Raul Fechete, Georg Kienesberger, Peter Minarik

Busy Wait Analyser

Bakkalaureatsarbeit

Betreuender Professor: Dr. Johann Blieberger

Wien, 30. Juni 2005

Inhaltsverzeichnis

1 Einleitung

Der *Busy Wait Analyser* ist ein kommandozeilengesteuertes Tool, welches allgemein gesprochen dazu dient, Java Programme nach bestimmten Anweisungen zu durchsuchen, die darauf hinweisen, dass das sogenannte *Busy Waiting* verwendet wurde. Die hierzu verwendete Analyse basiert auf dem von *Blieberger et al.* [BBS03] vorgestellten Algorithmus.

1.1 Worum geht es

Wie bereits erwähnt, geht es darum, in einem Java Programm die Verwendung von *Busy Waiting* zu finden und dem Autor der Software somit zu ermöglichen, diese zu eliminieren. Doch was genau beschreibt der Begriff *Busy Waiting*?

Dabei handelt es sich um eine Programmiertechnik, um einen Prozess (bzw. einen Thread) auf das Eintreten einer Bedingung warten zu lassen. Dabei läuft der Prozess so lange "im Kreis", bis die Bedingung erfüllt ist. Diese Technik wird vor allem für Spinlocks verwendet.

Das hat den großen Nachteil, dass der Prozess die volle Rechenleistung des Systems damit beansprucht, immer wieder die Bedingung zu prüfen, und damit andere Prozesse ausbremst. Das ist vor allem auch deshalb unvorteilhaft, weil es ja gerade die anderen Prozesse sind, die dafür sorgen müssen, dass die Bedingung erfüllt wird.

Busy Waiting gilt im allgemeinen als sehr schlechter Programmierstil und sollte nur als allerletzte Möglichkeit verwendet werden. Moderne Betriebssysteme bzw. Laufzeitumgebungen bieten in der Regel Locks oder Monitore an, die es erlauben, eine Bedingung nur genau dann erneut zu prüfen, wenn sich die Variablen, von denen sie abhängt, geändert haben.

1.2 Busy Waiting im Detail

Um sich das *Busy Waiting* (BW) besser vorstellen zu können, sieht man in Listing 1.1 ein konkretes Beispiel dafür in Java. Solange `flag` **`false`** ist, wird die Schleife immer weiterlaufen. Ob `flag` **`true`** wird hängt wiederum von anderen Threads ab, die, wie schon erwähnt, gerade durch das *Busy Waiting* zusätzlich ausgebremst werden.

1.2.1 Der Dekker Algorithmus

Das wohl bekannteste Paradebeispiel für *Busy Waiting* ist der so genannte *Dekker Algorithmus* [Dij68], den wir in Listing 1.2 veranschaulicht haben. Dieser Algo-

Listing 1.1: Ein konkretes Beispiel in Java

```
1 public class Bsp{
2     public boolean flag;
3
4     public void versuch{
5         while(!flag){
6             //tue nichts
7         }
8     }
9 }
```

Listing 1.2: Der Algorithmus von Dekker

```
1  public class Dekker{
2      int turn = 1;
3      boolean flag0 = false;
4      boolean flag1 = false;
5
6      public void p0(){
7          //claim critical section:
8          flag0 = true;
9          while(flag1 == true){
10             if(turn == 1){
11                 flag0 = false;
12                 while(turn == 1){
13                     //do nothing
14                 }
15                 flag0 = true;
16             }
17         }
18         //critical section:
19             //do something
20         //leave critical section:
21         turn = 1;
22         flag0 = false;
23     }
24 }
```

rithmus ist die erste bekannte Lösung für das *Mutual Exclusion Problem*, welche keine strikte Alternation verwendet.

1.2.2 Die Busy Wait Variablen

In der Analyse des *Busy Waiting* interessieren uns in weiterer Folge vor allem die Variablen, die in einer Schleife darüber entscheiden, ob diese weiter ausgeführt oder ob sie verlassen wird. Wir nehmen an, dass wir diese Variablen in den *Exit Conditions* der Schleife finden und überprüfen, ob diese überhaupt geschrieben werden. Ist dies nicht der Fall, liegt die Vermutung nahe, dass es sich um eine sogenannte *Busy Wait Variable* handelt, da sie nur von einem anderen Thread aus geschrieben werden kann.

Diese Variablen sind schließlich und endlich genau jene wichtigen Merkmale, auf die wir den Benutzer unseres Tools hinweisen wollen.

2 Die SootCom Schnittstelle

Bevor man mit der *Busy Wait Analyse* eines Programms beginnen kann, ist es nötig, den zugehörigen Kontrollflussgraphen zu ermitteln und Informationen über Variablen und deren Verwendung zu extrahieren. Diese Aufgabe wurde unter Zuhilfenahme von *Soot* realisiert und wird im Folgenden erläutert.

2.1 Soot

Soot ist ein *Java Optimization Framework* [soo05], also ein allgemeines und erweiterbares Programmgerüst um Java Bytecode zu inspizieren, zu optimieren und zu transformieren. *Soot* besitzt ein umfangreiches und starkes API, welches dem Benutzer erlaubt, High Level Analysen zu implementieren oder ganze Programme zu transformieren. Besteht das Ziel lediglich darin, einzelne, einfach gestrickte Optimierungen bzw. Analysen durchzuführen, kann *Soot* allerdings auch als Stand-Alone Tool verwendet werden.

Das Kernstück von *Soot* bilden die verschiedenen Repräsentationsformen des ursprünglichen Bytecodes, welche Code-Transformationen auf verschiedenen Abstraktionsebenen, vom Stack-Code bis zum *Typed Three-Address Code*, ermöglichen. Der Bytecode, welcher durch einen beliebigen Compiler erzeugt wurde, wird von *Soot* eingelesen und zum Beispiel als optimiertes *Classfile* wieder ausgegeben.

2.2 Interaktion mit Soot

Aus verschiedenen Gründen schien es vorteilhaft, eine klare Trennung zu *Soot* anzustreben, anstatt die vorgesehene Funktionalität in das Framework zu integrieren. Abgesehen von der klareren Strukturierung des Projekts und der erhöhten Wiederverwendbarkeit des Kernalgorithmus dürfte sich so vor allem auch die Laufzeit verbessern, da nur ein einmaliger Datenaustausch mit *Soot* stattfindet. *Soot* als umfassendes *Java Optimization Framework* stellt für die Zwecke dieser Arbeit viel zu träge, überdimensionierte Objekte und Methoden zur Verfügung, weshalb auf diese nach Erhalt der relevanten Daten nicht mehr zugegriffen wird. Diese Idee verwirklicht unsere SootCom Schnittstelle, welche im Grunde aus `bwa.sootcom.SootCom` und `bwa.sootcom.UnitEval` besteht.

Als Basis für die Analyse werden die bereits kompilierten *Classfiles* herangezogen und nicht etwa die zugehörigen Sourcen. Dieser Umstand ergibt sich aus der Natur von *Soot* als *Optimization Framework*, und bringt den Vorteil mit sich, dass auch Programme untersucht werden können, deren Sourcen nicht verfügbar sind. Allerdings können, sofern beim Kompilieren auf das Generieren der Debug Informationen verzichtet wurde (`javac -g`), weder Zeilennummern noch originale Variablennamen ausgegeben werden.

Die *SootCom* Klasse kann durch insgesamt drei Konstruktoren instantiiert werden, wobei der Unterschied im Umfang der analysierten Programmteile liegt. Im spezifischsten Fall wird durch Angabe von Methoden- und Klassenname nur diese eine, gewünschte Methode verarbeitet, sofern sie nicht überladen, also durch diese Kriterien eindeutig identifizierbar ist. Wird lediglich der Name einer Klasse übergeben, so werden alle darin enthaltenen Methoden der Analyse zugeführt, wobei auch diese Möglichkeit selten zur Auswertung umfangreicher Programme geeignet sein wird. In der Regel sollte also der dritte Konstruktor, welcher neben dem Klassennamen noch einen **boolean** Wert entgegen nimmt, zum Zug kommen. In diesem Fall werden all jene Methoden, welche von diesem Ausgangspunkt theoretisch erreichbar sind, ermittelt und schließlich in gleicher Art und Weise verarbeitet, wobei die besagte boolesche Variable darüber entscheidet, ob Pakete wie `java.*`, `sun.*`, `com.sun.*`, `org.apache.*` oder `org.xml.*` berücksichtigt werden. Es ist allerdings nicht möglich, diese Pakete bereits während der Erstellung des *Callgraphs*, welche generell sehr rechenintensiv ist, auszuschließen. Nachdem bereits kleine Beispielprogramme mit einfachen Aufrufen an das *Java API* einen stattlichen *Callgraph* haben können, setzt diese Methode, besonders für die repetitive Analyse eines Projekts, ein gewisses Maß an Geduld bzw. Rechenleistung voraus. Aus diesem Grund ist es möglich, dem Kommandozeileninterface (`bwa.Main`) gleichzeitig mehrere Klassen zu übergeben, welche durch wiederholten Aufruf des bereits erwähnten Konstruktors **public** `SootCom(String classname)` nacheinander abgearbeitet werden.

In jedem Fall müssen einige Optionen gesetzt werden, damit Zeilennummern und Variablennamen während der verschiedenen Transformationsschritte in *Soot* nicht gänzlich verloren gehen. Die Klasse `soot.options.Options` stellt zwar zum Wählen der so genannten *Phase Options* eine eigene Methode zur Verfügung, nicht aber was globale Einstellungen betrifft. Als Abhilfe ist es allerdings möglich, die gewünschten Optionen in einem `String[] args` abzulegen, als würde man sie `soot.Main` auf der Kommandozeile übergeben, um schließlich `Options.v().parse(args)` aufzurufen. Nach diversen weiteren Initialisierungsbefehlen kann schließlich damit begonnen werden, die Kontrollflussdaten abzufragen.

2.3 Units und Values

Soot besitzt mehrere Ebenen zur Repräsentation der Eingabeprogramme, welche in Abbildung 2.1 skizziert werden. Ganz oben in dieser Hierarchie steht `soot.Scene`, welche alle momentan geladenen Klassen organisiert. In diesem Zusammenhang ist vor allem der Aufruf `SootClass sClass = Scene.v().loadClassAndSupport(''Klassenname'');` wichtig, mit dem eine bestimmte Klasse der *Scene* hinzugefügt wird, wobei das korrespondierende *SootClass* Objekt praktischerweise gleich zurückgegeben wird. Dieses stellt die interne Repräsentation der zu bearbeitenden Klasse dar und bietet in erster Linie Zugriff auf Attribute, Methoden und organisatorische Informationen wie zum Beispiel Sichtbarkeit, Basisklasse oder Anzahl und Art der implementierten *Interfaces*. Analog dazu verhält sich eine *SootMethod*, wobei diese den zugehörigen Code nicht selbst enthält, sondern lediglich auf das

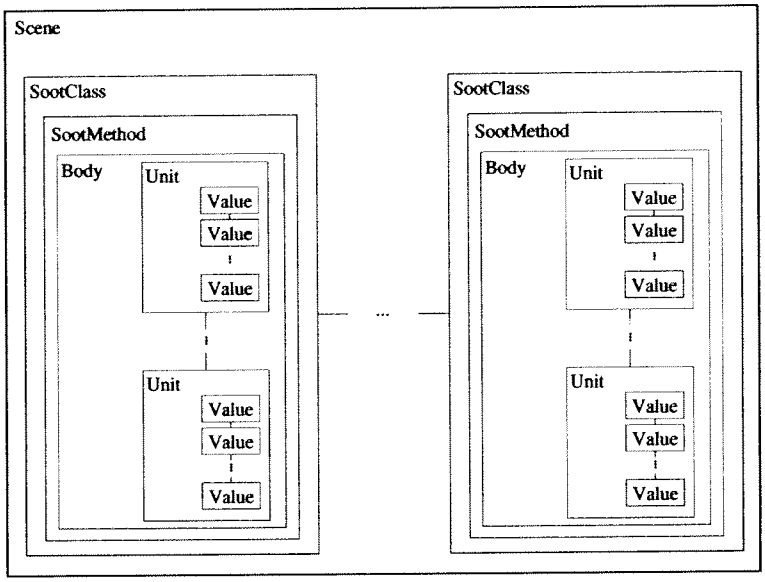


Abbildung 2.1: Repräsentation der Eingabeprogramme in Soot

entsprechende *Body* Objekt verweist, welches schließlich eine Liste von *Units* enthält. Eine *Unit* ist im Grunde die *Soot*-Repräsentation eines *Statements* bzw. einer *Instruction*, wobei in *Soot* 85 Implementierungen und 95 *Subinterfaces* zu `soot.Unit` existieren, welche leider in keiner Weise dokumentiert sind. Die meisten *Units* beinhalten wiederum verschiedene *Values*, wie lokale Variablen, Konstanten, *Expressions* und dergleichen, die wiederum aus *Values* bestehen können. So besteht beispielsweise die *Unit* `a = b + 1` aus den *Values* `a`, `b + 1`, `b` und `1`. Hier lässt sich bereits ein gewisses Problempotential erkennen, da `b + 1` zwar ein Value ist, der `b` und `1` enthält, diese allerdings auch schon für sich in der Liste der Values dieser Unit aufscheinen. Dieser und andere Umstände machen eine ausgeklügelte Auswertung von Units und Values nötig (siehe Abschnitt 2.3.2). Zwar gibt es zu `soot.Value` lediglich 47 *Subinterfaces* und 70 implementierende Klassen, die zugehörige Hierarchie gestaltet sich allerdings wesentlich flacher, weshalb für unsere Zwecke wesentlich mehr *Values* als *Units* berücksichtigt werden mussten. Oft wird von Soot an Stelle der eigentlichen Unit bzw. des eigentlichen Values eine *Box* (`soot.UnitBox` bzw. `soot.ValueBox`) geliefert, welche schließlich das gewünschte Objekt enthält. Mit *Boxes* wird also versucht *C/C++ Pointer* nachzuahmen [Lam00].

2.3.1 Auswertung von Units

Ziel ist es, die Namen der in einer Unit verwendeten Variablen und die Art des Zugriffs auf ebendiese in Erfahrung zu bringen. Zwar besitzen alle *Units* die Methoden `java.util.List getDefBoxes()` und `java.util.List getUseBoxes()`, welche die geschriebenen bzw. gelesenen Values retournieren, aber eine gesonderte Behandlung der verschiedenen Units ist dennoch nötig, und ist in `bwa.sootcom.UnitEval` implementiert.

Nachdem dem Konstruktor von UnitEval die gewünschte Unit übergeben wor-

den ist, wird in der Methode `private void evaluate()` zunächst mit Hilfe von `instanceof` überprüft um welche Art von Unit es sich hierbei handelt. Glücklicherweise kann hier auf einer relativ hohen Abstraktionsebene gearbeitet werden, weshalb lediglich 10 verschiedene Ausprägungen behandelt werden müssen.

Nach erfolgter Auswertung sind die Namen und Typen der linken, geschriebenen und der rechten, gelesenen Variablen, sowie der Typ der gesamten Unit als String getrennt verfügbar. Sollte die Unit direkt einer Zeile oder deren Anfang im Java Quelltext entsprechen, so ist, sofern sich Debug-Informationen im Bytecode befinden, auch die Zeilennummer verfügbar. Obwohl dies bei einem Schleifenkopf im Allgemeinen zutreffen sollte, wird im gegenteiligen Fall die Zeilennummer der nächstgelegenen Unit ausgegeben.

Die wohl häufigste Unit ist das *DefinitionStmt*, welches im Grunde eine Zuweisung darstellt. In der dem Bytecode nahen Zwischendarstellung, auf welcher Soot operiert, scheinen allerdings einige vereinfachende Einschränkungen gegenüber der üblichen hochsprachlichen Variante zu gelten, wobei es innerhalb dieses Projekts nicht nötig war, sich darauf zu stützen. Um ein *DefinitionStmt* auszuwerten, wird der Methode `private String[] extractValues(java.util.List boxes, boolean careLeft)` nacheinander die Liste der *DefBoxes* und der *UseBoxes* übergeben, um jeweils ein String Array mit Variablen zu erhalten, wobei dies im Falle der *DefBoxes* nur ein einzelnes Element enthalten kann, da auf der linken Seite einer Zuweisung nur genau eine Variable stehen darf. Der Boolean Wert `careLeft` ist notwendig geworden, da manche Values, die links stehen ein verändertes Vorgehen auf der rechten Seite erfordern (siehe Abschnitt 2.3.2).

Die einfachste Verzweigung, das *IfStmt*, ist zugleich auch in annähernd jedem Schleifenkopf zu finden, da Schleifen sowohl im Bytecode als auch in der von Soot verwendeten Darstellung auf bedingte Sprungstellen abgebildet sind. Der Hauptunterschied zum *DefinitionStmt* liegt im Fehlen der geschriebenen Variable, weshalb `public String getLeftVar()` immer den leeren String zurückliefert und sich die in der Bedingung vorkommenden Variablen nach der Evaluierung in dem von `getRightVars()` retournierten `String[]` befinden.

Die beiden Ausprägungen eines *Switch* Statements, *JTableSwitchStmt* und *JLookupSwitchStmt*, unterscheiden sich zwar lediglich im Opcode (*JTableSwitchStmt* für fortlaufende Schlüsselwerte) haben allerdings keine gemeinsame Basisklasse die spezifisch genug wäre, um diese stattdessen auszuwerten. Der mittels `ValueBox getKeyBox()` erhaltene Ausdruck wird wie üblich zur weiteren Zerlegung an `extractValues` weitergegeben.

Die Units *InvokeStmt*, *ReturnVoidStmt*, *ReturnStmt*, *GotoStmt*, *ThrowStmt* und *MonitorStmt* enthalten keine Variablen oder Bedingungen, weshalb sie ignoriert werden.

2.3.2 Auswertung von Values

Für jeden in der von `private void evaluate()` (Abschnitt 2.3.1) übergebenen Liste enthaltenen Value muss nun entschieden werden, ob und in welcher Form, er dem zu retournierenden String-Array hinzugefügt wird, wobei auch hier wieder mit Hilfe von `instanceof` die Zugehörigkeit zu einer bestimmten Klasse von Va-

lues geprüft wird. Im einfachsten zu behandelnden Fall ist der betreffende Value ein `soot.Local`, also eine lokale Variable, deren Name im Normalfall ohne weiteren Aufwand in das String Array aufgenommen wird, sofern dieser nicht bereits enthalten ist. (Für die Zwecke der *Busy Wait Analyse* ist es nicht notwendig, mehrfache Vorkommnisse derselben Variablen an der gleichen Stelle zu erfassen.) Zusammengesetzte Ausdrücke wie zum Beispiel `soot.jimple.BinopExpr`, also ein Ausdruck mit einem binären Operator, werden im Allgemeinen ignoriert, da die einzelnen Komponenten ohnehin in der Liste von Values enthalten sind, weshalb eigentlich kein *Local* unberücksichtigt gelassen werden dürfte. Dennoch kann es vorkommen, dass ein solcher *Local* unselbstständiger Teil eines bereits an anderer Stelle behandelten Values ist und deshalb ignoriert werden muss, wie z.B. im Falle der `soot.jimple.ArrayRef`. Eine *ArrayRef* ist eine Referenz auf ein bestimmtes Element eines *Arrays* der Form `base[index]`, wobei in der *Jimple* Darstellung an Stelle von `index` anscheinend nur eine lokale Variable, bzw. Konstante stehen darf.

Der Vergleich zweier solcher Referenzen ist im Rahmen einer statischen Analyse allerdigs nur eingeschränkt möglich, da `index` ausgewertet werden müsste, was im Allgemeinen nur zur Laufzeit möglich ist. Als beste Lösung im Rahmen dieser Arbeit schien es, eine *ArrayRef* direkt in einen String umzuwandeln. Zwar wird so die Tatsache ignoriert, dass `index` an unterschiedlichen Stellen verschiedene Zahlenwerte repräsentieren kann, die wesentlich größere Klasse an falsch erkannten *Busy Waits*, welche entsteht, würde man lediglich `base` speichern, wird allerdings vermieden.

Steht eine solche *ArrayRef* auf der rechten Seite einer Zuweisung, so werden *UseBoxes* mit `base`, `index` und `base[index]` in dieser Reihenfolge geliefert, weshalb die zuvor als *Local* erkannte `base` nachträglich entfernt werden muss. Der wesentlich unangenehmere Fall entsteht allerdings wenn dem Element eines Arrays ein Wert zugewiesen wird, also z.B. `base[index] = x`. Hier wird zuerst die Liste der *DefBoxes* ausgewertet, welche lediglich den Value `base[index]` enthält. Werden schließlich in einem separaten Aufruf von `extractValues` die *UseBoxes* mit `base`, `index` und `x` übergeben, so darf `base`, erkannt als lokale Variable, nicht aufgenommen werden.

Mit Hilfe von `extractValues` werden also über 20 verschiedene Klassen von Values erkannt und ihrer Bedeutung für die *Busy Wait Analyse* entsprechend aufbereitet.

2.4 Das CFG Objekt

Ziel der *SootCom* Schnittstelle ist es also, alle relevanten Kontrollflussdaten in einem Objekt abzulegen, welches dann dem eigentlichen Analysealgorithmus als Datenquelle dient. Die entsprechende Klassenbeschreibung findet sich in *CFG.java* (`bwa.data.CFG`) wobei hier von der allgemeinen *Graph* Klasse (`bwa.data.Graph`) geerbt wird. Diese beinhaltet abgesehen von der Knotenanzahl lediglich zwei dynamische Arrays welche für einen gegebenen Knoten die Nachfolger bzw. Vorgänger beinhalten und wird selbst nicht instantiiert. Zusätzlich zu diesen grundlegenden Informationen über den Kontrollflussgraphen enthält das *CFG* Objekt noch drei weitere Attribute welche von *SootCom* initialisiert werden.

Da die Kontrollflussgraphen methodenweise erzeugt werden, wird der vollständige Methodenname nach dem Schema `paket.Klasse.methode` abgelegt, um bei der späteren Ausgabe die entsprechende, unter Umständen überladene Methode zusammen mit der Zeilennummer eindeutig zu referenzieren. Weiters ist es nötig, die Nummer des Wurzelknotens zu speichern, da dies effizienter möglich ist, als einen festen Index zu garantieren. In `public Node[] nodes` findet sich schließlich Information bezüglich der in den Knoten enthaltenen Variablen (siehe Abschnitt 2.5).

2.4.1 Kontrollflussgraphen in Soot

Auch *Soot* verwendet zur Repräsentation des Kontrollflusses gerichtete Graphen, wobei hier unterschiedliche Ausprägungen zur Verfügung stehen. Während `soot.toolkits.graph.UnitGraph` für jede einzelne *Unit* einen eigenen Knoten enthält, werden diese in einem `soot.toolkits.graph.BlockGraph` soweit als möglich zusammengefasst, wodurch lange Ketten ohne Verzweigung vermieden werden. Da es auch in einem *BlockGraph* möglich ist, einzelne *Units* in einem Block anzusprechen, wurde ein solcher schließlich als Ausgangspunkt gewählt.

In *Soot* stehen vier verschiedene Varianten des *BlockGraphs* zur Auswahl, welche sich vor allem in der Behandlung von *Exceptions* unterscheiden. Nachdem es nicht zielführend schien, durch *Exceptions* beeinflusste Kontrollflussinformation in unsere Analyse aufzunehmen, da ein *Branch Predicate* ohne Variable nicht ausgewertet werden kann, fiel die Entscheidung zu Gunsten von `soot.toolkits.graph.BriefBlockGraph`, da eine Entfernung aller durch *Exceptions* verursachter Subgraphen hier am effizientesten möglich ist. Bei einem *BriefBlockGraph* handelt es sich allerdings nicht, wie in der *Soot* API Dokumentation behauptet, um einen Wald in dem jeder *Exception Handler* einen disjunkten Subgraphen induziert, sondern, wie in Abbildung 2.2 dargestellt, um einen zusammenhängenden Graphen, welcher für jeden *Exception Handler* einen zusätzlichen Wurzelknoten aufweist.

Bei der Konvertierung in ein *CFG* Objekt gilt es also jenen Wurzelknoten zu finden, welcher nicht durch eine *CaughtExceptionRef* (*Soot* Repräsentation eines **catch**) charakterisiert wird, und ausgehend von diesem, eine *Tiefensuche* zu starten, wobei dem weiteren Konvertierungsprozess ausschließlich besuchte Knoten zugeführt werden. In Abbildung 2.1 sind drei (grau hinterlegte) Wurzelknoten zu erkennen, von denen *Block 1* und *Block 5 Exception Handler* darstellen, welche bei *Block 6* wieder in den regulären Kontrollfluss münden. Ausgehend von der eigentlichen Wurzel (*Block 0*) wird also lediglich der umrandete Teilgraph berücksichtigt.

2.5 Die Node-Klasse

Das Array `nodes` schließlich, hält Informationen über die einzelnen Variablen eines jeden Knotens bereit, welcher auch aus mehreren Codezeilen bestehen kann. Für jede dieser Zeilen ist es nun möglich, mit `public String getLeft(int line)` bzw. `public String[] getRight(int line)` die Namen jener Variablen abzurufen, welche hier definiert bzw. gelesen werden, wobei die Gesamtzahl der Lines

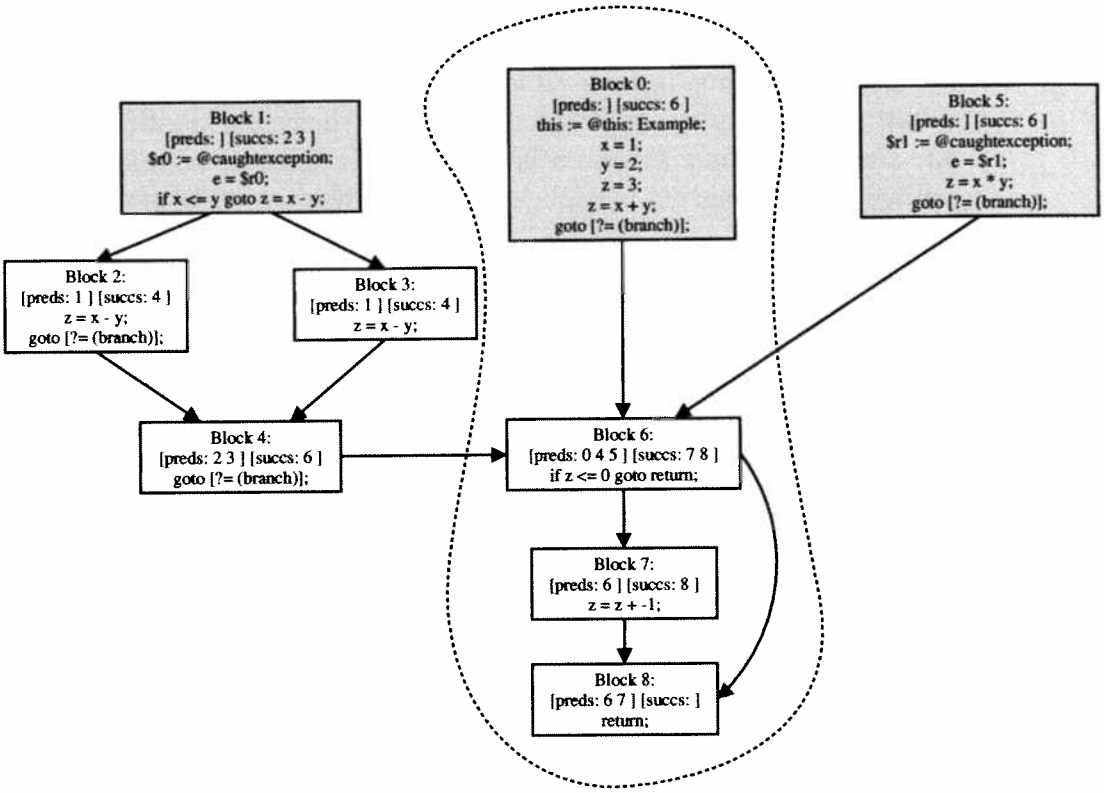


Abbildung 2.2: Ein *BriefBlockGraph* mit *Exception Handling*

Listing 2.1: Der Quelltext zu dem in Abbildung 2.2 gezeigten *BriefBlockGraph*

```
1  public void example(){
2      int x = 1;
3      int y = 2;
4      int z = 3;
5
6      try{
7          z = x + y;
8      }catch(Exception e){
9          if(x > y){
10             z = x - y;
11         }else{
12             z = x - y;
13         }
14     }catch(Error e){
15         z = x * y;
16     }
17
18     if(z > 0){
19         z--;
20     }
21 }
```

Listing 2.2: Ausgabe eines Nodes

```
1 DefinitionStmt: $i0 = bar.<Foo: int y>
2
3 left:
4       $i0 <-- Local
5 right:
6       bar.<Foo: int y> <-- InstanceFieldRef
```

mit `public int size()` ermittelt werden kann. Um eine aussagekräftige Ausgabe erzeugen zu können, ist außerdem die Zeilennummer und der Soot Typ einer Variablen verfügbar. Des weiteren kann die textuelle Repräsentation der Unit, welcher das Node Objekt zu Grunde liegt, und deren Typ abgefragt werden, um die in Abschnitt 2.3 beschriebene Auswertung zu verfolgen.

Listing 2.2 zeigt einen kleinen, exemplarischen Ausschnitt der mit Option `-vv` erzeugbaren Ausgabe einer Node. Die Unit `$i0 = bar.<Foo: int y>` ist demnach vom Typ *DefinitionStmt*, wobei der lokalen Variable `$i0` der Wert einer *InstanceFieldRef* zugewiesen wird. Eine *InstanceFieldRef* ist eine Referenz auf ein (nicht notwendigerweise fremdes) Attribut, in diesem Fall also auf die Integer-Variable `y` des Objekts `bar`, welches eine Instanz der Klasse `Foo` ist.

An dieser Stelle soll darauf hingewiesen werden, dass Variablen, deren Namen mit einem Dollar-Zeichen beginnen, ihren Ursprung in der Regel nicht im Sourcecode haben, sondern angelegt werden, um die in der Soot Darstellung (bzw. dem Bytecode) nicht erlaubten, komplexen Statements in einfachere aufzuteilen. So wird hier die *InstanceFieldRef* zuerst aufgelöst und ihr Wert in einer lokalen Variable abgelegt, bevor diese schließlich verwendet wird. Aus diesem Grund hat die in [BBS03] vorgeschlagene Erweiterung zum Erkennen von indirekten *Busy Wait Variablen* (siehe auch Abschnitt 3.2.3) besondere Bedeutung für Referenzen aller Art.

2.6 Ein Beispiel

Die in Listing 1.2 angeführte Implementierung des Algorithmus von Dekker wird von `soot.tools.CFGViewer`, wie in Abbildung 2.3 gezeigt, ausgegeben und ist somit eine grobe Darstellung des Inputs. Die SootCom-Schnittstelle interpretiert diesen nun wie in Abbildung 2.4 angedeutet, und stellt das entstandene CFG-Objekt dem Analysealgorithmus zur Verfügung.

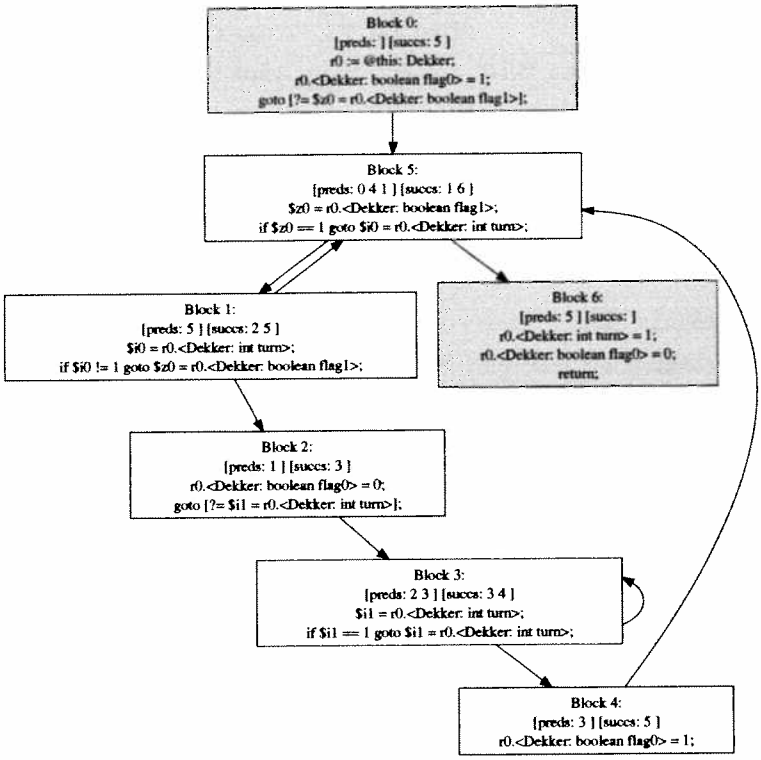


Abbildung 2.3: Ausgabe von `soot.tools.CFGViewer`

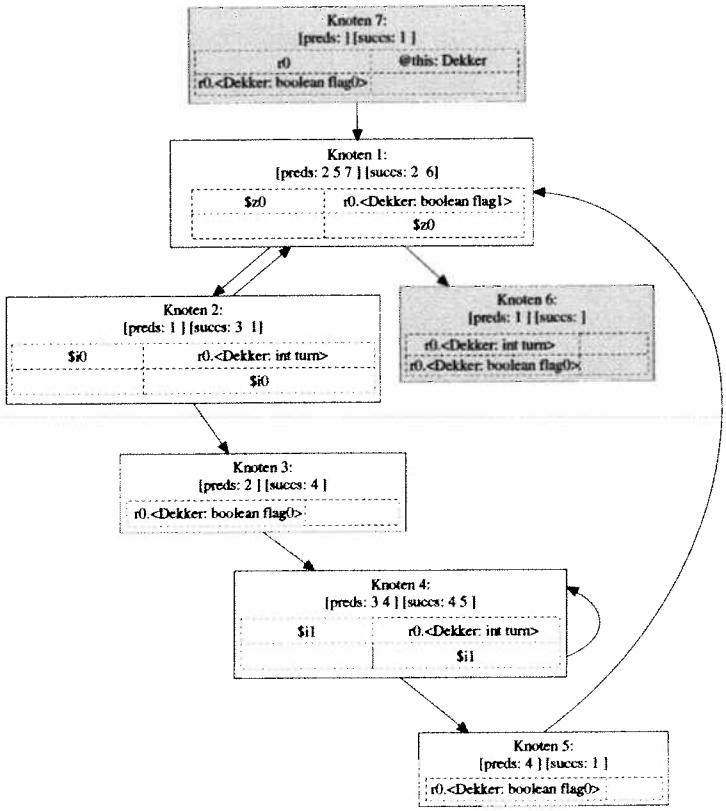


Abbildung 2.4: Interpretation durch SootCom

3 Die Analyse

Der Algorithmus wurde schrittweise, parallel mit der *SootCom* Schnittstelle entwickelt. Zuerst in einer groben Version geschrieben und geschliffen im Laufe der Testphasen, ist er einerseits eine genaue Implementierung des von *Blieberger et al.* [BBS03] vorgeschlagenen *Busy-Wait-Analysis*-Verfahrens, andererseits eine leichte Abweichung, die dem *BWA-Tool* eine in der Praxis größere Flexibilität gewährleisten soll.

3.1 Erkenntnisse

Im Laufe unserer Bakkalaureatsarbeit sind wir oft auf Fakten und Feinheiten gestoßen, die eine neue Überlegung der Vorgehensweise notwendig gemacht haben. Wegen Grenzfällen im algorithmischen Sinne, oder Limitierungen der Soot Schnittstelle, wurde nach Verbesserungen und neuen Wegen gesucht. Im Folgenden finden sich die wichtigsten Eckpunkte dieser.

3.1.1 Backedges und Dominatoren

Definition 1: Eine Kante (u,v) nennt man *Backedge*, wenn ihr Zielknoten v, den Quellknoten u dominiert.

Definition 2: Wir führen den Begriff *Backnode* für den Quellknoten einer *Backedge* ein.

Definition 3: Wir definieren den *Schleifenrumpf* einer Schleife mit dem Header h, im Graphen G(V,E) als:

$$\mathcal{R}(h) =_{defs} \{u \in \pi : \pi =< h, .., h >, u \in \mathrm{V}\}$$

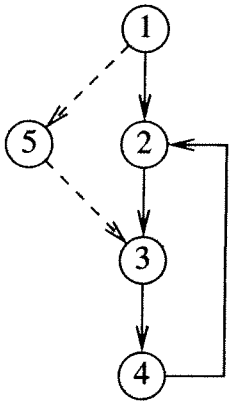


Abbildung 3.1: Loop mit *Alien Edge*

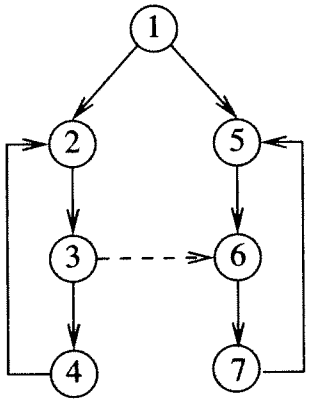


Abbildung 3.2: Edge Loop Crossover

Man kann den Algorithmus von *Blieberger et al.* verbessern, indem für die Berechnung der Backedges, anstatt der vorgeschlagenen Vorgangsweise - wo diese aus den *Dominatorbeziehungen* hergeleitet werden - einfach eine *DFS Suche* einsetzt. In *Reducible CFGs* sind jene Kanten Backedges, die Kreise schließen. Unsere wichtigste Annahme ist jedoch, wie bereits oben erwähnt, dass wir mit Reducible CFGs arbeiten.
Anmerkung: Wir verwenden im Folgenden den Namen *Alien Edge* für eine Kante (u,v) für die, wie in [HU72] beschrieben, für eine Schleife $\mathcal{R}(h)$ gilt, dass $u \notin \mathcal{R}(h)$ und $v \in \mathcal{R}(h)$. Wie von Hecht und Ullman [HU72] bewiesen, kann ein Reducible CFG keine *Alien Edges* enthalten.

3.1.2 Edge Loop Crossover

Zusammen mit der Dominatorbeziehung haben wir auch die Problematik eines *Edge Loop Crossovers* analysiert, die, wenn sie möglich gewesen wäre, einen großen Teil unserer Arbeit, in negativer Weise beeinflußt hätte. Ein solcher Fall (siehe Abbildung 3.2) ist aber nicht möglich, da wenn eine Kante ein Loop Crossover macht, gilt dann für zumindest eine der existierenden Backedges nicht mehr, dass ihr Zielknoten den Quellknoten dominiert. Somit kann sie laut der Definition eines Reducible CFGs nicht mehr entfernt werden, und die Schleife bleibt erhalten. Dadurch wäre der CFG nicht reduzibel, und auch nicht interessant für uns. **Anmerkung:** Eine solche Kante würde auch der Definition einer Alien Edge entsprechen.

3.1.3 Union Find

Ramalingam [Ram99] stellt einen nahezu linearen Algorithmus, zur Berechnung des Loop Forests vor. Er behauptet, man könnte mit dem Algorithmus von Tarjan eine Laufzeit von $\Theta(n \cdot \alpha(n,m))$ erzielen, wenn man die Standardalgorithmen für *Union Find* mit *Path Compression* und *Union by Rank* verwenden würde. Tarjan verwendet *find* Abfragen um herauszufinden, in welcher Schleife ein Knoten sich befindet. Genauer, welchen Schleifenkopf diese Schleife hat. So ist zum Beispiel find$(u)=v$, wenn der Knoten u sich in der Schleife mit dem Header v

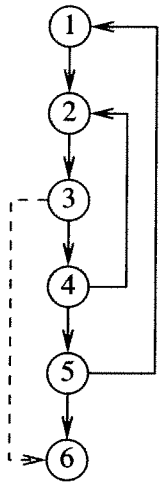


Abbildung 3.3: Multi Level Break

befindet. Somit sind die *union* Aufrufe immer gerichtet. union(u,v) wird verwendet, um den Knoten u auf den Knoten v zusammenzulegen. Da es in Union by Rank aber vorkommen kann (und das ist auch der Sinn des Verfahrens), dass im Aufruf union(u,v) auch umgekehrt zusammengelegt wird, sprich v auf u, was also die Berechnungen verfälschen würde, kann hier *Union by Rank* nicht verwendet werden.

3.1.4 Multi Level Breaks

Abbildung 3.3 ist ein Beispiel eines korrekten CFGs, der aber vom Algorithmus falsch bearbeitet wird. Das liegt daran, dass es sich hier um ein *Multi Level Break* handelt. Ein Break findet in der inneren Schleife statt (am Knoten 3), das aber über zwei Ebenen hinweggeht: aus der inneren Schleife bis ganz nach außen. Die falsche Behandlung finden wir bereits auf einer logischen Ebene. Als Knoten einer Schleife werden nur die Header, *Level 0 Nodes* (die unmittelbar in der Schleife stehenden Knoten) und die *Level 0 Header* (die Header der genau eine Ebene darunter liegenden Schleifen) betrachtet. Zum Beispiel haben wir in 3.3 als "echte" Knoten der äußeren Schleife nur 1, 2 und 5. 3 und 4 sind Knoten der Unterschleife. Bei einem Multi Level Break, wie hier, ist aber der Knoten 3 auch für die äußere Schleife ein Termination Statement. Die Kante (3,6) verlässt alle Schleifen.

3.1.5 "Tuning"

Die Implementierung setzt eine kleine Abweichung vom Standardalgorithmus, zwecks einer leichten Beschleunigung ein. *Blieberger et al.* schlagen einen 2-Schritt Algorithmus zum Überprüfen der Busy Wait Eigenschaft einer Variable vor. Zuerst wird ein Subgraph erzeugt, der die Schleife mit der Variable enthält und nur aus *Non Writing Nodes* (Knoten, die die Variable nicht schreiben) besteht, und dann wird eine Erreichbarkeitssuche darauf angewendet. Die Komplexität eines solchen Verfahrens ist etwas größer als unbedingt notwendig. Es sei $G'(V'$,

Listing 3.1: Variable Block-intern vor der Entscheidung beschreiben

```
1 while (++i<10) {
2     // etwas
3 }
```

E') der Subgraph, der die entsprechende Schleife enthält. Unabhängig vom verwendeten Verfahren, wird ein $\Omega(|V'| + |E'|)$ Aufwand für das Generieren dieses Untergraphens benötigt, da alle Kanten und Knoten zumindest einmal betrachtet werden müssen. Ein schnelles Erreichbarkeitsverfahren in diesem Subgraphen ist in $O(|V'| + |E'|)$ möglich und der Gesamtaufwand ist somit die Summe der beiden. Wir entscheiden uns, hier nur das Erreichbarkeitsverfahren direkt auf dem vollen Graphen auszuführen. Zu diesem Zweck verwenden wir eine *Backwards BFS* auf dem richtigen Loop. Diese inverse Breitensuche verwendet das `pred` Array (mit den Vorgängern eines Knotens) anstelle des üblichen `succ` Arrays (mit den Nachfolgern). Die Suche beginnt mit den Backnodes der Schleife, dann propagiert sie sich zurück, also hinauf bis zum Headerknoten. Die Suche hat zwei Eigenschaften: *Termination* nach endlich vielen Schritten, und *Bereichtreue*. Die erstere ergibt sich aus dem Einsatz eines Markierungsarrays, in dem ein Knoten als "markiert" gespeichert wird, wenn er bei der Suche bereits besucht worden ist. Man kann also jeden Knoten maximal einmal betrachten. Die Suche ist bereichtreu, i.e. sie verlässt die aktuelle Schleife nicht, weil (siehe Abschnitt 3.1.1) der Schleifenkopf alle Mitglieder der Schleife dominiert, und ein Rückweg zum Wurzelknoten dann unbedingt den Header passieren müsste. Angenommen wir überprüfen gerade die Busy Wait Eigenschaft der variable *var*. Damit die Bedingung hält, muss es einen *Non Writing Path* von den Backnodes zurück zum Schleifenkopf geben. Um einen Subgraphen, bestehend nur aus den *Non Writing Nodes*, zu simulieren, reicht es, die folgende Bedingung in den Algorithmus einzubauen: Kommt man im Laufe der Suche bei einem Knoten an, der die Variable *var* schreibt, so muss dieser Knoten ignoriert werden. Man kann die Gültigkeit dieser Annahme beweisen, indem man beobachtet, dass im Laufe der Rückwärts-BFS, nur die Kanten jener Knoten betrachtet werden, die in der Standard BFS Queue hinzugefügt worden sind. Ignoriert man einen Knoten, so existieren aus Sicht der BFS auch seine Kanten nicht. Aus der Bereichtreue der Suche ergibt sich eine $O(|\mathcal{R}(h)| + |E_{\mathcal{R}(h)}| - |E^W_{\mathcal{R}(h),var}|)$ Komplexität für das Verfahren. $\mathcal{R}(h)$ ist die Menge der Knoten der Schleife. $E_{\mathcal{R}(h)}$ ist die Menge der zu den Knoten der Schleife inzidenten Kanten. $E^W_{\mathcal{R}(h),var}$ ist die Menge der inzidenten Kanten für die Knoten der Schleife, die die Variable *var* schreiben. Aus rein theoretischer Sicht ist eine Erreichbarkeitsuntersuchung hinreichend für das Beweisen oder Widerlegen der Busy-Wait-Eigenschaft einer Variable. Eine nähere Betrachtung der von Soot zur Verfügung gestellten Datenstrukturen erweist jedoch in der Praxis die obige Überlegung als nicht zufriedenstellend. In manchen Fällen (siehe Abbildung 3.4 und Listing 3.1) kann es dazukommen, dass Zuweisungen vor der Entscheidungszeile stehen. Diese müssen auch berücksichtigt werden. Deswegen, sogar bevor die tatsächliche Erreichbarkeitsuntersuchung überhaupt gestartet wird, muss man überprüfen, ob der Schleifenkopf selber ein Non-Writing-Node ist.

Wenn man durch Anwendung dieses Verfahrens den Headerknoten erreicht,

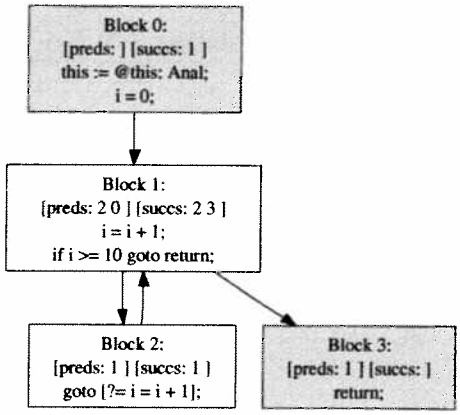


Abbildung 3.4: Variable Block-intern vor der Entscheidung beschreiben

Listing 3.2: Rekursiver Knoten

```
1 do {
2     i = i + 1;
3 } while (flag);
```

existiert ein Pfad der nur aus Non Writing Nodes besteht, und die Variable *var* ist eine Busy Wait Variable. Die zu einem späteren Zeitpunkt beschriebene `isBVW` Methode ist die exakte Umsetzung des hier geschilderten Verfahrens.

3.1.6 Rekursive Knoten

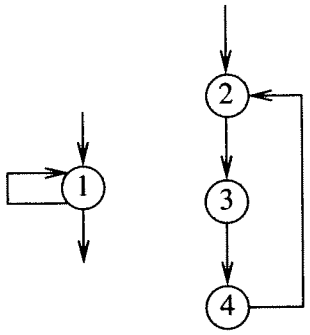
Ein erster Spezialfall der vom Busy Wait Analyser, in seiner "rohen" Implementierung nicht erkannt wurde, war jener eines rekursiven Knotens. Soot wird in dem Code aus dem Listing 3.2 die ganze Schleife als einen einzigen rekursiven Knoten darstellen. (Abbildung 3.5 a.) Diese Schleife wird vom Algorithmus von Tarjan nicht erkannt und somit auch nicht von unserem Algorithmus in Betracht gezogen. Da die Erkennung solcher Schleifen sehr leicht ist - man muss im CFG nur nach Kanten der Form (u,u) suchen, haben wir eine *Pre-Scan* Phase vor der Implementierung des Algorithmus vom *Blieberger et al.* eingebaut, die solche Schleifen aufspürt und analysiert. Für Implementierungsdetails siehe Abschnitt 3.2.4. Das Codebeispiel mag hier nicht sinnvoll sein, aber es ist nur ein Vertreter seiner Klasse. Soot spaltet Knoten nur im Fall von erzwungenen Verzweigungen auf (z.B. **`if`** Statements). Eine **`do..while`** Schleife, die nur aus einfachen Ausdrücken besteht (Zuweisungen oder Methodenaufrufen) wird nicht gespalten, stattdessen wird möglicherweise ein Knoten mit vielen Lines generiert.

3.1.7 No-T Loops

Eine anderer heikler Fall ist der Einsatz von *No-T Loops* (Listing 3.3; Abbildung 3.5 b.), Schleifen die keine Termination Statements haben. Weder enthält der Schleifenkopf eine Abbruchbedingung, noch existieren in der Schleife **`break`** Statements. Eine solche Konstellation kann eintreten, wenn ein Thread aktiv war-

Listing 3.3: No-T Loop

```
1 while (true) {
2     if (bedingung) {
3         doTerminate();
4     }
5 }
```



a) b)

Abbildung 3.5: Rekursive Knoten und No-T Loops

tet, eine Bedingung innerhalb der Schleife wiederholt abfragt, und danach eine Methode aufruft die den Thread terminiert. Da interprozedurale Aufrufe nicht verfolgt werden, kann auch nicht festgestellt werden, wie sich eine Methode auf den Control Flow des Programms auswirkt, sprich, in diesem Fall, ob der Aufruf tatsächlich den Thread terminiert oder nicht. Selbst wenn eine solche Schleife von Tarjans Algorithmus richtig erkannt wird, werden in den nächsten Phasen keine Termination Statemens gefunden, und somit auch keine Candidates. Ohne Candidates wird auch keinerlei Überprüfung durchgeführt, und die Schleife bleibt unentdeckt.

Da diese rekursiven Knoten und die no-T Loops eine gesonderte Behandlung brauchen, wurde eine solche extra vor der von *Blieberger et al.* hinzugefügt. Es wird darauf hingewiesen, dass es auch gemischte Schleifen gibt, i.e. ein rekursiver Knoten ohne Termination Statement.

3.1.8 "Schwellen"-variablen (Lite Scanning)

Besonders problematisch sind die Schwellenvariablen - hauptsächlich in den zählerartigen Konstrukten (siehe Listing 3.4 und Abbildung 3.6). Ziehen wir die Variable n in Betracht. Sie kommt im Entscheidungsblock vor, wird aber in der Schleife nicht verändert (was auch sinnentsprechend ist). Somit ist sie einerseits ein Candidate, andererseits ist sie laut Algorithmus eine Busy Wait Variable. Da eine hundertprozentige Korrektheit nur mit erheblichem Aufwand möglich ist, man denke z.B. nur an die Interprozeduralität, hat der Algorithmus keinen Anspruch auf eine solche Korrektheit, und diese falschen BWVs sind aus theoretischer Sicht akzeptabel. Anders sieht die Lage aus, wenn man das BWA Tool aus Sicht eines künftigen Benutzers betrachtet. In einem großen Projekt kommen

Listing 3.4: Einfaches For Loop

```
1 for (i=0; i<n; i++) {
2     // etwas
3 }
```

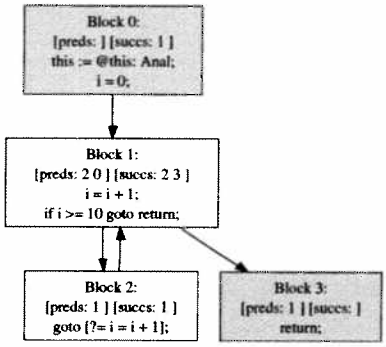


Abbildung 3.6: For Loop

solche Schleifen nur zu oft vor, was sich schließlich darin widerspiegelt, dass der User mit einer Unzahl an falschen Alarmen bombardiert wird. Mann kann aber eine besondere Eigenschaft der "echten" BWVs im Vergleich zu den "falschen" bemerken: Das *Busy Wait* Konzept macht nur dann einen Sinn, wenn die Schleife nur mit Einfluss "von außen" terminieren kann. Anders ausgedrückt, solange zumindest einer der Candidates eines Termination Knotens sich verändert, ist die Wahrscheinlichkeit, dass es sich dort tatsächlich um ein Busy Wait handelt nur sehr gering. Wir haben uns entschieden, dem Endbenutzer per Aufrufparameter zu ermöglichen, das *Lite Scanning* ein- oder auszuschalten. Im Lite Scan Modus werden Termination Statements, bei denen zumindest einer der Candidates kein BW-Suspect ist, ausgelassen. Somit bleiben nur mehr die Termination Statements übrig, deren Variablen alle BW-Suspects sind. Die Schleifen, die solche Knoten enthalten, haben die größte Wahrscheinlichkeit BW-Loops zu sein.

3.1.9 Source Aware Backpath Finding

Man betrachte den Code im Listing 3.5 mit dem CFG in der Abbildung 3.7. Die Variable `flag2` ist ein BW-Candidate, da sie in dem Termination-Statement/Knoten Nummer 3 vorkommt. Die Definition einer BW Variable ist laut *Blieberger et al.*:

$$\text{busy-var}(L_{(m,n)}, var) =_{defs} \exists \pi =< n, .., n >\in L_{(m,n)} : \neg \text{defined}_{var}(\pi)$$

Mit anderen Worten muss es einen nichtschreibenden Pfad vom Schleifenkopf ausgehend, und zurück zum Schleifenkopf geben. Hier existiert ein solcher Kreis: $1 \rightarrow 2 \rightarrow 5 \rightarrow 1$. Dieser Pfad verwendet aber die genannte Variable nicht, sprich: Sie muss auch nicht verändert werden, um ein Busy Wait zu vermeiden. Intuitiv wäre es wiederum sinnvoller, wenn es einen nichtschreibenden Pfad von dem Knoten ausgehend gäbe, der die Variable in einer Terminationsverzweigung einsetzt.

Listing 3.5: Source Aware Backpath Finding

```
1 while (flag0 == 0) {
2     if (flag1 == 0) {
3         flag2++;
4         if (flag2 > 10)
5             break;
6     }
7     else
8         flag1++;
9 }
```

Hier wäre $3 \rightarrow 1 \rightarrow 2 \rightarrow 3$ ein solcher. Über diesen Pfad muss die Schleife iterieren können, ohne dass die Verzweigungsvariable ihren Wert verändert, damit es sich um eine echte Busy Wait Variable handelt. Das ist aber nicht der Fall, da `flag2` sogar unmittelbar vor der Verzweigung beschrieben wird. Die obige Definition ist gültig, wenn die Candidates aus dem Schleifenkopf stammen. Für alle anderen Candidates kann es zu solchen Fällen kommen, wo sie fälschlicherweise als BWVs erkannt werden, selbst wenn sie es nicht sind. Wir erweitern daher diese Definition um ein beliebiges Termination Statement, verschieden von dem Loop Header:

$$\text{busy-var}(L_{(m,n)}, var) =_{defs} \exists \pi =< x, .., x >\in L_{(m,n)} : \neg\text{defined}_{var}(\pi) \wedge var \in \text{BP}(x)$$

$\text{BP}(x)$ ist das Branch Predicate des Knotens x.

3.2 Die Implementierung

Wir werden zuerst einen Blick auf die genaue Implementierung des vorher erwähnten Algorithmus werfen. Fortlaufend erworbene Kenntnisse haben dazu geführt, dass ganze Teile des Programms eingeführt, umgeschrieben oder gar weggelassen werden mussten.

3.2.1 Datenstrukturen

Der Kommunikation zwischen dem Analyseteil der Implementierung und der *SootCom* Schnittstelle liegen die Datenstrukturen im Unterverzeichnis `bwa/data/` zu Grunde. Nachdem das Projekt, wegen Kompatibilitätsgründen, in Java geschrieben werden musste, haben wir uns entschieden, feste Datenstrukturen, wie Arrays anstatt Java-Vektoren, zu verwenden. Dies mag in einem gewissen Sinne die Flexibilität des Codes beeinträchtigen, vermeidet aber einen riesigen Overhead, der beim Ausführen des Algorithmus, durch die in den Schleifen wiederholten Methodenaufrufe entstehen würde. Eben bei solchen großen Analysen darf dieser Overhead nicht außer Acht gelassen werden.

Die oberste Datenstruktur ist `Graph`. Sie stellt einen Graphen dar, und enthält zwei Arrays von Arrays, jeweils mit den Vorgängern (`pred`) bzw. Nachfolgern (`succ`) jedes Knotens. Die Nummer `n` gibt an wie viele Knoten im Graph enthalten sind. Als Faustregel für die Verwendung der Datenstrukturen, intern in

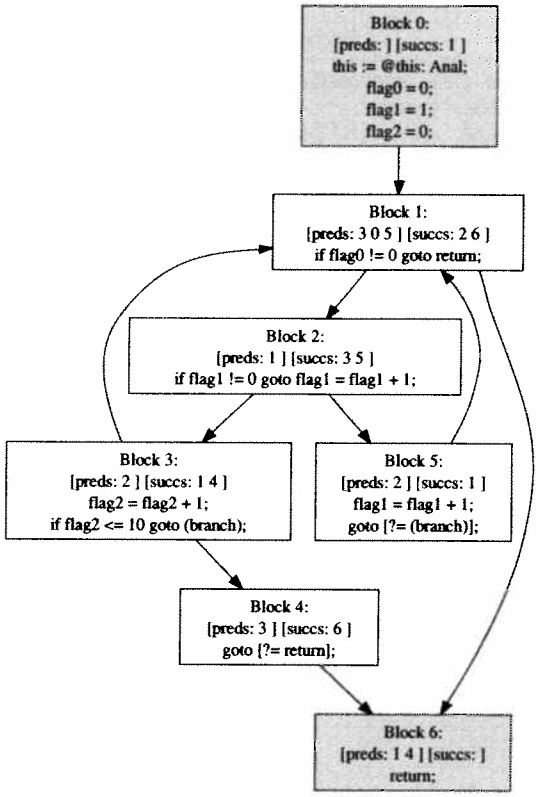


Abbildung 3.7: Source Aware Backpath Finding

Listing 3.6: Die Graph Datenstruktur

```
1 public class Graph {
2     public int[][] succ;
3     public int[][] pred;
4     public int n;
5 }
```

dem BWA Projekt gilt, dass ein Array immer auf Stelle `0` die Anzahl der in sich gespeicherten Elemente hält, und diese Elemente sind dann ab Stelle `1` bis einschließlich `Array[0]` befinden. `Graph.succ[k][0]` z.B. ist die Anzahl der Nachfolger des Knotens `k`, bzw. `Graph.succ[k][1]` ist der `1`-te Nachfolger des Knotens `k`. Analog werden auch die Vorgänger eines Knotens im Array `pred` gespeichert. Die Entscheidung für dieses eher ungünstige Speicherplatzmanagement wird von den oft wiederholten und abwechselnden Abfragen der Vorgänger und Nachfolger eines Knotens untermauert. Der dadurch, im Falle eines großen Graphens, entstehende Overhead rechtfertigt den kleinen Speicherplatzgewinn nicht.

Die `CFG`-Datenstruktur baut auf `Graph` auf und erweitert sie um speziell für die Darstellung eines *Control Flow Graphs* notwendigen Strukturen. `root` gibt an welcher Knoten die Wurzel des CFGs ist. `dfsNum` speichert für jeden Knoten die Tiefensuchenummer. `dfsNumInv` ist die Umkehrung des `dfsNum` Arrays, und gibt an welcher Knoten einer DFS Nummer entspricht. Diese ist eine spezielle Erweiterung der ursprünglichen Struktur. Als wir im Laufe der Programmierphase darauf gekommen sind, dass wir sie brauchen, haben wir uns entschieden, sie wie

Listing 3.7: Die CFG-Datenstruktur

```
1  public class CFG extends Graph {
2      public int root;
3      public int dfsNum[];
4      public int dfsNumInv[];
5      public int bfsNum[];
6      public int bfsBackArray[];
7      public int backEdge[][];
8      public String methodName;
9      public Node[] nodes;
10
11     ...
```

Listing 3.8: Die Loop-Datenstruktur

```
1  public class Loop {
2      public int id;
3      public int n;
4      public int header;
5      public int[] nodes;
6      public int[] backNodes;
7
8      ...
```

bei dem `pred` Array, ganz zu speichern, da wenn sie zusammen mit der DFS Suche berechnet wird, nur ein $\Theta(n)$ Aufwand, verteilt jeweils mit $\Theta(1)$ auf `n` Knoten verursacht wird. Eine nachträgliche Berechnung - erst wenn die Daten benötigt werden - würde jedes Mal einen $\Theta(n)$ Aufwand erzwingen. `bfsNum` hält, ähnlich wie die `dfsNum` die Breitensuchenummer. `bfsBackArray` enthält die Knoten des Baumes, sortiert in topologischer Reihenfolge. Dieses Array entspricht der Queue Struktur, die die BFS verwendet (die Knoten werden somit automatisch in der entsprechenden Reihenfolge eingetragen), was eine solche Sortierung zu einem späteren Zeitpunkt erspart. `backEdge` enthält die Backedges des CFGs, der Form (`backEdge[i][0]` $\rightarrow$ `backEdge[i][1]`). `methodName` ist der Name der Funktion die der CFG abbildet, und `nodes` sind die *Node* Strukturen, die von *SootCom* erzeugt werden.

Die `Loop`-Datenstruktur repräsentiert eine Schleife. `id` ist die Schleifen-ID, und identifiziert überall im BWA-Projekt eine Schleife eindeutig. `n` ist die Anzahl der Knoten, die in der Schleife enthalten sind, abgesehen von dem Schleifenkopf, und mit den Unterschleifen komprimiert auf den jeweiligen Header. z.B. Eine Schleife, die die Knoten 2, 3 und 4 enthält, mit der Backedge (4,2) wird nur mehr durch ihren Schleifenkopf angesprochen. Wenn also eine größere Schleife 1, 2, 3, 4 und 5, mit der Backedge (5,1), die obere enthält, so werden in der Loopstruktur als `n` nur 3 Knoten gezählt. (1, 2 und 5). Das dient der Schleifenhierarchiebildung. `header`, wie der Name sagt, ist der Schleifenkopf. `nodes` sind die Schleifenknoten (nach dem oben präsentierten *Loop Collapsing* Prinzip, gespeichert mit den jeweiligen IDs die auch in der `Graph` Struktur gelten. `backNodes` sind die Backnodes - die Quellknoten aller Backedges dieser Schleife.

Die `Forest` Klasse wird für das Speichern aller im CFG gefundenen Loops verwendet. Somit ist `loopn` ihre Anzahl. Die `loopf`-CFG-Datenstruktur enthält den Wald dieser Loops. Sie ist ein nichtzusammenhängender Graph, konstruiert an-

Listing 3.9: Die Forest Datenstruktur

```
1 public class Forest {
2     public int loopn;
3     public CFG loopf;
4     public int[] root;
5     public int[] loopParent;
6     public int[] loopId;
7     public Loop[] loops;
8
9     ...
```

hand der LoopIDs, der alle Schleifenbäume (*Loop Trees*) enthält. Er wird später für die bottom-up Analyse verwendet. Das Array `root` speichert für jede Schleife die Wurzel des Baumes, in dem sie sich befindet. `loopParent` und `loopID` enthalten für jeden Knoten den Kopf der Schleife, bzw. die ID der Schleife, in der er sich befindet. `loops` sind, wie der Name schon sagt, die Schleifen selbst.

Die `UnionFind` ist die übliche Union-Find Datenstruktur mit *Path Compression* und den Standardoperationen *add, find* und *union*. Sie wird vom Loop Forest Finding Algorithmus von Tarjan verwendet.

3.2.2 Subalgorithmen

Zunächst werden wir die `bwa/tools/` Algorithmen betrachten. Sie werden vom Hauptalgorithmus verwendet.

Die `BFS` Klasse stellt breitensucheverwandte Algorithmen zur Verfügung. `getBFSNums` berechnet nach dem *Breadth First Search* Prinzip die Breitensuchenummer jedes Knotens (ab einer vorgegebenen Wurzel), und überschreibt anschließend das `bfsBackArray` mit der von dem Algorithmus verwendeten Queue. Somit sind die Knoten in dem benannten Array automatisch topologisch sortiert. `getBFSIncomplete` unterscheidet sich von der ersteren nur darin, dass die gerade verwendete Queue `bfsBackArray` nicht überschreibt, sondern am Ende dessen angehängt wird. Das bewirkt, dass nach wiederholter Verwendung der `getBFSIncomplete` Methode das `bfsBackArray` des CFGs eine Reihe von topologisch sortierten Bäumen enthält. Die letztere Methode eignet sich für die Anwendung auf Wäldern.

`DFS` stellt eine einzige Mehrzweckmethode, `getBackEdges`, zur Verfügung. Sie führt eine Tiefensuche auf dem CFG aus, und markiert gleichzeitig auch die Backedges. Solche werden gefunden, indem im Laufe der Suche jedem Knoten eine "Farbe" zugewiesen wird. Ist ein Knoten noch nie besucht worden, so ist er weiß. Wird ein Knoten gerade verwendet, i.e. die Tiefensuche besucht gerade seine Kinder, so ist er grau, bzw. wurde ein Knoten und all seine Nachfolger bereits besucht, so ist er schwarz. Eine *Backedge* ist eine grau-grau Kante, i.e. sie "kehrt zurück" zu einem Knoten, der gerade verwendet wird (der sich also im Pfad von der Wurzel zum aktuellen Knoten befindet). Für einen Beweis der Korrektheit des Algorithmus siehe Abschnitt 3.1.1.

Die `Tarjan` Klasse berechnet den *Loop Forest* (`getLoopForest`) nach dem in [Ram99] vorgeführten Algorithmus von R.E.Tarjan. Zuerst werden für den CFG die DFS und BFS Nummern ermittelt. Im Prinzip ist der Algorithmus von

Tarjan ein *Backwards BFS* der, von einem potentiellen Loop Header ausgehend, seine Backedges speichert und dann anhand dieser die Kanten rückwärts verfolgt bis keine neuen Knoten gefunden werden, unter Berücksichtigung der Bedingung, dass aus der Schleife nicht hinaus "gesprungen" wird, sprich bis wieder der potentielle Loop Header erreicht wird. Dieses Verfahren haben wir später wieder für die Untersuchung der *Busy Wait Eigenschaft* der Variablen verwendet. Während der Erkennungsphase einer Schleife werden ihre Knoten auf den Header komprimiert, und sie wird nachher lediglich durch diesen angesprochen. In dieser *Collapse* Phase werden auch die entsprechenden `Loop` Datenstrukturen erzeugt. Am Ende dieses Subalgorithmus haben wir dann eine Liste aller vorhandenen Schleifen, mit den entsprechenden Informationen (ID, Header, Anzahl der enthaltenen Knoten und anschließend die Knoten selbst) in einem Array von Loop-Strukturen gespeichert, die weiter in einer Forest-Datenstruktur eingepackt ist. Was uns aber fehlt, sind die Beziehungen unter den einzelnen Schleifen, und wie sie in Bäumen organisiert sind. Die *Loop Gathering* Phase löst genau dieses Problem. Sie betrachtet nun das Loop Array. Um sorgfältig mit dem vorhandenen Speicherplatz umzugehen, werden in den meisten Teilen unserer Bakkalaureatsarbeit, Aufzählungen stattfinden, die die Größe der notwendigen Datenstrukturen ermitteln. In diesem Fall wird für jede Schleife die Anzahl ihrer Nachfolger berechnet. Wir führen die zwei Begriffe *Ober-* bzw. *Unterschleife* ein. Es gilt, dass wenn eine Schleife B sich innerhalb einer Schleife A befindet, A die *Oberschleife* und B ihre *Unterschleife* ist. Wir wissen nach der Ausführung des Algorithmus von Tarjan, welcher Schleife jeder Knoten angehört. z.B. Wir haben die Schleifen A und B, bzw. die Knoten a beliebig aus A, b der *Header* von B und c ein beliebiger Knoten aus B. Nun gilt `loopHeader[a]` = head(A) und `loopHeader[c]` = head(B) = b. Besonders ist `loopHeader[b]` = head(A). Das liegt daran, dass die *Unterschleife* B nur durch ihren Header angesprochen wird, und dieser wiederum als einfacher Knoten zu der Schleife A gehört. `succs[loopId[loopParent[forest.loops[i].header]]]` muss daher inkrementiert werden. `i` ist die aktuelle Schleife die bearbeitet wird. `forest.loops[i].header` ist ihr Header. `loopParent[forest.loops[i].header]` ist der Header der *Oberschleife*. `loopId[loopParent[forest.loops[i].header]]` ist die ID der Oberschleife. Und zu guter Letzt ist `succs[loopId[loopParent[forest.loops[i].header]]]` die Anzahl der Nachfolger dieser. Diesem Prinzip entsprechend folgt auch die tatsächliche Konstruktion des Schleifenwaldes. Ein zusätzlicher Schritt ist das Aktualisieren der Baumwurzelinformation, die in jeder Schleife gespeichert ist: Jedes Loop, weiß in welchem Baum es sich befindet. Dieser wird durch die ID der Wurzelschleife eindeutig identifiziert. Soll ein Schleifenunterbaum einem anderen hinzugefügt werden, so muss für jede Schleife des Unterbaumes die ID der neuen Wurzelschleife gespeichert werden. Als Nächstes wird der BFS Algorithmus auf jede der gefundenen Baumwurzelschleifen in der `getBFSNumsIncomplete` Form angewendet. Dadurch haben wir am Ende in dem `bfsBackArray` alle Schleifen des Waldes, topologisch sortiert. Als Letztes werden in der *Loop-Backedge Synchronisation* Phase für jede Schleife die entsprechenden Backedges gefunden. Es werden nur jene Backedges betrachtet, die dem Headerknoten "gehören". Dieser Schritt mag zu diesem Zeitpunkt überflüssig erscheinen, da diese Information

Listing 3.10: Die Standard Methoden

```
1  public class BWA {
2
3      /**
4       *  die Methoden der Standard Implementierung
5       */
6
7      // die Hauptmethode
8      public static void runBWA(cfg);
9
10
11     // die Termination Statements der Schleife 'loop'
12     private static int[] getT(forest, cfg, loop);
13
14     // die Level 0 Candidates der Termination Statements
15     //   aus der Schleife 'loop'
16     private static String[] getV(forest, cfg, loop, T[]);
17
18     // die Level n Candidates der Termination Statements
19     //   aus der Schleife 'loop'
20     private static String[] getVX(forest, cfg, loop, T[]);
21
22     // die Candidates des naechsten Levels fuer einen
23     //   bestimmten Knoten
24     private static void getNodeV(forest, cfg, node, NodeInfo);
25
26     // die BW Eigenschaft einer Variable
27     private static boolean isBWV(forest, cfg, loop, variable);
28
29     // Ueberpruefung ob ein Knoten eine Variable beschreibt
30     private static boolean writes(cfg, node, variable);
31
32
33     // Handler fuer eine erkannte BW Variable (Ausgabe)
34     private static void handleBWV(forest, cfg, root,
35                                   loop, variable);
36
37     ...
```

auch zu einem späteren Zeitpunkt ermittelt werden kann, ist aber in einer "auf einmal"-Fassung viel effektiver, da wir sie später wiederholt brauchen werden.

3.2.3 Der Standard-Busy-Wait-Algorithmus

Die Hauptmethode des Algorithmus, die für die Analyse jedes CFGs aufgerufen wird heißt `runBWA`. Zuerst wird der *Loop Forest* berechnet. Anhand dieser Information, werden für jede Schleife folgende Schritte durchgeführt: `getT` - die Berechnung der *Termination Statements*, `getV` - die Berechnung der *Level 0 Busy Wait Suspects* bzw. `getVX` - die Berechnung der *Level n Busy Wait Suspects*, und anschließend, für jede dieser gefundenen Variablen `isBWV` - überprüft ob diese Variable tatsächlich laut Algorithmus der Busy Wait Bedingung entspricht. Wir führen die zwei Begriffe *Level 0 Busy Wait Suspect* und *Level n Busy Wait Suspect* ein. Mit dem ersteren werden die Variablen, die sich unmittelbar innerhalb eines Termination Statements befinden, bzw. mit dem letzteren, die nach n Iterationen des von *Blieberger et al.* vorgeschlagenen Verbesserungsverfahrens gefundenen Variablen, gekennzeichnet. Am Ende wird für jede erkannte BW-Variable die `handleBWV` Methode aufgerufen. Diese Aufteilung erschien uns im

Sinne einer besseren Wartbarkeit und Erweiterbarkeit des Codes sinnvoll.

`getT` erkennt die Termination Statements des als Parameter übergebenen Loops, indem es nur dessen Knoten betrachtet, und jene auswählt, die Edges besitzen, die nicht ins aktuelle Loop führen. Das heißt für jeden Knoten werden die `succs` der Reihe nach analysiert. Sollte sich einer dieser Nachfolger nicht in der gleichen Schleife befinden, so heißt dies, die Kante führt hinaus. Wenn `loop` das aktuelle Loop ist und `dnode` der Zielknoten der aktuellen Kante (der aktuelle Nachfolger), dann muss eine der folgenden Bedingungen stimmen, damit der Quellknoten ein Termination Statement ist: `forest.root[loop]` $\neq$ `forest.root[forest.loopId[dnode]]` oder `loop < forest.loopId[dnode]` Anders ausgedrückt: Die Wurzel der aktuellen Schleife (wo der Quellknoten sich befindet) ist ungleich der Wurzel der Schleife in der sich der Zielknoten befindet (also zwei ganz unterschiedliche Schleifenbäume), bzw. die ID der Zielschleife ist größer als die ID der aktuellen. Das letztere deutet auf einen Sprung in die Oberschleife hin. Die Erklärung dafür ist, dass die Knoten, bei der Erkennung der Loops, in einer rückwärts DFS-Reihenfolge durchgemustert werden, und die IDs somit aufsteigend von unten nach oben vergeben werden. D.h. für zwei beliebige Loops A und B, mit A als Oberschleife von B gilt immer, dass ID(A) > ID(B). Sollte eine dieser zwei Bedingungen stimmen, dann bricht die Iteration über die Nachfolger des aktuellen Quellknotens ab, und markiert diesen als Termination Statement.

Nach dem Sammeln der Termination Statements müssen die *Busy Wait Candidates* gefunden werden. *Blieberger et al.* schlagen zwei Algorithmen vor: eine "straight forward" Variante die nur die *Level 0 Candidates* betrachtet und eine fortgeschrittene, die alle Candidates bis zu *Level n* berechnet.

In der `getV` Methode wird die erstere implementiert. Zuerst werden alle Candidates im Sinne einer effizienten Speicherplatzverwaltung gezählt. In den Node Datenstrukturen, die *SootCom*, liefert kann es vorkommen, dass eine Node Struktur mehrere *Lines* hat. Tatsache ist jedoch, dass wenn ein Node eine Verzweigung enthält, diese immer als letzte Line gespeichert ist. Abgesehen davon kann eine solche Line Variablen links und rechts haben (wie z.B. in `i < j`). Folgende Vereinbarung wurde aber in der Designphase des BWA-Projektes getroffen: Alle Verzweigungslines werden durch SootCom umgeformt, und die Variablen werden alle rechts gespeichert. Aus den oberen Überlegungen ergibt sich, dass in der Variablensammelphase nur die rechte Seite der letzten Line eines Nodes betrachtet werden muss. Es wird somit die rechte Seite der letzten Line eines jeden von `getT` gelieferten Termination Statements betrachtet. Diese Variablen sind die *Level 0 Candidates* - sie kommen unmittelbar in den Entscheidungen vor, und die eventuellen *Busy Wait Variables* befinden sich mit höchster Wahrscheinlichkeit unter diesen. Man muss aber in der Sammelphase auch bedenken, dass eine Variable mehrfach vorkommen kann, deswegen muss jedes Mal überprüft werden, ob wir sie bereits gespeichert haben. Am Ende werden die Variablen ein zweites Mal gezählt und ein Array entsprechender Größe alloziert. Diese Vorgehensweise wird notwendig, da die erste Zählung nur die Bruttoanzahl betrachtet und nicht das wiederholte Auftreten einer Variable berücksichtigt.

Die *Level n Candidates* werden von der `getVX` Methode berechnet. Grundlage für die fortgeschrittene Variablensuche sind die Level 0 Candidates. Sie werden

zuerst berechnet. Die erste Aufzählung zwecks Speicherplatzallozierung ist noch gröber, da eine genaue Berechnung zur Zeit nicht möglich ist. Somit werden alle n *Lines* der aktuellen Schleife gezählt und ein Speicherbereich der Größe $3{\cdot}n$ wird reserviert. Jede Line kann maximal 3 neue Variablen beitragen. Die Level 0 Candidates werden dann in dieses Array kopiert. Angenommen es wurden p solche Candidates gefunden. Es werden zwei Zeiger c_1 auf 1 und c_2 auf p gesetzt, die immer auf dem Bereich des letzten Levels im Array zeigen. Ein kompletter Durchlauf bedeutet alle Knoten in der Schleife, unter Berücksichtigung des vorigen Levels i.e. des $[v_{c_1}..v_{c_2}]$ Bereiches, zu scannen. Für alle Knoten der Schleife wird wiederholt eine eigene Methode `getNodeV` eingesetzt. Diese erzeugt dann im Bereich $[v_{c_2}..v_c]$ die Candidates des nächsten Levels für den als Parameter übergebenen Knoten. Nach dem Überarbeiten aller Schleifenknoten werden die Zeiger entsprechend justiert: c_1 auf $c_2{+}1$ und c_2 auf c. Diese Art von Bereichsuche wird so oft ausgeführt, bis keine neuen Candidates mehr gefunden werden. Da der Headerknoten den Schleifenknoten nicht gleichgestellt ist, muss er jedes Mal separat analysiert werden. Am Ende wird ein Array richtiger Größe alloziert und die Daten werden herüberkopiert.

`getNodeV` analysiert immer nur einen einzigen Knoten. Für alle Lines eines Knotens werden zuerst die linke (es darf nur eine geben), dann die rechten Variablen gelesen. Sollte es entweder links oder rechts keine Variablen geben, wird die Analyse der aktuellen Line unterbrochen. Angenommen wir berechnen gerade die Level k Candidates. Es wird zuerst überprüft, ob die linke Variable ein Level k-1 Candidate ist. Wenn sie es nicht ist, wird die Analyse unterbrochen. Die Definition, auf die sich diese Überlegung stützt, ist, dass die Level-k-Candidates jene Variablen sind, die ein Level k-1 Candidate schreiben (verändern). Sollte die linke Seite ein Level k-1 Candidate sein (also im Array in dem Bereich $[v_{c_1}..v_{c_2}]$ gespeichert), so werden der Reihe nach die "Rights" betrachtet. Ist eine solche Variable noch nie verwendet worden, ist sie automatisch ein Level k Candidate und wird entsprechend gespeichert. Ist die Variable bereits verwendet worden, dann wird sie ignoriert.

Die Untersuchung ob ein Candidate eine Busy Wait Variable ist, wird von der Methode `isBWV` durchgeführt. Laut *Blieberger et al.* muss es für diese Variable einen *Non Writing Path* - einen nichtschreibenden Pfad von einem Backnode (siehe oben) - zum Schleifenkopf geben. Somit wäre es möglich, in der Schleife zu iterieren, ohne diese Variable, die in der Abbruchbedingung vorkommt, zu verändern. Wir führen zuerst die Methode `writes(node,var)` ein, die **`true`** zurückliefert, falls der Knoten `node` die Variable `var` schreibt. Siehe Abschnitt 3.1.5 für eine Erklärung des Verfahrens, wie man entscheidet, ob für eine Variable die Busy Wait Eigenschaft hält. Die Implementierung ist die exakte Umsetzung dieses.

Um zu überprüfen, ob ein Knoten eine Variable verändert, wird die Methode `writes(node,var)` verwendet. Die Methode iteriert einfach über die Lines des Knotens `node`, bis eine Zeile gefunden wird, in der die angegebene Variable links in der Zuweisung steht. Die Suche wird dann aus Geschwindigkeitsgründen abgebrochen. Sollte keine solche Zeile gefunden werden, so ist der Knoten ein Non Writing Node.

3.2.4 Die Abweichungen

Rekursive Knoten (mit und ohne T)

Die Methode `scanRecursiveBackedges` betrachtet der Reihe nach alle Backedges des Control Flow Graphs und wählt jene aus, deren Startknoten gleich dem Zielknoten ist. Da ein solcher rekursiver Knoten gleichzeitig auch ein *no-T Loop* sein kann, muss man auch überprüfen, ob dieser Knoten auch andere ausgehende Kanten hat. Da das ganze Loop aus einem einzigen Node besteht, sind alle Kanten, abgesehen von der Backedge, automatisch Termination Statements. Da eine Kante nicht doppelt auftreten kann, heißt das, dass die Anzahl der Kanten ausschlaggebend ist. Ist nur eine einzige vorhanden (die Backedge), dann handelt es sich um eine nichtterminierende Schleife. Es wird die Methode `handleNoTRecursive` aufgerufen, die lediglich eine entsprechende Meldung auf den Bildschirm schreibt. Ganz anders sieht die Behandlung aus, wenn der rekursive Knoten doch mehrere ausgehende Edges hat. In einem solchen Fall wird eine leicht geänderte "mini"-Version des Standard BWA Algorithmus angewendet (`handleTRecursive`). Hier muss man erwähnen, dass in Soot pro Block maximal eine Verzweigung existiert, und sollte der Block mehrere Lines enthalten, so steht die Verzweigung immer auf dem letzten Platz. Angesichts dieser Tatsache und unserer Konvention, dass im CFG bei allen Verzweigungen alle Variablen *rechts* gespeichert werden, reicht ein einziger `getRight()` Aufruf auf der letzten Line des in Frage kommenden Knotens, um in O(1) alle Level 0 Candidates zu berechnen. Möchte der User eine *Deep Scan Analysis* durchführen (also auch die Level n Candidates in Betracht ziehen), so wird die Methode `getNodeV` wiederholt auf den aktuellen Knoten angewendet, bis keine neuen Candidates mehr gefunden werden. Die Überprüfung, ob eine Variable eine Busy Wait Variable ist, ist in diesem Kontext sehr einfach. Man braucht nur einmal `writes` für jede Variable des aktuellen Knotens aufzurufen. Auf Makro-Niveau handelt es sich um einen Pfad mit einem einzigen Knoten $u \rightarrow u$. Offen bleibt nur die Frage, ob die Variable in diesem Node geschrieben wird (und das finden wir wie oben beschrieben heraus). Auf einem Mikroniveau besteht dieser $u \rightarrow u$ Pfad aus möglicherweise mehreren Lines, die hintereinander im u-Knoten gereiht sind. Die einzig mögliche Verzweigung liegt aber nur am Ende und dadurch eignet sich die `writes` Methode perfekt für den *writing/non-writing* Check. Für jede der gefundenen BW-Variablen eines terminierenden rekursiven Knotens wird die Methode `handleBWVR` aufgerufen, die dem Benutzer eine entsprechende Meldung liefert. Die Methode `scanRecursiveBackedges` wird in der Implementierung vor dem Standardalgorithmus aufgerufen.

No-T Loops

Die Suche nach nichtterminierenden Schleifen ist direkt in der `runBWA` Methode eingebettet. Unmittelbar nach dem `getT` Aufruf, der die Termination Statements zurückliefert steht die Abfrage, ob überhaupt etwas gefunden wurde. Sollte nichts gefunden worden sein, dann werden alle anderen Überprüfungen storniert, und eine entsprechende Ausgabe wird mittels `handleNoTLoop` auf dem Bildschirm geschrieben.

Listing 3.11: Die Methoden der Abweichung

```
1  public class BWA {
2
3      /**
4       *  die Methoden der Abweichung
5       */
6
7      // die BW Eigenschaft einer Variable
8      //   mit Source Aware Backpath Finding
9      private static boolean isBWVX(forest, cfg, loop,
10                                   variable, xnode);
11
12     // suche nach rekursiven Backedges
13     private static void scanRecursiveBackedges(forest, cfg);
14
15     // Hashvalue fuer die ID einer erkannten BW Variable
16     //   - betrachtet alle Knoten der Schleife
17     private static string getVariableHashValue(forest, cfg,
18                                                root, loop,
19                                                variable);
20
21     // Hashvalue fuer die ID einer erkannten BW Variable
22     //   - betrachtet nur einen einzigen Knoten
23     private static string getVariableHashValue(forest, cfg,
24                                                header,
25                                                variable);
26
27     // Ueberpruefung ob der Benutzer in der Eingabedatei
28     //   diese Variable auskommentiert hat, und diese daher
29     //   ignoriert werden soll
30     private static boolean mustFilterVariable(hashValue);
31
32     // Handler fuer eine erkannte no-T Schleife (Ausgabe)
33     private static void handleNoTLoop(forest, cfg, root, loop);
34
35     // Handler fuer einen rekursiven no-T Knoten (Ausgabe)
36     private static void handleNoTRecursive(forest, cfg, header);
37
38     // Handler fuer einen rekursiven Knoten
39     //   - es werden seine Candidates untersucht
40     private static void handleTRecursive(forest, cfg, header);
41
42     // Handler fuer eine erkannte BW Variable eines
43     //   rekursiven Knotens (Ausgabe)
44     private static void handleBWVR(forest, cfg, header, variable);
45
46     ...
```

"Schwellen"-variablen (Lite Scanning)

Das Lite Scanning Prinzip ist in der `getV` Methode eingebettet. Da die genannte Methode die Termination Statements der Reihe nach durchgeht und analysiert, ist es am sinnvollsten, wenn vor Ort auch darüber entschieden wird, ob die Level-0-Candidates eines bestimmten Ts übernommen werden sollen. Zu diesem Zweck, wenn es sich um eine Lite Scan Suche handelt, führt die Methode automatisch `isBWV` Abfragen auf allen gefundenen Variablen aus. Ist eine von denen kein BW Suspect, so werden auch alle anderen Variablen des aktuellen Knotens weggelassen. Diese Filterung wirkt sich transparent auf die nächsten Bearbeitungsphasen aus. Der Overhead dieses Verfahrens (ist meistens nicht der Fall) kann jedoch unter Umständen, abhängig von der Anzahl der Knoten und Variablen, beträchtlich werden. Der Overhead rechtfertigt sich jedoch besonders bei großen Projekten, wo eine genaue Analyse von vielen Variablen sowieso nur sehr umständlich wäre.

Source Aware Backpath Finding

`isBWVX` implementiert den SABF-Algorithmus. Sie wird, sollte der Benutzer explizit mittels eines Aufrufparameters danach verlangen, anstatt der `isBWV` Methode verwendet. Der Algorithmus basiert im Grunde auch auf der typischen Rückwärts-BFS, die bei der `isBWV` Methode verwendet wurde. Die Hauptunterschiede bestehen einerseits darin, dass die Suche nicht mehr bei dem Loop-Header, sondern bei dem vorgegebenen Knoten `xnode` anfängt, andererseits darin, dass die Annahme bezüglich des Loop Headers, die in der Standardversion getroffen wurde, nicht mehr gilt. Die erste Version stützt sich auf die Tatsache, dass der Schleifenkopf alle Schleifenknoten dominiert, und mehr noch, dass wenn die Suche den Loop Header erreicht hat, man sie unterbrechen kann. Ein nicht schreibender Pfad wurde gefunden. Diese Eigenschaften bewirken, dass diese Art von Rückwärtssuche *bereichtreu* ist, und dass sie ohne weitere Überprüfungen durchgeführt werden kann. Wird die Suche (wie hier der Fall) woanders gestartet, muss man sie natürlich solange ausführen, bis entweder keine Kanten mehr zur Verfügung stehen oder bis der Startknoten der Suche wieder erreicht wurde. Das heißt, man soll über den Schleifenkopf hinweg fortfahren. Das Problem hier ist, dass der Loop-Header, anders als die Loop-Knoten, bei denen keine *Alien Edges* erlaubt sind, auch externe Kanten hat. Ohne eine gesonderte Behandlung des Schleifenkopfes würde sich jetzt die Suche über den ganzen Graphen ausbreiten, was im Fall eines relativ großen Graphen bestimmte Geschwindigkeitseinbußen bedeuten würde. Man beginnt nicht mit den Backnodes, wie bei der vorigen Implementierung, sondern direkt mit dem `xnode`. Die Vorgänger (das `pred` Array) werden iterativ durchgemustert, während man gleichzeitig auch Ausschau nach dem Loop-Header hält. Kommt man zum Loop-Header, werden aber nicht seine Vorgänger laut dem `pred` Array in Betracht genommen, sondern der Algorithmus iteriert in diesem Schritt über die Backnodes der Schleife. Somit vermeidet man das "Hinausrutschen" der Suche aus der Schleife. Das Verfahren terminiert, wenn entweder der `xnode` erreicht wurde, oder wenn alle Knoten der Schleife besucht worden sind.

BWV Filtering

Das zuletzt implementierte Feature ist die Möglichkeit, sich bei einer Analyse eine formatierte Datei generieren zu lassen, die zu einem späteren Zeitpunkt als Eingabedatei verwendet werden kann. Da der Algorithmus auch falsche Alarme erzeugen kann, können diese "Schein"-BWVs in der Ausgabedatei *auskommentiert* werden, damit sie bei einer erneuten Untersuchung nicht mehr ausgegeben werden. Um eine BW-Variable möglichst eindeutig zu identifizieren, verwendet man die `getVariableHashValue` Methode, die für die als Parameter übergebene Variable, abhängig von einem Toleranzwert, eine Art Fingerabdruck des umliegenden Codes berechnet. Eine hundertprozentige Toleranz bedeutet, dass alle Codezeilen (alle von Soot gelieferten Lines) der aktuellen Schleife betrachtet werden. Ein Prozent bedeutet hingegen, dass nur die erste Zeile betrachtet wird. Je größer die Toleranz desto mehr Zeilen werden miteinbezogen. Die Knoten der Schleife werden mittels einer Vorwärts-Breitensuche topologisch sortiert. Die ID wird als CRC Summe des Variablennamens, des Methodennamens und der Lines aller Knoten bis zu einer bestimmten Tiefe, mit den letzten zwei Stellen ersetzt durch den Integerwert der Toleranz, berechnet. Man muss die Toleranz speichern damit zu einem späteren Zeitpunkt eindeutig festgestellt werden kann, mit welcher Toleranz eine ID berechnet wurde. Zwei IDs, selbst wenn über den gleichen Code berechnet, sind unterschiedlich, wenn die verwendete Toleranz unterschiedlich ist. Eine solche ID-Berechnung hat die folgende Eigenschaft: Angenommen man hat eine 50-Prozentige Toleranz verwendet, dann wird eine Änderung im Code jenseits der Hälfte der Schleife vom BWA nicht wahrgenommen, und die IDs bleiben gleich - d.h. eine auskommentierte Variable mit dieser ID wird ignoriert. Ändert sich etwas im Code ganz vorne in der Schleife so spiegelt sich diese Änderung in der ID wider, und da nur Variablen, deren IDs auskommentiert sind, weggelassen werden, wird somit die gleiche Variable, aber mit einer neuen ID ausgegeben. Die Methode `mustFilterVariable` nimmt einen Hashwert als Parameter und überprüft ob sich diese ID in der Liste der zu ignorierenden Kennzeichen befindet. Rekursive Knoten brauchen eine gesonderte Behandlung was die Berechnung des Fingerabdruckes angeht. Da eine solche Schleife aus einem einzigen Knoten besteht muss darauf keine Tiefensuche angewendet werden. Die Methode `getVariableHashValueSimple` berechnet den Hashwert nur unter Berücksichtigung der Lines des als Parameter übergebenen Knotens. Diese drei Methoden werden von den `handler*`-Funktionen verwendet, um Variablen auszuschließen, wenn das BWA Filtering eingeschaltet ist.

4 BWA in der Praxis

Um einen Überblick über das Verhalten des *Busy Wait Analyser* in der Praxis zu erhalten, und nicht zuletzt um die korrekte Funktionsweise sicherzustellen, wurden während und nach Abschluss der Entwicklungsphase Teststläufe mit zufällig ausgewählten, im Quelltext verfügbaren Java-Programmen vorgenommen.

Hierbei zeigte sich früh die Problematik der falsch erkannten Busy-Wait-Variablen. So werden bei einem normalen Scanvorgang auf einem zwischen 50000 und 100000 Zeilen langen Programm normalerweise bereits über hundert Busy-Wait-Suspects ausgegeben. Führt man einen *Deep Scan* durch, so verdreifacht sich diese Anzahl in der Regel. Durch Einführung des *Lite Scan* (Abschnitt 3.1.8) und *Source Aware Backpath Finding* (Abschnitt 3.1.9) konnte allerdings, zugunsten der Usability, eine drastische Reduktion des Outputs auf etwa 12% erreicht werden. Bezogen auf die von uns untersuchten Programme bedeutete dies, dass zwischen 5 und 35 Busy-Wait-Suspects ausgegeben wurden.

Eine in jedem Fall notwendige manuelle Kontrolle ist bei dieser Anzahl von Busy-Wait-Suspects mehr als vertretbar, und wird durch das *BWV-Filtering* (Abschnitt 3.2.4) zusätzlich erleichtert.

So konnten schließlich, wie erwartet vorwiegend in *Thread-Klassen* und in Verbindung mit dem Aufruf von `sleep`, auch einige bestätigte Busy-Wait-Variablen gefunden werden, wobei lediglich eines der drei näher untersuchten Programme keine einzige Verwendung von Busy Wait aufwies.

Abbildungsverzeichnis

Listings

Literaturverzeichnis

[BBS03] Johann Blieberger, Bernd Burgstaller, and Bernhard Scholz. Busy wait analysis. *Reliable Software Technologies - Ada-Europe*, 2655:142–152, 2003.

[Dij68] E. W. Dijkstra. Co-operating sequential processes. In F. Genuys, editor, *Programing Languages*, pages 43–112. Academic Press, 1968.

[HU72] Matthew S. Hecht and Jeffrey D. Ullman. Flow graph reducibility. In *STOC '72: Proceedings of the fourth annual ACM symposium on Theory of computing*, pages 238–250, New York, NY, USA, 1972. ACM Press.

[Lam00] Patrick Lam. On the soot menagerie - fundamental soot objects [online]. März 2000 [zitiert am 16.04.2005]. URL: `http://www.sable.mcgill.ca/soot/tutorial/menagerie/index.html`.

[Ram99] G. Ramalingam. Identifying loops in almost linear time. *ACM Transactions on Programming Languages and Systems*, 21(2):175–188, März 1999.

[soo05] Soot: a java optimization framework [online]. Jänner 2005 [zitiert am 25.04.2005]. URL: `http://www.sable.mcgill.ca/soot/`.