# Entwicklung eines Dienstmodells für das Konzept des Program Slicing

## Diplomarbeit

**zur Erlangung des Grades eines Diplom-Informatikers
im Studiengang Informatik**

Hannes Schwarz
hschwarz@uni-koblenz.de

Betreuer:
Prof. Dr. Jürgen Ebert
Dr. Andreas Winter

Institut für Softwaretechnik
Fachbereich Informatik
Universität Koblenz-Landau

10. August 2006

# Erklärung

Ich erkläre hiermit, dass ich diese Diplomarbeit selbständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemässe Zitate als solche gekennzeichnet habe.

Ich bin mit der Einstellung dieser Arbeit in die Bibliothek einverstanden.

Ich stimme der Veröffentlichung dieser Arbeit im Internet zu.

Nörtershausen, den 10. August 2006

. . . . . . . . . . . . . . . . . . . . . . . . . . .
Unterschrift

## Abstract

*Program slicing* is a technique for extracting that parts of a program which influence a previously defined, so-called *slicing criterion*. The latter mostly takes the form of a source code statement and a set of variables. Over the years an abundance of variants and enhancements has evolved from the original notion of a program slice. One of these developments is called *chopping*. A chop only contains those statements which are necessary to sustain the effects of a certain statement on another statement. Corresponding to the slicing criterion, the two statements have to be available in advance.

This diploma thesis deals with the development of a service model in order to support the computation of a slice or a chop, given an original program and a slicing or chopping criterion respectively. A service model is a framework of communicating services so that each service performs a specific task within the concept of program slicing. The three main tasks, without considering further decomposition, are the mapping of program code into a representation suitable for slicing, the slicing itself and the display of the slice as code.

The key benefit of service-orientation is that a service encapsulates the underlying algorithms. Hence the possibility of improving or substituting them without changing the interfaces of the related service relieves the developer of the need of adapting adjoining services. Thus, service-orientation fosters maintainability and improvability.

Besides the definition of the services, this thesis also partially formally defines the data flow between them, i.e. their interfaces. It also features graph class definitions for most data structures. An accurate description of the interfaces encourages reuse, provided the services are of adequate granularity.

# Inhaltsverzeichnis


**Abbildungsverzeichnis** xi

**1. Einleitung** 1

1.1. Motivation und Gegenstand der Arbeit . . . 1
1.2. Aufbau der Arbeit . . . 2

**2. Einführung in das Konzept des Program Slicing** 3

2.1. Arten und Verfahren des Program Slicing . . . 3
2.1.1. Statisches Slicing . . . 4
2.1.2. Dynamisches Slicing . . . 12
2.1.3. Mischformen und verwandte Konzepte . . . 13
2.2. Anwendungsbeispiele des Program Slicing . . . 14
2.2.1. Debugging . . . 14
2.2.2. Programmdifferenzierung und -integration . . . 14
2.2.3. Softwarewartung und Programmverstehen . . . 15
2.3. Existierende Slicing-Werkzeuge . . . 16
2.3.1. CodeSurfer . . . 16
2.3.2. Indus . . . 16

**3. Grundlagen des Modells der Slicing-Dienste** 19

3.1. Auswahl der in dem Modell einbezogenen Verfahren . . . 19
3.1.1. Statisches Slicing und spezielle Programmeigenschaften . . . 19
3.1.2. Dynamisches Slicing . . . 21
3.1.3. Chopping . . . 21
3.2. Definition des Begriffs „*Dienst*“ . . . 22
3.3. Beispielsprache *EL* . . . 22

**4. Datenfluss der Slicing-Dienste** 27

4.1. Der Dienst *ProgramSlicing* . . . 27
4.2. Der Dienst *preprocessProgram* . . . 28
4.3. Der Dienst *sliceProgram* . . . 32
4.4. Der Dienst *convertESDGToCode* . . . 34
4.5. Überblick . . . 35

**5. Datenmodell** **37**
5.1. Hinweise zur Notation . . . 37
5.2. Abstrakter Syntaxgraph (*ASG*) . . . 38
5.3. Erweiterter Kontrollflussgraph (*ACFG*) . . . 39
5.4. Points-to-Graph (*PG*) . . . 52
5.5. Aufrufgraph/erweiterter Aufrufgraph (*CG*/*ECG*) . . . 53
5.6. Erweiterter Systemabhängigkeitsgraph (*ESDG*) . . . 54
5.6.1. Grundelemente des erweiterten Systemabhängigkeitsgraphen . . . 56
5.6.2. Interprozedurale Kanten . . . 61
5.6.3. Kontrollkanten . . . 64
5.6.4. Datenfluss- und Deklarationskanten . . . 68
5.6.5. Transitive Kanten . . . 74

**6. Spezifikationen der Dienste** **77**
6.1. *preprocessProgram* . . . 77
6.1.1. *computeAbstractSyntaxGraph* . . . 78
6.1.2. *computeAugmentedControlFlowGraphs* . . . 78
6.1.3. *computePointsToGraphs* . . . 79
6.1.4. *computeExtendedCallGraph* . . . 79
6.1.4.1. *computeCallGraph* . . . 80
6.1.4.2. *computeDefUseInformation* . . . 80
6.1.5. *computeExtendedSystemDependenceGraph* . . . 80
6.1.5.1. *computeBasicESDG* . . . 81
6.1.5.2. *computeInterMethodEdges* . . . 82
6.1.5.3. *computeControlDependenceEdges* . . . 82
6.1.5.4. *computeDataFlowEdges* . . . 82
6.1.5.5. *computeSummaryEdges* . . . 83
6.2. *sliceProgram* . . . 83
6.2.1. *markESDG* . . . 84
6.2.1.1. *markSlicingCriterion* . . . 85
6.2.1.2. *markChoppingCriterion* . . . 87
6.2.1.3. *markTrace* . . . 88
6.2.2. *computeStaticSlice* . . . 89
6.2.2.1. *computeStaticBackwardSlice* . . . 89
6.2.2.2. *computeStaticForwardSlice* . . . 90
6.2.3. *computeDynamicSlice* . . . 92
6.2.3.1. *computeDynamicBackwardSlice* . . . 94
6.2.3.2. *computeDynamicForwardSlice* . . . 95
6.2.4. *computeStaticChop* . . . 96
6.2.5. *computeDynamicChop* . . . 97
6.2.6. *computeExecutableBackwardSlice* . . . 100

6.3. *convertEDSGToCode* . . . . . . . . . . . . . . . . . . . . . . . . . . . . 101

**7. Zusammenfassung und Ausblick** **103**

**A. CD-ROM** **107**

**Literaturverzeichnis** **113**

# Abbildungsverzeichnis


2.1. Beispiel einer Slice nach Weiser . . . . . 5
2.2. Beispiel einer Slice nach Horwitz et al. . . . . . 9
2.3. CodeSurfer-Screenshot . . . . . 17
2.4. Kaveri-Screenshot . . . . . 17

3.1. Programm in der Beispielsprache EL . . . . . 23
3.2. Grammatik der Beispielsprache EL . . . . . 25

4.1. Kontextdiagramm . . . . . 27
4.2. Datenflussdiagramm 0 – *ProgramSlicing* . . . . . 28
4.3. Datenflussdiagramm 1 – *preprocessProgram* . . . . . 29
4.4. Datenflussdiagramm 1.4 – *computeExtendedCallGraph* . . . . . 31
4.5. Datenflussdiagramm 1.5 – *computeExtendedSystemDependenceGraph* . . 31
4.6. Datenflussdiagramm 2 – *sliceProgram* . . . . . 32
4.7. Datenflussdiagramm 2.1 – *markESDG* . . . . . 34
4.8. Datenflussdiagramm 2.2 – *computeStaticSlice* . . . . . 35
4.9. Datenflussdiagramm 2.3 – *computeDynamicSlice* . . . . . 35
4.10. Überblick über die Slicing-Dienste samt Ein- und Ausgaben . . . . . 36

5.1. Abstrakter Syntaxgraph des Beispielprogramms . . . . . 40
5.2. Schema des ASG . . . . . 41
5.3. Erweiterter Kontrollflussgraph der Methode *m1* des Beispielprogramms . . 42
5.4. Schema der ACFG-Menge . . . . . 43
5.5. Übersicht über die zulässigen Kontrollflusskanten . . . . . 46
5.6. Schema der PG-Menge . . . . . 52
5.7. Schema des ECG . . . . . 54
5.8. Schema des ESDG . . . . . 56
5.9. Auszug aus dem erweiterten Systemabhängigkeitsgraphen des Beispielprogramms . . . . . 57
5.10. Aufruf der Methode *pm(int x)* in Methode *m2()* des Beispielprogramms . . 62
5.11. Aufruf der Methode *m4(int *q)* in Methode *m2()* des Beispielprogramms . . 63
5.12. Kontrollkanten der Methode *m1* des Beispielprogramms . . . . . 65
5.13. ACFG der Methode *m1* des Beispielprogramms nd die daraus berechneten Kontrollkanten . . . . . 67

5.14. Übersicht über verschiedene Anweisungen des Beispielprogramms und den von diesen definierten oder benutzten Variablen . . . . . . . . . . . . . . . . 68
5.15. Datenfluss- und Deklarationskanten der Methode *m1* des Beispielprogramms 69

6.1. Beispiel einer statischen Rückwärtsslice . . . . . . . . . . . . . . . . . . . . 91
6.2. Beispiel einer statischen Vorwärtsslice . . . . . . . . . . . . . . . . . . . . . 93
6.3. Beispiel eines statischen Chops . . . . . . . . . . . . . . . . . . . . . . . . . 98

7.1. Überblick über die Funktionen der Slicing-Dienste . . . . . . . . . . . . . . 104
7.2. Überblick über die Graphen des Modells . . . . . . . . . . . . . . . . . . . . 105

# 1. Einleitung

## 1.1. Motivation und Gegenstand der Arbeit

Die vorliegende Arbeit befasst sich mit dem Konzept des *Program Slicing*. Program Slicing dient dazu, einen Ausschnitt eines Programms zu bestimmen, welcher nur diejenigen Programmanweisungen enthält, die eine Wirkung auf ein vorher zu spezifizierendes *Slicing-Kriterium* besitzen. Letzteres besteht in der ursprünglichen Ausprägung des Konzeptes aus einer Anweisung und einer Variablenmenge.

Die bisherige Forschung konzentrierte sich vor allem auf die Entwicklung von, meist graphenbasierten, Darstellungen eines Programms sowie zugehöriger Algorithmen, die, auf diesen Darstellungen aufsetzend, das eigentliche Slicing durchführen. Im Laufe der Zeit wurden die Algorithmen verbessert und spezielle Programmkonstrukte und -eigenschaften, wie etwa Sprunganweisungen oder Objektorientierung, in die Darstellungen mit aufgenommen. Zwar befassen sich eine Reihe von Arbeiten mit einem Überblick oder Vergleich der diversen Verfahren, jedoch wurde deren Integration in ein gemeinsames Modell bislang vernachlässigt. So existiert für verschiedene Gesichtspunkte des Program Slicing eine Vielzahl an Konzepten nebeneinander, bei denen nicht klar ist, wie, oder gar ob, sie sich miteinander kombinieren lassen.

Das Ziel dieser Arbeit ist die Entwicklung und Spezifikation eines Dienstmodells zur Unterstützung des Program Slicing, welches sich als Grundlage für die Implementation eines Softwarewerkzeuges eignen soll. Unter einem Dienstmodell ist eine Sammlung interagierender Softwarekomponenten zu verstehen, die mit ihrer Funktionalität jeweils einen abgegrenzten Teilaspekt des gesamten Slicingkonzeptes bedienen.

Im Vergleich zu monolithischen Systemen liegt der Vorteil eines derartigen Modells in der Vereinfachung der Erweiterung um zusätzliche Funktionalitäten: Zum einen sind die zu modifizierenden Programmkomponenten, also die Dienste, besser zu lokalisieren und zum anderen sind Seiteneffekte auf andere Dienste auszuschließen, solange die Schnittstellen beibehalten werden. Zweiterer Aspekt ergibt sich aus der Abstraktion von dem „Innenleben“ der Dienste, d.h. den Algorithmen, mit deren Hilfe aus den Eingaben die Ausgaben berechnet werden. Bei einer Änderung der Schnittstellen allerdings sind in Beziehung stehende Dienste gegebenenfalls anzupassen.

Ein weiterer Nutzen besteht in der Wiederverwendbarkeit einzelner Dienste. Aufgrund der definierten Ein- und Ausgabeschnittstellen wird eine Entscheidung über die Integration in ein anderes System erleichtert. Jedoch spielen hier auch andere Faktoren wie etwa die Granularität und Änderbarkeit der Dienste eine Rolle.

Zu der Erstellung des Dienstmodells ist es insbesondere notwendig, die Schnittstellen der Dienste exakt zu beschreiben. Dies geschieht, indem die zwischen den Diensten fließenden Daten als Graphenstrukturen verstanden und durch entsprechende Graphenschemata spezifiziert werden.

## 1.2. Aufbau der Arbeit

Der Aufbau dieser Arbeit orientiert sich an einer Datenflussbeschreibung der Slicing-Dienste. Nach dieser Einleitung folgt jedoch zunächst eine Einführung in das Konzept und die Verfahrensweisen des Program Slicing. Sie geht auch auf mögliche Anwendungen ein und stellt bereits existierende Werkzeuge kurz vor. Das dritte Kapitel bereitet die Modellierung der Dienste vor, indem es zum einen grundlegende Entscheidungen im Vorfeld der Modellierung darlegt, etwa welche der im vorhergehenden Kapitel erläuterten Verfahren Einzug in das Modell gefunden haben. Zum anderen beschreibt es die Programmiersprache, anhand derer das Modell erstellt wurde. Außerdem wird der Begriff des *Dienstes* definiert.

Die Kapitel vier bis sechs, der Hauptteil dieser Arbeit, behandeln das eigentliche Modell. Das vierte Kapitel führt die verschiedenen Dienste ein und stellt deren Interaktion anhand einer Datenflussbeschreibung dar. Aus dieser gehen auch die Ein- und Ausgabeschnittstellen sowie die Zerlegung in Teildienste hervor. Kapitel fünf hat das Datenmodell, d.h. die im vorangegangenen Abschnitt erwähnten Graphenstrukturen zum Thema. Das letzte Hauptkapitel betrachtet die Dienste anhand der Definition aus dem dritten Kapitel und geht dabei detailliert auf die Funktionalitäten ein.

Zum Abschluss erfolgt eine Zusammenfassung der Arbeit. Es wird auch versucht, Hinweise darauf zu geben, an welchen Stellen eine Weiterentwicklung des Modells anknüpfen könnte.

# 2. Einführung in das Konzept des Program Slicing

Dieses Kapitel verschafft einen Überblick über das Konzept des *Program Slicing* in seinen Grundzügen. Um das für das Verständnis des Dienstmodells notwendige Wissen zu vermitteln, wird auf dafür relevante Aspekte besonders detailliert eingegangen. Auch werden Anwendungsmöglichkeiten des Slicing aufgezeigt und bereits existierende Werkzeuge kurz beschrieben.

## 2.1. Arten und Verfahren des Program Slicing

Das Konzept des Program Slicing, zu deutsch wörtlich „ein Programm in Scheiben schneiden", wurde 1979 von Weiser in seiner Dissertation [Wei79] eingeführt. Seitdem wurde die Forschung auf diesem Gebiet weiter vorangetrieben: sei es die Verbesserung der Slicing-Algorithmen, die Untersuchung verschiedener Anwendungsmöglichkeiten oder auch die Entwicklung neuer, sich von Weisers erstem Konzept unterscheidenden Ausprägungen des Program Slicing.

Wie bereits im ersten Kapitel erwähnt, handelt es sich bei Program Slicing grob umrissen um eine Technik, ein Programm auf diejenigen Bestandteile zu reduzieren, welche in einem Abhängigkeitsverhältnis zu dem so genannten *Slicing-Kriterium* stehen. Die Gestalt dieses Kriteriums kann je nach betrachteter Ausprägung des Slicing differieren, involviert jedoch ursprünglich eine Programmanweisung und eine Teilmenge der im Programm verwendeten Variablen.

Neben Program Slicing existieren noch andere Arten des Slicing, beispielsweise *Specification Slicing* oder *Architectural Slicing*[1]. Ersteres ermittelt diejenigen Elemente einer Spezifikation, welche eine Wirkung auf den Wert einer bestimmten Variablen besitzen (siehe [Cha+94]). Architectural Slicing arbeitet auf einer höheren Abstraktionsebene: Es werden die eine Programmkomponente beeinflussenden Programmteile bestimmt (siehe [Zha98a]).

Die Strukturierung der folgenden Ausführungen wird anhand des ursprünglichen *statischen Slicing*, des *dynamischen Slicing*, daraus entstandenen Mischformen sowie verwandten Kon-

[1]Bei Verwendung des Begriffs *Slicing* ist in dieser Arbeit ausschließlich Program Slicing gemeint.

zepten vorgenommen. Weitere Möglichkeiten der Abgrenzung wären etwa *intraprozedurales* und *interprozedurales* Slicing oder die Berechnung *ausführbarer* und *nicht-ausführbarer* Slices. Auf diese und verschiedene andere Formen des Program Slicing wird in den nächsten Abschnitten ebenfalls eingegangen.

### 2.1.1. Statisches Slicing

Unter statischem Slicing versteht man Slicing anhand des Programmcodes ohne jegliche Laufzeitinformationen. Die nächsten Abschnitte beschreiben verschiedene Slicing-Verfahren und gehen auf spezielle Programmelemente und -eigenschaften ein, die einer Sonderbehandlung bedürfen: Sprunganweisungen, Zeiger sowie Objektorientierung.

#### Das Verfahren von Weiser

Statisches Slicing als erste Slicing-Ausprägung wurde von Weiser 1979 eingeführt. In [Wei84] beschreibt er ein Verfahren, Slices für ein Slicing-Kriterium $SliceCrit =< i, X >$ zu bestimmen. Dabei steht $i$ für eine Anweisung im Quellcode des Programms und $X$ für eine Menge von Variablen. Eine Slice $S$ ist nach Weiser ein ausführbares Programm, das aus der Streichung von Anweisungen des Ursprungsprogramms $P$ hervorgeht. Terminiert $P$, so waren die Werte der Variablen $x \in X$ bei jeder korrespondierenden Ausführung von $i$ in $S$ und $P$ identisch.

Weisers Idee beruht auf dem Kontrollflussgraphen (Abk. CFG, für engl. *control flow graph*), dessen Knoten die Programmanweisungen darstellen, zuzüglich eines *Entry*- und eines *Exit*-Knotens für Prozedureintritt bzw. -austritt. Eine Kontrollflusskante verläuft von Knoten $s$ zu Knoten $t$, falls $t$ unmittelbar nach $s$ ausgeführt werden kann[2]. Hängt die Ausführung von $t$ von einer Ausdrucksauswertung in $s$ ab, so ist die Kante abhängig von dem notwendigen Ergebnis der Auswertung entweder mit *True* oder mit *False* beschriftet.

Der CFG wird, von der Anweisung $i$ ausgehend, rückwärts traversiert. Für jeden Knoten des CFG werden zunächst „relevante" Variablen ermittelt. Unter Zuhilfenahme dieser Relevanzmengen erfolgt in einem zweiten Schritt die Bestimmung der in der Slice enthaltenen Anweisungen. Daraus folgt, dass mit Weisers Ansatz *rückwärtsgerichtete Slices* (auch: *Rückwärtsslices*) berechnet werden, d.h. es werden diejenigen Anweisungen gestrichen, die für die Werte der Variablen in $X$ irrelevant sind. Im Gegensatz dazu enthalten *vorwärtsgerichtete Slices* (auch: *Vorwärtsslices*) alle Anweisungen, deren Ausführung vom Slicing-Kriterium abhängig ist sowie alle Anweisungen, die Variablen verwenden, deren Werte vom Slicing-Kriterium abhängig sind (siehe [Tip95, S. 3]).

[2] Hier und im Folgenden werden ein Programmelement, in diesem Fall eine Anweisung, und der dieses Element in einem Graphen darstellende Knoten synonym verwendet, um eine sprachliche Vereinfachung zu erzielen.

Der von Weiser vorgestellte Algorithmus ist sowohl intraprozedural, als auch bei Programmen mit Prozeduren anwendbar. Im letzteren Fall könnten die berechneten Slices aufgrund der *Kontextinsensitivität* jedoch größer sein als notwendig. Informal ausgedrückt bedeutet kontextinsensitiv, dass der Algorithmus sich nicht „merkt", aus welcher übergeordneten Prozedur eine Prozedur $P$ aufgerufen wird. Bei Beendigung der Prozedur $P$ wird angenommen, der Kontrollfluss könne zu allen Prozeduren übergehen, die einen Aufruf von $P$ enthalten.

Abbildung 2.1 enthält ein Beispiel eines nach Weisers Verfahren geslicten Programms[3] für das Slicing-Kriterium $< 8, b >$. Es ist festzustellen, dass Zeile 5, die Definition der Variable $c$, nicht mehr in der Slice vorhanden ist, weil $c$ bei der Berechnung von $b$ keine Rolle spielt. Obwohl aber der Wert von $a$ gleichermaßen irrelevant sein sollte, ist auch Zeile 3 in der Slice vorzufinden. Dies lässt sich durch die Kontextsensitivität des Algorithmus erklären: Zwar wird erkannt, dass die Prozeduren *inc* und *add* in der Slice enthalten sein müssen, jedoch werden sowohl der Aufruf in Zeile 15 als auch derjenige in Zeile 6 der Slice hinzugefügt. Da letzterer die Variable $a$ benutzt, ist auch deren Definition mit einzubeziehen.

```
 1  int main()                  1  int main()
 2  {                           2  {
 3    a = 0                     3    a = 0
 4    b = 1                     4    b = 1
 5    c = 2
 6    add(a, b)                 6    add(a, b)
 7    inc(b)                    7    inc(b)
 8  }                           8  }

 9  void add(int x, int y)      9  void add(int x, int y)
10  {                          10  {
11    x = x + y                11    x = x + y
12  }                          12  }

13  void inc(int z)            13  void inc(int z)
14  {                          14  {
15    add(z, 1)                15    add(z, 1)
16  }                          16  }
```

Abbildung 2.1.: Beispiel eines mit Hilfe von Weisers Verfahren geslicten Programms: links das Originalprogramm, rechts die Slice für das Slicing-Kriterium $< 8, b >$. In Anlehnung an [Hor⁺90, S. 3]

[3] Es wird *call-by-reference* als Parameterübergabemechanismus vorausgesetzt.

### Das Verfahren von Ottenstein und Ottenstein

Ein anderer Ansatz zur Berechnung von intraprozeduralen und rückwärtsgerichteten Slices wurde von Ottenstein und Ottenstein [Ott+84] verfolgt. Sie zeigten, dass sich Slices durch Traversieren des Programmabhängigkeitsgraphen (Abk. PDG, für engl. *program dependence graph*), der Vereinigung von Kontroll- und Datenflussgraph, berechnen lassen. Ein Slicing-Kriterium besteht hier aus einem Knoten des PDG. Alle Knoten, von denen aus das Slicing-Kriterium erreichbar ist, sind in der Slice enthalten.

Sowohl im Kontroll- als auch im Datenflussgraphen stellen die Knoten die Programmanweisungen dar. Hinzu kommt ein spezieller *Entry*-Knoten. Eine Kante des Kontrollgraphen (*Kontrollkante*) verläuft von Knoten $s$ zu Knoten $t$, falls $s$ die Ausführung von $t$ beeinflussen kann (für eine Definition siehe [Fer+87, S. 322f]). Dies ist z.B. der Fall, wenn eine Ausdrucksauswertung in $s$ über die Ausführung von $t$ entscheidet. In strukturierten Programmiersprachen ohne jedwede Art von Sprunganweisungen spiegelt der Kontrollgraph die Schachtelung der Programmanweisungen wider. Alle Anweisungen der äußersten Schachtelungsebene einer Prozedur sind über eine Kontrollkante mit dem Entry-Knoten verbunden.

Eine Datenflusskante existiert zwischen zwei Knoten $s$ und $t$, falls von $s$ eine Variable definiert wird, die anschließend von $t$ genutzt werden kann, ohne dass es zwischen den Ausführungen von $s$ und $t$ zwingend zu einer Redefinition der Variable kommt. In Kapitel 5 erfolgt eine auf das in dieser Arbeit vorgestellte Modell bezogene, exakte Definition der Kanten des Kontroll- und des Datenflussgraphen. Beide lassen sich auf der Grundlage des Kontrollflussgraphen berechnen.

Im Vergleich zu Weisers Verfahren ist der Ansatz der Ottensteins aufgrund der Tatsache, dass das Slicing-Kriterium keine Angabe einer Variablenmenge erlaubt, ungenauer. Implizit wird nämlich angenommen, dass $X$ der Menge aller von $i$ benutzten oder definierten Variablen entspricht. Mittels der Modifikation aus [Tip95, S. 27] lässt sich dieser Umstand jedoch beseitigen: Als Slicing-Kriterium wird die Menge all jener Knoten verwendet, welche eine Variable aus $X$ definieren und deren Definitionen $i$ erreichen, d.h. dass die Variable auf mindestens einem Kontrollflusspfad nicht redefiniert wird. Dieser Sachverhalt ähnelt dem Konzept der Datenflusskanten, mit dem Unterschied, dass $x$ von $i$ nicht benutzt zu werden braucht.

### Das Verfahren von Horwitz, Reps und Binkley

Horwitz, Reps und Binkley [Hor+90] erweiterten den Programmabhängigkeitsgraphen von Ottenstein und Ottenstein zu einem Systemabhängigkeitsgraphen (Abk. SDG, für engl. *system dependence graph*), der die Darstellung von Programmen mit mehreren Prozeduren ermöglicht. Weiterhin entwickelten sie einen Algorithmus, der anhand des SDG interprozedurale, kontextsensitive und wahlweise rückwärts- oder vorwärtsgerichtete Slices berechnet.

Der Systemabhängigkeitsgraph besteht aus den PDGs der einzelnen Prozeduren. Jede eine Prozedur aufrufende Programmanweisung verfügt über eine Aufrufkante (engl. *call edge*) zu dem Entry-Knoten der aufgerufenen Prozedur.

Zur Modellierung des Parameterübergabemechanismus existieren so genannte *Parameterknoten*, wobei zwischen *Actual-in-* und *Actual-out-*Knoten auf der einen Seite sowie *Formal-in-* und *Formal-out-*Knoten auf der anderen Seite zu unterscheiden ist. Erstere sind über Kontrollkanten mit aufrufenden Anweisungen verbunden, während letztere, ebenfalls über Kontrollkanten, den Entry-Knoten zugeordnet werden. Parameterknoten realisieren die Parameterübergabe bei Prozeduraufrufen: Ein Actual-in-Knoten weist einen aktuellen Parameter einer Hilfsvariable zu. Der Wert dieser Hilfsvariable wird von einem Formal-in-Knoten für die Definition des entsprechenden formalen Eingabeparameters genutzt. Die Beziehung zwischen den Knoten wird über *Parameter-in-*Kanten von Actual-in- zu dem korrespondierenden Formal-in-Knoten hergestellt. Bei mehreren Aufrufen derselben Prozedur können mehrere Actual-in-Knoten mit demselben Formal-in-Knoten verbunden sein. In diesem Fall müssen alle Actual-in-Knoten dieselbe Hilfsvariable verwenden.

Analog verhält es sich mit Actual-out- und Formal-out-Knoten und den Ausgabewerten einer Prozedur. Allerdings sind die Knoten über *Parameter-out-*Kanten verknüpft, die von dem Formal-out- zu den Actual-out-Knoten ausgehen. Definiert oder benutzt eine von einer Anweisung aufgerufene Prozedur Variablen, so sind die Datenflusskanten mit den Actual-in- oder Actual-out-Knoten anstatt mit der Anweisung selbst verbunden. Auch an Formal-in- und Formal-out-Knoten können Datenflusskanten anknüpfen: Von einem eine Variable definierenden Formal-in-Knoten führen Datenflusskanten zu allen Knoten, die die Variable verwenden, ohne dass es dazwischen unbedingt zu einer Redefinition kommt. Wird eine Variable von einer Anweisung definiert und ohne vorherige zwingende Redefinition von einem Formal-out-Knoten benutzt, existiert zwischen den beiden Knoten ebenfalls eine Datenflusskante.

Weiterhin verfügt der SDG über transitive Kanten (engl. *summary edge*), die von Actual-in- zu Actual-out-Knoten verlaufen können. Eine solche Kante bedeutet, dass der Eingabeparameter den Wert des Ausgabeparameters beeinflussen kann. Einige weiterführende Arbeiten wie z.B. [Rep+94] oder [For+97] beschäftigen sich damit, effiziente Algorithmen zur Berechnung transitiver Kanten zu entwickeln. Das Verfahren von Reps et al. in [Rep+94] sieht vor, eine transitive Kante zwischen einem Actual-in- und einem Actual-out-Knoten zu konstruieren, wenn ein Pfad, bestehend aus Kanten des SDG, zwischen den beiden Knoten existiert. Einschränkend dabei ist, dass genau dann, wenn über eine im Pfad enthaltene Parameter-out-Kante von einer Methode zu deren Aufruf „aufgestiegen“ wird, der „Abstieg“ über Parameter-in- oder Aufrufkanten erfolgt sein muss, die an denselben Aufruf anknüpfen müssen wie die Parameter-out-Kante.

Die eigentlichen Berechnung der Slices dient ein Zwei-Phasen-Algorithmus. In der ersten Phase werden von den in dem Slicing-Kriterium enthaltenen Knoten aus Kontroll-, Datenfluss-, Parameter-in- und Aufrufkanten sowie transitive Kanten rückwärts traversiert.

Alle so erreichbaren Knoten sind in der Rückwärtsslice enthalten. In der zweiten Phase kommen diejenigen Knoten hinzu, welche von den bereits erreichten Knoten aus über das rückwärtige Traversieren von Kontroll-, Datenfluss- und Parameter-out-Kanten sowie transitiven Kanten erreichbar sind. Ohne transitive Kanten würde eine Berechnungsvorschrift zur Berechnung von Slices im SDG über einen reinen Graphentraversierungsalgorithmus hinausgehen. Sie müsste sich beim „Abstieg" in Prozeduren die Aufrufstelle „merken", um beim „Aufstieg" wieder zur selben zurückzukehren und so die Kontextsensitivität zu wahren.

Zur Ermittlung von Vorwärtsslices müssen zunächst Kontroll-, Datenfluss- und Parameter-out-Kanten sowie transitive Kanten vorwärts traversiert werden. Von den erreichten, in die Slice aufzunehmenden Knoten aus werden wiederum Kontroll-, Datenfluss-, Parameter-in- und Aufrufkanten sowie transitive Kanten vorwärts traversiert und die so erreichbaren Knoten der Slice hinzugefügt.

Die von dem in diesem Abschnitt beschriebenen Verfahren bestimmten Slices sind, im Gegensatz zu denjenigen, die mit den Verfahren von Weiser oder den Ottensteins berechnet wurden, nicht unbedingt ausführbar. Dies resultiert aus dem möglichen Wegfall von Parametern bei Prozeduraufrufen, falls zugehörige Parameterknoten in der Slice fehlen. Binkley präsentiert in [Bin93] eine Vorgehensweise, um ausführbare Slices zu produzieren: Es werden zusätzliche Actual-in- und Actual-out-Knoten für Prozeduraufrufe erzeugt, die nicht für jeden Formal-in- oder Formal-out-Knoten einen entsprechenden Actual-in- bzw. Actual-out-Knoten aufweisen können. Wird ein neuer Actual-in-Knoten eingefügt, so müssen für alle von diesem benutzten Variablen auch diejenigen Knoten in der Slice enthalten sein, welche für die Bestimmung der Variablenwerte verantwortlich sind.

Nach der Erstellung eines Systemabhängigkeitsgraphen können Slices mit linearem Zeitaufwand extrahiert werden. Daher eignet sich das Verfahren von Horwitz et al. insbesondere dann, wenn viele Slices desselben Programms zu berechnen sind. Weisers Ansatz hingegen ist ungleich aufwendiger. (siehe [Tip95, S. 29f])

Abbildung 2.2 zeigt das bereits aus Abbildung 2.1 bekannte Beispielprogramm. Zur Berechnung der Slice findet hier aber das Verfahren von Horwitz et al. Anwendung. Wie sich erkennen lässt, ist die ermittelte Slice genauer: Zeilen 3 und 6 sind nicht mehr enthalten.

### Slicing unstrukturierter Programme

Mit den oben vorgestellten Vorgehensweisen in unmodifizierter Form können nur strukturierte Programme ohne jegliche Sprunganweisungen bearbeitet werden. Enthält der Quellcode z.B. *goto*-, *break*-, *continue*-Anweisungen oder auch *try-catch*-Blöcke, können die ermittelten Slices fehlerhaft sein.

```
 1  int main()
 2  {
 3    a = 0
 4    b = 1
 5    c = 2
 6    add(a, b)
 7    inc(b)
 8  }

 9  void add(int x, int y)
10  {
11    x = x + y
12  }

13  void inc(int z)
14  {
15    add(z, 1)
16  }
```

```
 1  int main()
 2  {

 4    b = 1


 7    inc(b)
 8  }

 9  void add(int x, int y)
10  {
11    x = x + y
12  }

13  void inc(int z)
14  {
15    add(z, 1)
16  }
```

Abbildung 2.2.: Beispiel eines mit Hilfe des Verfahrens von Horwitz et al. geslicten Programms: links das Originalprogramm, rechts die Slice für das Slicing-Kriterium $< 8, b >$. In Anlehnung an [Hor$^{+}$90, S. 3]

Mit Bezug auf Weisers Verfahren machte Lyle [Lyl84] den Vorschlag, sämtliche goto-Anweisungen in die Slice einzubinden, die eine nicht-leere Relevanzmenge aufweisen. Dies führt jedoch oft zu zu großen Slices.

Die Einbindung von intraprozeduralen Sprunganweisungen[4] in den Programm- bzw. Systemabhängigkeitsgraphen wurde unter anderem von Choi und Ferrante [Cho$^{+}$94] untersucht, während Agrawal in [Agr94] vom SDG unabhängige Datenstrukturen verwendete, um die für die Slice relevanten Sprünge zu finden. Des Weiteren entwickelte er einen vereinfachten Algorithmus für Programme, die ausschließlich strukturierte Sprünge (also keine goto-Anweisungen) enthalten. Harman und Danicic steigerten in [Har$^{+}$98] die Genauigkeit von Agrawals Verfahren. Sinha et al. zeigen in [Sin$^{+}$99], wie Programme mit prozedurübergreifenden Sprüngen (z.B. mit Hilfe der C-Anweisungen *setjmp*, *longjmp* oder *try-catch*-Blöcke) geslict werden können.

Im Folgenden wird die Idee von Ball und Horwitz [Bal$^{+}$93] vorgestellt, die wie Choi und Ferrante Sprünge durch eine Ergänzung des Kontrollflussgraphen um zusätzliche Kanten repräsentieren. Sprunganweisungen werden ähnlich wie Verzweigungen behandelt: Eine mit *True* beschriftete Kante verläuft zu dem Sprungziel, während eine *False*-Kante in diejeni-

[4]Das Sprungziel liegt in derselben Prozedur wie die Sprunganweisung.

ge Anweisung eingeht, welche bei Nichtausführung des Sprunges ausgeführt würde. Aus dem Entry-Knoten gehen ebenfalls zwei beschriftete Kanten aus: zu der ersten Anweisung der Prozedur sowie zu dem Exit-Knoten. Erstere ist mit *True*, die andere mit *False* gekennzeichnet. Dieser erweiterte Kontrollflussgraph wurde von Ball und Horwitz *augmented control flow graph* genannt, hier mit ACFG abgekürzt. Aufgrund der zusätzlichen Kanten des ACFG ergeben sich bei der Berechnung des PDG/SDG weitere, aus den Sprunganweisungen ausgehende Kontrollkanten.

### Die Behandlung zusammengesetzter Datentypen und Zeiger

Die Behandlung von zusammengesetzten Datentypen wie Strukturen und Arrays kann vergleichsweise einfach sein. Strukturen müssen lediglich in ihre Elemente aufgespalten werden (siehe [Bin$^+$96, S. 10]) und bei Arrays besteht prinzipiell die Möglichkeit, sie als Ganzes zu betrachten und so ungenauere, d.h. zu große Slices[5] in Kauf zu nehmen. Genauere Lösungen werden in der Literatur (siehe z.B. [Agr$^+$91, S. 10ff]) erläutert.

Das Vorhandensein von Zeigern stellt die Anwendung der Slicing-Verfahren vor ein weiteres Hindernis, denn zur Berechnung der Datenabhängigkeiten muss bestimmt werden, welche Speicherstellen ein Zeiger referenzieren kann. Im Laufe der Zeit wurden verschiedene Ansätze zur Zeigeranalyse entwickelt (z.B. [Agr$^+$91], [And94], [Ema$^+$94], [Ste96], [Sha$^+$97], [Wil97], [Das00]), die sich in Genauigkeit und Effizienz unterscheiden. Hind und Pioli vergleichen in [Hin$^+$00] fünf Algorithmen miteinander und stellen fest, dass die Zeitkomplexität bei zunehmender Genauigkeit überproportional ansteigt. Aufgewogen werden könnte dieser Nachteil durch weniger zeitintensive Folgeberechnungen aufgrund der höheren Genauigkeit.

Laut [Sha$^+$97] lässt sich das Ergebnis einer Zeigeranalyse als Graph veranschaulichen, bei dem die Zeigervariablen und deren referenzierte Speicherstellen als Knoten und die „zeigt auf“-Beziehungen als Kanten dargestellt werden. Dieser Graph wird im Weiteren als PG, für engl. *points-to graph*, abgekürzt.

Zeigeranalyseverfahren lassen sich unter anderem anhand der Eigenschaften der Fluss- und Kontextsensitivität kategorisieren. Ist ein Verfahren flusssensitiv, so werden Kontrollflussinformationen verwendet. Dies macht es möglich, für jede Kante eines CFG einen PG zu ermitteln. Flussinsensitivität hingegen bedeutet, dass der Graph an jeder Stelle des Programms gültig sein muss. Kontextsensitive Algorithmen unterscheiden zwischen verschiedenen Aufrufen einer Prozedur, so dass für jeden Aufruf ein eigener PG existieren kann.

[5]Es könnten Datenabhängigkeiten berechnet werden, die in Wirklichkeit nicht vorhanden sind.

### Slicing objektorientierter Programme

Mit dem Aufkommen der objektorientierten Programmierung beschäftigten sich u.a. Tip et al. [Tip+96], Larsen und Harrold [Lar+96] sowie Liang und Harrold [Lia+98] mit dem Slicing von objektorientierten Programmen. So versuchen sich die beiden letztgenannten Arbeiten an der Darstellung von Klassen, Vererbung und Polymorphie im SDG. In jüngerer Zeit mehren sich die Artikel, die das Slicing von aspektorientierter Software thematisieren (z.B. [Zha02], [Zha+03b]).

Die Arbeit von Liang und Harrold basiert auf dem in Abschnitt 2.1.1 erläuterten Konzept von Horwitz et al. Die Herausforderungen beim Slicen objektorientierter Programme bestehen in der Behandlung von Objekten und Polymorphie. Zu deren Darstellung schlagen die Autoren einige Änderungen des SDG vor, die nachfolgend beschrieben werden sollen. Dabei nehmen sie an, dass alle Attribute privat sind, d.h. auf sie kann von außerhalb des Objekts nicht direkt, sondern nur durch Methodenaufrufe zugegriffen werden.

Objekte können als Empfänger einer Botschaft[6] oder als Parameter bei Methodenaufrufen auftreten. Im ersteren Fall werden die von einer Methode $m$ benutzten Attribute des Objekts wie die Eingabeparameter einer Prozedur als Actual-in- und Formal-in-Knoten behandelt. Dasselbe gilt für definierte Attribute und Actual-out- sowie Formal-out-Knoten. Es werden auch indirekt, d.h. von einer innerhalb des Methodenrumpfes von $m$ aufgerufenen Methode definierte und benutzte Attribute berücksichtigt.

Dient ein Objekt als Parameter bei einem Aufruf, so werden wie bereits bekannt Parameterknoten für das Objekt erzeugt, diese allerdings zu so genannten *Objektbäumen* erweitert. Diese Bäume bestehen aus einer das Objekt repräsentierenden Wurzel, dem ursprünglichen Parameterknoten, und seinen Attributen als Blätter. Ist ein Attribut wiederum ein Objekt, so wird auch dieses zu einem (Teil-)Baum erweitert etc. Dieses Verfahren führt bei Vorhandensein von rekursiven Definitionen zu unendlich großen Bäumen. Daher ist es sinnvoll, die Tiefe der Bäume zu begrenzen. Parameter-in- und Parameter-out-Kanten existieren nur zwischen den Blättern zweier Objektbäume.

Kann ein Objekt Instanz mehr als einer Klasse sein, so ist der den Methodenaufruf beinhaltende Knoten oder der Parameterknoten zunächst mit einem speziellen Knoten für jeden möglichen Typ des Objekts verbunden. An diese Knoten werden dann die Actual-in- und Actual-out-Knoten geknüpft bzw. sie bilden die Wurzel des Objektbaums.

Sofern eine Methode, hier $m_1$ genannt, eine andere Methode $m_2$ derselben Klasse aufruft, die in einer Subklasse redefiniert wird, sehen Liang und Harrold vor, dass im SDG eine Kopie aller Knoten von $m_1$ erstellt wird, so dass die Methode so behandelt werden kann, als würde sie selbst in der Subklasse redefiniert. Dies ist notwendig, weil sich die von der redefinierten Methode $m_2$ benutzten und definierten Attribute möglicherweise von denjenigen der

[6] ein Aufruf einer Methode des Objekts

Methode in der Oberklasse unterscheiden, was somit auch für die aufrufende Methode $m_1$ gilt (siehe die Aussage über indirekt benutzte oder definierte Attribute in Abschnitt 2.1.1).

Liang und Harrold machen keine Angaben über den Typ der die Knoten der Bäume verbindenden Kanten. Der Zwei-Phasen Algorithmus von Horwitz et al. müsste entsprechend modifiziert werden, falls es sich dabei nicht um Kantentypen handelt, die aus deren Version des SDG bereits bekannt sind.

### 2.1.2. Dynamisches Slicing

Dynamisches Slicing bezieht sich, anders als statisches Slicing, auf eine bestimmte Ausführung des Programms. Dazu wird als Eingabe zusätzlich zu dem Code und einem Slicing-Kriterium eine Historie der ausgeführten Programmanweisungen (engl. *trace*) benötigt.

#### Das Verfahren von Korel und Laski

Der Begriff des dynamischen Slicing wurde von Korel und Laski in [Kor$^{+}$88] geprägt. Angelehnt an Weiser definierten sie eine dynamische Slice $S$ als ein ausführbares Programm, welches durch Streichen von Anweisungen aus einem Ursprungsprogramm $P$ hervorgeht. Erst bei Betrachtung des Slicing-Kriteriums $< input, i^q, X >$ wird der Unterschied deutlich: $input$ steht für die Eingabe des Programms. Während bei statischem Slicing die Eingabe keine Rolle spielt, die Slice also für alle Eingaben gültig sein muss, ist eine dynamische Slice auf eine spezifische Eingabe und somit eine spezifische Programmausführung bezogen. $X$ bezeichnet eine Variablenmenge und $i^q$ einen bestimmten Punkt in der Ausführung des Programms: $i$ kennzeichnet eindeutig eine Anweisung (entspricht z.B. einer Zeilennummer) und $q$ gibt an, um die wievielte ausgeführte Anweisung seit Programmstart es sich handelt. Weiterhin müssen die Slices drei Bedingungen erfüllen (siehe [Tip95, S. 32]):

1. Bei Ausführung mit Eingabe $input$ sind die Trajektorien der Slice $S$ und des Programms $P$ identisch, wenn aus der Trajektorie von $P$ diejenigen $j^r$ entfernt werden, bei denen $r$ nicht in $S$ ausgeführt wird. Eine Trajektorie von $P$ ist die Sequenz der Anweisungsausführungen $j^r$, entspricht also der Trace.
2. Alle in $X$ enthaltenen Variablenwerte müssen bei Ausführung von $i^q$ in $S$ und $P$ identisch sein.
3. Anweisung $i$ ist in der Slice enthalten.

Aufbauend auf ihrer Definition veröffentlichten Korel und Laski ein Verfahren, dynamische Slices aus der Trajektorie und den Mengen der von den im Programm vorkommenden Anweisungen definierten und benutzten Variablen zu berechnen.

### Das Verfahren von Agrawal und Horgan

Agrawal und Horgan beschreiben in [Agr+90] vier Algorithmen zur Berechnung von dynamischen Slices. Die ersten beiden arbeiten auf dem PDG. Vereinfacht ausgedrückt markiert Algorithmus eins alle während der Programmausführung abgearbeiteten Anweisungen. Anschließend wird eine statische Slice über den sich aus den markierten Knoten ergebenden Graphen berechnet. Der zweite Algorithmus arbeitet ähnlich, außer dass er bei Programmausführung entsprechende Kanten des PDG markiert. Die Autoren weisen allerdings darauf hin, dass beide Ansätze zu ungenauen Ergebnissen führen könnten. Der dritte Algorithmus berechnet genaue Slices und benutzt dazu den dynamischen Abhängigkeitsgraphen (DDG, für engl. *dynamic dependence graph*), der einen Knoten für jede Ausführung einer Anweisung besitzt. Aufgrund dessen können DDGs sehr groß werden, weshalb Algorithmus 4 einen reduzierten DDG (RDDG, für engl. *reduced DDG*) verwendet, welcher, unter höherem Aufwand bei der Berechnung, Knoten zusammenfasst. In [Agr+91] behandeln Agrawal et al. Zeiger, zusammengesetzte Datentypen und Programme mit mehreren Prozeduren in Verbindung mit dynamischem Slicing.

Eine Auswahl an weiteren Arbeiten über dynamische Slicing-Verfahren ist [Far+01], [Far+02], [Zha+03a] und [Zha+04]. Zhao beschäftigt sich in [Zha98b] mit dem dynamischen Slicing objektorientierter Programme.

## 2.1.3. Mischformen und verwandte Konzepte

Ausgehend von statischem und dynamischem Slicing haben sich einige Mischformen entwickelt, die unter anderem Eigenschaften der beiden bisher erläuterten Ausprägungen kombinieren:

- *Hybrid Slicing*: die Einbindung von dynamischen Informationen, die durch das Einfügen von Haltepunkten beim Debuggen entstehen, in statische Slices (siehe [Gup+95])
- *Union Slicing*: die Berechnung von Slices für eine Menge von Eingaben, d.h. die Vereinigung mehrerer dynamischer Slices (siehe [Bes+02])
- *Conditioned Slicing*: Einfügen einer Bedingung in das Slicing-Kriterium, die von Ursprungsprogramm und Slice zu erfüllen sind

Ein mit Slicing verwandtes Konzept ist das so genannte Chopping, welches von Jackson und Rollins in [Jac+94] vorgestellt wurde. Deren lediglich intraprozeduraler Ansatz haben Reps und Rosay zu einem interprozeduralen Verfahren erweitert (siehe [Rep+95]). Ein Chop enthält nur diejenigen Anweisungen eines Programms, welche notwendig sind, Auswirkungen einer Anweisung $s$ auf eine Anweisung $t$ zu erhalten.

Nach Reps und Rosay sind die in einem Chop enthaltenen Knoten diejenigen, die auf einem Pfad zwischen den beiden Knoten liegen. Einschränkend dabei ist, dass, angenommen der Pfad sei in zwei aufeinanderfolgende Abschnitte unterteilt, der erste Abschnitt für jede enthaltene Parameter-in- oder Aufrufkante eine entsprechende Parameter-out-Kante aufweisen muss, die an denselben Aufruf anknüpft.

Im zweiten Teilpfad muss umgekehrt gelten, dass für jede Parameter-out-Kante, über die von einer Methode zu deren Aufruf „aufgestiegen" wird, eine entsprechende Parameter-in- oder Aufrufkante existieren muss, die an denselben Aufruf anknüpft.

## 2.2. Anwendungsbeispiele des Program Slicing

Program Slicing lässt sich für eine Vielzahl von Anwendungsgebieten verwenden. Nachfolgend werden mit *Debugging*, *Programmdifferenzierung und -integration* sowie *Softwarewartung und Programmverstehen* einige davon, inklusive beispielhafter Verfahrensweisen zum Einsatz von Program Slicing, vorgestellt.

### 2.2.1. Debugging

Debugging wird von Weiser in [Wei84] als primäre Anwendung des Program Slicing genannt. In seinem 1982 erschienenen Artikel *Programmers Use Slices When Debugging* [Wei82] erläutert er eine von ihm durchgeführte Studie, die zeigte, dass Programmierer beim Debuggen ein Programm „im Kopf" slicen. Bei Auftreten eines Programmfehlers, insbesondere eines Laufzeitfehlers, erlaubt dynamisches Slicing, Anweisungen auszuschließen, die für den Fehler nicht verantwortlich sein können.

Auch Chopping kann für das Debugging eine Rolle spielen, falls es beispielsweise gilt herauszufinden, aus welchem Grund eine Änderung in Anweisung $s$ einen Fehler in Anweisung $t$ verursacht.

### 2.2.2. Programmdifferenzierung und -integration

Unter Programmdifferenzierung versteht man die Gegenüberstellung zweier Programme bzw. Versionen eines Programms hinsichtlich semantischer Unterschiede (siehe [Bin$^+$96, S. 35]). Dazu werden die Rückwärtsslices aller Knoten der jeweiligen SDGs miteinander verglichen. Differieren die Slices zweier korrespondierender Knoten, so ist von einem unterschiedlichen Verhalten der beiden Programme an der betreffenden Stelle (genannt *Affected Point*, siehe [Tip95, S. 49]) auszugehen.

Programmintegration bezeichnet die Vereinigung mehrerer Programmvarianten und baut auf dem Verfahren der Differenzierung auf. Seien ein Programm $A$ und zwei unabhängige, auf der Basis von $A$ entwickelte Varianten $B$ und $C$ gegeben. Zunächst werden wie oben erklärt die Unterschiede zwischen $B$ und $A$ sowie $C$ und $A$ bestimmt. Die Bildung des integrierten Programms erfolgt durch Vereinigung der SDGs von $B$ und $C$ zuzüglich der vorher ermittelten *Affected Points*. Abschließend bleibt noch zu prüfen, ob sich die Unterschiede zwischen $B$ und $C$ im vereinigten Programm störend beeinflussen und ob sich der entstandene SDG überhaupt in ein Programm überführen lässt. Ersteres geschieht durch Vergleich der Slices der *Affected Points* in dem SDG des integrierten Programms mit denjenigen in dem SDG von $B$ bzw. $C$. Unterscheiden sich die Slices voneinander, so können die Programmvarianten nicht integriert werden. (siehe [Bin$^+$96, S. 35f] und [Tip95, S.48f])

### 2.2.3. Softwarewartung und Programmverstehen

Im Zusammenhang mit der Wartung von Software tut sich oft die Frage auf, ob und wenn ja, welche Auswirkungen das Hinzufügen, Ändern oder Löschen einer Anweisung an anderen Stellen im Programm haben. Gallagher und Lyle veröffentlichten in [Gal$^+$91] ihren Ansatz der so genannten *Decomposition Slices*. Diese beschränken sich nicht anhand eines Slicing-Kriteriums auf eine vorher spezifizierte Programmanweisung, sondern bestehen aus sämtlichen Anweisungen, welche den Wert einer Variable $x$ möglicherweise beeinflussen. Eine solche *decomposition slice* lässt sich auch als die Vereinigung der Slices für die Kriterien $< s_1, x >$ bis $< s_n, x >$ charakterisieren, wobei $s_{i,i\in\mathbb{N}}$ den $x$ ausgebenden Anweisungen zuzüglich der letzten Programmanweisung entsprechen. Bei einer *Output-restricted decomposition Slice*, abgekürzt ORD-Slice wurden die Ausgabeanweisungen entfernt.

Eine Anweisung, die in nur einer einzigen ORD-Slice auftaucht, wird *unabhängig*, ansonsten *abhängig* genannt. Variablen, welche ausschließlich von unabhängigen Anweisungen definiert werden, heißen ebenfalls *unabhängig*. Existieren Definitionen in abhängigen Anweisungen, sind die Variablen auch *abhängig*. Das *Komplement* einer ORD-Slice entspricht dem Ursprungsprogramm abzüglich den Ausgabeanweisungen und den unabhängigen Anweisungen der ORD-Slice. (siehe auch [Tip95, S. 49f])

Basierend auf ihrem Ansatz formulierten Gallagher und Lyle einige Richtlinien zur Änderung von Programmen, die das Verhalten des Komplements einer ORD-Slice unberührt lassen (siehe [Tip95, S. 50]):

- Unabhängige Anweisungen einer ORD-Slice können gelöscht werden.
- Definitionen unabhängiger Variablen lassen sich hinzufügen, ändern oder löschen.
- Neue Verzweigungen oder Schleifen, die abhängige Anweisungen beinhalten, wirken sich auf das Komplement aus.

Soll anschließend eine Qualitätssicherung des modifizierten Programms erfolgen, kann sich auf die ORD-Slice beschränkt werden, da sich für das Komplement keinesfalls etwas geändert hat.

## 2.3. Existierende Slicing-Werkzeuge

In diesem Abschnitt werden zwei Werkzeuge, *CodeSurfer* der Firma GrammaTech und das an der Kansas State University entwickelte *Indus* kurz präsentiert.

### 2.3.1. CodeSurfer

Der *CodeSurfer* von GrammaTech (siehe [Gra00], [And+01]) ist ein Werkzeug zur Unterstützung der Analyse und des Verstehens von C-Quellcode. Die Repräsentation der Programme basiert auf einem Systemabhängigkeitsgraphen. Neben einer Reihe anderer Programmanalyseverfahren bietet CodeSurfer auch Funktionen für vorwärts- und rückwärtsgerichtetes Slicing sowie Chopping. In der Dokumentation ist nicht eindeutig erwähnt, welche Algorithmen bei der Implementierung des Werkzeuges gewählt wurden. Die Seite unter dem Link `http://www.grammatech.com/products/codesurfer/codesurferpapers.htm` lässt aber den Schluss zu, dass man sich unter anderem auf die Arbeiten von Horwitz et al. ([Hor+90], [Hor+92], [Bal+93], [Rep+94]) und Reps ([Rep+95], [Rep97]) gestützt hat.

Die Zeiger-Analyse kann wahlweise mit den oben bereits erwähnten Algorithmen von Steensgaard [Ste96], Andersen [And94] oder Das [Das00] vorgenommen werden (Stand 2001, siehe [And+01]).

Abbildung 2.3 zeigt einen Screenshot von CodeSurfer: Ein Pop-up-Menü ermöglicht die Navigation innerhalb eines Programms anhand von Datenfluss-, Kontroll- oder Kontrollflussbeziehungen zwischen den Anweisungen.

### 2.3.2. Indus

Bei *Indus* handelt es sich um ein an der Kansas State University von John Hatcliff und seinen Mitarbeitern ins Leben gerufenes Projekt, welches die Entwicklung von Komponenten zur Analyse und Transformation von Java-Programmen verfolgt. Eine dieser Komponenten ist der *Java Program Slicer*, der mittels des Teilprojekts *Kaveri* in die Eclipse-Umgebung integriert wurde und kontextsensitives Slicing in beide Richtungen ermöglicht.

Ranganath erwähnt in seiner Beschreibung der Architektur des Java Program Slicers (siehe [Ran]), dass eine erste Version monolithisch aufgebaut war. Jedoch führte dies zu Problemen

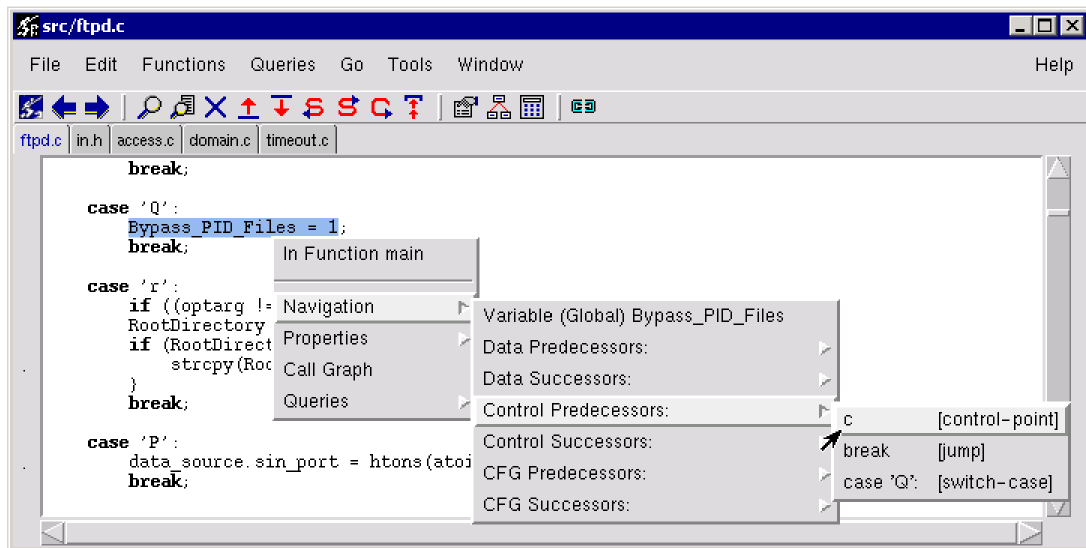


Abbildung 2.3.: Screenshot des Codeanalyse-Werkzeuges *CodeSurfer* von GrammaTech. Entnommen aus `http://www.grammatech.com/products/codesurfer/featuretour/ToControlPoints.html`. ©GrammaTech

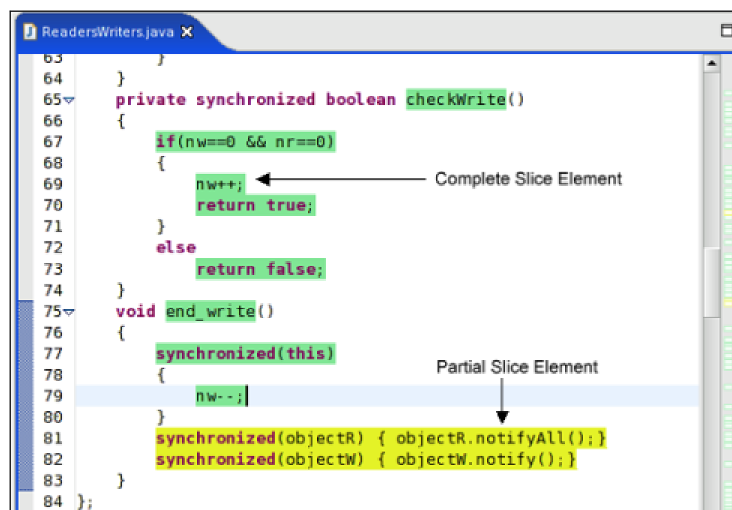


Abbildung 2.4.: Screenshot des Java-Slicing-Werkzeuges *Kaveri* der Kansas State University. Entnommen aus `http://indus.projects.cis.ksu.edu/projects/screenshots.shtml`.

bei der Berechnung der Slices, da sich verschiedene Formen und Eigenschaften der Slices nicht miteinander kombinieren ließen. Deshalb wurde entschieden, die eigentliche Berechnung und das Hinzufügen bestimmter Eigenschaften, wie z.B. Ausführbarkeit, zu trennen und jeweils in Teilkomponenten auszulagern.

Ein Screenshot von Kaveri ist Abbildung 2.4 zu entnehmen.

# 3. Grundlagen des Modells der Slicing-Dienste

Im Folgenden werden die Grundlagen der in den nächsten drei Kapiteln modellierten Slicing-Dienste behandelt. Nach der Darlegung der bei der Modellierung einbezogenen Verfahrensweisen folgt eine Definition des Begriffs „*Dienst*“. Im letzten Abschnitt wird eine einfache Beispiel-Programmiersprache beschrieben, die zur Erstellung des Modells genutzt wurde.

## 3.1. Auswahl der in dem Modell einbezogenen Verfahren

In diesem Abschnitt werden die bei der Entwicklung des Dienstmodells berücksichtigten Verfahrensweisen und Techniken sowie die Gründe für ihre Auswahl dargestellt.

### 3.1.1. Statisches Slicing und spezielle Programmeigenschaften

Die Alternativen bei der Modellierung der das statische Slicing durchführenden Dienste stehen das Verfahren nach Ottenstein und Ottenstein samt seiner Weiterentwicklungen und die Vorgehensweise nach Weiser zur Verfügung (siehe Abschnitt 2.1.1, S. 4). Letztere unterliegt allerdings folgenden Einschränkungen:

- Interprozedurale Slices sind aufgrund der Kontextinsensitivität unpräzise.
- Für Programme mit unstrukturiertem Kontrollfluss existieren nur ungenaue Erweiterungen der Vorgehensweise (siehe [Lyl84]).
- Konzepte der Objektorientierung werden nicht berücksichtigt.

Aufgrund dieser Nachteile kann eine Modellierung auf der Grundlage von Weisers Algorithmus nicht in Frage kommen. Zudem braucht ein Programmabhängigkeitsgraph nur einmal erstellt zu werden. Die Berechnung von Slices erfolgt dann mit linearem Zeitaufwand.

Für Programme mit mehreren Prozeduren werden Systemabhängigkeitsgraphen nach Horwitz et al. [Hor+90] erstellt, die jedoch in zwei Punkten von dem Vorbild abweichen. Erstens wird die Verbindung der Parameterknoten mit den Entry-Knoten bzw. den aufrufenden Knoten nicht über Kontrollkanten, sondern über spezielle Parameterkanten (engl. *parameter edge*) hergestellt. Dies geschieht, weil an Parameterknoten geknüpfte Kontrollkanten nicht mit der Definition dieser Kanten vereinbar sind (siehe [Fer+87, S. 322f]). Der Zwei-Phasen-Algorithmus zur Durchführung des Slicing auf dem SDG muss so angepasst werden, dass in beiden Phasen auch Parameterkanten traversiert werden.

Ein zweiter Unterschied entsteht durch das Hinzufügen von Deklarationskanten zwischen Deklaration und erster Definition einer Variablen. Dadurch wird sichergestellt, dass eine Slice die Deklarationen relevanter Variablen enthält.

Die Methode von Ball und Horwitz [Bal+93] findet bei der Behandlung von intraprozeduralen Sprunganweisungen Verwendung. Zur Unterscheidung von Verzweigungen und Schleifen im ACFG werden aus Sprunganweisungen ausgehende Kontrollflusskanten zum Sprungziel mit *always*, und zur nächsten Anweisung bei Nichtausführung des Sprunges mit *never* beschriftet. Auch „normale“, von einer nicht verzweigenden Anweisung ausgehende Kontrollflusskanten, tragen die Kennzeichnung *always*. Weiterhin verbindet eine *never*-Kante den Entry- mit dem Exit-Knoten. Das Modell sieht keine interprozeduralen Sprünge vor.

Zur Darstellung von Zeigerstrukturen benutzt das Modell für fluss- und kontextinsensitive Verfahren ausgelegte PGs (siehe Abschnitt 2.1.1). Dabei wird von dem Algorithmus zu ihrer Berechnung abstrahiert.

Die Integration der Konzepte der Objektorientierung richtet sich nach dem Ansatz von Liang und Harrold [Lia+98]. Jedoch wird auf die Erweiterung der Parameterknoten für Objekte zu Objektbäumen verzichtet. Auch bei der Berücksichtigung der Polymorphie werden hier eigene Wege gegangen. Die Duplizierung der Methoden, welche andere Methoden derselben Klasse aufrufen, die in einer Subklasse redefiniert werden, bewirkt im schlimmsten Fall eine Vervielfachung der SDG-Knoten. Daher wurde eine konservative Lösung gefunden, die zwar größere Slices[1] verursachen kann, aber ohne zusätzliche Knoten auskommt. Dazu werden die die möglichen Typen eines ein Objekt repräsentierenden Knotens weggelassen und die Entry-Knoten der möglicherweise aufgerufenen Methoden direkt mit dem aufrufenden Knoten verknüpft. Die Actual-in- und Actual-out-Knoten müssen der Vereinigung der benutzten und definierten Variablen aller mit dem Aufruf verbundenen Methoden entsprechen. Folglich können an einen Actual-in- oder Actual-out-Knoten mehrere Parameter-in- bzw. Parameter-out-Kanten knüpfen. Aus dieser Tatsache resultiert die Ungenauigkeit dieses Konzeptes: In der zweiten Phase des Slicing-Algorithmus müssen von einem Actual-out Knoten aus alle Parameter-out-Kanten traversiert werden, was unter Umständen zu der Inklusion eigentlich nicht aufgerufener Methoden führt.

[1] Die Slices enthalten eventuell mehr Anweisungen als solche, die mit Hilfe der SDG-Variante von Liang und Harrold berechnet wurden.

Der Systemabhängigkeitsgraphen wird fortan, aufgrund der obengenannten Modifikationen, erweiterter Systemabhängigkeitsgraph (Abk. ESDG, für engl *extended system dependence graph*) genannt. Die Basis zu dessen Berechnung bildet, neben ASG, ACFG und PG, ein erweiterter Aufrufgraph (Abk. ECG, für engl. *extended call graph*). *Erweitert* deshalb, weil dieser nicht nur die Aufrufbeziehungen zwischen Methoden, sondern auch deren benutzte und definierte Variablen darstellt.

Das ursprüngliche Slicing-Kriterium $< i, X >$ wird zu $< v, X >$ abgewandelt, bei dem $v$ denjenigen Knoten des ESDG bezeichnet, welcher der Anweisung $i$ entspricht.

### 3.1.2. Dynamisches Slicing

Zur Bestimmung dynamischer Slices stehen die Verfahren nach Korel und Laski [Kor$^+$88] sowie Agrawal und Horgan [Agr$^+$90] zur Verfügung. Ersteres leidet, da es auf Weisers Arbeit aufbaut, unter ähnlichen Einschränkungen. Die beiden letztgenannten Autoren stellen in ihrem Artikel insgesamt vier Algorithmen mit zunehmender Genauigkeit, aber auch abnehmender Effizienz vor. Laut [Tip95] sind der DDG (*Dynamic Dependence Graph*) und der RDDG (*Reduced DDG*), die von den Algorithmen drei und vier zur Bestimmung präziser Slices benötigt werden, von der Größenordnung $O(E)$ bzw. $O(2^n)$, wobei $E$ der Anzahl der ausgeführten Anweisungen und $n$ der Anzahl der Programmanweisungen im Quellcode entspricht. Daraus folgt, dass die präziseren Verfahren in der Praxis nicht relevant sind und für die dynamischen Slicing-Dienste auf den zweiten von Agrawal und Horgan beschriebenen Algorithmus als eine Kompromisslösung zurückgegriffen wird.

Dieser Algorithmus sieht vor, dass Kanten des ESDG, welche während der Programmausführung aufgetretene Kontrollabhängigkeiten, Aufrufe, Parameterübergaben etc. darstellen, gesondert zu markieren sind. Die Bestimmung der Slice selbst erfolgt durch den bekannten Zwei-Phasen-Algorithmus, mit dem Unterschied, dass in diesem Fall nur markierte Kanten traversiert werden.

Die Trace einer Programmausführung hat die Form $< v_1, v_2, \ldots, v_n >$ und entspricht der Sequenz der bei der Programmausführung abgearbeiteten Anweisungen $v_{i,i \in \mathbb{N}}$

### 3.1.3. Chopping

Das interprozedurale Choppingverfahren nach Reps und Rosay [Rep$^+$95] bietet den Vorteil, dass es auch auf dem Systemabhängigkeitsgraphen arbeitet und somit keine neuen Datenstrukturen notwendig macht. Der Algorithmus kann ohne Schwierigkeiten an den erweiterten Systemabhängigkeitsgraphen angepasst werden.

Auch wird ein *dynamisches Chopping* vorgesehen, bei dem ausschließlich wie in Abschnitt 3.1.2 dargestellte, markierte Kanten traversiert werden.

Die die beiden Anweisungen $s$ und $t$ darstellenden ESDG-Knoten $v$ und $w$, für die ein Chop zu berechnen ist, werden zu einem Chopping-Kriterium $ChopCrit =< v, w >$ zusammengefasst.

## 3.2. Definition des Begriffs „*Dienst*“

Der Begriff *Dienst* bezeichnet im Kontext dieser Arbeit eine Softwarekomponente, welche über eine definierte Schnittstelle verfügt. Genauer lassen sich folgende Eigenschaften aufzählen:

1. Ein Dienst benötigt eine definierte Eingabe zur Wahrung seiner Funktionalität.
2. Es wird eine definierte Ausgabe erzeugt.
3. Die Algorithmen zur Herstellung der Funktionalität sind nicht zwingend vorgegeben, solange die Schnittstelle beibehalten wird.
4. Ein Dienst kann aus mehreren (Teil-)Diensten bestehen. Die Summe der von außerhalb stammenden Eingaben, d.h. ausgenommen die zwischen den Teildiensten fließenden Daten, muss den Eingaben des zusammengesetzten Dienstes entsprechen. Dasselbe gilt für die Ausgaben.

Ein Slicing-Dienst erfüllt also eine im Rahmen des Program Slicing anfallende Teilaufgabe und interagiert zu diesem Zweck mit anderen Diensten, der Systemumgebung oder dem Benutzer. Die nächsten drei Kapitel befassen sich mit der Spezifikation der Dienste, ihrer Schnittstellen und Funktionalität.

## 3.3. Beispielsprache *EL*

Das in Kapitel fünf vorgestellte Datenmodell basiert auf der einfachen Beispielsprache *EL*, für engl. *Exemplary Language*. Das Ziel bei deren Gestaltung war es, das Slicing-Verfahren nach Ottenstein und Ottenstein samt seiner Weiterentwicklungen anhand dieser Sprache beschreiben zu können. Zu diesem Zweck muss sie mindestens folgende Eigenschaften aufweisen:

- Objektorientierung (durch Klassenbildung, Vererbung, Polymorphie[2]) zur Durchführung des Slicings objektorientierter Programme

[2] Polymorphie lässt sich nicht anhand aus einer Grammatik ersichtlichen syntaktischer Konstrukte darstellen. Es wird aber angenommen, dass ES Polymorphie ermöglicht.

```
 1  class A
 2  {
 3    int a
 4    int *p

 5    void m1()
 6    {
 7      B1 o
 8      a = o.pm(1)
 9      p = &a
10      while (a < 10)
11      {
12        *p = a + 2
13        if (*p > 3)
14          break
15      }
16    }

17    void m2()
18    {
19      B2 o
20      a = o.pm(4)
21      o.m3()
22      o.m4(&a)
23    }
24  }

25  class B1
26  {
27    int b

28    B1()
29    {
30      b = 0
31    }

32    int pm(int x)
33    {
34      x = x + b
35      return x
36    }

37    void m3()
38    {
39      b = pm(5)
40    }
41  }

42  class B2 extends B1
43  {
44    int c

45    B2()
46    {
47      c = 0
48    }

49    int pm(int x)
50    {
51      x = x + 6*b
52      c = x
53      return x
54    }

55    void m4(int *q)
56    {
57      *q = 7
58    }
59  }
```

Abbildung 3.1.: Beispielprogramm zu der mit der Grammatik in Abbildung 3.2, S. 25 beschriebenen Sprache EL.

- Verwendung von Prozeduren für interprozedurales Slicing (durch Objektorientierung – Methoden – bereits impliziert)
- Sprunganweisungen zur Behandlung unstrukturierten Kontrollflusses
- Zeiger zur Durchführung des Slicings bei Vorhandensein von Zeigern

Aus Gründen der Anschaulichkeit enthält die Sprache ausschließlich diejenigen Konstrukte, welche für die Realisierung der oben genannten Eigenschaften notwendig sind. Die Grammatik der Sprache ist, abgesehen von der goto-Anweisung und der expliziten Darstellung von Zeigern, an Java angelehnt und in ihrer EBNF-Notation Abbildung 3.2, S. 25 zu entnehmen.

Ein in EL geschriebenes Programm (siehe Abbildung 3.1) besteht aus Klassen (*class*), welche Subklasse maximal einer anderen Klasse sein können, ausgedrückt durch das Schlüssel-

wort “extends”. Klassen enthalten Attributdeklarationen (*memberVariableDeclaration*) und Methodendefinitionen (*memberFunctionDefinition*) als Elemente (*classMember*). Es wird angenommen, dass alle Attribute privat sind, d.h. ein Zugriff von außerhalb des Objekts kann nur über Methoden des Objekts erfolgen.

Methoden geben einen Wert zurück und können beliebig viele formale Parameter (*formalParameter*) aufweisen. Der Methodenrumpf besteht aus einem Block (*block*) mit beliebig vielen Blockelementen (*blockElement*). Blockelemente sind entweder nur innerhalb des Blocks gültige lokale Variablendeklarationen (*localVariableDeclaration*) oder Anweisungen (*statement*), denen eine Sprungmarke (*label*) vorangestellt werden kann. Anweisungen können zusammengesetzt (*compoundStatement*) oder einfach (*simpleStatement*) sein.

Zeigervariablen und Methoden, die einen Zeiger zurückgeben, werden, wie aus C++ bekannt, mit einem Stern (*) gekennzeichnet. Dabei seien nur einfache Zeiger erlaubt, d.h. eine Zeigervariable darf keine andere Zeigervariable referenzieren. Zum Umgang mit Zeigern werden, wie aus dem Beispielprogramm in Abbildung 3.1 ersichtlich, der Dereferenzoperator * sowie der Adressoperator & benutzt. Eine weitere Einschränkung ist, dass einer Zeigervariable ausschließlich mit Hilfe des Adressoperators die Adressen anderer Variablen zugewiesen werden dürfen: Zuweisungen wie z.B. `int *p; p = 2` sind nicht erlaubt.

Zusammengesetzte Anweisungen sind wiederum Blöcke oder aber mit einem Test verbunden (*compoundStatementWithTest*). Dabei kann es sich um Schleifen (*loop*) oder if-Anweisungen (*if*) handeln. Der aus einer Anweisung bestehende Schleifenrumpf wird ausgeführt, falls der den Test repräsentierende Ausdruck (*expression*) zu „wahr“ ausgewertet wird. An if-Anweisungen können maximal zwei (durch “else” abgetrennte) Anweisungen gebunden sein, die bei Auswertung des Tests zu „wahr“ bzw. „falsch“ entsprechend ausgeführt werden.

Einfache Anweisungen sind entweder Nicht-Sprunganweisungen (*nonJumpStatement*) oder Sprunganweisungen (*jumpStatement*). Erstere können die leere Anweisung (*emptyStatement*), Zuweisungen (*assignment*) oder Aufrufanweisungen[4] (*callStatement*) sein. Bei den Sprunganweisungen wird zwischen goto-Anweisungen (*gotoStatement*), die zu einer Sprungmarke innerhalb derselben Methode verzweigen, return-Anweisungen (*returnStatement*), die zur aufrufenden Methode springen und dabei ggf. das Ergebnis einer Ausdrucksauswertung zurückgeben, sowie strukturierten Sprung-Anweisungen (*structuredJumpStatement*) unterschieden. Letztgenannte können Teil eines Schleifenrumpfs sein und verzweigen entweder zu dem Schleifenkopf (*continue*), oder zur ersten Anweisung nach dem Schleifenrumpf (*break*).

Ausdrücke werden als Tests in zusammengesetzten Anweisungen mit Test, als aktuelle Parameter in Aufrufen oder in Zuweisungen, um einer Variable (*variable*) das Ergebnis der Auswertung zuzuweisen, verwendet. Ausdrücke können Variablen, Konstanten (*constant*),

[4] Aufrufanweisungen können, im Unterschied zu Funktionsaufrufen, nicht Teil eines Ausdrucks sein.

```
program                      ::==  {class}
class                        ::==  identifier [“extends” identifier] {classMember}
classMember                  ::==  memberVariableDeclaration | memberFunctionDefiniti-
                                   on
memberFunctionDefinition     ::==  type [*]memberFunction “(” [formalParameter {“,” formal-
                                   Parameter}] “)” block
memberFunction               ::==  identifier
formalParameter              ::==  type [*]identifier
memberVariableDeclaration    ::==  type [*]variable {“,” [*]variable}
localVariableDeclaration     ::==  type [*]variable {“,” [*]variable}

type                         ::==  identifier
statement                    ::==  [label“:”] (compoundStatement | simpleStatement)
label                        ::==  identifier
compoundStatement            ::==  compoundStatementWithTest | block
simpleStatement              ::==  nonJumpStatement | jumpStatement
compoundStatementWithTest    ::==  loop | if
block                        ::==  “{” {blockElement} “}”
blockElement                 ::==  localVariableDeclaration | statement
loop                         ::==  “while (” [expression] “)” statement
if                           ::==  “if (” expression “)” statement [“else” statement]
nonJumpStatement             ::==  emptyStatement | assignment | callStatement

emptyStatement               ::==  “”
callStatement                ::==  [variable (“.” | “->”)]memberFunction “(” [expression {“,”
                                   expression}] “)”
jumpStatement                ::==  structuredJumpStatement | returnStatement | gotoState-
                                   ment
structuredJumpStatement      ::==  “break” | “continue”
returnStatement              ::==  “return” [expression]
gotoStatement                ::==  “goto” label
assignment                   ::==  [“*”]variable operator expression
expression                   ::==  compoundExpression | variable | constant | functionCall
variable                     ::==  identifier
compoundExpression           ::==  [expression] operator expression
constant                     ::==  value
functionCall                 ::==  [variable (“.” | “->”)] memberFunction “(” [expression {“,”
                                   expression}] “)”
operator                     ::==  identifier
```

Abbildung 3.2.: Grammatik der Beispielsprache EL[3].

Funktionsaufrufe (*functionCall*) sein oder aus anderen, durch unäre oder binäre Operatoren (*operator*) verknüpften Ausdrücken (*compoundExpression*) bestehen.

Durch die Operatoren (.) und (->) kann angegeben werden, Element welchen Objekts eine Methode ist. Letzterer wird bei Zeigern auf Objekte verwendet.

Eine weitere Spracheigenschaft neben der Polymorphie, die nicht aus der Grammatik hervorgeht, ist der Parameterübergabemechanismus, für den *call-by-value* angenommen wird, d.h. es werden Kopien an die Methode übergeben, so dass sich etwaige Redefinitionen der Parameter innerhalb der Methode nicht auf die aufrufende Methode auswirken.

In einem möglichst kurzen Programmcode enthält das Beispielprogramm in Abbildung 3.1 die wesentlichen Sprachelemente, um in Kapitel 5 anhand weiterer Beispiele die zwischen den Diensten fließenden Daten zu verdeutlichen.

# 4. Datenfluss der Slicing-Dienste

In diesem Kapitel werden die Dienste, aus denen sich das in dieser Arbeit entwickelte Modell zusammensetzt, eingeführt. Der Datenaustausch zwischen den Diensten sowie die Interaktion mit der Systemumgebung und dem Benutzer wird mit Hilfe einer Datenflussbeschreibung erläutert. Aus dieser wird auch die Dienstehierarchie, d.h. die Zerlegung des das Gesamtsystem repräsentierenden Dienstes *ProgramSlicing*[1] in seine Teildienste, deutlich.

Die meisten der zwischen den Diensten fließenden Daten werden im nächsten Kapitel anhand von Graphenschemata spezifiziert. Ausgenommen sind die „von außen“ stammenden Eingaben: der Programmcode, die Slicing- und/oder Chopping-Kriterien sowie die Programmausführungstraces. Letztere drei Datenstrukturen werden in Kapitel 6 bei den sie verwendenden Diensten konkretisiert. Ebenfalls erfolgt dort eine detailliertere Beschreibung der Funktionalität der Dienste. Die Darstellung der unterstützten Programmiersprache wurde bereits im vorherigen Kapitel vorgenommen.

Das Kapitel gliedert sich in je einen Abschnitt für die Dienste der beiden obersten Hierarchieebenen. Der letzte Abschnitt vermittelt tabellarisch einen Überblick über die Dienste samt ihrer Ein- und Ausgabeschnittstellen.

## 4.1. Der Dienst *ProgramSlicing*

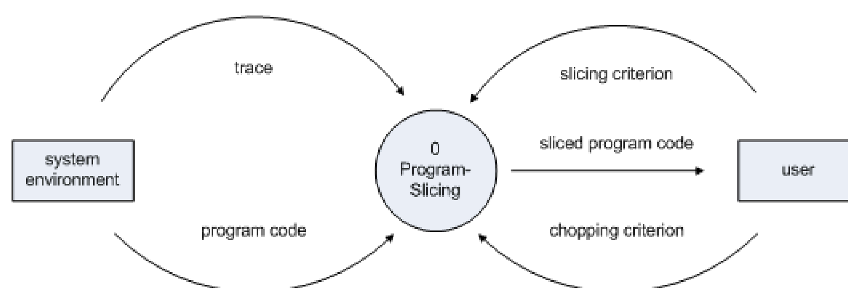


Abbildung 4.1.: Kontextdiagramm

[1] Begriffe in *serifenloser Schrägschrift* stehen für Elemente des Modells (z.B. Dienste oder Kanten- und Knotentypen in Kapitel 5)

Das Gesamtsystem der Slicing-Dienste, in dem Kontextdiagramm in Abbildung 4.1 durch den Dienst *ProgramSlicing* dargestellt, benutzt den zu bearbeitenden Programmcode sowie ein Slicing- oder Chopping-Kriterium, um die Slice bzw. den Chop zu ermitteln. Zur Bestimmung einer dynamischen Slice oder eines dynamischen Chops werden zusätzlich Tracedaten einer Programmausführung benötigt. Die Ausgabe erfolgt als Programmcode, bei dem die überflüssigen Syntaxelemente nicht mehr enthalten sind.

*ProgramSlicing* lässt sich in drei Teildienste aufspalten (siehe Abbildung 4.2). *preprocess-Program* (siehe Abbildung 4.3) hat die Aufgabe, aus dem Programmcode den erweiterten Systemabhängigkeitsgraphen (siehe Abschnitt 5.6) zu erstellen. *sliceProgram* berechnet, unter Eingabe des Slicing-Kriteriums und der Tracedaten, die Slice oder den Chop (zum Datenfluss *ACFGSet* siehe Abschnitt 5.3). Abschließend wandelt *convertESDGToCode* die im ESDG markierte Slice wieder in Programmcode um.

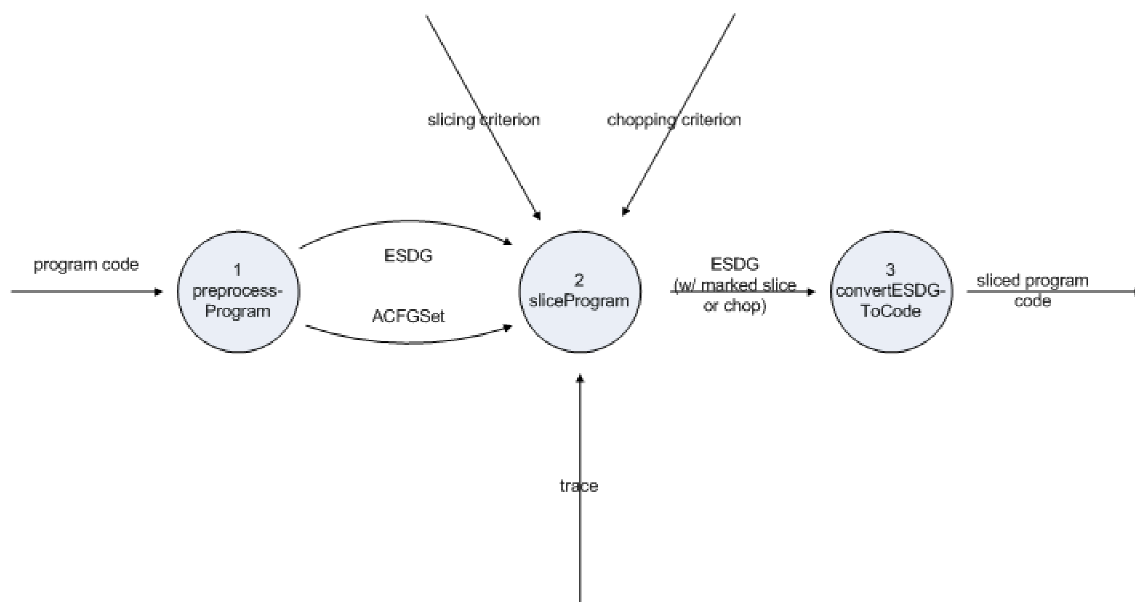


Abbildung 4.2.: Datenflussdiagramm 0 – *ProgramSlicing*

## 4.2. Der Dienst *preprocessProgram*

Zur Berechnung des ESDG aus dem Programmcode sind mehrere Zwischenschritte vonnöten, die in *preprocessProgram* zusammengefasst sind. Diese sind im Einzelnen (siehe Abbildung 4.3):

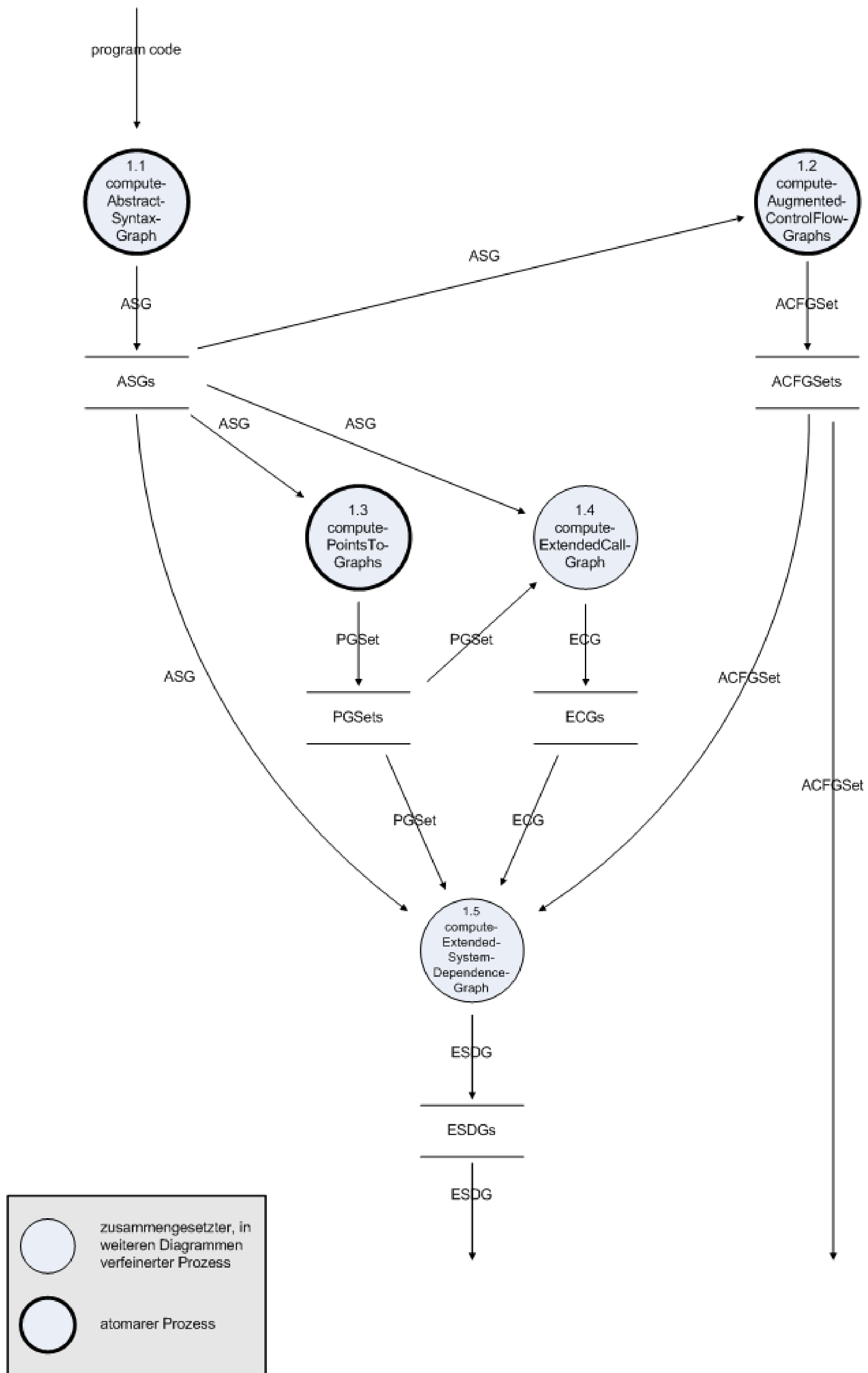


Abbildung 4.3.: Datenflussdiagramm 1 – *preprocessProgram*

**1.1** *computeAbstractSyntaxGraph*,

**1.2** *computeAugmentedControlFlowGraphs*[2],

**1.3** *computePointsToGraphs*,

**1.4** *computeExtendedCallGraph*,

**1.5** *computeExtendedSystemDependenceGraph*

Der Dienst *computeAbstractSyntaxGraph* erzeugt aus dem Programmcode unter Verwendung von Techniken des Compilerbaus (lexikalische Analyse, syntaktische Analyse) einen abstrakten Syntaxgraphen des Programms (Abk. ASG, für engl. *abstract syntax graph*, siehe Abschnitt 5.2 für eine detaillierte Beschreibung des Graphen).

Aus dem ASG wird von *computeAugmentedControlFlowGraphs* die Menge der erweiterten Kontrollflussgraphen (siehe Abschnitt 5.3) berechnet, bestehend aus den ACFGs der einzelnen Methoden.

Der abstrakte Syntaxgraph wird ebenfalls verwendet, um von dem Dienst *computePointsToGraphs* mit Hilfe von Zeigeranalyseverfahren die Menge der Points-to-Graphen (siehe Abschnitt 5.4), ebenfalls einen PG pro Methode, zu erstellen.[3]

In dem Dienst *computeExtendedCallGraph* (siehe Abbildung 4.4) sind die Teildienste

**1.4.1** *computeCallGraph*,

**1.4.2** *computeDefUseInformation*

enthalten. Ersterer ermittelt aus dem ASG und der PG-Menge, d.h. den PGs der einzelnen Methoden, den Aufrufgraphen (Abk. CG, für engl. *call graph*, siehe Abschnitt 5.5) zur Darstellung der Aufrufbeziehungen zwischen Methoden. *computeDefUseInformation* bestimmt durch Analyse des ASG und der PGs die von einer Methode veränderten bzw. benutzten Variablen. Diese werden mit dem CG verknüpft, so dass der erweiterte Aufrufgraph entsteht.

*computeExtendedSystemDependenceGraph* (siehe Abbildung 4.5) besteht aus den Teildiensten:

**1.5.1** *computeBasicESDG*,

**1.5.2** *computeInterMethodEdges*,

**1.5.3** *computeControlDependenceEdges*

[2]Der Plural deutet hier an, dass dieser und folgende Dienste mehr als einen Graphen pro Programm berechnen

[3]Bei flusssensitiven Zeigeranalyseverfahren müssten die ACFGs verwendet werden

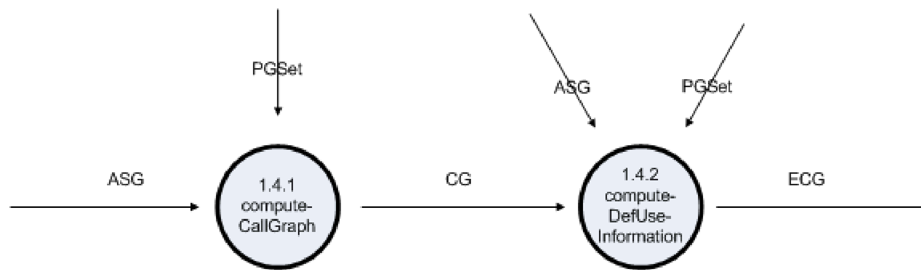


Abbildung 4.4.: Datenflussdiagramm 1.4 – *computeExtendedCallGraph*

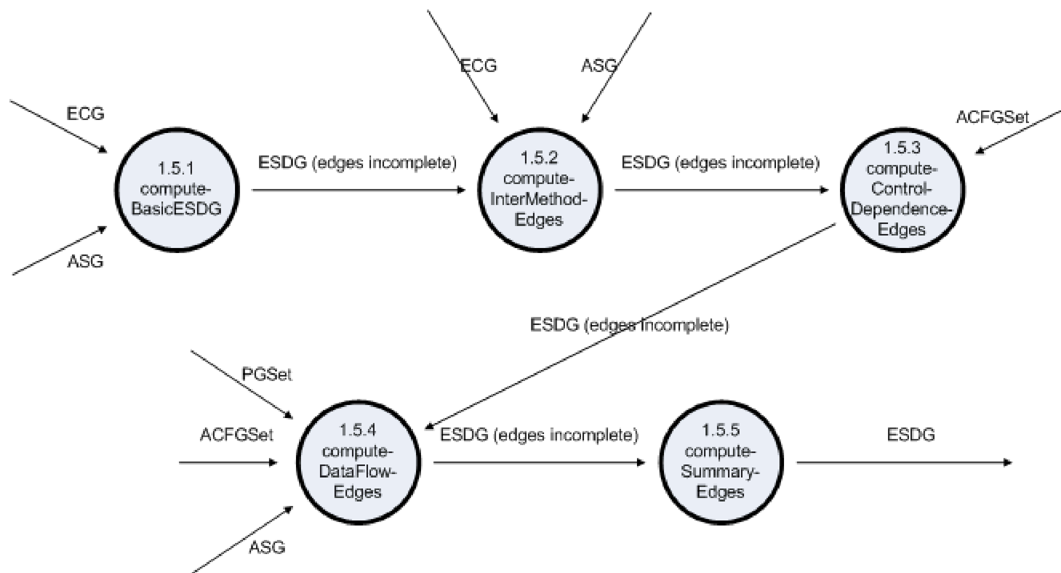


Abbildung 4.5.: Datenflussdiagramm 1.5 – *computeExtendedSystemDependenceGraph*

**1.5.4** *computeDataFlowEdges*,

**1.5.5** *computeSummaryEdges*,

Diese Teildienste erstellen den ESDG, indem *computeBasicESDG* zunächst dessen Grundgerüst für die Methoden aus dem ASG und dem ECG erzeugt. Anschließend verbindet *computeInterMethodEdges*, wieder unter Benutzung des ASG und ECG, die einzelnen Methoden durch Parameter-in-, Parameter-out- und Aufrufkanten (siehe Abschnitt 5.6.2). Zur Berechnung der Kontrollkanten (siehe Abschnitt 5.6.3) durch *computeControlDependenceEdges* wird lediglich die ACFG-Menge benötigt, während *computeDataFlowEdges* zusätzlich die Menge der PGs sowie den ASG nutzt, um die Datenflusskanten und Deklarationskanten (siehe Abschnitt 5.6.4) zu berechnen. Zuletzt ermittelt *computeSummaryEdges* die transitiven Kanten.

## 4.3. Der Dienst *sliceProgram*

Dieser Dienst berechnet Slices oder Chops unter Eingabe eines Slicing- bzw. Chopping-Kriteriums. Sollen dynamische Slices oder Chops ermittelt werden, ist zusätzlich die Trace einer bestimmten Programmausführung erforderlich. Die das Slicing und Chopping durchführenden Verfahren arbeiten auf dem ESDG und markieren die in den Programmausschnitten enthaltenen Knoten. Enthält die Variablenmenge $X$ des Slicing-Kriteriums $< v, X >$ nicht sämtliche von $v$ benutzten oder definierten Variablen, so wird die Menge der ACFGs benötigt, um das Kriterium anzupassen.

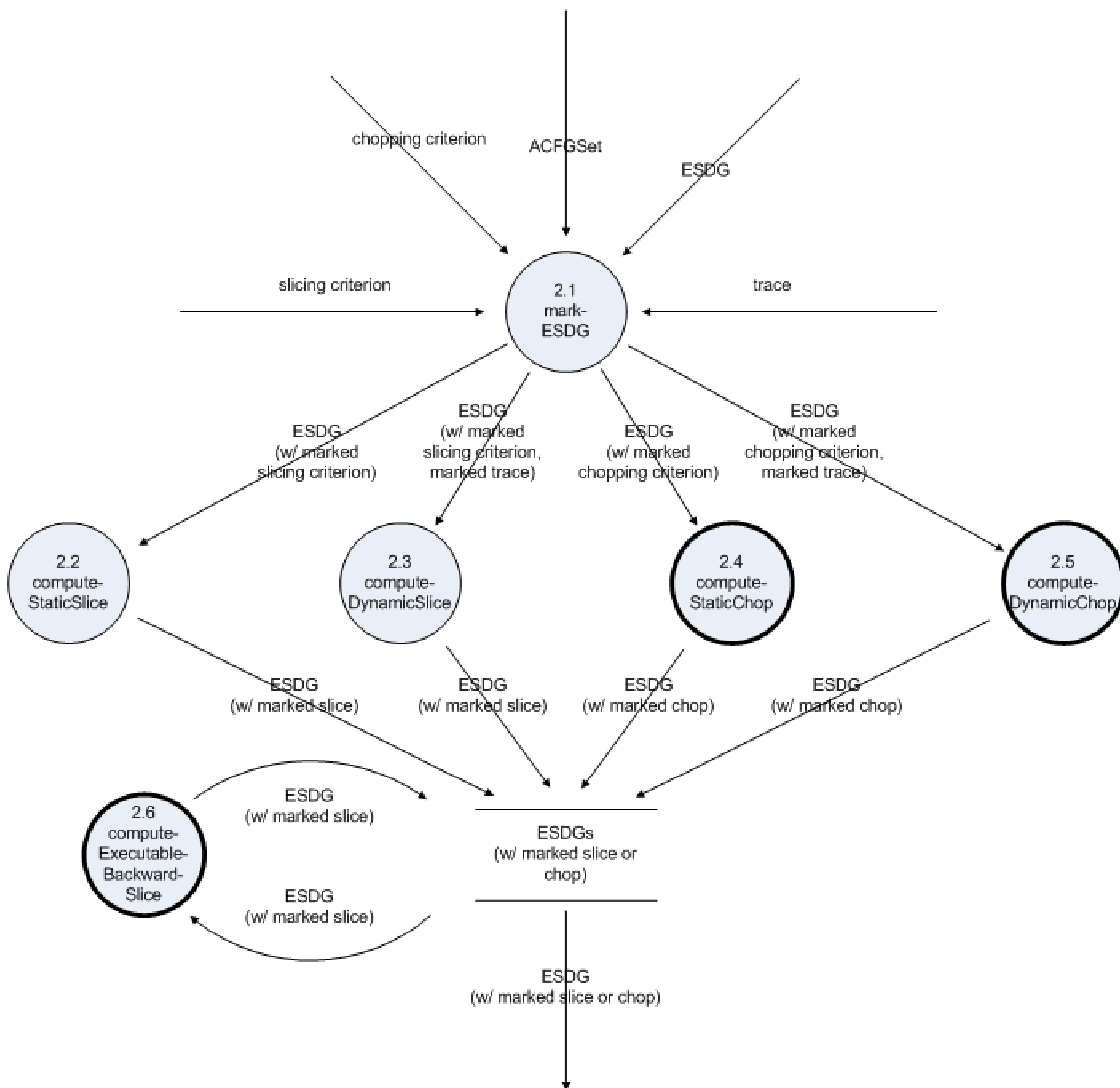


Abbildung 4.6.: Datenflussdiagramm 2 – *sliceProgram*

*sliceProgram* setzt sich aus den Diensten

**2.1** *markESDG*,

**2.2** *computeStaticSlice*,

**2.3** *computeDynamicSlice*,

**2.4** *computeStaticChop*,

**2.5** *computeDynamicChop*,

**2.6** *computeExecutableBackwardSlice*

zusammen (siehe Abbildung 4.6), wobei *markESDG* (siehe Abbildung 4.7) drei Teildienste enthält:

**2.1.1** *markSlicingCriterion*,

**2.1.2** *markChoppingCriterion*

**2.1.3** *markTrace*.

Der Dienst 2.1.1 markiert das Slicing-Kriterium im ESDG. Der ACFG wird benötigt, falls nicht genau ein Knoten das Slicing-Kriterium repräsentiert und deshalb mehrere zu markieren sind. *markChoppingCriterion* markiert die beiden im Chopping-Kriterium enthaltenen Knoten. Der dritte Dienst analysiert die Trace der Programmausführung und markiert Kanten des ESDG, um somit dynamisches Slicing vorzubereiten.

Dadurch, dass diese Dienste nicht *preprocessProgram* zugeordnet werden, wird verdeutlicht, dass die Programmdarstellung in Form des ESDG nur einmalig zu berechnen ist, um anschließend beliebig viele Slices oder Chops für verschiedene Kriterien zu bestimmen.

Die in *computeStaticSlice* (siehe Abbildung 4.8) und *computeDynamicSlice* (siehe Abbildung 4.9) eingebetteten Dienste für Rückwärts- und Vorwärtsslicing

**2.2.1** *computeStaticBackwardSlice*,

**2.2.2** *computeStaticForwardSlice*

bzw.

**2.3.1** *computeDynamicBackwardSlice*,

**2.3.2** *computeDynamicForwardSlice*

sowie *computeStaticChop* und *computeDynamicChop* führen das jeweilige Verfahren auf dem ESDG aus und markieren die in der Slice bzw. dem Chop befindlichen Knoten. Der Dienst *computeExecutableBackwardSlice* erstellt aus einer nicht ausführbaren Rückwärtsslice ein ausführbares Programm.

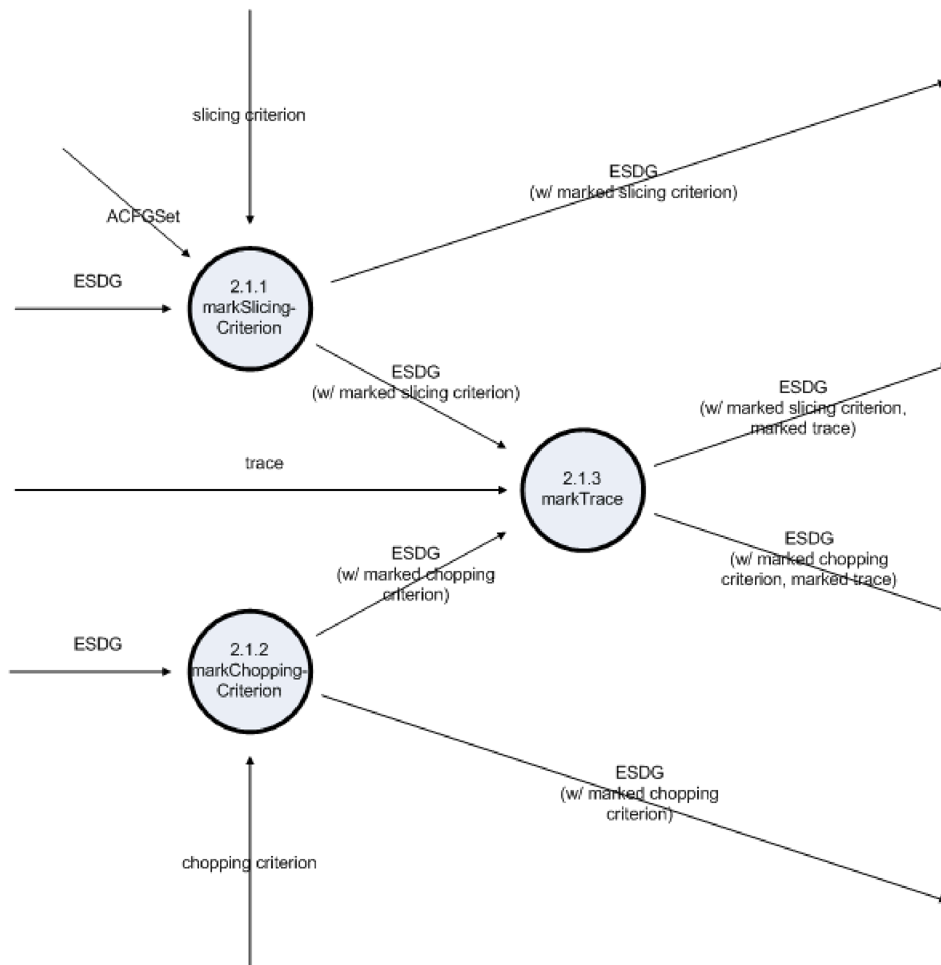


Abbildung 4.7.: Datenflussdiagramm 2.1 – *markESDG*

## 4.4. Der Dienst *convertESDGToCode*

Der Dienst *convertESDGToCode* wandelt den ESDG in Programmcode um. Dabei werden nur diejenigen Knoten berücksichtigt, welche als in der Slice oder in dem Chop befindlich markiert sind.

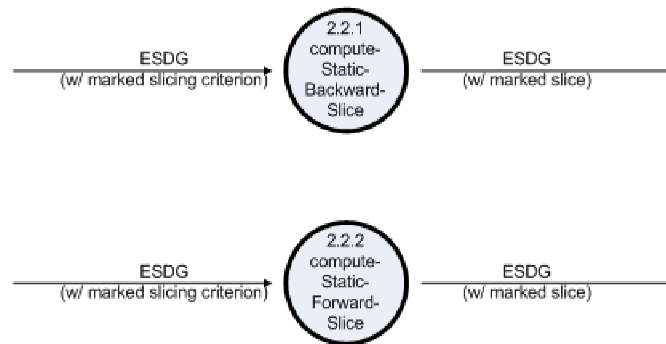


Abbildung 4.8.: Datenflussdiagramm 2.2 – *computeStaticSlice*

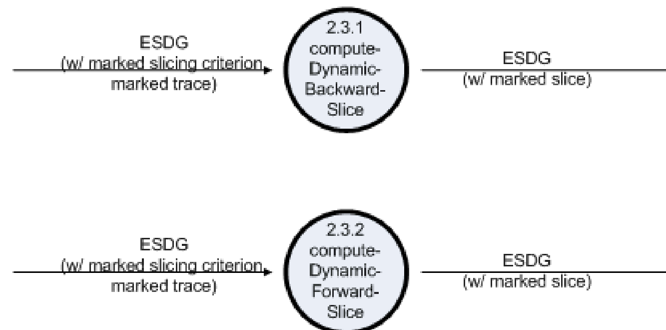


Abbildung 4.9.: Datenflussdiagramm 2.3 – *computeDynamicSlice*

## 4.5. Überblick

Die Tabelle in Abbildung 4.10 fasst die Dienste mit ihren Ein- und Ausgaben noch einmal zusammen.

| Nr. | Dienst | Eingaben | Ausgaben |
|---|---|---|---|
| 0 | ProgramSlicing | program code, slic. crit., chop. crit., trace | sliced program code |
| 1 | preprocessProgram | program code | ESDG, ACFGSet |
| 1.1 | computeAbstractSyntaxGraph | program code | ASG |
| 1.2 | computeAugmentedControl-FlowGraphs | ASG | ACFGSet |
| 1.3 | computePointsToGraphs | ASG | PGSet |
| 1.4 | computeExtendedCallGraph | ASG, PGSet | ECG |
| 1.4.1 | computeCallGraph | ASG, PGSet | CG |
| 1.4.2 | computeDefUseInformation | CG, ASG, PGSet | ECG |
| 1.5 | computeExtendedSystem-DependenceGraph | ASG, ECG, PGSet, ACFGSet | ESDG |
| 1.5.1 | computeBasicESDG | ASG, ECG | ESDG (edges incomplete) |
| 1.5.2 | computeInterMethodEdges | ESDG (edges incomplete), ASG, ECG | ESDG (edges incomplete) |
| 1.5.3 | computeControlDependence-Edges | ESDG (edges incomplete), ACFGSet | ESDG (edges incomplete) |
| 1.5.4 | computeDataFlowEdges | ESDG (edges incomplete), ASG, ACFGSet, PGSet | ESDG (edges incomplete) |
| 1.5.5 | computeSummaryEdges | ESDG (edges incomplete) | ESDG |
| 2 | sliceProgram | ESDG, ACFGSet, slic. crit., chop. crit., trace | ESDG (w/ mar. slice or chop) |
| 2.1 | markESDG | ESDG, ACFGSet, slic. crit., chop. crit., trace | ESDG (w/ mar. slic. crit.), ESDG (w/ mar. slic. crit., mar. trace), ESDG (w/ mar. chop. crit.), ESDG (w/ mar. chop. crit., mar. trace) |
| 2.1.1 | markSlicingCriterion | ESDG, ACFGSet, slic. crit. | ESDG (w/ mar. slic. crit.) |
| 2.1.2 | markChoppingCriterion | ESDG, chop. crit. | ESDG (w/ mar. chop. crit.) |
| 2.1.3 | markTrace | ESDG (w/ mar. slic. crit.), ESDG (w/ mar. chop. crit.), trace | ESDG (w/ mar. slic. crit., mar. trace), ESDG (w/ mar. chop. crit., mar. trace) |
| 2.2 | computeStaticSlice | ESDG (w/ mar. slic. crit.) | ESDG (w/ mar. slice) |
| 2.2.1 | computeStaticBackwardSlice | ESDG (w/ mar. slic. crit.) | ESDG (w/ mar. slice) |
| 2.2.2 | computeStaticForwardSlice | ESDG (w/ mar. slic. crit.) | ESDG (w/ mar. slice) |
| 2.3 | computeDynamicSlice | ESDG (w/ mar. slic. crit., mar. trace) | ESDG (w/ mar. slice) |
| 2.3.1 | computeDynamicBackward-Slice | ESDG (w/ mar. slic. crit., mar. trace) | ESDG (w/ mar. slice) |
| 2.3.2 | computeDynamicForward-Slice | ESDG (w/ mar. slic. crit., mar. trace) | ESDG (w/ mar. slice) |
| 2.4 | computeStaticChop | ESDG (w/ mar. chop. crit.) | ESDG (w/ mar. chop) |
| 2.5 | computeDynamicChop | ESDG (w/ mar. chop. crit., mar. trace) | ESDG (w/ mar. chop) |
| 2.6 | computeExecutable-BackwardSlice | ESDG (w/ mar. slice) | ESDG (w/ mar. slice) |
| 3 | convertESDGToCode | ESDG (w/ mar. slice or chop) | sliced program code |

Abbildung 4.10.: Tabellarischer Überblick über die Slicing-Dienste samt Ein- und Ausgaben.

# 5. Datenmodell

Dieses Kapitel beschreibt die zwischen den Diensten (siehe Kapitel 4) fließenden Daten mit Hilfe von Graphenschemata, aus denen sich die konkreten, ein bestimmtes Programm beschreibenden Graphen instanziieren lassen. Zur Veranschaulichung werden Beispielgraphen verwendet, die sich auf das Programm in Abbildung 3.1 beziehen.

Es ist wichtig zu verstehen, dass die Graphenschemata *Sichten* auf ein Gesamtschema darstellen, so dass sich letzteres aus der Vereinigung der verschiedenen Sichten ergibt. Somit sind gleichnamige Klassen, d.h. Knotentypen, in verschiedenen Schemata als identisch anzusehen. Beispielsweise handelt es sich bei einem die Zuweisung `p = &a` in Zeile 9 des Beispielprogramms repräsentierenden Knoten des abstrakten Syntaxgraphen und dem entsprechenden Knoten im Kontrollflussgraphen um denselben Knoten.

Die Abfolge der nächsten Abschnitte entspricht der Reihenfolge der Berechnung der durch die Schemata beschriebenen Graphen. Dabei enthalten die Abbildungen nur solche Strukturen, die im Vergleich zu den vorher dargestellten Schemata neu hinzugekommen sind.

Im kommenden Abschnitt werden zunächst einige Hinweise zu der in den Betrachtungen der Schemata verwendeten Notation gegeben.

## 5.1. Hinweise zur Notation

- $V_T$ bezeichnet die Menge aller Knoten des Typs $T$ oder eines seiner Untertypen.
- $\bullet_T$ bezeichnet einen (unbenannten) Knoten des Typs $T$.
- $\xrightarrow{i}_T^{A=a,B=b,\ldots}$ und $\xleftarrow{i}_T^{A=a,B=b,\ldots}$ bezeichnen die Menge derjenigen Kanten des Typs $T$, dessen Attribute $A, B, \ldots$ die Werte $a, b, \ldots$ annehmen. Bei Weglassen eines Attributs sind Kanten unabhängig von seinem Wert in der Menge vertreten. Werden die Attribute und $T$ weggelassen ($\rightarrow$, $\leftarrow$), sind beliebige Kanten gemeint.

  Bei geordneten Kantenmengen ist $i$ die Ordnungszahl der Kante, die die Position der Kante innerhalb der Menge angibt. Dabei sind die Kanten geordnet bezogen auf denjenigen Knoten, aus dem sie ausgehen. Die erste Kante in einer Menge erhält die Ordnungszahl 1.

Die Ordnungszahl $i$ wird weggelassen, sofern sie für eine Definition keine Rolle spielt. $\xrightarrow{last}$ steht für die letzte Kante in der geordneten Menge.

- $\rightarrow^*$ und $(\bullet_T \rightarrow)^*$ bilden die transitive Hülle einer Kantenmenge bzw. einer Abfolge von Knoten des Typs $T$ und ausgehender Kanten.
- $v \xrightarrow{i}_T = \{\}$ bedeutet, dass von dem Knoten $v$ keine i-te Kante des Typs $T$ ausgeht.
- $V_{G.T}$, $\rightarrow_{G.T}$, $\leftarrow_{G.T}$ geben, zur besseren Lesbarkeit, die Sicht $G$ an, in der ein Typ $T$ eingeführt wird. So steht z.B. $\rightarrow_{ECG.uses}$ für den Kantentyp *uses* im ECG.

## 5.2. Abstrakter Syntaxgraph (*ASG*)

**Zielsetzung**: Der abstrakte Syntaxgraph stellt die Syntax des Programmcodes dar. Er dient als Grundlage für die Berechnung der nachfolgend behandelten Graphen.

**erzeugt von**:
- *1.1 computeAbstractSyntaxGraph* aus dem Programmcode

**benutzt von**:
- *1.2 computeAugmentedControlFlowGraphs* zur Berechnung der erweiterten Kontrollflussgraphen
- *1.3 computePointsToGraph* zur Berechnung der PGs
- *1.4.1 computeCallGraph* zur Berechnung des Aufrufgraphen[1]
- *1.4.2 computeDefUseInformation* zur Ermittlung der von einer Methode definierten und benutzten Variablen
- *1.5.1 computeBasicESDG* zur Berechnung der Knoten des ESDG
- *1.5.2 computeInterMethodEdges* zur Berechnung der Aufruf-, Parameter-in- und Parameter-out-Kanten
- *1.5.4 computeDataFlowEdges* zur Berechnung der Datenfluss- und Deklarationskanten

Der ASG stellt den Programmcode nach Durchführung von Techniken des Compilerbaus wie lexikalischer und syntaktischer Analyse dar. Das Schema des ASG in Abbildung 5.2 lässt sich mit folgenden Heuristiken direkt aus der Grammatik von EL herleiten, indem

- Symbole als Klassen,
- Konkatenationen durch Aggregationen oder, falls sich ein Symbol aus ausschließlich einem anderen Symbol ableiten lässt, Kompositionen

[1] Es werden hier nur die nicht weiter zerlegbaren Dienste aufgeführt.

- Alternativen durch Spezialisierungen,
- Wiederholungen durch Verwendung der Kardinalität *0..** und
- Optionen durch Verwendung der Kardinalität *0..1*

dargestellt werden.

Eine Bearbeitung des ASG-Schemas nach Anwendung der oben angeführten Heuristiken zur Verbesserung der Lesbarkeit ist zusätzlich möglich. So werden etwa *identifier* und *value* als Attribute statt als eigenständige Klassen dargestellt und Zeigervariablen mit Hilfe des booleschen Attributs *isPointer* gekennzeichnet. Außerdem existiert in einem ASG kein Pendant für dem Parsen dienende Terminale wie "(", ")" oder "extends".

Eine Besonderheit ist die Gestaltung der Vererbungsbeziehung zwischen zwei Klassen eines Programms als einfache Assoziation anstatt als Aggregation, da hier nicht von einer Beziehung im Sinne von „ist Teil von" die Rede sein kann. Weiterhin werden zugunsten einer höheren Anschaulichkeit die Klassen *variableDeclaration* und *call* als Oberklassen von *localVariableDeclaration*, *memberVariableDeclaration* und *formalParameter* bzw. von *callStatement* und *functionCall* gebildet. Zudem wird die Aggregation *hasAsTest* an *compoundExpressionWithTest* anstatt an ihre Unterklassen geknüpft.

An dieser Stelle wird auf weitere Ausführungen zu den konkreten Sprachelementen, mit Verweis auf Abschnitt 3.3, verzichtet. Abbildung 5.1 zeigt den abstrakten Syntaxgraphen der Methode *m1* des Beispielprogramms in Abbildung 3.1.

## 5.3. Erweiterter Kontrollflussgraph (*ACFG*)

**Zielsetzung**: Die aus dem abstrakten Syntaxgraphen herleitbaren erweiterten Kontrollflussgraphen enthalten, entsprechend den „normalen", nicht erweiterten CFGs, alle möglichen Abarbeitungsreihenfolgen der in einem Methodenrumpf enthaltenen lokalen Variablendeklarationen und Anweisungen. Der ACFG besitzt jedoch zusätzliche Kontrollflusskanten, die zur Berechnung der Kontrollkanten (siehe Abschnitt 5.6.3) notwendig sind.

**erzeugt von**:
- *1.2 computeAugmentedControlFlowGraphs* aus dem ASG

**benutzt von**:
- *1.5.3 computeControlDependenceEdges* zur Berechnung der Kontrollkanten
- *1.5.4 computeDataFlowEdges* zur Berechnung der Datenfluss- und Deklarationskanten
- *2.1.1 markSlicingCriterion* zur Markierung des Slicing-Kriteriums, falls die Variablenmenge $X$ des Slicing-Kriteriums $< v, X >$ nicht der Menge aller von $v$ benutzten oder definierten Variablen entspricht

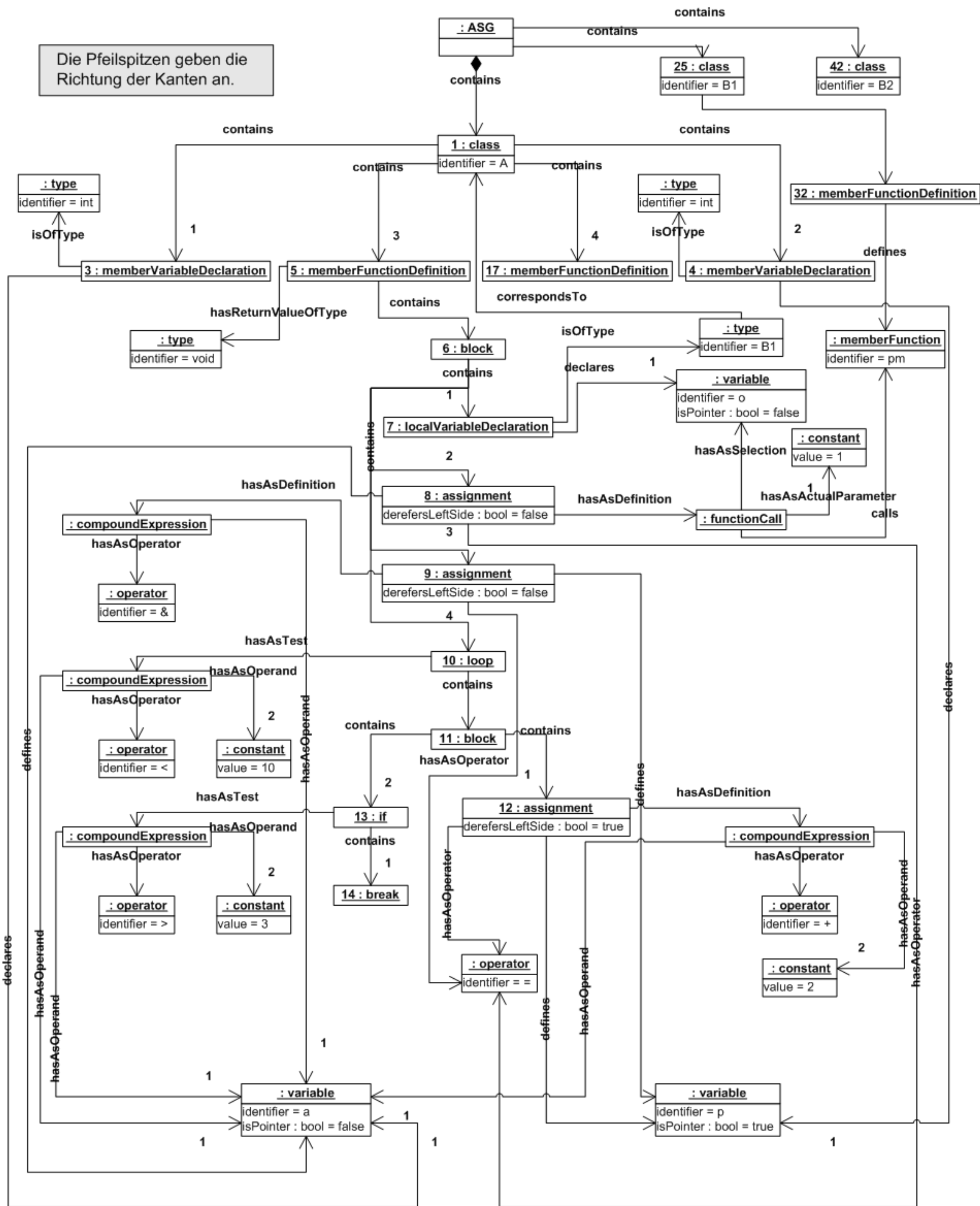


Abbildung 5.1.: Abstrakter Syntaxgraph des Beispielprogramms aus Beispiel 3.1. Aus Gründen der Übersichtlichkeit ist nur die Methode *m1* der Klasse *A* dargestellt.

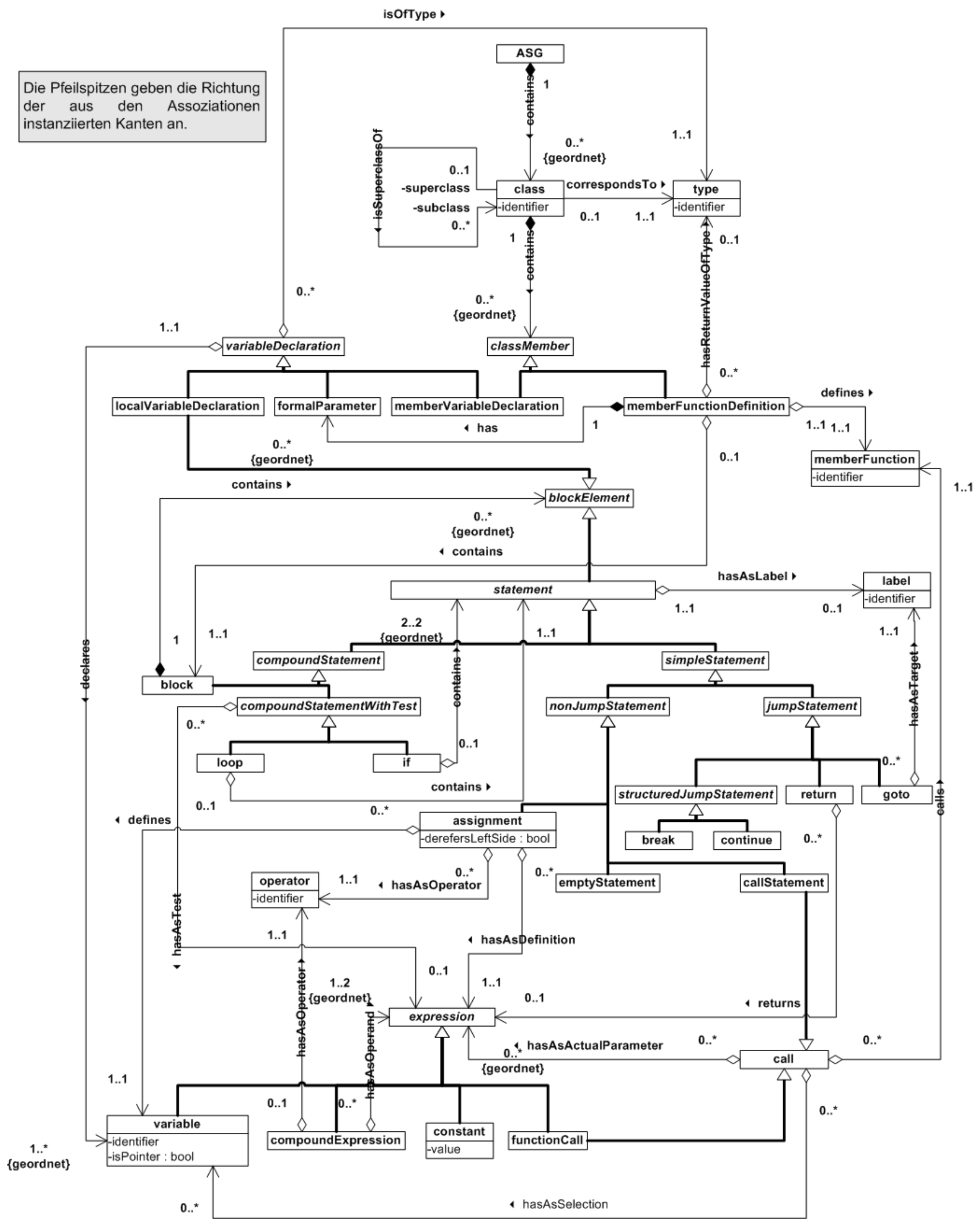


Abbildung 5.2.: Schema des ASG

Ein Beispiel eines ACFG, basierend auf dem Beispielprogramm in Abbildung 3.1, ist Abbildung 5.3 zu entnehmen.

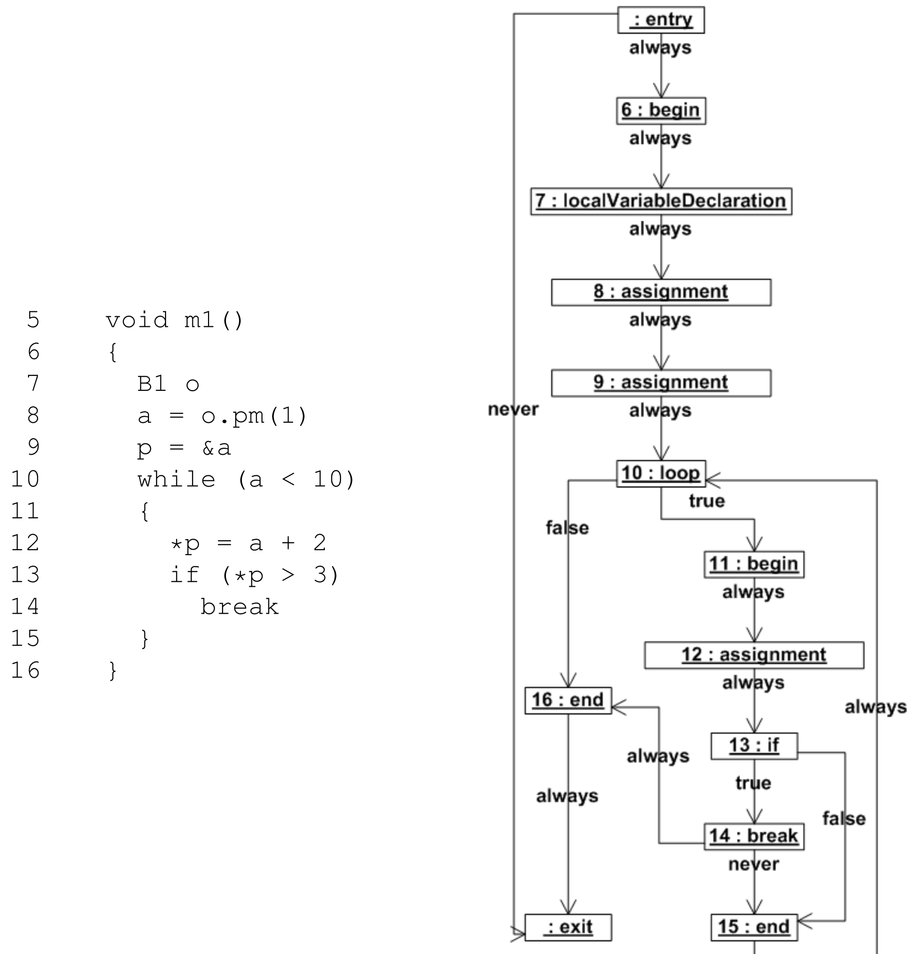


Abbildung 5.3.: Erweiterter Kontrollflussgraph der Methode *m1* des Beispielprogramms aus Beispiel 3.1. Es sind nur Kontrollflusskanten dargestellt.

Das Schema der Kontrollflussgraphenmenge (*ACFGSet*), die für jede im Programm enthaltene Methode einen ACFG enthält, ist in Abbildung 5.4 dargestellt. Wie aus dieser ersichtlich, ist die durch einen ACFG beschriebene Methode mit dieser über eine *describes*-Kante assoziiert.

Neben den beiden in jedem ACFG enthaltenen *entry*- und *exit*-Knoten können nur lokale Variablendeklarationen (*localVariableDeclaration*), einfache Anweisungen mit Test (*simpleStatementWithTest*) und einfache Anweisungen (ohne Test) (*simpleStatement*) Knoten des Graphen sein. Die beiden Anweisungsarten werden im Schema als Subklassen von *statement* dargestellt. Zwischen den Knoten können Kontrollflusskanten

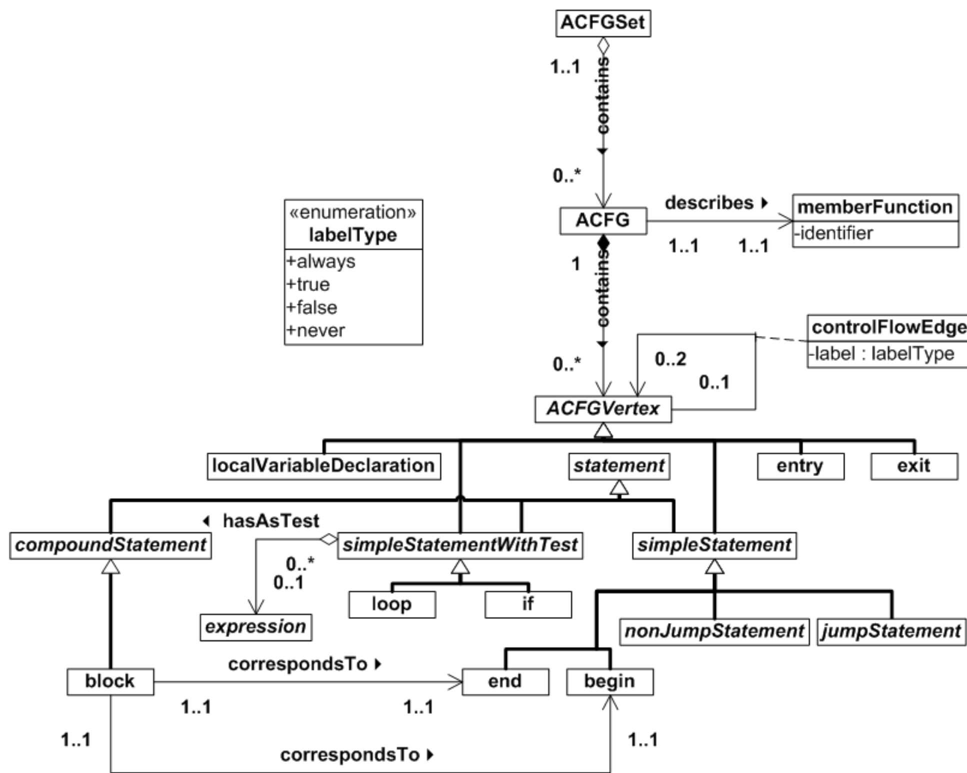


Abbildung 5.4.: Schema der ACFG-Menge

(*controlFlowEdge*) verlaufen, deren Attribut *label* die Werte *always*, *true*, *false* oder *never* annehmen kann.

Eine Kontrollflusskante von Knoten $s$ nach Knoten $t$ kann bedeuten, dass

- $t$ immer unmittelbar nach $s$ ausgeführt wird (Kante mit *always* beschriftet)
- $t$ unmittelbar nach $s$ ausgeführt wird, sofern eine Ausdrucksauswertung in $s$ zu wahr ausgewertet wird (Kante mit *true* beschriftet)
- $t$ unmittelbar nach $s$ ausgeführt wird, sofern eine Ausdrucksauswertung in $s$ zu falsch ausgewertet wird (Kante mit *false* beschriftet)
- $t$ unmittelbar nach der Sprunganweisung $s$ abgearbeitet würde, sofern es nicht zur Ausführung von $s$ kommt (Kante mit *never* beschriftet)

Im Folgenden wird das Vorhandensein von Kontrollflusskanten jedoch anhand der Syntax beschrieben.

Im Unterschied zum ASG existieren im ACFG keine zusammengesetzten Anweisungen, sondern die einfachen Anweisungen mit Test entsprechen den „Köpfen“ der zusammengesetzten Anweisungen mit Test aus der Grammatik, d.h. *“while (” [expression] “)”* oder *“if (” [expression] “)”*. Erstere weist eine mit *true* attributierte Kontrollflusskante zu der ersten Anweisung im Schleifenrumpf und eine mit *false* attributierte Kante zu dem ersten dem Rumpf nachfolgenden Methodenelement auf. Die ausgehende *true*-Kontrollflusskante eines *if*-Knotens verläuft zu der ersten mit dem Knoten im ASG verbundenen Anweisung, die *false*-Kante zu der zweiten, dem *“else”* nachfolgenden, Anweisung. Existiert letztere nicht, so geht die Kante in das nächste Element des Blocks ein, welcher auch die if-Anweisung enthält (oder in ein *end*).

Die Spezialisierungen von *simpleStatement* sind mit denjenigen im abstrakten Syntaxgraphen weitgehend identisch. Als Subklassen kommen hier zwei Anweisungen für Beginn und Ende eines Blocks (*begin* bzw. *end*) hinzu, die über *correspondsTo*-Kanten mit dem entsprechenden Block verknüpft sind. Einfache Anweisungen besitzen genau eine mit dem Wert *always* attributierte, ausgehende Kontrollflusskante zu dem nächsten Methodenelement bzw. zum Sprungziel. Sprunganweisungen (*jumpStatement*) verfügen darüber hinaus über eine mit *never* attributierte Kante zu dem Element, welches bei Nichtexistenz der Sprunganweisung abgearbeitet würde.

Ein Blockende als letzte in der Methode enthaltene Anweisung besitzt stattdessen allerdings eine *always*-Kante zum Ausgang. Der *entry*-Knoten besitzt zwei ausgehende Kontrollflusskanten: eine mit *always* attributierte zu dem zuerst abzuarbeitenden Methodenelement, d.h. einer begin-Anweisung, und eine mit *never* attributierte Kante zu dem *exit*-Knoten. Diese Kanten werden bei der Ermittlung der Kontrollkanten (siehe Abschnitt 5.6.3) benötigt.

Der Rest dieses Abschnitts beinhaltet eine exakte Definition der Knoten und Kanten des ACFG.

**Definition (Menge der erweiterten Kontrollflussgraphen)**
Sei $G = (V, E)$ ein Graph, der einen abstrakten Syntaxgraphen enthält.

Ein um die Menge der erweiterten Kontrollflussgraphen erweiterter Graph $G' = (V', E')$ geht aus $G$ hervor mit

$$\exists! acfgSet \in V'_{ACFGSet.ACFGSet} \Big( \forall m \in V_{ASG.memberFunction} \Big( \exists! acfg \in V'_{ACFGSet.ACFG}$$

$$\Big( acfgSet \rightarrow'_{ACFGSet.contains} acfg \rightarrow'_{ACFGSet.describes} m \tag{1}$$

$$\wedge \quad \forall elem \in V_{ASG.blockElement} \backslash V_{ASG.block} \tag{2}$$

$$(m \leftarrow_{ASG.defines} \bullet_{ASG.memberFunctionDefinition} \rightarrow^*_{ASG.contains} elem$$

$$\Rightarrow acfg \rightarrow'_{ACFGSet.contains} elem)$$

$$\wedge \quad \forall b \in V_{ASG.block} (\exists! bb \in V'_{ACFGSet.begin}, be \in V'_{ACFGSet.end} \tag{3}$$

$$(bb \leftarrow'_{ACFGSet.correspondsTo} b \rightarrow'_{ACFGSet.correspondsTo} be$$

$$\wedge (m \leftarrow_{ASG.defines} \bullet_{ASG.memberFunctionDefinition} \rightarrow^*_{ASG.contains} b$$

$$\Rightarrow bb \leftarrow'_{ACFGSet.contains}\; acfg \rightarrow'_{ACFGSet.contains}\; be)))$$

$$\wedge \quad \exists! entry \in V'_{ACFGSet.entry}, exit \in V'_{ACFGSet.exit} \quad (4)$$

$$\big(entry \leftarrow'_{ACFGSet.contains}\; acfg \rightarrow'_{ACFGSet.contains}\; exit\big)\Big)\Big)\Big)$$

(1) Zuordnung einer Methode $m$ zu dem sie beschreibenden ACFG $acfg$ über eine *describes*-Kante

(2) Zuordnung der Elemente $elem$ einer Methode $m$ zu dem die Methode beschreibenden ACFG $acfg$ über *contains*-Kanten

(3) Zuordnung eines *begin*-Knotens $bb$ und eines *end*-Knotens $be$ zu einem in $m$ enthaltenen Block $b$ über jeweils eine *correspondsTo*-Kante und Zuordnung von $bb$ und $be$ über jeweils eine *contains*-Kante zu demjenigen ACFG $acfg$, der $m$ beschreibt

(4) Zuordnung eines *entry*-Knotens $entry$ und eines *exit*-Knotens $exit$ zu einem ACFG $acfg$ über jeweils eine *contains*-Kante

Abbildung 5.5 zeigt die zulässigen, zwischen den Knoten eines ACFG verlaufenden Kontrollflusskanten. Dabei enthält die erste Spalte die Typen der Ausgangsknoten *s* und die erste Zeile die Typen der Eingangsknoten *t* der Kanten. Die Zellen enthalten „—“, falls zwischen Knoten der betreffenden Typen keine Kontrollflusskanten vorhanden sein können. Andernfalls zeigen die römischen Zahlen den entsprechenden Aufzählungspunkt der Definition an (siehe unten). Dieser beschreibt die Bedingung, unter der eine Kontrollflusskante zwischen *s* und *t* verläuft.

## Kontrollflusskanten mit dem Wert *always* für das Attribut *label*

$s \rightarrow'^{label=always}_{ESDG.controlFlowEdge}\; t \Leftrightarrow$ eine der folgenden Bedingungen gilt:

### mit den *entry*- und *exit*-Knoten verbundene Kanten

(i) von den *entry*-Knoten ausgehende Kanten zu den *begin*-Knoten der in der Schachtelungshierarchie an erster Stelle stehenden Blöcke[2] einer jeden Methode:

$s \in V'_{ACFGSet.entry}, t \in V'_{ACFGSet.begin}$
mit $t \leftarrow'_{ACFGSet.correspondsTo} \bullet_{ASG.block} \leftarrow_{ASG.contains}$
$\bullet_{ASG.memberFunctionDefinition} \rightarrow_{ASG.defines} \bullet_{ASG.memberFunction}$
$\leftarrow'_{ACFGSet.describes} \bullet_{ACFGSet.ACFG} \rightarrow'_{ACFGSet.contains}\; s$

[2] Blöcke gehören nicht zu einem ACFG, sondern sind ausschließlich in dem ASG enthalten. Stattdessen verlaufen Kanten von und zu Beginn und Ende eines Blocks.

| s \ t | *entry* | *exit* | *l.Var.Dec.* | *nonJump.* | *jump.* | *loop* | *if* | *begin* | *end* |
|---|---|---|---|---|---|---|---|---|---|
| *entry* | — | (xxiv) | — | — | — | — | — | (i) | — |
| *exit* | — | — | — | — | — | — | — | — | — |
| *l.Var.Dec.* | — | — | (vii) | (vii) | (vii) | (vii) | (vii) | (vii) | (viii) |
| *nonJump.* | — | — | (vii) | (vii) | (vii) | (vii)<br>(ix) | (vii) | (vii) | (viii) |
| *break* | — | — | (x)<br>(xxv) | (x)<br>(xxv) | (x)<br>(xxv) | (x)<br>(xxv)<br>(xxvii) | (x)<br>(xxv) | (x)<br>(xxv) | (xi)<br>(xxvi) |
| *continue* | — | — | (xxv) | (xxv) | (xxv) | (xii)<br>(xxv)<br>(xxvii) | (xxv) | (xxv) | (xxvi) |
| *goto* | — | — | (xiii)<br>(xxv) | (xiii)<br>(xxv) | (xiii)<br>(xxv) | (xiii)<br>(xxv)<br>(xxvii) | (xiii)<br>(xxv) | (xiii)<br>(xxv) | (xxvi) |
| *return* | — | (xiv) | (xxv) | (xxv) | (xxv) | (xxv)<br>(xxvii) | (xxv) | (xxv) | (xxvi) |
| *loop* | — | — | (xvii) | (xv)<br>(xvii) | (xv)<br>(xvii) | (xv)<br>(xvii)<br>(xix) | (xv)<br>(xvii) | (xv)<br>(xvii) | (xviii) |
| *if* | — | — | (xxi) | (xvi)<br>(xx)<br>(xxi) | (xvi)<br>(xx)<br>(xxi) | (xvi)<br>(xx)<br>(xxi)<br>(xxiii) | (xvi)<br>(xx)<br>(xxi) | (xvi)<br>(xx)<br>(xxi) | (xxii) |
| *begin* | — | — | (iii) | (iii) | (iii) | (iii) | (iii) | (iii) | (iv) |
| *end* | — | (ii) | (v) | (v) | (v) | (v)<br>(ix) | (v) | (v) | (vi) |

Abbildung 5.5.: Übersicht über die zulässigen Kontrollflusskanten

(ii) von den Enden (*end*) der in der Schachtelungshierarchie an erster Stelle stehenden Blöcke in die *exit*-Knoten eingehende Kanten:

$s \in V'_{ACFGSet.end}, t \in V'_{ACFGSet.exit}$
mit $s \leftarrow'_{ACFGSet.correspondsTo} \bullet_{ASG.block} \leftarrow_{ASG.contains}$
$\bullet_{ASG.memberFunctionDefinition} \rightarrow_{ASG.defines} \bullet_{ASG.memberFunction}$
$\leftarrow'_{ACFGSet.describes} \bullet_{ACFGSet.ACFG} \rightarrow'_{ACFGSet.contains} t$

**von dem Beginn eines Blocks (*begin*) ausgehende Kanten**

(iii) zu dem ersten Blockelement innerhalb des Blocks[3] (*ACFGVertex*-Knoten[4]):

$$s \in V'_{ACFGSet.begin}, t \in V'_{ACFGSet.ACFGVertex} \backslash V'_{ACFGSet.end}$$
mit $s \leftarrow'_{ACFGSet.correspondsTo} \bullet_{ASG.block} \xrightarrow{1}_{ASG.contains} t$
$\lor\ s \leftarrow'_{ACFGSet.correspondsTo} \bullet_{ASG.block} \xrightarrow{1}_{ASG.contains} \bullet_{ASG.block} \rightarrow'_{ACFGSet.correspondsTo} t$

(iv) zu dem Ende des Blocks, falls der Block leer ist:

$$s \in V'_{ACFGSet.begin}, t \in V'_{ACFGSet.end}$$
mit $\exists b \in V_{ASG.block}$
$(s \leftarrow'_{ACFGSet.correspondsTo} b \rightarrow'_{ACFGSet.correspondsTo} t \land b \xrightarrow{1}_{ASG.contains} = \{\})$

**von dem Ende eines Blocks ausgehende Kanten**

(v) zu dem nachfolgenden Blockelement des in der Schachtelungshierarchie unmittelbar übergeordneten Blocks[5]:

$$s \in V'_{ACFGSet.end}, t \in V'_{ACFGSet.ACFGVertex} \backslash V'_{ACFGSet.end}$$
mit $\exists i \in \mathbb{N}$
$(t \xleftarrow{i+1}_{ASG.contains} \bullet_{ASG.block} \xrightarrow{i}_{ASG.contains} (\bullet_{ASG.if} \rightarrow_{ASG.contains})^* \bullet_{ASG.block} \rightarrow'_{ACFGSet.correspondsTo} s$
$\lor\ t \leftarrow'_{ACFGSet.correspondsTo} \bullet_{ASG.block} \xleftarrow{i+1}_{ASG.contains} \bullet_{ASG.block} \xrightarrow{i}_{ASG.contains} (\bullet_{ASG.if} \rightarrow_{ASG.contains})^* \bullet_{ASG.block} \rightarrow'_{ACFGSet.correspondsTo} s)$

(vi) zu dem Ende des unmittelbar übergeordneten Blocks, falls dieser kein nachfolgendes Blockelement besitzt:

$$s \in V'_{ACFGSet.end}, t \in V'_{ACFGSet.end}$$
mit $t \leftarrow'_{ACFGSet.correspondsTo} \bullet_{ASG.block} \xrightarrow{last}_{ASG.contains} (\bullet_{ASG.if} \rightarrow_{ASG.contains})^* \bullet_{ASG.block} \rightarrow'_{ACFGSet.correspondsTo} s$

[3] Die zweite Bedingung (nach dem $\lor$) bezieht sich auf den Fall, dass das erste Element ein Block ist und die Kante somit zu seinem Beginn verläuft.

[4] Hier wird der Typ *ACFGVertex* anstatt *blockElement* verwendet, weil letzterer auch Supertyp des nicht in einem ACFG vorkommenden Typs *compoundStatement* ist.

[5] der zu $s$ gehörende Block kann sich in einer beliebig tiefen, ausschließlich aus *if*-Anweisungen bestehenden, Schachtelungshierarchie befinden - das nachfolgende Blockelement $t$ entspricht dem der äußersten *if*-Anweisung folgenden Element

**von einer Nicht-Sprunganweisung oder einer lokalen Variablendeklaration (*nonJumpStatement*- und *localVariableDeclaration*) ausgehende Kanten**

(vii) zu dem nachfolgenden Blockelement:

$$s \in (V'_{ACFGSet.nonJumpStatement} \cup V'_{ACFGSet.localVariableDeclaration}),\\ t \in V'_{ACFGSet.ACFGVertex} \setminus V'_{ACFGSet.end}\\ \text{mit } \exists i \in \mathbb{N}\\ (t \overset{i+1}{\longleftarrow}_{ASG.contains} \bullet_{ASG.block} \overset{i}{\longrightarrow}_{ASG.contains} (\bullet_{ASG.if} \rightarrow_{ASG.contains})^* s\\ \vee t \leftarrow'_{ACFGSet.correspondsTo} \bullet_{ASG.block} \overset{i+1}{\longleftarrow}_{ASG.contains} \bullet_{ASG.block} \overset{i}{\longrightarrow}_{ASG.contains}\\ (\bullet_{ASG.if} \rightarrow_{ASG.contains})^* s)$$

(viii) zu dem Ende des unmittelbar übergeordneten Blocks, falls kein nachfolgendes Blockelement existiert:

$$s \in (V'_{ACFGSet.nonJumpStatement} \cup V'_{ACFGSet.localVariableDeclaration}), t \in V'_{ACFGSet.end}\\ \text{mit } t \leftarrow'_{ACFGSet.correspondsTo} \bullet_{ASG.block} \overset{last}{\longrightarrow}_{ASG.contains} (\bullet_{ASG.if} \rightarrow_{ASG.contains})^* s$$

**von einer Nicht-Sprunganweisung oder einem Blockende ausgehende Kanten**

(ix) zu einer übergeordneten *loop*-Anweisung, falls sich die Nicht-Sprunganweisung bzw. der entsprechende Block in einem Schleifenrumpf befinden:

$$s \in V'_{ACFGSet.nonJumpStatement} \cup V'_{ACFGSet.end}, t \in V'_{ACFGSet.loop}\\ \text{mit } t \rightarrow_{ASG.contains} (\bullet_{ASG.if} \rightarrow_{ASG.contains})^* s\\ \vee t \rightarrow_{ASG.contains} (\bullet_{ASG.if} \rightarrow_{ASG.contains})^* \bullet_{ASG.block} \rightarrow'_{ACFGSet.correspondsTo} s$$

**von einer Sprunganweisung (*jumpStatement*) ausgehende Kanten**

(x) von einer *break*-Anweisung zu dem der unmittelbar übergeordneten Schleife nachfolgenden Blockelement:

$$s \in V'_{ACFGSet.break}, t \in V'_{ACFGSet.ACFGVertex} \setminus V'_{ACFGSet.end}\\ \text{mit } \exists i \in \mathbb{N}\\ (t \overset{i+1}{\longleftarrow}_{ASG.contains} \bullet_{ASG.block} \overset{i}{\longrightarrow}_{ASG.contains} \bullet_{ASG.loop} \rightarrow_{ASG.contains}\\ (\bullet_{ASG.blockElement \setminus ASG.loop} \rightarrow_{ASG.contains})^* s\\ \vee t \leftarrow'_{ACFGSet.correspondsTo} \bullet_{ASG.block} \overset{i+1}{\longleftarrow}_{ASG.contains} \bullet_{ASG.block} \overset{i}{\longrightarrow}_{ASG.contains}\\ \bullet_{ASG.loop} \rightarrow_{ASG.contains} (\bullet_{ASG.blockElement \setminus ASG.loop} \rightarrow_{ASG.contains})^* s)$$

(xi) von einer *break*-Anweisung zu dem Ende eines Blocks, falls kein der unmittelbar übergeordneten Schleife nachfolgendes Blockelement existiert:

$$s \in V'_{ACFGSet.break}, t \in V'_{ACFGSet.end}\\ \text{mit } t \leftarrow'_{ACFGSet.correspondsTo} \bullet_{ASG.block} \overset{last}{\longrightarrow}_{ASG.contains} \bullet_{ASG.loop} \rightarrow_{ASG.contains}\\ (\bullet_{ASG.blockElement \setminus ASG.loop} \rightarrow_{ASG.contains})^* s$$

(xii) von einer *continue*-Anweisung zu der unmittelbar übergeordneten *loop*-Anweisung:

$s \in V'_{ACFGSet.continue}, t \in V'_{ACFGSet.loop}$
mit $t \rightarrow_{ASG.contains} (\bullet_{ASG.blockElement \setminus ASG.loop} \rightarrow_{ASG.contains})^{*}\ s$

(xiii) von einer *goto*-Anweisung zu der Anweisung mit dem entsprechenden *label*:

$s \in V'_{ACFGSet.goto}, t \in V'_{ACFGSet.ACFGVertex} \setminus V'_{ACFGSet.end}$
mit $s \rightarrow_{ASG.hasAsTarget} \bullet_{ASG.label} \leftarrow_{ASG.hasAsLabel}\ t$
$\vee\ s \rightarrow_{ASG.hasAsTarget} \bullet_{ASG.label} \leftarrow_{ASG.hasAsLabel} \bullet_{ASG.block}$
$\rightarrow'_{ACFGSet.correspondsTo}\ t$

(xiv) von einer *return*-Anweisung zu dem *exit*-Knoten:

$s \in V'_{ACFGSet.return}, t \in V'_{ACFGSet.exit}$
mit $s \leftarrow'_{ACFGSet.contains} \bullet_{ACFGSet.ACFG} \rightarrow'_{ACFGSet.contains}\ t)$

## Kontrollflusskanten mit dem Wert *true* für das Attribut *label*

$s \rightarrow'^{label=true}_{ESDG.labeledControlFlowEdge}\ t \Leftrightarrow$ eine der folgenden Bedingungen gilt:

(xv) von einer *loop*-Anweisung zu der in der Schleife enthaltenen Anweisung:

$s \in V'_{ACFGSet.loop}$,
$t \in V'_{ACFGSet.simpleStatementWithTest} \cup (V'_{ACFGSet.simpleStatement} \setminus V'_{ACFGSet.end})$
mit $s \rightarrow_{ASG.contains}\ t \vee s \rightarrow_{ASG.contains} \bullet_{ASG.block} \rightarrow'_{ACFGSet.correspondsTo}\ t$

(xvi) von einer *if*-Anweisung zu der ersten enthaltenen Anweisung:

$s \in V'_{ACFGSet.if}$,
$t \in V'_{ACFGSet.simpleStatementWithTest} \cup (V'_{ACFGSet.simpleStatement} \setminus V'_{ACFGSet.end})$
mit $s \overset{1}{\rightarrow}_{ASG.contains}\ t \vee s \overset{1}{\rightarrow}_{ASG.contains} \bullet_{ASG.block} \rightarrow'_{ACFGSet.correspondsTo}\ t$

## Kontrollflusskanten mit dem Wert *false* für das Attribut *label*

$s \rightarrow'^{label=false}_{ESDG.labeledControlFlowEdge}\ t \Leftrightarrow$ eine der folgenden Bedingungen gilt:

**von einer *loop*-Anweisung ausgehende Kanten**

(xvii) zu dem der Schleife nachfolgenden Blockelement:

$s \in V'_{ACFGSet.loop}, t \in V'_{ACFGSet.ACFGVertex} \backslash V'_{ACFGSet.end}$
mit $\exists i \in \mathbb{N}$
$(t \xleftarrow{i+1}_{ASG.contains} \bullet_{ASG.block} \xrightarrow{i}_{ASG.contains} (\bullet_{ASG.if} \rightarrow_{ASG.contains})^* s$
$\vee\, t \leftarrow'_{ACFGSet.correspondsTo} \bullet_{ASG.block} \xleftarrow{i+1}_{ASG.contains} \bullet_{ASG.block} \xrightarrow{i}_{ASG.contains}$
$(\bullet_{ASG.if} \rightarrow_{ASG.contains})^* s)$

(xviii) zu dem Ende des in der Schachtelungshierarchie unmittelbar übergeordneten Blocks, falls kein nachfolgendes Blockelement existiert:

$s \in V'_{ACFGSet.loop}, t \in V'_{ACFGSet.end}$
mit $t \leftarrow'_{ACFGSet.correspondsTo} \bullet_{ASG.block} \xrightarrow{last}_{ASG.contains} (\bullet_{ASG.if} \rightarrow_{ASG.contains})^* s$

(xix) zu einer unmittelbar übergeordneten *loop*-Anweisung:

$s \in V'_{ACFGSet.loop}, t \in V'_{ACFGSet.loop}$
mit $t \xrightarrow{i}_{ASG.contains} (\bullet_{ASG.if} \rightarrow_{ASG.contains})^* s$

**von einer *if*-Anweisung ausgehende Kanten**

(xx) zu der zweiten enthaltenen Anweisung:

$s \in V'_{ACFGSet.if}$,
$t \in V'_{ACFGSet.simpleStatementWithTest} \cup (V'_{ACFGSet.simpleStatement} \backslash V'_{ACFGSet.end})$
mit $s \xrightarrow{2}_{ASG.contains} t \vee s \xrightarrow{2}_{ASG.contains} \bullet_{ASG.block} \rightarrow'_{ACFGSet.correspondsTo} t$

(xxi) zu dem nachfolgenden Blockelement, falls die *if*-Anweisung nur eine Anweisung enthält:

$s \in V'_{ACFGSet.if}, t \in V'_{ACFGSet.ACFGVertex} \backslash V'_{ACFGSet.end}$
mit $\exists i \in \mathbb{N}$
$(t \xleftarrow{i+1}_{ASG.contains} \bullet_{ASG.block} \xrightarrow{i}_{ASG.contains} (\bullet_{ASG.if} \rightarrow_{ASG.contains})^* s$
$\wedge\, s \xrightarrow{2}_{ASG.contains} = \{\}$
$\vee\, t \leftarrow'_{ACFGSet.correspondsTo} \bullet_{ASG.block} \xleftarrow{i+1}_{ASG.contains} \bullet_{ASG.block} \xrightarrow{i}_{ASG.contains}$
$(\bullet_{ASG.if} \rightarrow_{ASG.contains})^* s \wedge s \xrightarrow{2}_{ASG.contains} = \{\})$

(xxii) zu dem Ende des unmittelbar übergeordneten Blocks, falls kein nachfolgendes Blockelement existiert und die *if*-Anweisung nur eine Anweisung enthält:

$s \in V'_{ACFGSet.if}, t \in V'_{ACFGSet.end}$
mit $t \leftarrow'_{ACFGSet.correspondsTo} \bullet_{ASG.block} \xrightarrow{last}_{ASG.contains} (\bullet_{ASG.if} \rightarrow_{ASG.contains})^* s$
$\wedge\, s \xrightarrow{2}_{ASG.contains} = \{\}$

(xxiii) zu einer unmittelbar übergeordneten *loop*-Anweisung, falls die *if*-Anweisung nur eine Anweisung enthält:

$s \in V'_{ACFGSet.if}, t \in V'_{ACFGSet.loop}$
mit $t \xrightarrow{i}_{ASG.contains} (\bullet_{ASG.if} \rightarrow_{ASG.contains})^* s \wedge s \xrightarrow{2}_{ASG.contains} = \{\})$

## Kontrollflusskanten mit dem Wert *never* für das Attribut *label*

$s \rightarrow'^{label=never}_{ESDG.augmentedControlFlowEdge} t \Leftrightarrow$ eine der folgenden Bedingungen gilt:

(xxiv) von einem *entry*- zu einem *exit*-Knoten:

$s \in V'_{ACFGSet.entry}, t \in V'_{ACFGSet.exit}$
mit $s \leftarrow'_{ACFGSet.contains} \bullet_{ACFGSet.ACFG} \rightarrow'_{ACFGSet.contains} t$

**von einer Sprunganweisung (*jumpStatement*) ausgehende Kanten**

(xxv) zu dem nachfolgenden Blockelement (*ACFGVertex*):

$s \in V'_{ACFGSet.jumpStatement}, t \in V'_{ACFGSet.ACFGVertex} \setminus V'_{ACFGSet.end}$
mit $\exists i \in \mathbb{N}$
$(t \xleftarrow{i+1}_{ASG.contains} \bullet_{ASG.block} \xrightarrow{i}_{ASG.contains} (\bullet_{ASG.if} \rightarrow_{ASG.contains})^* s$
$\vee\ t \leftarrow_{ACFGSet.correspondsTo} \bullet_{ASG.block} \xleftarrow{i+1}_{ASG.contains} \bullet_{ASG.block} \xrightarrow{i}_{ASG.contains}$
$(\bullet_{ASG.if} \rightarrow_{ASG.contains})^* s)$

(xxvi) zu dem Ende des in der Schachtelungshierarchie unmittelbar übergeordneten Blocks, falls kein nachfolgendes Blockelement existiert:

$s \in V'_{ACFGSet.jumpStatement}, t \in V'_{ACFGSet.end}$
mit $t \leftarrow'_{ACFGSet.correspondsTo} \bullet_{ASG.block} \xrightarrow{last}_{ASG.contains} (\bullet_{ASG.if} \rightarrow_{ASG.contains})^* s$

(xxvii) zu einer unmittelbar übergeordneten *loop*-Anweisung, falls sich in der Schachtelungshierarchie kein Block zwischen *loop*- und Sprunganweisung befindet:

$s \in V'_{ACFGSet.jumpStatement}, t \in V'_{ACFGSet.loop}$
mit $t \xrightarrow{i}_{ASG.contains} (\bullet_{ASG.if} \rightarrow_{ASG.contains})^* s$ □

Die erweiterten Kontrollflussgraphen werden zur Berechnung des erweiterten Systemabhängigkeitsgraphen sowie, eventuell, zur Markierung des Slicing-Kriteriums im ESDG benötigt

## 5.4. Points-to-Graph (*PG*)

**Zielsetzung**: Die Funktion des Points-to-Graphen besteht darin, eine konservative Abschätzung der „zeigt auf“-Beziehungen zwischen Variablen innerhalb einer Methode darzustellen. Diese Variablen können entweder Attribute, formale Parameter oder lokale Variablen sein. *Konservative Abschätzung* bedeutet hier, dass die tatsächlich vorhandenen Beziehungen oder eine Obermenge davon, keinesfalls aber eine Untermenge abgebildet wird.
**erzeugt von**:
- *1.3 computePointsToGraphs* aus dem ASG

**benutzt von**:
- *1.4.1 computeCallGraph* zur Berechnung des Aufrufgraphen
- *1.4.2 computeDefUseInformation* zur Ermittlung der von einer Methode definierten und benutzten Variablen
- *1.5.4 computeDataFlowEdges* zur Berechnung der Datenfluss- und Deklarationskanten

Abbildung 5.6 zeigt das Schema der Menge der Points-to-Graphen (*PGSet*), das je einen Graphen (*PG*) mit jeder Methode (*memberFunction*) assoziiert. Dies impliziert, dass das hier verwendete Modell sich nur für fluss- und kontextinsensitive Zeigeranalyseverfahren eignet.

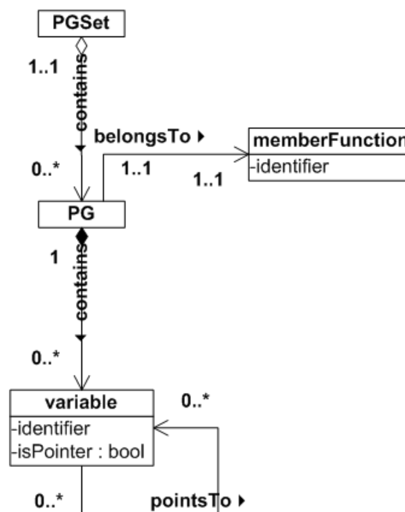


Abbildung 5.6.: Schema der PG-Menge

Enthält eine Variable (*variable*) in einer Ausführung der Methode möglicherweise die Speicherstelle einer anderen Variable als Wert, so verläuft zwischen den repräsentierenden Knoten eine *pointsTo*-Kante. Geht von einem Knoten keine *pointsTo*-Kante aus, so zeigt die Variable auf nichts.

## 5.5. Aufrufgraph/erweiterter Aufrufgraph (*CG*/*ECG*)

**Zielsetzung**: Der Aufrufgraph bildet die Aufrufbeziehungen zwischen den Methoden eines Programms ab. Der ECG zeigt außerdem die von einer Methode definierten und benutzten Variablen auf. Bei benutzten Variablen kann es sich um Attribute, formale Eingabeparameter oder Variablen, auf die ein Eingabeparameter zeigt, handeln. Definierte Variablen sind der Rückgabewert, von der Methode definierbare Attribute oder Variablen, auf die eine als Eingabeparameter übergebene Zeigervariable zeigt.
**erzeugt von**:

- *1.4.1 computeExtendedCallGraph*; Berechnung des Aufrufgraphen ohne definierte und benutzte Variablen aus dem abstrakten Syntaxgraphen und den Points-to-Graphen
- *1.4.2 computeDefUseInformation*; Erweiterung der Aufrufgraphen um die von den Methoden definierten und benutzten Variablen. Die Berechnung erfolgt mit Hilfe des ASG und den Points-to-Graphen.

**benutzt von**:

- *1.5.2 computeInterMethodEdges* zur Berechnung der Aufruf-, Parameter-in- und Parameter-out-Kanten
- *1.5.4 computeDataFlowEdges* zur Berechnung der Datenfluss- und Deklarationskanten

Sowohl der CG als auch der ECG (*extendedCallGraph*, Schema siehe Abbildung 5.7) verfügen für jede Methode über einen *memberFunction*-Knoten. Diese werden mit Hilfe von *contains*-Kanten mit Aufrufknoten (*call*) verbunden. Da sich bei Aufrufanweisungen und Funktionsaufrufen unter Umständen erst zur Laufzeit entscheidet, welche Methode aufgerufen wird (Polymorphie), können von einem Aufrufknoten mehrere *canCall*-Kanten zu jeweils verschiedenen *memberFunction*-Knoten ausgehen.

Zur Darstellung der definierten und benutzten Variablen existieren im ECG zusätzliche *defines*- und *uses*-Kanten zwischen den *memberFunction*-Knoten und den Variablen. Eine von einem *memberFunction*- zu einem *variable*-Knoten verlaufende *defines*-Kante bedeutet, dass die entsprechende Variable von der Methode definiert wird. Die *uses*-Kanten werden analog verwendet. Dabei sind auch indirekt, d.h. von einer aufgerufenen Methode, definierte und benutzte Attribute zu berücksichtigen, nicht jedoch lokale Variablen, die in der Methode selbst deklariert werden.

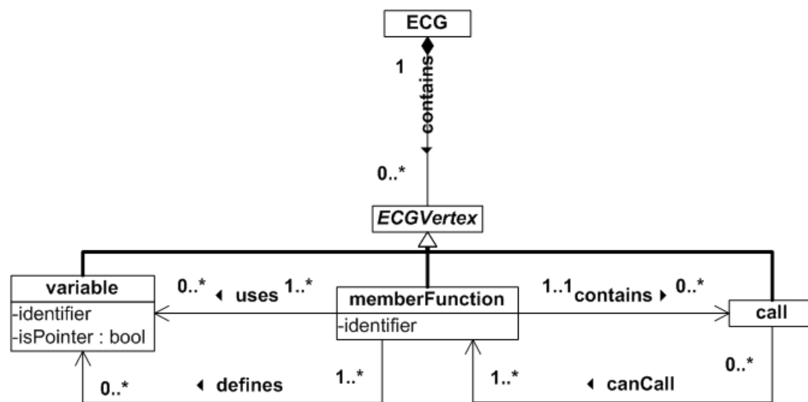


Abbildung 5.7.: Schema des ECG

Beispielsweise lautet die Menge der von der Methode *m1* des Beispielprogramms in Abbildung 3.1 definierten als auch die der benutzten Variablen $\{a, b, p\}$.

Für den Rückgabewert wird eine Hilfsvariable mit dem Bezeichner *returnValue*[6] erzeugt und über eine *defines*-Kante mit dem *memberFunction*-Knoten verbunden.

## 5.6. Erweiterter Systemabhängigkeitsgraph (*ESDG*)

**Zielsetzung**: Der erweiterte Systemabhängigkeitsgraph vereinigt die Knoten der Kontrollflussgraphen der einzelnen Methoden und verbindet sie mit Hilfe spezieller, unter anderem auf den Kontrollflusskanten basierender Kanten miteinander. Der ESDG ist die Grundlage für die Verfahren, die das Slicing im eigentlichen Sinne, d.h. abzüglich der Aufbereitung des Programmcodes, durchführen.
**erzeugt von**:

- *1.5.1 computeBasicESDG*; Erstellung der Knoten des ESDG aus dem ASG und dem ECG
- *1.5.2 computeInterMethodEdges*; Hinzufügen der Aufruf-, Parameter-in- und Parameter-out-Kanten, welche mit Hilfe des ASG und des ECG berechnet werden
- *1.5.3 computeControlDependenceEdges*; Hinzufügen der Kontrollkanten, welche mit Hilfe der ACFGs berechnet werden
- *1.5.4 computeDataFlowEdges*; Hinzufügen der Datenfluss- und Deklarationskanten, welche mit Hilfe des ASG, der ACFGs und den PGs berechnet werden
- *1.5.5 computeSummaryEdges*; Hinzufügen der transitiven Kanten

[6] Der Bezeichner darf von keiner anderen Variable verwendet werden.

**benutzt von**:

- *2.1.1 markSlicingCriterion* zur Markierung des Slicing-Kriteriums im ESDG
- *2.1.2 markChoppingCriterion* zur Markierung des Chopping-Kriteriums im ESDG
- *2.1.3 markTrace* zur Markierung der Trace einer Programmausführung im ESDG
- *2.2.1 computeStaticBackwardSlice* zur Berechnung einer statischen Rückwärtsslice. Das Slicing-Kriterium muss im ESDG markiert sein. Die in der Slice enthaltenen Knoten werden im ESDG durch Setzen eines Attributs gekennzeichnet.
- *2.2.2 computeStaticForwardSlice* zur Berechnung einer statischen Vorwärtsslice. Das Slicing-Kriterium muss im ESDG markiert sein. Die in der Slice enthaltenen Knoten werden im ESDG durch Setzen eines Attributs gekennzeichnet.
- *2.3.1 computeDynamicBackwardSlice* zur Berechnung einer dynamischen Rückwärtsslice. Das Slicing-Kriterium und die Trace müssen im ESDG markiert sein. Die in der Slice enthaltenen Knoten werden im ESDG durch Setzen eines Attributs gekennzeichnet.
- *2.3.2 computeDynamicForwardSlice* zur Berechnung einer dynamischen Vorwärtsslice. Das Slicing-Kriterium und die Trace müssen im ESDG markiert sein. Die in der Slice enthaltenen Knoten werden im ESDG durch Setzen eines Attributs gekennzeichnet.
- *2.4 computeStaticChop* zur Berechnung eines statischen Chops. Das Chopping-Kriterium muss im ESDG markiert sein. Die in dem Chop enthaltenen Knoten werden im ESDG durch Setzen eines Attributs gekennzeichnet.
- *2.5 computeDynamicChop* zur Berechnung eines dynamischen Chops. Das Chopping-Kriterium und die Trace müssen im ESDG markiert sein. Die in dem Chop enthaltenen Knoten werden im ESDG durch Setzen eines Attributs gekennzeichnet.
- *2.6 computeExecutableBackwardSlice* zur Berechnung einer ausführbaren Rückwärtsslice. Die Slice muss im ESDG markiert sein. Eine ausführbare Slice entsteht durch die Kennzeichnung zusätzlicher Knoten, um sie der Slice hinzuzufügen.
- *3 convertESDGToCode* zur Konvertierung der in dem ESDG markierten Slice in Programmcode.

Abbildung 5.9 zeigt den aus dem Beispielprogramm in Abbildung 3.1 schlussendlich berechneten ESDG. Das Schema des ESDG ist Abbildung 5.8 zu entnehmen.

Im ersten Teilabschnitt 5.6.1 werden die Knoten und Kanten des ESDG beschrieben, ausgenommen Aufrufkanten, Parameter-in-Kanten, Parameter-out-Kanten, Kontrollkanten, Datenfluss- und Deklarationskanten sowie transitive Kanten. Diese werden gesondert in den Abschnitten 5.6.2, 5.6.3, 5.6.4 und 5.6.5 behandelt.

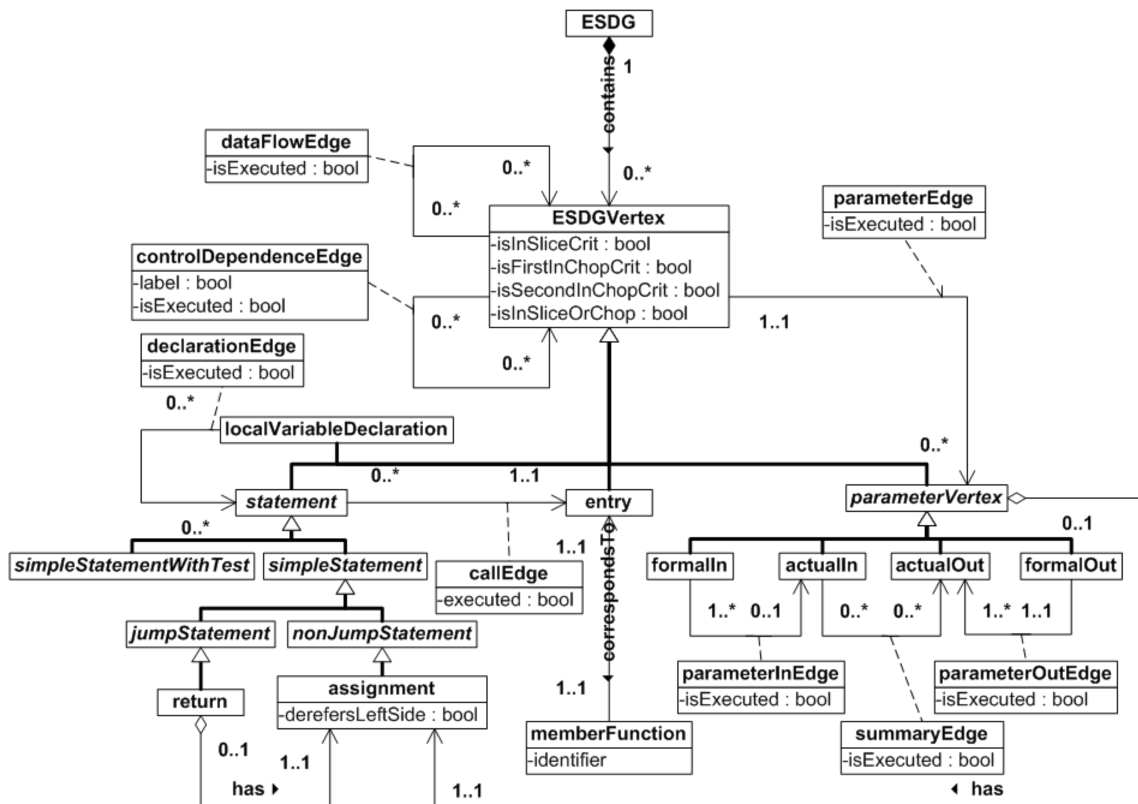


Abbildung 5.8.: Schema des ESDG

## 5.6.1. Grundelemente des erweiterten Systemabhängigkeitsgraphen

Der ESDG enthält sämtliche in den ACFGs enthaltenen Knoten abzüglich der *exit*-Knoten. Hinzu kommen die Parameterknoten (*parameterVertex*), bei denen sich zwischen *formalIn*- und *actualIn*- sowie *formalOut*- und *actualOut*-Knoten unterscheiden lässt. Sie stellen die von einer Methode benutzten (-In-Knoten) bzw. definierten Variablen (-Out-Knoten) dar.

Die *formalIn*- und die *formalOut*-Knoten sind mit dem *entry*-Knoten der entsprechenden Methode über Parameterkanten (*parameterEdge*), und mit einer eigenen Zuweisung über *has*-Kanten verbunden. Im Falle eines *formalIn*-Knotens weist diese Zuweisung einer von der Methode benutzten Variable $x$ die den aktuellen Wert repräsentierende, nicht im Ursprungsprogramm enthaltene Hilfsvariable $x_{in}$ zu. Die Zuweisung eines *formalOut*-Knoten weist eine von der Methode definierbare Variable $x$ einer Hilfsvariable $x_{out}$ zu.

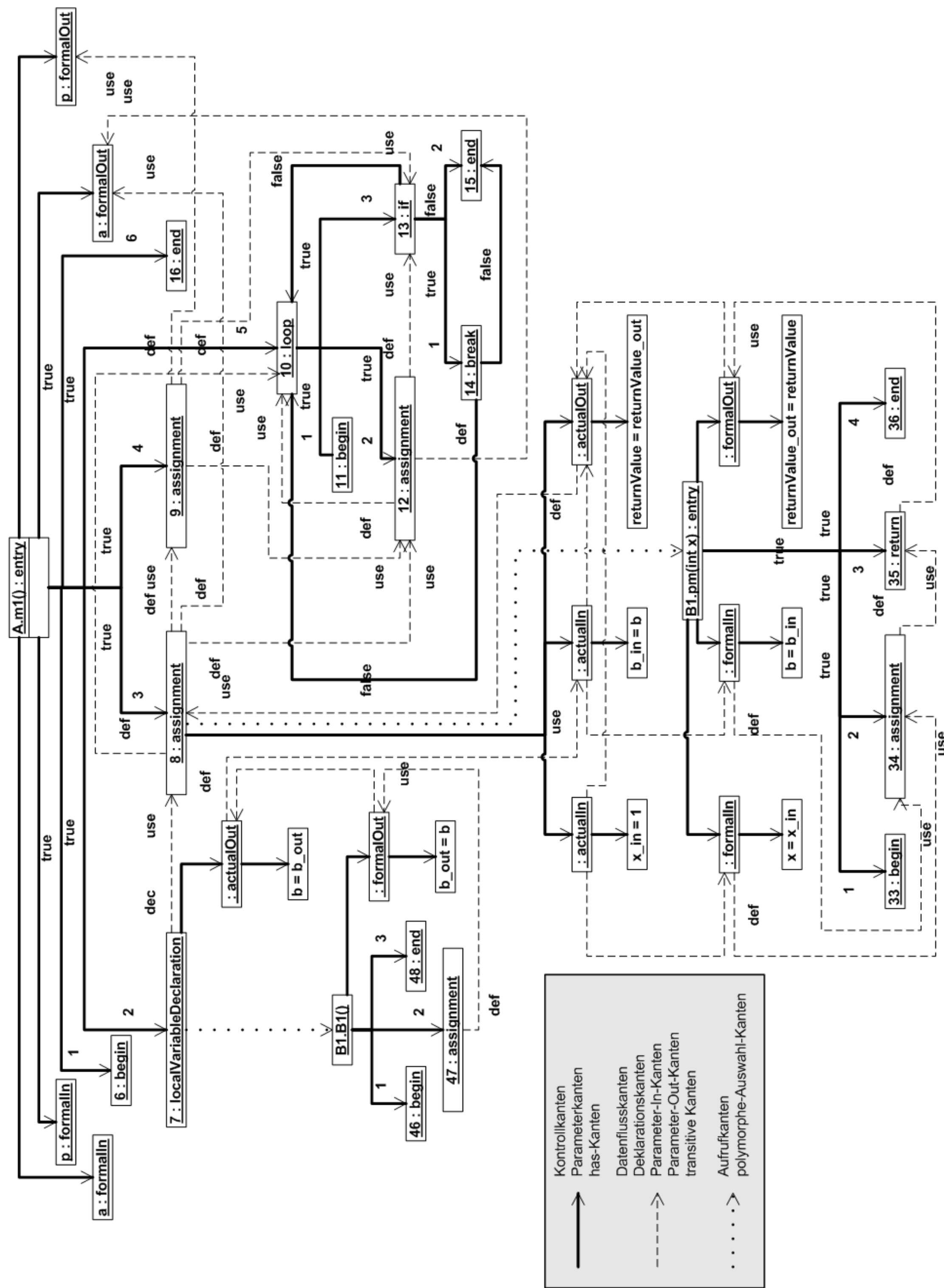


Abbildung 5.9.: Auszug aus dem erweiterten Systemabhängigkeitsgraphen des Beispielprogramms aus Abbildung 3.1. Dargestellt sind die Methoden *A.m1()*, *B1.B1()* und *B1.pm(int x)*)

Die *actualIn*- und *actualOut*-Knoten werden, ebenfalls über Parameterkanten, mit einem ESDG-Knoten verknüpft, der einer Aufrufanweisung entspricht oder einen Funktionsaufruf enthält. Analog zu *formalIn* und *formalOut* stellen sie die von einer aufgerufenen Methode benutzten bzw. definierten Variablen dar. *ActualIn*-Knoten verfügen über eine Zuweisung, die der Hilfsvariablen $x_{in}$ den aktuellen Wert (z.B. einen aktuellen Parameter) zuweist. Die Zuweisung der *ActualOut*-Knoten weist $x_{out}$ der von der Methode definierten Variablen zu.

Zur Berücksichtigung des Rückgabewertes einer Methode wird jeder *return*-Knoten über eine *has*-Kante an eine Zuweisung geknüpft, die den zurückzugebenden Ausdruck der im ECG eingeführten Hilfsvariable *returnValue* zuweist (siehe auch Abschnitt 5.5).

Im Folgenden wird die Knoten- und Kantenmenge des ESDG, ausgenommen oben angeführte Kanten, definiert:

**Definition (Grundgerüst des erweiterten Systemabhängigkeitsgraphen)**
Sei $G = (V, E)$ ein Graph, der einen abstrakten Syntaxgraphen und eine Menge erweiterter Kontrollflussgraphen enthält.

Ein um einen erweiterten Systemabhängigkeitsgraphen erweiterter Graph $G' = (V', E')$ geht aus $G$ hervor mit

$$\exists! esdg \in V'_{ESDG.ESDG} \Big( \forall m \in V_{ECG.memberFunction}$$

$$\Big( \exists! entry \in V_{ACFGSet.entry}$$

$$\Big( (m \leftarrow_{ACFGSet.describes} \bullet_{ACFGSet.ACFG} \rightarrow_{ACFGSet.contains} entry \quad (1)$$

$$\Rightarrow m \rightarrow'_{ESDG.correspondsTo} entry)$$

$$\wedge \forall v \in V_{ACFGSet.ACFGVertex} \backslash V_{ACFGSet.exit}, x \in V_{ASG.variable}$$

$$\Big( esdg \rightarrow'_{ESDG.contains} v \quad (2)$$

$$\wedge \Big( m \rightarrow_{ECG.uses} x \Rightarrow \exists! fi \in V'_{ESDG.formalIn}, assFi \in V'_{ESDG.assignment}, \quad (3)$$

$$x_{in} \in V'_{ESDG.variable}$$

$$\Big( esdg \rightarrow'_{ESDG.contains} fi \quad (4)$$

$$\wedge\ entry \rightarrow'_{ESDG.parameterEdge} fi \rightarrow'_{ESDG.has} assFi \quad (5)$$

$$\wedge\ x \leftarrow'_{ESDG.defines} assFi \rightarrow'_{ESDG.hasAsDefinition} x_{in} \quad (6)$$

$$\wedge \Big( (v \rightarrow_{ECG.canCall} m \vee v \rightarrow (\bullet_{ASG.compoundExpression} \rightarrow_{ASG.hasAsOperand})^* \quad (7)$$

$$\bullet_{ASG.functionCall} \rightarrow_{ECG.canCall} m) \Rightarrow$$

$$\exists! ai \in V'_{ESDG.actualIn}, assAi \in V'_{ESDG.assignment},$$

$$\Big( esdg \rightarrow'_{ESDG.contains} ai \quad (8)$$

$$\wedge\ v \rightarrow'_{ESDG.parameterEdge} ai \rightarrow'_{ESDG.has} assAi \quad (9)$$

$$\wedge\ assAi \rightarrow'_{ESDG.defines} x_{in} \tag{10}$$

$$\wedge\ \nexists ai_2 \in V'_{ESDG.actualIn}(v \rightarrow'_{ESDG.parameterEdge} ai_2 \rightarrow'_{ESDG.has} \bullet'_{ESDG.assignment} \rightarrow'_{ESDG.defines} x_{in}) \tag{11}$$

$$\wedge\ (x \leftarrow_{ASG.declares} \bullet_{ASG.memberVariableDeclaration} \Rightarrow assAi \rightarrow'_{ESDG.hasAsDefinition} x) \tag{12}$$

$$\wedge\ \exists e \in V_{ASG.expression}\Big(\forall i \in \mathbb{N}\Big(\Big(x \leftarrow_{ASG.declares} \bullet_{ASG.formalParameter} \overset{i}{\leftarrow}_{ASG.has} \bullet_{ASG.memberFunctionDefinition} \rightarrow_{ASG.defines} m \leftarrow_{ECG.canCall} v \overset{i}{\rightarrow}_{ASG.hasAsActualParameter} e \vee \exists fc \in V_{ASG.functionCall}(x \leftarrow_{ASG.declares} \bullet_{ASG.formalParameter} \overset{i}{\leftarrow}_{ASG.has} \bullet_{ASG.memberFunctionDefinition} \rightarrow_{ASG.defines} m \leftarrow_{ECG.canCall} fc (\leftarrow_{ASG.hasAsOperand} \bullet_{ASG.compoundExpression})^* \leftarrow v \overset{i}{\rightarrow}_{ASG.hasAsActualParameter} e)\Big) \Rightarrow assAi \rightarrow'_{ESDG.hasAsDefinition} e\Big)\Big)\Big)\Big)\Big)\Big) \tag{13}$$

$$\wedge\ \Big(m \rightarrow_{ECG.defines} x \Rightarrow \exists! fo \in V'_{ESDG.formalOut}, assFo \in V'_{ESDG.assignment}, x_{out} \in V'_{ESDG.variable} \tag{14}$$

$$\Big(esdg \rightarrow'_{ESDG.contains} fo \tag{15}$$

$$\wedge\ entry \rightarrow'_{ESDG.parameterEdge} fo \rightarrow'_{ESDG.has} assFo \tag{16}$$

$$\wedge\ x_{out} \leftarrow'_{ESDG.defines} assFo \rightarrow'_{ESDG.hasAsDefinition} x \tag{17}$$

$$\wedge\ \Big((v \rightarrow_{ECG.canCall} m \vee v \rightarrow (\bullet_{ASG.compoundExpression} \rightarrow_{ASG.hasAsOperand})^* \bullet_{ASG.functionCall} \rightarrow_{ECG.canCall} m) \Rightarrow \exists! ao \in V'_{ESDG.actualOut}, assAo \in V'_{ESDG.assignment}, \tag{18}$$

$$\Big(esdg \rightarrow'_{ESDG.contains} ao \tag{19}$$

$$\wedge\ v \rightarrow'_{ESDG.parameterEdge} ao \rightarrow'_{ESDG.has} assAo \tag{20}$$

$$\wedge\ x \leftarrow'_{ESDG.defines} assAo \rightarrow'_{ESDG.hasAsDefinition} x_{out} \tag{21}$$

$$\wedge\ \nexists ao_2 \in V'_{ESDG.actualOut}(v \rightarrow'_{ESDG.parameterEdge} ao_2 \rightarrow'_{ESDG.has} \bullet'_{ESDG.assignment} \rightarrow'_{ESDG.hasAsDefinition} x_{in})\Big)\Big)\Big)\Big)\Big)\Big)\Big)\Big) \tag{22}$$

$$\wedge\quad \forall rv \in V^{identifier=returnValue}_{ASG.variable}\Big(m \rightarrow_{ECG.defines} rv \Rightarrow \forall ret \in V_{ASG.return} \Big(\exists! e \in V_{ASG.expression} \Big((m \leftarrow_{ACFGSet.describes} \bullet_{ACFGSet.ACFG} \rightarrow_{ACFGSet.contains} ret \rightarrow_{ASG.returns} e \tag{23}$$

$$\Rightarrow ret \rightarrow_{ESDG.has} assRet)$$
$$\wedge\ rv \leftarrow'_{ESDG.defines} assRet \rightarrow'_{ESDG.hasAsDefinition} e\Big)\Big)\Big)\Big)$$

(1) Für alle Methoden (*memberFunction*) $m$ existiert im ESDG genau ein *entry*-Knoten $entry$, der über eine *correspondsTo*-Kante mit $m$ verbunden wird.

(2) Alle Knoten $v$ der ACFG-Menge vom Typ *ACFGVertex*, ausgenommen *exit*-Knoten, sind im ESDG enthalten.

(3) Für jede Variable (*variable*) $x$ gilt, dass, wenn sie von der Methode $m$ benutzt wird, d.h. im ECG existiert eine *uses*-Kante von $m$ zu $x$, genau ein *formalIn*-Knoten $fi$, eine Zuweisung (*assignment*) $assFi$ und eine Variable $x_{in}$ existieren, so dass

(4) $fi$ im ESDG enthalten,

(5) über eine Parameterkante (*parameterEdge*) mit $entry$ verknüpft und über eine *has*-Kante mit $assFi$ verbunden ist.

(6) $assFi$ definiert $x$ und enthält $x_{in}$ als Definition, d.h. dem von $m$ benutzten Attribut oder formalen Eingabeparameter $x$ wird die Hilfsvariable $x_{in}$ zugewiesen.

(7) Falls $m$ von $v$ aufgerufen werden kann, sei es mit $v$ als Aufrufanweisung (*callStatement*) oder durch einen in $v$ enthaltenen Funktionsaufruf (*functionCall*), dann existieren (für jede Variable $x$, die von $m$ benutzt wird, genau ein *actualIn*-Knoten $ai$ und eine Zuweisung $assAi$.

(8) Dabei ist $ai$ im ESDG enthalten,

(9) über eine Parameterkante (*parameterEdge*) mit $v$ verknüpft und über eine *has*-Kante mit $assAi$ verbunden.

(10) $assAi$ definiert $x_{in}$.

(11) Es gibt keinen anderen an $v$ geknüpften *actualIn*-Knoten, der $x_{in}$ definiert.

(12) Handelt es sich bei $x$ um ein Attribut, d.h. verläuft eine *declares*-Kante von einer Attributdeklaration (*memberVariableDeclaration*) zu $x$, so enthält $assAi$ $x$ als Definition. $x_{in}$ dient folglich als „Bindeglied“ zwischen *actualIn*- und *formalIn*-Knoten.

(13) Handelt es sich bei $x$ um einen formalen Eingabeparameter, d.h. verläuft eine *declares*-Kante von einer Parameterdeklaration (*formalParameter*) zu $x$, so enthält $assAi$ den entsprechenden aktuellen Parameter $e$ als Definition.

(14) Wird eine Variable (*variable*) $x$ von der Methode $m$ definiert, d.h. im ECG existiert eine *defines*-Kante von $m$ zu $x$, so existiert genau ein *formalOut*-Knoten $fo$, eine Zuweisung $assFo$ und eine Variable $x_{out}$, so dass

(15) $fo$ im ESDG enthalten,

(16) über eine Parameterkante (*parameterEdge*) mit $entry$ verknüpft und über eine *has*-Kante mit $assFo$ verbunden ist.

(17) $assFo$ definiert $x_{out}$ und enthält $x$ als Definition.

(18) Falls $m$ von $v$ aufgerufen werden kann, sei es mit $v$ als Aufrufanweisung oder durch einen in $v$ enthaltenen Funktionsaufruf, dann existieren (für jede Variable $x$, die von $m$ definiert wird, genau ein *actualOut*-Knoten $ao$ und eine Zuweisung $assAo$.

(19) Dabei ist $ao$ im ESDG enthalten,

(20) über eine Parameterkante (*parameterEdge*) mit $v$ verknüpft und über eine *has*-Kante mit $assAo$ verbunden.

(21) $assAo$ definiert $x$ und enthält $x_{out}$ als Definition. Analog zu $x_{in}$ dient $x_{out}$ als „Bindeglied“ zwischen *formalOut*- und *actualOut*-Knoten.

(22) Es gibt keinen anderen an $v$ geknüpften *actualOut*-Knoten, der $x_{out}$ als Definition enthält.

(23) Wird ein Rückgabewert $rv$, gekennzeichnet durch den Bezeichner *returnValue* von der Methode $m$ definiert, d.h. im ECG existiert eine *declares*-Kante von $m$ zu $rv$, so sind alle in der Methode enthaltenen *return*-Knoten $ret$, die einen Wert zurückgeben, über eine *has*-Kante mit einer Zuweisung $assRet$ verbunden. $assRet$ weist $rv$ den Rückgabewert $e$ zu. □

### 5.6.2. Interprozedurale Kanten

Die interprozeduralen Kanten verbinden die die einzelnen Methoden darstellenden Teilgraphen des ESDG miteinander. Dabei wird zwischen Aufrufkanten (*callEdge*) sowie Parameter-in- (*parameterInEdge*) und Parameter-out-Kanten (*parameterOutEdge*) unterschieden.

Aufrufkanten verlaufen von einem eine Methode aufrufenden ESDG-Knoten zu dem *entry*-Knoten der jeweiligen Methode. Unter Umständen gehen von einem Knoten mehrere Aufrufkanten aus, falls bei der Berechnung des ECG der genaue Typ des Objekts, dessen Methode aufgerufen wird, nicht bestimmt werden konnte.

Eine Parameter-in-Kante verbindet einen *actualIn*- mit einem *formalIn*-Knoten. Die Bedingungen dafür sind, dass die mit den Knoten in Beziehung stehenden Zuweisungen dieselbe Variable definieren bzw. als Definition verwenden und dass eine Aufrufkante zwischen dem entsprechenden die Methode aufrufenden Knoten und dem *entry*-Knoten existiert. Dasselbe

gilt für die Parameter-out-Kanten, die die *formalOut*- mit den *actualOut*-Knoten verbinden. Analog zu den Aufrufkanten kann auch hier aufgrund der Polymorphie ein *actualIn*- oder *actualOut*-Knoten mit mehreren *formalIn*- bzw. *formalOut*-Knoten verbunden sein.

Abbildung 5.10 zeigt exemplarisch den Aufruf der Methode *B2.pm(int x)* in Methode *m2()* des Beispielprogramms aus Beispiel 3.1. Es wird deutlich, wie sich der Übergabemechanismus mit Parameterknoten darstellt – für von einer Methode benutzte Attribute (`b`) und Eingangsparameter (`x`) sowie definierte Attribute (`c`) und den Rückgabewert. Die mit den Parameterknoten über *has*-Kanten verbundenen Zuweisungen sind vereinfachend in einem einzigen Knoten dargestellt. Es wird angenommen, dass der Typ des Objekts `o` nicht bekannt ist und somit ein möglicher Aufruf der Methode *B1.pm(int x)* berücksichtigt.

Abbildung 5.11 zeigt den Aufruf der Methode *B2.m4(int *q)* in Methode *m2()*. Es wird eine Zeigervariable übergeben und die von dieser referenzierte Variable redefiniert. Die Anzahl der Parameterknoten hängt hier von der Genauigkeit des Points-to-Graphen ab. Würde dieser z.B. anzeigen, dass *y* neben *a* noch auf eine andere Variable zeigen kann, so würden auch für diese je ein *actualOut*- und ein *formalOut*-Knoten vorhanden sein.

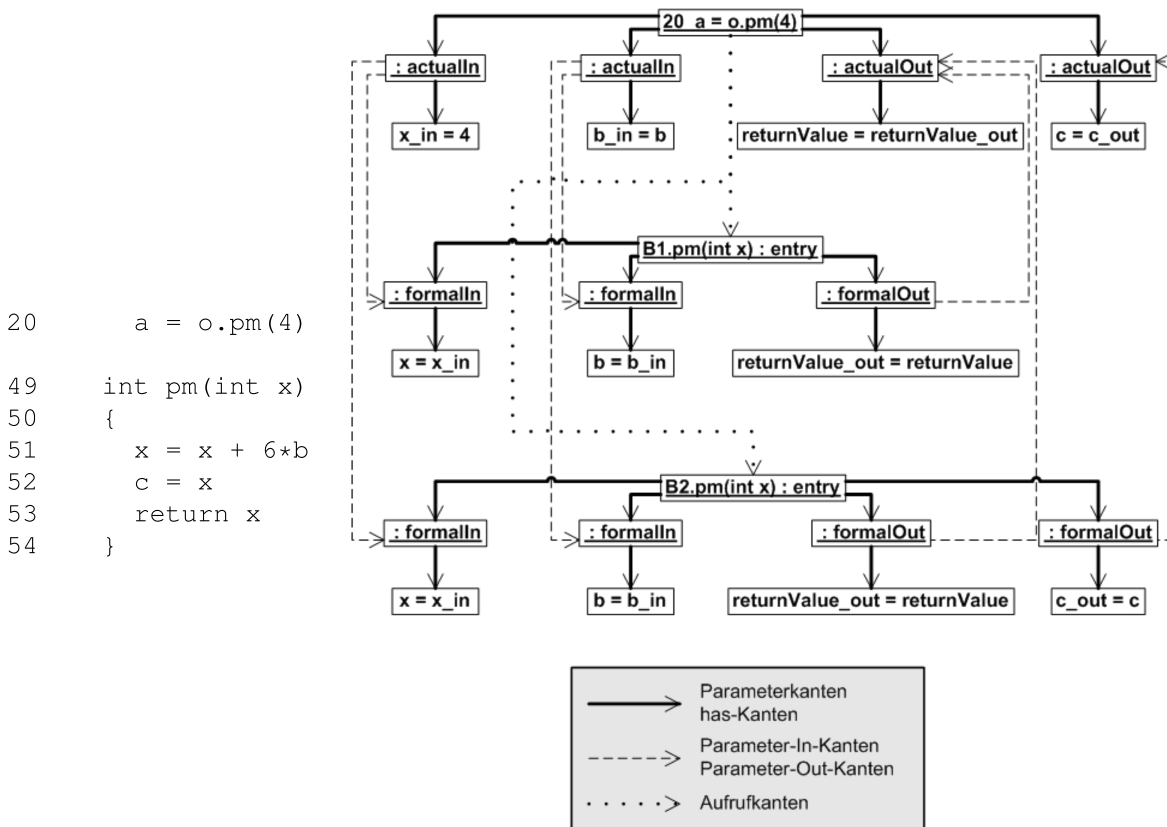


Abbildung 5.10.: Aufruf der Methode *pm(int x)* in Methode *m2()* des Beispielprogramms aus Abbildung 3.1

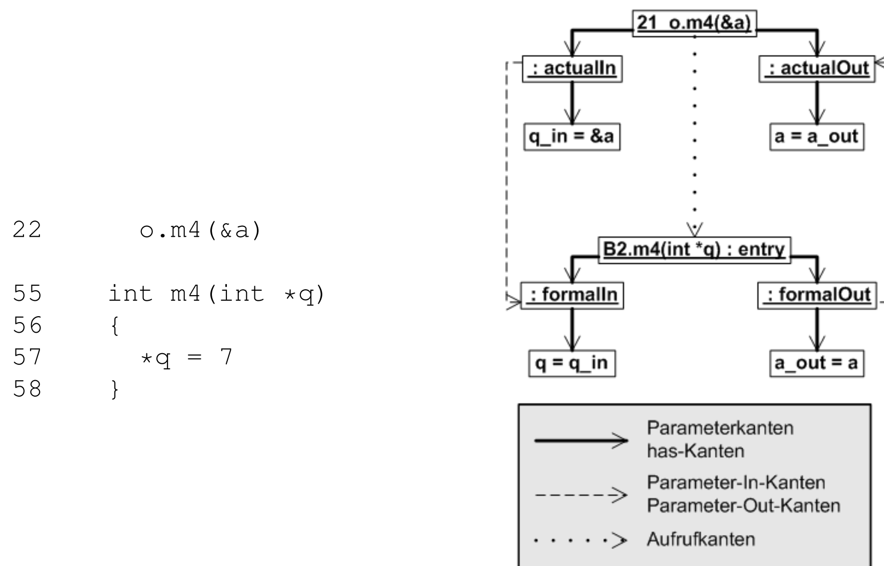


Abbildung 5.11.: Aufruf der Methode *m4(int *q)* in Methode *m2()* des Beispielprogramms aus Abbildung 3.1

Nachfolgend werden die interprozeduralen Kanten genau definiert:

**Definition (Aufrufkanten)**
Sei $G = (V, E)$ ein Graph, der einen abstrakten Syntaxgraphen, einen erweiterten Aufrufgraphen sowie einen erweiterten Systemabhängigkeitsgraphen, soweit in Abschnitt 5.6.1 definiert, enthält.

Ein um Aufrufkanten erweiterter Graph $G' = (V, E')$ geht aus $G$ hervor mit

$s \rightarrow'_{callEdge} t \Leftrightarrow$

> Ein ESDG-Knoten $s$ entspricht einer Aufrufanweisung oder enthält einen Funktionsaufruf. Von $s$ aus wird diejenige Methode aufgerufen, die über eine *correspondsTo*-Kante mit dem *entry*-Knoten $t$ verknüpft ist.
>
> $s \in V_{ESDG.ESDGVertex}, t \in V_{ESDG.entry}$
> mit $s \rightarrow_{ECG.canCall} \bullet_{ESDG.memberFunction} \rightarrow_{ESDG.correspondsTo} t$
> $\vee\ s \rightarrow (\bullet_{ASG.compoundExpression} \rightarrow_{ASG.hasAsOperand})^{*} \bullet_{ASG.functionCall}$
> $\rightarrow_{ECG.canCall} \bullet_{memberFunction} \rightarrow_{ESDG.correspondsTo} t$ □

**Definition (Parameter-in-Kanten)**
Sei $G = (V, E)$ ein Graph, der einen abstrakten Syntaxgraphen, einen erweiterten Aufrufgraphen sowie einen erweiterten Systemabhängigkeitsgraphen, soweit in Abschnitt 5.6.1 zuzüglich der Aufrufkanten definiert, enthält.

Ein um Parameter-in-Kanten erweiterter Graph $G' = (V, E')$ geht aus $G$ hervor mit

$s \rightarrow'_{parameterInEdge} t \Leftrightarrow$

> Ein *actualIn*-Knoten $s$ ist über eine *has*-Kante mit einer Zuweisung verbunden, die eine Variable $x_{in}$ definiert. Dieselbe Variable wird von einer mit einem *formalIn*-Knoten $t$ verbundenen Zuweisung als Definition verwendet. Ferner sind $s$ und $t$ über Parameterkanten an einen ESDG- bzw. an einen *entry*-Knoten geknüpft, zwischen denen eine Aufrufkante besteht.
>
> $s \in V_{ESDG.actualIn}, t \in V_{ESDG.formalIn}$
> mit $s \rightarrow_{has} \bullet_{ESDG.assignment} \rightarrow_{ESDG.defines} x_{in}$
> $\wedge\ t \rightarrow_{has} \bullet_{ESDG.assignment} \rightarrow_{ESDG.hasAsDefinition} x_{in}$
> $\wedge\ s \leftarrow_{ESDG.parameterEdge} \bullet_{ESDG.ESDGVertex} \rightarrow_{ESDG.callEdge} \bullet_{ESDG.entry}$
> $\rightarrow_{ESDG.parameterEdge} t$ □

**Definition (Parameter-out-Kanten)**
Sei $G = (V, E)$ ein Graph, der einen abstrakten Syntaxgraphen, einen erweiterten Aufrufgraphen sowie einen erweiterten Systemabhängigkeitsgraphen, soweit in Abschnitt 5.6.1 zuzüglich der Aufrufkanten definiert, enthält.

Ein um Parameter-out-Kanten erweiterter Graph $G' = (V, E')$ geht aus $G$ hervor mit

$s \rightarrow'_{parameterOutEdge} t \Leftrightarrow$

> Ein *formalOut*-Knoten $s$ ist über eine *has*-Kante mit einer Zuweisung verbunden, die eine Variable $x_{out}$ definiert. Dieselbe Variable wird von einer mit dem *actualOut*-Knoten $t$ verbundenen Zuweisung als Definition verwendet. Ferner sind $s$ und $t$ über Parameterkanten an einen ESDG- bzw. an einen *entry*-Knoten geknüpft, zwischen denen eine Aufrufkante besteht.
>
> $s \in V_{ESDG.formalOut}, t \in V_{ESDG.actualOut}$
> mit $s \rightarrow_{has} \bullet_{ESDG.assignment} \rightarrow_{ESDG.defines} x_{out}$
> $\wedge\ t \rightarrow_{has} \bullet_{ESDG.assignment} \rightarrow_{ESDG.hasAsDefinition} x_{out}$
> $\wedge\ s \leftarrow_{ESDG.parameterEdge} \bullet_{ESDG.ESDGVertex} \rightarrow_{ESDG.callEdge} \bullet_{ESDG.entry}$
> $\rightarrow_{ESDG.parameterEdge} t$ □

### 5.6.3. Kontrollkanten

Die Kontrollkanten (*controlDependenceEdge*) spiegeln in einer Methode ohne Sprünge die Schachtelung der Anweisungen wider, wobei Blöcke nicht berücksichtigt werden, sondern

Blockanfang und Blockende als einfache Anweisungen anzusehen sind. In Methoden, die Sprünge enthalten, besitzen die Sprunganweisungen ebenfalls ausgehende Kontrollkanten.

Abbildung 5.12 sind beispielhaft die zwischen den Knoten der Methode *m1* des Beispielprogramms aus Abbildung 3.1 verlaufenden Kontrollkanten zu entnehmen.

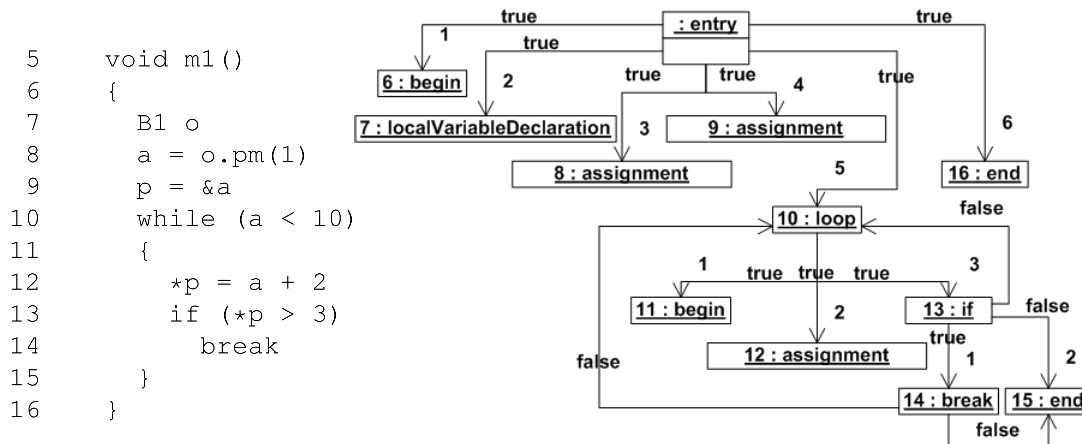


Abbildung 5.12.: Kontrollkanten der Methode *m1* des Beispielprogramms aus Abbildung 3.1

Kontrollkanten verlaufen einerseits von dem *entry*-Knoten zu den nicht in einer zusammengesetzten Anweisung enthaltenen Methodenelementen, d.h. lokalen Variablendeklarationen (*localVariableDeclaration*) und Anweisungen (*statement*), andererseits von den einfachen Anweisungen mit Test (*simpleStatementWithTest*) zu den Elementen im zugehörigen Rumpf. Dabei sind die Kanten immer mit *true* attributiert, es sei denn, sie verlaufen von einem *if*-Knoten zu den im *else*-Fall abzuarbeitenden Methodenelementen. Von den Sprunganweisungen (*jumpStatement*) gehen mit *false* attributierte Kontrollkanten zu denjenigen Elementen aus, zu deren Abarbeitung es nur bei Nichtausführung des Sprunges kommen würde. Dabei darf sich zwischen den Sprunganweisung und den besagten Elementen keine weitere Anweisung, die die Abarbeitung der Elemente verhindern könnte, befinden.

Die folgende Definition enthält zunächst das Konzept der Postdominanz, welches zur daran anschließenden, eigentlichen Definition der Kontrollkanten notwendig ist.

**Definition (Kontrollkante)**
Sei $G = (V, E)$ ein Graph der eine Menge erweiterter Kontrollflussgraphen sowie einen erweiterten Systemabhängigkeitsgraphen, soweit in den Abschnitten 5.6.1 und 5.6.2 definiert, enthält.

Die Menge $PD_s$ der von einem Knoten $s \in V_{ACFGSet.ACFGVertex}$ postdominierten Knoten ist gegeben durch diejenigen $t \in V_{ACFGSet.ACFGVertex}$ ungleich $s$, von denen aus jeder Pfad in der ACFG-Menge zu einem *exit*-Knoten $s$ enthält (siehe [Fer+87, S. 321]):

$$PD_s = \Big\{ t \in V_{ACFGSet.ACFGVertex} \Big| s \neq t \wedge \forall n_{1,...,k-1} \in V_{ACFGSet.ACFGVertex} \\ (t = n_0 \rightarrow_{ACFGSet.controlFlowEdge} n_1 \rightarrow_{ACFGSet.controlFlowEdge} \cdots \\ \rightarrow_{ACFGSet.controlFlowEdge} n_k = \bullet_{ACFGSet.exit} \Rightarrow s \in \{n_1, ..., n_k\}) \Big\}$$

Ein um Kontrollkanten erweiterter Graph $G' = (V, E')$ geht aus $G$ hervor mit

$s \rightarrow'^{label=true}_{ESDG.controlDependenceEdge} t \Leftrightarrow$

Von einem Knoten $s$ existiert in der ACFG-Menge ein Pfad zu einem Knoten $t$, so dass die erste, von $s$ ausgehende Kontrollflusskante mit *always* oder *true* attributiert ist und außerdem jeder in dem Pfad enthaltene Knoten von $t$ postdominiert wird, ausgenommen $s$ selbst:

$$s \in (V_{ESDG.entry} \cup V_{ESDG.simpleStatementWithTest} \cup V_{ESDG.jumpStatement}), \\ t \in V_{ESDG.ESDGVertex} \\ \text{mit } \exists n_{1,...,k-1} \in V_{ESDG.ESDGVertex} \\ (s = n_0 \rightarrow^{label=always \vee label=true}_{ACFGSet.controlFlowEdge} n_1 \rightarrow_{ACFGSet.controlFlowEdge} \cdots \\ \rightarrow_{ACFGSet.controlFlowEdge} n_k = t \wedge n_{i,1<i<k-1} \in PD_t) \wedge s \notin PD_t$$

$s \rightarrow'^{label=false}_{ESDG.controlDependenceEdge} t \Leftrightarrow$

Von einem Knoten $s$ existiert in der ACFG-Menge ein Pfad zu einem Knoten $t$, so dass die erste, von $s$ ausgehende Kontrollflusskante mit *false* attributiert ist und außerdem jeder in dem Pfad enthaltene Knoten von $t$ postdominiert wird, ausgenommen $s$ selbst:

$$s \in (V_{ESDG.entry} \cup V_{ESDG.simpleStatementWithTest} \cup V_{ESDG.jumpStatement}), \\ t \in V_{ESDG.ESGVertex} \\ \text{mit } \exists n_{1,...,k-1} \in V_{ESDG.ESDGVertex} \\ (s = n_0 \rightarrow^{label=false \vee label=never}_{ACFGSet.controlFlowEdge} n_1 \rightarrow_{ACFGSet.controlFlowEdge} \cdots \\ \rightarrow_{ACFGSet.controlFlowEdge} n_k = t \wedge n_{i,1<i<k-1} \in PD_t) \wedge s \notin PD_t \qquad \square$$

Anhand der Abbildung 5.13, die den ACFG der Methode *m1* den daraus berechneten Kontrollkanten gegenüberstellt, wird die obige Definition im Folgenden erläutert.

Aus der Definition geht hervor, dass ACFG-Knoten mit nur genau einer ausgehenden Kontrollflusskante keine ausgehenden Kontrollkanten aufweisen können, weil sie von jedem unmittelbaren Nachfolger selbst postdominiert werden. Dies trifft auf die Knoten 6-9, 11, 12,

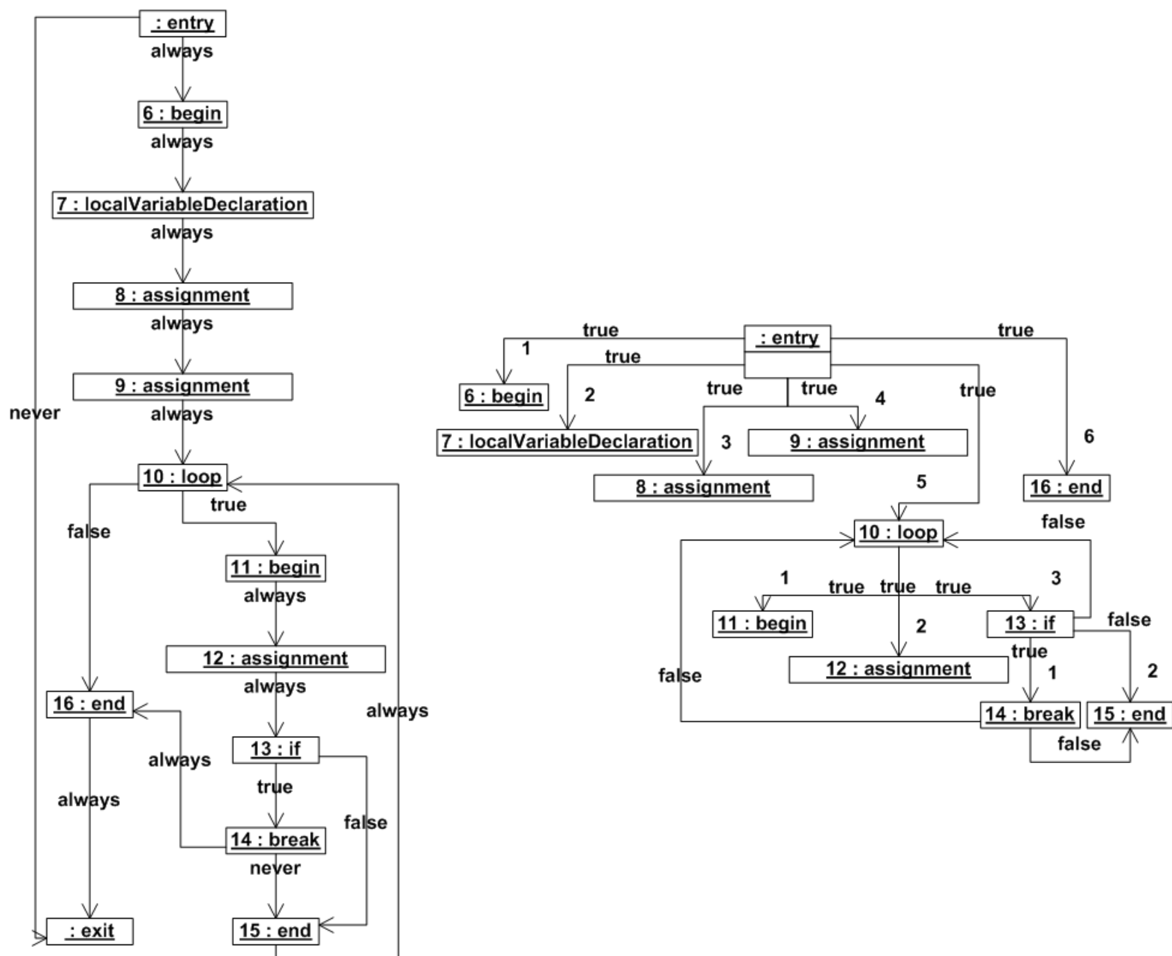


Abbildung 5.13.: ACFG der Methode *m1* des Beispielprogramms aus Abbildung 3.1 und die daraus berechneten Kontrollkanten

15 und 16 zu. Für die Pfade von dem *entry*-Knoten zu jedem Knoten $t \in \{6, 7, 8, 9, 10, 16\}$ wird jeder sich auf dem Pfad befindliche Knoten außer *entry* von $t$ postdominiert, so dass mit *true* attributierte Kontrollkanten von *entry* zu jedem Knoten aus der obengenannten Menge verlaufen. Analog dazu verhält es sich mit der *loop*-Anweisung 10 und der Knotenmenge $\{11, 12, 13\}$.

Auf den Pfaden von dem der *if*-Anweisung entsprechenden Knoten 13 zu den Knoten 14 und 15 liegen keine anderen Knoten und Knoten 13 wird nicht von 14 oder 15 postdominiert. Da Knoten 10 Knoten 15, aber nicht 13 postdominiert und es einen Pfad $13 \rightarrow_{ACFGSet.controlFlowEdge} 15 \rightarrow_{ACFGSet.controlFlowEdge} 10$ gibt, existiert eine Kontrollkante von 13 nach 10.

Die aus dem Knoten der *break*-Anweisung 14 ausgehenden Kontrollkanten begründen sich zum einen auf der Kontrollflusskante von 14 zu 15, wobei 14 nicht von 15 postdominiert

| Nr. | Anweisung | Menge der def. Variablen | Menge der ben. Variablen |
|---|---|---|---|
| 8 | `a = o.pm(1)` | {a} | {o, x, b}[8] |
| 9 | `p = &a` | {p} | {a} |
| 12 | `*p = a + 2` | {*p [a]}[9] | {p, a} |
| 13 | `if (*p > 3)` | {} | {p, *p [a]} |
| 34 | `x = x + b` | {x} | {x, b} |

Abbildung 5.14.: Übersicht über verschiedene Anweisungen des Beispielprogramms und den von diesen definierten oder benutzten Variablen

wird. Zum anderen wird Knoten 15, nicht jedoch 14 von 10 postdominiert und es existiert der Pfad $14 \rightarrow_{ACFGSet.controlFlowEdge} 15 \rightarrow_{ACFGSet.controlFlowEdge} 10$.

### 5.6.4. Datenfluss- und Deklarationskanten

Die Datenfluss- (*dataFlowEdge*) und Deklarationskanten (*declarationEdge*) bilden *def-use*- bzw. *dec-def/use*-Beziehungen zwischen Anweisungen einer Methode ab. Letztere bestehen zwischen der Deklaration und der ersten Definition oder Benutzung einer Variable[7]. Eine *def-use*-Beziehung besteht zwischen zwei Knoten $s$ und $t$, falls von $s$ eine Variable definiert wird, die anschließend von $t$ genutzt werden kann, ohne dass es zwischen den Ausführungen von $s$ und $t$ zwingend zu einer Redefinition der Variable kommt. Handelt es sich bei $s$ oder $t$ um aufrufende Anweisungen, knüpfen Datenflusskanten an die jeweiligen Parameterknoten an.

Zu beachten ist, dass es bei Definition oder Benutzung einer Variable, genau genommen, um die von den Variablen referenzierten Speicherstellen geht: So wird eine in `p = &x` definierte Zeigervariable `p` in einer nachfolgenden Zuweisung `*p = 2` nicht neu definiert. Stattdessen kommt es zu einer Redefinition von *x*. Die Verwendung eines Dereferenz- oder Adressoperators im Zusammenhang mit einer Variable führt zu einer Nutzung derselben, auch auf der linken Seite einer Zuweisung, wie in `*p = 2`. Abbildung 5.14 zeigt eine Übersicht über verschiedene Anweisungen des Beispielprogramms in Abbildung 3.1 und den in diesen definierten oder benutzten Variablen.

Abbildung 5.15 zeigt beispielhaft die zwischen den Knoten der Methode *m1* des Beispielprogramms aus Abbildung 3.1 verlaufenden Datenfluss- und Deklarationskanten. Es ist zu erkennen, dass die *formalIn*-Knoten keine ausgehenden Kanten besitzen, weil die Attribute vor der ersten Benutzung redefiniert werden.

[7] Deklarierte Variablen können auch ohne vorherige (explizite) Definition benutzt werden. Dies ist etwa bei Objekten der Fall, deren Konstruktor bei der Deklaration implizit aufgerufen wird.

[8] x und b werden von der Methode pm(int x) benutzt.

[9] Bezeichner in eckigen Klammern stehen für die von einem Zeiger referenzierte Variable.

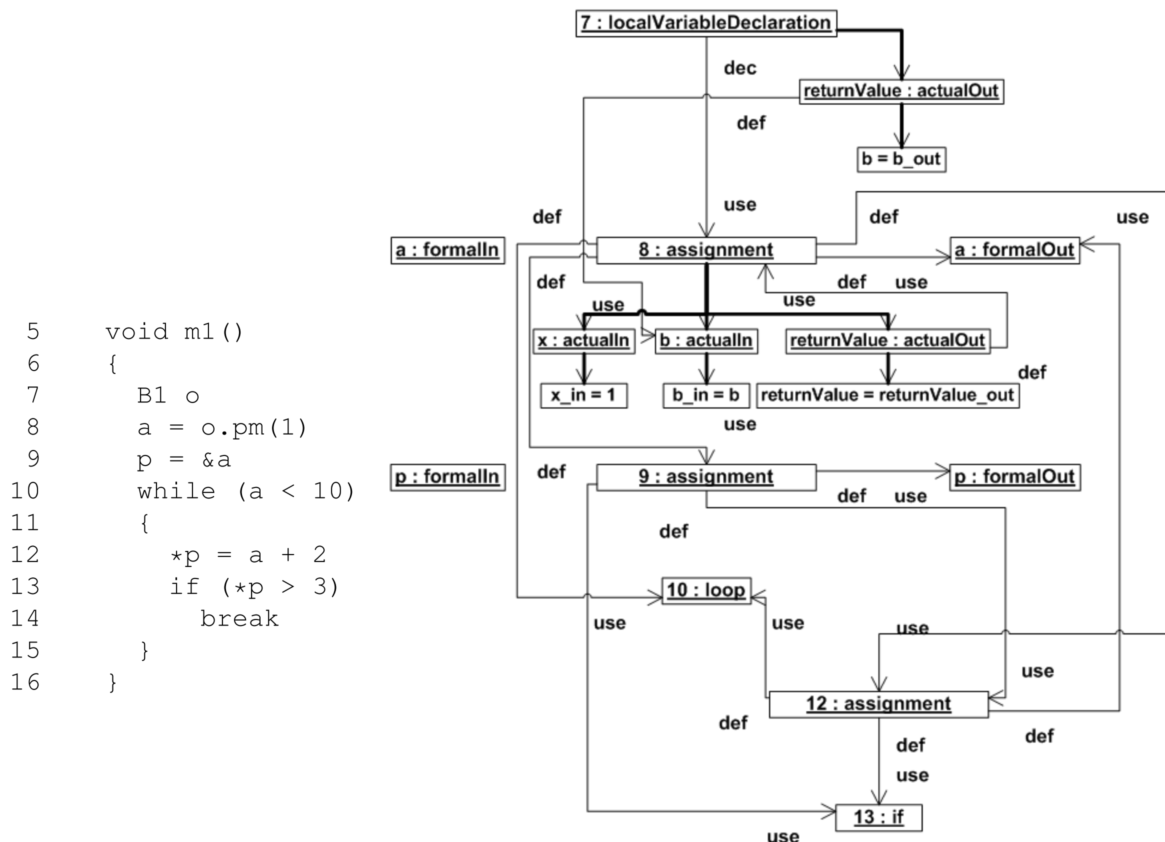


Abbildung 5.15.: Datenfluss- und Deklarationskanten der Methode *m1* des Beispielprogramms aus Abbildung 3.1

Die folgende Definition definiert zunächst die Mengen $DEC$, $DEF$ und $USE$, bevor sie die Datenfluss- und Deklarationskanten behandelt. Nicht zu deren Bestimmung herangezogen werden die mit *never* attributierten Kontrollflusskanten, da diese ausschließlich der Berechnung der Kontrollkanten dienen und nicht eine mögliche Abarbeitungsreihenfolge darstellen.

**Definition (Datenfluss- und Deklarationskanten)**
Sei $G = (V, E)$ ein Graph, der einen abstrakten Syntaxgraphen, eine Menge erweiterter Kontrollflussgraphen, einen erweiterten Aufrufgraphen, eine Menge der Points-to-Graphen sowie einen erweiterten Systemabhängigkeitsgraphen, soweit in den Abschnitten 5.6.1 bis 5.6.3 definiert, enthält. Sei weiterhin

$$\rightarrow_{cf} \quad = \quad \rightarrow^{label=always}_{ACFGSet.controlFlowEdge} \cup \rightarrow^{label=true}_{ACFGSet.controlFlowEdge} \cup \rightarrow^{label=false}_{ACFGSet.controlFlowEdge}$$

die Menge aller Kontrollflusskanten, die nicht mit *never* attributiert sind.

Die Menge $DEC_s$ der von einer lokalen Variablendeklaration

$$s \in V_{ASG.localVariableDeclaration}$$

deklarierten Variablen $x \in V_{variable}$ ist gegeben durch:

$$DEC_s = \left\{ x \in V_{ASG.variable} \middle| s \rightarrow_{ASG.declares} x \right\}$$

Die Menge $DEF_s$ der von einem Knoten

$$s \in (V_{ASG.assignment} \cup V_{ESDG.parameterVertex} \cup V_{ESDG.return})$$

definierten Variablen $x \in V_{variable}$ ist gegeben durch die Variable auf der linken Seite der Zuweisung bzw. der mit dem Parameterknoten oder *return*-Anweisung verbundenen Zuweisung:

$$\begin{aligned} DEF_s = \Big\{ x \in V_{ASG.variable} \Big| & \\ & s^{derefersLeftSide=false} \rightarrow_{ASG.defines} x && (1) \\ & \vee\ s^{derefersLeftSide=true} \rightarrow_{ASG.defines} \bullet_{ASG.variable} \rightarrow_{PGSet.pointsTo} x && (2) \\ & \vee\ s \rightarrow_{ESDG.has} \bullet^{derefersLeftSide=false}_{ASG.assignment} \rightarrow_{ASG.defines} x \Big\} && (3) \end{aligned}$$

(1) Variable (keine Zeigervariable) auf der linken Seite einer Zuweisung $s$

(2) von einer Zeigervariable auf der linken Seite einer Zuweisung $s$ referenzierte Variable

(3) Variable auf der linken Seite einer mit einer *return*-Anweisung oder einem Parameterknoten $s$ verbundenen Zuweisung

Die Menge $USE_s$ der von einem Knoten

$$s \in V_{ASG.assignment} \cup V_{ESDG.parameterVertex} \cup V_{ASG.callStatement} \cup V_{ASG.return}$$

benutzten Variablen $x \in V_{variable}$ ist gegeben durch die Variablen auf der rechten Seite einer Zuweisung in $s$ und durch die von in $s$ aufgerufenen Methoden benutzten Variablen:

$$\begin{aligned} USE_s = \Big\{ x \in V_{ASG.variable} \Big| & \\ & s^{derefersLeftSide=true} \rightarrow_{defines} x && (1) \\ & \vee\ s \rightarrow_{ASG.hasAsDefinition} \left(\bullet_{ASG.compoundExpression} \rightarrow_{ASG.hasAsOperand}\right)^{*} x && (2) \\ & \vee\ \exists e \in V_{ASG.compoundExpression} && (3) \\ & \quad (s \rightarrow_{ASG.hasAsDefinition} \left(\bullet_{ASG.compoundExpression} \rightarrow_{ASG.hasAsOperand}\right)^{*} e \end{aligned}$$

$$
\begin{aligned}
&\wedge \bullet^{identifier=*}_{ASG.operand} \overset{1}{\longleftarrow}_{ASG.hasAsOperator}\ e \overset{1}{\longrightarrow}_{ASG.hasAsOperand} \\
&\quad \bullet_{ASG.variable} \longrightarrow_{PGSet.pointsTo}\ x) \\
&\vee\ s \longrightarrow_{ASG.hasAsSelection}\ x && (4)\\
&\vee\ s \longrightarrow (\bullet_{ASG.compoundExpression} \longrightarrow_{ASG.hasAsOperand})^{*} \bullet_{ASG.functionCall} && (5)\\
&\quad \longrightarrow_{ASG.hasAsSelection}\ x \\
&\vee\ s \longrightarrow_{ESDG.has} \bullet_{ASG.assignment} \longrightarrow_{ASG.hasAsDefinition} && (6)\\
&\quad (\bullet_{ASG.compoundExpression} \longrightarrow_{ASG.hasAsOperand})^{*} x \\
&\vee\ \exists e \in V_{ASG.compoundExpression} && (7)\\
&\quad \Big(s \longrightarrow_{ESDG.has} \bullet_{ASG.assignment} \longrightarrow_{ASG.hasAsDefinition} \\
&\quad (\bullet_{ASG.compoundExpression} \longrightarrow_{ASG.hasAsOperand})^{*}\ e \\
&\wedge \bullet^{identifier=*}_{ASG.operand} \overset{1}{\longleftarrow}_{ASG.hasAsOperator}\ e \overset{2}{\longrightarrow}_{ASG.hasAsOperand} \\
&\quad \bullet_{ASG.variable} \longrightarrow_{PGSet.pointsTo}\ x\Big)\Big\}
\end{aligned}
$$

(1) Zeigervariable auf der linken Seite einer Zuweisung $s$

(2) Variablen (einschließlich Zeigervariablen) auf der rechten Seite einer Zuweisung $s$

(3) von einer Zeigervariable auf der rechten Seite einer Zuweisung $s$ referenzierte Variablen

(4) Objektvariable, an die die Aufrufanweisung $s$ eine Botschaft sendet

(5) Objektvariable, an die ein in $s$ enthaltener Funktionsaufruf eine Botschaft sendet

(6) Variable auf der rechten Seite einer mit einer *return*-Anweisung oder einem Parameterknoten $s$ verbundenen Zuweisung

(7) von einer Zeigervariable auf der rechten Seite einer mit einer *return*-Anweisung $s$ verbundenen Zuweisung referenzierte Variablen

Ein um Datenfluss- und Deklarationskanten erweiterter Graph $G' = (V, E')$ geht aus $G$ hervor mit

$s \longrightarrow'_{ESDG.dataFlowEdge}\ t \Leftrightarrow$ eine der folgenden Bedingungen gilt:

(i) Von einer $x$ definierenden Anweisung $s$ existiert in der ACFG-Menge ein Pfad zu einer $x$ benutzenden Anweisung $t$, auf dem $x$ nicht redefiniert wird:

$s \in V_{ESDG.statement}, t \in V_{ESDG.statement}$
mit $\exists x \in V_{ASG.variable}, n_{1,\ldots,k-1} \in V_{ACFGSet.ACFGVertex}$

$$(x \in DEF_s \land x \in USE_t \\ \land s = n_0 \rightarrow_{cf} n_1 \rightarrow_{cf} \ldots \rightarrow_{cf} n_k = t \\ \land x \notin (DEF_{n_1} \cup DEF_{n_2} \cup \ldots \cup DEF_{n_{k-1}}))$$

(ii) Von einem über eine Parameterkante mit dem $x$ benutzenden *formalOut*-Knoten $t$ verbundenen *entry*-Knoten existiert in der ACFG-Menge ein Pfad über eine $x$ definierende Anweisung $s$ zu einem *exit*-Knoten, so dass $x$ auf dem Teilpfad von $s$ zu dem *exit*-Knoten nicht redefiniert wird:

$$s \in V_{ESDG.statement}, t \in V_{ESDG.formalOut} \\ \text{mit } \exists x \in V_{ASG.variable}, n_{1,\ldots,k-1} \in V_{ACFGSet.ACFGVertex} \\ (x \in DEF_s \land x \in USE_t \\ \land t \leftarrow_{ESDG.parameterEdge} \bullet_{ACFGSet.entry} \rightarrow_{cf} (\bullet_{ACFGSet.ACFGVertex} \\ \rightarrow_{cf})^* s = n_0 \rightarrow_{cf} n_1 \rightarrow_{cf} \ldots \rightarrow_{cf} n_k = \bullet_{ACFGSet.exit} \\ \land x \notin (DEF_{n_1} \cup DEF_{n_2} \cup \ldots \cup DEF_{n_{k-1}}))$$

(iii) Von einer $x$ definierenden Anweisung $s$ existiert in der ACFG-Menge ein Pfad zu einer über Parameterkanten mit dem $x$ benutzenden *actualIn*-Knoten $t$ verbundenen ACFG-Knoten *ACFGVertex*, auf dem $x$ nicht redefiniert wird:

$$s \in V_{ESDG.statement}, t \in V_{ESDG.actualIn} \\ \text{mit } \exists x \in V_{ASG.variable}, n_{1,\ldots,k-1} \in V_{ACFGSet.ACFGVertex} \\ (x \in DEF_s \land x \in USE_t \\ \land s = n_0 \rightarrow_{cf} n_1 \rightarrow_{cf} \ldots \rightarrow_{cf} n_k = \bullet_{ACFGSet.ACFGVertex} \rightarrow_{ESDG.parameterEdge} t \\ \land x \notin (DEF_{n_1} \cup DEF_{n_2} \cup \ldots \cup DEF_{n_{k-1}}))$$

(iv) Von einem über eine Parameterkante mit dem $x$ definierenden *formalIn*-Knoten $s$ verbundenen *entry*-Knoten existiert in der ACFG-Menge ein Pfad zu einer $x$ benutzenden Anweisung $t$, auf dem $x$ nicht redefiniert wird:

$$s \in V_{ESDG.formalIn}, t \in V_{ESDG.statement} \\ \text{mit } \exists x \in V_{ASG.variable}, n_{1,\ldots,k-1} \in V_{ACFGSet.ACFGVertex} \\ (x \in DEF_s \land x \in USE_t \\ \land s \leftarrow_{ESDG.parameterEdge} \bullet_{ACFGSet.entry} = n_0 \rightarrow_{cf} n_1 \rightarrow_{cf} \ldots \rightarrow_{cf} n_k = t \\ \land x \notin (DEF_{n_1} \cup DEF_{n_2} \cup \ldots \cup DEF_{n_{k-1}}))$$

(v) Von einem über Parameterkanten mit dem $x$ definierenden *formalIn*-Knoten $s$ und dem $x$ benutzenden *formalOut*-Knoten $t$ verbundenen *entry*-Knoten existiert in der ACFG-Menge ein Pfad zu einem *exit*-Knoten, auf dem $x$ nicht redefiniert wird:

$$s \in V_{ESDG.formalIn}, t \in V_{ESDG.formalOut} \\ \text{mit } \exists x \in V_{ASG.variable}, n_{1,\ldots,k-1} \in V_{ACFGSet.ACFGVertex}, entry \in V_{ACFGSet.entry} \\ (x \in DEF_s \land x \in USE_t \\ \land entry = n_0 \rightarrow_{cf} n_1 \rightarrow_{cf} \ldots \rightarrow_{cf} n_k = \bullet_{ACFGSet.exit}$$

$\wedge\ x \notin (DEF_{n_1} \cup DEF_{n_2} \cup \ldots \cup DEF_{n_{k-1}})$
$\wedge\ s \leftarrow_{ESDG.parameterEdge}\ entry \rightarrow_{ESDG.parameterEdge}\ t)$

(vi) Von einem über eine Parameterkante mit dem $x$ definierenden *formalIn*-Knoten $s$ verbundenen *entry*-Knoten existiert in der ACFG-Menge ein Pfad zu einem mit dem $x$ benutzenden *actualIn*-Knoten $t$ verbundenen ACFG-Knoten, auf dem $x$ nicht redefiniert wird:

$s \in V_{ESDG.formalIn}, t \in V_{ESDG.actualIn}$
mit $\exists x \in V_{ASG.variable}, n_{1,\ldots,k-1} \in V_{ACFGSet.ACFGVertex}$
$(x \in DEF_s \wedge x \in USE_t$
$\wedge\ s \leftarrow_{ESDG.parameterEdge} \bullet_{ACFGSet.entry} = n_0 \rightarrow_{cf} n_1 \rightarrow_{cf} \ldots$
$\rightarrow_{cf} n_k = \bullet_{ACFGSet.ACFGVertex} \rightarrow_{ESDG.parameterEdge}\ t$
$\wedge\ x \notin (DEF_{n_1} \cup DEF_{n_2} \cup \ldots \cup DEF_{n_{k-1}}))$

(vii) Von einem über eine Parameterkante mit dem $x$ definierenden *actualOut*-Knoten $s$ verbundenen ACFG-Knoten existiert in der ACFG-Menge ein Pfad zu einem $x$ benutzenden ACFG-Knoten $t$, auf dem $x$ nicht redefiniert wird:

$s \in V_{ESDG.actualOut}, t \in V_{ESDG.statement}$
mit $\exists x \in V_{ASG.variable}, n_{1,\ldots,k-1} \in V_{ACFGSet.ACFGVertex}$
$(x \in DEF_s \wedge x \in USE_t$
$\wedge\ s \leftarrow_{ESDG.parameterEdge} \bullet_{ACFGSet.ACFGVertex} = n_0 \rightarrow_{cf} n_1 \rightarrow_{cf} \ldots \rightarrow_{cf} n_k = t$
$\wedge\ x \notin (DEF_{n_1} \cup DEF_{n_2} \cup \ldots \cup DEF_{n_{k-1}}))$

(viii) Von einem einen Rückgabewert definierenden *actualOut*-Knoten $s$ zu dem $x$ benutzenden ACFG-Knoten $t$, der mit $s$ über eine Parameterkante verbunden ist:

$s \in V_{ESDG.actualOut}, t \in V_{ESDG.statement}$
mit $\exists x \in V_{ASG.variable}$
$(x \in DEF_s \wedge x \in USE_t$
$\wedge\ s \leftarrow_{ESDG.parameterEdge}\ t$
$\wedge\ s \rightarrow_{ESDG.has} \bullet_{ESDG.assignment} \rightarrow_{ASG.defines} \bullet_{ASG.variable}^{identifier=returnValue})$

(ix) Von einem über eine Parameterkante mit dem $x$ benutzenden *formalOut*-Knoten $t$ verbundenen *entry*-Knoten existiert in der ACFG-Menge ein Pfad über einen mit dem $x$ definierenden *actualOut*-Knoten $s$ verbundenen Knoten $n_0$ zu einem *exit*-Knoten, so dass $x$ auf dem Teilpfad von $n_0$ zu dem *exit*-Knoten nicht redefiniert wird:

$s \in V_{ESDG.actualOut}, t \in V_{ESDG.formalOut}$
mit $\exists x \in V_{ASG.variable}, n_{1,\ldots,k-1} \in V_{ACFGSet.ACFGVertex}$
$(x \in DEF_s \wedge x \in USE_t$
$\wedge\ t \leftarrow_{ESDG.parameterEdge} \bullet_{ACFGSet.entry} \rightarrow_{cf} (\bullet_{ACFGSet.ACFGVertex}$
$\rightarrow_{cf})^* n_0 \rightarrow_{cf} n_1 \rightarrow_{cf} \ldots \rightarrow_{cf} n_k = \bullet_{ACFGSet.exit}$

$\wedge\, n_0 \rightarrow_{ESDG.parameterEdge} s$
$\wedge\, x \notin (DEF_{n_1} \cup DEF_{n_2} \cup \ldots \cup DEF_{n_{k-1}}))$

(x) Von einem über eine Parameterkante mit dem $x$ definierenden *actualOut*-Knoten $s$ verbundenen ACFG-Knoten existiert in der ACFG-Menge ein Pfad zu einem über Parameterkanten mit dem $x$ benutzenden *actualIn*-Knoten $t$ verbundenen ACFG-Knoten, auf dem $x$ nicht redefiniert wird. Des Weiteren wird von beiden ACFG-Knoten eine Botschaft an dasselbe Objekt gesendet:

$s \in V_{ESDG.actualOut}, t \in V_{ESDG.actualIn}$
mit $\exists x, o \in V_{ASG.variable}, n_{1,\ldots,k-1} \in V_{ACFGSet.ACFGVertex}$
$(x \in DEF_s \wedge x \in USE_t$
$\wedge\, s \leftarrow_{ESDG.parameterEdge} \bullet_{ACFGSet.ACFGVertex} = n_0 \rightarrow_{cf} n_1 \rightarrow_{cf} \ldots$
$\rightarrow_{cf} n_k = \bullet_{ACFGSet.ACFGVertex} \rightarrow_{ESDG.parameterEdge} t$
$\wedge\, x \notin (DEF_{n_1} \cup DEF_{n_2} \cup \ldots \cup DEF_{n_{k-1}})$
$\wedge\, (n_0 \rightarrow_{ASG.hasAsSelection} o \vee n_0 \rightarrow (\bullet_{ASG.compoundExpression}$
$\rightarrow_{ASG.hasAsOperand})^* \bullet_{ASG.functionCall} \rightarrow_{ASG.hasAsSelection} o)$
$\wedge\, (n_k \rightarrow_{ASG.hasAsSelection} o \vee n_k \rightarrow (\bullet_{ASG.compoundExpression}$
$\rightarrow_{ASG.hasAsOperand})^* \bullet_{ASG.functionCall} \rightarrow_{ASG.hasAsSelection} o)$

Für die Deklarationskanten gilt:

$s \rightarrow'_{ESDG.declarationEdge} t \;\Leftrightarrow$

Von einer $x$ deklarierenden lokalen Variablendeklaration (*localVariableDeclaration*) $s$ existiert in der ACFG-Menge ein Pfad zu einem $x$ definierenden oder benutzenden Knoten $t$, auf dem $x$ nicht definiert oder benutzt wird:

$s \in V_{ESDG.localVariableDeclaration}, t \in V_{ESDG.ESDGVertex}$
mit $\exists x \in V_{ASG.variable}, n_{1,\ldots,k-1} \in V_{ACFGSet.ACFGVertex}$
$(x \in DEC_s \wedge x \in (DEF_t \cup USE_t)$
$\wedge\, s = n_0 \rightarrow_{cf} n_1 \rightarrow_{cf} \ldots \rightarrow_{cf} n_k = t$
$\wedge\, x \notin (DEF_{n_1} \cup DEF_{n_2} \cup \ldots \cup DEF_{n_{k-1}} \cup USE_{n_1}$
$\cup\, USE_{n_2} \cup \ldots \cup USE_{n_{k-1}}))$ □

### 5.6.5. Transitive Kanten

Eine transitive Kante (*summaryEdge*) kann ausschließlich zwischen einem *actualIn*- und einem *actualOut*-Knoten verlaufen, falls diese an denselben Knoten geknüpft sind. Weiterhin muss der Wert des Ausgabeparameters von demjenigen des Eingabeparameters abhängen. Dies ist gegeben, wenn ein Pfad, bestehend aus Kanten des ESDG, zwischen den beiden

Knoten existiert. Einschränkend dabei ist, dass im Pfad enthaltene Parameter-out-Kanten, über die von einer Methode zu deren Aufruf „aufgestiegen“ wird, an denselben Aufruf anknüpfen müssen wie die Parameter-in- oder Aufrufkanten, über die der Pfad in die Methode „hinabgestiegen“ ist und umgekehrt.

**Definition**
Sei $G = (V, E)$ ein Graph, der einen erweiterten Systemabhängigkeitsgraphen, soweit in den Abschnitten 5.6.1 bis 5.6.4 definiert, enthält. Sei weiterhin

$$\begin{aligned} \rightarrow_{esdgEdge} \quad = \quad & \rightarrow_{ESDG.parameterEdge} \cup \rightarrow_{ESDG.parameterInEdge} \cup \rightarrow_{ESDG.parameterOutEdge} \cup \\ & \rightarrow_{ESDG.callEdge} \cup \rightarrow_{ESDG.controlDependenceEdge} \cup \rightarrow_{ESDG.declarationEdge} \cup \\ & \rightarrow_{ESDG.dataFlowEdge} \end{aligned}$$

die Menge aller ESDG-Kanten (Parameter, Parameter-in-, Parameter-out-, Aufruf-, Kontroll-, Deklarations- und Datenflusskanten).

Ein um transitive Kanten erweiterter Graph $G' = (V, E')$ geht aus $G$ hervor mit $s \rightarrow'_{ESDG.summaryEdge} t \Leftrightarrow$

> Von einem *actualIn*-Knoten $s$ existiert ein Pfad zu einem *actualOut*-Knoten $t$, bestehend aus den oben definierten Kanten vom Typ *esdgEdge*. Genau dann wenn dieser Pfad eine Parameter-out-Kante enthält, muss er auch eine Aufrufkante oder Parameter-in-Kante enthalten, so dass diese Kanten alle mit demselben Knoten $c$ verknüpft sind, bzw. mit den Parameterknoten $ai$ und $ao$, die dann ihrerseits mit $c$ verbunden sind.
>
> $s \in V_{ESDG.actualIn}, t \in V_{ESDG.actualOut}$
> mit $s \rightarrow^*_{esdgEdge} t$
> $\wedge \forall ai \in V_{ESDG.actualIn}, ao \in V_{ESDG.actualOut}, c \in V_{ESDG.ESDGVertex}$
> $(((c \rightarrow_{ESDG.parameterEdge} ai \wedge s \rightarrow^*_{esdgEdge} ai \rightarrow_{ESDG.parameterInEdge} \bullet_{ESDG.formalIn} \rightarrow^*_{esdgEdge} t) \vee s \rightarrow^*_{esdgEdge} c \rightarrow_{ESDG.callEdge} \bullet_{ESDG.entry} \rightarrow^*_{esdgEdge} t)$
> $\Leftrightarrow (c \rightarrow_{ESDG.parameterEdge} ao \wedge s \rightarrow^*_{esdgEdge} \bullet_{ESDG.formalOut} \rightarrow_{ESDG.parameterOutEdge} ao \rightarrow^*_{esdgEdge} t))$ □

# 6. Spezifikationen der Dienste

In diesem Kapitel werden die in Kapitel vier eingeführten Dienste anhand der Definition in Abschnitt 3.2 spezifiziert. Diese beinhaltet eine knappe Zusammenfassung der Funktion eines Dienstes, seine Ein- und Ausgaben sowie seine Zusammensetzung aus Teildiensten. Darauf folgt eine konkretere Beschreibung des Dienstes, wobei das Hauptaugenmerk auf dem Dienst *sliceProgram* und seinen Teildiensten liegt.

Die Ausführungen zu den Diensten *preprocessProgram* und *convertESDGToCode* wurden bewusst kurz gehalten, da entweder eine genauere Betrachtung die Grenzen dieser Arbeit überschreitet (z.B. bei *computeAbstractSyntaxGraph*, *computePointsToGraphs*), oder aber aus der Beschreibung des Datenmodells in Kapitel fünf bereits Informationen über die Arbeitsweise des Dienstes entnehmbar sind (z.B. bei *computeAugmentedControlFlowGraphs*, *computeExtendedSystemDependenceGraph*). Letzteres hängt damit zusammen, dass sich die Definition eines Graphen immer auf diejenigen Graphen stützt, welche der erzeugende Dienst als Eingabe fordert.

## 6.1. *preprocessProgram*

**Funktion**: Überführung des Programmcodes in einen erweiterten Systemabhängigkeitsgraphen

**Eingaben**:
- *program code*

**Ausgaben**:
- *ESDG*
- *ACFGSet*

**zusammengesetzt aus**:
- *1.1 computeAbstractSyntaxGraph*
- *1.2 computeAugmentedControlFlowGraphs*
- *1.3 computePointsToGraphs*
- *1.4 computeExtendedCallGraph*
- *1.5 computeExtendedSystemDependenceGraph*

*preprocessProgram* erstellt aus dem Programmcode als Eingabe den erweiterten Systemabhängigkeitsgraphen, der als Grundlage für die eigentlichen, in dem Dienst *sliceProgram* enthaltenen Slicing-Verfahren dient. Dazu wird der Code sukzessive in die Darstellungen abstrakter Syntaxgraph, erweiterte Kontrollflussgraphen, points-To-Graphen und erweiterter Aufrufgraph überführt. Aus diesen erfolgt abschließend die Berechnung des ESDG.

### 6.1.1. *computeAbstractSyntaxGraph*

**Funktion**: Überführung des Programmcodes in einen abstrakten Syntaxgraphen

**Eingaben**:
- *program code*

**Ausgaben**:
- *ASG*

Aus dem Programmcode wird unter Verwendung von Techniken wie der lexikalischen und syntaktischen Analyse ein abstrakter Syntaxgraph nach Maßgabe der Beschreibung in Abschnitt 5.2 erzeugt. Diese Verfahren werden im Compilerbau angewandt und sind beispielsweise [Aho+86] zu entnehmen.

### 6.1.2. *computeAugmentedControlFlowGraphs*

**Funktion**: Berechnung der Menge der erweiterten Kontrollflussgraphen

**Eingaben**:
- *ASG*

**Ausgaben**:
- *ACFGSet*

Dieser Dienst bestimmt die Menge der erweiterten Kontrollflussgraphen, d.h. ein ACFG pro Methode, unter Berücksichtigung der Definition in Abschnitt 5.3. Diese beruht allein auf dem abstrakten Syntaxgraphen.

Der ACFG legt alle möglichen Abarbeitungsabfolgen der in einer Methode enthaltenen Anweisungen dar. Zusätzlich enthält er Kanten, die von einer Sprunganweisung zu derjenigen Anweisung führen, welche bei Nichtausführung des Sprunges abgearbeitet würde.

### 6.1.3. *computePointsToGraphs*

**Funktion**: Berechnung der Menge der Points-to-Graphen

**Eingaben**:

- *ASG*

**Ausgaben**:

- *PGSet*

*computePointsToGraphs* bestimmt die Menge der Points-to-Graphen, die für jede Methode einen PG enthält. Die Anforderungen an die Points-to-Graphen sind Abschnitt 5.4 zu entnehmen. Da das in dieser Arbeit entwickelte Modell auf einer kontext- und flussinsensitiven Zeigeranalyse basiert, sind für diesen Dienst geeignete Algorithmen zu verwenden (z.B. [And94], [Ste96] oder [Sha$^{+}$97]).

Ein Points-to-Graph stellt eine konservative Abschätzung der „zeigt auf“-Beziehungen zwischen den Variablen einer Methode dar.

### 6.1.4. *computeExtendedCallGraph*

**Funktion**: Berechnung des erweiterten Aufrufgraphen

**Eingaben**:

- *ASG*
- *PGSet*

**Ausgaben**:

- *ECG*

**zusammengesetzt aus**:

- *1.4.1 computeCallGraph*
- *1.4.2 computeDefUseInformation*

Der Dienst *computeExtendedCallGraph* ermittelt den, in Abschnitt 5.5 beschriebenen, erweiterten Aufrufgraphen. Dazu wird von den beiden Teildiensten zunächst ein „normaler“, die Aufrufbeziehungen zwischen Methoden anzeigender Aufrufgraph berechnet und dieser dann um Informationen zu von den Methoden definierten und benutzten Variablen ergänzt.

#### 6.1.4.1. *computeCallGraph*

**Funktion**: Berechnung des Aufrufgraphen

**Eingaben**:
- *ASG*
- *PGSet*

**Ausgaben**:
- *CG*

Dieser Dienst berechnet den Aufrufgraphen unter Verwendung des abstrakten Syntaxgraphen und der Menge der Points-to-Graphen. Letztere wird benötigt, um Zeiger auf Objekte aufzulösen, deren Methoden aufgerufen werden.

#### 6.1.4.2. *computeDefUseInformation*

**Funktion**: Ergänzen des CG um Informationen über von den Methoden definierte und benutzte Variablen

**Eingaben**:
- *CG*
- *ASG*
- *PGSet*

**Ausgaben**:
- *ECG*

Mit Hilfe des abstrakten Syntaxgraphen und den Points-to-Graphen werden die von einer Methode $m$ definierten und die von ihr benutzten Variablen bestimmt. Dabei sind auch Variablen zu berücksichtigen, die von in $m$ aufgerufenen Methoden definiert oder benutzt werden. Die Integration dieser Informationen in den Aufrufgraphen lässt den erweiterten Aufrufgraphen entstehen.

### 6.1.5. *computeExtendedSystemDependenceGraph*

**Funktion**: Berechnung des erweiterten Systemabhängigkeitsgraphen

**Eingaben**:
- *ASG*
- *ACFGSet*

- *PGSet*
- *ECG*

**Ausgaben**:
- *ESDG*

**zusammengesetzt aus**:
- *1.5.1 computeBasicESDG*
- *1.5.2 computeInterMethodEdges*
- *1.5.3 computeControlDependenceEdges*
- *1.5.4 computeDataFlowEdges*
- *1.5.5 computeSummaryEdges*

Dieser aus fünf Teildiensten zusammengesetzte Dienst erzeugt unter Inanspruchnahme des abstrakten Syntaxgraphen, der Menge der erweiterten Kontrollflussgraphen, der Menge der Points-to-Graphen und des erweiterten Aufrufgraphen den erweiterten Systemabhängigkeitsgraphen (siehe Abschnitt 5.6). Dabei wird zunächst dessen Grundgerüst erstellt und nacheinander um interprozedurale Kanten, Kontrollkanten, Datenfluss- und Deklarationskanten sowie transitive Kanten ergänzt.

#### 6.1.5.1. *computeBasicESDG*

**Funktion**: Erzeugung des Grundgerüsts des erweiterten Systemabhängigkeitsgraphen (ohne die von den vier anderen Teildiensten von *computeExtendedSystemDependenceGraph* noch zu berechnenden Kanten)

**Eingaben**:
- *ASG*
- *ECG*

**Ausgaben**:
- *ESDG (edges incomplete)*

*computeBasicESDG* bestimmt, unter Benutzung des abstrakten Syntaxgraphen und des erweiterten Aufrufgraphen, die in dem erweiterten Systemabhängigkeitsgraphen enthaltenen Knoten. Außerdem werden über Parameterkanten die *formalIn*- und *formalOut*-Knoten an die *entry*-Knoten der Methoden sowie die *actualIn*- und *actualOut*-Knoten an die eine Methode aufrufenden *entry*-Knoten geknüpft (siehe Abschnitt 5.6.1). An *return*-Anweisungen wird eine Zuweisung geknüpft, die einer Hilfsvariable mit dem Bezeichner *returnValue* den Rückgabewert zuweist.

#### 6.1.5.2. *computeInterMethodEdges*

**Funktion**: Hinzufügen der Aufruf-, Parameter-in- und Parameter-out-Kanten zu dem ESDG

**Eingaben**:
- *ESDG (edges incomplete)*
- *ASG*
- *ECG*

**Ausgaben**:
- *ESDG (edges incomplete)*

Die *entry*-Knoten einer Methode sowie diejenigen Knoten, welche diese Methode aufrufen, werden über Aufrufkanten miteinander verbunden. Des Weiteren erfolgt die Verknüpfung der korrespondierenden *formalIn*- und *actualIn*- sowie der *formalOut*- und *actualOut*-Knoten über Parameter-in- bzw. Parameter-out-Kanten (siehe Abschnitt 5.6.2).

#### 6.1.5.3. *computeControlDependenceEdges*

**Funktion**: Hinzufügen der Kontrollkanten zu dem ESDG

**Eingaben**:
- *ESDG (edges incomplete)*
- *ACFGSet*

**Ausgaben**:
- *ESDG (edges incomplete)*

Dieser Dienst berechnet unter Zuhilfenahme der Menge der erweiterten Kontrollflussgraphen die Kontrollkanten (siehe Abschnitt 5.6.3) des ESDG.

Kontrollkanten spiegeln in Programmen ohne Sprünge die Schachtelungshierarchie der Anweisungen wider. Sind Sprunganweisungen enthalten, so verfügen diese ebenfalls über ausgehende Kontrollkanten zu denjenigen Knoten, zu deren Abarbeitung es lediglich bei Nichtausführung des Sprunges kommen würde.

#### 6.1.5.4. *computeDataFlowEdges*

**Funktion**: Hinzufügen der Datenfluss- und Deklarationskanten zu dem ESDG

**Eingaben**:
- *ESDG (edges incomplete)*

- *ASG*
- *ACFGSet*
- *PGSet*

**Ausgaben**:
- *ESDG (edges incomplete)*

Die Datenfluss- und Deklarationskanten (siehe Abschnitt 5.6.4) werden mit Hilfe des abstrakten Syntaxgraphen sowie den Mengen der erweiterten Kontrollflussgraphen und Points-to-Graphen ermittelt.

Eine Datenflusskante zwischen zwei Knoten $v$ und $w$ sagt aus, dass $w$ eine von $v$ definierte Variable nutzt, ohne dass diese zwischen der Ausführung der beiden Knoten zwingend redefiniert würde. Eine ähnliche Bedeutung haben die Deklarationskanten inne: Knoten $w$ definiert oder benutzt eine von $v$ deklarierte Variable.

#### 6.1.5.5. *computeSummaryEdges*

**Funktion**: Hinzufügen der transitiven Kanten zu dem ESDG

**Eingaben**:
- *ESDG (edges incomplete)*

**Ausgaben**:
- *ESDG (edges incomplete)*

*computeSummaryEdges* bestimmt die von *actualIn-* zu *actualIn*-Knoten verlaufenden transitiven Kanten (siehe Abschnitt 5.6.5). Eine existierende transitive Kante weist darauf hin, dass der Wert des Ausgabeparameters von dem des Eingabeparameters abhängt.

## 6.2. *sliceProgram*

**Funktion**: Durchführung der verschiedenen Slicing- und Choppingverfahren auf dem ESDG

**Eingaben**:
- *ESDG*
- *ACFGSet*
- *slicing criterion*
- *chopping criterion*
- *trace*

**Ausgaben**:
- *ESDG (with marked slice or chop)*

**zusammengesetzt aus**:
- *2.1 markESDG*
- *2.2 computeStaticSlice*
- *2.3 computeDynamicSlice*
- *2.4 computeStaticChop*
- *2.5 computeDynamicChop*
- *2.6 computeExecutableBackwardSlice*

Dieser Dienst führt das eigentliche Program Slicing, d.h. die Berechnung eines Programmausschnitts, durch. Dazu werden zunächst das Slicing- oder Chopping-Kriterium und gegebenenfalls die Trace im erweiterten Systemabhängigkeitsgraphen markiert. Anschließend wird die Slice bzw. der Chop ermittelt. Unterstützt wird die Berechnung von statischen und dynamischen Vorwärtsslices, Rückwärtsslices und Chops. Rückwärtsslices lassen sich von einem eigens dafür zuständigen Teildienst ausführbar machen.

### 6.2.1. *markESDG*

**Funktion**: Markierung des Slicing-Kriteriums, des Chopping-Kriteriums und/oder der Trace im ESDG

**Eingaben**:
- *ESDG*
- *ACFGSet*
- *slicing criterion*
- *chopping criterion*
- *trace*

**Ausgaben**:
- *ESDG (with marked slicing criterion)*
- *ESDG (with marked slicing criterion, marked trace)*
- *ESDG (with marked chopping criterion)*
- *ESDG (with marked chopping criterion, marked trace)*

**zusammengesetzt aus**:
- *2.1.1 markSlicingCriterion*
- *2.1.2 markChoppingCriterion*
- *2.1.3 markTrace*

Die Teildienste von *markESDG* sind für die Kennzeichnung des Slicing-Kriteriums, des Chopping-Kriteriums sowie der Trace einer Programmausführung im erweiterten Systemabhängigkeitsgraphen verantwortlich. Der erweiterte Kontrollflussgraph wird eventuell benötigt, um das Slicing-Kriterium zu markieren (siehe Abschnitt 6.2.1.1).

#### 6.2.1.1. *markSlicingCriterion*

**Funktion**: Markierung des Slicing-Kriteriums im ESDG

**Eingaben**:
- *ESDG*
- *ACFGSet*
- *slicing criterion*

**Ausgaben**:
- *ESDG (with marked slicing criterion)*

Falls die Variablenmenge $X$ des Slicing-Kriteriums $< v, X >$ allen von dem ESDG-Knoten $v$ definierten oder benutzten Variablen entspricht, wird $v$ markiert, indem dessen Attribut *isInSliceCrit* auf *true* gesetzt wird.

Für den Fall, dass $X \neq (DEF_v \cup USE_v)$, werden alle Knoten markiert, die eine Variable $x \in X$ definieren und deren Definitionen $v$ erreichen. Zur Berechnung dieser Knoten wird der ACFG benötigt (siehe [Tip95, S. 27]).

**Definition (ESDG-Knoten mit Attributwert *isInSliceCrit = true*)**
Sei $< v, X >$ ein Slicing-Kriterium, wobei $v \in V_{ESDGVertex}$ ein Knoten des ESDG und $X \subseteq V_{ASG.variable}$ eine Variablenmenge darstellt. Seien weiterhin die Mengen der von $v$ definierten und benutzten Variablen, $DEF_v$ bzw. $USE_v$, wie in Abschnitt 5.6.4 definiert sowie

$$\rightarrow_{cf} \quad = \quad \rightarrow^{label=always}_{ACFGSet.controlFlowEdge} \cup \rightarrow^{label=true}_{ACFGSet.controlFlowEdge} \cup \rightarrow^{label=false}_{ACFGSet.controlFlowEdge}$$

die Menge aller Kontrollflusskanten, die nicht mit *never* attributiert sind.

Die Menge $SliceCrit$ der ESDG-Knoten, die mit *isInSliceCrit = true* attributiert werden, ist gegeben durch:

$$SliceCrit \quad = \quad \Big\{v\Big\}, \text{ falls } X = (DEF_v \cup USE_v) \tag{1}$$

$$SliceCrit \quad = \quad \Big\{w \Big| \exists x \in X, n_{1,\ldots,k-1} \in V_{ACFGSet.ACFGVertex} \tag{2}$$

$$\Big(x \in DEF_w \tag{3}$$

$$\wedge \Big(w = n_0 \rightarrow_{cf} n_1 \rightarrow_{cf} \ldots \rightarrow_{cf} n_k = v \tag{4}$$

$$\vee\ v \leftarrow_{ESDG.parameterEdge} \bullet_{ACFGSet.entry} \rightarrow_{cf} (\bullet_{ACFGSet.ACFGVertex} \rightarrow_{cf})^{*} w = n_0 \rightarrow_{cf} n_1 \rightarrow_{cf} \ldots \rightarrow_{cf} n_k = \bullet_{ACFGSet.exit} \tag{5}$$

$$\vee\ w = n_0 \rightarrow_{cf} n_1 \rightarrow_{cf} \ldots \rightarrow_{cf} n_k = \bullet_{ACFGSet.ACFGVertex} \rightarrow_{ESDG.parameterEdge} v \tag{6}$$

$$\vee\ w \leftarrow_{ESDG.parameterEdge} \bullet_{ACFGSet.entry} = n_0 \rightarrow_{cf} n_1 \rightarrow_{cf} \ldots \rightarrow_{cf} n_k = v \tag{7}$$

$$\vee\ (entry = n_0 \rightarrow_{cf} n_1 \rightarrow_{cf} \ldots \rightarrow_{cf} n_k = \bullet_{ACFGSet.exit} \wedge v \leftarrow_{ESDG.parameterEdge} entry \rightarrow_{ESDG.parameterEdge} w) \tag{8}$$

$$\vee\ w \leftarrow_{ESDG.parameterEdge} \bullet_{ACFGSet.entry} = n_0 \rightarrow_{cf} n_1 \rightarrow_{cf} \ldots \rightarrow_{cf} n_k = \bullet_{ACFGSet.ACFGVertex} \rightarrow_{ESDG.parameterEdge} v \tag{9}$$

$$\vee\ w \leftarrow_{ESDG.parameterEdge} \bullet_{ACFGSet.ACFGVertex} = n_0 \rightarrow_{cf} n_1 \rightarrow_{cf} \ldots \rightarrow_{cf} n_k = v \tag{10}$$

$$\vee\ w \leftarrow_{ESDG.parameterEdge} v \wedge w \rightarrow_{defines} \bullet_{ASG.variable}^{identifier=returnValue} \tag{11}$$

$$\vee\ w \leftarrow_{ESDG.parameterEdge} \bullet_{ACFGSet.ACFGVertex} = n_0 \rightarrow_{cf} n_1 \rightarrow_{cf} \ldots \rightarrow_{cf} n_k = \bullet_{ACFGSet.entry} \rightarrow_{ESDG.parameterEdge} v \tag{12}$$

$$\vee\ \exists o \in V_{ASG.variable}$$

$$\Big(w \leftarrow_{ESDG.parameterEdge} \bullet_{ACFGSet.ACFGVertex} = n_0 \rightarrow_{cf} n_1 \rightarrow_{cf} \rightarrow_{cf} n_k = \bullet_{ACFGSet.ACFGVertex} \rightarrow_{ESDG.parameterEdge} v \tag{13}$$

$$\wedge\ (n_0 \rightarrow_{ASG.hasAsSelection} o \vee n_0 \rightarrow (\bullet_{ASG.compoundExpression} \rightarrow_{ASG.hasAsOperand})^{*} \bullet_{ASG.functionCall} \rightarrow_{ASG.hasAsSelection} o)$$

$$\wedge\ (n_k \rightarrow_{ASG.hasAsSelection} o \vee n_k \rightarrow (\bullet_{ASG.compoundExpression} \rightarrow_{ASG.hasAsOperand})^{*} \bullet_{ASG.functionCall} \rightarrow_{ASG.hasAsSelection} o)\Big)\Big)$$

$$\wedge\ x \notin (DEF_{n_1} \cup DEF_{n_2} \cup \ldots \cup DEF_{n_{k-1}})\Big)\Big\}, \tag{14}$$

falls $X \neq (DEF_v \cup USE_v)$

(1) Falls alle Variablen in $X$ von $v$ definiert oder benutzt werden, ist nur $v$ in $SliceCrit$ enthalten.

(2) Werden nicht alle Variablen in $X$ von $v$ definiert oder benutzt, so sind alle Knoten $w$ in $SliceCrit$ enthalten, die

(3) $x$ definieren und für die eine der folgenden Bedingungen zutrifft:

(4) von der Anweisung $w$ existiert in der ACFG-Menge ein Pfad zu der Anweisung $v$, oder

(5) von dem mit dem *formalOut*-Knoten $v$ über eine Parameterkante (*parameterEdge*) verbundenen *entry*-Knoten existiert in der ACFG-Menge ein Pfad über $w$ zu einem *exit*-Knoten, oder

(6) von der Anweisung $w$ existiert in der ACFG-Menge ein Pfad zu dem über eine Parameterkante mit dem *actualIn*-Knoten $v$ verbundenen Knoten, oder

(7) von dem mit dem *formalIn*-Knoten $w$ verbundenen *entry*-Knoten existiert in der ACFG-Menge ein Pfad zu der Anweisung $v$, oder

(8) von dem mit dem *formalIn*-Knoten $w$ und dem *formalOut*-Knoten $v$ verbundenen *entry*-Knoten existiert in der ACFG-Menge ein Pfad zu einem *exit*-Knoten, oder

(9) von dem mit dem *formalIn*-Knoten $w$ verbundenen *entry*-Knoten existiert in der ACFG-Menge ein Pfad zu dem über eine Parameterkante mit dem *actualIn*-Knoten $v$ verbundenen Knoten, oder

(10) von dem mit dem *actualOut*-Knoten $w$ verbundenen ACFG-Knoten existiert in der ACFG-Menge ein Pfad zu Knoten $v$, oder

(11) von einem einen Rückgabewert definierenden *actualOut*-Knoten $w$ zu dem Knoten $v$, der mit $w$ verbunden ist, oder

(12) von dem mit dem *actualOut*-Knoten $w$ verbundenen Knoten existiert in der ACFG-Menge ein Pfad zu dem mit dem *formalOut*-Knoten $v$ verbundenen *entry*-Knoten, oder

(13) von einem mit dem *actualOut*-Knoten $w$ verbundenen ACFG-Knoten existiert ein Pfad zu dem mit dem *actualIn*-Knoten $t$ verbundenen ACFG-Knoten und es wird von beiden ACFG-Knoten eine Botschaft an dasselbe Objekt gesendet.

(14) Die Variable $x$ wird von den Knoten $n_1$ bis $n_{k-1}$ auf dem Kontrollflusspfad nicht definiert. □

#### 6.2.1.2. *markChoppingCriterion*

**Funktion**: Markierung des Chopping-Kriteriums im ESDG

**Eingaben**:
- *ESDG*
- *chopping criterion*

**Ausgaben**:
- *ESDG (with marked chopping criterion)*

Dieser Dienst markiert die beiden im Chopping-Kriterium $< v, w >$, $v, w \in V_{ESDG.ESDGVertex}$ enthaltenen Knoten im ESDG, indem deren Attribute *isFirstInChopCrit* (für $v$) bzw. *isSecondInChopCrit* (für $w$) auf *true* gesetzt werden.

#### 6.2.1.3. *markTrace*

**Funktion**: Markierung der Trace im ESDG

**Eingaben**:
- *ESDG (with marked slicing criterion)*
- *ESDG (with marked chopping criterion)*
- *trace*

**Ausgaben**:
- *ESDG (with marked slicing criterion, marked trace)*
- *ESDG (with marked chopping criterion, marked trace)*

Mit Hilfe der Trace $< v_1, v_2, \ldots, v_n >$, der Sequenz der bei der Programmausführung abgearbeiteten Anweisungen $v_{i,i \in \mathbb{N}} \in V_{ESDG.statement}$, können die aufgetretenen Abhängigkeiten zwischen den Programmanweisungen analysiert werden. Die entsprechenden Kanten des ESDG werden markiert, indem deren Attribut *isExecuted* den Wert *true* erhält.

Die folgende Übersicht beschreibt, wie die Markierung der Kanten des Systemabhängigkeitsgraphen anhand der Trace erfolgt.

**Parameter-, Aufruf-, Parameter-in-, Parameter-out- und transitive Kanten**: Handelt es sich bei einem Knoten $v$ in $< v_1, v_2, \ldots, v_n >$ um eine Aufrufanweisung oder ist ein Funktionsaufruf enthalten, so werden die von $v$ ausgehenden Aufruf- und Parameterkanten sowie die an die *actualIn*- und *actualOut*-Knoten geknüpften Parameter-in- bzw. Parameter-out-Kanten als ausgeführt markiert. Dasselbe gilt für transitive Kanten zwischen diesen an $v$ gebundenen *actualIn*- und *actualOut*-Knoten. Ebenfalls markiert werden Parameterkanten, welche in die entsprechenden *formalIn*- und *formalOut*-Knoten eingehen.

**Kontrollkanten**: Verläuft zwischen zwei Knoten $v_i, v_j$ in $< v_1, v_2, \ldots, v_n >$ mit $i < j$ eine Kontrollkante, so wird diese als ausgeführt markiert.

**Datenfluss- und Deklarationskanten**: Existiert zwischen zwei Knoten $v_i, v_j$ in $< v_1, v_2, \ldots, v_n >$ mit $i < j$ eine Datenfluss- oder Deklarationskante von $v_i$ nach $v_j$, so wird diese als ausgeführt markiert. Eine Ausnahme kann bestehen, falls es einen Knoten $v_k$ in $< v_1, v_2, \ldots, v_n >$ gibt mit $i < k < j$ und einer Datenfluss- oder Deklarationskante von $v_k$ nach $v_j$. Ist die Menge der von $v_k$ definierten bzw. deklarierten Variablen eine Obermenge der von $v_i$ definierten oder deklarierten Variablen, so wird die Kante von $v_i$ nach $v_j$ nicht markiert.

### 6.2.2. *computeStaticSlice*

**Funktion**: Berechnung einer statischen Slice

**Eingaben**:
- *ESDG (with marked slicing criterion)*

**Ausgaben**:
- *ESDG (with marked slice)*

**zusammengesetzt aus**:
- *2.2.1 computeStaticBackwardSlice*
- *2.2.2 computeStaticForwardSlice*

*computeStaticSlice* enthält je einen Teildienst zur Berechnung statischer Rückwärts- und Vorwärtsslices.

#### 6.2.2.1. *computeStaticBackwardSlice*

**Funktion**: Berechnung einer statischen Rückwärtsslice

**Eingaben**:
- *ESDG (with marked slicing criterion)*

**Ausgaben**:
- *ESDG (with marked slice)*

Ausgehend von einer das Slicing-Kriterium repräsentierenden Knotenmenge im erweiterten Systemabhängigkeitsgraphen $\{v|v \in V_{ESDGVertex}^{isInSliceCrit=true}\}$ ist die Menge aller über das rückwärtige Traversieren von Kontroll-, Datenfluss-, Deklarations-, Parameter-, Parameter-in- und Aufrufkanten sowie transitiven Kanten erreichbaren Knoten in der statischen Rückwärtsslice enthalten. Von den bei der ersten Traversierung erreichten Knoten ausgehend, befindet sich in der Slice außerdem die Menge aller über das rückwärtige Traversieren von Kontroll-, Datenfluss-, Deklarations-, Parameter- und Parameter-out-Kanten sowie transitiven Kanten erreichbaren Knoten.

Das Attribut *isInSliceOrChop* der in der Menge enthalten Knoten wird auf *true* gesetzt.

Abbildung 6.1 enthält das Beispiel eines ESDG aus Abschnitt 5.6 mit der markierten statischen Rückwärtsslice für das Slicing-Kriterium $< 12, \{a, p\} >$. Wie der Abbildung zu entnehmen ist, besteht die Slice hier aus nahezu dem gesamten Programmausschnitt, ausgenommen die Parameterknoten der Methode *m1*.

**Definition (statische Rückwärtsslice)**
Die Menge $SBS$ der in einer statischen Rückwärtsslice enthaltenen Knoten ist gegeben durch:

$$SBS = \Big\{v, w \Big| \tag{1}$$

$$\bullet^{isInSliceCrit=true}_{ESDG.ESDGVertex}(\leftarrow_{ESDG.controlDependenceEdge} \vee \leftarrow_{ESDG.dataFlowEdge} \tag{2}$$
$$\vee \leftarrow_{ESDG.declarationEdge} \vee \leftarrow_{ESDG.parameterEdge} \vee \leftarrow_{ESDG.parameterInEdge}$$
$$\vee \leftarrow_{ESDG.callEdge} \vee \leftarrow_{ESDG.summaryEdge})^* \, v$$

$$(\leftarrow_{ESDG.controlDependenceEdge} \vee \leftarrow_{ESDG.dataFlowEdge} \vee \leftarrow_{declarationEdge} \tag{3}$$
$$\vee \leftarrow_{ESDG.parameterEdge} \vee \leftarrow_{ESDG.parameterOutEdge}$$
$$\vee \leftarrow_{ESDG.summaryEdge})^* w \Big\}$$

(1) $SBS$ enthält alle Knoten $v, w$, die

(2) von den Knoten des Slicing-Kriteriums rückwärts über Kontroll-, Datenfluss-, Deklarations-, Parameter-,Parameter-in- und Aufrufkanten sowie transitiven Kanten erreichbar und

(3) von den in (2) erreichten Knoten rückwärts über Kontroll-, Datenfluss-, Deklarations-, Parameter- und Parameter-out-Kanten sowie transitiven Kanten erreichbar sind. □

#### 6.2.2.2. *computeStaticForwardSlice*

**Funktion**: Berechnung einer statischen Vorwärtsslice

**Eingaben**:
- *ESDG (with marked slicing criterion)*

**Ausgaben**:
- *ESDG (with marked slice)*

Ausgehend von einer das Slicing-Kriterium repräsentierenden Knotenmenge im erweiterten Systemabhängigkeitsgraphen $\{v | v \in V^{isInSliceCrit=true}_{ESDGVertex}\}$ ist die Menge aller über das (Vorwärts-) Traversieren von Kontroll-, Datenfluss-, Deklarations-, Parameter- und Parameter-out-Kanten sowie transitiven Kanten erreichbaren Knoten in der statischen Vorwärtsslice enthalten. Von den bei der ersten Traversierung erreichten Knoten ausgehend, befindet sich in der Slice außerdem die Menge aller über das Traversieren von Kontroll-, Datenfluss-, Deklarations-, Parameter-, Parameter-in- und Aufrufkanten sowie transitiven Kanten erreichbaren Knoten.

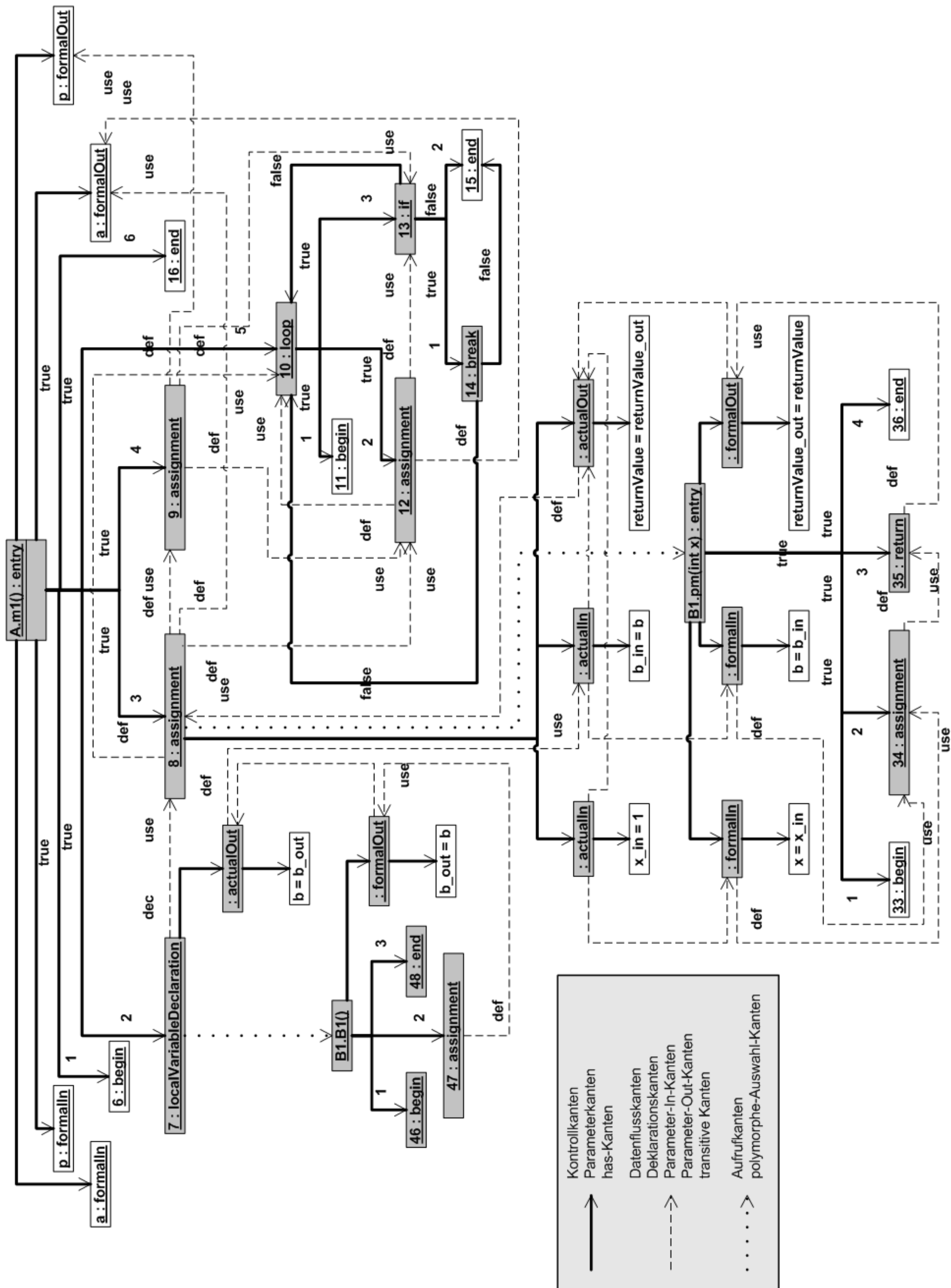


Abbildung 6.1.: Statische Rückwärtsslice für das Slicing-Kriterium $< 12, \{a, p\} >$. Die in der Slice enthaltenen Knoten sind grau unterlegt.

Das Attribut *isInSliceOrChop* der in der Menge enthalten Knoten wird auf *true* gesetzt.

Abbildung 6.2 enthält das Beispiel eines ESDG aus Abschnitt 5.6 mit der markierten statischen Rückwärtsslice für das Slicing-Kriterium $< 12, \{a, p\} >$.

**Definition (statische Vorwärtsslice)**
Die Menge $SFS$ der in einer statischen Rückwärtsslice enthaltenen Knoten ist gegeben durch:

$$
\begin{aligned}
SFS \;=\; \Big\{v, w \Big| \\
& \vee \; \bullet^{isInSliceCrit=true}_{ESDG.ESDGVertex} \big(\rightarrow_{ESDG.controlDependenceEdge} \vee \rightarrow_{ESDG.dataFlowEdge} && (1)\\
& \vee \rightarrow_{ESDG.declarationEdge} \vee \rightarrow_{ESDG.parameterEdge} \vee \rightarrow_{ESDG.parameterOutEdge} \\
& \vee \rightarrow_{ESDG.summaryEdge}\big)^* \; v \\
& \big(\rightarrow_{ESDG.controlDependenceEdge} \vee \rightarrow_{ESDG.dataFlowEdge} && (2)\\
& \vee \rightarrow_{ESDG.declarationEdge} \vee \rightarrow_{ESDG.parameterEdge} \vee \rightarrow_{ESDG.parameterInEdge} \\
& \vee \rightarrow_{ESDG.callEdge} \vee \rightarrow_{ESDG.summaryEdge}\big)^* \; w\Big)\Big\}
\end{aligned}
$$

(1) $SFS$ enthält alle Knoten $v, w$, die

(2) von den Knoten des Slicing-Kriteriums vorwärts über Kontroll-, Datenfluss-, Deklarations-, Parameter- und Parameter-out-Kanten sowie transitive Kanten erreichbar und

(3) von den in (2) erreichten Knoten vorwärts über Kontroll-, Datenfluss-, Deklarations-, Parameter-, Parameter-in- und Aufrufkanten sowie transitive Kanten erreichbar sind.□

### 6.2.3. *computeDynamicSlice*

**Funktion**: Berechnung einer dynamischen Slice

**Eingaben**:
- *ESDG (with marked slicing criterion, marked trace)*

**Ausgaben**:
- *ESDG (with marked slice)*

**zusammengesetzt aus**:
- *2.2.1 computeDynamicBackwardSlice*
- *2.2.2 computeDynamicForwardSlice*

*computeDynamicSlice* enthält je einen Teildienst zur Berechnung dynamischer Rückwärts- und Vorwärtsslices.

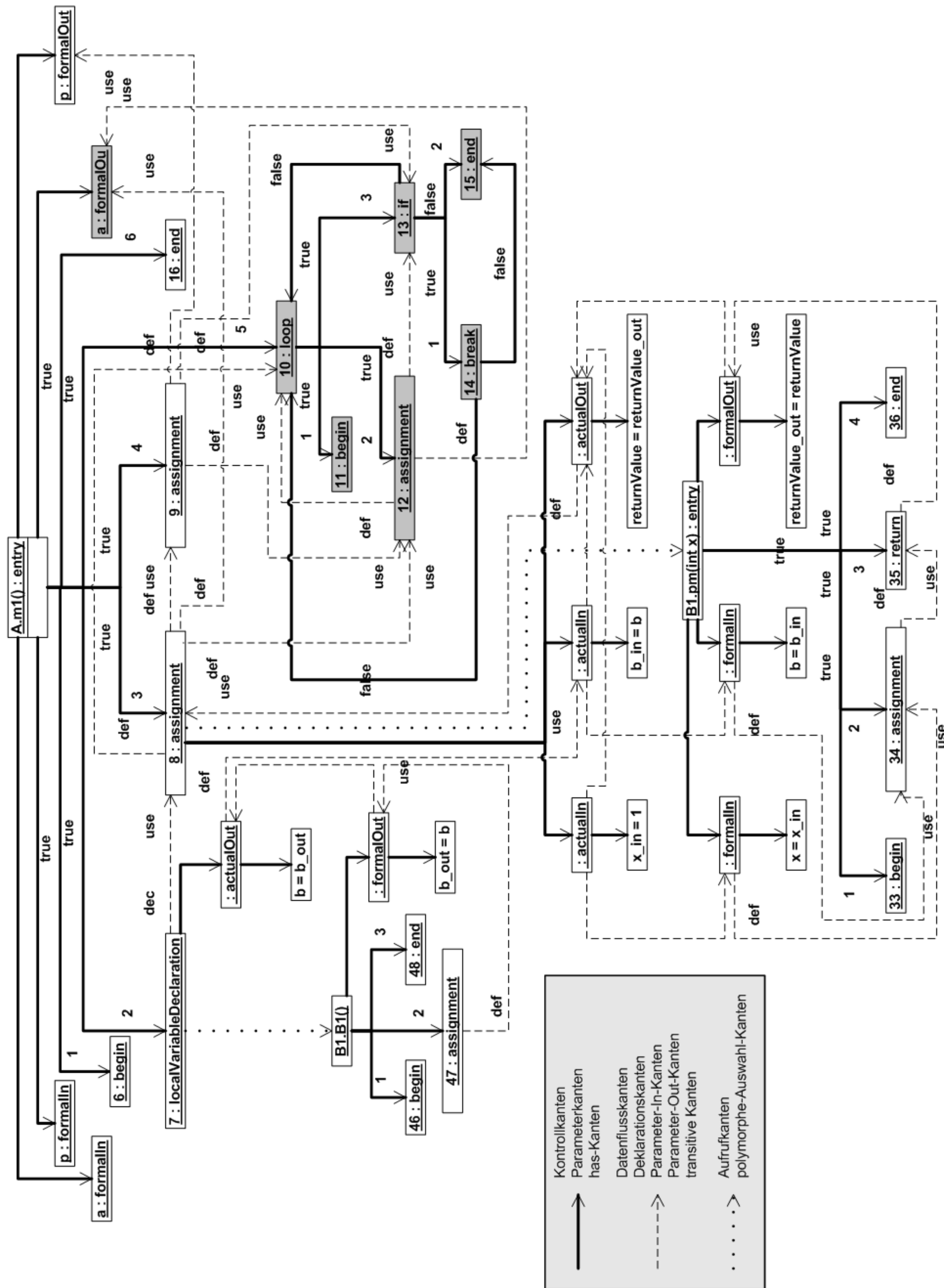


Abbildung 6.2.: Statische Vorwärtsslice für das Slicing-Kriterium $< 12, \{a, p\} >$. Die in der Slice enthaltenen Knoten sind grau unterlegt.

#### 6.2.3.1. *computeDynamicBackwardSlice*

**Funktion**: Berechnung einer dynamischen Rückwärtsslice

**Eingaben**:
- *ESDG (with marked slicing criterion, marked trace)*

**Ausgaben**:
- *ESDG (with marked slice)*

Analog zu der Berechnung einer statischen Rückwärtsslice entspricht die Knotenmenge einer dynamischen Rückwärtsslice der Menge aller über das rückwärtige Traversieren von Kontroll-, Datenfluss-, Deklarations-, Parameter-, Parameter-in- und Aufrufkanten sowie transitiven Kanten erreichbaren Knoten, ausgehend von der das Slicing-Kriterium repräsentierenden Knotenmenge $\{v|v \in V_{ESDGVertex}^{isInSliceCrit=true}\}$. Von den bei der ersten Traversierung erreichten Knoten ausgehend, befindet sich in der Slice außerdem die Menge aller über das rückwärtige Traversieren von Kontroll-, Datenfluss-, Deklarations-, Parameter- und Parameter-out-Kanten sowie transitiven Kanten erreichbaren Knoten. Im Unterschied zu dem Dienst *computeStaticBackwardSlice* (siehe Abschnitt 6.2.2.1) werden jedoch nur diejenigen Kanten berücksichtigt, deren Attribut *isExecuted* den Wert *true* aufweist.

Das Attribut *isInSliceOrChop* der in der Menge enthalten Knoten wird auf *true* gesetzt.

**Definition (dynamische Rückwärtsslice)**
Die Menge $DBS$ der in einer dynamischen Rückwärtsslice enthaltenen Knoten ist gegeben durch:

$$
\begin{aligned}
DBS \;=\; & \Big\{ v, w \Big| && (1)\\
& \bullet_{ESDG.ESDGVertex}^{isInSliceCrit=true} \big( \leftarrow_{ESDG.controlDependenceEdge}^{isExecuted=true} \vee \leftarrow_{ESDG.dataFlowEdge}^{isExecuted=true} && (2)\\
& \vee \leftarrow_{declarationEdge}^{isExecuted=true} \vee \leftarrow_{ESDG.parameterEdge}^{isExecuted=true} \vee \leftarrow_{ESDG.parameterInEdge}^{isExecuted=true} \\
& \vee \leftarrow_{ESDG.callEdge}^{isExecuted=true} \vee \leftarrow_{ESDG.summaryEdge}^{isExecuted=true} \big)^{*} \; v \\
& \big( \leftarrow_{ESDG.controlDependenceEdge}^{isExecuted=true} \vee \leftarrow_{ESDG.dataFlowEdge}^{isExecuted=true} && (3)\\
& \vee \leftarrow_{ESDG.declarationEdge}^{isExecuted=true} \vee \leftarrow_{ESDG.parameterEdge}^{isExecuted=true} \vee \leftarrow_{ESDG.parameterOutEdge}^{isExecuted=true} \\
& \vee \leftarrow_{ESDG.summaryEdge}^{isExecuted=true} \big)^{*} \; w \Big\}
\end{aligned}
$$

(1) $DBS$ enthält alle Knoten $v, w$, die

(2) von den Knoten des Slicing-Kriteriums rückwärts über ausgeführte Kontroll-, Datenfluss-, Deklarations-, Parameter-,Parameter-in- und Aufrufkanten sowie transitiven Kanten erreichbar und

(3) von den in Punkt (2) erreichten Knoten rückwärts über ausgeführte Kontroll-, Datenfluss-, Deklarations-, Parameter- und Parameter-out-Kanten sowie transitiven Kanten erreichbar sind. □

#### 6.2.3.2. *computeDynamicForwardSlice*

**Funktion**: Berechnung einer dynamischen Vorwärtsslice

**Eingaben**:

- *ESDG (with marked slicing criterion, marked trace)*

**Ausgaben**:

- *ESDG (with marked slice)*

Die Berechnungen dieses Dienstes entsprechen, wie auch schon bei *computeDynamicBackwardSlice*, seiner statischen Variante *computeStaticForwardSlice* (siehe Abschnitt 6.2.2.2). Die einzige Abweichung liegt darin, dass bei der Berechnung nur ausgeführte Kanten, d.h. solche mit dem Attributwert *isExecuted = true*, berücksichtigt werden. Das Attribut *isInSliceOrChop* der in der Menge enthalten Knoten wird auf *true* gesetzt.

Das Attribut *isInSlice* der in der Menge enthalten Knoten wird auf *true* gesetzt.

**Definition (dynamische Vorwärtsslice)**
Die Menge $DFS$ der in einer dynamischen Vorwärtsslice enthaltenen Knoten ist gegeben durch:

$$
\begin{aligned}
DFS \;=\; & \Big\{ v, w \Big| \\
& \wedge\ \bullet^{isInSliceCrit=true}_{ESDG.ESDGVertex} \big(\rightarrow^{isExecuted=true}_{ESDG.controlDependenceEdge} \vee \rightarrow^{isExecuted=true}_{ESDG.dataFlowEdge} \qquad (1)\\
& \vee \rightarrow^{isExecuted=true}_{ESDG.declarationEdge} \vee \rightarrow^{isExecuted=true}_{ESDG.parameterEdge} \vee \rightarrow^{isExecuted=true}_{ESDG.parameterOutEdge} \\
& \vee \rightarrow^{isExecuted=true}_{ESDG.summaryEdge}\big)^{*}\ v \\
& \big(\rightarrow^{isExecuted=true}_{ESDG.controlDependenceEdge} \vee \rightarrow^{isExecuted=true}_{ESDG.dataFlowEdge} \qquad (2)\\
& \vee \rightarrow^{isExecuted=true}_{ESDG.declarationEdge} \vee \rightarrow^{isExecuted=true}_{ESDG.parameterEdge} \vee \rightarrow^{isExecuted=true}_{ESDG.parameterInEdge} \\
& \vee \rightarrow^{isExecuted=true}_{ESDG.callEdge} \vee \rightarrow^{isExecuted=true}_{ESDG.summaryEdge}\big)^{*}\ w \Big) \Big\}
\end{aligned}
$$

(1) $DFS$ enthält alle Knoten $v, w$, die

(2) von den Knoten des Slicing-Kriteriums vorwärts über ausgeführte Kontroll-, Datenfluss-, Deklarations-, Parameter- und Parameter-out-Kanten sowie transitive Kanten erreichbar und

(3) von den in (3) erreichten Knoten vorwärts über ausgeführte Kontroll-, Datenfluss-, Deklarations-, Parameter-, Parameter-in- und Aufrufkanten sowie transitive Kanten erreichbar sind. □

## 6.2.4. *computeStaticChop*

**Funktion**: Berechnung eines statischen Chops

**Eingaben**:
- *ESDG (with marked chopping criterion)*

**Ausgaben**:
- *ESDG (with marked chop)*

Die in einem statischen Chop enthaltenen Knoten sind diejenigen, die auf einem Pfad zwischen den beiden Knoten des Chopping-Kriteriums liegen. Einschränkend dabei ist, dass, angenommen der Pfad sei in zwei aufeinanderfolgende Abschnitte unterteilt, im ersten Abschnitt enthaltene Parameter-in- oder Aufrufkanten, über die in eine Methode „hinabgestiegen“ wird, an denselben Aufruf anknüpfen müssen wie die Parameter-out-Kante, über die der Pfad aus der Methode „aufgestiegen“ ist.

Im zweiten Teilpfad muss umgekehrt gelten, dass Parameter-out-Kanten, über die von einer Methode zu deren Aufruf „aufgestiegen“ wird, an denselben Aufruf anknüpfen müssen wie die Parameter-in- oder Aufrufkanten, über die der Pfad in die Methode „hinabgestiegen“ ist (siehe [Rep$^+$95]).

Das Attribut *isInSliceOrChop* der in dem Chop enthalten Knoten wird auf *true* gesetzt.

Abbildung 6.3 enthält das Beispiel eines ESDG aus Abschnitt 5.6 mit dem markierten statischen Chop für das Chopping-Kriterium $< 9, 13 >$.

**Definition (statischer Chop)**
Die Menge $SC$ der in einem statischen Chop enthaltenen Knoten ist gegeben durch:

$$
\begin{aligned}
SC \quad = \quad & \Big\{ v \Big| & (1)\\
& \bullet^{isFirstInChopCrit=true}_{ESDG.ESDGVertex} \rightarrow^{*}_{esdgEdge} v \rightarrow^{*}_{esdgEdge} \bullet^{isSecondInChopCrit=true}_{ESDG.ESDGVertex} & (2)\\
& \wedge\ \exists w \in V_{ESDG.ESDGVertex} & (3)\\
& \Big( \forall ai \in V_{ESDG.actualIn}, ao \in V_{ESDG.actualOut}, c \in V_{ESDG.ESDGVertex} & \\
& ((c \rightarrow_{ESDG.parameterEdge} ai \wedge \bullet^{isFirstInChopCrit=true}_{ESDG.ESDGVertex} \rightarrow^{*}_{esdgEdge} ai & (4)\\
& \rightarrow_{ESDG.parameterInEdge} \bullet_{ESDG.formalIn} \rightarrow^{*}_{esdgEdge} w) \vee \bullet^{isFirstInChopCrit=true}_{ESDG.ESDGVertex} & \\
& \rightarrow^{*}_{esdgEdge} c \rightarrow_{ESDG.callEdge} \bullet_{ESDG.entry} \rightarrow^{*}_{esdgEdge} w & \\
& \Rightarrow c \rightarrow_{ESDG.parameterEdge} ao \wedge \bullet^{isFirstInChopCrit=true}_{ESDG.ESDGVertex} \rightarrow^{*}_{esdgEdge} & (5)\\
& \bullet_{ESDG.formalOut} \rightarrow_{ESDG.parameterOutEdge} ao \rightarrow^{*}_{esdgEdge} w) & \\
& \wedge\ (c \rightarrow_{ESDG.parameterEdge} ao \wedge w \rightarrow^{*}_{esdgEdge} \bullet_{ESDG.formalOut} & (6)\\
& \rightarrow_{ESDG.parameterOutEdge} ao \rightarrow^{*}_{esdgEdge} \bullet^{isSecondInChopCrit=true}_{ESDG.ESDGVertex} &
\end{aligned}
$$

$$\Rightarrow \big(c \rightarrow_{ESDG.parameterEdge} ai \wedge w \rightarrow^{*}_{esdgEdge} ai \rightarrow_{ESDG.parameterInEdge} \bullet_{ESDG.formalIn} \rightarrow^{*}_{esdgEdge} \bullet^{isSecondInChopCrit=true}_{ESDG.ESDGVertex}) \vee w \rightarrow^{*}_{esdgEdge} c \rightarrow_{ESDG.callEdge} \bullet_{ESDG.entry} \rightarrow^{*}_{esdgEdge} \bullet^{isSecondInChopCrit=true}_{ESDG.ESDGVertex})\big) \quad (7)$$

(1) $SC$ enthält alle Knoten $v$, die

(2) auf einem aus ausgeführten Kanten bestehenden Pfad zwischen den beiden Knoten des Chopping-Kriteriums liegen und

(3) es existiert ein auf dem Pfad liegender Knoten $w$, so dass dann,

(4) wenn der Teilpfad von dem ersten Knoten des Chopping-Kriteriums zu $w$ eine Aufrufkante oder Parameter-in-Kante enthält,

(5) er auch eine Parameter-out-Kante enthalten muss, so dass diese Kanten alle mit demselben Knoten $c$ verknüpft sind, bzw. mit den Parameterknoten $ai$ und $ao$, die dann ihrerseits mit $c$ verbunden sind.

(6) Analog zu (4) gilt, dass dann, wenn der Teilpfad von $w$ zu dem zweiten Knoten des Chopping-Kriteriums eine Parameter-out-Kante enthält,

(7) er auch eine Aufrufkante oder Parameter-in-Kante enthalten muss, so dass diese Kanten alle mit demselben Knoten $c$ verknüpft sind, bzw. mit den Parameterknoten $ai$ und $ao$, die dann ihrerseits mit $c$ verbunden sind. □

### 6.2.5. *computeDynamicChop*

**Funktion**: Berechnung eines dynamischen Chops

**Eingaben**:

- *ESDG (with marked chopping criterion, marked trace)*

**Ausgaben**:

- *ESDG (with marked chop)*

Wie bei der Berechnung von Slices unterscheidet sich die Ermittlung von dynamischen Chops von der statischen Variante lediglich dadurch, dass bei der Traversierung der Kanten ausschließlich solche berücksichtigt werden, deren Attribut *isExecuted* auf *true* gesetzt ist. Das Attribut *isInSliceOrChop* der in dem Chop enthalten Knoten wird auf *true* gesetzt.

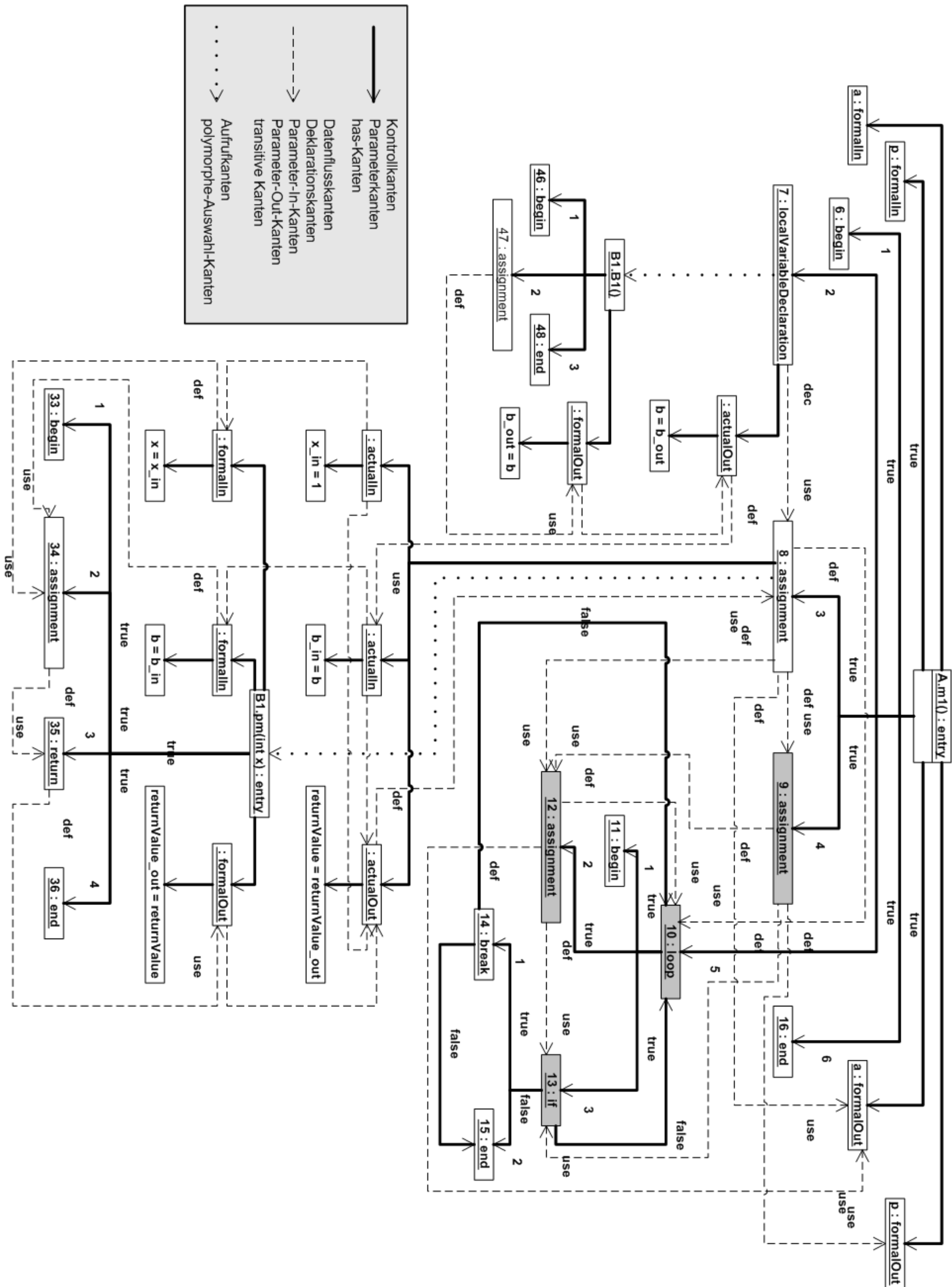


Abbildung 6.3.: Statischer Chop für das Chopping-Kriterium $< 9, 13 >$. Die in dem Chop enthaltenen Knoten sind grau unterlegt.

**Definition (dynamischer Chop)**
Sei

$$\begin{aligned}\rightarrow_{esdgEdge} \quad = \quad & \rightarrow_{ESDG.parameterEdge} \cup \rightarrow_{ESDG.parameterInEdge} \cup \rightarrow_{ESDG.parameterOutEdge} \cup \\ & \rightarrow_{ESDG.callEdge} \cup \rightarrow_{ESDG.controlDependenceEdge} \cup \rightarrow_{ESDG.declarationEdge} \cup \\ & \rightarrow_{ESDG.dataFlowEdge}\end{aligned}$$

die Menge aller ESDG-Kanten (Parameter, Parameter-in-, Parameter-out-, Aufruf-, Kontroll- und Datenflusskanten).

Die Menge $DC$ der in einem statischen Chop enthaltenen Knoten ist gegeben durch:

$$\begin{aligned}
DC \quad = \quad & \Big\{ v \Big| & (1)\\
& v \in V_{ESDG.ESDGVertex} & (2)\\
& \wedge \bullet^{isFirstInChopCrit=true}_{ESDG.ESDGVertex} \rightarrow^{*isExecuted=true}_{esdgEdge} v \rightarrow^{*isExecuted=true}_{esdgEdge} & (3)\\
& \bullet^{isSecondInChopCrit=true}_{ESDG.ESDGVertex} & \\
& \wedge \exists w \in V_{ESDG.ESDGVertex} & (4)\\
& \Big( \forall ai \in V_{ESDG.actualIn}, ao \in V_{ESDG.actualOut}, c \in V_{ESDG.ESDGVertex} & \\
& \quad ((c \rightarrow^{isExecuted=true}_{ESDG.parameterEdge} ai \wedge \bullet^{isFirstInChopCrit=true}_{ESDG.ESDGVertex} \rightarrow^{*isExecuted=true}_{esdgEdge} & (5)\\
& \quad ai \rightarrow^{isExecuted=true}_{ESDG.parameterInEdge} \bullet_{ESDG.formalIn} \rightarrow^{*isExecuted=true}_{esdgEdge} w) & \\
& \quad \vee \bullet^{isFirstInChopCrit=true}_{ESDG.ESDGVertex} \rightarrow^{*isExecuted=true}_{esdgEdge} c \rightarrow^{isExecuted=true}_{ESDG.callEdge} \bullet_{ESDG.entry} & \\
& \quad \rightarrow^{*isExecuted=true}_{esdgEdge} w & \\
& \quad \Rightarrow c \rightarrow^{isExecuted=true}_{ESDG.parameterEdge} ao \wedge \bullet^{isFirstInChopCrit=true}_{ESDG.ESDGVertex} \rightarrow^{*isExecuted=true}_{esdgEdge} & (6)\\
& \quad \bullet_{ESDG.formalOut} \rightarrow^{isExecuted=true}_{ESDG.parameterOutEdge} ao \rightarrow^{*isExecuted=true}_{esdgEdge} w) & \\
& \wedge (c \rightarrow^{isExecuted=true}_{ESDG.parameterEdge} ao \wedge w \rightarrow^{*isExecuted=true}_{esdgEdge} \bullet_{ESDG.formalOut} & (7)\\
& \quad \rightarrow^{isExecuted=true}_{ESDG.parameterOutEdge} ao \rightarrow^{*isExecuted=true}_{esdgEdge} \bullet^{isSecondInChopCrit=true}_{ESDG.ESDGVertex} & \\
& \quad \Rightarrow (c \rightarrow^{isExecuted=true}_{ESDG.parameterEdge} ai \wedge w \rightarrow^{*isExecuted=true}_{esdgEdge} ai & (8)\\
& \quad \rightarrow^{isExecuted=true}_{ESDG.parameterInEdge} \bullet_{ESDG.formalIn} \rightarrow^{*isExecuted=true}_{esdgEdge} & \\
& \quad \bullet^{isSecondInChopCrit=true}_{ESDG.ESDGVertex}) \vee w \rightarrow^{*isExecuted=true}_{esdgEdge} c \rightarrow^{isExecuted=true}_{ESDG.callEdge} & \\
& \quad \bullet_{ESDG.entry} \rightarrow^{*isExecuted=true}_{esdgEdge} \bullet^{isSecondInChopCrit=true}_{ESDG.ESDGVertex})\Big) &
\end{aligned}$$

(1) $SC$ enthält alle Knoten $v$, die

(2) ausgeführt wurden,

(3) auf einem aus ausgeführten Kanten bestehenden Pfad zwischen den beiden Knoten des Chopping-Kriteriums liegen und

(4) es existiert ein auf dem Pfad liegender Knoten $w$, so dass dann,

(5) wenn der Teilpfad von dem ersten Knoten des Chopping-Kriteriums zu $w$ eine Aufrufkante oder Parameter-in-Kante enthält,

(6) er auch eine Parameter-out-Kante enthalten muss, so dass diese Kanten alle mit demselben Knoten $c$ verknüpft sind, bzw. mit den Parameterknoten $ai$ und $ao$, die dann ihrerseits mit $c$ verbunden sind.

(7) Analog zu (4) gilt, dass dann, wenn der Teilpfad von $w$ zu dem zweiten Knoten des Chopping-Kriteriums eine Parameter-out-Kante enthält,

(8) er auch eine Aufrufkante oder Parameter-in-Kante enthalten muss, so dass diese Kanten alle mit demselben Knoten $c$ verknüpft sind, bzw. mit den Parameterknoten $ai$ und $ao$, die dann ihrerseits mit $c$ verbunden sind. □

### 6.2.6. *computeExecutableBackwardSlice*

**Funktion**: Berechnung einer ausführbaren Rückwärtsslice

**Eingaben**:
- *ESDG (with marked slice)*

**Ausgaben**:
- *ESDG (with marked slice)*

Dieser Dienst berechnet ausführbare Rückwärtsslices, indem für die Programmausführung notwendige Knoten in die Slice aufgenommen werden.

Weist ein in der Slice befindlicher, über eine Aufrufkante mit dem *entry*-Knoten einer Methode verbundener Knoten nicht für jeden *formalIn*- oder *formalOut*-Knoten einen entsprechenden *actualIn*- bzw. *actualOut*-Knoten auf, so werden die fehlenden *actualIn*- oder *actualOut*-Knoten der Slice hinzugefügt.

Sobald ein neuer *actualIn*-Knoten in die Slice eingefügt wird, müssen auch diejenigen Knoten in der Slice enthalten sein, welche für die Bestimmung der Werte der von der Zuweisung des *actualIn*-Knotens benutzten Variablen verantwortlich sind. Das bedeutet, dass sämtliche Knoten $w$ markiert werden, die eine von dieser Variablen definieren ohne dass es auf einem Kontrollflusspfad von einem Knoten $w$ zu dem mit dem *actualIn*-Knoten verbundenen aufrufenden Knoten zu einer Redefinition der Variablen kommt. Ausgehend von diesen markierten Knoten als Slicing-Kriterium wird eine neue Slice berechnet und deren Knotenmenge mit derjenigen der nicht-ausführbaren Slice vereinigt. Die Vereinigung der beiden Knotenmengen ergibt die ausführbare Slice.

Das Hinzufügen zusätzlicher *actualOut*-Knoten hat keine weiteren Konsequenzen, da die von diesen definierten Variablen im Programm nicht genutzt werden.

Weiterhin werden *begin*- sowie *end*-Knoten zu der Knotenmenge der Slice addiert, sofern sie über eine Kontrollkante mit einem Knoten der nicht-ausführbaren Slice verbunden sind.

## 6.3. *convertEDSGToCode*

**Funktion**: Konvertierung der im ESDG markierten Slice in Programmcode

**Eingaben**:
- *ESDG (with marked slice or chop)*

**Ausgaben**:
- *sliced program code*

Aus dem erweiterten Systemabhängigkeitsgraphen mit markierter Slice wird durch diesen Dienst der Programmcode wiederhergestellt. Dabei sind nur diejenigen Anweisungen im Code enthalten, deren ESDG-Knoten mit *isInSlice = true* attributiert ist. Aktuelle Eingabeparameter eines Methodenaufrufs werden weggelassen, sofern der entsprechende *actualIn*-Knoten nicht in der Slice enthalten ist. Dasselbe gilt für die Definitionen der formalen Parameter und *formalIn*-Knoten.

# 7. Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde ein Dienstmodell zur Unterstützung des Program Slicing entwickelt. Die dabei definierten Dienste bearbeiten jeweils eine abgegrenzte Teilaufgabe des gesamten Konzepts des Program Slicing. Das Modell, in dem Dienst 0 – *ProgramSlicing* zusammengefasst, lässt sich in drei übergeordnete Dienste gliedern, die sich wiederum aus Teildiensten zusammensetzen. Abbildung 7.1 stellt die Gesamtheit der Dienste samt einer Kurzbeschreibung ihrer Funktionalitäten tabellarisch dar.

Der erste Dienst *preprocessProgram* überführt den Programmcode schrittweise in den *erweiterten Systemabhängigkeitsgraphen* (ESDG), auf welchem die Verfahren des Dienstes *sliceProgram* zur Erstellung sowohl statischer als auch dynamischer Vorwärtsslices, Rückwärtsslices und Chops arbeiten. Auch unterstützt wird die Generierung ausführbarer Rückwärtsslices. Der dritte Dienst *convertESDGToCode* schließlich stellt den geslicten Systemabhängigkeitsgraphen wieder als Programmcode dar.

Der Vorteil dieses Dienstmodells liegt in der Kapselung definierter Segmente der Programmlogik in den Diensten. Dadurch lassen sich die verwendeten Algorithmen weiterentwickeln oder ersetzen, ohne dass sich dies auf andere Dienste auswirkt. Voraussetzung dafür ist, dass die Schnittstellen beibehalten werden, was allerdings je nach Umfang der Änderungen nicht immer realisierbar ist.

Neben der Definition der Dienste besteht die mit dieser Arbeit erbrachte Leistung in der, teils formalen, Definition der zwischen den Diensten fließenden Daten. Auf der Basis der formalen Definition ist eine Implementation des Modells denkbar.

Das Datenmodell basiert auf einer Darstellung der verwendeten Strukturen als Graphen (siehe Abbildung 7.2 für die Graphenschemata). Aus dem Programmcode wird zunächst, von einem Teildienst von *preprocessProgram*, ein *abstrakter Syntaxgraph* erstellt. Dieser bildet die Grundlage für die Berechnung der *erweiterten Kontrollflussgraphen* für die im Programm enthaltenen Methoden, mit dessen Hilfe die *Points-to-Graphen*, wiederum einer für jede Methode, bestimmt werden. Die Points-to-Graphen stellen „zeigt auf“-Beziehungen zwischen im Programm enthaltenen Variablen heraus. Aus dem abstrakten Syntaxgraphen und der Menge der Points-to-Graphen erfolgt die Konstruktion des *erweiterten Aufrufgraphen*, der neben den Aufrufbeziehungen der Methoden auch deren benutzte und definierte Variablen beinhaltet.

| Nr. | Dienst | Funktion |
|---|---|---|
| 0 | ProgramSlicing | Berechnung von statischen/dynamischen Rückwärts-/Vorwärtsslices sowie statischen/dynamischen Chops |
| 1 | preprocessProgram | Überführung des Programmcodes in einen ESDG. |
| 1.1 | computeAbstractSyntaxGraph | Überführung des Programmcodes in einen ASG. |
| 1.2 | computeAugmentedControl-FlowGraph | Berechnung der Menge der ACFGs. |
| 1.3 | computePointsToGraph | Berechnung der Menge der PGs. |
| 1.4 | computeExtendedCallGraph | Berechnung des ECG. |
| 1.4.1 | computeCallGraph | Berechnung des CG. |
| 1.4.2 | computeDefUseInformation | Ergänzung des CG um Informationen über von den Methoden definierte und benutzte Variablen zum ECG. |
| 1.5 | computeExtendedSystem-DependenceGraph | Berechnung des ESDG. |
| 1.5.1 | computeBasicESDG | Erzeugung des Grundgerüsts des ESDG. |
| 1.5.2 | computeInterMethodEdges | Hinzufügen der Aufruf-, Parameter-in- und Parameter-out-Kanten zu dem ESDG. |
| 1.5.3 | computeControlDependence-Edges | Hinzufügen der Kontrollkanten zu dem ESDG. |
| 1.5.4 | computeDataFlowEdges | Hinzufügen der Datenfluss- und Deklarationskanten zu dem ESDG. |
| 1.5.5 | computeSummaryEdges | Hinzufügen der transitiven Kanten zu dem ESDG. |
| 2 | sliceProgram | Durchführung von Slicing- und Choppingverfahren. |
| 2.1 | markESDG | Markierung des Slicing-Kriteriums, des Chopping-Kriteriums und/oder der Trace im ESDG. |
| 2.1.1 | markSlicingCriterion | Markierung des Slicing-Kriteriums im ESDG. |
| 2.1.2 | markChoppingCriterion | Markierung des Chopping-Kriteriums im ESDG. |
| 2.1.3 | markTrace | Markierung der Trace im ESDG. |
| 2.2 | computeStaticSlice | Berechnung einer statischen Slice. |
| 2.2.1 | computeStaticBackwardSlice | Berechnung einer statischen Rückwärtsslice. |
| 2.2.2 | computeStaticForwardSlice | Berechnung einer statischen Vorwärtsslice. |
| 2.3 | computeDynamicSlice | Berechnung einer dynamischen Slice. |
| 2.3.1 | computeDynamicBackward-Slice | Berechnung einer dynamischen Rückwärtsslice. |
| 2.3.2 | computeDynamicForwardSlice | Berechnung einer dynamischen Vorwärtsslice. |
| 2.4 | computeStaticChop | Berechnung eines statischen Chops. |
| 2.5 | computeDynamicChop | Berechnung eines dynamischen Chops. |
| 2.6 | computeExecutable-BackwardSlice | Berechnung einer ausführbaren Rückwärtsslice. |
| 3 | convertESDGToCode | Konvertierung der im ESDG markierten Slice in Programmcode. |

Abbildung 7.1.: Tabellarischer Überblick über die Funktionen der Slicing-Dienste samt Ein- und Ausgaben

| Graph | Beschreibung |
|---|---|
| abstrakter Syntaxgraph (ASG) | Der abstrakte Syntaxgraph stellt die Syntax des gesamten Programmcodes dar. Die Berechnung aller anderen Graphen in dieser Tabelle geschieht auf seiner Basis. |
| erweiterter Kontrollflussgraph (ACFG) | Anhand des erweiterten Kontrollflussgraphen einer Methode lassen sich alle möglichen Abarbeitungsabfolgen der in dieser Methode enthaltenen Anweisungen nachvollziehen. Zusätzlich enthält er Kanten, die von einer Sprunganweisung zu derjenigen Anweisung führen, welche bei Nichtausführung des Sprunges abgearbeitet würde. |
| Points-To-Graph (PG) | Der Points-to-Graph einer Methode stellt eine konservative Abschätzung der „zeigt auf"-Beziehungen zwischen den Variablen dar. |
| (erweiterter) Aufrufgraph (CG/ECG) | Ein Aufrufgraph beschreibt die Aufrufbeziehungen zwischen den Methoden eines Programms. Ergänzend dazu enthält der erweiterte Aufrufgraph Informationen über die von den Methoden definierten und benutzten Variablen. |
| erweiterter Systemabhängigkeitsgraph (ESDG) | Dieser Graph bildet die Grundlage zur Durchführung der Slicing- und Chopping-Algorithmen und lässt sich mit Hilfe des ASG, des ECG sowie den ACFGs und PGs der im Programm enthaltenen Methoden ermitteln. Die Slices und Chops werden direkt in dem Graphen markiert. |

Abbildung 7.2.: Tabellarischer Überblick über die im Modell verwendeten Graphen

Alle oben angeführten Graphen werden zur Erstellung des erweiterten Systemabhängigkeitsgraphen herangezogen. Die Markierung der Slices und Chops im ESDG erfolgt durch spezielle Attribute seiner Knoten.

Ansatzpunkte für eine Weiterentwicklung des Modells sind vor allem im Bereich der zugrundeliegenden Verfahren zu suchen. Das Modell stützt sich auf eine vorhergehende Analyse und Auswahl bereits existierender Slicingtechniken. Bei einigen dieser Techniken hat sich gezeigt, dass sie gravierende Nachteile aufweisen. Man denke hier etwa an die Duplizierung von Methoden im Zusammenhang mit Polymorphie bei Liang und Harrold [Lia+98]. Da diese Arbeit bestehende Verfahren aufgreift und sie in das Modell integriert, jedoch weitestgehend auf deren Weiterentwicklung verzichtet, bzw. deren größte Probleme durch konservative Ansätze beseitigt (siehe die Behandlung der Polymorphie), besteht hier ein Potential zur Verbesserung. Auch ist es fast sicher, dass bei der Vielzahl der vorhandenen Artikel und Konferenzbeiträge zum Thema Program Slicing nicht alle Ansätze und Techniken gesichtet und erfasst wurden. Die folgende Liste fasst einige Anstöße zur Weiterentwicklung zusammen:

- höhere Genauigkeit bei der Darstellung der Polymorphie

  Diese Weiterentwicklung betrifft die Berechnung des ESDG, so dass in jedem Fall der Dienst *computeExtendedSystemDependenceGraph* zu modifizieren ist. Solange das Schema des ESDG nicht angepasst werden muss, können andere Dienste unberührt bleiben.

- höhere Genauigkeit bei dynamischem Slicing

  Die dynamischen Slicingalgorithmen von Agrawal und Horgan [Agr+90, S. 6ff] sowie Zhang et al. [Zha+04] setzen sich vom hier vorgestellten ESDG unterscheidende Graphenstrukturen voraus. Deshalb wären sowohl das Schema des ESDG als auch der ihn berechnende Dienst *computeExtendedSystemDependenceGraph* zu ändern. Die Vorgehensweise von Agrawal und Horgan vervielfacht zwar die Knoten, führt aber keine neuen Kantentypen ein, so dass der Dienst *sliceProgram*, abgesehen von *markSlicingCriterion* und *markChoppingCriterion*, größtenteils unverändert bleiben kann.

- Verwendung fluss- und/oder kontextsensitiver Zeigeranalyseverfahren

  Neben der Verwendung eines anderen Zeigeranalyseverfahrens durch den Dienst *computePointsToGraph* ist auch das Schema der Points-to-Graphen anzupassen, da dieses in vorliegender Form lediglich auf fluss- und kontextinsensitive Verfahren zugeschnitten ist. Folglich müssten die die PGs benutzenden Dienste ebenfalls modifiziert werden.

- Implementation des Modells

Ein weiterer verbesserungswürdiger Aspekt ist der von dem Modell unterstützte Sprachumfang. Beispielsweise wäre hier an eine Erweiterung um interprozedurale Sprünge, Attribute mit beliebiger Sichtbarkeit, Felder, Funktionszeiger, Verkettung von Zeigern oder *switch*-Anweisungen zu denken.

# Anhang A. CD-ROM

Die beiliegende CD-ROM enthält dieses Dokument im PDF-Format.

# Glossar

**ACFG**: **A**ugmented **C**ontrol **F**low **G**raph. Siehe *erweiterter Kontrollflussgraph*.

**abstrakter Syntaxgraph**: Darstellung des Programmcodes nach lexikalischer und syntaktischer Analyse. Wird zur Berechnung des *ACFG*, des *ECG* sowie des *ESDG* verwendet. Siehe Abschnitt 5.2.

**ASG**: **A**bstract **S**yntax **G**raph. Siehe *abstrakter Syntaxgraph*.

**Aufrufkante**: Kante, die im *ESDG* den Aufruf mit dem Entry-Knoten der aufgerufenen Methode verbindet. Die Kante ist auf letzteren gerichtet. Siehe Abschnitt 5.6.2.

**Chop**: Programmausschnitt, welcher nur diejenigen Anweisungen enthält, welche notwendig sind, Auswirkungen einer Anweisung $s$ auf eine Anweisung $t$ zu erhalten.

- **Chop, statisch**: Chop, bei dessen Berechnung der gesamte Programmcode berücksichtigt wird.
- **Chop, dynamisch**: Chop, der sich auf eine bestimmte Programmausführung bezieht. Das heißt, es können nur diejenigen Anweisungen in dem Chop enthalten sein, welche bei der Programmausführung abgearbeitet wurden.

**Datenflusskante**: Kante des *ESDG*, die eine def-use-Beziehung darstellt. Diese besteht zwischen zwei Knoten $s$ und $t$, falls von $s$ eine Variable definiert wird, die anschließend von $t$ genutzt werden kann, ohne dass es zwischen den Ausführungen von $s$ und $t$ zwingend zu einer Redefinition der Variable kommt. Die Kante ist auf $t$ gerichtet. Siehe Abschnitt 5.6.4.

**Deklarationskante**: Kante des *ESDG*, die eine dec-def/use-Beziehung darstellt. Diese besteht zwischen der Deklaration und der ersten Definition oder Benutzung einer Variable. Die Kante ist auf $t$ gerichtet. Siehe Abschnitt 5.6.4.

**ECG**: **E**xtended **C**all **G**raph. Siehe *erweiterter Aufrufgraph*.

**erweiterter Aufrufgraph**: Stellt die Aufrufbeziehungen zwischen den Methoden sowie die von letzteren benutzten und definierten Variablen dar. Wird aus dem *ASG* und dem *PG* ermittelt und dient zur Berechnung des *ESDG*. Siehe Abschnitt 5.5.

**erweiterter Kontrollflussgraph**: Graph, dessen Knoten die Programmanweisungen einer Prozedur oder Methode darstellen, zuzüglich eines *Entry*- und eines *Exit*-Knotens für Prozedureintritt bzw. -austritt. Eine Kontrollflusskante verläuft von Knoten $s$ zu Knoten $t$, falls $t$ unmittelbar nach $s$ ausgeführt werden kann oder falls $t$ die Anweisung ist, welche bei Nichtausführung der Sprunganweisung $s$ ausgeführt würde. Wird aus dem *ASG* berechnet und ist Grundlage zur Erstellung des *PG* und des *ESDG*. Siehe Abschnitt 5.3.

**erweiterter Systemabhängigkeitsgraph**: Graph, der ein gesamtes Programm darstellt und durch Traversierung die Berechnung von *Slices* und *Chops* ermöglicht. Zur Berechnung des ESDG werden der *ASG*, der *ECG* sowie die *ACFGs* und *PGs* aller Prozeduren oder Methoden benötigt. Siehe Abschnitt 5.6.

**ESDG**:**E**xtended **S**ystem **D**ependence **G**raph. Siehe *erweiterter Systemabhängigkeitsgraph*.

**Kontrollkante**: Kante des *ESDG*, die von Knoten $s$ zu Knoten $t$ verläuft, falls $s$ die Ausführung von $t$ beeinflussen kann. Siehe Abschnitt 5.6.3.

**Parameter-in-Kante**: Von einem Actual-in-Knoten zu einem Formal-in-Knoten verlaufende Kante des *ESDG*. Siehe Abschnitt 5.6.2.

**Parameter-out-Kante**: Von einem Formal-out-Knoten zu einem Actual-out-Knoten verlaufende Kante des *ESDG*. Siehe Abschnitt 5.6.2.

**Parameterkante**: Kante des *ESDG*, die einen *Parameterknoten* mit einem Aufruf oder einem Entry-Knoten verbindet. Siehe Abschnitt 5.6.1.

**Parameterknoten**: Knoten, die den Parameterübergabemechanismus im *ESDG* darstellen. Es wird zwischen Formal-in-, Formal-out- sowie Actual-in- und Actual-out-Knoten unterschieden. Siehe Abschnitt 5.6.1.

**PG**: **P**oints-to-**G**raph. Siehe dort.

**Points-to-Graph**: Aus einer Zeigeranalyse hervorgehender Graph, der die „zeigt auf“-Beziehungen einer Prozedur oder Methode darstellt. Siehe Abschnitt 5.4.

**Slice**: Ausschnitt eines Programms, welcher nur diejenigen Programmanweisungen enthält, die eine Wirkung auf ein vorher zu spezifizierendes Slicing-Kriterium besitzen (Rückwärtsslice) bzw. auf die das Kriterium eine Wirkung hat (Vorwärtsslice). Das Kriterium besitzt die Form $< i, X >$, wobei $i$ eine Anweisung und $X$ eine Variablenmenge darstellt.

- **Slice, statisch**: Slice, bei deren Berechnung der gesamte Programmcode berücksichtigt wird.

- **Slice, dynamisch**: Slice, die sich auf eine bestimmte Programmausführung bezieht. Das heißt, es können nur Anweisungen in der Slice enthalten sein, welche bei der Programmausführung abgearbeitet wurden.

**transitive Kante**: Von einem Actual-in-Knoten zu einem Actual-out-Knoten verlaufende Kante des *ESDG*, falls der Wert des Ausgabeparameters von dem des Eingabeparameters abhängt. Siehe Abschnitt 5.6.5.

# Literaturverzeichnis


[Agr+90] AGRAWAL, Hiralal ; HORGAN, Joseph R.: Dynamic Program Slicing. In: *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation* Bd. 25, S. 246–256. http://www.research.telcordia.com/papers/hira/pldi90.pdf.
letzter Abruf: 08.08.2006

[Agr+91] AGRAWAL, Hiralal ; DEMILLO, Richard A. ; SPAFFORD, Eugene H.: Dynamic Slicing in the Presence of Unconstrained Pointers. In: *Proceedings of ACM Fourth Symposium on Testing, Analysis, and Verification – TAV4*, S. 60–73. http://homes.cerias.purdue.edu/~spaf/spyder/TR93P.pdf.
letzter Abruf: 08.08.2006

[Agr94] AGRAWAL, Hiralal: On Slicing Programs with Jump Statements. In: *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, http://www.research.telcordia.com/papers/hira/pldi94.pdf.
letzter Abruf: 08.08.2006

[Aho+86] AHO, Alfred V. ; SETHI, Ravi ; ULLMAN, Jeffrey D.: *Compilers. Principles, Techniques, and Tools*. Reading, Massachussets : Addison-Wesley, 1986

[And94] ANDERSEN, Lars O.: *Program Analysis and Specialization for the C Programming Language*, DIKU, Universität von Kopenhagen, Diss., 1994. http://www.cs.cornell.edu/courses/cs711/2005fa/papers/andersen-thesis94.pdf.
letzter Abruf: 08.08.2006

[And+01] ANDERSON, Paul ; TEITELBAUM, Tim: Software Inspection Using CodeSurfer. In: *Proccedings of the first Workshop on Inspection in Software Engineering – WISE'01*, http://www.grammatech.com/research/papers/AndersonTeitelbaum.pdf.
letzter Abruf: 08.08.2006

[Bal+93] BALL, Thomas ; HORWITZ, Susan: Slicing Programs with Arbitrary Control Flow. In: *Proceedings of the First International Workshop on Automated and*

*Algorithmic Debugging* Bd. 749, Springer-Verlag, S. 206–222. http://www.cs.wisc.edu/wpis/papers/aadebug93.ps.
letzter Abruf: 08.08.2006

[Bes+02] BESZÉDES, Árpád ; FARAGÓ, Csaba ; GERGELY, Tamás ; SZABÓ, Zsolt M. ; GYIMÓTHY, Tibor: Union Slices for Program Maintenance. In: *Proceedings of the International Conference on Software Maintenance, Montreal*, http://www.inf.u-szeged.hu/~beszedes/research/tech28_beszedes_a.pdf.
letzter Abruf: 08.08.2006

[Bin93] BINKLEY, David: Precise Executable Interprocedural Slices. In: *ACM Letters on Programming Languages and Systems* 2 (1993), S. 31–45. http://www.cs.loyola.edu/~binkley/papers/loplas93.ps.
letzter Abruf: 08.08.2006

[Bin+96] BINKLEY, David ; GALLAGHER, Keith B.: Program Slicing. In: *Advances in Computers* 43 (1996), S. 1–50

[Cha+94] CHANG, Juei ; RICHARDSON, Debra J.: Static and Dynamic Specification Slicing. In: *Proceedings of the Fourth Irvine Software Symposium, Irvine*, http://citeseer.ist.psu.edu/rd/0%2C34072%2C1%2C0.25%2CDownload/http://citeseer.ist.psu.edu/cache/papers/cs/2001/http:zSzzSzwww.ics.uci.eduzSzpubzSzarcadiazSzpaperszSzslicing.pdf/chang94static.pdf.
letzter Abruf: 08.08.2006

[Cho+94] CHOI, Jong-Deok ; FERRANTE, Jeanne: Static slicing in the presence of goto statements. In: *ACM Transactions on Programming Languages and Systems* 16 (1994), Nr. 4, S. 1097–1113

[Das00] DAS, Manuvir: Unification-based Pointer Analysis with Directional Assignments, http://research.microsoft.com/manuvir/Papers/pldi00.ps.
letzter Abruf: 08.08.2006

[Ema+94] EMAMI, Maryam ; GHIYA, Rakesh ; HENDREN, Laurie J.: Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers. In: *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, S. 242–256. http://www.cs.cornell.edu/courses/cs711/2005fa/papers/egh-pldi94.pdf.
letzter Abruf: 08.08.2006

[Far+01] FARAGÓ, Csaba ; GERGELY, Tamás: Handling the Unstructured Statements in the Forward Dynamic Slice Algorithm. In: *Proceedings of the 7th Symposium on Programming Languages and Software Tools – SPLST 2001*, S. 16–27. http://www.inf.u-szeged.hu/~gertom/Kutatas/handling01.pdf. letzter Abruf: 08.08.2006

[Far+02] FARAGÓ, Csaba ; GERGELY, Tamás: Handling the Unstructured Statements in the Forward Computed Dynamic Slice Algorithm. In: *Acta Cybernetica* 15 (2002), S. 489–508. http://www.inf.u-szeged.hu/~gertom/Kutatas/handling02.pdf. letzter Abruf: 08.08.2006

[Fer+87] FERRANTE, Jeanne ; OTTENSTEIN, Karl J. ; WARREN, Joe D.: The Program Dependence Graph and Its Use in Optimization. In: *ACM Transactions on Programming Languages and Systems* 9 (1987), S. 319–349. http://www.cs.utexas.edu/users/less/reading/spring00/ferrante.pdf. letzter Abruf: 08.08.2006

[For+97] FORGACS, Istvan ; GYIMÓTHY, Tibor: An Efficient Interprocedural Slicing Method for Large Programs. In: *Proceedings of SEKE'97, Madrid*, S. 279–287. http://historical.ncstrl.org/tr/ps/ercimsztaki/1997-7.ps. letzter Abruf: 08.08.2006

[Gal+91] GALLAGHER, Keith B. ; LYLE, James R.: Using program slicing in software maintenance. In: *IEEE Transactions on Software Engineering* 17 (1991), Nr. 8, S. 751–761

[Gra00] *Dependence Graphs and Program Slicing*. http://www.grammatech.com/research/slicing/slicingWhitePaper.pdf. 2000. letzter Abruf: 08.08.2006

[Gup+95] GUPTA, Rajiv ; SOFFA, Mary L.: Hybrid Slicing: An Approach for Refining Static Slices Using Dynamic Information. In: *Proceedings of the ACM SIGSOFT Third Symposium on the Foundations of Software Engineering*, S. 29–40. http://www.cs.arizona.edu/people/gupta/research/Publications/SE/fose95.ps. letzter Abruf: 08.08.2006

[Har+98] HARMAN, Mark ; DANICIC, Sebastian: A New Algorithm for Slicing Unstructured Programs. In: *Journal of Software Maintenance: Research and Practice* 10 (1998). http://www.dcs.kcl.ac.uk/staff/mark/jsm98.ps. letzter Abruf: 08.08.2006

[Hin+00] HIND, Michael ; PIOLI, Anthony: Which Pointer Analysis Should I Use? In: *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis – ISSTA*, http://www.cse.unsw.edu.au/~cs4133/Papers/hind.issta00.pdf.
letzter Abruf: 08.08.2006

[Hor+90] HORWITZ, Susan ; REPS, Thomas ; BINKLEY, David: Interprocedural Slicing Using Dependence Graphs. In: *ACM Transactions on Programming Languages and Systems* 22 (1990), S. 26–60. http://www.cs.wisc.edu/wpis/papers/toplas90.ps.
letzter Abruf: 08.08.2006

[Hor+92] HORWITZ, Susan ; REPS, Thomas: The Use of Program Dependence Graphs in Software Engineering. In: *Proceedings of the 14th International Conference on Software Engineering, Melbourne*, http://www.cs.wisc.edu/wpis/papers/icse92.ps.
letzter Abruf: 08.08.2006

[Jac+94] JACKSON, Daniel ; ROLLINS, Eugene J.: Chopping: A Generalization of Slicing / Carnegie Mellon University, School of Computer Science. 1994. http://historical.ncstrl.org/tr/ps/cmucs/CMU-CS-94-169.ps. – 21+ S. CS-94-169. – Technischer Bericht,
letzter Abruf: 08.08.2006

[Kor+88] KOREL, Bogdan ; LASKI, Janusz: Dynamic Program Slicing. In: *Information Processing Letters* 29 (1988), S. 155–163

[Lar+96] LARSEN, Loren ; HARROLD, Mary J.: Slicing Object-Oriented Software. In: *Proceedings of the 18th international conference on Software engineering, Berlin*, S. 495–505. http://www.cs.rutgers.edu/~ryder/oosem99/papers/harrold-slicing-icse96.pdf.
letzter Abruf: 08.08.2006

[Lia+98] LIANG, Donglin ; HARROLD, Mary J.: Slicing Objects Using System Dependence Graph. In: *Proceedings of the Interna-tional Conference on Software Maintenance – ICSM'98, Bethesda*, S. 358–367. http://www.cc.gatech.edu/aristotle/Publications/Papers/icsm98-slicing.pdf.
letzter Abruf: 08.08.2006

[Lyl84] LYLE, James R.: *Evaluating Variations of Program Slicing for Debugging*, University of Maryland, Diss., 1984

[Ott+84] OTTENSTEIN, Karl J. ; OTTENSTEIN, Linda M.: The Program Dependence Graph in a Software Development Environment. In: *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, 1984, S. 177–184

[Ran] RANGANATH, Venkatesh P.: *Indus – Java Program Slicer*. http://projects.cis.ksu.edu/docman/view.php/12/71/slicer-ug.pdf.
letzter Abruf: 08.08.2006

[Rep+94] REPS, Thomas ; HORWITZ, Susan ; SAGIV, Mooly ; ROSAY, Genevieve: Speeding up Slicing. In: *Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, S. 11–20. http://www.cs.wisc.edu/wpis/papers/fse94.ps.
letzter Abruf: 08.08.2006

[Rep+95] REPS, Thomas ; ROSAY, Genevieve: Precise Interprocedural Chopping. In: *Proceedings of the 3rd ACM Symposium on the Foundations of Software Engineering, Washington D.C.*, http://www.cs.wisc.edu/wpis/papers/fse95b.ps.
letzter Abruf: 08.08.2006

[Rep97] REPS, Thomas: *Program Analysis via Graph Reachability*. http://www.cs.wisc.edu/wpis/papers/ilps97.large_font.ps. 1997.
letzter Abruf: 08.08.2006

[Sha+97] SHAPIRO, Marc ; HORWITZ, Susan: Fast and Accurate Flow-Insensitive Points-To Analysis. In: *Proceedings of the 24th ACM SIGPLANSIGACT Symposium on Principles of Programming Languages*, S. 1–14. http://www.cs.wisc.edu/wpis/papers/popl97.ps.
letzter Abruf: 08.08.2006

[Sin+99] SINHA, Saurabh ; HARROLD, Mary J. ; ROTHERMEL, Gregg: System-Dependence-Graph-Based Slicing of Programs with Arbitrary Interprocedural Control Flow. In: *Proceedings of the 21st IEEE International Conference on Software Engineering – ICSE'99*, S. 432–441. http://www.cc.gatech.edu/aristotle/Publications/Papers/icse99.pdf.
letzter Abruf: 08.08.2006

[Ste96] STEENSGAARD, Bjarne: Points-to Analysis in Almost Linear Time. In: *Proceedings of 23rd Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, S. 32–41. http://pag.csail.mit.edu/6.883/readings/steensgaard-popl96.ps.
letzter Abruf: 08.08.2006

[Tip95] TIP, Frank: A Survey of Program Slicing Techniques. In: *Journal of Programming Languages* 3 (1995), Nr. 3, S. 121–189. http://www.research.ibm.com/people/t/tip/papers/jpl1995.pdf.
letzter Abruf: 08.08.2006

[Tip+96] TIP, Frank ; CHOI, Jong-Deok ; FIELD, John ; RAMALINGAM, G.: Slicing Class Hierarchies in C++. In: *Proceedings of the 11th conference on Object-Oriented Programming, Systems, Languages and Applications – OOPSLA'96), San Jose*, S. 179–197. http://www.research.ibm.com/people/t/tip/papers/oopsla1996.pdf.
letzter Abruf: 08.08.2006

[Wei79] WEISER, Mark: *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method.* 1979

[Wei82] WEISER, Mark: Programmers Use Slices When Debugging. In: *Communications of the ACM* 25 (1982), S. 446–452

[Wei84] WEISER, Mark: Program Slicing. In: *IEEE Transactions on Software Engineering* 10 (1984), S. 352–357

[Wil97] WILSON, Robert P.: *Efficient Context-Sensitive Pointer Analysis for C Programs*, Stanford University, Diss., 1997. http://suif.stanford.edu/papers/wilson97.ps.
letzter Abruf: 08.08.2006

[Zha98a] ZHAO, Jianjun: Applying Slicing Technique to Software Architectures. In: *Proceedings of the Fourth IEEE International Conference on Engineering of Complex Computer Systems*, S. 87–98. http://www.fit.ac.jp/~zhao/pub/ps/iceccs98.pdf.
letzter Abruf: 08.08.2006

[Zha98b] ZHAO, Jianjun: Dymanic Slicing of Object-Oriented Programs / Information Processing Society of Japan – IPSJ. 1998. http://www.fit.ac.jp/~zhao/pub/ps/se-tr98-119.pdf. – 17–23 S. SE-98-119. – Technischer Bericht,
letzter Abruf: 08.08.2006

[Zha02] ZHAO, Jianjun: Slicing Aspect-Oriented Software. In: *Proceedings of the 10th IEEE International Workshop on Programming Comprehension*, S. 251–260. http://www.fit.ac.jp/~zhao/pub/ps/iwpc2002.pdf.
letzter Abruf: 08.08.2006

[Zha+03a] ZHANG, Xiangyu ; GUPTA, Rajiv ; ZHANG, Youtao: Precise Dynamic Slicing Algorithms. In: *Proceedings of the 5th International Conference on Software Engineering*, S. 319–329. `http://www.cs.arizona.edu/people/gupta/research/Publications/Comp/icse03.pdf`.
letzter Abruf: 08.08.2006

[Zha+03b] ZHAO, Jianjun ; RINARD, Martin: System Dependence Graph Construction for Aspect-Oriented Programs / Laboratory for Computer Science, MIT. 2003. `http://www.fit.ac.jp/~zhao/pub/ps/mit-lcs-tr-891.pdf`. – 251–260 S. MIT-LCS-TR-891. – Technischer Bericht,
letzter Abruf: 08.08.2006

[Zha+04] ZHANG, Xiangyu ; GUPTA, Rajiv: Cost Effective Dynamic Program Slicing. In: *Proceedings of the ACM SIGPLAN Conference on Pro-gramming Language Design and Implementation*, `http://www.cs.arizona.edu/people/gupta/research/Publications/Comp/pldi04.pdf`.
letzter Abruf: 08.08.2006