Studiengang: Informatik

Prüfer: Prof. Dr. Erhard Plödereder

Betreuer: Dipl.-Inf. Jörg Czeranski,
Prof. Dr. Rainer Koschke

Diplomarbeit Nr. 2212

Kontrollflussanalyse objektorientierter Programme

Markus Müller

Institut für Softwaretechnologie
Universität Stuttgart
Universitätsstraße 38
70569 Stuttgart

begonnen am: 15. Juli 2004

beendet am: 14. Januar 2005

CR-Klassifikation: F.3.2, F.3.3

Ich möchte Murphy für sein unerschütterliches Gesetz danken. Ausserdem Newton für die Schwerkraft, einem Unbekannten für die Erfindung des Konfigurationsmanagements und Toshiba für die unglaublich schnelle Ersatzteillieferung.

Mein Dank gilt außerdem der gesamten Programmiersprachen-Abteilung für die gute Betreuung und die vielen Diskussionen, Gespräche und Ideen, sowie allen Personen, die mich unterstützt haben!!!

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Das Projekt bauhaus . . . 1
1.2 Aufgabenstellung . . . 1
1.3 Grundlagen . . . 2
1.4 Anforderungen an den Leser . . . 3
1.5 Gliederung des Dokuments . . . 3
1.6 Notation . . . 3
1.7 Phasen der Diplomarbeit . . . 4
1.8 Begriffsdefinitionen . . . 5

2 Ausnahmebehandlung in C++ und Java **6**
2.1 Sprachkonstrukte . . . 6
2.2 Eintreten in einen `try`-Block . . . 7
2.3 Verlassen eines `try`-Blocks . . . 7
2.3.1 Erreichen des Blockendes . . . 7
2.3.2 Auftreten einer Ausnahme . . . 7
2.3.3 Explizite Anweisung zum Verlassen des Blocks . . . 7
2.4 Ausnahmetypen . . . 8
2.5 Bestimmung des Handlers für eine Ausnahme . . . 8
2.6 Ausführung des `finally`-Blocks . . . 8
2.7 Verschwinden von Ausnahmen . . . 9
2.8 Propagierte Ausnahmetypen in der Routinendeklaration . . . 10
2.9 Sprachspezifische Besonderheiten . . . 11
2.9.1 Laufzeitausnahmen . . . 11
2.9.2 Das globale Ausnahmeobjekt in C++ . . . 11

3 Die IML **12**
3.1 Bisherige Version der IML . . . 12
3.2 Generierung des CFG . . . 13
3.2.1 Das Werkzeug `iml2cfg` . . . 13
3.2.2 Backpatching unbedingter Sprünge . . . 13
3.2.3 Optimierung des Zwischenergebnisses . . . 13
3.3 Der Kontrollflussgraph . . . 14
3.3.1 Beispiel intraprozeduraler CFG . . . 18
3.4 Erweiterung der IML . . . 18
3.4.1 Einschränkungen durch die Erweiterungen . . . 18
3.4.2 Grundblöcke . . . 20
3.4.3 Routine-Knoten . . . 21
3.5 Implizite Ausnahmen . . . 21
3.6 Der beliebige Ausnahmetyp . . . 23
3.6.1 Einschränkung der möglichen Typen . . . 23
3.6.2 Auftreten des beliebigen Ausnahmetyps . . . 24

4 Kontrollfluss und -abhängigkeit **26**
4.1 Kontrollfluss . . . 26
4.1.1 Routinenaufrufe . . . 26
4.1.2 Modellierung der `finally`-Blöcke . . . 26
4.1.3 Zusammenspiel von `catch` und `finally` . . . 27
4.2 Kontrollabhängigkeit . . . 27
4.2.1 Künstliche Hilfsknoten und -kanten . . . 29
4.2.2 Unzulänglichkeit des bisherigen Algorithmus . . . 29
4.2.3 Kontextsensitive Berechnung der Postdominanz . . . 30
4.2.4 Berechnung kontrollabhängiger Knoten . . . 31
4.2.5 Ausnahmen . . . 32
4.2.6 Interprozedurale Kontrollabhängigkeit . . . 32
4.2.7 Kontrollabhängigkeit bei Aufrufen . . . 37

5 Konzeption **39**
5.1 Ablauf der Analyse . . . 39
5.2 Automatische Zerstörung von Stackobjekten in `throw`-Anweisungen . 40
5.3 Anlegen und Entfernen von Grundblöcken . . . 41
5.4 Optimierung von Grundblöcken . . . 42
5.5 Temporäre Hilfskanten und -knoten . . . 42
5.5.1 Erreichbarkeit von Ausnahme-Handlern . . . 42
5.5.2 Erreichbarkeit des Local Super Exit . . . 42
5.5.3 Eindeutiger Exit-Knoten einer Routine . . . 43
5.6 Bestimmung nicht terminierender Routinen . . . 43
5.7 Entfernen unerreichbarer Kanten . . . 44
5.8 Entfernen unerreichbarer Exceptional_Exit_Basic_Blocks . . . 44
5.9 Typbestimmung einer Ausnahme . . . 45
5.9.1 Besonderheiten der IML-Modellierung . . . 45
5.9.2 Mögliche Argumente . . . 45
5.10 Rethrow von Ausnahmen . . . 47
5.11 Behandlung geworfener Ausnahmen . . . 47
5.11.1 Explizite Ausnahmen innerhalb einer Routine . . . 47
5.11.2 Modellierung propagierter Ausnahmen . . . 48
5.11.3 Iterative Analyse propagierter Ausnahmen . . . 49
5.12 Bestimmung möglicher Aufrufziele . . . 49
5.12.1 Polymorphe Aufrufe . . . 49
5.12.2 Einschränkung der Objekttypen . . . 50
5.12.3 Konservative Zielbestimmung von Funktionszeigern . . . 52
5.13 Die Konfigurationsdatei für benutzerdefinierte Deklarationen . . . 56
5.13.1 Motivation . . . 56
5.13.2 Aufbau und Inhalt . . . 56
5.13.3 Externe Instanziierung von Objekten . . . 56
5.13.4 Fehlerfälle und deren Behandlung . . . 57
5.13.5 Eingesetzte Werkzeuge . . . 57
5.14 Fehler im C++ Frontend . . . 57
5.14.1 Mehrfache Referenz auf Entity_L_Value . . . 57
5.15 Ausnahmen im Überblick: `cfg2html` . . . 58
5.16 Graphische CFG-Darstellung mit `cfg2dot` . . . 58

6 Entwurf **60**
6.1 Programmiersprache 60
6.2 Attributierte Kanten 60
6.3 Systemarchitektur 60
6.3.1 `Visitors.CFG_Gen`: IML-Traversierung 60
6.3.2 `Utils`: Hilfsfunktionen 62
6.3.3 `Types`: Allgemeine Typfunktionen 62
6.3.4 `Typefactory`: Typknoten finden und erstellen 65
6.3.5 `Try_Stacks`: Interne Darstellung der `try`-Anweisungen 65
6.3.6 `Support`: Grundblockunterstützung 65
6.3.7 `Userdefined_Routine_Attributes`: Deklarationen 66
6.3.8 `Exception_Source_Lists`: Ausnahmequellen 66
6.3.9 `Debug`: Unterstützung für den Softwareingenieur 66

7 Test **67**
7.1 Einschränkungen und Hilfskonstruktionen 67
7.2 Wegwerftests 68
7.3 Automatisiertes Testen 68
7.3.1 Darstellung der Objektidentität 68
7.3.2 Klassifizierung der Meldungen 69
7.3.3 Probleme durch nichtdeterministisches Verhalten 70
7.3.4 Wartung der Software 70
7.4 Organisation der Tests 70
7.5 Visualisierung der Ergebnisse 71
7.6 Testbereiche 71
7.6.1 Parser der externen Deklarationsdatei 71
7.6.2 Vererbung, Polymorphie und Funktionszeiger 72
7.6.3 Binden von Ausnahme und Handler 73
7.6.4 Propagierte Ausnahmen 76
7.6.5 Nicht zurückkehrende Routinen 76
7.6.6 `finally`-Aufruf 79
7.6.7 Intra- und interprozedurale Kontrollabhängigkeit 80
7.6.8 Grundblockoptimierung 82
7.6.9 `cfg2html` 82
7.6.10 `cfg2dot` 82

8 Messungen **83**
8.1 Auswahl des Quellcode 83
8.2 Umgebung 84
8.3 Mozilla 84
8.4 Xerces 84
8.4.1 Erstellungsprozess 84
8.4.2 Anwendung der Kontrollflussanalyse 85
8.5 Open Office 86

9 Verwendung der Analyseergebnisse **87**
9.1 Traversierung des erzeugten CFG 87
9.1.1 Genau ein unbedingter Nachfolger 87
9.1.2 Bedingte Nachfolger 87
9.1.3 Ausnahmenachfolger 88
9.1.4 `finally`-Aufruf 88
9.1.5 Unbedingter Nachfolger und Ausnahmenachfolger 88
9.1.6 Unbedingter Nachfolger und `finally`-Aufrufe 89

10 Zusammenfassung und Ausblick **90**
10.1 Zeitplan und aufgetretene Probleme 90
10.2 Erfahrungen 90
10.3 Bewertung des Projekts 91
10.4 Zukünftige Arbeiten 91
10.4.1 Performanceoptimierung 91
10.4.2 Verbesserung des Tests 91
10.4.3 Erweiterung der Deklarationsdatei 91

11 Glossar **92**

Literaturverzeichnis **97**

Erklärung **99**

Abbildungsverzeichnis

1.1 Generierung der IML 2

2.1 Aufbau der `try`-Anweisung 6
2.2 Qualifizierte Typen als Ausnahmen in C++ 9
2.3 Ausnahme während des Stack Unwinding 9
2.4 Verschwinden von Ausnahmen in Java 10

3.1 Intraprozeduraler Kontrollfluss mit Ausnahmebehandlung 17
3.2 Beispielquellcode Ausnahmebehandlung 19
3.3 `finally`-Block mit verschiedenen Nachfolgern 20
3.4 Klassendiagramm der IML-Grundblöcke (gekürzt) 22
3.5 Propagierung des beliebigen Ausnahmetyps 23
3.6 Der beliebige Ausnahmetyp 25

4.1 Geschachtelte `finally`-Blöcke 27
4.2 Aufruf des `finally`-Blocks von einer `throw`-Anweisung 28
4.3 Aufruf des `finally`-Blocks von einem `catch`-Block 28
4.4 Beispielquellcode für CFG-Hilfsknoten 29
4.5 CFG mit Hilfsknoten 30
4.6 Beispiel Kontrollabhängigkeit 32
4.7 Trennung von Aufruf und Rückkehr 34
4.8 RPN als Surrogat für potentiell terminierende Routine 35
4.9 Platzhalterknoten im CFG 36
4.10 Ersetzung eines EPN 37
4.11 Mehrfache Kontrollabhängigkeit 38
4.12 Kontrollabhängigkeit von Routine `fred` 38

5.1 Pseudocode für einfache IML-Graph-Traversierung 41
5.2 Ausgänge des CFG 43
5.3 C++ `throw`-Anweisungen in IML 45
5.4 IML-Hierarchie der Aufrufe 50
5.5 Sonderfall bei der Methodeneinschränkung 51
5.6 Funktionszeiger und Methoden: Quellcode 54
5.7 Funktionszeiger und Methoden: IML-Darstellung 55
5.8 Beispiel einer Deklarationsdatei 57
5.9 Ausgabe von `cfg2html` (Auszug) 59

6.1 Module von `iml2cfg` 61
6.2 IML-Darstellung verschiedener Zeigertypen. 64

7.1 Beispiel einer Spezifikation der Testmeldungen 70
7.2 Verzeichnisstruktur der Regressionstest 71
7.3 Klassenhierarchie der Ausnahmen 73

7.4 Quellcode mit explizit geworfenen Ausnahmen 74
7.5 CFG mit propagierten, ungefangenen Ausnahmen 75
7.6 CFG mit definitiv und potentiell nicht zurückkehrenden Routinen . . 77
7.7 CFG mit benutzerdefinierter DNRP 78
7.8 CFG mit `finally`-Aufruf am Ende eines `try`-Blocks 78
7.9 CFG mit `finally`-Aufrufen wegen Sprunganweisungen 79
7.10 CFG mit verschwundenen Ausnahmen 81
7.11 Beispiel einer `.dot`-Datei 82

9.1 Unbedingter Nachfolger 87
9.2 Bedingte Nachfolger 87
9.3 Nachfolger für versch. Ausnahmetypen 88
9.4 `finally`-Aufruf wegen Ausnahmen 88
9.5 uncond- und exceptional-Nachfolger 89
9.6 uncond- und exceptional-Nachfolger mit `finally`-Aufrufen 89
9.7 uncond-Nachfolger und finally_call-Kanten 89

Kapitel 1

Einleitung

Diese Ausarbeitung beschreibt die Diplomarbeit Nr. 2212 mit dem Titel "Kontrollflussanalyse objektorientierter Programme". In diesem Kapitel stelle ich kurz das bauhaus-Projekt vor, zu dem diese Diplomarbeit gehört, damit der Leser das Thema der Diplomarbeit einordnen kann. Darauf folgt eine Beschreibung der Aufgabenstellung und eine Vorstellung der Arbeiten, auf denen diese Diplomarbeit aufbaut. Zum Schluß wird ein Überblick über den Aufbau der Arbeit gegeben.

1.1 Das Projekt bauhaus

Das bauhaus-Projekt ist im Bereich Software-Reengineering angesiedelt. Es befasst sich mit der Unterstützung der Softwarewartung, insbesondere im Bereich des Programmverstehens. In der Praxis fehlen dem Wartungsingenieur oftmals Informationen zum Stand der Software. Dokumentationen sind veraltet oder fehlen ganz. Somit muss sich der Wartungsingenieur diese Informationen selbst beschaffen, was in vielen Fällen sehr zeitaufwändig ist. Dies schlägt sich auch darin nieder, dass der Wartungsaufwand für Software bei 70% bis 80% des Gesamtaufwandes liegt. Das bauhaus-Projekt will den Wartungsingenieur mit Methoden und Werkzeugen unterstützen, mit denen Architekturen extrahiert, dargestellt und manipuliert werden können.

Dazu wird eine eigene interne Darstellung der zu analysierenden Programme verwendet, die *intermediate language*, kurz IML. Sie repräsentiert syntaktische und semantische Informationen über das Programm in einem Graphen. Für die Erzeugung der IML aus dem Quellcode stehen verschiedene Frontends zur Verfügung. Nachgelagerte Analysen können aus dem IML-Graphen Informationen ableiten und diese wieder im Graphen speichern. Mit einem Visualisierungswerkzeug kann der Wartungsingenieur den Graphen unter verschiedenen Aspekten darstellen lassen und einfach durch den Graph navigieren. Zusätzlich existiert eine Palette an Werkzeugen, die ganz spezielle Informationen in textueller Form (wie z.B. HTML) ausgeben.

1.2 Aufgabenstellung

In dieser Diplomarbeit soll die schon bestehende Kontrollflussanalyse der IML auf Programme mit Ausnahmen erweitert werden. Diese sind typischerweise in objektorientierten Sprachen wie Java und C++ implementiert. Es bestehen die folgenden Anforderungen und Ausgrenzungen:

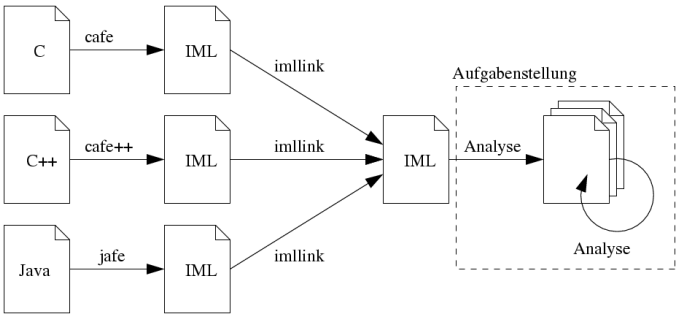

Abb. 1.1: Generierung der IML

- Es sollen nur explizite und propagierte Ausnahmen betrachtet werden. Implizit in Anweisungen und Ausdrücken ausgelöste Ausnahmen sollen nicht berücksichtigt werden.
- Alle in der IML vorhandenen Konstrukte, die in Zusammenhang mit Ausnahmen stehen, sollen berücksichtigt werden.
- Um auch Ausnahmen aus Bibliotheksroutinen in den Kontrollfluss einbringen zu können, soll der Benutzer spezifizieren können, welche Ausnahmen von welcher Routine propagiert werden können.
- Der bisher intraprozedurale Kontrollfluss soll interprozedural werden. Gleiches gilt für die Berechnung der Kontrollabhängigkeit. Dabei müssen sowohl Ausnahmen als auch Routinen berücksichtigt werden, welche das Programm terminieren.
- Es soll ein neues Werkzeug entwickelt werden, das für jede Routine die möglichen propagierten Ausnahmen bestimmt. Außerdem soll es feststellen, ob eine Routine alle möglicherweise in ihr auftretenden Ausnahmen behandelt.
- Die Bestimmung der aufgerufenen Methoden kann konservativ erfolgen, d.h. es kann angenommen werden, dass alle Implementierungen erreicht werden können.
- Der Aufwand für die Analysen soll durch Messungen ermittelt werden. Dabei sollen auch umfangreiche Programme analysiert werden.
- Optional kann die Bestimmung der aufgerufenen Methoden verfeinert werden, sofern dazu noch genügend Zeit bleibt.
- Die Software darf keine besonderen Einschränkungen oder Prämissen an die Eingabedaten stellen.
- Es wird großer Wert auf die Qualität der gesamten Software gelegt.

1.3 Grundlagen

Grundlage dieser Diplomarbeit sind hauptsächlich zwei Arbeiten von Mary Jean Harrold (et al.):

1. "Control-Flow Analysis of Programs with Exception-Handling Constructs" [Sin98]. Dieses Dokument beschreibt, welchen Einfluss Konstrukte zur Ausnahmebehandlung auf den Kontrollflussgraphen haben.

2. "Interprocedural Control Dependence" [Sin00]. Dieses Dokument geht auf die Berechnung der interprozeduralen Kontrollabhängigkeit für Sprachen ohne Ausnahmebehandlungskonstrukten ein.

Ich werde in den folgenden Kapiteln die Probleme und Lösungen beschreiben, die im Laufe dieser Diplomarbeit aufgetreten sind. Dennoch ist es für den interessierten Leser von Vorteil, die beiden genannten Arbeiten *vor* dieser Ausarbeitung zu lesen.

1.4 Anforderungen an den Leser

Der Leser sollte Kenntnisse in C++ und Java besitzen und mit den Konzepten der Objektorientierung vertraut sein. Wie die beiden Sprachen mit Ausnahmen umgehen, wird in dieser Arbeit genau beschrieben, grundlegende Kenntnisse sollten jedoch schon vorhanden sein. Auch die IML sollte dem Leser zumindest in ihren Grundzügen bekannt sein (siehe [Roh98], [Wue96], [Kna02] und [Kar03]).

1.5 Gliederung des Dokuments

Im zweiten Kapitel wird die Ausnahmebehandlung in C++ und Java beschrieben. Deren Verständnis ist Grundlage für die Modellierung des Kontrollflusses in der IML. Der bisherige Stand der IML und die notwendigen Erweiterungen werden im dritten Kapitel beschrieben. Das vierte Kapitel beschreibt Kontrollfluss und Kontrollabhängigkeit. In Kapitel fünf folgt eine genaue Beschreibung des Konzepts zur Realisierung der Aufgabenstellung. Kapitel sechs beschreibt die einzelnen Pakete von `iml2cfg` und stellt die wichtigsten Funktionen vor. Es folgt in Kapitel sieben der Test und abschließend in Kapitel acht die Beschreibung der Messungen von iml2cfg mit Open Source Quellcode.

1.6 Notation

Diese Ausarbeitung benutzt verschiedene Notationsstile, welche hier kurz erläutert werden:

- Quellcode, Kommandozeilen und Dateinamen sind in serifenloser, nichtproportionaler Schrift gesetzt, z.B. `System.exit (1);`. Schlüsselwörter werden durch **`Fettdruck`** hervorgehoben, Kommentare durch *`Kursivschrift`*.
- Bezeichner von IML-Konstrukten sind in serifenloser Schrift gesetzt, z.B. Regular_Basic_Block. Aufgrund der langen Namen werden sie teilweise auch nach den Unterstrichen getrennt. Man möge mir diesen typographischen Fauxpas verzeihen.
- Abbildungen sind in der UML-Notation gehalten, soweit nicht anders angegeben. IML-Graphen, die der Einfachheit wegen automatisch generiert wurden, bestehen aus einfachen Rechtecken und Pfeilen.
- IML-Graphen halten sich nicht an die UML-Notation. Rechtecke repräsentieren Knoten, Pfeile die Kanten des Graphen. Dies hat den Vorteil, dass die Abbildungen maschinell erzeugt werden können (s. Abschnitt 5.16, S. 58). Sofern es sich um Graphen handelt, die ein Modellierungsschema verdeutlichen sollen, sind alle Kanten als schwarze Pfeile mit Beschriftung dargestellt. Handelt es sich um IML-Graphen, die automatisch mit `cfg2dot` erzeugt wurden, sind die Kanten verschiedenfarbig dargestellt und ebenfalls beschriftet. Aus Platzgründen werden die Graphen teilweise gedreht abgebildet.

Soweit es sinnvoll ist, werden deutsche Begriffe verwendet. Nicht sinnvoll ist die Übersetzung von englischen Begriffen, die ihren festen Platz im deutschen Sprachgebrauch haben, bspw. "Computer" oder "Parser". Im Glossar (siehe Seite 92) sind außerdem Definitionen für die wichtigsten englischen und auch deutschen Begriffe zu finden.

Die Bezeichnung der Pakete wird im Text teilweise nicht vollständig wiedergegeben. Oftmals wird der Teil `Visitors.CFG_Gen` weggelassen, da fast alle Pakete zu diesem Paket gehören.

1.7 Phasen der Diplomarbeit

Im Rahmen dieser Diplomarbeit waren verschiedene Arbeiten durchzuführen. Hier wird ein kurzer Überblick über die wichtigsten Phasen gegeben:

- **Einarbeitung in die Aufgabenstellung:** In dieser Phase stand das Lesen und Verstehen der zu Grunde liegenden Arbeiten im Mittelpunkt. Dazu zählten auch die Ausarbeitungen anderer Studien- und Diplomarbeiten sowie der schon existierende Quellcode. Aufgrund der Erfahrung aus der Studienarbeit lag diese Phase vor der Unterzeichnung des Vertrags.

- **Projektplan ausarbeiten:** Nachdem die einzelnen Teilziele und Arbeitsschritte identifiziert worden waren, wurde ein Projektplan ausgearbeitet. Dazu entstand ein separates Dokument.

- **Konzept erstellen:** In dieser Phase wurde ein Konzept erstellt, wie die IML und `iml2cfg` erweitert werden mussten, um die Anforderungen erfüllen zu können.

- **Stufenweise Umsetzung:** Die Implementierung erfolgte in mehreren Stufen. Zunächst wurde die IML-Spezifikation erweitert, danach die Kontrollflussanalyse und abschließend die Berechnung der Kontrollabhängkeit.

- **Test:** Beim Testen wurden sämtliche Quellcodekonstrukte, die für die Analyse von Bedeutung waren und Sonderfälle, die während Spezifikation, Konzeption und Umsetzung identifiziert wurden, in Quellcode gegossen. Ausserdem wurde ein neues Testskript entwickelt und die notwendigen Makefiles erstellt, um den Test automatisieren zu können.

- **Messungen:** Der Analyseaufwand der Software sollte auch mit Quellcode von umfangreichen Programmen getestet werden. Dazu musste zunächst geeigneter Quellcode gefunden werden. Dabei wurde schnell deutlich, dass in C++-Quellcode äußerst selten Gebrauch von Ausnahmekonstrukten gemacht wird. Nachdem passender Quellcode gefunden war, musste die Ursache für Abstürze gefunden werden, die nach mehreren Stunden Laufzeit auftraten.

- **Ausarbeitung fertigstellen:** Als sehr nützlich erwies es sich, die Ausarbeitung parallel zur Spezifikation und Implementierung auf dem neuesten Stand zu halten. Hilfreich dazu war der Projektplan, der periodische Aufgaben enthielt, sowie der Einsatz entsprechender Software zur Erinnerung an diese Aufgaben. So musste zum Schluss nur noch der "letzte Schliff" erledigt werden.

1.8 Begriffsdefinitionen

In Programmiersprachen sind die Begriffe *Prozedur*, *Funktion* und *Methode* anzutreffen. Sie werden jedoch mit unterschiedlichen Bedeutungen verwendet. Diese Ausarbeitung versucht sich an die in bauhaus übliche Definition zu halten:

- **Routine** ist der allgemeine Ausdruck für eine benannte ausfürbahre Einheit innerhalb eines (Sub-)Systems.
- **Prozeduren** und **Funktionen** sind Routinen in prozeduralen Sprachen. Prozeduren geben im Gegensatz zu Funktionen keinen Wert zurück.
- Eine **Methode** ist eine Routine, die einer Klasse bzw. einer Instanz einer Klasse zugeordnet ist.

Kapitel 2

Ausnahmebehandlung in C++ und Java

In diesem Kapitel wird beschrieben, wie in C++ (s. Kapitel 15 in [Int98]) und Java (s. Kapitel 11 in [Gos00]) die Ausnahmebehandlung verankert ist, welche Anweisungen existieren, was sie im Detail bewirken und welche Unterschiede und Gemeinsamkeiten zwischen C++ und Java existieren. Die Betrachtung beschränkt sich auf diese beiden Sprachen, da sie die einzigen Sprachen mit Ausnahmebehandlungskonstrukten sind, für die in `bauhaus` Frontends existieren (oder sich in Planung befinden).

2.1 Sprachkonstrukte

C++ stellt die drei Schlüsselwörter **try**, **catch** und **throw** für das Exception Handling bereit. Java wurde um das Schlüsselwort **finally** erweitert, welches die Ausführung einer Anweisungssequenz nach einem try-Block garantiert (sog. *last whishes*), unabhängig davon, weshalb dieser verlassen wurde. Es ergibt sich folgende allgemeine Syntax für die `try`-Anweisung:

```
1   try{
2       ...
3   }catch (Ausnahmetyp_1  Variablenbezeichner_1){
4       ...
5   }catch (Ausnahmetyp_n  Variablenbezeichner_n){
6       ...
7   }finally{
8       ...
9   }
```

Abb. 2.1: Aufbau der `try`-Anweisung

In Zeile 1 von Abbildung 2.1 beginnt die Anweisung durch Öffnen eines neuen Blocks, dessen Inhalt in Zeile 2 durch die Punkte angedeutet ist. Dieser Block hat die besondere Eigenschaft, dass alle Ausnahmen, die innerhalb dieses Blocks geworfen werden, an den entsprechenden Handler weitergereicht werden (sofern vorhanden). Aus diesem Grund wird der `try`-Block auch als *guarded block* bezeichnet. Werden Ausnahmen im `catch`- oder `finally`-Block geworfen, so werden diese von

der **umgebenden** `try`-Anweisung bearbeitet. Die Zeilen 3 und 5 zeigen den Kopf eines Handlers. Im Kopf wird der Typ der Ausnahme und der formale Parameter für das Ausnahmeobjekt angegeben. Der formale Parameter ist nur im zugeordneten `catch`-Block sichtbar (Zeilen 4 und 6). Zeile 7 leitet den `finally`-Block ein (in C++ nicht unterstützt), der garantiert nach dem `try`-Block und einem evtl. auszuführenden `catch`-Block ausgeführt wird (Ausnahme: Das Programm wird zuvor terminiert, vgl. Abschnitt 2.6 auf S. 8).

2.2 Eintreten in einen `try`-Block

Beim Eintreten in einen `try`-Block werden diejenigen Stackobjekte angelegt und ggf. deren Konstruktor aufgerufen, die innerhalb des `try`-Blocks deklariert sind. Hier gibt es keinen Unterschied zum Eintreten in einen allgemeinen Block. Es ist jedoch nicht möglich, mit einem Sprungbefehl in einen `try`-Block hineinzuspringen [Int98].

2.3 Verlassen eines `try`-Blocks

2.3.1 Erreichen des Blockendes

Nachdem die letzte Anweisung in einem `try`-Block abgearbeitet wurde ohne eine Ausnahme zu werfen, wird der `try`-Block wie ein gewöhnlicher Block verlassen, d.h. es werden diejenigen Objekte in umgekehrter Reihenfolge vom Stapel entfernt, die zu Beginn des Blocks automatisch erzeugt wurden (*stack unwinding*). Danach wird der `finally`-Block ausgeführt (falls vorhanden). Siehe auch Abschnitt 5.2 auf Seite 40.

2.3.2 Auftreten einer Ausnahme

Wird innerhalb eines `try`-Blocks eine Ausnahme geworfen, so wird der `try`-Block an der Stelle verlassen, an der die Ausnahme geworfen wurde. Nach der Ausnahmebehandlung wird mit der Anweisung fortgefahren, die auf die `try`-Anweisung folgt (Recovery-Semantik). Eine benutzerdefinierte Ausnahme kann entweder explizit durch die `throw`-Anweisung geworfen werden oder von einer aufgerufenen Funktion propagiert worden sein. Auch in diesem Fall werden die bisher automatisch erzeugten Objekte in umgekehrter Reihenfolge vom Stapel entfernt. Anschließend wird der passende Handler für die Ausnahme bestimmt und ausgeführt, danach der `finally`-Block (falls vorhanden). Dieser Ablauf gilt für alle Arten von Ausnahmen.

2.3.3 Explizite Anweisung zum Verlassen des Blocks

Als letzte Möglichkeit kann ein `try`-Block auch durch eine der folgenden Anweisungen verlassen werden:

- `return` $value_{opt}$
- `break`
- `continue`
- `goto` *label* (nur C++)

Der Vollständigkeit wegen seien auch die nur in Java existierenden zusätzlichen Varianten genannt:

- `break` *label*
- `continue` *label*

Wie in anderen Fällen werden die bis zu diesem Zeitpunkt automatisch erzeugten Objekte in umgekehrter Reihenfolge vom Stapel entfernt. Falls dynamisch umschließende `finally`-Blöcke existieren, wird die Kontrolle nicht sofort an das Sprungziel übergeben, sondern es werden zuvor die umschließenden `finally`-Blöcke von innen nach aussen ausgeführt. Erst dann wird am Sprungziel fortgefahren!

2.4 Ausnahmetypen

In Java müssen sämtliche Ausnahmen von der Klasse `java.lang.Throwable` abgeleitet sein. Außerdem wird zwischen benutzerdefinierten Ausnahmen (in Java auch als *checked exceptions* bezeichnet) und Laufzeitausnahmen (*unchecked exceptions*) unterschieden (siehe Abschnitt 3.5, S. 21 und Glossar). Im Gegensatz dazu ist in C++ keine Einschränkung bzgl. des Typs einer Ausnahme vorhanden. So können auch Zeichenketten (`const char*`) oder Ganzzahlen (wie z.B. vom Typ `int`) als Ausnahmen geworfen werden. Eine weitere Besonderheit in C++ ist der Ausdruck `throw ()` als Erweiterung einer Funktionsdeklaration ([Int98], Abschnitt 15.4, [except.spec]). Diese Deklaration versichert, dass die Routine keine Ausnahme propagiert.

2.5 Bestimmung des Handlers für eine Ausnahme

Wird innerhalb eines `try`-Blocks eine Ausnahme vom Typ *ET* geworfen, so werden die `catch`-Blöcke der Anweisung in ihrer textuellen Reihenfolge traversiert. Wird kein passender Handler gefunden, wird die Suche mit den Handlern der textuell umgebenden `try`-Anweisung fortgesetzt. Wird auch hier kein passender Handler gefunden oder wird die Ausnahme außerhalb einer `try`-Anweisung geworfen, so wird die Suche in der dynamisch umgebenden `try`-Anweisung fortgesetzt (*Propagieren* der Ausnahme). Falls *ET* und der Ausnahmetyp eines `catch`-Blocks gleich sind oder *ET* ein vom Ausnahmetyp eines `catch`-Blocks abgeleiteter Typ ist, so passt der Handler zum Ausnahmetyp und erhält die Kontrolle. Falls kein passender Handler gefunden wird, so wird das Programm terminiert. Je nach Sprache werden zuvor noch bestimmte Routinen aufgerufen (wie z.B. `uncaught_exception` in C++). Beim Typvergleich werden weder Qualifizierer des Typs noch Referenzen auf den Typ mit einbezogen, was besonders bei C++ zu beachten ist. Diese Aussage gilt sowohl für den Fall, dass ein Handler mit konstantem Parameter einen nicht konstanten Typ fängt, also auch für den umgekehrten Fall!
Abbildung 2.2 zeigt verschiedene Handler mit unterschiedlichen Ausnahmetypen. Die Handler a) bis d) (einschl.) fangen sowohl Ausnahmen vom Typ `const MyType*` als auch vom Typ `MyType*`. Die Handler b) bis d) sind also unerreichbar.

2.6 Ausführung des `finally`-Blocks

(Nur Java) Sofern ein `finally`-Block vorhanden ist, so wird dieser immer dann ausgeführt, wenn der geschützte Bereich verlassen wird, unabhängig davon wie dies geschieht (vgl. Abschnitte 2.3.1 bis 2.7). Ein besonderer Fall existiert jedoch, in dem diese Aussage nicht gilt. Wenn innerhalb des `try`-Blocks ein Aufruf zu

```
/** C++ exception catching sample **/
try{
...
}catch (const MyType* constPtr){ ...        /** a) **/
}catch (const MyType*& constPtr){ ...       /** b) **/
}catch (MyType* ptr){ ...                   /** c) **/
}catch (MyType*& ptr){ ...                  /** d) **/
}catch (MyType** ptrPtr){ ...               /** e) **/
}
```

Abb. 2.2: Qualifizierte Typen als Ausnahmen in C++

`System.exit ()` stattfindet, so wird der `finally`-Block nicht ausgeführt! Sun Microsystems schreibt dazu in [Pur00]:

> *The System.exit method forces termination of all threads in the Java virtual machine. This is drastic. [...] System.exit should be reserved for a catastrophic error exit, or for cases when a program is intended for use as a utility in a command script that may depend on the program's exit code.*

Ausnahmen, die in einem `catch`-Block neu geworfen werden, werden im umgebenden Block behandelt. Dies gilt auch für `rethrows` (vgl. Abschnitt 5.10).

2.7 Verschwinden von Ausnahmen

Ausnahmen können sowohl in C++ als auch in Java verschwinden. In C++ verschwindet eine geworfene Ausnahme dann, wenn beim Kopieren des Ausnahmeobjekts eine neue Ausnahme geworfen wird (vgl. Abschnitt 2.9.2).

Auch wenn während des Stack Unwinding in einem Destruktor eine Ausnahme auftritt, beginnt erneut die Suche nach dem passenden Handler innerhalb der textuell umgebenden `try`-Anweisung. Auch in diesem Fall geht die ursprüngliche Ausnahme verloren.

```
1   try{
2     ClassA objA;
3     throw new ExceptionX;
4     ...
5   }catch (ExceptionX x){
6     ...
7   }catch (ExceptionClassA a){
8     ...
9   }
```

Abb. 2.3: Ausnahme während des Stack Unwinding

Sei im Beispiel (Abbildung 2.3) `ExceptionClassA` der Ausnahmetyp, den die Klasse `ClassA` bei der Ausführung des Destruktors wirft. Zeile 3 des Beispiels führt zum Verlassen des `try`-Blocks, was die Zerstörung von `objA` zur Folge hat. Der Destruktor-Aufruf wirft nun, während die Ausnahmebehandlung für `ExceptionX`

vorbereitet wird, die oben erwähnte Ausnahme vom Typ `ExceptionClassA`. Anstatt wie erwartet den Handler in Zeile 5 und 6 auszuführen, wird nun der Handler in Zeile 7 und 8 ausgeführt. Die ursprüngliche Ausnahme verschwindet.

In Java sorgen alle unbedingten Sprunganweisungen innerhalb des `finally`-Blocks, die die Kontrolle an Anweisungen außerhalb des `finally`-Blocks transferieren, für das Verschwinden der Ausnahme, die das Betreten des `finally`-Blocks ausgelöst hat. Abbildung 2.4 zeigt ein triviales Codebeispiel. Ganz offensichtlich wird eine Ausnahme geworfen (selbst wenn `getExceptionObject` einen NULL-Wert liefert), die das Betreten des `finally`-Blocks auslöst. Das `return` veranlasst die JVM jedoch, die Methode *normal* zu verlassen!

```
/** JAVA: exception disappears **/
void magic ()
{
  try{
    throw getExceptionObject ();
  }finally{
    return;
  }
}
```

Abb. 2.4: Verschwinden von Ausnahmen in Java

2.8 Propagierte Ausnahmetypen in der Routinendeklaration

In Java müssen alle potentiell von einer Routine propagierten Ausnahmetypen (oder deren Basisklassen) in der Routinendeklaration angegeben werden. Einzige Ausnahme bilden die *unchecked exceptions*, die Laufzeitausnahmen. Der Java-Compiler prüft bei der Übersetzung des Quellcodes, ob die propagierten Ausnahmetypen in der Funktionsdeklaration angegeben sind (daher die englischen Bezeichnungen *checked* und *unchecked exceptions*). Ist dies nicht der Fall, so enthält der Quellcode einen semantischen Fehler.

C++ erlaubt es, gar keine Ausnahmetypen in der Funktionsdeklaration anzugeben. Dies kann entweder bedeuten, dass tatsächlich keine Ausnahme propagiert wird oder dass beliebige Ausnahmen propagiert werden können. Auf vielen Internetseiten[1] zu C++ ist als erlaubte Deklaration auch `throw (...)` zu finden. Diese sagt aus, dass Ausnahmen geworfen werden können, schränkt deren Typ aber nicht ein. Somit ist die Deklaration äquivalent zur Deklaration ohne `throw`-Anhang. Die statische Analyse kann also nur dann Aussagen über die propagierten Ausnahmetypen treffen, wenn die Implementierung vorliegt, was besonders im Hinblick auf Bibliotheken zu Problemen führen kann. Allerdings handelt es sich hier um eine (streitbare) Spracherweiterung, die *nicht* Bestandteil des Standards ist (vgl. dazu [Int98], Abschnitt 15.4). `Cafe++` ist diesbezüglich standardkonform und wertet solche Deklarationen als Syntaxfehler.

[1] Als Beispiel sei das Microsoft Developer Network, MSDN, genannt.

2.9 Sprachspezifische Besonderheiten

2.9.1 Laufzeitausnahmen

Neben den explizit in der Funktionsdeklaration angegebenen Ausnahmen können in Java auch Laufzeitausnahmen (von der Klasse `java.lang.RuntimeException` oder abgeleiteten) geworfen werden. Diese werden von der JVM erzeugt, d.h. ihr Auftreten ist im Quellcode nicht explizit ersichtlich. Beispielsweise ist somit jede Dereferenzierung eine potientelle Quelle einer `NullPointerException`. Um die potentiellen Laufzeitausnahmen identifizieren zu können, muss jede einzelne Anweisung untersucht werden.

C++ kennt im Gegensatz zu Java keine Laufzeitausnahmen. Da in C++ keine virtuelle Maschine existiert und während der Laufzeit keine Typ- oder sonstigen Prüfungen[2] stattfinden, kann eine Ausnahme nur durch die `throw`-Anweisung geworfen werden.

Immerhin beschreibt [Int98] einige Fälle, in denen bestimmte Ausnahmen geworfen werden sollen. Beispielsweise soll bei einer erfolglosen Speicherallokation unter gewissen Umständen eine Ausnahme der Klasse `std::bad_alloc` geworfen werden.

2.9.2 Das globale Ausnahmeobjekt in C++

ANSI-C++ besitzt einen globalen Zeiger, der auf die aktuelle Ausnahme zeigt. Eine solche Variable genügt, da zu einem Zeitpunkt maximal eine Ausnahme existieren kann – sofern man nur einen Thread betrachtet. Allerdings trifft der ANSI-C++-Standard keine Aussage über Threads, sodass auch keine Aussagen über das Exception Handling in Multi-Thread-Umgebungen notwendig sind. Die `throw`-Anweisung kopiert das Ausnahmeobjekt in einen speziell dafür reservierten Speicherbereich und veranlasst die Laufzeit, nach einem passenden Handler zu suchen. Das Kopieren ist notwendig, da das Ausnahmeobjekt auf dem Stack angelegt worden sein könnte und damit beim Stack Unwinding zerstört werden würde. Gleichzeitig kann eine Ausnahme während des Kopierens zur Verdeckung der ursprünglichen Ausnahme führen (vgl. Abschnitt 2.7). `Cafe++` folgt dieser Modellierung. Es legt eine globale Instanz der Klasse O_Current_Exception an. Dieser wird beim Werfen einer Ausnahme das Ausnahmeobjekt zugewiesen. Für diese Diplomarbeit ist O_Current_Exception ohne Bedeutung, da der Typ statisch `void*` ist. Erst eine Datenflussanalyse könnte eine Aussage darüber treffen, welchen Typ die aktuelle Ausnahme besitzt. Dies ist jedoch nicht Aufgabe dieser Diplomarbeit.

[2] Der Standard [Int98] lässt bspw. offen, wie auf die Dereferenzierung von null-Pointern oder die Division durch null reagiert werden soll.

Kapitel 3

Die IML

IML ist die Abkürzung für *Intermediate Language*. Sie dient der Darstellung von Programmen, unabhängig von deren ursprünglicher Programmiersprache. Auf der IML-Darstellung eines Programms beruhen die Analysen, die bauhaus zur Verfügung stellt.

Die Spezifikation der IML ist in der Datei `iml1.spec` abgelegt, die einer eigenen, einfachen Grammatik folgt. Da die IML als Graph modelliert ist, werden hier die Knotentypen definiert. Ein Generator erzeugt aus der Spezifikation neben Ada-Quellcode auch C-Quellcode, der von verschiedenen Frontends benutzt wird. Der Aufruf des Generators geschieht aus dem Makefile heraus, sodass zur Generierung der IML ein einfacher Aufruf von `make` genügt. Die ebenfalls vorhandene `iml2.spec` hat informativen Charakter und stellt eine Art Fernziel für die IML dar. Aus ihr wird jedoch kein Quellcode generiert. Die Fähigkeiten des Generators sind beschränkt, sodass teilweise Wrapper-Klassen benutzt werden, um neben den generierten (einfachen) Funktionen weitere (komplexe) Funktionen bereitstellen zu können.

Ein sehr hilfreiches Dokument wird ebenfalls beim Generieren des Quellcodes erzeugt: Die Klassenhierarchie der IML-Knoten. Dieses Diagramm ist für jeden bauhaus-Entwickler von Interesse. Es ist unter dem Dateinamen `hierarchy.ps` im `generated`-Verzeichnis abgelegt.

3.1 Bisherige Version der IML

Dieser Abschnitt erläutert den aktuellen Stand der IML im Hinblick auf den CFG. Er ist ein Teilgraph von vielen, die im IML-Graphen vereint sind. Der CFG besteht aus zwei Teilen:

- Den Value-Knoten, die über die Attribute `CF_Next` und `CF_Previous` eine doppelt verkettete Liste aufbauen, welche eine Anweisungssequenz darstellt. Da ein Programm auch Verzweigungen enthalten kann, reicht diese Darstellung allein nicht aus. Deshalb muss jeder ausführbare Value-Knoten einen Verweis auf den Grundblock (siehe Abschnitt 3.1, S. 13) besitzen, dem er angehört.

- Den Grundblöcken, welche eine abstraktere Darstellung des Kontrollflusses erlauben. Sie fassen Anweisungssequenzen zusammen und modellieren zusätzlich Verzweigungen.

Die bisherige Form des CFG ist intraprozedural, d.h. der Kontrollfluss wird nur innerhalb einer Routine berechnet, Aufrufe anderer Routinen werden nicht oder nur teilweise abgebildet. Neben den Kontrollflussinformationen sind im Knotentyp Routine auch Informationen abgelegt, mit denen der Aufrufgraph erstellt werden kann.

Die beiden Attribute Callees und Call_Sites enthalten Verweise auf die aufgerufenen und aufrufenden Routinen. Zur Darstellung des Kontrollflusses werden u.a. die Grundblöcke generiert, die Bestandteil der IML-Spezifikation sind. Es existieren die abstrakte Basisklasse Basic_Block und drei Subklassen (siehe Abbildung 3.4). Ein Entry_Basic_Block stellt den Eintrittsknoten in eine Routine dar, ein Exit_Basic_Block den Austrittsknoten. Dabei besitzt jede Routine jeweils genau einen dieser Grundblöcke. Der IML-Knotentyp Routine besitzt auf beide einen entsprechenden Verweis. Auf jeden Entry_Basic_Block folgt genau ein Regular_Basic_Block. Ein Regular_Basic_Block fasst mehrere Value-Knoten des HPG-Graphen zusammen. Neben der unbedingten Folgerelation (Attribut Successor_Uncond, vgl. Abbildung 3.4 auf S. 22) ist auch die bedingte Folgerelation modelliert. Die beiden Attribute Successor_True bzw. Successor_False verweisen auf denjenigen Grundblock, der nach Auswertung des Prädikats (Attribut Predicate) zu wahr bzw. falsch ausgeführt wird.

3.2 Generierung des CFG

3.2.1 Das Werkzeug `iml2cfg`

Die Generierung des CFG ist Aufgabe des Tools `iml2cfg`[1]. Es liest eine bestehende IML-Datei ein, fügt die CFG-Informationen hinzu und schreibt eine neue IML-Datei. Voraussetzung hierfür ist, dass der IML-Graph ein komplettes System darstellt (also die Eingabedatei das Ergebnis von `imllink` ist). Das Tool implementiert das Visitor-Design-Pattern [Gam95]. Im Paket `Visitors.CFG_Gen` sind die für die jeweiligen Knotentypen zuständigen Visitor-Funktionen zusammengefasst. Im einfachsten Fall wird ein HPG-Knoten als letzter Eintrag in die Anweisungsliste eines bestehenden Grundblocks aufgenommen.

3.2.2 Backpatching unbedingter Sprünge

Unbedingte Sprünge stellen eine Besonderheit dar, die mit Hilfe des *Backpatchings* behandelt werden: Während der Traversierung eines Program_Unit-Knotens wird jeweils eine Liste aller Sprünge und Sprungmarken aufgebaut. Nach der Traversierung wird zwischen jedem Grundblock, der mit einem unbedingten Sprung endet und dem Grundblock, der mit der entsprechenden Sprungmarke beginnt, eine Successor_Uncond-Kante generiert. Diese nachgelagerte Bearbeitung ist notwendig, da Sprungmarken nicht vor ihrer ersten Verwendung in einem `goto` deklariert werden müssen. Nach dem Backpatching stehen die Ziele aller `gotos` fest, da Sprungmarken nur innerhalb der Routine gültig sind, in der sie deklariert sind ([Int98], Kapitel 6.1). Für Java-Programme erübrigt sich dieses Vorgehen, da in Java keine `goto`-Anweisung existiert ([Gos00], Kapitel 14.7). Für die interprozeduralen Analysen kann das Backpatching also unverändert übernommen werden.

3.2.3 Optimierung des Zwischenergebnisses

In einem abschließenden Schritt wurden bisher die nicht erreichbare Grundblöcke entfernt, und die restlichen Grundblöcke werden soweit möglich zusammengefasst (siehe unten). Diese Optimierung wurde bisher nach dem Besuch eines Program_Unit-Knotens ausgeführt, was für den interprozeduralen CFG nicht möglich ist (siehe Abschnitt 5.4, Seite 42). Es gilt:

Definition: *Grundblockverschmelzung*

[1] Verzeichnis $BAUHAUS/Ada-Code/Tools/iml2cfg/

Seien γ_1 und γ_2 jeweils ein Value-Knoten oder ein Regular_Basic_Block. γ_1 und γ_2 können genau zu einem einzigen Regular_Basic_Block verschmolzen werden,

- wenn γ_1 und γ_2 textuell aufeinander folgen, während der Ausführung auf γ_1 immer γ_2 folgt und γ_2 keinen anderen Vorgänger als γ_1 besitzt
- oder wenn γ_2 ein Sprungziel ist, das von genau der letzten Anweisung in γ_1 und von sonst keiner Anweisung angesprungen wird.

3.3 Der Kontrollflussgraph

In [Sin98] von M. J. Harrold findet sich eine genaue Definition eines intraprozeduralen CFG. Die beschriebene Darstellung ist mit der IML und den Grundblöcken leider nicht genau umzusetzen. Außerdem werden einige Feinheiten nicht behandelt, insbesondere was `finally`-Blöcke anbelangt. Deshalb gebe ich in diesem Abschnitt die formale Definition des CFG, wie er in dieser Diplomarbeit verwendet wird.

Definition: *Intraprozeduraler CFG für IML*
Sei M eine Funktion, $F = \{F_1, F_2, F_3, \ldots\}$ seien die `finally`-Blöcke in M, $TS = \{TS_1, TS_2, TS_3, \ldots\}$ sei die Menge der `try`-Anweisungen, zu denen die `finally`-Blöcke gehören. Ein CFG $G = (N, E, EN, EX, EXC, FN, FX)$ für M ist ein gerichteter Graph, wobei N die Knoten, $E = E_{uncond} \cup E_{true} \cup E_{false} \cup E_{exceptional} \cup E_{finally_call}$ die Kanten und EN, EX, FN sowie FX Mengen von Knoten mit bestimmten Eigenschaften sind (s.u.).

- N enthält für jeden Grundblock in M einen Knoten. Dabei wird jeder `catch`-Handler jeweils mit einem oder mehreren Grundblöcken dargestellt.
- N besitzt fünf spezielle Teilmengen: EN und EX, welche die Eintritts- und Austrittsknoten von G wiedergeben. EN besteht aus genau einem Knoten vom Typ Entry_Basic_Block. Der Knoten in EN besitzt keine Vorgänger in G. Der Knoten in EX hat keine Nachfolger in G; EX besteht aus genau einem Exit_Basic_Block . EXC ist die Menge aller Austrittsknoten von M, die eine Ausnahme propagieren. Für jeden Ausnahmetyp existiert ein Knoten in EXC. FN und FX sind die Eintritts- und Austrittsknoten der `finally`-Blöcke, wobei jeder `finally`-Block jeweils genau einen davon besitzt.
- In N werden Funktionsaufrufe in M nicht mit nur einem Knoten modelliert: Es existieren ein Grundblock mit dem Routine_Call-Knoten c als letzte Anweisung, der die Kontrolle an die aufgerufene Funktion übergibt und ein Grundblock mit dem Post_Call_Link-Knoten r als erste Anweisung, der nach dem Aufruf der Funktion als erster Knoten die Kontrolle wieder erhält, falls keine Ausnahmen propagiert wurden und der Aufruf zurückkehrt. Sofern der Aufruf zurückkehrt, existiert eine uncond-Kante von c nach r. Andernfalls existiert eine Kante von c zm Exit_Basic_Block aus EX.
- Für jeden `catch`-Handler in M existiert ein Teilgraph aus Regular_Basic_Blocks, der mit genau einem Grundblock als Wurzel beginnt.
- Jeder `finally`-Block in M wird als Teilgraph dargestellt, der mit genau einem Finally_Entry_Basic_Block beginnt. Sein Rumpf besteht aus mindestens einem Regular_Basic_Block und der normale Ausgangsknoten ist als Finally_Exit_Basic_Block dargestellt. "Normaler Ausgangsknoten" bedeutet in diesem Zusammenhang, dass auf dem Pfad vom Eingang zu diesem Ausgang weder Ausnahmen geworfen wurden noch Sprünge nach außerhalb des Blocks stattgefunden haben.

Kanten in G. Abbildung 3.1 auf Seite 17 stellt die folgenden Fälle grafisch dar.

1. Für zwei Grundblöcke m und n ($m, n \in N$) existiert in E_{uncond} dann eine Kante (m, n), wenn auf die letzte Anweisung in m immer n ausgeführt wird, falls in m keine Ausnahme geworfen wurde.

2. Für zwei Grundblöcke m und n ($m, n \in N$) existiert in E_{true} genau dann eine Kante (m, n), wenn die letzte Anweisung in m eine Bedingung c ist und n ausgeführt wird, falls c wahr ist und in m keine Ausnahme geworfen wurde.

3. Für zwei Grundblöcke m und n ($m, n \in N$) existiert in E_{false} genau dann eine Kante (m, n), wenn die letzte Anweisung in m eine Bedingung c ist und n ausgeführt wird, falls c nicht wahr ist und in m keine Ausnahme geworfen wurde.

4. Es existiert eine Kante $(t, f) \in E_{finally_call}$ zwischen dem letzten Grundblock t eines `try`-Blocks und dem ersten Grundblock des zugehörigen `finally`-Blocks f. Außerdem existiert diese Kante (t, f), wenn t ein Grundblock im `try`-Block ist, dessen letzte Anweisung den Block verlässt, ohne eine Ausnahme zu werfen oder wenn keine `catch`-Handler existieren.

5. Zwischen dem letzten Grundblock t innerhalb des `try`-Blocks und dem ersten Grundblock m nach einem `try`-Block existiert genau dann eine Kante $(t, m) \in E_{uncond}$, wenn der `try`-Block keinen `finally`-Block besitzt und t keine Ausnahme wirft.

6. Zwischen dem letzten Grundblock c eines `catch`-Blocks und den Eintrittsgrundblöcken f_i des `finally`-Blocks der `try`-Anweisung und der textuell umgebenden `finally`-Blöcke ($f_i \in FN$) existiert genau dann eine Kante $(c, f_i) \in E_{finally_call}$, wenn der `try`-Block einen `catch`-Block besitzt.

7. Zwischen dem ersten Grundblock m nach einem `try`-Block und dem letzten Grundblock c eines zugehörigen `catch`-Blocks existiert genau dann eine Kante $(c, m) \in E_{uncond}$, wenn in c keine Ausnahmen geworfen werden (unabhängig von `finally`-Blöcken).

8. Für zwei Grundblöcke m und c ($m, c \in N$) existiert in $E_{exceptional}$ dann eine Kante (m, c), wenn in m eine Ausnahme geworfen werden kann, m ein Grundblock im Teilgraphen eines `try`-Blocks ist und c der erste Grundblock des `catch`-Blocks ist, der diesen Ausnahmetyp behandelt.

9. Für zwei Grundblöcke m und e ($m \in N, e \in EXC$) existiert in $E_{exceptional}$ dann eine Kante (m, e), wenn in m eine Ausnahme geworfen werden kann, für die kein Handler in M existiert.

10. Für zwei Grundblöcke f und e ($f \in N, e \in EXC$) existiert in $E_{exceptional}$ dann eine Kante (f, e), wenn in f eine Ausnahme geworfen werden kann, für die kein Handler in M existiert und f Teil eines `finally`-Blocks ist. Dies ist ein Sonderfall zu 9.

11. Für zwei Grundblöcke c und e ($c \in N, e \in EXC$) existiert in $E_{exceptional}$ dann eine Kante (c, e), wenn in c eine Ausnahme geworfen werden kann, für die kein Handler in M existiert und c Teil eines `catch`-Blocks ist. Dies ist ein Sonderfall zu 9.

12. Zwischen einem Grundblock f, Teil eines `finally`-Blocks, und einem Grundblock x ($f \in N, x \in EX$) existiert in $E_{unconditional}$ dann eine Kante (f, x), wenn in f die Routine ohne Ausnahme verlassen wird.

13. Zwischen einem Grundblock m, nicht Teil eines `finally`-Blocks, und einem Grundblock x ($m \in N, x \in EX$) existiert in $E_{unconditional}$ dann eine Kante (m, x), wenn in m die Routine ohne Ausnahme verlassen wird.

14. Es existieren keine Kanten (f, m) ($f \in FX$, $m \in N$).

Der interprozedurale CFG basiert auf dem intraprozeduralen. Zusätzlich werden die Aufrufkanten der Routine_Call-Knoten mit aufgenommen, die wiederum Teil der Anweisungen innerhalb der Grundblöcke sind.

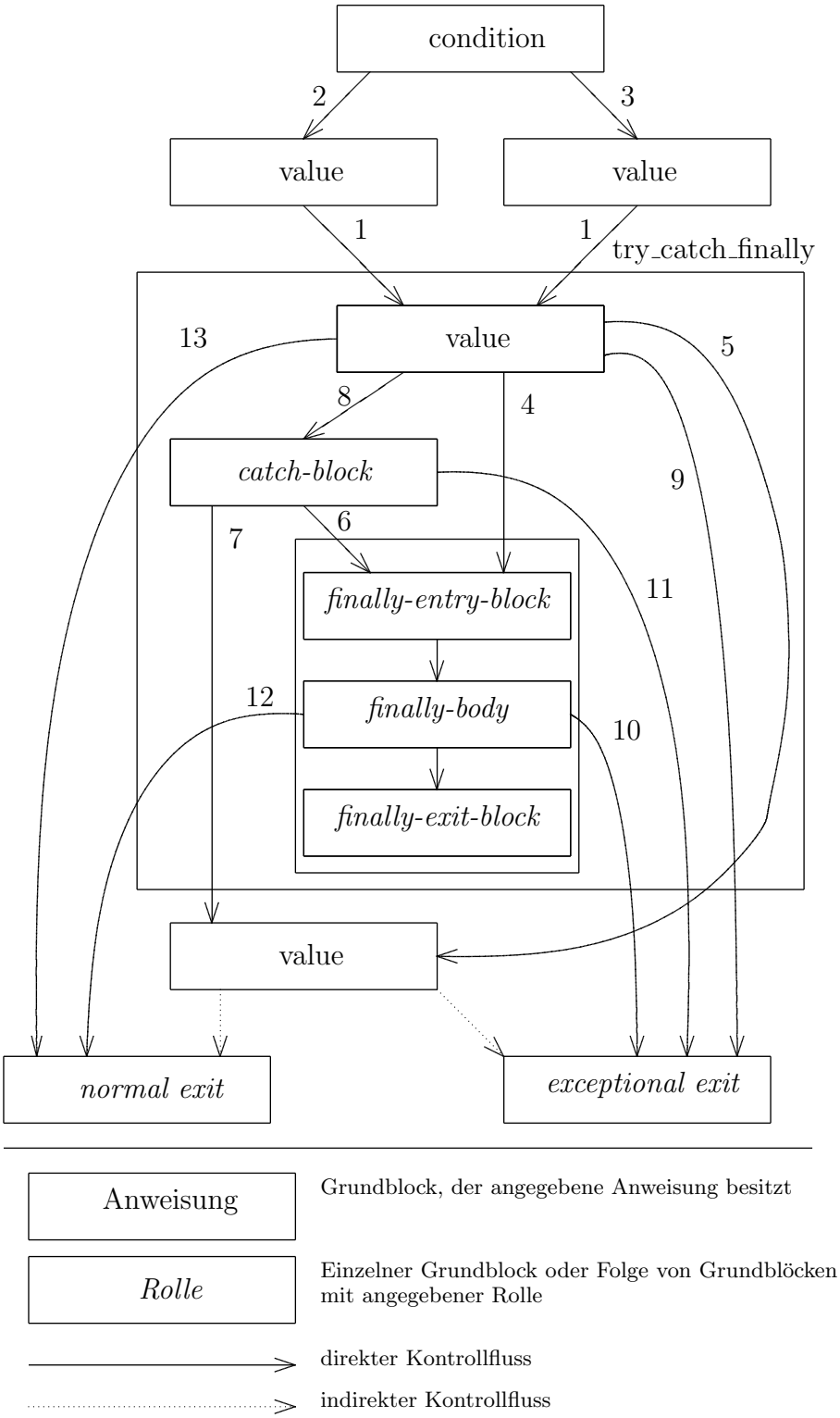

Abb. 3.1: Intraprozeduraler Kontrollfluss mit Ausnahmebehandlung

3.3.1 Beispiel intraprozeduraler CFG

Abbildung 3.2 auf Seite 19 zeigt ein Quellcodestück in Java, das verschiedene Fälle der Ausnahmebehandlung beinhaltet. Die folgenden Angaben in Klammern beziehen sich auf die Kommentare in der Abbildung.

(1) Ein unbedingter Sprung aus einem `try`-Block. Die Kontrolle geht jedoch an den `finally`-Block (f2), erst danach wird die Sprunganweisung ausgeführt.

(2) Aufruf einer Funktion, die Ausnahmen propagieren kann. Existiert für den Ausnahmetyp ein `catch`-Handler ((c2) oder (c3)), so geht die Kontrolle an diesen über, danach wird der `finally`-Block (f2) ausgeführt. Andernfalls wird die Ausnahme propagiert, nachdem der `finally`-Block (f2) ausgeführt wurde.

(3) Aufruf einer Funktion, die Ausnahmen propagieren kann. Falls (c1) diesen Ausnahmetyp behandelt, geht die Kontrolle an (c1), dann an (f1) und schließlich an (b) über. Falls (c1) kein passender Handler ist, wird (f1) ausgeführt und im (statisch) umgebenden `try`-Block nach passenden Handlern gesucht ((c2) und (c3)). Falls kein passender existiert, wird (f2) ausgeführt und die Ausnahme propagiert. Andernfalls wird der passende Handler, danach (f2) und schließlich (r) ausgeführt.

(4) Explizites Werfen einer Ausnahme. Der Ausnahmetyp ist entweder `Exception_A1` oder `Exception_B` (es existieren keine anderen Zuweisungen an `exc`). Falls der umgebende `try`-Block einen passenden Handler besitzt, geht die Kontrolle an diesen über (im Beispiel behandelt (c3) den Typ `Exception_A1`), anschließend werden der `finally`-Block (f2) und (r) ausgeführt. Falls kein passender Handler in den (statisch) umgebenden `try`-Blöcken existiert, wird der `finally`-Block (f2) ausgeführt und danach die Ausnahme propagiert.

3.4 Erweiterung der IML

In diesem Abschnitt wird beschrieben, welche Veränderungen und Erweiterungen an der IML vorgenommen wurden. Es werden Lösungsalternativen aufgezeigt und begründet, weshalb die realisierte Lösung gewählt wurde.

In der IML wird die Suche nach dem zuständigen `catch`-Block zu einer geworfenen Ausnahme mit einer `if`-Kaskade modelliert [Kna02]. Diese besteht aus einer Folge von Bedingungen, welche die von den `catch`-Blöcken behandelten Ausnahmetypen mit dem Typ der aktuellen Ausnahme vergleichen. Für die aktuelle Ausnahme generiert das C++-Frontend einen einzigen künstlichen, globalen O_Current_Exception-Knoten mit der Bezeichnung @o_Current_Exception@. Diese Modellierung ist sinnvoll, wenn eine virtuelle IML-Maschine den Code ausführt. Für die statische Analyse ist diese Darstellung belanglos.

3.4.1 Einschränkungen durch die Erweiterungen

Auf Hierarchical_Basic_Blocks wird in diesem Abschnitt *nicht* eingegangen, da sie nicht Teil der Aufgabenstellung sind. Ich möchte aber ausdrücklich darauf hinweisen, dass hier noch Anpassungen notwendig sind. Wenn die Generierung dieser Klasse von Grundblöcken aktiviert ist (also die Auskommentierung im Quellcode zurückgenommen wird), führt dies zu Abstürzen!

```
/** class hierarchy of the exceptions:
  - Exception_A extends Exception
  - Exception_A1, Exception_A2 extends Exception_A
  - Exception_B extends Exception */
void sampleMethod (BRANCH_CODE bc)
{
  Exception exc = null;
  try{
    switch (bc){
    case BRANCH_RETURN:
      return;                                  // <-- (1)
    case BRANCH_EXC1:
      exc = new Exception_A1;
      break;
    case BRANCH_EXC2:
      exc = new Exception_B;
      break;
    case BRANCH_METHOD:
      method ();                               // <-- (2)
      break;
    case BRANCH_NESTED:
      try{
        method ();                             // <-- (3)
      }catch (Exception_B eb){                 // <-- (c1)
        //...no throw...
      }finally{                                // <-- (f1)
        //...no throw...
      }
      break;                                   // <-- (b)
    default:
      break;
    }
    if (exc != null){
      throw exc;                               // <-- (4)
    }
  }case (Exception_A2 ea2){                    // <-- (c2)
    //...no throw...
  }case (Exception_A ea){                      // <-- (c3)
    //...no throw...
  }finally{                                    // <-- (f2)
    //...no throw...
  }
  return;                                      // <-- (r)
}
```

Abb. 3.2: Beispielquellcode Ausnahmebehandlung

```
1      try{
2          if (condition)
3              goto label0;
4      }finally{
5          // do something
6      }
7      ... // statement sequence B
8    label0:
9      ... // statement sequence C
```

Abb. 3.3: `finally`-Block mit verschiedenen Nachfolgern

3.4.2 Grundblöcke

Um die Auswirkungen der Ausnahmebehandlung im Kontrollfluss wiedergeben zu können, wurde zum einen der Regular_Basic_Block um einige Attribute erweitert, zum anderen wurden neue Knotentypen eingeführt. Jeder Regular_Basic_Block kann neben den bedingten und unbedingten Nachfolgern auch solche Grundblöcke als Nachfolger haben, die eine geworfene Ausnahme behandeln, also den Beginn eines `catch`-Blocks modellieren. Nach der obigen Definition bewirkt damit jeder Aufruf einer Funktion, die eine Ausnahme werfen kann und jede `throw`-Anweisung das Ende eines Regular_Basic_Block. Für jeden einzelnen Ausnahmetyp wird genau eine Kante in der Liste Successors_Exceptional angelegt. Zusätzlich wird ein Eintrag in der Liste Exception_Type angelegt (vgl. Abbildung 3.4). Auf diese Weise werden die Kanten mit dem Ausnahmetyp attributiert. [Sin98] verwendet für `finally`-Blöcke eine besondere Konstruktion, leider ohne diese zu erklären: Jeder `finally`-Block ist ein separater CFG, der ähnlich wie eine Routine aufgerufen wird. Diese Konstruktion löst geschickt das Problem, dass ein `finally`-Block mehr als einen unbedingten Nachfolger haben kann. Abbildung 3.3 zeigt ein Beispiel, in dem der `finally`-Block in Zeile 5 sowohl *statement sequence B* als auch *statement sequence C* als direkte, nicht bedingte Nachfolger haben kann.

Im Folgenden erläutere ich die mit der Abteilung diskutierten Alternativen zur Modellierung der `finally`-Anweisung und ihre Bewertung.

1. Statt einem Uncond-Nachfolger mehrere Uncond-Nachfolger erlauben: Bei dieser Variante kann bei einer Traversierung nicht entschieden werden, mit welchem der Uncond-Nachfolger die Traversierung fortzusetzen ist. Außerdem würde eine solche Veränderung die Anpassung aller Werkzeuge erzwingen, die den CFG benutzen.

2. Den `finally`-CFG als Routine in den CFG einfügen und diese mit einem Direct_Call aufrufen: Dies würde eine Vermischung von HPG und CFG sowie eine Manipulation des HPG bedeuten. Außerdem ist der `finally`-Block keine Routine im Sinn eines Routine-Knotens.

3. Erweiterung des Typs Regular_Basic_Block um eine Liste mit Verweisen auf Grundblöcke, die den Beginn eines `finally`-Blocks abbilden. Diese Verweise haben nicht die `goto`-Semantik wie die anderen Nachfolger, sondern Aufruf-Semantik: Diese Lösung ist eine Erweiterung der bisherigen Grundblöcke, sodass bestehende Werkzeuge auch mit diesen Grundblöcken ohne eine Anpassung lauffähig sind. Die neuen Kanten haben eine andere Semantik als die

bisherigen, was aber aus der Namensgebung der Attribute deutlich zu erkennen ist (Successor(s)_*nnn* vs. Finally_Calls).

Wegen des IML-Generators ist es leider nicht möglich, diese Attribute komplett abzukapseln und in ihren `Set`-Methoden eine Konsistenzprüfung durchzuführen. Das Paket `Basic_Block_Wrappers` stellt jedoch Funktionen bereit, um komfortabel und sicher mit Grundblöcken arbeiten zu können (das Paket existierte schon vor dieser DA). Zusätzlich wurde bei der Implementierung sämtlicher Funktionen, die Kanten zwischen den Grundblöcken verändern, darauf Wert gelegt, dass sämtliche Vor- und Nachbedingungen geprüft werden. Es ergeben sich folgende Konsistenzbedingungen für die Kanten eines Grundblocks:

- Wenn ein Uncond-Nachfolger existiert, kann kein True- und kein False-Nachfolger existieren.
- Wenn ein False-Nachfolger existiert, muss auch ein True-Nachfolger existieren.
- Wenn ein True-Nachfolger existiert, muss auch ein False-Nachfolger existieren.
- Exceptional-Nachfolger können unabhängig von den anderen Nachfolgern und Finally-Calls existieren.
- Finally-Calls können unabhängig von Uncond- oder Exceptional-Nachfolgern existieren.

Zur Darstellung der `finally`-Blöcke im CFG werden zwei weitere Grundblock-Klassen eingeführt: Finally_Entry_Basic_Block und Finally_Exit_Basic_Block, die den Beginn und das Ende eines `finally`-Blocks markieren (vgl. Abbildung 3.4, S. 3.4). Beide Klassen besitzen rein informativen Charakter, der CFG könnte auch ohne sie aufgebaut werden. Allerdings würden dann Teilgraphen im CFG existieren, die keine Nachfolger besitzen. Falls diese nur über Kanten aus finally_calls erreichbar wären, würde es sich um `finally`-Blöcke handeln. Die Einführung beider Klassen vermeidet diese Prüfung und unterstreicht den Aufrufcharakter eines `finally`-Blocks.

3.4.3 Routine-Knoten

Für den interprozeduralen CFG ist es notwendig, die von einer Routine propagierten Ausnahmen zu kennen. Deshalb wird der Routine-Knoten um das Attribut CFG_Exceptional_Exits erweitert. Diese Liste enthält die Zeiger auf alle Exceptional_Exit_Basic_Blocks der Routine. Über diese Grundblöcke können die propagierten Ausnahmetypen festgestellt werden.

3.5 Implizite Ausnahmen

Implizite Ausnahmen werden in dieser Diplomarbeit nicht berücksichtigt (s. Abschnitt 1.2, Seite 1). Ein Grund ist die immense Anzahl an Kanten, die durch die potentiellen Ausnahmen entstehen würden. Außerdem müsste jede Anweisung des Quellcode auf die von ihr möglicherweise generierten Ausnahmen untersucht werden. Ob und welche Ausnahmen auftreten können, ist von der Programmiersprache des Quellcodes abhängig. Daneben müsste man überlegen, ob weiterhin mehrere Anweisungen in einem Grundblock zusammengefasst werden. Es scheint sehr unwahrscheinlich zu sein, dass eine Folge von Anweisungen keine einzige Ausnahme auslösen kann (bezogen auf Java). Eine mögliche Lösung wäre, die Kontrollflussinformation nicht mehr in den Grundblöcken abzulegen, sondern direkt in den ausführbaren Anweisungen (Value-Knoten). Grundblöcke des Typs Regular_Basic_Block würden damit

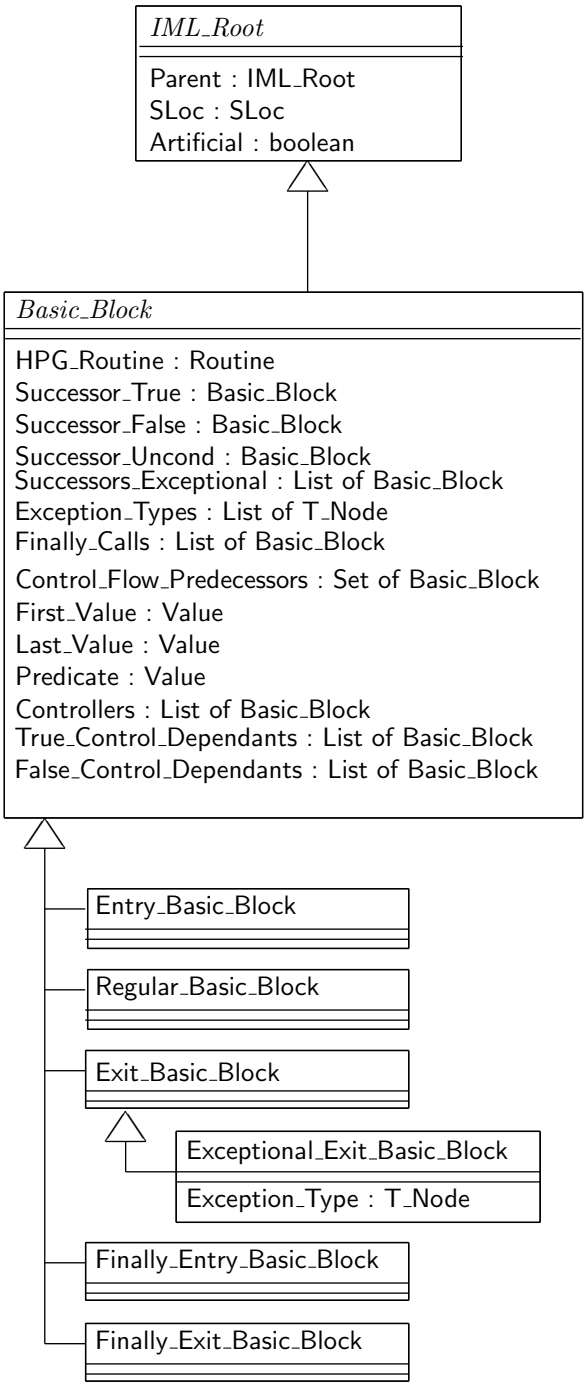

Abb. 3.4: Klassendiagramm der IML-Grundblöcke (gekürzt)

vielleicht überflüssig. Ein weiteres Problem wäre die Modellierung des Stack Unwinding. Hier wäre es denkbar, für alle Anweisungen $a_i \ \ldots \ a_k$ eine gemeinsame Folge von End_Of_Lifetime-Knoten anzulegen, wenn für alle $a_n, i \leq n \leq k$ gilt: a_n ist weder ein Begin_Of_Lifetime- noch ein End_Of_Lifetime-Knoten.

Die Arbeiten von M. J. Harrold verwenden eine Modellierung, bei der Anweisungen *nicht* zu Grundblöcken zusammengefasst werden. Der Grund dafür ist nicht bekannt.

3.6 Der beliebige Ausnahmetyp

> ***Hinweis:***
> *Dieser Abschnitt befasst sich mit einer C++-Erweiterung, die nicht Teil des ANSI-Standards ist und von* bauhaus *nicht unterstützt wird! Die folgenden Überlegungen wurden nicht realisiert!*

Manche C++-Dialekte erlauben als Typ von propagierten Ausnahmen den Ellipsisausdruck. Die IML bildet dies mit dem IML-Knotentyp T_Any_Exception ab, der einen beliebigen Ausnahmetyp darstellt.

3.6.1 Einschränkung der möglichen Typen

Angenommen, in einem System existieren zwei Ausnahmetypen, ET_1 und ET_2. Außerdem exisitieren die drei Methoden `alice`, `bob` und `claire`. Die Aufrufbeziehung, die propagierten Ausnahmen und die gefangenen Ausnahmen sind dem Beispiel in Abbildung 3.5 zu entnehmen (Pseudocode). In den Zeilen 8 und 15 werden äußerst

```
01   void alice (){
02     try{
03       bob ();
04     }(catch ET2* e2){
05       ...
06     }
07   }
08   void bob () throw (...){
09     try{
10       claire ();
11     }(catch ET1* e1){
12       ...
13     }
14   }
15   void claire () throw (...){
16     ...
17     throws new ET1;
18     ...
19     throws new ET2;
20     ...
21   }
```

Abb. 3.5: Propagierung des beliebigen Ausnahmetyps

ungenaue Angaben zu den propagierten Ausnahmen gemacht, die durch die Quellcodeanalyse eingeschränkt werden können. Außerdem wird es notwendig, den Typ T_Any_Exception zu erweitern: Er erhält als weiteres Attribut eine Negativliste von Ausnahmetypen. Dazu ein Beispiel: Ohne diese Erweiterung würden sowohl `bob` als auch `claire` Ausnahmen des beliebigen Typs propagieren. Dadurch würde die Analyse ergeben, dass auch `alice` Ausnahmen des beliebigen Typs propagieren würde. Unter der Annahme, dass `alice` der Wurzelknoten im Aufrufgraph ist, würde diese Modellierung dazu führen, dass `alice` ebenfalls den beliebigen Ausnahmetyp propagiert, also global nicht behandelte Ausnahmen existieren, die das Programm terminieren könnten. Das ist nicht korrekt.

Die Erweiterung um die Negativliste verhindert zwar nicht, dass im CFG für `alice` ein Exceptional_Exit_Basic_Block angelegt wird. Ist jedoch die Menge aller Ausnahmetypen bekannt und existieren für alle entsprechende Handler, so ergibt die Menge aller möglichen Ausnahmen (also der beliebige Ausnahmetyp) abzüglich der Menge in der Negativliste die leere Menge. Damit ist die Aussage korrekt.

Die Menge aller möglichen Ausnahmetypen wird auf die expliziten Ausnahmetypen beschränkt. Somit könnten alle Ausnahmetypen durch die statische Analyse ermittelt werden, indem jede `throw`-Anweisung und jede Funktionsdeklaration mit `throws`-Ausdruck analysiert wird und deren potentielle Ausnahmetypen zentral werden.

3.6.2 Auftreten des beliebigen Ausnahmetyps

In diesem Abschnitt stelle ich die drei möglichen Stellen vor, an denen der beliebige Ausnahmetyp auftreten kann und erläutere deren denkbare Abbildung im CFG.

3.6.2.1 Universeller `catch`-Handler

Die erste Stelle, an der man auf den beliebigen Ausnahmetyp treffen kann, ist ein `catch`-Block, der alle Ausnahmetypen behandelt. Dies hat für die Generierung des CFG keine Konsequenzen. Sollten weitere `catch`-Blöcke folgen, so haben diese keine eingehenden Kanten, da sie unerreichbar sind. Ein guter Compiler wird in einem solchen Fall schon bei der Umwandlung des Quellcodes entsprechende Warnungen ausgeben.

3.6.2.2 Explizite Propagierung

In der Routinendeklaration wurde die Ellipsis angegeben (C++-Erweiterung). Steht bei der Analyse die Implementierung der Routine zur Verfügung, werden zunächst typspezifische Ausnahmeausgänge (Exceptional_Exit_Basic_Block) angelegt, falls notwendig. Zusätzlich wird ein Ausnahmeausgang für T_Any_Exception angelegt. Deren Negativliste beinhaltet die Typen der anderen Ausnahmeausgänge. Steht keine Implementierung zur Verfügung (z.B. bei Bibliotheken), so wird nur der Ausnahmeausgang für T_Any_Exception angelegt (mit leerer Negativliste).

Mit dieser Modellierung werden zwei Ziele erreicht: Quellentreue und Genauigkeit. Alternativ hätten die Ausnahmeausgänge auch zu einem Ausgang für beliebige Ausnahmetypen zusammengefasst werden können. Damit hätte jedoch die Genauigkeit auf Kosten der Quellentreue gelitten.

3.6.2.3 Routinenaufruf

Der Aufruf einer Routine ist die einzige Möglichkeit, wie eine Ausnahme beliebigen Typs überhaupt entstehen kann. Tritt dieser Fall ein, so werden Kanten zu den `catch`-Blöcken gezogen, die den Aufruf umgeben, sofern der vom Block behandelte Ausnahmetyp nicht in der Negativliste von T_Any_Exception enthalten ist. Danach

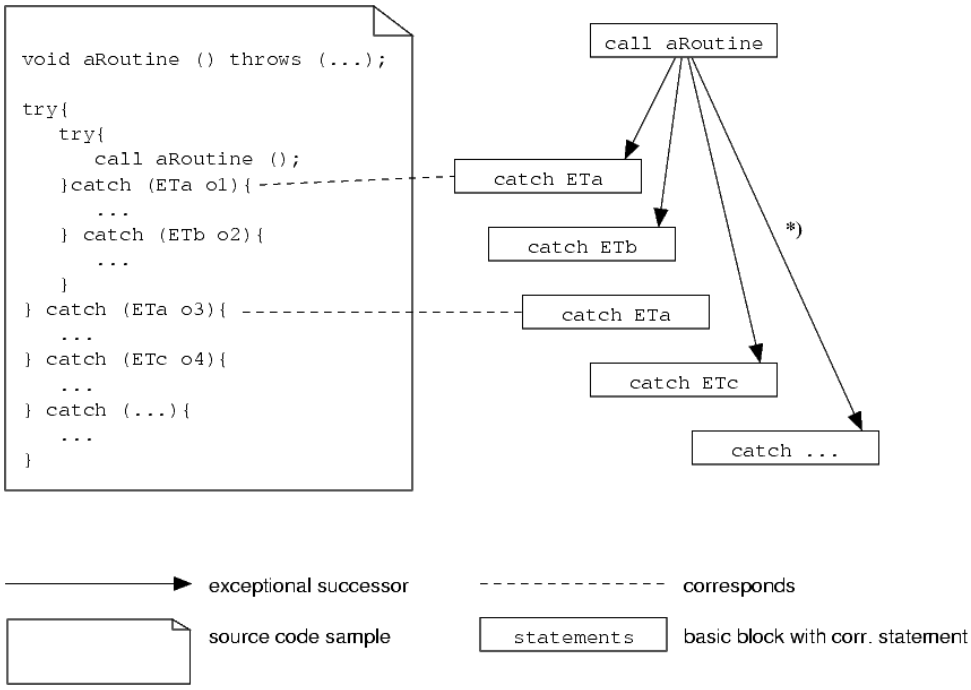

Abb. 3.6: Der beliebige Ausnahmetyp

wird der Ausnahmetyp des `catch`-Blocks in die Negativliste aufgenommen, sofern es sich dabei nicht ebenfalls um T_Any_Exception handelt. Falls ein `catch`-Block für T_Any_Exception existiert, gelten alle Ausnahmen als behandelt und es werden keine Kanten zu evtl. folgenden `catch`-Blöcken generiert. Abbildung 3.6 illustriert dies mit einem Quellcodeausschnitt und dessen Abbildung in IML. Die mit *) markierte Nachfolgerkante gilt für eine Instanz von T_Any_Exception mit der Negativliste $\{ETa, ETb, ETc\}$.

Hinweis: *Ende der C++-Erweiterung.*

Kapitel 4

Kontrollfluss und -abhängigkeit

4.1 Kontrollfluss

4.1.1 Routinenaufrufe

Bei Routinenaufrufen wird der Kontrollfluss während der Generierung des CFG zwischen dem Aufruf und der darauffolgenden Anweisung unterbrochen, falls die aufgerufene Routine Ausnahmen propagieren kann oder (eventuell) nicht zurückkehrt. Der temporär zwischen die beiden Knoten eingefügte Platzhaltergraph wird in Abschnitt 4.2.6.3 beschrieben, da er für die Kontrollabhängigkeit besondere Bedeutung besitzt.

4.1.2 Modellierung der `finally`-Blöcke

`finally`-Blöcke bereiten einige Probleme bei der Modellierung der IML-Knoten. Die jetzige Modellierung ist nicht als optimal anzusehen, aber dennoch die beste mir bekannte Lösung. Dieser Abschnitt beschreibt, was bei der Traversierung des CFG im Hinblick auf finally_calls zu beachten ist.

Anders als bei den anderen Nachfolgeknoten wird bei einem `finally`-Block keine Nachfolgerkante benutzt, sondern ein finally_call. Damit wird der besonderen Semantik Rechnung getragen. Jeder `finally`-Block kann mehrere Vorgänger besitzen (alle Anweisungen, die ein Verlassen eines `try`-Blocks veranlassen). Somit kann der `finally`-Block auch mehrere potentielle Nachfolger haben, wobei die zulässigen Nachfolger kontextsensitiv ermittelt werden müssen. Beim Besuch eines Knotens k mit einem finally_call ist der erste zu besuchende Nachfolger der Eintrittsknoten des `finally`-Blocks, auf den der finally_call verweist. Sofern k andere (normale) Nachfolger besitzt, existiert ein Pfad innerhalb des `finally`-Blocks von dessen Eintrittsknoten zu dessen Austrittsknoten. Beim Erreichen dieses Austrittsknotens ist die Traversierung mit den normalen Nachfolgern von k fortzusetzen. Existieren keine solchen Nachfolger, so existieren im `finally`-Block nur solche Pfade, welche den Kontrollfluss unbedingt von innerhalb des `finally`-Block zu Grundblöcken außerhalb des Blocks lenken (`return, ...`). Bei der Traversierung des CFG ist es ohne zusätzliche Information nicht möglich festzustellen, wann ein `finally`-Block unbedingt verlassen wurde. Eine Möglichkeit, dieses Problem zu lösen, scheint die Einführung eines neuen Grundblocktyps wie bspw. Regular_Finally_Basic_Block zu sein, der reguläre Grundblöcke innerhalb eines `finally`-Blocks repräsentiert. Da jedoch auch `finally`-Blöcke geschachtelt auftreten können (siehe Abbildung 4.1),

```
try{
  if (condition)
    return;
  if (anotherCondition)
    throw new Exception ();
}finally{
  try{
  ...
  }finally{
    if (condition)
      return;
    if (anotherCondition)
      throw new Exception ();
  }
}
```

Abb. 4.1: Geschachtelte `finally`-Blöcke

trägt diese Modellierung nicht: Eine Kante von einem Regular_Finally_Basic_Block zu einem Regular_Basic_Block würde als solch ein Übergang erkannt werden. Eine Kante von einem Regular_Finally_Basic_Block zu einem anderen Regular_Finally_Basic_Block könnte aber genauso ein solcher Übergang sein. Es ist ohne zusätzliche Knotentypen oder Attribute bei den bestehenden Grundblöcken nicht möglich, diese Übergänge im CFG zu erkennen. Da diese Information für spätere Auswertungen nicht sehr wichtig ist, wird diese Unzulänglichkeit bewusst in Kauf genommen. Die möglichen Kombinationen von Nachfolgekanten sind detailliert in Kapitel 9 ab Seite 87 dargestellt.

4.1.3 Zusammenspiel von `catch` und `finally`

`finally`-Blöcke können zwei verschiedene Aufrufer besitzen: Zum einen den letzten Grundblock innerhalb eines `catch`-Blocks, zum anderen Grundblöcke, die Ausnahmen werfen, welche aus mindestens einer `try`-Anweisung propagiert werden (dieser Fall schließt auch solche Grundblöcke in `catch`-Blöcken mit ein, die diese mit einem Transferstatement oder dem Werfen einer neuen Ausnahme verlassen). Falls ein `catch`-Block verlassen wird, weil seine letzte Anweisung abgearbeitet wurde, so wird anschließend der `finally`-Block der `try`-Anweisung aufgerufen (falls vorhanden). Somit gilt für alle Grundblöcke, von denen aus Successors_Exceptional-Kanten auf einen `catch`-Block innerhalb derselben Anweisung verweisen, dass diese *keine* finally_call-Kanten besitzen dürfen, da der entsprechende Aufruf im `catch`-Block stattfindet. In allen anderen Fällen dürfen Successors_Exceptional-Kanten und finally_call-Kanten vom selben Grundblock ausgehen. Beide Fälle sind in den Abbildungen 4.2 und 4.3 (S. 28) jeweils mit Beispielquellcode und Kontrollflussgraph illustriert.

4.2 Kontrollabhängigkeit

In diesem Abschnitt beschreibe ich die Berechnung der Kontrollabhängigkeit. Die Änderungen und Erweiterungen an den bisherigen Algorithmen werden detailliert beschrieben.

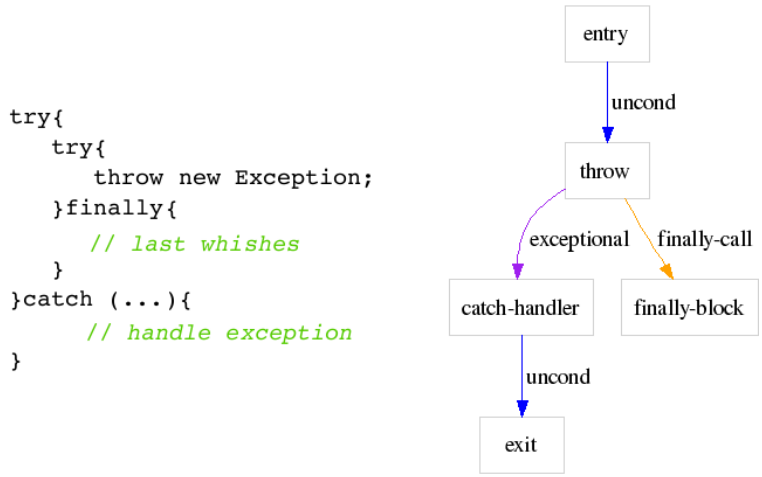

Abb. 4.2: Aufruf des `finally`-Blocks von einer `throw`-Anweisung

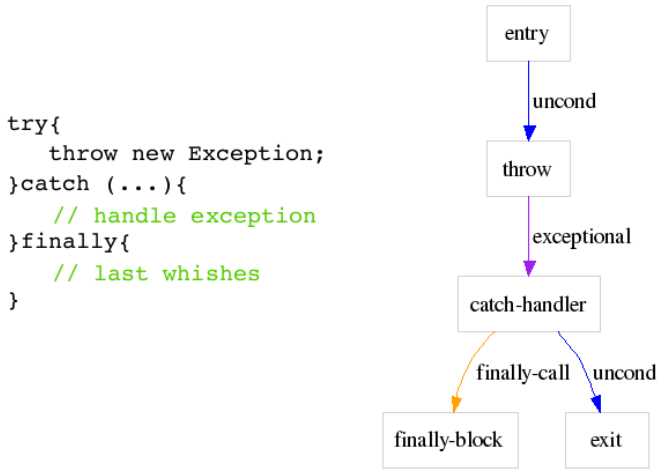

Abb. 4.3: Aufruf des `finally`-Blocks von einem `catch`-Block

4.2.1 Künstliche Hilfsknoten und -kanten

Der Begriff CFG-Entry beschreibt den Grundblock vom Typ Entry_Basic_Block, der den Beginn einer Routine im CFG darstellt. Entsprechend steht CFG-Exit für den Grundblock vom Typ Exit_Basic_Block, der den normalen Ausgang einer Routine im CFG darstellt.

Für die Berechnung der Kontrollabhängigkeit werden die intraprozeduralen CFGs um verschiedene Knoten und Kanten erweitert. Bisher war der CFG-Entry-Knoten konditional, die Wahr-Kante führte zum ersten Regular_Basic_Block, die Falsch-Kante zum CFG-Exit. Diese Konstruktion sorgte dafür, dass alle Knoten innerhalb des CFG mindestens vom CFG-Entry kontrollabhängig sind. Die Wahr-Kante bleibt von den Erweiterungen unberührt, die Falsch-Kante führt nun jedoch zum *Local Super Exit*-Knoten (LSX) (vgl. [Har98]). Dieser Knoten ist per Definition der letzte Knoten des CFG einer Routine, er besitzt keine Nachfolger. Seine Vorgänger sind alle Knoten, die die Routine entweder verlassen oder das Programm terminieren. Damit ist dieser Knoten einzig vom CFG-Entry kontrollabhängig.

Ein weiterer Hilfsknoten ist der sog. *Collective Exit* (CLX). Alle Knoten, welche die Routine verlassen (aber nicht terminieren), sind Vorgänger dieses Knotens – also der CFG-Exit-Knoten und alle Knoten der Klasse Exceptional_Exit_Basic_Block. Dieser Knoten ist entweder kontrollabhängig vom CFG-Entry (gdw. die Routine definitiv zurückkehrt) oder von dem Knoten, der kontrolliert, ob die Routine terminiert.

Sowohl Local Super Exit als auch Collective Exit sind Instanzen der Klasse Regular_Basic_Block, die jedoch als temporär markiert sind und nach der Berechnung der interprozeduralen Kontrollabhängigkeit entfernt werden.

Das folgende Beispiel zeigt Quellcode (Abb. 4.4) und dessen temporäre Darstellung (Abb. 4.5).

```
void foo (int action)
{
  if (action == RET) {                    //'if-1'
    return;                               //'return'
  }else if (action == EXC) {              //'if-2'
    trow new Exception ();                //'throw'
  }else{
    System.exit ();                       //'terminate'
  }
}
```

Abb. 4.4: Beispielquellcode für CFG-Hilfsknoten

4.2.2 Unzulänglichkeit des bisherigen Algorithmus

Bisher wurde zur Berechnung der Kontrollabhängigkeit das Paket `Basic_Block_Graph_Mapping.Domination` aus der `Reuse`-Bibliothek verwendet, um die Postdominanz zu berechnen. Eine Prämisse dieses Pakets ist, dass in einem Graphen keine unerlaubten Pfade existieren.

Definition: *Pfad*
Sei $G = (E, V)$ ein gerichteter Graph und $W = \{v_1, v_2..., v_n\}$ eine Folge von Knoten aus V. Falls für alle i aus $\{1..., n-1\}$ gilt: $(v_i, v_{i+1}) \in E$ und für alle i und j aus $\{1..., n\}$ gilt: $v_i \neq v_j \leftrightarrow i \neq j$, dann ist W ein Pfad in G.

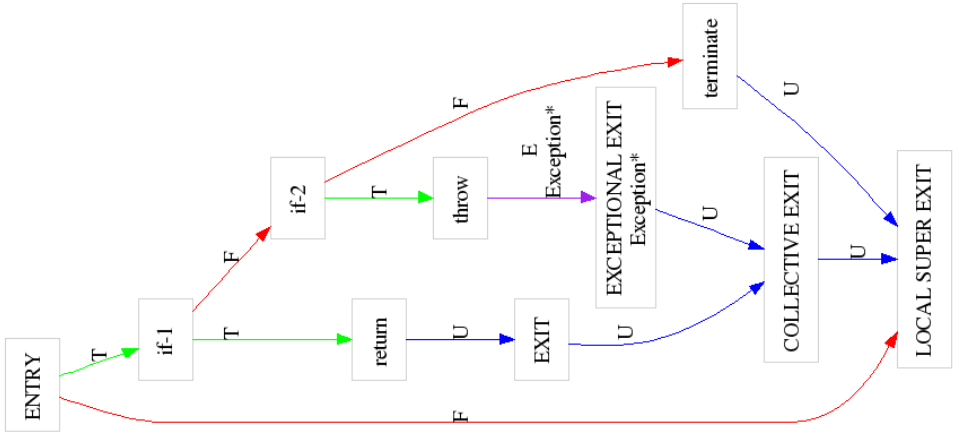

Abb. 4.5: CFG mit Hilfsknoten

Definition: *Unerlaubter Pfad*
Sei $G = (E, V)$ ein CFG und W ein Pfad in G. W ist ein unerlaubter Pfad genau dann, wenn der Pfad einen nicht möglichen Kontrollfluss darstellt.

Das Problem der unerlaubten Pfade kann sich dann ergeben, wenn Routinen-Aufrufe im CFG durch eine Kante zwischen aufrufendem Knoten und dem Eintrittsknoten der aufgerufenen Routine dargestellt werden. Bei n Aufrufen einer Routine hat deren Eintrittsknoten n eingehende Kanten. Entsprechend hat ihr Austrittsknoten n potentielle Nachfolger. Bei der statischen Analyse sind sowohl die n eingehenden als auch die n ausgehenden Kanten im Graphen abgelegt. Bei der Traversierung des Graphen muss also darauf geachtet werden, dass keine unerlaubten Pfade beschritten werden. Also muss die Traversierung kontextsensitiv sein.

Neben den Routinen-Aufrufen wird auch die Ausführung der `finally`-Blöcke als Aufruf (finally_call) modelliert. Deshalb ist selbst bei der Berechnung der Kontrollabhängigkeit auf intraprozeduraler Ebene eine kontextsensitive Traversierung notwendig. Der bisherige Algorithmus arbeitet außerdem vorwärtsgerichtet, das heisst, die Kontrollabhängigkeit wird vom kontrollierenden Knoten gesetzt. Allerdings beschränkt sich der Algorithmus darauf, nur die direkt folgenden Knoten als kontrollabhängig zu erkennen. Dieses Vorgehen ist unter der Annahme richtig, dass alle Folgeknoten soweit wie möglich vereint wurden (siehe auch Abschnitt 3.2.3, S. 13 bzw. Abschnitt 5.4, S. 42), also keine Folgen von Grundblöcken existieren, die einzig durch uncond-Kanten verbunden sind. Schon allein die Einführung des Super Exit-Knotens verstößt gegen diese Annahme, da zwischen dem Collective Exit-Knoten und dem Super Exit-Knoten *immer* eine uncond-Kante existiert. Somit liefert der bisherige Algorithmus für alle Routinen falsche Ergebnisse, die beim Aufbau der interprozeduralen Abhängigkeiten falsche Folgeergebnisse verursachen.

4.2.3 Kontextsensitive Berechnung der Postdominanz

Definition: *Dominanz*
Sei $G = (V, E)$ ein gerichteter Graph. Ein Knoten $x \in V$ dominiert genau dann einen Knoten $y \in V, x \neq y$, wenn jeder Pfad von der (eindeutigen) Wurzel des Graphen zu y den Knoten x beinhaltet oder $x = y$.

Definition: *Postdominanz*
Sei $dom(G)$ die Funktion, die zu einem gerichteten Graphen $G = (V, E)$ die Dominanz berechne. Sei F der Graph $F = (V', E)$, bei dem für alle $(v_i, v_j) \in V'$ gilt: $(v_j, v_i) \in V$. Sei $pdom(G)$ die Funktion, die die Postdominanz von G berechnet. Dann ist $pdom(G) = dom(F)$.

4.2.3.1 Algorithmus zur Dominanzberechnung

Der realisierte Algorithmus ist recht einfach und besitzt sicherlich noch Optimierungspotenzial. Er basiert auf dem Algorithmus aus [Coo01], der aus mehreren Schritten besteht. Zunächst wir die Postorder-Reihenfolge der Knoten des CFG einer Routine ermittelt. Sie startet mit dem Local Super Exit-Knoten der Routine, da alle Pfade innerhalb des CFG an diesem Knoten enden (zumindest während der Analyse, bevor die temporären Knoten entfernt wurden)[1]. Dabei werden auch die Finally-Teilgraphen beachtet: Wird ein Grundblock angetroffen, der einen Finally-Aufruf besitzt, so sind dessen Vorgänger (aus Sicht des CFG, nicht aus Sicht der umgekehrten Postorder-Traversierung) sowohl seine direkten Vorgänger als auch der Finally-Exit-Grundblock des aufgerufenen Finally-Teilgraphen. Eine Annahme des Algorithmus in [Coo01] ist, dass alle Pfade vom Startknoten (hier der Local Super Exit) des Algorithmus aus erreichbar sind. Dies trifft für Endlosschleifen, die mit unbedingten Sprungbefehlen aufgebaut sind, nicht zu. Aus diesem Grund wird nach der Bestimmung der Postorder-Reihenfolge für jeden Knoten k untersucht, ob alle Nachfolger von k ebenfalls besucht wurden. Ist dies nicht der Fall, wird derjenige Pfad verfolgt, der mit dem noch nicht besuchten Knoten beginnt. Dieser Pfad kann nicht am Local Super Exit enden, muss also einen Zyklus besitzen. Derjenige Knoten, der diesen Zyklus schließt, postdominiert alle Knoten in diesem Zyklus. Von diesem Knoten beginnend werden die Knoten dieses Teilgraphen in die Postorder-Reihenfolge vor k eingefügt. Diese Erweiterung erlaubt die korrekte Berechnung auch für zyklische Graphen. Ansonsten wurde der Algorithmus aus [Coo01] unverändert übernommen.

4.2.4 Berechnung kontrollabhängiger Knoten

Dieser Abschnitt beschreibt die Erweiterung des bisherigen Algorithmus. Kern der Erweiterung ist die Bestimmung transitiv kontrollabhängiger Knoten (in diesem Zusammenhang repräsentiert ein Knoten immer einen Grundblock). Außerdem sollte der vorhandene Quellcode möglichst wenig verändert werden. Die gefundene Lösung erfüllt beide Anforderungen.

Da der bisherige Algorithmus als rekursive Tiefensuche implementiert war, muss der kontrollierende Knoten über einen neuen Parameter mitgegeben werden. Allerdings reicht es nicht aus, nur den momentan kontrollierenden Knoten zu übergeben. Vielmehr muss ein Stapel mit allen bisher kontrollierenden Knoten übergeben werden. Aufgrund der Tiefensuche muss für jeden rekursiven Aufruf eine Kopie des Stapels angelegt werden. Der oberste Knoten des Stapels ist der aktuell kontrollierende Knoten. Beim Besuch des Eintrittsknotens einer Routine ist der Stapel leer. Der Stapel wird wie folgt verändert:

- Bei jeder Rekursion wird überprüft, ob der besuchte Knoten den aktuell kontrollierenden Knoten postdominiert. Ist dies der Fall, so wird der oberste Knoten vom Stapel entfernt, da alle von ihm kontrollabhängigen Knoten auf diesem Pfad erreicht wurden. Damit wird derjenige Knoten zum aktuell kontrollierenden, der den bisher kontrollierenden kontrolliert.

[1] Außer bedingungslosen Endlosschleifen, deren Pfad zyklisch ist.

- Wird ein Knoten κ mit bedingten Nachfolgern besucht und existiert ein Knoten π, sodass $\pi = pdom(\kappa)$, dann existieren Knoten, die von κ kontrollabhängig sind und als nächstes besucht werden. Der aktuell besuchte Knoten wird auf den Stapel gelegt.

Zu Beginn jeder Rekursion wird geprüft, ob der Stapel leer ist. Falls nicht, wird der besuchte Knoten als kontrollabhängig vom obersten Element des Stapels gekennzeichnet. Der veränderte Algorithmus arbeitet also rückwärtsgerichtet. Als Beispiel eines CFG mit Kontrollabhängigkeiten inkl. RPN (RPNs werden in Abschnitt 4.2.6.3.2 eingeführt) und Super Exit-Knoten siehe Abbildung 4.6, Seite 32.

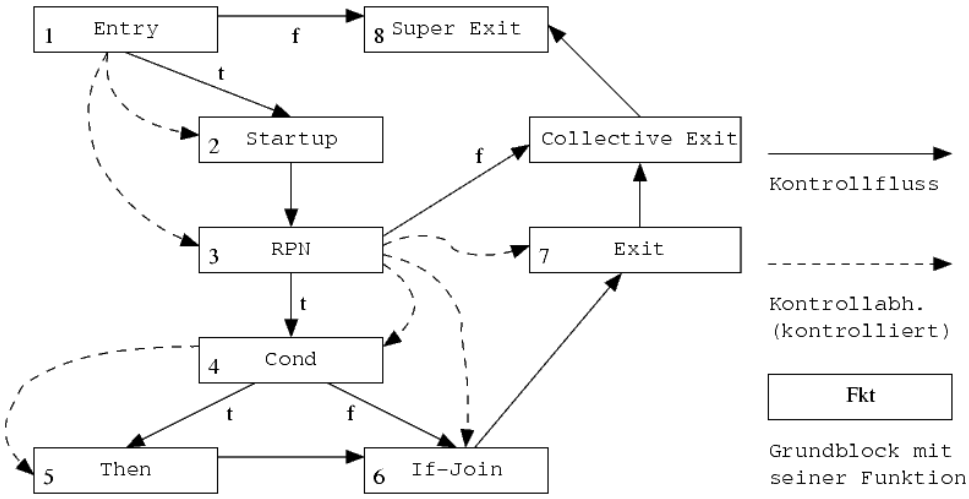

Abb. 4.6: Beispiel Kontrollabhängigkeit

4.2.5 Ausnahmen

Der Leser mag sich fragen, warum im o.g. Algorithmus nur Knoten mit bedingten Nachfolgern als kontrollierende Knoten betrachtet werden. Mit der Erweiterung der IML um Ausnahmen können schließlich auch Knoten mit einem unbedingten Nachfolger zusätzlich solche Nachfolger haben, die zur Ausnahmebehandlung dienen (`catch`- und `finally`-Blöcke). Warum werden solche Knoten nicht als kontrollierend angesehen? Es gibt zwei Ausnahmequellen: (i) Explizites Werfen einer Ausnahme oder (ii) Aufruf einer Routine, die eine Ausnahme propagiert. In (i) gibt es genau einen Nachfolger (`finally`- oder `catch`-Block oder Exceptional_Exit_Basic_Block). In diesem Fall existiert also keine Verzweigung, der Knoten kann nicht kontrollierend sein. Anders in Fall (ii). Ein solcher Knoten kann entweder keinen (Aufruf einer terminierenden Routine) oder beliebig viele Nachfolger haben (aufgerufene Routine kehrt zurück und propagiert Ausnahmen). Aber selbst in diesem Fall ist der Knoten nicht kontrollierend. Zumindest nicht direkt, denn in der aufgerufenen Routine muss ein Knoten mit bedingten Nachfolgern existieren, der kontrolliert, ob bzw. welche Ausnahme geworfen und propagiert wird. Wie diese interprozedurale Abhängigkeit aufgelöst wird, folgt im nächsten Abschnitt.

4.2.6 Interprozedurale Kontrollabhängigkeit

In der bisherigen Version von `iml2cfg` wird die Kontrollabhängigkeit für jede Routine einzeln berechnet. Dies bleibt auch weiterhin so. Eine Alternative wäre die Be-

rechnung aufgrund eines interprozeduralen CFG gewesen. Bei diesem Ansatz stehen zwei Varianten zur Auswahl (vgl. auch [Ezi99]): Zum einen das Ausrollen des CFG, zum anderen die Aufahme von Aufruf- und Rückkehrkanten in den CFG. Beide Varianten werden in den folgenden Abschnitten beschrieben.

4.2.6.1 Ausrollen des CFG

Zum einen kann der CFG ausgerollt werden (vgl. *inlined interprocedural control flow graph* in [Sin00]). Dabei wird für den Routinenaufruf der Teilgraph der aufgerufenen Routine in den CFG an der Aufrufstelle einkopiert. Hier treten zwei Probleme auf:

1. Rekursive Aufrufe machen den CFG unendlich groß.
2. Von einem Knoten können beliebig viele Kopien existieren, sodass für die Berechnung der Kontrollabhängigkeit ein Algorithmus benötigt würde, der dies berücksichtigt.

Allein der erste Punkt schließt diese Alternative aus.

4.2.6.2 Aufrufkanten

Zum anderen kann ein Aufruf auch als normale Kante vom Aufruf zum Eintrittsknoten der gerufenen Routine dargestellt werden. Dies kann zu sehr vielen Knoten führen, die Vorgänger des Eintrittsknotens sind. Entsprechend können die Austrittsknoten viele Nachfolger haben. Dabei entstehen Pfade im CFG, die bei der Ausführung des dargestellten Programms nie auftreten können. Zum Beispiel erlaubt diese Darstellung einen Pfad, der über einen Aufruf in einer Routine *alice* zum Eintrittsknoten einer aufgerufenen Routine *bob* führt und von einem Austrittsknoten in *bob* weiter zu einem Knoten in einer Routine *claire* geht ($alice \neq claire$), falls sowohl *alice* als auch *claire* Aufrufe von *bob* besitzen. Solche Pfade müssen bei sämtlichen Berechnungen ausgeschlossen werden. Neben der Größe des zu untersuchenden Graphen spricht auch die zu erwartende Komplexität der neu zu entwickelnden Algorithmen gegen diese Alternative. Die gleiche Problematik ergibt sich bei den finally_calls, wie in Abschnitt 4.2.3, Seite 30, beschrieben.

4.2.6.3 Lichte Platzhaltergraphen

Anstatt also die Berechnung der interprozeduralen Kontrollabhängigkeit von Grund auf neu zu entwickeln, wird die in [Sin00] beschriebene Methode in einer erweiterten Form angewandt. Die Grundidee ist es, Routinen weiterhin isoliert zu betrachten und die interprozeduralen Abhängigkeiten mit temporären Hilfskonstrukten zu modellieren, die als Platzhalter für tatsächliche Knoten anderer Routinen stehen.

4.2.6.3.1 Definitiv terminierende Routinen Harrold et al widmen in [Sin00] solchen Routinen besondere Aufmerksamkeit, die ein Programm *definitiv* beenden (sog. *definitely non returning procedures, DNRPs*) oder *möglicherweise* beenden (sog. *potentially non returning procedures, PNRPs*). Falls dieser Aspekt bei der Berechnung der Kontrollabhängigkeit bzw. des Kontrollflusses nicht beachtet wird, ist das Ergebnis nicht korrekt. Eine Routine ist genau dann eine DNRP, wenn

- sie eine `halt`-Anweisung besitzt oder
- der Collective Exit-Knoten statisch unerreichbar ist[2] oder

[2] Dies ist gleichbedeutend damit, dass weder der CFG-Exit noch ein Exceptional-Exit erreichbar sind.

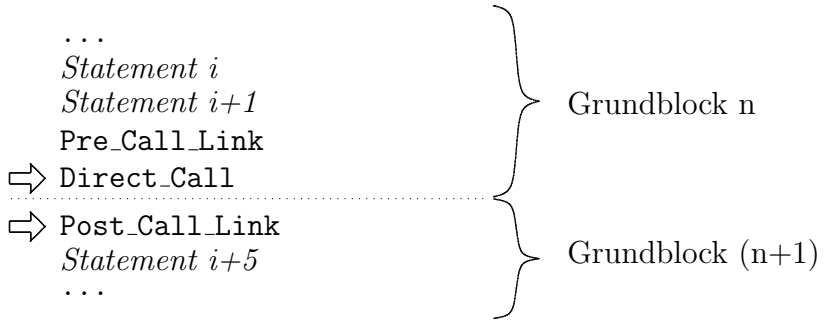

Abb. 4.7: Trennung von Aufruf und Rückkehr

- der Benutzer die Routine als definitiv terminierend definiert (beschrieben in Abschnitt 5.13, S. 56)

Es wird also nicht unterschieden, ob eine Routine aufgrund eines terminierenden Aufrufs (`System.exit` in Java o.ä.) nicht zurückkehrt oder aufgrund einer Endlosschleife[3]. Im CFG darf somit kein Pfad zwischen dem aufrufenden Knoten und dem darauffolgenden Knoten existieren. Die Trennung zwischen "Aufruf" und "folgenden Knoten" erfolgt zwischen den IML-Knotentypen Routine_Call und Post_Call_Link, da nur bei einer zurückkehrenden Routine die auf Routine_Call folgenden Anweisungen ausgeführt werden können. Siehe auch Beispiel in Abbildung 4.7.

4.2.6.3.2 Potentiell terminierender Routinen: RPNs Für den Aufruf einer potentiell nicht zurückkehrenden Routine (PNRPs) wird ein künstlicher Knoten in den CFG eingefügt, der als Rückkehrprädikat dient. Solche Knoten werden in [Sin00] als *return predicate node* (kurz RPN) bezeichnet. Sie werden zwischen dem aufrufenden Knoten und dessen Nachfolger platziert. Der RPN steht für das Prädikat der aufgerufenen Routine das über die Rückkehr entscheidet. Der auf den PNRP-Aufruf folgende Knoten wird somit kontrollabhängig von diesem Prädikat (was später durch einen oder mehrere konditionale Grundblöcke ersetzt wird). Die Wahr-Kante des Prädikats zeigt auf den dem Aufruf folgenden Konten, die Falsch-Kante zeigt auf den Super Exit-Knoten der aufrufenden Routine. Ein einfaches Beispiel zeigt Abbildung 4.8.

Diese Modellierung weicht von der von Harrold ab: Harrold fügt in den CFG einen *globalen* Endknoten n_{sx} (*super exit node*) ein, der Nachfolger *aller* Routinen-Endknoten und der Falsch-Kanten aller RPNs ist, was jedoch zu grösseren Problemen bei der Berechnung der Postdominanz führt. Es reicht jedoch aus, einen lokalen Super Exit-Knoten zu verwenden: Die Berechnung der Kontrollabhängigkeit einer Routine geschieht intraprozedural. Ob der Knoten global oder lokal ist, hat keinen Einfluss auf das Ergebnis der Berechnung.

Bei Aufrufen, deren Ziel nicht eindeutig ist (Multiple_Callees-Knoten), wird genau dann ein RPN platziert, wenn mindestens ein mögliches Aufrufziel eine PNRP ist.

[3] Wobei nicht alle Endlosschleifen bei der stat. Analyse als solche erkannt werden können.

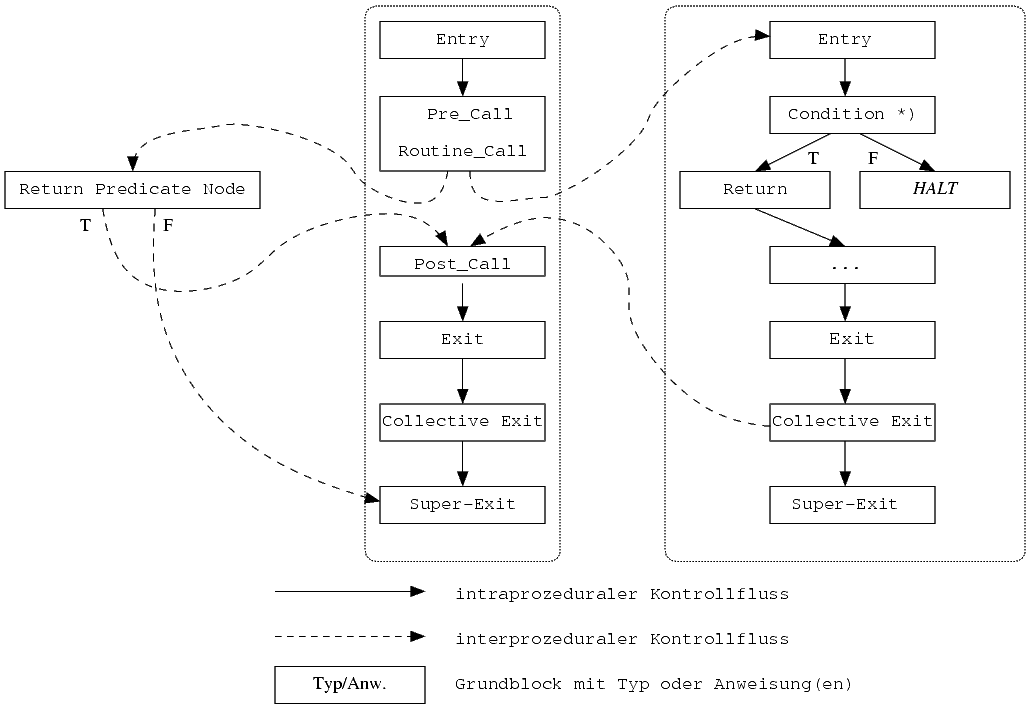

Abb. 4.8: RPN als Surrogat für potentiell terminierende Routine

4.2.6.3.3 Platzhalter für propagierte Ausnahmen: EPNs Analog wird für jeden Exceptional_Exit_Basic_Block der aufgerufenen Routine(n) ein Exceptional_Exit_Predicate-Knoten (kurz EPN) eingefügt, der stellvertretend für den Knoten steht, der diesen Ausgang kontrolliert ([Sin00] berücksichtigt keine Ausnahmen). Somit entsteht bei jedem Aufruf eine Platzhalterkonstruktion für die aufgerufene Routine. Per Definition ist der RPN immer der letzte Knoten in der Folge der Platzhalter, falls ein RPN notwendig ist. Abbildung 4.9 (S. 36) zeigt ein Beispiel.

Die true-Kante eines EPN steht für den Fall, dass eine Ausnahme geworfen wird. Sie verweist aber nicht direkt auf den zuständigen Handler im Aufrufer, sondern auf einen weiteren künstlichen Knoten mit dem Namen AES, *Artificial Exception Source*. Diese Knoten sind normale Regular_Basic_Blocks und werden als Ausnahmequelle benutzt. Notwendig wird dies durch eventuell auftretende `finally`-Aufrufe, die durch das Werfen der Ausnahme entstehen. Dieser Fall tritt genau dann ein, wenn eine propagierte Ausnahme auch vom Aufrufer propagiert wird, und mindestens ein `finally`-Block gerufen werden muss. Falls ein Handler existiert, so wird am Ende des Handlers der `finally`-Block aufgerufen. Existiert aber kein passender Handler, muss der `finally`-Block explizit gerufen werden. Dies geschieht im AES. Ohne AES müsste der `finally`-Aufruf direkt am Prädikat erfolgen, was nicht zulässig ist (ruft der false- oder der true-Ast den `finally`-Block? Oder beide?).

Falls eine Routine definitiv zurückkehrt, aber Ausnahmen propagieren kann, so entfällt der RPN im CFG. Die false-Kante des letzten EPN verweist in diesem Fall auf den Grundblock, der dem Routinenaufruf folgt.

4.2.6.4 Entfernen der Platzhalterknoten

Nach der Berechnung der Kontrollabhängigkeit werden sämtliche Kanten von und zu Platzhalterknoten entfernt. Die Reihenfolge, in der die Knoten bearbeitet werden,

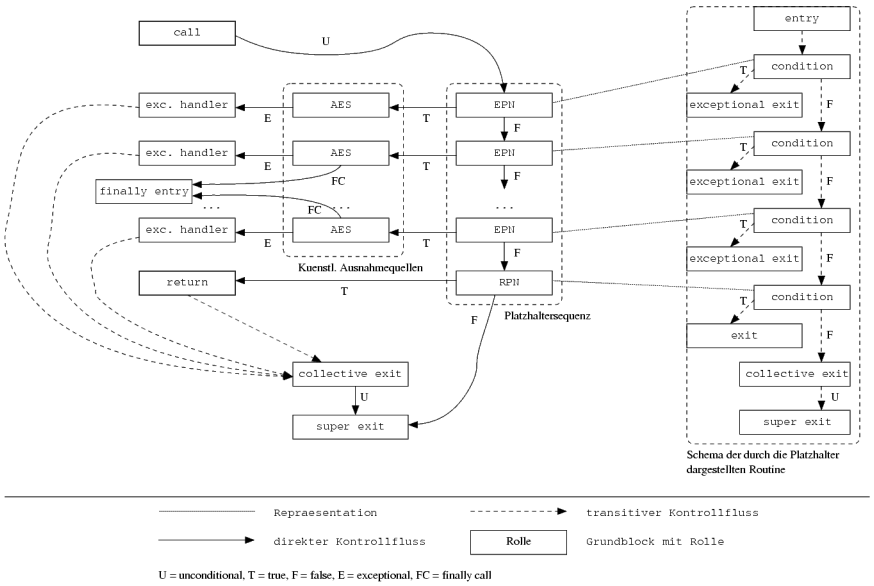

Abb. 4.9: Platzhalterknoten im CFG

ist nicht festgelegt. Erst nachdem die Kanten aller Knoten entfernt wurden, werden auch die Knoten selbst entfernt. Diese Vorgehensweise vereinfacht den Algorithmus ungemein. Je nach Knotentyp werden folgende Korrekturen vorgenommen:

- **EPNs und RPNs:** Zwischen dem eindeutigen Vorgänger des Knotens und dem folgenden EPN, RPN oder Return-Knoten wird eine neue Kante angelegt. Falls ein EPN entfernt wird, der einen unbedingten Nachfolger besitzt, wird eine Kante vom Vorgänger zu diesem Nachfolger (ein AES) angelegt. Dieser Fall tritt genau dann ein, wenn eine Routine definitiv eine bestimmte Ausnahme propagiert und nie ohne Ausnahme zurückkehrt. In Abbildung 4.10 auf Seite 37 besitzt der zu ersetzende EPN, (als "(v)" gekennzeichnet) einen bedingten Nachfolger. Der Typ (true, false, uncond) der neuen Kante (e1) entspricht dem Typ der Kante (e0) zwischen Vorgänger und zu entfernendem Knoten. Da dieser Knoten ein Prädikat darstellt, existieren Knoten, die von ihm kontrollabhängig sind. Durch die Entfernung des Knotens geht diese Information verloren. Aus diesem Grund wird der Grundblock, der den Aufruf beinhaltet zu dem Knoten, der die kontrollabhängigen Knoten des zu entfernenden Knotens kontrolliert. Dieser Grundblock ist der erste (indirekte) Vorgänger, der nicht temporär ist.

- **AESs:** Alle von einem AES ausgehenden finally_call- und successors_exceptional-Kanten (in Abbildung 4.10 mit "(e2)" gekennzeichnet) werden dem ersten nicht temporären Vorgänger zugeordnet (als "(e3)" gekennzeichnet). Dieser Vorgänger ist der Aufrufknoten, der die propagierte Ausnahme auslöst. Anschließend werden alle Kanten vom und zum AES entfernt. Ein AES kann keinen Knoten kontrollieren, da er nur unbedingte Nachfolger besitzt. Ein AES-Knoten wird genau dann entfernt, wenn der zugehörige EPN entfernt wird. Somit wird sichergestellt, dass vom AES der Aufrufer ermittelt werden kann.

- **Super Exit:** Ein Super Exit besitzt keine Nachfolger, nur genau einen Vorgänger (CLX). Die Kanten vom Vorgänger zum Super Exit werden entfernt,

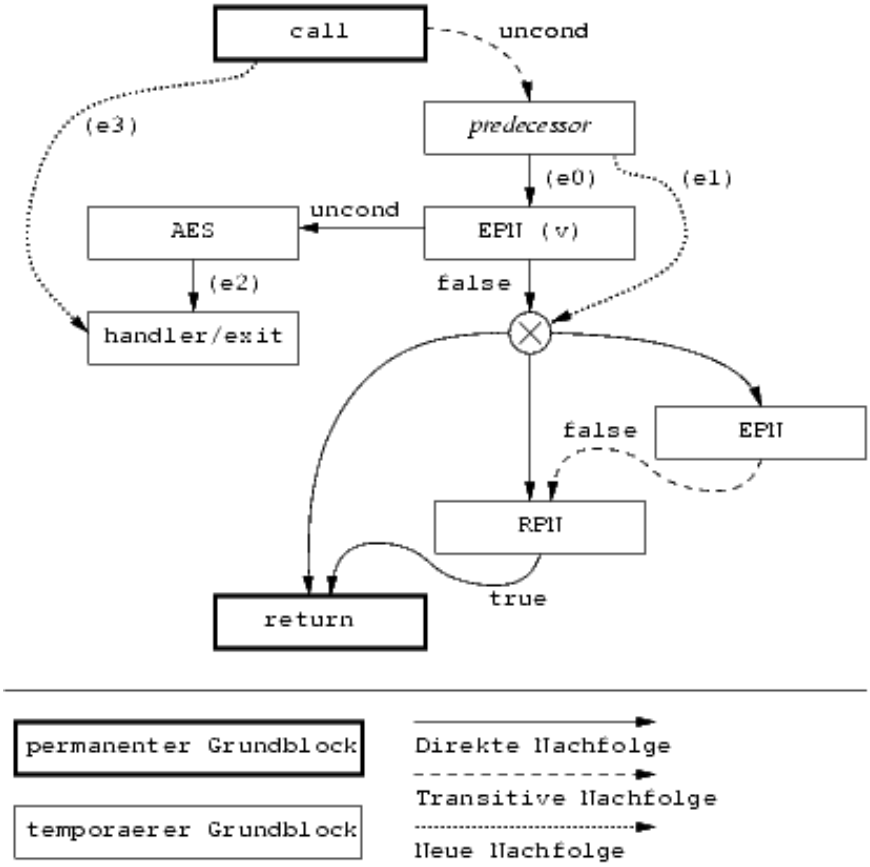

Abb. 4.10: Ersetzung eines EPN

ansonsten sind keine Korrekturen vorzunehmen. Ein Super Exit kann keinen Knoten kontrollieren, da er keinen Nachfolger besitzt.

- **Collective Exit:** Ein Collective Exit besitzt einen Nachfolger (LSX) und mindestens einen Vorgänger. Die Kanten von jedem Vorgänger zum Collective Exit werden zum LSX umgelenkt, sofern dieser noch vorhanden ist. Falls nicht, werden die Kanten entfernt.

4.2.7 Kontrollabhängigkeit bei Aufrufen

Abweichend von der allgemeinen Definition der Kontrollabhängigkeit (siehe [Kos03]) folge ich bei der Berechnung der Kontrollabhängigkeit (KA) der Definition in [Sin00]. Nach dieser Definition ist der erste Grundblock einer Routine kontrollabhängig von *jedem* Grundblock, der eine Aufrufstelle der Routine kontrolliert. Leider wird in der o.g. Arbeit diese Definition kaum begründet. Deshalb folgt hier eine kurze Argumentation.

Eine Bedingung c kontrolliert, ob bzw. wie oft eine Anweisung ausgeführt wird, die von c kontrollabhängig ist. Es existieren Fälle, in denen eine Anweisung von mehr als einer Bedingung kontrollabhängig ist, wie Abbildung 4.11 (S. 38) zeigt. Zeile 4 ist sowohl von der Bedingung in Zeile 1 als auch von der Bedingung in Zeile 5 kontrollabhängig. Zeile 1 kontrolliert, ob die Zeilen 2 bis 7 überhaupt ausgeführt werden. Zeile 5 kontrolliert die Anzahl der Schleifendurchläufe.

```
1  if (x > 42)
2  {
3  label0:
4    // do something
5    if (x > 23)
6      goto label0;
7  }
```

Abb. 4.11: Mehrfache Kontrollabhängigkeit

Das Beispiel in Abbildung 4.12 ist an [Sin00] angelehnt. Angenommen, es existieren keine anderen Aufrufstellen von `fred`, als die im Beispiel vorhandenen. Ein Aufruf von `fred` findet genau dann statt, wenn `wilma` aufgerufen wird. Nach der allgemeinen Definition der KA wäre der Eintrittsknoten von `fred` also kontrollabhängig vom Eintrittsknoten von `wilma`. Dieser kontrolliert aber nur, ob `fred` *überhaupt* aufgerufen wird. Die *Anzahl* der Aufrufe ist jedoch auch abhängig von der Bedingung in Zeile 3.

```
1  void wilma (int c)
2  {
3    while (c > 0){
4      c = c - 1;
5      fred ();
6    }
7    fred ();
8  }
```

Abb. 4.12: Kontrollabhängigkeit von Routine `fred`

Kapitel 5

Konzeption

5.1 Ablauf der Analyse

Dieser Abschnitt gibt einen Überblick über die Kontrollflussanalyse. In den folgenden Abschnitten werden die einzelnen Schritte genau beschrieben. Die Kontrollflussanalyse setzt nun an zwei Punkten an: (a) Nach dem Besuch von Knoten des Typs Routine[1] und (b) nach dem Besuch von Knoten des Typs System.

In (a) werden die folgenden Schritte ausgeführt, die keine direkten Auswirkungen auf andere Routinen haben:

1. Falls noch keine Annotation zu der besuchten Routine existiert, wird eine solche angelegt. Eine solche Annotation hält Informationen über eine Routine bereit, welche während der Analyse berechnet und verwendet werden. Im Einzelnen sind dies:
   - Typ der Routine bezüglich der Terminierung (terminiert die Routine definitiv nicht, potentiell, definitiv).
   - Die Menge aller Routinenaufrufe innerhalb der Routine (*call sites*).
   - Ein Verweis auf den lokalen Super Exit-Grundblock der Routine.
   - Eine Liste von Stapeln, die Informationen über die `try`-Blöcke incl. deren Handlern etc. enthalten. Siehe Abschnitt 6.3.5, S. 65.
2. Erzeugung von Exceptional_Exit_Basic_Blocks für alle noch unbehandelten Ausnahmen, die innerhalb der Routine direkt geworfen oder von aufgerufenen Routinen propagiert wurden.
3. Erzeugung von Exceptional_Exit_Basic_Blocks für alle Ausnahmen, die vom Benutzer explizit für diese Routine spezifiziert wurden.
4. Backpatching der Sprunganweisungen.

In (b) folgen diese Schritte, die zum Aufbau des interprozeduralen CFG notwendig sind:

1. Entfernung aller nicht erreichbaren Ausnahmekanten (s. Abschnitt 5.7). Bei der Traversierung wurden für alle gefundenen Ausnahmehandler entsprechende Grundblöcke angelegt, selbst dann, wenn diese nicht erreichbar sind.

[1] Aus historischen Gründen jedoch im Visitor für Program_Unit-Knoten platziert.

2. Entfernung aller nicht erreichbaren Exceptional_Exit_Basic_Blocks (siehe auch Abschnitt 5.8). Unerreichbare Ausnahmeausgänge entstehen dann, wenn Ausnahmen zunächst keinem Handler zugeordnet werden können, gleichzeitig aber ein `finally`-Block existiert, dessen normaler Ausgang ebenfalls unerreichbar ist. In diesem Fall geht die Ausnahme verloren, wird also nicht propagiert.

3. Binden aller propagierten Ausnahmen. Dies kann zu neuen Exceptional_Exit_Basic_Blocks führen. Die Schritte 1. und 2. dienen dazu, die Anzahl der in diesem Schritt neu entstandenen Exceptional_Exit_Basic_Blocks zu minimieren. Siehe auch Abschnitt 5.11.3.

4. Erneute Entfernung aller nicht erreichbaren Ausnahmekanten, die im vorigen Schritt neu entstanden sein können.

5. Erneute Entfernung aller nicht erreichbaren Exceptional_Exit_Basic_Blocks. Diese können im vorigen Schritt neu entstanden sein.

6. Berechnung aller potentiell nicht terminierenden Routinen (beschrieben in Abschnitt 4.2.6.3.1).

7. Erzeugen der Prädikatknoten und anderer Hilfsknoten und -kanten für propagierte Ausnahmen und potentiell nicht terminierende Routinen zur Berechnung der interprozeduralen Kontrollabhängigkeit basierend auf dem intraprozeduralen CFG.

8. Optimierung der Grundblöcke: Verschmelzen von Grundblöcken und Entfernen aller nicht erreichbaren Grundblöcke (s. Abschnitt 5.4).

9. Berechnung der Kontrollabhängigkeit (s. Abschnitt 4.2.6).

10. Entfernen aller Hilfsknoten und -kanten, die zur Berechnung der Kontrollabhängigkeit notwendig waren.

11. Erneute Optimierung der Grundblöcke, da durch die Änderungen aus Schritt 10 optimierbare Konstellationen entstehen können.

5.2 Automatische Zerstörung von Stackobjekten in `throw`-Anweisungen

Die bisherige Kontrollflussanalyse ignoriert die `throw`-Anweisung komplett. Zwar implementiert die Klasse Try_Catch_Finally_Statement das Interface End_Lifetime, es wird bisher aber nicht ermittelt, welche Stackobjekte von der Laufzeitumgebung bei einer `throw`-Anweisung zerstört werden müssen. Die Bestimmung dieser Objekte ist jedoch recht einfach. Wird ein Begin_Of_Lifetime-Knoten besucht, der sich innerhalb des guarded blocks einer `try`-Anweisung befindet, so wird ein Zeiger auf diesen Knoten in einer Liste gespeichert, die mit dem `try`-Block assoziiert ist. Analog werden die Zeiger aus der Liste entfernt, wenn ein End_Of_Lifetime-Knoten im selben `try`-Block besucht wird. Somit ist innerhalb des `try`-Blocks immer bekannt, welche Stackobjekte erzeugt wurden bzw. zu diesem Zeitpunkt existieren. Beim Erreichen eines Throw_Statement-Knotens werden die über die Liste erreichbaren Stackobjekte in dessen `End_Objects`-Menge aufgenommen. Falls die geworfene Ausnahme über mehrere `try`-Blöcke propagiert wird, werden die Stackobjekte der assoziierten Listen den jeweiligen Throw_Statement-Knoten zugeordnet.

Bezüglich des Zeitpunkts der Objekterstellung gibt es offensichtlich Unterschiede zwischen verschiedenen Compilern. So erzeugt das edg-Frontend alle Stackobjekte eines Blocks gleich zu Beginn des Blocks, während gcc (Version 2.8.1) die Objekte

```
procedure traverseSubgraph (Root : Basic_Block)
  Queue          : Stack of Basic_Blocks;
  Current        : Basic_Block;
  Statement      : Value;
begin
  Queue.Push (Root);
  while not Queue.Is_Empty do
    Current := Queue.Pop;
    mark Current as visited;
    Statement := Current.First_Statement;
    while Statement.Unequal (No_Statement) do
      ... process current statement ...
      Statement := Statement.Next_Statement;
    end while;
    enqueue unvisited successors of Current;
  end while;
end;
```

Abb. 5.1: Pseudocode für einfache IML-Graph-Traversierung

nach und nach anlegt, sich also genau an den Quellcode hält. Das oben beschriebene Vorgehen ist für beide Varianten korrekt.

Die Traversierung des IML-Graphen unter Benutzung des Visitor-Design-Patterns [Gam95] stellt eine Methode dar, wie der Graph auf sehr detaillierte Weise durchwandert werden kann. Für den Software-Entwickler bedeutet dies aber auch einen großen Aufwand, da für jeden Knotentyp eine eigene Visitor-Routine erstellt werden muss, sofern dieser Knotentyp Referenzen auf andere IML-Knoten enthält. Bei der Berechnung der Kontrollabhängigkeit war mir dieser Aufwand viel zu groß. Als effiziente Alternative kommt jetzt ein iterativer Algorithmus zum Einsatz, der die Kontrollflussinformationen ausnutzt. Mögliche Verzweigungen sind durch die Grundblöcke modelliert. Innerhalb der Grundblöcke werden die Anweisungen immer strikt sequentiell abgearbeitet (Value.CF_Next und Value.CF_Prev). Also kann der gesamte Teilgraph einer Routine vollständig und effizient traversiert werden (siehe Abbildung 5.1).

5.3 Anlegen und Entfernen von Grundblöcken

Während der Berechnung des CFG und der Kontrollabhängigkeiten werden nicht nur neue Grundblöcke angelegt, sondern auch immer wieder Grundblöcke entfernt. Zum einen ist beim Entfernen darauf zu achten, dass evtl. eingehende Kanten entsprechend behandelt werden. Dies lässt sich realisieren, indem eine Menge von Prozeduren bereitgestellt wird, die diese Funktionalität kapseln. Schwieriger wird es bei den sämtlichen Strukturen, die Referenzen auf Grundblöcke enthalten. Anders als in der IML existieren hier keine Kanten ausgehend vom Grundblock in die referenzierenden Strukturen. Um diese dennoch aktuell halten zu können, kommt das Observer-Pattern ([Gam95]) zum Einsatz. Die zentrale Prozedur namens `Notify_BB_Death` im Paket `Visitors.CFG_Gen.Support` muss immer dann aufgerufen werden, bevor ein Grundblock aus dem Speicher entfernt wird. Diese Prozedur ist das Observable. Jede Struktur, die Grundblöcke referenziert, registriert sich als Observer. `Notify_BB_Death` ruft alle Observer auf und ermöglicht es ihnen, Referenzen zu aktualisieren oder zu entfernen, wenn ein Grundblock entfernt wird.

Dieses Design erlaubt es auch zukünftigen Erweiterungen, Grundblöcke effizient und einfach zu verwalten. Ungültigen Zeigern auf Grundblöcke wird entgegengewirkt.

5.4 Optimierung von Grundblöcken

Die Größe des IML-Graphen soll minimal gehalten werden, entsprechend ist die Anzahl der Knoten zu minimieren. `iml2cfg` erzeugt hauptsächlich Grundblöcke. Ihre Anzahl wird durch die Optimierung minimal.

Bisher wurde eine Optimierung der Grundblöcke durchgeführt, sobald das Ende einer Routine erreicht wurde. Dabei wurden zum einen nicht erreichbare Grundblöcke entfernt (z.B. nach einem Sprungbefehl) und zum anderen Grundblöcke zusammengefasst, falls möglich (siehe 3.2.3, S. 13). Bei der Generierung des interprozeduralen CFG darf die Optimierung erst erfolgen, wenn der Kontrollfluss komplett berechnet worden ist. Ursache sind die Grundblöcke, welche einen Funktionsaufruf enthalten. Dieser *kann* Ausnahmen werfen. Welche Ausnahmen dies tatsächlich sind, ist erst dann bekannt, wenn die von der aufgerufenen Routine propagierten Ausnahmen bekannt sind. Die Optimierung würde also den Grundblock mit dem Funktionsaufruf und den folgenden Grundblock stets zu einem Grundblock verschmelzen, was nur genau dann korrekt ist, wenn der Aufruf definitiv zurückkehrt und keine der gerufenen Routinen Ausnahmen propagiert. Daher wird die Optimierung dann ausgeführt, wenn alle Kinder des System-Knotens besucht wurden. Es wurde folgende Einschränkung bei der Optimierung vorgenommen: Zwei Grundblöcke dürfen nur dann verschmolzen werden, falls beide vom Typ Regular_Basic_Block sind. Alle anderen Grundblocktypen besitzen eine besondere Bedeutung, die durch eine Verschmelzung verloren gehen kann. Eine weitere Sonderbehandlung ist für Exceptional_Exit_Basic_Blocks notwendig. Falls für eine Routine ein Ausnahmeausgang nur aufgrund einer Benutzerdefinition angelegt wurde, so ist dieser Ausgang nicht erreichbar. Im Zuge der Bereinigung der Ausnahmeausgänge (vgl. Abschnitt 5.8) würde ein solcher Ausgang entfernt werden. Um dies zu verhindern, können Grundblöcke als "unentfernbar" markiert werden (`Mark_BB_Unremovable`). So markierte Grundblöcke bleiben immer bestehen.

5.5 Temporäre Hilfskanten und -knoten

Hilfskanten und -knoten sind nur während der Berechnung vorhanden und werden später entfernt. In diesem Abschnitt wird erläutert, zu welchem Zeitpunkt und zu welchem Zweck bestimmte Knoten und Kanten hinzugefügt werden. Knoten und Kanten, die schon im Abschnitt 4.2 (Seite 27) beschrieben wurden, werden hier nicht nochmals erläutert.

5.5.1 Erreichbarkeit von Ausnahme-Handlern

Wird während der IML-Traversierung ein neuer Ausnahme-Handler angelegt, so wird gleichzeitig eine `exceptional`-Kante vom CFG-Entry zum ersten Grundblock des Handlers angelegt. Diese Kanten sind jederzeit einfach zu identifizieren, da im CFG ohne Hilfskanten der CFG-Entry-Knoten ausschließlich einen `uncond`-Nachfolger besitzt. Dies verhindert die Entfernung des Grundblocks während der Analyse.

5.5.2 Erreichbarkeit des Local Super Exit

Für die Berechnung der Kontrollabhängigkeit wird beim Anlegen des CFG-Entry-Knotens eine `true`-Kante zum nachfolgenden Grundblock (den sog. *Startup*-Grund-

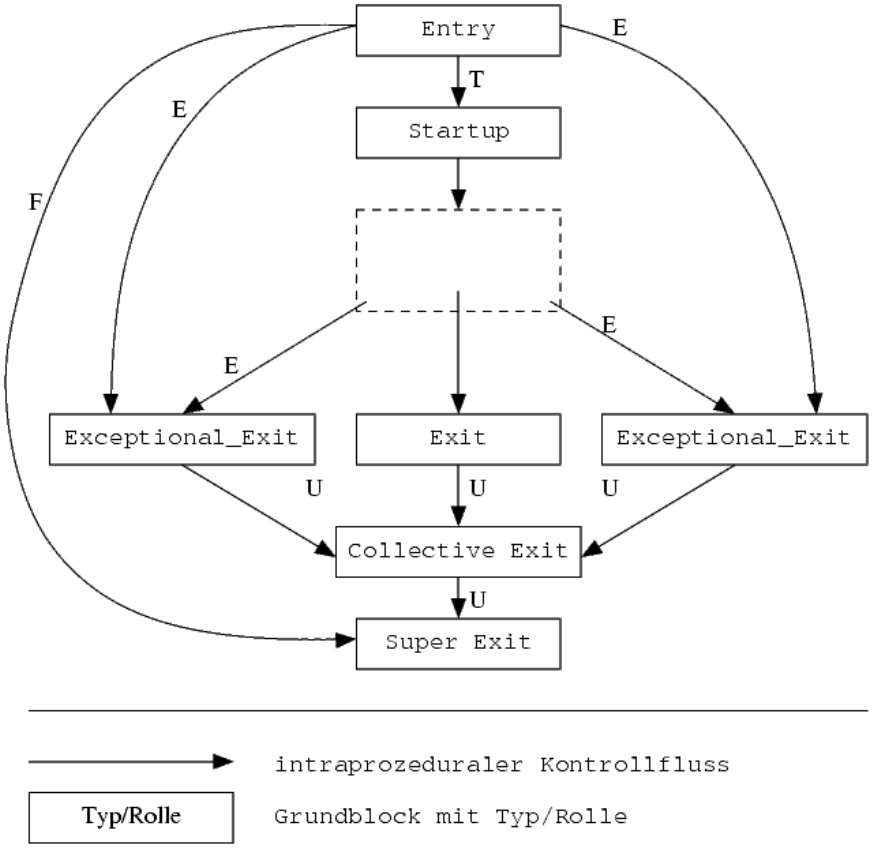

Abb. 5.2: Ausgänge des CFG

block) erzeugt. Außerdem wird eine `false`-Kante zum Local Super Exit-Knoten erzeugt. Damit wird garantiert, dass dieser erreichbar ist, und alle Grundblöcke werden durch diese Hilfskonstruktion vom CFG-Entry kontrollabhängig.

5.5.3 Eindeutiger Exit-Knoten einer Routine

Neben dem CFG-Exit-Knoten kann eine Routine auch über Exceptional-Exit-Knoten verfügen. Für die Berechnung der Kontrollabhängigkeit muss in jeder Routine genau ein Knoten existieren, der den Austritt aus der Routine modelliert. Diese Rolle wird vom Local Super Exit-Knoten übernommen, der (transitiv) Nachfolger des CFG-Exit-Knotens und aller Exceptional-Exit-Knoten ist.

5.6 Bestimmung nicht terminierender Routinen

Für die Kontrollflussanalyse muss zunächst ermittelt werden, welche Routinen definitiv oder potentiell das Programm terminieren. Dazu werden ausgehend von der Routine, welche beim Programmstart von der Laufzeit aufgerufen wird (in C/C++ und Java die Funktion `main`), alle erreichbaren Grundblöcke ermittelt. Falls keine Eintrittsfunktion existieren sollte (wie z.B. bei Bibliotheken), so werden alle Routinen als erreichbar angenommen[2] und alle Grundblöcke ermittelt, die von

[2] Diese Annahme kann noch verfeinert werden. Bspw. sind nicht öffentliche Methoden von Klassen vom Bibliotheksbenutzer nicht aufrufbar.

den CFG-Entries aller Routinen erreichbar sind. Diejenigen Routinen, deren Exit-Grundblöcke (`Exit_Basic_Block` und `Exceptional_Basic_Blocks`) nicht erreichbar sind, sind definitiv terminierend. Zusätzlich sind alle vom Benutzer als "definitively terminating" deklarierten Routinen definitiv terminierend. Es existieren verschiedene Ursachen für die Terminierung:

1. Die Routine kehrt wegen einer Endlosschleife nicht zurück.
2. Die Routine ruft (immer) eine terminierende Routine auf.

Basierend auf diesen Informationen können alle potentiell terminierenden Routinen bestimmt werden: Alle Routinen, welche eine DNRP aufrufen können (aber nicht müssen), sind potentiall terminierend (PNRP).

5.7 Entfernen unerreichbarer Kanten

Zunächst möchte ich unerreichbare Kanten definieren. Besitzt ein Knoten k sowohl die finally_call-Kanten F als auch die exceptional-Kanten E, so existieren unerreichbare Kanten, sofern der normale Ausgang des von Kante $f_i \in F$ gerufenen `finally`-Blocks unerreichbar ist. Unerreichbar sind in diesem Fall alle Kanten $e \in E$. Ebenfalls unerreichbar sind alle uncond-, true- oder false-Kanten von k. Immer noch erreichbar sind alle $f_j \in F, j > i$ (finally_call-Kanten bilden eine Liste, keine Menge).

Falls der normale Ausgang eines `finally`-Blocks unerreichbar ist, existiert eine unbedingte Transferanweisung. Dadurch wird die implizite Steuerung des Kontrollflusses verändert. Konstruktionen dieser Art sind für den Wartungsingenieur ähnlich schlecht nachzuvollziehen wie `goto`s.

Die Erkennung solcher Kanten ist aufwändig. Es werden alle Grundblöcke in allen Routinen untersucht, die mindestens eine finally_call-Kante besitzen. Ob der normale Ausgangsknoten eines `finally`-Blocks erreichbar ist, wird mit einer Tiefensuche entschieden. Um diese nicht mehrmals auf demselben `finally`-Block ausführen zu müssen, wird das Ergebnis in einer Hashtabelle gespeichert.

Wurde eine Kante als unerreichbar identifiziert, wird sie entfernt. Dadurch können unerreichbare Grundblöcke entstehen. Zu beachten sind besonders unerreichbar gewordene Exceptional_Exit_Basic_Blocks.

5.8 Entfernen unerreichbarer Exceptional_Exit_Basic_Blocks

Im vorigen Abschnitt wurde ein Grund beschrieben, weshalb Kanten wegfallen können. Aufgrund der Exceptional_Exit_Basic_Blocks werden die propagierten Ausnahmen berechnet (s. Abschnitt 5.11.3). Wenn unerreichbare Exceptional_Exit_Basic_Blocks dabei berücksichtigt werden, dann entstehen propagierte Ausnahme, die gar keine Quelle besitzen und so das Ergebnis verfälschen. Also muss hier eine Bereinigung stattfinden. Die einfachste Lösung wäre die Optimierung der Grundblöcke, da dabei auch alle unerreichbaren Grundblöcke entfernt werden (s. Abschnitt 5.4). Dabei werden *alle* Grundblöcke untersucht. Dieser Mehraufwand kann vermieden werden, da jeder Routine-Knoten eine Liste seiner Exceptional_Exit_Basic_Blocks besitzt. Die Vorgänger dieser Knoten besitzen immer eine successor_exceptional-Kante zu dem Ausgang. Es genügt also, die Anzahl der direkten Vorgänger eines Exceptional_Exit_Basic_Block zu prüfen. Existieren keine, so ist der Knoten unerreichbar. Natürlich kann ein Knoten auch dann unerreichbar sein, wenn er einen direkten Vorgänger besitzt (transitiv unerreichbar). Solche Knoten müssen zuvor mit der Grundblockoptimierung entfernt worden sein. Durch das Entfernen unerreichbarer Kanten (s.o.) können aber keine neuen transitiv unerreichbaren Knoten entstehen.

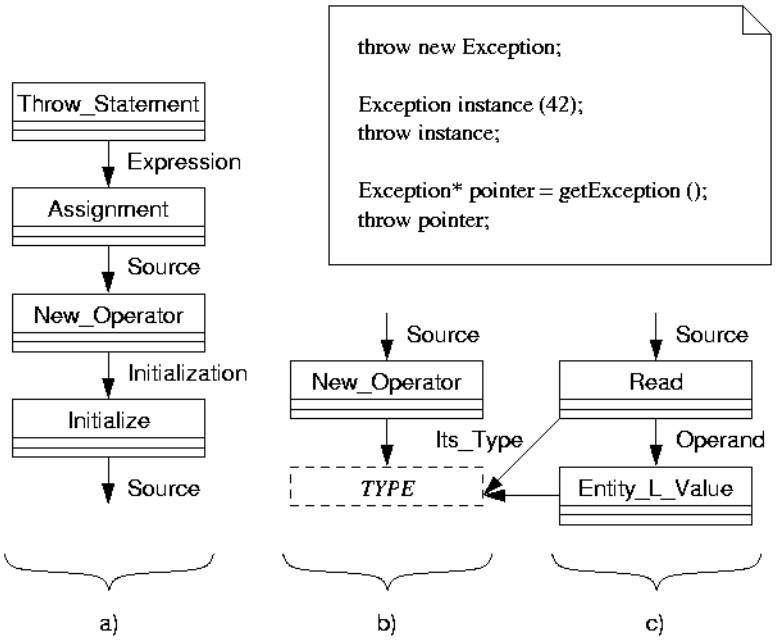

Abb. 5.3: C++ `throw`-Anweisungen in IML

5.9 Typbestimmung einer Ausnahme

Die Typbestimmung ist nur bei `throw`-Anweisungen notwendig. Propagierte Ausnahmen beruhen letztendlich entweder auf einer solchen oder auf einer expliziten Deklaration vom Benutzer. Der Throw_Statement-Typ besitzt das Attribut Expression vom Typ Value. Der von ihm gelieferte Typ muss bestimmt werden. Außerdem muss ermittelt werden, ob das Argument der Anweisung ein `operator new` ist. In diesem Fall kann davon ausgegangen werden, dass der Typ der Ausnahme eindeutig bestimmt ist. Andernfalls muss der Ausnahmetyp um seine Basistypen erweitert werden, falls es sich um einen Zeiger handelt.

5.9.1 Besonderheiten der IML-Modellierung

`Cafe++` folgt dem ANSI-Standard, was das Werfen einer Ausnahme betrifft. Bei jedem `throw` wird zunächst eine Kopie des Ausnahmeobjekts angelegt (vgl. Abschnitt 2.9.2, S. 11). Also ist das Expression-Attribut immer eine Zuweisung, die dem globalen Ausnahmezeiger eine neue Ausnahmeinstanz zuweist (Teilgraph a) in Abbildung 5.3), die auf verschiedene Arten initialisiert werden kann. Die Initialisierung entscheidet darüber, ob der Ausnahmetyp eindeutig bestimmt werden kann. Der Typ ist mittels des Its_Type-Attributs am Initialize-Knoten festgelegt. Die Teilgraphen b) und c) zeigen verschiedene Quellen bei der Initialisierung.

Auch das Java-Frontend `jafe` folgt dieser Modellierung. Es stehen jedoch aus genannten Gründen keine von `jafe` generierten IML-Dateien zur Verfügung, um die Angaben in [Kna02], Kapitel 3.2.7 zu verifizieren.

5.9.2 Mögliche Argumente

Die Typbestimmung einer zu werfenden Ausnahme kann in die folgenden Fälle aufgegliedert werden:

5.9.2.1 Explizite Instanziierung

Ist das Argument ein `new`-Aufruf, so bestimmt dessen Typ den Ausnahmetyp eindeutig. Im folgenden Beispiel wird eine Ausnahme des Typs `Exception_Type` geworfen. Es kann keine Ausnahme eines anderen Typs geworfen werden.

Beispiel: `throw new Exception_Type(...);`

5.9.2.2 Primitive Typen

Ist das Argument ein primitiver Typ, so ist der Ausnahmetyp eindeutig festgelegt (nur C++).

Beispiel: `throw ''flight altitude below ground is not allowed!'';`

5.9.2.3 Objektreferenz

Ebenfalls eindeutig ist die Angabe eines Objekts (Beispiel in C++-Syntax).

Beispiel:
```
Exception exc;
...
throw exc;
```

5.9.2.4 Zeiger

Mehrdeutig ist die Angabe einer Klasse t (obiges Beispiel in Java-Syntax). Die konservative Typbestimmung ermittelt hier alle möglichen Typen wie folgt:

Definition: *Konservative Typbestimmung bei Klassentypen*

- Sei *subclasses* (c) die Menge aller Klassen, die von c abgeleitet sind.
- Sei *implementing_classes* (i) die Menge aller Klassen, die das Interface i implementieren.

Die Menge aller Typen, auf die t zeigen kann, *possible_types* (t), berechnet sich wie folgt:

- Falls t ein Klassentyp ist, so gilt:
  $possible_types\ (t) = \{t\} \cup subclasses\ (t)$
- Falls t ein Interfacetyp ist, so gilt:
  $c = implementing_classes\ (t)$
  $possible_types\ (t) = \{c\} \cup subclasses\ (c)$

5.9.2.5 Direkte und indirekte Funktionsaufrufe

Eine `throw`-Anweisung kann als Argument auch einen Funktionsaufruf besitzen. Aufgerufene Funktionen liefern einen Rückgabewert. Je nach dessen Typ wird wie oben beschrieben verfahren. Der Typ ergibt sich aus der Quelle der Initialisierung der neuen Ausnahmeinstanz im new-Operator (siehe c) in Abbildung 5.3).

Beispiel: `throw getException ();`

5.9.2.6 Zusammengesetzte Ausdrücke

Die letzte Variante des `throw`-Arguments sind zusammengesetzte Ausdrücke. Auch diese liefern letztendlich einen Rückgabewert eines definierten Typs. Mit diesem Typ wird wie in den vorangegangenen Abschnitten beschrieben verfahren. Auch hier ergibt sich der Typ aus der Quelle der Initialisierung der neuen Ausnahmeinstanz im new-Operator.

Beispiel: `throw (*(functionPtr())).getException ();`

5.10 Rethrow von Ausnahmen

Das erneute Werfen einer Ausnahme ist nur innerhalb eines `catch`-Blocks möglich (es existiert sonst nirgends eine aktuelle Ausnahme). Sowohl C++ als auch Java bieten diese Möglichkeit, indem für die `throw`-Anweisung kein Parameter angegeben wird. Trifft die statische Analyse bei der IML-Traversierung auf eine solche Anweisung, kann nicht *genau* entschieden werden, welche Ausnahmetypen in die umgebende `try`-Anweisung weitergeworfen werden. Nur eine sehr ungenaue Berechnung ist möglich: Der im `catch`-Handler angegebene Typ und alle seine Subtypen könnten von diesem Handler gefangen und damit auch weitergeworfen werden. Fängt der Handler jedoch Ausnahmen eines beliebigen Typs (T_Any_Exception), so könnte er alle im Typsystem bekannten Typen (C++) bzw. alle von `java.lang.Throwable` abgeleiteten Klassen (Java) weiterwerfen. Dies würde die Anzahl der propagierten Ausnahmen dramatisch erhöhen und die Ergebnisse stark verfälschen. Die Menge der weitergeworfenen Ausnahmetypen ist die Menge der Ausnahmetypen, die von diesem Handler tatsächlich gefangen werden. Diese Information wird nach der vollständigen Traversierung.

5.11 Behandlung geworfener Ausnahmen

In den vorigen Abschnitten habe ich beschrieben, wie Ausnahmen entstehen können und wie ihr Typ bestimmt wird. In diesem Abschnitt gehe ich nun genauer darauf ein, was mit geworfenen Ausnahmen passiert, welche Strukturen während der Traversierung einer Routine aufgebaut werden und wie Ausnahmen an ihren Handler gebunden werden. Außerdem gehe ich detailliert auf die Propagierung von Ausnahmen ein und begründe die gewählte Modellierung.

5.11.1 Explizite Ausnahmen innerhalb einer Routine

Während der Traversierung einer Routine wird für jeden beginnenden `try`-Block ein Datensatz angelegt (vgl. Paket `Try_Stacks`, Abschnitt 6.3.5). Dieser enthält Verweise auf den ersten und letzten Grundblock des `finally`-Blocks sowie eine Liste aller Eintrittsgrundblöcke der Handler der `try`-Anweisung und der von ihnen gefangenen Ausnahmetypen. Aufgrund der möglichen Schachtelung von `try`-Anweisungen enthält der Datensatz ebenfalls eine Liste aller eingeschachtelten `try`-Anweisungen. Zusätzlich wird die Liste der noch nicht behandelten Ausnahmen gesichert und durch eine neue, leere Liste ersetzt. Wird eine `throw`-Anweisug während der Traversierung angetroffen, so wird zunächst die Menge der von ihr möglicherweise geworfenen Ausnahmetypen bestimmt. Jeder einzelne Ausnahmetyp wird anschließend zusammen mit dem Grundblock der `throw`-Anweisung registriert. Zwei Listen sind davon betroffen: Zum einen die globale Liste aller Ausnahmequellen, zum anderen die Liste der noch nicht behandelten Ausnahmen. Wird während der Traversierung das Ende eines `try`-Anweisung angetroffen, so sind folgende Informationen bekannt:

- Erster Grundblock des `finally`-Blocks (falls vorhanden)
- Alle Handler der `try`-Anweisung mit dem von ihnen behandelten Ausnahmetyp
- Alle Ausnahmen, die innerhalb des `try`-Blocks explizit geworfen wurden.

Es folgt eine Iteration über alle Ausnahmetypen aller Handler der `try`-Anweisung. Existiert eine passende Ausnahme in der Liste der nicht behandelten Ausnahmen, so wird diese an den Handler gebunden und aus der Liste entfernt. Für diejenigen Ausnahmen, die in der Liste verbleiben, wird der `finally`-Block aufgerufen, und die Ausnahmen werden an die umgebende `try`-Anweisung weitergegeben. Die gesicherte Liste der noch nicht behandelten Ausnahmen wird wieder hergestellt.

5.11.2 Modellierung propagierter Ausnahmen

Falls eine Ausnahme nicht innerhalb einer Routine behandelt wird, wird sie zur aufrufenden Routine propagiert. Es ist also notwendig, im IML-Graphen diese Information unterzubringen. Wird eine Ausnahme propagiert, so wird die Routine, in der sie entstand, verlassen. Also stellt dies eine besondere Art von Ausgang dar. Anders als beim normalen Verlassen wird nun aber im Aufrufer nach einem passenden Handler gesucht, etc. Es ist also naheliegend, einen speziellen Exit_Basic_Block zu verwenden. Die Klasse Exceptional_Exit_Basic_Block ist von Exit_Basic_Block abgeleitet und besitzt das zusätzliche Attribut Exception_Type. Somit existiert für jeden propagierten Ausnahmetyp genau ein solcher Ausgang. Ich habe schon an verschiedenen Stellen erwähnt, dass auf die Größe des IML-Graphen geachtet werden muss. Die gerade beschriebene kanonische Lösung ist offensichtlich nicht optimal in Bezug auf den Platzbedarf.

Eine Idee, die ich verfolgt habe, war in einem Ausgang eine Ausnahmeklasse abzüglich einer Negativmenge zusammenzufassen. Ein Ausgang mit leerer Negativmenge würde einen Ausnahmetypen mit allen seinen Subklassen zusammenfassen. Hintergrund dieser Überlegung ist, dass neben genau einem Ausnahmetypen eben auch ganze Klassen von Ausnahmetypen geworfen werden können (`throw` mit Zeiger, siehe Abschnitt 5.9). Ein Ausnahme-Handler kann entweder alle Typen dieser Klasse behandeln oder einen Teilbaum aus deren Subtyphierarchie. Für die behandelten Klassen ist die Negativmenge gedacht. Diese Modellierung braucht in fast allen Fällen weniger Platz als die kanonische. Sie besitzt aber Nachteile. Einerseits sind Funktionen auf den Klassenmengen zu implementieren, die sich als recht komplex herausgestellt haben. Andererseits sind Fälle denkbar, bei denen viele Informationen verloren gehen: Existiert im Quellcode genau eine Basisklasse für alle Ausnahmen, und wird ein Zeiger auf diese Basisklasse in einer `throw`-Anweisung verwendet, so besitzt die Routine genau einen Exceptional_Exit_Basic_Block. Alle weiteren Ausnahmen, die von dieser Routine propagiert werden, werden mit demselben Ausgang dargestellt, da alle Ausnahmetypen Subklassen der Basisklasse sind. Der kontrollierende Grundblock dieses Ausgangs würde um so mehr an Aussagekraft verlieren, je mehr Stellen existieren, von denen Ausnahmen propagiert werden. Die kanonische Darstellung besitzt neben dem Nachteil des Platzbedarfs diese Vorteile:

- Die notwendigen Funktionen sind nicht komplex.
- Die Darstellung ist leicht verständlich.
- Die Darstellung ist präzise.

Somit stellt die kanonische Darstellung die beste mir heute bekannte Lösung dar. Sie ist aber einer der Punkte, wo Optimierungspotenzial besteht.

5.11.3 Iterative Analyse propagierter Ausnahmen

Nach der vollständigen Berechnung der intraprozeduralen Kanten steht eine erste Menge von Ausnahmetypen fest, die eine Routine propagiert. Diese umfasst zunächst jedoch nur die explizit in der Routine geworfenen und nicht gefangenen Ausnahmen. Um die von aufgerufenen Routinen propagierten Ausnahmen ebenfalls berücksichtigen zu können, werden für jede Routine alle in ihr enthaltenen Aufrufstellen untersucht. Die von der aufgerufenen Funktion propagierten Ausnahmen werden an die entsprechenden Handler gebunden oder an den entsprechenden Exceptional_Exit_Basic_Block weitergeleitet. Dazu werden intern verschiedene Strukturen verwaltet. Die wichtigsten sind

- eine Liste aller Routinen im IML-Graph,
- eine Hashmap, die zu einer gegebenen Routine eine Liste aller Aufrufstellen dieser Routine liefert und
- eine Hashmap, die zu einem Grundblock einen Stapel der umgebenden `catch`-Handler liefert. Hier kann man sich auf die Grundblöcke beschränken, die eine Aufrufstelle beinhalten, da es keine anderen Quellen von propagierten Ausnahmen geben kann.

Auf den ersten Blick scheint es ausreichend, die Routinen in der Reihenfolge zu untersuchen, wie sie entstehen würde, wenn man den Aufrufgraph an den Blättern beginnend zur Wurzel hin durchläuft. Zyklische Aufrufe blieben dadurch aber völlig unbeachtet!

Hat der Aufrufgraph die Tiefe d, so muss diese Berechnung maximal $d + 1$ mal ausgeführt werden, da bei jeder Berechnung einer Routine genau diejenigen Ausnahmen neu berücksichtigt werden, die von einer direkt aufgerufenen Routine stammen. Nach d Iterationen wurden die Ausnahmen also auch von dem am weitesten entfernten Blatt bis zur Wurzel propagiert. Der größtmögliche Zyklus besteht aus $d+1$ Knoten, sodass nach $d + 1$ Iterationen alle Knoten im Zyklus vollständig berechnet sind.

5.12 Bestimmung möglicher Aufrufziele

Es existieren bereits Zeigeranalysen in bauhaus, wie Steensgard oder DAS (vgl. [Fro04]). Diese benötigen jedoch den Kontrollfluss, können also nicht benützt werden. Dieser Abschnitt beschreibt, wie die Ziele verschiedener Aufruftypen ermittelt werden.

5.12.1 Polymorphe Aufrufe

Bei der Ausführung objektorientierter Programme wird bei virtuellen Methoden erst zur Laufzeit bestimmt, welche Implementation aufgerufen wird. Dieser Vorgang ist auch als *dynamic dispatch* bekannt. Ausschlaggebend ist der konkrete Typ des Objekts, auf das ein Zeiger zeigt. In der IML werden solche Aufrufe als Virtual_Call-Knoten dargestellt. Eine konservative Bestimmung der möglicherweise aufgerufenen Methoden kann anhand der Vererbungshierarchie erfolgen: Die Menge aller Methoden, die Ziel eines polymorphen Aufrufs sein können, $possible_callees\ (t, s)$, wobei t der Typ des Objekts ist, auf den der Zeiger zeigt und s die Signatur der aufgerufenen Methode ist, berechnet sich wie folgt:

- Falls p ein Klassentyp ist, so gilt:
  $possible_classtypes\ (t) = \{t\} \cup subclasses\ (t)$

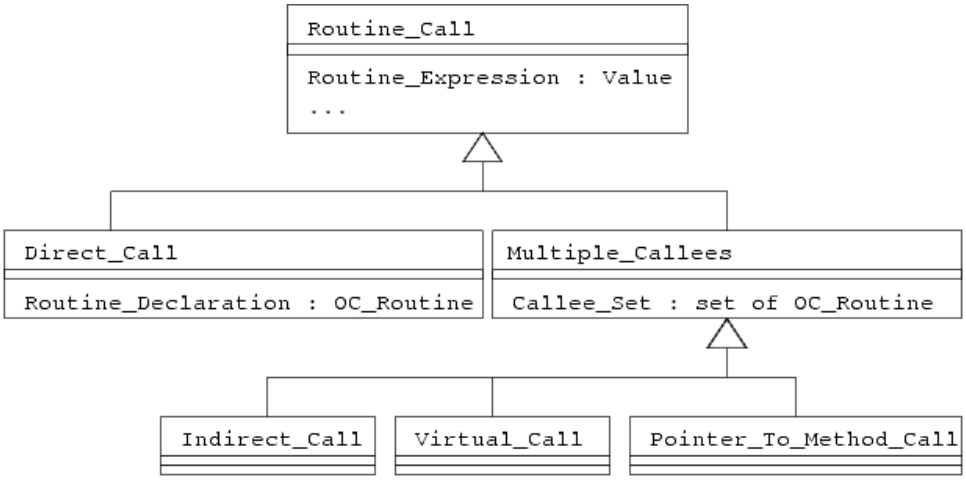

Abb. 5.4: IML-Hierarchie der Aufrufe

- Falls t ein Interfacetyp ist, so gilt:
  $c = \mathit{implementing_classes}\ (t)$
  $\mathit{possible_classtypes}\ (t) = \{c\} \cup \mathit{subclasses}\ (c)$
- $\mathit{possible_callees}\ (t, s) =$ Alle virtuellen Methoden mit der Signatur s, die zu den Klassen in $\mathit{possible_classtypes}\ (t)$ gehören.

Diese Berechnung ist äußerst konservativ und ermittelt alle *möglichen* Ziele. Typischerweise sind die *tatsächlichen* Ziele eine Teilmenge davon. Es existieren verschiedene Möglichkeiten, die Menge einzuschränken. Die Methoden unterscheiden sich stark in ihrer Komplexität. Im nächsten Abschnitt wird eine der einfachsten Methoden beschrieben.

Im Paket `types` ist die Funktion `Determine_Possible_Callees` untergebracht, die zu jedem Routine_Call-Knoten die Menge der möglichen Ziele zurückgibt. Die oben beschriebene Auflösung polymorpher Aufrufe wird nur für Knoten des Typs Virtual_Call angewendet. Die nachfolgend beschriebene Einschränkung kann in dieser Funktion angewendet werden.

5.12.2 Einschränkung der Objekttypen

Am einfachsten lassen sich die möglichen Objekttypen ganz allgemein einschränken, wenn man verfolgt, von welchen Klassen überhaupt Instanzen angelegt werden. Objekte können entweder automatisch auf dem Stack oder mit dem `operator new` auf dem Heap angelegt werden. Die Instanziierung auf dem Stack hat einen Aufruf des Konstruktors der Klasse zur Folge. In der IML ist dieser Aufruf als Direct_Call abgebildet. Ist die aufgerufene Routine ein O_Constructor-Knoten, handelt es sich um einen Konstruktoraufruf. Die Klasse, zu der er gehört, kann einfach ermittelt werden. Da für die Berechnung der Kontrollabhängigkeit alle Aufrufe im `Utils`-Paket registriert werden (`Register_Call`), wird in dieser Routine auch geprüft, ob ein Konstruktor aufgerufen wird. Ist dies der Fall, so wird die Klasse des Konstruktors in einer Hashtabelle vermerkt. Bei der Instanziierung auf dem Heap ist der Aufruf des Konstruktors nur über einen Knoten des Typs Entity_L_Value zu erreichen. Die Quellen der Entity_L_Values werden bei der Traversierung aber nicht untersucht. Der Typ des neuen Objekts kann aber von dem Typ, den `operator new` liefert, bestimmt werden (`operator new` liefert immer einen Zeiger). Der Visitor des Operators vermerkt diesen Typ ebenfalls in der Hashtabelle. Damit sind alle Quellen

```
class alice : public someBaseclass
{
// whether constructor nor destructor shown
public: virtual int foo () return 42;
};

class bob : public alice
{
// whether constructor nor destructor shown
// no method redefinitions of 'alice'
};

void sampleProcedure ()
{
  bob* obj = new bob;
  int retVal = obj->foo ();          // <-- (A)
};
```

Abb. 5.5: Sonderfall bei der Methodeneinschränkung

einer Objektinstanziierung abgedeckt. Nach der vollständigen Traversierung eines Systems sind in der Hashtabelle genau die Klassen vermerkt, von denen Instanzen erzeugt wurden. Diese Information wird zur Einschränkung der möglichen Ausnahmetypen und der Ziele polymorpher Aufrufe verwendet. Von abstrakten Klassen können keine Instanzen erzeugt werden, sie müssen also je nach Verwendung besonders behandelt werden. Bei polymorphen Aufrufen werden Methoden aus abstrakten Klassen dann nicht beachtet, wenn sie ebenfalls abstrakt sind. Außerdem werden solche Methoden nicht als mögliche Aufrufziele eingestuft, die zu Klassen gehören, von denen keine Instanzen erzeugt wurden. Diese Einschränkung ist jedoch zu ungenau, wie das Beispiel in Abbildung 5.5 zeigt: Der Aufruf (A) der Methode `foo` veranlasst die Ausführung der Implementierung in der Klasse `alice`, von der jedoch keine Instanz existiert. Ist eine virtuelle Methode in einer Klasse nicht redefiniert, so muss diejenige (Re-) Definition der Methode als erreichbar gelten, die in der am wenigsten weit entfernten Basisklasse vorhanden ist. Falls diese nicht existiert, ist die Methode abstrakt.

Bei den Ausnahmetypen werden abstrakte Klassen beibehalten. Dafür sprechen folgende Argumente:

1. Würden nur konkrete Klassen betrachtet, gingen Informationen verloren und die Übersichtlichkeit könnte leiden, obwohl insgesamt weniger Ausnahmetypen vorhanden wären.

2. Der Programmierer wird sich etwas dabei gedacht haben, wenn er eine abstrakte Klasse definiert oder als Ausnahmetyp verwendet.

3. Neben konkreten Klassen können auch abstrakte Klassen von einem `catch`-Handler gefangen werden.

4. Am IML-Graph ist leicht zu erkennen, ob eine nicht behandelte Ausnahme einen abstrakten Typ besitzt. Falls eine Routine nur abstrakte Ausnahmen propagiert, muss der Benutzer entscheiden, wie er dies wertet. Sofern der komplette Quellcode bei der IML-Generierung zur Verfügung stand, kann die Routine keine Ausnahme propagieren, da alle tatsächlich auftretenden Ausnahmen gefangen wurden. Andernfalls können durchaus noch Ausnahmen existieren, die ihren Ursprung z.B. in Bibliotheksroutinen haben.

Gegen dieses Vorgehen spricht:

1. Es können niemals Instanzen abstrakter Klassen geworfen werden. Also kann man diese Ausnahmen weglassen.

2. C++ und Java geben dem Programmierer sehr viel Freiraum, er braucht nicht über jedes Detail nachzudenken. Programme enthalten also auch eine gewisse Menge Unüberlegtes.

3. Der Benutzer muss Entscheidungen treffen.

5.12.3 Konservative Zielbestimmung von Funktionszeigern

Die Routinen, die von einem Funktionszeiger[3] aufgerufen werden, können z.B. durch eine Datenflussanalyse genau bestimmt werden. Eine solche Analyse ist aber nicht Teil dieser Diplomarbeit. Eine konservative Abschätzung kann jedoch sehr einfach getroffen werden. Bei jeder Zuweisung der Adresse einer Routine R (in Form einer Funktion oder Prozedur, nicht jedoch einer Methode) an eine Variable kann R global als Ziel eines Funktionszeigers betrachtet werden. Diese einfache, aber effiziente Vorgehensweise wurde in `iml2cfg` umgesetzt. Die o.g. Zuweisungen besitzen immer ein Entity_L_Value. Mit der schon vorhandenen Funktion `Address_Taken.Of_Routine` kann sehr einfach festgestellt werden, ob die Quelle der Zuweisung eine Funktionsadresse ist. Ist dies der Fall, wird die Routine in die globale Liste der Funktionszeigerziele aufgenommen. Eine notwendige Erweiterung ist die Betrachtung aller Knoten vom Typ Pointer_To_Member_Literal, da diese zur Zuweisung von Methoden benutzt werden, wie im folgenden Beispiel:

```
int (Vehicle::*vfp)() = &Vehicle::virtMethod;
```

Routinen, auf die Pointer_To_Member_Literal verweist, werden ebenfalls in die Liste der Funktionszeigerziele aufgenommen.

Werden Funktionszeiger auf Methoden von Klassen oder Strukturen verwendet, so muss auch hier die Polymorphie beachtet werden. Die Deklaration eines Funktionszeigers enthält keinerlei Aussagen darüber, ob die gerufene Methode virtuell ist oder nicht. Aus diesem Grund ist die Menge der möglicherweise gerufenen Methoden

[3] Ich verwende den bekannten Begriff Funktionszeiger, obwohl es Routinenzeiger heißen müsste.

dann besonders groß, wenn der Funktionszeiger auf eine virtuelle Methode verweisen kann. Die Menge der möglichen Funktionszeigerziele wird wie folgt ermittelt und eingeschränkt:

1. Zunächst werden alle Methoden derjenigen Klasse untersucht, die zu der Klasse B gehören, auf die T_Pointer_To_Method verweist (siehe auch Abbildung 5.7, S. 55). Wird eine Methode gefunden,

   - deren Parameter zur Definition des Funktionszeigers passt und
   - deren Rückgabetyp zur Definition des Funktionszeigers passt und
   - deren Adresse auf der rechten Seite einer Zuweisung steht und
   - die entweder nicht virtuell ist oder eine Instanz der Basisklasse B existiert,

   so ist diese Methode ein mögliches Ziel.

2. Außerdem werden alle virtuellen Methoden untersucht, die in von B abgeleiteten Klassen definiert sind und Methoden der Basisklasse B redefinieren. Passen deren Parameter und Rückgabetyp zur Definition des Funktionszeigers, so sind auch sie ein mögliches Ziel. Diese Menge wird auf diejenigen Methoden eingeschränkt, die zu Klassen gehören, von denen mindestens eine Instanz vorhanden ist.

Damit können nur solche Methoden als Ziel erkannt werden, die in derjenigen Klasse vorhanden sind, für die der Funktionszeiger definiert ist. Die Bedingungen sind hier etwas komplexer, da ein Funktionszeiger wie oben erwähnt sowohl virtuelle als auch nicht virtuelle Routinen aufrufen kann. Bei Aufrufen nicht virtueller Methoden kann nicht angenommen werden, dass von den Klassen, zu denen die aufgerufenen Methoden gehören, explizit Instanzen erzeugt wurden. In diesem Fall können Methoden von Basisklassen aufgerufen werden, sofern diese in abgeleiteten Klassen nicht redefiniert wurden und von einer abgeleiteten Klasse Instanzen existieren.

Das Beispiel in Abbildung 5.6 (Seite 54) zeigt zwei verschiedene Funktionszeiger, `nvfp` und `vfp`. `nvfp` zeigt auf eine nicht virtuelle Methode, `vfp` auf eine virtuelle und in den abgeleiteten Klassen überschriebene Methode. Abbildung 5.7 auf Seite 55 zeigt den Ausschnitt für den mit (A) markierten Aufruf aus dem entsprechenden IML-Graphen.

Beide Funktionszeiger verweisen auf Routinen mit derselben Signatur. Aus diesem Grund sind die von der Analyse ermittelten Ziele beider Zeiger gleich. Als mögliche Ziele werden identifiziert:

- `Amphibian::virtMethod`: Die Methode ist virtuell und in `Amphibian` redefiniert. Von `Amphibian` existiert eine Instanz.
- `Car::virtMethod`: Die Methode ist virtuell und in `Car` redefiniert. Von `Car` existiert eine Instanz.
- `Vehicle::method`: Die Methode ist nicht virtuell, daher wird die Implementierung in `Vehicle` aufgerufen, auch wenn von `Vehicle` keine Instanz existiert. `Car` und `Amphibian` sind aber von `Vehicle` abgeleitet.

Dieses Ergebnis ist nicht falsch, nur konservativ. Für ein genaueres Ergebnis wäre eine separate Analyse notwendig, die jedoch nicht Teil der Aufgabenstellung ist und den zeitlichen Rahmen dieser Arbeit völlig gesprengt hätte.

```
class Vehicle
{
public: virtual int virtMethod ();
public:         int method ();
};

class Boat : public Vehicle
{
public: virtual int virtMethod ();
public:         int method ();
};

class Car : public Vehicle
{
public: virtual int virtMethod ();
public:         int method ();
};

class Amphibian : public Car, public Boat
{
public: virtual int virtMethod ();
public:         int method ();
};

int main (int argc, char** argv)
{
  ...
  Boat *pBoat = new Amphibian;
  Car *pCar = new Car;
  // virtual method call using function pointer
  int (Vehicle::*vfp)() = &Vehicle::virtMethod;
  x = (pBoat->*vfp)();                  // <-- (A)
  // non-virtual method call using function pointer
  int (Vehicle::*nvfp)() = &Vehicle::method;
  y = (pBoat->*nvfp)();
  ...
};
```

Abb. 5.6: Funktionszeiger und Methoden: Quellcode

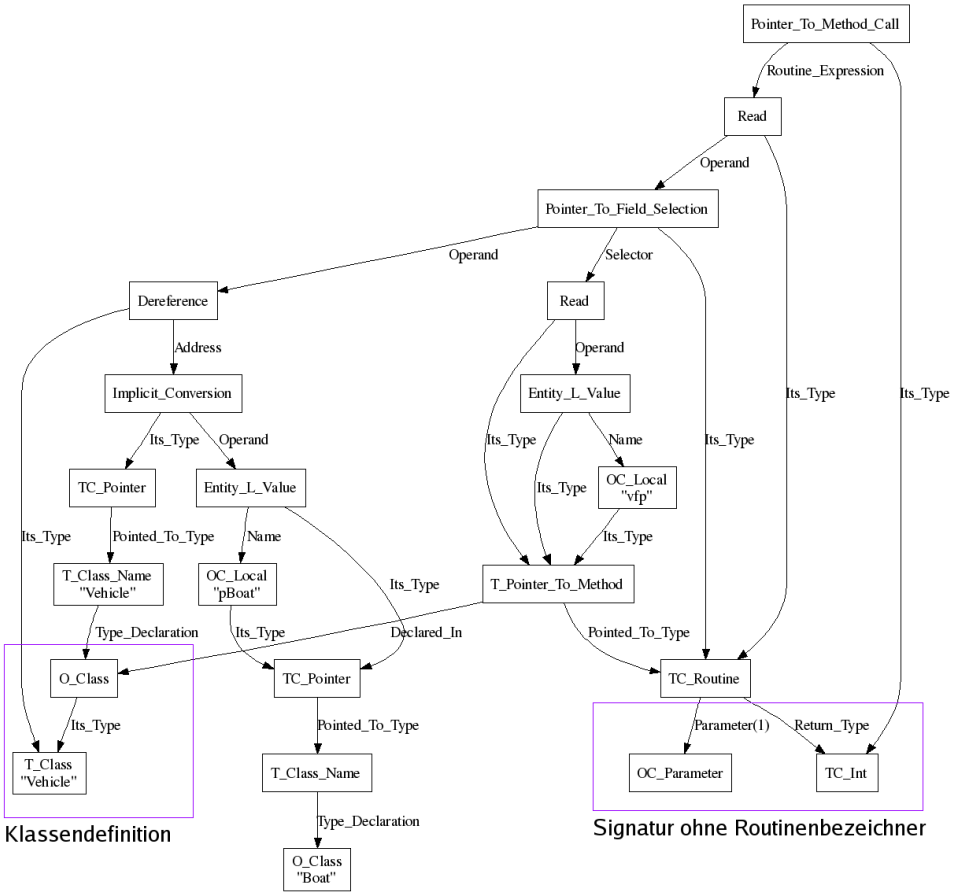

Abb. 5.7: Funktionszeiger und Methoden: IML-Darstellung

5.13 Die Konfigurationsdatei für benutzerdefinierte Deklarationen

5.13.1 Motivation

Liegt der Analyse zu einer Routinen kein Rumpf vor, so kann nicht ermittelt werden, ob die Routine terminiert oder welche Ausnahmen sie werfen kann. Dies ist bei Routinen der Fall, die als extern deklariert sind. Gerade bei der Verwendung von Bibliotheken ist dies der Fall. Beispielsweise fehlt beim Aufruf von `System.exit` in Java-Programmen der Analyse die Implementation der Routine. Um diese Informationslücke zu schließen, kann der Benutzer in einer optionalen Deklarationsdatei Angaben zu einzelnen Routinen machen. Vorgesehen sind zwei Aspekte:

- Angabe der Ausnahmetypen, die eine Routine propagieren kann.
- Angabe der Routinen, die ein Programm (oder einen Thread) beenden.

5.13.2 Aufbau und Inhalt

Grundsätzlich muss für beide Aspekte die Signatur der Routine angegeben werden. Für den Benutzer wäre es am einfachsten und vertrautesten, die Signatur in der Syntax der zu Grunde liegenden Programmiersprache anzugeben. Da die CFG-Generierung aber auf der IML beruht, ist diese Variante ein Bruch mit der Architektur. Außerdem würde für jede neue Programmiersprache, die von bauhaus unterstützt wird, ein neues Frontend für den CFG-Generator notwendig. Als Alternative zu diesem aufwändigen Design kam im Gespräch mit den Mitarbeitern der Abteilung die Idee auf, als Signatur den in der IML verwendeten *mangled name* zu benutzen. Dieser ist global eindeutig, dem bauhaus Anwender vertraut und vereinfacht den Parser für das Dateiformat erheblich. Ich habe mich für diese Variante entschieden, die eine einfache und dennoch vollständige Lösung des Problems darstellt. Die Information, ob eine Routine einen Thread beenden kann, ist in der aktuellen Form der IML noch belanglos: Threads werden momentan nicht modelliert. Die Konfigurationsdatei unterscheidet schon jetzt zwischen Thread und Prozess. Eine Routine kann einen Thread oder einen Prozess oder beides beenden. In der Datei sind neben den Deklarationen auch Kommentare erlaubt. Diese halten sich an die Ada-Notation und beginnen mit zwei Gedankenstrichen. Abbildung 5.8 auf Seite 57 zeigt ein Beispiel einer solchen Deklarationsdatei. Als Routinen sind sowohl Funktionen als auch Methoden erlaubt, wobei "Funktion" hier für eine Routine steht, die an keine Klasse gebunden ist, während sich "Methode" auf eine Routine bezieht, die zu einer Klasse gehört. Bei den Ausnahmetypen sind neben Klassentypen alle benutzerdefinierten und primitiven C++ Typen erlaubt, um das in C++ mögliche Spektrum an Ausnahmetypen nutzen zu können.

5.13.3 Externe Instanziierung von Objekten

Die oben beschriebene Datei lässt die Deklaration von Ausnahmen zu, die ausserhalb des verfügbaren Quellcodes geworfen werden. Gleichzeitig werden bei der Analyse aber die Ausnahmeklassen auf diejenigen Klassen beschränkt, von denen Instanzen erzeugt wurden. Da momentan die Deklaration von Instanziierungen in der Deklarationsdatei nicht möglich ist, werden alle Ausnahmen, die in dieser Datei deklariert werden, als instanziiert betrachtet.

```
--
-- declare exceptions thrown by external routines
--
__19DocumentManagerFRC19GetDocument
    throws
        (SecurityException,
         InvalidKeyException,
         LockExcetion,
         char*);
--
-- declare terminating routines
--
terminate__Fv terminates process;
```

Abb. 5.8: Beispiel einer Deklarationsdatei

5.13.4 Fehlerfälle und deren Behandlung

Aufgrund des Aufbaus der Datei kann davon ausgegangen werden, dass die Datei erst dann vom Benutzer geschrieben wird, nachdem der IML-Graph erzeugt wurde. Also kann vorausgesetzt werden, dass alle Bezeichner eindeutig einem IML-Knoten zugeordnet werden können müssen. Die Datei wird erst ausgewertet, nachdem der IML-Graph von `iml2cfg` gelesen wurde. Die semantischen Aktionen des Parsers versuchen den passenden IML-Knoten im Graphen zu finden. Ist dies nicht möglich, liegt ein Fehler vor und die Kontrollflussanalyse bricht nach dem Parsen der Datei mit entsprechenden Fehlermeldungen ab. Je Routine ist maximal ein Eintrag in der Datei erlaubt. Eine Mehrfachdeklaration ist ein semantischer Fehler, der zum Abbruch des Werkzeugs führt. Außerdem führen auch alle Syntaxfehler zum Abbruch.

5.13.5 Eingesetzte Werkzeuge

AFLEX und AYACC [Int02] werden als Scanner- bzw. Parsergenerator eingesetzt. Als gute Vorlage erwies sich das Teilprojekt `concepts`, das ebenfalls diese Werkzeuge benutzt. Auch die notwendige Erweiterung des `Makefile` wurde aus diesem Teilprojekt kopiert und angepasst. Der Parser erzeugt eine Liste von `Userdefined_Routine_Attributes`. Diese Struktur besteht aus einem Verweis auf das IML-Objekt, zu dem Attribute hinterlegt sind, und den Attributen selbst. Während der Kontrollflussanalyse kann über eine Hashmap effizient auf die Struktur zugegriffen werden.

5.14 Fehler im C++ Frontend

In diesem Abschnitt werden Fehler beschrieben, die während der Entwicklung in `cafe++` festgestellt wurden. Neben der Fehlerbeschreibung soll vor allem dargelegt werden, welche Hilfskonstruktionen eingesetzt wurden, um die Fehler zu umgehen.

5.14.1 Mehrfache Referenz auf `Entity_L_Value`

Während der Weiterentwicklung an `iml2cfg` trat unter gewissen Umständen ein reproduzierbarer Fehler auf, dessen Ursache erst nach zeitraubender Suche gefunden wurde. Das Programm stürzte beim Speichern des IML-Graphen entweder ab ("stack overflow (or erroneous memory access)") oder ging in eine Endlosschleife über. Der Fehler trat nur dann auf, wenn Grundblöcke optimiert wurden und statische Variablen mit einer neuen Objektinstanz initialisiert wurden. Allerdings

musste die Initialisierung mit der Deklaration erfolgen, andernfalls trat der Fehler nicht auf.

Ursache ist die Verwendung des IML-Knotentyps Entity_L_Value in `cafe++`. Zwar ist dieser von Value abgeleitet, wird jedoch wie eine Variable verwendet, d.h. es können mehrere von Value abgeleitete Knoten existieren, die denselben Entity_L_Value-Knoten referenzieren. Wird der Entity_L_Value-Knoten nun in eine Anweisungssequenz aufgenommen, geschieht das eventuell mehrfach, sodass alle außer der letzten Sequenzen verfälscht werden. Als Nebeneffekt können dadurch Zeiger so verändert werden, dass sie auf nicht mehr existente Objekte zeigen. Dies führt zum Absturz.

Um diesen Fehler zu beseitigen, werden die Entity_L_Value-Knoten nur dann in Anweisungssequenzen aufgenommen, falls sie noch nicht Teil einer solchen sind. Dies gilt auch für alle davon abgeleiteten IML-Knoten.

5.15 Ausnahmen im Überblick: `cfg2html`

Ein Punkt der Aufgabenstellung ist die Implementierung eines Werkzeugs, das die von einer Routine propagierten Ausnahmen anzeigt (vgl. Abschnitt 1.2 auf Seite 1). Ich habe mich gegen eine rein textuelle Ausgabe entschieden, wie sie beispielsweise `dumpiml` liefert. HTML scheint mir wesentlich besser geeignet, da der Benutzer mit Hilfe von Hyperlinks navigieren kann. Ein HTML-fähiger Browser kann heute auf jedem Rechner vorausgesetzt werden (`gravis`[4] setzt eine graphische Benutzeroberfläche voraus.). Das neue Werkzeug trägt den Namen `cfg2html`. Dieser ist zwar nicht ganz treffend, da nur der Fluss von Ausnahmen und der Aufrufgraph dargestellt werden, er passt aber sehr gut in die bisherige Nomenklatur der bauhaus-Werkzeuge. Das Werkzeug ist sehr einfach gebaut. Es basiert auf `iml2cfg` und `iml2html`, von denen sehr viele Teile wieder verwendet werden. Nun zur Ausgabe: Zu jeder Routine werden alle Aufrufer der Routine und die propagierten Ausnahmen aufgelistet. Außerdem werden ihre Quellen genannt (`throw` oder Routinenaufruf, jeweils mit Angabe der Quellcodeposition). Falls eine Ausnahme einen abstrakten Typ besitzt, wird dies ebenfalls angezeigt (vgl. Abschnitt 5.12.2, S. 50) Die Eintrittsroutine eines Programms ist besonders gekennzeichnet. Wenn sie Ausnahmen propagiert, wird eine entsprechende Meldung angezeigt. Abbildung 5.9 (S. 59) zeigt ein Beispiel der Ausgabe.

5.16 Graphische CFG-Darstellung mit `cfg2dot`

Im Rahmen der Ausarbeitung zu dieser Diplomarbeit habe ich das Werkzeug `cfg2dot` zum größten Teil[5] an die Erweiterungen angepasst. Es liest eine IML-Datei ein und generiert eine Ausgabedatei im `.dot`-Format, die den CFG beschreibt. Von `http://www.graphviz.org/` kann das Werkzeug `dot` heruntergeladen werden, das eine solche `.dot`-Datei einliest und einen Graphen erstellt. Es werden verschiedene Ausgabeformate unterstützt (PS, PNG, etc.). Alle neuen Kanten und Knoten des CFG werden in den Kontrollflussgraphen, die das Werkzeug erzeugt, angezeigt. Ebenfalls neu ist der Kommandozeilenparameter `-sloc`, mit dem neben den generischen Namen der Grundblöcke auch die Zeilennummer der ersten und der letzten Anweisung in den Knoten angezeigt wird. Außerdem wurden die generischen Namen um den Typ des Grundblocks erweitert, falls es sich nicht um einen regulären Grundblock handelt. Ausnahmekanten werden mit dem Typ der Ausnahme attributiert. Ein Beispiel zeigt Abbildung 7.5 auf Seite 75. Abbildungen komplexerer Kontrollflussgraphen in dieser Ausarbeitung wurden mit diesem Werkzeug erstellt.

[4] `gravis` ist ein GRAph VISualisierungswerkzeug in bauhaus

[5] Bei der Ausgabe der interprozeduralen KA treten noch Fehler auf.

Other Node: Routine ("func_B__Fb") --> 002, ID=1305

```
Callers: : ...
    1. Routine ("func_A__Fb") . . . . . . cyclic.cxx ( 90 : 6 )
    2. Routine ("func_C__Fb") . . . . . . cyclic.cxx ( 112 : 6 )
Propagated exception types:
    pointer to class "Exception";
      --> explicitely thrown at cyclic.cxx ( 107 : 7 )
    pointer to class "Test_Exception";
      --> propagated by routine call to Routine ("func_C__Fb") . . . . . . cyclic.cxx ( 112 : 6 )
    pointer to int ;
      --> propagated by routine call to Routine ("func_C__Fb") . . . . . . cyclic.cxx ( 112 : 6 )
    pointer to class "Numeric_Exception";
      --> propagated by routine call to Routine ("func_C__Fb") . . . . . . cyclic.cxx ( 112 : 6 )
 Root
```

Other Node: Routine ("func_C__Fb") --> 003, ID=1276

```
Callers: : ...
    1. Routine ("func_B__Fb") . . . . . . cyclic.cxx ( 103 : 6 )
    2. Routine ("func_D__Fb") . . . . . . cyclic.cxx ( 127 : 6 )
Propagated exception types:
    pointer to class "Test_Exception";
      --> explicitely thrown at cyclic.cxx ( 123 : 7 )
    pointer to int ;
      --> explicitely thrown at cyclic.cxx ( 119 : 10 )
    pointer to class "Numeric_Exception";
      --> propagated by routine call to Routine ("func_A__Fb") . . . . . . cyclic.cxx ( 90 : 6 )
    pointer to class "Exception";
      --> propagated by routine call to Routine ("func_B__Fb") . . . . . . cyclic.cxx ( 103 : 6 )
 Root
```

Abb. 5.9: Ausgabe von `cfg2html` (Auszug)

Kapitel 6

Entwurf

In diesem Kapitel wird die Architektur von `iml2cfg` dargestellt. Es sind einige Pakete neu hinzugekommen und bisherige Pakete teilweise stark erweitert worden. Bisher war der Visitor-Datensatz *die* zentrale Struktur. Inzwischen werden jedoch viele Informationen während der Traversierung gesammelt und in internen Strukturen abgelegt, um später darauf zugreifen zu können. Der Ablauf wurde in mehrere Phasen aufgeteilt (s. Abschnitt 5.1, S. 39).

Die folgenden Abschnitte geben einen Überblick über den Entwurf und die Aufgaben der einzelnen Pakete. Es folgt eine Beschreibung aller wichtigen Aspekte des Entwurfs.

6.1 Programmiersprache

Da `iml2cfg` in seiner bisherigen Version in Ada implementiert und Ada die Muttersprache von bauhaus ist, bestand kein Grund die Programmiersprache zu wechseln. Viele C-typische Werkzeuge, wie z.B. Lex und Yacc sind ebenfalls für Ada verfügbar, sodass durch Ada keine Nachteile zu erwarten waren.

6.2 Attributierte Kanten

Gerichtete Kanten werden im IML-Graphen nicht durch Kanten-Objekte dargestellt, sondern durch Listen, die Zeiger auf die Zielknoten enthalten. Um die Kanten mit zusätzlichen Informationen versehen zu können, wird parallel zu der Zeigerliste eine zweite Liste geführt, welche die Attribute für die Kanten enthält. Die Assoziation wird über die Position in der Liste hergestellt.

6.3 Systemarchitektur

Abbildung 6.1 (S. 61) zeigt die Pakete von `iml2cfg`. In den folgenden Abschnitten wird kurz darauf eingegangen, welches Paket welche Funktionalität bietet.

6.3.1 `Visitors.CFG_Gen`: IML-Traversierung

Dieses Paket beinhaltet alle Visitors für die IML-Knoten. Im Vergleich zur alten Version kamen einige Visitor neu hinzu:

- `Visit_Indirect_Call`: Aufruf einer *Methode* mittels eines Funktionszeigers. Per Definition steht die Objektinstanz des Aufrufs fest.

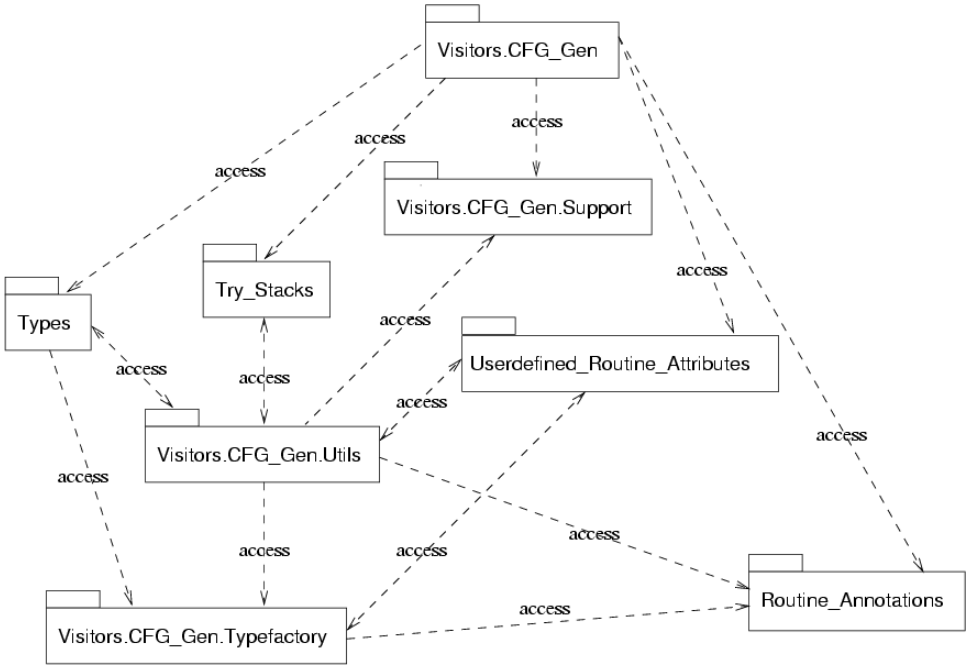

Abb. 6.1: Module von `iml2cfg`

- `Visit_Initialize`: Initialisierung von neu angelegten Objekten.
- `Visit_Member_Selection`: Auswahl eines Members aus einem Compound.
- `Visit_Multiple_Callees`: Aufruf mit mehreren möglichen Zielen.
- `Visit_New_Operator`: Operator in C++ und Java zur Speicherallokation und implizitem Konstruktoraufruf.
- `Visit_Pointer_To_Member_Literal`: Zeiger auf ein Literal, das ein Member eines Compounds bezeichnet.
- `Visit_Pointer_To_Member_Selection`: Zeiger aus einer Member-Selektion.
- `Visit_Pointer_To_Method_Call`: Zeiger auf einen Methodenaufruf.
- `Visit_Static_Member_Selection`: Statische Auswahl eines Members.
- `Visit_Throw_Statement`: Werfen einer Ausnahme.
- `Visit_Try_Catch_Finally_Statement`: `try`-Anweisung.
- `Visit_Virtual_Call`: Aufruf einer virtuellen Methode, verlangt dynamische Bindung.

Einige der bestehenden Visitors wurden angepasst. Diese Änderungen sind jedoch meist marginal. Teilweise führte die ursprüngliche Implementation jedoch zu Abstürzen.

6.3.2 `Utils`: Hilfsfunktionen

Das `Utils`-Paket bündelt sämtliche Hilfsfunktionen, die während Traversierung und Analyse notwendig sind. Dazu zählen Routinen zum Registrieren bestimmter IML-Knoten (Anweisungen wie `goto` oder Einheiten wie Routinen) und Routinen zur Instanziierung von Iteratoren über die registrierten Knoten. Auch andere Routinen, die auf den registrierten Knoten arbeiten, sind im Paket enthalten. Auch die Liste der Klassen, von denen Objekte instanziiert wurden, ist im `utils`-Paket untergebracht.

Ebenfalls in diesem Paket enthalten sind die Funktionen zur Berechnung der definitiv oder potentiell terminierenden Routinen, die Erweiterung des CFG zur Berechnung der interprozeduralen Kontrollabhängigkeit (*CFG augmentation*), die Funktionen zur Berechnung der interprozeduralen Kontrollabhängigkeit und zur Ermittlung geworfener bzw. gefangener Ausnahmetypen.

6.3.3 `Types`: Allgemeine Typfunktionen

In diesem Paket werden allgemeine Typfunktionen zusammengefasst. Da diese Funktionen auch in anderen Projekten Verwendung finden können, sind sie nicht Teil von `Visitors.CFG_Gen`. Es wäre vorstellbar, dieses Paket in Zukunft in `Class_Relations` umzubennen, da sein Fokus auf Klassentypen liegt. Die wichtigsten Funktionen möchte ich im Folgenden kurz erläutern.

6.3.3.1 Funktion `Is_Subclass`

Diese Funktion prüft, ob eine gegebene Klasse (genauer gesagt ein T_Compound_Type) von einer anderen gegebenen Klasse abgeleitet ist. Dabei gilt: Eine Klasse ist gleichzeitig Super- und Subklasse von sich selbst.

6.3.3.2 Funktion `Is_Subtype`

Dies ist eine Generalisierung von `Is_Subclass`. Dabei werden folgende Fälle unterschieden:

1. Wird diese Funktion auf Klassentypen angewandt, ist sie äquivalent zu `Is_Subclass`.

2. Bei Zeigertypen, die in der IML als `TC_Pointer`-Knoten dargestellt werden, kann der Vergleich mit oder ohne Einbeziehung von Qualifizierern (z.B. `const` bzw. T_Type_Qualifier) einfach durchgeführt werden. Zwei Zeigertypen sind dann identisch,

   - wenn sie vom selben IML-Knoten dargestellt werden (Identität) oder
   - wenn die beiden Typen, auf die sie zeigen, identisch sind. Dieser Sonderfall tritt bei benutzerdefinierten Typen auf, da die IML teilweise Knoten vom Typ T_Compound_Type und teilweise Knoten vom Typ TC_User_Type_Name verwendet. Knoten des Typs TC_User_Type_Name besitzen ein Attribut Type_Declaration, das auf einen O_Node-Knoten verweist. Dessen Its_Type-Attribute verweist auf einen T_Compound_Type-Knoten. Durch die Verwendung beider Knotentypen existieren zwangsläufig verschiedene Zeigerknoten, die letztendlich vom selben Typ sind!

   Der Vergleich von Typen arbeitet iterativ. Bei jedem Schritt wird festgelegt, ob die aus der aktuellen Iteration resultierenden Typknoten verglichen werden müssen oder nicht.

Ist mindestens einer der zu vergleichenden Typen ein Typqualifizierer und sollen Typqualifizierer ignoriert werden[1] (über einen Parameter wählbar), so wird der unqualifizierte Typ zum zu vergleichenden Typ, am Ende der Iteration wird kein Typvergleich durchgeführt. Sollen Typqualifizierer in den Vergleich mit einbezogen werden, so müssen beide Typen ein Typqualifizierer vom selben Typ sein. Ist dies nicht der Fall, sind die Typen nicht gleich. Ansonsten wird mit den unqualifizierten Typen fortgefahren.

Ist mindestens einer der zu vergleichenden Typen ein Zeigertyp und sollen Zeiger ignoriert werden (über einen Parameter wählbar), so wird mit dem Typ, auf den der Zeiger verweist, fortgefahren. Am Ende der Iteration wird kein Typvergleich durchgeführt. Sollen Zeiger in den Vergleich mit einbezogen werden, so müssen beide Typen ein Zeiger sein. Ist dies nicht der Fall, sind die Typen nicht gleich. Ansonsten wird mit den Typen, auf die die Zeiger verweisen, fortgefahren.

Ist mindestens einer der zu vergleichenden Typen aus der Klasse TC_User_Type_Name, so wird mit dem T_Compound_Type-Typ fortgefahren, auf dem der Typ beruht. Es wird kein Typvergleich am Ende der Iteration durchgeführt.

In allen anderen Fällen wird ein Typvergleich durchgeführt:

- Sind beide Typknoten identisch (derselbe IML-Knoten), liegt eine Subtypbeziehung vor (da ein Typ T hier per Definition immer Subtyp von T ist).
- Sind beide Typen aus der Klasse T_Compound_Type, wird der Vergleich an `Is_Subclass` weitergegeben.
- Sind beide Typen Arrays, werden die beiden Arrays verglichen.
- Ansonsten liegt keine Subtypbeziehung vor.

Abbildung 6.2 auf Seite 64 zeigt die Darstellung von verschiedenen C++-Zeigertypen in IML. Es fällt auf, dass der `const`-Qualifizierer direkt am Klassenknoten hängt, sodass es unterschiedliche Zeigerknoten für den konstanten und den nicht konstanten Klassentyp gibt. In diesem Fall werden die Pfade von den Ausgangsknoten (die zu vergleichenden Typen) zum T_Compound_Type-Knoten knotenweise verglichen. Sind beide Knoten vom Typ TC_Pointer, wird mit dem nächsten Knoten auf dem Pfad fortgefahren. Ist ein Knotentyp in der Klasse T_Type_Qualifier, wird er ignoriert und der nächste Knoten auf dem Pfad betrachtet. Sind beide Knoten vom Typ T_Compound_Type, endet der Vergleich mit dem Ergebnis von `Is_Subclass` auf diese beiden Knoten. Bei anderen Typknoten der Klasse T_Node wird auf die Identität beider Knoten geprüft. Alle anderen Fälle beenden den Vergleich mit dem Ergebnis "kein Subtyp".

Anmerkung i: Das Beispiel in Abbildung 6.2 zeigt auch die Verwendung des `typedef`-Konstrukts. Für diese Funktion ist es ohne Bedeutung, da es nicht von T_Node, sondern von O_Node abgeleitet ist.

Anmerkung ii: In kommenden Versionen der IML kann es notwendig werden, diesen Algorithmus entsprechend den Neuerungen anzupassen!

3. Wird diese Funktion auf primitive Typen angewandt[2], so wird auf Gleichheit geprüft, da C++ und Java im Gegensatz zu Ada das Konzept der Subtypen nicht kennen. Zwei solche Typen sind gleich, falls beide durch denselben Typknoten im IML-Graphen repräsentiert werden. In diesem Fall ist der Typ per Definition Subtyp von sich selbst.

[1] Bei der Bestimmung des passenden Exception Handlers ist dies der Fall.
[2] Liste der primitive Typen in IML siehe Glossar.

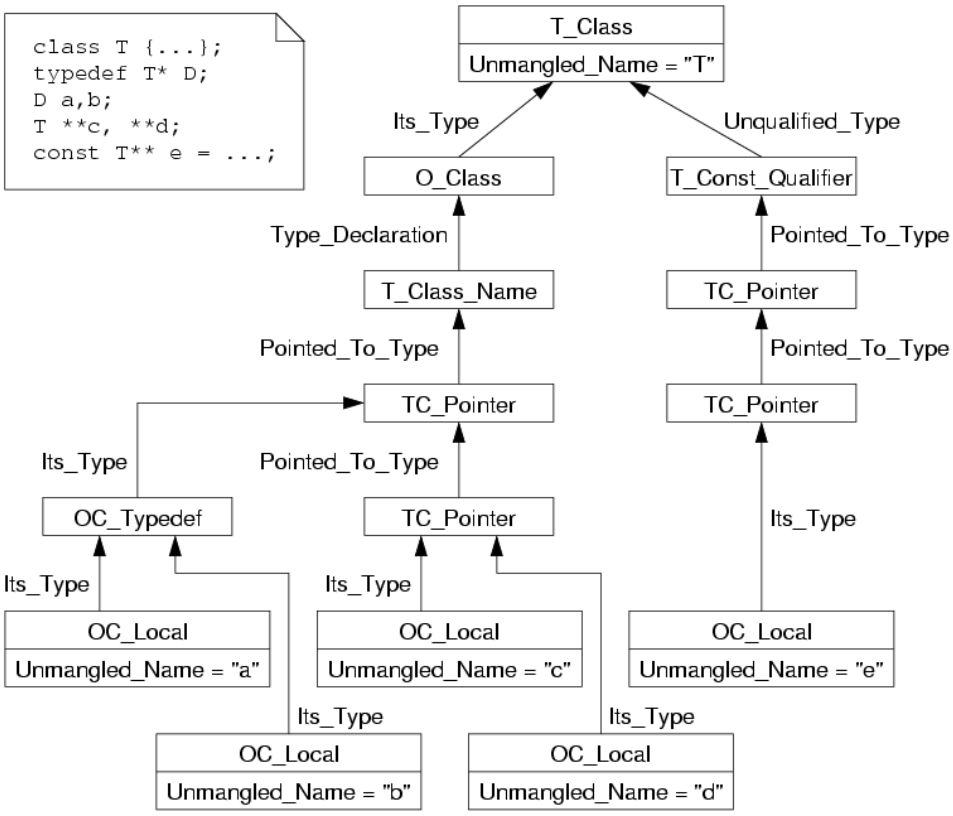

Abb. 6.2: IML-Darstellung verschiedener Zeigertypen.

4. Beim Vergleich von Arrays werden nur die zugrunde liegenden Typen miteinander verglichen, die Grenzen des Arrays werden nicht betrachtet.

6.3.3.3 Funktion `Get_Subclasses`

Als Parameter wird der Funktion eine Liste mit Compoundtypen übergeben. Diese Liste wird um alle davon abgeleiteten Klassen erweitert. Eine leere Menge bleibt leer.

6.3.3.4 Funktion `Get_Implementors`

Diese Funktion liefert eine Liste aller Klassentypen, die eine gegebene Schnittstelle implementieren.

6.3.3.5 Funktion `Determine_Possible_Callees`

In dieser Funktion werden die Ziele sämtlicher Aufrufe ermittelt. Die verschiedenen Konzepte sind den Abschnitten 5.12.1, S. 49 und 5.12.3, S. 52 zu entnehmen.

Zwei wichtige Eigenschaften dieser Funktion seien hier genannt:

1. Die Funktion liefert nur dann korrekte Ergebnisse, wenn das gesamte System des IML-Graphen traversiert wurde. Nur dann sind alle Objektinstanziierungen und alle Funktionszeigerzuweisungen bekannt.

2. Nachdem für einen Aufruf die möglichen Ziele ermittelt wurden, werden sie im Callee_Set des Knotens abgelegt. Jeder Aufruf dieser Funktion prüft zunächst,

ob diese Menge leer ist. Nur dann werden die Ziele bestimmt und die Menge verändert.

6.3.4 `Typefactory`: Typknoten finden und erstellen

Dieses Paket wird benutzt, um Zeigertypen und primitive Typen zu finden oder zu erstellen, falls sie noch nicht existieren. Es mag auf den ersten Blick seltsam erscheinen, bei einer Analyse neue Typen zu erstellen. Es existieren jedoch zwei Fälle, in denen dies notwendig ist:

1. Der Benutzer definiert in der externen Deklarationsdatei Ausnahmen mit primitivem Typ. Während der Analyse müssen diese Ausnahmen in den IML-Graph eingebracht werden. Da primitive Typen jeweils einen eigenen Knotentyp besitzen, wäre die einfachste Möglichkeit gewesen, je einen Knoten des entsprechenden Typs zu instanziieren. Damit wäre die Identität zwischen Typ und Knoten verletzt worden. Also darf genau dann ein neuer Knoten instanziiert werden, wenn noch kein Knoten diesen Typs existiert.

2. Bei der Auflösung der möglichen Ausnahmetypen kann es notwendig sein, alle Subtypen eines Typs ebenfalls zu berücksichtigen. In diesen Fällen handelt es sich immer um Zeigertypen. Sofern ein Zeigertyp noch nicht im IML-Graph vorhanden ist, muss er angelegt werden. Es gilt wie o.g. die Identität zwischen Zeigertyp und IML-Typknoten.

Primitive Typen der Sprache C/C++ werden mit ihrem Bezeichner eindeutig identifiziert ("int", "bool", etc.). Für Java besteht keine Unterstützung. Zeigertypen werden durch den Typ, auf den sie zeigen, eindeutig bestimmt.

6.3.5 `Try_Stacks`: Interne Darstellung der `try`-Anweisungen

Ein essentieller Bestandteil der Analyse ist der Aufbau einer Struktur, über die zu jedem Grundblock die umgebenden `catch`- und `finally`-Blöcke effizient ermittelt werden können. Außerdem werden hier die automatischen Objektinstanziierungen festgehalten, sodass die notwendigen Informationen zum Stack Unwindung nach einem beliebigen Grundblock ebenfalls verfügbar sind. Der möglichen Schachtelung von `try`-Anweisungen wird mit einer Stapelstruktur Rechnung getragen. Diese Struktur wird während der Traversierung aufgebaut.

6.3.6 `Support`: Grundblockunterstützung

Das `Support`-Paket beinhaltet alle Funktionen zur Erstellung, Traversierung und Vernichtung von Grundblöcken. Die Funktionen können in die folgenden drei Gruppen gegliedert wurden:

1. Verwaltung normaler Grundblöcke: Normale Grundblöcke müssen mit der Funktion `Make_BB` erstellt werden. Sie sorgt für die korrekte Initialisierung und die Aktualisierung interner Strukturen. Wird ein Grundblock entfernt, muss die Prozedur `Notify_BB_Death` aufgerufen werden, die alle Observer über den Verlust benachrichtigt (vgl. Abschnit 5.3).

2. Verwaltung temporärer Grundblöcke: Diese müssen mit der Funktion `Make_Temporary_BB` erstellt werden. Sie initialisiert sie und aktualisiert die interne Liste der temporären Grundblöcke. Außerdem werden verschiedene Funktionen zum Zugriff auf die temporären Grundblöcke bereitgestellt.

3. Zuordnung der Grundblöcke zu `finally`-Blocks: Wird innerhalb eines `finally`-Blocks ein neuer Grundblock angelegt, so muss dieser den Eingangsgrundblock des `finally`-Blocks kennen. Zu diesem Zweck sind verschiedene Funktionen im Paket enthalten.

6.3.7 `Userdefined_Routine_Attributes`: Deklarationen

Dieses Paket kapselt die Verwaltung der Deklarationen, die vom Benutzer stammen (vgl. Abschnitt 5.13, S. 56).

6.3.8 `Exception_Source_Lists`: Ausnahmequellen

Während der Traversierung des IML-Graphen können Anweisungen auftreten, die Ausnahmen verursachen. Da diese erst dann definitiv einem Handler zugeordnet werden können, nachdem das gesamte System traversiert worden ist, müssen die Quellen der Ausnahmen gespeichert werden. Die dazu notwendigen Funktionen sind in diesem Paket zusammengefasst. Neben dem Registrieren neuer Ausnahmen erlaubt das Paket auch das Entfernen von Ausnahmen (nachdem sie an einen Handler gebunden wurden), die Suche nach Ausnahmen eines bestimmten Typs, die Iteration über alle Ausnahmen einer Quelle und die Iteration über alle bekannten Quellen.

6.3.9 `Debug`: Unterstützung für den Softwareingenieur

In diesem Paket sind alle Routinen zusammengefasst, die interne Darstellungen textuell ausgeben oder textuelle Bezeichner zu Objekten erstellen, wie sie beim Testen verwendet werden. Die wichtigste Routine ist `Name`. Diese erwartet genau ein Argument. Je nach dessen Typ wird ein Bezeichner generiert. Bei zukünftigen Erweiterungen sollte diese Routine mit evtl. neu hinzugekommenen Typen überladen werden.

Kapitel 7

Test

7.1 Einschränkungen und Hilfskonstruktionen

bauhaus stellt für Java das Frontend `jafe`, für C++ das Frontend `cafe++` bereit. Allerdings verwendet nur `cafe++` die aktuelle Version der IML, sodass ein Test mit beiden Frontends zwei unterschiedliche Quellcode-Revisions voraussetzt. Erfahrungsgemäß stellt es einen recht großen Aufwand dar, bauhaus komplett neu aus dem CVS zu laden und umzuwandeln. Selbst wenn eine lauffähige Version von `jafe` verfügbar ist, stellt dies keine Garantie dar, dass andere Werkzeuge wie z.B. `imllink`, `iml2html`, `dumpiml`, usw. ebenfalls fehlerfrei arbeiten. Anpassungen an den in dieser Diplomarbeit entwickelten Programmen wären ebenfalls wahrscheinlich. Somit ist mit vertretbarem Aufwand ein umfangreicher Test von Java-Quellcode nicht machbar. Im Besonderen betrifft dies folgende Sprachkonstrukte:

- Das `finally`-Schlüsselwort existiert nicht in C++. Die Behandlung von `finally`-Blöcken ist jedoch recht anspruchsvoll und sollte deshalb getestet werden.
- Für explizite Deklarationen von Schnittstellen existiert in C++ kein `interface`-Sprachkonstrukt wie in Java. In der Praxis wird die Semantik durch `struct`-Deklarationen nachgeahmt. In IML sind Strukturen, Schnittstellen und Klassen durch unterschiedliche Knotentypen dargestellt. Alle diese Knotentypen sollten bei der Bestimmung von Superklassen, abgeleiteten Klassen oder potentiellen Kandidaten von Methodenaufrufen im Test beachtet werden.

Um diese Unsicherheiten und den Aufwand zu umgehen, habe ich einen Workaround eingebaut: `iml2cfg` prüft bei allen `catch`-Handlern den Typnamen der gefangenen Ausnahme. Stimmt dieser Typname mit einem Platzhalternamen überein, so wird der `catch`-Block von der Kontrollflussanalyse als `finally`-Block angesehen. Auf diese Weise kann auch mit C++-Quellcode die `finally`-Anweisung getestet werden. Der Typname wird als optionaler Parameter `finally_placeholder` in der Kommandozeile von `iml2cfg` definiert. Eine Besonderheit ist an dieser Stelle unbedingt zu beachten: Typen, die in C/C++ mittels der `typedef`-Direktive definiert wurden, werden von `cafe++` auf ihren ursprünglichen Typ zurückgeführt. Der Typname, wie er in der `typedef`-Direktive steht, ist in einigen Elementen der IML nicht mehr zu bestimmen, so auch in den `catch`-Handlern! Es empfiehlt sich, eine leere Struktur zu definieren und deren Typnamen als Platzhalter zu verwenden.

Ein weiterer Sprachbestandteil, der nur in Java vorhanden ist, sind Interfaces. In C++ ist die Verwendung von Strukturen mit virtuellen Funktionen ein pragmatischer Ersatz für Interfaces. In IML werden aber Interfaces und Strukturen mit unterschiedlichen Knotentypen repräsentiert. Für C++-Quellcode existiert somit

keine Möglichkeit, mit einfachen Mitteln Interfaces im IML-Graph zu erzeugen, ohne den Graph explizit zu manipulieren. Aus diesem Grund sind alle Funktionen, die auf Interfaces operieren, *nicht getestet*!

7.2 Wegwerftests

Während der Entwicklung der Software wurden permanent Wegwerftests durchgeführt. Diese Tests werden hier nicht dokumentiert. Typischerweise handelte es sich um den gezielten Test von Änderungen oder Erweiterungen am Quellcode. Auch während der Konzeption wurden verschiedene Wegwerftests durchgeführt. Besonders im Hinblick auf den Kontrollfluss bei `finally`s wurden etliche Tests mit Java durchgeführt.

7.3 Automatisiertes Testen

Vor der Spezifikation der Testfälle war die Frage zu klären, wie ein Programm wie `iml2cfg` getestet werden kann. Zur Auswahl standen verschiedene, schon bekannte Alternativen. Als Beispiele möchte ich zwei Extreme nennen: Entwicklung eines kompletten Testtreibers als eigenständige Anwendung einerseits und andererseits ein einfaches Skript, welches Soll- und Istausgaben miteinander textuell vergleicht (→`diff`). Der Testtreiber stellt die mächtigste Alternative dar, ist aber auch mit hohem Aufwand verbunden. Es ist unwahrscheinlich, in der gegebenen Zeit einen brauchbaren Testtreiber zu erstellen. Ein Skript ist einfach zu entwickeln, jedoch sind die Möglichkeiten stark eingeschränkt (außerdem tendieren Skripte mit der Zeit zur Unleserlichkeit). Ein Zugriff auf den IML-Graphen ist nur über Umwege möglich. Es bleiben also die Formate übrig, die den Graphen in textueller Form wiedergeben, wie die Ausgaben von `dumpiml` oder `iml2html`. Bei diesen Formaten besitzt jeder IML-Knoten eine eindeutige numerische ID. Diese ist abhängig von der Traversierungsreihenfolge des Graphen. Sie kann in den Testfällen also nicht als bekannt vorausgesetzt werden. Es wäre vorstellbar, mit `grep`, `sed` und anderen Werkzeugen einen IML-Graphen in Textform so zu analysieren, dass eine Verbindung zwischen den IDs und den (statischen) Bezeichnern aus dem Quelltext hergestellt wird. Wäre diese Zuordnung nicht notwendig, würde sich der Test stark vereinfachen.

Während der Entwicklung habe ich von den `-verbose` und `-veryverbose` Optionen regen Gebrauch gemacht, um die Resultate von Wegwerftests zu analysieren. Eine einfache Lösung scheint mir daher zu sein, die Ausgaben im `veryverbose`-Modus für den Test zu gebrauchen.

7.3.1 Darstellung der Objektidentität

Ein grundlegendes Problem beim Test ist die Darstellung der Identität der Objekte. Die einfachste Lösung ist es, ihre Speicheradresse zu verwenden. Analog zu den o.g. IML-IDs ist sie aber nicht vorhersagbar und damit für die automatische Testauswertung nicht geeignet. Für alle Objekte, die einen eindeutig benannten Teil des Quellcodes repräsentieren, wie z.B. Routinen, Typen oder Variablen, kann als eindeutiger Bezeichner der `mangled_name`[1] aus der IML verwendet werden. Falls kein IML-Bezeichner zur Verfügung steht, kann die Position bzw. der Bereich innerhalb des Quellcodes, auf den sich ein Objekt bezieht, als eindeutiges Merkmal verwendet werden. Zwar kann auf diese Weise nicht allen Objekten ein Bezeichner zugeordnet werden, aber für einen recht großen Teil trägt diese Lösung.

[1] Es wird angenommen, dass dasselbe Frontend bei jedem Lauf die selben Bezeichner erzeugt.

Ein zentraler Bestandteil von `iml2cfg` sind die Grundblöcke. Für sie war in der vorliegenden Implementierung in der Funktion `Make_BB` ein Argument `Name` vorgesehen, das jedoch nicht weiter verwendet wurde. Der Name gibt Auskunft über die Verwendung des Grundblocks, z.B. `IF-JOIN`, `LOCAL-SUPER-EXIT` oder `UNREACHABLE`. In der aktuellen Version von `iml2cfg` wird eine Hashtabelle benutzt, um eine Assoziation zwischen einem Grundblock und seinem Namen herzustellen. Diese Lösung hat zwei Vorteile: Erstens bleiben die `Basic_Block`-Klassen unverändert. Zweitens kann der dadurch entstehende Mehraufwand optional ein- oder ausgeschaltet werden. Momentan steht die Funktionalität nur im `veryverbose`-Modus zur Verfügung. Ein Nachteil dieser Lösung ist die Mehrdeutigkeit des Namens. Zwar bringt die Bezeichnung im Sinne der eindeutigen Benennung bzw. des automatischen Tests keinerlei Vorteile. Für den Benutzer ist sie aber sehr hilfreich, um das Protokoll besser zu verstehen.

Der Bezeichner eines Grundblocks besteht beim Test aus seinem Namen (evtl. leer) und dem Bereich von der ersten bis zur letzten Anweisung, die zu ihm gehört. Beispielsweise bezeichnet "`STARTUP;SomeRoutine__Fv;;`" den Startup-Grundblock der Routine mit dem IML-Bezeichner SomeRoutine__Fv, zu dem keine Anweisungen gehören. "`THEN;OtherRoutine__Fv;main.cxx; 168; 192`" bezeichnet einen Grundblock der Routine mit dem IML-Bezeichner OtherRoutine__Fv, zu dem die Anweisungen von Zeile 168 bis 192 (einschl.) gehören. Im Paket `Debug` sind sämtliche Funktionen zusammengefasst, die die Bezeichner verschiedenster Objekte liefern. Es unterstützt sowohl die Speicheradressen als auch Testnamen als Bezeichner. Wird der neue Kommandozeilenparameter `-trace_pointers` beim Aufruf von `iml2cfg` angegeben, werden Speicheradressen als Bezeichner verwendet.

7.3.2 Klassifizierung der Meldungen

Die Meldungen im `-veryverbose`-Modus sind sehr detailliert. Ihre Anzahl ist entsprechend groß. Ein direkter Vergleich der Meldungen eines Tests mit den erwarteten Meldungen würde bei vielen Änderungen am Quellcode dazu führen, dass Tests mit Fehlern enden. Eine Einteilung der Meldungen in die Kategorien *notwendig*, *belanglos* und *unerlaubt* vereinfacht den Test. Belanglose Meldungen werden bei der Auswertung des Testprotokolls ignoriert. Ein Test ist genau dann erfolglos, wenn im Protokoll

- keine unerlaubten Meldungen aufgetreten sind und
- alle notwendigen Meldungen vorhanden sind.

Für jeden Test werden die Soll-Ergebnisse zeilenweise spezifiziert. Jede Zeile beginnt mit einem Präfixzeichen, das die Kategorie der Zeile bestimmt. Ein Pluszeichen leitet notwendige Zeilen ein, ein Minuszeichen leitet unerlaubte Zeilen ein. Belanglose Zeilen beginnen mit einem Gatterzeichen. Sie können als Kommentarzeilen benutzt werden. Als Bonbon kann die Datei auch den Namen des Tests beinhalten. Dazu muss die Zeile mit einem Dollarzeichen beginnen. Ob eine Meldung im Protokoll vorhanden ist, wird mit `grep` ermittelt. Dieses Werkzeug benutzt reguläre Ausdrücke. Diese können in der Spezifikation ebenfalls verwendet werden, wie Abbildung 7.1 zeigt. Ein Nachteil an dieser Stelle ist, dass sämtlichen Zeichen, die eine besondere Bedeutung in regulären Ausdrücken besitzen, ein Backslash vorangestellt werden muss, damit sie als normales Zeichen behandelt werden. Die erste Zeile der Spezifikationsdatei muss `#type=xlg` sein (xlg steht für *expected log*). Daran erkennt das Testskript, dass es sich um eine Spezifikationsdatei handelt.

```
#type=xlg
#
$Beispiel
#-------------------------
# a) notwendige Meldungen
#-------------------------
+Diese Meldung muss vorhanden sein
+Auch [a-zA-Z]* muss existieren
#-------------------------
# b) unerlaubte Meldungen
#-------------------------
-Fehler [0-9]*
-Warnung W[0-9][0-9][0-9]
```

Abb. 7.1: Beispiel einer Spezifikation der Testmeldungen

7.3.3 Probleme durch nichtdeterministisches Verhalten

Es wurde bisher mehrmals beschrieben, dass Objekte in Hashmaps verwaltet werden, um einen effizienten Zugriff auf sie zu erhalten. Diese Hashmaps haben beim Test eine negative Auswirkung: Einige Operationen verhalten sich nichtdeterministisch. Der Grund liegt in der nicht vorhersagbaren Speicheradresse dynamisch erzeugter Objekte, die von Hashmaps als Schlüssel verwendet wird. Eine Iteration über alle Elemente einer Hashmap liefert also eine nichtdeterministische Reihenfolge. Bei der Optimierung der Grundblöcke führt dies dazu, dass nicht vorhergesagt werden kann, in welcher Reihenfolge Grundblöcke zusammengefasst werden. Deshalb fehlen bei den erwarteten Protokollen einige Meldungen, die sich auf die Grundblockverschmelzung beziehen.

7.3.4 Wartung der Software

Bei der Wartung der Testskripte ist unbedingt auf die Verwendung der richtigen Hochkommas zu achten. Sie stellen eine große Fehlerquelle dar. Einen guten Überblick über bash-Skripte mit genügend Tiefe bietet [bas02].

Die für die Tests verwendeten Meldungen sind im Quellcode mit einem entsprechenden Kommentar gekennzeichnet. Damit soll verhindert werden, dass durch eine Veränderung des Meldungstextes die Testergebnisse verfälscht werden.

7.4 Organisation der Tests

Der Systemtest besteht aus mehreren Testbereichen (siehe nächster Abschnitt), die einzelne Testfälle thematisch zusammenfassen. Auch die Verzeichnisstruktur richtet sich nach dieser Gliederung. Alle Testdaten liegen im Verzeichnis `test`, das wiederum Unterverzeichnisse für alle Testbereiche hat. Innerhalb dieser Bereichsverzeichnisse liegen die C++-Quelltexte und die Soll-Protokolle (Suffix `.xlg`). Die IML-Dateien des C++-Frontends (Suffix `.iml`) bzw. die gebundenen IML-Dateien (Suffix `.sys`) werden im Unterzeichnis `obj` des `test`-Verzeichnisses abgelegt. Zur Generierung der `.sys`-Dateien dient das Ziel `test` des Makefile. Neben den Daten liegen in den Unterverzeichnissen auch die verschiedenen Makefiles, die aus dem Haupt-Makefile heraus aufgerufen werden. Die Shell-Skripte, die zum Testen benötigt werden, befinden sich im `bin`-Verzeichnis (vgl. Abbildung 7.2).

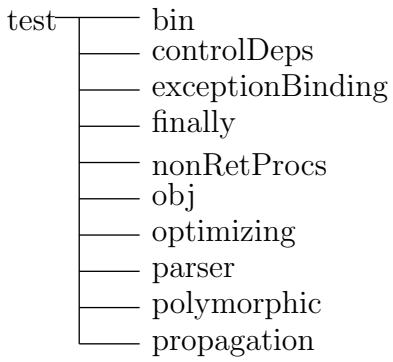

Abb. 7.2: Verzeichnisstruktur der Regressionstest

7.5 Visualisierung der Ergebnisse

Beim Testen stellte sich heraus, dass eine grafische Darstellung sowohl der rein textuellen als auch der HTML-Darstellung bei weitem überlegen war. Außerdem bestand die Notwendigkeit, Grafiken für diese Ausarbeitung zu erzeugen. Um zukünftige Tester bei ihrer Arbeit zu unterstützen, wurde in den Makefiles der einzelnen Tests das Ziel `png` definiert. Dieses veranlasst die Generierung einer grafischen Darstellung der CFGs aller Testfälle im PNG-Format. Die Grafiken werden im selben Verzeichnis wie die Quellen abgelegt. Voraussetzung ist das Vorhandensein (und die Korrektheit) der Werkzeuge `cfg2dot` (vgl. Abschnitt 7.6.10) und `dot`.

7.6 Testbereiche

Beim Test sind mehrere thematische Bereiche zu unterscheiden. Jeder Bereich wird in einem eigenen Abschnitt mit seinen Testfällen beschrieben. Angaben zu Dateien beziehen sich immer auf das `test`-Verzeichnis.

7.6.1 Parser der externen Deklarationsdatei

Für den Parsertest wird ein einfacher Quellcode (`parser/parser.cxx`) verwendet, der die Routinen `func1`...`func5` und die Ausnahmeklassen `exception1`...`exception4` definiert. Diese werden in den folgenden Deklarationsdateien verwendet.

7.6.1.1 Parsertest

Hier wird getestet, ob sämtliche syntaktischen Konstruktionen fehlerfrei gelesen werden. Das Programm würde terminieren, falls es die Deklarationsdatei nicht parsen könnte (was ein Fehler wäre). Alle Deklarationen in der Datei beziehen sich auf die o.g. Routinen und Klassen.
→ Deklarationsdatei: `parser/correct.decl`

7.6.1.2 Nicht existierende Routine

Es werden Deklarationen zu nicht existierenden Routinen (`GeorgeWBush` und `HarryKrishna`) und einem nicht existierenden Ausnahmetyp (`heavenlyException`) gemacht. Das Programm muss mit einer entsprechenden Fehlermeldung abbrechen.
→ Deklarationsdatei: `parser/nonexisting.decl`

7.6.1.3 Primitive Datentypen

Einer bestehenden Routine werden primitive Ausnahmetypen zugeordnet. Dieser Test prüft, ob der Parser diese korrekt erkennt und ob die Literale den entsprechenden IML-Knoten zugeordnet werden.
→ Deklarationsdatei: `parser/primitive.decl`

7.6.1.4 Mehrfachdeklarationen

In diesem Test wird der Fehlerfall der Mehrfachdeklarationen geprüft. Eine bestehende Routine wird mehrfach als terminierend deklariert. Der Parser muss dies erkennen und mit einem Fehler abbrechen.
→ Deklarationsdatei: `parser/redefinition.decl`

7.6.2 Vererbung, Polymorphie und Funktionszeiger

Dieser Abschnitt beschreibt die Tests, die sich mit der Typbestimmung von Klassentypen und der Ermittlung der Zielmethoden von Aufrufen befassen. Beides ist grundlegend für den Aufbau des interprozeduralen CFG. Zukünftige Projekte im bauhaus-Umfeld, die sich mit objektorientierten Programmen beschäftigen, finden hier eine reichhaltige Funktionssammlung.

7.6.2.1 Subtypbestimmung

Die Funktion `Is_Subtype` (s. Abschnitt 6.3.3.2, S. 62) wurde mit mehreren Wegwerftests getestet. Eine automatische Prüfung ist nicht möglich, da die Funktion von der Kommandozeile nie isoliert aufgerufen werden kann. Die folgenden Tests zur Polymorphie testen diese Funktion implizit mit.
→ Quellcodedatei: *keine*

7.6.2.2 Virtuelle und redefinierte Methoden

Implizit testet dieser Testfall auch die Subtypbestimmung. Im Quellcode werden verschiedene Strukturen und Klassen definiert, die teilweise voneinander abgeleitet sind. Diese besitzen teilweise virtuelle Methoden, die zum Teil auch abstrakt sind. Das Programm muss die möglichen Ziele der polymorphen Aufrufe korrekt bestimmen.
→ Quellcodedatei: `polymorphic/virtual.cxx`

Dieselben Klassen wie in Abbildung 5.6 (S. 54) vorausgesetzt, muss der folgende Quellcode

```
int x;
Boat *pBoat = new Amphibian;
Car *pCar = new Car;
x = pBoat->method ();          // (1)
x = pBoat->virtMethod ();      // (2)
x = pCar->method ();           // (3)
x = pCar->virtMethod ();       // (4)
```

diese Methoden als Ziele der Funktionsaufrufe ermitteln:

(1) `Boat::method` (Direct_Call)

(2) `Amphibian::virtMethod` (Virtual_Call)

(3) `Car::method` (Direct_Call)

(4) `Amphibian::virtMethod`, `Car::virtMethod` (Virtual_Call)

7.6.2.3 Funktionszeiger

Dieser Testfall prüft, ob die Ziele von Methodenaufrufen mit Funktionszeigern korrekt berechnet werden. Im Quellcode werden dieselben Klassen wie in Abbildung 5.6 (S. 54) definiert. Die folgenden Aufrufe (jeweils ein Pointer_To_Method_Call)

```
Boat *pBoat = new Amphibian;
Car *pCar = new Car;
int (Vehicle::*vfp)() = &Vehicle::virtMethod;
int x = (pBoat->*vfp)();            // (1)
int (Vehicle::*dfp)() = &Vehicle::method;
int y = (pBoat->*dfp)();            // (2)
```

müssen diese Methoden als Ziele der Funktionsaufrufe ermitteln:

(1) `Amphibian::virtMethod`, `Car::virtMethod`.

(2) `Vehicle::method`

Dieses Beispiel entspricht dem in Abschnitt 5.12.3 (S. 52). Erläuterungen siehe dort.
→ Quellcodedatei: `polymorphic/functionpointer.cxx`

7.6.2.4 Ermittlung der Ausnahmetypen

Implizit testet dieser Testfall auch die Subtypbestimmung und die Einschränkung der Ausnahmetypen. In diesem Test müssen die potentiellen Typen einer Ausnahme ermittelt werden, die der `throw`-Anweisung als Zeiger übergeben wird. Die beiden folgenden Abbildungen zeigen zum einen die Klassenhierarchie der Ausnahmen in einer Pseudoprogrammiersprache (Abbildung 7.3) und den Quellcode zur Funktion `main` (Abbildung 7.4). Abbildung 7.5 (S. 75) zeigt den CFG mit den propagierten Ausnahmekanten und die implizite Erweiterung der Zeiger um deren Unterklassen.
→ Quellcodedatei: `polymorphic/exception.cxx`

```
class exception1;
class exception2 : public exception1;
class exception3 : public exception1;
class exception4 : public exception2;
class exception5 : public exception1;
class exception6 : public exception2, public exception3;
```

Abb. 7.3: Klassenhierarchie der Ausnahmen

7.6.3 Binden von Ausnahme und Handler

Die folgenden Testfälle benutzen keine `finally`-Blöcke. Diese werden gesondert getestet.

```
exception1* excA;
exception4* excB;
//
switch (argc){
case 2:
     throw new exception1;
case 3:
     throw excA;
case 4:
     throw excB;
case 5:
     throw (int) 42;
case 6:
     throw ''string exception'';
default:
     return 0;
}
```

Abb. 7.4: Quellcode mit explizit geworfenen Ausnahmen

7.6.3.1 Ausnahmequelle und Handler in derselben `try`-Anweisung

In diesem Test werden Ausnahmen explizit mit der `throw`-Anweisung geworfen. Der Typ der Ausnahmen ist eindeutig. Folgende Konstellationen zwischen Ausnahmetyp und behandeltem Typ werden getestet:

- Gleicher Typ von Ausnahme und Handler. Als Typen werden Klassen, Strukturen und vordefinierte Typen verwendet, außerdem Zeiger und Referenzen auf die genannten Typen. Zusätzlich werden verschiedene Qualifizierer gestestet (`const`, ...).
- Der Ausnahmetyp ist Subtyp des behandelten Typs. Auch hier werden als Typen Klassen, Strukturen und vordefinierte Typen verwendet, außerdem Zeiger und Referenzen auf die genannten Typen. Zusätzlich werden verschiedene Qualifizierer gestestet (`const`, ...).
- Aufgrund der Einschränkungen von C++ kann *nicht* getestet werden, ob ein Ausnahmetyp, der ein Interface implementiert, an einen Handler für dieses Interface gebunden wird.

Der Kontrollfluss ist dann korrekt, wenn keine Ausnahmen propagiert werden. Alle Ausnahmen müssen an den richtigen Handler gebunden sein.
→ Quellcodedatei: `exceptionBinding/statementLocal.cxx`

7.6.3.2 Ausnahmequelle und Handler innerhalb derselben Routine

Die getesteten Konstellationen sind wie im vorigen Abschnitt. Dieser Test prüft, ob nicht gefangene Ausnahmen an die umgebende `try`-Anweisung weitergereicht werden. Der Kontrollfluss ist dann korrekt, wenn keine Ausnahmen propagiert werden. Alle Ausnahmen müssen an den richtigen Handler gebunden sein.
→ Quellcodedatei: `exceptionBinding/routineLocal.cxx`

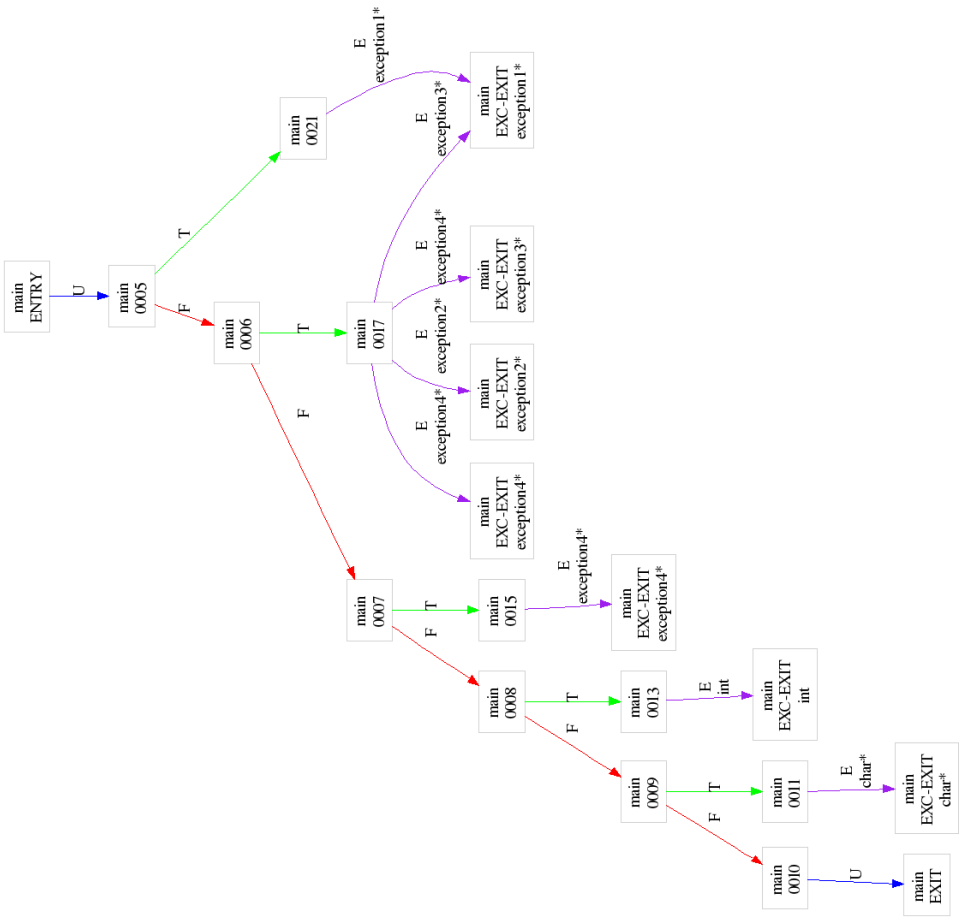

Abb. 7.5: CFG mit propagierten, ungefangenen Ausnahmen

7.6.4 Propagierte Ausnahmen

7.6.4.1 Ausnahmequelle ohne Handler innerhalb der Routine

Dieser Test prüft, ob nicht gefangene Ausnahmen aus einer Routine propagiert werden. Ausnahmen werden explizit mit der `throw`-Anweisung geworfen und besitzen einen eindeutigen Typ.
→ Quellcodedatei: `propagation/propagated.cxx`

7.6.4.2 Routinenaufruf mit propagierten Ausnahmen

Ausnahmen werden in diesem Test sowohl explizit geworfen als auch von Routinenaufrufen propagiert. Der Quellcode ist so angelegt, dass keine zyklischen Aufrufe vorhanden sind. Alle Ausnahmen werden an einen Handler gebunden, und jeder Handler fängt mindestens eine Ausnahme.
→ Quellcodedatei: `propagation/non-cyclic.cxx`

7.6.4.3 Zyklische Aufrufe

Abschnitt 5.11.3 (S. 49) beschreibt das Problem zyklischer Aufrufe mit propagierten Ausnahmen. Dieser Test widmet sich diesem Problem. Im Quellcode ist eine Reihe von Funktionen definiert, von der jede eine bestimmte Ausnahme propagiert. Keine dieser Funktionen fängt eine Ausnahme. Das Ergebnis ist dann korrekt, wenn die `main`-Funktion alle Ausnahmen propagiert.
→ Quellcodedatei: `propagation/cyclic.cxx`

7.6.5 Nicht zurückkehrende Routinen

7.6.5.1 `goto`-Endlosschleife

In diesem Testfall ist die Routine `endless` so gebaut, dass sie aufgrund einer `goto`-Schleife niemals zurückkehren kann. Die Routine `alwaysEndless` besteht nur aus einem Aufruf von `endless`, kann also ebenfalls nicht zurückkehren. Die beiden Routinen `sometimesEndless` und `main` sind potentiell nicht zurückkehrend, da die Aufrufe von `endless` bzw. `alwaysEndless` von einer Bedingung abhängig sind. Die Kontrollflussanalyse muss diese Besonderheiten erkennen und darf keine Kontrollflusskanten zu den entsprechenden CFG-Exit-Knoten bzw. den Folgeknoten der Aufrufe anlegen. Abbildung 7.6 zeigt den CFG mit den unerreichbaren Exits und den Aufrufen ohne Nachfolgern.
→ Quellcodedatei: `nonRetProcs/endless.cxx`

7.6.5.2 Benutzerdefinierte nicht zurückkehrende Routine

In diesem Testfall sind zwei Routinen `alice` und `bob` definiert. `alice` besteht aus einem leeren Rumpf, `bob` ist als extern deklariert: Somit existieren weder ein `Routine`-Knoten noch Grundblöcke für diese Routine! In der Konfigurationsdatei werden beide Routinen als definitiv nicht zurückkehrend festgelegt. Die Analyse muss dies bei der Generierung des CFG berücksichtigen. Im Gegensatz zu tatsächlich nicht zurückkehrenden Routinen existiert in solchen Fällen ein Pfad zwischen dem CFG-Entry und dem CFG-Exit. Die Aufrufstellen solcher Routinen besitzen jedoch keinen Nachfolger, wie in Abbildung 7.7, S. 78 zu sehen ist.
→ Quellcodedatei: `nonRetProcs/forUserDef.cxx`

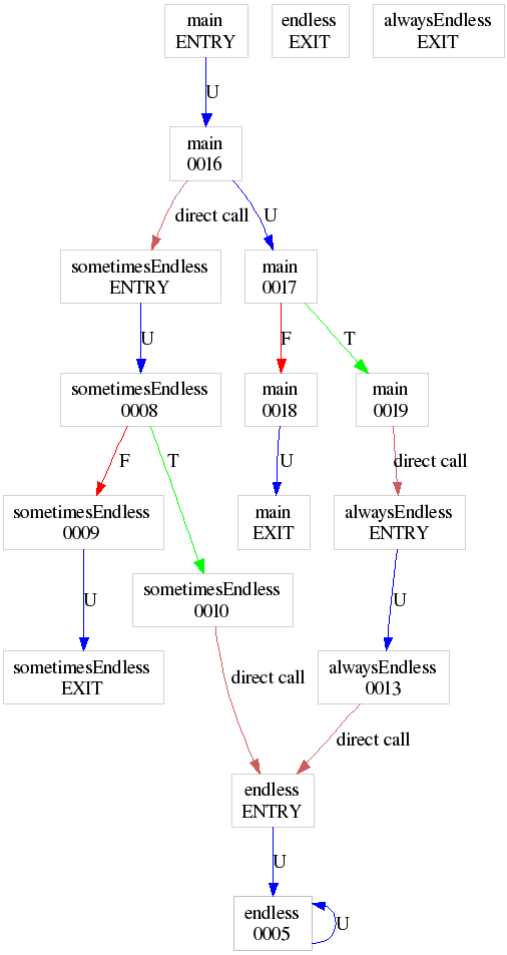

Abb. 7.6: CFG mit definitiv und potentiell nicht zurückkehrenden Routinen

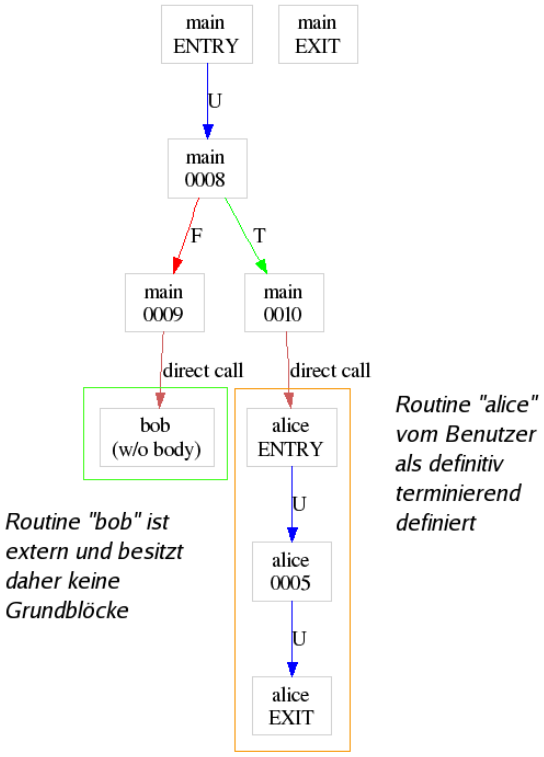

Abb. 7.7: CFG mit benutzerdefinierter DNRP

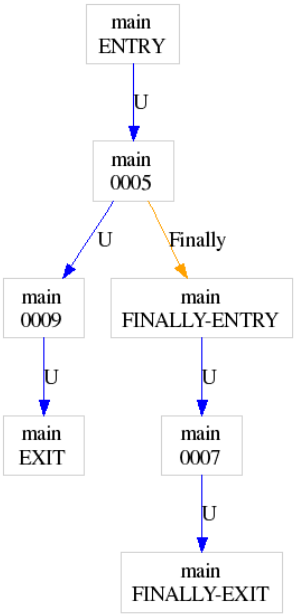

Abb. 7.8: CFG mit `finally`-Aufruf am Ende eines `try`-Blocks

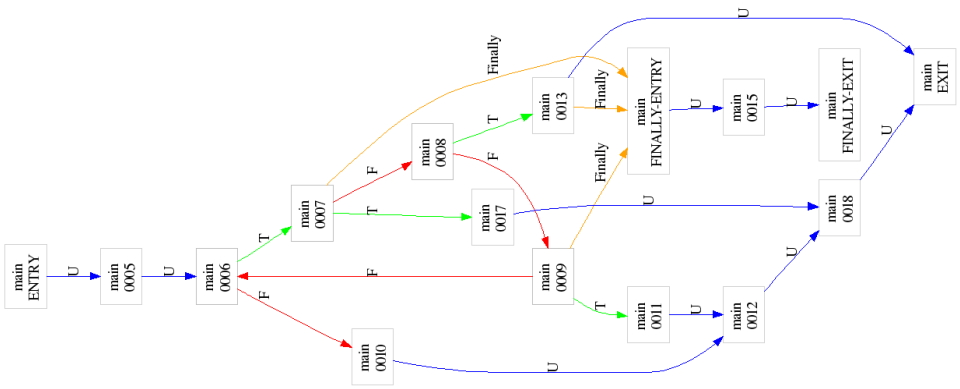

Abb. 7.9: CFG mit `finally`-Aufrufen wegen Sprunganweisungen

7.6.6 `finally`-Aufruf

Bei allen folgenden Tests wird der Strukturtyp mit dem Namen `FINALLY` als Platzhalter für den `finally`-Block verwendet.

7.6.6.1 Substitution eines `catch`-Handlers mit einem `finally`-Block

Um mit `cafe++` überhaupt `finally`-Aufrufe verwenden zu können, muss ein `catch`-Handler durch einen `finally`-Block ersetzt werden. Diese Hilfskonstruktion ist in Abschnitt 7.1 (S. 67) beschrieben. Mit diesem Test wird überprüft, ob diese Substitution korrekt erfolgt. `iml2cfg` gibt bei einer Ersetzung eine entsprechende Meldung aus.
→ Quellcodedatei: `finally/substitution.cxx`

7.6.6.2 Normales Verlassen des `try`-Blocks

In diesem Testfall wird überprüft, ob nach dem Ausführen der letzten Anweisung innerhalb eines `try`-Blocks der `finally`-Block ausgeführt wird. Der Aufruf des `finally`-Blocks ist in Abbildung 7.8 (S. 78) dargestellt.
→ Quellcodedatei: `finally/normalExit.cxx`

7.6.6.3 Explizites Verlassen des `try`-Blocks

Unbedingte Sprünge innerhalb einer `try`-Anweisung müssen die textuell umgebenden `finally`-Blöcke ausführen, bevor der Sprung selbst ausgeführt wird. Es wird mit den folgenden drei Anweisungen getestet:

- `return`
- `goto`
- `break`

Der Testquellcode besteht aus einer einzigen Routine, die die o.g. Anweisungen innerhalb eines `try`-Blocks mit `finally`-Block verwendet. Der resultierende CFG ist in Abbildung 7.9, S. 79 abgebildet.
→ Quellcodedatei: `finally/explicitExit.cxx`

7.6.6.4 Verschwinden von Ausnahmen

Das Verschwinden von Ausnahmen ist in Abschnitt 2.7 (S. 9) erläutert. Da der CFG alle Anweisungen in eine sequentielle Reihenfolge bringt, muss das Verschwinden von Ausnahmen erkannt werden. Dieser Test prüft, ob die verdeckten Ausnahmen tatsächlich nicht mehr im CFG erscheinen. Zwei Möglichkeiten, wie Ausnahmen verschwinden können, werden getestet:

1. Der Konstruktor einer Klasse `Exception` wirft eine weitere Ausnahme, `CtorException`. Es darf also keine Ausnahme an den Handler für `Exception` gebunden werden.

2. Im `finally`-Block wird die `main`-Routine mit einem `return` verlassen. Somit darf `main` keine Ausnahmen propagieren.

Den resultierenden CFG, in Abbildung 7.10 auf Seite 81 dargestellt, möchte ich ein wenig erläutern: Der Knoten *Exception/EXIT* (oben rechts) besitzt keine eingehende Kante, da der Konstruktor der Klasse *Exception* immer eine Ausnahme wirft. *Exception/0008* instanziiert mittels des `operator new` ein Objekt der Klasse `CtorException` und wirft dieses als Ausnahme, deren Propagierung mit dem Knoten *Exception/EXC-EXIT/CtorException** abgebildet ist. Dadurch kann *Exception/EXIT* kein Nachfolger von *Exception/0008* sein. Die aufrufende Routine, `main`, besitzt zwar keinen Grundblock, der eine Ausnahme behandeln könnte, aber *main/0012* besitzt einen `finally`-Block (*main/FINALLY-ENTRY*), der die Routine unbedingt fehlerfrei verlässt, wie an der Kante zu *main/EXIT* zu sehen ist. Damit wird keine der beiden Ausnahmen propagiert, beide gehen verloren. Eventuell vorhandene Anweisungen, die nach dem `finally`-Block ausgeführt werden sollten, sind unerreichbar.
→ Quellcodedatei: `finally/disappearing.cxx`

7.6.7 Intra- und interprozedurale Kontrollabhängigkeit

Bisher wurde die Kontrollabhängigkeit intraprozedural berechnet. Mit dieser Diplomarbeit kommt die interprozedurale Kontrollabhängigkeit (KA) neu hinzu. Die folgenden Tests decken beide ab, wobei der Schwerpunkt deutlich auf der interprozeduralen KA liegt. Wird sie korrekt berechnet, so muss auch die intraprozedurale KA korrekt berechnet werden.

7.6.7.1 Bedingung

Dieser Testfall untersucht die intraprozedurale Kontrollabhängigkeit anhand mehrerer geschachtelter `if`-Anweisungen.
→ Quellcodedatei: `controlDeps/nestedConditions.cxx`

7.6.7.2 Routinenaufruf

Dieser Testfall untersucht die interprozedurale Kontrollabhängigkeit, besonders die Abhängigkeit der Eintrittsgrundblöcke von Aufrufstellen.
→ Quellcodedatei: `controlDeps/routineCalls.cxx`

7.6.7.3 Aufruf potentiell terminierender Routinen

Bei PNRPs sind die Grundblöcke, die (evtl.) auf deren Aufruf folgen, von dem Prädikat abhängig, das in der aufgerufenen Routine darüber entcheidet, ob die Routine terminiert oder zurückkehrt. Dieser Fall wird hier getestet.
→ Quellcodedatei: `controlDeps/pnrp.cxx`

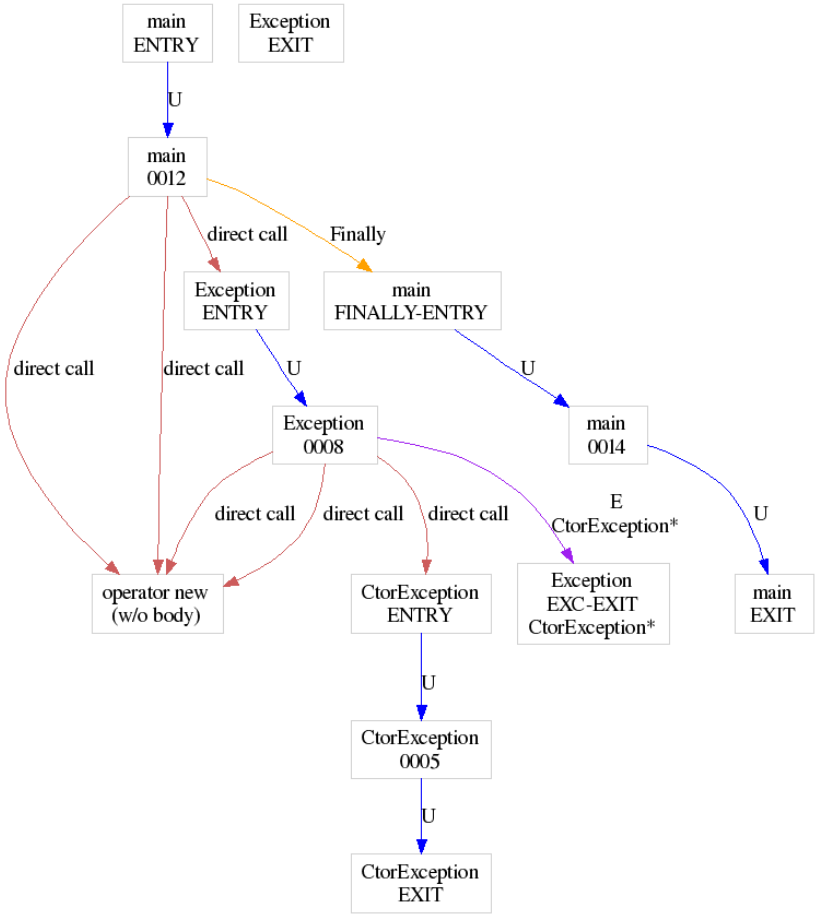

Abb. 7.10: CFG mit verschwundenen Ausnahmen

```
digraph G
{
        0 [label="entry", color=lightgrey, shape=box];
        1 [label="throw", color=lightgrey, shape=box];
        2 [label="catch", color=lightgrey, shape=box];
   0 -> 1 [label=" uncond",      color="blue",   width=3];
   1 -> 2 [label=" exceptional", color="purple", width=3];
}
```

Abb. 7.11: Beispiel einer `.dot`-Datei

7.6.8 Grundblockoptimierung

Die Optimierung besteht aus zwei Teilen: Dem Entfernen nicht erreichbarer Grundblöcke und dem Vereinen von Grundblöcken. Der Testquellcode beinhaltet sowohl nicht erreichbare Teile als auch Anweisungen, die das Anlegen von mehreren Grundblöcken provozieren (Routinenaufrufe, `if`-Anweisungen, etc.). Um eine Sequenz von Grundblöcken zu erzwingen werden etliche Sprungmarken definiert, die jedoch niemals Ziel einer Sprunganweisung sind. Diese Grundblöcke müssen zu einem einizgen Grundblock vereint werden.
→ Quellcodedatei: `optimizing/bb.cxx`

7.6.9 `cfg2html`

Das Werkzeug `cfg2html` wird nicht automatisch getestet. Das Format der HTML-Ausgabe hängt von verschiedenen Faktoren ab, sodass ein zuverlässiger Test ein anderes Konzept als hier verwendet benötigt. Die Anpassungen an `cfg2html` sind außerdem in keiner Weise Bestandteil der Aufgabenstellung dieser DA.

7.6.10 `cfg2dot`

Das Werkzeug `cfg2dot` wird nicht automatisch getestet, da die Nummerierung der Knoten in der `.dot`-Datei (siehe Beispiel in Abbildung 7.11) von der Traversierungsreihenfolge abhängig ist. Die Reihenfolge kann sich durch Änderungen am Quellcode leicht ändern, sodass in diesem Fall jedesmal der Test angepasst werden müsste. Die Anpassungen an `cfg2html` sind ebenfalls in keiner Weise Bestandteil der Aufgabenstellung dieser DA.

Kapitel 8

Messungen

8.1 Auswahl des Quellcode

Mein erster Ansatz zur Suche nach Quellcode war, im Internet auf `http://www.sourceforge.net` nach passenden Projekten zu suchen. Als Auswahlkriterien galten:

a) Die Programmiersprache musste C++ sein.

b) Die Plattform musste Linux oder Unix sein. C/C++-Quellcode für Windows ist typischerweise für einen Microsoft-Compiler geschrieben. Da Microsoft eine Version von C/C++ mit eigenen Erweiterungen verwendet, sind Syntaxfehler beim Umwandeln mit dem edg-Frontend (`cafe++`) sehr wahrscheinlich.

c) Im Quellcode müssen `try`- und `catch`-Anweisungen vorhanden sein.

Es stellte sich heraus, dass viele Projekte die Punkte a) und b) erfüllen. Punkt c) entwickelte sich jedoch zum Problem. Als Beispiel sei die Open Source Datenbank *MySQL* genannt. Der Quellcode des DB Servers ist mehrere Megabyte groß. Es existiert jedoch nur eine einzige Ausnahme-Klasse, und dies auch nur in einem Beispielprogramm.

Letztendlich stellte sich heraus, dass die Suche im Internet der falsche Weg war. Ein Blick auf den eigenen Rechner lieferte genügend Testkandidaten:

- **Xerces**, der XML-Parser des Apache Project, wird von bauhaus verwendet. Der C++-Quellcode ist ebenfalls Teil von bauhaus (XML4Ada95). Die Software kann von der Apache-Homepage bezogen werden:
  `http://xml.apache.org/xerces-c`

- **Mozilla**, der freie Internetbrowser und Nachfolger von Netscape. Die Sourcen können direkt vom Mozilla-CVS-Server heruntergeladen werden. Die Entwickler beschreiben den Download und den Übersetzungsvorgang auf `http://www.mozilla.org/cvs.html`.

- Die Quellen zu **Open Office** können von deren Website[1] heruntergeladen werden. Ausgepackt ergeben sich knapp 800MB Quellen (inkl. Skripten, Grafiken, etc.).

[1] `http://www.openoffice.org/dev_docs/source/get_source.html`

8.2 Umgebung

Für die Messungen wurden die Rechner pslx1 und pslx2 der Abteilung benutzt, da sie über genügend Speicherplatz und Rechenleistung verfügten. Beide Maschinen sind Intel-Rechner mit Linux (SMP-Kernel, Version 2.6.5) als Betriebssystem, besitzen jeweils vier Xeon CPUs mit 3.06 GHz und 512 MB Cache und 4 GB Hauptspeicher.

8.3 Mozilla

Dieser Abschnitt beschreibt die Messungen mit Mozilla, Version 1.8a5, Codename *Gecko*.

Mozilla besteht aus 1.512 Dateien C-Quellcode, 4.407 Dateien C++-Quellcode und 8.079 Header-Dateien. Insgesamt ergibt dies knapp 500 MB.

Eine Analyse der Ausnahmetypen brachte keine Ergebnisse. Offensichtlich wird an einigen Stellen vom Ausnahmekonzept Gebrauch gemacht. Eine klare Klassenhierarchie scheint es bei den Ausnahmen nicht zu geben. So werden von Klassen, die keine Ausnahme darstellen, Ausnahmeklassen abgeleitet. Allgemein scheinen die Ausnahmen von Xerces zu stammen. Mozilla selbst definiert keine weiteren Aushametypen. Auf der Homepage von Mozilla existieren viele Dokumente, die die Architektur beschreiben. Auch zum Thema "Building Mozilla" wird eine Vielzahl an Dokumenten abgeboten.

Um die Quellen nach IML zu übersetzen, wurden alle Quellen sowohl mit `gcc/g++` als auch mit `cafe++` compiliert. Die Generierung der normalen Objektdateien ist notwendig, da der Make-Prozess aus diesen Dateien Bibliotheken und andere Zwischenergebnisse baut. Scheitert dies, bricht der Prozess ab. Eine Anpassung des Prozesses war aus Zeitgründen und wegen der Komplexität unmöglich. Die Übersetzung der Quellen in einzelne IML-Dateien war nach einigen Anlaufschwierigkeiten problemlos möglich, nur in wenige Quelldateien fand `cafe++` Fehler. Mozilla selbst wurde vom Makeprozess erstellt und war lauffähig. Das Linken der IML-Dateien entwickelte sich jedoch zu einem Problem. Anzahl und Größe der Dateien ließen auf lange Laufzeiten der Werkzeuge schließen. Die Unkenntnis der Architektur machte es unmöglich, einzelne Komponenten isoliert zu betrachten.

Nach einem Tag Einarbeitung in die Dokumentation von Mozilla sah ich kaum Erfolgschancen, in vertretbarer Zeit erkennbare Fortschritte zu erzielen. Ich habe keine weitere Zeit in Mozilla investiert und mich Xerces gewidmet.

8.4 Xerces

Dieser Abschnitt beschreibt die Messungen mit Xerces-C, Version 2.6.0. Die Quellen von Xerces-C bestehen aus 94 Dateien C-Quellcode, 512 Dateien C++-Quellcode und 1.977 Header-Dateien. Insgesamt macht dies etwa 25,5 MB Quellcode oder 576 KLOC. Von Ausnahmen wird reger Gebrauch macht. Es existieren fast 1.000 `throw`-Anweisungen.

8.4.1 Erstellungsprozess

Wie auch bei Mozilla wurden die Quellen sowohl mit `gcc/g++` als auch mit `cafe++` compiliert. Dabei konnte auf die zuvor für Mozilla erstellten Skripte zurückgegriffen werden. Die Übersetzung der Quellen in einzelne IML-Dateien war problemlos möglich, nur in wenigen Quelldateien fand `cafe++` Fehler. Die Xerces-Bibliothek wurde vom Makeprozess erstellt. Das Linken der IML-Dateien erfolgte manuell. Die gebundene Datei hatte eine Größe von 214,5 MB, bestehend aus 344 Modulen

mit 70.140 Definitionen. 468.945 Deklarationen waren externen, zu 5.881 Entitäten wurde keine Definition gefunden.

Anders als bei den selbst erstellten Testfällen ist Xerces kein in sich abgeschlossenes System. Xerces verwendet einige Bibliotheken, zu denen kein Quellcode vorlag. Durch diese Lücken in der IML-Darstellung wurde von `iml2cfg` eine gewisse Robustheit verlangt.

8.4.2 Anwendung der Kontrollflussanalyse

8.4.2.1 Aufgetretene Probleme

Erste Versuche, die Kontrollflussanalyse anzuwenden endeten mit Abstürzen von `iml2cfg` nach Laufzeiten zwischen 30 und 60 Minuten. Die Fehlerursachen waren von den vorangegangenen Tests nicht identifiziert worden. Bei der Ursachenforschung stellte sich heraus, dass die `if`- und `switch`-Anweisung nicht korrekt verarbeitet wurde (Altcode). Daneben verursachte die Grundblockoptimierung einige Fehler. Nach diesen ersten Bugfixes wurde das Programm wegen Verletzung von Annahmen (`asserts`) nach Laufzeiten von mehreren Stunden abgebrochen.

8.4.2.2 Analyse einzelner Dateien

Wegen der fortgeschrittenen Zeit entschied ich mich zweigleisig fortzufahren. Neben der Fehlersuche und -korrektur wurden die IML-Objektdateien von Xerces einzeln von `iml2cfg` bearbeitet, um Messergebnisse zu erhalten. Diese Analysen verliefen ohne Abstürze. 320 IML-Objektdateien wurden untersucht. Dazu wurden die von `cafe++` erstellten IML-Dateien gebunden (`imllink`) und von `iml2cfg` jeweils einzeln analysiert. Die Größe der gebundenen IML-Objektdateien lag im Schnitt bei 1,83 MB (minimal 9 KB, maximal 2,9 MB), der durchschnittliche Größenzuwachs bei 18% (minimal 1%, maximal 25%). Die Laufzeit von `iml2cfg` wurde nicht in CPU-Sekunden gemessen, sondern als Zeitspanne zwischen Aufruf und Terminierung. Da auf dem Testrechner zu diesem Zeitpunkt keine anderen Benutzer angemeldet waren und keine (anderen) rechenintensiven Prozesse aktiv waren, ist diese Messung ohne nennenswerten Messfehler. `iml2cfg` benötigte zwischen 1 und 14 Sekunden, im Durchschnitt 5 Sekunden pro Datei. Dies ergibt eine mittlere Laufzeit von 2,2 Sekunden/MB (minimal 0,5 Sek./MB, maximal 108 Sek./MB).

8.4.2.3 Analyse eines Teilsystems

Nach der erfolgreichen Analyse einzelner Dateien habe ich eine Teilmenge der IML-Objektdateien zu einem System gebunden. Es wurden die 256 kleinsten Dateien ausgewählt. Diese Auswahl war willkürlich. Um einen Eintrittspunkt zu erhalten, wurde neben den Bibliotheksobjekten auch das mitgelieferte Beispiel `DOMCount` übersetzt und eingebunden, das eine `main`-Routine besaß. Die resultierende IML-Datei hatte eine Größe von 98,8 MB. `iml2cfg` analysierte sie in 1 Stunde 32 Minuten und 38 Sekunden (reine CPU-Zeit). Die Laufzeit betrug weit mehr als drei Stunden, was auf den Bedarf an Hauptspeicher und damit verbundenes Swapping zurückgeführt werden konnte. Die von `iml2cfg` generierte Datei hatte eine Größe von 115,25 MB, was einem Größenzuwachs von 16,6% entspricht. Dieser Wert korrespondiert mit den Ergebnissen des letzten Abschnitts. Pro 1 MB Eingabedaten ergibt sich eine CPU-Zeit von ca. 52 Sekunden, was die Werte des letzten Abschnitts deutlich überschreitet. Dies lässt auf eine exponentiell wachsende CPU-Zeit bezogen auf die Größe der Eingabedaten schließen.

8.5 Open Office

Nach den Versuchen mit Xerces und Mozilla blieb keine Zeit mehr, Open Office mit `iml2cfg` zu analysieren. Aufgrund der Erfahrungen mit Mozilla stellt sich die Frage, ob der Zeitaufwand und die zu erwartenden Resultate in einer sinnvollen Relation stünden. Seine Praxistauglichkeit hat `iml2cfg` mit Xerces unter Beweis gestellt.

Kapitel 9

Verwendung der Analyseergebnisse

9.1 Traversierung des erzeugten CFG

Während der Implementierung der Werkzeuge wurde häufig die Frage gestellt, wie mit den Kontrollflussinformationen im IML-Graph umzugehen sei. Dieses Kapitel fasst die wichtigsten Aspekte aus den vorausgehenden Kapiteln nochmals kurz zusammen und erklärt, wie ein CFG zu traversieren ist. Der Fokus in diesem Abschnitt liegt auf der vollständigen Beschreibung aller Kontrollflusskanten.

9.1.1 Genau ein unbedingter Nachfolger

Besitzt ein Grundblock nur einen möglichen direkten Nachfolger, der bei der normalen Ausführung des Programms erreicht wird, so wird dies mit einer uncond-Kante modelliert. Der Knoten wirft keine Ausnahmen. Wird bei der Traversierung nur eine solche Kante vorgefunden, so muss diese Kante verfolgt werden.

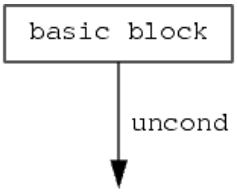

Abb. 9.1: Unbedingter Nachfolger

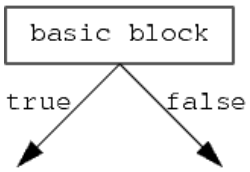

Abb. 9.2: Bedingte Nachfolger

9.1.2 Bedingte Nachfolger

Ein Grundblock, dessen letzte Anweisung ein Prädikat ist, besitzt zwei mögliche Nachfolger, je nach Auswertung des Prädikats. Wird bei der Traversierung ein solcher Grundblock vorgefunden, so müssen beide Kanten verfolgt werden. Ein Grundblock mit bedingten Nachfolgern kann niemals eine Ausnahme werfen oder einen `finally`-Block aufrufen.

9.1.3 Ausnahmenachfolger

Ein Grundblock, dessen letzte Anweisung eine Ausnahme erzeugt (explizit oder propagiert), besitzt für jeden möglichen Typ der Ausnahme eine exceptional-Kante. Wird bei der Traversierung ein solcher Grundblock vorgefunden, so müssen alle Kanten verfolgt werden. Eine Reihenfolge existiert nicht.

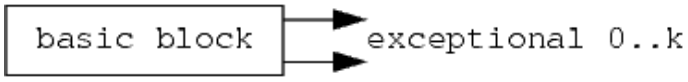

Abb. 9.3: Nachfolger für versch. Ausnahmetypen

9.1.4 `finally`-Aufruf

Ein Grundblock, dessen letzte Anweisung eine Ausnahme erzeugt (explizit oder propagiert), muss von jedem umschließenden `try`-Block den `finally`-Block aufrufen, bis ein `try`-Block die Aufnahme behandelt. Wird bei der Traversierung ein solcher Grundblock vorgefunden, so müssen

1. alle finally_call-Kanten in der Reihenfolge traversiert werden, wie sie in der Liste gespeichert sind
2. alle exceptional-Kanten traversiert werden – diese sind nicht geordnet.

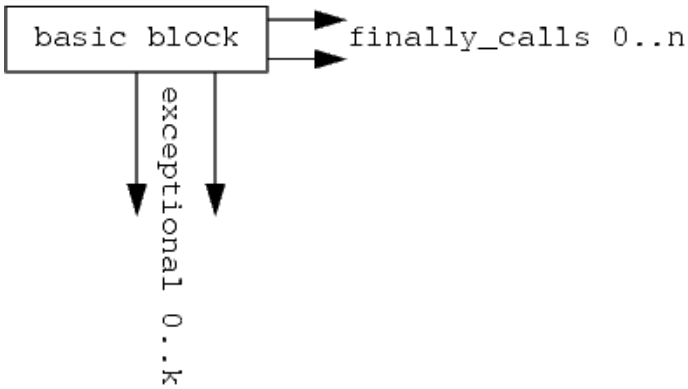

Abb. 9.4: `finally`-Aufruf wegen Ausnahmen

9.1.5 Unbedingter Nachfolger und Ausnahmenachfolger

Ein Grundblock, dessen letzte Anweisung eine Ausnahme erzeugen kann, besitzt für den Fall, dass keine Ausnahme auftritt, einen unbedingten Nachfolger. Für den Ausnahmefall besitzt er außerdem genau eine exceptional-Kante für jeden möglichen Ausnahmetyp. Diese Konfiguration kommt bei Routineaufrufen zustande, bei denen die Routine Ausnahmen propagieren kann. Abbildung 9.5 zeigt genau diesen Fall. Wird bei der Traversierung ein solcher Grundblock vorgefunden, so müssen sowohl die exceptional-Kanten als auch die uncond-Kante traversiert werden. Eine Reihenfolge muss nicht beachtet werden.

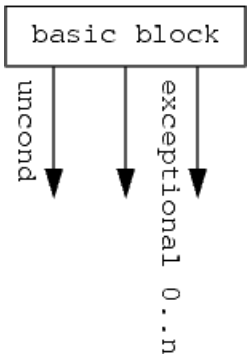

Abb. 9.5: uncond- und exceptional-Nachfolger

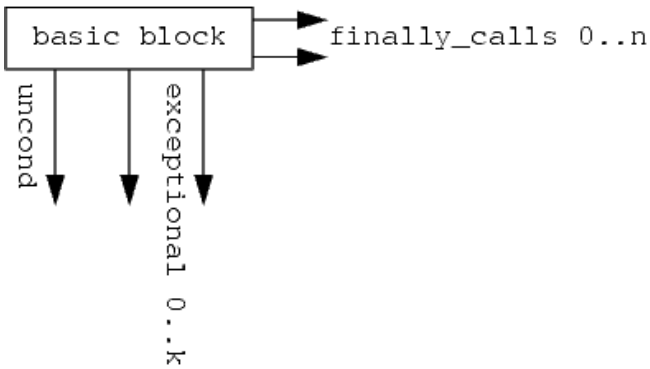

Abb. 9.6: uncond- und exceptional-Nachfolger mit `finally`-Aufrufen

9.1.6 Unbedingter Nachfolger und `finally`-Aufrufe

Ein Grundblock, der einen `try`-Block unbedingt und ohne eine Ausnahme zu werfen verlässt, muss diejenigen `finally`-Blöcke aufrufen, die Teil der zu verlassenden `try`-Anweisungen sind. Typischerweise sind dies unbedingte Transferanweisungen wie `return`, `goto`, etc. Abbildung 9.7 zeigt genau diesen Fall. Wird bei der Traversierung ein solcher Grundblock vorgefunden, so müssen zuerst alle finally_call-Kanten in der Reihenfolge traversiert werden, in der die gespeichert sind. Danach folgt die uncond-Kante. Neben den hier aufgezählten Konfigurationen existieren keine weiteren.

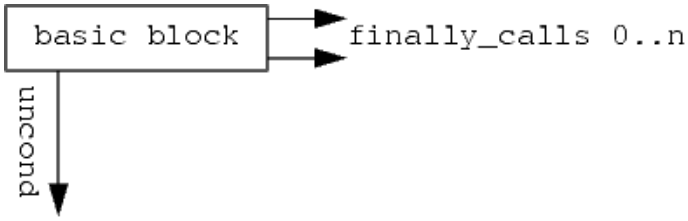

Abb. 9.7: uncond-Nachfolger und finally_call-Kanten

Kapitel 10

Zusammenfassung und Ausblick

10.1 Zeitplan und aufgetretene Probleme

Trotz einer großzügig bemessenen Vorbereitungs- und Einarbeitungsphase war es immer wieder notwendig, recht viel Zeit in das Verstehen der IML zu investieren. Es enstanden unzählige Zeilen Quellcode und IML-Graph-Skizzen. Auch der Aufwand, den das Wort "objektorientiert" des DA-Titels verursachte, wurde von mir stark unterschätzt. Dass die Arbeit an dieser Stelle nicht umsonst war, zeigte das Interesse der Mitarbeiter der Abteilung. Kaum war eine erste Version des `types`-Pakets fertig, wollten sie es gleich haben. Erst während Entwicklung und Dokumentation wurden zusätzliche Aufgaben deutlich, die zwar nicht Teil der Aufgabenstellung waren, aber als äußerst hilfreich erschienen: Die beiden Werkzeuge `cfg2dot` und `iml2html`. Für die Messungen mit Open Source Software habe ich sowohl Beschaffungs- als auch Anlaufschwierigkeiten einkalkuliert. Dass ein Programmlauf jedoch eine ganze Nacht dauern würde, habe ich nicht erwartet. Entsprechend sind die Messungen nicht so detailliert ausgefallen, wie ich es geplant hatte. Fazit: Die Zeit hat gereicht, wenn auch knapp.

10.2 Erfahrungen

Im Gegensatz zur Studienarbeit, in der C/C++ zur Implementierung verwendet wurde, war die Verwendung von Ada wesentlich angenehmer. Es gab keinerlei Unterschiede zwischen verschiedenen Ada-Compilern bzw. Compiler-Versionen. Durch die parallel stattfindende Weiterentwicklung der benutzten bauhaus-Bibliotheken (IML, Reuse) entstanden jedoch hohe Integrationsaufwände, die in der Planung nicht in diesem Maße berücksichtigt wurden. Die Annahme, die bisher vorhandene Kontrollflussanalyse wäre fehlerfrei, war zwar optimistisch, aber (fast) korrekt. Die daraus gezogene Schlussfolgerung, es wären kaum Aufwände zur Fehlerkorrektur an der vorhandenen Software notwendig, war ebenfalls optimistisch - aber falsch. Kleine Fehler hatten oftmals fatale Auswirkungen. Die Fehlersuche dauerte oft mehrere Tage, die Fehlerbeseitigung warf oft Fragen auf, die bisher überhaupt nicht berücksichtigt wurden. Der zeitliche Aufwand erhöhte sich dadurch merklich. Als positive Folge ergaben sich häufig Diskussionen mit den Mitarbeitern der Abteilung, die immer ein offenes Ohr hatten. Die verschiedenen Blickwinkel und vielen Ideen halfen mir immer, eine Lösung zu finden. Diese Zusammenarbeit hat sehr viel Spaß gemacht.

10.3 Bewertung des Projekts

Die in der Aufgabenstellung beschriebenen Anforderungen scheinen mir erfüllt zu sein. In Puncto Werkzeuge für den Benutzer habe ich mehr getan, als gefordert. Dies geschah zwar auch zum Eigennutz, bringt aber dem ganzen Projekt Vorteile, da keine Integrationsarbeit mehr stattfinden muss.

Anfangs waren die Arbeiten von Harrold et al. noch die Grundpfeiler dieser Diplomarbeit. Es hatte den Anschein, als würden sie den größten Teil der Konzepte für die DA beinhalten. Es stellte sich aber sehr schnell heraus, dass Harrold einige Details ignoriert (oder vergessen?) hatte, für die Lösungen gefunden werden mußten. Zwar sind nicht alle meine Lösungen optimal, aber es sind Lösungen, die tragen.

10.4 Zukünftige Arbeiten

10.4.1 Performanceoptimierung

Sowohl Laufzeit als auch Speicherbedarf des entwickelten Werkzeugs sind recht groß. Aus Zeitmangel wurden keine Maßnahmen geplant, dies zu verbessern. Der Produktiveinsatz der Software wird zeigen, in welchem Umfang Performanceverbesserungen notwendig sind.

10.4.2 Verbesserung des Tests

Die Verwendung von Meldungen für die Tests stellen eine potentielle Fehlerquelle für die Ergebnisse der einzelnen Tests dar. An dieser Stelle wäre es denkbar, sowohl für die Testskripte als auch für den Quellcode eine gemeinsame Ressourcendatei zu verwenden, in der die Meldungstexte mit Platzhaltern für Variablen abgelegt sind. Auf diese Art könnte die Fehlerquelle beseitigt werden.

Als eine einfachere Alternative dazu mag das Suchen und Ersetzen über mehrere Dateien erscheinen. Da die Texte aber aus mehreren Teilen zusammengesetzt werden, kann zwar mit regulären Ausdrücken gesucht werden, die Ersetzung muss aber manuell geschehen.

10.4.3 Erweiterung der Deklarationsdatei

Die bisherige Deklarationsdatei könnte um extern instanziierte Ausnahmeklassen erweitert werden. Dadurch würde die Analyse an Genauigkeit gewinnen. Andererseits soll die Deklarationsdatei kein Ersatz für Quellcode werden.

Kapitel 11

Glossar

AES/artificial exception source
Künstlicher →Grundblock, der für den interprozeduralen →CFG eingefügt wird. Ein AES-Grundblock ist bei propagierten Ausnahmen notwendig, um `finally`-Blöcke zu rufen. Diese Aufrufe können nicht direkt vom Call-Knoten ausgehen. Siehe Abschnitt 4.2.6.3.3, Seite 35.

Ausnahme/exception
Element moderner Programmiersprachen für die Behandlung von Fehlern. Man spricht vom "Werfen einer Ausnahme", wenn eine Ausnahme erzeugt wird und vom "Fangen einer Ausnahme", wenn der passende Handler die Ausnahme bearbeitet. Siehe auch Kapitel 11 in [Gos00].

Anreicherung des CFG/CFG augmentation
Zur Berechnung der interprozeduralen Kontrollabhängigkeit wird der intraprozedurale CFG um temporäre Prädikatsknoten (→EPN/→RPN) erweitert, die als Platzhalter für diejenigen Bedingungen fungieren, die in der gerufenen Routine darüber entscheiden, ob die aufgerufene Routine zurückkehrt (→RPN) oder eine bestimmte Ausnahme propagiert (→EPN). Die Platzhalter werden später entfernt und die Kontrollabhängigkeitsattribute entsprechend angepasst.

Ausnahmebehandlung/exception handling
Ausnahmebehandlung bezeichnet ein Verfahren, Informationen über bestimmte Programmzustände - meistens Fehlerzustände - an andere Programmebenen zur Behandlung weiterzureichen. Der Code zur Ausnahmebehandlung wird vom normalen Anwendungscode getrennt gehalten. Anstatt bei jedem Funktionsaufruf einen Rückgabewert zu prüfen und darauf zu reagieren, kann die betreffende Funktion in Ausnahmesituationen eine →Ausnahme auslösen, die alle für die Problemerkennung und -behandlung erforderlichen Informationen in sich trägt.

Ausnahmepropagierung/exception propagation
Wenn in einer Routine eine Ausnahme ausgelöst, aber nicht behandelt wird, reicht die Laufzeitumgebung die →Ausnahme an die aufrufende Routine weiter. Dies bezeichnet man als "Ausnahmepropagierung".

Austrittsgrundblock/exit basic block
Auch als CFG-Exit bezeichnet.
Ein →Grundblock, der den normalen Austrittspunkt einer Routine darstellt. Wird eine Routine über diesen Grundblock verlassen, propagiert sie keine Ausnahmen. Jede Routine besitzt genau einen Austrittsgrundblock, auch dann, wenn dieser unerreichbar ist. Austrittsgrundblöcke sind Instanzen der Klasse `Exit_Basic_Block`.

Block/block
Ein Block ist eine Menge von aufeinanderfolgenden Deklarationen, Anweisungen usw., die in Java und C++ von geschweiften Klammern eingefasst sind. Beim Eintreten in einen Block werden dessen lokalen Objekte auf dem Stack angelegt und initialisiert. Beim Verlassen des Blocks werden diejenigen lokalen Objekte in umgekehrter Reihenfolge zerstört, die angelegt wurden (→stack unwinding).
Vorsicht: *Block* hat damit eine andere Bedeutung als →*Grundblock*!

-/CFG-Entry
Siehe Eintrittsgrundblock.

-/CFG-Exit
Siehe Austrittsgrundblock.

DNRP/definitely non-returning procedure
Eine Routine, die definitiv nicht zurückkehrt, wird als DNRP bezeichnet. Dieser Begriff stammt aus [Har98].

Eintrittsgrundblock/entry basic block
Auch als CFG-Entry bezeichnet.
Ein →Grundblock, der den Eintrittspunkt einer Routine darstellt. Jede Routine besitzt genau einen Eintrittsgrundblock. Eintrittsgrundblöcke sind Instanzen der Klasse `Entry_Basic_Block`.

EPN/exception predicate node
→Künstlicher →temporärer →Grundblock, der für die Berechnung der interprozeduralen →Kontrollabhängigkeit eingefügt wird. Ein EPN-Grundblock fungiert als Platzhalter für denjenigen Grundblock, der kontrolliert, ob die aufgerufene Routine eine Ausnahme propagiert oder nicht. Siehe Abschnitt 4.2.6.3.3, Seite 34.

Erweiterter Name/mangled name
Global eindeutiger Bezeichner eines Objekts, der auf dem im Quellcode verwendeten Bezeichner beruht und vom →Frontend generiert wird.

Explizite Ausnahmen/checked exception
Bei diesen →Ausnahmen stellt der Compiler sicher, dass sie nur dann von Methoden propagiert werden können, wenn dies in der Funktionsdeklaration angegeben ist. Die Unterscheidung zwischen *checked* und *unchecked exceptions* existiert nur zur Compile-Zeit, die →JVM kennt diese Unterscheidung nicht. *unchecked exceptions*

können nur explizit mit einer `throw`-Anweisung ausgelöst werden, sind also immer synchron und immer statisch erkennbar.

Frontend/front end (Übersetzerbau)
Subsystem zur syntaktischen und semantischen Analyse von Quelltexten.

Grundblock/basic block (BB)
Ein →Grundblock stellt eine Zusammenfassung unbedingt aufeinanderfolgender Anweisungen im →Kontrollflussgraphen dar. Grundblöcke abstrahieren von den einzelnen Anweisungen. Die Knoten des Kontrollflussgraphen der IML sind Grundblöcke. Es existieren verschiedene Typen, die verschiedene Bedeutungen haben.

IML/IML (bauhaus)
Abkürzung für Intermediate Language. Eine Zwischendarstellung für Quelltexte im Projekt bauhaus.

Instanziierung/instantiation (OO-Sprachen)
Erzeugung eines neuen Objektes einer Klasse.

JVM/java virtual machine
Die Java Virtual Machine (abgekürzt Java VM oder JVM) ist eine Laufzeitumgebung und virtuelle Maschine für Software, die den so genannten Java-Bytecode ausführt. Bytecode entsteht durch Compilierung von Programmen. Der Sinn der Java Virtual Machine und des Bytecodes ist es, plattformunabhängige Programme zu ermöglichen. Die JVM dient dabei als Schnittstelle zur Maschine und zum Betriebssystem.

Kontrollflussgraph (KFG)/control flow graph (CFG)
Als Kontrollflussgraph bezeichnet man einen Graphen, der den Ablauf eines Programms darstellt. Ein Knoten des Graphen ist entweder eine einzelne Anweisung, eine Gruppe von Anweisungen oder er stellt eine besondere semantische Entität dar, der keine Anweisungen zugeordnet sind. Die Kanten stellen den möglichen Ablauf dar.

Künstlicher Knoten/Grundblock)/-
Künstliche →Grundblöcke werden während der Analyse erzeugt. Sie stellen Konstrukte dar, die vom Quellcode abgeleitet sind. Im Kontext des →CFG werden Grundblöcke auch als Knoten bezeichnet.

Laufzeitausnahme/runtime exception
Bei Laufzeitausnhamen fordert der Compiler nicht, dass die →Ausnahme in der Definition der Routinen deklariert wird, die die Ausnahme werfen können. Ebenso prüft der Compiler bei Laufzeitausnhamen nicht, ob passende Handler existieren. In Java sind dies Ausnahmen, die von `RuntimeException` oder `Error` abgeleitet sind. In C++ prüft der Compiler prinzipiell keine Ausnahmen.

LOC/LOC
Abkürzung für *lines of code*. Größenmaß für Quelltexte. Bezeichnet die Anzahl an Quelltextzeilen (einschließlich Kommentaren und leeren Zeilen).

Methode/method (OO-Sprachen)
Auch als Nachricht bezeichnet (Smalltalk). Ausführbare Einheit, die zu einem Objekt oder einer Klasse gehört.

Objektorientierung (OO)/object orientation
Ein Paradigma bei der Softwareentwicklung. Traditionelle Programmiermethoden unterscheiden zwischen passiven Datenstrukturen und aktiven Funktionen. Bei der Objektorientierung werden Daten und Funktionen, die auf diesen Daten operieren, in Klassen zusammengefasst. Die Eigenschaften Kapselung, Polymorphie und Vererbung sind die wesentlichen Kriterien, die eine objektorientierte Programmiersprache ausmachen. Als erste konsequent objektorientierte Sprache gilt Smalltalk (1981 in Kalifornien bei Xerox entwickelt). Sie fußt auf Simula. Vollständig objektorientierte Programmiersprachen wie Smalltalk oder Eiffel sind eher selten anzutreffen, verbreitet sind heute objektorientierte Erweiterungen traditioneller Programmiersprachen (wie C++).

PNRP/possibly non-returning procedure
Eine Routine, die eventuell nicht zurückkehrt, wird als PNRP bezeichnet. Dieser Begriff stammt aus [Har98].

Primitiver Typ/primitive type
Primitive Typen sind Typen, die von der Programmiersprache vordefiniert sind. Sie werden auch als "eingebaute Typen" (engl. *built-in types*) bezeichnet. Je nach Programmiersprache ist der Wertebereich plattformabhängig (C, C++) oder fest vorgegeben (z.B. Ada, Java). In IML unterstützte primitive Typen sind: `void`, `bool`, `char`, `int`, `float` und `double`.

RPN/return predicate node
→Künstlicher →temporärer →Grundblock, der für die Berechnung der interprozeduralen →Kontrollabhängigkeit eingefügt wird. Ein RPN-Grundblock fungiert als Platzhalter für denjenigen Grundblock, der kontrolliert, ob die aufgerufene Routine zurückkehrt oder nicht. Siehe Abschnitt 4.2.6.3.2, Seite 34.

Signatur/signature (C++, Java)
Name und Typliste einer Routine. Die Typliste enthält die Typen der Parameter. In anderen Programmiersprachen kann auch der Rückgabetyp zur Signatur zählen (Ada).

Sprungmarke/label
Benannter Punkt im Quellcode, der Ziel einer Sprunganweisung sein kann.

Temporärer Knoten/Grundblock)/-
Temporäre →Grundblöcke werden während der Analyse erzeugt. Sie werden am Ende der Analyse wieder entfernt. Im Kontext des →CFG werden Grundblöcke auch als Knoten bezeichnet.

Transferanweisung/transfer statement
Eine Anweisung, die den Kontrollfluss an eine andere Anweisung als die direkt folgende lenkt. Beispiele: `break`, `goto`, `return` und `throw`.

Überwachter Block/guarded block
Der →Block in einer `try`-Anweisung, der auf Ausnahmen überwacht wird. Sofern ein passender →Handler in der `try`-Anweisung existiert, wird in diesem die Ausnahme behandelt. Falls ein `finally`-Block existiert, wird dieser bei Verlassen des überwachten Blocks ausgeführt.

ursprünglicher Name/unmangled name
Bezeichner eines Objekts, wie er im Quellcode verwendet wird. Ein solcher Bezeichner kann global mehrdeutig sein.

-/collective exit (CLX)
Jede Routine besitzt genau einen →künstlichen →temporären →Grundblock, der die Rolle des *collective exit* inne hat. Dieser Grundblock dient bei der Berechnung der Kontrollabhängigkeit als künstlicher Endknoten, in dem alle diejenigen Pfade des CFGs der Routine enden, über die die Routine verlassen werden kann (normal oder mit →propagierter Ausnahme). Ein solcher Knoten ist für die Berechnung der interprozeduralen Kontrollabhängigkeit mit Hilfe von Platzhaltern notwendig. Der einzige Nachfolgder des CLX ist der →LSX.

-/local super exit (LSX)
Jede Routine besitzt genau einen →künstlichen →temporären →Grundblock, der die Rolle des *local super exit* inne hat. Dieser Grundblock dient bei der Berechnung der Kontrollabhängigkeit als künstlicher Endknoten, in dem *alle* Pfade des CFGs der Routine enden. Für die Berechnung der Postdominanz ist ein solcher Knoten unverzichtbar. Der →CLX ist immer ein Vorgänger des LSX.

-/stack unwinding
Beschreibt den Prozess beim Verlassen eines →Blocks, bei dem die automatisch auf dem Stack angelegten Objekte zerstört werden. Die Zerstörung erfolgt in umgekehrter Reihenfolge wie die Erstellung. Da ein Block auch abrupt verlassen werden kann, ist nicht garantiert, dass alle innerhalb eines Blocks deklarierten lokalen Objekte vor dem Verlassen schon angelegt worden sind!

Literaturverzeichnis

[Aho86] Aho, A. V.; Sethi, R.; Ullman, J.D. *Compilers: Principles, Techniques und Tools*. Addision-Wesley, Reading, MA, USA, 1986.

[bas02] *Bash Reference Manual*. Free Software Foundation, Inc., 2002. `http://www.gnu.org/software/bash/manual/bashref.html`.

[Coo01] Cooper, Keith D.; Harvey, Timothy J.; Kennedy, Ken. A Simple, Fast Dominance Algorithm. Technical report, Rice University, Houston, Texas, USA, 2001.

[Dei02] Deininger, Marcus; Lichter, Horst; Prof. Dr. Ludewig, Jochen; Schneider, Kurt. *Studien-Arbeiten – ein Leitfaden zur Vorbereitung, Durchführung und Betreuung von Studien-, Diplom-, und Doktorarbeiten am Beispiel Informatik*. vdf Hochschulverlag AG an der ETH Zürich, 4. Auflage, Rielasingen, Schweiz, 2002.

[Ezi99] Ezick, James; Wagstaff, Kiri. Interprocedural Control Dependence. Seminarausarbeitung CS 612, Cornell University, Ithaca, NY, USA, Mai 1999. `http://www.cs.cornell.edu/Info/Courses/Spring-99/CS612/lecture%s/ezick-wagstaff.ppt`.

[Fro04] Frohn, Simon. Implementierung einer unifizierenden Zeigeranalyse mit gerichteten Zuweisen. Studienarbeit Nr. 1941, Universität Stuttgart, Institut für Softwaretechnologie, August 2004.

[Gam95] Gamma; Helm; Johnson; Vlissides. *Design Patterns: Elements of Reusable Object–Oriented Software*. Addison Wesley, 1995.

[Gos00] Gosling; Joy; Steele; Bracha. *The Java Language Specification*. Sun Microsystems, 2. Auflage, April 2000. `http://java.sun.com/docs/books/jls/second_edition/html/j.title%.doc.html`.

[Har98] Harrold, Mary Jean; Rothermel, Greg; Sinha, Saurabh. Computation of Interprocedural Control Dependence. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing und Analysis*, pages 11–20, März 1998.

[Inta] Internetseite. *Projekt Bauhaus. Software-Architektur, Software-Reengineering und Programmverstehen*. `http://www.bauhaus-stuttgart.de/`.

[Intb] Internetseite. *Richtlinien für Studien-, Diplomarbeiten und Fachstudien*. `http://www.informatik.uni-stuttgart.de/fakultaet/studienberatu%ng/richtlinien/richtlinien.html`.

[Int98] International Standard. *Programming Languages – C++*. Number ANSI/ISO/IEC 14882:1998(E) in ASC X3. American National Standards Institute, 1. Auflage, September 1998. ISO/IEC JTC 1.

[Int02] Internetseite. *Lex and Yacc primer / HOWTO*. `http://ds9a.nl/lex-yacc/`, 2002.

[Kar03] Karaca, Tahir; Setzer, Sebastian. Erweiterung und Generierung einer Zwischendarstellung IML für C++ Programme. Diplomarbeit Nr. 2048, Universität Stuttgart, Institut für Informatik, März 2003.

[Kna02] Knauß, Markus. Erweiterung und Generierung der Zwischendarstellung IML für Java-Programme. Diplomarbeit Nr. 2006, Universität Stuttgart, Institut für Informatik, Oktober 2002.

[Kos03] Koschke, Rainer; Plödereder, Erhard. Zwischendarstellungen und Programminformationen – Compiler-Technologie für das Reengineering. Technical report, Universität Stuttgart, 2003.

[Obj03] Object Management Group. *Unified Modelling Language Specifications, Version 1.5*. OMG, 2003. `http://www.omg.org/uml/`.

[Pur00] *100% Pure Java^{TM} Cookbook*. Sun Microsystems, 4. Revision, 2000. `http://java.sun.com/products/archive/100percent/4.1.1/100Perce%ntPureJavaCookbook-4_1_1.pdf`.

[Roh98] Rohrbach, Jürgen. Erweiterung und Generierung einer Zwischendarstellung für C-Programme. Studienarbeit Nr. 1662, Universität Stuttgart, Institut für Informatik, Januar 1998.

[Sin98] Sinha, Saurabh; Harrold, Mary Jean. Control-Flow Analysis of Programs with Exception-Handling Constructs. Technical Report OSU-CISRC-7/98-TR25, The Ohio State University, Columbus, OH, USA, Juli 1998.

[Sin00] Sinha, Saurabh; Harrold, Mary Jean; Rothermel, Gregg. Interprocedural Control Dependence. Technical Report OSU-CISRC-7/97-TR36, The Ohio State University, Columbus, OH, USA, Juni 2000.

[Str00] Stroustrup, Bjarne. *Die C++ Programmiersprache*. Addision-Wesley, Boston, USA, 2000.

[Wue96] Wuerthner, Martin. Entwurf und Implementierung einer Interndarstellung für die Analyse von Ada-Programmen. Studienarbeit Nr. 1567, Universität Stuttgart, Institut für Informatik, Oktober 1996.

Erklärung

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

Markus Müller