Präziser Kontrollfluss für Exceptions durch interprozedurale Datenflussanalyse

Markus Herhoffer

11. Juni 2012

Zusammenfassung

Die vorliegende Studienarbeit erweitert eine vorhandene Analyse, die nicht auftretende Nullpointer-Exceptions innerhalb einer einzelnen Methode erkennt (intraprozedurale Analyse), auf den interprozeduralen Fall und ergänzt diese mit einem Testframework und einer Klasse zur anschaulichen Darstellung der Kontrollflussgraphen als Plot. Die interprozedurale Erweiterung erkennt doppelt so viele nicht auftretende Nullpointer-Exceptions wie die intraprozedurale Analyse. Insbesondere können mehr als 50% aller potentiell auftretenden Nullpointer-Exceptions ausgeschlossen werden.

1 Motivation

Ein Kontrollflussgraph ist ein Graph, der den Programmfluss innerhalb einer Methode abbildet. Kontrollflussgraphen sind die Basis vieler Analysen und Optimierungen in jSDG/Joana (s. [Gra10] und [HS09]), wie z. Bsp. IFC mit PDGs. Die Mehrzahl dieser Optimierungen ist abhängig von einem möglichst präzisen Kontrollfluss und hat exponentielle oder größere Laufzeiteigenschaften $O(E)$, wobei

$$E = \textit{Anzahl der Kanten im Kontrollflussgraphen}$$

Exceptions sind in Java ein Mechanismus zum Behandeln von Ausnahmefällen. Jede Exception unterbricht das laufende Programm und erzeugt neuen Kontrollfluss im Kontrollflussgraphen.

Am statistisch häufigsten tritt die spezielle Nullpointer-Exception auf. Darum ist sie auch das Subjekt der vorliegenden Analyse.

Nullpointer-Exceptions können in Java theoretisch an jeder Stelle auftreten, an der auf eine Referenz-Variable zugegriffen wird. Wird aus dem Java-Bytecode also ein Kontrollflussgraph zur Analyse erzeugt, beinhaltet dieser bei jedem Zugriff auf Referenz-Variablen eine Behandlung für eine potentiell auftretende Nullpointer-Exception.

Um die vorhandenen und zukünftigen Analysen in jSDG/Joana schneller und präziser zu machen, ist es also von Interesse, die Anzahl der Kanten in den Kontrollflussgraphen zu minimieren. In einem vorgelagerten Prozess kann dies erreicht werden, indem sicher nicht auftretende Exceptions erkannt und aus dem Kontrollflussgraphen gelöscht werden.

Zudem soll die erarbeitete Analyse in das WALA-Framework integriert werden.

2 Grundlagen und theoretische Überlegungen

2.1 Definition Kontrollflussgraphen

Eine Datenflussanalyse wie die vorliegende arbeitet auf verschiedenen Datenstrukturen, die Aspekte eines Programms repräsentieren (nach [Muc97]).

Grundlegend ist dabei die Instruktion, die im Nachfolgenden formal mit einem Block gleichgesetzt wird. Die explizite Einführung von Grundblöcken[1] ist für die Analyse nicht nötig, ein Block wird also im Folgenden mit einer Anweisung gleichgesetzt.

Um den Kontrollfluss zwischen den Blöcken zu modellieren, werden Kontrollflussgraphen (*Control Flow Graph*, CFG) definiert. Ein Kontrollflussgraph besteht aus einer Menge von Anweisungen als Knoten sowie den beiden speziellen Knoten `ENTRY` und `EXIT`. Eine Kante zwischen zwei Anweisungen `B1` und `B2` verläuft dann, wenn bei der Ausführung des Programms `B2` direkt nach `B1` ausgeführt werden kann. [Mor98]. Die Kante ist dann von `B1` nach `B2` gerichtet.

Ein Kontrollflussgraph existiert für jede Prozedur eines Programms und wird aus dem in einem vorgelagerten Prozess aufgestellten Flussdiagramm erzeugt [Muc97].

Formal ergibt sich also folgender Zusammenhang [Muc97]: Ein Kontrollflussgraph ist ein Graph $G = \langle N, E \rangle$ mit einer Knotenmenge N und einer Kantenmenge $E \subseteq N \times N$. Es gilt zudem $\texttt{ENTRY} \in N$ und $\texttt{EXIT} \in N$. Der Einfachheit wegen werden Kanten im Folgenden mit $\texttt{B1} \rightarrow \texttt{B2}$ dargestellt.

Seien die Nachfolger und Vorgänger einer Anweisung `b` wie folgt definiert [Muc97]

$$\mathit{Nachfolger}(\texttt{b}) = \{\texttt{n} \in N \mid \exists e \in E : e = \texttt{b} \rightarrow \texttt{n}\}$$

$$\mathit{Vorgaenger}(\texttt{b}) = \{\texttt{n} \in N \mid \exists e \in E : e = \texttt{n} \rightarrow \texttt{b}\}$$

$\mathit{Vorgaenger}^*$ bzw. $\mathit{Nachfolger}^*$ ist jeweils die transitive Hülle.

Es gilt:

[1] eine maximale Abfolge von Instruktionen, sodass die Flusssteuerung immer durch die erste Instruktion in den Grundblock gelangt und ihn durch die letzte verlässt (s. [ALSU06, S. 642])

1. `ENTRY` ist transitiver Vorgänger von allen Knoten. Es gilt für alle `b` $\in (N \setminus$ `ENTRY`$)$:
$$\texttt{ENTRY} \in Vorgaenger^*(\texttt{b})$$

2. `EXIT` ist transitiver Nachfolger von allen Knoten. Es gilt für alle `b` $\in (N \setminus$ `EXIT`$)$:
$$\texttt{EXIT} \in Nachfolger^*(\texttt{b})$$

3. Aus diesen beiden Eigenschaften ergibt sich zudem
$$Vorgaenger^*(\texttt{ENTRY}) = \emptyset$$
$$Nachfolger^*(\texttt{EXIT}) = \emptyset$$

D.h. alle Anweisungen im CFG können von `ENTRY` aus erreicht werden und haben einen Pfad zu `EXIT`.

2.1.1 Beispiel

Quelltext 1 Beispielcode einer einfachen Prozedur `funct`

```
1 static void funct(){
2   int a = 3;
3   if (a<2) {
4     b = foo(a);
5   } else {
6     a--;
7   }
8 }
```

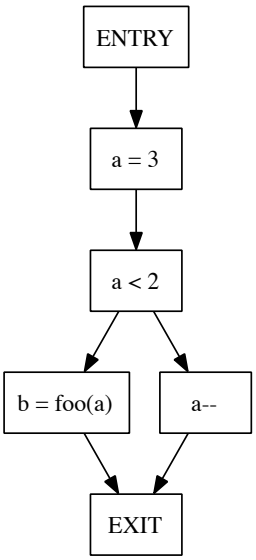
In Quelltext 1 ist beispielhaft eine Prozedur in Java gelistet. Abb. 1 zeigt den zugehörigen Kontrollflussgraphen.

2.2 Callgraphen und Interprozedurale Kontrollflussgraphen

Callgraphen zeigen wie sich Prozeduren in einem Programm gegenseitig aufrufen. Sie bilden den Kontrollfluss zwischen Prozeduren ab und schaffen somit eine interprozedurale Verbindung der einzelnen CFG. Ein Callgraph repräsentiert also ein Programm oder einen Programmausschnitt, ein Kontrollflussgraph nur eine einzige Prozedur.

Callgraphen bestehen aus einer Menge von Prozeduren als Knoten. Eine Kante zwischen den Prozeduren p_1 und p_2 besteht genau dann, wenn p_2 innerhalb von p_1 aufgerufen wird. Ein solcher Aufruf wird als *callsite* bezeichnet.

Abbildung 1: Kontrollflussgraph zu dem Code aus Quelltext 1



[Muc97] definiert den Callgraphen G_P eines Programmes P mit den Prozeduren $p_1, \ldots, p_n$ als

$$G_P = \langle N, S, E, r \rangle$$

Die Knotenmenge N ist dabei die Menge der Prozeduren $N = \{p_1, \ldots, p_2\}$, S ist eine Menge von Callsite-Labels und die Kantenmenge E ist eine Menge von ausgezeichneten Kanten $E \subseteq N \times N \times S$. r ist der Startknoten[2] von P. Für jede Kante $e \in E$ gilt $e = \langle p_i, s_k, p_j \rangle$. s_k benennt einen Aufruf in p_i, von der aus p_j aufgerufen wird. Das Label s_k ist nötig, da mehrere Aufrufe in p_i auf p_j zeigen können.

Diese Datenstruktur erweist sich jedoch beim Vorhaben, eine interprozedurale Analyse zur Optimierung von Exceptions zu entwickeln, als nicht besonders hilfreich. Die Trennung von Callgraphen und Kontrollflussgraphen ist ungünstig, da insbesondere die genaue Anweisung im Kontrollflussgraphen, von der aus eine Prozedur aufgerufen wird (callsite), für die Exception-Analyse benötigt wird. Die Kanten im Callgraphen zeigen nur von Prozedur zu Prozedur und erschweren somit unnötig die Identifikation der Aufrufstellen der Exceptions. Deshalb verwendet die interprozedurale Analyse eine andere Datenstruktur, die besser geeignet ist: Der interprozedurale CFG.

Die Interprozeduralen Kontrollflussgraphen ICFG sind wie folgt defi-

[2] entspricht der `main`-Methode in den meisten Java-Programmen

niert, etwa bei [Gif11]:

$$G_{ICFG} = \langle G_q, \texttt{main}, Call \rangle$$

wobei G_q die Menge der Kontrollflussgraphen (Abs. 2.1) der Prozeduren im Programm q ist sowie `main` die ausgezeichnete Startprozedur von q ist. $Call$ ist eine Menge aus Paaren von `call`- und `return`-Kanten, sodass gilt:

- Für jedes Paar von Prozeduren p_i und p_j aus dem Programm q sind die Knotenmengen N_{p_i} und N_{p_j} sowie die Kantenmengen E_{p_i} und E_{p_j} disjunkt.
- Sei c_p ein Knoten im CFG G_p der Prozedur p, der die Prozedur p' aufruft. Dann existiert ein Kantenpaar

  $$(b_p \rightarrow_{call} \texttt{ENTRY}_{p'}, \texttt{EXIT}_{p'} \rightarrow_{ret} b_p)$$

  wobei $c_p \rightarrow_{call} \texttt{ENTRY}_{p'}$ eine Aufruf-Kante von einem Block im CGF G_p zum Wurzelknoten `ENTRY` des CFG $G_{p'}$ ist und $\texttt{EXIT}_{p'} \rightarrow_{ret} b'_p$ eine Return-Kante vom `EXIT`-Knoten des CFG $G_{p'}$ zu b im CFG G_p ist.[3]
- Für beide CFG G_p und $G_{p'}$ gelten die in Abschnitt 2.1 vorgestellten Eigenschaften.

2.2.1 Beispiel

Zur Methode aus Quelltext 1 wird die Methode `foo()` aus Quelltext 2 hinzugefügt.

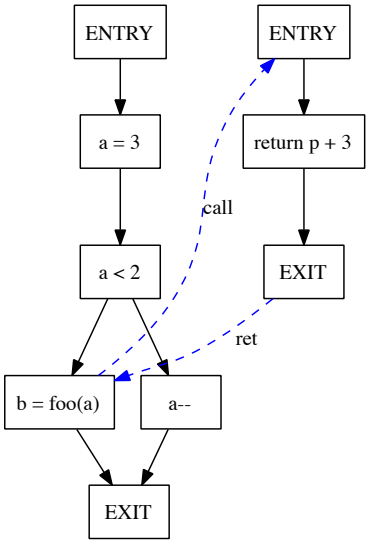
Quelltext 2 Beispielcode einer einfachen Prozedur `foo(int p)`

```
1 static void foo(int p){
2         return p + 3;
3 }
```

Der aus beiden Prozeduren resultierende ICFG ist in Abb. 2 dargestellt. Links ist der CFG G_{funct}, rechts der CFG G_{foo}. Die blauen, gestrichelten Kanten sind die in Absatz 2.2 eingeführten interprozeduralen Kanten $(\texttt{b} = \texttt{foo(a)}_{funct} \rightarrow_{call} \texttt{ENTRY}_{foo}, \texttt{EXIT}_{foo} \rightarrow_{ret} \texttt{b} = \texttt{foo(a)}_{funct}) \in Call$.

[3] Hier unterscheidet sich die Definition von [Gif11], der die Return-Kante auf *Nachfolger*(b) richtet. In der vorliegenden Analyse erweist es sich aber als pragmatischer, die Return-Kante direkt zum Aufruf zu richten, da dies auch der Modellierung in WALA entspricht.

Abbildung 2: Interprozeduraler Kontrollflussgraph zu dem Code aus Quelltext 1 und Quelltext 2



2.3 Exceptions in Java

Java verwendet das Konzept der Ausnahmen (Exceptions) zur Behandlung von Fehlern, die nur zur Laufzeit auftreten. Welche Fehler welche Exception auslösen ist in der Java-Spezifikation [GJSB05] definiert.

Wenn eine Ausnahme auftritt, wird die normale Programmausführung abgebrochen und die Ausnahmebehandlung (Exceptionhandler) aufgerufen. Dabei entsteht ein neuer Kontrollfluss von der Stelle im Programm, an der die Exception ausgelöst wurde, hin zu jener Stelle, die die Exception behandelt.

2.4 Nullpointer-Exceptions in Java

Eine besondere Exception ist die `NullPointerException` (kurz: NPE). Sie tritt unter allen Exceptions am statistisch häufigsten auf (siehe Anhang A). Sie kann potenziell von ca. 40% aller Anweisungen in einem Programm geworfen werden. Darum ist das Analysieren von NPE auch Ziel der vorliegenden Analyse.

Ihre Einbettung in die Klassenhierarchie der Java Programming Language ist (nach [Gra97]):

`Object` > `Throwable` > `Exception` > `RuntimeException` > `NullPointerException`

Eine NullPointerException wird dann geworfen, wenn das Programm `null` an Stellen verwendet, an denen ein Objekt notwendig ist. Dies tritt nach [GJSB05, Abs. 15.6] genau dann auf wenn:

- ein Feldzugriff erfolgt, dessen Wert im Objektreferenz-Ausdruck `null` ist
- ein Methodenaufruf auf eine Instanz-Methode erfolgt, wenn die Zielreferenz `null` ist
- ein Arrayzugriff erfolgt, wenn der Wert des Ausdrucks der Array-Referenz `null` ist

2.5 Exceptions in Kontrollflussgraphen

Java selbst strebt nicht an, die Stellen, welche `NullPointerExceptions` werfen können, einzuschränken. [GJSB05]:

> It is beyond the scope of the Java programming language, and perhaps beyond the state of the art, to include sufficient information in the program to reduce to a manageable number the places where these can be proven not to occur.

Die in dieser Arbeit entwickelte Analyse soll nun möglichst viele jener Stellen finden, an denen im Programm keine `NullPointerException` auftreten kann. Die Datenstruktur, welche dieser Analyse zugrunde liegt, sind die in Abschnitt 2.2 vorgestellten Interprozeduralen Kontrollflussgraphen.

Die Stelle, an denen eine Exception (insbesondere eine `NullPointerExceptions`) auftreten kann, ist immer eine Anweisung. Da eine Exception den normalen Kontrollfluss unterbricht, generiert sie also eine Kante von der Anweisung, in der sie geworfen wird, hin zum Ende der Prozedur oder zum Beginn des Exception-Handler. Betrachtet man also einen CFG wie unter Abs. 2.1 definiert ($G = \langle N, E \rangle$), so gibt es die Kantenmenge $E_{Exc} \subset E$, welche den Kontrollfluss aufgrund einer geworfenen Exception definiert. Eine Kante in E_{Exc} wird mit $\rightarrow_{Exc}$ dargestellt. E_{Exc} ist eine echte Teilmenge von E, da die Kanten von `ENTRY` ausgehend nie einen Exceptionkontrollfluss darstellen können und es immer einen Kontrollfluss hin zu einer Anweisung geben muss, der eine Exception wirft.

Formal definieren wir analog die Menge $N_{Exc} \subset N$ mit $\texttt{b} \in N_{Exc}$ gdw. `b` eine Exception wirft.

Es gilt:

- (intraprozedural) Wenn eine Anweisung `b` eine Exception wirft, so gibt es einen Exceptionkontrollfluss zum Knoten `h` des CFG: $\texttt{b} \in N_{Exc} \Rightarrow \texttt{b} \rightarrow_{Exc} \texttt{h}$, wobei $\texttt{h} \in \{\texttt{EXIT}, \textit{Handler}\}$

- (interprozedural) Wenn im CFG der Prozedur `p` eine Exception geworfen wird, dann existiert eine Exception-Kontrollflusskante $\rightarrow_{Exc}$ von der Anweisung `inv` in der Prozedur `q` hin zu `h` von `q`: $\texttt{inv}_q \rightarrow_{call} \texttt{ENTRY}_q \wedge N_{Exc,p} \neq \emptyset \Rightarrow \texttt{inv}_q \rightarrow_{exc} \texttt{h}_q$, wobei wieder gilt $\texttt{h} \in \{\texttt{EXIT}_q, Handler_q\}$

Handler ist die Menge aller Blöcke, welche Exceptionhandler beinhalten.[4]

2.5.1 Beispiel

Quelltext 3 Beispielcode für einen Prozeduraufruf mit Nullpointer-Exception-Behandlung

```
1  public class D {
2    int i;
3    public void invokesMethod(){
4      D d = new D();
5      needsAParameter(d);
6      return;
7    }
8
9    private void needsAParameter(D d) {
10     int j = d.i;
11     j++;
12     this.i = 3;
13     return;
14   }
15 }
```

Der Code in Alg. 3 zeigt zwei Prozeduren. Die Prozedur `invokesMethod()` ruft die Prozedur `needsAParameter(D d)` in Zeile 5 auf. In Zeile 10 erfolgt ein Zugriff auf ein Attribut, der eine NPE werfen kann (s. Abs. 2.4). Daher ist Zeile 10 $\in N_{exc}$.

Im zugehörigen ICFG sind Anweisungen, die eine Exception werfen können, rot markiert. Die rot markierte Anweisung `this.i = 3` gehört zu Codezeile 10. Wie in Abs. 2.5 beschrieben, gibt es von diesem Block ausgehend eine Exceptionkante $\in E_{exc}$, welche ebenfalls rot markiert ist.

In dem Aufrufkontext des Codebeispiels Alg. 3 wurde der Parameter `d` von `needsAParameter(D d)` in Zeile 4 bereits initialisiert. Die Variable `d` ist also ganz sicher nicht `null`. Sonst kann die Nullpointer-Exception ausgehend

[4] Welcher Handler genau für das Fangen der Exception zuständig ist, ist für die vorliegende Analyse nicht wichtig; allein die Existenz eines zusätzlichen Kontrollflusses ist relevant.

Abbildung 3: Nicht optimierter Kontrollflussgraph zu Quelltext 3

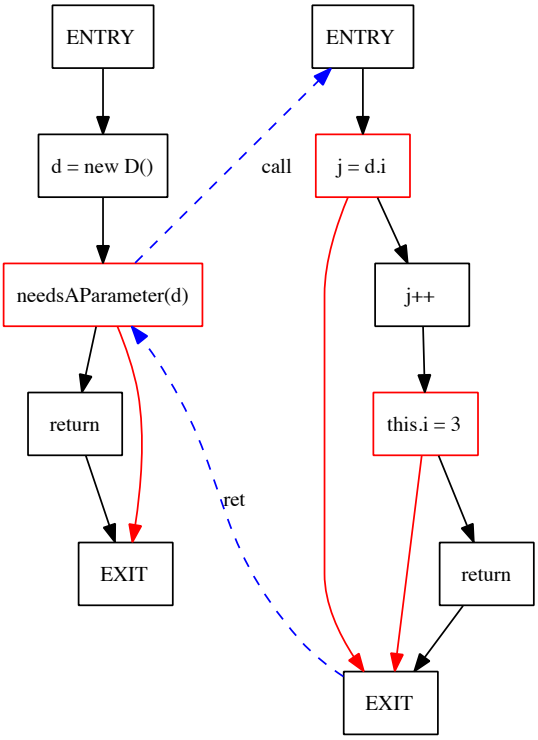


Abbildung 4: Mit intraprozeduraler Analyse optimierter Kontrollflussgraph zu Quelltext 3

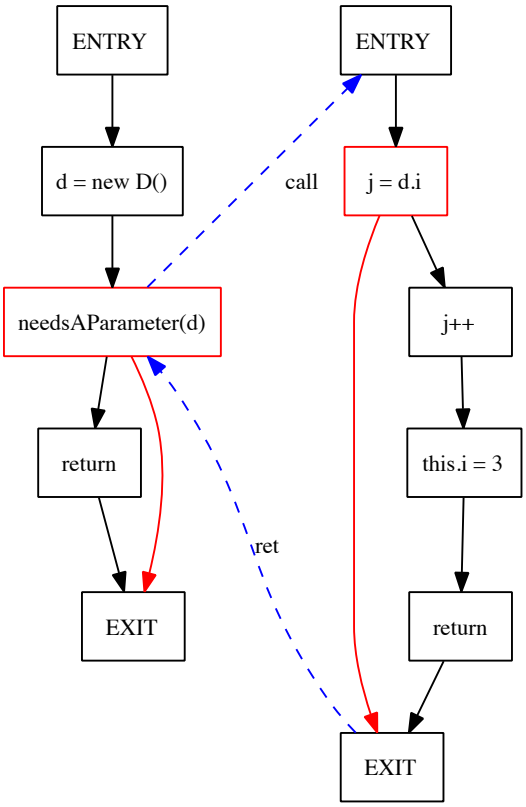
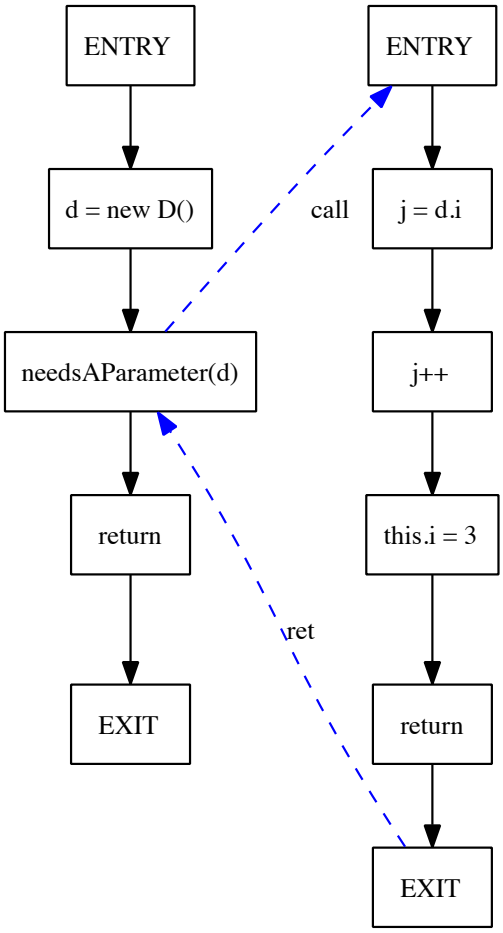


Abbildung 5: Mit interprozeduraler Analyse optimierter Kontrollflussgraph zu Quelltext 3



von Block 2 unter keinen Umständen (im vorliegenden Kontext) geworfen werden. Die entsprechende Kante kann aus dem Kontrollflussgraphen also herausgenommen werden.

2.6 Motivation und Strategie

Bisher war es möglich mit Hilfe einer intraprozeduralen Analyse einen Teil der unmöglichen Ausnahmen zu erkennnen. So kann man z. Bsp. im Programm aus Quelltext 3 nachweisen, dass der Zugriff auf den This-Zeiger in Zeile 12 keine Ausnahme auftreten kann. Abb. 4 zeigt den durch intraprozedurale Analyse präzisierten CFG.

Dass hingegen der Zugriff auf den Parameter `d` in Zeile 10 ebenfalls, keine Ausnahme auftreten kann, wird nicht erkannt (ebensowenig die „weitergereichte Exception“ im Aufruf `needsAParameter(d)`). Hierfür wird die interprozedurale Analyse benötigt, die im Rahmen dieser Arbeit erstellt wurde. Dabei wurde die bereits vorhandene intraprozedurale Analyse erweitert.

Abb. 3 zeigt den Code aus Quelltext 3 in nicht optimierter Form. Die vorhandene intraprozedurale Analyse kann die NPE, welche in Zeile 9 auftritt, nicht als unnötig erkennen, da sie keine Informationen über den Parameter `d` besitzt. Die Exception in Zeile 11 hingegen kann als unnötig erkannt werden, da die intraprozedurale Analyse sicher davon ausgeht, dass ein This-Zeiger nicht `null` sein kann. Daraus ergibt sich der mit intraprozeduraler Analyse optimierte ICFG aus Abb. 4.

Übrig bleiben nun noch zwei Exception-Kanten. Aber auch diese können nach der Theorie entfernt werden. `d` in Zeile 9 ist sicher nicht `null`. Daraus resultierend kann auch die Kante vom Aufruf von `needsAParameter(d)` zum `EXIT`-Knoten entfernt werden, wie der ICFG in Abb. 5 zeigt. Der ICFG besteht also, nach Entfernen aller unnötigen Exception-Kanten, aus zwei einfachen linearen CFG.

Die beiden zuletzt genannten Exceptions lassen sich mit einer intraprozeduralen Analyse nicht erkennen. Sie sind repräsentativ für eine Menge an Exceptions, die nur mit interprozeduraler Sicht analysiert werden können[5].

3 Implementierung

Zu implementieren ist nun eine interprozedurale Analyse, die eine bereits vorhandene intraprozedurale Analyse erweitert. Die intraprozedurale Analyse (im Folgenden `intra`) wird sowohl in der Modellierung als auch in der Implementierung von ihrer tatsächlichen Aufgabe (nämlich dem Optimieren

[5]Die A-Posteriori-Messungen haben ergeben, dass diese Menge statistisch etwa gleich groß ist wie die Menge der Exceptions, die mit intraprozeduraler Sichtweise analysiert werden kann. Siehe hierzu die Evaluation unter Abs. 1

von NPEs) so weit wie möglich abstrahiert[6] und als Blackbox betrachtet. So ist das implementierte interprozedurale Analyse-Framework auch für andere interprozedurale Analysen (ggf. mit kleinen Anpassungen) zu verwenden.

`inta` liefert nach der Analyse eines CFG Informationen über die SSA-Variablen der Zwischencodedarstellung wie folgt:

$$Answeisung \rightarrow (SSAVar \rightarrow \{?, 0, 1, *\})$$

Es wird also für jeden Block im CFG jede SSA-Variable mit genau einem Status aus *unbekannt*, *sicher null*, *sicher nicht null* sowie *entweder sicher nicht null oder sicher null* belegt (Reihenfolge wie in oben dargestellter Formel).

3.1 WALA

Die vorliegende Analyse verwendet als Framework die *T. J. Watson Libraries for Analysis*[7], kurz *WALA*. WALA wird hier verwendet, um ein im Quelltext vorliegendes Java-Programm in die WALA-eigene SSA-basierte Registertransfer-Repräsentation (IR) zu übersetzen. Für diese Repräsentation stellt WALA zudem Klassen für die Darstellung als CFG bereit (`ControlFlowGraph`).

3.2 Design der interprozeduralen Analyse

3.2.1 Überblick

Die vorliegende interprozedurale Analyse arbeitet auf dem ICFG eines Java-Programmes und ist in zwei Phasen eingeteilt: `run` und `collect`.

`run`: Die interprozedurale Analyse wird mit einer als Startpunkt definierten Prozedur `s` initialisiert. Auf dieser Prozedur wird dann `intra` ausgeführt. Wenn die Prozedur komplett behandelt ist, werden alle in `s` aufgerufenen Prozeduren als neuer Startpunkt `s` definiert; der Prozess wiederholt sich dann rekursiv. Dies entspricht der Aufruf-Ordnung (invocation order), wie sie in [Muc97, S. 609] definiert ist, also einer Tiefensuche auf dem ICFG.

`collect`: Beim zweiten Durchlauf werden in rückwärtiger Aufruf-Ordnung (reverse invocation order, [Muc97, S. 609]) die Aufruf-Anweisungen ein weiteres mal besucht, um bei einer Elimination aller NPEs in der aufgerufenen Prozedur `p` die Exception-Kante (s. Abs. 2.5) `invoke` $\rightarrow_{exc}$ `EXIT` ebenfalls zu entfernen. Im Falle einer Rekursion wird beim zweiten Durchlauf abgebrochen. Dies erfolgt unter Verlust von Präzision[8].

[6] Das Propagieren der Parameter ist unabhängig von der eigentlichen Funktion von `intra`. Das Verarbeiten der Ergebnisse von `intra` kann aber nur mit Kenntnis der Funktion und über daran angepasste Schnittstellen erfolgen.

[7] http://wala.sourceforge.net/wiki/index.php/Main_Page

[8] Ein komplexerer Lösungsansatz mit Fixpunktiteration anstatt Reverse Invocation Order über Rekursion würde diesen Kompromiss nicht eingehen. Eine Fixpunktiteration, die alle ICFG eines durchschnittlich großen Java-Programms mit allen Zuständen aller SSA-

Die Analyse muss in diesem Prozess insbesondere den This-Zeiger korrekt behandeln sowie erhebliche Mengen an Daten protokollieren.

In vereinfachtem Pseudocode ist dies in Quelltext 4 aufgeführt.

Quelltext 4 Rekursive Prozedur zur Propagation der Parameter-Stati zwischen intra- und interprozeduraler Analyse

```
 1: procedure findAndInject(startnode, parameterstatus)
 2:     if node already visited then
 3:         return
 4:     end if
 5:     if parameterState ≠ null then
 6:         startnode.inject(parameterState)
 7:     end if
 8:     cfg ← intra(cfg)                          ▷ analysiere mit intra
 9:     for all invoke ∈ cfg do
10:         parameters[invokeBlock] ← extractParameterState(invoke)
11:         invokeSet ← getInvokeTarget(invoke)
12:     end for
13:     for all target ∈ invokeSet do
14:         findAndInject(target, parameters)▷ Reverse Invocation Order
15:     end for
16:     findAndInject(this, null)        ▷ wiederholter Durchlauf (collect)
17: end procedure
```

3.2.2 Einbindung und Erweiterung von `intra`

Die der Analyse zu Grunde liegende Repräsentation WALA-IR ist SSA-basiert, demnach arbeitet die vorliegende Analyse mit SSA-Variablen. Dies vereinfacht die Analyse, da eine Variable nur einmal zugewiesen werden kann (s. Definition der SSA-Form u.A. bei [ALSU06, S. 446]). Die vorhandene Analyse `intra` markiert bei einem Durchlauf auf einem CFG jede SSA-Variable entweder mit `null`, `not null`, `unknown` oder `both`[9]. Die Sichtweite ist nur intraprozedural. Parameter werden mit `unknown` initialisiert, obwohl ggf. genauere Informationen (`null`, `not null`) verfügbar wären.

In WALA-IR sind per Konvention die ersten n SSA-Variablen[10] einer Prozedur die Parameter (wenn die Prozedur n Parameter empfängt). Dies schließt den `this`-Zeiger mit ein. Er ist im nicht-statischen Fall implizit der erste Parameter, also auch die erste SSA-Variable und immer `not null`.

Variablen im Speicher vorrätig hält, erscheint jedoch viel zu kostenintensiv.

[9] Diese Struktur der Variablenbelegungen ist ein Verband. Da die daraus resultierenden Eigenschaften für die Analyse aber nicht relevant sind (relevant sind sie nur innerhalb der Fixpunktiteration von `intra`), wird die zugehörige Theorie an dieser Stelle nicht behandelt.

[10] WALA beginnt die Zählung mit SSA-Variable 1, nicht mit 0

Die vorliegende Analyse betrachtet diesen Sonderfall und initialisiert einen vorhandenen `this`-Zeiger immer mit `not null`.

In der `run`-Phase werden wie in Abs. 3.2.1 beschrieben, `intra` auf dem Start-CFG `s` ausgeführt, sodass alle SSA-Variablen markiert sind. Danach werden die aufgerufenen Methoden in `s` gesammelt. In diesem Prozess wird anschließend für jeden eingesammelten Knoten `intra` so initialisiert, dass die SSA-Variablen, die Parameter (insbeondere den `this`-Zeiger) repräsentieren, mit dem Ergebnis der `intra`-Analyse initialisiert.

Demnach wird in WALA-IR nicht unterschieden, ob eine SSA-Variable in einer Prozedur eine lokale Variable oder ein Parameter ist. Das interprozedurale Framework muss also bei einem `invoke p` in der Prozedur `q` die in `q` über `intra` gewonnenen Erkenntnisse einsammeln und an `p` übergeben. Dies geschieht über eine Erweiterung des Interfaces von `intra` mit der Methode `setParameterState(ParameterState state)`. (Die Klasse `ParameterState` ist schlicht ein Wrapper, der einem Parameter seinen Status zuweist.) Zu den Constraints des `intra`-Interfaces gehört nun, dass bei einer Initialisierung generell davon ausgegangen werden muss, dass bereits Informationen über SSA-Variablen vorhanden sind (und nicht überschrieben werden sollten).

3.2.3 Implementierung des ICFG

Das Verwalten mehrerer CFGs ist sehr speicherintensiv. Daher werden die ICFG effizient gespeichert, indem nur eine Liste der in Abs. 2.2 definierten Call-Kanten im Speicher vorgehalten wird. Dies erlaubt es, die für die `intra`-Analyse ohnehin notwendigen normalen CFG zu verwenden. WALA bietet zwar die Klasse `InterproceduralCFG` an. Ausgehend von der Definition in Abs. 2.2 wird jedoch nur die Menge *Call* als Erweiterung gewöhnlicher CFG benötigt. Der rekursive Abstieg erlaubt den Aufruf direkt beim „Einsammeln“ (`collect`) der Invoke-Blöcke eines CFG. Der ICFG wird also nicht direkt gespeichert; die Call/Return-Kanten in *Call* sind also nur im Augenblick des Aufrufs der Intra-Analyse tatsächlich bekannt.

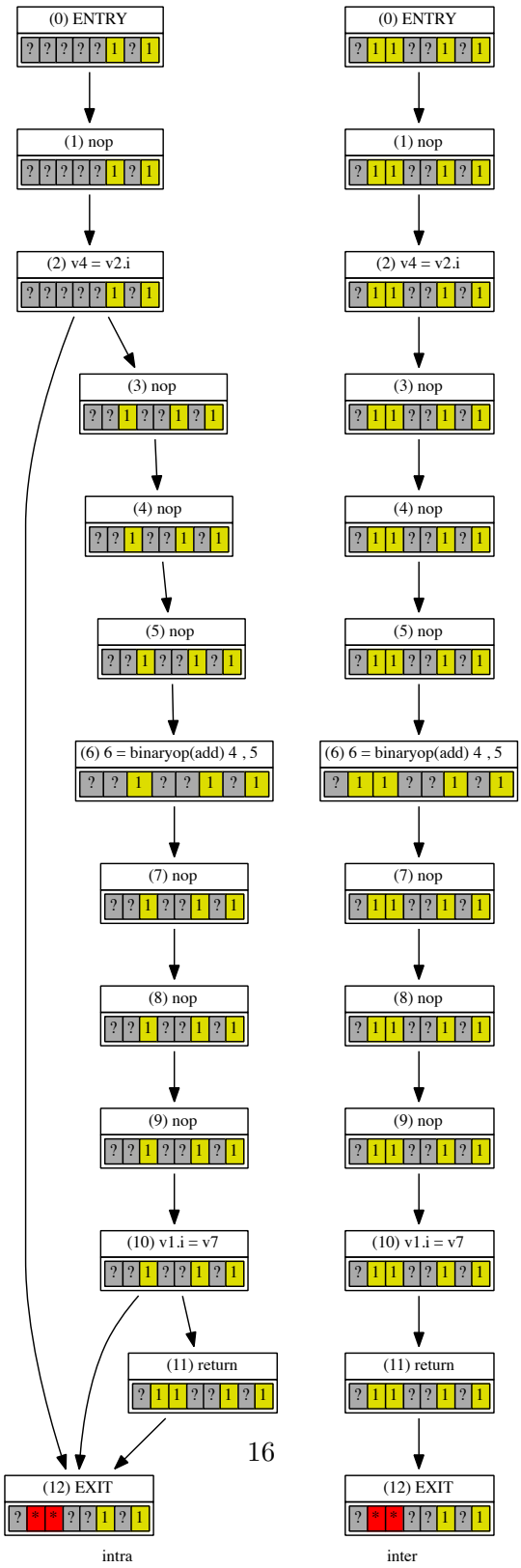
Damit verbessert sich die Skalierbarkeit. Der Speicherverbrauch ist (innerhalb des interprozeduralen Frameworks) praktisch unabhängig von der Größe des Programms.

3.3 Beispiel

Betrachten wir das bereits aufgeführte Code-Beispiel aus Quelltext 3. Wie in Abs. 2.5.1 erklärt, ist bei dem Zugriff auf `d.i` in Zeile 9 keine NPE möglich, da in Zeile 4 der Parameter `d` vor dem Aufruf initialisiert wurde.

Abb. 6 zeigt die beiden Analysen nun im Vergleich. Die Darstellung ist die direkte Ausgabe der Analyse, ohne manuelles Einfügen roter Exception-Kanten. Unter dem Namen des Blockes sind alle SSA-Variablen in kleinen

Abbildung 6: Ergebnis der Analyse von Quelltext 3 (nur Prozedur `needsAParameter`). $v4 \equiv j$, $v2 \equiv d$, $v1 \equiv$ `this`



Blöcken dargestellt. Eine gelbe 1 bedeutet, dass die Variable sicher nicht null ist. Das Initialisieren von SSA-Varaible 1 (this-Pointer) und 2 (`new D()`) ist beim Vergleich von Block 0 zu erkennen. Ausgehend von dieser Initialisierung können die beiden NPE, die in Zeile 9 und Zeile 11 theoretisch geworfen werden können, nicht auftreten. Der CFG nach inter-Analyse hat diese beiden Kanten entfernt (rechts). Der CFG der Prozedur `needsAParameter` ist also linear geworden.

Die Prozedur `needsAParameter` wirft nun keine Exception mehr, insbesondere auch keine NPE. Dadurch kann auch der Aufruf in `invokesMethod` keine Exception mehr werfen. Die Analyse entfernt darum auch die in der intra-Analyse noch vorhandene Exception-Kante, wie Abb. 7 zeigt. Zudem ist zu erkennen, dass auch in diesem Fall der This-Zeiger als SSA-Variable 1 korrekt initialisiert wird.

Die von der Analyse erzeugten präziseren CFG aus Abb. 6 und Abb. 7 sind damit deckungsgleich mit dem theoretisch hergeleiteten ICFG aus Abb. 5. Der Ausdruck „call *Klassennamen*“ ist jeweils die WALA-interne Schreibweise eins Aufrufes eines Prozedur.

3.4 Weitere Infrastruktur

Im Rahmen dieser Studienarbeit entstanden neben der interprozeduralen Analyse noch weitere Artefakte, die das Entwickeln, Evaluieren und das Finden von Fehlern erleichtern. Diese sind:

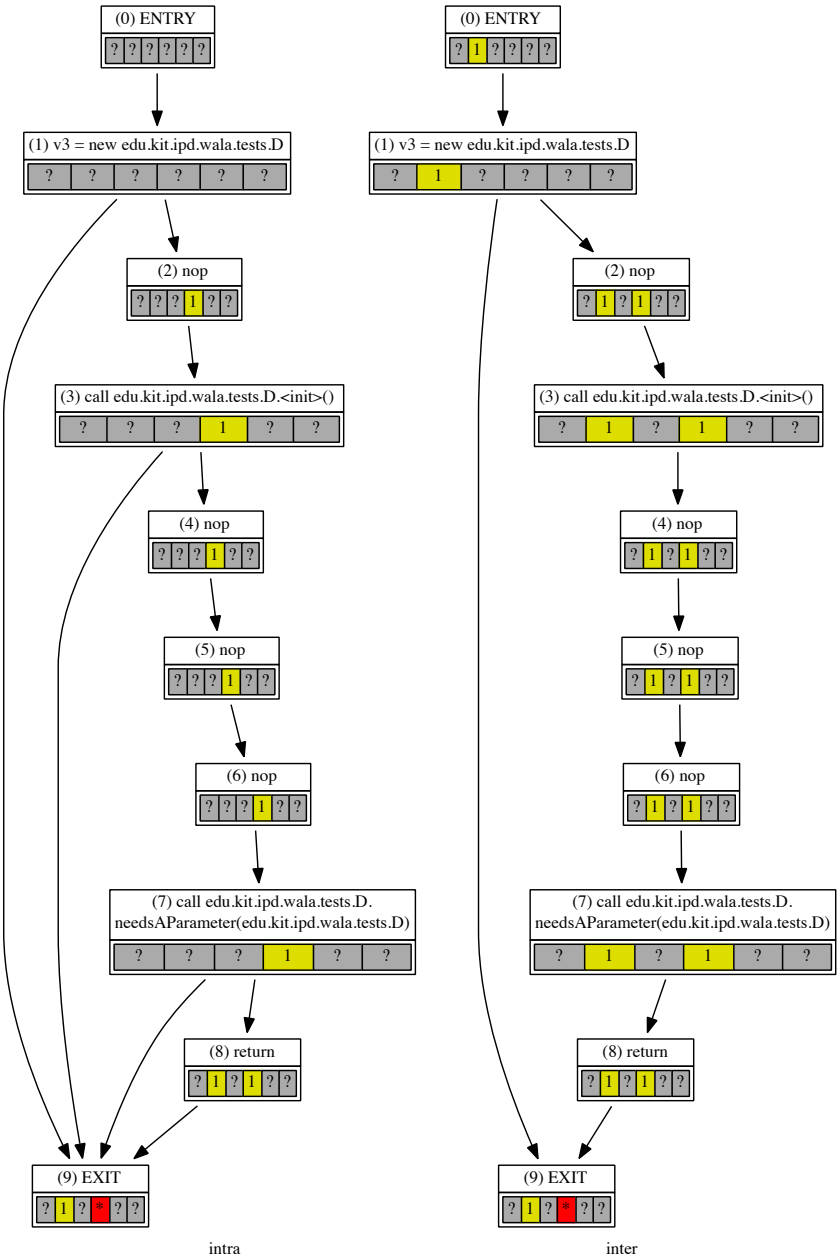
3.4.1 Visualisierung

Das Debuggen von Datenstrukturen wie den CFG ist unter Umständen sehr schwer, da Graphen mit klassischem Logging nur schwer darstellbar sind. So entstand im Rahmen der vorliegenden Arbeit ein Graph-Decorator, welcher eine umfassende Darstellung von CFG ermöglicht. Seine Ausgabe ist in den Abb. 6 und 7 sichtbar. Er verarbeitet insbesondere auch die Statūs der SSA-Variablen in einem Block und gibt diese in einer eingefärbten Tabelle in jedem Block aus.

3.4.2 Initialisierung

Die Vorbereitungen und Prozesse, die in WALA nötig sind, um von einem Programm-Quelltext im Textformat zu einem Callgraphen bzw. zu den verschiedenen CFGs zu gelangen, sind komplex. Wie weiter oben erklärt, ist eine Zwischenübersetzung in IR nötig. Diese Schritte werden in der Klasse `AnalysisBuilder` gekapselt durchgeführt.

Abbildung 7: Ergebnis der Analyse von Quelltext 3 (nur Prozedur `invokesMethod`) mit $v3 \equiv d$



3.4.3 Mittel zur Evaluation

Zur Evaluation müssen verschiedene Zustände der CFG zu verschiedenen Zeitpunkten festgehalten werden. Hierfür wurde eine Wrapper-Klasse `Textbox` geschrieben, die für ein perfomantes und bequemes Testen alle relevanten Informationen kapselt. Die Klasse `EvaluationInfo` sammelt für jeden CFG eines Analysedurchlaufes kompakt alle Informationen. Um Vergleiche anstellen zu können werden sowohl Daten vor jeglicher Analyse als auch die Ergebnisse der inter- und intraprozeduraler Analyse protokolliert.

Eine Klasse `EvaluationRunner` dient zur letztendlichen Konfiguration der Evaluation. Sie protokolliert die Testergebnisse in einer CSV-Datei. Zudem lässt sich eine Menge von Prozeduren definieren, für die jeweils ein visueller Graph in den Zuständen vor der Analyse, nach der `intra`-Analyse und nach der interprozeduralen Analyse geplottet wird.

3.4.4 Testgetriebene Entwicklung

Die Entwicklung der vorliegenden Analyse entstand zu weiten Teilen testgetrieben mit JUnit. Um bequem Test schreiben zu können, beinhaltet die Klasse `TestHelper` einige Methoden, um über Assertions erwartete Optimierungen validieren zu können. So lassen sich einfache und prägnante Tests formulieren.

Im Code Quelltext 3 muss die interprozedurale Analyse 2 Kanten entfernen. Ein entsprechender beispielhafter Testcase in JUnit ist in Code Quelltext 5 aufgeführt. In Zeile 9 und 10 wird sichergestellt, dass im Vergleich zur intra-Analyse 2 weitere Kanten entfernt wurden.

Für viele exemplarische oder spezielle Fälle gibt es umfassende Testfälle. Dieser Ansatz hat sich beim Debuggen der Analyse sehr bewährt, vor allem als die Komplexität der Analyse im Verlauf der Arbeit angestiegen ist.

Quelltext 5 Beispielcode eines JUnit-Testcases zur Optimierung von Code 3

```
1  public void shouldRunD(){
2    inter.run();
3
4    HashMap<CGNode, OptimizationInfo> intraCFG
5      = inter.getIntraCFGs();
6    HashMap<CGNode, OptimizationInfo> interCFG
7      = inter.getInterCFGs();
8
9    assertEquals(2,
10     TestHelper.removedEdges(intraCFG, interCFG));
11 }
```

4 Evaluierung

4.1 Metriken

Ziel der Analyse ist es, unnötigen Kontrollfluss aufgrund nicht auftretender Exceptions zu zu erkennen. Dadurch können die entsprechenden Kanten aus dem CFG entfernt werden. Man eliminiert somit die Anzahl der Kanten im CFG, um weitere Analysen performanter ausführen zu können. Offensichtliche Metrik ist also die Anzahl der Kanten in den analysierten und optimierten Graphen bzw. ihr Verhältnis zueinander. Interessant ist zudem, wie viele Exception tatsächlich in welchem Schritt entfernt werden können. Hierzu wird auch die Anzahl der tatsächlich potentiell geworfenen NPE erfasst.

Protokolliert werden pro Prozedur folgende Werte, jeweils für den originalen CFG (im Folgenden "orig"), für einen mit rein intraprozeduraler Sichtweise ("intra") sowie die Analyse mit interprozeduraler Sichtweite ("inter"):

1. Anzahl der Kanten im CFG
2. Anzahl der potentiell auftretenden NPE
3. Anzahl der Verzweigungen im CFG

Daraus berechnen lässt sich:

1. Anzahl bzw. Anteil der wegoptimierten Kanten in den CFG
2. Anzahl bzw. Anteil der wegoptimierten NPE in den CFG
3. Anzahl bzw. Anteil der wegoptimierten Verzweigungen in den CFG

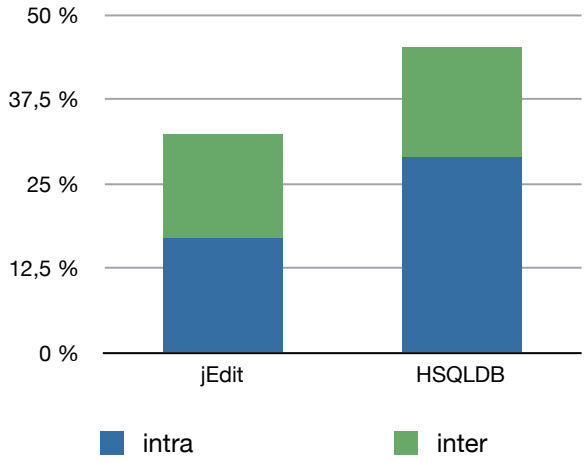
4.2 Messergebnisse

Die Analyse wird mit dem quelloffenen Programmcode von jEdit[11] und HSQLDB[12] getestet und evaluiert. Beide Quelltexte sind sehr umfangreich und repräsentativ für Java-Programme "in the wild". jEdit ist ein gutes Beispiel für ein Anwendungsprogramm mit GUI und einem umfangreichen Klassenmodell, HSQLDB ist ein Beispiel für eine effizient und auf Performance optimierte Server-Anwendung ohne GUI.

In den Statistiken werden Fälle, in denen keine Optimierung stattfinden kann, nicht ausgeklammert. "0-Zeilen" sind also insbesondere in die Berechnungen von Verhältnissen mit eingenommen und wirken sich entsprechend auf den Durchschnitt und auf den Anteil optimierter NPE aus. Ausgeklammert und nicht erfasst werden jedoch einige Prozeduren in der Java-API

[11] ein Texteditor geschrieben in Java mit 176668 LOC, `http://jedit.org`
[12] eine Datenbank geschrieben in Java mit 277122 LOC, `http://hsqldb.org/`

Abbildung 8: Anteil eliminierter NPE. 100% entspricht aller im Programm potentiell auftretende NPE



selbst, insbesondere solche in `Java.lang.*`. Sie werden nicht analysiert. Der Umfang dieses Filters lässt sich im `EvaluationRunner` konfigurieren.

Zuerst betrachtet wird die Anzahl der optimierten NPE. Die Ergebnisse:

Tabelle 1: Automatisch erkannte, nicht auftretende NPE

Programm	Anzahl NPE	eliminiert intra	eliminiert inter
jEdit	41612	7038	13464
HSQLDB	30540	8850	13821

Die vorliegende Erweiterung von `intra` ist demnach eine Verbesserung der Optimierung um 56,16% bei HSQLDB und um 91,30%. Dieser große Unterschied im Erkennen von Exception ist nach Analyse einiger Methoden in in jEdit und HSQLDB auf die unterschiedliche Qualität des Programmcodes zurückzuführen. HSQLDB ist als möglichst performante Datenbank sehr effizient programmiert und vermeidet im Programmierstil generell Schreib- und Lesezugriffe aus Gründen besserer Performance. jEdit bietet mit einer GUI und einem komplexen Objektmodell viele Möglichkeiten für nicht initialisierte Referenzvariablen.

Mit beiden Analysen zusammen ist es nun möglich, bis zu ca. der Hälfte (HSQLDB: 45,26%) aller generell auftretenden NPE im CFG zu eliminieren.

Abb. 8 zeigt die eliminierten NPE anteilig an allen im Programm vorhandenen NPE. Im Balkendiagramm durch Stapel abgetrennt sind die jeweiligen Anteile von intra- und interprozeduraler Sichtweise.

4.2.1 Erweiterte Messung

Das Eliminieren von NPE ist kein Selbstzweck, sondern dient dem generellen Eliminieren von Kontrollflusskanten (siehe Motivation in Abs. 1). Darum ist eine weitere interessante Kenngröße die Anzahl der eliminierten Kontrollflusskanten im CFG, die entfernt wurden. Auch diese Anzahl wird im `EvaluationRunner` protokolliert. Die Datenstruktur `ControlFlowGraph` von WALA entfernt Kanten automatisch, sobald ein anhängiger Knoten entfernt wurde. Das Entfernen geschieht also implizit und muss nicht implementiert werden.

Tabelle 2: Eliminierte Kontrollflusskanten

Programm	Anzahl Kanten	eliminiert intra		eliminiert inter	
jEdit	249728	7302	2,92%	14232	5,70%
HSQLDB	178698	9048	5,06%	14603	8,17%

Zu Erkennen ist, dass insgesamt ein Anteil von unter 10% an eliminierten Kontrollflusskanten zu erwarten ist. Wieder manfifestiert sich eine Verbesserungs-Rate von 40% bis 50% zwischen einer Analyse mit intra- und jener mit intraprozeduraler Sichtweise.

Die Anzahl der Kanten allein ist jedoch keine hinreichend charakterisierende Metrik. Vielmehr die Verzweigungen in einem CFG verursachen Kosten in den aufbauenden Analysen. Tab. 3 zeigt die Daten einer Messung, bei der die Verzweigungen vor einer Analyse und jeweils nach intra- und interprozeduraler Analyse gezählt wurden. Mit intraprozeduraler Analyse können die Verzweigungen um 14% reduziert werden, mit interprozeduraler Analyse um 23%. Der Unterscheid zwischen intra- und interprozedural liegt wieder im Bereich von 40-50% Verbesserung.

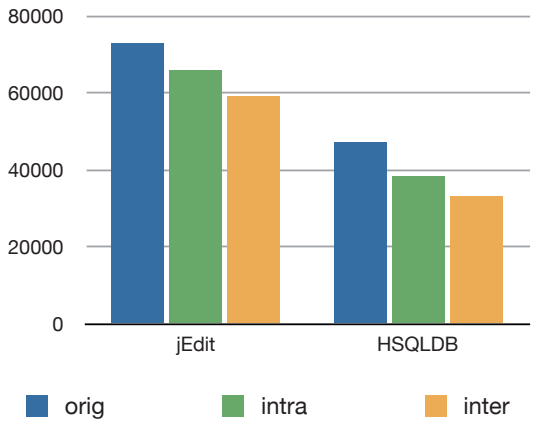
Tabelle 3: Anzahl der Verzweigungen in den CFGs der Programme ohne Analyse, mit intra- und mit interprozeduraler Analyse

Programm	original	intra	inter
jEdit	72960	65747	59172
HSQLDB	47170	38269	33156
anteilig	100,00%	86,59%	76,86%

5 Fazit und Ausblick

Die vorliegende Arbeit hat gezeigt, dass es generell möglich ist, annähernd die Hälfte aller möglicher NPE in einem Java-Programm sicher ausschließen

Abbildung 9: Balkendiagramm der Daten von Tab. 3



zu können. Wenn die vorliegende Analyse noch um Sonderfälle erweitert wird, ist auch ein Wert von über 50% denkbar. Eine Reduktion aller Kontrollflusskanten um bis zu 8% bringt den aufbauenden Analysen eine höhere Präzision und erkennbare Performance-Gewinne.

Die generelle Erweiterung einer intraprozeduralen Analyse zu einer interprozeduralen Analyse bringt (orientiert an den durchgeführten Evaluationen) eine Verbesserung um 40 bis 50%.

5.1 Mögliche Verbesserungen und Erweiterungen der vorliegenden Analyse

Die vorliegende interprozedurale Analyse ist zu weiten Teilen sehr abstrakt gehalten. Einzig der zweite Durchlauf (`collect`, s. 3.2.1) ist speziell auf das Arbeiten mit NPE bezogen. Wenn diese Funktion etwa mit dem Entwurfsmuster "Strategie" entkoppelt wird, kann der vorliegende Code als Basis eines ganzen interprozeduralen Frameworks verwendet werden, der für alle intraprozeduralen Analysen verwendet werden kann, welche durch Parameter-Propagation verbessert werden können.

Verwendet wird in der vorliegenden Implementierung der pragmatische Ansatz mit Rekursion in Reverse Invocation Order ([Muc97, S. 609]). Ein theoretischerer Ansatz könnte die Verwendung von Fixpunktiteration sein. Der implizite Aufbau der ICFG durch Rekursion ist dann jedoch hinfällig; man müsste eine eigene Datenstruktur des ICFG implementieren (oder jene in WALA verwenden). Dadurch könnte insbesondere die Präzision noch verbessert werden.

Da die Analyse aufbauend auf verwandte Optimierungen (wie etwa Constant Propagation) ist, wirken sich Verbesserungen in diesen Optimierungen auch auf die Präzision der Exception-Analyse aus.

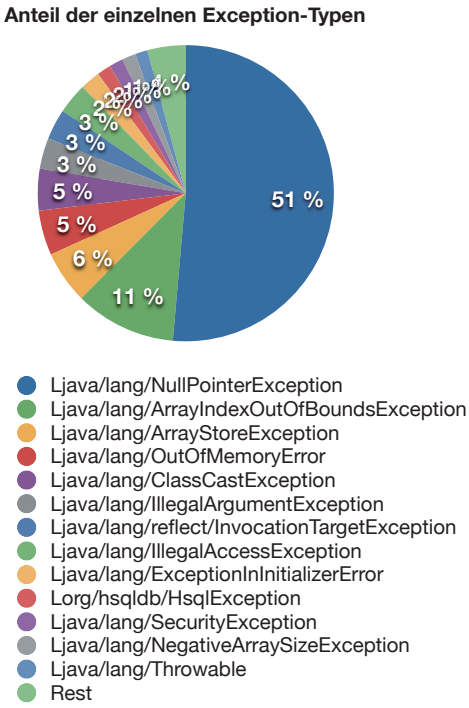
Der Ansatz des testgetriebenen Entwickelns hat sich bewährt. Das im Rahmen dieser Arbeit entstandene Test-Framework kann erweitert und ausgebaut werden. Insbesondere intuitive Test-Helper (wie etwa `shouldEliminateNPE(3, inter, intra)`) ermöglichen eine hohe Testabdeckung und transparentes Debugging.

A Häufigkeitsanalyse Nullpointer-Exceptions

Folgende Tabelle und zugehöriges Diagramm (Abb. 10) zeigen, welche Exceptions in welcher Anzahl in den beiden Beispielprogrammen jEdit und HSQLDB vorkommen. Die Auflistung zeigt, dass Nullpointer-Exceptions die signifikante Mehrheit ausmachen. Ihr Anteil an allen Exceptions liegt bei über 50%.

Abbildung 10: Häufigkeitsanalyse Exceptions

	Anzahl	HSQLDB	jEdit
Ljava/lang/NullPointerException	94745	41086	53659
Ljava/lang/ArrayIndexOutOfBoundsException	20393	5237	15156
Ljava/lang/ArrayStoreException	10600	1699	8901
Ljava/lang/OutOfMemoryError	9029	3294	5735
Ljava/lang/ClassCastException	8387	1386	7001
Ljava/lang/IllegalArgumentException	6205	63	6142
Ljava/lang/reflect/InvocationTargetException	6169	21	6148
Ljava/lang/IllegalAccessException	6163	20	6143
Ljava/lang/ExceptionInInitializerError	4028	0	4028
Lorg/hsqldb/HsqlException	2827	2827	0
Ljava/lang/SecurityException	2793	0	2793
Ljava/lang/NegativeArraySizeException	2759	1049	1710
Ljava/lang/Throwable	2525	1103	1422
Rest	7716		



Abbildungsverzeichnis

Literatur

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2 edition, August 2006.

[Gif11] Dennis Giffhorn. *Slicing of Concurrent Programs and its Application to Information Flow Control*. PhD thesis, Karlsruhe Institute of Technology, to appear in 2011.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Longman, Amsterdam, 3 edition, June 2005.

[Gra97] Mark Grand. *Java language reference*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1997.

[Gra10] Jürgen Graf. Speeding up context-, object- and field-sensitive sdg generation. In *9th IEEE International Working Conference on Source Code Analysis and Manipulation*, September 2010. Accepted at SCAM 2010.

[HS09] Christian Hammer and Gregor Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8(6):399–422, December 2009. Supersedes ISSSE and ISoLA 2006.

[Mor98] Robert Morgan. *Building an optimizing compiler*. Digital Press, Newton, MA, USA, 1998.

[Muc97] Steven S. Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.