Institut für Softwaretechnik
Fachbereich 4

Program Slicing

Ein komponentenbasiertes und adaptierbares Referenztool, exemplarisch angepasst für C

Diplomarbeit

zur Erlangung des Grades eines Diplom-Informatikers
im Studiengang Informatik

vorgelegt von:
Elmar Brauch
elmar.brauch@onlinehome.de
Matrikelnummer: 203110017
am 26.03.2008

Betreuer:

Prof. Dr. Jürgen Ebert, Hannes Schwarz, Institut für Softwaretechnik

Koblenz, im März 2008

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden. Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.

. . . . . .

(Ort, Datum) (Unterschrift)

Abstract

Im Rahmen dieser Diplomarbeit wird das Dienstmodell für Program Slicing, welches von Hannes Schwarz in [Sch07] vorgestellt wurde, zu einem komponentenbasierten Referenztool weiterentwickelt und implementiert.

Dieses Referenztool verwendet zum Slicen ein Referenzschema für Programmiersprachen, welches ebenfalls in dieser Arbeit eingeführt wird. Die hier eingesetzten Slicing-Verfahren basieren auf diesem Referenzschema. Somit kann das Referenztool als Grundlage zum Slicen von Quellcode in beliebigen Programmiersprachen genutzt werden, wenn das Referenzschema vorab an die Gegebenheiten dieser Sprachen angepasst wird.

Exemplarisch wird in dieser Diplomarbeit ein Program Slicing Tool für C-Quellcode entwickelt. Dieses Slicing Tool basiert auf dem Referenztool, in dem es die einzelnen Komponenten des Referenztools bei Bedarf spezialisiert oder in der ursprünglichen Form übernimmt.

Damit das Program Slicing Tool als Referenz gelten kann, wird in dieser Arbeit eine einfache, erweiterbare und komponentenbasierte Architektur entwickelt. Diese entsteht durch den Einsatz aktueller Prinzipien der Softwaretechnik.

Danksagung

Zu Beginn meiner Diplomarbeit möchte ich allen Personen danken, die direkt oder indirekt zum Gelingen dieser Arbeit beigetragen haben.

An erster Stelle danke ich meinen beiden Betreuern Prof. Dr. Jürgen Ebert und Hannes Schwarz für ihre intensive Betreuung. In unseren regelmäßigen Treffen haben sie durch konstruktive Kritik und gute Ideen zum Erfolg dieser Arbeit maßgeblich beigetragen. Ein spezieller Dank gilt Hannes, der mir jederzeit alle Fragen beantwortet hat.

Des Weiteren möchte ich mich bei Daniel Bildhauer bedanken, weil er mir des Öfteren in Sachen Linux und GReQL geholfen hat.

Schließlich danke ich meinen Eltern und meiner Freundin für ihre Unterstützung in jeder Hinsicht.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

1.1. Motivation

Program Slicing [Wei79] ist ein Konzept zur Analyse von Programmen. Anhand eines vorher zu spezifizierenden *Slicing-Kriteriums* wird ein möglichst kleiner Programmausschnitt bestimmt, welcher nur Programmanweisungen enthält, die sich entweder auf das Slicing-Kriterium auswirken oder auf die sich das Slicing-Kriterium auswirkt.

Das Konzept des Program Slicing wurde schon in mehreren Werkzeugen umgesetzt. Beispiele hierfür sind das *Indus*[1] Projekt der Kansas State University und der *CodeSurfer*[2] [AT01] der Firma GrammaTech.

Diese Werkzeuge basierten in ihren ursprünglichen Versionen auf einer monolithischen Architektur. Monolithische Architekturen haben im Vergleich zu komponentenbasierten Architekturen einige Nachteile bezüglich der Flexibilität und Wiederverwendbarkeit.

Es gibt viele verschiedene Program Slicing-Verfahren, wie zum Beispiel statisches Slicing [OO84], dynamisches Slicing [KL88] oder Chopping [RR95]. Wenn diese verschiedenen Verfahren in einzelnen Komponenten realisiert wären, so könnte man neue Werkzeuge durch Wiederverwendung dieser Komponenten leichter und schneller produzieren.

1.2. Ziel der Diplomarbeit

Ziel dieser Diplomarbeit ist die Implementation eines Dienstmodells für Program Slicing. Dabei versteht man unter einem *Dienstmodell* eine Sammlung zusammenarbeitender Softwarekomponenten oder Software-Dienste[3], deren Funktionalitäten jeweils abgegrenzte Teil-

[1] `http://indus.projects.cis.ksu.edu/`

[2] `http://www.grammatech.com/products/codesurfer/overview.html`

[3] Die Begriffe Softwarekomponente und Software-Dienst werden in dieser Arbeit synonym verwendet.

aspekte des gesamten Program Slicing-Konzeptes realisieren. Dieses Dienstmodell basiert auf dem Buch von Hannes Schwarz [Sch07].

Die Implementation des *Slicing-Konzeptes* soll so allgemein gehalten werden, dass Program Slicing für möglichst viele Programmiersprachen möglich ist. Slicing[4] für eine konkrete Programmiersprache kann dann durch Spezialisierung eines zu definierenden Referenzschemas für Programmiersprachen geschehen. In dieser Diplomarbeit wird exemplarisch das Slicen von *C-Programmen* umgesetzt.

Die Berechnung einer Slice geschieht auf der Basis von Graphen. Das zu slicende Programm wird in einen *TGraphen* [EF95] überführt, um anschließend die Slice-Berechnung anhand des Slicing-Kriteriums auf einem TGraphen durchzuführen.

Das zu entwickelnde Slicing-Werkzeug wird aus einigen wesentlichen Teilkomponenten bzw. Teildiensten bestehen, die sich selbst auch aus mehreren Teilkomponenten zusammensetzen können.

Es gibt drei Hauptaufgaben, die von unterschiedlich vielen Komponenten realisiert werden. Die erste Hauptaufgabe ist das Erstellen eines erweiterten *Systemabhängigkeitsgraphen* [Sch07] aus dem zu slicenden Programmcode. Die Teildienste zur Realisierung der zweiten Hauptaufgabe berechnen auf dem Systemabhängigkeitsgraphen in Abhängigkeit von Eingaben, wie zum Beispiel dem Slicing-Kriterium, die Slice. Die Slice wird als Markierung im Systemabhängigkeitsgraphen an die letzte Komponente übergeben. Diese markiert im Programmcode die Slice.

Diese Diplomarbeit stellt einen kompletten Software-Entwicklungsprozess dar. Zuerst werden Anforderungen an das zu entwickelnde Slicing-Werkzeug erhoben. Anschließend wird eine komponentenbasierte Software-Architektur entwickelt, die sich an den Datenflussbeschreibungen aus [Sch07] orientiert. Die Implementation des Slicing-Werkzeuges geschieht in Java. Zur Darstellung des Programmcodes wird das Java Graphenlabor (JGraLab) [Kah06] benutzt und zur Berechnung der Slices wird möglichst die Graph Query Language 2 (GReQL2) [Mar06] verwendet. Abschließend werden die einzelnen Dienste sowie das gesamte Werkzeug getestet und anhand von Beispiel-Slices evaluiert.

[4] Bei Verwendung des Begriffs Slicing ist in dieser Diplomarbeit ausschließlich Program Slicing gemeint.

2. Grundlagen

In diesem Kapitel werden Grundlagen zum besseren Verständnis dieser Arbeit dargestellt. Daher werden im Folgenden einige wichtige Begriffe ausführlich erläutert.

2.1. Program Slicing

Das Program Slicing Tool, welches in dieser Diplomarbeit entwickelt wird, bietet eine Plattform für verschiedene Slicing-Verfahren. Konkret wird das *statische Rückwärts- und Vorwärtsslicen* implementiert. Alternative Verfahren, wie dynamisches Slicen, Choppen oder die Berechnung ausführbarer Slices, werden in [KL88], [AH90], [RR95] und [Bin93] vorgestellt.

2.1.1. Statisches Slicen

Program Slicing wurde 1979 in Weisers Dissertation [Wei79] eingeführt. Die deutsche Übersetzung *„ein Programm in Scheiben schneiden“* [Sch07] beschreibt, was Programmierer laut Weisers Studie [Wei82] im Kopf tun, wenn sie ein Programm debuggen. So zerlegt oder schneidet der Programmierer den Quellcode in kleinere Teile, die nur die relevanten Stellen des Codes enthalten. Diese Teile können dann auch als Scheiben des Quellcodes betrachtet werden.

Die Reduzierung des Quellcodes auf die relevanten Stellen geschieht beim Program Slicing in Abhängigkeit von einem *Slicing-Kriterium*. Dieses Kriterium variiert je nach angewendetem Slicing-Verfahren. Im ursprünglichen Verfahren besteht es jedoch aus einer bestimmten Stelle im Quellcode und einer Teilmenge aller Variablen des Quellcodes.

Die hier implementierten statischen Slicing-Verfahren verwenden ein Slicing-Kriterium *SliceCrit* = <*i,X*>, wobei *i* für eine Anweisung im Quellcode steht und *X* eine Variablenmenge

festlegt. Eine Slice ist nun ein Ausschnitt des Quellcodes, welche aus Streichungen von Anweisungen im ursprünglichen Quellcode hervorgeht. Diese Streichungen basieren auf der *Traversierung* des erweiterten Systemabhängigkeitsgraphen (ESDG). Beim Rückwärtsslicen wird ausgehend von der Anweisung *i* rückwärts traversiert; analog dazu wird beim Vorwärtsslicen vorwärts traversiert. Während der Traversierung werden alle Anweisungen gestrichen, die keine Abhängigkeiten zu den Variablen in der Menge *X* besitzen.

Die Berechnung des erweiterten Systemabhängigkeitsgraphen ist in [Sch07] erklärt. Sie basiert auf einigen anderen Graphen, die in Abschnitt 3.4 vorgestellt werden.

Listing 2.1 enthält ein kleines C-Programm, das anhand des Slicing-Kriteriums „<10,{aLocal}>“ mittels statischem Rückwärtsslicen geslicet werden soll. Die entsprechende Slice ist in Listing 2.2 zu sehen. Da sich die Variable „aLocal“ und Anweisung „10“ im Slicing-Kriterium befinden, müssen alle Anweisungen ausgehend von der letzten Zeile „10“ auf Abhängigkeiten zu „aLocal“ analysiert werden. Man stellt fest, dass die Anweisungen in den Zeilen „4“, „7“ und „9“ keine Abhängigkeit zu „aLocal“ besitzen, so dass sie auch nicht in der Slice enthalten sind. Die Anweisung in Zeile „6“ besitzt im Gegensatz zu den Anweisungen in den Zeilen „3“, „5“ und „8“ zwar keine direkte Abhängigkeit zu „aLocal“, ist aber dennoch in der Slice enthalten, da sie die Variable „bLocal“ definiert, welche in Zeile „8“ zur Definition von „aLocal“ benutzt wird.

```
1  int main()
2  {
3    int aLocal, bLocal;
4    int cLocal;
5    aLocal = 1;
6    bLocal = 2;
7    cLocal = 3;
8    aLocal = aLocal + bLocal;
9    cLocal = cLocal + 5;
10 }
```

Listing 2.1: Beispiel: C-Programm

```
1  int main()
2  {
3    int aLocal, bLocal;
4
5    aLocal = 1;
6    bLocal = 2;
7
8    aLocal = aLocal + bLocal;
9
10 }
```

Listing 2.2: Beispiel: Slice des C-Programms aus Listing 2.1 mit Slicing-Kriterium <10,{aLocal}>

2.1.2. Datenfluss der Slicing-Dienste

Dieser Abschnitt fasst das Dienstmodell für das Konzept des Program Slicing aus [Sch07] zusammen. Die Idee des Dienstmodells ist die hierarchische Zerlegung von Diensten in kleinere Teildienste. So wird in [Sch07] der Dienst *ProgramSlicing* in drei Teildienste zerlegt, welche selbst so lange zerlegt werden, bis weitere Zerlegungen keine sinnvollen Teildienste mehr liefern. Die Dienste auf niedrigster Ebene werden als atomar bezeichnet.

Mit Hilfe von Datenflussdiagrammen wird die Zerlegung der Dienste visualisiert, da hier insbesondere die Abhängigkeiten der einzelnen Dienste bezüglich ihrer Ein- und Ausgaben deutlich werden. Die Funktionalität der Dienste bzw. der Komponenten, welche die Dienste umsetzen, wird in den Kapiteln 4.2.1 und 6 genauer erläutert.

Abb. 2.1 zeigt ein Datenflussdiagramm, das die Teildienste des Dienstes *ProgramSlicing* enthält. Im ersten Schritt werden durch den Teildienst *preprocessProgram* aus dem Quellcode ein erweiterter Systemabhängigkeitsgraph und erweiterte Kontrollflussgraphen (ACFGs) erzeugt. Diese Graphen und von außen kommend ein Slicing- oder Chopping-Kriterium sowie eine optionale Trace bilden die Eingabe für den zweiten Teildienst *sliceProgram*. Für dynamische Slicing- oder Chopping-Verfahren ist die Eingabe der Trace zwingend erforderlich, während die Trace in statischen Verfahren nicht berücksichtigt wird. Der zweite Teildienst markiert im ESDG die Slice oder den Chop und übermittelt diese an den letzten Teildienst *convertESDGToCode*, welcher für die Umwandlung des ESDG in ein geslicetes Programm zuständig ist, das als Ausgabe geliefert wird. Die Eingabedaten Quellcode, Trace, Slicing- und Chopping-Kriterium kommen vom Benutzer oder von der Systemumgebung.

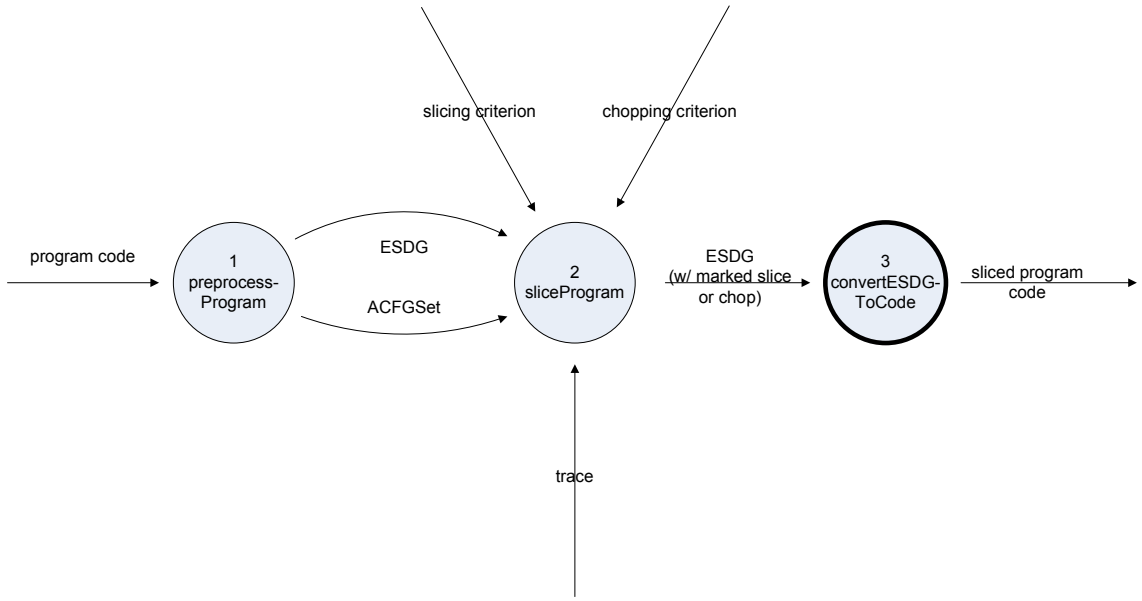

Abbildung 2.1.: Datenflussdiagramm: *ProgramSlicing* [Sch07]

Der Dienst *preprocessProgram* zerlegt sich ebenfalls in mehrere Teildienste, da die Berechnung des ESDG einige Zwischenschritte erfordert. Abb. 2.2 veranschaulicht das Zusammenspiel der einzelnen Teildienste. Zuerst wird aus dem Quellcode durch den atomaren Teildienst *computeAbstractSyntaxGraph* ein abstrakter Syntaxgraph (ASG) berechnet, der als Eingabe für alle anderen Teildienste von *preprocessProgram* dient. Den atomaren Teildiensten *computeAugmentedControlFlowGraphs* und *computePointsToGraphs* genügt der ASG als Eingabe, so dass sie anschließend erweiterte Kontrollflussgraphen bzw. Points-to-Graphen (PGs) produzieren. Der Teildienst *computeExtendedCallGraph* erzeugt einen erweiterten Aufrufgraph (ECG) und zerlegt sich in die Dienste *computeCallGraph* und *computeDefUseInformation*. Da diese beiden Dienste aber eine einfache Sequenz bilden, wird dies in keinem weiteren Datenflussdiagramm dargestellt. Ein wesentlich komplexerer Teildienst ist *computeExtendedSystemDependenceGraph*. Dieser Teildienst benötigt zur Erzeugung des ESDG die Ausgabe aller anderen Teildienste als Eingabe, nämlich den ASG, die ACFGs, die PGs und den ECG.

Abb. 2.3 zeigt detailliert die Zerlegung von *computeExtendedSystemDependenceGraph* in seine atomaren Teildienste. Die Berechnung des ESDG entspricht hierbei einer Sequenz der Dienste *computeBasicESDG*, *computeInterMethodEdges*, *computeControlDependenceEdges*, *computeDataFlowEdges* und *computeSummaryEdges*.

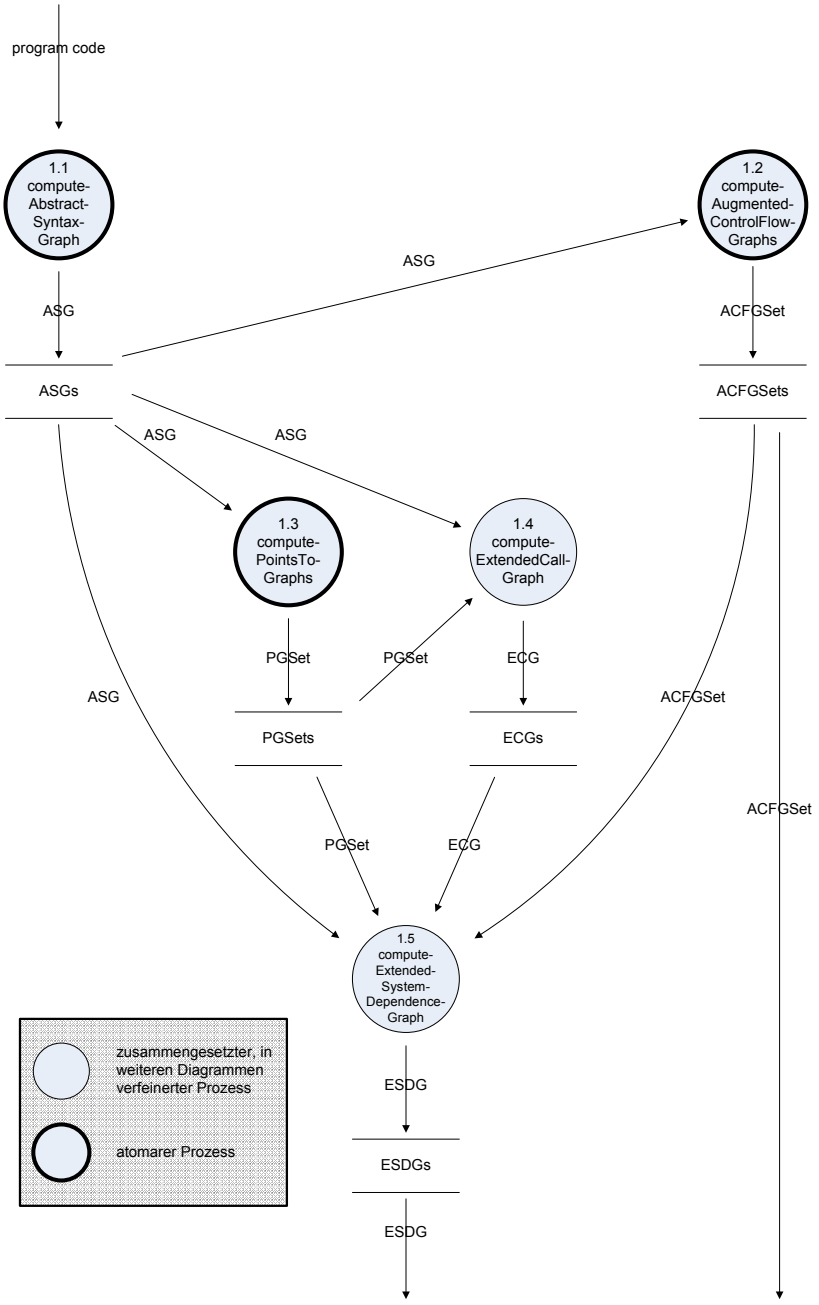

Abbildung 2.2.: Datenflussdiagramm: *preprocessProgram* [Sch07]

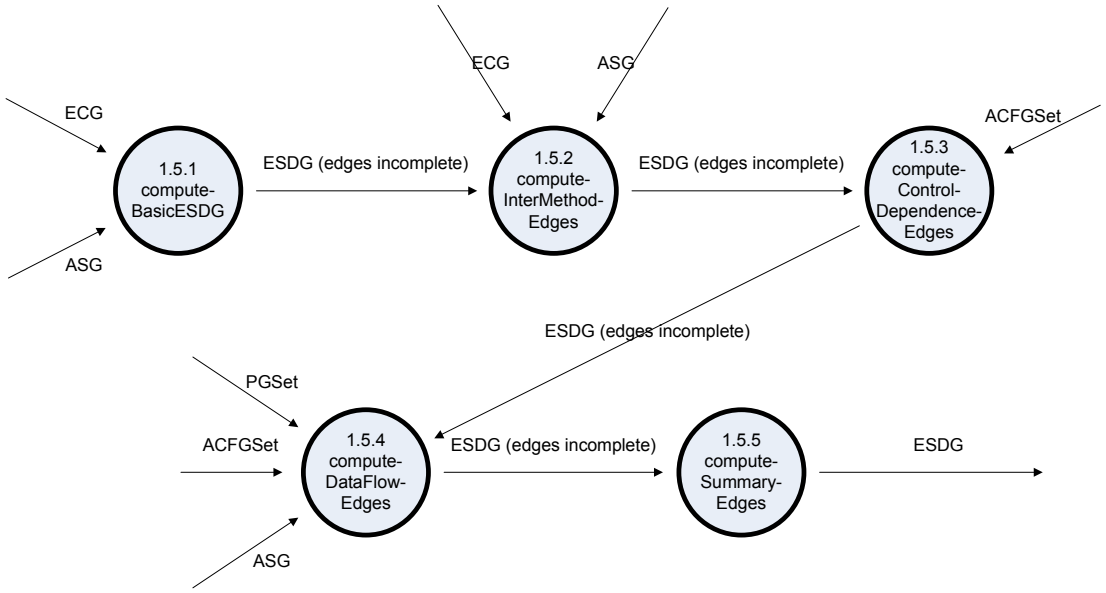

Abbildung 2.3.: Datenflussdiagramm: *computeExtendedSystemDependenceGraph* [Sch07]

Eine Datenflussbeschreibung für den Dienst *sliceProgram* befindet sich in Abb. 2.4. Die Markierung des Slicing- oder Chopping-Kriteriums im ESDG durch den atomaren Teildienst *markESDG* geschieht anhand der Eingabedaten ESDG, ACFG-Menge und optional Trace, sowie dem Slicing- oder Chopping-Kriterium. Im zweiten Schritt wird die Slice oder der Chop unter Zuhilfenahme des markierten ESDG in einem der atomaren Dienste *computeStaticBackwardSlice*, *computeStaticForwardSlice*, *computeDynamicBackwardSlice*, *computeDynamicForwardSlice*, *computeStaticChop* oder *computeDynamicChop* berechnet. Diese Dienste sind Alternativen zueinander und liefern alle als Ausgabe einen ESDG mit markierter Slice oder markiertem Chop. Der Teildienst *computeExecutableBackwardSlice* kann optional auf die Ausgabe der Dienste *computeStaticBackwardSlice* und *computeDynamicBackwardSlice* zur Ermittlung einer ausführbaren Slice angewendet werden.

2.2. TGraph

Dieser Abschnitt beschreibt TGraphen, indem ihre Eigenschaften und ihre zu Grunde liegenden Schemata beschrieben werden.

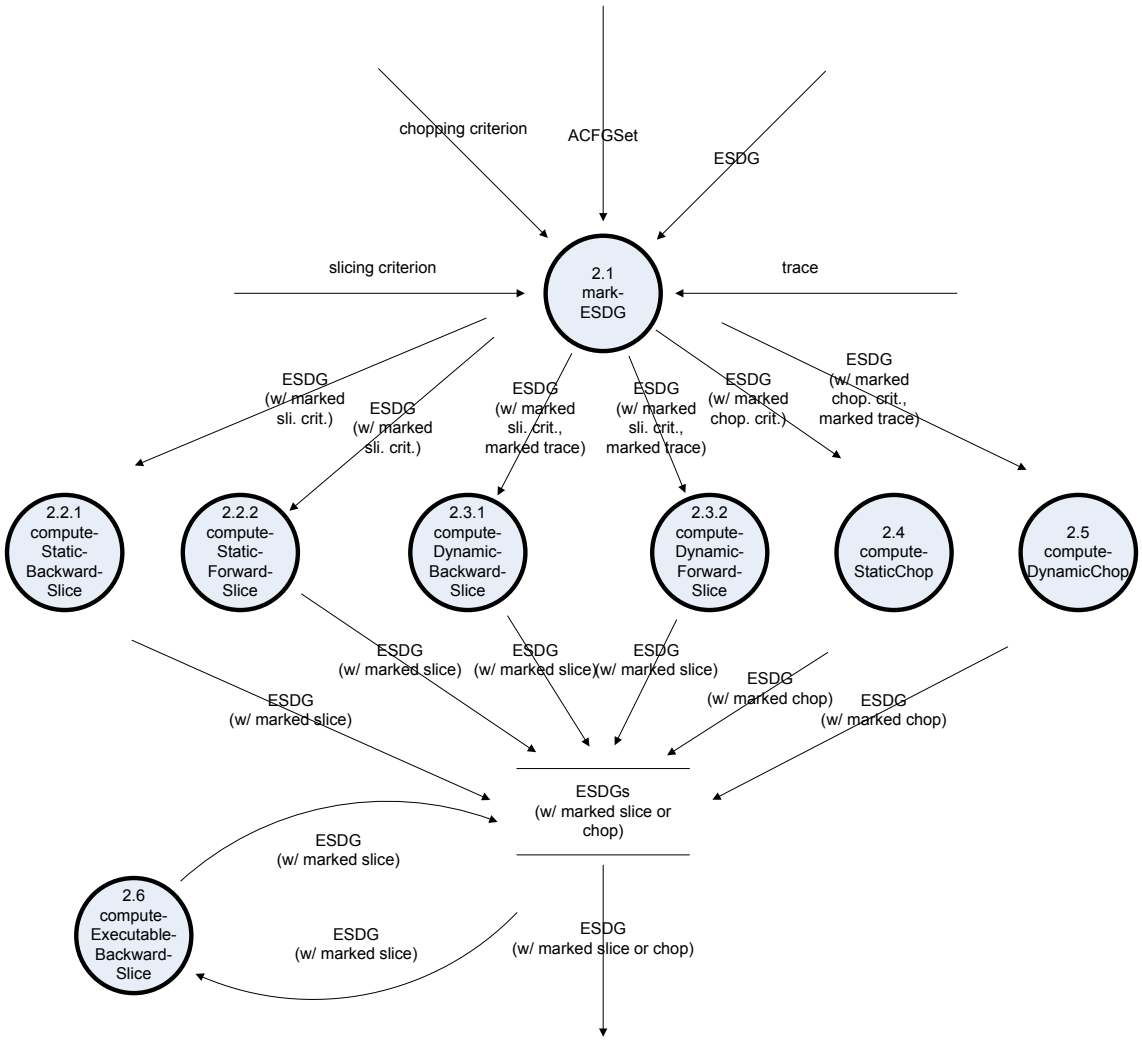

Abbildung 2.4.: Datenflussdiagramm: *slicеProgram*, angepasst aus [Sch07]

2.2.1. TGraph Eigenschaften

Ein Graph besteht aus Knoten und Kanten, wobei eine Kante zwei Knoten verbindet. Weiterführendes zu Graphen im Allgemeinen findet der Leser in [Cha85].

TGraphen sind eine spezielle Art von Graphen. Sie werden durch die folgenden Eigenschaften definiert:

- typisiert: Der TGraph selbst und all seine Knoten und Kanten besitzen jeweils einen festgelegten Typ. Knoten, Kanten oder TGraphen, welche den gleichen Typ besitzen, werden in Knoten-, Kanten- bzw. Graphklassen zusammengefasst.
- attributiert: Alle Elemente des Graphen dürfen Attribut-Werte-Paare besitzen.
- gerichtet: Die Kanten in TGraphen sind immer gerichtet. Allerdings können sie ohne Änderung der Darstellung auch als ungerichtete Graphen betrachtet werden, so dass zum Beispiel gleichzeitig eine gerichtete und eine ungerichtete Graphen-Traversierung durchgeführt werden kann.
- angeordnet: Die Graphelemente innerhalb eines Graphen befinden sich prinzipiell in Anordnungen untereinander. So gibt es eine Anordnung der Knoten, eine Anordnung der Kanten und für jeden Knoten eine Anordnung der ein- und ausgehenden Kanten.

Durch ihre charakteristischen Eigenschaften eignen sich TGraphen zum Beispiel zur Beschreibung von *abstrakten Syntaxgraphen* (siehe dazu Abschnitt 3.4.1). Abstrakte Syntaxgraphen können Parse-Bäume abstrakt beschreiben. Der Parse-Baum ist der erste Schritt auf dem Weg vom ursprünglichen Programmcode zum gesliceten Programmausschnitt. Er entsteht durch Anwendung der Compilerbautechniken [ASU86] lexikalische und syntaktische Analyse.

```
1 ...
2 {
3       if (a > 0)
4             a = 0;
5       b = a;
6 }
7 ...
```

Listing 2.3: Ausschnitt eines C-Programms

Abb. 2.5 zeigt einen Ausschnitt eines Parse-Baums, wie er beim Parsen des C-Programmausschnitts in Listing 2.3 entsteht. Dieser Parse-Baum kann in abstrakter Form auch als TGraph dargestellt werden, siehe dazu Abb. 2.6. Sowohl der Parse-Baum als auch der TGraph entsprechen dem in Kapitel 3.3 vorgestellten C-Schema.

Anhand von Abb. 2.6 erkennt man sehr gut den Nutzen der einzelnen Eigenschaften von TGraph:

- Aufgrund der Typisierung kann man im Beispiel-TGraph aus Abb. 2.6 erkennen, dass es hier jeweils einen Knoten vom Typ `CompoundStatement`, `SelectionStatement` und `OperatorExpression` und zwei Knoten vom Typ `ExpressionStatement` gibt. Zusätzlich erkennt man die Typen der Kanten: `IsExprIn` und `IsStmtIn`.

- Durch die Attributierung wird im Beispiel-TGraph festgelegt, dass Knoten *v2* eine if-Anweisung repräsentiert. Hierfür sind im Parse-Baum zusätzliche Blätter nötig.

- Da die Kanten innerhalb des TGraphen gerichtet sein können, ist es möglich, die Baumstruktur des Parse-Baums im TGraphen wiederzugeben. Da zum Beispiel Knoten *v1* nur eingehende Kanten besitzt, kann man ihn als Wurzel identifizieren.

- Die Anordnung der einzelnen Elemente ist beim Parsen von zentraler Bedeutung, damit man die Abarbeitungsreihenfolge der einzelnen Anweisungen erkennen kann. Im Parse-Baum ist die Anordnung nur grafisch dargestellt, Elemente auf gleicher Ebene werden von links nach rechts abgearbeitet. Innerhalb des TGraphen sind sowohl die Knoten als auch die Kanten angeordnet. In Abb. 2.6 ist dies für die Knoten durch Bezeichnungen wie *v3* (dritter Knoten) dargestellt. Mit den Kanten wird analog verfahren. Zusätzlich besitzt jeder Knoten eine Anordnung seiner ein- und ausgehenden Kanten. Im Beispiel-TGraph ist dies durch Nummern in geschweiften Klammern dargestellt. Auf diese Weise erkennt man, dass zum Beispiel innerhalb der if-Anweisung *v3* vor *v4* abgearbeitet werden muss, weil die Nummer der in *v2* eingehenden Kante *e2* kleiner[1] als die von *e3* ist.

Eine formale Definition von TGraphen kann in [EF95] gefunden werden. Weiterführende Erklärungen zu TGraphen besonders im Bezug zu JGraLab befinden sich in [Kah06].

[1] Hier wird eine aufsteigende Ordnung der Zahlen angenommen.

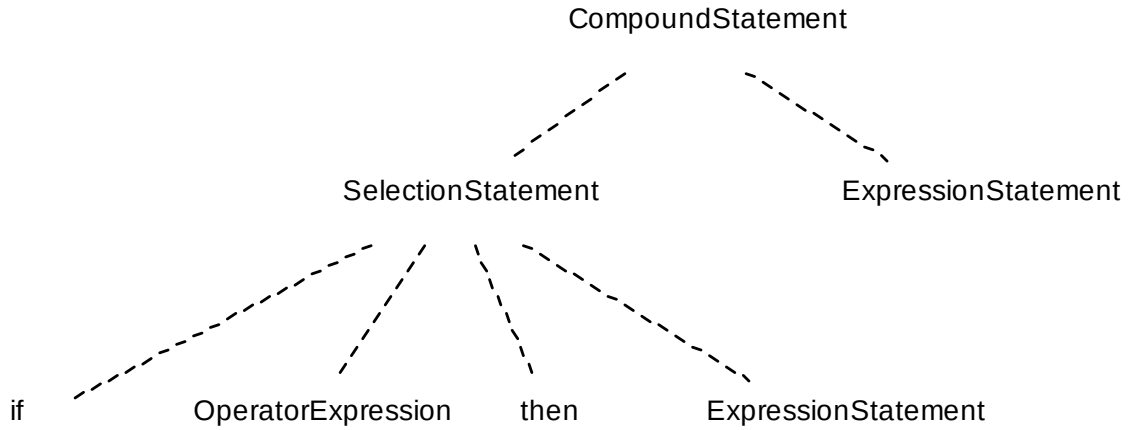

Abbildung 2.5.: Parse-Baumausschnitt eines C-Programms

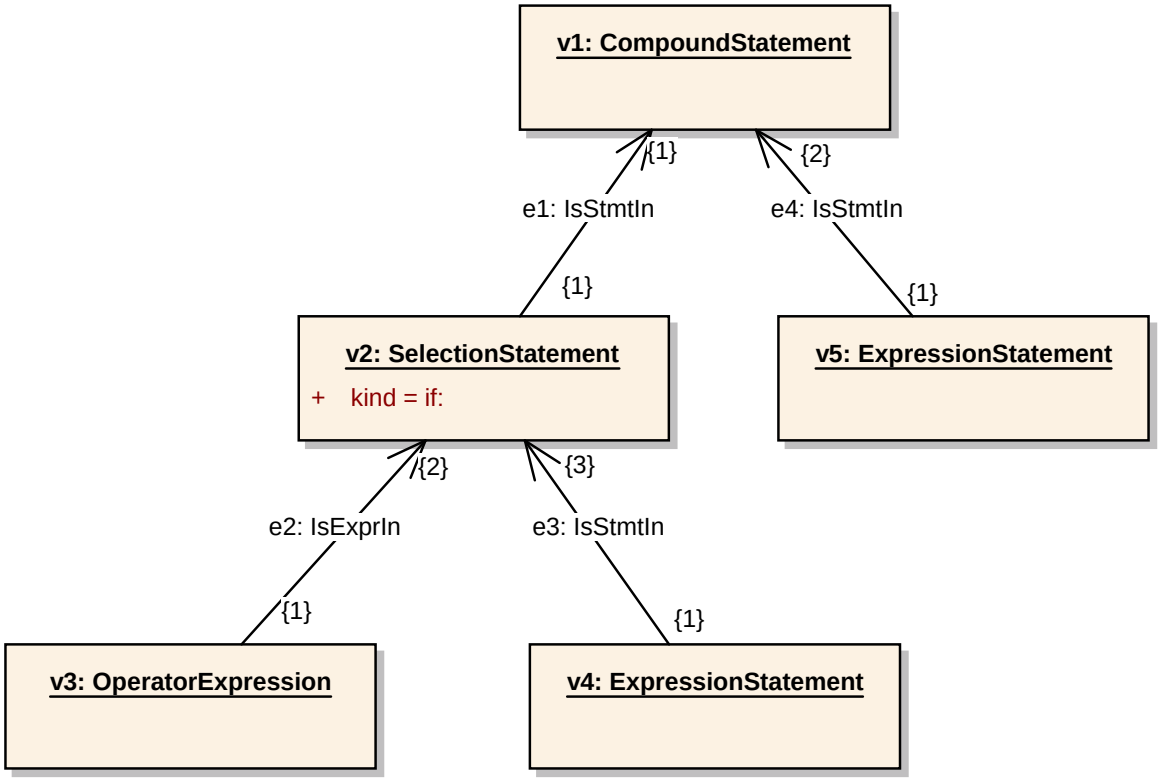

Abbildung 2.6.: Darstellung des Parse-Baums aus Abb. 2.5 als TGraph

2.2.2. Schema

Der Begriff Schema wird in der Informatik häufig im Zusammenhang mit Datenbanken und deren Datenmodellen [KE04] verwendet. In diesem Zusammenhang definiert ein *Datenmodell* generische Strukturen und Operationen, welche zur Modellierung einer konkreten Anwendung genutzt werden. Es bietet dem Benutzer somit die Möglichkeit zur Beschreibung von Datenobjekten und zur Bestimmung von anwendbaren Operatoren und deren Wirkung.

Ein *Schema* entspricht einem Datenmodell, das auf einen *„speziellen Einsatzfall“* [Bal00] angewendet wird. Im Kontext dieser Diplomarbeit wird ein Schema genutzt, um die möglichen Strukturen von TGraphen festzulegen. So wird zum Beispiel durch ein Schema festgelegt, dass ein konkreter TGraph zur abstrakten Beschreibung eines Parse-Baums nur ein Wurzelelement haben kann.

Konkrete Schemata zur Beschreibung von Programmiersprachen werden in Kapitel 3 vorgestellt.

Graphklasse Im Kontext von JGraLab wird der Schema-Begriff überladen. So versteht man im Kontext von JGraLab unter einem Schema eine Menge von Graphklassen, welche wiederum Mengen von Knoten- und Kantenklassen enthalten. Dabei können zwischen den einzelnen Klassen Beziehungen und Vererbungshierarchien bestehen.

In dieser Diplomarbeit werden *Graphklassen* verwendet, um die Struktur von Quellcode in einem TGraph darzustellen. Dabei gibt es für jede Programmiersprache eine eigene Graphklasse, so dass beispielsweise die Struktur eines C-Programms mit Hilfe der C-Graphklasse dargestellt wird.

2.3. Referenzschema

Ein *Referenzschema* ist ein Schema, das als Vorbild, Empfehlung oder Bezug für speziellere Schemata dient. Die folgende Definition aus [Win00] legt den Begriff Referenzschema genauer fest.

Definition: Referenzschema [Win00] *Ein Referenzschema[2] ist ein ausgewiesenes Schema, das charakteristische Eigenschaften einer Klasse gleichartiger Systeme beschreibt und als Bezugspunkt für spezielle Schemata dient.*

Aus einer Menge von Schemata auf gleicher Abstraktionsebene kann ein Schema ausgewählt werden und als Referenzschema bezeichnet werden. Dieses ausgewählte Schema sollte im Vergleich zu den anderen Schemata auf gleicher Abstraktionsebene möglichst allgemein sein, um die gemeinsamen, problembezogenen Aspekte der Schemata durch Abstraktion möglichst einfach darzustellen. In diesem Zusammenhang kann das Referenzschema durchaus einen konkreten Fall beschreiben, da es sich auf der gleichen Ebene wie die anderen Schemata befindet. Jedoch sollte es möglichst in der Menge der anderen Schemata keines geben, das eine größere Menge als das ausgewählte Referenzschema an Gemeinsamkeiten zu den übrigen Schemata besitzt.

Ein Referenzschema kann als Basis zur Entwicklung speziellerer Schemata dienen, indem es Entwicklern einen *„vorgefertigte[n] Lösungsrahmen“* [Seu97] bereitstellt. Somit ist das Entwickeln eines speziellen Schemas auf der Basis eines Referenzschemas meist schneller und führt zu höherer Qualität als die vergleichbare Neuentwicklung des Schemas ohne Vorbild. Daher benötigt das Referenzschema eine hohe Qualität, damit es seiner Rolle als Vorbild oder Empfehlung gerecht werden kann. Qualitätskriterien zur Bewertung eines Referenzschemas sind zum Beispiel Korrektheit, Vollständigkeit, Allgemeingültigkeit, Erweiterbarkeit und Anwendbarkeit. Weiterführendes ist in [Win00] zu finden.

In der Informatik gibt es bereits eine Vielzahl von Referenzschemata. So kann zum Beispiel UML (Unified Modeling Language) [OMG05] als ein Referenzschema betrachtet werden, da sich UML [BRJ06] gegenüber einer Vielzahl anderer Entwurfsbeschreibungssprachen, wie zum Beispiel OMT [Rum96], OML [FHSG98] oder Syntropy [CD94], als Standard durchgesetzt hat. Ein Beispiel für ein spezielleres Schema, welches das UML-Schema als Basis bzw. Referenzmodell verwendet, ist das Schema der Modellierungssprache grUML [BRSS07].

[2] In [Win00] wurde der Begriff „Referenzmodell“ anstelle von „Referenzschema“ definiert. Nach Absprache mit dem Autor darf im Zitat „Referenzschema“ bzw. „Schema“ anstelle von „Referenzmodell“ bzw. „Modell“ verwendet werden.

3. Entwicklung eines Referenzschemas für Programmiersprachen

Dieses Kapitel beschreibt die Entwicklung des Referenzschemas und die Abbildung zwischen diesem und dem C-Schema, welche die Voraussetzung zum Slicen von C-Programmen mit dem Program Slicing Referenztool schafft.

3.1. Entwicklung des Referenzschemas

Jede Programmiersprache besitzt eine *abstrakte Syntax*, welche als Schema dargestellt werden kann. Durch dieses Schema wird der Aufbau von syntaktisch korrekten, abstrakten Syntaxgraphen in der entsprechenden Sprache festgelegt. Die Schemata von verschiedenen Programmiersprachen haben sowohl Unterschiede als auch Gemeinsamkeiten. Im Zuge dieser Diplomarbeit wird an einem Referenzschema gearbeitet, welches die Gemeinsamkeiten von möglichst vielen Sprachen beschreibt.

Als Grundlage für dieses *Referenzschema* dient das Schema der Beispiel-Sprache *EL* [Sch07]. EL (engl. Exemplary Language) ist eine relativ kompakte Sprache zur imperativen Programmierung, welche die Konzepte Objektorientierung, Prozeduren, Sprunganweisungen und Zeiger umsetzt. Abb. 3.1 zeigt das EL-Schema zur Beschreibung von abstrakten Syntaxgraphen (ASG).

[KG98] führt einen *generalisierten abstrakten Syntaxbaum* (GAST) ein, um Programmcode von verschiedenen Programmiersprachen, wie zum Beispiel C und Ada, zu repräsentieren. Die Überführung von Programmcode in einen GAST ermöglicht eine Datenflussanalyse, sowie die Berechnung von verschiedenen Sichten auf den Programmcode. Dieser Ansatz wird hier jedoch nicht weiter verfolgt, da es in [KG98] weder eine Grammatik noch ein Schema zur genauen Definition des GASTs gibt.

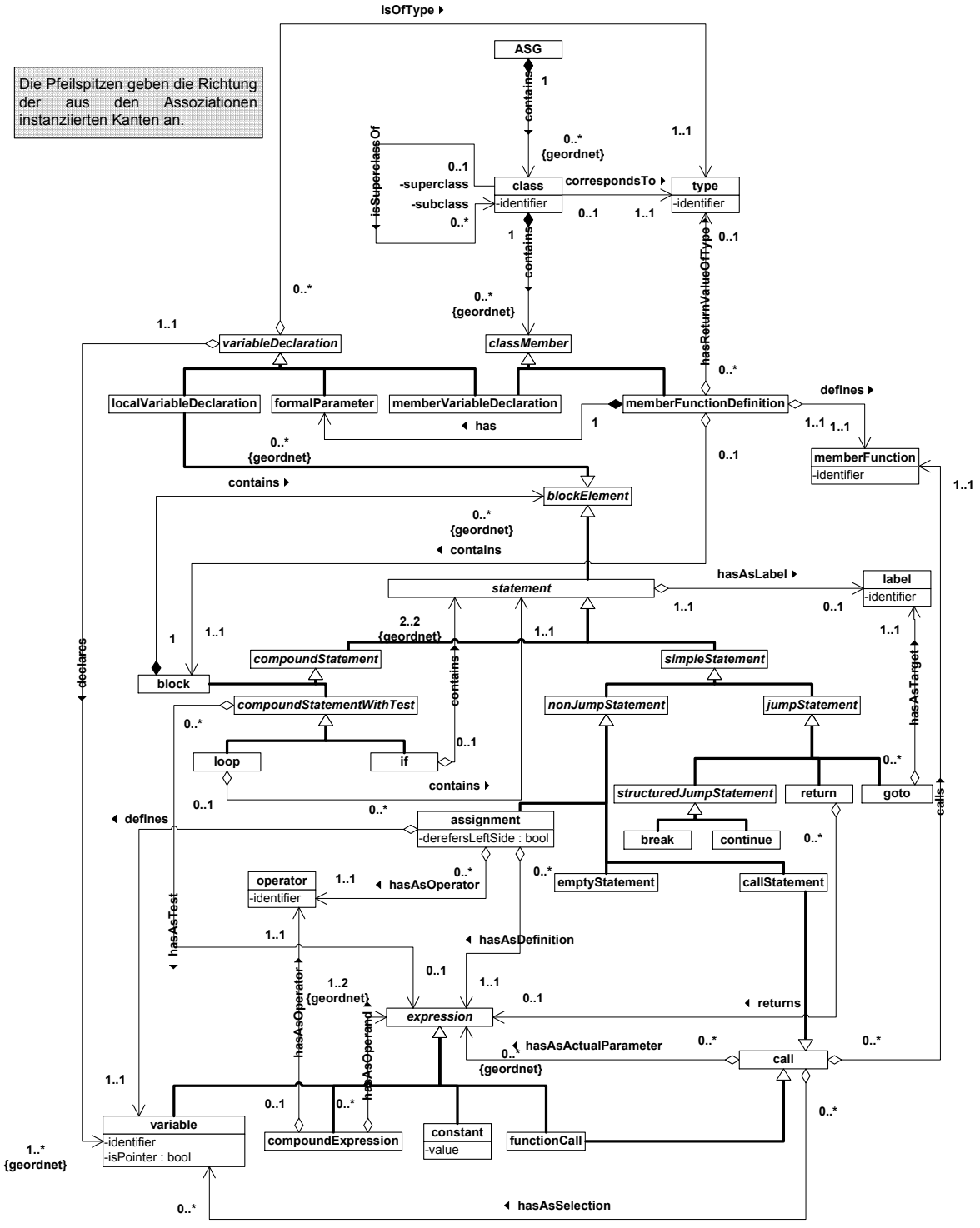

Abbildung 3.1.: EL-Schema zur Beschreibung von ASG [Sch07]

Um ein Program Slicing Tool, das im Folgenden auch als *Reference Program Slicing Tool* (RePST) bezeichnet wird, nach dem beschriebenen Dienstmodell in [Sch07] zu entwickeln, benötigt man je nach gewünschter Programmiersprache eine Abbildung zwischen dem Referenzschema und dem Schema der konkreten Programmiersprache. Mit Hilfe dieser Abbildung kann dann ein konkretes Program Slicing Tool mit möglichst geringem Aufwand entwickelt werden, da es eine Abwandlung des abstrakten, auf dem Referenzschema arbeitenden Program Slicing Tools ist.

3.2. Erweiterung des EL-Schemas zum Referenzschema

Das Schema von EL dient als Grundlage für das Referenzschema. Im Hinblick auf das Program Slicing von konkreten Programmiersprachen, wie zum Beispiel C, sind einige Erweiterungen oder Änderungen am EL-Schema nötig. Das Ergebnis dieser Änderungen wird dann als Referenzschema für Program Slicing oder *Reference Program Slicing Schema* (RePSS) bezeichnet. In diesem Abschnitt werden diese Veränderungen aufgezählt.

1. Nach den Code-Konventionen in [Sun99] beginnen Klassennamen stets mit einem Großbuchstaben. Dies wurde im Referenzschema entsprechend angepasst. Im Quellcode von RePST besitzen alle Knoten des Referenzschemas den Präfix „EL“. Dies wird in der schriftlichen Ausarbeitung jedoch nicht weiter berücksichtigt.

2. Aufgrund des Analogieprinzips enden im Referenzschema alle Klassen, die von `Statement` erben, mit dem Postfix „Statement“.

3. Die Bezeichner der Klassen `if` und `loop` innerhalb des EL-Schemas klingen zu sehr nach einer konkreten Programmiersprache. So ist zum Beispiel „if“ in Java oder C ein Schlüsselwort. Daher wird die EL-Schema-Klasse `if` in `SelectionStatement` und `loop` in `IterationStatement` umbenannt.

4. Da EL eine objektorientierte Sprache ist, können durch das EL-Schema keine prozeduralen Programmiersprachen beschrieben werden. Deshalb besitzt das Referenzschema die abstrakte Oberklasse `Unit`, von dieser erben die Klassen `Class` und `SourceUnit`. So-

mit gibt es im Referenzschema für objektorientierte Sprachen die Klasse `Class` und für prozedurale `SourceUnit`.

5. Durch das Einfügen der Oberklasse `Unit` wurden folgende Klassen umbenannt:

- Die Klasse `classMember` heißt nun `UnitMember`.
- Die Klasse `memberVariableDeclaration` heißt nun `UnitVariable-Declaration`.
- Die Klasse `memberFunctionDefinition` heißt nun `UnitFunction-Definition`.
- Die Klasse `memberFunction` heißt nun `UnitFunction`.

6. Die Multiplizität zwischen `SelectionStatement` und `Statement` wurde so angepasst, dass im Referenzschema eine `SelectionStatement`-Instanz mit „else“ zwei `Statement`-Instanzen besitzt, während eine `SelectionStatement`-Instanz ohne „else“ nur eine `Statement`-Instanz besitzt. Im ursprünglichen EL-Schema muss eine `if`-Instanz stets zwei `statement`-Instanzen besitzen, wobei die zweite eine `empty-Statement`-Instanz sein muss, falls das `if`-Statement keinen „else“-Zweig besitzt.

7. Zur besseren Unterscheidung zwischen Programmiersprachen mit und ohne Zeiger-Konzept, gibt es im Referenzschema die `Variable`-Spezialisierung `Pointer`. Die Klasse `Pointer` ersetzt das EL-Schema-Attribut `-isPointer`.

Abb. 3.2 zeigt das fertige Referenzschema bzw die Änderungen an EL.

3.3. Abbildung des C-Schemas auf das Referenzschema

Da in dieser Diplomarbeit ein Program Slicing Tool für die Programmiersprache C entwickelt wird, benötigt man eine Abbildung vom Schema der Programmiersprache C auf RePSS. Die Abb. 3.3 und 3.4 zeigen das C-Schema [Rie01a]. Dieses C-Schema orientiert sich an der ANSI C-Spezifikation [BCM+98]. Im Folgenden werden die Gemeinsamkeiten und Unterschiede zwischen dem C-Schema und dem Referenzschema gezeigt.

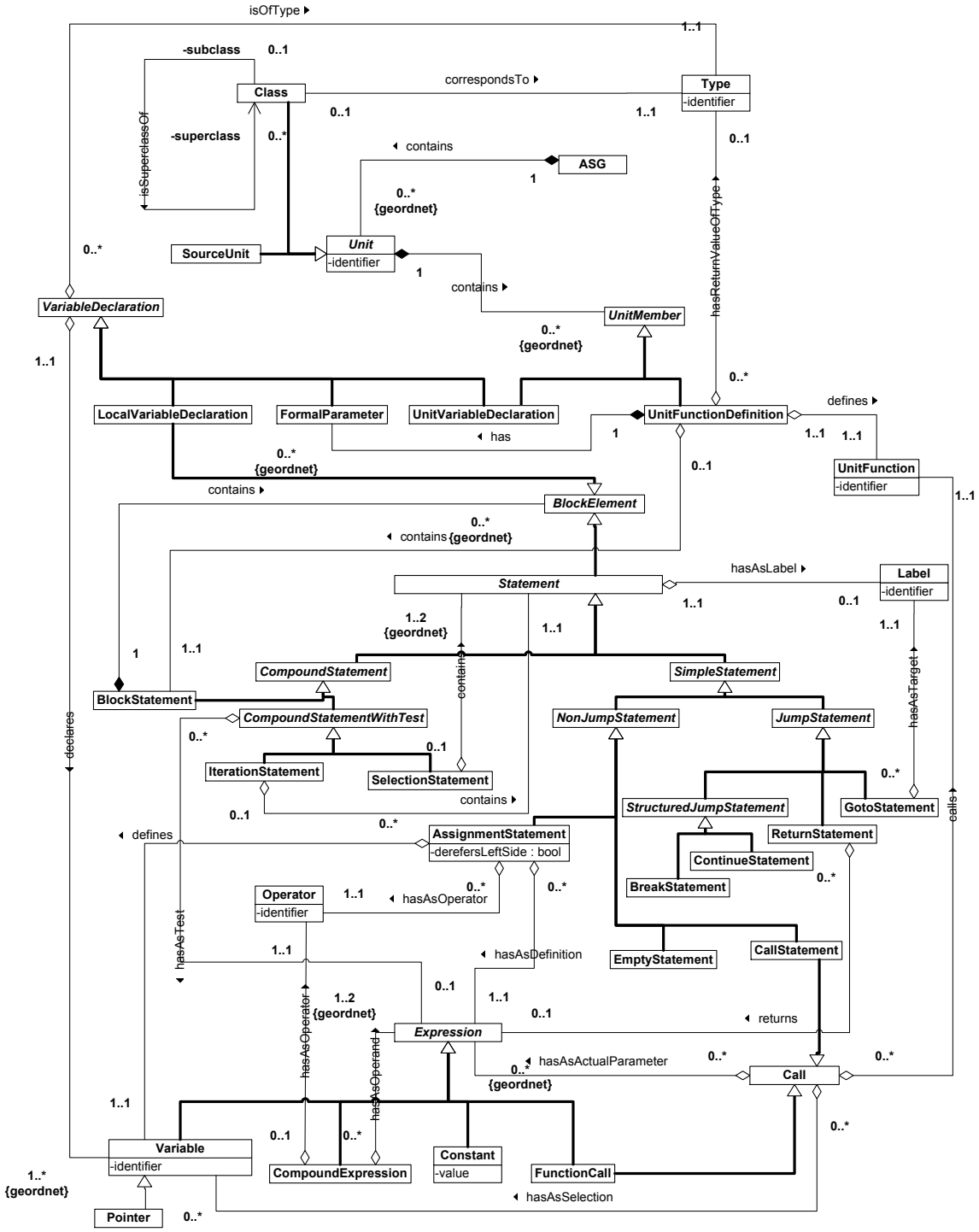

Abbildung 3.2.: Das Referenzschema für Program Slicing (RePSS)

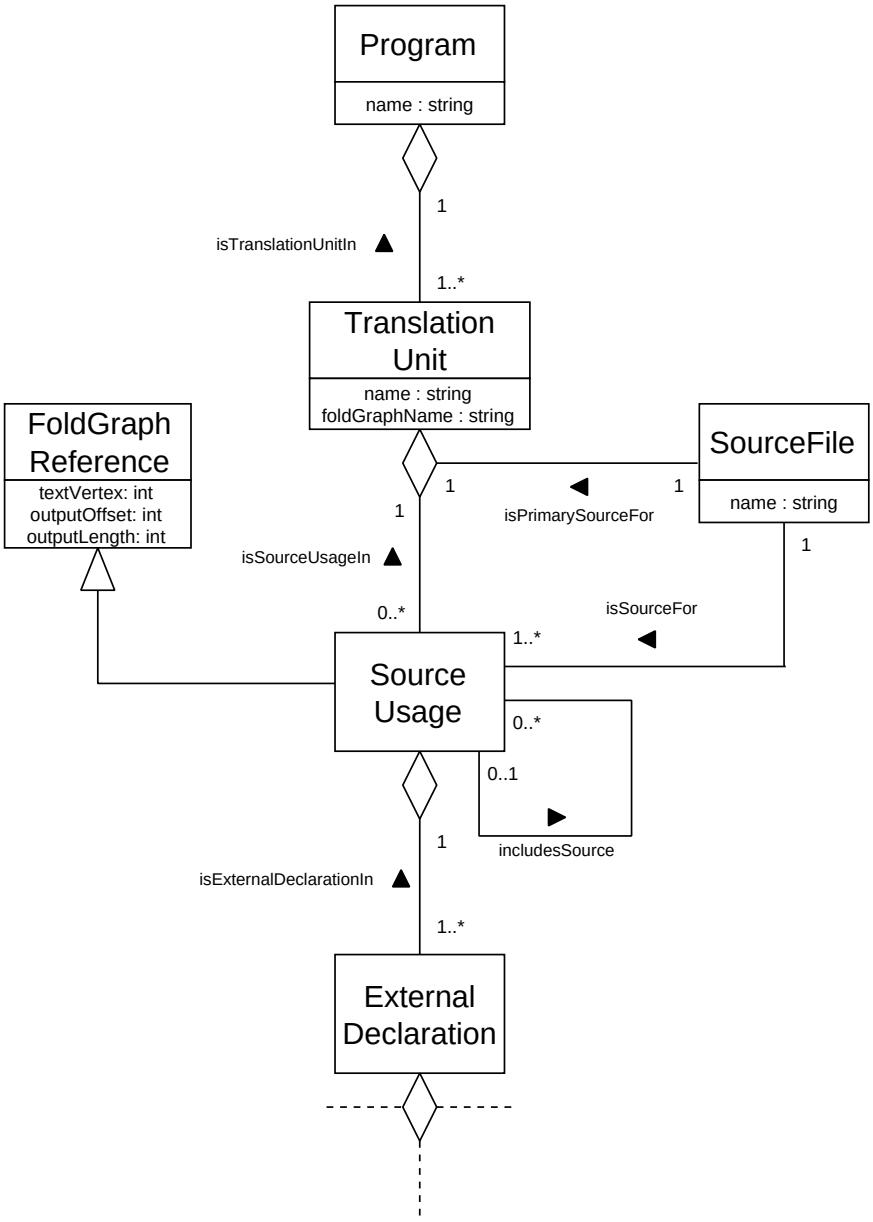

Abbildung 3.3.: C-Schema (Teil 1) [Rie01a]

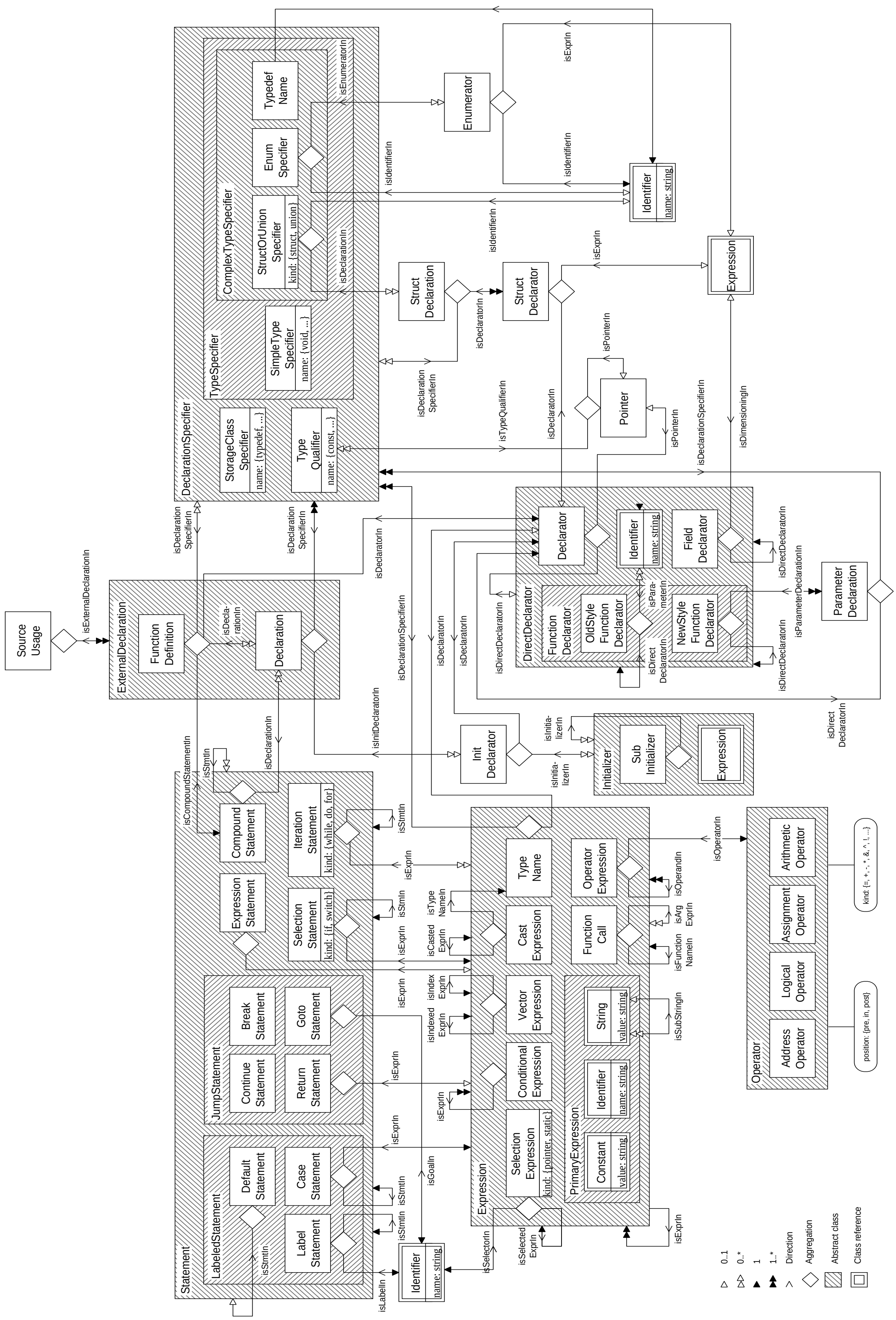

Abbildung 3.4.: C-Schema (Teil 2) [Rie01a]

Da die grafischen Darstellungen des C-Schemas und des Referenzschemas beide bereits relativ groß sind, wird aus Gründen der Übersichtlichkeit auf eine grafische Darstellung der Abbildung zwischen dem C-Schema und dem Referenzschema verzichtet. Die grafische Darstellung der Abbildung müsste mindestens beide Schemata enthalten und zusätzliche Spezialisierungspfeile zwischen den Klassen, die aufeinander abgebildet werden. Dabei sind die Klassen des Referenzschemas die Oberklassen und die Klassen des C-Schemas die Spezialisierungen.

Die Abbildung zwischen den beiden Schemata wird durch die Tabellen 3.1, 3.2 und 3.3 beschrieben.

3.3.1. Direkte Abbildungen vom RePSS auf das C-Schema

Tabelle 3.1 enthält die *direkten Abbildungen* vom RePSS auf das C-Schema. Diese Abbildungen lassen sich relativ einfach durch *Spezialisierungen* umsetzen. Das heißt, es existieren Klassen im Referenzschema, die Superklasse von C-Schema-Klassen sind. So kann zum Beispiel der Tabelle 3.1 entnommen werden, dass die RePSS-Klasse `LocalVariable-Declaration` Oberklasse der C-Schema-Klasse `Declaration` ist oder dass die C-Schema-Klasse `CompoundStatement` eine Spezialisierung der RePSS-Klasse `Block-Statement` ist.

Als Beleg für die Korrektheit der Tabelle 3.1 dient ein Vergleich zwischen dem Referenzschema und dem C-Schema.

1 Die Abbildung zwischen `Program` und `Program` wird gemacht, weil beides als Startsymbol der jeweiligen Grammatik betrachtet werden kann.

2 RePSS wurde um `SourceUnit` erweitert, damit eine Abbildung zur `SourceUsage` im C-Schema gemacht werden kann. Siehe dazu Abschnitt 3.2.

3 `DeclarationSpecifier` wurde als Spezialisierung von `Type` ausgewählt, weil die Klasse `DeclarationSpecifier` an höchster Vererbungshierarchiestufe zur Spezifikation von Deklarationen steht. Die Unterklassen des `DeclarationSpecifier` beschreiben Details, die im Referenzschema nicht gezeigt werden. Außerdem wird dem Analogieprinzip entsprechend die Klasse `Type` bzw. `DeclarationSpecifier` von den Klassen

	RePSS	**C-Schema**
1	Program	Program
2	SourceUnit	SourceUsage
3	Type	*DeclarationSpecifier*
4	LocalVariableDeclaration	Declaration
5	UnitVariableDeclaration	*ExternalDeclaration*
6	UnitFunctionDefinition	FunctionDefinition
7	*Statement*,	*Statement*
8	BlockStatement	*CompoundStatement*
9	IterationStatement	IterationStatement
10	SelectionStatement	SelectionStatement
11	*JumpStatement*	*JumpStatement*
12	ContinueStatement	ContinueStatement
13	BreakStatement	BreakStatement
14	ReturnStatement	ReturnStatement
15	GotoStatement	GotoStatement
16	AssignmentStatement	ExpressionStatement
17	*Expression*	*Expression*
18	Constant	Constant
19	CompoundExpression	OperatorExpression
20	Operator	*Operator*
21	Label	*LabeledStatement*

Tabelle 3.1.: Abbildungen zwischen RePSS und dem C-Schema durch Spezialisierung

FunctionDefinition und Declaration bzw. UnitFunctionDefinition und VariableDeclaration aggregiert. Da VariableDeclaration nicht im C-Schema abgebildet werden kann, müssen hier die entsprechenden Unterklassen verglichen werden. Siehe dazu Tabelle 3.1 und Abschnitt 3.3.2.

4 Die Klasse LocalVariableDeclaration beschreibt Variablendeklarationen, die sich in einem Block befinden. Somit entspricht sie der Klasse Declaration, deren Instanzen sich ebenfalls in einem CompoundStatement oder einer FunctionDefinition befinden. Die von VariableDeclaration geerbten Aggregationen der Klassen Variable und Type entsprechen den Aggregationen der Klasse Declaration.

5 Durch die Klasse UnitVariableDeclaration werden globale Variablen beschrieben, die von der Klasse Unit beinhaltet werden. Im C-Schema beinhaltet die Klasse SourceUsage, als Spezialisierung von SourceUnit, die abstrakte Klasse ExternalDeclaration. Somit muss eine globale Variable laut C-Schema Instanz der Klasse ExternalDeclaration sein. Im konkreten Fall ist die globale Variable somit auch Instanz der Klasse Declaration. Da aber nicht jede Declaration-Instanz eine globale Variable ist, besteht die Abbildung zwischen ExternalDeclaration und UnitVariableDeclaration.

6 FunctionDefinition wird als Spezialisierung von UnitFunctionDefinition betrachtet, da FunctionDefinition genau eine CompoundStatement-Instanz besitzt, die der BlockStatement-Instanz von UnitFunctionDefinition entspricht. Außerdem aggregiert FunctionDefinition eine DeclarationSpecifier-Instanz, welche das UnitFunctionDefinition-Aggregat Type spezialisiert. Eine direkte Abbildung der Klasse FormalParameter gibt es im C-Schema nicht. Um entsprechendes zur Komposition zwischen FormalParameter und UnitFunctionDefinition im C-Schema zu finden, sei auf Abschnitt 3.3.3 verwiesen.

7 Aufgrund der gleichen Bezeichner ist die Abbildung von Statement auf Statement naheliegend. Außerdem befinden sich diese beiden abstrakten Klassen in CompoundStatement bzw. BlockStatement, in SelectionStatement und in IterationStatement. Zudem sind beide Klassen mit Label-Klassen assoziiert.

8 Die Klasse `CompoundStatement` ist innerhalb des C-Schemas die Klasse, die beliebig viele andere `Statement`-Instanzen beinhalten kann. Innerhalb des Referenzschemas übernimmt die Klasse `BlockStatement` diese Rolle, da eine `BlockStatement`-Instanz beliebig viele `Statement`-Instanzen aggregieren kann. Zusätzlich sind beide Klassen Spezialisierungen von `Statement` und werden von `FunctionDefinition` bzw. `UnitFunctionDefinition` aggregiert.

9 Die RePSS-Klasse `IterationStatement` als Unterklasse von `Statement` entspricht der Klasse `IterationStatement` im C-Schema, da das C-`IterationStatement` analog dazu von `Statement` erbt und beide Klassen genau ein `Statement` enthalten.

10 Die C-`SelectionStatement`-Klasse kann als Spezialisierung von `SelectionStatement` im Referenzschema betrachtet werden, da beide Klassen die gemeinsame Oberklasse `Statement` haben und mindestens eine Instanz dieser Oberklasse aggregieren. Bei dieser Abbildung wird vernachlässigt, dass `SelectionStatement` in Abb. 3.4 genau ein `Statement` beinhaltet, da dies ein Fehler in der Grafik des C-Schema ist.

11 Da C-`JumpStatement` und `JumpStatement` im Referenzschema den gleichen Klassennamen haben, die gleichen Unterklassen (siehe 12 bis 15) und beide von `Statement` erben, wird zwischen beiden Klassen eine Abbildung hergestellt.

12 Die C-`ContinueStatement`-Klasse wird aufgrund der gemeinsamen Oberklasse `JumpStatement` und des gleichen Bezeichners auf `ContinueStatement` im Referenzschema abgebildet.

13 `BreakStatement` im C-Schema ist eine Spezialisierung von `BreakStatement` in RePSS, weil beide Klassen die Oberklasse `JumpStatement` und gleiche Bezeichner haben.

14 Die Klassen `ReturnStatement` und `ReturnStatement` haben gleiche Bezeichner, die gleiche Position in der Vererbungshierarchie und enthalten höchstens eine Instanz von `Expression`. Daher kann `ReturnStatement` im C-Schema als Spezialfall von `ReturnStatement` im Referenzschema betrachtet werden.

15 Die Klassen `GotoStatement` und `GotoStatement` haben gleiche Bezeichner und die gleiche Position in der Vererbungshierarchie. Ihre Abbildung aufeinander wird zusätzlich bestätigt durch die Aggregation der Klasse `Label` bzw. der Klasse `Identifier`, die sich in einem `LabelStatement` befindet.

16 Die Klassen `AssignmentStatement` und `ExpressionStatement` besitzen unterschiedliche Assoziationen, so dass die Verwendung der Klasse `Assignment-Statement` genauer definiert ist als die Verwendung der Klasse `Expression-Statement`. Da beide Klassen Unterklasse von `Statement` sind und `Expression-Statement` die einzige `Statement`-Spezialisierung ist, die eine `Expression` enthält, werden die beiden Klassen aufeinander abgebildet. Jedoch ist zu beachten, dass das Definieren einer Variable im C-Schema durch die Klasse `OperatorExpression` geschieht, die in einem `ExpressionStatement` enthalten ist.

17 C-`Expression` kann aufgrund des gleichen Bezeichners und ähnlicher Unterklassen auf `Expression` in RePSS abgebildet werden.

18 C-`Constant` wird auf `Constant` im Referenzschema abgebildet, weil beide Klassen den gleichen Bezeichner haben, Unterklassen von `Expression` sind und das Attribut `value` besitzen.

19 Die RePSS-Klasse `CompoundExpression` ist Oberklasse der C-Schema-Klasse `OperatorExpression`, da beide Klassen genau eine `Operator`-Instanz und mindestens eine `Expression`-Instanz aggregieren. Außerdem sind beide Klassen Unterklasse von `Expression`.

20 Die C-Klasse `Operator` wird auf die RePSS-Klasse `Operator` abgebildet, weil beide Klassen gleiche Bezeichner haben und von der Klasse `OperatorExpression` bzw. `CompoundExpression` aggregiert werden.

21 Labels werden in beiden Schemata unterschiedlich mit der Klasse `Statement` verbunden. Im C-Schema aggregiert das `LabeledStatement` die Klasse `Statement`, während im Referenzschema die Klasse `Label` von der Klasse `Statement` aggregiert wird.

Da aber in beiden Schemata eine Beziehung zwischen `Label` und `Statement` bzw. zwischen `LabeledStatement` und `Statement` besteht, wird `Label` als Superklasse von `LabeledStatement` definiert.

3.3.2. Klassen des C-Schemas und von RePSS ohne Abbildung

Die Tabelle 3.2 zeigt Klassen beider Schemata, die kein entsprechendes Gegenstück im anderen Schema haben. Dies kann unterschiedliche Ursachen haben:

- Die Schemata sind in manchen Stellen unterschiedlich detailliert.
- Der Funktionsumfang von C ist reichhaltiger als der von EL.

Tabelle 3.2 beschreibt einige *Differenzen* zwischen den beiden Schemata. In diesem Abschnitt wird das weitere Verfahren bezüglich dieser Differenzen beschrieben.

1 Die Klasse `Class` in RePSS besitzt im C-Schema kein Gegenstück, da C nicht objektorientiert ist. Siehe dazu auch Abschnitt 3.2.

2 Für die Klasse `EmptyStatement` existiert ebenfalls keine entsprechende Klasse im C-Schema. Dieser Fall ist jedoch weniger relevant, da das `EmptyStatement` beim Programmieren keine Relevanz hat.

3 Da `UnitFunction` von der Klasse `UnitFunctionDefinition` aggregiert wird, benötigt man im Kontext des Program Slicings nur die Abbildung der Klasse `UnitFunctionDefinition`. Die entsprechende Abbildung zwischen `UnitFunctionDefinition` und `FunctionDefinition` befindet sich in Tabelle 3.1.

4 Im C-Schema gibt es verschiedene Varianten zur Modellierung der Klasse `FormalParameter`. Zum einen gibt es eine Umsetzung durch beliebig viele `Identifier`-Instanzen, die verwendet wird, wenn im `Declarator` eine Instanz von `OldStyleFunctionDeclarator` genutzt wird. Zum anderen existiert eine Umsetzung durch `ParameterDeclaration`-Instanzen, die nur in Kombination mit einer `NewStyleFunctionDeclarator`-Instanz Verwendung findet.

	RePSS	**C-Schema**
1	Class	-
2	EmptyStatement	-
3	UnitFunction	-
4	FormalParameter	-
5	-	TypeQualifier, StorageClassSpecifier, *TypeSpecifier*, SimpleTypeSpecifier, *ComplexTypeSpecifier*, TypedefName, EnumSpecifier, StructOrUnionSpecifier
5	-	*PrimaryExpression*, String
5	-	VectorExpression, SelectionExpression, CastExpression, TypeName, ConditionalExpression
5	-	DefaultStatement, CaseStatement, LabelStatement
6	-	SourceFile
6	-	FoldGraphReference
6	-	Translation Unit
6	-	*FunctionDeclarator*, OldStyleFunctionDeclarator, NewStyleFunctionDeclarator, *DirectDeclarator*, Declarator, FieldDeclarator
6	-	StructDeclaration
6	-	StructDeclarator, InitDeclarator
6	-	Enumerator
7	-	*Initializer*, SubInitializer
8	-	ParameterDeclaration

Tabelle 3.2.: Elemente eines Schemas ohne direkte Abbildung ins andere Schema

5 Die in Tabelle 3.2 unter 5 gelisteten C-Schema Klassen besitzen alle eine Oberklasse im C-Schema, die Unterklasse einer Klasse im Referenzschema ist, siehe dazu Tabelle 3.1 und Abb. 3.4. Bezüglich dieser Klassen ist das C-Schema detaillierter als RePSS. Im Kontext des Program Slicings können diese Klassen jedoch vernachlässigt werden, da die Abstraktionsebene des Referenzschemas ausreichend ist.

6 Die unter 6 gelisteten C-Schema-Klassen beschreiben Aspekte, die innerhalb des Referenzschemas nicht berücksichtigt werden. Daher sind diese Klassen auch nicht relevant für das Program Slicing nach [Sch07] und können somit vernachlässigt werden.

7 Die Klassen `Initializer` und `SubInitializer` sind bei der Berechnung des ACFG von Bedeutung. Da es keine passende Oberklasse im Referenzschema gibt, wird die Klasse `Initializer` als Unterklasse von `ACFGVertex` festgelegt. Siehe dazu Abschnitt 3.4.2.

8 `ParameterDeclaration` ist eine der unter 4 angesprochenen Varianten zur Modellierung der Klasse `FormalParameter`.

3.3.3. Unterschiede zwischen dem C-Schema und RePSS

Tabelle 3.3 zeigt *Unterschiede*, die teilweise durch relativ schwierige Abbildungen gelöst werden müssen. Diese Unterschiede haben verschiedene Ursachen:

- Beide Schemata sind in manchen Stellen unterschiedlich detailliert.
- RePSS verwendet Mehrfachvererbung.
- Entitäten werden als Attribut oder als eigenständige Klasse in beiden Schemata unterschiedlich eingesetzt.
- Die Vererbungshierarchie des Referenzschemas ist an bestimmten Stellen tiefer als die des C-Schemas.

Tabelle 3.3 enthält nicht nur Klassen, sondern auch Attribute von Klassen. Attribute werden durch das Präfix „-“ gekennzeichnet.

	RePSS	**C-Schema**
1	*CompoundStatementWithTest*	SelectionStatement, IterationStatement
2	*SimpleStatement*	*JumpStatement*, ExpressionStatement, FunctionCall
3	*NonJumpStatement*	ExpressionStatement, FunctionCall
4	*StructuredJumpStatement*	ContinueStatement, BreakStatement
5	*BlockElement*	*Statement*, LocalVariableDeclaration
6	*CompoundStatement*	Block, CompoundStatementWithTest
7	*UnitMember*	UnitVariableDeclaration, UnitFunctionDefinition
8	*VariableDeclaration*	LocalVariableDeclaration, FormalParameter UnitVariableDeclaration
9	Call, CallStatement, FunctionCall	FunctionCall
10	Variable, Pointer	Identifier, Pointer
11	Variable, -identifier	Identifier

Tabelle 3.3.: Unterschiede zwischen RePSS und dem C-Schema

Häufig kommt es zu Unterschieden zwischen beiden Schemata, weil unterschiedlich tiefe Vererbungshierarchien verwendet werden.

1 So gibt es im Referenzschema das `CompoundStatementWithTest`. Da innerhalb des C-Schemas keine entsprechende Klasse in der gleichen Vererbungshierarchiestufe existiert, müssen alle Klassen in der nächsttieferen Vererbungshierarchiestufe betrachtet werden. Dadurch ergibt sich dann die Abbildung zwischen `CompoundStatementWithTest` und {`SelectionStatement`, `IterationStatement`}. Diese Abbildung wird jedoch nicht explizit im Referenzschema modelliert, da die Spezialisierung bereits auf der tieferen Ebene vorgenommen wurde, siehe Abschnitt 3.3.1.

2 bis 8 Der gleiche Sachverhalt führt zu den Abbildungen

- von `NonJumpStatement` auf {`ExpressionStatement`, `FunctionCall`},
- von `StructuredJumpStatement` auf {`ContinueStatement`, `BreakStatement`},

- von `SimpleStatement` auf {`JumpStatement`, `ExpressionStatement`, `FunctionCall`},
- von `BlockElement` auf {`Statement`, `LocalVariableDeclaration`},
- von `CompoundStatement` auf {`Block`, `CompoundStatementWithTest`},
- von `UnitMember` auf {`UnitVariableDeclaration`, `UnitFunctionDefinition`} und
- von `VariableDeclaration` auf {`LocalVariableDeclaration`, `FormalParameter`, `UnitVariableDeclaration`}.

Analog zur Abbildung des `CompoundStatementWithTest` werden auch diese Abbildungen nicht modelliert.

Die restlichen Einträge der Tabelle 3.1 beschreiben die schwierigeren Abbildungen zwischen dem Referenzschema und dem C-Schema.

9 Die Abbildung zwischen {`FunctionCall`, `Call`, `CallStatement`} und `FunctionCall` beschreibt einen Fall, in dem RePSS detaillierter als das C-Schema ist. Da aber innerhalb des Referenzschemas eine Vererbungshierarchie mit `Call` als Superklasse besteht, kann hier `FunctionCall` als Spezialisierung von `Call` definiert werden.

10 Die Modellierung des Zeiger-Konzeptes unterscheidet sich im ursprünglichen EL-Schema sehr stark von der Modellierung im C-Schema. Aufgrund dieses Unterschiedes kann keine Abbildung zwischen den beiden Klassen `Pointer` gefunden werden, so dass eine spezielle Lösung nötig ist. Zum Erkennen von Zeigern gibt es in RePSS die neue Klasse `Pointer`, hierzu sei auf Abschnitt 3.2 verwiesen. Durch diese neue Klasse verliert die RePSS-Klasse `Variable` das Attribut `isPointer`, so dass nun eine Abbildung zwischen der Klasse `Variable` und der C-Schema-Klasse `Identifier` gemacht werden kann. Diese Abbildung ermöglicht das Erkennen von `Identifier`-Instanzen, die eine Variable oder einen Zeiger repräsentieren.

11 Die unterschiedliche Modellierung von Entitäten als eigenständige Klasse oder als Attribut einer Klasse führt ebenfalls zu Unterschieden zwischen den beiden Schemata. Eine

Beispiel-Abbildung hierfür ist {`Variable`, `-identifier`} mit `Identifier`. Diese Abbildung lässt sich leicht umsetzen, da im Grunde eine 1:1 Beziehung zwischen zwei Klassen vorliegt. Dazu wird das Attribut `-identifier` im Referenzschema als eigenständige Klasse mit dem Attribut `-name` modelliert und dann als Superklasse von `Identifier` definiert. Da in JGraLab keine Attribute überschrieben werden können, muss das Attribut `-name` von der C-Schema-Klasse `Identifier` entfernt werden.

3.3.4. Kantenklassen im C-Schema und in RePSS

In den bisherigen Abschnitten wurden die Gemeinsamkeiten und Unterschiede zwischen beiden Schemata im Bezug auf die Knotenklassen diskutiert. Dabei wurde stets die *Richtung der Kanten* vernachlässigt. In diesem Abschnitt werden nun die Gemeinsamkeiten und Unterschiede bezüglich der Kantenklassen beschrieben.

Analog zu den Abbildungen zwischen den Knotenklassen in beiden Schemata könnte man Abbildungen zwischen den Kantenklassen der unterschiedlichen Schemata vornehmen. Allerdings sprechen zwei entscheidende Gründe dagegen:

1. Kantenklassen, die zwischen aufeinander abgebildeten Knotenklassen verlaufen, haben immer unterschiedliche Richtungen.

2. Die Typ-Bezeichnungen der einzelnen Kantenklassen sind im C-Schema nicht ausreichend differenziert.

Im C-Schema sind die Bezeichner und die Richtungen der Kantenklassen immer nach dem Muster „isIn“ aufgebaut. Das bedeutet, dass eine solche Kante immer vom aggregierten Knoten zum aggregierenden Knoten verläuft, da alle Kanten[1] des C-Schemas Aggregationen repräsentieren. Zum Beispiel verläuft von der Knotenklasse `Statement` eine `IsStmtIn`-Kantenklasse zu der Knotenklasse `CompoundStatement`.

Innerhalb des Referenzschema ist die Ausrichtung der Kanten genau spiegelverkehrt zur Kantenausrichtung im C-Schema. Im Referenzschema gibt es im Wesentlichen nur Aggregationen und Kompositionen, die verwendeten Assoziationen stehen im Zusammenhang mit der Objektorientierung und können daher keine Abbildung im C-Schema besitzen. Das verwendete Muster für die Aggregations- und Kompositionsklassen könnte man mit „contains“ bezeichnen. Das hat zur Folge, dass Kanten immer vom äußeren zum inneren Knoten verlau-

[1] Ausnahmen, die im C-Schema als Assoziation dargestellt werden, sind vernachlässigbar, da man sie auch als Aggregation modellieren kann.

fen und somit eine spiegelverkehrte Ausrichtung zu den Kanten im C-Schema besitzen. Zum Beispiel verläuft eine `contains`-Kante vom `BlockStatement` zum `BlockElement`.

Die Bezeichnungen der Kantenklassen im Referenzschema wurden im Hinblick auf die Verwendung innerhalb von JGraLab so gewählt, dass jede Kantenklasse einen individuellen Namen hat. In Abb. 3.2 wurde aus Gründen der Übersichtlichkeit auf die ausführliche Bezeichnung verzichtet. Tatsächlich besitzt aber jeder Kantenklasse als Präfix die Bezeichnung der Alpha[2]-Knotenklasse und als Postfix die Bezeichnung der Omega[3]-Knotenklasse. Zum Beispiel ist die tatsächliche Bezeichnung der Kantenklasse, die zwischen den Knotenklassen `BlockStatement` und `BlockElemement` verläuft, `ELBlockStatement-ContainsELBlockElement`[4].

Innerhalb des C-Schemas ist die Bezeichnerwahl der Kantenklassen nicht so differenziert wie in RePSS, da bei der Erstellung auf Prä- und Postfixe verzichtet wurde. So existiert zum Beispiel die Kantenklasse `IsPointerIn` zum einen zwischen den Knotenklassen `Pointer` und `Declarator` und zum anderen zwischen `Pointer` und `Pointer`.

Würde man nun die Ausrichtung und die Bezeichnung der Kanten innerhalb des C-Schemas oder des Referenzschemas anpassen, so könnte man Abbildungen zwischen den Kantenklassen beider Schemata vornehmen. Diese Änderung wird am Referenzschema jedoch nicht durchgeführt, weil somit das Problem der entgegengesetzten Kantenausrichtungen in C- und Referenzschema nicht gelöst wird und bei den Schemata anderer Programmiersprachen erneut auftauchen kann. Eine Änderung des C-Schemas würde eine entsprechende Anpassung des C-Parsers erfordern. Diese Lösung wird aufgrund des hohen Aufwands nicht umgesetzt.

Damit die Komponenten von RePST dennoch effizient mit Kantenklassen arbeiten können, wird die Superkantenklasse `Edge` für Berechnungen verwendet. Es gibt zwei Möglichkeiten, um die Problematik bezüglich der Kantenausrichtung anzugehen. Zum einen gibt es die JGraLab-Methoden `getThis` und `getThat`, deren Benutzung den gezielten Zugriff auf die Knoten an den beiden Enden einer Kante erlaubt. Zum anderen bietet die Implementierung von RePST Hilfsmethoden (siehe Abschnitt 6.22) an, deren Verwendung die Traversierung eines TGraphen in Richtung der Wurzel oder der Blätter ermöglichen, sofern der TGraph einen abstrakten Parse-Baum beschreibt.

[2]Der Knoten, von dem eine Kante ausgeht, wird in JGraLab als Alpha bezeichnet.
[3]Der Knoten, an dem eine Kante eingeht, wird in JGraLab als Omega bezeichnet.
[4]Knotenklassen innerhalb des Referenzschemas besitzen das Präfix „EL“

3.4. Datenmodelle des C-Schemas

Zwischen den einzelnen Diensten des Program Slicing Tools fließen Daten, die ein bestimmtes, zu slicendes Programm beschreiben. Die Beschreibung des Programms geschieht mit Hilfe von Graphen. Daher entspricht der Aufbau dieser Daten einem Graphschema. Das Graphschema setzt sich aus mehreren Graphklassen zusammen. Jede Programmiersprache (C, C++, Java, ...) sowie RePSS hat eine eigene Graphklasse. Dabei erlaubt jede Graphklasse mehrere, unterschiedliche *Sichten* auf das zu slicende Programm. Diese Sichten sind im Einzelnen:

- der abstrakte Syntaxgraph (ASG)
- der erweiterte Kontrollflussgraph (ACFG)
- der Points-to-Graph (PG)
- der Aufrufgraph (CG)
- der erweiterte Aufrufgraph (ECG)
- der erweiterte Systemabhängigkeitsgraph (ESDG)

Zum Slicen von C-Programmen muss das Graphschema sowohl das C-Schema als auch das Referenzschema enthalten. In den nächsten Unterabschnitten werden die Sichten auf C-Programme vorgestellt, welche durch die C-Graphklasse möglich sind.

Weiterführende Erklärungen und Beispiele zum Aufbau und zur Bedeutung aller Sichten findet der Leser in [Sch07]. Abschnitt 7.2 beinhaltet ebenfalls Beispiele zu den Teilgraphen aller Sichten.

3.4.1. Abstrakter Syntaxgraph (ASG) für C

Der *abstrakte Syntaxgraph* repräsentiert die Syntax des zu slicenden Programms. Er dient als Grundlage zur Berechnung der nachfolgend beschriebenen Graphen. Die Abb. 3.3 und 3.4 zeigen das Schema des ASG für die Programmiersprache C.

Der ASG wird durch lexikalische und syntaktische Analyse erstellt. Diese Compilerbautechniken arbeiten mit der C-Grammatik [BCM+98]. Daher ist das ASG-Schema für C aus der C-Grammatik hergeleitet worden.

3.4.2. Erweiterter Kontrollflussgraph (ACFG) für C

Ein *Kontrollflussgraph* beschreibt alle möglichen Abarbeitungsreihenfolgen von Anweisungen und lokalen Variablendeklarationen innerhalb eines Funktionsrumpfes. Der *ACFG* erweitert diesen Kontrollflussgraphen um weitere Kontrollflusskanten, die zur Berechnung von Kontrollkanten innerhalb eines erweiterten Systemabhängigkeitsgraphen nötig sind.

Eine *ACFG-Menge* (`ACFGSet`) besitzt für jede Methode innerhalb des zu beschreibenden Programms einen ACFG (`ACFG`). Dieser ACFG enthält für jede Anweisung oder lokale Variablendeklaration einen ACFG-Knoten (`ACFGVertex`). Die einzelnen ACFG-Knoten innerhalb eines ACFG sind durch Kontrollflusskanten (`ACFGVertexHasControlFlowEdgeToACFGVertex`) miteinander verbunden, wobei von jedem ACFG-Knoten maximal zwei Kontrollflusskanten ausgehen dürfen.

Jede Kontrollflusskante von Knoten *s* nach Knoten *t* besitzt ein Attribut `label`. Dieses legt die Bedeutung der Kontrollflusskante fest. Es gibt vier mögliche Werte für `label`:

1. *always*: *t* wird unmittelbar nach *s* ausgeführt.

2. *never*: *t* würde nur dann nach *s* (der Typ von s kann `Entry` oder Unterklasse von `JumpStatement` sein) ausgeführt werden, wenn *s* keine `Entry`- oder `JumpStatement`-Instanz wäre.

3. *true*: *t* wird unmittelbar nach *s* (der Typ von s kann hier nur Unterklasse von `CompoundStatementWithTest` sein) ausgeführt, wenn die Ausdrucksauswertung in *s* wahr ergibt.

4. *false*: *t* wird unmittelbar nach *s* (der Typ von s kann hier nur Unterklasse von `CompoundStatementWithTest` sein) ausgeführt, wenn die Ausdrucksauswertung in *s* falsch ergibt.

Abb. 3.5 zeigt die *ACFG-Sicht* auf die C-Graphklasse. Diese Sicht wird zur Berechnung einer ACFG-Menge für die Programmiersprache C genutzt. Alle Graphelementklassen, außer der Graphelementklasse `Initializer`, stammen aus RePSS. Diese Klassen besitzen Spezialisierungen in der C-Graphklasse, die durch direkte Abbildungen umgesetzt werden können, siehe dazu Abschnitt 3.3.1.

Die Berechnung der ACFG-Menge für C entspricht der Berechnung der ACFG-Menge des Referenzschemas (siehe dazu [Sch07]). Es gibt Ausnahmen, welche eine spezielle Berechnung auf Basis des C-Schemas benötigen.

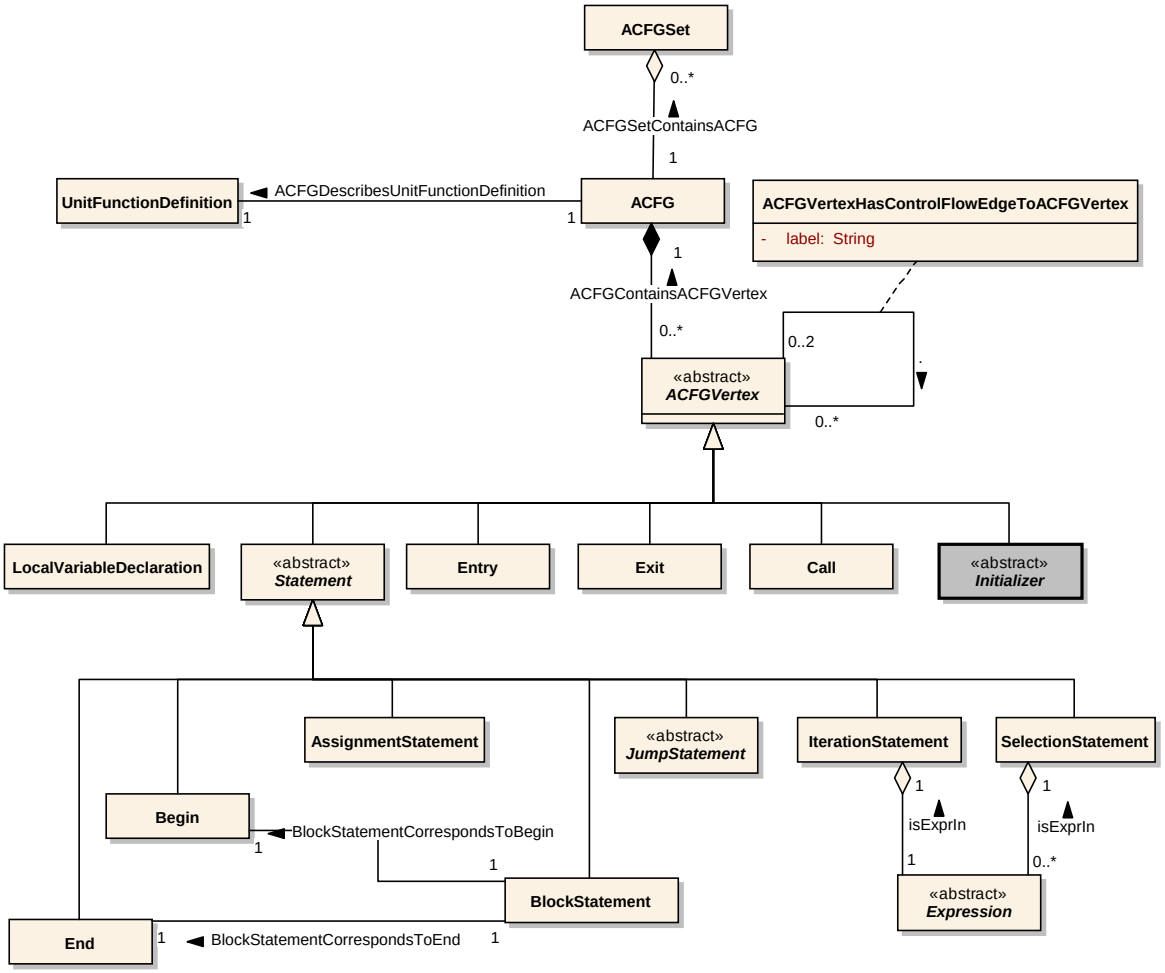

Abbildung 3.5.: Schema der ACFG-Menge für C

- Die Berechnung der Sprungmarke bei einer Goto-Anweisung kann nicht auf Basis des Referenzschemas durchgeführt werden, da der Pfad vom Goto-Knoten zum Namen der Sprungmarke in RePSS länger als im C-Schema ist.
- Die Suche nach möglichen Folgeknoten innerhalb eines Blocks benötigt eine spezielle Filterfunktion, die auf Basis des C-Schemas geschrieben werden muss.
- In EL ist es unmöglich, Variablen beim Deklarieren einen Wert zuzuweisen. Da dies aber in C möglich ist, muss es auf Basis des C-Schemas gemacht werden. Deshalb wird die C-Klasse `Initializer` als Unterklasse von `ACFGVertex` definiert. Da `Initializer` Oberklasse der C-Klasse `Expression` ist, muss stets überprüft werden, dass jede `ACFGVertex`-Instanz ausschließlich mit `Expression`-Instanzen verknüpft ist, welche eine Kante zu einer `InitDeclarator`-Instanz besitzen.

3.4.3. Points-to-Graph (PG) für C

Ein *Points-to-Graph* beschreibt die „zeigt auf"-Beziehungen zwischen Variablen innerhalb einer Methode. Variablen können in diesem Fall Attribute, lokale Variablen und formale Parameter sein. Es ist zwingend notwendig, dass der Points-to-Graph eine *konservative Abschätzung* der „zeigt auf"-Beziehungen vornimmt. Unter einer konservativen Abschätzung versteht man hier, dass die tatsächlich vorhandenen „zeigt auf"-Beziehungen oder eine Obermenge davon abgebildet werden. Eine Untermenge der tatsächlich vorhandenen „zeigt auf"-Beziehungen darf unter keinen Umständen verwendet werden.

Eine *PG-Menge* (`PGSet`) besitzt für jede Methode innerhalb des zu beschreibenden Programms einen PG (`PG`). Der PG enthält alle Variablen. Variablen können Zeiger sein und somit auf andere Variablen zeigen.

Abb. 3.6 zeigt die *PG-Sicht* auf die C-Graphklasse. Zur Berechnung der PG-Menge für die Programmiersprache C wird diese Sicht benutzt. Die Berechnung der PG-Menge für C unterscheidet sich von der Berechnung anhand des Referenzschemas, da Variablen und somit auch Zeiger in beiden Schemata unterschiedlich dargestellt werden (siehe dazu Abschnitt 3.3.3).

Die PG-Sicht auf die C-Graphklasse enthält zur *Pointer-Erkennung* die C-Schema-Klassen `Declarator` und `Pointer`. Ein `Variable`-Objekt ist genau dann eine `ELPointer`-

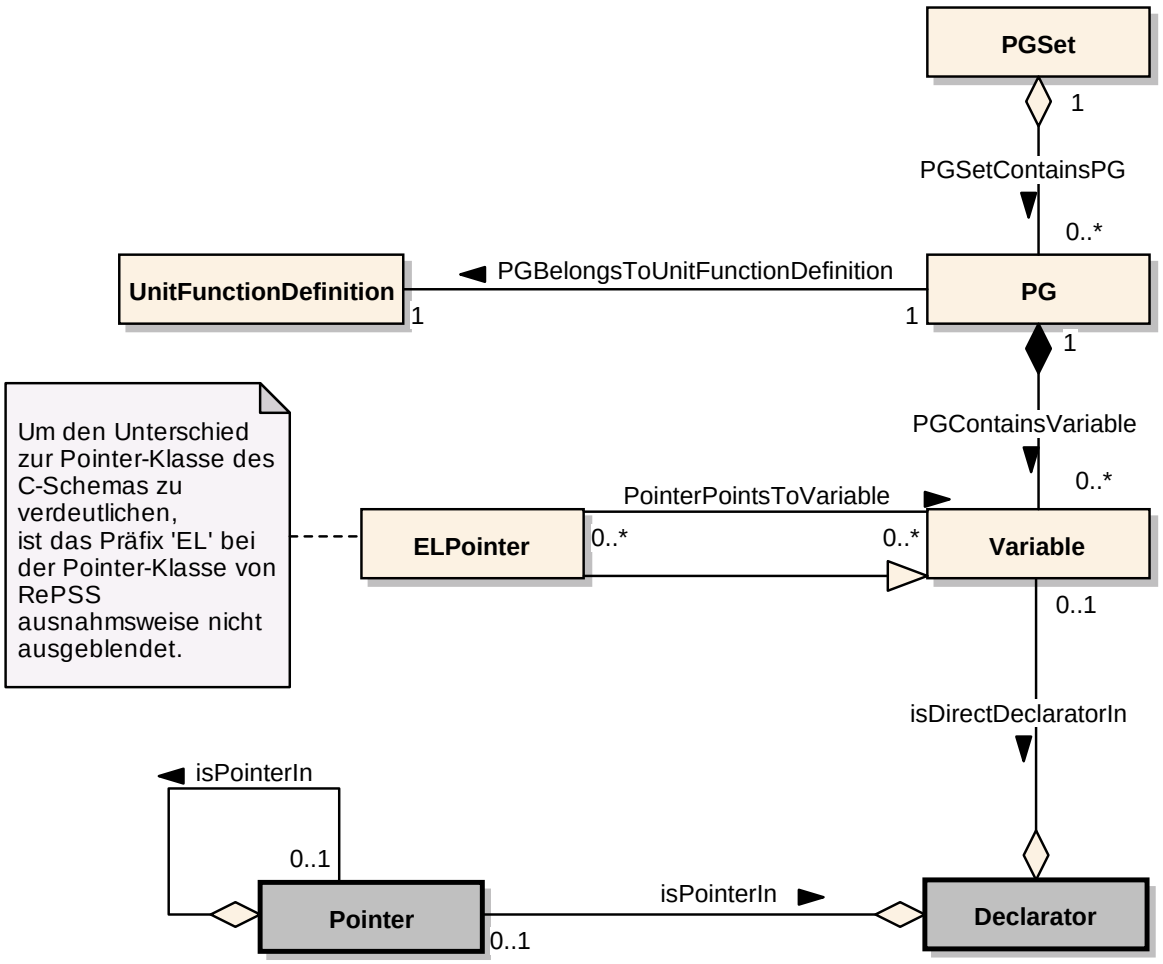

Abbildung 3.6.: Schema der PG-Menge für C

Instanz[5], wenn seine `Declarator`-Instanz per eingehender `IsPointerIn`-Kante mit einem C-`Pointer` verbunden ist. C-`Pointer` und `Declarator` sind in Abb. 3.6 grau unterlegt, da sie keine Oberklasse in RePSS besitzen.

Das Erkennen von Variablen ist im C-Schema schwieriger als in RePSS, da die Klasse `Identifier` nicht nur für Variablen benutzt wird sondern zum Beispiel auch für Methodennamen. Eine `Identifier`-Instanz ist nur dann keine Variable, wenn:

- die `Identifier`-Instanz eine Kante zu einer `LabelStatement`-Instanz besitzt.
- die `Identifier`-Instanz eine Kante zu einer `GotoStatement`-Instanz besitzt.
- die `Identifier`-Instanz eine `IsFunctionNameIn`-Kante besitzt.
- die `Identifier`-Instanz eine `IsOldStyleFunctionDirectDeclaratorIn`-Kante besitzt.
- die `Identifier`-Instanz eine `IsNewStyleFunctionDirectDeclaratorIn`-Kante besitzt.

3.4.4. Aufrufgraph (CG) und erweiterter Aufrufgraph (ECG) für C

Durch einen Aufrufgraph werden die Aufrufbeziehungen zwischen den Funktionen[6] eines Programms wiedergegeben. Innerhalb eines *erweiterten Aufrufgraph* werden neben den Aufrufbeziehungen auch die von einer Funktion definierten und benutzten Variablen mit Hilfe von Defines- und Uses-Kanten angezeigt.

Ein *ECG* (`ECG`) enthält alle ECG-Knoten (`ECGVertex`). ECG-Knoten können Instanzen vom Typ `Call`, `Variable` oder `UnitFunctionDefinition` sein.

Abb. 3.7 zeigt die *ECG-Sicht* auf die C-Graphklasse, welche zur Berechnung des ECG für die Programmiersprache C verwendet wird. Diese Sicht unterscheidet sich nicht von der ECG-Sicht auf RePSS, da keine Klassen aus dem C-Schema ohne Oberklasse in RePSS

[5] Hier ist das Präfix „EL“ zur einfacheren Differenzierung zwischen der RePSS- und C-`Pointer`-Klasse ausnahmsweise nicht ausgeblendet

[6] In dieser Diplomarbeit wird zur Vereinfachung von Funktionen gesprochen, tatsächlich sind aber Funktionen und Prozeduren in prozeduralen Programmiersprachen bzw. Methoden in objektorientierten Programmiersprachen gemeint.

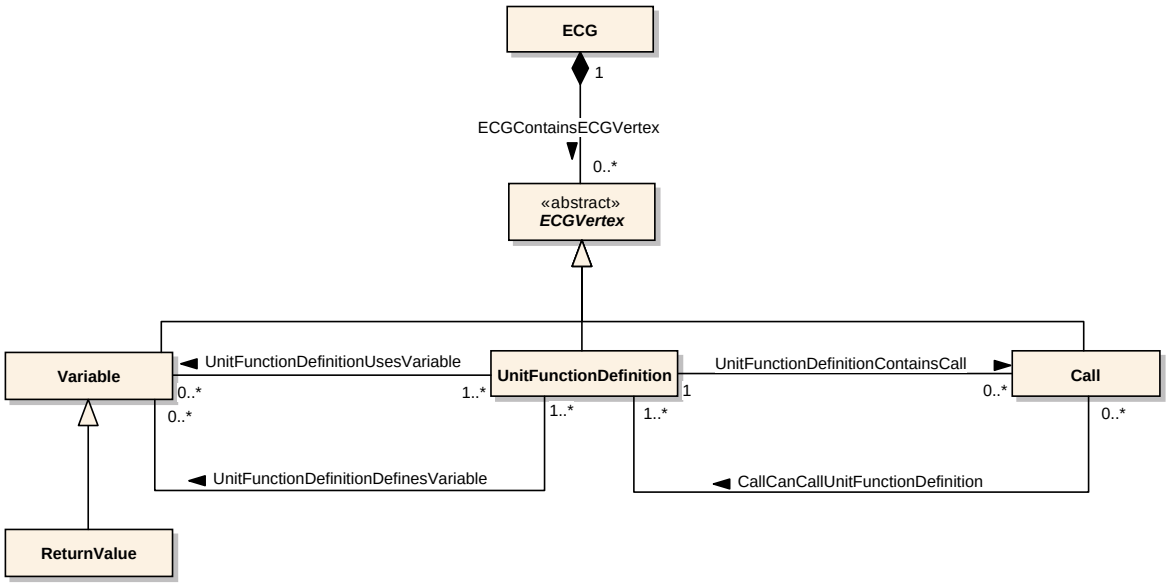

Abbildung 3.7.: Schema des ECG für C

benötigt werden. Da zur Berechnung des ECG die PG-Menge benötigt wird, ist keine erneute Sonderbehandlung zur Erkennung von `Variable`-Instanzen erforderlich.

Die Kanten des CG werden zwischen `UnitFunctionDefinition`- und `Call`-Knoten gezogen, wobei für Graphen zur Beschreibung von C-Programmen gilt:

- Eine Kante von einem `UnitFunctionDefinition`- zu einem `Call`-Knoten ist vom Typ `UnitFunctionDefinitionContainsCall`, wenn sich der Aufruf-Knoten einer anderen Funktion innerhalb der Funktion befindet, die durch den `UnitFunctionDefinition`-Knoten repräsentiert wird.

- Eine Kante von einem `Call`- zu einem `UnitFunctionDefinition`-Knoten ist vom Typ `CallCanCallUnitFunctionDefinition`, wenn der `Call`-Knoten die Ausführung der Funktion zur Folge hat, welche durch den `UnitFunctionDefinition`-Knoten vertreten wird. Da es innerhalb von C-Programmen keine Vererbung gibt, kann der Aufruf-Knoten nur eine Funktion aufrufen. In Abb. 3.7 ist dies anders dargestellt, weil durch Polymorphie unter Umständen mehrere, unterschiedliche Methoden aufrufbar sind.

Innerhalb des ECG für C-Programme werden von `UnitFunctionDefinition`- nach `Variable`-Knoten *def/use-Kanten* gezogen, für die folgendes gilt:

- Eine def/use-Kante ist vom Typ `UnitFunctionDefinitionUsesVariable`, wenn die `Variable`-Instanz innerhalb der Funktion genutzt wird. Die Nutzung einer Variable innerhalb einer Funktion erkennt man daran, dass sie einen Pfad, bestehend aus ausgehenden Kanten, zu einer `OperatorExpression`-Instanz mit =-Operator besitzt und rechts von dem =-Operator steht oder als aktueller Parameter im Funktionsaufruf vorkommt. Außerdem darf die `Variable`-Instanz keine lokale Variable repräsentieren.

- Eine def/use-Kante ist vom Typ `UnitFunctionDefinitionDefinesVariable`, wenn die `Variable`-Instanz innerhalb der Funktion definiert wird. Das heißt, dass sie eine Kante zu einer `OperatorExpression`-Instanz mit =-Operator besitzt und links von dem =-Operator steht. Die `Variable`-Instanz darf analog zur Uses-Kante keine lokale Variable und außerdem keinen Parameter repräsentieren.

Operatoren zur abkürzenden Schreibweise [BCM+98] können zum einen durch den Preprocessor so ausgeschrieben werden, dass die Regeln zur Bestimmung von def/use-Kanten greifen oder zum anderen mit Hilfe von zusätzlichen Regeln erkannt werden. Wenn man also in C für die Operatoren *=, /=, %=, +=, -=, «=, »=, &=, |=, ˆ=, ++ und -- die Regel definiert, dass globale Variablen auf der linken Seite sowohl definiert als auch benutzt werden, kann man diese Problematik auch ohne Hilfe des Preprocessors lösen.

Variablen, auf die ein Zeiger als Parameter zeigt, werden in dieser Diplomarbeit nicht berücksichtigt. Eine Liste aller Einschränkungen für C-Programme ist in Abschnitt 8.2.5 zu finden.

Uses- und Defines-Kanten können auch implizit zwischen einem Funktions-Knoten und einem Variable-Knoten entstehen. Dies passiert immer dann, wenn letzterer eine globale Variable bzw. ein Attribut ist und innerhalb einer direkt oder indirekt aufgerufenen Funktion verwendet oder definiert wird.

Die Klasse `ReturnValue` repräsentiert Hilfsvariablen, die den Rückgabewert einer Funktion darstellen. Sofern eine Funktion einen Rückgabewert besitzt, wird ein neuer `ReturnValue`-Knoten erzeugt und der Funktions-Knoten wird per `UnitFunctionDefinitionDefinesVariable`-Kante mit dem `ReturnValue`-Knoten verbunden. `ReturnValue`-Knoten besitzen keine eingehenden `UnitFunctionDefinitionUsesVariable`-Kanten.

3.4.5. Erweiterter Systemabhängigkeitsgraph (ESDG) für C

Ein *erweiterter Systemabhängigkeitsgraph* [HRB88] enthält die Knoten aller ACFGs und verbindet diese durch spezielle, auf Kontrollfluss- und def/use-Kanten basierende Kanten miteinander. Dadurch fungiert der ESDG als Grundlage für die eigentlichen Slicing-Verfahren, das sind alle Verfahren außer die zur Programmcode-Aufbereitung.

Ein *ESDG* (`ESDG`) enthält alle ESDG-Knoten (`ESDGVertex`). ESDG-Knoten können Instanzen vom Typ `ACFGVertex` oder `ParameterVertex` sein. `Exit`-Knoten werden jedoch ausgeschlossen, obwohl sie Unterklasse von `ACFGVertex` sind. Die Klasse der Parameterknoten (`ParameterVertex`) wird durch die Klassen `FormalInParameterVertex`, `FormalOutParameterVertex`, `ActualIn-ParameterVertex` und `ActualOutParameterVertex` spezialisiert.

Abb. 3.8 zeigt die *ESDG-Sicht* auf die C-Graphklasse, welche zur Berechnung des ESDG für die Programmiersprache C verwendet wird. Diese Sicht unterscheidet sich nur in einem Detail von der Sicht auf die RePSS-Graphklasse. So ist die Klasse `Initializer` zusätzlich Unterklasse von `ESDGVertex`, weil sie auf Grund eines Unterschiedes zwischen dem C-Schema und RePSS als Spezialisierung der `ACFGVertex`-Klasse berücksichtigt werden muss.

Neben den ESDG-Knoten gibt es auch noch einige Kanten, welche zur Menge der Grundelemente eines ESDG gehören. Diese Kanten sind:

- Die `ESDGVertexHasParameterEdgeToParameterVertex`-Kante, welche jedem `ParameterVertex`-Objekt genau einen ESDG-Knoten zuweist.
- Die `EntryCorrespondsToUnitFunctionDefinition`-Kante, welche Paare aus zusammengehörigen `Entry`- und `UnitFunctionDefinition`-Knoten bildet.
- Die `ReturnStatementHasAssignmentStatement`-Kante, welche jedem `ReturnStatement`-Objekt genau einen `AssignmentStatement`-Knoten zuweist.
- Die `ParameterVertexHasAssignmentStatement`-Kante, welche jedem `ParameterVertex`-Objekt genau einen `AssignmentStatement`-Knoten zuweist.

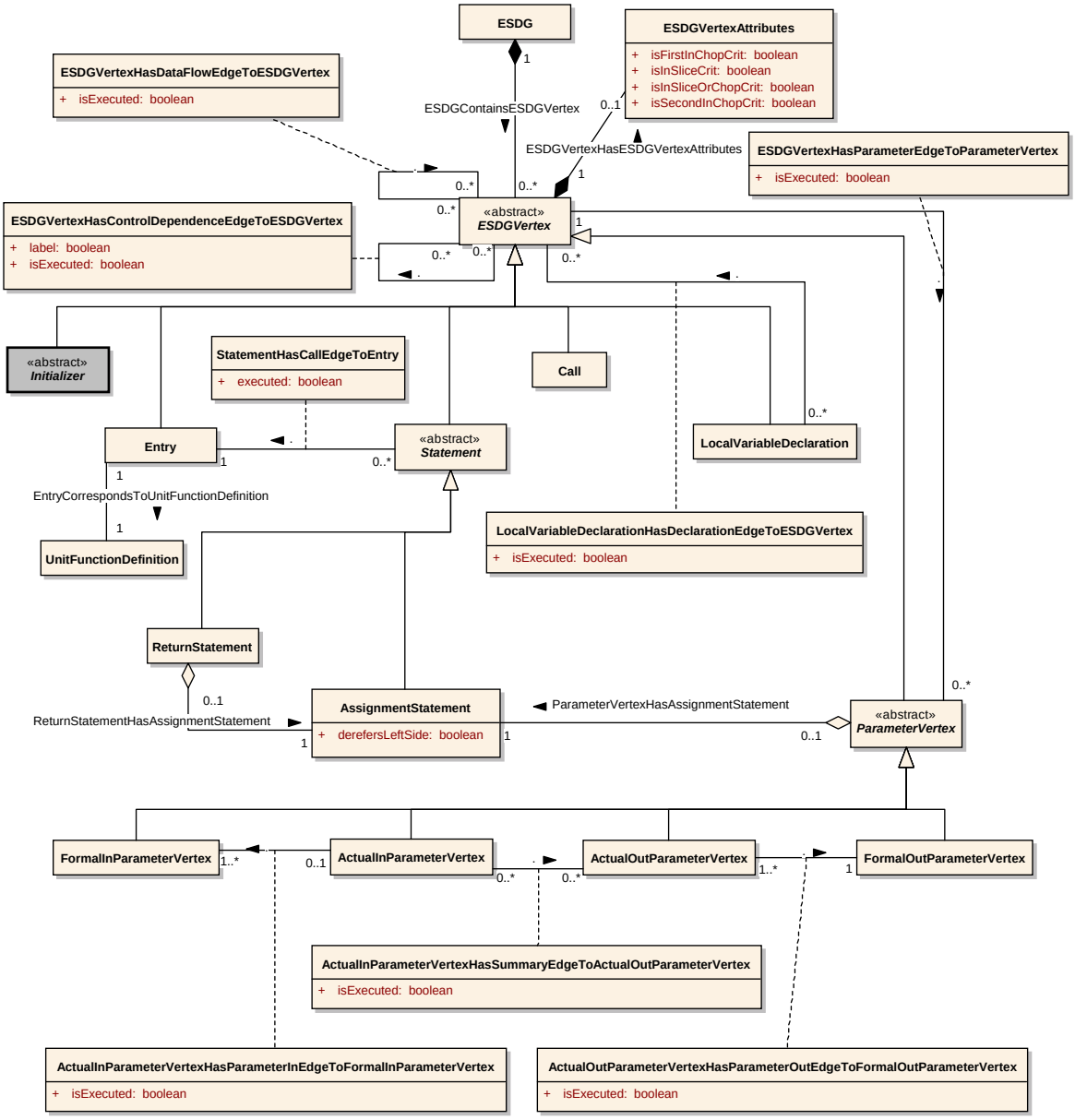

Abbildung 3.8.: Schema des ESDG für C

- Zur Gruppe der interprozeduralen Kanten gehören die Aufrufkanten (`StatementHasCallEdgeToEntry`), die Parameter-in-Kanten (`ActualInParameterVertexHasParameterInEdgeToFormalInParameterVertex`) und die Parameter-out-Kanten (`ActualOutParameterVertexHasParameterOutEdgeToFormalOutParameterVertex`). Interprozedurale Kanten beschreiben die Verbindungen zwischen den Teilgraphen zur Repräsentation von Funktionen innerhalb des ESDG.

- Kontrollkanten `ESDGVertexHasControlDependenceEdgeToESDGVertex` reflektieren die Schachtelung der Anweisungen in einer Funktion ohne Sprünge, wobei Blöcke als einfache Anweisungen betrachtet werden. Sofern Sprunganweisungen vorhanden sind, besitzen diese ebenfalls ausgehende Kontrollkanten.

- Der Datenfluss innerhalb des ESDG wird mit Hilfe von Datenflusskanten (`ESDGVertexHasDataFlowEdgeToESDGVertex`) und Deklarationskanten (`LocalVariableDeclarationHasDeclarationEdgeToESDGVertex`) dargestellt.

- Transitive Kanten [FG97] `ActualInParameterVertexHasSummaryEdgeToActualOutParameterVertex` beschreiben Beziehungen zwischen `ActualInParameterVertex`- und `ActualOutParameterVertex`-Knoten.

3.5. Fazit zu RePSS

Durch die Umsetzung der in diesem Kapitel beschriebenen Abbildungen erhält man ein neues Graphschema, welches das C-Schema und RePSS als Graphklassen enthält. Dieses Graphschema wird als *Slicing-Schema* bezeichnet.

Zur Beschreibung des Slicing-Schemas gibt es zwei Möglichkeiten [SBR06]:

1. Das Slicing-Schema kann durch eine *„tg“-Datei* [Kah06] definiert werden. Mit Hilfe von JGraLab können dann die Java-Klassen zur Beschreibung des Slicing-Schemas aus dieser „tg“-Datei generiert werden.

2. Alternativ kann die Beschreibung des Slicing-Schemas auch direkt in Java-Quellcode geschehen.

Diese Diplomarbeit enthält das Slicing-Schema als „tg“-Datei (`slicingschema.tg`).

Das Slicing-Schema enthält die Graphklasse von RePSS. Zum Slicen von Code in anderen Programmiersprachen, muss das Slicing-Schema um die Graphklasse dieser Programmiersprache ergänzt werden. In diesem Zusammenhang müssen Abbildungen zwischen den Knoten in der Graphklasse der konkreten Programmiersprache und den Knoten in der RePSS-Graphklasse so gemacht werden, wie es in diesem Kapitel exemplarisch für die Programmiersprache C vorgeführt ist. Daher enthält das Slicing-Schema die C-Graphklasse, die RePSS-Graphklasse und die Abbildungen zwischen diesen beiden Graphklassen.

Die innerhalb des Slicing-Schemas definierten Graphklassen enthalten Sichten auf abstrakte Syntaxgraphen, erweiterte Kontrollflussgraphen, Points-to-Graphen, Aufrufgraphen, erweiterte Aufrufgraphen und erweiterte Systemabhängigkeitsgraphen. Diese Sichten entsprechen dem innerhalb dieses Kapitels beschriebenen Datenmodell.

Konkretes Program Slicing kann mit dem in dieser Diplomarbeit vorgestellten RePST auf Graphen ausgeführt werden, die Quellcode beschreiben und dem Slicing-Schema entsprechen.

4. Definition von RePST

Dieses Kapitel beschreibt die *Definitionsphase* des Program Slicing Tools. In dieser Phase werden die Anforderungen an das Tool definiert. Als Grundlage für die *Anforderungsdefinition* dienen unter anderem die *Anwendungsfälle* zu Beginn diesen Kapitels.

4.1. Anwendungsfälle

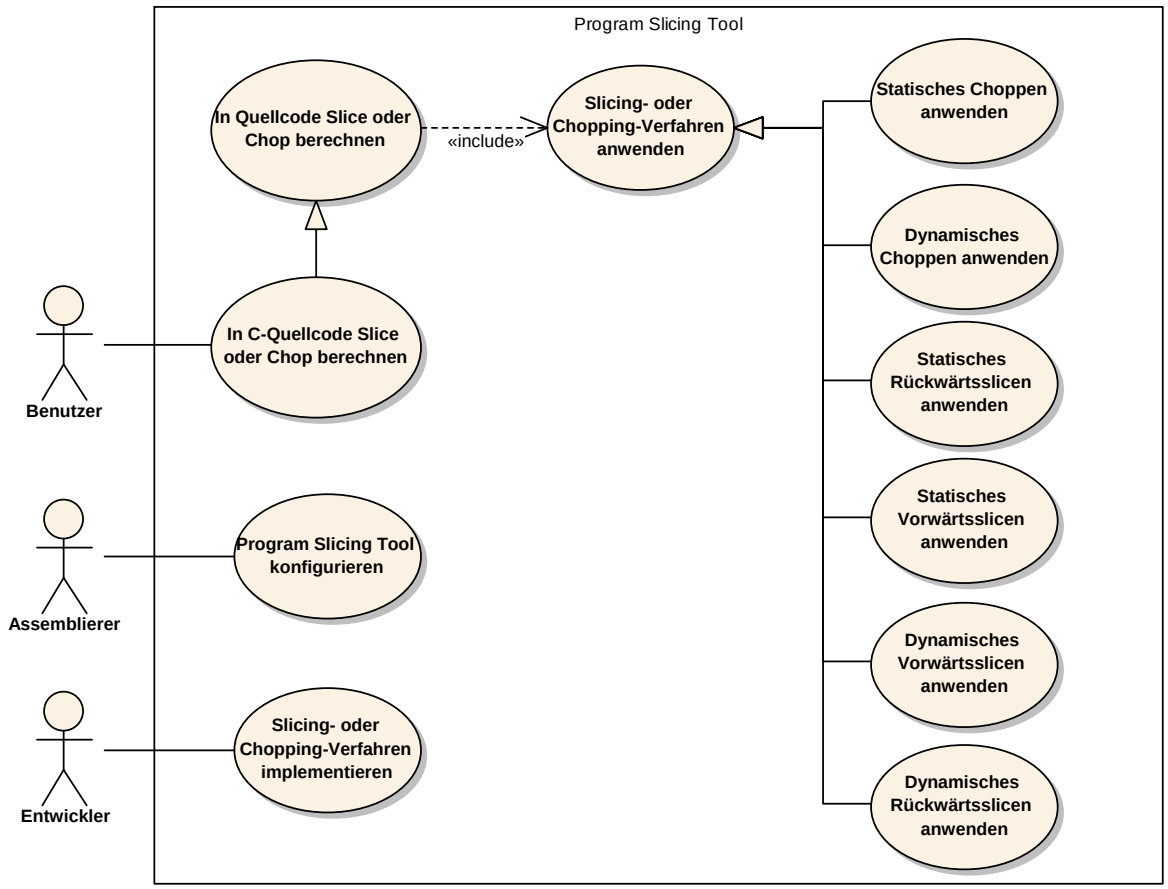

Abbildung 4.1.: Anwendungsfalldiagramm: RePST

Im Kontext dieser Diplomarbeit wird ein generisches Tool zum Slicen oder Choppen von Quellcode in beliebigen Programmiersprachen entwickelt. Da dieses Tool auf einem Dienstmodell basiert, wird es durch eine Vielzahl von Komponenten realisiert.

Entwickler können das Tool um weitere Komponenten ergänzen, indem sie die Komponenten des generischen Program Slicing Tools spezialisieren. Diese Komponenten können neue Slicing- oder Chopping-Verfahren bereitstellen oder konkrete Verfahren zum Slicen oder Choppen von Quellcode in anderen Programmiersprachen enthalten.

Assemblierer erstellen Konfigurationen von RePST, die dem Benutzer verschiedene Slicing- oder Chopping-Verfahren für konkrete Programmiersprachen anbieten. Dazu wählt der Assemblierer aus einer Menge von Komponenten diejenigen aus, welche die gewünschten Verfahren und Programmiersprachen unterstützen.

Benutzer können sich dann mit Hilfe von RePST eine Slice oder einen Chop von Quellcode berechnen lassen. Dabei kann der Benutzer zwischen verschiedenen Verfahren zur Slice- oder Chop-Berechnung wählen:

- statisches Rückwärtsslicen,
- statisches Vorwärtsslicen,
- dynamisches Rückwärtsslicen,
- dynamisches Vorwärtsslicen,
- statisches Choppen oder
- dynamisches Choppen.

Diese Situation wird in Abb. 4.1 in einem *Anwendungsfalldiagramm* dargestellt. Da in dieser Diplomarbeit exemplarisch ein konkretes Program Slicing Tool für die Programmiersprache C geschrieben wird, vertritt der Anwendungsfall „In C-Quellcode Slice oder Chop berechnen“ die Menge aller Anwendungsfälle für imperative Programmiersprachen, wie zum Beispiel Java, C++, Pascal usw.

4.2. Anforderungen

In diesem Abschnitt werden die *Anforderungen an RePST* bzw. seine Komponenten definiert. Ein großer Teil dieser Anforderungen, insbesondere der funktionalen Anforderungen, ist aus den „Spezifikationen der Dienste“ in [Sch07] abgeleitet.

Die Anforderungen sind in sechs Kategorien eingeteilt: es gibt funktionale Anforderungen, technische Anforderungen, Anforderungen an die Architektur, Qualitätsanforderungen, Anforderungen an die Benutzerschnittstelle und Anforderungen an die Dokumentation. Die Gewichtung der Anforderungen wird mit Hilfe der Modalverben: „muss“, „soll“ und „kann“ vorgenommen. Dabei bedeutet „muss“, dass eine Anforderung zwingend umgesetzt wird und somit Pflicht ist. „soll“ hat die zweithöchste Priorität. Die niedrigste Priorität hat „kann“; es ist ein Vorschlag, der nur als optional gilt.

Die folgende Anforderungsliste beschreibt eine idealisierte Version, die vor dem eigentlichen Entwurf erstellt wurde.

4.2.1. Funktionale Anforderungen

Funktionale Anforderungen an RePST

1. Alle unmarkierten Dienste aus den Tabellen 4.1 und 4.2 müssen durch Komponenten realisiert werden.

2. Alle mit * markierten Dienste aus den Tabellen 4.1 und 4.2 können durch Komponenten realisiert werden.

3. RePST muss so konfigurierbar sein, dass es beliebige Slicing- oder Chopping-Verfahren verwenden kann, sofern diese durch entsprechende Komponenten realisiert werden.

4. Jede Komponente muss die Funktionalität laut den Tabellen 4.1 und 4.2 des ihr zugeordneten Dienstes umsetzen, sofern sie realisiert wird.

5. RePST muss als Eingabe eine Quellcode-Datei akzeptieren.

6. RePST muss als Ausgabe eine Quellcode-Datei liefern, die den berechneten Slice oder Chop enthält.

7. RePST und dessen Komponenten müssen anderen Anwendungen ein Application Programming Interface (API) bereitstellen.

Dienst	**Funktion**
ProgramSlicing	berechnet gesliceten Programmcode
preprocessProgram	berechnet aus Programmcode einen ESDG
computeAbstractSyntaxGraph	berechnet aus Programmcode einen ASG
compute-AugmentedControlFlowGraphs	berechnet aus einem ASG einen ACFG
computePointsToGraphs	berechnet aus einem ASG einen PG
computeExtendedCallGraph	berechnet aus einem ASG und PGs einen ECG
computeCallGraph	berechnet aus einem ASG und PGs einen CG
computeDefUseInformation	berechnet aus einem ASG, PGs und einem EG einen ECG
computeESDG	berechnet aus einem ASG, einem ECG, PGs und ACFGs einen ESDG
computeBasicESDG	berechnet aus einen ASG und einem ECG einen ESDG mit unvollständigen Kanten
computeInterMethodEdges	ergänzt einen ESDG mit unvollständigen Kanten um Aufruf-, Parameter-in- und Parameter-out-Kanten
computeControlDependenceEdges	ergänzt einen ESDG mit unvollständigen Kanten um Kontrollkanten
computeDataFlowEdges	ergänzt einen ESDG mit unvollständigen Kanten um Datenfluss- und Deklarationskanten
computeSummaryEdges	ergänzt einen ESDG mit unvollständigen Kanten um transitive Kanten

Tabelle 4.1.: Tabellarischer Überblick über die Dienste und deren Funktion. (Teil 1)

Funktionale Anforderungen an das Program Slicing Tool für C

1. Alle unmarkierten Dienste aus den Tabellen 4.1 und 4.2 müssen durch Komponenten realisiert werden, die Slicing oder Chopping von C-Programmen durchführen.

2. Alle mit * markierten Dienste aus den Tabellen 4.1 und 4.2 können durch Komponenten realisiert werden, die Slicing oder Chopping von C-Programmen durchführen.

Dienst	**Funktion**
sliceProgram	markiert in einem ESDG den Slice oder den Chop
markESDG	markiert in einem ESDG das Slicing oder Chopping Kriterium und eventuell die Trace
computeStaticBackwardSlice*	berechnet die statische Rückwärtsslice
computeStaticForwardSlice*	berechnet die statische Vorwärtsslice
computeDynamicBackwardSlice*	berechnet die dynamische Rückwärtsslice
computeDynamicForwardSlice*	berechnet die dynamische Vorwärtsslice
computeStaticChop*	berechnet den statischen Chop
computeDynamicChop*	berechnet den dynamischen Chop
computeExecutableBackwardSlice*	berechnet die ausführbare Rückwärtsslice
convertESDGToCode	wandelt die innerhalb eines ESDG als Slice-zugehörig markierten Knoten in Programmcode um.

Tabelle 4.2.: Tabellarischer Überblick über die Dienste und deren Funktion. (Teil 2)

3. Das Program Slicing Tool für C muss so konfigurierbar sein, dass es beliebige Slicing- oder Chopping-Verfahren verwenden kann, sofern diese durch entsprechende Komponenten realisiert werden.

4. Jede Komponente für Slicing oder Chopping von C-Programmen muss die Funktionalität laut den Tabellen 4.1 und 4.2 des ihr zugeordneten Dienstes umsetzen, sofern sie realisiert wird.

5. Das Program Slicing Tool muss als Eingabe eine „c“-Datei akzeptieren.

6. Das Program Slicing Tool muss als Ausgabe eine „c“-Datei liefern, die den berechneten Slice oder Chop enthält.

7. Das Program Slicing Tool für C muss mit Hilfe der Konsole nutzbar sein.

8. Für das Program Slicing Tool für C kann eine grafische Benutzungsoberfläche erstellt werden.

9. Das Program Slicing Tool für C und dessen Komponenten müssen anderen Anwendungen ein Application Programming Interface (API) bereitstellen.

4.2.2. Technische Anforderungen

1. Die Komponenten müssen in Java 6 geschrieben werden.
2. Die Komponenten müssen zur Darstellung von Graphen die JGraLab-Bibliothek verwenden.
3. Die Komponenten sollen eine public-Methode zur Erfüllung ihres Dienstes anbieten.
4. Falls erforderlich sollen die Komponenten durch public-Methoden konfektionierbar sein.
5. Die Komponenten sollen als Ein- und Ausgabe Graph-Instanzen, „tg“-Dateien oder Programmcode-Dateien verwenden.
6. Die Komponenten sollen mit Hilfe von Eclipse[1] entwickelt werden.
7. Zum Parsen von C-Programmen innerhalb der Komponente zur Realisierung des Dienstes „computeAbstractSyntaxGraph“ soll der GUPRO[2] C-Parser verwendet werden.
8. Das Schema soll automatisch generiert werden.
9. Automatisches Erstellen soll durch das Build-Tool Apache Ant[3] umgesetzt werden.
10. Die Ant-Datei kann der Common Apache Ant-Datei für GUPRO Projekte [Fal07] entsprechen.
11. Zum Testen der Komponenten soll JUnit[4] verwendet werden.

4.2.3. Anforderungen an die Architektur

1. Jeder Dienst muss durch eine Komponente realisiert werden.
2. Komponenten, die externe Verzeichnisse und Dateien benötigen, müssen konfigurierbar sein.

[1] `http://www.eclipse.org`
[2] `http://www.uni-koblenz.de/FB4/Contrib/GUPRO/Site/Home`
[3] `http://ant.apache.org/`
[4] `http://www.junit.org/`

3. Komponenten, die keine externen Verzeichnisse und Dateien benötigen, sollen möglichst nur eine public-Methode als Schnittstelle anbieten.

4. Komponenten, die Program Slicing anhand des Referenzschema durchführen, müssen ausschließlich die RePSS-Graphklasse verwenden.

5. Komponenten, die Program Slicing anhand des Referenzschemas durchführen, können abstrakt sein.

6. Komponenten, die Program Slicing anhand des C-Schemas durchführen, müssen konkret sein.

7. Komponenten, die Program Slicing anhand des C-Schemas durchführen, müssen neben der RePSS-Graphklasse nur die C-Graphklasse verwenden.

8. Zwischen den einzelnen Komponenten soll eine möglichst lose Kopplung bestehen.

9. Komponenten sollen ohne Architektur-Kenntnisse benutzbar sein.

4.2.4. Qualitätsanforderungen

1. Die Komponenten sollen ihre Berechnungen stabil ausführen.

2. Die Komponenten sollen eventuell auftretende, nicht innerhalb der Komponente zu behandelnde Exceptions werfen.

3. Die Komponenten können dazu in der Lage sein, Exceptions selbstständig zu behandeln oder Fehlermeldungen direkt auf der Konsole auszugeben.

4. Die Methoden innerhalb einer Komponente sollen mit JavaDoc kommentiert sein.

5. Der Quellcode kann weitere Kommentare enthalten.

6. Die Komponenten sollen nach eventueller Anpassung in anderen Werkzeugen wiederverwendbar sein.

7. Auf dem Referenzschema basierende Komponenten sollen mit möglichst wenig Aufwand zum Slicen von Programmcode in anderen Programmiersprachen nutzbar sein.

8. Auf dem C-Schema basierende Komponenten können dazu in der Lage sein, mehr als 80% des Quellcodes der auf dem Referenzschema basierenden Superkomponente zu nutzen.

9. Die Komponenten sollen erweiterbar sein.

10. Die Komponenten sollen austauschbar sein.

11. Die Komponenten können dazu in der Lage sein, möglichst performant zu arbeiten.

12. Komponenten zum konkreten Program Slicing für C-Programme sollen durch Unit-Tests getestet werden.

13. Verwendete Fremdsoftware kann dazu in der Lage sein, plattformunabhängig zu sein.

14. Verwendete externe Bibliotheken können plattformunabhängig zu sein.

15. Komponenten ohne Fremdsoftware sollen plattformunabhängig sein.

4.2.5. Anforderungen an die Benutzerschnittstelle

1. Der Benutzer muss mit Hilfe der Konsole das Program Slicing Tool für C-Programme bedienen können.

2. Es kann eine grafische Benutzungsoberfläche entwickelt werden.

3. Die grafische Benutzungsoberfläche kann zum Slicen eines C-Programms genutzt werden.

4. Die grafische Benutzungsoberfläche kann zum Auswählen einer „c“-Datei genutzt werden.

5. Die grafische Benutzungsoberfläche kann dem Benutzer Fehlermeldungen anzeigen.

4.2.6. Anforderungen an die Dokumentation

1. Die Entwicklung von RePST muss in der zugehörigen Diplomarbeit dokumentiert werden.

2. Die Dokumentation innerhalb der Diplomarbeit kann in deutscher Sprache verfasst werden.

3. Die Dokumentation des Quellcodes muss in englischer Sprache formuliert sein.

4. Fehlermeldungen müssen in englischer Sprache sein.

5. Die Struktur der einzelnen Komponenten soll durch Diagramme veranschaulicht werden.

6. Die Architektur von RePST soll durch Diagramme gezeigt werden.

5. Architektur

Die Architektur-Beschreibung der größten Komponenten innerhalb des Program Slicing Tools geschieht in diesem Kapitel. Die größten Komponenten sind RePST selbst und seine beiden Komponenten ProgramPreprocessor und ProgramSlicer, da sich diese Komponenten aus einer Vielzahl kleinerer Komponenten zusammensetzen. Der detaillierte Aufbau dieser kleineren Komponenten wird in Kapitel 6 genauer beschrieben. Ein grober Überblick zur Architektur von RePST befindet sich in Abb. 5.1.

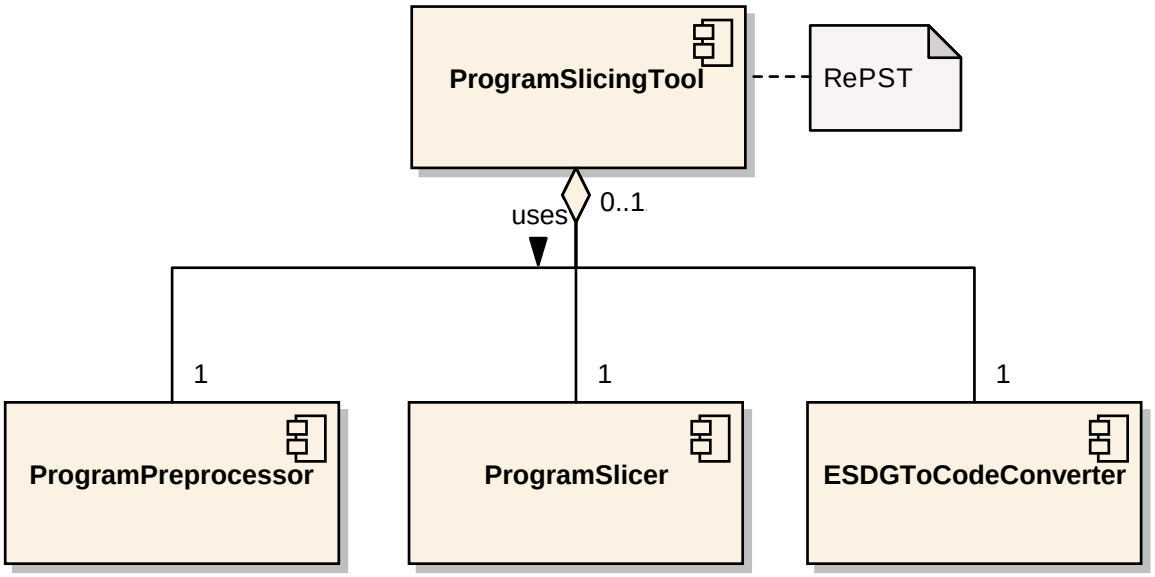

Abbildung 5.1.: Komponentendiagramm: grober Architekturüberblick

5.1. Architektur-Muster

Die Architektur des Program Slicing Tools basiert auf dem Ansatz der *Komponentenorientierung*. Durch die komponentenbasierte Software-Entwicklung [Bal00] werden im Rahmen dieser Arbeit Komponenten geschaffen, die nicht nur im Kontext des Program Slicings Verwendung finden, sondern auch in anderen Projekten zum Einsatz kommen könnten. So ist es

zum Beispiel durchaus denkbar, dass die hier entwickelten Komponenten in anderen Projekten im Kontext von GUPRO [EGSW98] zum Einsatz kommen.

Call-Return *Call-Return* [SG96] ist ein *Architektur-Muster*, welches die Aufrufbeziehungen zwischen Teilsystemen beschreibt. Somit wird es eingesetzt, um die Problematik der Bereitstellung von benötigten Diensten durch Teilsysteme zu behandeln. Die Idee besteht darin, dass das aktuell aktive Teilsystem die Kontrolle an ein anderes Teilsystem weitergeben (Call) kann, um nach der Rückgabe (Return) der Kontrolle weiterarbeiten zu können. Dieses Weiter- und Zurückgeben kann sich hierarchisch über mehrere Teilsysteme erstrecken.

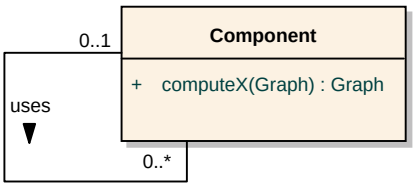

Abbildung 5.2.: Viewpoint zum Architektur-Muster: Call-Return

Abb. 5.2 zeigt den *Viewpoint* zum Architektur-Muster Call-Return. Der Viewpoint eines Architektur-Musters wird durch ein Schema festgelegt. Weiterführende Erläuterungen zu diesem Architektur-Muster findet der Leser in [SG96].

Um den komponentenbasierten Ansatz durch die Architektur zu unterstützen, wird für RePST Call-Return als Architektur-Muster gewählt. Dabei entsprechen die Komponenten von RePST den hierarchisch aufgebauten Teilsystemen von Call-Return. Die Methode `computeX` nach eventueller Anpassung ihres Rückgabetyps oder ihrer Parameter vertritt die public-Methoden jeder Komponente für Berechnung an einem Graphen.

Den Klassendiagrammen in den Abb. 5.3, 5.4, 5.5 und 5.6 liegt der Viewpoint des Call-Return-Musters zugrunde. In diesen Diagrammen gibt es jedoch noch weitere Details, die später in diesem Kapitel erläutert werden.

5.2. Architektur Eigenschaften von RePST

Als Grundlage für die Architektur dient die Beschreibung der Dienste durch *Datenflussdiagramme* in [Sch07]. Daraus ergeben sich folgende Eigenschaften der Architektur:

Dienst	**RePST-Komponente**	**Komponente für C**
programSlicing	ProgramSlicingTool	-
preprocessProgram	ProgramPreprocessor	-
computeAbstractSyntaxGraph	ASGComputer	ASGComputerForC
computeAugmented-ControlFlowGraphs	ACFGComputer	ACFGComputerForC
computePointsToGraphs	PGComputer	-
computeExtendedCallGraph	ECGComputer	-
computeCallGraph	CGComputer	CGComputerForC
computeDefUseInformation	DefUseInfComputer	DefUseInf-ComputerForC
computeExtended-SystemDependenceGraph	ESDGComputer	-
computeBasicESDG	BasicESDGComputer	BasicESDG-ComputerForC
computeInterMethodEdges	IntMetEdgesComputer	-
compute-ControlDependenceEdges	ConDepEdgesComputer	-
computeDataFlowEdges	DataFlowEdgesComputer	DataFlowEdges-ComputerForC
computeSummaryEdges	SumEdgesComputer	-
sliceProgram	ProgramSlicer	-
markESDG	ESDGMarker	ESDGMarkerForC
computeStaticBackwardSlice	StaticBackwardSliceComputer	-
computeStaticForwardSlice	StaticForwardSliceComputer	-
computeDynamic-BackwardSlice	DynamicBackward-SliceComputer	-
computeDynamic-ForwardSlice	DynamicForward-SliceComputer	-
computeStaticChop	StaticChopComputer	-
computeDynamicChop	DynamicChopComputer	-
computeExecutable-BackwardSlice	ExecutableBackward-SliceComputer	-
convertESDGToCode	ESDGToCodeConverter	ESDGToCode-ConverterForC

Tabelle 5.1.: Tabellarischer Überblick über die Dienste und deren Umsetzung durch RePST-Komponenten und programmiersprachenabhängige Komponenten.

- Falls möglich wird jeder Dienst[1] durch eine einzelne RePST-Komponente umgesetzt. Andernfalls wird jeder Dienst durch eine eventuell abstrakte RePST-Komponente und pro Programmiersprache eine weitere konkrete Komponente umgesetzt. Tabelle 5.1 listet die Dienste und die Komponenten auf, welche die Dienste umsetzen.

- Aus dem Namen eines Dienstes ergeben sich die Bezeichnung der Komponente und der public-Methode, welche die Funktion der Komponente ausführt.

- Die Schnittstellen ergeben sich durch den Fluss der Daten. Das heißt, dass Argumente und Rückgabewert der public-Methoden den Ein- und Ausgaben der Dienste entsprechen. Dies ist in Tabelle 5.2 notiert.

- Aufgrund der Zerlegung von größeren Diensten in kleinere oder atomare Dienste ergeben sich „uses“-Beziehungen zwischen den Komponenten.

Die Umsetzung der Dienste durch *RePST-Komponenten* und *programmiersprachenabhängige Komponenten* ergibt sich durch die Nutzung von RePSS. Das Slicen von Programmen, die dem Schema einer beliebigen, imperativen Programmiersprache entsprechen, geschieht anhand des Referenzschemas und, falls erforderlich, anhand des Schemas der Programmiersprache. Falls das Schema dieser Programmiersprache zu stark von RePSS abweicht und die Ausführung des Dienstes zu sehr vom Schema der Programmiersprache geprägt wird, so benötigt die Komponente zur Realisierung des Dienstes eine Spezialisierung. Die Umsetzung dieser Spezialisierung geschieht in Form einer zusätzlichen, konkreten Komponente für die entsprechende Programmiersprache. Diese Komponente arbeitet direkt auf dem Schema der Programmiersprache. Daraus folgt:

- Dienste, deren Ausführung ausschließlich auf RePSS basiert, können durch eine konkrete Komponente für alle Programmiersprachen umgesetzt werden.

- Dienste, deren Ausführung zusätzlich das Schema einer Programmiersprache erfordern, müssen durch eine eventuell abstrakte Komponente und pro Programmiersprache eine konkrete Komponente realisiert werden. Hierbei kommt das Design Pattern *Template Method* [GHJV95] zum Einsatz. Der Algorithmus, der den Dienst der Komponente erfüllt, wird in mehrere teilweise primitive Methoden aufgeteilt. Diese primitiven Methoden werden dann von der konkreten, programmiersprachenabhängigen

[1] Die in [Sch07] verwendeten Dienste markSlicingCriterion, markChoppingCriterion und markTrace werden in dem Dienst markESDG zusammengefasst und dort als Methoden realisiert. Außerdem werden die Dienste computeStaticSlice und computeDynamicSlice aus [Sch07] nicht realisiert, da sie während der Entwicklung des Program Slicing Tools als abstrakte Oberklasse `SliceOrChopComputer` identifiziert wurden.

RePST-Komponente	**Eingabe**	**Ausgabe**
ProgramSlicingTool	Quellcode-Datei	Quellcode-Datei
ProgramPreprocessor	Quellcode-Datei	ESDG, ACFGs
ASGComputer	Quellcode-Datei	ASG
ACFGComputer	ASG	ACFGs
PGComputer	ASG	PGs
ECGComputer	ASG, PGs	ECG
CGComputer	ASG, PGs	CG
DefUseInfComputer	CG, ASG, PGs	ECG
ESDGComputer	ASG, ECG, PGs, ACFGs	ESDG
BasicESDGComputer	ASG, ECG	ESDG'
IntMetEdgesComputer	ESDG[unvoll.], ASG, ECG	ESDG[unvoll.]
ConDepEdgesComputer	ESDG[unvoll.], ACFGs	ESDG[unvoll.]
DataFlowEdgesComputer	ESDG[unvoll.], ASG, ACFGs, PGs	ESDG[unvoll.]
SumEdgesComputer	ESDG[unvoll.]	ESDG
ProgramSlicer	ESDG, ACFGs, (Tr.), Sl./Ch. Krit.	ESDG [Sl./Ch. mark.]
ESDGMarker	ESDG, ACFGs, (Tr.), Sl./Ch. Krit.	ESDG [Sl./Ch. Krit. mark.]
StaticBackwardSliceCo.	ESDG[Sl. Krit. mark.]	ESDG[Sl. mark.]
StaticForwardSliceCo.	ESDG[Sl. Krit. mark.]	ESDG[Sl. mark.]
DynamicBackwardSliceCo.	ESDG[Sl. Krit.&Tr. mark.]	ESDG[Sl. mark.]
DynamicForwardSliceCo.	ESDG[Sl. Krit.&Tr. mark.]	ESDG[Sl. mark.]
StaticChopComputer	ESDG[Ch. Krit. mark.]	ESDG[Ch. mark.]
DynamicChopComputer	ESDG[Ch. Krit.&Tr. mark.]	ESDG[Ch. mark.]
ExecutableBackwardSliceCo.	ESDG[Sl. mark.]	ESDG[Sl. mark.]
ESDGToCodeConverter	ESDG[Sl./Ch. mark.]	Quellcode-Datei

Tabelle 5.2.: Tabellarischer Überblick über die Ein- und Ausgaben der Komponenten.

Komponente überschrieben, so dass die entsprechende Berechnung anhand des Programmiersprachenschemas durchgeführt wird.

Da es keinen Parser für EL gibt, werden RePST-Komponenten teilweise nur durch abstrakte Komponenten umgesetzt.

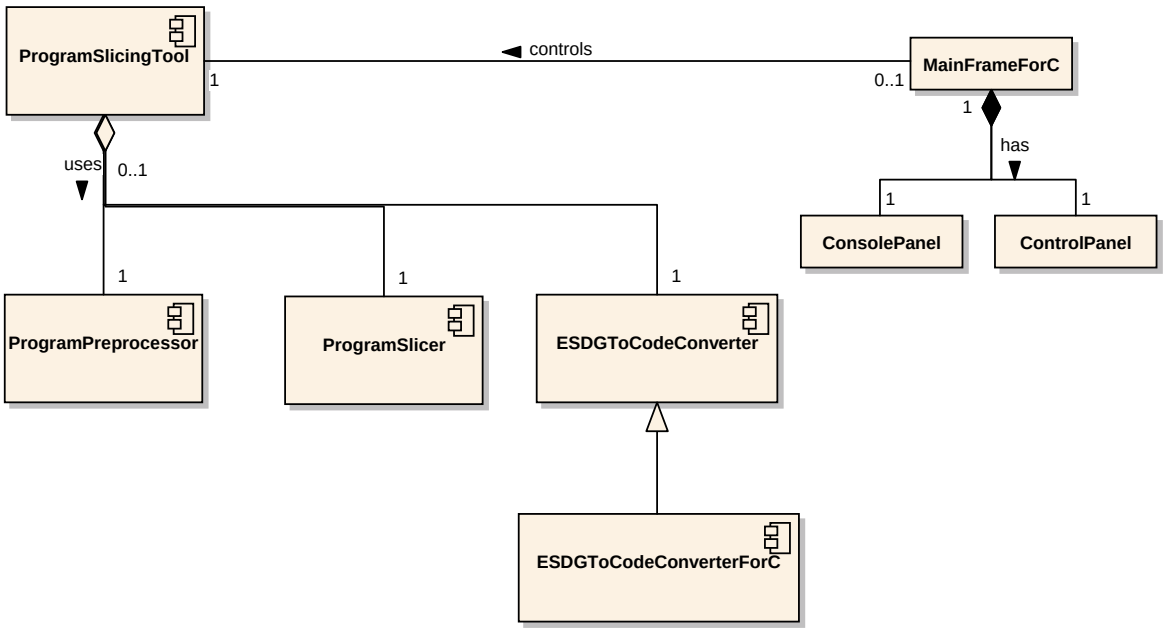

Abbildung 5.3.: Hauptkomponenten und grafischer Client des Program Slicing Tools

In Abb. 5.3 sind die größten Komponenten und, falls erforderlich, ihre Spezialisierungen zum Slicen von C-Programmen zu sehen. So gibt es zum Beispiel die RePST-Komponente ESDGToCodeConverter, welche die Konvertierung eines ESDG in Quellcode vornimmt. Da diese Komponente laut Tabelle 5.1 eine programmiersprachenabhängige Komponente ist, muss sie spezialisiert werden. Diese Spezialisierung heißt hier exemplarisch ESDGToCodeConverterForC. Programmiersprachenunabhängige Komponenten, wie zum Beispiel ProgramPreprocessor, werden nicht spezialisiert. Dieses Prinzip wird analog auf alle anderen Komponenten angewendet.

Bisher wurde oft davon gesprochen, dass sich Komponenten aus mehreren anderen Komponenten zusammensetzen können. Diese Aussage bezieht sich genau genommen auf die Funktionalität der Komponenten. Das heißt, dass sich die Funktionalität einer größeren Komponente aus den Funktionalitäten mehrerer kleinerer Komponenten zusammensetzt. Dieser Sachverhalt ist in Abb. 5.3 und in allen folgenden Abb. durch „uses“-Assoziationen für die Komponente ProgramSlicingTool dargestellt. Ihre Funktionalität ergibt sich also durch die Benutzung der Komponenten ProgramPreprocessor, ProgramSlicer und ESDGToCodeConverter.

Tatsächlich sind Komponenten, die andere Komponenten verwenden, im Kontext von RePST *Fassaden*. Das bedeutet, dass zum Beispiel die Komponente ProgramSlicingTool Fassade für die Komponenten PreprocessProgram, ProgramSlicer und ESDGToCodeConverter ist. Durch die Anwendung des Facade-Patterns [GHJV95] muss ein Client lediglich die public-Methoden der Komponente ProgramSlicingTool aufrufen, um den Dienst ProgramSlicing auszuführen.

In allen Klassendiagrammen dieser Diplomarbeit werden Fassaden durch den Stereotyp `facade` gekennzeichnet.

5.3. Architektur der Komponente ProgramSlicingTool

Alle Komponenten sind Spezialisierung der Superklasse `ProgramSlicingComponent`. Dies ist nur in Abb. 5.4 gezeigt und wird in den folgenden Diagrammen ausgeblendet.

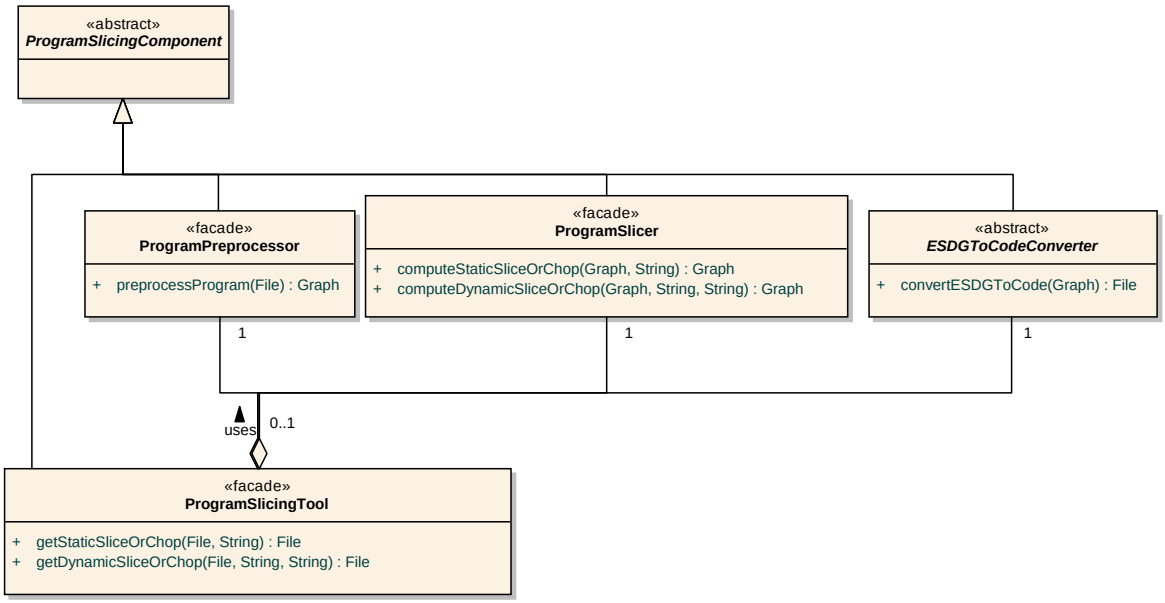

Abbildung 5.4.: Klassendiagramm: RePST

Außerdem sind in Abb. 5.4 die public-Methoden der Komponenten ProgramSlicingTool, ProgramPreprocessor, ProgramSlicer und ESDGToCodeConverter zu sehen. Die Methode `getStaticSliceOrChop` bzw. `getDynamicSliceOrChop` der Komponente ProgramSlicingTool berechnet die statische bzw. dynamische Slice oder Chop der übergebenen Quellcode-Datei, indem sie die Methoden `preprocessProgram`, `computeStaticSliceOrChop` bzw. `computeDynamicSliceOrChop` und `convertESDGToCode`

der entsprechenden Komponenten je nach Bedarf ausführt. So wird `preprocessProgram` zum Beispiel nur einmal ausgeführt, falls der Benutzer die statische und dynamische Slice für denselben Quellcode berechnen möchte.

Weiterführende Details befinden sich in Abschnitt 6.1.

5.4. Architektur der Komponente ProgramPreprocessor

Die Komponente ProgramPreprocessor ist eine der größten Komponenten, weil sie sich aus fünf anderen Komponenten zusammensetzt, die teilweise selbst aus mehreren anderen Komponenten bestehen. In Abb. 5.5 ist dies mit Hilfe von „uses“-Assoziationen dargestellt. Durch die „uses“-Beziehungen lassen sich die Komponenten ProgramPreprocessor, ECGComputer und ESDGComputer als Fassaden identifizieren.

Zusätzlich zeigt Abb. 5.5 die public-Methoden der einzelnen Komponenten bzw. Klassen, welche die Funktionalität laut Tabelle 4.1 des entsprechenden Dienstes ausführen.

Weitere Informationen zu dieser Komponente befinden sich in Abschnitt 6.2.

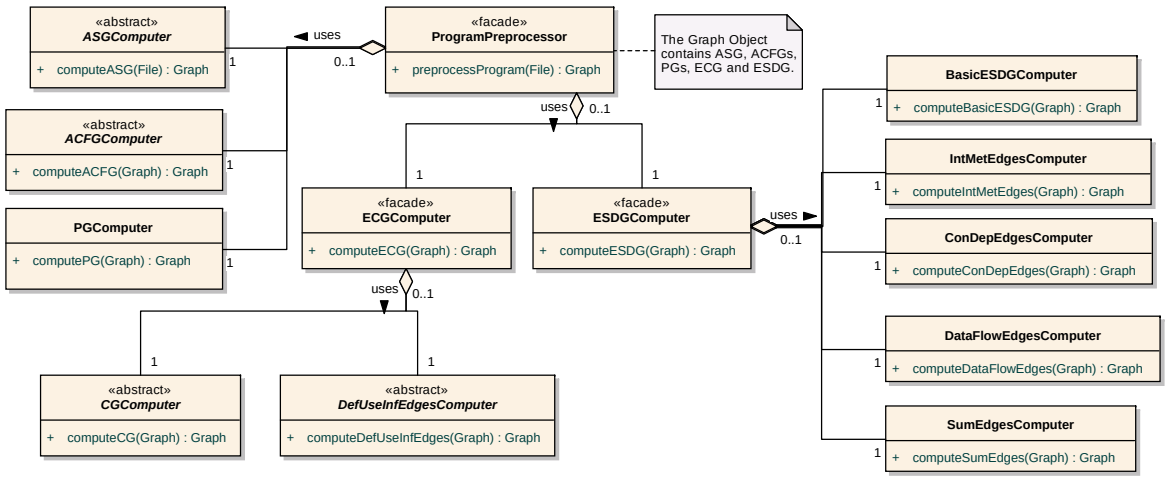

Abbildung 5.5.: Klassendiagramm: ProgramPreprocessor

5.5. Architektur der Komponente ProgramSlicer

Der Architekturstil der Komponente ProgramSlicer ist vergleichbar mit dem Architekturstil der Komponente ProgramPreprocessor. Beide Architekturen verwenden „uses“-

Beziehungen, um die Call-Return-Struktur zu verdeutlichen. So erkennt man in Abb. 5.6, dass die Komponente ProgramSlicer Fassade für andere Komponenten ist.

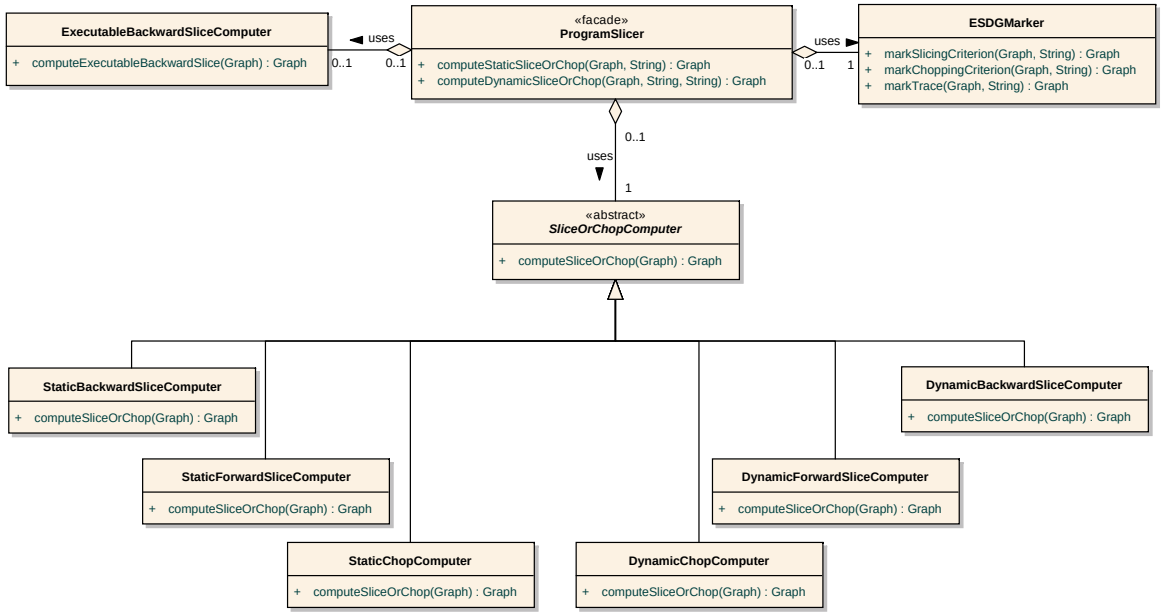

Abbildung 5.6.: Klassendiagramm: ProgramSlicer

Die Komponente ProgramSlicer bietet die beiden public-Methoden `computeStaticSliceOrChop` und `computeDynamicSliceOrChop`, um durch statische oder dynamische Verfahren eine Slice oder einen Chop zu berechnen. Es gibt acht verschiedene Arten von Slices und Chops, deren Berechnung durch die folgenden acht Verfahren geschieht:

1. Statisches Rückwärtsslicen wird durch die Komponente StaticBackwardSliceComputer ausgeführt.

2. Statisches Rückwärtsslicen mit ausführbarer Rückwärtsslice wird durch die Komponenten StaticBackwardSliceComputer und ExecutableBackwardSliceComputer ausgeführt.

3. Statisches Vorwärtsslicen wird durch die Komponente StaticForwardSliceComputer ausgeführt.

4. Dynamisches Rückwärtsslicen wird durch die Komponente DynamicBackwardSliceComputer ausgeführt.

5. Dynamisches Rückwärtsslicen mit ausführbarer Rückwärtsslice wird durch die Komponenten DynamicBackwardSliceComputer und ExecutableBackwardSliceComputer ausgeführt.

6. Dynamisches Vorwärtsslicen wird durch die Komponente DynamicForwardSliceComputer ausgeführt.

7. Statisches Choppen wird durch die Komponente StaticChopComputer ausgeführt.

8. Dynamisches Choppen wird durch die Komponente DynamicChopComputer ausgeführt.

Für alle Verfahren wird außerdem die Komponente ESDGMarker benötigt.

Zur Ausführung eines statischen Verfahrens gibt es die Methode `computeStaticSliceOrChop`, die aufgrund dynamischer Bindung eine `SliceOrChopComputer`-Instanz benutzt, welche ein statisches Verfahren implementiert. Analog dazu gibt es die Methode `computeDynamicSliceOrChop`, deren Ausführung auf eine Komponente zugreift, welche ein dynamisches Verfahren implementiert. Dabei ist zu beachten, dass sowohl das Slicing- als auch das Chopping-Kriterium durch einen String in den Argumenten der Methoden vertreten werden. Beide Methoden benötigen neben dem String für das Slicing- bzw. Chopping-Kriterium eine Graph-Instanz als weiteres Argument. Dynamische Verfahren benötigen als drittes Argument einen weiteren String, der die Trace repräsentiert.

Weiterführendes findet der Leser in Abschnitt 6.15.

5.6. Konfiguration von RePST

Die *Konfiguration* von RePST wird mit Hilfe des Design Patterns *FactoryMethod* [GHJV95] umgesetzt. Dazu gibt es die abstrakte Klasse `ProgramSlicingToolFactory`, deren Factory-Methode `createProgramSlicingTool` für die Erzeugung von `ProgramSlicingTool`-Objekten zuständig ist. Der Assemblierer schreibt nun für jede zum Slicen oder Choppen gewünschte Quellcode-Programmiersprache eine Unterklasse von `ProgramSlicingToolFactory`, welche die Factory-Methode überschreibt. In Abb. 5.7 ist dies exemplarisch mit der Unterklasse `ProgramSlicingToolForCFactory` gezeigt.

In der überschriebenen Factory-Methode `createProgramSlicingTool` müssen die zur gewünschten Programmiersprache passenden Komponenten ausgewählt werden. Außerdem wird das gewünschte Slicing- oder Chopping-Verfahren durch den Parameter dieser Methode festgelegt.

Ein Client, der das Program Slicing Tool nutzen möchte, instanziert die zum Quellcode passende `ProgramSlicingToolFactory`-Unterklasse. Anschließend ruft er die `createProgramSlicingTool`-Methode des `ProgramSlicingToolFactory`-Objektes auf und übergibt ihr die zum gewünschten Verfahren passende Konstante als Parameter. Die Konstanten, welche die möglichen Verfahren festlegen, sind in der abstrakten Oberklasse `ProgramSlicingToolFactory` definiert, siehe dazu Abb. 5.7.

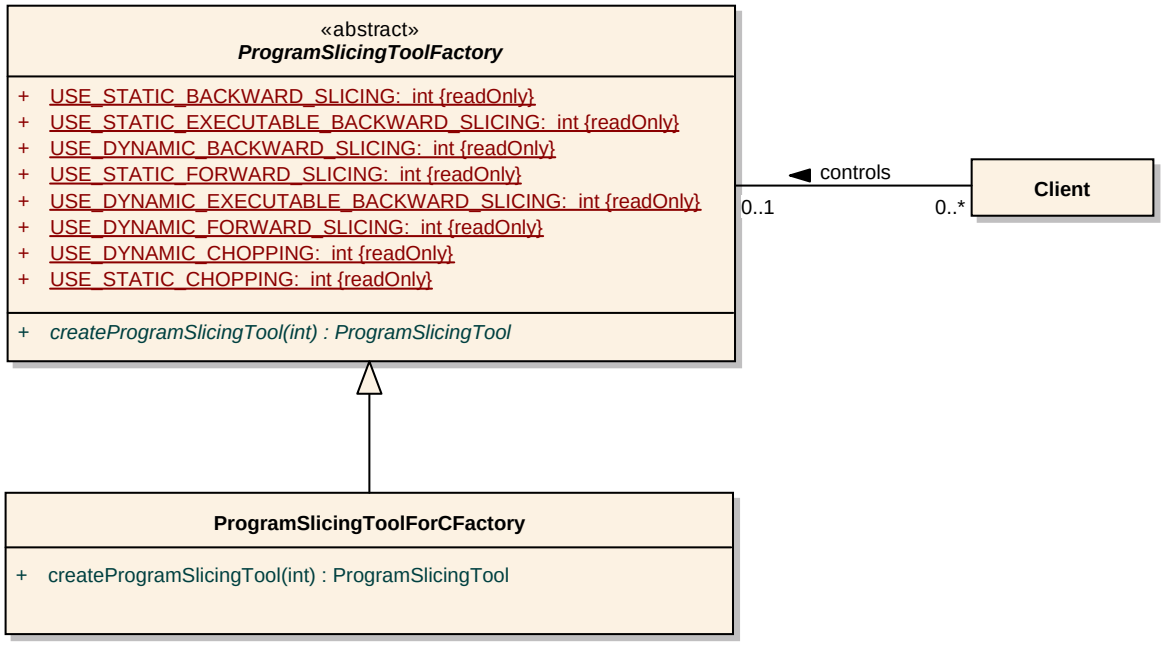

Abbildung 5.7.: Klassendiagramm: Factory zur Konfiguration von RePST.

5.7. Verwendung gemeinsamer Komponenten

Zur Vermeidung von redundantem Quellcode gibt es die Komponente SharedComponent. Diese enthält Hilfsmethoden, die von mehr als einer anderen Komponente verwendet werden. Das ist zum Beispiel die Methode, welche zur Erkennung von globalen Variablen bzw. Attributen im ASG benötigt wird.

Damit andere Komponenten auf die SharedComponent zugreifen können, müssen sie Unterklasse der abstrakten Klasse `ProgramSlicingComponent` sein. Dadurch erben sie das Attribut `sharedComponent`, welches für jede Programmiersprache die passende `SharedComponent`-Instanz referenziert, sofern sie während der Konfiguration korrekt gesetzt wurde.

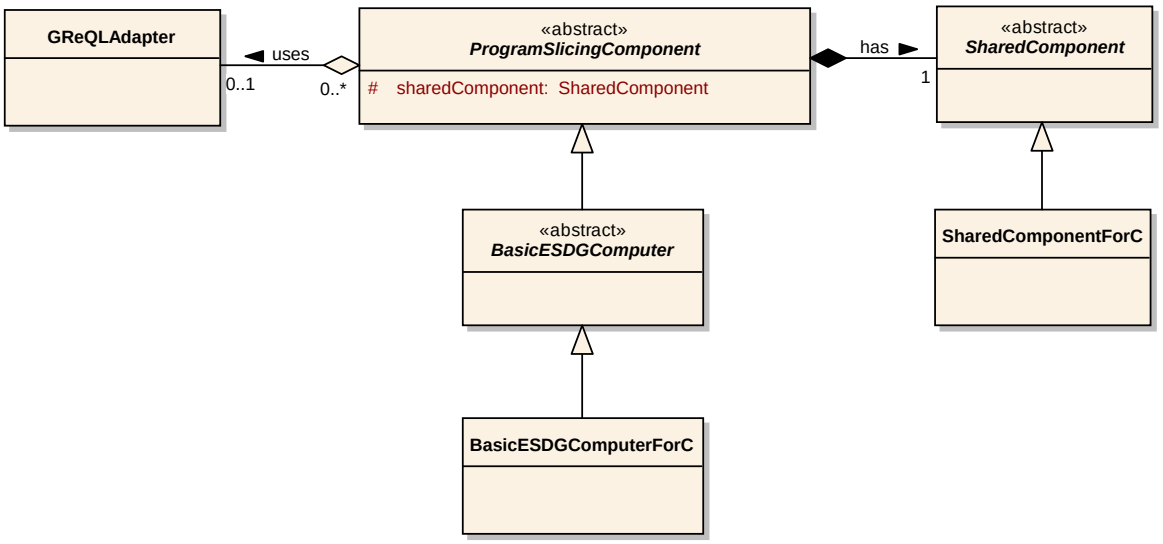

Abbildung 5.8.: Klassendiagramm: Verwendung der gemeinsamen Komponente

Abb. 5.8 zeigt diesen Sachverhalt am Beispiel des BasicESDGComputers. Die Komponente `BasicESDGComputer` hat `ProgramSlicingComponent` als Superklasse. Der ESDG für C wird unter anderem mit Hilfe der Komponente `BasicESDGComputerForC` berechnet, welche Spezialisierung der Komponente `BasicESDGComputer` ist.

Während der Initialisierung des BasicESDGComputer muss innerhalb der Factory für C festgelegt werden, dass es sich um eine Instanz des `BasicESDGComputerForC` handelt, die eine `SharedComponentForC`-Instanz nutzt. Diese `SharedComponentForC`-Instanz enthält die gemeinsamen Methoden, die für Berechnungen an Graphen zur Beschreibung von C-Quellcode angepasst sind.

Damit alle Komponenten für ihre Berechnungen *GReQL2* verwenden können, gibt es die Klasse `GReQLAdapter`. Diese Klasse fungiert als *Adapter* [GHJV95] für die Klasse `GreqlEvaluator` [Bil06]. Da die Komponenten nicht parallel rechnen können und die Nutzung von GReQL2 während der Entwicklung Speicherprobleme (siehe dazu Kapitel 7.1) verursachte, wird die Klasse `GReQLAdapter` als *Singleton* [GHJV95] umgesetzt, um überflüssige `GReQLAdapter`-Instanzen einzusparen.

6. Realisierung der Komponenten

Dieses Kapitel beschreibt ausführlich die *Realisierung* der Komponenten zur Umsetzung der Dienste aus [Sch07]. Außerdem ergänzt es die *Spezifikation* aus [Sch07] um Aspekte bezüglich des Slicens anhand des Referenzschemas und des C-Schemas.

Zur Veranschaulichung werden die Komponenten durch *Klassendiagramme* dargestellt. Um einen besseren Überblick zu bieten, werden einige Details in den Klassendiagrammen ausgeblendet.

- Private Methoden und private Attribute werden nur dann dargestellt, wenn sie von zentraler Bedeutung sind, da die wesentlichen Methoden und Attribute aufgrund des intensiven Einsatzes von Vererbung protected oder public sind.
- Private Attribute, auf die per Getter- und Setter-Methode zugegriffen wird, werden dargestellt, damit auf die Darstellung von Getter- und Setter-Methoden verzichtet werden kann.
- Methoden, die in der Unterklasse überschrieben werden, weil hier eine andere Berechnung als in der Oberklasse nötig ist, werden in beiden Klassen angezeigt.
- Methoden, die in der Unterklasse nicht überschrieben werden, werden in den Unterklassen nicht aufgeführt, um deutlicher zu machen, dass hier die Funktionalität der Oberklasse vollständig übernommen wird.
- Konstruktoren werden nicht angezeigt.

6.1. ProgramSlicingTool

Funktion: Slicing oder Chopping einer „c“-Datei

Eingabe: „c“-Datei ohne Includings, Slicing- oder Chopping-Kriterium, optional Trace

Ausgabe: „c“-Datei mit Slice oder Chop

Fassade für: ProgramPreprocessor, ProgramSlicer, ESDGToCodeConverter

Das Program Slicing Tool bzw. der Dienst *programSlicing* wird durch die Komponente ProgramSlicingTool umgesetzt. Diese Komponente ermöglicht dem Benutzer je nach Konfiguration die statische oder dynamische Berechnung einer Slice oder eines Chops. Dazu bietet das API dieser Komponente die beiden public-Methoden `getStaticSliceOrChop` und `getDynamicSliceOrChop` an, welche als Parameter eine Quellcode-Datei sowie das Slicing- oder Chopping-Kriterium erfordern. Dynamische Berechnungen benötigen die Trace als zusätzlichen Parameter.

Zum Slicen oder Choppen führt die Klasse `ProgramSlicingTool` durch ihre public-Methoden die public-Methoden der Komponenten ProgramPreprocessor, ProgramSlicer und ESDGToCodeConverter der Reihe nach aus, da sie Fassade für diese drei Komponenten ist. Abb. 5.4 zeigt in einem Klassendiagramm die Komponente ProgramSlicingTool. Das Ergebnis des ProgramPreprocessor wird hierbei zwischengespeichert, damit derselbe Quellcode nicht mehrfach in einen ESDG überführt werden muss.

6.2. ProgramPreprocessor

Funktion: Umwandlung des C-Programmcodes in einen ESDG

Eingabe: „c“-Datei ohne Includings

Ausgabe: ESDG, ACFGSet, „i“-Datei

Fassade für: ASGComputer, ACFGComputer, PGComputer, ECGComputer, ESDGComputer

Zur Realisierung des Dienstes *preprocessProgram* existiert die Komponente ProgramPreprocessor. Diese Komponente bildet durch die Klasse `ProgramPreprocessor` eine Fassade für die Komponenten ASGComputer, ACFGComputer, PGComputer, ECGComputer und ESDGComputer. Die Umwandlung von Quellcode in einen erweiterten Systemabhängigkeitsgraph wird von der public-Methode `preprocessProgram` durchgeführt. Dazu ruft der ProgramPreprocessor die public-Methoden der Komponenten auf, für die er eine Fassade bildet. Das Klassendiagramm in Abb. 5.5 enthält den ProgramPreprocessor.

6.3. ASGComputer

Funktion: Umwandlung des C-Programmcodes in einen ASG, der dem C-Schema entspricht

Eingabe: „c“-Datei ohne Includings

Ausgabe: ASG, „i“-Datei

Die Komponente *ASGComputer* setzt den Dienst *computeAbstractSyntaxGraph* um. Zur Erzeugung des ASGs werden innerhalb dieser Komponente Compilerbautechniken verwendet, wie sie zum Beispiel in [ASU86] oder speziell für objektorientierte Sprachen in [RH98] beschrieben sind.

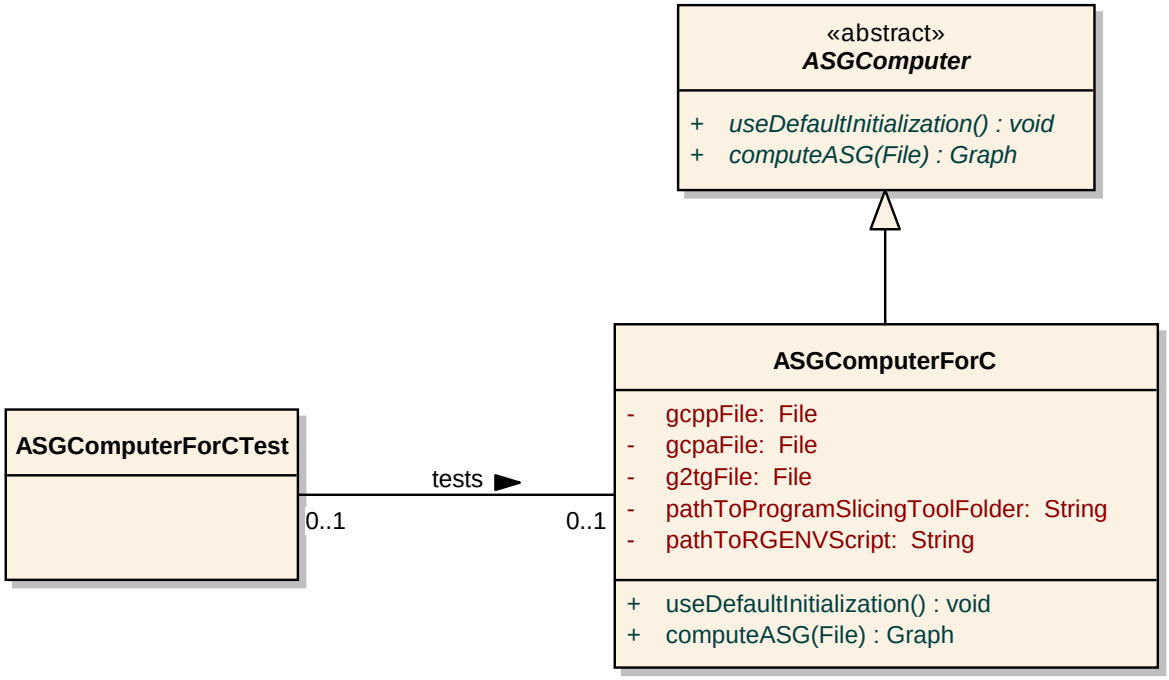

Abbildung 6.1.: Klassendiagramm: ASGComputer

Die Struktur dieser Komponente wird durch das Klassendiagramm in Abb. 6.1 beschrieben. Die abstrakte Klasse `ASGComputer` enthält nur die beiden abstrakten Methoden `useDefaultInitialization` zur Standardkonfiguration der Komponente und `computeASG` zur Berechnung des ASGs. Diese beiden Methoden müssen in spezielleren Komponenten überschrieben werden, da zur Berechnung des ASGs externe Parser nötig sind, die sich für jede Programmiersprache unterscheiden. Der Parameter der Funktion `computeASG` beschreibt den Pfad zur Quelldatei. Dazu wird eine Instanz der Klasse `File`

[Krü06] verwendet, da mit Hilfe dieser Klasse eine abstrakte, plattformunabhängige Beschreibung der Dateien und Verzeichnisse möglich ist.

Die Klasse `ASGComputerForC` repräsentiert die konkrete Komponente `ASGComputerForC` für die Programmiersprache C. Zur Erstellung eines ASGs, der dem C-Schema entspricht, werden folgende Tools benötigt:

- GCPP (GUPRO C-Preprocessor) [Rie01b] - referenziert durch das Attribut `gcppFile`
- GCPA (GUPRO C-Parser) [Rie01a] - referenziert durch das Attribut `gcpaFile`
- G2TG (Dateikonverter von „g“ nach „tg“) - referenziert durch das Attribut `g2tgFile`

Um den ASGComputer konfektionierbar zu machen, gibt es für diese drei externen Tools Setter-Methoden, die das Setzen der Attribute zur Referenzierung der externen Tools erlauben. Zum Ausführen des Tools G2TG ist das RGEnv-Skript zwingend erforderlich. RGEnv ist ein Skript für Linux, das der Systemumgebung die Pfade zu den benötigen GUPRO-Tools und GraLab 4.4 hinzufügt. Zur Konfiguration des ASGComputer muss der Pfad zu diesem Skript und zum Verzeichnis des Program Slicing Tools gesetzt werden. Daher gibt es Setter-Methoden für die Attribute `pathToRGENVScript` und `pathToProgramSlicingToolFolder`.

Das Umwandeln der C-Programmcode-Datei in einen Graph, der den ASG beschreibt, geschieht in vier Schritten. Diese und der dabei entstehende Datenfluss sind in Abb. 6.2 gezeigt.

1. Zuerst werden durch den C-Preprocessor GCPP einige Ersetzungen, bedingte Übersetzungen sowie das Einfügen von Dateien an dem Programmcode vorgenommen. Diese Änderungen werden in einer „i“-Datei gespeichert. Zusätzlich wird eine „c.fg“-Datei erzeugt, die aber nicht weiter verwendet wird.

2. Anschließend wandelt der C-Parser GCPA den geänderten Programmcode in einen ASG um. Der ASG wird als GraLab 4.4 [DW98] Graph in einer „g“-Datei gespeichert. Außerdem erzeugt GCPA eine „pg“-Datei, welche ignoriert werden kann.

3. Danach muss der durch GCPP und GCPA erzeugte GraLab 4.4 Graph in das JGraLab-Format konvertiert werden. Dies wird durch den Konverter G2TG vorgenommen. Dieser Konverter liefert eine „tg“-Datei, die dem Speicherformat von JGraLab entspricht.

4. Zuletzt wird eine `Graph`-Instanz aus der „tg“-Datei geladen.

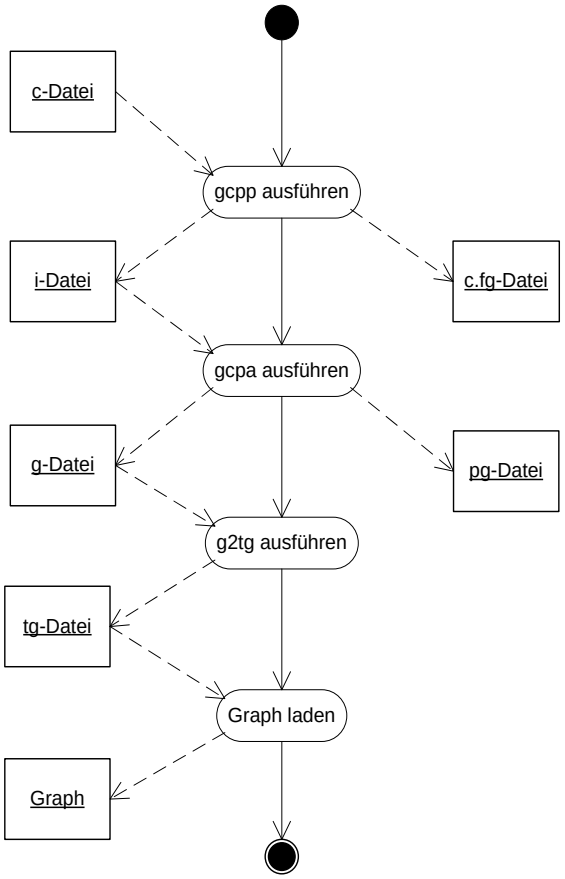

Abbildung 6.2.: Aktivitätsdiagramm: Berechnung des ASGs für C

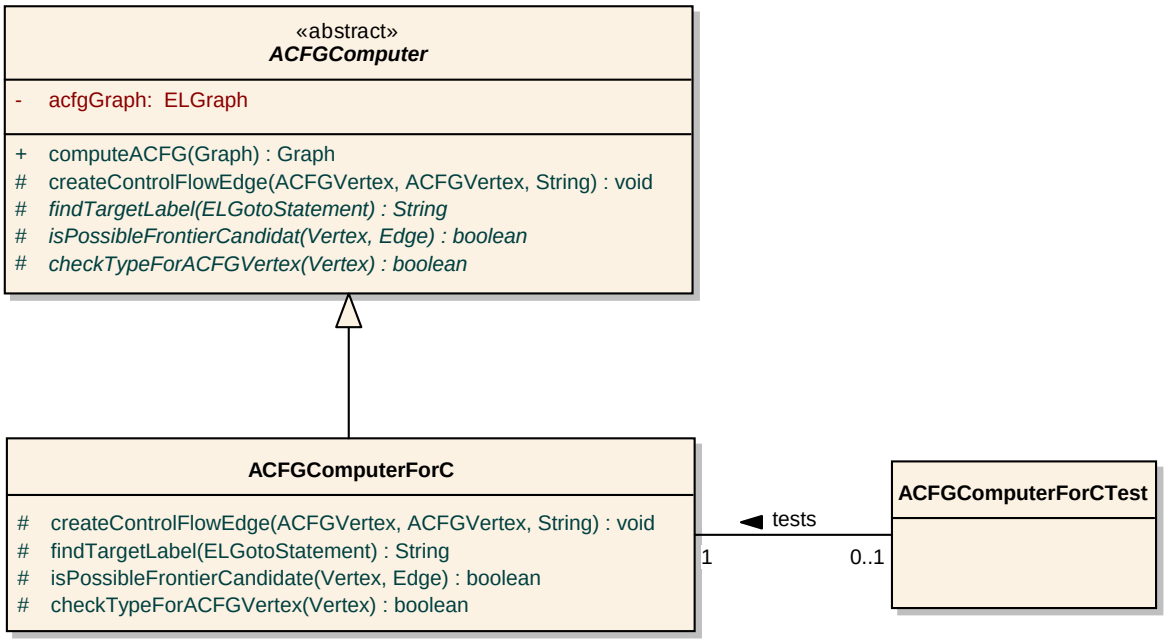

Abbildung 6.3.: Klassendiagramm: ACFGComputer

Der ASGComputer wird durch die Testklasse `ASGComputerForCTest` getestet.

6.4. ACFGComputer

Funktion: Berechnung der ACFG-Menge für ein C-Programm

Eingabe: ASG

Ausgabe: ASG, ACFGSet

Die Komponente ACFGComputer setzt den Dienst *computeAugmentedControlFlowGraphs* um. Auf Basis des abstrakten Syntaxgraphen ermittelt der ACFGComputer die *Menge der erweiterten Kontrollflussgraphen*. Pro Funktion wird ein ACFG berechnet, der alle möglichen *Abarbeitungsfolgen der Anweisungen* in einer Methode mit *Kontrollflusskanten* [BH93] beschreibt. Dabei werden ausschließlich intraprozedurale Sprunganweisungen[1] [CF94] berücksichtigt.

Das Klassendiagramm in Abb. 6.3 zeigt die Klassen des ACFGComputers. Die abstrakte Klasse `ACFGComputer` enthält alle Methoden zur Berechnung der ACFG-Menge. Diese

[1] Sprunganweisung und Sprungziel liegen in derselben Funktion.

Berechnung wird mit Hilfe der public-Methode `computeACFG` gestartet und besteht aus zwei Schritten:

1. Im ersten Schritt werden alle relevanten `ACFGVertex`-Objekte erkannt und mit der `ACFG`-Instanz der entsprechenden Methode verknüpft. Da eine Verschachtelung von `ACFGVertex`-Objekten durch „contains“-Kanten dargestellt wird, wird eine rekursive Methode [Sed02] verwendet, welche beispielsweise `ACFGVertex`-Objekte innerhalb einer `BlockStatement`-Instanz verarbeitet.

2. Im zweiten Schritt werden die `ACFGVertex`-Knoten eines ACFG mit gelabelten Kontrollflusskanten verbunden. Dazu wird in einer Schleife über alle miteinander verbundenen `ACFG`- und `ACFGVertex`-Objekte iteriert, so dass für bestimmte ACFG-Knoten je nach Typ und erfüllten Bedingungen (siehe dazu [Sch07]) entsprechend gelabelte Kontrollflusskanten zum Folge-ACFG-Knoten kreiert werden. Zur Erzeugung der Kontrollflusskanten gibt es die Methode `createControlFlowEdge`, deren Parameter den Start- und Endknoten sowie das Label definieren. Die Klasse `ACFGComputerForC` überschreibt diese Methode, weil hier eine zusätzliche mit „always“ gelabelte Kontrollflusskante `LabelStatement`-Knoten mit dem aggregierten `Statement`-Knoten verbinden muss.

Zur Bestimmung der ACFGs für C-Programme wird die Klasse `ACFGComputerForC` verwendet. Diese Klasse überschreibt einige primitive Methoden, die aufgrund von Unterschieden zwischen dem C-Schema und dem Referenzschema eine spezielle Behandlung erfordern. So werden zum Slicen von C-Programmen folgende Methoden spezialisiert:

- `findTargetLabel` operationalisiert die Namenserkennung des Sprungziels einer Goto-Anweisung.

- `isPossibleFrontierCandidate` dient dem Aufbau einer Grenze während einer Breitensuche [Sed02], welche angewendet wird, um Verschachtelungen zwischen ACFG-Knoten zu erkennen.

- `checkTypeForACFGVertex` leistet die Erkennung von `ACFGVertex`-Instanzen.

Der Test des ACFGComputers erfolgt durch die Testklasse `ACFGComputerForCTest`.

6.5. PGComputer

Funktion: Berechnung der PG-Menge für ein C-Programm

Eingabe: ASG

Ausgabe: ASG, PGSet

Der Dienst *computePointsToGraphs* wird durch die Komponente PGComputer realisiert. Die Komponente bestimmt anhand des ASGs die PG-Menge, das heißt pro Methode im ASG wird ein PG berechnet. Ein PG gibt eine konservative Abschätzung der „zeigt auf“-Beziehungen zwischen den Zeigern und Variablen innerhalb einer Methode wieder.

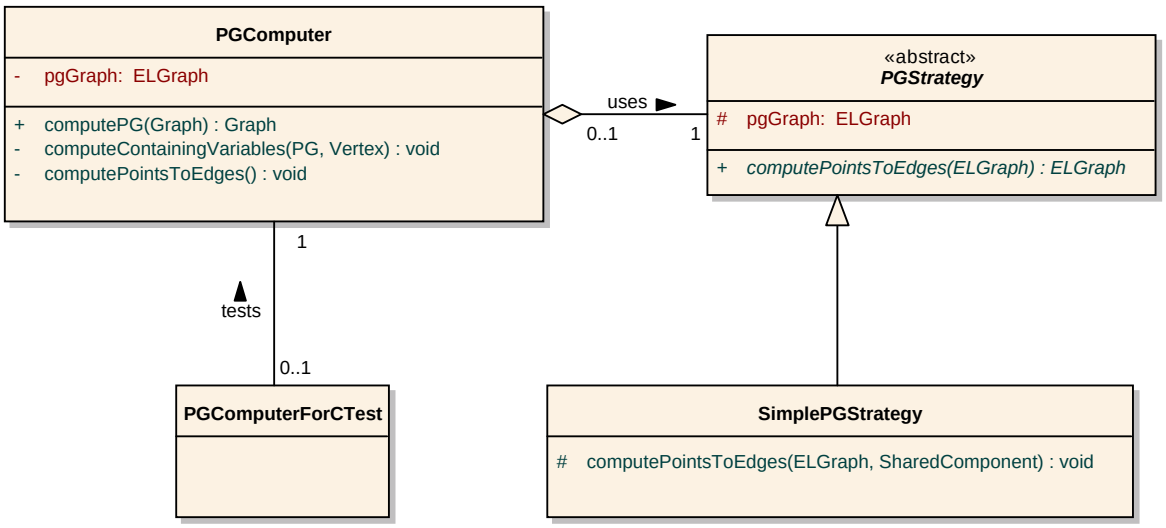

Abbildung 6.4.: Klassendiagramm: PGComputer

Die innere Struktur der Komponente wird in Abb. 6.4 gezeigt. Die Klasse `PGComputer` enthält alle Methoden zur Berechnung der PG-Menge. Die public-Methode `computePG` führt diese Berechnung aus. Für jede Methode innerhalb des ASGs wird ein PG berechnet. Dies geschieht in zwei Schritten:

1. Im ersten Schritt werden alle in einer Methode verwendeten Variablen im PG aufgenommen. Dazu wird die rekursive Methode `computeContainingVariables` verwendet.

2. Anschließend werden für alle Variablen, die als Zeiger erkannt werden, die „zeigt auf“-Beziehungen berechnet. Dies geschieht durch die Methode `computePointsTo-Edges`.

Die Methode `computePointsToEdges` wendet das *Strategy-Pattern* [GHJV95] an, um das Hinzufügen von anderen Points-To-Algorithmen zu erleichtern. Im Moment bietet die Klasse `SimplePGStrategy` einen rudimentären Algorithmus, der jeden Zeiger auf alle anderen Variablen und Zeiger zeigen lässt. Effizientere Algorithmen werden zum Beispiel in [And94], [Ste96] oder [IH97] beschrieben.

Der PGComputer wird durch die Testklasse `PGComputerForCTest` getestet.

6.6. ECGComputer

Funktion: Berechnung des ECG für ein C-Programm

Eingabe: ASG, PGSet

Ausgabe: ECG

Fassade für: CGComputer, DefUseInfComputer

Die Komponente ECGComputer setzt den Dienst *computeExtendedCallGraph* um. Da sich dieser Dienst aus den beiden Diensten *computeCallGraph* und *computeDefUseInformation* zusammensetzt, bildet die Klasse `ECGComputer` eine Fassade für die beiden Komponenten CGComputer und DefUseInfComputer. Die public-Methode `computeECG` ruft zur Berechnung des ECG die beiden public-Methoden der Klassen `CGComputer` und `DefUseInfComputer` auf. Das Klassendiagramm in Abb. 6.5 veranschaulicht die Struktur des ECGComputers.

6.7. CGComputer

Funktion: Berechnung des CG für ein C-Programm

Eingabe: ASG, PGSet

Ausgabe: CG

Die Realisierung des Dienstes *computeCallGraph* geschieht durch die Komponente CGComputer. Diese Komponente berechnet den Aufrufgraph auf der Basis des ASG und der PGs.

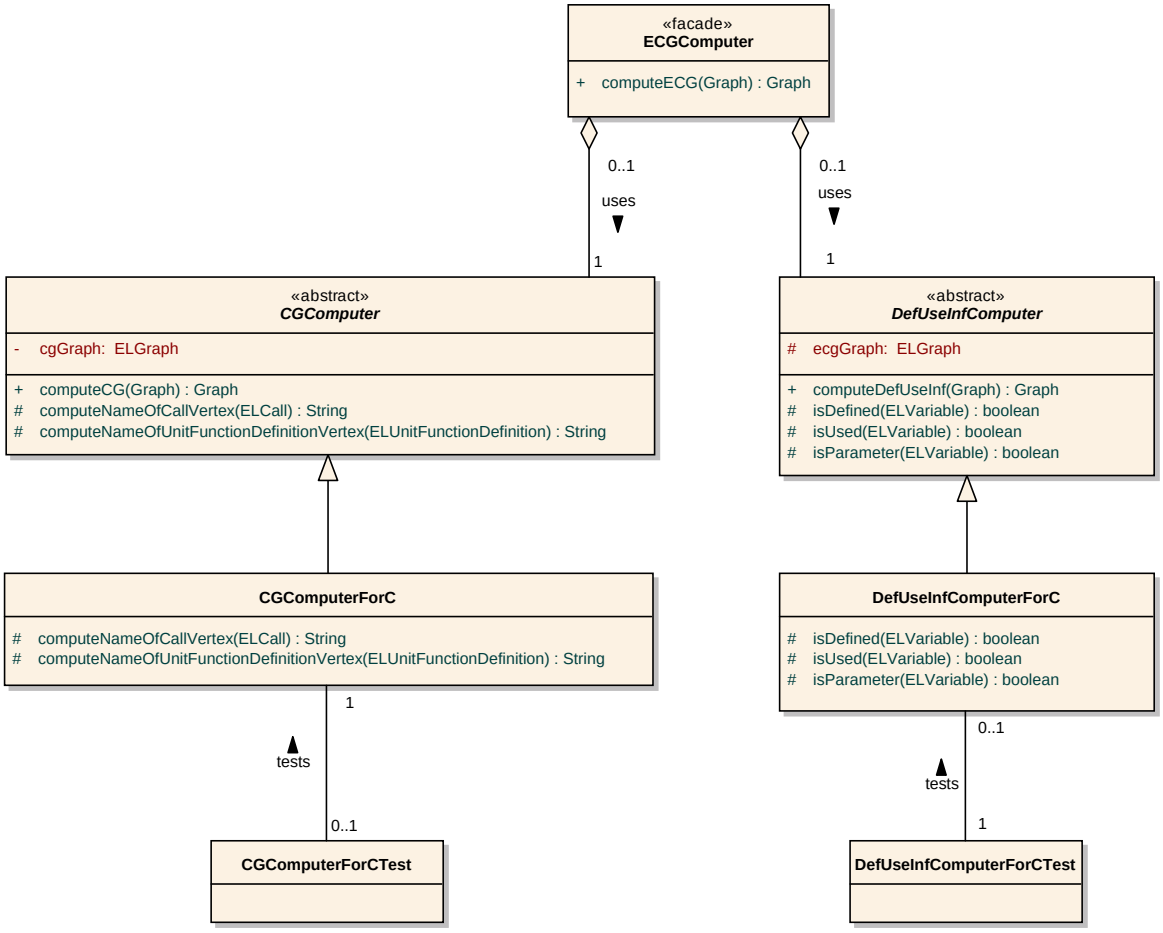

Abbildung 6.5.: Klassendiagramm: ECGComputer, CGComputer und DefUseInfComputer

Abb. 6.5 enthält die Klassen `CGComputer` und `CGComputerForC`, welche das Berechnen des Aufrufgraphen für C-Programme ermöglichen. Dazu muss die public-Methode `computeCG` ausgeführt werden.

Die Berechnung des CG besteht aus drei Schritten. Zuerst werden die Knoten bestimmt, die im ECG enthalten sind. Anschließend werden für jede `UnitFunctionDefinition`-Instanz, die in ihr enthaltenen Funktionsaufrufe gesucht und per `UnitFunctionDefinitionContainsCall`-Kante mit ihr verknüpft. Im dritten Schritt wird für jeden `Call`-Knoten der `UnitFunctionDefinition`-Knoten gesucht, welcher durch ihn aufgerufen werden kann. Die gefundenen Paare werden mit `CallCanCallUnitFunctionDefinition`-Kanten verbunden.

Dieser Algorithmus verwendet zur Bestimmung des Namens einer Funktion die Methode `computeNameOfUnitFunctionDefinitionVertex`. Diese primitive Methode muss zur Berechnung eines CG für C-Programme in der Klasse `CGComputerForC` überschrieben werden, weil sich C-Schema und RePSS an der betroffenen Stelle zu sehr unterscheiden. Ein analoger Sachverhalt liegt bei der Methode `computeNameOfCallVertex` zur Bestimmung des Namens eines Funktionsaufrufs vor.

Getestet wird der CGComputer durch die Testklasse `CGComputerForCTest`.

6.8. DefUseInfComputer

Funktion: Ergänzung des CG um Informationen über Variablen, die in Methoden definiert und benutzt werden

Eingabe: ASG, PGSet, CG

Ausgabe: ECG

Die Umsetzung des Dienstes *computeDefUseInformation* geschieht durch die Komponente DefUseInfComputer. Dieser Dienst ergänzt den CG um Informationen über die Definition oder Benutzung von Variablen innerhalb einer Funktion.

Abb. 6.5 zeigt die Klassen `DefUseInfComputer` und `DefUseInfComputerForC`, die zur Berechnung des ECG erforderlich sind.

Die public-Methode `computeDefUseInf` ergänzt den CG um die Kanten vom Typ `UnitFunctionDefinitionDefinesVariable` und `UnitFunctionDefinitionUsesVariable`. Diese Kanten verbinden `UnitFunctionDefinition`-Knoten mit den in der entsprechenden Funktion definierten bzw. benutzten `Variable`-Knoten.

Zur Berechnung dieser Kanten wird ein Algorithmus gestartet, der die primitiven Methoden `isDefined`, `isUsed` und `isParameter` zur Erkennung von definierten Variablen, benutzten Variablen und Variablen als Parameter einsetzt. Zur Ergänzung des ECG für C-Quellcode müssen diese Methoden in der Klasse `DefUseInfComputerForC` überschrieben werden.

Der Test des DefUseInfComputers wird durch den Unittest `DefUseInfComputerForCTest` umgesetzt.

6.9. ESDGComputer

Funktion: Berechnung des ESDG für ein C-Programm

Eingabe: ASG, ACFGSet, PGSet, ECG

Ausgabe: ESDG

Fassade für: BasicESDGComputer, IntMetEdgesComputer, ConDepEdgesComputer, DataFlowEdgesComputer, SumEdgesComputer

Die Komponente ESDGComputer realisiert den Dienst *computeExtendedSystemDependenceGraph*, indem sie durch die Klasse `ESDGComputer` eine Fassade für die Komponenten BasicESDGComputer, IntMetEdgesComputer, ConDepEdgesComputer, DataFlowEdgesComputer und SumEdgesComputer bildet. Mit Hilfe seiner public-Methode `computeESDG` ermittelt der ESDGComputer den erweiterten Systemabhängigkeitsgraphen durch Ausführung der public-Methoden, die sich in den Komponenten hinter der Fassade befinden. Die Struktur des ESDGComputers wird in Abb. 6.6 durch ein Klassendiagramm gezeigt.

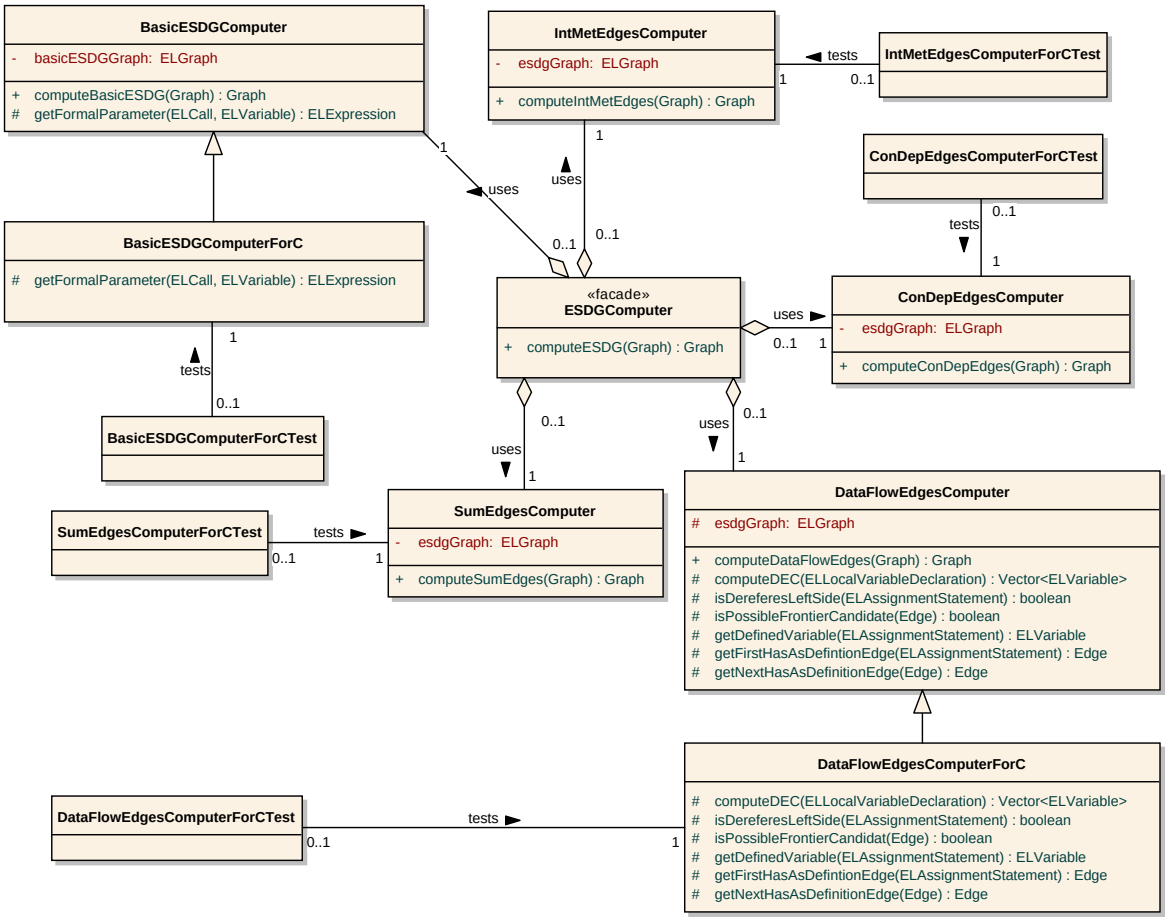

Abbildung 6.6.: Klassendiagramm: ESDGComputer, BasicESDGComputer, IntMetEdgesComputer, ConDepEdgesComputer, DataFlowEdgesComputer und SumEdgesComputer

6.10. BasicESDGComputer

Funktion: Grundgerüstkonstruktion des ESDG für ein C-Programm

Eingabe: ASG, ECG

Ausgabe: ESDG[unvollständig]

Die Komponente BasicESDGComputer realisiert den Dienst *computeBasicESDG*. Abb. 6.6 enthält die Klassen `BasicESDGComputer` und `BasicESDGComputerForC`, welche gemeinsam für die Erzeugung der Basis eines erweiterten Systemabhängigkeitsgraphen für C-Quellcode verantwortlich sind.

Durch die public-Methode `computeBasicESDG` werden alle `ESDGVertex`-Knoten erkannt und zum ESDG hinzugefügt. Wenn eine `Entry`-Instanz gefunden wird, so wird sie mit der korrespondierenden `UnitFunctionDefinition`-Instanz verknüpft. Des Weiteren werden an diese Funktion für alle Parameter `FormalInParameterVertex`- und `FormalOutParameterVertex`-Knoten geknüpft. An `Call`-Knoten, die zu in dieser Funktion aufgerufenen Funktionen gehören, werden entsprechende `ActualIn-ParameterVertex`- und `ActualOutParameterVertex`-Knoten geknüpft. Zuletzt werden Verbindungen zwischen `ReturnStatement`-Objekten und ihren korrespondierenden `ReturnValue`-Objekten hergestellt. Ausführlicheres dazu findet der Leser in [Sch07].

Aufgrund der komplexen Modellierung von Parametern im C-Schema ist eine spezialisierte Methode `getFormalParameter` in der Klasse `BasicESDGComputerForC` erforderlich. Diese Methode bestimmt das Argument des übergebenen Funktionsaufrufs, welches der zusätzlich übergebenen Variablen entspricht.

Der Unittest `BasicESDGComputerForCTest` überprüft die Funktion des BasicESDG-Computers.

6.11. IntMetEdgesComputer

Funktion: Ergänzung des ESDG um Aufruf-, Parameter-in- und Parameter-out-Kanten

Eingabe: ESDG[unvollständig], ASG, ECG

Ausgabe: ESDG[unvollständig]

IntMetEdgesComputer ist die Komponente zur Umsetzung des Dienstes *computeInterMethodEdges*. Durch diesen Dienst werden die ESDG-Teilgraphen zur Darstellung von Funktionen miteinander verbunden.

Weil diese Komponente nicht direkt vom C-Schema abhängt, wird sie nur durch die Klasse `IntMetEdgesComputer` realisiert. Diese Klasse und ihr Unittest `IntMetEdgesComputerForCTest` befinden sich im Klassendiagramm von Abb. 6.6.

Um die interprozeduralen Kanten zu erzeugen, führt die public-Methode `computeIntMetEdges` drei Schritte aus:

1. Zuerst werden die Aufrufkanten erzeugt, welche Funktionsaufruf-Knoten mit dem `Entry`-Knoten der aufgerufenen Funktion verbinden.

2. Anschließend werden die Parameter-in-Kanten berechnet. Diese verbinden Actual-in- mit Formal-in-Knoten, sofern die zugehörigen Funktionen per Aufrufkante verknüpft sind und der Actual-in-Knoten die Variable definiert, welche der Formal-in-Knoten als Definition verwendet.

3. Zuletzt werden die Parameter-out-Kanten analog zu den Parameter-in-Kanten bestimmt, wobei diese Kanten unter analogen Bedingungen Formal-out- mit Actual-out-Knoten zusammenfügen.

In C beschreiben interprozedurale Kanten 1:1-Beziehungen, während in objektorientierten Programmiersprachen aufgrund der Polymorphie 1:N-Beziehungen bestehen können.

6.12. ConDepEdgesComputer

Funktion: Ergänzung des ESDG um Kontrollkanten

Eingabe: ESDG[unvollständig], ASG, ACFGSet, PGSet

Ausgabe: ESDG[unvollständig]

Der Dienst *computeControlDependenceEdges* wird durch die Komponente ConDepEdgesComputer verwirklicht. Anhand der erweiterten Kontrollflussgraphen werden die Kontrollkanten berechnet, welche die Schachtelungshierarchie der Anweisungen wiedergeben.

ConDepEdgesComputer ist eine ausschließlich auf RePSS basierende Komponente, so dass zur Realisierung die Klasse `ConDepEdgesComputer` ausreichend ist. Diese Klasse und Ihre Testklasse `ConDepEdgesComputerForCTest` werden in Abb. 6.6 veranschaulicht.

Die Ausführung der Methode `computeConDepEdges` erzeugt von jedem `Entry`-Knoten ausgehend einen Graphen, der die Postdominanz-Beziehungen [FOW87] zwischen ESDG-Knoten beschreibt. Eine Kontrollkante wird von einem `JumpStatement`-, `CompoundStatementWithTest`- oder `Entry`-Knoten zu einem anderen ESDG-Knoten genau dann erzeugt, wenn der Startknoten den Typ `CompoundStatementWithTest`, `JumpStatement` oder `Entry` hat und vom Startknoten zum Endknoten ein Pfad aus Kontrollflusskanten besteht, der keine anderen `CompoundStatementWithTest`- oder `JumpStatement`-Knoten enthält.

Das Label der erzeugten Kontrollkante ist *true*, sofern der Pfad mit einer *always* oder *true* gelabelten Kontrollflusskante beginnt. Startet der Pfad mit einer *false* oder *never* gelabelten Kontrollflusskante, so wird eine mit *false* gelabelte Kontrollkante zwischen den beiden ESDG-Knoten erstellt.

Weiterführende Erklärungen zu Kontrollkanten und Postdominanz findet man in [Sch07].

6.13. DataFlowEdgesComputer

Funktion: Ergänzung des ESDG um Datenfluss- und Deklarationskanten

Eingabe: ESDG[unvollständig], ASG, ACFGSet, PGSet

Ausgabe: ESDG[unvollständig]

Die Komponente DataFlowEdgesComputer realisiert den Dienst *computeDataFlowEdges*. Dieser Dienst erzeugt eine Datenflusskante zwischen zwei Knoten, wenn der Anfangsknoten dieser Datenflusskante eine Variable definiert, die während eines beliebigen Pfades im Kontrollfluss zum Endknoten nicht redefiniert und im Endknoten benutzt wird. Ähnliches gilt für die erzeugten Deklarationskanten. Im Startknoten wird die Variable deklariert, im Endknoten benutzt oder definiert und während eines beliebigen Pfades im Kontrollfluss zum Endpunkt darf sie weder benutzt noch definiert werden.

Abb. 6.6 enthält die Klassen `DataFlowEdgesComputer` und `DataFlowEdgesComputerForC` zur Realisierung des DataFlowEdgesComputers und die Klasse `DataFlowEdgesComputerForCTest`, um ihn zu testen.

Um die Datenfluss- und Deklarationskanten zu erzeugen, führt die public-Methode `computeDataFlowEdges` drei Schritte aus:

1. Zuerst werden die DEF- und USE-Mengen aller relevanten ESDG-Knoten ermittelt und in den `HashMap`-Singletons der Oberklasse `ProgramSlicingComponent` gespeichert (siehe Abschnitt 6.21). Dies geschieht anhand der Definition für DEF- und USE-Mengen in [Sch07].

2. Im zweiten Schritt werden die Deklarationskanten mittels der Definition für DEC-Menge in [Sch07] berechnet. Dazu wird der Kontrollfluss ausgehend von den `LocalVariableDeclaration`-Knoten entlang der Kontrollflusskanten mit einem Label ungleich *never* traversiert. Wird während der Traversierung ein ESDG-Knoten gefunden, dessen DEF- oder USE-Menge die im `LocalVariableDeclaration`-Knoten deklarierte Variable enthält, so wird eine Deklarationskante vom `LocalVariableDeclaration`-Knoten zu diesem ESDG-Knoten gezogen, sofern diese Variable auf den Knoten des traversierten Kontrollflusspfades weder benutzt noch definiert wird. Zur Traversierung wird vom `LocalVariableDeclaration`-Knoten ausgehend für den Endknoten jeder nicht *never*-gelabelten, ausgehenden Kontrollflusskante eine rekursive Methode aufgerufen, welche den Endknoten als Eingabe erhält und diesen zur Erzeugung von Deklarationskanten nach benutzten und definierten Variablen untersucht.

3. Zuletzt geschieht die Berechnung der Datenflusskanten. Der hierzu eingesetzte Traversierungs-Algorithmus ist im Prinzip analog zu dem für die Deklarationskanten. Allerdings ist der vollständige Algorithmus zur Bestimmung der Datenflusskanten wesentlich komplexer, da es nach den Definitionen in [Sch07] als mögliche Startpunkte für eine Datenflusskante `Statement`-, `FormalInParameterVertex`- und `ActualOutParameterVertex`-Knoten gibt, die mit den möglichen Endpunkten `Statement`-, `FormalOutParameterVertex`- und `ActualInParameterVertex`-Knoten kombiniert werden müssen. Zudem enden Datenflusskanten ausschließlich bei ESDG-Knoten, welche die vorab definierte Variable benutzen.

Die Algorithmen des DataFlowEdgesComputers benötigen an einigen Stellen direkten Zugriff auf das C-Schema. Daher werden in der Klasse `DataFlowEdgesComputerForC` folgende Methoden überschrieben:

- `computeDEC`: Diese Methode berechnet die DEC-Menge eines `LocalVariable-Declaration`-Knoten und wird aufgrund der unterschiedlichen Modellierung von Variablendeklarationen im C- und Referenzschema überschrieben.

- `isDerefersLeftSide`: Diese Methode simuliert das Attribut `derefers-LeftSide` in RePSS, da die Dereferenzierung von Variablen im C-Schema nicht an `AssignmentStatement`-Instanzen gebunden ist.

- `isPossibleFrontierCandidate` dient dem Aufbau einer Grenze während einer Breitensuche zur Berechnung der USE-Menge eines ESDG-Knotens.

- `getDefinedVariable` liefert die definierte Variable in einem `Assignment-Statement`-Objekt und muss aufgrund der Differenzen in der Modellierung von Assignments im C-Schema und in RePSS überschrieben werden.

- Aufgrund der Unterschiede in der Modellierung von Assignments gibt es auch die beiden überschriebenen Methoden `getFirstHasAsDefinitionEdge` und `getNextHasAsDefinitionEdge`, welche die gleichnamigen, von JGraLab generierten Methoden für die generierte Klasse `AssignmentStatement` simulieren.

6.14. SumEdgesComputer

Funktion: Ergänzung des ESDG um transitive Kanten

Eingabe: ESDG[unvollständig]

Ausgabe: ESDG

Transitive Kanten werden durch den Dienst *computeSummaryEdges* ermittelt, welcher durch die Komponente SumEdgesComputer umgesetzt wird. Transitive Kanten verbinden `ActualInParameterVertex`-Instanzen mit `ActualOutParameterVertex`-Instanzen, sofern diese Instanzen die Bedingungen aus [RHSR94] erfüllen.

Der SumEdgesComputer besteht aus der Klasse `SumEdgesComputer` und wird durch den Unittest `SumEdgesComputerForCTest` überprüft. Diese Klasse und ihr Test sind in Abb. 6.6 zu sehen.

Durch das Aufrufen der public-Methode `computeSumEdges` werden alle `Actual-InParameterVertex-ActualOutParameterVertex`-Knotenpaare gesucht, wel-

che per `ESDGVertexHasParameterEdgeToParameterVertex`-Kante mit demselben ESDG-Knoten verknüpft sind. Anschließend wird überprüft, ob zwischen dem Knotenpaar ein Pfad aus ESDG-Kanten besteht, der andere `ActualInParameterVertex-ActualOutParameterVertex`-Knotenpaare nur dann enthalten darf, wenn diese ebenfalls mit demselben ESDG-Knoten verbunden sind. Zur Überprüfung dieses Pfades wird Rekursion genutzt. Jedes Paar mit gültigem Pfad aus ESDG-Kanten wird per `ActualInParameterVertexHasSummaryEdgeToActualOutParameterVertex`-Kante verknüpft.

6.15. ProgramSlicer

Funktion: Markierung der Slice oder des Chops im ESDG

Eingabe: ESDG, Slicing- oder Chopping-Kriterium, optional Trace, „i“-Datei

Ausgabe: ESDG[mit markierter Slice oder markiertem Chop]

Fassade für: ESDGMarker, StaticBackwardSliceComputer, StaticForwardSliceComputer, StaticChopComputer, DynamicBackwardSliceComputer, DynamicForwardSliceComputer, DynamicChopComputer, ExecutableBackwardSliceComputer

Die Komponente ProgramSlicer verwirklicht den Dienst *sliceProgram*. Deren Klasse `ProgramSlicer` bildet eine Fassade für die Komponenten ESDGMarker, StaticBackwardSliceComputer, StaticForwardSliceComputer, StaticChopComputer, DynamicBackwardSliceComputer, DynamicForwardSliceComputer, DynamicChopComputer und ExecutableBackwardSliceComputer. Durch die public-Methoden `computeStaticSliceOrChop` und `computeDynamicSliceOrChop` wird entweder eine statische oder eine dynamische Slice bzw. Chop im ESDG markiert. Die Abb. 6.7 und 6.8 beinhalten den ProgramSlicer und seine Beziehungen.

Das verwendete Slicing- oder Chopping-Verfahren hängt von der dynamischen Bindung während der Konfiguration ab, siehe dazu Abschnitt 5.6. Daher gibt es zur Konfiguration die beiden Methoden `setSlicingMode` und `setChoppingMode`, welche sicherstellen, dass falsche Benutzer-Eingaben des Slicing- oder Chopping-Kriteriums mit regulären Ausdrücken [Fri07] richtig erkannt werden. Die eingegebene Trace wird ebenfalls mit einem regulären Ausdruck überprüft. Tabelle 6.1 zeigt die regulären Ausdrücke für die beiden Kriterien und die Trace.

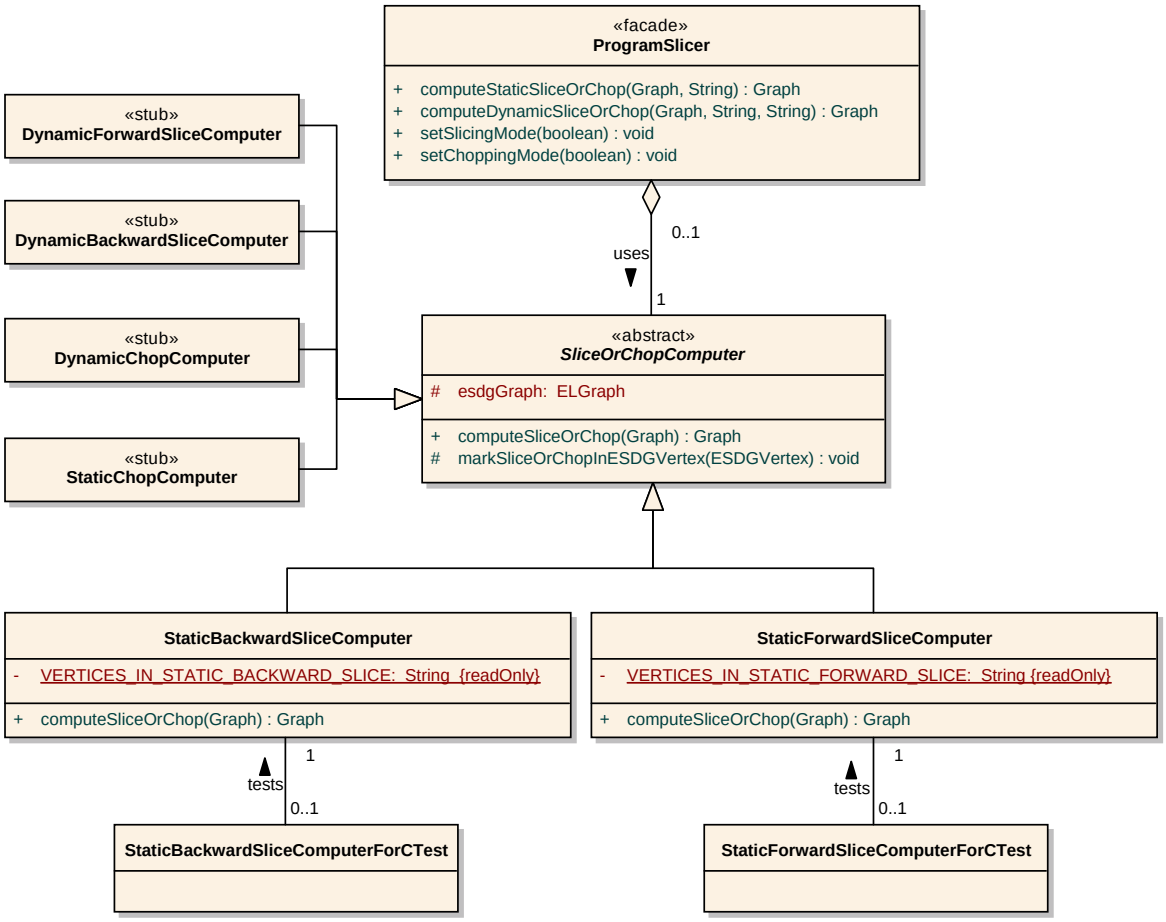

Abbildung 6.7.: Klassendiagramm: ProgramSlicer, StaticBackwardSliceComputer, StaticForwardSliceComputer, StaticChopComputer, DynamicBackwardSliceComputer, DynamicForwardSliceComputer und DynamicChopComputer

Eingabe	**Regulärer Ausdruck**	**Beispiel**
Slicing-Krit.	’<’ [’0’ - ’9’] + ’,’ ’{’ [’a’ - ’z’ ’A’ - ’Z’] [’a’ - ’z’ ’A’ - ’Z’ ’0’ - ’9’] * (’,’ [’a’ - ’z’ ’A’ - ’Z’] [’a’ - ’z’ ’A’ - ’Z’ ’0’ - ’9’] *) * ’}’ ’>’	<1,{a1,c}>
Chopping-Krit.	’<’ [’0’ - ’9’] + ’,’ [’0’ - ’9’] + ’>’	<123,17>
Trace	’<’ [’0’ - ’9’] + ’(’ ’,’ [’0’ - ’9’] + ’)’ * ’>’	<1,20,3,7>

Tabelle 6.1.: Reguläre Ausdrücke für Benutzer-Eingaben (Terminale befinden sich in Hochkommata)

6.16. ESDGMarker

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

Eingabe: ESDG, Slicing- oder Chopping-Kriterium, optional Trace, „i“-Datei

Ausgabe: ESDG[mit markiertem Slicing- oder Chopping-Kriterium und optional markierter Trace]

Der Dienst *markESDG* wird durch die Komponente ESDGMarker realisiert. Diese Komponente ist für die *Markierung* des Slicing- und Chopping-Kriteriums sowie der Trace zuständig. Da in dieser Diplomarbeit nur statische Slicing-Verfahren konkret implementiert werden, markiert der ESDGMarker auch nur das Slicing-Kriterium innerhalb des ESDG. Der Markierungs-Algorithmus befindet sich in der Klasse `ESDGMarker`. Abb. 6.8 zeigt neben dieser Klasse auch noch die zugehörige Testklasse `ESDGMarkerForCTest`.

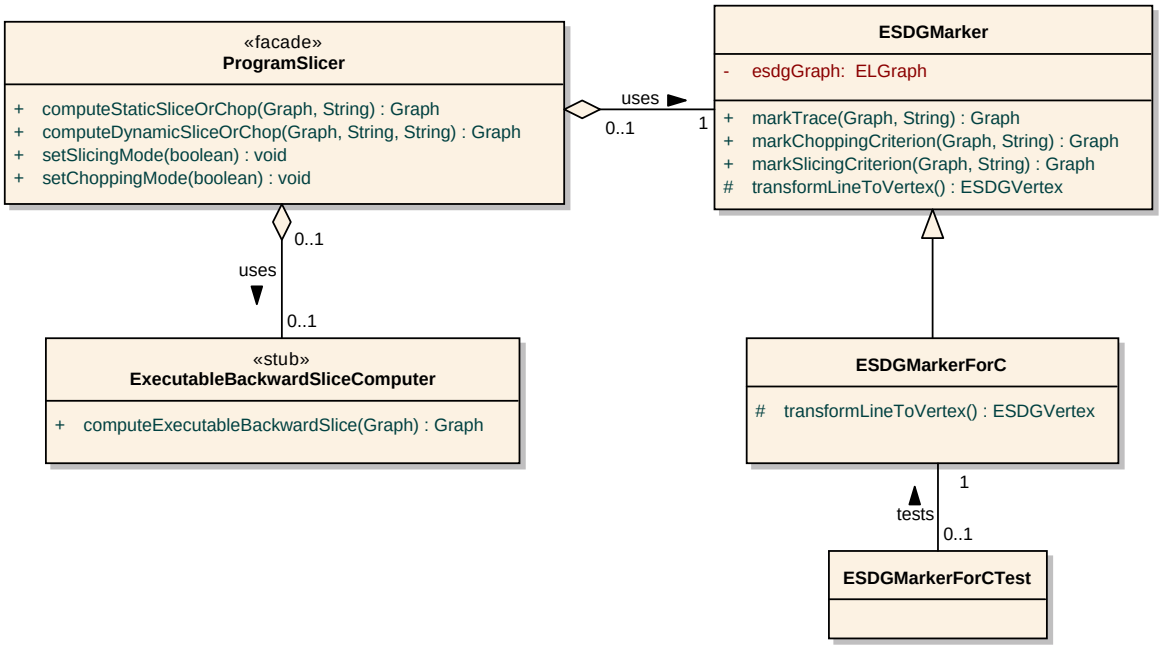

Abbildung 6.8.: Klassendiagramm: ProgramSlicer, ESDGMarker und ExecutableBackwardSliceComputer

Der ESDGMarker bietet zur Markierung des Slicing-Kriteriums die public-Methode `markSlicingCriterion` an. Außerdem werden die public-Methoden `markTrace` und `markChoppingCriterion` zur Markierung der Trace und des Chopping-Kriteriums als Stubs[2] angeboten.

[2]Die hier verwendeten Stubs geben ihre Eingaben unverändert als Ausgaben zurück.

Zur Markierung des Slicing-Kriteriums wird vorab überprüft, ob das übergebene Slicing-Kriterium den regulären Ausdruck in Tabelle 6.1 erfüllt. Anschließend wird der Knoten bestimmt, welcher der Anweisung entspricht, die im Slicing-Kriterium festgelegt ist. Dazu wird in der Klasse `ESDGMarkerForC` die Methode `transformLineToVertex` so überschrieben, dass sie anhand der im Slicing-Kriterium gegebenen Zeilennummer die ID des ESDG-Knotens bestimmt, der eine Anweisung vertritt und aus den meisten Zeichen in der gegebenen Zeile besteht. Außerdem werden durch das Slicing-Kriterium alle Knoten gesammelt, die der definierten Variablenmenge entsprechen.

Anschließend werden die ESDG-Knoten markiert, welche zum Slicing-Kriterium gehören. Dazu werden zwei Fälle unterschieden:

1. Im einfacheren Fall werden alle Variablen des Slicing-Kriteriums innerhalb der Anweisung des Slicing-Kriteriums benutzt oder definiert, so dass ausschließlich der Knoten markiert wird, welcher die Anweisung des Slicing-Kriteriums vertritt.

2. Im komplexeren Fall definiert oder benutzt der Knoten, welcher die Anweisung des Slicing-Kriteriums vertritt, nicht alle Variablen des Slicing-Kriteriums. Dadurch müssen alle Knoten markiert werden, die eine Variable der Variablenmenge des Slicing-Kriteriums definieren und deren Definition die Anweisung im Slicing-Kriterium erreicht.

Zur Markierung des Slicing-Kriteriums wird ein Algorithmus eingesetzt, welcher die Definition der zu markierenden ESDG-Knoten aus [Sch07] umsetzt. Dieser Algorithmus setzt ähnliche Techniken wie der Algorithmus des DataFlowEdgesComputers ein, da in beiden Algorithmen der Kontrollfluss traversiert werden muss.

6.17. StaticBackwardSliceComputer

Funktion: Markierung der Slice im ESDG durch statisches Rückwärtsslicen

Eingabe: ESDG[mit markiertem Slicing-Kriterium]

Ausgabe: ESDG[mit markierter Slice]

Die Umsetzung des Dienstes *computeStaticBackwardSlice* geschieht durch die Komponente StaticBackwardSliceComputer. Ausgehend vom markierten Slicing-Kriterium im ESDG werden die Knoten der Slice durch Rückwärts-Traversierung entlang der Kontroll-,

Datenfluss-, Deklarations-, Parameter-, Parameter-in-, Parameter-out-, Aufruf- und transitiven Kanten ermittelt. Dabei ist zu beachten, dass weder eine Parameter-out-Kante vor einer Aufruf- oder Parameter-in-Kante, noch eine Aufruf- oder Parameter-in-Kante nach einer Parameter-out-Kante traversiert werden darf. Die GReQL2-Anfrage zur Realisierung dieser Traversierung befindet sich in Listing 6.1. Sie entspricht der Definition der statischen Rückwärtsslice aus [Sch07]. Die Ergebnismenge der Anfrage enthält alle ESDG-Knoten, die in der statischen Rückwärtsslice enthalten sind. Dabei besteht die Ergebnismenge aus Knotenpaaren, um die Abhängigkeit zwischen den Parameter-in-, Aufruf- und Parameter-out-Kanten wiederzugeben. Redundanz in der Ergebnismenge wird durch redundantes Markieren der ESDG-Knoten behandelt.

```
1  USING esdgA:
2  FROM esdgV:V{ESDGVertex}, esdgW:V{ESDGVertex}
3  WITH esdgA.isInSliceCrit
4       AND esdgA <--{ESDGVertexHasESDGVertexAttributes}
5                 (<--{ESDGVertexHasControlDependenceEdgeToESDGVertex}
6                 |<--{ESDGVertexHasDataFlowEdgeToESDGVertex}
7                 |<--{ELLocalVariableDeclarationHasDeclarationEdgeToESDGVertex}
8                 |<--{ESDGVertexHasParameterEdgeToParameterVertex}
9                 |<--{ActualInParameterVertexHasParameterInEdgeToFormalInParameterVertex}
10                |<--{ELStatementHasCallEdgeToEntry}
11                |<--{ActualInParameterVertexHasSummaryEdgeToActualOutParameterVertex})*
12            esdgV
13                (<--{ESDGVertexHasControlDependenceEdgeToESDGVertex}
14                |<--{ESDGVertexHasDataFlowEdgeToESDGVertex}
15                |<--{ELLocalVariableDeclarationHasDeclarationEdgeToESDGVertex}
16                |<--{ESDGVertexHasParameterEdgeToParameterVertex}
17                |<--{FormalOutParameterVertexHasParameterOutEdgeToActualOutParameterVertex}
18                |<--{ActualInParameterVertexHasSummaryEdgeToActualOutParameterVertex})*
19            esdgW
20 REPORT esdgV, esdgW END
```

Listing 6.1: GReQL2-Anfrage zur Berechnung der statischen Rückwärtsslice

Das Klassendiagramm in Abb. 6.7 enthält den StaticBackwardSliceComputer und seine Testklasse `StaticBackwardSliceComputerForCTest`. Die Klasse `StaticBackwardSliceComputer` überschreibt die public-Methode `computeSliceOrChop` der abstrakten Oberklasse `SliceOrChopComputer`. Diese public-Methode verwendet

die *GReQL2-Anfrage* aus Listing 6.1, um die Knoten der Slice zu berechnen, welche anschließend mit Hilfe der geerbten Methode `markSliceOrChopInESDGVertex` als Teil der Slice markiert werden.

6.18. StaticForwardSliceComputer

Funktion: Markierung der Slice im ESDG durch statisches Vorwärtsslicen

Eingabe: ESDG[mit markiertem Slicing-Kriterium]

Ausgabe: ESDG[mit markierter Slice]

Die Realisierung des Dienstes *computeStaticForwardSlice* durch die Komponente StaticForwardSliceComputer ist analog zum StaticBackwardSliceComputer. Der einzige Unterschied ist die GReQL2-Anfrage aus Listing 6.2 zum statischen Vorwärtsslicen anstelle der GReQL2-Anfrage zum statischen Rückwärtsslicen aus Listing 6.1. Die GReQL2-Anfrage des StaticForwardSliceComputer ist der GReQL2-Anfrage des StaticBackwardSliceComputer sehr ähnlich. Anstelle einer Rückwärts-Traversierung wird eine Vorwärts-Traversierung eingesetzt, welche Parameter-out-Kanten stets vor Aufruf- oder Parameter-in-Kante traversiert.

Der StaticForwardSliceComputer wird durch die Klasse `StaticForwardSliceComputer` verwirklicht und durch `StaticForwardSliceComputerForCTest` getestet. Diese Klassen werden in Abb. 6.7 gezeigt.

6.19. Weitere Slice- und ChopComputer

Funktion: Markierung der Slice oder des Chops im ESDG

Eingabe: ESDG[mit markiertem Slicing- oder Chopping-Kriterium und optional markierter Trace]

Ausgabe: ESDG[mit markierter Slice oder markiertem Chop]

Die Komponenten StaticChopComputer, DynamicBackwardSliceComputer, DynamicForwardSliceComputer, DynamicChopComputer und ExecutableBackwardSliceComputer bieten

```
1  USING esdgA:
2  FROM esdgV:V{ESDGVertex}, esdgW:V{ESDGVertex}
3  WITH esdgA.isInSliceCrit
4       AND esdgA <--{ESDGVertexHasESDGVertexAttributes}
5                 (-->{ESDGVertexHasControlDependenceEdgeToESDGVertex}
6                 |-->{ESDGVertexHasDataFlowEdgeToESDGVertex}
7                 |-->{ELLocalVariableDeclarationHasDeclarationEdgeToESDGVertex}
8                 |-->{ESDGVertexHasParameterEdgeToParameterVertex}
9                 |-->{FormalOutParameterVertexHasParameterOutEdgeToActualOutParameterVertex}
10                |-->{ActualInParameterVertexHasSummaryEdgeToActualOutParameterVertex})*
11            esdgV
12                (-->{ESDGVertexHasControlDependenceEdgeToESDGVertex}
13                |-->{ESDGVertexHasDataFlowEdgeToESDGVertex}
14                |-->{ELLocalVariableDeclarationHasDeclarationEdgeToESDGVertex}
15                |-->{ESDGVertexHasParameterEdgeToParameterVertex}
16                |-->{ActualInParameterVertexHasParameterInEdgeToFormalInParameterVertex}
17                |-->{ELStatementHasCallEdgeToEntry}
18                |-->{ActualInParameterVertexHasSummaryEdgeToActualOutParameterVertex})*
19            esdgW
20 REPORT esdgV, esdgW END
```

Listing 6.2: GReQL2-Anfrage zur Berechnung der statischen Vorwärtsslice

alternative Slicing- oder Chopping-Verfahren an, siehe dazu Abschnitt 2.1. Diese Komponenten werden jedoch nur als *Stubs*[3] angeboten. Zur nachträglichen Implementation dieser Komponenten muss die geerbte Methode `computeSliceOrChop` überschrieben werden, da diese Methode in der abstrakten Oberklasse `SliceOrChopComputer` als Stub angeboten wird. In den Klassendiagrammen der Abb. 6.7 und 6.8 sind diese Komponenten mit dem Stereotyp `stub` gekennzeichnet.

6.20. ESDGToCodeConverter

Funktion: Konvertierung einer im ESDG markierten Slice in C-Code

Eingabe: ESDG[mit markierter Slice oder markiertem Chop], „i“-Datei

Ausgabe: „c“-Datei mit Slice oder Chop

Die Komponente ESDGToCodeConverter verwirklicht den Dienst *convertESDGToCode*. Sie übersetzt die markierte Slice oder den markierten Chop in Code, der in einer Datei gespeichert wird.

Der ESDGToCodeConverter wird durch die abstrakte Klasse `ESDGToCodeConverter` und die Spezialisierungsklasse `ESDGToCodeConverterForC` implementiert. Diese Klassen und die zugehörige Testklasse befinden sich im Klassendiagramm in Abb. 6.9.

Beim ESDGToCodeConverter handelt es sich um eine Komponente, die sehr stark vom Schema der zu slicenden Programmiersprache abhängig ist. Das kommt daher, dass die Abbildung zwischen den Knoten des ESDG und den entsprechenden Stellen im Quellcode nicht einheitlich umgesetzt ist.

Der abstrakte Syntaxgraph für ein C-Programm besitzt Kanten vom Typ `Links` [Rie01b], welche die Knoten des ASG auf Stellen der „i“-Datei abbilden. Dazu besitzt jede `Links`-Kante die Attribute `lengthAlpha`, `lengthOmega`, `offsetAlpha` und `offset-Omega`. Durch `offset-` wird die Startposition der Codestelle zur Repräsentation des Knotens innerhalb der „i“-Datei festgelegt. Mit Hilfe von `length-` wird in der „i“-Datei die Zeichenanzahl der Codestelle ab dem `offset-` definiert. `-Alpha` und `-Omega` stellen den Bezug zum Knoten am Alpha- bzw. Omega-Ende der Kante her.

[3] Der hier verwendete Stub gibt seine Eingabe unverändert als Ausgabe zurück.

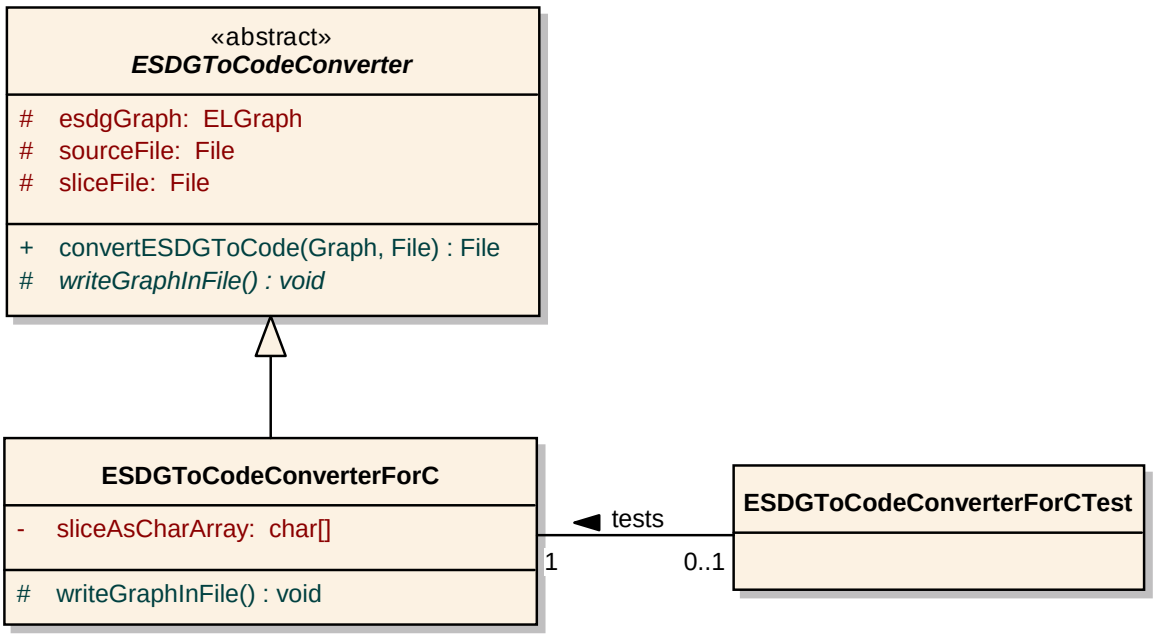

Abbildung 6.9.: Klassendiagramm: ESDGToCodeConverter

Durch den Aufruf der public-Methode `convertESDGToCode` wird die Konvertierung des ESDG gestartet. Neben dem ESDG braucht diese Methode auch noch die ursprüngliche Quellcode-Datei als Eingabe, weil der Graph nur Verweise auf den Quellcode enthält. Die Konvertierung besteht aus einem allgemeinen Schritt und einem Schritt, der speziell für die Konvertierung in C-Code ist. Im allgemeinen Schritt wird eine leere Datei angelegt, die später die Slice enthalten wird. Diese Datei befindet sich im selben Ordner und besitzt die selbe Endung wie die Quellcode-Datei. Ihr Name ist jedoch „slice“.

Der spezielle Schritt wird durch die protected-Methode `writeGraphInFile` vollzogen. Daher muss diese Methode auch in der `ESDGToCodeConverterForC`-Klasse implementiert werden. Diese Methode besteht zum Konvertieren in C-Code aus vier Teilschritten:

1. Zuerst wird der relevante Inhalt der „i“-Datei, welche anhand des Pfades zur C-Quellcode-Datei gefunden wird, im Character-Array `sliceAsCharArray` zwischengespeichert.

2. Anschließend werden alle Anweisungen, die nicht zur Slice gehören und keine anderen Anweisungen enthalten, aus dem zwischengespeicherten Character-Array entfernt.

3. Der nächste Schritt wird solange wiederholt, bis nichts mehr aus dem Character-Array gelöscht werden darf. In diesem Schritt werden alle Anweisungen eliminiert, welche ursprünglich andere Anweisungen enthielten, die im vorherigen Schritt entfernt wur-

den. Funktionsdefinitionen, deren kompletter Inhalt gelöscht wurde, werden ebenfalls eliminiert.

4. Zuletzt wird der Character-Array, welcher nun die Slice enthält, in die im allgemeinen Schritt erzeugte `slice.c`-Datei geschrieben.

6.21. ProgramSlicingComponent

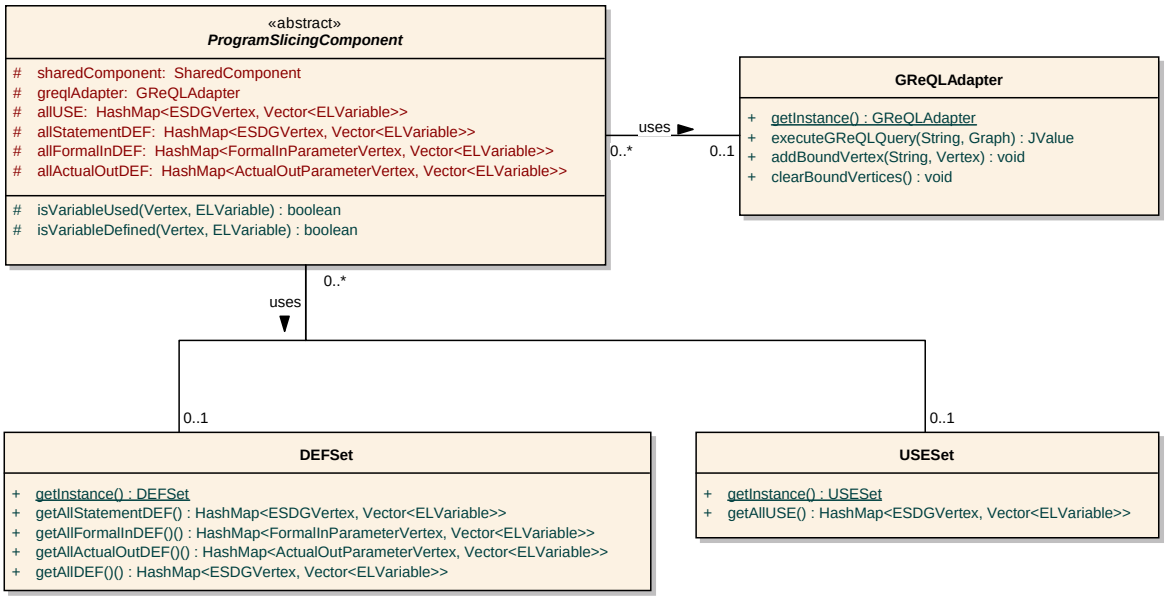

Abbildung 6.10.: Klassendiagramm: GReQLAdapter und ProgramSlicingComponent

`ProgramSlicingComponent` ist die *abstrakte Superklasse aller RePST-Komponenten* und ist in Abb. 6.10 veranschaulicht. Von dieser Klasse erben alle anderen Klassen den Zugriff auf eine Instanz der Klasse `SharedComponent` (siehe Abschnitt 6.22) und das `GReQLAdapter`-Objekt (siehe Abschnitt 6.23).

Die `HashMap`-Objekte `allUSE`, `allStatementDEF`, `allFormalInDEF` und `allActualOutDEF` sind Referenzen auf die gleichnamigen Objekte in den Singleton-Klassen `DEFSet` und `USESet`. Aus Performancegründen beinhalten diese beiden Singletons die USE- bzw. DEF-Mengen für ESDG-Knoten [Sch07]. Die protected-Methoden `isVariableUsed` und `isVariableDefined` bieten den anderen Komponenten die Möglichkeit mittels der entsprechenden `HashMap`-Instanz zu überprüfen, ob sich eine Variable in der USE- bzw. DEF-Menge eines Knotens befindet.

6.22. SharedComponent

Die SharedComponent ist eine Komponente, die Methoden sammelt, welche von mehr als einer anderen Komponente benutzt werden. Somit vermeidet sie redundanten Java-Quellcode in den Komponenten von RePST. Ihre allgemeine Funktion wurde bereits im Abschnitt 5.7 beschrieben. Dieser Abschnitt soll anhand des Klassendiagramms in Abb. 6.11 einen Überblick der durch die SharedComponent angebotenen Methoden bieten.

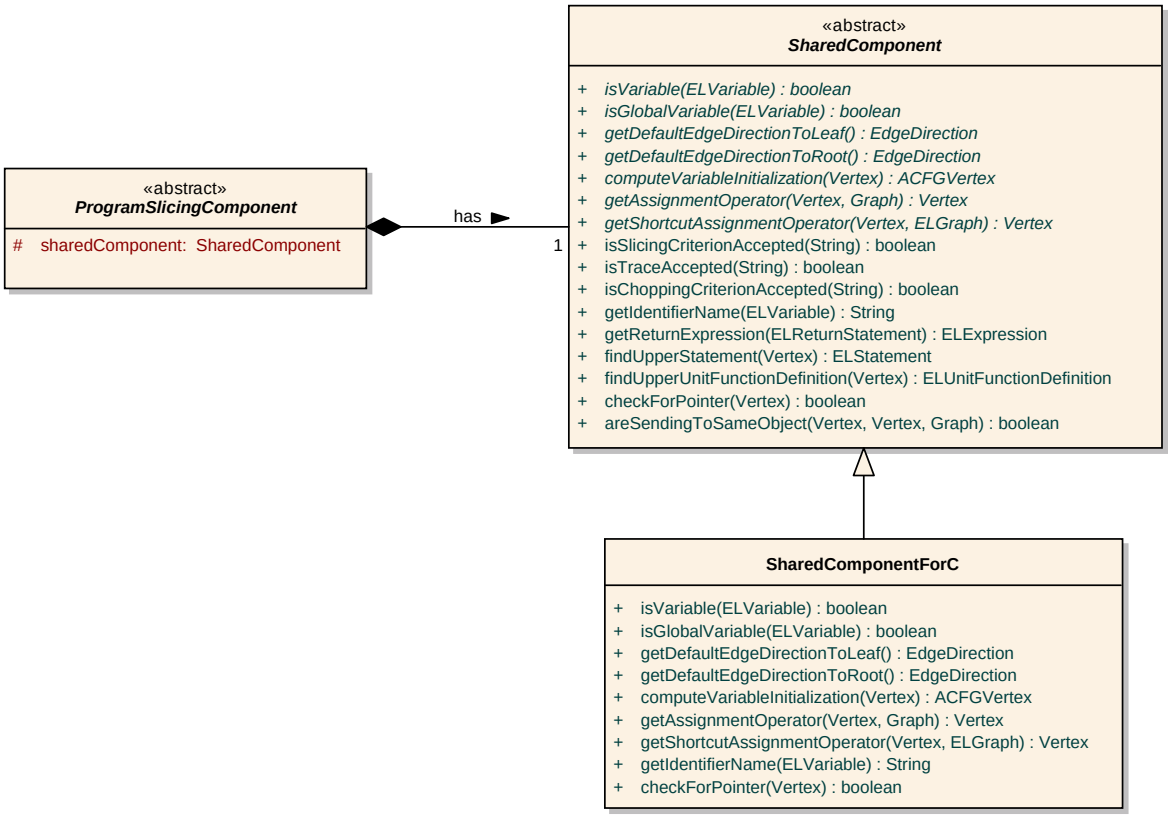

Abbildung 6.11.: Klassendiagramm: SharedComponent

In der Klasse `SharedComponent` werden public-Methoden angeboten, die entweder für das Slicen beliebiger Programmiersprachen gelten oder abstrakt sind. Die abstrakten Methoden werden dann beispielsweise zum Slicen von C-Programmen in der Klasse `SharedComponentForC` implementiert. Je nach Unterschieden zwischen Referenzschema und C-Schema müssen auch konkrete Methoden der Klasse `SharedComponent` in der `SharedComponentForC`-Klasse überschrieben werden.

Im Folgenden werden die public-Methoden der `SharedComponent` sowie eine kurze Beschreibung ihrer Funktion aufgelistet:

- `isVariable` und `isGlobalVariable`: Diese Methoden dienen zur Erkennung von Variablen bzw. globalen Variablen.

- `getDefaultEdgeDirectionToLeaf` und `getDefaultEdgeDirectionToRoot`: In Abschnitt 3.3.4 wird das Problem bezüglich der standardmäßig entgegengesetzten Kantenausrichtung zwischen den Kanten des Referenzschemas und den Kanten des C-Schemas diskutiert. Diese beiden Methoden lösen das Problem, in dem sie die Kantenausrichtung zu den Blättern bzw. der Wurzel des ASG liefern.

- `isSlicingCriterionAccepted`, `isChoppingCriterionAccepted` und `isTraceAccepted`: Diese Methoden kontrollieren das Slicing-Kriterium, das Chopping-Kriterium bzw. die Trace anhand der regulären Ausdrücke aus Tabelle 6.1.

- `getIdentifierName`: Durch diese Methode wird der Name einer `Identifier`-Instanz bestimmt und als String zurückgegeben.

- `getReturnExpression`: Zur Bestimmung des Rückgabewertes eines `ReturnStatement`-Objektes wird diese Methode eingesetzt.

- `findUpperStatement` und `findUpperUnitFunctionDefinition`: Um den `Statement`- bzw. `UnitFunctionDefinition`-Knoten mittels Breitensuche zu finden, welcher den übergebenen Knoten enthält, gibt es diese beiden Methoden.

- `checkForPointer`: Diese Methode wird zur Erkennung von Zeigern eingesetzt.

- `computeVariableInitialization`: Der Knoten, welcher innerhalb einer Deklaration für die Initialisierung einer Variable sorgt, wird durch diese Methode bestimmt.

- `getAssignmentOperator` und `getShortcutAssignmentOperator`: Diese beiden Methoden werden eingesetzt, um `AssignmentStatement`-Knoten zu erkennen. Wobei `getShortcutAssignmentOperator` speziell für die Erkennung von Zuweisungen mit abkürzenden Schreibweisen, wie ++ oder *=, genutzt wird.

- `areSendingToSameObject`: Um herauszufinden, ob zwei Aufrufanweisungen an dieselbe Objektvariable eine Botschaft senden, wird diese Methode verwendet.

6.23. GReQLAdapter

Zur Nutzung von GReQL2 [Mar06] gibt es den Adapter `GReQLAdapter`. Dieser vereinfacht die Nutzung von GReQL2, indem er die Komplexität der Schnittstellen reduziert und sämtliche GReQL2 spezifischen Klassen kapselt.

Das Klassendiagramm zum `GReQLAdapter` ist in Abb. 6.10 zu sehen. Durch die Anwendung des Singleton-Patterns verwenden alle `ProgramSlicingComponent`-Spezialisierungen dieselbe `GReQLAdapter`-Instanz. Zugriff auf dieses Objekt erhalten sie durch die statische Methode `getInstance`. Die anderen Methoden stellen die benötigte GReQL2-Funktionalität bereit:

- Die Methode `executeGReQLQuery` führt die in den Parametern übergebene Anfrage auf dem übergebenen Graphen aus und liefert als Rückgabewert eine `JValue`-Instanz [Bil06].
- Zur Vereinfachung des Bindens von Variablen an gespeicherte Anfrageergebnisse gibt es die Methode `addBoundVertex`. Jeder Aufruf verknüpft die übergebene `Vertex`-Instanz mit der durch den String-Parameter festgelegten Variable.
- Die Methode `clearBoundVertices` entfernt alle bisher gebundenen Variablen.

7. Implementation

7.1. TGraph-Anfragen

Die Implementation des in dieser Diplomarbeit entwickelten Programm Slicing Tools basiert auf TGraphen, die mittels JGraLab implementiert werden. JGraLab bietet dem Entwickler zwei Möglichkeiten, um Anfragen an TGraphen zu stellen. Zum einen können die Graphen mittels des JGraLab-APIs analysiert werden, zum anderen kann GReQL2 dazu genutzt werden.

GReQL [Kam96] ist eine Anfragesprache für TGraphen, die zur Analyse der anfallenden Datenbestände innerhalb des GUPRO Projektes [EGSW98] konzipiert wurde. GReQL2 [Mar06] ist eine Weiterentwicklung von GReQL, deren Implementation [Bil06] in die JGraLab-Bibliothek integriert ist.

7.1.1. Anfragen mit GReQL2

Anfragen, die mit *GReQL2* formuliert sind, erinnern vom Aufbau her an SQL-Anfragen [Bea06]. Sie bestehen typischerweise aus einem „from“-, einem „with“- und einem „report“-Teil. Hierbei werden im „from“-Teil Variablen deklariert, welche anschließend verwendet werden können. Von außen gebundene Variablen werden nicht im „from“-Teil deklariert, sondern müssen in der „using“-Klausel definiert werden. Der optionale „with“-Teil definiert ein Prädikat, dem die innerhalb des „from“-Teils deklarierten Variablen genügen müssen. Das Ergebnis der Anfrage wird im „report“-Teil beschrieben.

Weiterführendes zum Aufbau von GReQL2-Anfragen findet der Leser in [Mar06].

Da die Komponenten des Program Slicing Tools auf TGraphen operieren, können die Anfragen der Komponenten prinzipiell in GReQL2 formuliert werden. Zum Beispiel berechnet

```
1 USING ufd:
2 FROM acfg:V{ACFG}, rest:V{ELReturnStatement}
3 WITH ufd <--{ACFGDescribesELUnitFunctionDefinition} acfg
4       -->{ACFGContainsACFGVertex} rest
5 REPORT rest
6 END
```

Listing 7.1: GReQL2-Anfrage zur Bestimmung aller `ELReturnStatement`-Knoten innerhalb einer vorgegebenen `ELUnitFunctionDefinition`-Instanz

die Anfrage aus Listing 7.1 alle `ELReturnStatement`-Knoten `rest`, die sich in der gleichen, vorgegebenen `ELUnitFunctionDefinition`-Instanz `ufd` befinden.

Dazu werden im „*from*“-Teil die verwendeten Variablen `acfg` und `rest` deklariert. Die Typen dieser Variablen entsprechen den Klassen `ACFG` und `ELReturnStatement` in RePSS. Die von außen gebundene Variable `ufd` wird vorab in der „*using*“-Klausel definiert.

Der „*with*“-Teil stellt *Bedingungen* an die Variablen. So wird hier festgelegt, dass nur Tupel bestehend aus der vorgegebenen `ELUnitFunctionDefinition`-Instanz, einem `ACFG`- und einem `ELReturnStatement`-Knoten für das Ergebnis relevant sind, sofern der `ELReturnStatement`-Knoten per eingehender `ACFGContainsACFGVertex`-Kante mit dem `ACFG`-Knoten verknüpft ist, welcher eine ausgehende `ACFGDescribes-ELUnitFunctionDefinition` zur vorgegebenen `ELUnitFunctionDefinition`-Instanz besitzt.

Die Ergebnismenge wird anschließend durch den „*report*“-Teil gebildet. Hier werden aus den relevanten Tupeln die `ELReturnStatement`-Knoten herausgefiltert und zurückgegeben.

Listing 7.2 zeigt einen Ausschnitt aus einem Java-Programm, in dem die Beispiel-Anfrage aus Listing 7.1 ausgeführt wird. Zum Auswerten der Anfrage wird ein `GreqlEvaluator`-Objekt erzeugt. Dieses Objekt benötigt die Anfrage als `String`, eine `Graph`-Instanz und eine `HashMap`-Instanz, die eventuell gebundene Variablen enthält. Die Methode `startEvaluation` führt die Berechnung der Anfrage aus und mittels der Methode `getEvaluationResult` werden die Anfrage-Ergebnisse als `JValue`-Objekt abgefragt.

Die in dieser Diplomarbeit entwickelten Komponenten besitzen eine einfachere Schnittstelle zu GReQL2, die von der Klasse `GReQLAdapter` bereitgestellt wird. Die abstrakte Klasse

```
String query = "USING␣ufd:␣" +
  "FROM␣acfg:V{ACFG},␣rest:V{ELReturnStatement}␣" +
  "WITH␣ufd␣<−−{ACFGDescribesELUnitFunctionDefinition}␣acfg␣" +
  "−−>{ACFGContainsACFGVertex}␣rest␣" +
  "REPORT␣rest␣" +
  "END";

Graph graph = getGraph();
HashMap<String, JValue> boundVariables = new HashMap<String, JValue>();
boundVariables.put("ufd", new JValue(getUnitFunctionDefinition(graph)));

GreqlEvaluator greqlEvaluator = new GreqlEvaluator(query, graph, boundVariables);
greqlEvaluator.startEvaluation();
JValue result = greqlEvaluator.getEvaluationResult();
```

Listing 7.2: Java-Code zur Ausführung einer GReQL2-Anfrage

`ProgramSlicingComponent` vererbt allen anderen RePST-Komponenten den Zugriff auf die `GReQLAdapter`-Instanz. Siehe dazu die Abschnitte 6.21 und 6.23.

Listing 7.3 zeigt erneut die Beispiel-Anfrage aus Listing 7.1. Jedoch werden diesmal die Methoden der `GReQLAdapter`-Klasse verwendet. Die Methoden `executeGReQLQuery` und `addBoundVertex` kapseln dabei alle GReQL2-Ausführungsdetails, die den Code von Listing 7.2 wesentlich komplexer machen.

7.1.2. Algorithmen mit dem RePSS-API

Als Alternative zu Anfragen mit GReQL2 gibt es das *RePSS-API*, welches mit Hilfe von JGraLab generiert wird. JGraLab ist die Weiterentwicklung und Implementation von GraLab [DW98] mittels der Programmiersprache Java. Näheres dazu findet man in [Kah06].

Die Beispiel-Anfrage aus Listing 7.1 zur Berechnung der `ELReturnStatement`-Knoten innerhalb eines vorgegebenen `ELUnitFunctionDefinition`-Objektes kann auch auf direktem Weg durch Nutzung der RePSS-API geschehen. Dies ist in Listing 7.4 gezeigt.

Das RePSS-API bietet dem Entwickler verschiedene Möglichkeiten, um über die Knoten und Kanten eines Graphen zu iterieren. Im Beispiel aus Listing 7.4 werden die `ACFG-`

```
String query = "USING␣ufd:␣" +
  "FROM␣acfg:V{ACFG},␣rest:V{ELReturnStatement}␣" +
  "WITH␣ufd␣<−−{ACFGDescribesELUnitFunctionDefinition}␣acfg␣" +
  "−−>{ACFGContainsACFGVertex}␣rest␣" +
  "REPORT␣rest␣" +
  "END";

GReQLAdapter greqlAdapter = GReQLAdapter.getInstance();
Graph graph = getGraph();
greqlAdapter.addBoundVertex("ufd", getUnitFunctionDefinition(graph));
JValue result = greqlAdapter.executeGReQLQuery(query, graph);
```

Listing 7.3: Java-Code zur Ausführung einer GReQL2-Anfrage mit dem GReQL-Adapter

```
Graph graph = getGraph();
ELUnitFunctionDefinition unitFunctionDefinitionVertex = getUnitFunctionDefinition(graph);

ACFG acfgVertex = (ACFG)unitFunctionDefinitionVertex.
      getFirstACFGDescribesELUnitFunctionDefinition().getAlpha();
ACFGContainsACFGVertex containsEdge = acfg.getFirstACFGContainsACFGVertex();
while(containsEdge != null) {
      if (containsEdge.getOmega() instanceof ELReturnStatement) {
            ELReturnStatement returnStatementVertex =
                  (ELReturnStatement)containsEdge.getOmega();
            // do something with returnStatementVertex...
      }
      containsEdge = containsEdge.getNextACFGContainsACFGVertex();
}
```

Listing 7.4: Java-Code zur Nutzung des RePSS-APIs

`ContainsACFGVertex`-Kanten ihrer Anordnung zu dem `ACFG`-Knoten, der mit dem vorgegebenen `ELUnitFunctionDefinition`-Objekt verknüpft ist, entsprechend mittels einer while-Schleife und den beiden Methoden `getFirstACFGContainsACFGVertex` und `getNextACFGContainsACFGVertex` der Reihe nach durchlaufen. Sofern der Endpunkt einer `ACFGContainsACFGVertex`-Kante vom Typ `ELReturnStatement` ist, handelt es sich um einen der gesuchten `ELReturnStatement`-Knoten. Dies wird mit Hilfe der Methode `getOmega` und des `instanceof`-Operators überprüft.

7.1.3. Bewertung der TGraph-Anfragen

Wie in diesem Kapitel gezeigt, gibt es zwei verschiedene Möglichkeiten zur Informationsextraktion aus Graphen. Aufgrund ihrer unterschiedlichen Mächtigkeit ist ein Vergleich beider Ansätze schwierig. Sofern man nur die Teile der Algorithmen betrachtet, welche auch mit GReQL2 in Form von Anfragen realisierbar sind, stellt man fest, dass Algorithmen mit dem RePSS-API programmiersprachennah sind, während GReQL2-Anfragen eine abstraktere Darstellung für Anfragen bieten.

Während der Entwicklung hat sich herausgestellt, dass das RePSS-API effizienter und schneller als sein GReQL2-Gegenstück ist, weil dieses in der Version von [Bil06] aufgrund des fehlenden Anfrageoptimierers teilweise starke Performanceverluste erfährt. Aktuell wird in [Hor08] ein Anfrageoptimierer, wie man ihn beispielsweise für SQL aus [KE04] kennt, für GReQL2 entwickelt.

Das RePSS-API bietet dem Entwickler zusätzlich Möglichkeiten zur Manipulation des TGraphen. So können Knoten und Kanten dem zugrundeliegenden Schema entsprechend erstellt, gelöscht oder bearbeitet werden. Diese Möglichkeiten zur Manipulation bietet GReQL2 nicht. Dadurch ist das RePSS-API wesentlich mächtiger als GReQL2.

GReQL2 bietet dem Benutzer eine Anfragesprache auf einem programmiersprachenunabhängigen Niveau. Dadurch sind GReQL2-Anfragen verständlicher und leichter zu warten. Allerdings brauchen GReQL2-Anfragen mehr Speicher als vergleichbare Anfrage-Algorithmen mit Hilfe des RePSS-APIs.

7.1.4. GReQL2-Nutzung in RePST

Prinzipiell soll GReQL2 in den Komponenten dieser Diplomarbeit nutzbar sein. Daher gibt es auch die Klasse `GReQLAdapter`. Jedoch kommt GReQL2 nicht in allen Komponenten zum Einsatz. So wird in den Komponenten, die zwingend auf das Schema der zu slicenden Programmiersprache zugreifen müssen, primär das RePSS-API für Anfragen an TGraphen genutzt, da GReQL2 hier erhöhte Redundanz produzieren würde. Diese Redundanz entsteht, weil GReQL2-Anfragen komplett neu geschrieben werden müssen, wenn sich zum Beispiel der Typ einer einzelnen Variablen ändert. Im Vergleich dazu muss in Algorithmen durch den Einsatz des Template Method Patterns [GHJV95] lediglich eine primitive Methode, welche den Typ kapselt, überschrieben werden. In den übrigen Komponenten, die ausschließlich auf RePSS arbeiten, werden Anfragen in erster Linie mit GReQL2 formuliert, sofern sie eine gewisse Komplexität besitzen.

Abschließend soll ein kompakter Überblick bezüglich der GReQL2-Nutzung in den einzelnen Komponenten von RePST gegeben werden. Dazu werden die Komponenten in fünf Kategorien eingeteilt, welche den Umfang der GReQL2-Nutzung beschreiben:

- Kategorie 1: Komponenten, deren Kernfunktion vollständig durch GReQL2-Anfragen umgesetzt wird. Hierbei ist zu beachten, dass Manipulationen des TGraphen trotzdem mit dem RePSS-API umgesetzt werden müssen. In die erste Kategorie gehören die Komponenten IntMetEdgesComputer, StaticForwardSliceComputer und StaticBackwardSliceComputer.

- Kategorie 2: Komponenten, deren Kernfunktion algorithmisch umgesetzt wird, wobei der Algorithmus mittleren Gebrauch von GReQL2-Anfragen macht. Die zweite Kategorie enthält den DefUseInfComputer, den BasicESDGComputer, den DataFlowEdgesComputer und den ESDGMarker.

- Kategorie 3: Komponenten, deren Kernfunktion algorithmisch umgesetzt wird, wobei der Algorithmus marginalen Gebrauch von GReQL2-Anfragen macht. Zur dritten Kategorie zählen die Komponenten ConDepEdgesComputer, SumEdgesComputer und ESDGToCodeConverter.

- Kategorie 4: Komponenten, die GReQL2 nicht verwenden. In dieser Kategorie befinden sich der ASGComputer, der ACFGComputer, der PGComputer und der CGComputer.

- Kategorie 0: Komponenten, die Fassaden für andere Komponenten sind und somit indirekt GReQL2-Anfragen verwenden. Hierzu gehören das ProgramSlicingTool, der ProgramPreprocessor, der ECGComputer, der ESDGComputer und der ProgramSlicer.

7.2. Unittests gegen ein C-Testprogramm

Im Rahmen der Qualitätssicherung wird jede Komponente, die keine Fassade ist, durch einen eigenen Unittest kontrolliert. Diese Unittests sind so systematisch aufgebaut, dass sie nicht nur die Komponenten selbst testen, sondern auch in ihrer Gesamtheit den kompletten Slicing-Vorgang verifizieren. Die Funktionalität des kompletten Systems und aller Fassaden wurde durch Systemtests bestätigt.

Dieser Abschnitt beschreibt das *Slicen eines C-Beispielprogramms*, indem die Ergebnisse der einzelnen Unittests als Zwischenschritte vorgestellt werden. Die *Unittests* sind mit Hilfe von JUnit 4 [Dos05] implementiert. Da sich alle Testklassen im Verzeichnis „testit“ befinden, ist eine Test-Suite aufgrund der Verwendung von JUnit 4 nicht nötig.

Die Testklassen führen verschiedene Tests aus, wobei für jede Komponente in der Regel folgendes getestet wird:

- Es wird getestet, ob die Ausführung der Komponente neue Knoten und Kanten im übergebenen Graphen erzeugt hat.
- Die Anzahl der erzeugten Knoten oder Kanten wird mit der Anzahl der erwarteten Kanten eines bestimmten Typs (z.B. `ACFGVertexHasControlFlowEdgeTo-ACFGVertex`) verglichen.
- Eventuell werden im Test allgemeingültige Konsistenzprüfungen durchgeführt.
- Alle mittels komplexer Berechnung erzeugten Knoten oder Kanten werden in der Konsole zum manuellen Überprüfung ausgegeben.

Die beiden Tools GCPP und GCPA des ASGComputers verwenden GraLab zur Konstruktion des ASG. GraLab weist jedem erzeugten Knoten und jeder erzeugten Kante eine eindeutige ID zu, die einer natürlichen Zahl größer Null entspricht. Die Objektdiagramme in diesem Abschnitt verwenden als Namen der Objekte diese eindeutige ID. Falls ein solches Objekt einer Zeile im Beispielprogramm zugeordnet werden kann, so wird diese Zeilennummer zum besseren Verständnis der Diagramme in spitzen Klammern angegeben.

Die Objektdiagramme in diesem Abschnitt entsprechen Ausprägungen der Schemata aus Kapitel 3. Zur Vereinfachung werden Knoten und Kanten ausgeblendet, die ohne aufwändige Berechnung erzeugt werden. So werden beispielsweise *contains*-Kanten nicht visualisiert, da jedes Objekt eines ACFG, eines ECG oder eines ESDG eine eingehende `ACFGContainsACFGVertex`-, eine `ECGContainsECGVertex`- oder eine `ESDGContainsESDGVertex`-Kante besitzt. Ausprägungen des ESDG werden in mehrere kleine Graphen zerlegt, um den Test der entsprechenden Komponente klarer wiederzugeben.

7.2.1. Testprogramm

Im Interface `ProgramSlicingForCTestInterface` wird ein C-Programm für die Tests festgelegt, so dass alle Testklassen dieses Interface implementieren müssen, um die zu erwartenden Werte des C-Programms zu kennen. Außerdem kann das C-Testprogramm in diesem Interface an einer Stelle ausgetauscht werden, so dass die einzelnen Testklassen nicht verändert werden müssen, wenn ein neues Beispielprogramm genutzt wird.

RePST wurde gegen verschiedene C-Programme getestet. In diesem Abschnitt werden die Tests gegen ein ausgewähltes C-Programm gezeigt, welches viele verschiedene Anweisungen besitzt, obwohl es relativ klein ist. Dieses C-Testprogramm ist in Listing 7.5 aufgeführt.

```
1  int aGlobal;
2  int bGlobal;
3
4  void init() {
5      aGlobal = 4;
6      bGlobal = 1 + 3 + aGlobal − 3 − 2;
7  }
8
9  int m2(int cParam) {
10     int cLocal = 5;
11     cLocal ∗= aGlobal;
12     return cLocal;
13 }
14
15 void m1(int aParam, int bParam) {
16     while (bParam < aParam) {
17         jumpLabel: bParam = bParam + 1;
```

```
18              if (bParam <= 2) {
19                  continue;
20              }
21              if (bGlobal == 3) {
22                  goto jumpLabel;
23              }
24              if (m2(aParam) >= 4) {
25                  break;
26              }
27          }
28  }
29
30  int main() {
31          int aLocal, bLocal;
32          init();
33          if (aGlobal > 0) {
34              aLocal = aGlobal + bGlobal;
35              m1(aLocal, 1);
36              return aLocal;
37          } else {
38              aLocal = -1;
39              return -1;
40          }
41  }
```

Listing 7.5: C-Testprogramm

7.2.2. ASGComputerForCTest

Der Test des ASGComputers besitzt eine Sonderstellung, da die eigentliche Funktionalität, also das Parsen von C-Quellcode, nicht in dieser Diplomarbeit entwickelt wurde (siehe Abschnitt 6.3). Deshalb überprüft der Test lediglich die Ausführung der externen Tools GCPP, GCPA und G2TG und nicht deren Korrektheit. Aus diesem Grund wird auch auf die Darstellung des ASG verzichtet.

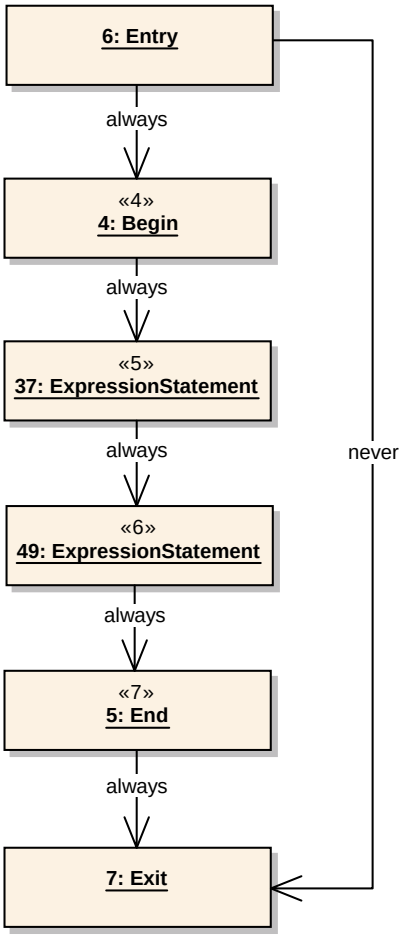

Abbildung 7.1.: ACFG der Methode `init`

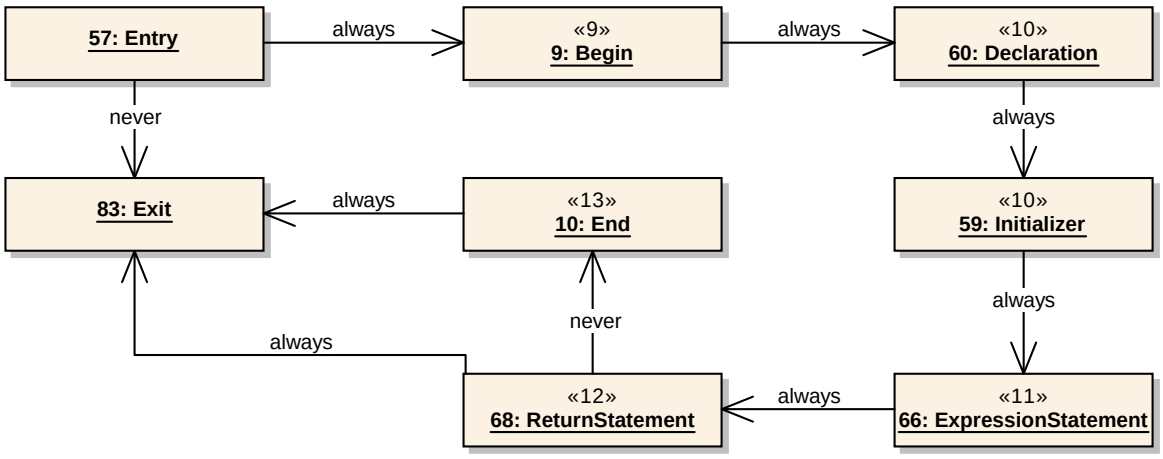

Abbildung 7.2.: ACFG der Methode `m2`

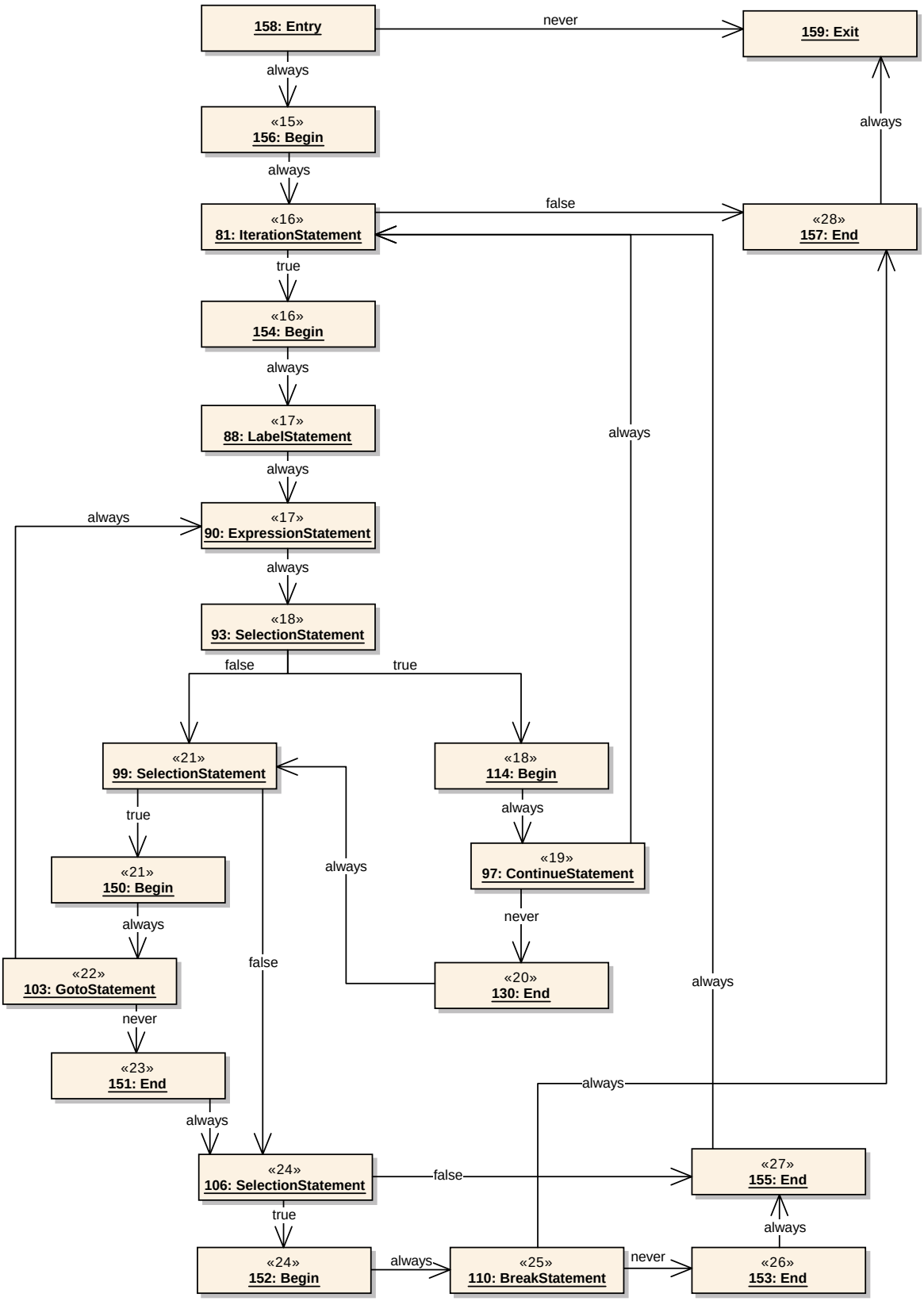

Abbildung 7.3.: ACFG der Methode `m1`

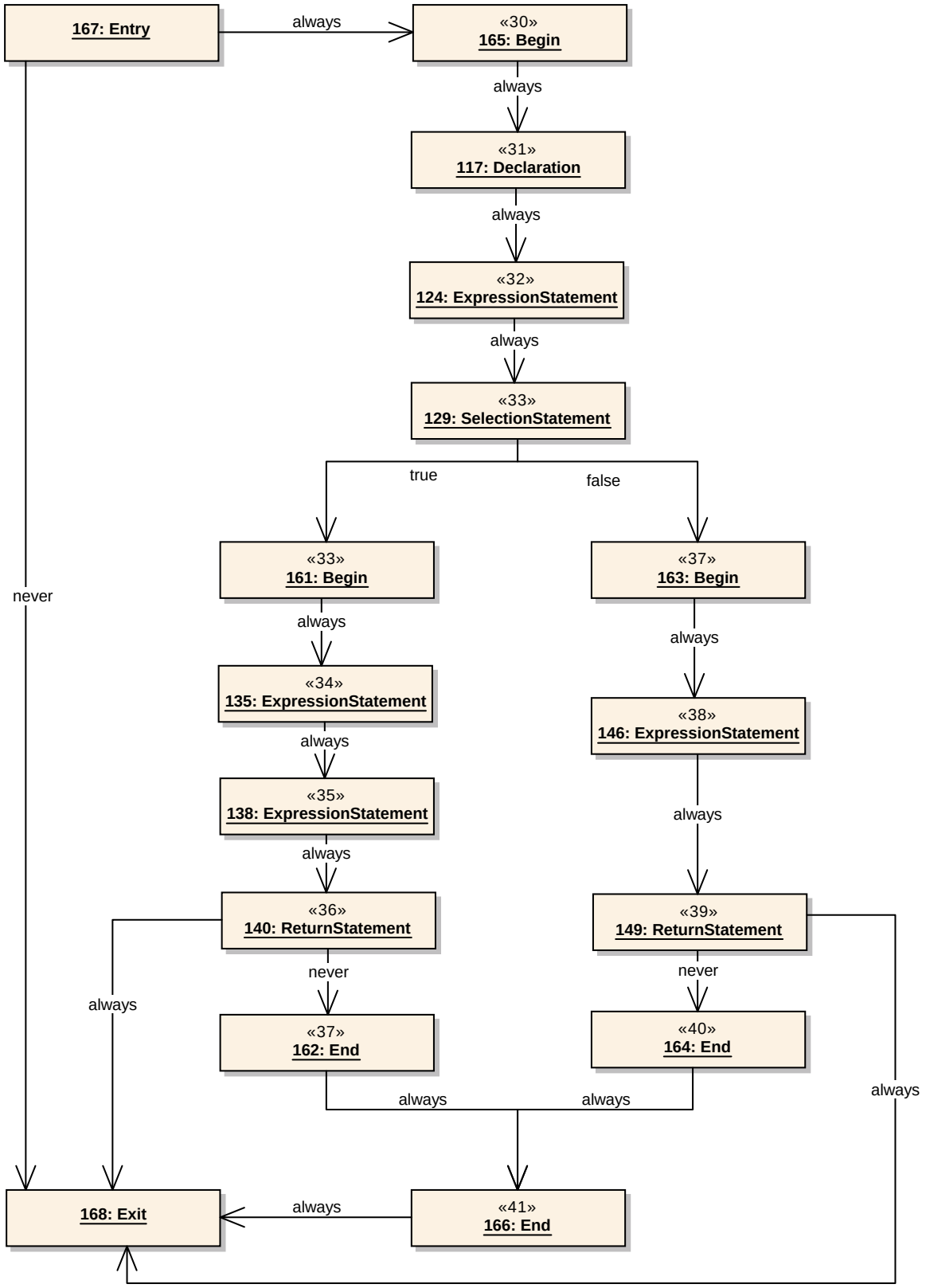

Abbildung 7.4.: ACFG der Methode `main`

7.2.3. ACFGComputerForCTest

Für die berechneten ACFGs wird durch den Test des ACFGComputers kontrolliert, ob jeder ACFG-Knoten maximal zwei ausgehende Kontrollflusskanten besitzt. Außerdem wird der Anzahl der erwarteten Kontrollflusskanten die Anzahl der tatsächlich berechneten entgegengesetzt. Für das Testprogramm aus Listing 7.5 werden insgesamt 62 Kontrollflusskanten mit beliebigem Label erwartet. Die Abb. 7.1, 7.2, 7.3 und 7.4 zeigen die erwarteten ACFGs des C-Beispielprogramms. Hierbei werden die möglichen Labels der Kontrollflusskanten (*always*, *true*, *false* oder *never*) als Beschriftung an den Kanten dargestellt.

Der ACFGComputer ermittelt für das Testprogramm die geforderten 62 Kontrollflusskanten und beschreibt deren Verlauf entsprechend den Abb. 7.1, 7.2, 7.3 und 7.4. Zusätzlich kann man den Darstellungen der ACFGs entnehmen, dass die Ausführung des ACFGComputers die Konsistenzbedingung wahrt, dass die ACFG-Knoten vom Typ `Entry`, `JumpStatement` oder `CompoundStatementWithTest` genau zwei und alle anderen nicht mehr als eine ausgehende Kontrollflusskante besitzen.

7.2.4. PGComputerForCTest

Durch den PGComputer werden Variablen entsprechend ihrem Vorkommen Funktionen zugeordnet. Dabei ist es egal, ob es sich um Parameter-, lokale oder globale Variablen handelt. Da das C-Testprogramm keine Zeiger enthält, benötigt man keine Pointeranalyse, um die in einer Methode verwendeten Variablen zu erkennen. Listing 7.5 kann man entnehmen, dass durch alle vier Methoden insgesamt 12 `PGContainsVariable`-Kanten erzeugt werden müssen. Tabelle 7.1 zeigt neben diesen 12 Variablen zusätzlich die Funktionen, welche die 12 Variablen enthalten. Der PGComputer wird durch die Erfüllung dieser erwarteten Beziehungen zwischen Funktionen und Variablen im Test validiert.

Funktion	**PG-Menge**
main	{aLocal, bLocal, aGlobal, bGlobal}
init	{aGlobal, bGlobal}
m1	{bGlobal, aParam, bParam}
m2	{cLocal, aGlobal, cParam}

Tabelle 7.1.: PG-Mengen des Testprogramms aus Listing 7.5

Die Berechnung der „zeigt auf“-Beziehungen wird im Test des PGComputers vernachlässigt, weil diese Funktionalität nur rudimentär implementiert ist.

7.2.5. CGComputerForCTest

Der Test des CGComputers erwartet für Listing 7.5 jeweils 3 `UnitFunctionDefinitionContainsCall`- und 3 `CallCanCallUnitFunctionDefinition`-Kanten. Dies begründet sich dadurch, dass das Testprogramm an drei Stellen `Call`-Instanzen verwendet und dass aufgrund der nicht vorhandenen Polymorphie in C eine `Call`-Instanz genau eine Funktion aufruft.

Abb. 7.5 visualisiert unter anderem die Ergebnisse des CGComputers. Dabei ist zu beachten, dass `UnitFunctionDefinitionContainsCall`- bzw. `CallCanCallUnitFunctionDefinition`-Kanten aus Platzgründen durch Kanten mit der Beschriftung „contains“ bzw. „canCall“ vertreten werden.

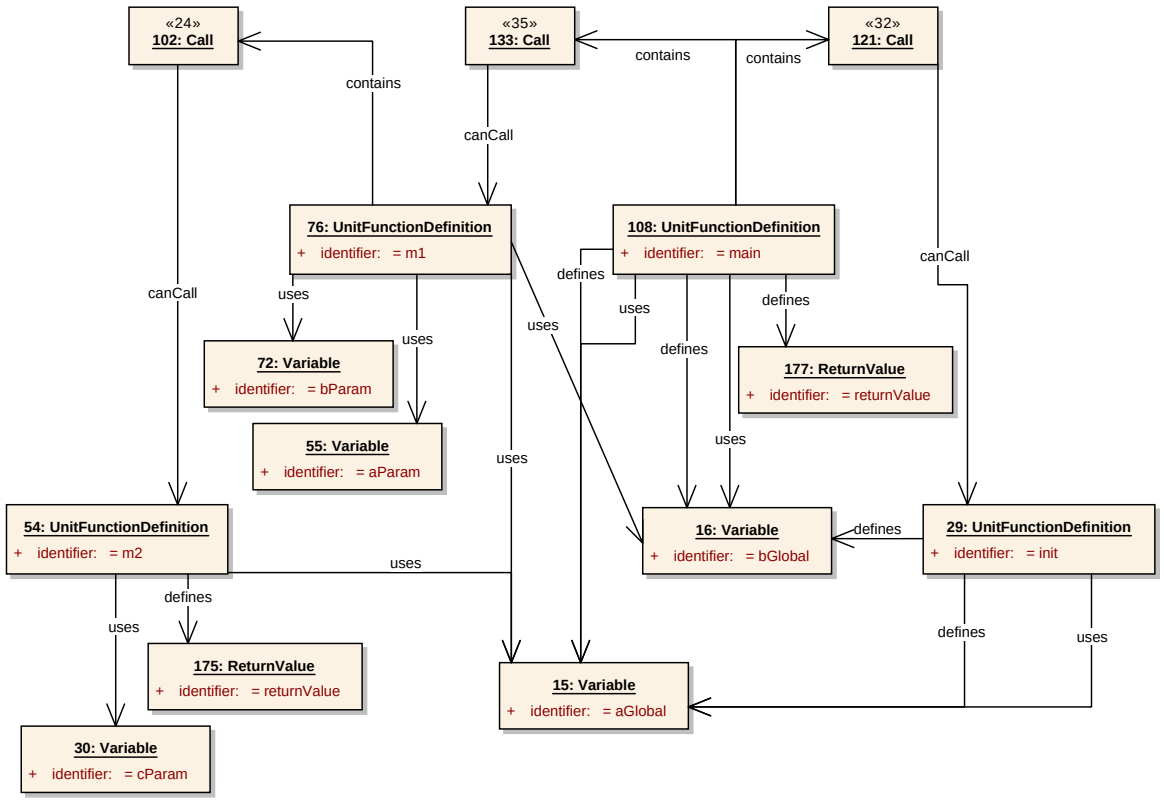

Abbildung 7.5.: ECG des C-Programms aus Listing 7.5

7.2.6. DefUseInfComputerForCTest

Ebenfalls im ECG aus Abb. 7.5 sind die Ergebnisse des DefUseInfComputers enthalten. Diese stimmen mit den Erwarteten überein, da man im Testprogramm erkennt, dass insgesamt 9 „uses“-Beziehungen und 6 „defines“-Beziehungen zwischen den `UnitFunctionDefinition`- und `Variable`-Objekten existieren.

7.2.7. BasicESDGComputerForCTest

Der BasicESDGComputer schafft das Grundgerüst des ESDG. Im Test wird kontrolliert:

- ob alle `Entry`-Knoten per `EntryCorrespondsToUnitFunctionDefinition`-Kante mit einem `UnitFunctionDefinition`-Knoten verbunden sind,
- ob jede `ParameterVertex`-Instanz mindestens 3 (vergleiche Abb. 3.8) ein- oder ausgehende ESDG-Kanten besitzt,
- ob für jede `UnitFunctionDefinitionDefinesVariable`-Kanten genau ein `FormalOutParameterVertex`-Objekt existiert,
- ob für jede `UnitFunctionDefinitionUsesVariable`-Kanten genau ein `FormalInParameterVertex`-Objekt existiert,
- ob die Anzahl der tatsächlich erzeugten `ActualOutParameterVertex`- und `ActualInParameterVertex`-Knoten der Anzahl der Erwarteten entspricht,
- und ob alle `ReturnValue`-Objekte korrekt mit `ReturnStatement`-Objekten verknüpft sind.

Eine kompakte Auflistung aller `ParameterVertex`-Instanzen, der durch eine `ESDGVertexHasParameterEdgeToParameterVertex`-Kante an sie gebundenen ESDG-Knoten und die durch das zugehörige `AssignmentStatement` definierten und als Definition benutzten Knoten für das Testprogramm findet der Leser in Tabelle 7.2. Dabei stehen „Xin“ und „Xout“ für die Namen erzeugter Hilfsvariablen und in den Klammern steht als ergänzende Informationen die Zeilennummer. Die ID des Knotens steht vor dem Typ bzw. vor der Bezeichnung im Fall von Variablen oder Konstanten.

ParameterVertex	**ESDGVertex**	**defines**	**hasAsDefinition**
180: FormalIn	6: Entry (4)	15: aGlobal	182: Xin
184: ActualIn	124: Expr.Stmt. (32)	182: Xin	15: aGlobal
186: FormalOut	6: Entry (4)	188: Xout	15: aGlobal
190: ActualOut	124: Expr.Stmt. (32)	15: aGlobal	188: Xout
192: FormalOut	6: Entry (4)	194: Xout	16: bGlobal
196: ActualOut	124: Expr.Stmt. (32)	16: bGlobal	194: Xout
198: FormalIn	57: Entry (9)	30: cParam	200: Xin
202: ActualIn	106: Sel.Stmt. (24)	200: Xin	55: aParam
204: FormalIn	57: Entry (9)	15: aGlobal	206: Xin
208: ActualIn	106: Sel.Stmt. (24)	206: Xin	15: aGlobal
210: FormalOut	57: Entry (9)	212: Xout	175: returnValue
214: ActualOut	106: Sel.Stmt. (24)	175: returnValue	212: Xout
216: FormalIn	158: Entry (15)	55: aParam	218: Xin
220: ActualIn	138: Expr.Stmt. (35)	218: Xin	113: aLocal
222: FormalIn	158: Entry	72: bParam	224: Xin
226: ActualIn	138: Expr.Stmt. (35)	224: Xin	35: '1'
228: FormalIn	158: Entry (15)	16: bGlobal	230: Xin
232: ActualIn	138: Expr.Stmt. (35)	230: Xin	16: bGlobal
234: FormalIn	158: Entry (15)	15: aGlobal	236: Xin
238: ActualIn	138: Expr.Stmt. (35)	236: Xin	15: aGlobal
240: FormalIn	167: Entry (30)	15: aGlobal	242: Xin
244: FormalIn	167: Entry (30)	16: bGlobal	246: Xin
248: FormalOut	167: Entry (30)	250: Xout	177: returnValue
252: FormalOut	167: Entry (30)	254: Xout	15: aGlobal
256: FormalOut	167: Entry (30)	258: Xout	16: bGlobal

Tabelle 7.2.: `ParameterVertex`-Instanzen des Testprogramms aus Listing 7.5

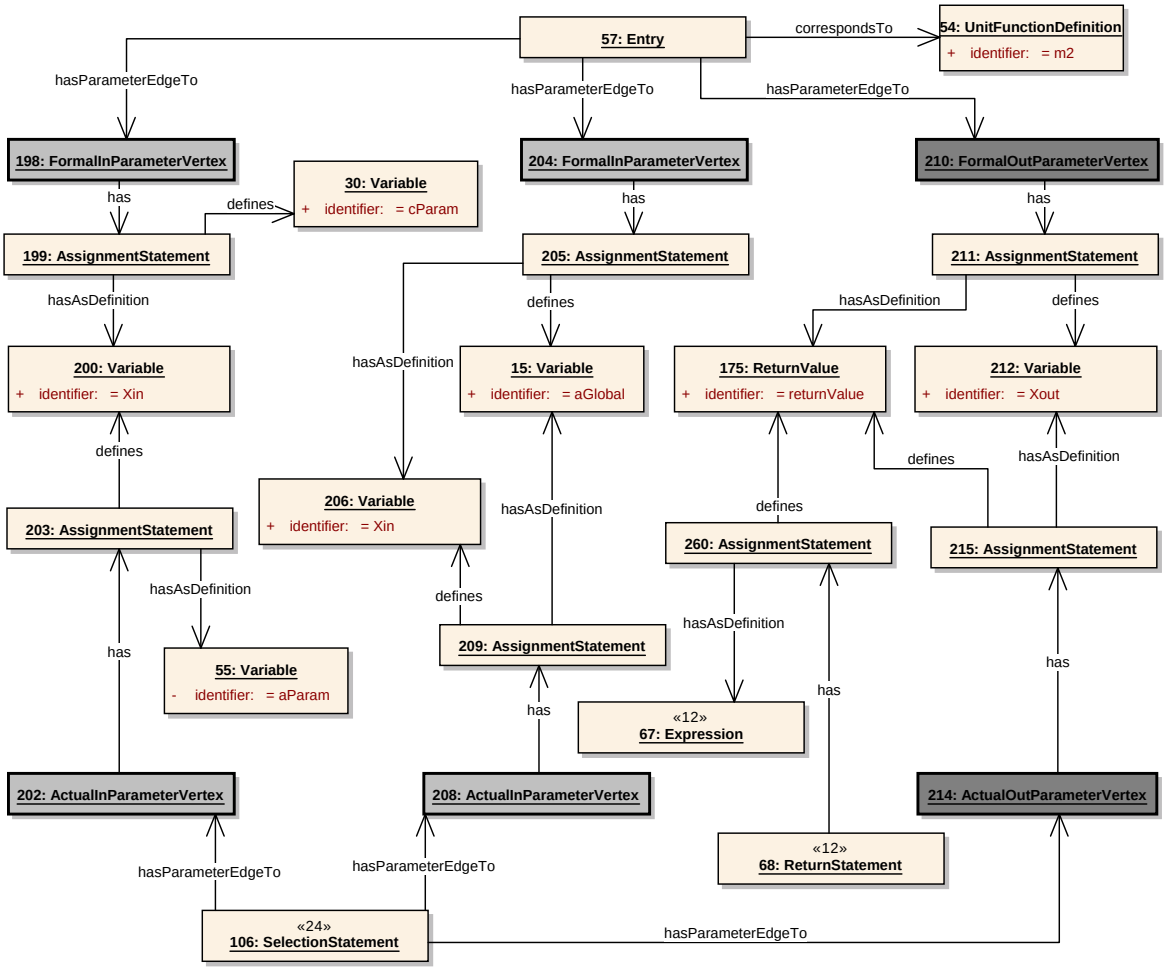

Abbildung 7.6.: Grundgerüst des ESDG für die Methode `m2` aus Listing 7.5

Anhand von Tabelle 7.2 ist es möglich Graphen zu konstruieren, welche die durch den BasicESDGComputer erzeugten ESDG-Ausschnitte verdeutlichen. In Abb. 7.6 ist dies exemplarisch für die relevanten Knoten und Kanten der Funktion `m2` des C-Beispielprogramms vorgeführt. Es ist zu beachten, dass die Beschriftungen der Kanten aus Gründen der Übersichtlichkeit auf Prä- und Postfixe verzichten. Zusätzlich werden `ParameterVertex`-Knoten farblich hervorgehoben.

7.2.8. IntMetEdgesComputerForCTest

Die interprozeduralen Kanten werden durch den Test der Komponente IntMetEdgesComputer verifiziert. Für Listing 7.5 gibt es insgesamt 3 `StatementHasCallEdgeToEntry`-Kanten. Außerdem muss es für jede `ActualInParameterVertex`-Instanz bzw. `ActualOutParameterVertex`-Instanz eine korrespondierende `FormalInParameterVertex`-Instanz bzw. `FormalOutParameterVertex`-Instanz geben, die per Parameter-in-Kante bzw. Parameter-out-Kante verbunden sind.

In Abb. 7.6 werden demnach folgende interprozeduralen Kanten durch den IntMetEdgesComputer ergänzt:

- `106: SelectionStatement` wird per `StatementHasCallEdgeToEntry`-Kanten mit `57: Entry` verknüpft.
- `202: ActualInParameterVertex` bildet mit `198: FormalInParameterVertex` ein per `ActualInParameterVertexHasParameterInEdgeToFormalInParameterVertex`-Kante verlinktes Paar.
- `208: ActualInParameterVertex` bildet mit `204: FormalInParameterVertex` ein per `ActualInParameterVertexHasParameterInEdgeToFormalInParameterVertex`-Kante verlinktes Paar.
- `210: FormalOutParameterVertex` bildet mit `214: ActualOutParameterVertex` ein per `FormalOutParameterVertexHasParameterOutEdgeToActualOutParameterVertex`-Kante assoziiertes Paar.

7.2.9. ConDepEdgesComputerForCTest

Im Test des ConDepEdgesComputer wird die Anzahl der erzeugten Kontrollkanten inspiziert. Es wird für alle vier Methoden des Testprogramms insgesamt mit 33 *true*-gelabelten und 19 *false*-gelabelten Kontrollkanten gerechnet. Wobei die Methode `m1` 16 der 33 *true*- und 11 der 19 *false*-Kanten enthält, siehe dazu Abb. 7.7. Außerdem befinden sich die 5 *true*- und 1 *false*-Kanten von `m2` in Abb. 7.8.

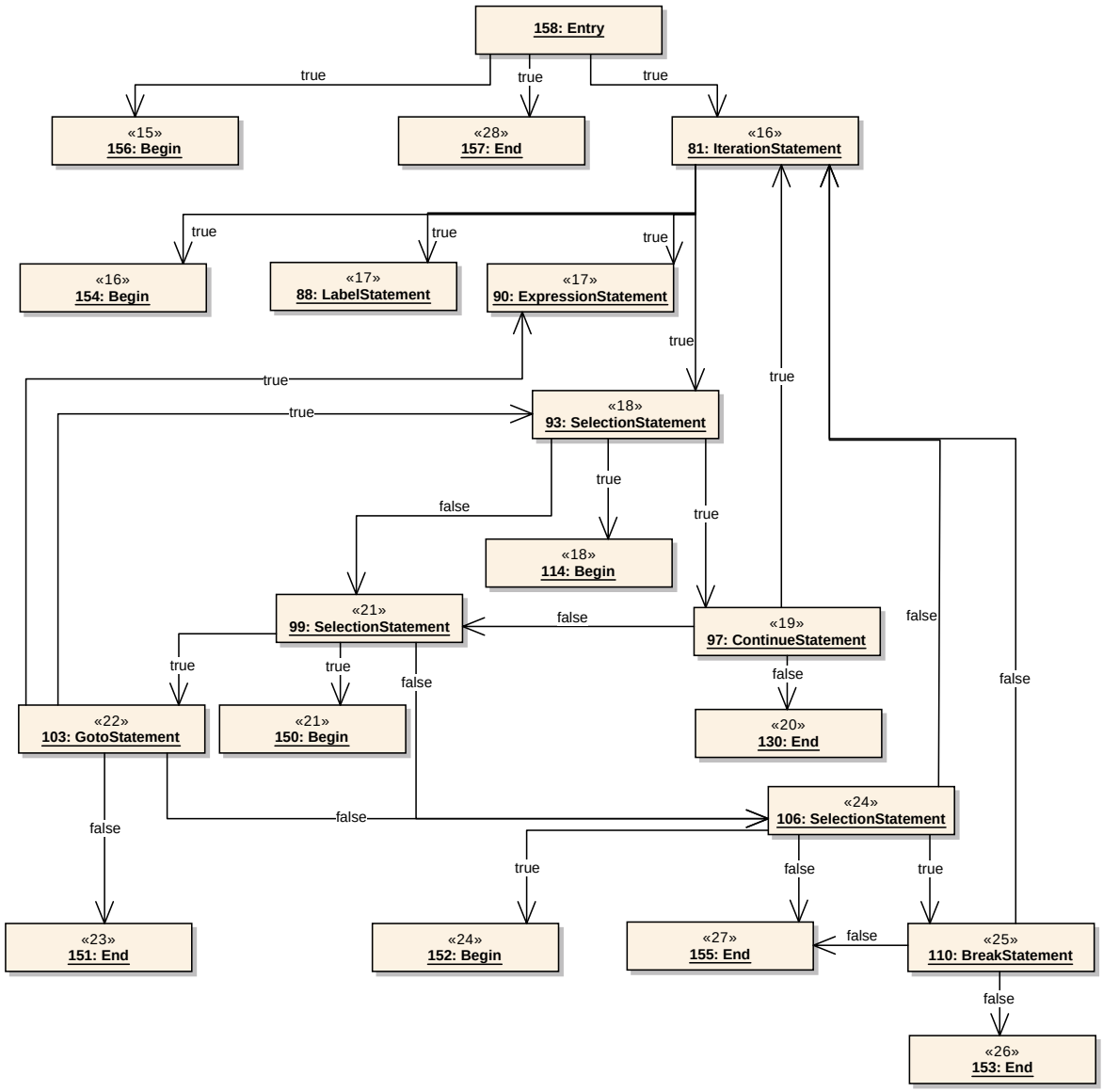

Abbildung 7.7.: Kontrollkanten der Methode `m1` aus Listing 7.5

Zur Kontrolle kann der Leser die Kontrollkanten der beiden anderen Methoden, wie in Unterabschnitt 6.12 beschrieben, relativ einfach aus den Abb. 7.1 und 7.4 der ACFGs herleiten.

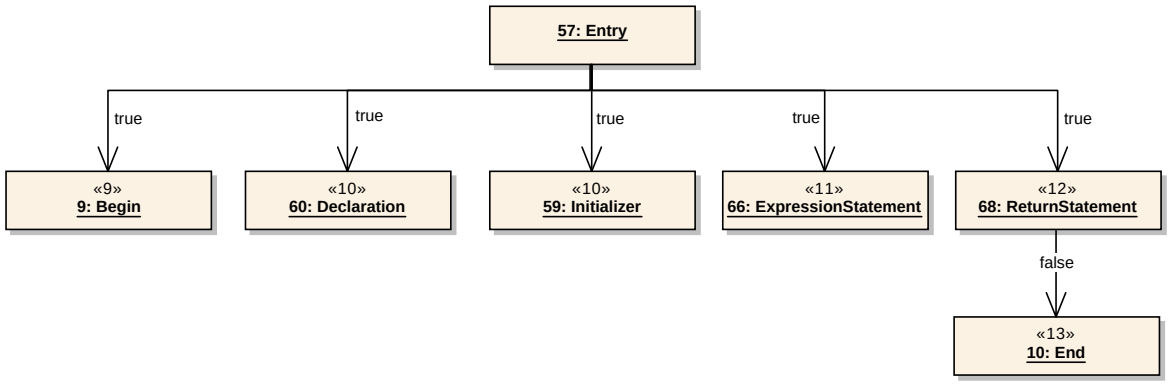

Abbildung 7.8.: Kontrollkanten der Methode `m2` aus Listing 7.5

7.2.10. DataFlowEdgesComputerForCTest

Aufgrund der hohen Komplexität des DataFlowEdgesComputers wird dessen Test mit Hilfe der sehr kompakten, aber repräsentativen Funktion `m2` des Testprogramms veranschaulicht. Man betrachte dazu Abb. 7.9 und beachte, dass der `FormalInParameterVertex`-Knoten „204“ mit dem `Variable`-Knoten für „aGlobal“ in Abb. 7.6 verbunden ist. Diese Funktion enthält nämlich eine Deklarationskante (*dec-use*) und 4 Datenflusskanten (*def-use*) mit jeweils unterschiedlichen Typkombinationen für Start- und Endpunkt.

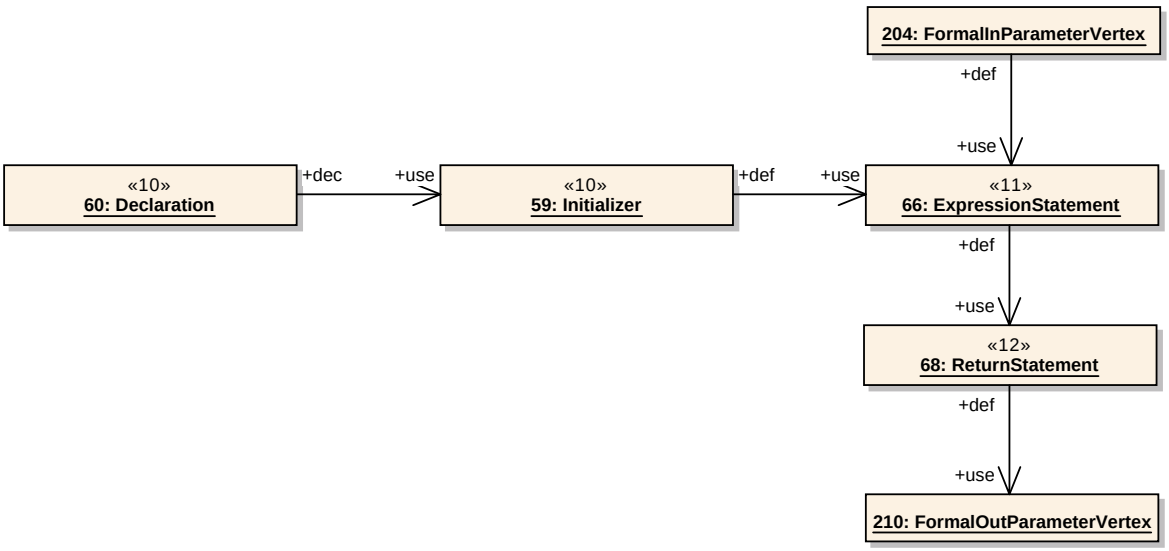

Abbildung 7.9.: Datenflusskanten der Methode `m2` aus Listing 7.5

Insgesamt müssen vom DataFlowEdgesComputer für Listing 7.5 3 Deklarationskanten und 34 Datenflusskanten erzeugt werden, wobei `main` 2 Deklarations- und 18 Datenfluss-, `init` 4 Datenfluss-, `m1` 8 Datenfluss- und `m2` 1 Deklarations- und 4 Datenflusskanten enthält.

7.2.11. SumEdgesComputerForCTest

Die Erzeugung transitiver Kanten wird durch den Unittest des SumEdgesComputers kontrolliert.

Insgesamt wird für das Testprogramm aus Listing 7.5 eine transitive Kante erwartet. Diese Kante verknüpft den `ActualInParameterVertex`-Knoten „208“ mit dem `ActualOutParameterVertex`-Knoten „214“ aus Abb. 7.6. Diese Kante repräsentiert den Einfluss der Variable `aGlobal` auf den Rückgabewert der Methode `m2`.

7.2.12. ESDGMarkerForCTest

Der ESDGMarker wird anhand 8 verschiedener Slicing-Kriterien getestet. Dazu wird überprüft, ob für jedes Test-Slicing-Kriterium die richtigen ESDG-Knoten als zum Slicing-Kriterium gehörend markiert werden. Tabelle 7.3 enthält die Test-Slicing-Kriterien und die Menge der ESDG-Knoten, welche durch den ESDGMarker markiert werden.

Slicing-Kriterium	**markierte ESDG-Knoten**
<6,{aGlobal,bGlobal}>	{(49,*6*)}
<7,{aGlobal,bGlobal}>	{(49,*6*), (37,*5*)}
<36,{bLocal}>	{}
<36,{bGlobal}>	{(196,-)}
<17,{bGlobal,bParam,aLocal,cLocal}>	{(90,*17*), (228,-)}
<24,{aGlobal,cLocal}>	{(234,-)}

Tabelle 7.3.: Verschiedene Slicing-Kriterien und ihre Markierung durch den ESDGMarker im ESDG zu Listing 7.5 (ESDG-Knoten werden hier als Tupel bestehend aus Knoten-ID und entsprechender Zeilennummer im Testprogramm angegeben)

7.2.13. StaticBackwardSliceComputerForCTest

Von den Ergebnissen des ESDGMarkers ausgehend, kontrolliert der Unittest des StaticBackwardSliceComputer die zur Slice gehörenden und markierten ESDG-Knoten. Die erwarteten Ergebnisse für die Slicing-Kriterien, die bereits im Test des ESDGMarkers verwendet wurden, befinden sich in Tabelle 7.4.

Slicing-Kriterium	**markierte ESDG-Knoten**
<6,{aGlobal,bGlobal}>	{(49,*6*), (37,*5*), (6,*4*), (124,*32*), (167,*30*)}
<7,{aGlobal,bGlobal}>	{(49,*6*), (37,*5*), (6,*4*), (124,*32*), (167,*30*)}
<36,{bLocal}>	{}
<36,{bGlobal}>	{(196,-), (192,-), (49,*6*), (37,*5*), (6,*4*), (124,*32*), (167,*30*)}
<17,{bGlobal,bParam,aLocal,cLocal}>	{(49,*6*), (37,*5*), (6,*4*), (124,*32*), (167,*30*), (192,-), (196,-), (90,*17*), (228,-), (220,-), (232,-), (244,-), (97,*19*), (129,*33*), (186,-), (110,*25*), (81,*16*), (226,-), (106,*24*), (222,-), (103,*22*), (135,*34*), (93,*18*), (99,*21*), (190,-), (216,-), (138,*35*), (117,*31*), (158,-), (240,-)}
<24,{aGlobal,cLocal}>	{(234,-), (238,-), (138,*35*), (158,*15*), (129,*33*), (124,*32*), (186,-), (190,-), (37,*5*), (6,*4*), (167,*30*), (240,-)}

Tabelle 7.4.: Verschiedene Slicing-Kriterien und ihre Markierung durch den StaticBackwardSliceComputer im ESDG zu Listing 7.5 (ESDG-Knoten werden hier als Tupel bestehend aus Knoten-ID und entsprechender Zeilennummer im Testprogramm angegeben)

7.2.14. StaticForwardSliceComputerForCTest

Analog zum Test des StaticBackwardSliceComputer werden beim Test des StaticForwardSliceComputer ausgehend von den Ergebnissen des ESDGMarkers die zur Slice gehörenden, markierten ESDG-Knoten überprüft. Tabelle 7.5 listet ausgewählte, erwartete Ergebnisse nach Ausführung des StaticForwardSliceComputer.

Slicing-Kriterium	**markierte ESDG-Knoten**
<36,{bLocal}>	{}
<17,{bGlobal,bParam,aLocal,cLocal}>	{(151,*23*), (152,*24*), (204,-), (10,*13*), (81,*16*), (99,*21*), (9,*9*), (130,*20*), (66,*11*), (103,*22*), (150,*21*), (59,*10*), (88,*17*), (155,*27*), (60,*10*), (68,*12*), (97,*19*), (153,*26*), (154,*16*), (202,-), (106,*24*), (57,*9*), (114,*18*), (93,*18*), (90,*17*), (210,-), (214,-), (208,-), (198,-), (110,*25*), (228,-)}
<24,{aGlobal,cLocal}>	{(234,-), (208,-), (204,-), (66,*11*), (210,-), (214,-), (68,*12*), (10,*13*)}

Tabelle 7.5.: Verschiedene Slicing-Kriterien und ihre Markierung durch den StaticForwardSliceComputer im ESDG zu Listing 7.5 (ESDG-Knoten werden hier als Tupel bestehend aus Knoten-ID und entsprechender Zeilennummer im Testprogramm angegeben)

7.2.15. ESDGToCodeConverterForCTest

Der Test des ESDGToCodeConverter ist kein Test im eigentlichen Sinne, da hier nur überprüft wird, ob die Komponente eine Datei mit Inhalt produziert. Der Inhalt dieser Datei ist dann die Slice, welche durch den Test auf der Konsole ausgegeben wird. Die Korrektheit der Slice wurde manuell überprüft. Die beiden Listings 7.6 und 7.7 zeigen exemplarisch den erwarteten Inhalt der Datei `slice.c` für eine mit dem StaticBackwardSliceComputer berechnete statische Rückwärtsslice bzw. eine mit dem StaticForwardSliceComputer berechnete statische Vorwärtsslice. Es ist zu beachten, dass in den beiden Listings die entfernten Codezeilen in grauer Schrift angezeigt werden.

```
1  int aGlobal;
2  int bGlobal;
3
4  void init() {
5      aGlobal = 4;
6      bGlobal = 1 + 3 + aGlobal − 3 − 2;
7  }
8
9  int m2(int cParam) {
10     int cLocal = 5;
11     cLocal *= aGlobal;
12     return cLocal;
13 }
14
15 void m1(int aParam, int bParam) {
16     while (bParam < aParam) {
17         jumpLabel: bParam = bParam + 1;
18         if (bParam <= 2) {
19             continue;
20         }
21         if (bGlobal == 3) {
22             goto jumpLabel;
23         }
24         if (m2(aParam) >= 4) {
25             break;
26         }
27     }
28 }
29
30 int main() {
31     int aLocal, bLocal;
32     init();
33     if (aGlobal > 0) {
34         aLocal = aGlobal + bGlobal;
35         m1(aLocal, 1);
36         return aLocal;
37     } else {
38         aLocal = -1;
39         return -1;
```

```
40      }
41  }
```

Listing 7.6: Inhalt der Datei `slice.c` nach Anwendung des statischen Rückwärtsslicen anhand des Slicing-Kriteriums <7,{aGlobal,bGlobal}>

```
1   int aGlobal;
2   int bGlobal;
3
4   void init() {
5       aGlobal = 4;
6       bGlobal = 1 + 3 + aGlobal - 3 - 2;
7   }
8
9   int m2(int cParam) {
10      int cLocal = 5;
11      cLocal *= aGlobal;
12      return cLocal;
13  }
14
15  void m1(int aParam, int bParam) {
16      while (bParam < aParam) {
17          jumpLabel: bParam = bParam + 1;
18          if (bParam <= 2) {
19              continue;
20          }
21          if (bGlobal == 3) {
22              goto jumpLabel;
23          }
24          if (m2(aParam) >= 4) {
25              break;
26          }
27      }
28  }
29
30  int main() {
31      int aLocal, bLocal;
32      init();
33      if (aGlobal > 0) {
```

```
34              aLocal = aGlobal + bGlobal;
35              m1(aLocal, 1);
36              return aLocal;
37          } else {
38              aLocal = -1;
39              return -1;
40          }
41      }
```

Listing 7.7: Inhalt der Datei `slice.c` nach Anwendung des statischen Vorwärtsslicen anhand des Slicing-Kriteriums <17,{bGlobal,bParam,aLocal,cLocal}>

7.3. Ausnahmebehandlung

Dieser Abschnitt erläutert die *Exceptions*, die durch RePST bzw. dessen Komponenten ausgelöst werden können. Dazu wird die *Exception-Hierarchie* [Ull07] von Java um `Exception`-Unterklassen erweitert, welche Ausnahmen in den Komponenten vertreten.

7.3.1. Exception-Hierarchie in RePST

Abb. 7.10 visualisiert die dreistufige Exception-Hierarchie des Program Slicing Tools. In der ersten Ebene befindet sich die Klasse `Exception` [Ull07], von ihr erben die Klassen der zweiten Ebene `EvaluateException`, `ProgramSlicingToolException` und `JGraLabException`. Durch diese Ebene wird der Ursprung der Ausnahme festgelegt.

- Die `EvaluateException` hat ihren Ursprung in der GReQL2-API. Sie kann auftreten, wenn während der Ausführung von GReQL2-Anfragen Ausnahmen entstehen. Weiterführendes hierzu findet der Leser in [Bil06].

- Die `ProgramSlicingToolException` entspringen direkt aus den Komponenten des Program Slicing Tools. Sie werden im nächsten Unterabschnitt 7.10 ausführlich erläutert.

- Die `JGraLabException` stammt aus der externen API von JGraLab. Solche Ausnahmen werden geworfen, sofern Methoden der JGraLab-API zu unerwarteten Ergebnissen führen. Eine detailliertere Beschreibung der Ausnahmen in JGraLab wird in [Kah06] dargelegt.

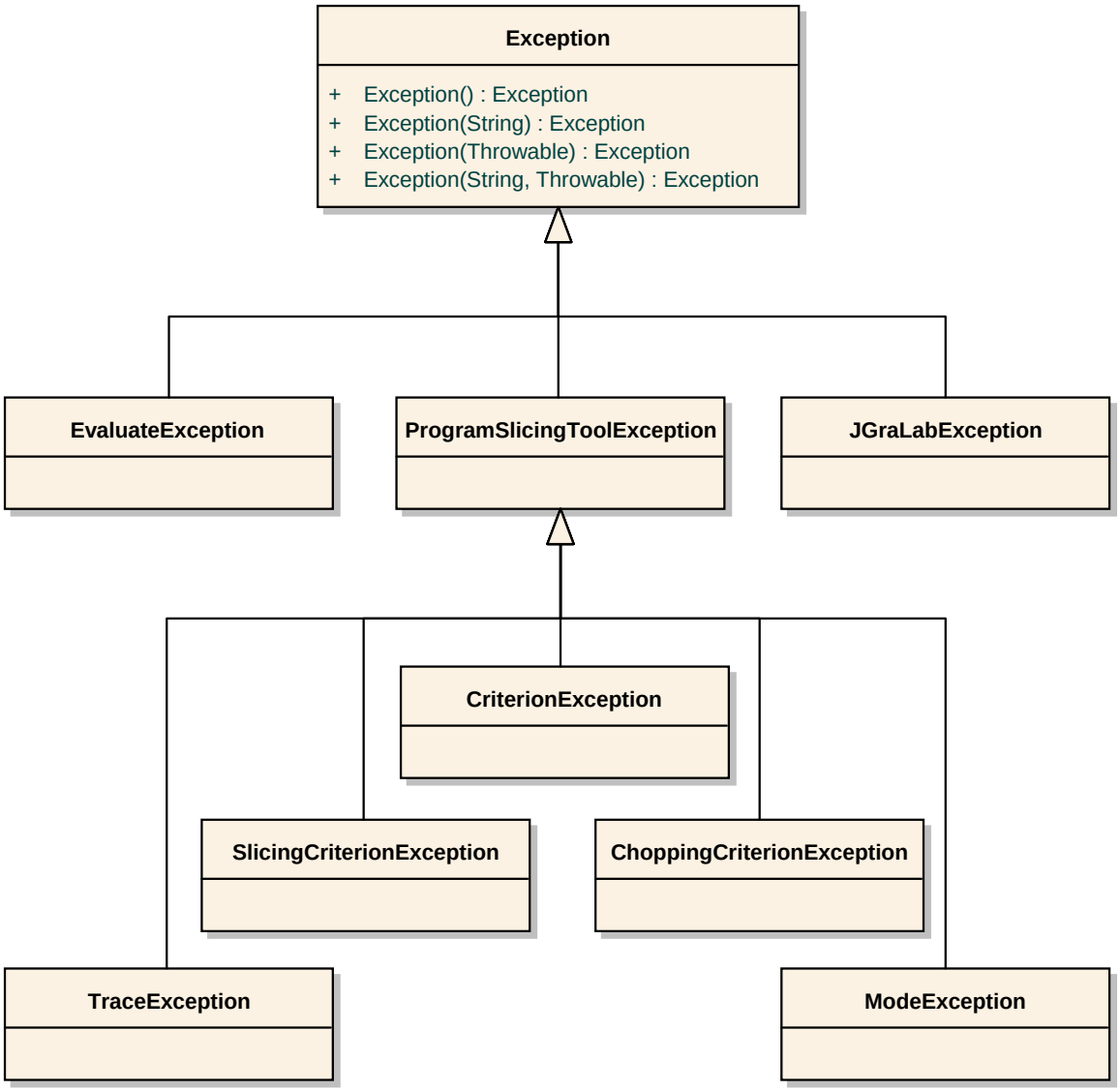

Abbildung 7.10.: Klassendiagramm: Exception-Hierarchie in RePST

7.3.2. Mögliche Auslöser der ProgramSlicingToolExceptions

In der dritten Ebene der Exception-Hierarchie wird der Auslöser der Ausnahme noch enger eingegrenzt. Im Folgenden werden die dazu genutzten `ProgramSlicingToolException`-Unterklassen mit ihren möglichen Auslösern gelistet.

CriterionException: Die Komponente ProgramSlicer entscheidet anhand eines Strings, ob ein Slicing- oder ein Chopping-Verfahren eingesetzt wird. Wenn diese Entscheidung nicht getroffen werden kann, erzeugt der ProgramSlicer eine `CriterionException`.

SlicingCriterionException: Innerhalb des ESDGMarkers oder des ProgramSlicers können `SlicingCriterionExceptions` hervortreten, wenn den beiden Komponenten ein Slicing-Kriterium geliefert wird, welches das benötigte Pattern für Slicing-Kriterien nicht erfüllt. Außerdem tritt diese Ausnahme auf, falls sich die im Slicing-Kriterium definierte Anweisung oder eine der definierten Variablen nicht im Quellcode befindet.

ChoppingCriterionException: `ChoppingCriterionExceptions` treten auf, sofern dem ESDGMarker oder dem ProgramSlicer ein String übergeben wird, der ein Chopping-Kriterium definiert, welches nicht dem vorausgesetzten Muster (siehe Abschnitt 6.16) für Chopping-Kriterien gleichkommt.

TraceException: `TraceExceptions` entstehen immer dann, wenn dem ESDGMarker oder dem ProgramSlicer eine Trace als String übergeben wird, welche nicht dem geforderten Pattern (siehe Abschnitt 6.16) für zu akzeptierende Traces entspricht.

ModeException: Die konkreten Factories (siehe Abschnitt 5.6) produzieren `ModeExceptions`, wenn der Methode `createProgramSlicingTool` ein Wert übergeben wird, der kein unterstütztes Program Slicing-Verfahren repräsentiert.

7.4. Installation von RePST

Für die Installation von RePST oder einzelner Komponenten benötigt man den RePST-Quellcode. Dieser befindet sich als *jar*-Datei `repst.jar` und als Sammlung von *java*-Dateien auf der CD im Anhang B.

Alle Komponenten benötigen zwingend JGraLab in der Version Anatotitan, daher muss sich JGraLab im *Java-Classpath* befinden. Die Komponente `ProgramSlicingTool` braucht zusätzlich die Datei `getopt.jar` im Classpath. JGraLab in Form der Datei `jgralab.jar` und `getopt.jar` befinden sich ebenfalls auf der CD im Anhang.

Die Komponente ASGComputer ist plattformabhängig, da sie das externe Tool G2TG verwendet, welches nur unter Linux funktioniert. Demnach muss zur Nutzung der Komponenten ProgramSlicingTool, ProgramPreprocessor und ASGComputer *Linux* als Betriebssystem verwendet werden. Die restlichen Komponenten sind plattformunabhängig.

Neben dem externen Tool G2TG benötigt der ASGComputer auch noch die externen Tools GCPA und GCPP. Diese beiden Tools befinden sich sowohl in einer Version für Linux `gcpa.dat` und `gcpp.dat` als auch in einer Windows-Version `gcpa.exe` und `gcpp.exe` auf der CD im Anhang. G2TG ist nur in einer Linux-Version `g2tg` auf der CD vorhanden.

Die externen Tools des ASGComputers benötigen die ReGroup-Umgebung, welche aus verschiedenen GUPRO-Tools und GraLab 4.4 besteht. Da die Installation der ReGroup-Umgebung recht aufwändig ist, befindet sich auf der CD im Anhang eine „readme“-Datei von Daniel Bildhauer, welche die Installation erklärt.

Anschließend muss der Ant [ES06] Build-Prozess gestartet werden, um das Schema aus der Datei `slicingschema.tg` zu generieren. Dazu muss das Target „generateschema“ in der Datei `build.xml` ausgeführt werden. Diese Datei ist nach den Maßgaben von [Fal07] aufgebaut.

Zur abschließenden Konfiguration des ASGComputers enthält dessen API für jedes benötigte externe Tool eine Setter-Methode. Dabei braucht jede Setter-Methode einen String als Parameter, welcher den Pfad zum externen Tool beschreiben muss.

Um das Program Slicing Tool oder dessen Komponenten zu starten, muss Java 6 installiert sein. Eventuell muss der Java Virtual Machine zusätzlicher Speicher zugewiesen werden, für die Tests haben sich 512 Megabyte als geeigneter Wert erwiesen.

7.5. Die Benutzungsoberfläche von RePST

Zur Demonstration der Funktionalität von RePST gibt es eine *grafische Benutzungsoberfläche*, die dem Benutzer das Slicen von C-Programmen ermöglicht. Die Benutzungsoberfläche besteht aus einem Fenster, welches drei Buttons, ein Textfeld und eine Textarea enthält. Die Buttons und das Textfeld werden zur Steuerung des Tools verwendet, während die Textarea den Quellcode bzw. den gesliceten Quellcode mit zugehörigen Zeilennummern anzeigt. Ein Screenshot der grafischen Benutzungsoberfläche befindet sich in Abb. 7.11.

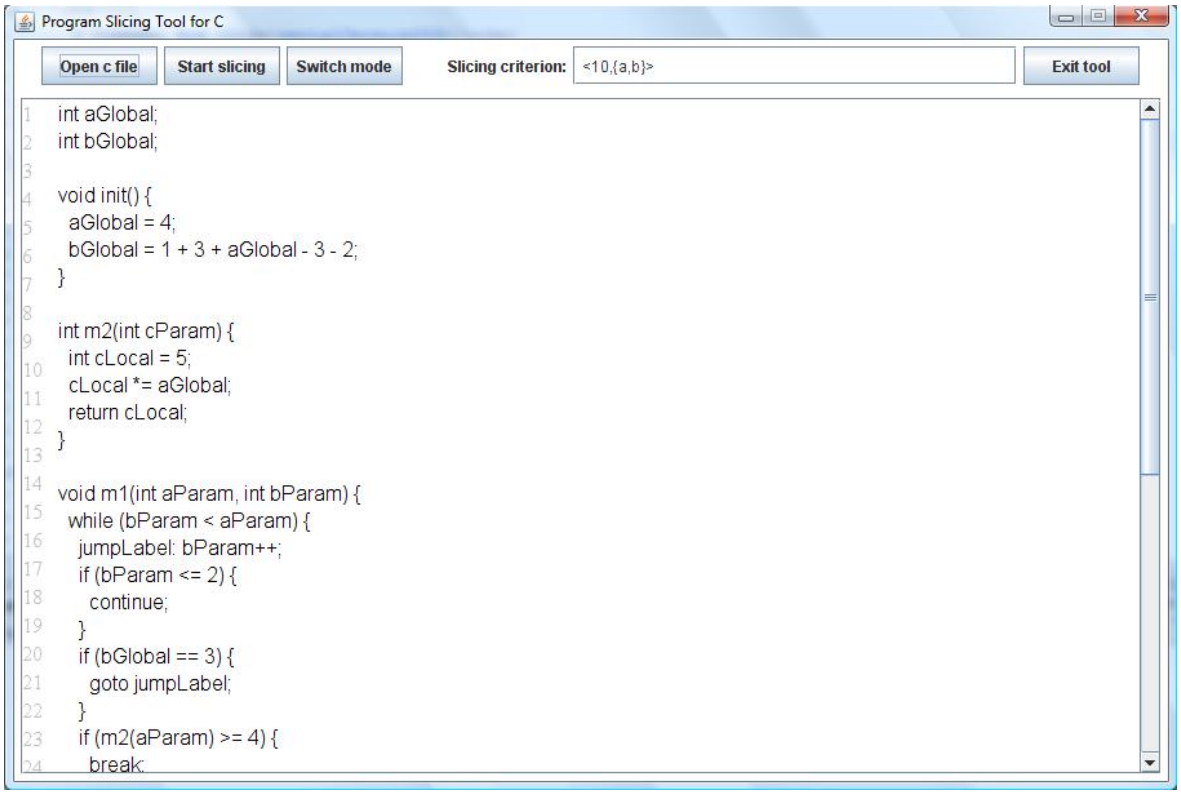

Abbildung 7.11.: Grafische Benutzungsoberfläche für RePST zum Slicen von C-Quellcode

„Open c file“. Bevor Quellcode geslicet werden kann, muss zuerst eine „c“-Datei geöffnet werden. Dazu gibt es den Button „Open c file“. Dieser startet einen Dateibrowser, mit dessen Hilfe eine beliebige Datei mit der Endung „.c“ geöffnet werden kann. Die geöffnete „c“-Datei wird anschließend in der Textarea angezeigt.

„Start Slicing“. Das Drücken des Buttons „Start Slicing“ startet den Slicing-Vorgang, sofern vorher eine „c“-Datei geöffnet wurde. Das Ergebnis des Slicing-Vorgangs wird anschließend in der Textarea angezeigt und in der Datei `slice.c` gespeichert, welche im Verzeich-

nis der Quelldatei angelegt wird. Wenn der Slicing-Vorgang erfolgreich war, wird in der Textarea die Slice des Quellcodes angezeigt. Dabei bleiben die alten Zeilennummern in der Slice erhalten. Andernfalls befindet sich eine Fehlermeldung in der Textarea.

„Switch Mode“. Die Betätigung des Buttons „Switch Mode“ tauscht das eingesetzte Slicing-Verfahren durch das nächste, unterstützte Verfahren aus. Aktuell werden die Verfahren statisches Rückwärtsslicen und statisches Vorwärtsslicen unterstützt. Standardmäßig ist statisches Rückwärtsslicen aktiv.

„Slicing Criterion:“. In das Textfeld „Slicing Criterion“ muss der Benutzer vor dem Starten des Slicing-Vorgangs ein Slicing-Kriterium eintragen. Dieses Slicing-Kriterium sollte dem Muster für Slicing-Kriterien entsprechen, welches in Abschnitt 6.16 beschrieben ist.

„Exit tool“. Das Tool wird durch Betätigen des Buttons „Exit tool“ beendet. Alternativ kann das Tool auch durch den „x“-Button des Fensters geschlossen werden.

7.6. Steuerung von RePST per Konsole

Alternativ zur Steuerung von RePST mittels grafischer Benutzungsoberfläche gibt es die Möglichkeit RePST per *Konsole* zu bedienen.

```
1 java (−cp | −−classpath) <classpath> ProgramSlicingTool
2       ((−m | −−mode) <number>
3       (−s | −−sourcefilepath) <path><.c>
4       (−c | −−criterion) <slicing or chopping criterion>
5       [(−t | −−trace) <trace>])
```

Listing 7.8: Regulärer Ausdruck zum Starten von RePST per Konsole

Optionen. In der Konsole kann das Slicen oder Choppen von Quellcode mit Hilfe von Optionen im Startbefehl konfiguriert werden. Diese Optionen sind mit „GNU getopt“ [GNU06] realisiert. Der reguläre Ausdruck in Listing 7.8 definiert den Startbefehl von RePST und die

möglichen Kombinationen der Optionen, wobei die Reihenfolge der Optionen unerheblich ist.

- Die Option „mode“ oder kurz „m“ legt das zu verwendende Slicing- oder Chopping-Verfahren fest. Daher entspricht das Argument dieser Option einem Wert zwischen 1 und 8, wobei folgende Wert-Verfahren-Zuordnung gilt:

  1. Statisches Rückwärtsslicen
  2. Statisches Rückwärtsslicen mit Berechnung einer ausführbaren Slice
  3. Statisches Vorwärtsslicen
  4. Dynamisches Rückwärtsslicen
  5. Dynamisches Rückwärtsslicen mit Berechnung einer ausführbaren Slice
  6. Dynamisches Vorwärtsslicen
  7. Statisches Choppen
  8. Dynamisches Choppen

- Die Option „sourcefilepath“ oder kurz „s“ definiert den Pfad zur Quellcode-Datei, welche geslicet oder gechoppt werden soll. Hier muss das Argument den entsprechenden Pfad zu einer „c“-Datei enthalten. Nach eventuellen Weiterentwicklungen können auch andere Quellcode-Dateitypen verarbeitet werden. Das Ergebnis des Slicens wird in einer Datei mit dem Namen „slice“ gespeichert, die im Verzeichnis der Quelldatei erzeugt wird.

- Die Option „criterion“ oder kurz „c“ enthält als Argument das Slicing- oder Chopping-Kriterium. Beide Kriterien müssen einem regulären Ausdruck nach Tabelle 6.1 entsprechen.

- Die einzige optionale Option „trace“ oder „t“ besitzt als Argument die Trace, welche bei dynamischen Verfahren erforderlich ist. Sie muss dem regulären Ausdruck nach Tabelle 6.1 entsprechen.

- Die Option „help“ oder kurz „h“ zeigt den Hilfetext zur Verwendung von RePST per Konsole an. Sie besitzt keine Argumente.

Classpath. Außer den Optionen zur Steuerung von RePST muss noch der Java-Classpath gesetzt werden. Dieser muss die Pfadangaben zu den Dateien von JGraLab, von getopt und von dem Program Slicing Tool selbst enthalten.

main-Methode. Die zum Starten von RePST relevante main-Methode befindet sich in der Klasse `ProgramSlicingTool`, welche innerhalb des Paketes `de.uni_koblenz.programSlicing` ist. Soll die grafische Benutzungsoberfläche von RePST verwendet werden, so muss die main-Methode innerhalb der Klasse `MainFrameForC` aufgerufen werden. Diese Klasse befindet sich innerhalb des Paketes `de.uni_koblenz.programSlicingToolGUI`. Die Optionen zum Steuern von RePST werden in der main-Methode der Klasse `MainFrameForC` nicht verarbeitet.

Listing 7.9 zeigt einen Beispiel-Befehl, welcher RePST startet und die Datei `testprogram.c` anhand des Slicing-Kriteriums „<6,aGlobal,bGlobal>“ statisch rückwärtsslicet.

```
1 java −cp ./build/classes:./lib/getopt.jar:./lib/jgralab.jar
2       de.uni_koblenz.programSlicing.ProgramSlicingTool
3       −m 1 −s ./tmp/testprogram.c −c "<6,{aGlobal,bGlobal}>"
```

Listing 7.9: Beispiel: Statisches Rückwärtsslicen einer „c“-Datei per Konsole

7.7. API von RePST

Das Program Slicing Tool selbst und jede Komponente des Program Slicing Tools bietet dem Programmierer ein *API* an. Das API einer Komponente besteht aus den *public-Methoden* dieser Komponente, deren Ausführung die Funktion der Komponente verwirklichen. In der Regel besitzt jede Komponente nur eine public-Methode.

Die Komponente ACFGComputer besitzt zum Beispiel nur die public-Methode `computeACFG`. Demnach sieht das API dieser Komponente so aus: **computeACFG(Graph): Graph**.

Da Kapitel 6 sowohl die Funktionalität als auch die entsprechenden public-Methoden jeder Komponente beschreibt, wird an dieser Stelle auf eine redundante Beschreibung der API verzichtet.

8. Fazit

Zum Schluss wird in diesem Kapitel ein Fazit gezogen, welches RePST anhand der Anforderungen bewertet, künftige Weiterentwicklungsmöglichkeiten vorstellt und die Ergebnisse zusammenfasst.

8.1. Bewertung

Die Bewertung von RePST geschieht durch einen Vergleich des Entwicklungsstandes am Ende dieser Diplomarbeit mit den Anforderungen aus Abschnitt 4.2. Dieser Vergleich besteht aus mehreren Punkten, welche den jeweiligen Anforderungskategorien entsprechen.

8.1.1. Bewertung: Funktionale Anforderungen

Die obligatorischen, funktionalen Anforderungen an RePST und an das Program Slicing Tool für C wurden vollständig erfüllt. Es wurden jedoch nicht alle optionalen Aspekte vollständig erfüllt, so dass es einige Einschränkungen gibt, welche durch mögliche Weiterentwicklungsmaßnahmen (siehe Abschnitt 8.2) aufgehoben werden können.

8.1.2. Bewertung: Technische Anforderungen

Die technischen Anforderungen wurden vollständig erfüllt.

8.1.3. Bewertung: Anforderungen an die Architektur

Die Anforderungen an die Architektur wurden vollständig erfüllt.

8.1.4. Bewertung: Qualitätsanforderungen

Die Qualitätsanforderungen wurden bis auf die Punkte 7, 8, 11, 12 und 13 vollständig erfüllt.

Rückblick: Qualitätsanforderungen

7. Auf dem Referenzschema basierende Komponenten sollen mit möglichst wenig Aufwand zum Slicen von Programmcode in anderen Programmiersprachen nutzbar sein.

8. Auf dem C-Schema basierende Komponenten können dazu in der Lage sein, mehr als 80% des Quellcodes der auf dem Referenzschema basierenden Superkomponente zu nutzen.

11. Die Komponenten können dazu in der Lage sein, möglichst performant zu arbeiten.

12. Komponenten zum konkreten Program Slicing für C-Programme sollen durch Unit-Tests getestet werden.

13. Verwendete Fremdsoftware kann dazu in der Lage sein, plattformunabhängig zu sein.

Qualitätsanforderung 7 ist im Prinzip erfüllt, wobei der Entwicklungsaufwand zum Slicen anderer Programmiersprachen geringer wäre, wenn alle abstrakten RePST-Komponenten durch konkrete ersetzt wären.

Die Auswertung von Punkt 8 erfordert einen Vergleich der lines of code (LOC) [Bal00] zwischen der RePST-Komponente und der konkreten Komponente für Slicing von C-Programmen. Die hier angewendete Metrik zählt zum LOC-Wert einer RePST-Komponente alle Quellcodezeilen mit mehr als drei Zeichen ungleich dem Leerzeichen. Kommentare werden beim Zählen ignoriert. Der LOC-Wert einer RePST-Komponente setzt sich aus den LOC-Werten aller benötigten Klassen zusammen, wobei die SharedComponent-Klasse und Testklassen ignoriert werden. Tabelle 8.1 listet alle Komponenten und deren prozentualen Anteil von Quellcode in den RePST-Klassen.

In Tabelle 8.1 erkennt man, dass die Komponenten ASGComputer, CGComputer, DefUseInfComputer, BasicESDGComputer, ESDGMarker, ESDGToCodeConverter und SharedComponent den geforderten Prozentsatz von 80% nicht erfüllen. Für die Komponenten ASGComputer und ESDGToCodeConverter ist dies naheliegend, da sie Quellcode lesen und schreiben. Die SharedComponent kapselt Methoden, die von mehreren Komponenten genutzt werden. Da diese Methoden sehr oft direkten Zugriff auf das C-Schema erfordern, ist ein Wert größer 80% zum Slicen von C-Programmen hier nicht möglich. Im Zuge der Weiterentwicklung ist

Komponente	LOC-RePST	LOC-C	Verhältnis
ProgramSlicingTool	144	0	100%
ProgramPreprocessor	67	0	100%
ASGComputer	50	112	30,8%
ACFGComputer	303	42	87,8%
PGComputer	126	0	100%
ECGComputer	56	0	100%
CGComputer	86	25	77,5%
DefUseInfComputer	172	68	71,7%
ESDGComputer	65	0	100%
BasicESDGComputer	168	51	76,7%
IntMetEdgesComputer	124	0	100%
ConDepEdgesComputer	115	0	100%
DataFlowEdgesComputer	383	63	85,9%
SumEdgesComputer	114	0	100%
ProgramSlicer	107	0	100%
ESDGMarker	220	70	75,9%
StaticBackwardSliceComputer	71	0	100%
StaticForwardSliceComputer	71	0	100%
DynamicBackwardSliceComputer	46	0	100%
DynamicForwardSliceComputer	46	0	100%
StaticChopComputer	46	0	100%
DynamicChopComputer	46	0	100%
ExecutableBackwardSliceComputer	46	0	100%
ESDGToCodeConverter	73	178	29,1%
SharedComponent	171	108	61,3%
GReQLAdapter	29	0	100%
Insgesamt	2945	717	80,4%

Tabelle 8.1.: Tabellarischer Überblick über den prozentualen Anteil von Quellcode in RePST-Komponenten.

ein Wert größer 80% für die Komponenten CGComputer, DefUseInfComputer, BasicESDG-Computer und ESDGMarker zu erreichen, wenn die abstrakten Methoden und Stubs dieser Komponenten durch konkrete Methoden ersetzt werden.

Die Performance von RePST hängt teilweise von der Performance von GReQL2 ab. Da GReQL2 noch keinen fertigen Optimierer hat, ist eine Performancesteigerung mit einer neuen GReQL2-Version möglich. Somit hängt die vollständige Erfüllung von Punkt 11 vom GReQL2-Entwicklungsstand ab.

Die Unittests beschränken sich auf Komponenten die keine Fassaden sind. Damit ist Qualitätsanforderung 12 eigentlich erfüllt, da Fassaden hier nicht als konkrete Komponenten betrachtet werden.

Zur Berechnung des abstrakten Syntaxgraph wird der GUPRO C-Parser verwendet. Dieser ist nicht plattformunabhängig, da das Tool G2TG nur unter Linux funktioniert. Punkt 13 kann also nur durch Weiterentwicklung oder Austausch dieses externen Tools erfüllt werden.

8.1.5. Bewertung: Anforderungen an die Benutzerschnittstelle

Die Anforderungen an die Benutzerschnittstelle wurden vollständig erfüllt.

8.1.6. Bewertung: Anforderungen an die Dokumentation

Die Anforderungen an die Dokumentation wurden vollständig erfüllt.

8.2. Weiterentwicklung

An dieser Stelle soll ein Ausblick auf mögliche Verbesserungen und Weiterentwicklungen von RePST gegeben werden.

8.2.1. Program Slicing von Quellcode in anderen Programmiersprachen

Diese Diplomarbeit wurde hauptsächlich umgesetzt, um eine Basis für Program Slicing in beliebigen Programmiersprachen zu schaffen. Dies ist mit RePST gelungen, so dass nun ein

adaptierbares Tool bereitsteht, welches durch geringe Anpassungen das Slicen von Quellcode in anderen Programmiersprachen ermöglicht. Diese Anpassungen beziehen sich auf Spezialisierungen von einigen Komponenten, sofern vorab eine Abbildung zwischen dem Schema der gewünschten Programmiersprache und RePSS gemacht wurde. Da RePSS ein Schema ist, das sich relativ stark an Java orientiert, würde sich eine Weiterentwicklung zum Slicen von Java-Code besonders anbieten.

8.2.2. Alternative Slicing- und Chopping-Verfahren

Als Ergebnis dieser Diplomarbeit bietet RePST die Slicing-Verfahren statisches Rückwärtsslicen und statisches Vorwärtsslicen an. Zusätzlich werden Stubs für die Verfahren statisches Choppen, dynamisches Rückwärtsslicen, dynamisches Vorwärtsslicen, dynamisches Choppen, statisches und dynamisches Rückwärtsslicen mit ausführbarer Slice angeboten. Durch Weiterentwicklung können die angebotenen Stubs durch konkrete Verfahren ersetzt werden, so wie sie in [Sch07] vorgestellt sind.

8.2.3. PGComputer

Aktuell verwendet der PGComputer einen rudimentären Algorithmus zur Berechnung der „zeigt auf“-Beziehungen zwischen Zeigern und Variablen. Dieser kann im Zuge der Weiterentwicklung durch einen effizienteren Algorithmus ersetzt werden. Beispiele für effizientere Algorithmen sind in [And94], [Ste96] oder [IH97] zu finden.

8.2.4. Implementierung der abstrakten RePST-Komponenten

Wie bereits in Abschnitt 8.1 angedeutet, sollten die abstrakten RePST-Komponenten durch konkrete ausgetauscht werden. Dies hätte zusätzlich zur Folge, dass Programme, deren abstrakte Syntaxgraphen konform zu RePSS sind, geslicet werden können. Die abstrakten Komponenten teilen sich anhand des Konkretisierungs-Aufwands in zwei Gruppen:

1. Komponenten mit geringem Konkretisierungs-Aufwand: ACFGComputer, CGComputer, DefUseInfComputer, BasicESDGComputer, SharedComponent

2. Komponenten mit hohem Konkretisierungs-Aufwand: ASGComputer, ESDGToCodeConverter

Exemplarisch wurde die Konkretisierung der Komponente DataFlowEdgesComputer bereits in dieser Diplomarbeit durchgeführt.

8.2.5. Program Slicing für C-Programme

Das Slicen von C-Programmen unterliegt den folgenden *Einschränkungen*:

1. Es können nur „c“-Dateien ohne Includings verarbeitet werden.
2. Variablen, auf die ein Zeiger als Parameter zeigt, werden nicht berücksichtigt.
3. Verbesserungswürdige Aspekte, die bereits in [Sch07] aufgeführt sind, schränken den C-Quellcode ebenfalls ein. Hierzu zählen beispielsweise interprozedurale Sprünge, Attribute mit beliebigen Sichtbarkeiten, Felder, Funktionszeiger, Verkettung von Zeiger oder `switch`-Anweisungen.

Das Weiterentwickeln der C-spezifischen Komponenten sollte die erste Einschränkung beheben. Die Erweiterungen bezüglich der restlichen Einschränkungen sollten direkt in RePST und RePSS integriert werden, damit sie sich nach eventueller Anpassung automatisch auf das Slicen von C-Code auswirken.

8.2.6. Integration in fremde Umgebungen

Durch die komponentenbasierte Struktur von RePST und die einfachen Schnittstellen, sollte sich RePST relativ leicht in fremde Umgebungen integrieren lassen. Für die Zukunft wäre zum Beispiel eine Integration in Eclipse [GB03] oder GUPRO denkbar.

8.3. Zusammenfassung

Hauptziel dieser Diplomarbeit war die Umsetzung des Dienstmodells für Program Slicing, welches in [Sch07] entwickelt wurde. Zusätzlich wurde dieses Dienstmodell erweitert, um seine Nutzbarkeit für prozedurale Programmiersprachen zu zeigen. Durch die erfolgreiche Entwicklung von RePST, einem komponentenbasierten Program Slicing Referenztool, wurde das Hauptziel dieser Arbeit erreicht. Außerdem wurde durch die Spezialisierung von

RePST das Slicen von C-Quellcode ermöglicht und somit gezeigt, dass RePST auch für andere Sprachen neben EL als Referenztool geeignet ist.

Ein weiteres Produkt dieser Diplomarbeit ist RePSS, das als Referenzschema für Program Slicing betrachtet werden kann. RePSS vereint grundlegende Aspekte objektorientierter und prozeduraler Programmiersprachen und bietet somit eine fundierte Basis zum Slicen von Quellcode in beliebigen Programmiersprachen.

Diese Diplomarbeit unterstützt durch die Bereitstellung von RePST und RePSS die Entwicklung künftiger Program Slicing Tools und bietet zusätzlich einen Leitfaden zur Entwicklung solcher Tools. Dem Entwickler wird gezeigt, wie er das Schema einer Programmiersprache in RePSS integrieren kann und wie er ein Program Slicing Tool als Spezialisierung von RePST entwickeln kann.

Aktuelle Entwicklungen in der Softwaretechnik, wie Komponentenorientierung, Design Patterns und moderne Graphentechnologie werden aufgegriffen und umgesetzt, so dass eine technologisch hochwertige Basis geschaffen wird. Die Phasen des Software-Lebenslaufs spiegeln sich in den einzelnen Kapiteln dieser Diplomarbeit wieder. Dadurch dokumentiert sie zuletzt auch einen für die Softwaretechnik typischen Produktentwicklungsprozess.

A. Glossar

abstrakter Syntaxgraph: Graph zur Repräsentation des Programmcodes nach Ausführung von Compilerbautechniken. Er wird zur Erstellung der *ACFGs*, der *PGs*, des *ECG* und des *ESDG* benötigt. Siehe Abschnitt 3.4.1

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

Actual-in-Knoten: Knoten, welcher die benutzen Variablen in einer Funktion in Abhängigkeit von dem zugehörigen Funktionsaufruf-Knoten vertritt.

Actual-out-Knoten: Knoten, welcher die definierten Variablen in einer Funktion in Abhängigkeit von dem zugehörigen Funktionsaufruf-Knoten vertritt.

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

Aufrufkante: Kante innerhalb des *ESDG* zur Verbindung eines Aufrufs mit dem *Entry-Knoten* der aufgerufenen Funktion.

C-Schema: Schema der Programmiersprache C. Siehe Kapitel 3.

Chop: Programmausschnitt, der nur die notwendigen Anweisungen enthält, um die Auswirkungen einer Anweisung *s* auf die Anweisung *t* zu beschreiben.

- **Chop, statisch**: Die Berechnung des *Chops* berücksichtigt den vollständigen Programmcode.
- **Chop, dynamisch**: Die Berechnung des *Chops* bezieht sich auf eine bestimmte Ausführung des Programms.

Chopping-Kriterium: besteht aus zwei Anweisung *s* und *t*. Es hat die Form < *s*, *t* >.

Datenflusskante: Kante innerhalb des *ESDG* zur Beschreibung des Datenflusses. Sie verläuft von einem Knoten *s* nach einem Knoten *t*, falls *s* eine Variable definiert, welche anschließend von *t* benutzt wird.

def/use-Kante: Kante innerhalb des *ECG* zur Beschreibung der Beziehung zwischen einer Funktion und einer in ihr definierten oder verwendeten Variable.

Deklarationskante: Kante innerhalb des *ESDG* zur Beschreibung der Beziehung zwischen Deklaration und erster Definition oder Benutzung einer Variable.

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

EL: **E**xemplary **L**anguage. Siehe [Sch07].

Entry-Knoten: Knoten, der den Eingang einer Funktion repräsentiert.

Exit-Knoten: Knoten, der den Ausgang einer Funktion repräsentiert.

erweiterter Aufrufgraph: Graph zur Darstellung von Aufrufbeziehungen zwischen Funktionen und zur Beschreibung von innerhalb einer Funktion benutzten und definierten Variablen. Dieser Graph wird aus dem *ASG* und den *PGs* errechnet und unterstützt die Berechnung des *ESDG*. Siehe Abschnitt 3.4.4.

erweiterter Kontrollflussgraph: Dieser Graph beschreibt mit seinen Knoten die Programmanweisungen einer Funktion. Zusätzlich besitzt er *Entry*- und *Exit*-Knoten, um den Funktionseintritt bzw. -austritt darzustellen. Die Knoten des erweiterten Kontrollflussgraphen können mittels Kontrollflusskanten verbunden sein. Dieser Graph wird aus dem *ASG* berechnet und bietet die Grundlage zur Erstellung der *PGs* und des *ESDG*. Siehe Abschnitt 3.4.2.

erweiterter Systemabhängigkeitsgraph: Dieser Graph repräsentiert ein komplettes Programm und ermöglicht durch Traversierung die Berechnung von *Slices* oder *Chops*. Dieser Graph wird aus dem *ASG*, den *ACFGs*, den *PGs* und dem *ECG* berechnet. Siehe Abschnitt 3.4.5.

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

Formal-in-Knoten: Knoten, welche die benutzen Variablen in einer Funktion in Abhängigkeit von ihrem *Entry-Knoten* vertritt .

Formal-out-Knoten: Knoten, welche die benutzen Variablen in einer Funktion in Abhängigkeit von ihrem *Entry-Knoten* vertritt.

Kontrollkante: Kante innerhalb des *ESDG*, welche von einem Knoten *s* zu einem Knoten *t* verläuft, sofern die Ausführung von *t* durch *s* beeinflusst werden kann.

Parameter-in-Kante: Kante innerhalb des *ESDG*, die Beziehungen von *Actual-in-Knoten* nach *Formal-in-Knoten* herstellt.

Parameter-out-Kante: Kante innerhalb des *ESDG*, die Beziehungen von *Formal-out-Knoten* nach *Actual-out-Knoten* herstellt.

Parameterkante: Kante innerhalb des *ESDG* zur Beschreibung der Parameterübergabe. Dabei wird zwischen *Formal-in-*, *Formal-out-*, *Actual-in-* und *Actual-out-Knoten* differenziert.

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

Points-to-Graph: Graph der aufgrund einer Zeigeranalyse entsteht und die „zeigt auf"-Beziehungen einer Funktion wiedergibt.

Slice: Programmausschnitt, der nur Programmanweisungen enthält, welche in Beziehung zu dem vorab zu spezifizierenden *Slicing-Kriterium* stehen.

- **Slice, statisch**: Die Berechnung der *Slice* berücksichtigt den vollständigen Programmcode.
- **Slice, dynamisch**: Die Berechnung der *Slice* bezieht sich auf eine bestimmte Ausführung des Programms.
- **Rückwärtsslice**: *Slice*, die eine Auswirkung auf das *Slicing-Kriterium* hat.
- **Vorwärtsslice**: *Slice*, auf die das *Slicing-Kriterium* Auswirkung hat.
- **Rückwärtsslice, ausführbar**: *Slice*, die um für die Programmausführung notwendige Knoten und Kanten ergänzt ist.

Slicing-Kriterium: besteht aus einer Anweisung i und einer Variablenmenge X. Es hat die Form $< i, X >$.

Reference Program Slicing Schema: Referenzschema für Program Slicing von imperativen Programmiersprachen basierend auf der Sprache *EL*. Siehe Kapitel 3.

Reference Program Slicing Tool: Referenztool für Program Slicing in beliebigen imperativen Programmiersprachen.

RePSS: **Re**ference **P**rogram **S**licing **S**chema. Siehe dort.

RePST: **Re**ference **P**rogram **S**licing **T**ool. Siehe dort.

transitive Kante: Kante innerhalb des *ESDG* zur Beschreibung von Abhängigkeiten zwischen den Werten der Aus- und Eingabeparametern.

B. CD-ROM

Die beiliegende CD-ROM enthält:

- dieses Dokument im *pdf*-Format (`Diplomarbeit.pdf`),
- die Installationsanleitung für die ReGroup-Umgebung (`install-ReGroup.txt`),
- den Quellcode des Program Slicing Tools als *jar*-Datei (`repst.jar`) und Sammlung von *java*-Dateien,
- JGraLab als *jar*-Datei (`jgralab.jar`),
- das Tool GCPA für Windows (`gcpa.exe`) und für Linux (`gcpa.dat`),
- das Tool GCPP für Windows (`gcpp.exe`) und für Linux (`gcpp.dat`),
- das Tool G2TG für Linux (`g2tg`),
- das Slicing-Schema (`slicingschema.tg`),
- das C-Schema (`cschema.tg`) und
- die Ant Build-Datei (`build.xml`).

Literaturverzeichnis

[AH90] AGRAWAL, Hiralal ; HORGAN, Joseph R.: Dynamic Program Slicing. In: *PLDI*, 1990, S. 246–256

[And94] ANDERSEN, Lars O.: *program analysis and specialization for the c programming language*, University of Copenhagen, Diss., May 1994

[ASU86] AHO, Alfred V. ; SETHI, Ravi ; ULLMAN, Jeffrey D.: *Compilers: principles, techniques, and tools*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1986. – ISBN 0–201–10088–6

[AT01] ANDERSON, Paul ; TEITELBAUM, Tim: Software Inspection Using CodeSurfer. (2001), Juli. `http://www.grammatech.com/research/papers/AndersonTeitelbaum.pdf`

[Bal00] BALZERT, Helmut: *Lehrbuch der Software-Technik*. Heidelberg : Spektrum, Akad. Verl., 2000. – ISBN 3–8274–0480–0

[BCM+98] BANTA, D. ; CARROLL, D. ; MAURICH, S. ; ROCCATAGLIATA, J. ; SHANAHAN, P. ; TURNER, C. ; WAITE, W.: *ANSI C Specification*. `http://eli-project.sourceforge.net/c_html/c.ps.Z`, January 1998

[Bea06] BEAULIEU, Alan: *Einführung in SQL*. 1. O'Reilly, 2006. – ISBN-10: 3897214431

[BH93] BALL, Thomas ; HORWITZ, Susan: Slicing Programs with Arbitrary Controlflow. In: *Automated and Algorithmic Debugging*, 1993, 206-222

[Bil06] BILDHAUER, Daniel: *Ein Interpreter für GReQL 2 - Entwurf und prototypische Implementation*, Universität Koblenz-Landau, Institut für Softwaretechnik, Diplomarbeit, 2006. `http://www.uni-koblenz.de/~dbildh/diplomathesis.pdf`

[Bin93] BINKLEY, David: Precise Executable Interprocedural Slices. In: *LOPLAS* 2 (1993), Nr. 1-4, S. 31–45

[BRJ06] BOOCH, Grady ; RUMBAUGH, James ; JACOBSON, Ivar: *Das UML-Benutzerhandbuch*. München [u.a.] : Addison-Wesley, 2006. – ISBN 3–8273–2295–2

[BRSS07] BILDHAUER, Daniel ; RIEDIGER, Volker ; SCHWARZ, Hannes ; STRAUSS, Sascha: *grUML - Eine UML-basierte Modellierungssprache für T-Graphen*. noch nicht veröffentlicht, Januar 2007

[CD94] COOK, Steve ; DANIELS, John: *Designing object systems : object-oriented modelling with Syntropy*. New York : Prentice Hall, 1994. – ISBN 0132038609

[CF94] CHOI, Jong-Deok ; FERRANTE, Jeanne: Static Slicing in the Presence of Goto Statements. In: *ACM Transactions on Programming Languages and Systems* 16 (1994), July, Nr. 4, 1097–1113. `citeseer.ist.psu.edu/choi94static.html`

[Cha85] CHARTRAND, Gary: *Introductory Graph Theory*. Ney York : Dover Publications Inc., 1985. – ISBN 978–0486247755

[Dos05] DOSHI, Gunjan: JUnit 4.0 in 10 minutes. (2005), Juni, 1–6. `http://www.cs.unm.edu/~rstehwien/CS580/reading/JUnit4.pdf`

[DW98] DAHM, Peter ; WIDMANN, Friedbert: Das Graphenlabor / Universität Koblenz-Landau, Institut für Informatik. Version: 1998. `http://www.uni-koblenz.de/~ist/documents/Dahm1998DG.pdf`. 1998 (11/98). – Fachberichte Informatik

[EF95] EBERT, Jürgen ; FRANZKE, Angelika: A Declarative Approach to Graph Based Modeling. In: MAYR, E. (Hrsg.) ; SCHMIDT, G. (Hrsg.) ; TINHOFER, G. (Hrsg.): *Graphtheoretic Concepts in Computer Science*. Berlin : Springer Verlag, 1995 (LNCS 903), 38–50

[EGSW98] EBERT, Jürgen (Hrsg.) ; GIMNICH, Rainer (Hrsg.) ; STASCH, Hans H. (Hrsg.) ; WINTER, Andreas (Hrsg.): *GUPRO - Generische Umgebung zum Programmverstehen*. Föllbach, 1998 (Koblenzer Schriften zur Informatik)

[ES06] EDLICH, Stefan ; STAUDEMEYER, Jörg: *Ant - kurz & gut*. O'Reilly, 2006. – ISBN 3897215195

[Fal07] FALKOWSKI, Kerstin: *Common Apache ANT file*. noch nicht veröffentlicht, September 2007

[FG97] FORGÁCS, Istvan ; GYIMÓTHY, Tibor: An Efficient Interprocedural Slicing Method for Large Programs. In: *Proceedings of SEKE'97*. Madrid, 1997, 279–287

[FHSG98] FIRESMITH, Donald G. ; HENDERSON-SELLERS, Brian ; GRAHAM, Ian: *OPEN Modeling Language (OML) reference manual*. New York : Cambridge University Press, 1998. – ISBN 0521648238

[FOW87] FERRANTE, Jeanne ; OTTENSTEIN, Karl J. ; WARREN, Joe D.: The program dependence graph and its use in optimization. In: *ACM Trans. Program. Lang. Syst.* 9 (1987), Nr. 3, S. 319–349. `http://dx.doi.org/http://doi.acm.org/10.1145/24039.24041`. – DOI http://doi.acm.org/10.1145/24039.24041. – ISSN 0164–0925

[Fri07] FRIEDL, Jeffrey E. F.: *Reguläre Ausdrücke*. 3. Auflage. Köln : O'Reilly, 2007. – ISBN 3897217201

[GB03] GAMMA, Erich ; BECK, Kent: *Contributing to Eclipse: Principles, Patterns, and Plugins*. Redwood City, CA, USA : Addison Wesley Longman Publishing Co., Inc., 2003. – ISBN 0321205758

[GHJV95] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns. Elements of Reusable Object-Oriented Software*. 1st ed. Amsterdam : Addison Wesley, 1995. – ISBN 0–201–63361–2

[GNU06] GNU: *GNU getopt - Java port*. `http://www.urbanophile.com/arenn/hacking/getopt/Package-gnu.getopt.html`, August 2006

[Hor08] HORN, Tassilo: *Ein Optimierer für GReQL 2*. noch nicht veröffentlicht, 2008

[HRB88] HORWITZ, Susan ; REPS, Thomas ; BINKLEY, David: Interprocedural slicing using dependence graphs. In: *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation* Bd. 23. Atlanta,

GA, June 1988, 35–46

[IH97] II, Marc S. ; HORWITZ, Susan: Fast and Accurate Flow-Insensitive Points-To Analysis. In: *Symposium on Principles of Programming Languages*, 1997, 1-14

[Kah06] KAHLE, Steffen: *JGraLab: Konzeption, Entwurf und Implementierung einer Java-Klassenbibliothek für TGraphen*, Universität Koblenz-Landau, Diplomarbeit, Juni 2006

[Kam96] KAMP, Manfred: GReQL - eine Anfragesprache für das GUPRO–Repository 1.1 / Universität Koblenz-Landau, Institut für Softwaretechnik. Koblenz, 1 1996 (8/96). – Projektbericht

[KE04] KEMPER, Alfons ; EICKLER, Andre: *Datenbanksysteme. Eine Einführung*. 5. Oldenbourg, 2004. – ISBN-10: 3486273922 ISBN-13: 978-3486273922

[KG98] KOSCHKE, Rainer ; GIRARD, Jean-Francois: An Intermediate Representation for Reverse Engineering Analyses. In: *Working Conference on Reverse Engineering*, 1998, 241-250

[KL88] KOREL, Bogdan ; LASKI, Janusz: Dynamic program slicing. In: *Inf. Process. Lett.* 29 (1988), Nr. 3, S. 155–163. http://dx.doi.org/http://dx.doi.org/10.1016/0020-0190(88)90054-3. – DOI http://dx.doi.org/10.1016/0020–0190(88)90054–3. – ISSN 0020–0190

[Krü06] KRÜGER, Guido: *Handbuch der Java-Programmierung*. 4. Auflage. München : Addison-Wesley, 2006. – ISBN 3827324475

[Mar06] MARCHEWKA, Katrin: *GReQL 2*, Universität Koblenz-Landau, Institut für Softwaretechnik, Diplomarbeit, 2006

[OMG05] OMG: *Unified Modeling Language: Superstructure*. http://www.omg.org/cgi-bin/doc?ptc/06-04-02.pdf, August 2005

[OO84] OTTENSTEIN, Karl J. ; OTTENSTEIN, Linda M.: The program dependence graph in a software development environment. In: *SIGSOFT Softw. Eng. Notes* 9 (1984), Nr. 3, S. 177–184. http://dx.doi.org/http://doi.acm.org/10.1145/390010.808263. – DOI http://doi.acm.org/10.1145/390010.808263. – ISSN 0163–5948

[RH98] R. HÖLLERER, B. B.: *Übersetzung objektorientierter Programmiersprachen*. Berlin : Springer, 1998. – ISBN 3540642560

[RHSR94] REPS, Thomas W. ; HORWITZ, Susan ; SAGIV, Shmuel ; ROSAY, Genevieve: Speeding up Slicing. In: *SIGSOFT FSE*, 1994, S. 11–20

[Rie01a] RIEDIGER, V.: *The GUPRO C Parser. Projektbericht 5/01, Universität Koblenz-Landau, Institut für Softwaretechnik*. Koblenz : nicht veröffentlicht, 2001

[Rie01b] RIEDIGER, V.: *The GUPRO C Preprocessor. Projektbericht 4/01, Universität Koblenz-Landau, Institut für Softwaretechnik*. Koblenz : nicht veröffentlicht, 2001

[RR95] REPS, Thomas ; ROSAY, Genevieve: Precise interprocedural chopping. In: *SIGSOFT '95: Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*. New York, NY, USA : ACM Press, 1995. – ISBN 0–89791–716–2, S. 41–52

[Rum96] RUMBAUGH, James: *OMT insights : perspectives on modeling from the Journal of Object-Oriented Programming*. New York : SIGS Books, 1996. – ISBN 1884842585

[SBR06] SCHWARZ, Hannes ; BILDHAUER, Daniel ; RIEDIGER, Volker: *JGraLab Citymap Tutorial*. noch nicht veröffentlicht, November 2006

[Sch07] SCHWARZ, Hannes: *Program Slicing - Ein dienstorientiertes Modell*. Vdm Verlag Dr. Müller, 2007

[Sed02] SEDGEWICK, Robert: *Algorithmen*. 2. Addison-Wesley, 2002. – ISBN-10: 3827370329

[Seu97] SEUBERT, Michael: Business Objekte und objektorientiertes Prozeßdesign. In: *Entwicklungsstand und Entwicklungsperspektiven der Referenzmodellierung*, 1997, 46-64

[SG96] SHAW, Mary ; GARLAN, David: *Software Architecture - Perspectives on an Emerging Discipline*. N. J. : Prentice Hall, 1996. – ISBN 0–13–182957–2

[Ste96] STEENSGAARD, Bjarne: Points-to Analysis in Almost Linear Time. In: *Symposium on Principles of Programming Languages*, 1996, 32-41

[Sun99] SUN: *Code Conventions for the Java Programming Language*. `http://java.sun.com/docs/codeconv/`, April 1999

[Ull07] ULLENBOOM, Christian: *Java ist auch eine Insel*. 6., aktualisierte und erweiterte Auflage. Bonn : Galileo Computing, 2007 `http://www.galileocomputing.de/openbook/javainsel6/`

[Wei79] WEISER, Mark D.: *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*, Diss., 1979

[Wei82] WEISER, Mark: Programmers Use Slices When Debugging. In: *Commun. ACM* 25 (1982), Nr. 7, S. 446–452

[Win00] WINTER, Andreas: *Referenz-Metaschema für visuelle Modellierungssprachen*. Wiesbaden : Deutscher Universitätsverlag, 2000 `http://www.uni-koblenz.de/~ist/documents/Winter2000RFV.pdf`. – zugl. Dissertation, Institut für Informatik. Universität Koblenz-Landau.