typs von einer Komponente abfragen (Lesezugriffe) und Anweisungen, die solchen Feldern Werte zuweisen (Schreibzugriffe). Darüber hinaus werden diejenigen Bedingungen identifiziert, von denen Komponentenoperationen kontrollabhängig [Mor98] sind.

Um Komponentenoperationen zu identifizieren, muss jede Operation darauf untersucht werden, ob sie auf der untersuchten Komponente arbeitet oder nicht. Treten im Quellcode Zeiger auf, dann muss eine Points-To-Analyse vorgenommen werden. Dabei ist zu beachten, dass Points-To-Analysen Objekte nicht beliebig präzise identifizieren können. In Bauhaus

2 Grundlegendes

wird bei der Points-To-Analyse ein konservativerer Ansatz verfolgt, wodurch mindestens alle Objekte identifiziert werden, auf die tatsächlich gezeigt wird. Es können jedoch auch mehr als diese identifiziert werden, wodurch sequenzielle Abhängigkeiten in der extrahierten Spur entstehen können, die es im Quellcode nicht gibt.

Sind alle Operationen und Bedingungen identifiziert, dann werden sie entsprechend dem Kontrollfluss mit Kontrollflusskanten verbunden. Dadurch entsteht der Objekt-Prozess-Graph.

Mit Hilfe von Aufrufpfaden können nun einzelne Objekte ausgemacht werden, die in derselben Äquivalenzklasse liegen, also an derselben Stelle erzeugt werden. Damit ist es möglich, einzelne statisch feststellbare Objekte zu bestimmen, für die Objekt-Prozess-Graphen extrahiert werden.

Eine Spur ist ein Pfad in einem solchen Objekt-Prozess-Graph. Die graphische Repräsentation von Spuren wird im Folgenden *Spurgraph* genannt.

2.4 Extrahierte Spurgraphen – Beschreibung

Ausgabe des Bauhaus-Werkzeugs zur Spurextraktion sind Spurgraphen im RFG-Format. Im nächsten Abschnitt werden extrahierte Spurgraphen exakt definiert. In diesem Abschnitt sollen sie zunächst informell beschrieben werden.

Abbildung 2.1 auf der nächsten Seite gibt einen Überblick über alle Knoten- und Kantentypen, die in extrahierten Spurgraphen vorkommen. Dabei sollen die Kanten an den Knoten andeuten, welche Knotentypen mit welchen Kantentypen auf welche Art verbunden werden dürfen. *Seq.* steht dabei für alle Sequenzkanten. Namen von Kantentypen, die in Klammern stehen, deuten an, dass nicht jeder Knoten dieses Typs eine Kante dieses Typs braucht.

2.4.1 Sequenz-Kanten

Wie zu sehen ist, gibt es drei Arten von Sequenz-Kanten: *Unconditional*, *True* und *False*. Die beiden Letzteren dürfen dabei nur von *Condition*-Knoten ausgehen, die Bedingungen repräsentieren. Eine ausgehende *True*-Kante wird beschriftet, wenn die Bedingungen, die der *Condition*-Knoten repräsentiert, wahr ist. Entsprechend wird eine ausgehende *False*-Kante

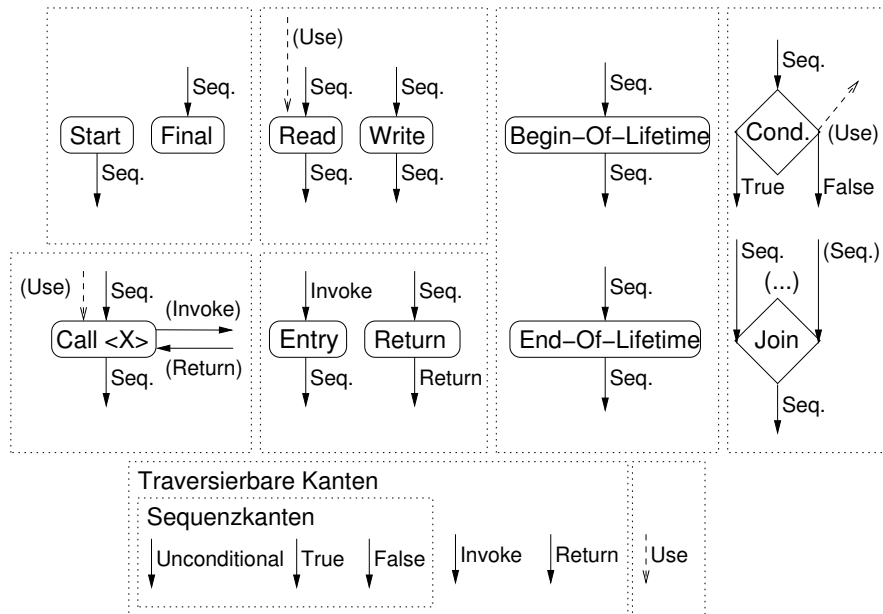


Abbildung 2.1: Kanten- und Knotentypen von Spurgraphen

beschriftet, wenn die Bedingung falsch ist. Im Gegensatz dazu geben *Unconditional*-Kanten (oder exakter: *Unconditional-Sequence*-Kanten) unbedingte Sequenzen an. *Unconditional* ist der Kantentyp, der am häufigsten in Spur- und Protokollgraphen vorkommt. Daher werden von nun an *Unconditional*-Kanten in den Abbildungen aus Gründen der Übersichtlichkeit nicht mehr beschriftet.

2.4.2 Anfang und Ende

Ein Spurgraph beginnt mit einem *Start*- und endet mit einem *Final*-Knoten.

Wie schon erwähnt unterscheiden sich Stapel- und Haldenobjekte in ihrer Lebensdauer. Bei Stapelobjekten wird an der Stelle der Deklaration in den Spurgraphen ein *Begin-Of-Lifetime*-Knoten eingefügt. Am Ende des Gültigkeitsbereichs steht ein *End-Of-Lifetime*-Knoten. Bei Haldenobjekten hingegen wird die Lebensdauer implizit durch *Start*- und *Final*-Knoten des Spurgraphen angegeben.

2.4.3 Direkte Zugriffe

Direkte Zugriffe sind Anweisungen, in denen einem Teil eines Objektes ein Wert zugewiesen wird (*Write*-Knoten) oder ein Teil eines Objektes in einem Ausdruck verwendet wird (*Read*-Knoten). In Abbildung 2.2 ist Beispielcode für direkte Zugriffe zu sehen.

```
typedef struct {...} MyADT;

MyADT object;
...
object.a = x; // Write
...
y := object.b; // Read
...
```

Abbildung 2.2: C-Code-Beispiel für direkte Zugriffe

Dabei ist zu beachten, dass *Read*- und *Write*-Knoten in der gegenwärtigen Implementierung der Spurextraktion keine Auskunft darüber geben, welcher Teil des Objekts gelesen oder geschrieben wird. Aussagen der Art „Bevor Routine *getContent()* ausgeführt werden darf, muss zunächst *object.hasContent* abgefragt werden“ können daher nicht gemacht werden. Diese Einschränkung ist jedoch zu verschmerzen, weil viele Programme derlei Zugriffe durch Routinen gekapselt haben (z.B.: *int hasContent(MyADT *obj)*).

2.4.4 Routinen

In den Spurgraphen können zwei Arten von Routinen vorkommen: Routinen, die primitive Operationen der untersuchten Komponente darstellen und Routinen, die aus beliebigem Quellcode stammen, der die Komponenten benutzt. Erstere werden ab jetzt Komponentenroutinen, letztere Benutzer-routinen genannt. Denselben Schema entsprechend definieren sich Komponentencode und Benutzercode.

2.4.5 Routinenaufrufe

Routinen-Aufrufe können in drei verschiedenen Konstellationen auftreten, die wir als Fall 1 bis 3 deklarieren.

Fall 1: Call-, Entry- und Return-Knoten

Abbildung 2.3 veranschaulicht eine Fall-1-Konstellation. Hier wird ein beliebiger ADT mit einer primitiven Operation *doSomething()* definiert. Darüberhinaus existiert eine Funktion *routine1()*, in der *doSomething()* aufgerufen wird (wir setzen an dieser Stelle voraus, dass das Objekt in *routine1()* korrekt instanziiert und initialisiert wird).

In Abbildung 2.4 auf der nächsten Seite sieht man den Spurgraphen, der aus dem Code-Beispiel entsteht. Der Aufruf *doSomething()* wird durch einen *Call*-Knoten repräsentiert. Der Routinenrumpf beginnt mit einem *Entry*- und endet mit einem *Return*-Knoten. Eine *Invoke*-Kante zeigt vom *Call*-Knoten auf den *Entry*-Knoten und eine *Return*-Kante zeigt vom *Return*-Knoten zurück auf den *Call*-Knoten.

```
typedef struct {...} MyADT;

// Manipulates object
void doSomething(MyADT *object) {
    object.a = ...;
    ...
}

void routine1() {
    MyADT object;
    ...
    doSomething(object); //Call Case 1
    ...
}
```

Abbildung 2.3: C-Code-Beispiel für Fall 1

Dieser Fall kann sowohl für Benutzer- als auch für Komponentenroutinen auftreten. In [Eis04] wird zwar gesagt, dass bei der Spurextraktion primitive Operationen als atomar angesehen werden und daher ihre Rümpfe nicht in den Spurgraphen erscheinen, tatsächlich ist dies aber nur eine Option des Werkzeugs zur Spurextraktion. Je nachdem, ob der Wartungs-Ingenieur diese Option zuschaltet oder nicht, können Fall-1-Konstellationen auch für Komponentenroutinen auftreten.

Diesem Fall kommt eine besondere Bedeutung zu, weil hier nicht nur intra-, sondern auch inter-prozeduraler Kontrollfluss modelliert wird. Damit können automatisch auch rekursive Aufrufe und mehrfache Aufrufe derselben Routine auftreten.

2 Grundlegendes

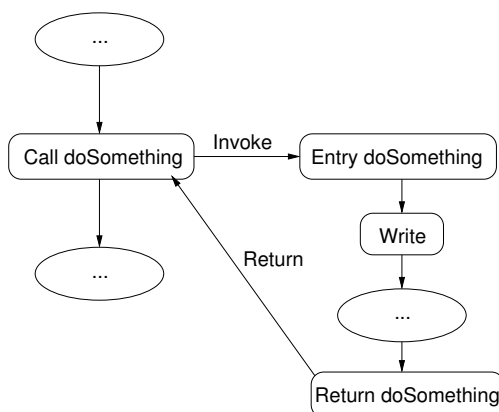


Abbildung 2.4: Spurgraph-Beispiel für Fall 1

Das Auftreten von rekursiven Aufrufen hat zunächst zur Folge, dass die Sprache, die durch den Spurgraph definiert wird, nicht mehr regulär, sondern kontext-frei ist.¹ Wie in Abschnitt 1.1 auf Seite 7 schon erwähnt wurde, können Spuren nur dann automatisch gegen Protokolle validiert werden, wenn sowohl die Sprachen der Spuren als auch die Sprachen der Protokolle regulär sind. In Kapitel 5 auf Seite 81 werden rekursive Aufrufe eingehend untersucht und es wird ein Verfahren gezeigt, wie Spurgraphen so verändert werden, dass ihre Sprache wieder regulär ist. Dabei muss zwangsläufig die Semantik der Spurgraphen verändert werden.

Betrachten wir nun die Eigenschaften mehrfach aufgerufener Routinen. Wird eine Routine innerhalb einer Spur mehrfach aufgerufen, dann ist der Routinenrumpf nur einmal enthalten, jedoch gibt es pro Aufruf einen *Call*-Knoten, der wie beschrieben über eine *Invoke*-Kante mit dem *Entry*-Knoten und über eine *Return*-Kante mit dem *Return*-Knoten des Routinenrumpfs verbunden ist. Eine solche Konstellation ist in Abbildung 2.5 auf der nächsten Seite zu sehen. Bei der Traversierung dieses Graphen muss darauf geachtet werden, dass beim Passieren des *Return*-Knotens zum korrekten *Call*-Knoten gegangen wird. Andernfalls betritt man Pfade, die nicht der Semantik des Spurgraphen entsprechen, siehe auch Definiti-

¹Genaugenommen ist sie kontext-sensitiv, denn zur Laufzeit werden ja immer nur bestimmte, aber nie alle Sequenzen durchlaufen. Allerdings sind etliche Probleme bei kontext-sensitiven Sprachen nicht entscheidbar, weswegen diese Einschränkung hingenommen werden muss.

on 2.5.5 auf Seite 39. Die korrekte Traversierung der Beispielkonstellation aus Abbildung 2.5 ist durch die Nummerierung der Kanten dargestellt.

Damit ist klar, dass Spurgraphen pfad-sensitiv traversiert werden müssen, z.B. durch Mitführen eines Stapels, auf den beim Betreten eines Routinenrumpfs der entsprechende *Call*-Knoten geschoben und beim Verlassen wieder heruntergenommen wird. Dieser Knoten ist automatisch immer der richtige Nachfolger eines Fall-1-*Return*-Knotens.

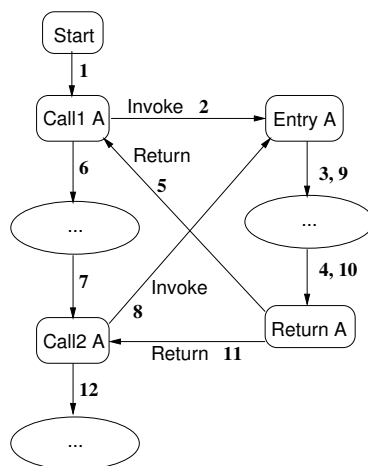


Abbildung 2.5: Traversierung bei mehrfach aufgerufenen Routinen

Fall 2: Call- und Return-Knoten

In Abbildung 2.6 auf der nächsten Seite ist ein Codebeispiel für den zweiten Fall zu sehen. Wieder wird ein ADT definiert. Diesmal gibt es eine Routine zum Anlegen neuer Variablen dieses Typs (*create()*). An einer beliebigen anderen Stelle (*routine1()*) wird die *create()*-Routine aufgerufen. Der Unterschied zu Fall 1 besteht darin, dass der Spurgraph erst nach dem Einsprung in die *create()*-Routine beginnt, weil hier der *malloc*-Aufruf steht, mit dem die Lebensdauer des Objekts beginnt. Ein solcher Fall kann offensichtlich nur bei einem Haldenobjekt auftreten.

Der passende Spurgraph ist in Abbildung 2.7 auf der nächsten Seite zu sehen. Da der Einsprung in die *create()*-Routine nicht Teil des Spurgraphen ist, existiert hier kein *Entry*-Knoten. Der *Call*-Knoten steht außerdem direkt hinter dem *Return*-Knoten, weil er an dieser Stelle zum ersten Mal

2 Grundlegendes

vorkommt. Wie in Fall 1 führt auch hier eine *Return*-Kante vom *Return*- zum *Call*-Knoten.

In Fall 2 existieren also nur *Call*- und *Return*-Knoten, jedoch kein *Entry*-Knoten.

Beachten Sie, dass beim Verlassen von *routine1* ebenfalls eine Fall-2-Konstellation im Spurgraph entsteht.

```
typedef struct {...} MyADT;  
  
// Creates object  
MyADT *create() {  
    ...  
    return (MyADT *)malloc(sizeof(MyADT));  
}  
  
void routine1() {  
    MyADT myObject;  
    ...  
    myObject = create(); //Call Case 2  
    ...  
}
```

Abbildung 2.6: C-Code-Beispiel für Fall 2

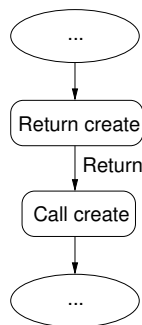


Abbildung 2.7: Spurgraph-Beispiel für Fall 2

Fall 3: Call- Knoten

In Abbildung 2.8 ist beispielhafter C-Code für den dritten Fall beschrieben. Wieder wird ein ADT definiert, diesmal ist jedoch der *malloc*-Aufruf interessant. Hier wird eine Routine aufgerufen, die nicht Teil des untersuchten Codes ist.

Daher erscheint im Spurgraph (siehe Abbildung 2.9) nur der *Call*-Knoten. *Entry*- und *Return*-Knoten sind in diesem Fall nicht vorhanden.

Wie bei Fall 1 beschrieben, werden bei der Spurextraktion abhängig von einer zuschaltbaren Option die Rümpfe von Komponentenroutinen eingefügt oder nicht. Daher können Fall-3-*Call*-Knoten neben dem *malloc*-Aufruf auch bei Komponentenroutinen auftreten.

```
typedef struct {...} MyADT;

// Creates object
void routine1() {
    MyADT myObject;
    ...
    myObject = malloc(sizeof(MyADT)); //Call Case 3
    ...
}
```

Abbildung 2.8: C-Code-Beispiel für Fall 3

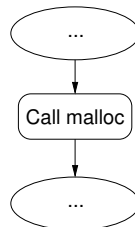


Abbildung 2.9: Spurgraph-Beispiel für Fall 3

2.4.6 Bedingungen

In Abbildung 2.10 sind zwei verschiedene Bedingungen zu sehen.

```
typedef struct {...} MyADT;

MyADT myObject;
...

if (getState(myObject)) // (1)
    manipFirst(myObject);
else
    manipSecond(myObject);
...

if (x) // (2)
    manipFirst(myObject);
else
    manipSecond(myObject);
...
```

Abbildung 2.10: C-Code-Beispiel für Bedingungen

Bei der *if*-Anweisung (1) wird zunächst anhand der primitiven Operation *getState()* der Zustand des Objektes abgefragt. Abhängig davon wird das Objekt entweder durch *manipFirst()* oder durch *manipSecond()* verändert.

Bei der *if*-Anweisung (2) passiert dasselbe mit dem Unterschied, dass die Bedingung (*x*) nicht offensichtlich vom Objekt abhängt. Es ist natürlich möglich, dass *x* zuvor der Rückgabewert von *getState()* zugewiesen wurde. Dies ist bei den hier beschriebenen und verwendeten Techniken jedoch nicht feststellbar. Es wäre möglich, mit Hilfe von Slicing [Wei84][Hor90] derlei Zusammenhänge herauszufinden und dem Spurgraphen hinzuzufügen.²

Abbildung 2.11 auf der nächsten Seite stellt die Spurgraphen für die beiden Codefragmente dar.³ In beiden Spurgraphen ist zu sehen, wie die Bedingung der *if*-Anweisung durch einen *Condition*-Knoten repräsentiert wird. Von ihm gehen jeweils eine *True*- und eine *False*-Kante aus, die zu den

²„Doch das ist eine andere Geschichte und soll ein andermal erzählt werden.“ – [End79]

³Die Routinenrümpfe wurden aus Übersichtlichkeitsgründen durch Platzhalterknoten (...) ersetzt.

2.4 Extrahierte Spurgraphen – Beschreibung

Call-Knoten von *manipFirst()* und *manipSecond()* führen. Von letzteren führt jeweils eine *Unconditional*-Kante zu einem *Join*-Knoten.

Join-Knoten werden an denjenigen Stellen im Spurgraph eingefügt, an denen der Kontrollfluss aus verschiedenen Zweigen zusammenläuft. Dies kommt z.B. am Ende von Schleifen oder von *if*-Anweisungen vor.

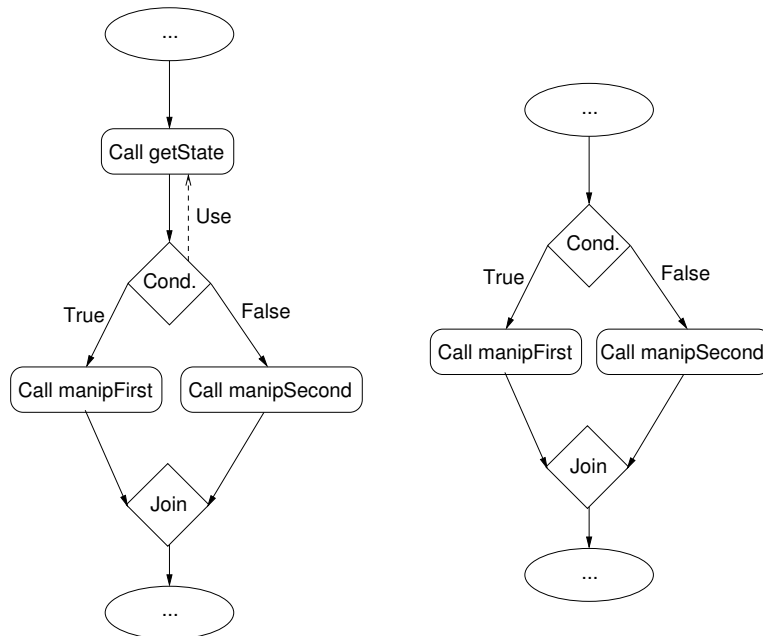


Abbildung 2.11: Spurgraph-Beispiel für Bedingungen.

Zurück zu den beiden Bedingungen – der Unterschied zwischen den beiden *Condition*-Knoten ist offensichtlich: Im linken Spurgraph ist der Ausdruck der Bedingung enthalten (*Call getState*). Der *Condition*-Knoten hat eine ausgehende *Use*-Kante, die auf diesen Ausdruck zeigt. Im rechten Spurgraph fehlt sowohl der Ausdruck als auch die *Use*-Kante. Das liegt daran, dass der Ausdruck in diesem Fall nicht in offensichtlicher Weise vom Komponenten-Objekt abhängt.

Der Bedingungsknoten wird zwar in den Spurgraphen eingefügt, weil Komponentenoperationen von ihm kontrollabhängig sind, der Ausdruck x selbst erscheint allerdings nicht im Spurgraph. Es sei jedoch vorweggenommen, dass der Wartungsingenieur beim Betrachten des Spurgraphen mit Gravis (siehe Abschnitt 2.6 auf Seite 42) eine komfortable Möglichkeit hat, sich den zugehörigen Quellcode anzusehen.

2 Grundlegendes

Besondere Beachtung gilt auch den *Use*-Kanten. Sie suggerieren zunächst, dass es mit ihrer Hilfe möglich ist, Bedingungen eindeutig zu charakterisieren, um z.B. *Condition*-Knoten auf Gleichheit zu überprüfen. Dies ist jedoch nicht der Fall. Beispielsweise ergeben zwei Anweisungen *if (!getState(object))* und *if (getState(object))* dasselbe Muster im Spurgraph: einen *Condition*-Knoten, dessen ausgehende *Use*-Kante auf einen *Call getState*-Knoten zeigt. Es gibt bei den Elementen des Spurgraphen keine Repräsentation für zusammengesetzte Ausdrücke, daher dient die *Use*-Kante tatsächlich in erster Linie dem Wartungs-Ingenieur zur besseren Übersicht in Protokoll- und Spurgraphen.

Nichtsdestoweniger kann aufgrund fehlender Alternativen zum Vergleich zweier *Condition*-Knoten die *Use*-Kante verwendet werden. Jedoch muss der Wartungs-Ingenieur sich im Klaren darüber sein, dass die Ergebnisse solcher Vergleiche in jedem Fall von ihm überprüft werden müssen.

Die Bedeutung und Behandlung von *Condition*-Knoten, die keine ausgehende *Use*-Kante besitzen, wird in Abschnitt 4.3 auf Seite 78 diskutiert

2.4.7 Relevante Knoten und Kanten

Für das Protokoll einer Komponente in unserem Sinne sind nur sequenzielle Abhängigkeiten von Operationen der Komponente relevant. Folglich gibt es in Spurgraphen relevante Knoten und Kanten sowie irrelevante Knoten und Kanten. Dabei sind folgende Knoten und Kanten relevant:

- *Start*- und *Final*-Knoten
- *Call*-Knoten, die Komponentenroutinen aufrufen, jedoch nicht die Knoten dieser Routinenrumpfe (siehe Unterabschnitt 2.4.4 auf Seite 24)
- Routinenrumpfe von Benutzerrountinen, jedoch nicht deren *Call*-Knoten
- *Read*-Knoten
- *Write*-Knoten
- *Unconditional*-Kanten zwischen den eben genannten Operationen
- *Condition*-Knoten, die über eine *Use*-Kante mit einer der eben genannten Knotenarten verbunden sind

2.5 Extrahierte Spurgraphen – Definition

- *True*- und *False*-Kanten, die von den eben genannten *Condition*-Knoten ausgehen.

Irrelevant sind somit folgende Knoten und Kanten:

- *Condition*-Knoten ohne ausgehende *Use*-Kante
- *True*- und *False*-Kanten, die von eben genannten *Condition*-Knoten ausgehen
- *Join*-Knoten
- *Call*-Knoten, die Benutzerrouninen aufrufen
- Knoten und Kanten aus Routinenrümpfen von Komponentenroutinen
- *Begin*- und *End_Of_Lifetime*- sowie *Entry*- und *Return*-Knoten

Diese Beschreibung ist nicht ganz exakt, nachdem es ja Kanten gibt, die von einem relevanten zu einem irrelevanten Knoten führen und umgekehrt. Für solche Fälle gilt diejenige Kante als relevant, die vom relevanten zum irrelevanten Knoten führt, wird jedoch so betrachtet, als führe sie zum nächsten relevanten Knoten.

Bei der Spurbereinigung (Kapitel 4 auf Seite 75) werden Spurgraphen derartig manipuliert, dass relevante Knoten und Kanten anschließend explizit von irrelevanten unterschieden werden können.

2.5 Extrahierte Spurgraphen – Definition

Nachdem extrahierte Spurgraphen informell beschrieben sind, folgen nun Definitionen ihrer Syntax und Semantik. Die Definitionen sind z.T. formal und z.T. natürlichsprachlich angegeben. An einigen Stellen empfiehlt es sich, zum besseren Verständnis die Abbildungen und informellen Beschreibungen des letzten Abschnitts heranzuziehen.

2.5.1 Syntax des Spurgraphen

Zunächst wird ein allgemeiner Graph eingeführt, den wir *Sequenzgraph* nennen. Auf dieser Definition wird dann die Definition des *extrahierten*

2 Grundlegendes

Spurgraphen aufgebaut, die durch zusätzliche Einschränkungen exakt die Graphen beschreibt, die das Werkzeug zur Spurextraktion ausgibt.

Definition 2.5.1 (Sequenzgraph) Ein Sequenzgraph ist ein 6-Tupel $(V, E, Labels, l, Start, Final)$ mit

- V ist eine endliche Menge (Knotenmenge)
- $E \subseteq V \times V$ (Kantenmenge)
- $Labels$ ist eine endliche Menge (Knotenlabels)
- $l: V \rightarrow Labels$. (Funktion zum Beschriften der Knoten)
- $Starts \subseteq V$ (Menge der Startknoten)
- $Finals \subseteq V$ (Menge der Endknoten).

Definition 2.5.2 (Hilfsdefinitionen) Für einen Sequenzgraph, $v \in V$ und $e \in E$ sei

$$\begin{aligned} Predecessors(v) &= \{w \mid w \in V \wedge (w, v) \in E\} \\ Successors(v) &= \{w \mid w \in V \wedge (v, w) \in E\} \\ Source(e) &= \{v \mid v, w \in V \wedge e = (v, w)\} \\ Target(e) &= \{v \mid v, w \in V \wedge e = (w, v)\} \\ Incomings(v) &= \{e \mid e \in E \wedge v \in Target(e)\} \\ Outgoings(v) &= \{e \mid e \in E \wedge v \in Source(e)\} \end{aligned}$$

Wenn nicht anders angegeben, beziehen sich Knoten- und Kantenmengen und deren Teilmengen ($V, E, Starts, Predecessors(v)$, etc.) von nun an immer auf den im Kontext der Definition angegebenen Graphen.

Definition 2.5.3 (Extrahierter Spurgraph) Ein extrahierter Spurgraph ist ein Sequenzgraph mit folgenden Einschränkungen:

- $Starts, Finals, Begin_Lifetimes, End_Lifetimes, Conditions, Joins, Calls, Entries, Returns, Reads, Writes$ sind endliche Mengen, die paarweise disjunkt sind (Repräsentation der Knotentypen).
- $Unconds, Trues, Falses, Invokes, Return_Edges, Uses \subseteq E$. $Unconds, Trues, Falses, Invokes, Return_Edges, Uses$ sind paarweise disjunkt (Repräsentation der Kantentypen).
- $V = Starts \cup Finals \cup Begin_Lifetimes \cup End_Lifetimes \cup Conditions \cup Joins \cup Calls \cup Entries \cup Returns \cup Reads \cup Writes$ (nur Knoten von diesen Typen sind erlaubt).

2.5 Extrahierte Spurgraphen – Definition

- $E = Unconds \cup Trues \cup Falses \cup Invokes \cup Return_Edges \cup Uses$ (nur diese Kantentypen sind erlaubt).
- $Operations = Calls \cup Reads \cup Writes$ (diese Knoten repräsentieren Operationen).
- $Sequences = Unconds \cup Trues \cup Falses$ (diese Kanten repräsentieren Sequenzen).
- $Traversables = Sequences \cup Invokes \cup Return_Edges$ (diese Kanten sind traversierbar).
- Es gibt genau einen Start- und einen Final-Knoten.
- *Start*-Knoten haben keine eingehenden Kanten und nur eine ausgehende Kante, die den Typ *Unconditional* haben muss.
- *Final*-Knoten haben keine ausgehende Kante und nur eine eingehende Kante, die den Typ *Sequences* haben muss.
- Alle *Begin_Of_Lifetime*-, *End_Of_Lifetime*- und *Write*-Knoten haben genau eine eingehende und eine ausgehende Kante. Die eingehende muss eine *Sequence*, die ausgehende eine *Unconditional*-Kante sein.
- *Read*-Knoten müssen eine eingehende *Sequence*-Kante haben, dürfen eine eingehende *Uses*-Kante haben, jedoch keine weiteren eingehenden Kanten. Sie müssen genau eine ausgehende Kante haben, die eine *Unconditional*-Kante sein muss.
- Alle und nur *Call*-, *Entry*- und *Return*-Knoten sind beschriftet durch die Beschriftungsfunktion $l : V \rightarrow Labels$ (Vorgriff auf die Semantik: Die Zugehörigkeit von *Call*-, *Entry*- und *Return*-Knoten wird durch diese Beschriftungsfunktion repräsentiert, d.h. die *Labels* enthalten die Namen aller Routinen, die in der Spur vorkommen).
- Fall-1-Aufrufe (*Calls_Case_1*) haben mindestens zwei, höchstens drei eingehende Kanten. Die eine ist eine *Sequence*-, die andere eine *Return*-Kante, die dieselbe Beschriftung wie der *Call*-Knoten hat. Falls eine dritte eingehende Kante existiert, muss es eine *Use*-Kante sein.

Fall-1-Aufrufe haben genau zwei ausgehende Kanten. Eine davon ist eine *Unconditional*-Kante, die andere eine *Invoke*-Kante, die zu

2 Grundlegendes

einem *Entry*-Knoten führt, der dieselbe Beschriftung wie der *Call*-Knoten hat.

- Fall-2-Aufrufe (*Calls_Case_2*) haben eine eingehende *Return*-Kante, die von einem *Return*-Knoten mit derselben Beschriftung stammt wie der *Call*-Knoten. Sie dürfen eine eingehende *Use*-Kante haben. Weitere eingehende Kanten gibt es nicht.

Fall-2-Aufrufe haben genau eine ausgehende Kante, die eine *Unconditional*-Kante sein muss.

- Fall-3-Aufrufe (*Calls_Case_3*) haben eine eingehende *Sequence*-Kante und können eine eingehende *Use*-Kante haben. Weitere eingehende Kanten haben sie nicht.

Fall-3-Aufrufe haben genau eine ausgehende Kante, die eine *Unconditional*-Kante sein muss.

- $Calls = Calls_Case_1 \cup Calls_Case_2 \cup Calls_Case_3$ (außer diesen drei Arten von Aufrufknoten darf es keine weiteren geben).

- *Entry*-Knoten haben mindestens eine eingehende Kante. Alle eingehenden Kanten müssen vom Typ *Invoke* sein. Sie haben genau eine ausgehende Kante, die vom Typ *Unconditional* sein muss.

- *Return*-Knoten haben genau eine eingehende Kante, die vom Typ *Sequences* sein muss. Sie haben mindestens eine ausgehende Kante. Alle ausgehenden Kanten müssen vom Typ *Return_Edges* sein.

- Für jede Routine, deren Rumpf im Graph enthalten ist, gibt es genau einen *Entry*- und einen *Return*-Knoten. *Entry*- und *Return*-Knoten, die derselben Routine angehören, müssen über *Invoke*- bzw. *Return*-Kanten mit denselben *Call*-Knoten verbunden sein.

- *Condition*-Knoten haben genau eine eingehende Kante, die vom Typ *Sequences* sein muss. Sie haben genau eine ausgehende *True*- und genau eine ausgehende *False*-Kante. Sie können eine ausgehende *Use*-Kante haben. Weitere ausgehende Kanten haben sie nicht. *Condition*-Knoten mit einer ausgehenden Kante gehören *Conditions_With_Uses*, was eine Teilmenge und damit einen Subtyp von *Conditions* darstellt.

Definition 2.5.4 (Hilfsdefinitionen mit Typ-Prädikaten) Oftmals müssen Vorgänger und Nachfolger eines Knotens bestimmt werden, die einem speziellen Typ angehören. Dasselbe gilt für eingehende und ausgehende Kanten. Um solche Konstellationen kompakter formulieren zu können, werden nun die Hilfsdefinitionen aus Definition 2.5.2 auf Seite 34 um Typ-Prädikate erweitert.

Für einen Sequenzgraph, $v \in V$ und $P \subseteq E$ sei

$$\begin{aligned} \text{Predecessors}(v, P) &= \{w \mid w \in V \wedge (w, v) \in P\} \\ \text{Successors}(v, P) &= \{w \mid w \in V \wedge (v, w) \in P\} \\ \text{Incomings}(v, P) &= \{e \mid e \in P \wedge v \in \text{Target}(e)\} \\ \text{Outgoings}(v, P) &= \{e \mid e \in P \wedge v \in \text{Source}(e)\} \end{aligned}$$

2.5.2 Semantik des Spurgraphen

Nachdem die Syntax eines extrahierten Spurgraphen formal beschrieben wurde, folgt nun die Beschreibung der Semantik, d.h. der Sequenzen, die ein extrahierter Spurgraph definiert.

Grundlage dafür sind die Pfade eines Spurgraphen. Sind diese definiert, dann muss nur noch beschrieben werden, wie aus den Pfaden die relevanten Informationen ausgegeben werden, d.h. Typ und Namen von relevanten Knoten und Kanten. Zu diesem Zweck wird eine Funktion eingeführt, die eine Zeichenkette für Knoten und Kanten ausgibt, die deren Typ beschreiben. Außerdem werden zusätzlich zu Pfaden noch relevante Pfade beschrieben, d.h. Pfade, in denen nur relevante Knoten und Kanten vorkommen. Aus den Definitionen für Pfade werden Definitionen für Teilpfade abgeleitet, die Wege genannt werden.

Betrachten wir zunächst die naheliegendste Beschreibung von Pfaden eines gegebenen Spurgraphen: Ein Pfad ist eine Folge von Knoten des Spurgraphen, die mit dem *Start*-Knoten des Spurgraphen beginnt und mit dem *Final*-Knoten des Spurgraphen endet, wobei jeder Knoten der Folge mit seinem direkten Nachfolger über eine *Traversable*-Kante verbunden sein muss.

Diese Definition der Pfade gilt für alle Knoten, die intra-prozeduralen Kontrollfluss modellieren, das sind alle Knoten außer Fall-1- und Fall-2-Aufrufen. Da bei Fall-2-Aufrufen ein Routinenrumpf lediglich verlassen,

2 Grundlegendes

jedoch nicht betreten wird, stellen diese Knoten keine Ausnahme dar und können ebenfalls durch die gerade umrissene Definition erfasst werden.

Die Ausnahmen bilden Fall-1-Aufrufe, da sie inter-prozeduralen Kontrollfluss beschreiben. Um diese Aufrufe erfassen zu können, werden die Routinenrumpfe so interpretiert, dass sie den Fall-1-*Call*-Knoten untergeordnet sind. Diese Sichtweise schlägt sich in der untenstehenden Definition durch die Menge *No_Subs* nieder, in der alle Knoten zusammengefasst sind, die keine untergeordneten Knoten besitzen. Anhand der Zugehörigkeit zu dieser Menge kann dann eine Fallunterscheidung vorgenommen werden, so dass die oben umrissene Beschreibung von Pfaden für alle Knoten aus *No_Subs* gilt.

Bei Fall-1-*Call*-Knoten muss darauf geachtet werden, dass zu jeder *Invoke*-Kante, über die ein Routinenrumpf betreten wird, die korrespondierende *Return*-Kante gewählt wird, um den Rumpf wieder zu verlassen (siehe auch Unterabschnitt 2.4.5 auf Seite 24). Dies lässt sich folgendermaßen bewerkstelligen: Stößt man bei der Erstellung eines Pfades auf einen Fall-1-*Call*-Knoten, dann wird zunächst dieser Knoten zum Pfad hinzugefügt. Anschließend wird ein gültiger Weg vom *Entry*- zum *Return*-Knoten der aufgerufenen Routine hinzugefügt. Danach wird der Pfad weitergeführt, indem die ausgehende *Unconditional*-Kante des *Call*-Knotens beschriftet wird. Für die Erstellung des Wegs im Routinenrumpf gelten rekursiv die gerade beschriebenen Definitionen für Fall-1-Aufrufe und für alle anderen Knoten.

Um auch korrekte Wege für Routinen zu erstellen, die rekursive Aufrufe enthalten, muss noch Folgendes beachtet werden: Zunächst kann man alle Wege durch einen Routinenrumpf als korrekt betrachten, die mit einem *Entry*-Knoten beginnen, der zur aufgerufenen Routine gehört und mit einem *Return*-Knoten enden, der zur aufgerufenen Routine gehört. Bei rekursiven Aufrufen reicht diese Bedingung nicht aus, weil der Routinenrumpf z.B. drei Mal betreten, jedoch nur ein Mal verlassen werden könnte, ohne gegen die gerade aufgestellte Bedingung zu verstoßen. Deshalb muss als weitere Bedingung hinzukommen, dass jeder Weg gleich viele *Entry*-Knoten der Routine wie *Return*-Knoten der Routine enthält. Zur Veranschaulichung:

$$\underbrace{(EntryA, \dots, \overbrace{EntryA, \dots, EntryA, \dots, Ret.A, \dots, Ret.A, \dots, Ret.A}})}_{}$$

Auf diese Art werden bei der Definition der Pfade auch rekursive Aufrufe korrekt behandelt.

2.5 Extrahierte Spurgraphen – Definition

Die Erstellung des Weges, der durch einen Routinenrumpf führt, wird durch die Funktion $Body(v)$ bewerkstelligt, wohingegen die Funktion $Sub(v)$ die oben beschriebene Fallunterscheidung zwischen Fall-1-Aufrufen und No_Subs macht und damit entweder $Body(v)$ liefert oder den fraglichen Knoten.

Definition 2.5.5 (Pfade) Für einen extrahierten Spurgraphen S sei:

$$No_Sub = V \setminus Calls_Case_1$$

$$\begin{aligned} Paths &= \{(v_1, Sub(v_2), Sub(v_3), \dots, Sub(v_{n-1}), v_n) \mid \\ &\quad v_1 \in Starts, v_n \in Finals \\ &\quad \wedge ((w_1, \dots, w_m) \\ &\quad = (v_1, Sub(v_2), Sub(v_3), \dots, Sub(v_{n_1}), v_n) \\ &\quad \Rightarrow \forall i \in [1, m-1] : (w_i, w_{i+1}) \in Traversables)\} \end{aligned}$$

$$Sub(v \in V) = \begin{cases} v & \text{für } v \in No_Sub \\ b \in Body(v) & \text{für } v \in Calls_Case_1 \end{cases}$$

$$\begin{aligned} Body(v) &= \{(v_i, Sub(v_{i+1}), Sub(v_{i+2}), \dots, Sub(v_j)) \mid \\ &\quad v = v_i \\ &\quad \wedge v_{i+1} \in Entries \\ &\quad \wedge v_j \in Returns \\ &\quad \wedge l(v_i) = l(v_{i+1}) = l(v_j)\} \end{aligned}$$

Basierend auf dieser Definition können nun Wege ($Walks$) definiert werden. Ein Weg eines Spurgraphen S ist dabei eine Folge von Knoten, die Teilfolge eines Pfades von S ist. Zur Veranschaulichung:

$$\begin{array}{ll} \text{Pfad: } & (w_1, \dots, w_k, \dots, w_l, \dots, w_m) \\ \text{Weg: } & (v_1, \dots, v_n) \end{array}$$

Definition 2.5.6 (Wege) Für einen extrahierten Spurgraphen S sei:

$$\begin{aligned} Walks &= \{(v_1, \dots, v_n) \mid \exists w = (w_1, \dots, w_m) \in Paths \\ &\quad \exists k, l \in [1, m] \\ &\quad \forall i \in [1, n] : \\ &\quad v_i = w_{k+i-1}\} \end{aligned}$$

(siehe letzter Absatz).

2 Grundlegendes

$$Walks_From(v) = \{(v_1, \dots, v_k) \mid (v_1, \dots, v_k) \in Walks(S) \\ \wedge v_1 = v\}$$

(alle Wege, die mit v anfangen).

$$Walks_To(v) = \{(v_1, \dots, v_k) \mid (v_1, \dots, v_k) \in Walks \wedge v_k = v\}$$

(alle Wege, die mit v aufhören).

$$Walks_From_To(v, w) = \{(v_1, \dots, v_k) \mid (v_1, \dots, v_k) \in Walks \\ \wedge v_1 = v \wedge v_k = w\}$$

(alle Wege, die mit v_1 anfangen und mit v_k aufhören).

Nun sind sowohl Pfade als auch Wege definiert. Um darauf aufbauend Sequenzen zu definieren, muss zunächst noch eine Möglichkeit geschaffen werden, die Typinformation von Knoten und Kanten explizit anzugeben. Dies geschieht durch eine Typ-Beschriftungsfunktion, die die Beschriftungsfunktion l aus Definition 2.5.1 auf Seite 34 ergänzt.

Definition 2.5.7 (Typbeschriftung) Für einen extrahierten Spurgraphen S sei:

$$t : V \cup E \rightarrow Strings \text{ mit}$$

$$t(v) = \begin{cases} \text{Start,} & \text{für } v \in Starts, \\ \text{Call,} & \text{für } v \in Calls, \\ \dots & \\ \text{Final,} & \text{für } v \in Finals. \end{cases}$$

$$t(e) = \begin{cases} \text{Uncond,} & \text{für } e \in Unconds, \\ \text{True,} & \text{für } e \in Trues, \\ \dots & \\ \text{Use,} & \text{für } e \in Uses. \end{cases}$$

In Unterabschnitt 2.4.7 auf Seite 32 wurde beschrieben, welche Knoten und Kanten relevant sind. Dies wird im Folgenden definiert.

Definition 2.5.8 (Relevante Knoten) Für einen extrahierten Spurgraphen S , eine Komponente K , und $Primitives(K)$, die die Menge der Namen der primitiven Operationen von K repräsentiert, sei:

$$Relevant_Calls(K) = \{v \mid v \in Calls \wedge l(v) \in Primitives(K)\}$$

(alle Calls, die eine primitive Routine von K aufrufen).

2.5 Extrahierte Spurgraphen – Definition

$$\begin{aligned} Irrelevant_Bodies(K) = \{v | (\exists w = (w_1, \dots, w_n) \in Walks : \\ w_1 \in Entries \wedge w_n \in Returns \\ \wedge l(w_1) = l(w_n) \\ \wedge l(w_1) \notin Primitives(K)) \\ \wedge v \in \{w_1, \dots, w_n\}\} \end{aligned}$$

(alle Knoten, die auf einem Weg vom *Entry*-Knoten einer Benutzer-routine zum *Return*-Knoten der Benutzeroutine liegen).

$$\begin{aligned} Relevant_Conditions(K) = \{v | v \in Conditions_With_Uses \\ \wedge (\exists (v, w) \in Uses : w \in Relevant_Calls(K) \cup Reads)\} \end{aligned}$$

(alle Bedingungsknoten, deren Use-Kante auf ein *Read* oder auf einen *Call*-Knoten zeigt, der eine primitiven Operation von K aufruft).

$$\begin{aligned} Relevant_Nodes(K) \\ = (Reads \cup Writes \cup Relevant_Calls(K) \\ \cup Relevant_Conditions(K)) \setminus Irrelevant_Bodies \end{aligned}$$

(zu den eben beschriebenen relevanten Knoten sind zusätzlich *Reads* und *Writes* relevant, abzüglich aller Knoten, die in einem Routinenrumpf einer Komponentenroutine vorkommen).

Mit Hilfe dieser Definition können nun, basierend auf Wegen und Pfaden, relevante Wege und Pfade formuliert werden:

Definition 2.5.9 (Relevante Pfade und Wege) Für einen extrahierten Spurgraph S sei:

$Relevant_Path(path) = (w_1, \dots, w_n)$ die Folge von Knoten eines $path \in Paths$, aus der alle irrelevanten Knoten im Sinne obiger Definition entfernt wurden.

Dies gelte analog für

$$Relevant_Walk(walk, v), walk \in Walks(v)$$

$$Relevant_Walk_From(walk, v), walk \in Walks_From(v)$$

$$Relevant_Walk_To(walk, v), walk \in Walks_To(v)$$

$$Relevant_Walk_From_To(walk, v, w), w \in Walks_From_To(v, w)$$

Es seien $Relevant_Walks$, $Relevant_Walks_From(v)$, $Relevant_Walks_To(v)$, $Relevant_Walks_From_To(v, w)$, $Relevant_Paths$ jeweils die Menge aller Wege bzw. Pfade aus S nach obigen Definitionen.

2 Grundlegendes

Nachdem nun alle notwendigen Voraussetzungen eingeführt wurden, folgt schließlich die Definition der Sequenzen eines extrahierten Spurgraphen.

Definition 2.5.10 (Sequenzen eines extrahierten Spurgraphen)

Für einen extrahierten Spurgraph S , für den alle gerade vorgestellten Definitionen gelten und eine Funktion \circ , die Zeichenketten konkateniert, sei:

$$\begin{aligned} \text{Sequences} = & \\ & \{t(v_1) \circ l(v_1), t((v_1, w_{i_1})), t(v_2) \circ l(v_2), \dots, t((v_{k-1}, w_{i_m})), t(v_k) \circ l(v_k) | \\ & w = (w_1, \dots, w_n) \in \text{Paths} \\ & v = (v_1, \dots, v_k) = \text{Relevant_Path}(w) \\ & j, i_j, m \in [1, n]\}. \end{aligned}$$

(Sequenzen ergeben sich aus den relevanten Pfaden, indem man von jedem Knoten Typ und Name ausgibt und von jeder Kante den Typ. Da relevante Knoten nicht immer direkt durch Kanten verbunden sind, sondern irrelevante Knoten dazwischen liegen können, werden jeweils die ausgehenden Kanten der relevanten Knoten zur Ausgabe des Kantentyps herangezogen.)

2.6 Gravis

Gravis (Abbildung 2.12 auf der nächsten Seite) ist ein Grapheneditor, der speziell dafür entwickelt wurde, RFGs darzustellen und zu bearbeiten.

Zunächst kann der Wartungsingenieur verschiedene Sichten (Views) eines RFG öffnen und betrachten. Jede Sicht zeigt eine Teilmenge aller Knoten des momentan geladenen RFGs. So werden z.B. die extrahierten Spuren in einer Sicht gespeichert. Oftmals enthalten Sichten eine Vielzahl von Knoten und Kanten. Daher gibt es diverse Layout-Algorithmen, die auf ein Sichtfenster angewendet werden können. Darüber hinaus kann zu allen Knoten, die Quellcode repräsentieren, durch einen Doppelklick der entsprechende Quellcode in einem zusätzlichen Fenster angezeigt werden. Dies kann z.B. sehr schön bei den oben erwähnten *Condition*-Knoten verwendet werden. Gravis bietet außerdem eine Vielzahl nützlicher Such-, Markier- und Selektierfunktionen.

Außerdem können beliebige Knoten und Kanten angelegt und bearbeitet werden. Zum Bearbeiten zählt insbesondere das Umbenennen von Knoten.

Zuguterletzt bietet Gravis die Möglichkeit, Bauhaus-Werkzeuge interaktiv aufzurufen. Sowohl die Komponentenerkennung als auch die Spurextrak-

tion kann von Gravis aus aufgerufen werden. Bei der Spurextraktion kann der Wartungs-Ingenieur interaktiv genau diejenigen Objekte im Objekt-Prozess-Graph bestimmen, für die er Spuren extrahiert haben möchte (siehe Abschnitt 2.3 auf Seite 21).

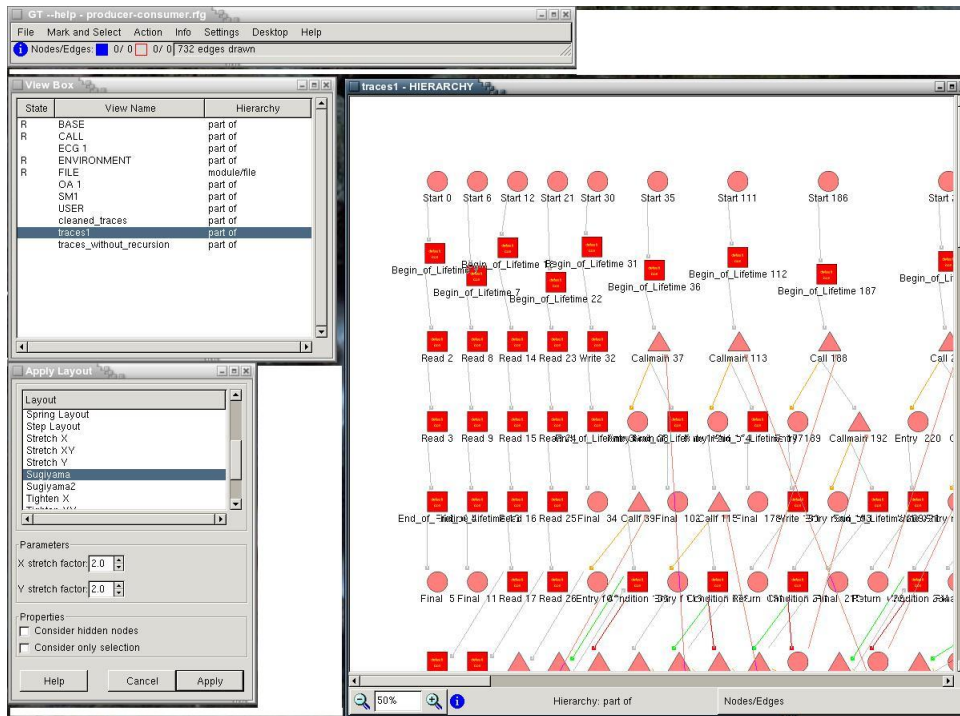


Abbildung 2.12: Screenshot von Gravis

2.7 Verwandte Forschung

In diesem Abschnitt werden Forschungsergebnisse beschrieben, die mit dem Thema dieser Arbeit verwandt sind. Einige wurden schon in Abschnitt 1.2 auf Seite 12 erwähnt. Für etliche der unten aufgeführten diente [Eis04] als Quelle.

2.7.1 Herleitung von Protokollen

Die Extraktion von Spuren wird in diesem Abschnitt als der Herleitung von Protokollen zugehörig betrachtet. Dies macht insofern Sinn, da es ein der Herleitung vorgelagerter Schritt ist.

2 Grundlegendes

ISVis [Jer97a][Jer97b] ist ein System, das es dem Wartungs-Ingenieur ermöglicht, aus Low-Level-Verhalten auf Muster auf Entwurfsebene zu schließen. Dazu protokolliert es interessante Ereignisse zur Laufzeit mit, die der Wartungs-Ingenieur anschließend mit Hilfe der graphischen Darstellungen von ISVis analysieren kann. Er identifiziert daraus Interaktionsmuster und fügt diese zu einem Verhaltensmodell zusammen. Allerdings gelten diese Ergebnisse – wie bei allen dynamischen Analysen – nur für genau die Szenarien, unter denen die Ereignisse protokolliert wurden.

Auch Cook and Wolf [Coo98] beschreiben eine dynamische Analyse. Dabei werden wiederkehrende Verhaltensmuster automatisch aus einer Menge dynamisch gesammelter Ereignisse erzeugt. Damit führen sie das Problem der Herleitung von Protokollen auf das Problem der Grammatik-Inferenz zurück. In der Grammatik-Inferenz wird versucht, eine reguläre Grammatik aus einer Menge von Wörtern herzuleiten.

Whaley, Martin und Lam [Wha02] leiten Protokolle von objekt-orientierten Komponenten statisch her, ergänzt durch eine dynamische Analyse. Die Schnittstellen dieser Komponenten überwachen ihre Benutzung, indem sie paarweise Kombinationen von Lese- und Schreibzugriffen auf ihre Attribute suchen, die Exceptions auslösen. Im Gegensatz zur oben beschriebenen Technik der Spurextraktion, die den Quellcode analysiert, der die Komponente benutzt, werden hier die inneren Abhängigkeiten der Komponente untersucht.

In Kapitel 3 auf Seite 47 werden die Ansätze zur Herleitung von Protokollen genauer beschrieben, die für diese Arbeit in Betracht gezogen werden.

2.7.2 Validierung mit Protokollen

Hat man ein Protokoll hergeleitet, dann kann es verwendet werden, um statische Spuren auf Korrektheit zu testen. Wie schon in Abschnitt 1.2 auf Seite 12 angesprochen, muss dafür gezeigt werden, dass die Sprache, die vom Spurgraph beschrieben wird, eine Teilmenge der Sprache ist, die der Protokollgraph beschreibt. Dieses Problem ist allerdings nur für reguläre Sprachen entscheidbar. Beschreibt man Protokolle und Spuren als endliche Automaten (oder in einer Form, die äquivalent zu endlichen Automaten ist, wie die hier beschriebenen Spurgraphen), dann können zur Validierung bekannte Verfahren aus der Automatentheorie verwendet werden, wie z.B. die Bildung eines Potenzmengenautomaten nach Rabin-Scott [Hop79] über

Spuren und Protokoll. Dieses Verfahren wird in Abschnitt 3.2 auf Seite 57 umrissen und in Abschnitt 6.3 auf Seite 101 beschrieben.⁴

Olender und Osterweil vertreten einen alternativen Ansatz, indem sie mit Hilfe eines Datenfluss-Rahmenwerks Zustandsübergänge durch den Kontrollfluss-Graphen propagieren. Damit können nicht nur universell, sondern auch existenziell quantifizierte Bedingungen überprüft werden. Der Ansatz ist jedoch insofern eingeschränkt, dass die Parameter von validierten Routinen nicht berücksichtigt werden. Bei parameterlosen Routinen ist die Analyse des Kontrollfluss-Graphen zwar ausreichend, will man jedoch die Beziehungen zwischen Routinen und Objekten mitbetrachten, kann dieser Ansatz nicht direkt verwendet werden.

Das, Lerner und Seigle beschreiben in [Das02] ein Verfahren zur pfad-sensitiven Validierung von Programmen in polynomieller Zeit. Dies steht im Gegensatz zu Ansätzen wie der oben angesprochenen Potenzmengenbildung, die im schlimmsten Fall exponentiellen Aufwand haben. Das und Kollegen führen mit der „Property Simulation“ ein Verfahren ein, das entlang eines Kontrollflussgraphen den Zustand nur von relevanten Zweigen mitführt. Relevante Zweige sind diejenigen, die den Zustand des endlichen Automaten ändern, der das spezifizierten Protokolls repräsentiert. Dieses Verfahren wird mit der Analyse des Datenflusses kombiniert, die eine Points-To-Analyse einschließt. Dadurch können, wie im hier vorgestellten Ansatz der Spurextraktion, einzelne statisch nachweisbare Objekte verfolgt werden.

⁴Abschnitt 6.3 behandelt zwar die Validierung und nicht die Herleitung von Protokollen, in Abschnitt 3.2 wird jedoch erklärt, wie dieses Verfahren auch zur Validierung mit Protokollen verwendet werden kann.

2 Grundlegendes

3 Ansätze zur Herleitung von Protokollen

Nachdem nun das Umfeld der Arbeit und alle vorgelagerten Schritte beschrieben wurden, wird in diesem Kapitel die Aufgabenstellung nochmals genau beleuchtet. Der darin beschriebene Ansatz wird einem weiteren Ansatz gegenübergestellt. Am Ende des Kapitels wird einer davon ausgewählt, der im Rest der Arbeit umgesetzt wird.

3.1 Induktiver Ansatz

Heiber beschreibt in [Hei00] ausführlich, wie ein Protokollgraph aus Spurgraphen hergeleitet und anschließend automatisch sowie semi-automatisch vereinfacht werden kann. Sein Ansatz ist in Abschnitt 1.2 auf Seite 12 zusammengefasst. Dieses Vorgehen wird im Weiteren als induktiv bezeichnet, weil dabei aus den Spurgraphen ein Protokollgraph induziert wird.

Um diesen Ansatz weiterzuverfolgen, wie in Abschnitt 1.2 auf Seite 12 gefordert, müssen zunächst die Ausführungen und die Implementierung von Heiber auf die aktuellen Spurgraphen übertragen werden. Dann müssen weitere Muster zur Vereinfachung gefunden werden, die häufig vorkommen und nicht unmittelbar von der untersuchten Komponente abhängen. Des Weiteren müssen Muster charakterisiert werden, die der Benutzer graphisch und textuell spezifizieren kann und die in der Regel nur auf eine spezielle Komponente anwendbar sind. Mit Hilfe der beiden Arten von Mustern muss dann ein Graphtransformationssystem spezifiziert werden, in das alle Vereinfachungen eingegeben und auf dem Protokollgraph ausgeführt werden können. Ein solches Graphtransformationssystem muss anschließend entweder neu implementiert oder an die Bauhaus-Werkzeuge angebunden werden.

Im Folgenden werden die einzelnen Schritte genauer beleuchtet.

3.1.1 Übertragen von Heibers Ergebnissen

Heibers Ausführungen bauen auf der Spurextraktion von Hanssen [Han00] auf. Die extrahierten Spurgraphen haben sich mittlerweile jedoch maßgeblich geändert. Daher müssen die Überlegungen zur Reduzierung der Knotenzahl und zur Vereinfachung der induktiv erzeugten Protokollgraphen erweitert werden.

Bereinigung der Spuren

In Unterabschnitt 2.4.7 auf Seite 32 erläutert, dass es relevante und irrelevante Knoten gibt. Bei einigen der irrelevanten Knoten ist es sinnvoll, sie zu entfernen, um die Spurgraphen zu verkleinern. Darüber hinaus werden wir sehen, dass *Condition*-Knoten ohne ausgehende *Use*-Kante als nicht-deterministisch zu betrachten sind und daher ihre ausgehenden *True*- und *False*-Kanten durch *Unconditional*-Kanten ersetzt werden können. Kapitel 4 auf Seite 75 beschäftigt sich eingehend mit diesem Thema.

Minimierung

Heiber beschreibt in [Hei00], wie der Minimierungsalgorithmus für endliche Automaten aus [Hop79] zur Minimierung von induktiv erzeugten Protokollgraphen verwendet werden kann. Der Algorithmus findet gemeinsame Präfixe und Suffixe verschiedener Zweige, die vom Start-Knoten bzw. Endknoten ausgehen und fasst diese zusammen.

In Abschnitt 2.4 auf Seite 22 wurden die Spurgraphen charakterisiert, die in der vorliegenden Arbeit verwendet werden. Dabei wurde beschrieben, dass durch die Modellierung von Routinenaufrufen und Rümpfen rekursive Aufrufe und Mehrfachaufrufe in Spurgraphen vorkommen können.

Betrachten wir nun beispielhaft einen Spurgraphen, der Mehrfachaufrufe enthält und versuchen darauf, die Präfix-Minimierung anzuwenden¹.

¹Rekursive Aufrufe werden zunächst außer Acht gelassen, da in Kapitel 5 auf Seite 81 ein Verfahren beschrieben wird, wie Rekursive Aufrufe aus Spurgraphen entfernt werden können.

Abbildung 3.1 zeigt einen solchen Spurgraphen². Die Routinen X , Y und Z seien Komponentenroutinen, A hingegen sei eine Benutzeroutine.

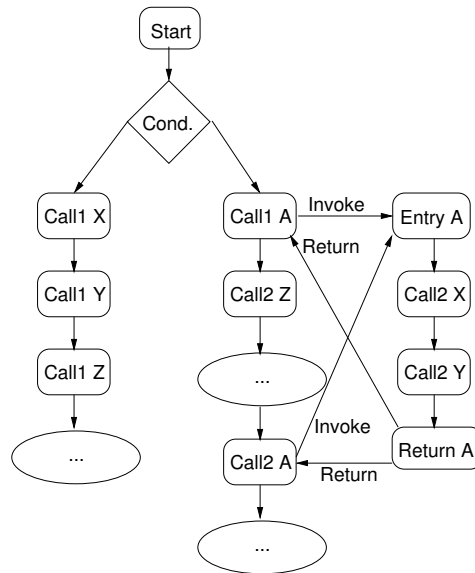


Abbildung 3.1: Neuer Fall bei der Prafix-Reduktion

Der automatische Minimierungsalgorithmus identifiziert ausgehend vom Startknoten zwei gleiche Prafixe:

- $Call1 X$, $Call1 Y$, $Call1 Z$.
- $Call2 X$, $Call2 Y$, $Call2 Z$.

$Call1 A$ wird ignoriert, weil es sich dabei um eine Benutzeroutine handelt. Beachten Sie, dass die Routine A noch von einer spateren Stelle des Protokollgraphen aus aufgerufen wird.

Wurde man nun die Prafixminimierung nach gewohntem Muster vornehmen, dann mussten dabei $Call A$, $Entry A$ und $Return A$ gesondert behandelt werden. Angenommen, man eliminierte sie einfach, dann wurde damit der Protokollgraph unerlaubt verandert werden: $Call2 A$ konnte nicht mehr mit dem Rumpf von A verbunden werden. Die einzige Alternative besteht darin, in einem Zwischenschritt den Routinenrumpf von A

² Die Routinenrumpfe von X , Y und Z sind aus Grunden der ubersichtlichkeit weglassen worden.

3 Ansätze zur Herleitung von Protokollen

zu kopieren, wie in Abbildung 3.2 dargestellt. Dadurch erhöht sich jedoch die Zahl der Knoten des Protokollgraphen um die Zahl der Knoten im Routinenrumpf von *A*.

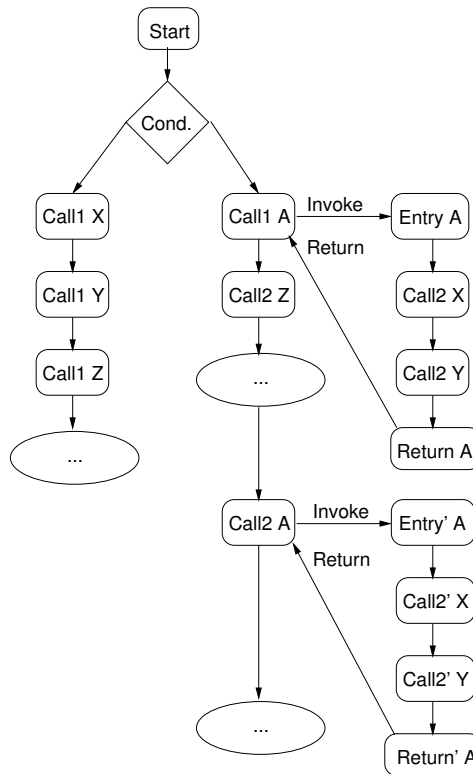


Abbildung 3.2: Zwischenschritt der Minimierung – Routinenrumpf *A* kopiert

Aufgrund dieser Konstellation muss der Minimierungsalgorithmus also zunächst prüfen, ob in den zusammenfassenden Zweigen eine oder mehrere Routinen betreten werden. Dann muss er die Zahl der Knoten aller Routinen, die vor der Minimierung kopiert werden müssen, mit der Zahl der Knoten vergleichen, die durch die Minimierung eingespart werden. Abhängig davon führt er die Minimierung durch oder nicht.

3.1.2 Finden von Vereinfachungsmustern

In [Hei00] werden schon diejenigen Muster angegeben, die durch Reflektieren über Spur- und Protokollgraphen herausgefunden werden können. Das Finden von weiteren Mustern dieser Art durch Analysieren von Spurgraphen ist semi-entscheidbar, denn es ist nicht sicher, ob es weitere Muster dieser Art überhaupt gibt.

Bei einer ersten Betrachtung von Spurgraphen eines C-Programms hat sich bei mir der Verdacht erhärtet, dass Spuren zwar Aussagen über die sequenziellen Abhängigkeiten der Komponente treffen, die sie benutzen, jedoch gibt es wenig Information, die von diesen speziellen Abhängigkeiten auf Vereinfachungsmuster schließen lassen, die auf Spurgraphen von anderen Komponenten übertragbar sind. Die Chance, weitere allgemeingültige Muster durch Suche in Spuren zu finden, ist meiner Meinung nach sehr gering.

Nichtsdestoweniger müssen Muster gefunden werden, um das Graphtransformationssystem spezifizieren zu können.

3.1.3 Graphtransformationssysteme

Es gibt eine Vielzahl von Ansätzen bei Graphtransformationssystemen. Die meisten Berichte beschreiben jedoch lediglich theoretische Untersuchungen von einzelnen Aspekten. Daher liegt das Augenmerk bei den meisten Ansätzen nicht auf praktischer Anwendbarkeit für reale Probleme, sondern auf beschränkten Definitionen, die eine Untersuchung des gewünschten Aspekts möglich machen. [Ehr91] enthält eine Vielzahl solcher Ansätze. Eine Ausnahme bildet [Sch91]. Darin wird eine komplexe Graphtransformationssprache formal beschrieben, die in einem Werkzeug zur visuellen Programmierung (PROGRES) zum Einsatz kommt.

Graphersetzungsregeln

Wir werden nun anhand eines Beispiels die wichtigsten Kriterien von Graphtransformationssystemen herausarbeiten.

Ein Graphtransformationssystem akzeptiert Regeln zur Ersetzung von Teilgraphen. Eine beispielhafte Ersetzung ist in Abbildung 3.3 auf der nächsten Seite zu sehen. Darin werden die Knoten C und D samt der

3 Ansätze zur Herleitung von Protokollen

zwischen ihnen liegenden Kante durch einen Knoten X ersetzt. X erhält als eingehende Kanten nach der Ersetzung alle eingehenden Kanten von beiden Knoten C und D , mit den ausgehenden Kanten verhält es sich genauso. Eine Ausnahme bilden die Kanten zum Knoten F , denn ursprünglich waren C und D mit F verbunden, X hat jedoch nur eine ausgehende Kante zu F . Grund dafür könnte sein, dass bei der zugrunde liegenden Definition von Graphen keine Multikanten erlaubt sind. Darüber hinaus wird von X noch eine Kante zu H gezogen.

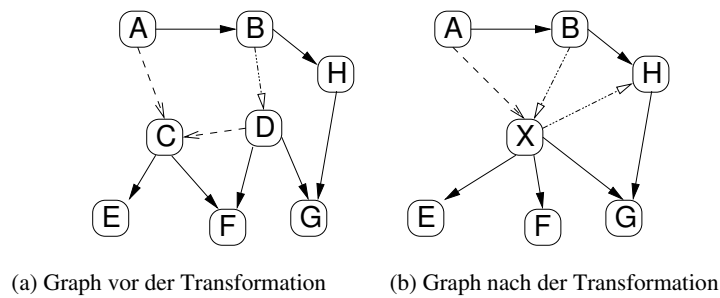


Abbildung 3.3: Beispiel für eine Graphtransformation

An diesem recht simplen Beispiel kann man sehen, dass eine Ersetzungsregel mindestens aus drei Teilen besteht:

1. Linke Seite des Ersetzungsmusters (hier: C , D und die dazwischenliegende gestrichelte Kante)
2. Rechte Seite des Ersetzungsmusters (hier: X)
3. Einbettungsregeln (hier: X bekommt alle eingehenden und ausgehenden Kanten von C und D . Multikanten werden eliminiert. X bekommt eine ausgehende Kante zu H .)

Ein weiterer Teil einer Ersetzungsregel können Anwendbarkeitsbedingungen sein, die genau definieren, unter welchen Umständen die Ersetzung angewendet werden darf. Über die Regeln hinaus kann es Konsistenzbedingungen geben, die der Graph einhalten muss und die nach jeder Anwendung einer Ersetzungsregel überprüft werden.

Wird eine solche Graphersetzung von einem Graphtransformationssystem ausgeführt, dann muss zunächst der Graph nach dem Muster der linken Seite durchsucht werden. Falls vorhanden, werden dann die Anwendbarkeitsbedingungen überprüft. Jede gefundene Stelle, die die Anwendbar-

keitsbedingungen erfüllt, wird als nächstes durch die rechte Seite des Musters ersetzt. Zuguterletzt werden die eingefügten Knoten durch die Einbettungsregeln mit den restlichen Teilgraphen verbunden.

Textuelle und graphische Spezifikation

Betrachten wir nun die Möglichkeiten der Darstellung von den verschiedenen Teilen einer Graphersetzungssprache. In der Ausschreibung (Abschnitt 1.2 auf Seite 12) ist gefordert, sowohl eine textuelle als auch eine graphische Repräsentation einer Graphersetzungssprache zu entwerfen.

Am gerade gezeigten Beispiel kann man sehen, dass die graphische Darstellung sehr gut für die linke und rechte Seite eines solchen Musters geeignet ist, bei komplexen Einbettungsregeln und Anwendbarkeitsbedingungen jedoch schnell an ihre Grenzen stößt. Umgekehrt ist die textuelle Repräsentation für Einbettungsregeln und Anwendbarkeitsbedingungen sehr gut geeignet, linke und rechte Seite eines Ersetzungsmusters sind jedoch bei nicht-trivialen Mustern sehr unübersichtlich (siehe hierzu auch [Sch91]). Als Schlussfolgerung daraus verwendet PROGRES eine Kombination beider Ansätze: linke und rechte Seite des Musters werden graphisch angegeben, Einbettungsregeln und Anwendbarkeitsbedingungen textuell.

Es gibt Ansätze, die versuchen, auch die Einbettungsregeln graphisch zu repräsentieren. Dazu gehört u.a. die X-Notation [Gö88]. In Abbildung 3.4 auf der nächsten Seite ist das obige Beispiel in der X-Notation zu sehen. 1 entspricht dabei der linken Seite der Ersetzungsregel, 2 entspricht der rechten. 3 enthält alle Teile des Graphen, die mit der linken und/oder der rechten Seite verbunden sind und identisch ersetzt werden. In 4 sind diejenigen Knoten enthalten, die zur Einbettung der rechten Seite über den identisch zu ersetzenden Teil hinaus notwendig sind.

Grobe Einordnung der Ansätze

Als nächstes versuchen wir eine – wenn auch sehr grobe – Einteilung der verschiedenen Ansätze der Graphtransformationssysteme zu beschreiben. Dieser Unterabschnitt ist im Wesentlichen aus Teilen von [Sch91] zusammengefasst.

Neben den Graphtransformationssystemen gibt es noch das Gebiet der Graphgrammatiken. Oft werden beide Begriffe als Synonym verwendet.

3 Ansätze zur Herleitung von Protokollen

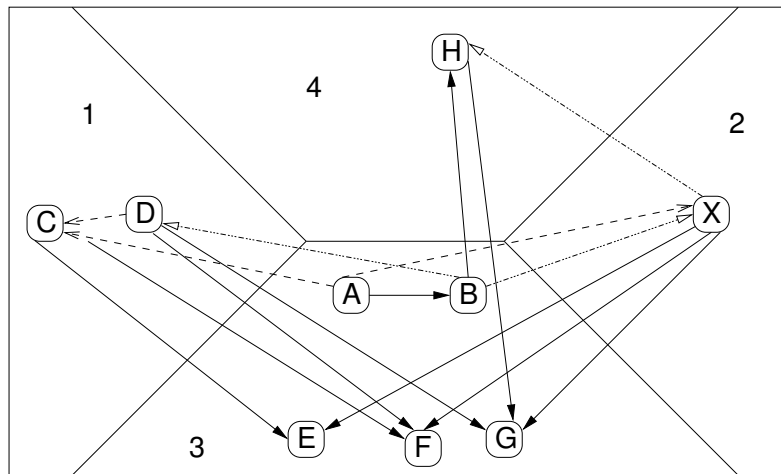


Abbildung 3.4: Beispiel für Graphersetzung in X-Notation

Jedoch beschäftigen sich Graphgrammatiken – ähnlich wie Grammatiken über Zeichenketten – eher mit der Erzeugung von Graphen nach gewissen Regeln, wohingegen Graphtransformationssysteme vorhandene Graphen manipulieren, indem sie Teil- oder Untergraphen ersetzen.

Die Ansätze von Graphtransformationssystemen und Graphgrammatiken lassen sich grob in zwei Kategorien unterteilen: algorithmische und algebraische Ansätze.

Algorithmische Ansätze basieren auf Mengen und Mengenoperationen. So sind Kanten (wie auch bei den hier definierten Graphen, siehe z.B. Definition 2.5.1 auf Seite 34) in der Regel als Relationen über Knotenmengen definiert. Dies hat zur Folge, dass es zwischen zwei Knoten nicht mehr als eine Kante desselben Typs in dieselbe Richtung geben kann. Beschreibungen komplexer Konsistenz- und Anwendbarkeitsbedingungen werden von den algorithmischen Ansätzen schlecht unterstützt. Ihre Stärke liegt dafür bei den Ersetzungsregeln. So können z.B. Ersetzungsmuster mit Anweisungen wie „ X erhält alle eingehenden und ausgehenden Kanten von C “ formuliert werden, ohne dass diese Kanten explizit im Muster spezifiziert sind.

Algebraische Ansätze hingegen stammen aus der Kategorientheorie. Ihr Hauptaugenmerk liegt auf der deklarativen Beschreibung des Graphen vor und nach der Ersetzung. Bei den meisten dieser Ansätze sind Kan-

ten eigenständige Objekte, so dass die gerade beschriebene Schwäche der algorithmischen Ansätze hier nicht vorhanden ist. Des Weiteren sind sie gut geeignet für die Formulierung komplexer Konsistenz- und Anwendbarkeitsbedingungen, haben jedoch Schwächen bei der Ersetzung. Die linke Seite des Musters wird vollständig angegeben, weswegen keine Kantensmengen mit Kanten unbekanntem Typs oder unbekannter Kardinalität durch Ersetzungen manipuliert werden können.

Auch hier liegt es nahe, die Vorgehensweise von PROGRES zu verfolgen und beide Ansätze zu kombinieren.

Beispiel eines Vereinfachungsmusters

Betrachten wir als Nächstes beispielhaft ein bekanntes Vereinfachungsmuster von Spurgraphen, um damit erste Überlegungen im Hinblick auf die Spezifikation, aber auch Aufwands- und Risikoabschätzung eines zu implementierenden Graphtransformationssystems anstellen zu können.

In Abschnitt 1.2 auf Seite 12 ist folgendes komponentenunabhängiges Vereinfachungsmuster beschrieben:

Es gebe eine Operation O , die an mindestens einer Stelle in den Spurgraphen durch ein Prädikat P bedingt ist, an mindestens einer weiteren Stelle jedoch nicht. Überall, wo O direkt nach P auftritt, wird nun O vor P gezogen, so dass es an keiner Stelle mehr von P bedingt wird.

Exakt formuliert heißt das Muster der linken Seite (basierend auf einem extrahierten Spurgraph S nach Definition 2.5.3 auf Seite 34):

$$\begin{aligned}
 & o_1, o_2, p_1, p_2, \in \text{Operations}, \\
 & c_1, c_2 \in \text{Conditions_With_Uses}, \\
 & s \in \text{Starts} : \\
 & \quad (\exists w_1 \in \text{Walk_From_To}(S, s, o_1) \\
 & \quad \quad \exists(p_1, c_1) \in \text{Unconds} \\
 & \quad \quad \exists(c_1, p_1) \in \text{Uses} \\
 & \quad \quad \exists(c_1, o_1) \in \text{Trues} \cup \text{Falses} : p_1, c_1 \in w_1) \\
 & \quad \wedge (\exists w_2 \in \text{Walk_From_To}(S, s, o_2) : \\
 & \quad \quad (p_2, c_2) \in \text{Unconds} \\
 & \quad \quad \wedge (c_2, p_2) \in \text{Uses} \\
 & \quad \quad \wedge (c_2, o_2) \in \text{Trues} \cup \text{Falses} \Rightarrow p_2, c_2 \notin w_2) \\
 & \quad \wedge l(p_1) = l(p_2) \\
 & \quad \wedge l(o_1) = l(o_2).
 \end{aligned}$$

3 Ansätze zur Herleitung von Protokollen

Es wird klar, dass eine Sprache, die derlei Muster akzeptiert, einen erheblichen Entwicklungsaufwand erfordert, sowohl bei der Spezifikation als auch beim Entwurf, der Implementierung und dem Test. Die Entwicklung einer solchen Sprache würde den Rahmen einer Diplomarbeit sprengen.

Daher liegt es nahe zu versuchen, ein vorhandenes Graphtransformationssystem einzubinden.

Vorhandene Graphtransformationssysteme

Wie schon erwähnt gibt es nur wenige implementierte Graphtransformationssysteme, die die Anforderungen aus realen Anwendungsbereichen erfüllen, d.h. die mit großen Graphen und komplexen Graphtransformationen umgehen können. Für Muster der oben vorgestellten Art kommt neben PROGRES nur noch GenSet in Frage.

PROGRES

PROGRES ist, wie schon erwähnt, ein Werkzeug zur visuellen Programmierung. Dabei kann der Entwickler über eine integrierte graphische Oberfläche Graphtransformationen angeben. Die Ersetzungsmuster und die dabei entstehenden Graphen können von PROGRES in C-Code umgewandelt werden. Daher eignet sich PROGRES besonders gut für Rapid Prototyping.

Nun stellt sich die Frage, inwiefern PROGRES für diese Arbeit verwendet werden könnte. Ideal wäre es, wenn zunächst ein Spurgraph in PROGRES eingelesen, dann über eine Programmierschnittstelle transformiert und zuletzt wieder als Spurgraph gespeichert werden könnte.

Leider bietet PROGRES von sich aus keine Möglichkeit zur Anbindung an externe Programme. Es gibt zwar einen Ansatz, PROGRES an Corba³ anzubinden, jedoch ist diese Entwicklung erst im Anfangsstadium.

Darüber hinaus wäre der einzige gemeinsame Nenner zwischen Bauhaus und PROGRES in puncto Graph-Formate GXL, die Graph eXchange Language [GXL04]. Auf Bauhaus-Seite wurde in zeitlicher Überschneidung mit dieser Diplomarbeit im Rahmen der Diplomarbeit von Vran-

³Component Object Request Broker Architecture [COR04] – eine Architektur, mit deren Hilfe Programmierschnittstellen von Komponenten angesprochen werden können, die auf entfernten Rechnern liegen.

3.2 Hypothesengetriebener Ansatz

dečić eine XML-Anbindung für Bauhaus geschaffen [Vra03], mit deren Hilfe es problemlos möglich sein sollte, Bauhaus-Pakete zum Austausch via GXL zu schreiben. Jedoch war das End-Datum jener Arbeit nicht früh genug, um die Anbindung nutzen zu können. Außerdem hat auch die GXL-Anbindung von PROGRES erst Versuchs-Stadium erreicht und könnte nur mit erheblichem Aufwand und großen Risiken benutzt werden.

GenSet

GenSet [Fis03] [Gen04] ist ein Werkzeug, das es ermöglicht, Graphen aus verschiedenen Quellen zu integrieren und einfache Transformationen auszuführen. Primärer Anwendungsbereich ist die Analyse, Erzeugung und Herleitung von Architektur- und Entwurfsinformationen im Software-(Re)Engineering. GenSet verfolgt einen rein algebraischen Ansatz.

Wie oben beschrieben, sind algebraische Ansätze nicht gut für die Ersetzungen geeignet. Mit hoher Wahrscheinlichkeit könnte also nur die Mustersuche durch GenSet abgedeckt werden.

GenSet kann momentan GXL und RSF (Rigi Standard Format) ein- und auslesen. RFGs können ebenfalls in RSF konvertiert werden. Jedoch war zu Beginn dieser Diplomarbeit auch GenSet nur als einzelstehendes Werkzeug mit graphischer Oberfläche und ohne Anbindungsmöglichkeit für externe Programme vorhanden. Zeitgleich mit dieser Arbeit wurde eine Anbindungsmöglichkeit für externe Programme geschaffen, jedoch wäre nicht sicher gewesen, ob sie rechtzeitig ein Stadium erreicht, in dem sie ohne hohen Aufwand und größere Risiken hätte verwendet werden können.

3.2 Hypothesengetriebener Ansatz

Neben dem induktiven Ansatz gibt es noch eine weitere, bisher unerforschte Möglichkeit, Spurglyphen zur Herleitung von Komponentenprotokollen zu verwenden.

Der Wartungsingenieur wird in jedem Fall ein – evtl. sehr grobes – Verständnis von der untersuchten Komponente haben und damit auch Hypothesen über das Komponentenprotokoll. Dieses Verständnis kann er zunächst durch Betrachtung des Quellcodes der Komponente erweitern. Die Grundidee des hypothesengetriebenen Ansatzes ist es nun, seine Hypothese iterativ gegen die vorhandenen Spuren zu validieren. Dabei be-

kommt er u.a. Auskunft darüber, ob alle korrekten Spuren von seinem Protokoll akzeptiert werden, ob falsche Spuren akzeptiert werden und wie gut sein Protokollgraph von den Spuren überdeckt wird. Dadurch lernt der Wartungs-Ingenieur etwas Neues über das Komponentenprotokoll, kann seine Hypothese entsprechend anpassen und in einer weiteren Iteration erneut gegen die Spurgraphen validieren. Dieser Ansatz wird nun eingehender beleuchtet.

3.2.1 Korrekte und falsche Spuren

In [Hei00] ist das Problem beschrieben, Spuren, die die untersuchte Komponente korrekt benutzen, von solchen zu unterscheiden, die sie falsch benutzen. Dieses Problem ist bis heute ungelöst. Existiert jedoch Code von Testfällen für die untersuchte Komponente, dann können unter Umständen daraus Spuren gewonnen werden, die die Komponente definitiv korrekt bzw. definitiv falsch verwenden. Eine gute Auswahl an Testfällen testet nämlich sowohl das Verhalten der Komponente bei korrekter Benutzung als auch das Verhalten bei falscher Benutzung.

Validiert man den Protokollgraph gegen eine korrekte Spur, dann ist zunächst interessant, ob die Spur von dem Graph akzeptiert wird (Abschnitt 6.4 auf Seite 103 beschreibt einen Algorithmus zur Validierung gegen korrekte Spuren). Falls der Spurgraph akzeptiert wird, dann ist die Hypothese des Wartungs-Ingenieurs mindestens umfassend genug für diese Spur. Wird der Spurgraph nicht akzeptiert, dann ist der zuletzt akzeptierte Knoten im Spurgraph von Interesse und der damit korrespondierende Knoten des Protokollgraphen. Mit Hilfe dieser Informationen und Gravis kann der Wartungsingenieur die Stelle im Protokoll und in der Spur untersuchen und damit einschätzen, wie und warum sein Protokollgraph angepasst werden muss. Natürlich besteht auch die Möglichkeit, dass die Hypothese über das Protokoll korrekt, die vermeintlich korrekte Spur jedoch falsch ist. Durch die Informationen, die der Wartungs-Ingenieur hier bekommt, wird er gezielt auf solche Stellen aufmerksam gemacht und hat damit eine Chance, falsche Spuren zu entdecken.

Bei der Validierung gegen Spuren, die definitiv falsch sind, ist wieder zunächst interessant, ob der Protokollgraph die Spur akzeptiert. Ist dies der Fall, dann sollte derjenige Knoten des Protokollgraphen in Augenschein genommen werden, der bei der Validierung mit dem falschen Knoten des Spurgraphen assoziiert wurde. Dazu muss selbstverständlich der falsche Knoten im Spurgraph als falsch markiert sein.

3.2 Hypothesengetriebener Ansatz

Betrachten wir nun ein Beispiel: Gegeben sei eine Komponente mit den primitiven Operationen A, B, C, D und E. Weiterhin seien die Spurgraphen aus Abbildung 3.5 gegeben. Spur 1 bis Spur 3 sind dabei korrekt, Spur 4 ist falsch. Der falsche Knoten in Spur 4 ist gekennzeichnet (gestrichelte Umrandung).

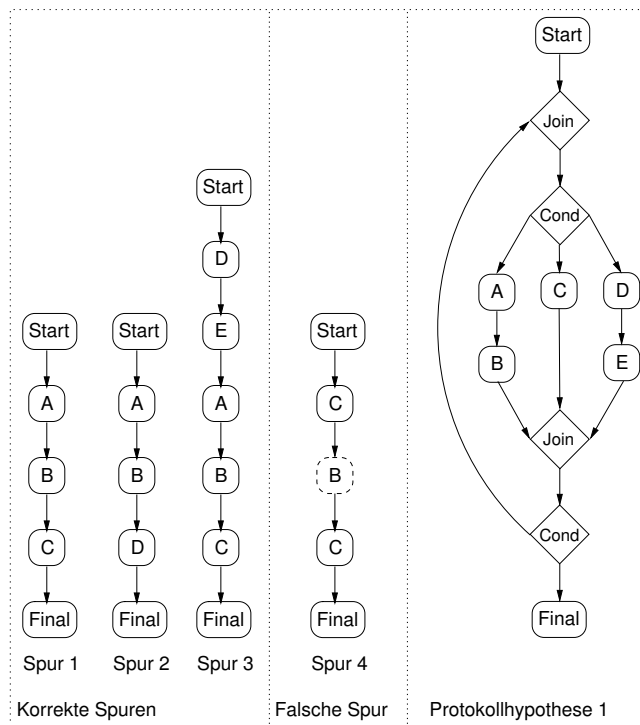


Abbildung 3.5: Extrahierte Spurgraphen und Protokollhypothese 1

Die erste Hypothese des Wartungs-Ingenieurs über das Komponentenprotokoll ist rechts in Abbildung 3.5 zu sehen⁴. Die Sequenzen (A, B) , (C) und (D, E) dürfen also in beliebiger Reihenfolge und beliebig oft durchlaufen werden. Allerdings muss mindestens eine der Sequenzen durchlaufen werden.

Nun wird Spur 1 validiert. Diese wird vollständig vom Protokollgraph akzeptiert. Bei Spur 2 verhält es sich anders: Sie hat eine Kante von D zum

⁴Die *Condition*-Knoten sind nicht-deterministisch und dürfen beliebig viele ausgehende *Uncond*-Kanten besitzen. Eine ausführliche Diskussion von nicht-deterministischen *Condition*-Knoten folgt in Abschnitt 4.4 auf Seite 79

3 Ansätze zur Herleitung von Protokollen

Endknoten. Laut Protokoll-Hypothese muss jedoch nach D immer E aufgerufen werden. Der Wartungs-Ingenieur muss nun also überprüfen, ob er seine Hypothese anpasst und wenn ja, auf welche Art. Auf Anhieb erscheinen zwei plausible Alternativen, die in Abbildung 3.6 zu sehen sind⁵. In Variante (a) wurde als weitere erlaubte Sequenz in der Schleife D hinzugefügt. Damit wird Spur 2 akzeptiert, es bleibt jedoch die Einschränkung erhalten, dass E immer nur nach D aufgerufen werden darf. Diese Einschränkung existiert in Variante (b) nicht mehr. Hier dürfen nun D und E unabhängig voneinander beliebig oft aufgerufen werden.

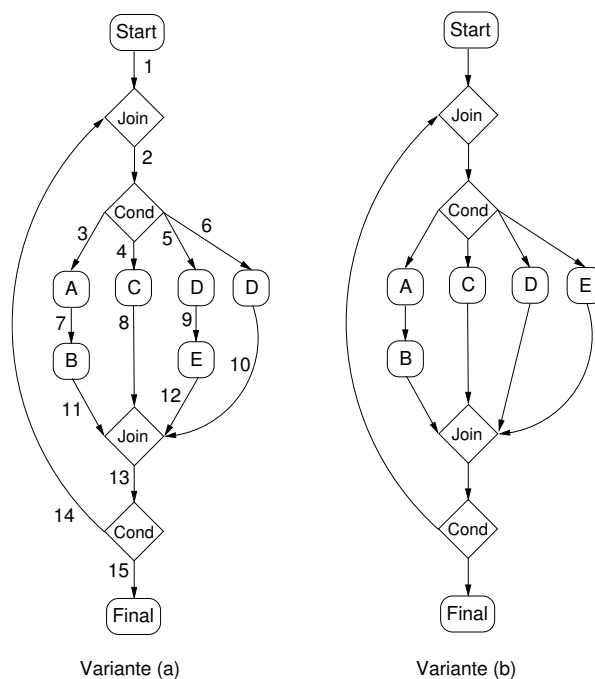


Abbildung 3.6: Protokollhypothese 2

Der Wartungs-Ingenieur kann nun aus verschiedene Quellen Informationen heranziehen, auf deren Basis er eine Alternative auswählt. Zum einen kann er sich über Gravis direkt den Quellcode der Spur ansehen. Eine weitere Option ist die Bestimmung der obligatorischen und verbotenen Vorgänger und Nachfolger, die in Unterabschnitt 3.2.3 auf Seite 62 beschrieben wird.

⁵Die Nummerierung der Kanten gehört nur zur Darstellung der Graphen, nicht zum Graph selbst. Sie spielt erst im nächsten Abschnitt eine Rolle.

Bei der Formulierung von Protokollhypothesen erscheint die Regel „Lieber zu restriktiv als zu freigiebig“ sinnvoll. Denn unnötige Restriktionen werden – bei ausreichend vorhandenen nicht-trivialen, korrekten Spuren – früher oder später bei der Validierung erkannt. Wählen wir daher in unserem Beispiel Variante (a) als Protokollhypothese der zweiten Iteration aus.

Spur 3 wird problemlos akzeptiert.

Nun wird der Protokollgraph gegen die falsche Spur 4 validiert. Der erste Knoten (C) wird akzeptiert, der zweite – falsche – Knoten (B) jedoch nicht. Der Protokollgraph ist also in Bezug auf Spur 4 korrekt, weil er den falschen Knoten nicht akzeptiert.

Damit ist der erste Teil der Validierung abgeschlossen.

3.2.2 Überdeckung

Ein weitere interessante Frage ist: Wieviel von der Protokollhypothese wird durch die korrekten Spuren abgedeckt? Bei der bisher beschriebenen Validierung gegen korrekte Spuren wurden im vorliegenden Beispiel lediglich unnötige Restriktionen aus dem Protokollgraph entfernt. Sollte der Graph jedoch zu allgemein formuliert sein (Extrembeispiel wäre ein Graph, der alle möglichen Sequenzen der Operationen zulässt), dann kann dies in vielen Fällen nicht bei der Validierung gegen korrekte, sondern allenfalls bei der Validierung gegen falsche Spuren aufgedeckt werden. Die Überdeckung ist eine weitere Möglichkeit, Stellen im Protokollgraph aufzudecken, die zu allgemein formuliert sind.

Dabei wäre insbesondere die Pfadüberdeckung interessant. Würden alle Pfade eines Protokollgraphen durch die Spuren abgedeckt, dann hätte man den idealen Protokollgraphen in Bezug auf die Spuren. Jedoch ist es selbst bei trivialen Beispielen wie dem momentan vorliegenden nicht möglich, eine vollständige Pfadüberdeckung zu erreichen, weil der Graph aufgrund von Schleifen unendlich viele Pfade enthält.

Begnügen wir uns daher mit der Zweigüberdeckung⁶. In Tabelle 3.1 auf der nächsten Seite ist zu sehen, welche Kanten im Protokollgraph von

⁶Sicher wäre es möglich, etwas zwischen Zweig- und Pfadüberdeckung zu erreichen, z.B. indem man alle schleifenfreien Pfade einmal durchläuft und dann exemplarisch jede Schleife n -mal. Dieser Ansatz wird hier jedoch nicht weiterverfolgt.

3 Ansätze zur Herleitung von Protokollen

welcher der korrekten Spuren überdeckt werden. Wie man sofort sieht, erreichen schon Spur 2 und Spur 3 eine vollständige Zweigüberdeckung des Protokollgraphen.

Spur	Überdeckte Kanten
Spur 1	1, 2, 3, 4, 7, 8,11,13,14,15
Spur 2	1, 2, 3, 6, 7, 10,11,13,14,15
Spur 3	1, 2, 3, 4, 5, 7,8,9,11,12,13,14,15

Tabelle 3.1: Zweigüberdeckung durch korrekte Spuren

3.2.3 Obligatorische und verbotene Vorgänger und Nachfolger

Motivation

Der Protokollgraph erfüllt zwar bisher alle Kriterien, d.h. er akzeptiert alle korrekten Spuren, wird von ihnen vollständig überdeckt und akzeptiert den fehlerhaften Knoten aus der falschen Spur nicht, trotzdem enthalten die Spuren noch weitere Informationen, die nicht im Protokollgraph berücksichtigt sind: In allen korrekten Spuren erscheint C immer nur nach A und B . Im Protokollgraph ist es jedoch auch erlaubt, C direkt auszuführen.

Beschreibung

Um Informationen dieser Art aus den Spuren zu gewinnen, wird für jede Operation o folgende Untersuchung angestellt: Welche Operationen werden in allen Spuren mindestens vor o ausgeführt? Diese Operationen nennen wir die *obligatorischen Vorgänger*⁷ von o .

Umgekehrt werden alle Operationen gesucht, die in keiner Spur vor o ausgeführt werden. Diese werden ab jetzt *verbotene Vorgänger* von o genannt.

Zuguterletzt werden noch diejenigen Operationen bestimmt, die in allen Spuren immer nach o ausgeführt werden. Der Name hierfür ist *obligatorische Nachfolger*.

⁷Der korrekte Ausdruck wäre eigentlich *obligatorische transitive Vorgänger*, da Vorgänger ja als unmittelbar durch eine eingehende Kante verbunden definiert sind, hier jedoch alle Knoten gemeint sind, die auf einem Weg vom Startknoten zum aktuellen Knoten liegen. Um jedoch die Sätze verständlich zu halten, wird diese sprachliche Ungenauigkeit in Kauf genommen.

Verbotene Nachfolger müssen nicht bestimmt werden, da sie sich direkt aus den verbotenen Vorgängern ergeben. Denn wenn es keinen Weg vom Startknoten zu o gibt, auf dem p liegt, dann kann es, da Wege als Teile von Pfaden definiert sind und ein Pfad immer vom Start- zum Endknoten führt, auch keinen Weg von p zum Endknoten geben, auf dem o liegt (siehe Abbildung 3.7).

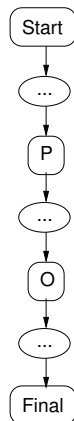


Abbildung 3.7: $Verb._Vorgänger(O, P) \Leftrightarrow Verb._Nachfolger(P, O)$

Besonderheit: Prädikate

Eine Besonderheit bei diesen Untersuchungen bilden Prädikate (Bedingungen), die obligatorische Vorgänger für einen Knoten o sind. Tritt ein solcher Fall auf, dann ist natürlich nicht nur interessant, um welche Bedingung es sich handelt, sondern auch, ob die Bedingung wahr oder falsch sein muss. Auf diese Art würde man z.B. herausfinden, dass ein $pop()$ -Aufruf auf einem Stapel nur ausgeführt werden darf, wenn zuvor mittels $isEmpty()$ abgefragt wurde, ob Elemente auf dem Stapel vorhanden sind.

In Unterabschnitt 2.4.6 auf Seite 30 wurde beschrieben, wie Bedingungen in Spurgraphen modelliert sind. Dabei wurde auch gesagt, dass es in Spurgraphen, wie sie in dieser Arbeit verwendet werden, keine Repräsentation für Ausdrücke gibt, d.h. im gerade genannten Beispiel könnte nicht unterschieden werden, ob das Prädikat $isEmpty()$ oder $!isEmpty()$ lautet. Die einzige Aussage, die über den Wahrheitswert gemacht werden kann, lässt sich aus dem Kantentyp der ausgehenden Sequenz-Kante ($True$ oder $False$) ableiten. Damit diese Information von Nutzen ist, muss

3 Ansätze zur Herleitung von Protokollen

der Wartungs-Ingenieur jedoch manuell herausfinden, wie die Bedingung tatsächlich aussieht. Um den Wartungs-Ingenieur auf solche Fälle hinzuweisen und um den Grundstein für Untersuchungen zu legen, die gemacht werden können, sobald Spurgraphen eine detailliertere Repräsentation für Prädikate enthalten, werden obligatorische Vorgängerprädikate als solche gekennzeichnet und es wird angegeben, ob die ausgehende Sequenz-Kante des Prädikats eine *True*- oder eine *False*-Kante ist.

Erläuterung am Beispiel

Betrachten wir nun wieder unser Beispiel. Tabelle 3.2 stellt die obligatorischen Vorgänger aller Routinen von K dar. Dabei werden senkrecht die Routinen abgetragen, waagrecht erhält jeder ihrer obligatorischen Vorgänger ein X.

	A	B	C	D	E
A	–	–	–	–	–
B	x	–	–	–	–
C	x	x	–	–	–
D	–	–	–	–	–
E	–	–	–	x	–

Tabelle 3.2: Obligatorische Vorgänger

Daraus wird ersichtlich, dass es für drei Knoten obligatorische Vorgänger gibt. Zwei davon sind schon im Protokollgraph berücksichtigt: Vor B muss immer A ausgeführt werden und vor E immer D . Die dritte Abhängigkeit ist die oben angesprochene: C wird immer nur aufgerufen, nachdem A und B aufgerufen wurden. Der Wartungs-Ingenieur kann nun wiederum seine Hypothese überprüfen und diese Restriktion ggfs. in den Protokollgraph einbauen, z.B. indem er die eingehende Kante von C durch A , B und entsprechende Kanten ersetzt, siehe Abbildung 3.8 auf der nächsten Seite.

In Tabelle 3.3 auf der nächsten Seite sind die verbotenen Vorgänger dargestellt. Wiederum sind senkrecht die Routinen abgetragen und waagrecht bei allen verbotenen Vorgängern ein X eingetragen. Sie sind hier nur der Vollständigkeit halber aufgeführt. Bei diesem Beispiel macht es aufgrund der wenigen und kurzen Spuren keinen Sinn, die verbotenen Vorgänger mit einzubeziehen.

Zuguterletzt seien noch die obligatorischen Nachfolger des Beispiels gezeigt: Tabelle 3.4 auf Seite 66. Es ist zu sehen, dass nur zwei Routinen

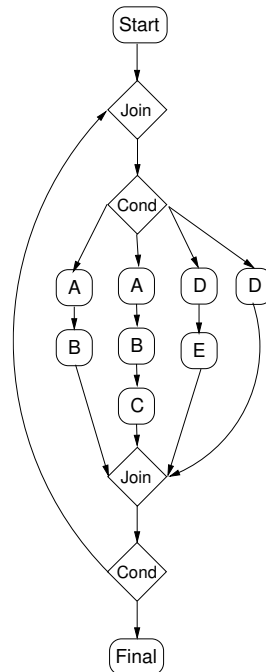


Abbildung 3.8: Protokollhypothese 3

	A	B	C	D	E
A	x	x	x	–	–
B	–	x	x	–	–
C	–	–	x	–	–
D	–	–	x	x	x
E	x	x	x	–	x

Tabelle 3.3: Verbotene Vorgänger

obligatorische Nachfolger besitzen: *A* und *E*. Der obligatorische Nachfolger von *A* ist schon im Protokollgraph berücksichtigt, da in allen Wegen, in denen *A* vorkommt, automatisch anschließend *B* vorkommt. Anders ist es bei den obligatorischen Nachfolgern von *E*. Wenn sich der Wartungs-Ingenieur davon überzeugt hat, dass die Protokollhypothese aufgrund dieser Information überarbeitet werden muss, dann kann er z.B. in den Zweig, in dem *E* auftritt, die Knoten *A*, *B* und *C* einfügen (siehe Abbildung 3.9 auf der nächsten Seite).

3 Ansätze zur Herleitung von Protokollen

	A	B	C	D	E
A	-	x	-	-	-
B	-	-	-	-	-
C	-	-	-	-	-
D	-	-	-	-	-
E	x	x	x	-	-

Tabelle 3.4: Obligatorische Nachfolger

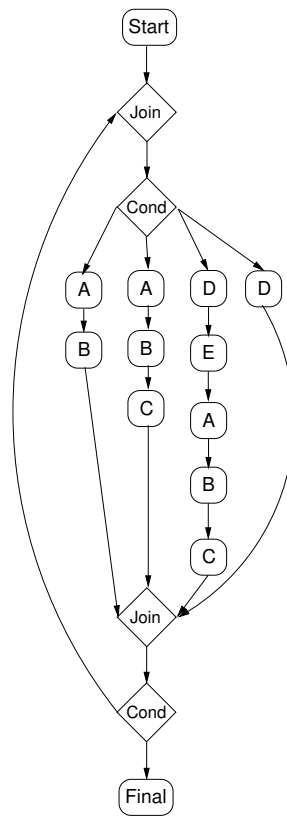


Abbildung 3.9: Protokollhypothese 4

Nutzen und Erweiterungsmöglichkeiten

Über den tatsächliche Nutzen von obligatorischen und verbotenen Vorgängern und Nachfolgern können selbstverständlich erst nach einer Fallstudie mit realem Code Aussagen gemacht werden.

Die hier vorgestellten Überlegungen könnten u.a. durch das Verfahren von Whaley, Martin und Lam [Wha02] erweitert werden, das in Abschnitt 2.7 auf Seite 43 umrissen ist.

3.2.4 Prozess

In diesem Abschnitt wird der Prozess beschrieben, der dem hypothesengetriebenen Ansatz zugrunde liegt. Zur Beschreibung des Prozesses werden IDEF0-Diagramme verwendet. Für eine kurze Einführung in IDEF0 siehe Anhang B auf Seite 121.

Der Prozess gibt zugleich Aufschluss über die Werkzeuge, die entwickelt werden müssen, wenn der hypothesengetriebene Ansatz umgesetzt werden soll.

Zu Beginn wird davon ausgegangen, dass die zu untersuchende Komponente samt ihren primitiven Operationen bekannt ist und dass aus vorhandenem Code Spurgraphen für diese Komponente extrahiert wurden.

Abbildung 3.10 auf der nächsten Seite stellt das Diagramm PH/A1 dar, welches die erste Aktivität des Prozesses beschreibt: Die Vorbereitung der Spurgraphen.

Die Vorbereitung enthält drei Unter-Aktivitäten:

- In Unter-Aktivität 1 werden die Spurgraphen mit Hilfe eines Werkzeugs bereinigt. Das Spurbereinigungswerkzeug benötigt dabei die primitiven Operationen der Komponente. Es entfernt überflüssige Knoten und kennzeichnet irrelevante. Eine genaue Beschreibung ist in Kapitel 4 auf Seite 75 nachzulesen. Die Spurbereinigung verändert dabei die Semantik des Protokollgraphen in Bezug auf die sequenziellen Abhängigkeiten der Operationen nicht.
- Unter-Aktivität 2 erkennt und entfernt rekursive Aufrufe. Wie wir in Kapitel 5 auf Seite 81 sehen werden, wird dabei die Semantik des Protokollgraphen verändert. Für eine genauere Diskussion siehe dort.

3 Ansätze zur Herleitung von Protokollen

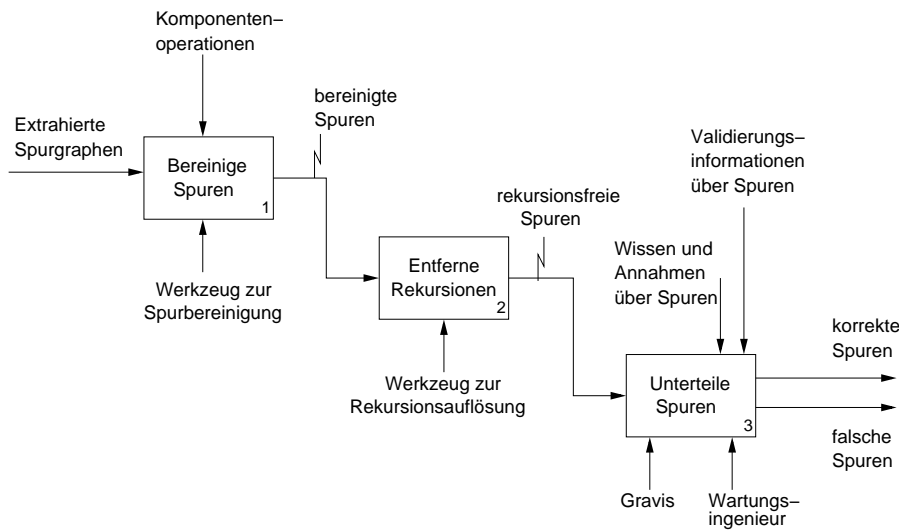


Abbildung 3.10: Prozess des hypothesengetriebenen Ansatzes – IDEF0-Diagramm PH/A1

- Unter-Aktivität 3 führt der Wartungs-Ingenieur mit Hilfe von Gravis aus. Dabei trennt er korrekte von falschen Spuren nach dem Wissen oder den Annahmen, die er darüber hat.

Diese drei Unter-Aktivitäten sind in der ersten Aktivität des Diagramms PH/A0 (Abbildung 3.11 auf der nächsten Seite) zusammengefasst.

Aktivität 2 von PH/A0 beinhaltet die Erstellung eines Protokollgraphen nach der Ausgangshypothese des Wartungs-Ingenieurs. Dazu setzt er in Gravis den Graphen zusammen.

In Aktivität 3 wird die Validierung des Protokollgraphen gegen die Spuren vorgenommen. Dabei kommt alles zum Tragen, was in diesem Abschnitt bisher beschrieben wurde: Validierung gegen korrekte und falsche Spuren, Bestimmung der Überdeckung durch die korrekten Spuren, Bestimmung der obligatorischen und verbotenen Vorgänger und Nachfolger, und zuletzt Analyse der Spuren und des Quellcodes durch den Wartungs-Ingenieur. Werden bei der Validierung Schwächen oder Fehler aufgedeckt, dann kann der Wartungs-Ingenieur nun entweder seine Hypothese und damit den Protokollgraph überarbeiten (Aktivität 4) oder er unterteilt die Spuren aufs Neue. Sind die Ergebnisse durchweg positiv, dann ist der Prozess an dieser Stelle beendet und hat als Ausgabe einen validierten Protokollgraphen.

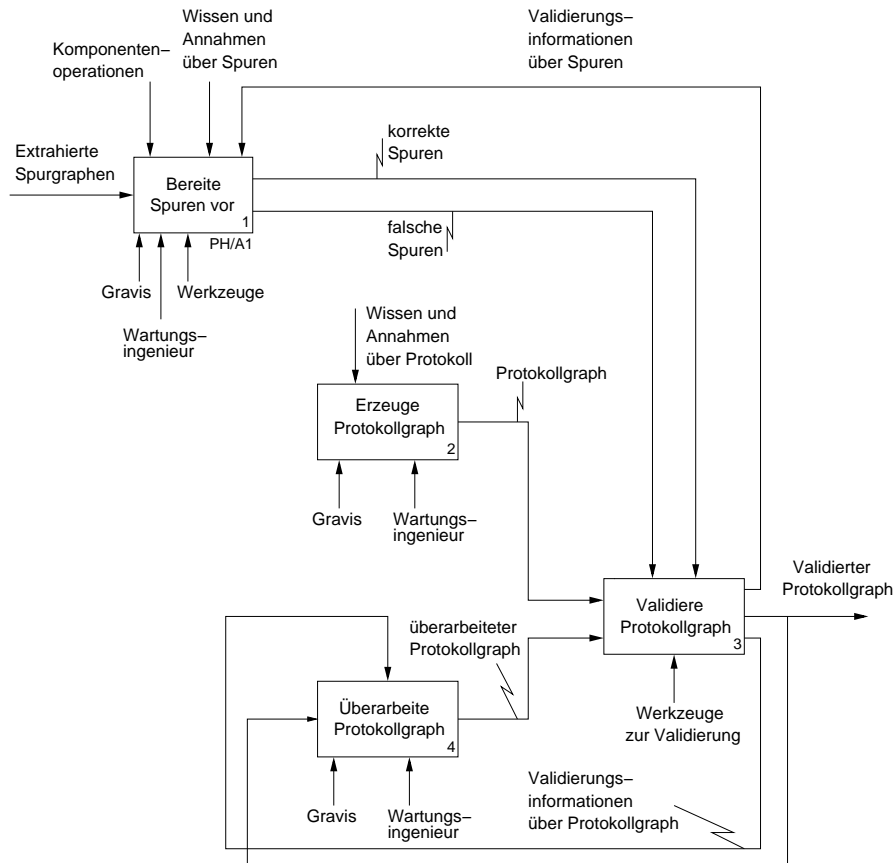


Abbildung 3.11: Prozess des hypothesengetriebenen Ansatzes – IDEF0-Diagramm PH/A0

3.3 Auswahl des Ansatzes

Nachdem nun zwei mögliche Ansätze beschrieben wurden, die als Ziel haben, mit Hilfe von Werkzeugen aus Spuren einen Protokollgraphen herzuleiten, stellt sich die Frage, welcher der beiden Ansätze umgesetzt und evaluiert werden soll. Dabei werden fünf Kriterien aufgestellt, anhand derer die beiden Ansätze verglichen werden.

3.3.1 Größe und Lesbarkeit des Protokollgraphen

Der induktive Ansatz leitet zunächst aus den gegebenen Spuren ein Protokoll her, welches automatisch minimiert wird. Es gibt bisher noch keine Evaluierung der automatischen Minimierung anhand von realem Code, allerdings habe ich die Vermutung, dass die automatisch erfassbaren Fälle in der Realität nur selten auftauchen, bzw. die Zahl der eingesparten Knoten nicht sehr groß ist. Daher schätze ich die Chancen eher gering ein, einen induktiv erzeugten Protokollgraphen, der aus vielen und/oder komplexen Spuren hergeleitet wurde, so weit zu minimieren, dass er für Menschen leicht verständlich ist.

Beim hypothesengetriebenen Ansatz hingegen wird der Wartungs-Ingenieur mit einem verhältnismäßig kompakten Protokollgraph beginnen. Durch nicht-deterministische *Condition*-Knoten ist es möglich, von komplexen Strukturen auf einfache Formen zu abstrahieren. So wird es z.B. meiner Vermutung nach häufig möglich sein, eine oder mehrere Entscheidungsschleifen einzubauen, wie schon in [Hei00] beschrieben und im Beispiel aus dem letzten Abschnitt gezeigt. Dabei gibt es jeweils einen nicht-deterministischen *Condition*-Knoten, von dem aus Wege zu einem *Join*-Knoten modelliert sind, die Operationen mit sequenziellen Abhängigkeiten enthalten. Von diesem *Join*-Knoten führt eine Kante zum *Condition*-Knoten zurück, so dass diese Wege beliebig oft und in beliebiger Reihenfolge durchlaufen werden können. Mit Hilfe solcher Ausdrucksmöglichkeiten werden abstrakte Hypothesen im Protokollgraph umgesetzt, die ihn von vornherein klein und verständlich halten. Er wird im Laufe der Validierung zwar wachsen, jedoch höchstwahrscheinlich nicht auf die Größe eines induktiv erzeugten Graphen. Es ist jedoch anzumerken, dass dabei die Gefahr besteht, das Protokoll zu abstrakt zu formulieren und damit unerlaubte Sequenzen zuzulassen. Diese Gefahr kann zwar durch die Überdeckung reduziert, jedoch nicht völlig eliminiert werden.

3.3.2 Unterstützung des Wartungs-Ingenieurs

Spurgraphen, die aus realem Code erzeugt wurden, sind häufig sehr komplex. Ein wichtiger Aspekt der Werkzeuge ist es, den Wartungs-Ingenieur im Umgang mit der Fülle von Informationen zu unterstützen.

Beim induktiven Ansatz werden nach der automatischen Minimierung Vereinfachungsmuster angewendet, deren Vorbedingungen der Wartungs-Ingenieur bestätigen muss. Dazu muss er die entsprechenden Spurgraphen und ggfs. den Quellcode analysieren. Als Nächstes ist es die Aufgabe des Wartungs-Ingenieurs, Vereinfachungsmuster zu finden, die typisch für diese Komponente sind. Diese spezifiziert er dann und stößt ein Graphtransformationssystem an, das die Vereinfachung durchführt. Um solche Vereinfachungsmuster zu finden, muss der Wartungs-Ingenieur jedoch ohne jegliche Werkzeugunterstützung (mit Ausnahme von Gravis) die Spurgraphen und den Quellcode durchsuchen. Dabei beschäftigt er sich nicht direkt mit sequenziellen Abhängigkeiten, sondern er sucht Beziehungen zwischen den Operationen, die Vereinfachungen ermöglichen. Damit wird die ursprüngliche Aufgabe des Wartungs-Ingenieurs, das Protokoll einer Komponente manuell aus dem Quellcode und Spurgraphen zu extrahieren auf das Problem verlagert, manuell aus dem Quellcode und Spurgraphen Beziehungen zur Vereinfachung der Spurgraphen zu finden.

Der hypothesengetriebene Ansatz hingegen unterstützt den Wartungs-Ingenieur unmittelbar bei der Herleitung des Protokolls.

3.3.3 Benötigtes Vorwissen

Der induktive Ansatz verlangt in den ersten beiden Schritten (induktive Erzeugung und automatische Minimierung der Protokollgraphen) keinerlei Vorwissen vom Wartungs-Ingenieur über die sequenziellen Abhängigkeiten der Komponente. Der Wartungs-Ingenieur hat dabei schon am Ende dieser beiden Schritte ein Protokoll, das zur Validierung von Spuren verwendet werden kann. Erst bei der Anwendung komponentenunabhängiger und komponentenabhängiger Vereinfachungsmuster muss er Annahmen oder Wissen über sequenzielle Abhängigkeiten besitzen.

Der hypothesengetriebene Ansatz hingegen setzt voraus, dass der Wartungs-Ingenieur auf eine beliebige Weise (z.B. durch manuelle Analyse des Quellcodes der Komponente) Wissen und Annahmen über die Komponen-

3 Ansätze zur Herleitung von Protokollen

te und insbesondere über sequenzielle Abhängigkeiten ihrer Operationen gesammelt hat. Lediglich durch das Bestimmen der obligatorischen und verbotenen Vorgänger besteht die Möglichkeit, initiale Hypothesen aus den Spuren zu gewinnen. Diese Hypothesen müssen jedoch erst gegen die Spuren validiert werden, bevor der erste Protokollgraph entsteht, der zur Validierung von Spuren verwendet werden kann.

3.3.4 Lernprozess

Wie schon in der Einleitung skizziert, muss der Wartungs-Ingenieur die Komponenten der untersuchten Software verstehen lernen, um seine Aufgabe erledigen zu können. Daher ist es sinnvoll, wenn er bei der Herleitung des Protokolls zugleich ein Verständnis desselben entwickelt. Die beiden Ansätze unterstützen den Lernprozess des Wartungs-Ingenieurs auf verschiedene Art und Weise.

Der induktive Ansatz bietet wenig Möglichkeiten, das Verständnis des Wartungs-Ingenieurs vom Protokoll zu formulieren und in den Prozess einzubringen. Lediglich im zweiten Schritt, in dem der Wartungs-Ingenieur die Vorbedingungen von allgemeinen Vereinfachungsmustern bestätigen muss, wird er angehalten, einzelne, überschaubare Aspekte zu betrachten, ein Verständnis davon zu entwickeln und dieses als Eingabe für den Prozess zu verwenden.

Beim hypothesengetriebenen Ansatz hat der Wartungs-Ingenieur hingegen die Möglichkeit, Schritt für Schritt etwas Neues zu lernen. In jeder Iteration bekommt er von den Validierungswerkzeugen jeweils einen Einzelaspekt vermittelt, den er dann betrachtet und in Bezug zum Rest seiner Hypothese bringen kann. Auf diese Art kann er sein Verständnis des Protokolls stückchenweise verfeinern und korrigieren. Er hat dabei die Möglichkeit, seine Hypothese explizit als Protokollgraph zu formulieren und kann somit jeden Lernschritt speichern und – sollte er eine Fehlentscheidung bemerken – zu einem früheren Schritt zurückkehren. Darüber hinaus kann er wählen, wann er gegen korrekte, wann gegen falsche Spuren validiert, wann er sich die Überdeckung berechnen lässt und wann er die Information der obligatorischen und verbotenen Vorgänger einbringt. Eine solche Wahlmöglichkeit erlaubt es, Lernprozesse nach eigenen Wünschen und Neigungen zu entwickeln, was die Akzeptanz der Werkzeuge und die Effizienz des Ausführenden steigert.

3.3.5 Weitere Anwendungsbereiche

Bei beiden Ansätzen entstehen Werkzeuge, die auch in verwandten Bereichen zum Einsatz kommen können.

Ein Graphtransformationssystem, wie es beim induktiven Ansatz erstellt werden muss, findet im gesamten Bauhaus-Projekt Anwendungsmöglichkeiten.

Nachdem beim hypothesengetriebenen Ansatz die Techniken aus der Validierung verwendet werden, können folglich mit den Werkzeugen, die bei diesem Ansatz entwickelt werden, alle Bereiche abgedeckt werden, die mit der Validierung zu tun haben (siehe Kapitel 1 auf Seite 7): Zum Einen kann mit Hilfe der Überdeckung Testcode evaluiert werden. Wenn ein Protokollgraph nur schlecht von Testcode überdeckt wird, sind die Tests offensichtlich noch nicht ausführlich genug. Darüber hinaus kann der Algorithmus zur Validierung von Protokollen gegen Spuren auch umgekehrt verwendet werden. Dies findet zunächst schon innerhalb des beschriebenen Prozesses statt, denn der Wartungs-Ingenieur kann bei jeder Spur, die nicht vom Protokoll akzeptiert wird, anhand der exakten Information, welche Stelle nicht akzeptiert wird, entscheiden, ob er seine Hypothese anpasst oder ob die Spur möglicherweise falsch ist. Des Weiteren kann neu geschriebener Code durch das Validierungswerkzeug gegen den Protokollgraph getestet werden.

3.3.6 Risiken in der Durchführung

Da eine Diplomarbeit einen definierten zeitlichen Rahmen einhalten muss, sind Risiken in der Durchführung bei der Auswahl des Ansatzes ein nicht von der Hand zu weisender Faktor.

In Abschnitt 3.1 auf Seite 47 wurde herausgearbeitet, dass die Weiterführung des induktiven Ansatzes, wie sie in der Ausschreibung gefordert ist, an verschiedenen Stellen große Risiken birgt. Zunächst ist nicht klar, ob es überhaupt weitere allgemeingültige Muster gibt. Dasselbe gilt für komponentenabhängige Muster. Von beiden muss jedoch eine gewisse Anzahl gefunden werden, um das Graphtransformationssystem spezifizieren zu können. Eine textuelle und graphische Repräsentation, die jeweils alle Teile eines Graphersetzungsmusters umfasst, ist sehr komplex, wenn überhaupt machbar. Ein solches System zu implementieren, sprengt den

3 Ansätze zur Herleitung von Protokollen

Rahmen einer Diplomarbeit. Die beiden einzigen vorhandenen implementierten Graphtransformationssysteme bergen Risiken, weil nicht sicher ist, dass sie an Bauhaus in irgendeiner Form angebunden werden können. Abgesehen davon ist es aus Sicht der Benutzerfreundlichkeit unzumutbar, ein weiteres komplexes Werkzeug in den Prozess einzubinden, mit dem der Wartungs-Ingenieur lernen muss, umzugehen. Zuguterletzt müssen die Spuren bereinigt und von Rekursionen befreit werden, was zusätzlichen Aufwand kostet.

Letzteres müssen sie beim hypothesengetriebenen Ansatz zwar auch, jedoch beruht der Hauptteil dieses Ansatzes (Validierung gegen korrekte und falsche Spuren) auf wohlbekanntem Algorithmen, wie wir in Kapitel 6 auf Seite 99 sehen werden und ist somit frei von größeren Risiken.

3.3.7 Auswahl

Nach sorgfältigem Abwägen der Vor-, Nachteile und Risiken beider Ansätze, habe ich mich – im Einverständnis mit meinen Betreuern – dazu entschieden, den hypothesengetriebenen Ansatz umzusetzen.

4 Spurbereinigung

In Abschnitt 2.4 auf Seite 22 wurden extrahierte Spurgraphen beschrieben und in Abschnitt 2.5 auf Seite 33 definiert. Dabei wurde klar, dass Spurgraphen irrelevante Information enthalten, so dass eine Bereinigung der Spurgraphen notwendig ist.

Dieses Kapitel beschreibt die Umwandlung von extrahierten Spurgraphen in bereinigte Spurgraphen. Dabei werden überflüssige Knoten entfernt und nicht-deterministische *Condition*-Knoten in eine einheitliche Form gebracht. Es wird darauf geachtet, die Semantik des Spurgraphen in Bezug auf die sequenziellen Abhängigkeiten der Komponentenoperationen nicht zu verändern.

Am Ende des Kapitels wird, basierend auf der Definition von *extrahierten Spurgraphen*, die Definition von *bereinigten Spurgraphen* angegeben.

4.1 Anfang und Ende

Begin_Of_Lifetime- und *End_Of_Lifetime*-Knoten geben Beginn und Ende der Lebensdauer eines Stapelobjekts an. Diese Information ist jedoch schon durch die *Start*- und *Final*-Knoten eindeutig bestimmt. Daher sind sie überflüssig und werden bei der Spurbereinigung gelöscht.

Beim Löschen eines solchen Knotens wird seine eingehende Sequenz-Kante mit seinem Nachfolger verbunden. Anschließend wird der Knoten und seine ausgehende *Unconditional*-Kante gelöscht.

4.2 Routinen und Routinenaufrufe

Unterabschnitt 2.4.4 auf Seite 24 beschreibt den Unterschied zwischen Benutzer- und Komponentenroutinen und Unterabschnitt 2.4.5 auf Seite 24 erklärt die drei Fälle, die bei Routinenaufrufen vorkommen können.

4 Spurbereinigung

Der folgende Abschnitt beschreibt den Teil der Spurbereinigung, der sich mit Routinen und Routinenaufrufen beschäftigt.

4.2.1 Fall 1

Fall-1-Routinen sind diejenigen Routinen, deren Rumpf vollständig im Spurgraph enthalten sind.

Komponentenroutinen

Komponentenroutinen werden als atomar angesehen, deswegen ist der Rumpf irrelevant. Der Routinenrumpf wird daher entfernt. Damit wird der Fall-1-Aufruf in einen Fall-3-Aufruf (d.h. ein Aufruf einer Routine, deren Rumpf nicht im Spurgraph enthalten ist) umgewandelt.

Benutzerrountinen

Von Benutzerrountinen sind im Sinne der Sequenzen eines Spurgraphen nur die Rümpfe wichtig, nicht aber die Aufrufe. Daher liegt der Gedanke nahe, die *Call*-, *Entry*- und *Return*-Knoten von Benutzerrountinen zu entfernen.

Allerdings verringert sich die Knotenzahl des Graphen, wenn eine Benutzerrountine mehrfach aufgerufen wird (siehe Unterabschnitt 3.1.1 auf Seite 48). Aus diesem Grund werden nur bei solchen Routinen die gerade genannten Knoten entfernt, die nur einmal aufgerufen werden.

Vor dem Löschen der Knoten einer einfach aufgerufenen Benutzerrountine wird die eingehende Sequenzkante des *Call*-Knotens mit dem Nachfolger des *Entry*-Knotens verbunden. Die eingehende Sequenzkante des *Return*-Knotens wird mit demjenigen Nachfolger des *Call*-Knotens verbunden, der die ausgehende *Unconditional*-Kante des *Call*-Knotens als eingehende Kante hat. Anschließend werden *Call*-, *Entry*- und *Return*-Knoten samt aller ein- und ausgehenden Kanten gelöscht. In Abbildung 4.1 auf der nächsten Seite ist die Löschoperation dargestellt. Die Nummern der Kanten vermitteln, welche Kante zum Verbinden der nicht gelöschten Knoten verwendet werden.

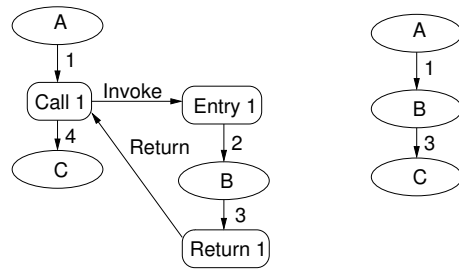


Abbildung 4.1: Ersetzen eines Aufrufs durch Routinenrumpf (Fall 1)

4.2.2 Fall 2

Komponentenroutinen

Fall-2-Aufrufe von Komponentenroutinen werden beibehalten.

Benutzerroutinen

Hier gilt dieselbe Argumentation wie bei Fall-1-Aufrufen von Benutzerroutinen. Allerdings werden durch Fall-2-Aufrufe keine Knoten eingespart, daher löscht die Spurbereinigung den *Return*- und den *Call*-Knoten.

Die eingehende Kante des *Return*-Knotens wird zunächst mit dem Nachfolger des *Call*-Knotens verbunden. Dann werden beide Knoten samt all ihren ein- und ausgehenden Kanten gelöscht (siehe Abbildung 4.2).

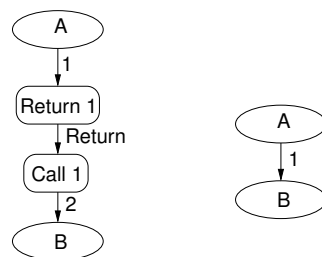


Abbildung 4.2: Löschen eines Fall-2-Aufrufes

4.2.3 Fall 3

Komponentenroutinen

Aufrufe von Komponentenroutinen sind relevant für die Sequenzen von Spurgraphen und werden daher beibehalten.

Benutzerrountinen

Fall-3-Aufrufe von Benutzerrountinen können nur Aufrufe zur Speicherallokierung (*malloc()*) sein. Diese Information ist nützlich, wenn die Erzeugung eines Haldenobjekts nicht durch einen Konstruktor gekapselt ist. Daher werden diese Aufrufe beibehalten.

4.3 Bedingungen

In Unterabschnitt 2.4.6 auf Seite 30 wurde beschrieben, dass es zwei Arten von Bedingungen gibt: Solche, bei denen der Ausdruck der Bedingung im Spurgraph erscheint und solche, bei denen dies nicht der Fall ist. Abbildung 4.3 auf der nächsten Seite zeigt nochmals die dabei entstehenden Spurgraphen.

Aus Sicht des Benutzercodes sind beide Bedingungen deterministisch. Aus Sicht des Spurgraphen ist die rechte Bedingung jedoch nicht-deterministisch, weil das Entscheidungskriterium dafür, welcher Zweig genommen werden soll, nicht von der Komponente abhängt. Daher macht in diesem Fall weder die *True*- noch die *False*-Kante Sinn.

In Unterabschnitt 3.3.1 auf Seite 70 haben wir gesehen, dass es in Protokollgraphen sinnvoll ist, nicht-deterministische *Condition*-Knoten zu verwenden. Es liegt nahe, die hier vorliegenden Knoten in dieselbe Form zu bringen, die nicht-deterministische *Condition*-Knoten in Protokollgraphen haben: eine eingehende Sequenz-Kante und beliebig viele ausgehende *Unconditional*-Kanten.

Daher ersetzt die Spurbereinigung bei nicht-deterministischen *Conditional*-Knoten die ausgehenden *True*- und *False*-Kanten durch *Unconditional*-Kanten.

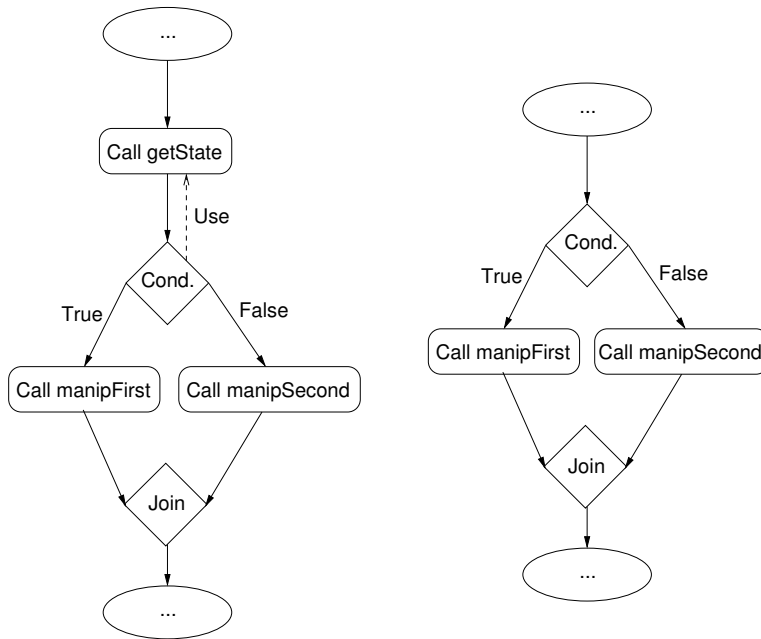


Abbildung 4.3: Spurgraph-Beispiel für Bedingungen.

4.4 Bereinigte Spurgraphen – Definition

Nachdem die Bereinigung von Spurgraphen beschrieben wurde, folgt die Definition der Syntax und Semantik von bereinigten Spurgraphen, basierend auf der Definition von extrahierten Spurgraphen (Definition 2.5.3 auf Seite 34).

4.4.1 Syntax

Definition 4.4.1 (Bereinigter Spurgraph) Ein bereinigter Spurgraph sei ein extrahierter Spurgraph nach Definition 2.5.3 auf Seite 34 mit folgenden Einschränkungen und Redefinitionen:

- Es gibt keine *Begin*- und *End_Of_Lifetime*-Knoten.
- Es gibt keine Fall-1-Aufrufe von Komponentenroutinen.
- Es gibt Fall-1-Aufrufe nur an Routinen, die mehrfach aufgerufen werden.

4 Spurbereinigung

- Es gibt keine Fall-2-Aufrufe von Benutzerrouninen.
- Deterministische *Condition*-Knoten haben genau eine eingehende Kante, die vom Typ *Sequence* sein muss. Sie haben genau eine ausgehende *True*-, eine ausgehende *False*- und eine ausgehende *Uses*-Kante. *Deterministic_Conditions* sei eine Teilmenge von *Conditions*.
- Nicht-deterministische *Condition*-Knoten haben genau eine eingehende Kante, die vom Typ *Sequence* sein muss. Sie haben beliebig viele ausgehende *Unconditional*-Kanten. Weitere ausgehende Kanten besitzen sie nicht. *Non_Deterministic_Conditions* sei eine Teilmenge von *Conditions*.
- Außer *Deterministic_Conditions* und *Non_Deterministic_Conditions* gebe es keine weiteren *Condition*-Knoten.

4.4.2 Semantik

Die Semantik eines extrahierten Spurgraphen ändert sich durch die Bereiniung nicht, da bei der Bereiniung nur solche Knoten und Kanten manipuliert werden, die in Unterabschnitt 2.4.7 auf Seite 32 und Definition 2.5.8 auf Seite 40 als irrelevant beschrieben wurden.

5 Rekursionsauflösung

Durch die Modellierung von Routinenaufrufen und -rümpfen können, wie schon angesprochen, rekursive Aufrufe in Spurgraphen vorkommen. Dieses Kapitel beschreibt zunächst die Schwierigkeiten, die rekursive Aufrufe mit sich bringen. Dann wird ein Muster zur Auflösung von rekursiven Aufrufen eingeführt und die genauen Folgen der Auflösung erläutert. Anschließend wird ein Algorithmus zur Identifizierung und Auflösung von Rekursionen in Spurgraphen beschrieben, der anhand eines Beispiels erläutert wird.

5.1 Motivation

Das Hauptproblem von Rekursionen wurde schon mehrfach angesprochen: Will man eine Sprache daraufhin testen, ob sie Teilmenge einer anderen Sprache ist (was beim Validieren von Spuren gegen Protokolle und umgekehrt geschieht), dann ist dies nur für reguläre Sprachen entscheidbar. Durch rekursive Aufrufe gehören die Spurgraphen jedoch nicht mehr zu den regulären, sondern nur noch zu den kontextfreien Sprachen.

Darüber hinaus kommt noch ein praktischer Aspekt hinzu. Wie in Unterabschnitt 2.4.5 auf Seite 24 beschrieben, können Spurgraphen nur pfadabhängig traversiert werden, z.B. durch Mitführen eines Stapels, auf den beim Eintreten in eine Routine der aufrufende Knoten abgelegt und beim Verlassen der Routine wieder heruntergenommen wird. Bei rekursiven Aufrufen kann hierbei ein Stapelüberlauf auftreten.

Damit ist klar, dass rekursive Aufrufe in Spurgraphen speziell behandelt werden müssen. Im Folgenden wird beschrieben, wie rekursive Aufrufe so aufgelöst werden können, dass die Spurgraphen anschließend wieder äquivalent zu regulären Sprachen sind.

5.2 Änderung der Semantik

Bei einer solchen Umwandlung wird zwangsläufig die Semantik des Graphen verändert. Es gibt verschiedene Forderungen, die an die Veränderung der Semantik gestellt werden. Diese werden nun anhand eines Beispiels herausgearbeitet.

Ein einfaches Beispiel für einen rekursiven Aufruf ist in Abbildung 5.1 zu sehen¹. Der rekursive Aufruf gewährleistet, dass immer gleich viele *Push*-Aufrufe wie *Pop*-Aufrufe vorkommen, so dass kein illegaler Zugriff auf den Stapel auftreten kann. Abbildung 5.2 auf der nächsten Seite zeigt den Spurgraphen, der dem Codebeispiel entspricht.

```
typedef struct {...} Stack;

int pop(Stack *object) {...}

void push(Stack *object, int arg) {...}
...
Stack myStack;
...
void routine () {
    if (...) {
        push(myStack, x);
        routine (); // recursive call
        y = pop(myStack);
    }
}

routine ();
```

Abbildung 5.1: C-Code-Beispiel mit einfachem rekursiven Aufruf

Alternativ könnte man die *routine* ohne rekursiven Aufruf mit Hilfe von Schleifen implementieren, wobei immer noch die Einschränkung gelten würde, dass gleich viele *Push*- wie *Pop*-Aufrufe erfolgen müssen. Diese alternative Implementierung ist in Abbildung 5.3 auf Seite 84, der zugehörige Spurgraph in Abbildung 5.4 auf Seite 84 dargestellt.

Es ist zu sehen, dass der Spurgraph aus Abbildung 5.4 auf Seite 84 u.a. folgende Sequenzen zulässt, die im Sinne des tatsächlichen Komponentenprotokolls illegal sind:

¹Eine Rekursion im weitesten Sinne: „siehe [Haa04]“.

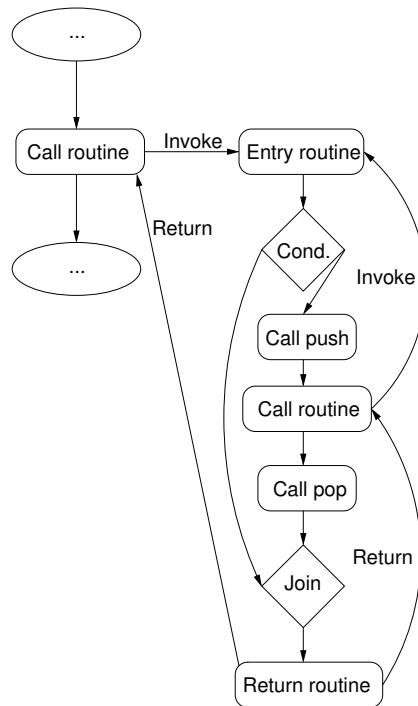


Abbildung 5.2: Spurgraph mit einfachem rekursivem Aufruf

- *Push, Pop, Pop,*
- *Pop, Pop.*

Diese Sequenzen können im ursprünglichen Code nicht vorkommen, weil durch die Bedingung der *for*-Schleife sichergestellt wird, dass beide Schleifen gleich oft durchlaufen werden. Da aber im hier verwendeten Verfahren zur Spurextraktion die Datenflussanalyse lediglich bestimmt, auf welcher Komponente eine Routine arbeitet, nicht aber, welche weiteren Argumente der Routine übergeben werden, kann diese Einschränkung nicht festgestellt werden. Stattdessen wird angenommen, dass alle Sequenzen eines Spurgraphen auch tatsächlich vorkommen können (*konservative Annahme*). Dadurch ist die Menge der Sequenzen, die der Spurgraph definiert, größer als die Menge der Sequenzen, die der Code definiert. Da die Sprache eines Spurgraphen durch dessen Sequenzen definiert ist, wird also die Sprache der Spur bei der Umwandlung in einen Spurgraphen erweitert.

5 Rekursionsauflösung

```
int number() // returns always same number
```

```
void routine () {  
  for (i = 0; i < number(); i++) {  
    push(myStack, x);  
  }  
  for (i = 0; i < number(); i++) {  
    y = pop(myStack);  
  }  
}
```

Abbildung 5.3: C-Code-Beispiel für alternative Implementierung mit Schleifen

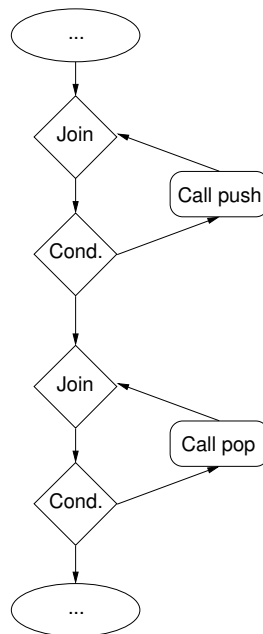


Abbildung 5.4: Spurgraph der alternativen Implementierung mit Schleifen

Bei der Validierung von Spuren gegen Protokolle hat diese Erweiterung Konsequenzen, die zu verschmerzen sind: Ein Protokoll wird weiterhin alle falschen Spuren ablehnen. Es kann lediglich sein, dass eine korrekte Spur durch die Erweiterung der Sprache in ihrer graphischen Repräsentation Sequenzen enthält, die das Protokoll nicht akzeptiert. Dieser Zusammenhang ist in Abbildung 5.5 auf der nächsten Seite dargestellt, wobei $L(S)$

die Sprache der ursprünglichen Spur repräsentiert, $L(S')$ die Sprache des Spurgraphen und $L(P)$ die Sprache des Protokollgraphen.

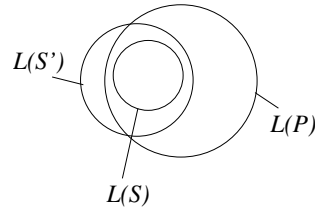


Abbildung 5.5: Zusammenhang zwischen Sprachen von Spuren, von Spurgraphen und von Protokollgraphen

Anders ist es, wenn ein hypothesengetriebenes Protokoll gegen eine Spur validiert wird. Durch die Erweiterung der Sprache der Spur, die illegale Sequenzen enthalten kann, wird ein korrektes Protokoll evtl. als fehlerhaft betrachtet. Eine mögliche Folge ist, dass der Wartungs-Ingenieur den Protokollgraph so erweitert, dass dieser die falschen Sequenzen von nun an akzeptiert. Dieser Gefahr muss sich der Wartungs-Ingenieur bewusst sein, um gegebenenfalls durch manuelle Analyse der Spur oder durch Slicing solche Abhängigkeiten zu suchen und den Protokollgraph somit frei zu halten von illegalen Sequenzen. Im konkreten Beispiel könnte er, um die Korrektheit von Spuren zu gewährleisten, ein Zusicherung `assert(!isEmpty(myStack))` vor dem `pop()`-Aufruf einfügen.

Wir haben bisher gesehen, wie eine einfache Rekursion durch eine Schleife repräsentiert werden kann und welche Auswirkungen die Umwandlung von Spuren, die Schleifen enthalten, auf die Sprache der Spuren hat.

Laut [Sed92] kann jede beliebige Rekursion in eine Schleife umgewandelt werden. Für Endrekursionen ist diese Umwandlung trivial, bei komplexeren Rekursionen müssen in der Regel die impliziten Bedingungen, die Rekursionen beinhalten, explizit ausprogrammiert werden, z.B. indem ein Stapel gehalten wird, auf den Elemente in einer bestimmten Reihenfolge geschoben und anschließend abgearbeitet werden. Dabei werden implizite Bedingungen in den Datenfluss verlagert. Genau wie bei dem gezeigten einfachen Beispiel kann auch hier argumentiert werden, dass derlei Datenabhängigkeiten bei der Umwandlung in Spurgraphen verloren gehen.

Nichtsdestoweniger ist die Umwandlung von Rekursionen in Schleifen eine Möglichkeit, das vorliegende Problem der Rekursionen zu lösen. Es liegt

daher nahe, rekursive Aufrufe in Spurgraphen durch Strukturen zu ersetzen, die Schleifen repräsentieren.

Im folgenden Abschnitt wird ein solches Ersetzungsmuster eingeführt.

5.3 Ersetzungsmuster

Betrachten wir dazu den Kontrollfluss am rekursiven *Call*-Knoten (siehe hierzu Abbildung 5.6 auf der nächsten Seite, linker Teil). Vom *Call push*-Knoten her kommend, muss, nachdem der *Call routine*-Knoten besucht wurde, die ausgehende *Invoke*-Kante beschriftet werden (oberer gestrichelter Pfeil). Kommt man vom *Return routine*-Knoten über die *Return*-Kante, dann muss als nächstes die ausgehende *Unconditional*-Kante zum *Call pop*-Knoten beschriftet werden (unterer gestrichelter Pfeil). Daher erscheint es sinnvoll, den *Call routine*-Knoten in zwei Knoten aufzuteilen, um den Kontrollfluss exakter zu modellieren. Diese Aufteilung ist rechts in Abbildung 5.6 auf der nächsten Seite zu sehen²

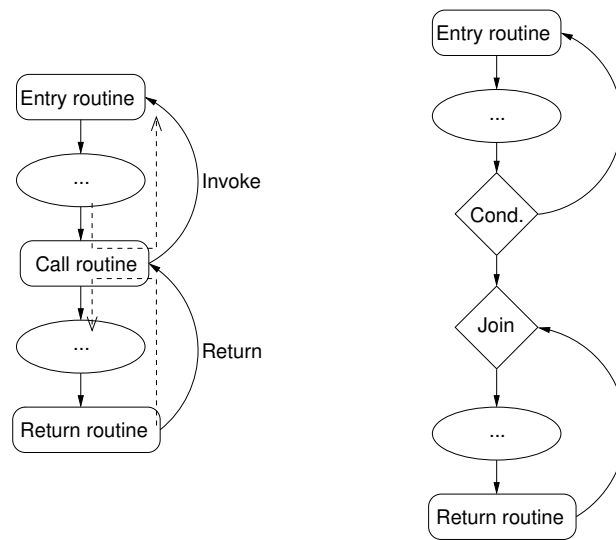
Wendet man dieses Ersetzungsmuster auf unser ursprüngliches Beispiel (Abbildung 5.2 auf Seite 83) an, dann erhält man einen Spurgraphen, der dieselben Sequenzen und somit dieselbe Sprache definiert wie die alternative Implementierung mit Schleifen (Abbildung 5.4 auf Seite 84). Der Spurgraph ist in Abbildung 5.7 auf Seite 88 zu sehen.

Damit ist das grundlegende Muster zur Auflösung von Rekursionen spezifiziert. Als nächstes wird ein Algorithmus beschrieben, der Rekursionen in Spurgraphen erkennt und mit Hilfe des gerade beschriebenen Musters auflöst.

5.4 Algorithmus zur Rekursionsauflösung

Zunächst wird ein kurzer Überblick über den Algorithmus gegeben. Danach folgt ein Auszug aus dem Code des Algorithmus und eine Beschreibung dieses Auszugs. Der Algorithmus wird anschließend anhand eines

² Dieses Aufteilungsmuster führt offensichtlich zu einem Graphen, der gegen die Definition von extrahierten und bereinigten Spurgraphen verstößt. Dies ist jedoch nur der erste Schritt zur Rekursionsauflösung, in weiteren Schritten wird dafür gesorgt, dass die Konformität zur Definition wieder hergestellt wird.

Abbildung 5.6: Kontrollfluss an *Call*-Knoten und Ersetzungsmuster

Beispiels veranschaulicht. Dann werden Details angesprochen, die im Code aus Gründen der Übersichtlichkeit weggelassen wurden und schließlich werden noch Möglichkeiten zur Optimierung des Algorithmus vorgestellt.

5.4.1 Überblick

Man traversiert einen Spurgraphen durch eine pfadsensitive Tiefensuche. Durch die Pfadsensitivität ist an jeder Stelle bekannt, welche Routinen auf dem gerade beschrifteten Weg betreten, jedoch noch nicht wieder verlassen wurden. Ein Aufruf ist genau dann rekursiv, wenn er eine solche Routine aufruft.

Von allen besuchten Knoten werden Kopien angelegt, die in einen neuen Graphen kopiert werden. Rekursive Aufrufe werden dabei in der Kopie durch das oben beschriebene Muster ersetzt.

Durch das Kopieren während der Tiefensuche wird automatisch jede mehrfach aufgerufene Routine pro Aufruf einmal in den resultierenden Graph kopiert (siehe Unterabschnitt 5.4.3 auf Seite 93). Dadurch werden Schwierigkeiten, die bei überlagerten Zyklen und bei Zyklen mit mehreren Eingängen auftreten, umgangen. Auf diese Art entsteht ein gerichteter Graph, der azyklisch ist in Bezug auf *Invoke*-Kanten.

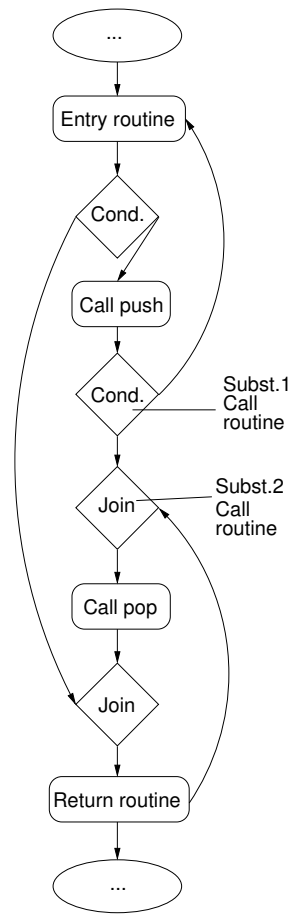


Abbildung 5.7: Graph aus Abbildung 5.2 auf Seite 83 mit aufgelöster Rekursion

In diesem Graphen werden nun alle *Call*-Knoten durch ihre Routinenrumpfe ersetzt. Dieser Schritt ist deswegen sinnvoll, weil die *Call*, *Entry*- und *Return*-Knoten von Fall-1-Routinen in einem bereinigten Spurgraphen nur enthalten sind, um bei mehrfach aufgerufenen Routinen Knoten einzusparen (siehe Unterabschnitt 4.2.1 auf Seite 76). Dadurch, dass der Graph hier kopiert wird, geht dieser Vorteil verloren, wodurch die *Call*-, *Entry*- und *Return*-Knoten von Fall-1-Aufrufen nutzlos werden.

5.4.2 Code-Auszug und Beschreibung

Abbildung 5.8 auf der nächsten Seite stellt eine Skizze des Algorithmus dar. Aus Gründen der Übersichtlichkeit wurden bestimmte Details (z.B. das Kopieren von Kanten) und Optimierungen weggelassen, siehe hierzu Unterabschnitt 5.4.4 auf Seite 96.

Der Algorithmus verwendet zwei Stapel, um die korrekte Traversierung und die Erkennung von rekursiven Aufrufen sicherzustellen. Auf dem *Call-Stack* werden alle Fall-1-*Call*-Knoten abgelegt, deren Routine momentan betreten, jedoch noch nicht wieder verlassen wurde. Der *Visited_Nodes-Stack* hingegen hält nicht einzelne Knoten, sondern Knotenmengen. In der Knotenmenge, die zu oberst liegt, befinden sich genau diejenigen Knoten, die im momentan betretenen Routinenrumpf (oder zwischen Start- und Endknoten, falls keine Routine betreten wurde) besucht wurden. Besuchte Knoten werden u.a. benötigt, um intra-prozedurale Zyklen, d.h. Schleifen, zu erkennen. Die besuchten Knoten können nicht auf einer globalen Liste gehalten werden, weil sonst keine mehrfach aufgerufene Routine mehrfach betreten werden würde.

Nachdem die beiden Stapel zu Beginn initialisiert wurden, wird die Prozedur *Traverse* aufgerufen und dabei der Startknoten, der Knotentyp *Finals* und eine leere Zeichenkette übergeben.

Die *Traverse*-Prozedur traversiert den Graphen ausgehend vom gegebenen Knoten. Die beiden anderen Parameter (*Type* und *Label*) geben die Abbruchbedingung für die Traversierung an. Wenn ein Knoten des gegebenen Typs gefunden wird, der den gegebenen Namen trägt, dann werden keine weiteren Nachfolger dieses Knotens traversiert. Da die *Traverse*-Prozedur jedoch rekursiv aufgerufen wird, ist sichergestellt, dass alle Knoten besucht werden, die auf einem Weg vom gegebenen Knoten zu demjenigen liegen, auf den die Abbruchbedingung zutrifft. Die ist in Abbildung 5.9 auf Seite 91 veranschaulicht, wobei *Node* für den gegebenen Knoten steht, *Type/Label* für den Knoten, auf den die Abbruchbedingung zutrifft und die Nummern an den Knoten eine mögliche Traversierungsreihenfolge angeben.

Der erste Aufruf von *Traverse* veranlasst die Routine also, den Graphen vom *Start*- bis zum *Final*-Knoten zu traversieren. Das *Label* ist in diesem Fall leer, da *Final*-Knoten ja per Definition (siehe Definition 2.5.3 auf Seite 34) keine Beschriftung tragen.

5 Rekursionsauflösung

```
procedure Visit (Node) is
begin
  Insert (Top (Visited_Nodes_Stack), Node);

  if Node  $\in$  Calls_Case_1 then
    if ( $\exists V \in$  Call_Stack :  $l(V) = l(Node)$ ) then
      // recursive Call
      Handle_Recursive_Call (Node);
    else
      // non-recursive Call
      Add_To_Copied_Graph (Node);

      Push (Call_Stack, Node);
      Push (Visited_Nodes_Stack,  $\emptyset$ );

      Entry_Node := First (Successors (Node, Invokes));

      Traverse (Entry_Node, Returns,  $l(Entry\_Node)$ );

      for S in Top (Visited_Nodes_Stack) loop
        Add_To_Copied_Graph (S);
      end loop;

      Pop (Visited_Nodes_Stack);
      Pop (Call_Stack);
    end if;
  else
    Add_To_Copied_Graph (Node);
  end if;
end Visit;

procedure Traverse (Node, Type, Label) is
begin
  if Node  $\notin$  Top (Visited_Nodes_Stack) then
    Visit (Node);
    if not (Node  $\in$  Type  $\wedge$   $l(Node) = Label$ ) then
      for V in Successors (Node, Sequences) loop
        Traverse (V, Type, Label);
      end loop;
    end if;
  end if;
end Traverse;

Call_Stack :=  $\emptyset$ ;
Visited_Nodes_Stack :=  $\emptyset$ ;
Traverse (Start_Node, Finals, "");
```

Abbildung 5.8: Skizze des Algorithmus zur Rekursionsauflösung

5.4 Algorithmus zur Rekursionsauflösung

Nach Betreten der *Traverse*-Prozedur wird zunächst überprüft, ob der übergebene Knoten innerhalb der momentan betretenen Routine (bzw. auf dem Weg vom Startknoten, falls wir außerhalb aller Routinen sind) schon einmal besucht wurde. Da die besuchten Knoten zu oberst auf dem *Visited_Nodes_Stack* liegen, wird einfach auf Zugehörigkeit zu dieser Knotenmenge geprüft. Trifft dies zu, dann wird nichts weiter unternommen, wodurch die *Traverse*-Prozedur unverrichteter Dinge wieder verlassen wird. Wurde der Knoten noch nicht besucht, dann geschieht dies nun durch Aufruf der *Visit*-Prozedur. Anschließend wird überprüft, ob die gegebene Abbruchbedingung (Knotentyp und Beschriftung) zutrifft. Wenn das der Fall ist, wird *Traverse* wiederum ohne weitere Aktion verlassen. Wenn nicht, dann wird über alle Nachfolger des Knotens iteriert, die über Sequenzkanten erreichbar sind, und jeder dieser Nachfolger mit der eingegebenen Abbruchbedingung durch einen rekursiven Aufruf der *Traverse*-Prozedur traversiert.

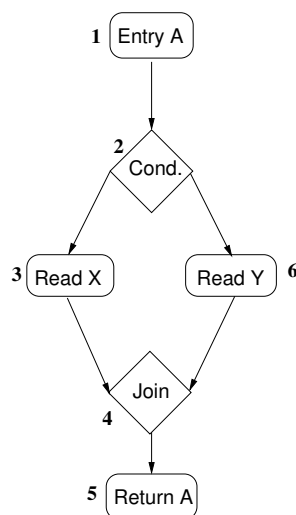


Abbildung 5.9: Traversierung mit *Visited_Nodes_Stack*

Betrachten wir nun die *Visit*-Prozedur. Sie markiert zunächst den gegebenen Knoten als besucht, indem sie ihn der Knotenmenge hinzufügt, die zu oberst auf dem *Visited_Nodes_Stack* liegt. Anschließend bestimmt sie den Typ des gegebenen Knotens und behandelt ihn entsprechend.

Für alle Knoten außer Fall-1-*Call*-Knoten (*else*-Zweig der äußeren *if*-Anweisung) wird einfach eine Kopie des gegebenen Knotens zum Gra-

5 Rekursionsauflösung

phen hinzugefügt, der während der Traversierung erstellt wird (diesen Graphen nennen wir von nun an *Ergebnis-Graph*). Dies geschieht in der *Add_To_Copied_Graph*-Prozedur, die aus Gründen der Übersichtlichkeit nicht dargestellt wird.

Für Fall-1-*Call*-Knoten muss unterschieden werden, ob es sich um einen rekursiven Aufruf handelt oder nicht. Dies geschieht, indem überprüft wird, ob auf dem *Call_Stack* schon ein Aufruf mit derselben Beschriftung liegt. Wenn ja, handelt es sich um einen rekursiven Aufruf.

Rekursive Aufrufe werden in der (nicht dargestellten) Prozedur *Handle_Recursive_Call* behandelt. Dort wird das Ersetzungsmuster erstellt, das im letzten Abschnitt beschrieben wurde, und dem Ergebnis-Graphen anstelle des rekursiven *Call*-Knotens hinzugefügt.

Nicht-rekursive Fall-1-*Call*-Knoten werden folgendermaßen behandelt: Zunächst wird eine Kopie des *Call*-Knotens in den Ergebnis-Graphen eingefügt. Dann wird der *Call*-Knoten auf den *Call_Stack* und eine leere (Knoten)-Menge auf den *Visited_Nodes_Stack* geschoben. Anschließend wird der *Entry*-Knoten der Routine bestimmt. Dieser wird zur Traversierung des Routinenrumpfs der *Traverse*-Prozedur übergeben. Die Traversierung des Rumpfs soll dann abbrechen, wenn ein *Return*-Knoten gefunden wird, der dieselbe Beschriftung trägt wie der *Entry*-Knoten, d.h. der zur selben Routine gehört. Die *Traverse*-Prozedur gewährleistet, dass alle Knoten, die auf Wegen vom *Entry*- zum *Return*-Knoten liegen, traversiert wurden, wenn die *Traverse*-Prozedur wieder verlassen wird. Daher befinden sich nun alle Knoten des Routinenrumpfs in der Knotenmenge, die zu oberst auf dem *Visited_Nodes_Stack* liegt. Über diese Knoten wird nun iteriert, wobei von allen Knoten Kopien angelegt und zum Ergebnis-Graph hinzugefügt werden. Zuguterletzt müssen die obersten Elemente von den beiden Stapeln entfernt werden, da ja gerade ein Routinenrumpf verlassen wurde.

Ausgabe dieses Algorithmus ist, wie oben beschrieben, ein Graph, der azyklisch in Bezug auf *Invoke*-Kanten ist. In diesem Graph werden nun alle Fall-1-*Call*-Knoten durch die Routinenrumpfe ersetzt. Zuguterletzt werden die *Entry*- und *Return*-Knoten durch *Join*- und *Condition*-Knoten ersetzt, wobei sie als Beschriftung den Namen desjenigen Knotens erhalten, den sie ersetzen³.

³Nachdem das Muster zur Auflösung von Rekursionen ja Kanten einführt, die nicht definiert sind (z.B. *Invoke*-Kanten von einem *Condition*- zu einem *Entry*-Knoten),

5.4.3 Veranschaulichung

Veranschaulichen wir uns nun den Algorithmus zur Rekursionsauflösung anhand eines Beispiels. Dieses ist in Abbildung 5.10 dargestellt.

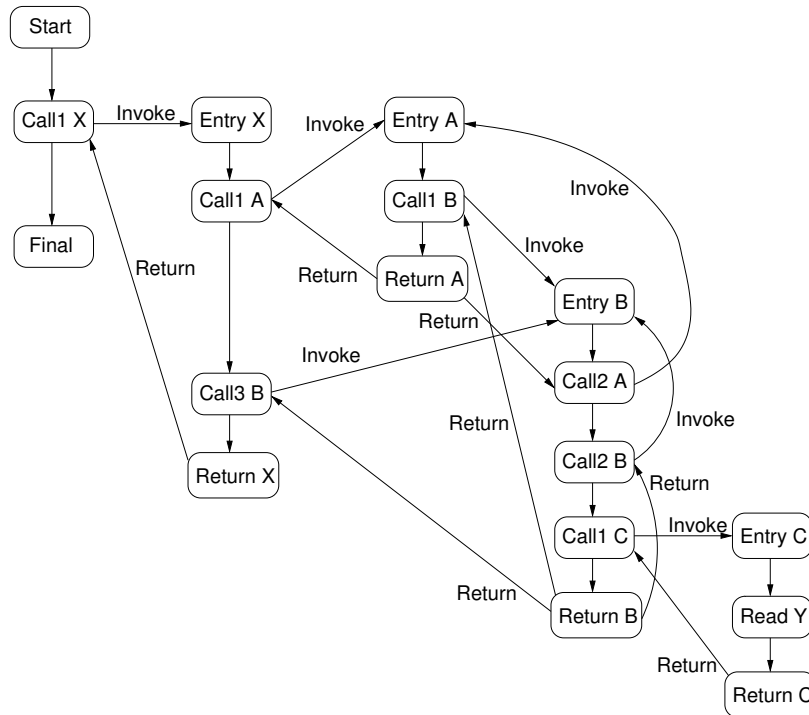


Abbildung 5.10: Beispiel für den allgemeinen Fall

Der Algorithmus identifiziert zunächst die folgenden beiden Wege, die Zyklen enthalten (es werden jeweils nur die *Call*-Knoten von solchen Routinen aufgeführt, die betreten aber nicht wieder verlassen wurden):

entsprechen die Graphen nun fast wieder ihrer Definition. „Fast“ deswegen, weil per Definition *Condition*- und *Join*-Knoten keine Beschriftung haben. Diese Erweiterung wird jedoch bei der Definition von rekursionsfreien Spurgraphen eingeführt, siehe Definition 6.1.1 auf Seite 99

5 Rekursionsauflösung

$$\underbrace{Call1X, Call1A, Call1B, Call2A}_{\text{Zyklus}} \quad (5.1)$$

$$\underbrace{Call1X, Call1A, Call1B, Call2B}_{\text{Zyklus}} \quad (5.2)$$

In Abbildung 5.11 ist der Graph zu sehen, der entsteht, wenn der Algorithmus *Call1A* komplett abgearbeitet hat und als nächstes *Call3B* besuchen will. Wie im Schaubild sichtbar ist, wurden *Call2A* und *Call2B* als rekursive Aufrufe erkannt und durch das beschriebene Muster in Schleifenkonstrukte umgewandelt.

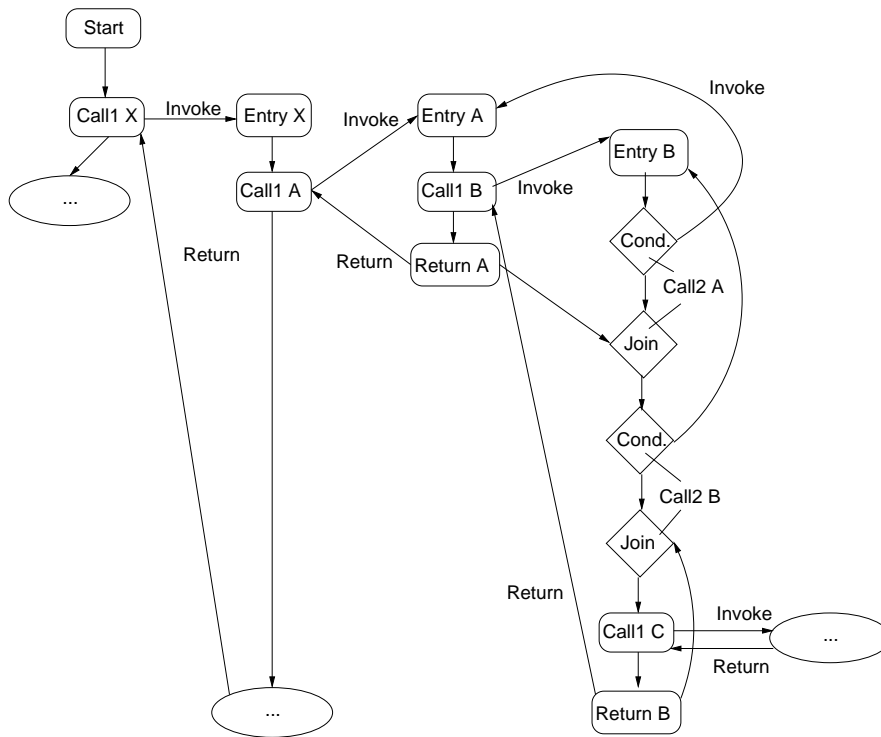


Abbildung 5.11: Aufgelöste rekursive Aufrufe – oberer Teil

5.4 Algorithmus zur Rekursionsauflösung

Beim Besuchen der Routinen unterhalb von *Call3B* werden folgende Wege gegangen:

$$Call1X, \underbrace{Call3B, Call2A, Call1B}_{\text{Zyklus}} \quad (5.3)$$

$$Call1X, \underbrace{Call3B, Call2B}_{\text{Zyklus}} \quad (5.4)$$

Das Ergebnis dieses Teils der Traversierung ist in Abbildung 5.12 dargestellt. Hier wurden die beiden rekursiven Aufrufe *Call1B* und *Call2B* ersetzt.

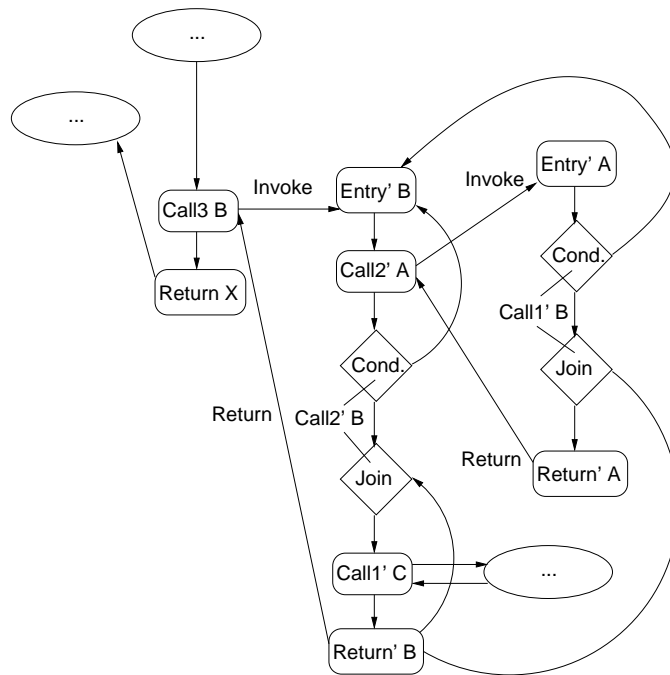


Abbildung 5.12: Aufgelöste rekursive Aufrufe – unterer Teil

Im letzten Schritt werden, wie beschrieben, alle *Call*-Knoten durch ihre Routinenrumpfe ersetzt. Das Ergebnis ist in Abbildung 5.13 auf Seite 97 und Abbildung 5.14 auf Seite 98 zu sehen.

5.4.4 Vervollständigung und Optimierungen

Wie schon angesprochen enthält die Skizze des Algorithmus einige Lücken und lässt Raum für Optimierungen.

So beschreibt der Algorithmus bisher nur die Behandlung von Knoten, nicht aber von Kanten. Um ihn zu vervollständigen, müssen von allen traversierten Kanten Kopien angelegt werden, die dann in den Ergebnis-Graphen so eingefügt werden, dass sie die passenden kopierten Knoten verbinden. Die einzige Ausnahme bilden hier rekursive *Call*-Knoten. Die eingehende Sequenz-Kante eines solchen Knotens wird in der Kopie an den *Condition*-Knoten des Ersetzungsmusters gehängt, die ausgehende Sequenz-Kante an den *Join*-Knoten, siehe auch Abbildung 5.6 auf Seite 87.

Eine mögliche Optimierung liegt darin, die Ersetzung der *Call*-Knoten durch ihre Routinenrümpfe nicht in einem zusätzlichen Schritt, sondern schon während der Traversierung vorzunehmen. Dazu wird die *Traverse*-Routine so abgewandelt, dass sie eine Menge von Knoten zurückliefert. Auf diese Art werden alle Routinenrümpfe rekursiv nach oben durchgereicht und können anstelle ihres aufrufenden Knotens in den Ergebnis-Graph kopiert werden.

Eine weitere Möglichkeit zur Optimierung besteht darin, nicht alle Knoten aller gegangenen Wege zu kopieren, sondern nur diejenigen, bei denen eine Kopie wirklich notwendig ist, wodurch der Vorteil eingesparter Knoten durch mehrfach aufgerufene Routinen nicht völlig verloren geht. Offensichtlichstes Beispiel ist die Routine *C*, die keinem Zyklus über *Invoke*-Kanten des vorgestellten Graphen angehört. Aufgrund dieser Eigenschaft, die sich ebenfalls leicht durch den Traversierungs-Algorithmus herausfinden lässt, kann automatisch entschieden werden, dass ihr Rumpf auch im Ergebnis-Graph nur einmal vorhanden ist, dafür jedoch mehrfach aufgerufen wird. Genauso kann mit allen anderen *Call*-Knoten verfahren werden, die zu keinem Zyklus gehören.

In der Implementierung des Werkzeugs zur Rekursionsauflösung, das im Rahmen dieser Arbeit entstanden ist, sind diese Optimierungen umgesetzt.

5.4 Algorithmus zur Rekursionsauflösung

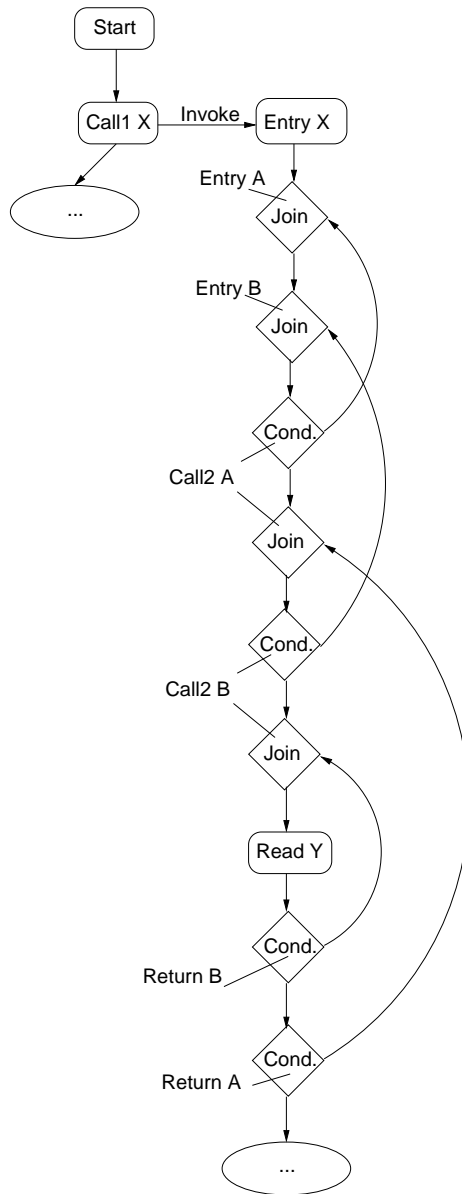


Abbildung 5.13: Ergebnis der Rekursionsauflösung – oberer Teil

5 Rekursionsauflösung

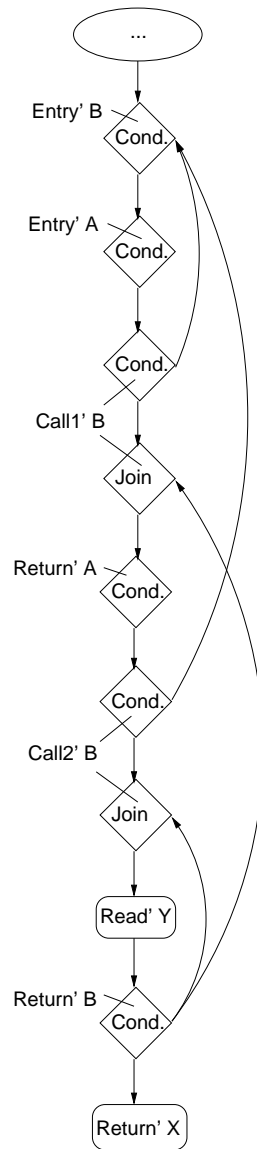


Abbildung 5.14: Ergebnis der Rekursionsauflösung – unterer Teil

6 Validierung

Nachdem die letzten beiden Kapitel gezeigt haben, wie die Spuren bereinigt und von Rekursionen befreit werden können, beschäftigt sich dieses Kapitel mit der Validierung von Protokollgraphen gegen rekursionsfreie Spurgraphen.

Dazu wird zunächst ein rekursionsfreier Spurgraph, dann ein Protokollgraph definiert. Anschließend wird gezeigt, wie man einen Protokoll-/Spurgraphen in einen endlichen Automaten umwandeln kann.

In den darauf folgenden Abschnitten werden basierend auf der Umwandlung von nicht-deterministischen in deterministische Automaten nach Rabin und Scott [Rab59] [Hop79] die Schritte beschrieben, die bei der Validierung gegen korrekte und falsche Spuren und bei der Überdeckung vollzogen werden. Zuguterletzt wird gezeigt, wie die obligatorischen und verbotenen Vorgänger und Nachfolger u.a. mit Hilfe von Dominanz- und Post-Dominanz-Analyse bestimmt werden.

6.1 Protokollgraphen – Definition

Um einen Protokollgraphen zu definieren, wird zunächst ein rekursionsfreier Spurgraph definiert, wie er entsteht, wenn man die Rekursionsauflösung auf einen bereinigten Spurgraphen anwendet, die im letzten Kapitel beschrieben wurde.

Definition 6.1.1 (Rekursionsfreier Spurgraph) Ein rekursionsfreier Spurgraph S ist ein bereinigter Spurgraph mit folgenden Einschränkungen und Erweiterungen:

- Er enthält keine rekursiven Aufrufe.
- *Call*-, *Entry*- und *Return*-Knoten müssen beschriftet sein. *Condition*- und *Join*-Knoten dürfen beschriftet sein. Andere Knoten dürfen nicht beschriftet sein.

Definition 6.1.2 (Protokollgraph) Ein Protokollgraph ist ein rekursionsfreier Spurgraph nach Def. 6.1.1.

Aufgrund dieser Definition wird von nun an „Protokollgraph“ verwendet, wenn Eigenschaften von Protokollgraphen und rekursionsfreien Spurgraphen beschrieben werden und damit der Unterschied zwischen Spur und Protokoll unerheblich ist.

6.2 Umwandlung in endlichen Automaten

Heiber beschreibt in [Hei00] wie ein Protokollgraph in einen endlichen Automaten umgewandelt werden kann (siehe Abbildung 6.2 auf Seite 102). Die Protokoll- und Spurgraphen aus [Hei00] unterscheiden sich jedoch von den hier definierten in drei Punkten:

1. Die hier definierten Graphen enthalten Routinenrümpfe. Diese können jedoch, wie beim Algorithmus zur Rekursionsauflösung in Abschnitt 5.4 auf Seite 86 beschrieben, an Stelle ihrer Aufrufe in den Graph kopiert werden, womit dieser Unterschied ausgemerzt ist.
2. *Condition*-Knoten bei Heiber geben den Ausdruck der Bedingungen durch eine Beschriftung anstatt durch eine *Use*-Kante an. Dieser Unterschied kann jedoch leicht eliminiert werden, indem alle *Condition*-Knoten mit *Use*-Kanten nach dem Muster in Abbildung 6.1 auf der nächsten Seite ersetzt werden.
3. Im Gegensatz zu den Graphen bei Heiber haben die hier definierten Graphen *Join*-Knoten. Dieser Unterschied kann behoben werden, indem die eingehenden Kanten der *Join*-Knoten direkt auf ihre Nachfolger gesetzt und die *Join*-Knoten anschließend entfernt werden.

Mit diesen vorbereitenden Schritten können die hier definierten Protokollgraphen nach dem Verfahren von Heiber in endliche Automaten umgewandelt werden.

Um die Diagramme in den folgenden Abschnitten übersichtlich zu halten, werden alle irrelevanten Zustände und Kanten der Automaten weggelassen, d.h. alle ε -Kanten und alle Zustandsknoten, die bei der Umwandlung wegen den ε -Kanten eingefügt wurden. Die Sprache des Automaten ändert sich dadurch nicht.

6.3 Umwandlung in deterministischen Automaten

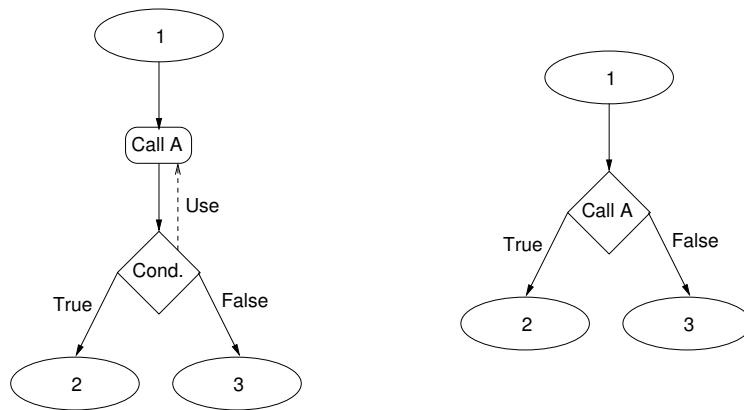


Abbildung 6.1: Umwandlung von *Condition*-Knoten in die in [Hei00] beschriebene Form

Beispiel 6.2.1 Abbildung 3.5 auf Seite 59 zeigt die Graphen von Spur 2 und von Protokollhypothese 1, die in Unterabschnitt 3.2.1 auf Seite 58 beschrieben wurden. Darüberhinaus sieht man dort die endlichen Automaten, die sich aus diesen Graphen ergeben. *SP* und *SS* repräsentieren dabei die Startzustände, *FP* und *FS* die Endzustände, alle anderen Zustände sind fortlaufend nummeriert. *P* markiert die Zustände des Protokolls, *S* die Zustände der Spur.

6.3 Umwandlung in deterministischen Automaten

Will man Protokollgraphen gegen Spurgraphen validieren, dann kann man wie folgt vorgehen:

1. Man wandelt beide Graphen in endliche Automaten um.
2. Die Zustände der beiden Automaten werden so markiert, dass erkennbar ist, ob sie zum Spur- oder zum Protokollgraphen gehören.
3. Beide werden zu einem nicht-deterministischen endlichen Automaten vereinigt, indem ein Startknoten gebildet wird, der als Beschriftung die Beschriftung und Markierung der ursprünglichen Startknoten enthält. Er ersetzt beide Startknoten.
4. Das Gleiche wird mit den Endknoten gemacht.

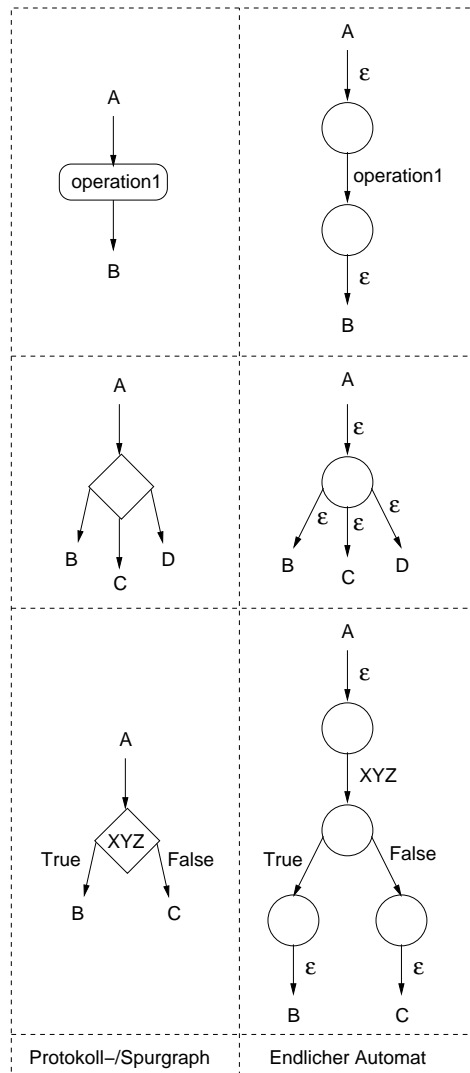


Abbildung 6.2: Umwandlung von Protokollgraphen in endliche Automaten nach [Hei00]

- Der so entstandene nicht-deterministische Automat wird nach dem bekannten Verfahren von Rabin und Scott [Hop79] durch Bildung der Potenzmengen Zustände in einen deterministischen Automaten umgewandelt.

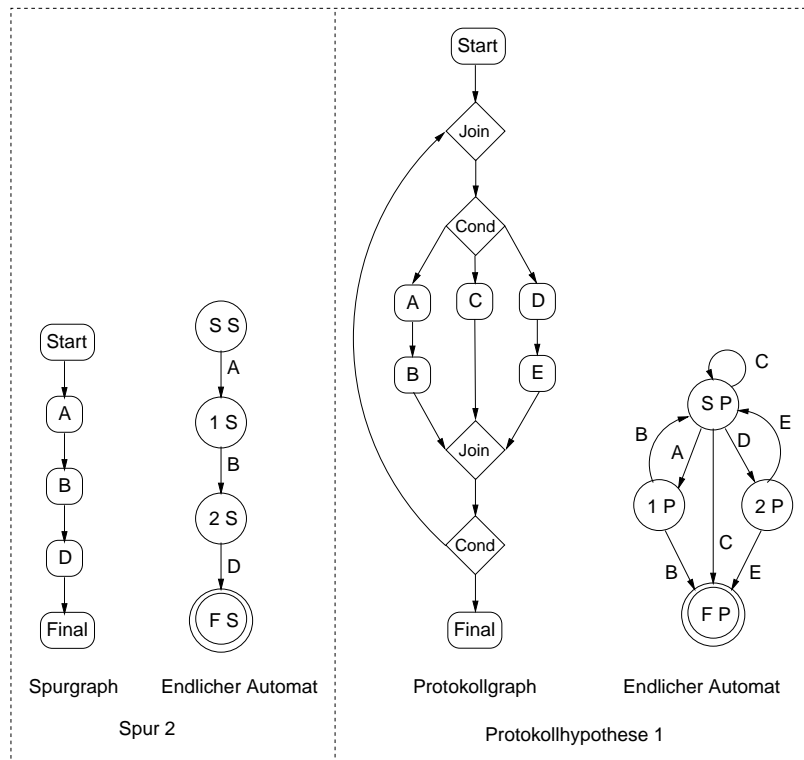


Abbildung 6.3: Protokoll- und Spurgraph aus Abbildung 3.5 auf Seite 59 als endlicher Automat

Anhand dieses deterministischen Automaten kann nun die Validierung gegen korrekte und falsche Spuren und die Überdeckung vorgenommen werden.

Ein Nachteil dieser Vorgehensweise ist offensichtlich, dass der deterministische Automat im schlimmsten Fall exponentiell anwächst. Es gibt jedoch – wenn auch nur begrenzt – Möglichkeiten für Optimierungen, von denen eine im nächsten Abschnitt umrissen wird.

6.4 Validierung gegen korrekte Spuren

Die Fragestellung bei der Validierung eines Protokolls gegen korrekte Spuren ist: Wird eine korrekte Spur von einem Protokoll akzeptiert? Wenn

6 Validierung

nein, welcher Knoten im Spurgraph wird nicht akzeptiert? Wie ist der Weg vom Startknoten zu diesem Knoten?

Diese Informationen erhält man durch den gerade beschriebenen deterministischen Automaten auf folgende Art:

Man traversiert den deterministischen Automaten ausgehend vom Startknoten und merkt sich dabei den gegangenen Weg. Beim Traversieren sucht man nach zwei Arten von Zuständen:

1. Zustände, die nur Markierungen des Spurgraphen, nicht aber des Protokollgraphen enthalten. Ein solcher Zustand kann nur dann auftreten, wenn eine Sequenz von Operationen zwar in der Sprache der Spur, nicht aber in der Sprache des Protokolls liegt.
2. Zustände, die zwar Endzustände der Spur, nicht aber des Protokolls beinhalten. In diesem Fall wird die Sequenz, die zu diesem Zustand führt, von der Spur akzeptiert, im Protokoll hingegen ist sie lediglich der Beginn einer akzeptierten Sequenz.

Findet man einen der gerade beschriebenen Zustände, dann gibt man die Kantenbeschriftungen des gegangenen Weges aus. Auf diese Art kann der Wartungs-Ingenieur sowohl im Spur- als auch im Protokollgraph nachvollziehen, welcher Weg gegangen wurde, um zu dem nicht akzeptierten Knoten zu gelangen.

Dieses Vorgehen hat außerdem den Vorteil, dass schon bei der Erstellung des deterministischen Automaten abgebrochen werden kann, sobald ein deterministischer Knoten erstellt wird, der eine der gerade genannten Bedingungen erfüllt.

Beispiel 6.4.1 Wenden wir nun dieses Verfahren auf die endlichen Automaten von Spur 2 und Protokollhypothese 1 an. Damit ergibt sich der deterministische Automat aus Abbildung 6.4 auf der nächsten Seite.

Wie zu sehen ist, enthalten alle Zustände des Automaten mindestens ein P , d.h. es gibt keinen Zustand, der nicht auf das Protokoll abgebildet werden kann. Allerdings enthält der Zustand FS , $2P$ keinen Endzustand des Protokolls. Damit ist klar, dass der Protokollgraph die Spur nicht akzeptiert. Die Ausgabe ist in diesem Fall (A, B, D) . Der Wartungs-Ingenieur kann nun in den Originalgraphen die Wege mit diesen Knoten abgehen und auf diese Art die fehlerhafte Stelle im Protokollgraph herausfinden.

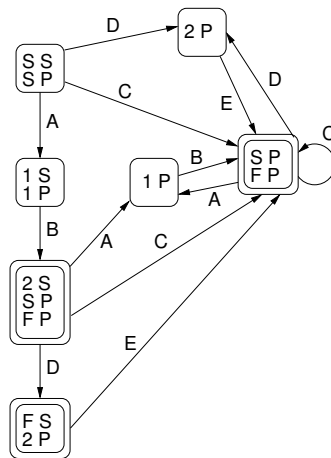


Abbildung 6.4: Deterministischer endlicher Automat aus Spur 2 und Protokollhypothese 1

6.5 Validierung gegen falsche Spuren

Das Vorgehen beim Validieren eines Protokolls gegen falsche Spuren ist dasselbe wie beim Validieren gegen korrekte Spuren. Hier kann jedoch automatisch überprüft werden, ob der als fehlerhaft markierte Knoten genau der ist, der nicht vom Protokollgraph akzeptiert wird. Ist das der Fall, dann bestätigt das Validierungs-Werkzeug den Protokollgraphen. Ist dies nicht der Fall, dann gibt es zwei Möglichkeiten. Entweder wird die fehlerhafte Spur vom Protokoll akzeptiert. In dem Fall wird der Weg zum als falsch markierten Knoten zusammen mit einem entsprechenden Hinweis ausgegeben. Oder sie wird an einer anderen als der falschen Stelle nicht akzeptiert, dann wird der Weg zu dieser Stelle mit entsprechender Meldung ausgegeben.

6.6 Zweigüberdeckung

Bei der Zweigüberdeckung sich folgende Frage: Welche Zweige des Protokollgraphen werden durch die korrekten Spuren abgedeckt und welche nicht?

6 Validierung

Auch für diese Frage kann der deterministische Automat verwendet werden. Zunächst werden alle Spurgraphen und der Protokollgraph in endliche Automaten verwandelt. Dann werden diese Automaten nach oben beschriebenen Verfahren in einen deterministischen endlichen Automaten umgewandelt.

Für jede Kante im deterministischen Automaten werden die entsprechenden Kanten des nicht-deterministischen Protokoll-Automaten markiert. Somit sind am Ende alle Kanten im Protokoll-Automaten markiert, die von den Spuren überdeckt werden.

Die Zuordnung zwischen dem deterministischen Automaten und dem Protokoll-Automaten wird anhand von Abbildung 6.5 erläutert: Von jeder Kante des deterministischen Automaten werden Start- und Zielzustand bestimmt. Für die Kante $(2S, SP, FP \xrightarrow{C} SP, FP)$ sind die Startzustände des Protokollgraphen SP, FP und die Zielzustände ebenfalls SP, FP . Nun werden alle Kombinationen von Startzuständen mit Zielzuständen gebildet, in unserem Beispiel: (SP, SP) , (SP, FP) , (FP, SP) und (FP, FP) . Anschließend wird für jede dieser Kombinationen die entsprechende Kante im Protokoll-Automaten markiert, sofern sie vorhanden ist. Im Beispiel sind die Kanten (SP, SP) und (SP, FP) vorhanden, die Kanten (FP, SP) und (FP, FP) jedoch nicht. Abbildung 6.5 veranschaulicht die Zuordnung der Kante im deterministischen Automaten und im Protokoll-Automaten.

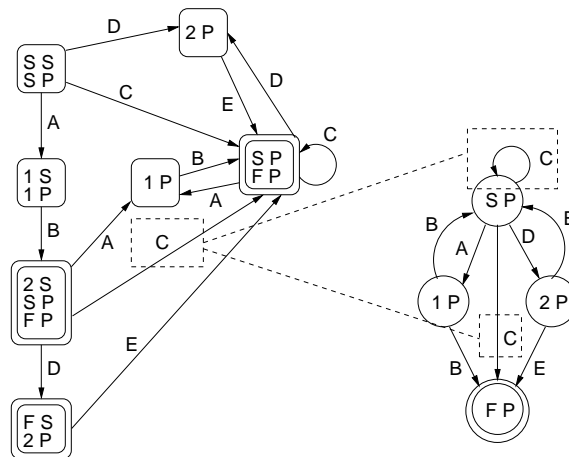


Abbildung 6.5: Zuordnung zwischen deterministischem Automaten und Protokoll-Automaten

Nun ist für den Wartungs-Ingenieur jedoch nicht die Überdeckung des Protokoll-Automaten, sondern die des ursprünglichen Protokollgraphen interessant. Um vom Protokollautomat auf den ursprünglichen Graphen schließen zu können, muss schon bei der Umwandlung in den endlichen Automaten eine Abbildung angelegt werden, die jeder Kante des endlichen Automaten eindeutig den entsprechenden Weg des ursprünglichen Graphen zuordnet. Diese Abbildung wird in Abbildung 6.6 verdeutlicht. Die Kante im Protokoll-Automat, die mit 1 markiert ist (SP,SP) , entspricht dem Weg im Protokollgraph, der mit $1a$ bis $1f$ gekennzeichnet ist. Das Gleiche gilt entsprechend für die mit 2 markierte Kante (SP,FP) und den Weg $2a$ bis $2f$ im Protokollgraph.

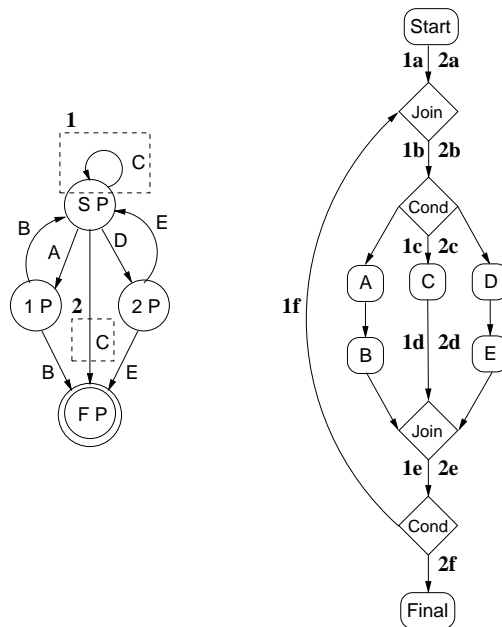


Abbildung 6.6: Zuordnung zwischen Protokoll-Automat und ursprünglichem Protokollgraph

Auf diese Art kann sich der Wartungs-Ingenieur eine Liste aller Knoten und Kanten eines Protokollgraphen ausgeben lassen, die von Spurgraphen überdeckt werden.

6.7 Obligatorische und verbotene Vorgänger und Nachfolger

Wie schon in Unterabschnitt 3.2.3 auf Seite 62 beschrieben, gibt es weitere Informationen in den Spurgraphen, die zur Herleitung von Protokollen eine Rolle spielen:

1. Die obligatorischen Vorgänger jeder primitiven Operation o , d.h. alle primitiven Operationen, die in allen Spuren auf allen Wegen vom Startknoten zu o vorkommen.
2. Die verbotenen Vorgänger jeder primitiven Operation o , d.h. alle primitiven Operationen, die in keinem Weg einer Spur vom Startknoten zu o vorkommen.
3. Die obligatorische Nachfolger jeder primitiven Operation o , d.h. alle primitiven Operationen, die in allen Spuren auf allen Wegen von o zum Endknoten vorkommen.
4. Die verbotenen Nachfolger ergeben sich direkt aus den verbotenen Vorgängern, wie schon in Unterabschnitt 3.2.3 auf Seite 62 erwähnt.

Betrachtet man die Beschreibung von Punkt 1 und Punkt 3 (obligatorische Vorgänger und Nachfolger), dann fällt die Ähnlichkeit zur Dominanz und Postdominanz auf.

6.7.1 Dominanz und Postdominanz

Definition 6.7.1 (Dominanz) Ein Knoten A dominiert einen Knoten B , wenn A auf allen Pfaden vom Startknoten zu B liegt.

Definition 6.7.2 (Post-Dominanz) Ein Knoten A post-dominiert einen Knoten B , wenn A auf allen Pfaden von B zum Endknoten liegt.

Die Post-Dominanz entspricht der Dominanz des Graphen mit umgedrehten Kanten und vertauschten *Start*- und *Final*-Knoten.

Veranschaulichen wir nun Dominanz und Post-Dominanz anhand eines Beispiels. In Abbildung 6.7 auf der nächsten Seite, Teil (a), ist ein Graph dargestellt. In diesem Graph wird u.a. Knoten 5 von Knoten 2 dominiert, Knoten 1 wird von Knoten 4 post-dominiert. Man kann die Dominanz-

6.7 Obligatorische und verbotene Vorgänger und Nachfolger

und Post-Dominanz-Beziehungen der Knoten jeweils in einem Baum darstellen. (b) drückt dabei die Dominanz-Beziehungen des Graphen aus, (c) stellt die Post-Dominanz-Beziehungen dar. So ist z.B. in (b) zu sehen, dass der Knoten 6 nur vom *Start*-Knoten dominiert wird, denn 6 ist sowohl über 2 als auch über 3 zu erreichen.

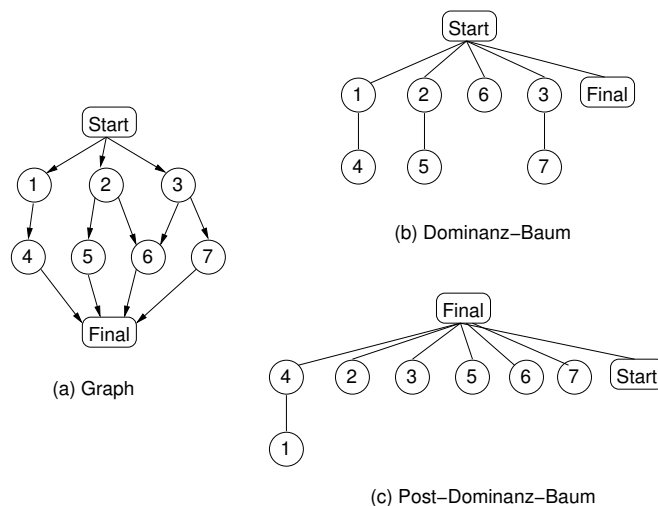


Abbildung 6.7: Beispiel für Dominanz und Post-Dominanz

Stellen wir nun die Verbindung zu den obligatorischen Vorgängern und Nachfolgern her. Die Gemeinsamkeit von Dominanz und obligatorischen Vorgängern liegt darin, dass in beiden Fällen nach Knoten gesucht wird, die auf allen Pfaden vom Startknoten zu einem bestimmten Knoten x liegen (dasselbe gilt für obligatorische Nachfolger und Post-Dominanz). Allerdings unterscheiden sie sich darin, dass bei der Dominanz jeder Knoten einzeln untersucht wird. Bei den obligatorischen Vorgängern hingegen sollen alle Pfade vom Startknoten zu allen Knoten mit dem selben Namen untersucht werden. Die Namen stellen ja, wie in Definition 2.5.3 auf Seite 34 beschrieben, die Namen der Komponentenroutinen dar.

Nun ist es wünschenswert, das ursprüngliche Problem („finde die obligatorischen Vorgänger und Nachfolger aller Komponentenroutinen in Bezug auf alle Spurgraphen“) auf die Dominanz zurückzuführen, um auf bekannte Algorithmen zurückgreifen zu können.

6 Validierung

Dies wird für eine Komponentenroutine x folgendermaßen bewerkstelligt:

1. Man fasst alle Spuren zu einem Graphen zusammen, indem man alle Startknoten durch einen einzelnen Startknoten und alle Endknoten durch einen einzelnen Endknoten ersetzt. Dabei erhält der neue Startknoten alle ausgehenden Kanten der ursprünglichen Startknoten, dasselbe gilt für die eingehenden Kanten der Endknoten.
2. Von diesem Graph erstellt man den Dominanz- und den Post-Dominanz-Baum. Ein effizienter Algorithmus zur Erstellung dieser Bäume findet sich in [Tar79].
3. Nun bestimmt man die Namen der Dominatoren von jedem Knoten mit Namen x und bildet aus diesen die Schnittmenge. Damit hat man alle obligatorischen Vorgänger von x . Auf dieselbe Weise bekommt man mit Hilfe der Post-Dominatoren die obligatorischen Nachfolger.

Betrachten wir dies am Beispiel der Spuren aus Abschnitt 3.2 auf Seite 57. Diese sind in Abbildung 6.8 auf der nächsten Seite, Teil (a), nochmals dargestellt, wobei die Knoten zusätzlich zu ihren Namen Nummern erhalten haben, die ihre Zugehörigkeit zur jeweiligen Spur darstellt. A , B , C , D und E seien weiterhin die Namen im Sinne der Beschriftungsfunktion l aus Definition 2.5.1 auf Seite 34 und Definition 2.5.3 auf Seite 34. (b) zeigt die Spurgraphen, nachdem *Start*- und *Final*-Knoten wie gerade beschrieben zusammengefasst wurden.

In Abbildung 6.9 auf der nächsten Seite sind der Dominanz- und der Post-Dominanz-Baum für den Graphen dargestellt, der sich aus den zusammengefassten Spuren ergibt. Da die Spuren dieses Beispiels keine Verzweigungen enthalten, ist es nicht weiter verwunderlich, dass der Dominanz-Baum, abgesehen vom *Final*-Knoten, dem Originalgraphen gleicht. Entsprechendes gilt für den Post-Dominanz-Baum, wenn man im Originalgraphen die Kanten umdreht und *Start*- und *Final*-Knoten vertauscht.

Führen wir nun Schritt 3 des oben vorgestellten Verfahrens für die Bestimmung der obligatorischen Vorgänger beispielhaft auf alle Knoten mit Namen C aus. Tabelle 6.1 auf Seite 112 zeigt die Ergebnisse der Teilschritte, die dabei auszuführen sind.

Daraus ergeben sich A und B als obligatorische Vorgänger von C . Dieses Ergebnis stimmt mit dem überein, das bei der Einführung des hypothesengetriebenen Ansatzes dargestellt wurde, siehe Tabelle 3.2 auf Seite 64.

6.7 Obligatorische und verbotene Vorgänger und Nachfolger

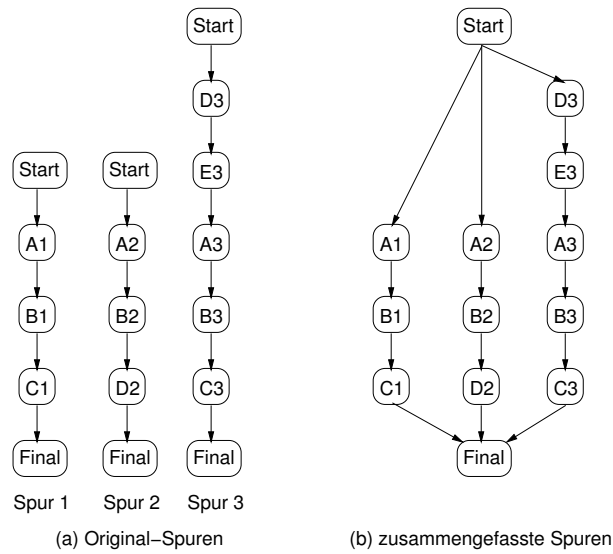


Abbildung 6.8: Spurgraphen und Vorbereitung für Dominanz-Analyse

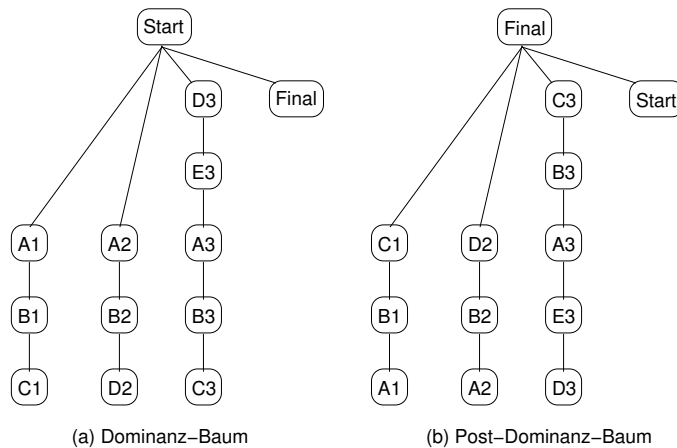


Abbildung 6.9: Dominanz und Post-Dominanz für die zusammengefassten Spuren (b) in Abbildung 6.8

Als Beispiel für obligatorische Nachfolger nehmen wir *A*. Die Knoten mit diesem Namen sind *A1*, *A2* und *A3*. Tabelle 6.2 auf der nächsten Seite zeigt wiederum die Teilergebnisse.

6 Validierung

Knoten	Dominatoren	Namen d. Dom.	Schnittmenge
C1	B1, A1	B, A	A, B
C3	B3, A3, E3, D3	B, A, E, D	

Tabelle 6.1: Bestimmung der obligatorischen Vorgänger von C

Knoten	Post-Dom.	Namen d. Post-Dom.	Schnittmenge
A1	B1, C1	B, C	B
A2	B2, D2	B, D	
A3	B3, C3	B, C	

Tabelle 6.2: Bestimmung der obligatorischen Nachfolger von A

Der einzige obligatorische Nachfolger von A ist also B . Auch dies stimmt mit den Ergebnissen aus Unterabschnitt 3.2.3 auf Seite 62 überein.

Damit ist klar geworden, dass sich die Bestimmung obligatorischer Vorgänger und Nachfolger auf die Dominanz- und Post-Dominanz-Analyse zurückführen lässt. Damit kann auf bekannte, effiziente Algorithmen zurückgegriffen werden.

6.7.2 Verbotene Vorgänger und Nachfolger

Die Bestimmung von verbotenen Vorgängern lässt sich nicht auf Dominanz oder Post-Dominanz zurückführen. Mit Hilfe der gerade vorgestellten Methode zur Zusammenfassung von Spurgraphen können verbotene Vorgänger jedoch recht einfach bestimmt werden.

Man hält die Namen aller Komponentenroutinen in einer Menge und läuft vom *Start*-Knoten aus alle Wege zum untersuchten Knoten ab. Schleifen müssen jeweils einmal vollständig durchlaufen werden. Aus der Menge werden die Namen aller besuchten Knoten entlang dieser Wege abgestrichen, solange bis entweder die Menge leer ist oder alle Wege abgelaufen wurden. Die Komponentenroutinen, deren Namen anschließend noch in der Menge enthalten sind, sind die verbotenen Vorgänger vom untersuchten Knoten.

7 Schlüsse und Schluss

7.1 Ergebnisse dieser Arbeit

In dieser Arbeit, die im Rahmen des Reengineering-Projektes Bauhaus [Bau04] stattfand, wurden zwei Ansätze zur werkzeuggestützten Herleitung von Protokollen aus Spuren untersucht: Der induktive Ansatz und der hypothesengetriebene.

Dazu wurde zunächst die Arbeit von Heiber [Hei00] aufgegriffen, die den Prozess des induktiven Ansatzes formuliert, den ersten Schritt ausarbeitet und umsetzt und die weiteren Schritte beschreibt. Nun wurde untersucht, inwiefern die Ergebnisse von Heiber erweitert und verändert werden müssen, um sie auf die aktuelle Repräsentation von extrahierten Spurgraphen in Bauhaus anzupassen und welchen Einschränkungen die Spurgraphen unterworfen sind, insbesondere bei der Modellierung von Bedingungen.

Dann wurde das Vorgehen zur Weiterführung des induktiven Ansatzes ausgearbeitet. Dabei wurde klar, dass die Weiterführung dieses Ansatzes mit erheblichen Risiken verbunden ist, weil einerseits die Spuren zunächst bereinigt und von Rekursionen befreit werden müssen, und andererseits ein Graphtransformationssystem implementiert oder eingebunden werden muss.

Dem induktiven Ansatz wurde anschließend der hypothesengetriebene gegenübergestellt. Beim hypothesengetriebenen Ansatz beginnt der Wartungs-Ingenieur damit, seine Hypothese vom Protokoll der untersuchten Komponente als Graph zu formulieren und validiert diese dann iterativ gegen Spurgraphen. Zur Validierung können ihm verschiedene Werkzeuge zur Verfügung gestellt werden: Validierung gegen korrekte und falsche Spuren, Zweigüberdeckung des Protokollgraphen durch korrekte Spuren, Bestimmung der obligatorischen und verbotenen Vorgänger und Nachfolger aller Komponentenroutinen in korrekten Spuren.

7 Schlüsse und Schluss

Nach einer Gegenüberstellung beider Ansätze wurde entschieden, den hypothesengetriebenen Ansatz in dieser Arbeit weiterzuverfolgen.

Des Weiteren wurde beschrieben, auf welche Art Spuren vorbereitet werden müssen, um sie zur Herleitung und Validierung von und mit Protokollen verwenden zu können. Die wichtigste Anforderung dabei ist, dass Spurgraphen und Protokollgraphen reguläre Sprachen definieren müssen, was sie nicht tun, wenn rekursive Routinenaufrufe vorkommen. Es wurde ein Verfahren vorgestellt und implementiert, das rekursive Aufrufe in Spurgraphen durch Schleifenkonstrukte ersetzt, so dass die Sprache des entstehenden Spurgraphen regulär und zugleich eine Obermenge der Sprache des ursprünglichen Graphen ist. Die Folgen dieser Spracherweiterung für den hypothesengetriebenen Ansatz sowie für die Validierung von Spuren gegen Protokolle wurden erläutert.

Abschließend wurde die Funktionsweise der verschiedenen Werkzeuge zur Validierung beschrieben, die im Rahmen dieser Arbeit implementiert worden sind.

7.2 Offene Punkte

An folgenden Punkten kann angesetzt werden, wenn die Aufgabe dieser Arbeit weitergeführt werden soll:

Der praktische Nutzen des hypothesengetriebenen Ansatzes und die Effizienz der Validierungswerkzeuge sollte evaluiert werden, z.B. durch eine Fallstudie, die anhand von realem Quellcode durchgeführt wird.

Wie an verschiedenen Stellen beschrieben, gibt es keine Repräsentation von Bedingungen in Spurgraphen, was zu Schwierigkeiten bei der Validierung führt. Dies könnte auf zwei Arten gelöst werden: Entweder es wird eine solche Repräsentation eingebaut, was sich mit dem nächsten Punkt (Datenflussanalyse) kombinieren ließe. Oder man erstellt Werkzeuge, die es dem Wartungs-Ingenieur erlauben, Bedingungen als äquivalent zu kennzeichnen.

In einem weiteren Schritt könnte man die inneren Abhängigkeiten der Komponente und den Datenfluss mitberücksichtigen, um auch Fälle formulieren zu können wie „Wenn in einer Hashtabelle der Wert des Schlüssels x abgefragt wird, muss vorher abgefragt werden, ob die Hashtabelle einen Wert zu Schlüssel x enthält“.

Eine mögliche Erweiterung der Bestimmung von obligatorischen und verbotenen Vorgängern und Nachfolgern könnte dem Wartungs-Ingenieur helfen, falsche Spuren aufzudecken. Gibt man nämlich die Vorgänger und Nachfolger nicht absolut, sondern pro untersuchter Spur an, dann kann z.B. der Fall auftreten, dass 95% der Spuren A als obligatorischen Vorgänger von B identifizieren, 5% jedoch nicht. Diese 5% erscheinen verdächtig und können durch eine manuelle Analyse entweder als falsch deklariert oder als korrekt bestätigt werden.

7.3 Rückblick und Erfahrungen

Als nach sechs Wochen schließlich klar wurde, dass es weder möglich ist, ein Graphtransformationssystem im Rahmen dieser Arbeit zu implementieren, noch ein vorhandenes einzubinden, sah dies zunächst nach einem Unglücksfall aus. Im Nachhinein betrachtet erweist es sich jedoch als Glücksfall, weil dieser Umstand die Geburt des hypothesengetriebenen Ansatzes möglich gemacht hat, der dem induktiven Ansatz zumindest gleichgestellt ist.

Einziger Wermutstropfen ist, dass aufgrund der Neu-Ausrichtung des Themas während der Arbeit und auch weil sich die Rekursionsauflösung in der Implementierung als komplexer erwiesen hat, als zunächst ersichtlich war, zum Schluss die Zeit für eine Evaluierung des hypothesengetriebenen Ansatzes und der Werkzeuge, die im Rahmen dieser Arbeit implementiert wurden, gefehlt hat.

Alles in allem war diese Arbeit intensiv, lehrreich und wertvoll. Die dabei gewonnenen Erfahrungen, insbesondere über Entwurf und Implementierung von Algorithmen, verleihen meinem Studium den letzten Schliff.

7.4 ... und Schluss.

„Jetzt fängt alles erst richtig an. Was du bisher erfahren hast, das war doch nur eine Vorahnung vom Anfang und längst noch nicht alles.“ [Bem03]

7 Schlüsse und Schluss

A Glossar

Dieses Glossar definiert Begriffe wichtige Begriffe, die in der vorliegenden Arbeit verwendet werden. In den meisten Fällen sind deutsche und englische Begriffe angegeben, da Definitionen und Algorithmen in der vorliegenden Arbeit englische Bezeichner verwenden. In Fällen, in denen eine Übersetzung nicht sinnvoll erscheint, wird entweder nur der deutsche oder nur der englische Begriff angegeben.

ADO – Abstraktes Datenobjekt (*Abstract Data Object*) Untereinheit eines Programms (z.B. Modul), die eine globale Variable hält und Routinen exportiert, die zu dieser globalen Variablen gehören.

ADT – Abstrakter Datentyp (*Abstract Data Type*) Untereinheit eines Programms (z.B. Modul), die einen Typ und Routinen exportiert, die zu diesem Typ gehören.

Benutzercode (-) Code, der eine \rightarrow *Komponente* benutzt. Benutzercode steht im Gegensatz zu \rightarrow *Komponentencode*.

Benutzerroutinen (-) Routinen, die zu \rightarrow *Benutzercode* gehören. Benutzerroutinen stehen im Gegensatz zu \rightarrow *Komponentenroutinen*.

Black Box Understanding (-) Analyse von \rightarrow *Benutzercode*, um daraus Schlüsse über eine \rightarrow *Komponente* zu ziehen, im Gegensatz zu \rightarrow *Glass Box Understanding*. Siehe auch [Kos02].

Fall-Eins-Routine (*Case-One-Routine*) Eine Routine in einem \rightarrow *Spurgraph*, deren Rumpf und damit *Entry*- und *Return*-Knoten ebenfalls im Spurgraph vorhanden sind. Siehe auch Unterabschnitt 2.4.5 auf Seite 24.

Fall-Zwei-Routine (*Case-Two-Routine*) Eine Routine in einem \rightarrow *Spurgraph*, in der ein Haldenobjekt allokiert wurde, weswegen ihr Rumpf erst von der Allokierung an im Spurgraph enthalten ist und damit kein *Entry*-, sondern nur der *Return*-Knoten Teil des Spurgraph ist. Siehe auch Unterabschnitt 2.4.5 auf Seite 24.

Fall-Drei-Routine (*Case-Three-Routine*) Eine Routine in einem \rightarrow Spurgraph, deren Rumpf nicht im Spurgraph enthalten ist. Siehe auch Unterabschnitt 2.4.5 auf Seite 24.

Glass Box Understanding (-) Analyse von \rightarrow Komponentencode, um daraus Schlüsse über die \rightarrow Komponente zu ziehen, im Gegensatz zu \rightarrow Black Box Understanding. Siehe auch [Kos02].

Haldenobjekt (*heap object*) Instanz einer \rightarrow Komponente, deren Lebensdauer zur Laufzeit bestimmt werden kann, im Gegensatz zu \rightarrow Stapelobjekt.

Komponente (*component*) Im Sinne dieser Arbeit: \rightarrow ADTs und \rightarrow ADOs. Im weiteren Sinne u.a. auch: Module und Klassen.

Komponentenabhängige Muster (-) Muster zur Vereinfachung von Spurgraphen, nur anwendbar auf Spuren einer speziellen Komponente, im Gegensatz zu \rightarrow Komponentenunabhängigen Mustern.

Komponentenunabhängige Muster (-) Muster zur Vereinfachung von Spurgraphen, anwendbar auf Spuren von beliebigen Komponenten, wobei die Vorbedingungen zum Anwenden der Muster durch den Wartungs-Ingenieur bestätigt werden müssen. Steht im Gegensatz zu \rightarrow Komponentenabhängigen Mustern.

Komponentencode (-) Code, der eine \rightarrow Komponente implementiert, im Gegensatz zu \rightarrow Benutzercode.

Komponentenroutinen (-) Routinen, die zu \rightarrow Komponentencode gehören, im Gegensatz zu \rightarrow Benutzerrountinen. Sie sind Teil der \rightarrow Primitive Operationen einer Komponente.

Operationen (*operations*) Routinenaufrufe und Lese- und Schreibzugriffe auf \rightarrow Stapel- und \rightarrow Haldenobjekte.

Primitive Operationen (*primitive operations*) \rightarrow Operationen, die zu einer \rightarrow Komponente gehören. Dabei wird unterschieden zwischen Routinen und Zugriffen auf Teile der exportierten Typen oder globalen Variablen der Komponente.

RFG (*Resource Flow Graph*) Graphformat aus dem Bauhaus-Projekt. RFG-Graphen werden zur Speicherung und visuellen Darstellung von Informationen verwendet, die über ein untersuchtes Software-System mit Hilfe von Bauhaus-Analysewerkzeugen gewonnen wurden.

Slicing (-) Technik zum Herausfinden derjenigen Teile im Quellcode, die durch eine bestimmte Anweisung beeinflusst werden (Forward Slicing) bzw. von denen eine bestimmte Anweisung beeinflusst wird (Backward Slicing). Siehe auch [Wei84] und [Hor90].

Spur (*trace*) Diejenigen \rightarrow Operationen von \rightarrow Benutzercode die auf eine \rightarrow Komponente zugreifen.

Spurgraph (*trace graph*) Ein Graph, der eine \rightarrow Spur repräsentiert. In dieser Arbeit immer in Form eines \rightarrow RFG.

Stapelobjekt (*heap object*) Globale oder lokale Variable. Die Lebensdauer eines Stapelobjekts ist durch den Gültigkeitsbereich seiner Variable bestimmt. Gegenteil von \rightarrow Haldenobjekt.

A Glossar

B Kurzeinführung in IDEF0

IDEF-0-Diagramme [IDE04] entsprechen SADT-Diagrammen [Ros77][Ros84], nur dass hier Prozesse beschrieben werden. Eine Aktivität wird in einer Box dargestellt. Boxen haben eingehende und ausgehende Kanten. Die Seite, an der eine Kante an einer Box angebracht ist, definiert die Bedeutung der Kante (siehe Abb. B.1):

- Links angebrachte Kanten sind immer eingehend. Sie repräsentieren die Eingabedaten, die während der Funktion manipuliert werden.
- Rechts angebrachte Kanten sind immer ausgehend. Sie repräsentieren die Ausgabedaten der Funktion.
- Oben angebrachte eingehende Kanten stellen Daten dar, die die Funktion steuern.
- Unten angebrachte eingehende Kanten stehen für Ressourcen, die für die Funktion benötigt werden. In dieser Arbeit werden sowohl Software-Werkzeuge als auch der Wartungs-Ingenieur als Ressource betrachtet.

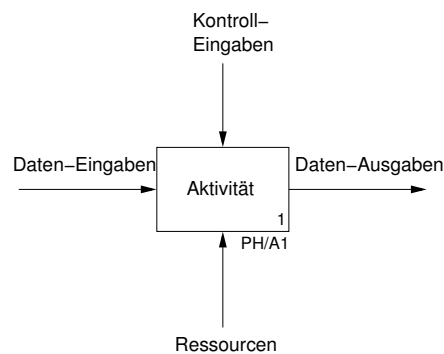


Abbildung B.1: Beispiel einer Aktivität eines IDEF0-Diagramms

Diagrammen können in hierarchischer Beziehung zueinander stehen, so dass eine Aktivität eines Diagramms durch ein anderes Diagramm repräsentiert wird, das diese Aktivität genauer spezifiziert.

B Kurzeinführung in IDEF0

Die Benennung von IDEF0-Diagrammen ist eindeutig. Es gibt üblicherweise ein Diagramm der obersten Ebene, das den Namen *PH/A-0* trägt. Dessen Aktivität wird durch ein Diagramm mit Namen *PH/A0* repräsentiert, welches weitere Aktivitäten enthält. Die Aktivitäten werden dabei – sofern es geht – diagonal von links oben nach rechts unten angeordnet und auch in dieser Reihenfolge pro Diagramm durchnummeriert. Um Aktivitäten global eindeutig anzugeben, fügt man den Namen des Diagramms an, in dem sie enthalten sind. So trägt z.B. die 4. Aktivität des Diagramms *PH/A1* die eindeutige Nummer *PH/A14*.

Die Nummerierung der Aktivitäten soll dabei jedoch keine vorgegebene Ausführungsreihenfolge angeben. Aktivitäten können in beliebiger Reihenfolge und auch parallel ausgeführt werden, solange alle Kontroll- und Dateneingaben zur Verfügung stehen, die zur Ausführung einer Aktivität notwendig sind.

Literaturverzeichnis

- [Bau93] BAUER, F. L.: *Software Engineering – wie es begann*. Informatik Spektrum, 16(5):259–260, Oktober 1993.
- [Bau04] *Projekt Bauhaus*. <http://www.informatik.uni-stuttgart.de/iste/ps/bauhaus>, 2004.
- [Bem03] BEMMANN, H.: *Stein und Flöte und das ist noch nicht alles*. Piper, 2003.
- [But00] BUTKEVICH, S., RENEDO, M., BAUMGARTNER, G. UND YOUNG, M.: *Compiler and Tool Support for Debugging Object Protocols*. In Proceedings of the Eighth International Symposium on Foundations of Software Engineering for Twenty-First Century Applications, Seiten 50–59, 2000.
- [Coo98] COOK, J. E., WOLF, A. L.: *Discovering Models of Software Processes from Event-Based Data*. ACM Transactions on Software Engineering and Methodology, 7(3):215–249, July 1998.
- [Cor00] CORBETT, J. C.: *Bandera: Extracting Finite-State Models from Java Source Code*. In Proceedings of the International Conference on Software Engineering, Seiten 439–448, 2000.
- [COR04] *OMG's CORBA Website*. <http://www.corba.org>, 2004.
- [Das02] DAS, M., LERNER, S., SEIGLE, M.: *ESP: Path-Sensitive Program Verification in Polynomial Time*. In: *SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2002.
- [Ehr91] EHRIG, H., KREOWSKI, H.-J., ROZENBERG, G. (Herausgeber): *Graph Grammars and Their Application to Computer Science*. Lecture Notes in Computer Science. Springer-Verlag, 1991. 4th International Workshop, Bremen, Germany, March 1990, Proceedings.

- [Eis02] EISENBARTH, T., KOSCHKE, R., VOGEL, G.: *Static Trace Extraction*. In Proceedings of the Working Conference on Reverse Engineering, October 2002.
- [Eis04] EISENBARTH, T., KOSCHKE, R., VOGEL, G.: *Static Trace Extraction for Programs with Pointers*. Journal of Systems and Software, 2004. To Appear.
- [End79] ENDE, M.: *Die unendliche Geschichte*. Thienemann Verlag, 1979.
- [Fis03] FISKIO-LASSETER, J., YOUNG, M.: *Flow equations as a generic programming tool for manipulation of attributed graphs*. In: *SIGPLAN-SIGSOFT Workshop on Programming Analysis for Software Tools and Engineering*, Seiten 69–76. ACM, 2003.
- [Gen04] *GenSet – Pacemaker Project*. <http://www.cs.uoregon.edu/research/perpetual/dasada/Software/GenSet/>, 2004.
- [Gö88] GÖTTLER, H.: *Graphgrammatiken in der Softwaretechnik*. Springer-Verlag, 1988.
- [GXL04] *Graph eXchange Language*. <http://www.gupro.de/GXL>, 2004.
- [Haa04] HAAK, D.: *Werkzeuggestützte Herleitung von Protokollen*. Diplomarbeit Nr. 2135, Institut für Softwaretechnologie, Universität Stuttgart, 2004. Noch nicht veröffentlicht.
- [Han00] HANSEN, SVEN: *Extraktion statischer Traces zur Wiedergewinnung von Protokollen*. Studienarbeit Nr. 1768, Institut für Informatik, Universität Stuttgart, 2000.
- [Hei00] HEIBER, TIMO: *Semi-automatische Herleitung von Komponentenprotokollen aus statischen Verwendungsmustern*. Diplomarbeit Nr. 1822, Institut für Informatik, Universität Stuttgart, 2000.
- [Hop79] HOPCROFT J. E., ULLMAN, J.D.: *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
- [Hor90] HORWITZ, S., REPS, T., BINKLEY, D.: *Interprocedural Slicing Using Dependence Graphs*. ACM Transactions on Programming Languages and Systems, 12(1):26–60, January 1990.
- [IDE04] *IDEF0 Overview*. <http://www.idef.com/idef0.html>, 2004.

- [Jer97a] JERDING, D. F., STASKO, JOHN T., BALL, THOMAS: *Visualizing Interactions in Program Executions*. In: *Proceedings of the International Conference on Software Engineering*. ACM, 1997.
- [Jer97b] JERDING, D. F., S. RUGABER: *Using Visualization for Architectural Localization and Extraction*. In: *Proceedings of the Working Conference on Reverse Engineering*. IEEE Computer Society Press, 1997.
- [Kos00] KOSCHKE, R.: *Atomic Architectural Component Detection for Program Understanding and System Evolution*. Doktorarbeit, University of Stuttgart, Universitätsstrasse 38, 70569 Stuttgart, Germany, 2000.
- [Kos02] KOSCHKE, R., ZHANG, Y.: *Component Recovery, Protocol Recovery and Validation*. In 3. Workshop Software-Reengineering, Bad Honnef (10./11.Mai 2001), Fachberichte Informatik, Universität Koblenz-Landau, 1/2002:73–76, January 2002.
- [Mor98] MORGAN, R.: *Building an Optimizing Compiler*. Digital Press, 1998.
- [Ole92] OLENDER, K. M., OSTERWEIL, L. J.: *Interprocedural Static Analysis of Sequencing Constraints*. ACM Transactions on Software Engineering and Methodology, 1(1):21–52, January 1992.
- [Par72] PARNAS, D. L.: *On the Criteria to be Used in Decomposing Systems into Modules*. Communications of the ACM, 5(12):1053–1058, December 1972.
- [Rab59] RABIN, M.O., SCOTT, D.: *Finite automata and their decision problems*. IBM J. Res. Devel., 3, 1959.
- [Ros77] ROSS, D., SCHOMAN, K.: *Structured Analysis for Requirements Definition*. IEEE Computer Society Transactions on Software Engineering, 3(1):6–15, January 1977.
- [Ros84] ROSS, D.: *Applications and Extensions of SADT*. IEEE Computer, 18(4):25–35, April 1984.
- [Sch91] SCHÜRR, A.: *Operationales Spezifizieren mit programmierten Graphersetzungssystemen*. Deutscher Universitäts-Verlag, 1991.
- [Sed92] SEDGEWICK, R.: *Algorithms*. Addison-Wesley, 2 Auflage, 1992.

Literaturverzeichnis

- [Tar79] TARJAN, R. E., LENGAUER, T.: *A Fast Algorithm for Finding Dominators in a Flowgraph*. ACM Transactions on Programming Languages and Systems, 1(1):121–141, July 1979.
- [Vra03] VRANDEČIĆ, ZDENKO: *XML4Ada95*. Diplomarbeit Nr. 2093, Institut für Softwaretechnologie, Universität Stuttgart, 2003.
- [Wei84] WEISER, MARK: *Program Slicing*. IEEE Computer Society Transactions on Software Engineering, 10(4), July 1984.
- [Wha02] WHALEY, J., MARTIN, M. C., LAM, M. S.: *Automatic Extraction of Object-Oriented Component Interfaces*. In: *Proceedings of the International Symposium on Software Testing and Analysis*, July 2002.

Erklärung

Hiermit versichere ich, diese Arbeit selbstständig verfasst und nur die angegebenen Quellen benutzt zu haben.

(Dietrich Haak)

