Konvertierung von Systemabhängigkeitsgraphen aus dem kommerziellen Tool CodeSurfer

Tobias Blaschke

17.12.2012

Inhaltsverzeichnis

1. Einleitung

1.1. Motivation und Ziel

Joana ist ein aktuelles Forschungsprojekt des KIT. Ziel dieses Projektes ist es durch ein genaues Abbild der Informationspfade innerhalb eines Programms automatisierte Analysen beispielsweise bezüglich der Sicherheit eines Programms durchführen zu können. Nahm diese Projekt seinen Ursprung in *Valsoft*, einem Werkzeug zur Analyse von C-Programmen, ist es nunmehr auf Java beschränkt: Es analysiert den Bytecode.

Bei CodeSurfer der Firma GrammaTech handelt es sich zunächst um ein Debuggingwerkzeug, welches dem Entwickler Informationen darüber liefert, wodurch eine Variable an einer Programmstelle beeinflusst werden kann (es berechnet dafür sogenannte *Slices*, siehe Kapitel 2.2.5). Der für diese Berechnung benötigte *Systemabhängigkeitsgraph* (siehe Kapitel 2.2.6) bildet konzeptionell die Grundlage der mittels Joana durchgeführten Analysen.

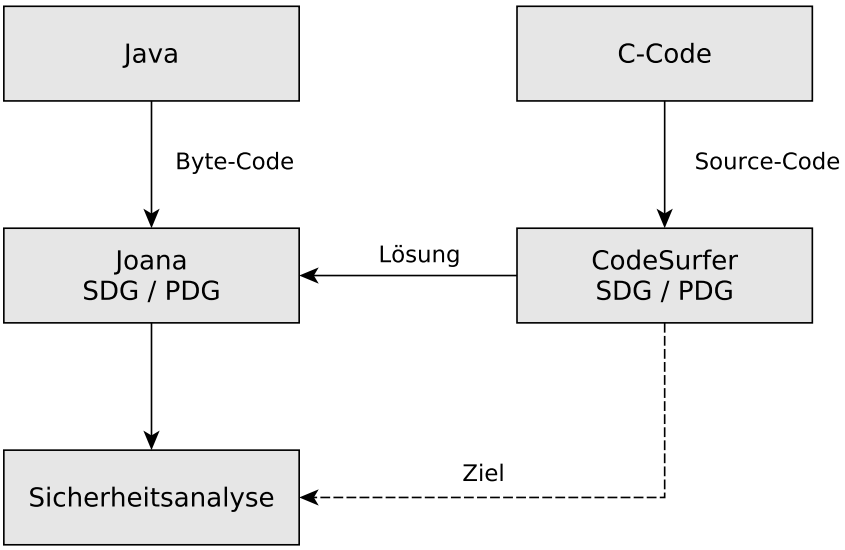

Abbildung 1: *Motivation und Ziel der Arbeit*

Motivation dieser Arbeit ist es nunmehr die Analyse von C-Code dadurch wieder zu ermöglichen, dass die dafür benötigten Informationen aus CodeSurfer extrahiert werden.

Das Ziel liegt somit in der Implementierung eines Konverters begründet, welcher die aus CodeSurfer zu erlangenden Informationen in ein Joana-lesbares Format bringt.

Abschließend sind beide Formate gegenüberzustellen.

1.2. Struktur der Arbeit

Zunächst wird ein kurzer Abriss über die Grundlagen von Systemabhängigkeitsgraphen sowie ein Beispiel für deren Verwendung gegeben. Anschließend erfolgt eine Analyse der

jeweiligen Formate, sowie die Feststellung der Randbedingungen der späteren Implementierung des Konverters. Schließlich wird auf die Implementierung selbst eingegangen, um zuletzt die Qualitäten beider Programme gegeneinander aufzuwiegen.

2. Grundlagen

Gegenstand dieser Arbeit ist die Konvertierung von Informationen aus CodeSurfer zu Joana. Nach einem kurzen Abriss beider Werkzeuge wird auf die Grundlagen der generierten Daten eingegangen, um schließlich die für die Implementierung benötigten Hilfsmittel anzusprechen.

2.1. Die beiden Werkzeuge

Bei CodeSurfer der Firma GrammaTech handelt es sich zunächst um einen Browser für C-Sourcecode, welcher die Effekte von Anweisungen mit berücksichtigt. Dadurch soll das Debuggen und das Verständnis eines Programms erleichtert werden.

Bei Joana handelt es sich um ein anfangs unter dem Namen *Valsoft* entwickeltes Werkzeug, welches zunächst ebenfalls auf C-Programmen operierte und aufgrund der erhobenen Daten automatisierte Sicherheitsanalysen bietet; wurde es später mit der Adaption auf Java-Bytecode in *Joana* umbenannt. Schließlich wurde die Unterstützung für C-Code aufgegeben.

2.2. Die verarbeiteten Daten

Kernpunkt der von CodeSurfer bzw. Joana erstellten Daten bildet der sogenannte *Systemabhängigkeitsgraph* (SDG - System Dependence Graph). In diesem Kapitel sollen zunächst die Grundlagen zum Verständnis eines SDG geschaffen werden um diesen schließlich selbst zu erklären.

Zur exemplarischen Verdeutlichung der dargestellten Konstrukte wird im Folgenden der in Listing 1 dargestellte C-Sourcecode zugrundegelegt.

```
1  int i = 5;
2  int j = 3;
3
4  if (i > j) {
5      i = j;
6      j = 3;
7  } else {
8      j = 2;
9  }
10
11 print(j);
12 j=1;
13 print(j);
```

Listing 1: *C-Sourcecode der späteren Beispiele*

2.2.1. Kontrollfluss

Die Reihenfolge, in der Anweisungen eines Programms ausgeführt werden, nennt man Kontrollfluss. Dieser lässt sich ein einem Graphen darstellen: die Knoten des Graphen bilden dann die Anweisungen; die Kanten legen die Ausführungsreihenfolge fest [1]. Ein Knoten hat mehr als eine Ausgangskanten, wenn es sich bei ihr um die Abfrage einer Bedingung mit Ziel der Kontrolflussänderung handelt.

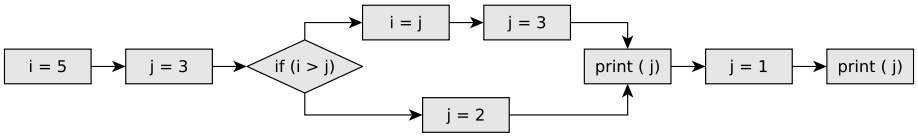

Abbildung 2: *Programmablaufplan mit Kontrollflusskanten*

2.2.2. Kontrollabhängigkeit

Kontrollabhängigkeiten [13] kann man mit Hilfe von *Dominators* (Siehe Kapitel 2.3.1) berechnen. Im Gegensatz zum *Kontrollfluss* wird durch *Kontrollabhängigkeiten* keine Ausführungsreihenfolge vorgegeben. Diese ist bei nicht Vorhandensein von Datenabhängigkeiten (siehe Kapitel 2.2.3) auch in der Tat unerheblich. Durch dieses Vorgehen ist es möglich alle unterschiedlichen Ausführungsreihenfolgen, die bei der Übersetzung entstehen können[1] geschlossen zu Betrachten.

Kontrollabhängigkeiten fußen zunächst grundsätzlich im Startknoten des Graphen oder in einem Prädikat [7]. Weiterhin können in Joana und CodeSurfer zusammenfassende Knoten als Fußpunkt auftreten. Deren Verhalten lässt sich jedoch auf das Vorgenannte zurückführen.

Das Vorhandensein einer Kontrollabhängigkeit zeigt also an, dass bei Ausführung des referenzierenden Knotens – unter Berücksichtigung einer optionalen Bedingung – immer auch der referenzierte Knoten ausgeführt wird, wenn auch nicht zwangsläufig unmittelbar. [7]

Der somit entstehende gerichtete Graph nennt sich *Kontrollabhängigkeitsgraph (CDG)*.

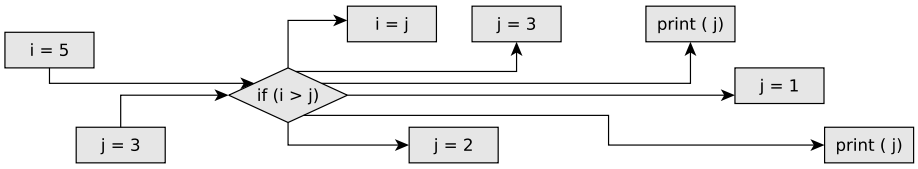

Abbildung 3: *Kontrollflussgraph (CFG) mit vorgezogenen Kontrollabhängigkeitskanten*

[1] Compiler können bei der Code-Optimierung die Ausführungsreihenfolge verändern. Dazu werden allerdings keine CDGs herangezogen, da diese zu aufwändig zu berechnen sind. Viel mehr wird dies durch ein Verfahren, wie BUPM bewerkstelligt.

2.2.3. Datenabhängigkeit

Eine Datenabhängigkeit [13] entsteht, wenn eine Anweisung auf einen von einer anderen Anweisung generierten Wert zugreift. Greifen also zwei Anweisungen nur lesend auf diesen zu, so existiert zwischen ihnen keine Datenabhängigkeit.

Datenabhängigkeiten werden transparent gegenüber *Aliasing*, also der Variablen und deren möglichen Referenzierung, erkannt.

Voraussetzung für das Vorhandensein einer Datenabhängigkeit ist die Existenz eines Pfades auf dem Kontrollflussgraph.

Selbstverständlich lassen sich Datenabhängigkeiten wieder als Graph darstellen. Dieser Graph nennt sich *Datenabhängigkeitsgraph (DDG)*.

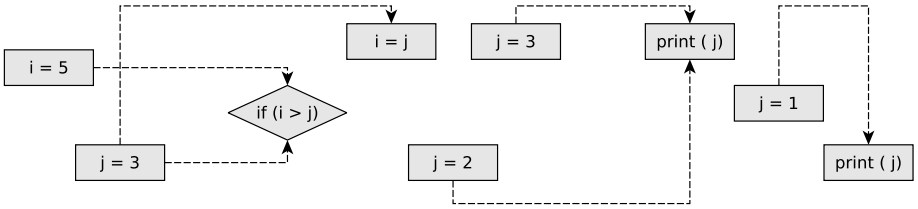

Abbildung 4: *Datenabhängigkeitsgraph (DDG)*

2.2.4. Programmabhängigkeitsgraphen (PDG)

Aus den beiden vorgenannten Graphen lässt sich ein *Programmabhängigkeitsgraph (PDG)* [12] [6] erstellen, in dem die Graphen "übereinandergelegt" werden.

Nach Definition in [7] sind in einem PDG zunächst nur skalare Variablen, Zuweisungen, Bedingte Anweisungen und *while*-Schleifen zugelassen. Der Graph selbst ist von gerichteter Natur, wobei es verschiedene Typen von Kanten gibt: Die Datenabhängigkeit, die unbedingte Kontrollabhängigkeit und die bedingte Kontrollabhängigkeit, welche mit einem *Label* zur Beschreibung des Verhaltens versehen ist.

Zusätzlich zu dieser Definition fügen sowohl Joana als auch CodeSurfer Kontrollflusskanten zu dem Graph hinzu.

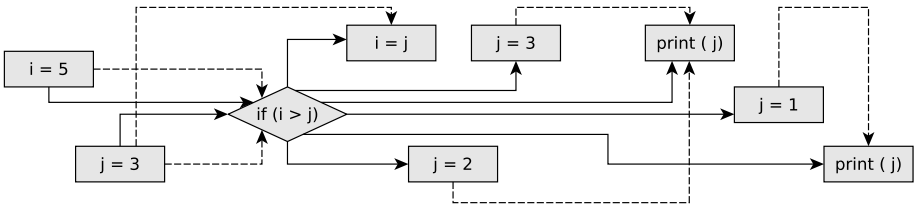

Abbildung 5: *Programmabhängigkeitsgraph (PDG)*

Weiterhin ist zu berücksichtigen, dass in dieser ursprünglichen Definition der PDGs die zugrundeliegende Sprache stark eingeschränkt ist. Insbesondere sind keine Funktionsauf-

rufe erlaubt. Somit eignet sich dieser Graph nur zur Abbildung *Intraprozeduralen* Verhaltens. Diese Einschränkung wird in den später folgenden *Systemabhängigkeitsgraphen* angegangen.

2.2.5. Program Slicing

Ein *Slice* kann bezüglich eines Punktes der Programmausführung (p) und einer Variable (x) gebildet werden. Es handelt sich dabei um die Menge aller Anweisungen dieses Programms, welche nach der Ausführung von p den Wert der Variable x potentiell beeinflussen können [7].

Der *Slice* ist somit vergleichbar mit der Bildung der *transitiven Hülle* auf dem DDG eines Programms, jedoch genauer als diese, sobald man den *Slice interprozedural* berechnet (s.u).

Analog zu einem *Slice* definiert man den *Backwards Slice* als alle Anweisungen eines Programms, welche eine Variable x beeinflussen können und zwischen Programmstart und einer Anweisung p liegen. Zur expliziteren Differenzierung wird der anfangs genannte *Slice* auch *Forward Slice* genannt.

Berechnung eines interprozeduralen Slices Einen ersten Ansatz für die *interprozedurale Berechnung* lieferte [14]. Diese verläuft in zwei Schritten: Zunächst wird der *Slice intraprozedural* in der Prozedur berechnet, welche die Slicing-Kriterien (also den Programmpunkt p und die Variable x) enthält. Bezüglich des Aufrufs von Prozeduren werden *Summary-* Informationen dieser Prozedur genutzt (siehe *Summary Kanten* in Kapitel 4.4.9), allerdings wird die Prozedur für die Berechnung nicht "betreten".

Im zweiten Schritt wird anschließend ein neues Slicing-Kriterium für jede Variable x_n in jeder Prozedur p_n, welche x beeinflussen könnte, generiert.

Anschließend springt man in den ersten Schritt zurück, solange bis keine neuen Kriterien mehr erzeugt werden.

Bei dieser *Fixpunktiteration* handelt es sich um eine *konservative Approximation*[2] der Berechnung des *Slices*: Der Kontext der einzelnen Aufrufe wird hier wenig berücksichtigt.

Um das Ergebnis des *Interprozeduralen Slicings* zu präzisieren wird in Horwitz04 [7] ein zu Ottenstein84 [12] ein zunächst ähnlicher Ansatz verfolgt. Allerdings wird dort der *Systemabhängigkeitsgraph* eingeführt, welcher mehr Informationen über Aufrufumgebungen enthält.

2.2.6. Definition von Systemabhängigkeitsgraphen (SDG)

Ein Problem bei der Erstellung von Programmabhängigkeitsgraphen entsteht, wenn in dem Quellprogramm Funktionen definiert sind: In der ursprünglichen Definition der *PDG* waren Funktionsaufrufe noch nicht vorgesehen.

Zur Darstellung *Interprozeduralen* Verhaltens mittels eines *SDG* wird zunächst für jede Prozedur ein *PDG* erstellt. Dieser wird daraufhin an den *call-sites* erweitert: Enthielt

[2] Die *Konservative Approximation* umfasst mehr potentielle Codezeilen als nötig wären, um die Abhängigkeiten am fraglichen Programmpunkt aufzulösen.

der *PDG* (siehe Kapitel 2.2.4) nur direkte Abhängigkeiten, so ist der *SDG* um durch Prozeduraufrufe induzierte transitive Datenabhängigkeiten [7] ergänzt.

Für diese Ergänzungen werden zunächst neue Typen von Knoten zum Zusammenfassen des Programmzustands[3], sowie Kanten zur Übergabe dieses Zustandes in die Prozeduren hinein induziert. Durch diese Entitäten werden die jeweiligen Aufrufstellen, sowie Funktionseinsprungspunkte umschlossen.

Schließlich wird das Verhalten von Funktionen durch *Summary-Kanten* zusammengefasst.

Eine detaillierte Aufstellung dieser Knoten und Kanten, sowie deren jeweilige Funktion findet sich, am Beispiel von Joana, in den Kapiteln 4.3 und 4.4.

2.2.7. Komplettes Beispiel eines SDG

Um ein Beispiel eines Kompletten SDG zu geben muss das zuvor verwendete Beispiel noch um die Implementierung der *print()*-Funktion erweitert werden. Da der SDG in dem Fall, dass das Programm auf einem PC ausgeführt wird noch um weiteren Initialisierungscode und komplexeres Verhalten erweitert werden müsste, ist der Code zur direkten Ausführung (ohne Betriebssystem) auf einem *Atmel AVR-8* Mikrocontroller konzipiert.

Der nun komplette Source-Code findet sich in Listing 2. Die *print()*-Funktion gibt eine Integer-Zahl rückwärts auf einer Seriellen Schnittstelle (*UART* - Universal Asynchronous Receiver Transmitter) des Mikrocontrollers aus.

[3] Durch eine *Umgebung* werden Variablen und der Programmzustand (insbesondere der *Instruction Pointer*) zusammengefasst. Neben der *Globalen Umgebung* existiert bei modernen Sprachen eine Umgebung je Funktion. Diese wird auf Low-Level-Ebene durch ein *Activation Record* (*Stack Frame*) realisiert.

```
1  /* UCSRA and UDR are memory-adressed registers
2     UDRE is a #defined constant */
3
4  int print(int v) {
5      int pos = 0;
6      while (v > 0) {
7          while(!(UCSRA & (1<<UDRE)));     // Warten: UART-Ready
8          UDR= (char)('0' + (v % 10));     // Ausgabe auf UART
9          v /= 10;
10         pos++;
11     }
12     return pos;
13 }
14
15 void main(void) {
16     int i = 5;
17     int j = 3;
18
19     if (i > j) {
20         i = j;
21         j = 3;
22     } else {
23         j = 2;
24     }
25
26     print(j);
27     j=1;
28     print(j);
29 }
```

Listing 2: *Erweiterung des Beispiel Source-Codes*

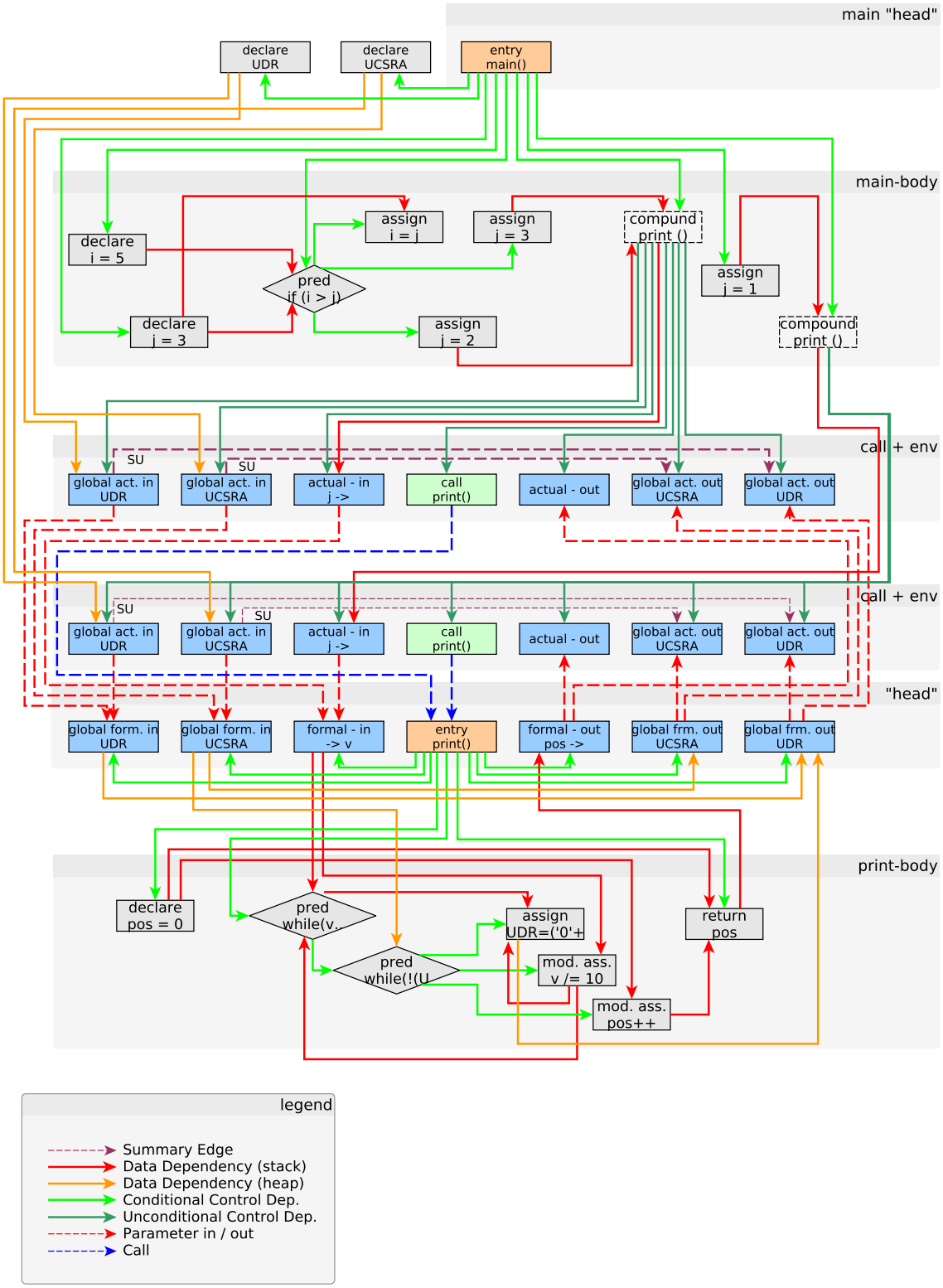

Abbildung 6: *Beispiel eines Systemabhängigkeitsgraphen*

Das Beispiel stellt den SDG in seiner einfachen Form dar: Bei einem *Finegrained SDG* würden die *Expression*-Knoten weiter aufgespalten. Zu erkennen sind Datenabhängigkeiten (Rot) und Kontrollabhängigkeiten (Grün) sowie Parameter-Kanten und schließlich Call-Kanten. Die Nomenklatur richtet sich hierbei weitestgehend nach Joana, lediglich die Benennung der Formalen Parameter Knoten richtet sich nach CodeSurfer.

Eine detaillierte Betrachtung der auftretenden Knoten und Kanten befindet sich in den Kapiteln 4.3 und 4.4.

Weiterhin sieht man, dass die aktuelle Umgebung[4] an den jeweiligen Aufrufstellen zusammengefasst wird und dass das Verhalten der Funktion über Summary-Kanten zusammengefasst wird.

CodeSurfer und Joana enthalten weiterhin Kontrollflusskanten, welche allerdings nicht zum eigentlichen SDG gehören und aus Übersichtlichkeitsgründen nicht dargestellt sind.

2.3. Grundlagen der Implementierung

Nachdem die zu verarbeitenden Daten kurz umrissen wurden, sei im Folgenden weiterhin auf Themen eingegangen, welche für die Implementierung als bekannt vorausgesetzt werden.

2.3.1. Dominatoren

Ein Werkzeug zur Berechnung von *Kontrollabhängigkeiten* ist die Berechnung von sogenannten *Dominatoren* aus den Kontrollflussverläufen eines Programms. Im späteren Verlauf der Arbeit wird diese Begrifflichkeit benötigt, weshalb sie hier kurz angerissen sei; eine Analyse zum Erkennen von Kontrollabhängigkeiten ist hingegen nicht Gegenstand dieser Arbeit.

Ein Knoten A eines Graphen heißt von einem anderen Knoten B *dominiert* [13] [9], genau dann wenn alle Eingangspfade des Kontrollflusses in A über B führen. Die Dominatoren eines Knotens bilden somit eine Untermenge der Vorgänger dieses Knotens auf dem Kontrollflusspfad, wobei Bereiche verzweigten Programmablaufes keine Elemente dieser Menge bilden (Siehe Abbildung 7).

Offensichtlich dominiert sich jeder Knoten selbst. Weiterhin wird jeder Knoten eines Programms durch den Start-Knoten des Programms dominiert.

Die im späteren verwendete Notation für Dominanz lautet $B \succeq A$, wenn A von B dominiert wird. Bei $B \succ A$ ist $B = A$ ausgeschlossen.

Interessant ist weiterhin der *Immediate Dominator* [13] ($idom(A)$) eines Knotens. Dabei handelt es sich um den Dominator, welcher den Pfad mit der geringsten Distanz zu A aufweist, jedoch nicht A selbst ist.

Schließlich sei noch auf die *Semi Dominators* [9] ($sdom(A)$) eingegangen, welche auch in einer verzweigten Programmstelle liegen können und somit keinen Dominator im strengen

[4] Durch eine *Umgebung* werden Variablen und der Programmzustand (insbesondere der *Instruction Pointer*) zusammengefasst. Neben der *Globalen Umgebung* existiert bei modernen Sprachen eine Umgebung je Funktion. Diese wird auf Low-Level-Ebene durch ein *Activation Record* (*Stack Frame*) realisiert.

Sinne darstellen.

Analog zu *Doninators* definiert man die *Postdominators* [8] als Knoten, über die alle Pfade zum Programmende verlaufen.

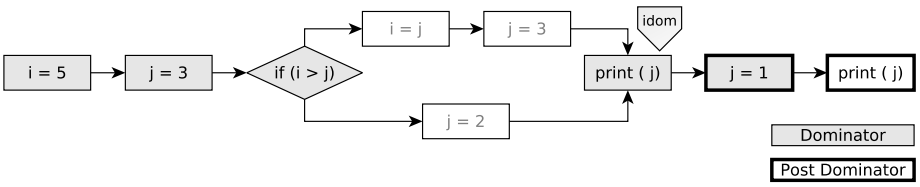

Abbildung 7: *Dominatoren und Immediate-Dominator*

2.3.2. Abstrakter Syntaxbaum (AST)

Ein *Abstrakter Syntaxbaum* ist eine Zwischenrepäsentation eines Programms, welche im Zuge einer Programmübersetzung vom Parser erstellt wird. Diese Baumstruktur wird weiterhin mit zusätzlichen Informationen, wie beispielsweise Datentypen bei Variablen, attributiert. Hochsprachennotationen werden beim Parsen bereits teilweise aufgebrochen, Expressions in ihre einzelnen Rechenoperationen usw.

CodeSurfer emuliert diverse Compiler um einen AST zu erstellen. Ein exemplarischer AST findet sich in Abbildung 8.

2.3.3. XML, XSLT, XPath und DTD

Bei XML [5] handelt es sich um ein generisches textbasiertes Dateiformat, welches sich besonders zur Ablage attributierter Baumstrukturen eignet. Zu Anfang einer solchen Datei kann man deren Inhalt mittels einer *Document-Type-Definition* (DTD [5]) optional festschreiben. XML-Parser werden dann strukturelle Fehler in der Datei automatisch feststellen können.

Die einzelnen Knoten eines in XML spezifizierten Baumes lassen sich mittels einer speziellen Abfragesprache, *XPath* [4], adressieren. Diese Adressierung kann anschließend in einem Ersetzungssystem wie beispielsweise *XSL-Transformations* [10] herangezogen werden, um die Daten in ein anderes Format zu bringen bzw. bei Verwendung temporärer Bäume sogar Baumersetzungsregeln auf die Datenstrukturen anzuwenden.

Das Ergebnis einer XSL-Transformation muss dabei nicht zwangsläufig selbst wieder in einem XML-Format sein.

Da XSLT selbst in einem XML-Format verfasst ist, ist es beispielsweise auch möglich XSLT mittels XSLT aus einer XML-Datei zu generieren.

Innerhalb einer XML-Datei lassen sich Strukturen diverser Dokumentfestschreibungen durch Verwendung von Namespaces verwenden. Diese Namespaces sind am Dateianfang festzulegen. Somit lässt sich eine XML-Datei auch zu späteren Zeitpunkten leicht erweitern.

In der späteren Implementierung kommen XML in der Version 1.0, sowie XSLT und XPath in den jeweiligen Versionen 2.0 zum Einsatz.

3. CodeSurfer

Bei *CodeSurfer* handelt es sich um ein kommerzielles Analysewerkzeug der Firma GrammaTech, welches Entwickler beim Debuggen ihrer Programme – hauptsächlich durch Berechnung von *Slices* (siehe Kapitel 2.2.5) – unterstützen soll. Hierzu berechnet CodeSurfer einen Systemabhängigkeitsgraph (siehe Kapitel 2.2.6), auf welchen man über eine API zugreifen Kann.

3.1. Analyse der CodeSurfer Daten

CodeSurfer [2] legt seine Daten in einem Binärformat ab, was einen direkten Zugriff auf diese als wenig praktikabel herausstellt. Stattdessen gibt es die Möglichkeit der Verwendung der API CodeSurfers; die C-Variante der API erlaubt leider keinen Komplettzugriff auf alle Daten. Aus diesem Grund muss auf die Scheme-API zurückgegriffen werden.

CodeSurfer induziert hier - im Gegensatz zur initialen Definition von Systemabhängigkeitsgraphen von [7] - nur *Kontroll-* und *Datenkanten*. Die weitere Spezifizierung der jeweiligen Eigenschaften der Kanten spiegelt CodeSurfer durch das Erstellen passender *Vertices* wieder.

Weiterhin erlaubt CodeSurfer den Zugriff auf den *Abstrakten Syntaxbaum* (AST) sowohl in einer *normalisierten*, als auch in einer ursprünglichen Variante, welche im Folgenden als μAST referenziert wird.

Die normalisierte Variante "vereinfacht" Sachverhalte in einer Form, die nicht der eigentlichen Coderepräsentation entspricht. Beispielsweise wird $i{+}{+}$ zu $i = i + 1$.

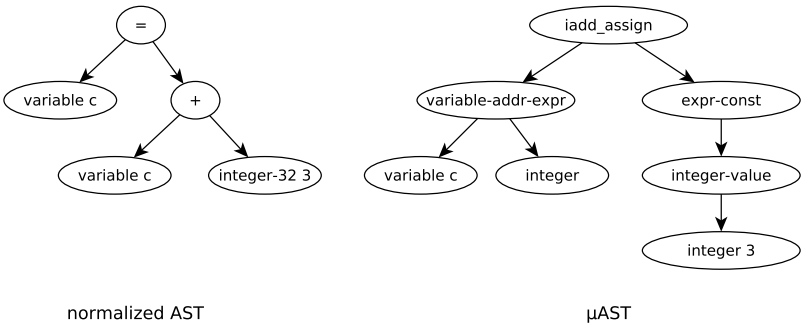

Abbildung 8: *Vergleich normalized- und μ-AST*

Der μAST wird im Folgenden interessant, da CodeSurfer teilweise weniger spezifisch als Joana ist.

4. Joana

Bei *Joana* handelt es sich um ein Werkzeug, welches auf Programmen automatisierte Analysen bezüglich der Sicherheit durchführt. Dabei operiert Joana ebenfalls auf Sys-

temabhängigkeitsgraphen (siehe Kapitel 2.2.6), welche durch Debuggen- und Typinformationen weiter attributiert sind.

4.1. Sprachdefinition des Joana SDG-Formats

Im Gegensatz zu CodeSurfer setzt Joana auf ein menschenlesbares Textformat. Joana kennt mehrere Typen von Knoten (siehe Kapitel 4.3) zwischen denen Kanten gemäß Kapitel 4.4 vorhanden sein können. Weiterhin weisen Knoten folgende (teils optionale) Attribute auf:

Operation (O) dient zur weiteren Spezifizierung des Knotentyps gemäß Kapitel 4.3.1

Wert (V) enthält keine Semantik, sondern Informationen für den "menschlichen Betrachter", meist eine Anweisung aus dem Source-Code

Typfeld (T) der Typ des Rückgabewerts der Anweisung ist abhängig von der Quellsprache. Eine Aufstellung möglicher Werte für ein mittels des im Umfang der Arbeit entstehenden Konverters findet sich in Kapitel 4.1.1, die Vorgehensweise des Konverters in Kapitel 5.11.

Prozedur (P) die numerische ID der Prozedur, zu der der Knoten gehört

Source Referenz (S) die Datei sowie Zeilen- und Spaltennummer im Sourcecode

Bytecodereferenz (B) die Position der Knoten im Compilat. Wie die *Komponente* ist auch diese Referenz für C-Programme nicht wirklich bestimmbar. Jedoch existieren hier Sonderbelegungen, auf welche in Kapitel 5.10 eingegangen wird.

Komponente (C) die Java-Komponente wird für den *Classloader* benötigt, ist für C-Programme also nicht anwendbar. Dennoch wird der Versuch unternommen hier Platzhalter zu setzen.

In so genannten *Fine Grained SDG*s werden weiterhin alle Vertices so weit wie möglich aufgespalten. Dies trifft vor allem auf *Expression-Vertices* zu. Aufgrund der dadurch entstehenden massiven Größen der Graphen ist dies jedoch nicht grundsätzlich erwünscht.

4.1.1. Spezifikation des Typfelds

Die aus der PDG-Generierung von Java-Bytecode bekannten Werte des Typfelds lassen sich nicht bijektiv auf Typen abbilden, wie sie bei der Generierung mittels CodeSurfer auftreten. Aus diesem Grund wird hierfür eine Eigene Notation eingeführt:

T "*typedef*:*size*";

Wobei *size* die Größe des verwendeten Speicherbereichs in Byte (z.B. 4 für ein 32-Bit Integer) und *typedef* den Typen in einer an C angelehnten Notation angibt. Beispiele:

Ein *size*-Wert von *0* tritt auf, wenn die Größe nicht explizit angegeben werden kann bzw. der Typ noch nicht komplett ist. Unvollständige Typen sollten während der Konvertierung aufgelöst worden sein.

T	"*(int):4";	Pointer auf einen Integer-Wert in einem 32-Bit Compilat
T	"int (int):0";	Typ einer Funktion mit Integer-Parameter und Integer-Return-Type

P	Das Quellprogramm
G_J	Den Programmabhängigkeitsgraphen bzw. SDG von Joana
G_{CS}	Den Programmabhängigkeitsgraphen bzw. SDG von CodeSurfer
V	Die Menge der Knoten
E	Die Menge der Kanten
$(v_i \rightarrow_\tau v_j)$	Eine Kante des Typs τ von $v_i \in V$ nach $v_j \in V$

Tabelle 1: *Formelle Bezeichner*

4.2. Formale Notation

An dieser Stelle sei kurz auf die im folgenden Verwendete formale Notation eingegangen. Neben den Bezeichnern in Tabelle 1 sei weiterhin:

$$\Gamma_{CS} \vdash v : \tau \tag{1}$$

Den Typ τ des Vertex v im Kontext von CodeSurfer, sowie

$$\Gamma_J \vdash v : \tau' \tag{2}$$

den Typ τ' im Kontext von Joana. Das Verhalten von Γ sei über Tabelle 7 in Anhang A beschrieben.

4.3. Beschreibung der Joana-Knoten

Ein Knoten besitzt zunächst einen *Typ*, welcher dann durch eine dem Knoten zugeordnete *Operation* weiter spezifiziert wird.

Es existieren folgende Typen:

NORM Beinhalten Deklarationen und einfache Anweisungen.

EXPR Beinhalten eine Berechnung. Sie liefern von daher im Programm einen Wert zurück. Auch Wertzuweisungen an Variablen sind von diesem Typ.

PRED Prädikate sind spezielle Expressions, die über den weiteren Programmverlauf entscheiden. Sie bilden Fußpunkte für *CD-Kanten* (siehe 4.4.5).

CALL Zeigen einen Funktionsaufruf an. Sie treten idR. in Kombination mit *ACTI*- und *ACTO*-Knoten auf.

ACTI Entsteht für jede Variable der aktuellen Umgebung, auf die durch die aufzurufende Funktion zugegriffen wird. Insbesondere also für Parameter der Funktion, aber auch für *Globale Variablen*. Einem *ACTI* ist jeweils ein *FRMI* zugeordnet.

ACTO Ist ebenfalls auf der *Caller*-Seite vorzufinden. Diese Knoten dienen dazu Änderungen, die durch den Funktionsaufruf an Variablen vorgenommen wurden, auf die aktuelle Umgebung anzuwenden. Ihnen ist ein *FRMO* zugeordnet.

ENTR Der Einsprungspunkt einer Funktion. Ausgehend von diesem Knoten werden zunächst alle Variablen durch *FRMI* abgeholt um schließlich den eigentlichen Funktionskörper auszuführen. Alle Knoten auf dem CF-Pfad zwischen *ENTR* und *EXIT* gehören zu der aufgerufenen Funktion.

EXIT Letzter Knoten einer Funktion. Von ihm aus wird das Programm beim Aufrufer (idR. an einer *ACTO*-Knoten) fortgesetzt. Durch eine *EXIT*-Knoten wird eine Variable an den Aufrufer zurückgegeben. Für weitere Parameter werden *FRMO*-Knoten angelegt.

FRMI Repräsentiert einen formalen Eingangsparameter. Dieser Knoten dient zur Zuordnung einer Variable der aufrufenden Umgebung zu der aktuellen Umgebung.

FRMO Steht für jeweils einen Rückgabeparameter der Funktion bzw. auch für Änderungen globaler Variablen.

Nicht näher betrachtet sind hier die Typen *SYNC*, *JOIN* und *FOLD*, welche bei Behandlung von Threads bzw. zum Zusammenfassen auftreten.

4.3.1. Vertex-Operationen

Operationen dienen zur weiteren Spezifizierung des Typs eines Knotens. Mögliche Operationen sind:

declaration (erlaubt in *NORM*) Eine Deklaration, jedoch ohne Initialisierung der Variable. Diese treten an Funktionsanfängen auf.

IF (erlaubt in *NORM* und *PRED*) Nicht zu verwechseln mit dem Shannon-Operator ($\cdot?\cdot : \cdot$, der durch *question* repräsentiert wird). Hauptsächlich sind *IF*-Operationen in *PRED*-Knoten zur Programmverzweigung vorzufinden.

loop (erlaubt in *NORM*) Zeigt eine Schleife an, wobei die Schleifenbedingungen gesondert zu prüfen sind. Schleifen finden sich jedoch meistens in der Form eines *PRED*-Knotens, wobei die Schleife selbst durch einen entsprechenden Kontrollflussverlauf angezeigt wird.

jump (erlaubt in *NORM*) Entsteht bei *goto*, *break*, *continue*, also bei unbedingten Sprüngen. Sie bilden den Fußpunkt für *Jump-Flow (JF)* Kanten.

compound (erlaubt in *NORM*) Dient der Zusammenfassung mehrerer Knoten, hat selbst aber keine Bedeutung: die Kanten könnten genau so gut an alle transitiv durch gereicht werden.

form-in (erlaubt in *FRMI*) Ist die Standardoperation eines *FRMI*-Knotens, welche diese zu einer "normalen" Parameterübergabe veranlasst (im Gegensatz zu *form-ellip*).

form-ellip (erlaubt in *FRMI*) Ellipse-Parameter[5] werden aktuell nicht gesondert behandelt.

act-in (erlaubt in *ACTI*) Ist die Standardoperation eines *ACTI*-Knotens und zeigt somit die Übergabe einer "echten" Variable (also einer nicht konstanten Variable) an.

intconst (erlaubt in *ACTI*, *PRED* und *EXPR*) Konstanten ziehen ein anderes Verhalten des Compilers nach sich. Auch ist z.B. bei eines *ACTI*-Knotens mit einer Konstanten kein *ACTO*-Knoten sinnvoll zuordenbar.

shortcut (erlaubt in *ACTI*, *PRED* und *EXPR*) Entsteht bei binärem "und" und "oder" zur Repräsentation von *Lazy Evaluation*.

derefer (erlaubt in *ACTI*, *PRED* und *EXPR*) Zur Darstellung der Auflösung einer Referenz.

rerefer (erlaubt in *ACTI*, *PRED* und *EXPR*) Eine Referenzierung findet man z.B. in *Call-by-Reference*-Situationen vor.

rereference (erlaubt in *ACTI*, *PRED* und *EXPR*)

Da die Angabe einer Operation zwingend ist, existiert eine feste Zuordnung zwischen: *ACTO* und *act-out*, *FRMO* und *form-out*, *EXIT* und *exit*, *ENTR* und *entry*, sowie *CALL* und *call*.

Möchte man einen feingranularen SDG haben, also Expressions weiter aufspalten, so existieren für diese (sowie teilweise für Prädikate) weitere mögliche Operationen, die aber soweit selbsterklärend sein sollten: *floatconst*, *charconst*, *stringconst*, *functionconst*, *question* (Shannon-Operator: $\cdot ? \cdot : \cdot$), *binary*, *unary*, *array*, *select*, *modify*, *modassign*, *assign*

Nicht weiter betrachtet werden hier die Operationen *monitor*, *summary* und *empty*.

4.4. Beschreibung der Joana-Kanten

Kanten werden durch Attributierung des Knotenobjekts erzeugt, indem je Kante eine Zeile angefügt wird, welche aus Knotentyp und Ziel besteht. Die Struktur eines Funktionsaufrufs ist hierbei sehr streng: die erste UN-Kante führt zu dem Ausdruck (bzw. der Funktionskonstante), der die aufgerufene Funktion bestimmt. Die anderen UN-Kanten führen zu den In- und den Out-Parametern.

4.4.1. Kontrollflusskante (CF)

Jeder Knoten des Graphen ist Teil eines Kontrollflusses, welcher den Ablauf des Programms repräsentiert. Knoten haben immer mindestens einen[6] Kontrollflusseingang. Wenn an ihnen Entscheidungen getroffen werden, haben sie mehrere *CF*-Ausgänge. Es ist zu

[5] Ein Ellipse-Parameter wird verwendet, wenn die Anzahl der zu übergebenden Parameter zuvor nicht bekannt ist - beispielsweise die Ersetzungen im Format-String von *printf*.

[6] Außer natürlich der den Programmstart bezeichnende Knoten.

beachten, dass das Auftreten einer *Exception* ebenfalls einen Entscheidungsprozess darstellt, an dem jedoch keine *CD-Kante* (4.4.5) erstellt wird. Ein mögliches Label für eine *CF-Kante* ist *"exc"*.

Ein Spezialfall bilden die *Exit*-Knoten. Hier bezeichnet CodeSurfer die Rücksprünge auch als *Kontrollfluss*, in Joana sind hier ***RF*-Kanten** vorzufinden.

Weiterhin ist zu beachten, dass der Kontrollfluss nicht eins zu eins aus CodeSurfer zu übernehmen ist: An gewissen Stellen muss dieser neu geführt werden, sodass auch alle Knoten besucht werden (siehe Kapitel 5.7)

Regeln für die Erstellung

$$(v_i \rightarrow_{RF} v_j) \in G_J \Leftarrow \exists (v_i \rightarrow_{cflow} v_j) \in G_{CS} \wedge \Gamma_{CS} \vdash v_i : \text{exit} \tag{3}$$

Die restlichen Kontrollflüsse werden übernommen:

$$(v_i \rightarrow_{CF} v_j) \in G_J \Leftarrow \exists (v_i \rightarrow_{cflow} v_j) \in G_{CS} \wedge \text{keine vorherige Regel anwendbar} \tag{4}$$

4.4.2. Kante zum Ausschluss eines Kontrollflusses (NF)

Diese Kanten werden optional gezogen, wenn kein Kontrollfluss vorliegt. Dies ist deshalb sinnvoll, da Joana voraussetzt, dass jeder Knoten des Graphen über einen Kontrollfluss ausgehend von einem *entry*-Knoten erreichbar ist. Tritt in einem Programm jedoch Toter Code auf, so muss man diesen durch *NF-Kanten* kennzeichnen.

Ausgehend von den rohen CodeSurfer Daten sind die Auftrittsstellen schwer festzustellen.

In Kapitel 5.8 ist die Heuristik, die im Post-Processing angewandt wird kurz skizziert. Sie resultiert in einer Konservativen Approximation.

4.4.3. Sprungkante (JF)

Sie ist eine spezielle Kontrollflusskante, die beim Vorhandensein von Sprüngen entsteht, also bei *break*, *continue* und natürlich *goto*. Fußpunkte von *JF-Kanten* bilden demnach *NORM-jump*-Knoten. Als Ziele existieren in CodeSurfer Label-Vertices, wofür es in Joana kein direktes Äquivalent gibt. In der Tat sind diese auch nicht zwangsläufig nötig: Eine Referenzierung der auf ein Label folgenden Anweisung genügt.

Regeln für die Erstellung Der Konverter erstellt keine expliziten JF-Kanten. Stattdessen werden CF-Kanten herangezogen.

4.4.4. Allgemeine Kontrollabhängigkeitskante (CE, UN)

Diese Kante wird genau dann gewählt, wenn bei Erreichen des Kantenfußes entscheidungsfrei eine Kontrollabhängigkeit zu ihrem Ziel besteht. Eine weitere Unterscheidung, ob sich diese Kante innerhalb eines Ausdruckes (dann *CE*) oder nicht (dann *UN*) befindet wird hier nicht getroffen.

Regeln für die Erstellung

$$(v_i \rightarrow_{UN} v_j) \in G_J \Leftarrow \exists (v_i \rightarrow_{control} v_j) \in G_{CS} \wedge \text{keine andere Regel anwendbar} \tag{5}$$

$$(v_i \rightarrow_{CE} v_j) \in G_J \Leftarrow \exists (v_i \rightarrow_{control} v_j) \in G_{CS} \wedge \Gamma_{CS} \vdash v_i : \text{control-point} \wedge \Gamma_{CS} \vdash v_j : \text{control-point} \tag{6}$$

4.4.5. Bedingte Kontrollabhängigkeit (CD)

CD-Kanten zeigen bedingte Programmverzweigungen an. Sie werden von *control-points* abgehend zu Knoten erstellt, die von ihm abhängen. Laut Definition treten *CD-Kanten* weiterhin abgehend vom *entry*-Vertex einer Funktion auf. Die Behandlung von *Exceptions* hingegen weist keine *CD-Kante* auf.

Diese Kante wird statt einer *CE-Kante* gezogen, falls die Gültigkeit abhängig von dem Programmzustand ist. Für die Evaluierung sind sie in der Regel mit einem Label versehen. Fußen sie in einem Prädikat, so ist das Label *true* oder *false*. Fußen sie hingegen in einer *switch*-Anweisung, so sind sie mit dem Variablen-wert beschriftet.

Weiterhin sind sie per Definition nach dem Eingangsknoten einer Methode zu finden; dort allerdings ohne Beschriftung.

Regeln für die Erstellung

$$(v_i \rightarrow_{CD} v_j) \in G_J \Leftarrow \exists (v_i \rightarrow_{control} v_j) \in G_{CS} \wedge \left\{ \begin{array}{lll} \Gamma_{CS} \vdash v_i : \text{entry} & \wedge \Gamma_{CS} \vdash v_j : \text{exit} & \vee \\ \Gamma_{CS} \vdash v_i : \text{entry} & \wedge \Gamma_{CS} \vdash v_j : \text{body} & \vee \\ \Gamma_{CS} \vdash v_i : \text{control-point} & & \end{array} \right. \tag{7}$$

4.4.6. Helper-Kante (HE)

Diese Kante verläuft immer entlang einer *CD-* oder *CE*-Kante. Im Vergleich zu diesen wird sie allerdings ausgelassen, sobald Zyklen entstehen. Hauptzweck dieser Kante ist es für das Layout von Graphen mittels *graphviewer* [11] zu dienen.

Regeln für die Erstellung Bei der Feststellung des Vorhandenseins einer *HE*-Kante handelt es sich um einen iterativen Prozess, auf den weiter in Kapitel 5.9 eingegangen wird. Aufgrund dieser Eigenschaft liest sich die formale Darstellung etwas umständlich:

$$\begin{array}{l} G_J^0 = \{\} \\ (v_i \rightarrow_{HE} v_j) \in G_J^{t+1} \Leftarrow \exists (v_i \rightarrow_{control} v_j) \in G_{CS} \wedge \nexists (v_i \rightarrow_{HE} v_j) \in G_J^t \\ (v_i \rightarrow_{HE} v_j) \in G_J \Leftarrow (v_i \rightarrow_{HE} v_j) \in \lim_{t \rightarrow \infty} G_J^t. \end{array} \tag{8}$$

4.4.7. Call-Kante (CL)

Zeigen einen Prozeduraufruf an und gehen dementsprechend von einer *Call-Site* zu einem *Entry*-Knoten. Ein mögliches Label für eine CL-Kante ist *"virtual"*.

Regeln für die Erstellung

$$(v_i \rightarrow_{CL} v_j) \in G_J \Leftarrow \exists (v_i \rightarrow_{control} v_j) \in G_{CS} \wedge \Gamma_{CS} \vdash v_i : \text{call} \wedge \Gamma_{CS} \vdash v_j : \text{entry} \tag{9}$$

4.4.8. Datenabhängigkeitskante (DD, DH)

Sie dient zur Repräsentation einer allgemeinen Datenabhängigkeit. Ist eine speziellere Datenabhängigkeitskante wählbar, so wird dieser natürlich der Vorzug gegeben. Eine *DD*-Kante wird dann gewählt, wenn sich die Daten auf dem Stack befinden. Eine *DH*-Kante zeichnet eine Datenabhängigkeit auf dem Heap aus.

Regeln für die Erstellung

$$\begin{array}{ll} (v_i \rightarrow_{DH} v_j) \in G_J \Leftarrow \exists (v_i \rightarrow_{data} v_j) \in G_{CS} \wedge & \left\{ \begin{array}{l} \Gamma_{CS} \vdash v_i : \text{global-formal-in} \\ \Gamma_{CS} \vdash v_j : \text{global-formal-out} \end{array} \right. \vee \\ (v_i \rightarrow_{DD} v_j) \in G_J \Leftarrow \exists (v_i \rightarrow_{data} v_j) \in G_{CS} \wedge & \text{keine vorherige Regel anwendbar} \end{array} . \tag{10}$$

Wie man sieht erfolgt bei der Konvertierung keine korrekte Auszeichnung von Heap-Abhängigkeiten: Lediglich die erste und letzte Kante innerhalb einer Prozedur werden korrekt beschriftet. Dies liegt darin begründet, dass die Information, ob eine Variable auf dem Heap liegt, in CodeSurfer nur an diesen Stellen vorliegt.

4.4.9. Summary-Kante (SU)

Diese Kante steht für eine transitive Datenabhängigkeit. Vorzufinden ist sie zwischen *formal-in* und *formal-out* einer Funktion. Diese Kante ist in CodeSurfer nicht vorhanden und wird immer zusätzlich erstellt.

Regeln für die Erstellung

$$\begin{array}{l} (v_i \rightarrow_{SU} v_j) \in G_J \Leftarrow \exists (v_i \rightarrow_{data} v_j) \in G_{CS} \wedge \Gamma_{CS} \vdash v_i : \text{actual-in} \wedge \Gamma_{CS} \vdash v_j : \text{actual-out} \vee \\ \Gamma_{CS} \vdash v_i : \text{global-actual-in} \wedge \Gamma_{CS} \vdash v_j : \text{global-actual-out} \end{array} . \tag{11}$$

4.4.10. Parametereingangskante (PI)

Dient zur Übergabe von Parametern an Funktionen. Dementsprechend ist sie zwischen *actual-in* und *formal-in* vorzufinden, laufen also über Prozedurgrenzen hinweg.

Regeln für die Erstellung

$$\begin{array}{l} (v_i \rightarrow_{PI} v_j) \in G_J \Leftarrow \exists (v_i \rightarrow_{data} v_j) \in G_{CS} \wedge \Gamma_{CS} \vdash v_i : \text{actual-in} \wedge \Gamma_{CS} \vdash v_j : \text{formal-in} \vee \\ \Gamma_{CS} \vdash v_i : \text{global-actual-in} \wedge \Gamma_{CS} \vdash v_j : \text{global-formal-in} \end{array} . \tag{12}$$

4.4.11. Parameterstrukturkante (PS)

Diese Kante wird herangezogen um strukturelle Abhängigkeiten zwischen Parameterknoten darzustellen. Sie dient zur Extraktion eines Wertes aus einem Feld (z.B. *C-struct*).

4.4.12. Parameterausgabekante (PO)

Hierbei handelt es sich um das passende Gegenstück der *PI*-Kante. Sie werden von dem *Exit*-Knoten zum *Actual-Out*-Knoten der aufrufenden Funktion gezogen.

Regeln für die Erstellung

$$(v_i \rightarrow_{PO} v_j) \in G_J \Leftarrow \exists (v_i \rightarrow_{data} v_j) \in G_{CS} \wedge \Gamma_{CS} \vdash v_i : \text{formal-out} \wedge \Gamma_{CS} \vdash v_j : \text{actual-out} \vee \Gamma_{CS} \vdash v_i : \text{global-formal-out} \wedge \Gamma_{CS} \vdash v_j : \text{global-actual-out} \quad . \tag{13}$$

4.5. Unterschiede der Formate

Im Vergleich zu Joana generiert CodeSurfer nach Funktionseintritten Declaration-Vertices, um nötigen Speicherplatz für die Ausführung zu reservieren. Dies erinnert etwas an frühere Vorgehen von Compilern, da mittlerweile nicht mehr direkt klar sein muss, dass dieser Platz auch benötigt wird: Variablen können sich während der Ausführung der Funktion auch rein in Registern bewegen.

Ein weiterer Unterschied ist darin festzustellen, dass CodeSurfer Berechnungen auch in den *Actual-Out* Knoten verschiebt. An dieser Stelle stünde in einem mit Joana erstellten SDG ein extra Knoten mit entsprechenden Abhängigkeiten. Dieses Verhalten wird in der Konvertierung übernommen. Als Resultat daraus kann die Graphabdeckung verschiedener Analysen schwanken (siehe Kapitel 6.3).

Im Vergleich zu Joana verläuft bei CodeSurfer der Kontrollfluss nicht durch sämtliche Knoten: *Declaration*, *Formal-In* werden beispielsweise nicht berücksichtigt (siehe Kapitel 5.7). Da sich aber einige Joana-Algorithmen darauf verlassen, wurde dieses Verhalten nachgebildet. In der XML-Datei weisen diese Kontrollflusskanten das Attribut *type=special* auf.

Das Vorgehen von CodeSurfer bei der Wertrückgabe unterscheidet sich dahingehend von Joana, dass der Wert zunächst in ein *Declaration-Vertex* überführt wird. Auch hier ist die Kontrollflussreihenfolge sehr unterschiedlich und wurde von daher auf den Stil von Joana adaptiert.

Der Mechanismus des Labelns unterscheidet sich sehr stark zwischen beiden Programmen. In CodeSurfer sind Kontrollabhängigkeiten nie beschriftet, dafür jedoch jeder Kontrollfluss. Sollte keine Beschriftung notwendig sein, so lautet die Beschriftung immer auf *true*. Will man eine Kontrollabhängigkeit überprüfen, so muss man das Label des ersten Kontrollflussschritts transitiv durchreichen (siehe Kapitel 5.5).

Die Attributierung von PDG-Vertices mit Datentypen ist in CodeSurfer nicht direkt vorhanden. Man erhält diese Information ausschließlich durch Aktivierung der Erstellung von ASTs. Der AST muss schließlich selbst analysiert werden um den passenden Datentyp herauszufinden (siehe Kapitel 4.1.1).

IDs werden in CodeSurfer nicht programmeindeutig, sondern lediglich PDG-eindeutig vergeben. Sie müssen daher umgeschrieben werden.

Die durch CodeSurfer erstellten SDGs befinden sich sehr nahe am Source-Code: Durch Aneinanderreihung der Source-Code-Referenzen der einzelnen Vertices erhält man fast den kompletten Text. Da Joana SDGs aus Java-Bytecode erstellt, überrascht es wenig, dass das hier nicht der Fall ist.

CodeSurfer ordnet *call-* und *acti*-Knoten auf dem *CF*-Pfad umgekehrt zu Joana an (siehe Kapitel 5.7.1).

Joana spaltet im Gegensatz zu CodeSurfer *switch-case*-Konstrukte in mehrere *PRED*-Knoten auf, die jeweils *true-*/*false*-gelabelt.

Die Abbildung von Exceptions erfolgt bei CodeSurfer wiederum durch spezielle Vertices (*exceptional-exit* und *exceptional-return*). Joana verwendet hier die typischen Knoten, setzt jedoch auf Kanten, die mit *"exc"* gelabelt sind.

CodeSurfer zieht im Falle einer *modassign*-Operation (z.B. *c++*) eine selbstrekursive Datenabhängigkeitskante, die bei Joana in dieser Form nicht existiert. Diese Kante wird im Postprocessing entfernt.

4.6. Analyse der vorherigen Softwareversion

Im Rahmen seiner Diplomarbeit entwickelte Bernd Nürnberger 2008 [3] das Programm *valsoftize*. Dabei handelt es sich um ein mittels der STK[7]-Engine CodeSurfers interpretierten Scheme-Code. Dieser Code liest die CodeSurfer-Strukturen in eigene Hash-Maps, auf denen im weiteren Analysen durchgeführt werden um schließlich direkt einen Joana-kompatiblen PDG auszugeben.

Leider hat CodeSurfer seit 2008 seine API derart stark überarbeitet, dass *valsoftize* auf aktuellen Versionen nicht mehr lauffähig ist.

[7] STK ist ein Scheme-Dialekt, welcher auf Graphische ausgaben mittels TK optimiert ist.

5. Implementierung

Da CodeSurfer seine Daten in einem Binärformat ablegt, wird davon Abstand genommen diese direkt einzulesen. Statt dessen wird die API-Schnittstelle von CodeSurfer verwendet.

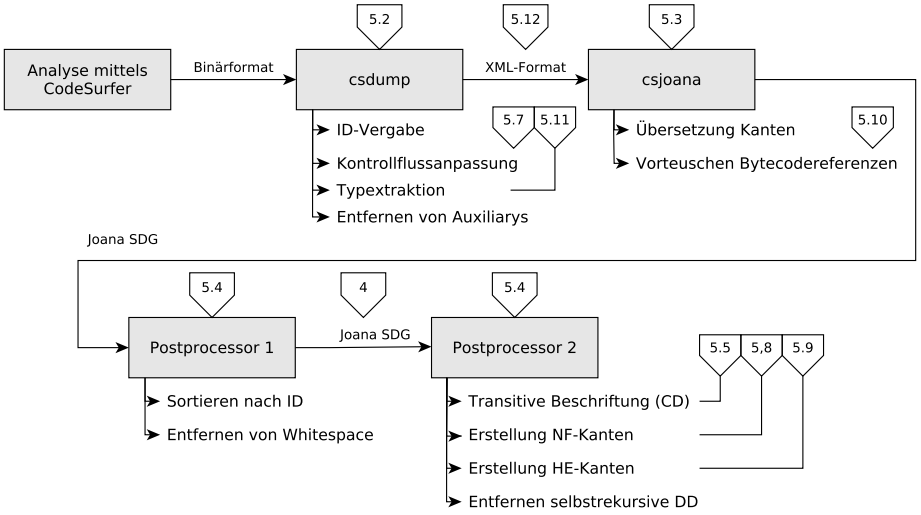

Abbildung 9: *Struktur der Implementierung*

5.1. Grobe Gliederung

Der initiale Gedanke der Implementierung lag dahingehend die Datenkonvertierung von der Informationsakquise abzuspalten. Letzteres sollte durch ein Scheme-Script erfolgen ersteres durch ein XSLT-Script.

Der dahinterstehende Gedanke lag darin begründet, dass man durch dieses Vorgehen eine weitgehende Abspaltung von der CodeSurfer-API gewinnen könne, um gegenüber struktureller Änderungen - wie sie seit 2008 vorkamen - gefeit zu sein; weiterhin lägen die Zusammenhänge beider Darstellungsformen gekapselt vor.

Leider konnte dies eingedenk der sprachlichen Beschränkungen XSLTs nicht strikt durchgezogen werden: Diese Beschränkungen der funktionalen Sprache äußern sich darin, dass es nicht möglich ist, eine eine Rekursion umgebende Variable während des Abstiegs zu redefinieren. Dadurch ist es beispielsweise nicht möglich eine pfadsuchende Rekursion im Falle des Fundes eines erfolgreichen Pfades auf anderen Pfaden abzubrechen. Vielmehr muss man eine Heuristik, wie beispielsweise eine maximale Rekursionstiefe heranziehen, um einen Suchlauf als vergeblich kennzuzeichnen. Resultierend daraus ergibt sich eine nicht tragbare Laufzeit dieses Ansatzes.

Im weiterführenden Vorgehen wurden unter Aufweichung der ersten Idee Berechnungen in das Scheme-Script ausgelagert: Die Berechnung des Kontrollfluss- Verlaufs wurde

vorgezogen. Aufgrund der Besuchsreihenfolge der Knoten, die CodeSurfer vorgibt musste das Script weiterhin um einen zweiten Baumdurchlauf erweitert werden. Bei dieser Anpassung wurde Wert darauf gelegt, dass die CodeSurfer anhaftenden Eigenheiten als alternative Beschreibung beibehalten werden.

Ein weiteres Problem lag bei der Propagierung der Labels von CD-Kanten vor. Um dies zu bewerkstelligen ist offenbar das Wissen über ihre genaue Lage im Joana-SDG vonnöten. Ein Vorziehen dieser Berechnung ist daher ausgeschlossen. Da für die zugehörige Evaluierung allerdings eine Suche entlang der Kontrollflusskante, unter Unkenntnis des erfolgreichen Pfades vonnöten ist, schließt sich eine Implementierung in der XSL-Transformation aus oben dargelegten Gründen ebenfalls aus. Resultierend daraus wurde ein Ansatz in Java gewählt, da hierfür bereits ein Parser zum Einlesen des Joana-Formats vorliegt.

Im Endergebnis erhalten wir nun einen vierfachen Baumdurchlauf mit dreifachem Neuparsen der Daten. Die daraus resultierenden Einflüsse auf die Umwandlungsgeschwindigkeit werden wir im späteren betrachten.

5.2. Datenextraktion aus CodeSurfer

Die Daten werden unter Beibehaltung der von CodeSurfer vergebenen Bezeichnern in einem XML-Format ausgegeben; lediglich die IDs werden geändert, sodass sie Projektglobal eindeutig sind.

Hierzu kommt das Script *csdump.stk* zum Einsatz. Es ist in dem Scheme-Dialekt *STK* verfasst und wird an CodeSurfer übergeben.

Die CodeSurfer-API stellt fast alle benötigten Traversal-Funktionen bereit, um die Datenstrukturen zyklenfrei zu durchlaufen. Mittels dieser Funktionen lässt sich bereits ein erster *SDG* ausgeben. Dieser kann allerdings wie später beschrieben noch erweitert werden.

5.3. Mapping der Vertices

Die so entstandene XML-Datei kann schließlich mittels diverser XSLT-Stylesheets in andere Formate Konvertiert werden.

Hier findet zunächst eine eins zu eins Zuordnung gemäß den in Kapiteln 4.3 und 4.4 statt. Weiterhin werden an dieser Stelle die "Bytecodereferenzen" (Kapitel 5.10) erstellt.

Als Resultat erhält man eine erste Version einer Joana PDG-Datei, die allerdings noch nachbearbeitet werden muss.

5.4. Postprocessing

Im ersten Schritt der Nachbearbeitung werden zunächst die Joana-Vertices mittels Shell-Scripten nach ID sortiert, sowie überflüssiger Whitespace entfernt. Erst jetzt lässt sich die PDG-Datei durch Joana überhaupt zuverlässig parsen.

Im nächsten Schritt werden schließlich die Beschriftungen der CD-Kanten vorwärts propagiert (Kapitel 5.5), sowie die Helper-Kanten eingefügt (Kapitel 5.9).

Schließlich erhält man die fertig konvertierte Datei.

5.5. Beschriftung von CD-Kanten

Da die Beschriftung der Kanten in CodeSurfer und Joana, wie bereits erwähnt, unterschiedlich vonstatten geht, müssen diese wie folgt im Post-Processing umgeschrieben werden:

5.5.1. Beschriftung der ersten CD-Kante

Die jeweils erste CD-Kante wird mittels *XSLT* beschriftet. Sie läuft entlang des Kontrollflusses und übernimmt lediglich die Beschriftung der CF-Kante. Die Implementierung in *XSLT* könnte auch weitere Ersetzungen vornehmen, dies ist aber Aufgrund der untragbaren Geschwindigkeit (siehe Kapitel 5.1) deaktiviert. Stattdessen erfolgt dies im Post-Processing.

5.5.2. Beschriftung der weiteren CD-Kanten

Eine unbeschriftete CD-Kante erhält ihre Beschriftung dadurch, dass die Kontrollflusskanten (CF-Kanten) des Knotens, welcher das Ziel der CD-Kante ist, rückwärts traversiert werden. Bei diesem Vorgang werden weitere Knoten, die auf dem selben Niveau wie der Knoten, auf der der Algorithmus gestartet wurde, in eine Menge der gleich gelabelten Knoten aufgenommen, sofern eine CD-Beziehung vom Fußpunkt der ursprünglichen Kante besteht.

$$S(r \rightarrow_{CD} t_0) = \{r \rightarrow_{CD} t_1 | t_1 = \text{idom}(t_0) \wedge (r \rightarrow_{CD} t_1) \in G_J \wedge r \succ t_1\} \tag{14}$$

Die genannte Traversierung läuft so lange, bis eine bereits beschriftete CD-Kante gefunden wird (die Beschriftung der CD-Kanten, die entlang dem ersten Kontrollflussschritts ausgehend vom Fußpunkt der Gewählten Kante verlaufen, wurde im Kapitel 5.5.1 festgelegt).

Eine Sonderbehandlung muss demnach dann herangezogen werden, wenn in einem CF-Zweig einer Entscheidungsstelle eine weitere Entscheidungsstelle vorkommt: in diesem Fall spalten sich die CF-Rückwärtskanten in mehrere Wege auf und die Verarbeitung muss am *Immediate Dominator* fortgesetzt werden.

Der *Immediate Dominator* wird wie in Kapitel 5.6 beschrieben Gefunden. Dieses Vorgehen terminiert natürlich nicht, wenn eine selbst-rekursive Kante, also eine Endlosschleife, auf dem Pfad liegt. Dies ist in den Bibliothekshelfern von CodeSurfer leider häufig der Fall. In diesem Fall wird der Oben genannte Algorithmus an dieser Stelle unterbrochen und auf einen langsameren vorwärts-schreibenden Algorithmus umgestellt.

Eine alternative Herangehensweise zur Lösung dieses Problems wäre die Verwendung des *Lengauer-Tarjan-Algorithmus*.

5.6. Ermittlung des Dominators der Vorgänger

Achtung:
Im Folgenden werden die dominanzbezogenen Termini nicht gänzlich korrekt benutzt: Die Verwendung wäre lediglich in *GOTO*-Freien Programmen korrekt. Im Folgenden werden *GOTO*s einfach ignoriert, was das gewünschte Verhalten liefert.

Zur Ermittlung des *idoms* gibt es mehrere Möglichkeiten:

- Eine Breitensuche über die Rückwärtsverfolgung der *Kontrollflusskanten*
- Berechnung eines Dominatorbaums mittels des *Lengauer-Tarjan-Algorithmus*
- Analyse der *Kontrollabhängigkeitskanten*

Die hier Implementierte Methode nutzt aus, dass der Fußpunkt einer *Kontrollabhängigkeitskante* selbst ein Dominator ist.

Diese Methode wird rekursiv bis zum Fund des *idoms* ausgeführt[8].

$$\mathrm{pred}_{CF}(x) = \{z \mid (z \rightarrow_{CF} x) \in G_J\} \tag{15}$$

$$\mathrm{pred}_{CD}(y) = \{z \mid (z \rightarrow_{CD} y) \in G_J\} \tag{16}$$

$$\mathrm{sdom}(x) = \{w \mid \forall y \in \mathrm{pred}_{CF}(x)\ \exists w \in \mathrm{pred}_{CD}(y)\} \tag{17}$$

$$\mathrm{idom}(x) = \max_{\succ}(\mathrm{sdom}(x)) \tag{18}$$

Die programmatische Umsetzung lautet etwas unterschiedlich:

function idom(x) :

$tier_1 := \{y \mid (x \rightarrow_{CD} y)\}$
$sdom := \oslash$
while $(sdom \cap tier_1 = \oslash)$: erspart Max-Berechnung, ignoriert Overshoot
$sdom := sdom \cup_{y \in sdom \cup \{x\}} pred_{CD} \circ pred_{CF}(y)$
return $sdom \cap tier_1$ Das Element der einelementigen Menge

5.7. Rerouting des Kontrollflusses

Die nötigen Änderungen des Kontrollflussverlaufs werden beim ersten Baumdurchlauf festgestellt. Dies erfolgt im Scheme-Script derart, dass beim Auftreten eines *entry*- oder *call-site*-Vertex eine Suche stattfindet. Beide Typen seien im folgenden dargestellt.

[8] Durch diese Rekursion sollte kein spürbarer Geschwindigkeitsnachteil entstehen, da die Schachtelungstiefe in üblichen Programmen nicht so tief ist.

5.7.1. Rerouting des Funktionseintritts

Der von CodeSurfer standardmäßig auftretende Kontrollflussverlauf ausgehend von einem *entry*-Vertex zeigt direkt auf ein *body*-Vertex, welches den Funktionskörper enthält (das *body*-Vertex wird durch einen *NORM-compund*-Knoten realisiert). Joana erwartet beim Funktionseintritt jedoch zunächst ein Ablaufen der *formal-in*- und *declaration*-Vertices.

Zum Umschreiben des Kontrollflussverlaufs werden alle vom *entry*-Vertex abgehende *Kontrollabhängigkeiten* betrachtet. Aus diesen wird anschließend eine Kette derart gebildet, dass diese alle referenzierten *global-formal-in*-, *formal-in*- und *declaration*-Vertices (in dieser Reihenfolge) enthält. Abschließend werden die beiden Enden dieser Kette am *entry*- und *body*-Vertex "eingehängt".

5.7.2. Rerouting eines Funktionsaufrufs

Im Falle eines Funktionsaufrufs gibt CodeSurfer folgende Besuchsreihenfolge aus: Zunächst der *call-site*-, daraufhin die *global-formal-in*- und *formal-in*- und schließlich die *global-formal-out*- und *formal-out*-Vertices. Innerhalb dieser Kette existieren weiterhin abzweigende Kontrollflussverläufe für auftretende Exceptions.

Hier werden dementsprechend zum Schreiben der neuen Reihenfolge zunächst alle vom *call-site*-Vertex abgehenden *Kontrollabhängigkeiten* betrachtet, deren Ziel zunächst ein (*global-*)*actual-out* ist. Beim Erzeugen der zugehörigen Kontrollflusskette werden dabei weiterhin von den Elementen abgehende Kontrollflüsse gemerkt, es sei den sie zeigen auf ein (*global-*)*actual-in*.

In einem weiteren Schritt werden alle *Kontrollabhängigkeiten* durchlaufen, deren Ziel ein *actual-in* oder *formal-out* ist. Auch hier werden wieder ausgehende Kontrollflüsse gemerkt, denn bei nicht Vorhandensein von *formal-out*s liegt der im späteren benötigte Verlauf hier.

Abschließend wird die zuvor erstellte *acti*-Kette an allen Vertices eingehängt, die zuvor einen Kontrollfluss zur *call-site* aufwiesen, es sei denn bei diesem Vorgänger liegt bereits ein neugeschriebener Kontrollfluss vor (in diesem Fall muss eine Sonderbehandlung vorgenommen werden (s.u.)). Beim Einhängen ist weiterhin zu beachten, dass wenn der Vorgänger ein *actual-in* ist (das ist der Fall, wenn der Vorgänger ein Funktionsaufrufgebilde ist, bei dem kein *actual-out* vorhanden ist) man entlang des neu geschriebenen Kontrollflusses dieser *call-site* zunächst den korrekten Ausgang finden muss.

Die im vorherigen Absatz genannte Sonderbehandlung sieht derart aus, dass wenn es sich bei dem Vorgänger um einen *control-point* (bzw. einen *PRED*-Knoten) handelt man wie gewohnt verfährt. Andernfalls wird ein Fehler ausgegeben und auf das Einhängen verzichtet. Dies muss anschließend leider manuell vorgenommen werden.

Der Kopf der *actual-in*-Kette ist nun eingehängt. Diese Kette wird zur *call-site* verlassen.

Anschließend muss noch die *actual-out*-Kette eingehängt werden (wenn sie vorhanden ist). Hier ist wiederum zu beachten, dass wenn der Zuvor gemerkte Nachfolger selbst eine *call-site* ist, man entlang der Ketten dieser *call-site* den korrekten Knoten finden muss. Es kann aber auch vorkommen, dass zuvor kein Nachfolger gemerkt wurde (z.B.

bei einer Funktion ohne *actual-in*s und *actual-out*s, wie z.B. *exit();*). In diesem Fall wird der Nachfolgeknoten gesucht.

5.7.3. Datenstruktur eines Kontrollflussreroutingtabelleneintrags

Kontrollflussänderungen werden in einer Hash-Map, deren Schlüssel Vertices sind, gespeichert. Ein Eintrag besteht aus dem neuen Ziel, einer optionalen Beschriftung (z.B. für Exceptions) und einer Referenz auf den Ursprünglichen Kontrollfluss, welcher durch den neuen ersetzt werden soll. Dies ist nötig, falls ein Funktionsaufruf als erste Aktion innerhalb eines If-Statements ausgeführt wird. Diese Referenz kann auch "*" lauten, falls explizit alle anderen Kontrollflüsse ersetzt werden sollen.

5.8. Ausschluss von Kontrollfluss

In Fällen unerreichbaren Codes wird in Joana eine spezielle Auszeichnung dieser Regionen benötigt. Um dies zu bewerkstelligen operiert der Post-Processor derart, dass beim Auftritt von *jump*- und *loop*-Knoten Kanten ausgehend von diesen Knoten gezogen werden.

Das Ziel dieser Kanten bilden zunächst alle Knoten der selben Prozedur, welche keinen eingehenden Kontrollfluss haben, es sei denn es handelt sich bei ihnen um einen *entry*-, *formal-in*- oder *formal-out*-Knoten. Besitzt ersterer von Natur aus keinen eingehenden Kontrollfluss, sind letztere von einem Sprung aus nicht direkt erreichbar.

Weiterhin wird eine *NF-Kante* zu allen Knoten gezogen, welche all ihre Vorgänger dominieren (siehe Kapitel 5.6). Dies ist dann der Fall, wenn innerhalb des Toten Codes eine Schleife existiert.

Dieses Verfahren resultiert somit in einer konservativen Approximation der Erstellung von *NF-Kanten* und weist dadurch weiterhin das Problem auf, dass Fehlverhalten des Konverters (bei fehlendem Kontrollfluss zu einer Stelle, die erreichbar sein sollte) verschleiert werden. Zum Debuggen des Konverters muss somit entweder die Berechnung dieser Kanten deaktiviert werden oder es muss der SDG vor Anwendung des Postprocessors untersucht werden.

5.9. Erstellung der Helper-Kanten

Zur Feststellung, an welchen Stellen Helper-Kanten (gemäß Kapitel 4.4.6) einzufügen sind, muss der SDG daraufhin untersucht werden, durch welche *Kontrollabhängigkeiten* (namentlich *CD*-, *CE*- und *UN-Kanten*) Zyklen entstehen. An diesen Stellen ist die *HE-Kante* auszulassen.

Dies funktioniert durch eine Traversierung der *Kontrollabhängigkeiten*, wobei bereits besuchte Knoten markiert werden. Eine *HE-Kante* wird erstellt, wenn der Zielknoten der Abhängigkeit noch nicht markiert ist.

Die Berechnung der *HE-Kanten* erfolgt im Postprocessing.

5.10. Ausgabe von Bytecodereferenzen

Offensichtlich existiert für C-Compilate kein Bytecode. Dennoch verlassen sich einige Programme auf das Vorhandensein von Bytecodereferenzen. Als Ziel für diese Referenzen sind folgende Spezialwerte gemäß Tabelle 2 vorgesehen.

Num.	Bezeichner
−1	UNDEFINED_POS_IN_BYTECODE
−2	ROOT_PARAMETER
−3	STATIC_FIELD
−4	OBJECT_FIELD
−5	ARRAY_FIELD
−6	BASE_FIELD
−7	ARRAY_INDEX
−8	PHI

Tabelle 2: *Spezielle Bytecodereferenzen*

Demnach erhalten so gut wie alle Knoten nach der Konvertierung den Wert *UNDEFINED_POS_IN_BYTECODE*. In der aktuellen Implementierung bilden hiervon *exit-*, *formal-in-* und *formal-out*-Knoten die einzige Ausnahme. Sie erhalten den Wert *ROOT_PARAMETER*.

Eine weitere Differenzierung ist hier nicht trivial vornehmbar und bildet, wenn überhaupt sinnvoll, Gegenstand einer weiteren Arbeit.

Die Erstellung dieser Referenzen erfolgt im XSLT-Processing.

5.11. Extraktion des Rückgabetyps

Für die Extraktion des Rückgabetyps eines Vertex müssen abhängig vom Vertex-Typ unterschiedliche Herangehensweisen gewählt werden. Diese unterscheiden sich unter anderem auch im Detailgrad. An folgenden Stellen kann der Typ abgefragt werden:

μ-AST Einem Großteil der Vertices ist ein μ-AST zugeordnet. Mit der hierauf angewandten Pretty-Print Funktion findet man Typen, wie sie in C-Deklarationen stehen (d.h. ein mittels *typedef* umbenannter Typ wird unter seinem Alias ausgegeben). Nicht vorhanden ist diese Methode unter anderem bei Vertices der Typen *declaration*, Parameterknoten, *variable-initialization*. In der XML-Datei wird für diese Quelle *source="natural"* gesetzt.

Variable im normalized ast Diese Methode wird gewählt, wenn der Wurzelknoten des dem Vertex assoziierten normalisierten AST vom Typ *variable* ist. Die Quellenangabe in XML lautet auf *source="variable"*

Assignment im normalized AST Hier erhält man die ungenauste Angabe: *char*-Typen werden beispielsweise als *int* angegeben. Das Ergebnis dieser Methode findet lediglich dann ihren Weg in den Joana-SDG, wenn keine andere Methode zu einem Ergebnis führt. Die XML-Quellenangabe lautet *source="assignment"*

Abstrakter Ort einer Deklaration Diese Methode wird bei allen *declaration*-Vertices gewählt. Sie ist mit *source=abs-loc* beschriftet. Bei diesen Typangaben tritt pro Mitglied eines Feldes eine Instanz auf. Bei dieser Art werden umbenannte Typen zu ihrem Basistyp aufgelöst.

Abstrakter Parameter Die Beschriftung lautet hier *source="abs-loc-param"*. Diese Quellen treten bei Parameterübergabe teils zusätzlich zu anderen Quellen auf. Auch hier werden Felder aufgespalten, in diesem Fall muss pro Mitglied ein Vertex angelegt werden, diese Vertices werden dann mittels *PS-Kanten* eingehängt. Bei dieser Art werden umbenannte Typen zu ihrem Basistyp aufgelöst.

Nun muss man die endgültige Typangabe gemäß folgender Heuristik zusammenbauen: Wenn ein einzelner *abs-loc-param* vorliegt, so wird dieser gegenüber allen anderen Typen bevorzugt. Hat eine Typangabe eine Größe von 0, so wird eine andere Typangabe bevorzugt.

Die Typübersetzung erfolgt in der XSL-Transformation.

5.12. Inhalt der XML-Datei

Wie bereits erwähnt ist der Inhalt der XML-Datei an den Bezeichnern von CodeSurfer orientiert.

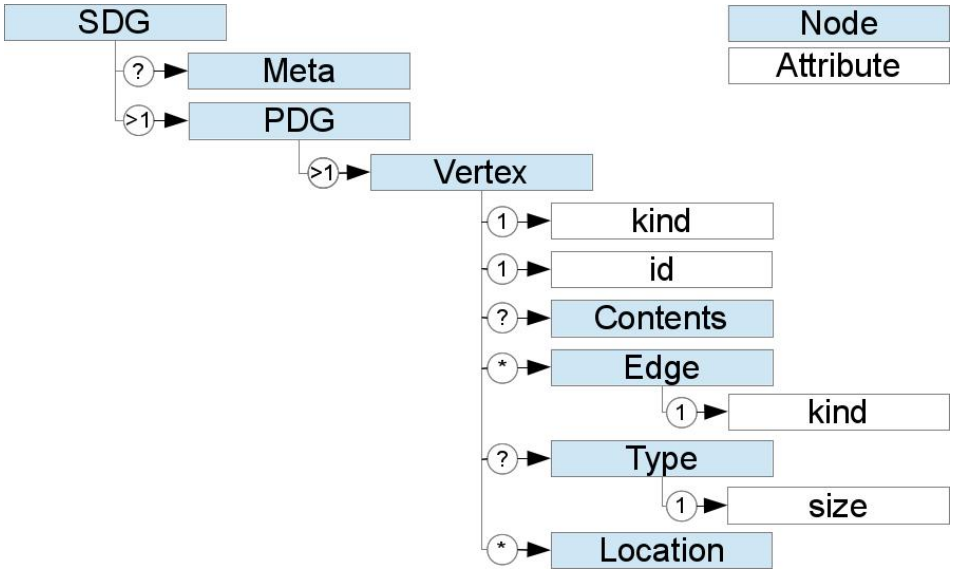

Abbildung 10: *Inhalt der XML-Datei (Auszug)*

Die Datei besteht also aus einem *SDG* bestehend aus mehreren *PDG*s, welche Typisiert und einer Funktion Zugeordnet sind.

Die Art der jeweiligen Elemente ist in dem Feld *kind* in der CodeSurfer Nomenklatur hinterlegt.

Das Tag *contents* enthält den jeweils zugehörigen Quelltext, jedoch ohne explizite Referenz auf die Originaldateien.

6. Analyse der Implementierung

Für die Analyse der Eigenschaften von Programmgraphen, welche mit dem im Umfang dieser Arbeit entstandenen Konverter übersetzt wurden, wurden die selben Programme herangezogen, die schon in der Diplomarbeit von B.Nürnberger [3] gewählt wurden. Im einzelnen sind dies:

agrep
: Eine approximierende Variante des bekannten Tools *grep*.
3 968 Codezeilen in **65** Funktionen

bison
: Ein Werkzeug zur Generierung eines Parsers.
8 557 Codezeilen in **132** Funktionen

ctags
: Dieses Tool extrahiert Bezeichner aus C-Quelltexten.
2 933 Codezeilen in **63** Funktionen

d3des
: Eine Implementierung des bekannten Verschlüsselungsalgorithmus DES, welcher von diesem Programm drei mal hintereinander ausgeführt wird.
832 Codezeilen in **19** Funktionen

ed
: Ein bekannter recht rudimentärer Texteditor für die Konsole.
3 052 Codezeilen in **120** Funktionen

moria
: Ein "Rogue-like" Spiel, welches mittels curses in der Konsole gespielt wird.
25 755 Codezeilen in **454** Funktionen

patch
: Ein Programm zur Anwendung mittels *diff* erstellter Differenzdateien auf Text.
7 998 Codezeilen in **106** Funktionen

plot2fig
: Ein Tool zur Konvertierung des Graphikformates *plot*.
1 538 Codezeilen in **27** Funktionen

sudoku
: Eine Implementierung des bekannten Rätsels für die Konsole.
714 Codezeilen in **10** Funktionen

tscp
: Ein Schachsimulator für die Konsole.
2 214 Codezeilen in **38** Funktionen

6.1. Laufzeit der Konvertierung

Die Laufzeit wird mittels des Unix-Kommandos *time* gemessen. Das den Messungen zugrundeliegende System findet sich im Anhang F.

Programm	Gesamtzeit	CodeSurfer	csdump	csjoana	valsoftize[a]	VALSOFT[b]
agrep	30.721s	7.802s	9.632s	13.287s	113s	28s
bison	43.829s	10.705s	20.188s	12.936s	46s	95s
ctags	16.845s	3.002s	6.969s	6.874s	23s	5s
d3des	4.973s	1.394s	0.942s	2.637s	3s	2s
ed	67.126s	7.390s	32.444s	27.292s	55s	285s
moria		77.463s	1504.112s	N/A[c]	4 162s	N/A[d]
patch	54.489s	15.279s	18.066s	21.144s	107s	158s
plot2fig	16.607s	9.270s	3.004s	4.333s	6s	1s
sudoku	6.475s	1.430s	1.917s	3.128s	5s	0s
tscp	15.107s	4.973s	4.687s	5.447s	18s	8s

Tabelle 3: *Laufzeit der Konvertierung*

Da keine aktuellen Daten für *valsoftize* und *VALSOFT* vorliegen sind die oben dargestellten Daten leider nur sehr begrenzt aussagekräftig.

6.2. Vergleich der Resultate

Im Folgenden soll das Resultat der jeweiligen Graphen verglichen werden, um eine Abschätzung der Qualität zu ermöglichen.

Zunächst sei die Anzahl der jeweils generierten Knoten und Kanten in Tabelle 4 gegenübergestellt. Die Daten für die Programme *valsoftize* und *VALSOFT* sind hierbei wieder aus [3] entnommen. Sie wurden mit dem Tool *sdgtool* ermittelt. Die Daten für *csjoana* wurden mit Hilfe des Skriptes *graphstats* ausgelesen.

Bei der Anzahl der erzeugten Kanten sind nur jene berücksichtigt, welche zum eigentlichen *PDG* gehören, die Typen *HE*, *CF*, *NF* und *JF* werden nicht gezählt.

Aus Tabelle 4 erkennt man, dass sowohl die Knoten- als auch die Kantenzahl bei *csjoana* geringer ausfällt. Dies liegt zum einen darin begründet, dass hier Berechnungen in die *ACTO-Knoten* verschoben wurde, anstatt wie in den beiden anderen Versionen einen eigenen Knoten erzeugen, zum anderen werden bei *csjoana* keine *ACTO-Knoten* erstellt, wenn das Ergebnis einer Funktion nicht beim Aufrufer verwendet wird. Dadurch fällt weiterhin die *SU-Kante* an dieser Stelle weg. Dies schlägt sich auch massiv in Tabelle 5 nieder.

In Tabelle 5 erkennt man weiterhin bei *csjoana* eine erheblich höhere Zahl an *DD-Kanten* auftritt.

[a] Da das *valsoftize* mit der neuen Version von CodeSurfer nicht mehr lauffähig ist, wurden die Werte von [3] übernommen. Leider sind dort keine Angaben über das Testsystem zu finden.
[b] Diese Daten wurden ebenfalls von [3] übernommen
[c] Auf dem Testsystem konnte Java nicht ausreichend Arbeitsspeicher zugewiesen werden (32-Bit-VM
[d] Konnte nicht mit VALSOFT analysiert werden

	csjoana		valsoftize		VALSOFT	
Programm	Knoten	Kanten	Knoten	Kanten	Knoten	Kanten
agrep	15 550	73 409	28 019	93 707	22 820	71 328
bison	28 073	122 624	35 946	106 245	33 158	141 232
ctags	12 972	56 921	15 216	51 264	12 958	51 924
d3des	1 468	4829	3 806	8 661	4 236	13 364
ed	32 647	284 948	38 925	316 555	35 863	574 725
patch	27 170	142 163	37 803	150 692	30 349	235 913
plot2fig	4 958	14 399	4 287	10 597	2 938	8 154
sudoku	2 538	6 142	2 295	4 910	1 463	3 191
tscp	8 876	46 892	12 870	51 611	11 704	68 817

Tabelle 4: *Anzahl der generierten Knoten und Kanten*

Es bleibt zu erwähnen, dass die Werte für Knoten- und Kantenzahlen - gerade bei kleinen Programmen - extrem vom System-Initialisierungscode beeinflusst werden. Dieser ist Abhängig von mehreren Faktoren: CodeSurfer kann unterschiedliche Compiler emulieren[9]; weiterhin nehmen die Bibliotheks-Stubs einen erheblichen Einfluss darauf.

	csjoana			valsoftize		VALSOFT	
Programm	DD	DH	SU	DD	SU	DD	SU
agrep	37 979	6 051	3 036	22 201	22 261	15 484	13 174
bison	47 363	13 689	6 696	26 333	17 124	50 878	32 688
ctags	22 478	5 481	4 471	8 169	17 138	8 085	20 620
d3des	1 564	443	364	1 482	461	3 223	2 750
ed	190 013	16 128	8 271	33 397	213 948	179 483	326 573
patch	67 354	12 253	9 361	29 367	53 871	87 132	92 619
plot2fig	3 320	1734	747	2 168	886	1 572	953
sudoku	1 613	879	128	833	111	606	43
tscp	24 771	3 895	1 732	8 433	19 823	13 648	33 226

Tabelle 5: *Anzahl der generierten DD- und SU-Kanten*

6.3. Graphabdeckung des Summary-Slicer-Backwards

Als weiteres Maß zur Bewertung der Resultate wird die Graphabdeckung des *Summary-Slicer-Backwards* herangezogen. Für die in Tabelle 6 wurde dieser Slice auf 1000 zufällig ausgewählten Knoten des Graphen berechnet. Als Schätzwert wird eine Graphabdeckung von 40% angestrebt.

Für *valsoftize* und *VALSOFT* wurden in Tabelle 6 nur ausgewählte Ergebnisse dargestellt, da die jeweiligen PDG-Dateien per Hand nachbearbeitet werden mussten.

[9]Hier wurde die Einstellung *gcc* verwendet

Programm	csdump	valsoftize	VALSOFT
agrep	49%		
bison	37%		
ctags	25%		
d3des	8%	12%	10%
ed	57%		
patch	57%		
plot2fig	6%	13%	23%
sudoku	5%	42%	75%
tscp	42%	66%	

Tabelle 6: *Graphabdeckung des Summary-Slicer-Backwards*

7. Fazit

Es ist festzustellen, dass Joana von Haus aus die Möglichkeiten für eine feingranularere Abbildung einer Analyse bietet. Die Möglichkeit in CodeSurfer die ASTs ebenfalls einzublenden kommt dem offenbar nicht gleich, da auf diesen Abhängigkeitskanten nicht in gewünschter Form vorliegen.

Weiterhin ist die Vorgehensweise von CodeSurfer in der Hinsicht unschön, dass es seine Daten binär hinterlegt, was einem keinen direkten Zugriff verschafft.

Jedoch ist meiner Meinung nach auch das Format von Joana alles andere als optimal, zumindest im Hinblick auf die Zugriffsmöglichkeiten; weiterhin ist es störend, dass es in diesem Format keine Kommentare gibt. Jedes Programm, welches Dateien von Joana benutzen will, muss diese selbst interpretieren (möglicherweise mit Hilfe des AntLR Parsers in Java).

Die Entscheidung für das bei Joana verwendete Dateiformat ist unter Berücksichtigung des Entstehungszeitpunkts des Projektes nachvollziehbar. Es ist allerdings nicht mehr zeitgemäß und weißt einige Spuren des Wachstums der Generatoren und einhergehend veraltete, nicht mehr verwendete Darstellungsmöglichkeiten auf.

8. Future Work

Der im Zuge dieser Arbeit entstandene Konverter könnte dahingehend erweitert werden, dass Heap-Data-Dependencies besser dargestellt werden. Dafür wäre ein tiefgehendere Analyse des CodeSurfer-SDG vonnöten. Auch könnte man Bytecodereferenzen besser emulieren.

Weiterhin bestünde ein weiteres Arbeitsfeld in der Überarbeitung des Dateiformats von Joana. So könnte man Konzepte der CodeSurfer-Darstellung in das Format übernehmen. Weiterhin würde sich eine Modularisierung des Joana-Formates anbieten, um es besser auf Sprachspezifische Eigenheiten anzupassen: Die feste Einbindung von Bytecode-Referenzen ist für C-Programme nicht passend. Im Gegenzug wäre es in einer modularisierten Form möglich weitere Debugginginformationen für C-Programme in das Format aufzunehmen.

Ein weiterer Vorteil der Modularisierung wäre die Möglichkeit der Aufnahme von Ergebnissen von Analysetools in das Format. Dadurch wäre es zum einen möglich das Analyseergebnis abzuspeichern, zum anderen würde es eine Kaskadierung von verschiedenen Analyseprogrammen ermöglichen.

Das aktuelle Joana-Format weist weiterhin mit der Angabe von HE-Kanten ein unschönes Vorgehen auf, da jeder Entwickler eine Vorberechnung durchzuführen gezwungen ist, welche nicht eigentlicher Gegenstand seiner Software ist.

Ein weiterer Mangel des Joana-Formats ist das Fehlen jeglicher Metainformationen (abgesehen vom Programmnamen) des gesamten SDG. So wären hier folgende Angaben sinnvoll:

- Dateireferenzen auf die Bestandteile des analysierten Programms.

- Sourcecodeversion des analysierten Programms um bei Aktualisierung des Codes eine Warnung ausgeben zu können.
- Programm und dessen Version mit welchem der SDG erstellt wurde um bei veralteten Versionen zu warnen.

Weiterhin weist das Format auch grundlegende Mängel auf: Das Fehlen von Escape-Zeichen, Kommentaren und eine Redundanz in der Bezeichnung von Knotentypen.

Im aktuellen Joana-Format ist es schwierig auf Datentypen außerhalb der generischen Typen der Sprache zu verweisen. Ein neueres Format könnte einen Bereich zur Definition benutzerspezifizierter Typen aufweisen.

Auch Sourcecodereferenzen sind in Joana nicht in vollem Detailgrad darstellbar: Zum einen ist die bloße Angabe des Dateinamens zur Referenzierung der Quelldatei nicht hinreichend, zum anderen ergeben sich Probleme dadurch, dass pro Knoten nur eine Referenz angegeben werden kann: So lassen sich eingebettete Kommentare und Referenzen auf Makros nicht darstellen.

A. Zuordnungstabellen

Die Tabelle 7 stellt die Zuordnung zwischen Joana-Knoten und CodeSurfer-Vertices dar. Sie dient somit weiterhin zur Auflösung der Γ-Abbildung aus Kapitel 4.2.

Joana-Knoten (Γ_J)	CodeSurfer (Γ_{CS})
NORM	declaration
EXPR	expression
ENTR	entry
EXIT	exit
EXIT	normal-exit
EXIT	exceptional-exit
NORM	return
NORM	normal-return
NORM	exceptional-return
CALL	call-site
ACTI	actual-in
ACTI	global-actual-in
ACTO	actual-out
ACTO	global-actual-out
FRMI	formal-in
FRMI	global-formal-in
FRMO	formal-out
FRMO	global-formal-out
	auxiliary
NORM	body
PRED / NORM	control-point
	indirect-call
NORM	label
	pi
EXPR	phi
	switch-case
	unavailable
	unknown
EXPR	variable-initialization
NORM	jump

Tabelle 7: *Zuordnung der Bezeichner von Knoten beider Formate*

Knoten	Erlaubte Operationen
NORM	declaration, IF, jump, compund
EXPR	intconst, shortcut, defer, refer, reference, floatconst, charconst, stringconst, functionconst, question, binary, unary, array, select, modify, modassign, assign
EXIT	exit
ENTR	entry
CALL	call
ACTI	act-in, intconst, shortcut, defer, refer, reference
ACTO	act-out
FRMI	form-in, form-ellip
FRMO	form-out
PRED	IF, intconst, chortcut, defer, refer, reference

Tabelle 8: *Erlaubte Operationen nach Joana-Knoten*

B. Source Dokumentation

Hier findet sich ein kurzer Abriss darüber, welche Funktionen implementiert worden sind und mit welchen Parametern diese aufgerufen werden.

B.1. Scheme-Code

Die Funktionen sind im Source-Code selbst in einer an Doxygen angelehnten Notation dokumentiert. Mit "ausgeben" ist im Folgenden gemeint, dass Daten in XML-Formatierter Form auf *STDOUT* geschrieben werden.

toString (Helper Funktion)
: Wandelt diverse Datentypen zu ihrer String-Darstellung.

xmlify (Helper Funktion)
: Escaped einen gegebenen String derart, dass er keine XML-Steuerungszeichen enthält.

translate-filename (Helper Funktion)
: CodeSurfer gibt lediglich absolute Pfadnamen aus. Diese Funktion übersetzt diese in relative.

expr-pos (AST-Traversal Funktion)
: Wird ein FainGrained-SDG (**ref**) erstellt, so muss die Source-Code Referenz entsprechend aufgespalten werden. Dies ist vor allem bei Expressions von Nöten. Diese Funktion gibt eine Position zurück, falls die Aufspaltung erfolgreich war - *false* andernfalls. Als Seiteneffekt ändert diese Funktion eine Variable, die den Anteil der verarbeiteten Eingabe enthält.

global_id (Helper Funktion)
: Gibt zu einem Vertex eine Projekt-global eindeutige numerische ID aus. IDs starten bei 1 und zählen lückenlos aufwärts. Dies ist nötig, da in CodeSurfer IDs nur innerhalb eines PDGs eindeutig sind.

equalVertex? (Helper Funktion)
: Gibt zurück, ob es sich bei den Vertices a und b um den selben handelt.

pdg-vertex-control-edge? (Helper Funktion)
: Gibt zurück, ob eine Kontroll-Abhängigkeit von Vertex a zu Vertex b existiert.

dumpMeta (Ausgabe Funktion)
: Schreibt Projektinformationen in die Ausgabe: Den Projekt-Namen, die Compilation-Units, den Erstellungszeitpunkt.

special-control-flow-push (Helper Funktion)
: Erstellt ein Special-Control-Flow Objekt und fügt es in die Special-Control-Flow-Liste hinzu (Parameter: source, destination, label, insteadOf).

special-control-flow-generate (Traversal Funktion)
Erstellt eine Special-control-Flow-Chain (falls nötig) für einen gegebenen Vertex.

first-traversal (Main Funktion)
Die ”main-Funktion” für den ersten Baumdurchlauf. Hier wird *special-control-flow-generate* angestoßen. Diese Funktion wird für jeden PDG angestoßen.

dump-declarations (Ausgabe Funktion)
Erzeugt die Ausgabe für einen Declaration-Vertex.

dump-special-control-flow (Ausgabe Funktion)
Erzeugt im Falle des Vorhandenseins eines speziellen Kontrollflussverlaufs die passende Ausgabe.

dump-location (Ausgabe Funktion)
Gibt die Source-Code Referenz aus.

dump-return-type-pretty-print (Ausgabe Funktion)
Helper-Funktion für *dump-return-type* zur Formatierung des Typs.

dump-return-type (Ausgabe Funktion)
Gibt den Return-Typ eines Vertex aus.

dump-edges (Ausgabe Funktion)
Gibt alle mit einem Vertex assoziierten Kontroll- und Daten- Kanten aus.

dump-control-flow (Ausgabe Funktion)
Gibt alle Kontrollflusskanten aus.

dump-vertices (Ausgabe Funktion)
Gibt alle Vertices eines PDGs aus. Das ist die ”main-Funktion” für den zweiten Durchlauf.

dump-ast (Ausgabe Funktion)
Gibt sowohl den AST als auch den μAST für ein Vertex aus. (Normalerweise nicht verwendet, da die Ausgabedatei zu groß wird).

dispatch-expression (Ausgabe Funktion)
Gibt sowohl den AST als auch den μAST für ein Vertex aus.

acti-chain-head (Helper Funktion)
Gibt zu einem gegebenen acti-Vertex den Anfang der Chain aus, in dem dieser sich befindet.

Fehler und Log-Einträge werden in Form eines XML-Kommentars ebenfalls auf STDOUT geschrieben.

B.2. XSLT-Code

Bezüglich dieses Codes sind folgende Konzepte zu unterscheiden:

- Templates, welche auf Eingabezeichen hin ausgeführt werden.
- Benannte Templates, welche explizit aufgerufen werden müssen.
- Funktionen, welche selbst keine Ausgabe erzeugen, jedoch einen Rückgabewert haben und ebenfalls explizit aufgerufen werden müssen.

Den Ausgangspunkt bildet die Dateien *nodes.xslt*, von ihr aus werden die Dateien *nd_ *.xslt* aufgerufen.

Named Template Joana Stellt die einzelnen Vertices im Joana SDG-Format zusammen

Template Match: vertex Ruft das Joana-Template für jedes Vertex auf

Template Match: control-Edge Ruft das Named Template *rewrite-edges-control* auf

Template Match: data-Edge Ruft das Named Template *rewrite-edges-data* auf

Template Match: cf-Edge Ruft das Named Template *rewrite-edges-control-flow* auf

Knoten werden behandelt von:

Function rewrite-kind Mapt Vertice-Typen von CodeSurfer nach Joana

Function rewrite-operation Findet die passende Joana-Operation heraus

Function return-type Extrahiert den Typ-Parameter des Knotens

Die Kontrollabhängigkeiten werden behandelt von:

Funktion fetch-transitive-label Schreibt Labels von CE-Kanten auf CD-Kanten um

Funktion rewrite-edges-control Übersetzt isomorphe Kontrollabhängigkeiten und verwirft diese, welche in Joana nicht existieren explizit

Named Template generate-edges-control erstellt Kontrollabhängigkeiten, die in dieser Form in CodeSurfer nicht vorhanden sind

Die Datenabhängigkeiten werden behandelt von:

Funktion rewrite-edges-data Übersetzt isomorphe Datenabhängigkeiten und verwirft diese, welche in Joana nicht existieren explizit

Named Template generate-edges-data Erstellt Datenabhängigkeiten, die in dieser Form in CodeSurfer nicht vorhanden sind

Der Kontrollfluss wird behandelt von:

Named Template rewrite-edges-control-flow Übersetzt den Kontrollfluss. Beachtet dabei den Wert *type="special"* und *insteadOf=ID*

Named Template append-edges-control-flow Dient zum Kontrollflussrerouting

Byte Code Interface:

Named Template fake-bci Erstellt eine Fake-Bytecodereferenz

Funktion fake-component Erstellt den Inhalt des Joana-C-Attributes

Funktion fake-class Wrapt Klassennamen *app* und *sysinit* ...

Helper Funktionen:

Named Template location Fasst mehrere Locations zu einer zusammen und generiert die passende Zeile

Named Template procnum Übersetzt einen Prozedurnamen in die zugehörige numerische ID

Function strip-quotes Entfernt Kontrollzeichen den Joana SDG aus Strings

B.3. Bash Postprocessor

XSLT gibt teilweise am Anfang und Ende der Zeilen Whitespace aus, außerdem sind die Vertices zunächst noch nicht nach ID sortiert.

B.4. Java-Code

Der Iterator zur Graphtraversierung wurde derform angepasst, dass man sich vor der Iteration für bestimmte Vorkommnisse im Graphen registrieren kann und dann per *Call-Back-Funktion* informiert wird[10]. Dadurch ist es möglich einen einzelnen Baumdurchlauf modular zu erweitern. Hierzu wurden die folgenden Klassen erstellt:

Filter Dient zur Angabe, bei Auftritt welcher Knoten oder Kanten man informiert werden will.

FilteredObserver Basisklasse für den Observer

CsJoanaWalk Das Observable, welches den Graphen durchläuft

Hier sind nur die neu erstellten Klassen aufgeführt

FilteredSDG(SDG, SDGNode.Kind) extends Iterator Iterator über alle Knoten eines bestimmten Typs

csJoanaPostprocessor Die Main-Klasse des Postprocessors:

[10] Observer Design-Pattern

- Liest einen Joana-SDG ein
- Schreibt die Beschriftung von CD-Kanten transitiv entlang des CF-Pfades weiter
- Berechnet die HE-Kanten (Zyklenfrei)
- Schreibt den geänderten SDG wieder aus

C. Fehlentscheidungen bei der Implementierung und verworfene Konzepte

In dieser Stelle sollen auch kurz die Fehlentscheidungen nicht unerwähnt bleiben, auf dass diese Liste bei späteren Arbeiten eventuell eine Stütze bei der Vermeidung eben dieser sei und Zeit spare.

Die erste Implementierung der Erkennung der **Zyklenfreiheit** zur Erstellung der **HE-Kanten** sah derart aus, dass die CodeSurfer eigene Funktion zur Feststellung der Zyklenbehaftetheit verwendet wurde. Daraufhin wurden sukzessive weiter Knoten zu einem PDG hinzugefügt und jeweils diese Funktion angewandt. Die daraus resultierende Geschwindigkeit war denkbar untragbar.

Das **Durchlaufen** der **Kontrollflusspfade** mittels **XSLT** ist zum einen sehr umständlich, zum anderen aus in 5.1 dargestellten Gründen viel zu langsam.

Die Funktion zum Auffinden des gemeinsamen Vorgängers auf Pfaden mittels einer Breitensuche war ungeschickt gewählt: Leider fiel mir erst beim Schreiben des Textes auf, dass es sich dabei um den *Immediate-Dominator* handelt, und dafür bereits ein schneller Algorithmus von Lengauer und Tarjan existiert.

Es bestand die Idee eine XSL-Transformation zur Erstellung von **graphviz**-Dotfiles zu erstellen. Die auftretenden Graphen sind allerdings zu groß und komplex um sinnvolle Ergebnisse zu erzielen.

D. Abhängigkeiten der Software

Zur Ausführung der im Rahmen dieser Studienarbeit erstellten Software sollten auf einem System die folgenden Programme installiert sein:

CodeSurfer (mit Scripting) Für den Befehl *csdump* wird CodeSurfer mit Scripting Unterstützung benötigt. Erhältlich ist CodeSurfer auf:
http://www.grammatech.com

XSLT-2.0-Processor Als XSLT-Processor kommt *Saxon-9-HE* zum Einsatz. *Saxon* basiert auf einer Java-Implementation, die Home-Edition ist kostenfrei erhältlich unter:
http://www.saxonica.com

Shell Als Interpreter der Shell-Scripte ist */bin/sh* angegeben. Zur Verfügung gestellt wird dieser Befehl beispielsweise durch die Bourne again shell, die man beziehen kann von:
http://tiswww.case.edu/php/chet/bash/bashtop.html

GNU Core-Utils Aus den Core-Utils, beziehbar von:
http://www.gnu.org/software/coreutils/
werden die Programme *md5sum*, *sort* und *cat* benötigt.

grep Für das Post-Processing wird *egrep* benötigt. Erhältlich ist dies unter:
http://www.gnu.org/software/grep/

sed Der Streameditor wird ebenfalls im Post-Processing benötigt. Man bekommt ihn unter:
http://sed.sourceforge.net/

Java VM Getestet wurden hier die JVMs *IcedTea 6* und die Version von Sun (Oracle). Erhältlich sind sie jeweils unter:
http://icedtea.classpath.org
http://www.oracle.com/technetwork/java/javase/

Perl-Interpreter Der Perl-Interpreter wird lediglich für das Statistikscript benötigt. Er ist für Linux und Mac-OS erhältlich unter:
http://www.perl.org/
Für Windows existiert eine Version von:
http://www.activestate.com/

Der XSLT-Prozessor von TrollTech (Nokia) repräsentiert u.a. durch das Programm *xmlpatterns* ist nicht geeignet, da er benötigte Teile der XSLT-2.0 Spezifikation nicht unterstützt.

E. Benutzerschnittstelle

Zur einfacheren Verwendung der oben dargestellten Konzepte wurden einige Bash-Scripte geschrieben. Diese rufen die einzelnen Teilprogramme auf und warnen bei Gefahr auf inkonsistente Daten, wenn z.B. das Programm geändert wurde allerdings noch keine erneute Analyse ausgeführt wurde.

Diese Scripte suchen die benötigten Dateien an einschlägigen Stellen des Dateisystems, sie lassen sich daher z.B. nach */usr/bin/* kopieren.

E.1. Extraktion as CodeSurfer

Das zugehörige Wrapper-Script hierzu heißt *csdump*. Es liest das CodeSurfer-Projekt ein und generiert eine XML-Datei. Weiterhin existieren Optionen wie folgt:

-h Zeigt die verfügbaren Optionen

-v Verbose Ausgabe

-b Fügt Strukturen zur Anzeige im Webbrowser hinzu (siehe E.3)

-o DIR Ändert das Ausgabeverzeichnis (Standard: *./csdout/*)

-a Fügt den *normalized-* und μ*-AST* zur XML-Datei hinzu

-d lvl Ändert den Debug-level und damit die Anzahl der Kommentare in der XML-Datei

-p Befehle zum Erstellen des CodeSurfer-Projektes ausführen

Im Ausgabeverzeichnis wird neben der XML-Datei weiterhin die Datei *versions.md5* aktualisiert, welche zur Konsistenzprüfung dient.

E.2. Konvertierung zu Joana

Hierzu dient *csjoana*, welches die zuvor erstellte XML-Datei einliest.

-h Zeigt die verfügbaren Optionen

-v Verbose Ausgabe

-i DIR Ändert das Arbeitsverzeichnis (Standard: *./csdout/*)

Das Script führt die XSLT-Transformation durch und führt alle nötigen Post-Processing Schritte aus.

E.3. Darstellung im Web-Browser

Wenn man *csjoana* mit der Option *-b* ausgeführt hat, so werden alle Dateien, die zur Darstellung der XML-Datei im Browser benötigt werden, angelegt.

Die Web-Browser Darstellung ist eher experimenteller Natur.

Da die Formatierung eines SDG im Browser recht rechenaufwändig ist, muss man dem Browser eventuell gestatten Scripte mit langer Laufzeit auszuführen.

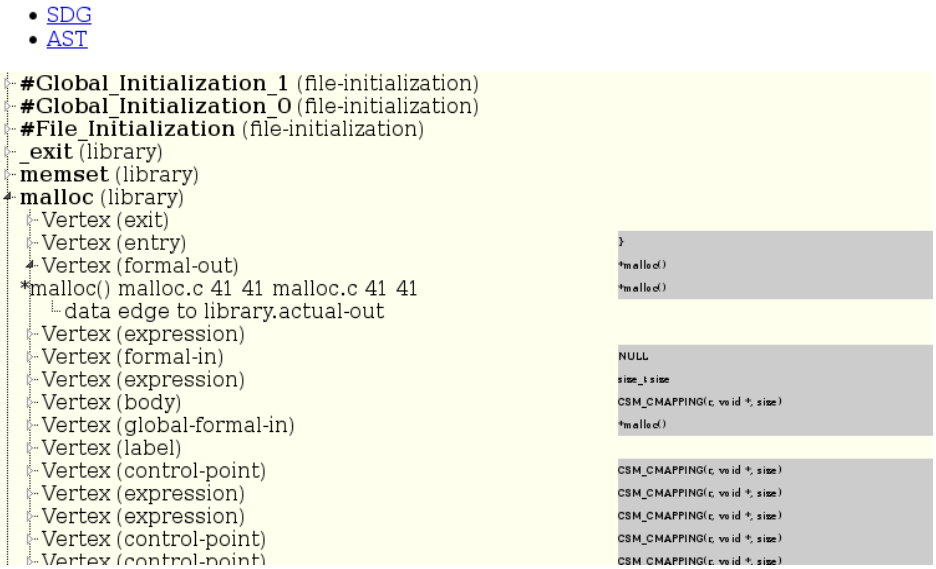

Abbildung 11: *SDG im Web-Browser*

F. Test System

Die in Kapitel 6.1 vorgenommenen Messungen fanden auf folgendem System statt:

- CPU: Intel(R) Core(TM) i3 CPU 550
  - Taktung: 3.2GHz
  - Cache: 4MB
  - Stepping: 5
- OS: GNU/Linux, Ubuntu
  - Kernel: 2.6.32-38-generic
  - Version: #83-Ubuntu
  - Target: i686, SMP
- RAM: 1878 MB
- JRE-Version: IcedTea6 1.9.13, 6b20-1.9.13-0ubuntu1 10.04.1
- XSLT-Parser: Saxon-9 HE

Abbildungsverzeichnis

Tabellenverzeichnis

Beispielverzeichnis

Literatur

[1] Din66001. Din, DIN, Sept. 1966.

[2] Anderson, P., and Zarins, M. The codesurfer software understanding platform. In *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on* (may 2005), pp. 147 – 148.

[3] B., N. Slicing und pfadbedingungen mit codesurfer. Diplomarbeit, Universität Passau, 2008.

[4] Berglund A., Boag S., C. D. e. a. Xml path language (xpath) 2.0 (second edition). W3c recommendation, W3C, Dec. 2010.

[5] Bray T., Paoli J., S.-M. C. e. a. Extensible markup language (xml) 1.0 (fifth edition). W3c recommendation, W3C, Nov. 2008.

[6] Ferrante, J., Ottenstein, K. J., and Warren, J. D. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst. 9*, 3 (July 1987), 319–349.

[7] Horwitz, S., Reps, T., and Binkley, D. Interprocedural slicing using dependence graphs. *SIGPLAN Not. 39*, 4 (Apr. 2004), 229–243.

[8] Krinke, J. Advanced slicing of sequential and concurrent programs. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on* (sept. 2004), pp. 464 – 468.

[9] Lengauer, T., and Tarjan, R. E. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst. 1*, 1 (Jan. 1979), 121–141.

[10] Michael, K. Xsl transformations (xslt) version 2.0. W3c recommendation, W3C, Jan. 2007.

[11] Mustermann, M. Graphviewer. Studienarbeit, Universität Karlsruhe (TH), 2008.

[12] Ottenstein, K. J., and Ottenstein, L. M. The program dependence graph in a software development environment. *SIGPLAN Not. 19*, 5 (Apr. 1984), 177–184.

[13] Podgurski, A., and Clarke, L. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *Software Engineering, IEEE Transactions on 16*, 9 (sep 1990), 965 –979.

[14] Weiser, M. Program slicing. In *Proceedings of the 5th international conference on Software engineering* (Piscataway, NJ, USA, 1981), ICSE '81, IEEE Press, pp. 439–449.