Technische Universität Braunschweig

Studienarbeit

Approximate Dynamic Slicing

Stefan Konst

Aufgabenstellung und Betreuung:

Dipl.-Inform. Jens Krinke

Februar 1999

Institut für Programmiersprachen und Informationssysteme
Abteilung Softwaretechnologie

Prof. Dr. Gregor Snelting

Erklärung

Ich versichere, die vorliegende Arbeit selbständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

Braunschweig, den 1. Februar 1999

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Im Rahmen der Software-Entwicklung besteht eine wesentliche Aufgabe darin, die Korrektheit der entwickelten Software sicherzustellen. Hierzu muß überprüft werden, ob die implementierten Funktionen zum einen die von ihnen geforderten Eigenschaften haben und sich zum anderen nicht in einer ungewünschten Art und Weise gegenseitig beeinflussen. Abhängig vom Einsatzgebiet der überprüften Software muß die Prüfung selbst gewissen Mindestanforderungen genügen.

Eine der Aufgaben der Physikalisch-Technischen-Bundesanstalt (PTB) besteht in der Prüfung von Baumustern eichpflichtiger Meßgeräte. Dabei müssen bei Meßgeräten, die durch Software gesteuert werden, nicht nur deren physikalische Eigenschaften überprüft werden, sondern auch die sie steuernde Software. Besonderes Augenmerk ist hierbei auf den Pfad der Daten vom Sensor zur Anzeige, dem sogenannten Eichpfad, zu legen. Es muß ausgeschlossen werden, daß dieser durch externe Faktoren beeinflußt wird.

Um solch hohen Anforderungen gerecht zu werden, wurde im Verbundprojekt VALSOFT („Validierung softwaregesteuerter Meßsysteme durch Program-Slicing und Constraint-Solving“) ein Werkzeug entwickelt, daß die Zusammenhänge der Anweisungen und Variablen in einem C-Programm analysiert und darstellt. Die Auswertung erfolgt allerdings nicht vollautomatisch, sondern es obliegt dem Anwender dieses Werkzeugs, die Zusammenhänge zu deuten und so zum Beispiel ungewünschte Einwirkungen auf den Eichpfad, seien es Entwicklungsfehler oder mutwillige Manipulationen, zu erkennen [KSR].

Die Analyse eines Programms erfolgt in mehreren Schritten. Zunächst werden die C-Quellen geparst, um daraus einen attributierten abstrakten Syntaxbaum aufzubauen. Unter Analyse der Abhängigkeiten wird im nächsten Schritt ein System Dependence Graph (SDG) erstellt. Diese beiden Schritte wurden im Programm `mess` zusammengefaßt. Der erzeugte SDG dient als Grundlage für weitere Analysen, Vereinfachungen und die Visualisierung der Zusammenhänge (s. a. Abb. 1) [KSR].

Unter Slicing versteht man die Analyse eines Programms mit dem Ziel herauszufinden, welche Anweisungen eines Programms eine bestimmte Anweisung in einem bestimmten Programmpunkt beeinflussen bzw. von ihr beeinflußt werden. Bei Static Slicing erfolgt diese Analyse unabhängig von einer bestimmten Startkonfiguration, während sie bei Dynamic Slicing für eine bestimmten Startkonfiguration erfolgt.

In dieser Studienarbeit wird Approximate Dynamic Slicing (ADS) vorgestellt. Darunter ist zu verstehen, daß für die Methode des Static Slicings nicht das gesamte Programm herangezogen wird, sondern nur diejenigen Teile des Programms, welche bei einer speziellen Startkonfiguration durchlaufen wurden. Die praktische Umsetzung unterscheidet zwei Phasen. Zunächst werden die durchlaufenden Programmteile mit Hilfe von Breakpoints bestimmt. Dazu wird das Programm durch einen Debugger ausgeführt. Danach kann in einer zweiten Phase analysiert werden, welche Anweisungen dieser Programmteile eine bestimmte Anweisung beeinflussen oder von ihr beeinflußt werden und welche nicht. Um diese Beeinflussungen visualisieren zu können muß der SDG ent-

sprechend markiert werden. Im Rahmen dieser Studienarbeit wurde die zweite Phase jedoch nicht implementiert. Zur besseren Übersicht werden in Abbildung 1 die Zusammenhänge in ihren wesentlichen Zügen dargestellt. Ziel dieser (approximativen) Vorgehensweise ist es, gegenüber Static Slicing eine höhere Genauigkeit und gegenüber exaktem Dynamic Slicing einen zeitlichen Vorteil zu erreichen.

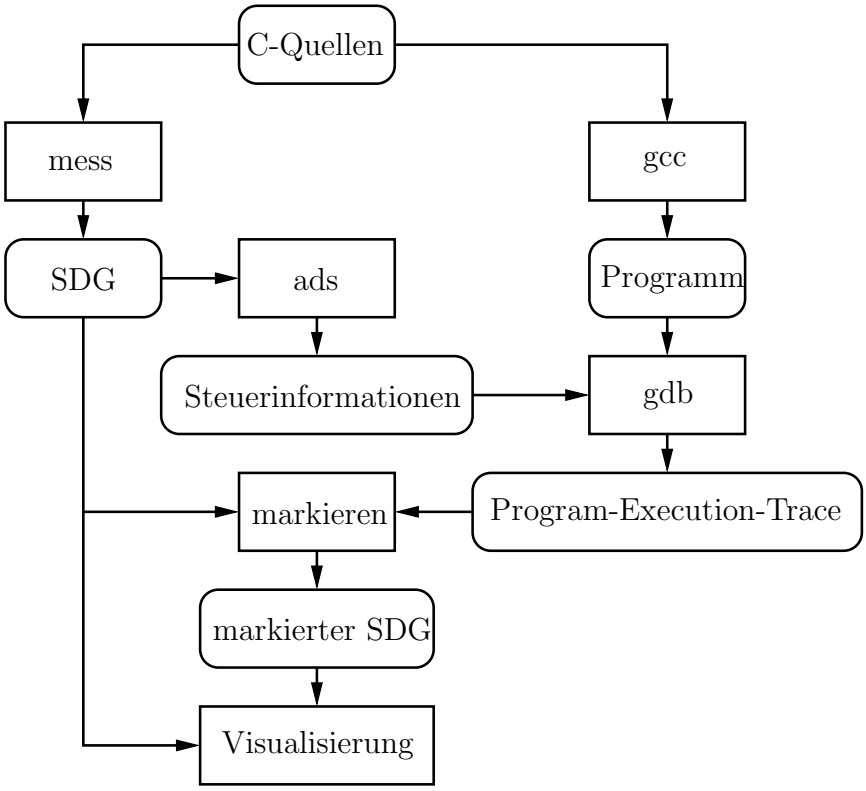

Abbildung 1: Zusammenhänge

Nach dieser Einleitung werden zunächst die allgemeinen Grundlagen behandelt. Im Anschluß daran wird auf die hier gewählte Vorgehensweise und Implementierung eingegangen. Es folgt eine Beschreibung zur Anwendung dieses Werkzeugs. Den Abschluß dieser Arbeit bildet eine Zusammenfassung, gepaart mit einem Ausblick auf mögliche Ergänzungen.

Eine weitergehende Einführung in das Verbundprojekt VALSOFT ist in dem bereits zitierten Artikel [KSR] und in Artikel [KS98] zu finden. Die Studienarbeiten [Bru97] und [Kön98] entstanden ebenfalls im Rahmen von VALSOFT.

2 Allgemeine Grundlagen

2.1 Graphen

Wegen des besseren Verständisses, werden die Bedeutungen der folgenden Begriffe hier nocheimal definiert:

Definition 2.1 *Ein* Control Flow Graph *(CFG) für ein Programm P ist ein gerichteter Graph, bei dem jeder Knoten mit einer Anweisung von P assoziiert wird und die Kanten den Programmfluß in P darstellen [BG96].*

Definition 2.2 *Ein* Program Dependence Graph *(PDG) für ein Programm P ist ein gerichteter Graph, dessen Knoten mit Anweisungen und Prädikaten von P assoziiert werden und dessen Kanten Daten- und Kontrollabhängigkeiten in P darstellen [Tip94].*

Definition 2.3 *Ein* System Dependence Graph *(SDG) ist ein Multigraph, der bei einem aus mehreren Prozeduren bestehenden Programm die PDGs der einzelnen Prozeduren mit Kanten verbindet, die die Aufrufabhängigkeiten und Parameterübergaben widerspiegeln [HRB90].*

2.2 Programmstruktur

Programme lassen sich in strukturiert und unstrukturiert einteilen. Strukturierte Programme bestehen aus

- Funktionen/Prozeduren,
- Verzweigungen (IF-THEN, IF-THEN-ELSE, CASE) und
- Schleifen (WHILE, REPEAT-UNTIL, FOR),

während unstrukturierte Programme

- GOTOs mit entsprechenden
- Labels

enthalten. Die Strukturen strukturierter Programme haben gemeinsam, daß sie sich in einen Kopf (Head) und in einen Rumpf (Body) aufteilen lassen:

- Funktionen/Prozeduren verweisen im Kopf auf die zu übergebenen und zurückzugebenen Parameter und enthalten im Rumpf die eigentlichen Anweisungen.
- Verzweigungen und Schleifen enthalten im Kopf die Bedingungen (Prädikate) für das Ausführen der Anweisungen des Rumpfes dieser Strukturierungsmechanismen. Der Rumpf selbst enthält die eigentlichen Anweisungen, zu denen wiederum Verzweigungen und Schleifen gehören können. Bei IF-THEN-ELSE- und CASE-Verzweigungen muß für jeden einzelnen Fall ein eigener Rumpf berücksichtigt werden.

Ein wesentliches Merkmal ist, daß diese Strukturen nicht miteinander verzahnt werden können, sondern immer nur vollständig in eine übergeordnete Struktur eingeschachtelt werden (s. a. Abb. 2). Für die Untersuchung strukturierter Programme sollte daher eine rekursive Vorgehensweise erwogen werden.

```
1
2
3
4   WHILE ( )
5     IF ( )
6
7   ENDWHILE
8
9     ENDIF
10
11
```

(a) verzahnt (nicht möglich)

```
1   PROC OddNumsSum( n )
2     i=0
3     sum=0
4     WHILE ( i <= n )
5       IF ( i mod 2 == 1 )
6         sum=sum+i
7       ENDIF
8       i=i+1
9     ENDWHILE
10    RETURN sum
11  ENDPROC
```

(b) eingeschachtelt

Abbildung 2: Beziehungen der Programmstrukturen zueinander

3 Program Slicing

Im folgenden Abschnitt findet eine weitgehend informale Einführung in die Grundlagen des Slicings statt. Teile dieser Einführung basieren auf den ausführlichen Übersichtsartikeln [BG96] und [Tip94]. Ihre Literaturverzeichnisses verweisen auf eine Vielzahl weiterer Artikel, die in Zusammenhang mit Slicing stehen.

3.1 Slice

Ein Programm wird aus einzelnen Anweisungen aufgebaut. In einer Vielzahl von Situationen, z. B. Debugging, Verstehen unbekannter Programme usw., ist es notwendig zu untersuchen, welche Anweisungen eine Variable an einem bestimmten Programmpunkt beeinflussen oder welche Anweisungen von einer Variablen an einem bestimmten Programmpunkt beeinflußt werden.

Definition 3.1 *Ein* Slice *ist eine Menge von Anweisungen eines Programms P, die eine Variable v an einem bestimmten Programmpunkt s beeinflussen oder von ihr beeinflußt werden können. Im ersteren Fall spricht man von* Backward Slice*, im letzteren Fall von* Forward Slice*.*

Backward und Forward deshalb, weil dies die Richtung angibt, in welcher ein PDG durchsucht werden muß, um den gewünschten Slice zu erhalten. In der vorliegenden Literatur werden zumeist Backward Slices gemeint, wenn nur von Slices gesprochen wird.

Definition 3.2 *Das Tupel aus Variable v und deren Lokalisierung im Programm s wird* Slicing-Kriterium (v, s) *genannt.*

Ein Slice ist immer relativ zu seinem Slicing-Kriterium zu betrachten.

Bei einem Backward Slice kann unter „beeinflussen“ ein sowohl direkter, als auch indirekter „schreibender Zugriff“ auf die Variable des Slicing-Kriteriums verstanden werden. Für den Slice ist dieser Zugriff nur dann von Bedeutung, wenn er vor Erreichen des Slicing-Kriteriums stattfindet. Anweisungen, die nur lesend zugreifen, können ignoriert werden, da ein lesender Zugriff keine Beeinflussung darstellt.

Dagegen kann bei einem Forward Slice unter „beeinflußt werden“ ein sowohl direkter, als auch indirekter „lesender Zugriff“ auf die Variable des Slicing-Kriteriums verstanden werden. Dieser Zugriff ist für den Slice nur dann von Bedeutung, wenn er nach Verlassen des Slicing-Kriteriums stattfindet. Schreibende Zugriffe sollten insoweit beachtet werden, als daß dadurch der alleinige Einfluß der Variable des Slicing-Kriteriums endet.

Entsprechend sind auch solche Anweisungen zu berücksichtigen, welche Einfluß auf Prädikate in Verzweigungen und Schleifen haben und dadurch den Programmfluß steuern [KSR, Tip94]. Weiterhin ist ein Slice immer nur eine Teilmenge der Menge aller Anweisungen eines Programms, welche maximal alle Anweisungen eines Programms beinhalten kann.

Definition 3.3 *Die* Startkonfiguration *eines Programms ist die Menge der Werte aller Parameter eines Programms, welche beim Programmaufruf übergeben bzw. während des Programmlaufs eingelesen werden können.*

Die Berechnung eines Slice kann allgemein für alle möglichen Startkonfigurationen eines Programms erforderlich sein oder speziell für eine ganz bestimmte Startkonfiguration. Anhand dieser Unterscheidung ergeben sich zwei weitere Arten von Slices, nämlich *Static Slices* und *Dynamic Slices* [ADS90, AH90]:

Definition 3.4 *Ein* Static Backward Slice *beinhaltet alle Anweisungen, die unabhängig von einer Startkonfiguration die Variable des Slicing-Kriteriums beeinflussen können.*

Definition 3.5 *Ein* Dynamic Backward Slice *beinhaltet nur die Anweisungen, die abhängig von einer Startkonfiguration die Variable des Slicing-Kriteriums beeinflussen.*

Abbildung 3 zeigt jeweils ein Beispiel für einen Static Backward Slice und einen Dynamic Backward Slice.

Liegt das Slicing-Kriterium innerhalb einer Schleife, soll hier der Einfachheit halber bei einem Dynamic Backward Slice immer vom letztmaligen Schleifendurchlauf ausgegangen werden. Die Anzahl der Schleifendurchläufe kann dabei entweder fest im Programm vorgegen sein oder durch die dem Programm übergebenen Parameter bestimmt werden.

Wenn das Slicing-Kriterium innerhalb einer Schleife liegt, muß für Backward Slices beachtet werden, daß auch Anweisungen hinter dem Slicing-Kriterium innerhalb der Schleife dieses beeinflussen können (s. a. Abb. 3(b)).

Definition 3.6 *Ein* Static Forward Slice *beinhaltet alle Anweisungen, die unabhängig von der Startkonfiguration von der Variable des Slicing-Kriteriums beeinflusst werden können.*

```
1   PROC OddNumsSum( n )
2     i=0
3
4     WHILE ( i <= n )
5
6
7
8       i=i+1
9     ENDWHILE
10
11  ENDPROC
```

(a) Static Backward Slice für `(i,8)`

```
1   PROC OddNumsSum( n )
2     i=0
3     sum=0
4     WHILE ( i <= n )
5       IF ( i mod 2 == 1 )
6         sum=sum+i
7       ENDIF
8       i=i+1
9     ENDWHILE
10
11  ENDPROC
```

(b) Dynamic Backward Slice für `(sum,6)` und `n=1`

Abbildung 3: Backward Slices

Definition 3.7 *Ein* Dynamic Forward Slice *beinhaltet nur die Anweisungen, die abhängig von der Startkonfiguration von der Variable des Slicing-Kriteriums beeinflusst werden.*

In Abbildung 4 sind jeweils ein Beispiel für einen Static Forward Slice und einen Dynamic Forward Slice zu sehen. Im Gegensatz zu Dynamic Backward Slices soll hier vom erstmaligen Schleifendurchlauf ausgegangen werden, wenn das Slicing-Kriterium innerhalb einer Schleife liegt.

```
1   PROC OddNumsSum( n )
2
3
4     WHILE ( i <= n )
5       IF ( i mod 2 == 1 )
6         sum=sum+i
7       ENDIF
8       i=i+1
9     ENDWHILE
10    RETURN sum
11  ENDPROC
```

(a) Static Forward Slice für `(i,8)`

```
1   PROC OddNumsSum( n )
2
3
4
5
6         sum=sum+i
7
8
9
10    RETURN sum
11  ENDPROC
```

(b) Dynamic Forward Slice für `(sum,6)` und `n=1`

Abbildung 4: Forward Slices

Es gibt verschiedene Definitionen darüber, ob ein Slice ein ausführbares (übersetzbares) Programm darstellen muß oder nicht [Tip94], [AH90]. Nicht

zuletzt hängt dies von der Verwendung des Slices ab und ist nicht in allen Fällen nötig. Bei Dynamic Slices ist zu beachten, daß eine eventuelle Ausführbarkeit wesentlich von der Startkonfiguration des Programms abhängt, für welche der Slice berechnet wurde. Wird die Startkonfiguration geändert, kann eine Ausführung fehlschlagen, da nun beispielsweise Prädikate zum Durchlaufen bestimmter Programmteile erfüllt werden, die vorher nicht in den Slice aufgenommen wurden.

3.2 Prinzipien

Unter *Slicing* versteht man den Vorgang Slices zu berechnen. Allgemein läßt es sich als ein Verfahren zur Reduzierung der Komplexität bei der Betrachtung der Zusammenhänge für einen bestimmten Programmpunkt einordnen. Das heißt, daß die relevanten Anweisungen extrahiert werden [BG96].

Analog zu den vier verschiedenen Arten von Slices lassen sich vier Grundrichtungen des Slicings unterscheiden:

- *Static Backward Slicing*
- *Dynamic Backward Slicing*
- *Static Forward Slicing*
- *Dynamic Forward Slicing*

Die allgemeine Vorgehensweise zur Berechnung eines Slice läßt sich grob in zwei Schritte unterteilen, wobei die Ausgestaltung dieser Schritte abhängig von der Grundrichtung des Slicings ist:

- Analyse des Programms, um Informationen über Datenfluß und Kontrollabhängigkeiten zu gewinnen.
- Durch Analyse der Informationen über Datenfluß und Kontrollabhängigkeiten den Slice berechnen.

Unabhängig von den Grundrichtungen des Slicings gibt es verschiedene Problemformulierungen zur Berechnung von Slices [BG96]:

- Datenfluß im Control Flow Graph:
  Zunächst wird der Datenfluß der relevanten Variablen berechnet. In einem zweiten Schritt können daraus die relevanten Anweisungen des Slices extrahiert werden.
- Erreichbarkeit im Program Dependence Graph:
  Der Slice besteht aus allen Knoten, die vom Knoten des Slicing-Kriteriums aus erreichbar sind.
- Alternative Methoden:
  - Denotational Program Slicing: Functional Semantics
  - Denotational Program Slicing: Descriptive Semantics

  - Information Flow Relations
  - Parametric Program Slicing
  - Dynamic Slicing using Dependence Relations
  - Parallel Slicing
  - Value Dependence Graphs

Nach [Tip94] haben all diese Problemformulierungen zur Berechnung von Slices folgendes gemeinsam:

- Nicht relevante Anweisungen des Programms werden verworfen.
- Die Verfolgung der Daten- und Kontrollabhängigkeiten hat ihre Wurzel im Slicing-Kriterium.

Vergleichen lassen sich die Methoden in bezug auf die Genauigkeit der einzelnen Methoden und dem dafür zu betreibenden Aufwand [Tip94]. Zu überlegen ist, ob für eine bestimmte Zielsetzung eine Kombination mehrerer Methoden die Ergebnisse verbessern kann. Vor- und Nachteile des Einsatzes einer bestimmten Methode können im Einzelfall auch abhängig von den speziellen Gegebenheiten der jeweiligen Programmiersprache sein.

Als weitere Nebenbedingungen müssen die Struktur und die Elemente des Programms beachtet werden [BG96]:

- Nicht-verzweigende Programme:
  Sie enthalten nur Zuweisungen, die eine nach der anderen ausgeführt werden und bilden die einfachsten Programme.
- Strukturierte Programme:
  Sie enthalten z. B. Verzweigungen oder Schleifen. Die Ausführung der Anweisungen innerhalb dieser Konstrukte hängt von der Nicht-/Erfüllung ihrer Prädikate ab.
- Unstrukturierte Programme:
  Sie enthalten z. B. GOTOs oder BREAKs, die die Daten- und Kontrollabhängigkeiten unnötig verkomplizieren können. Die Überführung des Programms in eine GOTO-freie Version könnte dieses erheblich verändern, was vielleicht nicht wünschenswert ist und neue Risiken mit sich bringen kann.
- Arrays, Records, Pointer:
  Bei Arrays und Records ist nicht immer ersichtlich, auf welche Elemente zugegriffen wird. So müssen unter Umständen Anweisungen in den Slice aufgenommen werden, die das betrachtete Element nicht beeinflussen, dieses aber durch ihren Zugriff auf andere Elemente nicht ausgeschlossen werden kann. Pointer sind als die GOTOs der Datenstrukturen zu betrachten.
- Interprozedural:
  Es müssen zusätzliche Kanten für die Aufrufabhängigkeiten und Parameterübergaben eingesetzt werden.

3.2.1 Static Slicing

In diesem Abschnitt wird nur Static Backward Slicing behandelt. Hierunter versteht man die Berechnung eines Static Backward Slice. Auf die verschiedenen Methoden, die zur Berechnung von Static Slices herangezogen werden können, soll hier nicht gesondert eingegangen werden. Die wesentlichen Problemformulierungen wurden bereits allgemein im Abschnitt 3.2 genannt. Eine ausführlichere Beschreibung ist beispielsweise in [BG96] oder [Tip94] zu finden. Hier werden vielmehr die zu beachtenden Nebenbedingungen und die Ergebnisse in bezug auf die Genauigkeit und den zu betreibenden Aufwand aufgeführt.

Ein Static Backward Slice soll, unabhängig von einer speziellen Startkonfiguration, all diejenigen Anweisungen eines Programms enthalten, die Einfluß auf das Slicing-Kriterium haben können:

- Für nicht-verzweigende Programme müssen alle Anweisungen aufgenommen werden, die direkt oder indirekt das Slicing-Kriterium beeinflussen.
- Kommen Strukturierungsmechanismen, wie Verzweigungen oder Schleifen, hinzu, müssen auch deren Prädikate und weitere diese Prädikate beeinflussenden Anweisungen aufgenommen werden, wenn innerhalb des Rumpfes dieser Mechanismen das Slicing-Kriterium beeinflußt wird.
- Bei Arrays müssen vier Fälle unterschieden werden. Zum einen zwischen den Anweisungen, die das Array beeinflussen, und den Anweisungen, die mittels Zugriff auf das Array das Slicing-Kriterium beeinflussen. Zum anderen, ob die Zugriffe mittels konstanter oder variabler Indices erfolgen:
  - Wird nur mittels konstanter Indices auf das Array zugegriffen, können die einzelnen Elemente des Arrays wie eigenständige Variablen behandelt werden (s. a. Abb. 5(b)).
  - Wenn der das Slicing-Kriterium beeinflussende Zugriff auf das Array mittels eines konstanten Index erfolgt, müssen von den das Array beeinflussenden Zugriffen nur solche mit variablem Index und solche mit dem selben konstanten Index aufgenommen werden (s. a. Abb. 5(c)). Zugriffe mittels anderer konstanter Indices können ignoriert werden.
  - Erfolgt der das Slicing-Kriterium beeinflussende Zugriff auf das Array mittels eines variablen Index, müssen alle das Array beeinflussenden Zugriffe aufgenommen werden (s. a. Abb. 5(d)).
- Für Records gilt die gleiche Unterscheidung wie für Arrays, wobei es nur sehr wenige Programmiersprachen (z. B. PV-Wave) erlauben, über einen variablen Index auf ein Element eines Records zuzugreifen.
- Pointer können entsprechend dem Index eines (einelementigen) Arrays verstanden werden.
- Manche Programmiersprachen bieten die Möglichkeit mittels verschiedener Indizierungen auf einen bestimmten Speicherbereich zuzugreifen. In

```
1  PROC CombineData( i )
2    a=[3,4,5,6,7,8,9]
3    a(3)=0
4    c=a(2)
5    a(i mod 7)=0
6    d=a(3)
7    e=a(i+1 mod 7)
8    RETURN c+d+e
9  ENDPROC
```

(a) Beispielprogramm

```
1  PROC CombineData( i )
2    a=[ , ,5, , , , ]
3
4    c=a(2)
5
6
7
8
9  ENDPROC
```

(b) Static Backward Slice für (c,4)

```
1  PROC CombineData( i )
2    a=[3,4,5,6,7,8,9]
3
4
5    a(i mod 7)=0
6    d=a(4)
7
8
9  ENDPROC
```

(c) Static Backward Slice für (d,6)

```
1  PROC CombineData( i )
2    a=[3,4,5,6,7,8,9]
3    a(3)=0
4
5    a(i mod 7)=0
6
7    e=a(i+1 mod 7)
8
9  ENDPROC
```

(d) Static Backward Slice für (e,7)

Abbildung 5: Static Slices bei Array-Zugriffen

diesem Fall muß eine weitere Abstraktionsebene eingeführt und dadurch der Speicherbereich in einzelne Speicherzellen aufgeteilt werden. Im Prinzip können diese dann wieder wie normale Arrays oder Records behandelt werden, wobei zu beachten ist, daß eine Anweisung der übergeordneten Abstraktionsebene gleich auf mehrere benachbarte Speicherzellen zugreifen kann.

Als Ergebnis enthält ein Static Backward Slice alle möglicherweise das Slicing-Kriterium beeinflussenden Anweisungen. Dabei können Anweisungen enthalten sein, die bei einem Programmlauf tatsächlich keine Auswirkungen auf das Slicing-Kriterium haben, weil etwa ihre Ausführung von einem Prädikat abhängig ist, welches nie erfüllt wird (Prädikat abhängig von Variablen) oder erfüllt werden kann (Prädikat abhängig von Konstanten). Von der Präzision her besehen stellt ein Static Slice daher nur eine konservative Approximation dar.

Der zu betreibende Aufwand wird übersichtsartig in [Tip94] besprochen. Er ist bei einem CFG im wesentlichen abhängig von der Anzahl der Variablen des Programms und der Knoten und Kanten des CFG, bei einem PDG (SDG) von der Anzahl der Knoten und Kanten des PDG (SDG). Die Überführung eines

Programms mittels syntaktischer Analyse, z. B. in einen PDG, hat den Vorteil, daß ein PDG wesentlich leichter weiter analysiert und bearbeitet werden kann. Dies ist vor allem dann der Fall, wenn mehrere Slices berechnet werden, da der PDG nur einmal berechnet werden muß. Die Berechnung eines Static Backward Slice kann bei einem endlichen Programm stets in endlich vielen Schritten erfolgen, solange nur eine syntaktische Analyse des Programms vorgenommen wird.

3.2.2 Dynamic Slicing

Analog zum Abschnitt über Static Slicing, wird hier nur (Exact)[1] Dynamic Backward Slicing vorgestellt. Darunter ist die Berechnung eines Dynamic Backward Slice zu verstehen. In dieser Ausarbeitungen sollen dazu, wie im Abschnitt über Static Slicing, nur die zu beachtenden Nebenbedingungen und die Ergebnisse in bezug auf die Genauigkeit und den zu betreibenden Aufwand dargestellt werden. Eine ausführliche Übersicht über Dynamic Slicing ist z. B. in [BG96] oder [Tip94] zu finden.

Ein Dynamic Backward Slice soll, abhängig von einer speziellen Startkonfiguration, genau diejenigen Anweisungen eines Programms enthalten, die Einfluß auf das Slicing-Kriterium haben. Bei der Betrachtung eines Dynamic Backward Slice muß also neben dem Slicing-Kriterium auch die Startkonfiguration hinzugezogen werden. Weiter folgt daraus, daß für ein Slicing-Kriterium abhängig von der Startkonfiguration mehrere verschiedene Slices berechnet werden können, wofür Abbildung 6 ein Beispiel liefert.

```
1   PROC OddNumsSum( n )
2     i=0
3     sum=0
4     WHILE ( i <= n )
5       IF ( i mod 2 == 1 )
6
7       ENDIF
8       i=i+1
9     ENDWHILE
10    RETURN sum
11  ENDPROC
```

(a) Dynamic Backward Slice für `(sum,10)` und `n=0`

```
1   PROC OddNumsSum( n )
2     i=0
3     sum=0
4     WHILE ( i <= n )
5       IF ( i mod 2 == 1 )
6         sum=sum+i
7       ENDIF
8       i=i+1
9     ENDWHILE
10    RETURN sum
11  ENDPROC
```

(b) Dynamic Backward Slice für `(sum,10)` und `n=1`

Abbildung 6: Dynamic Backward Slices

Zur Berechnung eines Dynamic Backward Slice muß eine semantische Analyse des Programms vorgenommen werden, als deren Ergebnis man einen vollständigen Program-Execution-Trace erhält. Dies kann auf zwei Arten erfolgen:

[1] Vgl. Approximate Dynamic Slicing.

- Das Programm wird um Anweisungen zur Erzeugung eines Program-Execution-Trace ergänzt und danach ausgeführt.

- Der Sourcecode des Programm wird interpretiert, um daraus einen Program-Execution-Trace zu berechnen.

Probleme können beispielsweise dadurch auftreten, daß das Programm eine Endlos-Schleife beinhaltet. In diesem Fall kann kein Dynamic Slice für diese Startkonfiguration berechnet werden. Da das Auftreten einer Endlos-Schleife von der Startkonfiguration abhängig sein kann, muß diese für jeden konkreten Fall auch hier mit einbezogen werden.

Anhand des Programms und des Program-Execution-Trace läßt sich schließlich für die spezielle Startkonfiguration der Dynamic Slice des Slicing-Kriteriums berechnen. Sollte der Slice Anweisungen enthalten, die für die spezielle Startkonfiguration nicht das Slicing-Kriterium beeinflussen, so sind diese Abweichungen in den Ungenauigkeiten der verwendeten Verfahren zu suchen. Sie können vor allem aufgrund von Arrays, Records und Pointern auftreten, weil deren semantische Analyse eventuell zu kompliziert oder zu aufwendig ist und deshalb entsprechend alle Anweisungen wie bei einem Static Slice aufgenommen werden.

Der Aufwand zur Berechnung eines Dynamic Slice kann im Gegensatz zur Berechnung eines Static Slice sehr groß werden und in keinem Verhältnis zur erzielten Genauigkeit stehen. Der Aufwand ist nicht zuletzt abhängig vom jeweils untersuchten Programm, da dieses in irgendeiner Art und Weise ausgeführt oder interpretiert werden muß.

3.2.3 Approximate Dynamic Slicing

Im Einzelfall kann die Berechnung eines Exact Dynamic Slice unnötig aufwendig sein. Andererseits kann der Static Slice für das diesem Fall zugrundeliegende Slicing-Kriterium zu ungenau (zu groß) sein. Dieser Spannungsbogen zwischen Aufwand und Genauigkeit liefert den klassischen Hintergrund für approximative Vorgehensweisen.

In [ADS90] wurde ein *Approximate Dynamic Slice* als Durchschnitt aus dem Program-Execution-Trace für die aktuelle Startkonfiguration und dem Static Slice des zugrundeliegenden Slicing-Kriteriums definiert. Über den Program-Execution-Trace kann dabei die zu erreichende Genauigkeit bestimmt werden und die Berechnung des Static Slice liefert die untere Grenze für den zu betreibenden Aufwand. Da die Reihenfolge der Berechnungen für den Approximate Dynamic Slice von Bedeutung ist, wird auf sie am Ende dieses Abschnitts explizit eingegangen.

Die Genauigkeit des Program-Execution-Trace läßt sich anhand folgender Informationen schrittweise steigern:

1. Prozedur wurde angesprungen/verlassen, wobei alle Anweisungen der Prozedur zum Program-Execution-Trace gehören würden.

2. Rumpf einer Verzweigung oder einer Schleife wurde angesprungen/verlassen bzw. das zugehörige Prädikat wurde erfüllt oder nicht erfüllt, wobei

alle Anweisungen des Rumpfes zum Program-Execution-Trace gehören würden, die nicht zum Rumpf einer eingeschachtelten Struktur gehören.

3. Anweisung wurde ausgeführt.

Im Falle des Auftretens von GOTOs mit entsprechenden Labels kommen noch die Informationen

- GOTO wurde erreicht und
- Label wurde angesprungen

hinzu. Diese sind parallel zu den Informationen zu sehen, die besagen, daß eine Prozedur bzw. der Rumpf einer Verzweigung oder einer Schleife angesprungen/verlassen wurde, da strukturierte GOTO-freie Programme in „GOTO-Programme“ überführt werden können und umgekehrt (s. a. Abb. 7).

```
1   PROC OddNumsSum( n )
2     i=0
3     sum=0
4     WHILE ( i <= n )
5       IF ( i mod 2 == 1 )
6         sum=sum+i
7       ENDIF
8       i=i+1
9     ENDWHILE
10    RETURN sum
11  ENDPROC
```

(a) Strukturiertes GOTO-freies Programm

```
1   OddNumsSum:
2       i=0
3       sum=0
4   l:  IF NOT( i <= n ) GOTO el
5         IF NOT( i mod 2 == 1 ) GOTO ei
6           sum=sum+i
7   ei:
8         i=i+1
9       GOTO l
10  el:
11
```

(b) GOTO-Programm

Abbildung 7: Überführung

Unter GOTO können z. B. auch BREAK und EXIT eingeordnet werden, durch welche eine Verzweigung, eine Schleife oder eine Prozedur vor deren Ende verlassen werden können.

Die Genauigkeit des Program-Execution-Trace hängt weiterhin davon ab, ob die vorgenannten Informationen nur für das erstmalige oder für jedes Auftreten der zugrundeliegenden Ereignisse zur Verfügung stehen:

- Stehen die Informationen nur für erstmaliges Auftreten zur Verfügung, gibt der Program-Execution-Trace nur an, in welcher Reihenfolge die einzelnen Anweisungen erstmalig ausgeführt wurden. Diese Genauigkeit ist für Approximate Dynamic Slicing ausreichend, nicht jedoch für Exact Dynamic Slicing.

- Stehen dagegen die Informationen für jedes Auftreten zur Verfügung, gibt der Program-Execution-Trace genau an, in welcher Reihenfolge und wie oft die Anweisungen ausgeführt wurden. Diese Genauigkeit wird für Exact Dynamic Slicing benötigt.

Für die Abarbeitung der einzelnen Anweisungen eines Programms wird vorausgesetzt, daß sie sequentiell erfolgt. Daraus folgt, daß der Program-Execution-Trace allein von der Nicht-/Erfüllung der Prädikate von Strukturierungsmechanismen, wie Verzweigungen oder Schleifen, abhängig ist. Er kann daher allein anhand der Informationen, ob die einzelnen Prädikate dieser Strukturierungsmechanismen erfüllt wurden oder nicht, rekonstruiert werden. Natürlich immer unter dem Aspekt, ob die Informationen nur für erstmaliges oder für jedes Auftreten zur Verfügung stehen. Der Aufruf (Call) einer Prozedur oder Sprung (GOTO) zu einem Label sind zwingend und können so z. B. aus dem SDG ersehen werden. Abbildung 8 soll verdeutlichen wie die Anweisung in Zeile 6 des Programms `OddNumsSum` über mehrere Stufen vom Programmaufruf abhängig ist.

Die einzelnen Informationen können nun miteinander verglichen werden. Da sich mittels der Informationen welche Prädikate der einzelnen Strukturierungsmechanismen erfüllt wurden und welche nicht, der Program-Execution-Trace rekonstruieren läßt, sind die übrigen Informationen hierzu redundant. Das heißt, daß sich aus den Informationen

- „Prozedur wurde angesprungen/verlassen“,

- „Anweisung wurde ausgeführt“ und

- „GOTO wurde erreicht“,

- „Label wurde angesprungen“

keinerlei zusätzliche Informationen über den Program-Execution-Trace gewinnen lassen. Allerdings können sie hier zur (gegenseitigen) Verifizierung eingesetzt werden. Denn der Program-Execution-Trace läßt sich auch mittels der Information „Anweisung wurde ausgeführt“ rekonstruieren. Stehen dagegen nur die Informationen

```
1
2
3
4
5
6          sum=sum+i
7
8
9
10
11
```

(a) abhängig von (b)

```
1
2
3
4
5       IF ( i mod 2 == 1 )
6
7       ENDIF
8
9
10
11
```

(b) abhängig von (c)

```
1
2
3
4     WHILE ( i <= n )
5
6
7
8
9     ENDWHILE
10
11
```

(c) abhängig von (d)

```
1   PROC OddNumsSum( n )
2
3
4
5
6
7
8
9
10
11  ENDPROC
```

(d) abhängig vom Aufruf

Abbildung 8: Abhängigkeiten

- „Prozedur wurde angesprungen/verlassen“ und
- „GOTO wurde erreicht“,
- „Label wurde angesprungen“

zur Verfügung, ist eine Rekonstruktion des Program-Execution-Trace nur dann möglich, wenn das Programm keine von Prädikaten abhängigen Strukturierungsmechanismen enthält. Auch hier muß unterschieden werden, ob die Informationen nur für erstmaliges oder für jedes Auftreten zur Verfügung stehen.

Die Reihenfolge der Berechnungen des Program-Execution-Trace und des Static Slice ist in bezug auf den resultierenden Approximate Dynamic Slice nicht kommutativ. Die „richtige“, d. h. optimale Vorgehensweise ist diejenige,

1. zunächst den Program-Execution-Trace (PET) zu berechnen und
2. danach den Static Slice (SS), wobei hierfür nur die im Program-Execution-Trace vorhandenen Anweisungen herangezogen werden dürfen.

Ansonsten könnte der Approximate Dynamic Slice Anweisungen enthalten, die ganz offensichtlich, d. h. ohne semantische Analyse, nicht das Slicing-Kriterium beeinflussen können, wie Abbildung 9 verdeutlicht.

Wird zunächst für das Beispielprogramm (Abb. 9(a)) der Program-Execution-Trace berechnet, entfällt Zeile 4 mit der Anweisung `a=b`, da die IF-Abfrage nicht erfüllt wird. Folglich entfallen für den auf der Grundlage des Program-Execution-Trace berechneten Static Backward Slice des Slicing-Kriteriums `(c,6)`, die Zeilen 2, 3 und 5, wie in Abbildung 9(b) dargestellt wird.

Abbildung 9(c) zeigt das Ergebnis der „falschen“, d. h. suboptimalen Vorgehensweise. Für das Slicing-Kriterium `(c,6)` müssen zunächst alle Zeilen in den Static Backward Slice aufgenommen werden. Danach kann anhand des Program-Execution-Trace nur Zeile 4 entfernt werden, so daß die Zeilen 2, 3, und 5 enthalten bleiben, obwohl sie keinen Einfluß auf das Slicing-Kriterium haben.

```
1  a=...
2  b=...
3  IF ( FALSE )
4    a=b
5  ENDIF
6  c=a
```

(a) Beispielprogramm

```
1  a=...
2
3
4
5
6  c=a
```

(b) `(c,6)`, richtig: 1. PET, 2. SS

```
1  a=...
2  b=...
3  IF ( FALSE )
4
5  ENDIF
6  c=a
```

(c) `(c,6)`, falsch: 1. SS, 2. PET

Abbildung 9: Berechnungsreihenfolge beim Approximate Dynamic Slice

3.3 Operationen

Auf Slices können verschiedene Operationen, wie die Vereinigung, der Durchschnitt und die Differenz angewendet werden, aus denen sich Rückschlüsse für die Analyse eines Programms ziehen lassen (s. a. [Tip94]). Allgemein kann (s. a. [ZK97])

- der Schnitt zweier Slices als *Backbone-Slice* bezeichnet werden, welcher für gemeinsam genutzte Anweisungen steht,
- die Differenzmenge eines Slices zu einem anderen Slice, welche die Menge der Anweisungen darstellt, die nur in dem einen Slice vorhanden ist, als *Dice* bezeichnet werden und
- der Schnitt zwischen einem Forward- und einem Backward-Slice als *Chop* bezeichnet werden, welcher die Menge derjenigen Anweisungen darstellt, die bestimmte Eingaben zu bestimmten Ausgaben transformieren.

3.4 Vergleich

Hier werden die drei genannten Prinzipien des Slicings unter den Gesichtspunkten der Genauigkeit der erzielten Ergebnisse und dem zu betreibenden Aufwand miteinander verglichen. Der Vergleich erfolgt anhand ihrer Definitionen, die sie bei ihrer Anwendung idealerweise erfüllen (sollen) und nicht anhand bestimmter Implementierungen.

Static Slicing liefert, in bezug auf die das Slicing-Kriterium beeinflussenden oder von ihm beeinflußten Anweisungen, eine Grenze nach oben, während Dynamic Slicing eine Abschätzung nach unten liefert. Das heißt, daß es nicht mehr Anweisungen geben kann als im Static Slice enthalten sind. Andererseits kann es sehr wohl für ein Slicing-Kriterium einen kleineren Dynamic Slice geben, wenn eine andere Startkonfiguration gewählt wird.

Der Aufwand zur Berechnung eines Static Slice ist gering, d. h. linear, wenn die Berechnung auf der Grundlage eines SDG erfolgt. Die Berechnung eines Exact Dynamic Slice kann unter Umständen einen so großen Aufwand erforderlich machen, daß das Ergebnis in keinem Verhältnis zur eingesetzten Zeit steht.

Approximate Dynamic Slicing bietet einen Kompromiß zwischen dem zu betreibenden Aufwand und der Genauigkeit der Ergebnisse, wobei auch hier die Möglichkeit des Versagens besteht. Das heißt, daß der Aufwand entweder zu groß oder das Ergebnis zu ungenau werden kann. Abhängig ist dies beispielsweise davon, ob im Programm oft zu wiederholende Schleifen oder viele Pointer vorkommen.

3.5 Anwendungen

Unabhängig von einer bestimmten Methode, kann Slicing in einer Vielzahl von Gebieten eingesetzt werden, z. B. (s. a. [BG96, Tip94]) in:

- Debugging:
  Eingrenzung eines Fehlers.
- Differenzanalyse und Integration:
  Unterschiede zwischen zwei Programm(version)en herausfinden, Integration einer Entwicklungslinie in eine andere.
- Wartung:
  Einfluß einer Änderung auf andere Teile des Programms, ungenutzten Code entfernen.
- Test und Verifikation:
  Einfluß einer Anweisung auf das Ergebnis, Ausführung aller Programmteile.
- Optimierende Compiler:
  Mehrfaches Vorkommen eines Teilausdrucks.

Slicing kann als Hilfsmittel zur Lösung eines Problems herangezogen werden. Allerdings kann z. B. der Einsatz von Dynamic Slicing aufgrund des Aufwands

selbst wieder problematisch sein. Welche Slicing Methode letztendlich erfolgreich eingesetzt werden kann, hängt nicht zuletzt von der Erfahrung und Intuition des Entwicklers ab. Eventuell kann eine Kombinierung der verschiedenen Methoden den gewünschten Erfolg liefern.

Beispielsweise könnte ein Static Slice zu ungenau und ein Dynamic Slice zu aufwendig zu berechnen sein. Etwa in einem Fall, wo ein Slicing-Kriterium von einer Vielzahl von Prädikaten abhängig ist und zusätzlich im Program-Execution-Trace vor dem Slicing-Kriterium sehr viele Anweisungen oder gar Schleifen ausgeführt werden müssen, die keinen Einfluß auf das Slicing-Kriterium haben. In solch einem Fall können zunächst mittels Static Backward Slicing die überflüssigen Anweisungen heraussortiert werden und anschließend kann mit Hilfe von Dynamic Backward Slicing, welches auf den ermittelten ausführbaren Static Backward Slice angewendet wird, die Anzahl der Prädikate und der damit zusammenhängenden Programmteile reduziert werden.

Die Anweisungen einer Endlos-Schleife können, z. B. durch Static Slicing herausgefunden werden. Dynamic Slicing würde in diesem Fall versagen.

Kommen hingegen in einem Programm sehr viele Datenstrukturen mit Pointern vor, kann der berechnete Static Slice leicht das gesamte Programm umfassen. Hier können Dynamic Slices eventuell sehr viel kleiner sein [ADS90].

Slicing kann für das Debugging von Programmen eingesetzt werden. Zur Eingrenzung eines Fehlers berechnet man für die Stelle, an der der Fehler sichtbar wird, einen Slice. Dieser muß dann den Fehler beinhalten. Allerdings ist unter dieser Aussage nicht nur zu verstehen, daß der Slice eine falsche Anweisung enthält, etwa Addition statt Subtraktion, sondern auch, daß eine Anweisung gänzlich fehlt oder zum Slice gehören sollte, es aber wegen falscher Schreibweise nicht tut. Daß im Slice etwas fehlt, kann man aber erst feststellen, wenn man ihn gefunden hat. In diesem Sinne liegt der Fehler im Slice [AH90].

4 Algorithmen

Für einen Approximate Dynamic Slice müssen der Program-Execution-Trace der gegebenen Startkonfiguration und darauf basierend der Static Slice des zu untersuchenden Slicing-Kriteriums berechnet werden. In diesem Abschnitt werden die wesentlichen für die Implementierung benötigten Algorithmen beschrieben. Der Schwerpunkt liegt dabei auf der Ermittlung des Program-Execution-Trace, da die Berechnung eines Static Slice im Rahmen dieser Studienarbeit nicht implementiert wurde.

Die Ermittlung des Program-Execution-Trace soll mit Hilfe von Breakpoints geschehen, wozu das Programm in einem Debugger ausgeführt werden muß. Daraus ergibt sich folgende verfeinerte Vorgehensweise zur Berechnung eines Approximate Dynamic Slice:

1. Basierend auf dem SDG werden die Positionen der zu setzenden Breakpoints ermittelt. Gleichzeitig werden jedem Breakpoint entsprechend die Nummern der Knoten des SDG zugeordnet.

2. Das ausführbare Programm und die Positionen der zu setzenden Break-

points werden in den Debugger geladen. Nach dem Setzen der Breakpoints wird das Programm durch den Debugger ausgeführt.

3. Bei Erreichen eines Breakpoints werden die Nummern, der dem Breakpoint zugeordneten Knoten des SDG, ausgegeben und der Breakpoint gelöscht.

4. Abschließend kann anhand des SDG der Static Slice berechnet werden, wobei nur solche Knoten berücksichtigt werden dürfen, deren Nummer ausgegeben wurde. Als Ergebnis erhält man den Approximate Dynamic Slice.

4.1 Program-Execution-Trace

Für Approximate Dynamic Backward Slicing kann die Berechnung des Program-Execution-Trace für eine gegebene Startkonfiguration entweder bis zum Slicing-Kriterium oder bis zum Programmende erfolgen. Dabei kann sich aus der Berechnung bis zum Programmende zum einen ein gewaltiger Overhead ergeben, zum anderen ist es aber dadurch problemlos möglich die Position des Slicing-Kriteriums nach hinten zu verschieben, ohne den Program-Execution-Trace ständig neu berechnen zu müssen.

Für Approximate Dynamic Forward Slicing muß die Berechnung des Program-Execution-Trace für eine gegebene Startkonfiguration mindestens vom Slicing-Kriterium aus bis zum Programmende erfolgen. Da das Programm bis zum Slicing-Kriterium in diesem Fall sowieso ausgeführt werden muß, fällt hier die Berechnung des Program-Execution-Trace für das gesamte Programm nicht ins Gewicht.

Zusammengefaßt kann der Program-Execution-Trace für drei Bereiche berechnet werden:

1. Vom Programmanfang bis zum Slicing-Kriterium,

2. vom Slicing-Kriterium bis zum Programmende und

3. vom Programmanfang bis zum Programmende.

Da die ersten beiden Bereiche jeweils Teilbereiche des dritten Bereichs sind, soll hier, wenn nicht anders angegeben, der Program-Execution-Trace einer gegebenen Startkonfiguration stets vom Programmanfang bis zum Programmende betrachtet werden. Daraus ergibt sich die Erleichterung, daß nicht berücksichtigt werden braucht, ob das Slicing-Kriterium innerhalb einer Schleife liegt oder nicht.

Die benötigten Informationen für die Erstellung des Program-Execution-Trace können mittels Breakpoints gewonnen werden. Nachdem diese gesetzt wurden, muß das Programm ausgeführt werden. Aus den einzelnen Informationen, welche Breakpoints erreicht wurden, läßt sich der Program-Execution-Trace in der gewünschten Genauigkeit zusammensetzen. Hierzu müssen all die Anweisungen hergenommen werden, die hinter dem jeweiligen Breakpoint folgen und nicht in der Folge eines anderen Breakpoints liegen.

Das Setzen der Breakpoints muß analog zu Abschnitt 3.2.3 erfolgen:

- Prozedur wurde angesprungen/verlassen:
  Vor der ersten Anweisung der Prozedur bzw. hinter der letzten.
- Rumpf einer Verzweigung oder einer Schleife wurde angesprungen/verlassen:
  Vor der ersten Anweisung des Rumpfes bzw. hinter der letzten.
- Anweisung wurde ausgeführt:
  Hinter der Anweisung, vor der nächsten.
- GOTO wurde erreicht:
  Hinter der letzten Anweisung, vor dem GOTO.
- Label wurde angesprungen:
  Vor der ersten Anweisung, hinter dem Label.

Im Prinzip ist dieses Verfahren an keine spezielle Programmiersprache gebunden. Eventuelle Einschränkungen ergeben sich nur aus den Möglichkeiten des verwendeten Debuggers. Der hier verwendete `gdb` erlaubt es, z. B. Breakpoints auf der Ebene des Sourcecodes und auf der Ebene des Binärcodes zu setzen. Während er auf der Ebene des Binärcodes den Breakpoint an einer bestimmten Adresse setzen kann, kann er auf der Ebene des Sourcecodes den Breakpoint nur an den Anfang und nicht innerhalb einer Zeile setzen.

Je nachdem, mit welcher Genauigkeit der Program-Execution-Trace berechnet wurde, besteht dieser aus einer Menge von Anweisungen,

- die ausgeführt wurden, aber keinen Einfluß auf das Slicing-Kriterium hatten, oder
- die ausgeführt wurden und das Slicing-Kriterium beeinflußt haben.

In den folgenden Unterabschnitten sollen nun Algorithmen zur Ermittlung eines Program-Execution-Trace für eine konkrete Implementierung vorgestellt und diskutiert werden.

Zunächst ist für das zu untersuchende Programm der zugehörige SDG zu erzeugen. Dieser dient als Grundlage der Algorithmen und neben den Abhängigkeiten, muß aus ihm der Bezug zum Sourcecode hervorgehen, wie Dateiname, Zeile und Spalte. Das Programm selbst ist mit entsprechenden Optionen für den Debugger zu compilieren und durch den Debugger auszuführen.

4.1.1 Setzen der Breakpoints

Nachdem der SDG erzeugt wurde, müssen die Breakpoints gesetzt werden. Sie sollen sich stets vor der ersten Anweisung

- einer Prozedur,
- des Rumpfes einer Verzweigung oder einer Schleife oder
- hinter einem Label

befinden. Hierzu muß der SDG nach solchen Knoten durchsucht werden, die den Anfang des jeweiligen Konstruktes signalisieren. Hat man einen entsprechenden Knoten gefunden, kann die zugehörige Position im Sourcecode ausgelesen werden, an die dann der Breakpoint gesetzt werden kann. Die Ausführungen über GOTOs und Labels sind eher theoretischer Natur, da GOTOs mit entsprechenden Labels zur Zeit noch nicht im SDG implementiert sind. Das Hauptaugenmerk soll daher allein auf strukturierte Programme gerichtet werden.

Für das Auffinden der Knoten im SDG bieten sich zunächst zwei Strategien an:

- Durchwandern des SDG entlang seiner Kanten, um zu den einzelnen Knoten zu gelangen, oder
- sequentielles Auswerten aller Knoten.

Das alleinige Durchwandern des SDG entlang seiner Kanten hätte zur Folge, daß neben den Kanten für unbedingte Kontrollabhängigkeiten (UN) auch Kanten für Prozeduraufrufe (Call) weiterverfolgt werden müßten. Da in erster Linie strukturierte Programme untersucht werden sollen, würde auf der anderen Seite ein rein sequentielles Auswerten aller Knoten nicht den in Abschnitt 2.2 beschriebenen Beziehungen aus übergeordneter und eingeschachtelter Struktur gerecht werden.

Die Lösung für strukturierte Programme bietet eine Kombination beider Strategien, wodurch der SDG praktisch in seine PDGs zerlegt wird und jeder PDG einzeln für sich analysiert wird (Abb. 10):

1. In einer Schleife wird im SDG sequentiell nur nach den Anfangsknoten von Prozeduren gesucht.
2. Der vom Anfangsknoten einer Prozedur abgehende PDG wird rekursiv entlang seiner Kanten ausgewertet. Eventuelle Prozeduraufrufe brauchen nicht berücksichtigt zu werden.

Sobald GOTOs mit entsprechenden Labels auftauchen, müßten diese in einem weiteren Durchlauf durch den SDG separat behandelt werden.

Die Anfangsknoten einer Prozedur, einer Verzweigung oder einer Schleife verweisen auf den jeweiligen Kopf und Rumpf. Die Knoten des Kopfes können zunächst völlig vernachlässigt werden. Aus den Knoten des Rumpfes kann die Position (Zeile, Spalte) der ersten Anweisung ausgelesen werden, welche als Position für einen zu setzenden Breakpoint gilt. Verteilen sich die einzelnen Teile eines Programms über mehrere Dateien, müssen auch die entsprechenden Dateinamen ausgelesen und beim Setzen des Breakpoints mit angegeben werden. Nach dem Setzen des Breakpoints an diese Position, müssen die übrigen Knoten des Rumpfes daraufhin getestet werden, ob sie ihrerseits Anfangsknoten einer eingeschachtelten Struktur sind. Knoten des Rumpfes, die keine Anfangsknoten einer eingeschachtelten Struktur sind, können vernachlässigt werden, während diese Routine ansonsten mit dem Anfangsknoten der eingeschachtelten Struktur wieder (rekursiv) aufgerufen werden muß. Abbildung 11 gibt hierfür den entsprechenden Pseudocode an.

```
1   PROCEDURE main()
2   {
3     lies SDG ein
4
5     WHILE ( noch ungeteste Knoten )
6     {
7       IF ( Knoten == Anfangsknoten einer Prozedur )
8       {
9         ProcessNode( Knoten )
10      }
11
12      teste nächsten Knoten
13    }
14  }
```

Abbildung 10: Sequentielles Durchsuchen des SDG

```
1   PROCEDURE ProcessNode( Anfangsknoten )
2   {
3     Überspringe Knoten des Kopfes
4
5     Lies Position der ersten Anweisung aus Knoten des Rumpfes
6     Setze Breakpoint an die Position der ersten Anweisung
7
8     WHILE ( noch ungeteste Knoten des Rumpfes )
9     {
10      IF ( Knoten ==
11           Anfangsknoten eines Strukturierungsmechanismus )
12      {
13        ProcessNode( Knoten )
14      }
15
16      teste nächsten Knoten des Rumpfes
17    }
18  }
```

Abbildung 11: Rekursives Auswerten des PDG

Eine Schwierigkeit ergibt sich allerdings daraus, daß man für den `gdb` auf der Ebene des Sourcecodes einen Breakpoint nur auf den Zeilenanfang und nicht innerhalb einer Zeile setzen kann. Hierfür können, bezogen auf die Zeilennummern, drei Fälle unterschieden werden:

1. Letzte Zeile des Kopfes kleiner erster Zeile des Rumpfes (s. a. Abb. 12(a)).
2. Letzte Zeile des Kopfes gleich erster Zeile des Rumpfes und es gibt eine Zeile des Rumpfes, die größer ist als die letzte Zeile des Kopfes (s. a.

Abb. 12(b)). Die Zeilen einer eingeschachtelten Struktur, welche in der letzten Zeile des Kopfes beginnt, sind dabei von der Menge der Zeilen des Rumpfes auszunehmen (s. a. Abb. 12(c)).

3. Es gibt die vorgenannte Zeile nicht (s. a. Abb. 12(d) und Abb. 12(e)).

```
1  if ( c != d ) {
2    e=a+b;
3    f=a*b;
4  }
```

(a) Fall 1

```
1  if ( c != d ) { e=a+b;
2    f=a*b;
3  }
```

(b) Fall 2, 1. Teil

```
1  if ( a == b ) { if ( c != d ) {
2      e=a+b;
3      f=a*b;
4    }
5    g=f-e;
6  }
```

(c) Fall 2, 2. Teil

```
1  if ( c != d ) { e=a+b; f=a*b;
2  }
```

(d) Fall 3, 1. Teil (s. Fall 2)

```
1  if ( a == b ) { if ( c != d ) {
2      e=a+b;
3      f=a*b; } g=f-e;
4  }
```

(e) Fall 3, 2. Teil (s. Fall 2)

Abbildung 12: Zeilen zum Setzen eines Breakpoints

Solange im Sourcecode eines Programms nur der erste Fall vorkommt, reicht der bisher beschriebene Algorithmus vollkommen aus, weil dann von der Positionsangabe im Sourcecode nur die Zeilenangabe berücksichtigt werden braucht und die Spaltenangabe vernachlässigt werden kann.

Wenn aber die Möglichkeit besteht, daß auch die Fälle 2 und 3 vorkommen, was immer dann der Fall ist, wenn die letzte Zeile des Kopfes gleich der ersten Zeile des Rumpfes sein darf, muß der Algorithmus erweitert werden. Es gilt daher, eine unter 2. beschriebene Zeile ausfindig zu machen, wodurch der erste Fall mit abgedeckt wird. Eine Lösung besteht darin, die erste Zeile des Rumpfes zu suchen, die größer als die letzte Zeile des Kopfes ist und nicht zu den Zeilen des Rumpfes einer eingeschachtelten Struktur gehört. Sollte es keine solche Zeile geben, muß eine eventuell eingeschachtelte Struktur komplett mit in den Program-Execution-Trace aufgenommen werden, auch wenn deren Anweisungen des Rumpfes möglicherweise nicht ausgeführt wurden. Auf diesen Fall wird im Abschnitt 4.1.2 genauer eingegangen.

Um diesen Algorithmus bei der hier verwendeten Implementierung eines SDG, sowohl für Prozeduren als auch für Verzweigungen und Schleifen verwenden zu können, muß darauf eingegangen werden, daß der Anfangsknoten nicht immer direkt auf die Anweisungen verweist. Man muß daher die erste Zeile Sourcecode des vom Knoten abgehenden Unterbaums bestimmen, die größer als die letzte Zeile des Kopfes ist und nicht zum Rumpf einer Verzweigung oder einer Schleife gehört (Abb. 13). Sollte es keine solche Zeile geben, wird vom übergebenen Knoten die erste Zeile zurückgegeben.

Einen Spezialfall stellen IF-THEN-ELSE- und CASE-Verzweigungen dar, wo die Möglichkeit besteht, daß sich mehrere Rümpfe in einer Zeile befinden:

- Für eine IF-THEN-ELSE-Verzweigung soll die Eigenschaft des `gdb` ausgenutzt werden, daß er den Breakpoint auf den ersten Rumpf in einer Zeile bezieht, also den THEN-Rumpf. Für die Bestimmung einer geeigneten Zeile für den ELSE-Rumpf, braucht daher nur die Nummer der letzten Zeile des Kopfes auf die Nummer der letzten Zeile des THEN-Rumpfes gesetzt zu werden. Danach kann der ELSE-Rumpf genauso wie der THEN-Rumpf behandelt werden.

- Für CASE-Verzweigungen sollen die Anforderungen, an eine zum Setzen eines Breakpoints geeignete Zeile, dahingehend erweitert werden, daß diese Zeile nur genau einem Rumpf (s. a. Abschnitt 2.2) angehört.

Wie in Abbildung 14 dargestellt, ist es in der Programmiersprache C möglich mittels eines CASE (Zeile 4) in den Rumpf eines anderen CASE (Zeile 2–6) zu springen. Diese Konstellation soll hier als Programmierfehler verstanden werden, da es ohne weiteres möglich ist, für jedes CASE einen eigenen klar abgegrenzten Rumpf zu erstellen. In der Implementierung wird diese Konstellation nicht weiter berücksichtigt werden, da sie bereits bei der Erstellung des SDG angemahnt wird. Für `case 2` in Zeile 4 wird daher kein Breakpoint gesetzt.

4.1.2 Bestimmung der Knoten und Kanten

Die Breakpoints dienen dazu, Informationen darüber zu erlangen, welche Rümpfe von Prozeduren, Verzweigungen und Schleifen angesprungen wurden. In einem zweiten Schritt müssen diese Informationen den Anweisungen des Programms zugeordnet werden. Sequentielle Abarbeitung eines Programms bedeutet, daß, nachdem der Rumpf einer Prozedur, einer Verzweigung oder einer

```
1   PROCEDURE FirstGreaterRow( Anfangsknoten,
2                              Vorgegebene_Zeile )
3   {
4     //--- Rückgabewert initialisieren.
5     firstGreaterRow=Anfangsknoten.Erste_Zeile
6
7     WHILE ( noch ungeteste Knoten ) {
8       IF ( Knoten == Anfangsknoten einer Struktur ) {
9         firstSubTreeRow=Erste Zeile der Struktur
10      }
11      ELSE {
12        firstSubTreeRow=FirstGreaterRow( Knoten,
13                                         Vorgegebene_Zeile )
14      }
15
16      IF ( firstSubTreeRow > Vorgegebene_Zeile ) {
17        //--- Die erste Zeile des Unterbaums ist größer,
18        //    als die Vorgabe.
19        IF ( firstGreaterRow > Vorgegebene_Zeile ) {
20          //--- Die bisherige erste größere Zeile ist größer,
21          //    als die Vorgabe.
22          IF ( firstGreaterRow > firstSubTreeRow ) {
23            //--- Da die erste Zeile des Unterbaums
24            //    kleiner als die bisherige erste größere Zeile und
25            //    größer als die Vorgabe ist,
26            //    ist sie nun die erste größere Zeile.
27            firstGreaterRow=firstSubTreeRow
28          }
29        }
30        ELSE {
31          //--- Die bisherige erste größere Zeile
32          //    ist nicht größer als die Vorgabe.
33          //    Aber die erste Zeile des Unterbaums
34          //    ist größer als die Vorgabe.
35          firstGreaterRow=firstSubTreeRow
36        }
37      }
38
39      teste nächsten Knoten
40    }
41
42    RETURN firstGreaterRow
43  }
```

Abbildung 13: Erste größere Zeile bestimmen

```
1   switch ( ... ) {
2     case 1: {
3       ...
4       case 2:
5       ...
6     }
7     case 3: {
8       ...
9     }
10  }
```

Abbildung 14: GOTO der CASE-Verzweigung

Schleife angesprungen wurde, zunächst die erste Anweisung abgearbeitet wird, danach die zweite, die dritte usw. bis zur letzten Anweisung des Rumpfes. Unter der Voraussetzung, daß das Programm bis zum Ende abgearbeitet wurde, kann daraus gefolgert werden, daß alle Anweisungen des Rumpfes, die in Folge der ersten Anweisung des Rumpfes liegen, ausgeführt wurden, wenn der Rumpf angesprungen wurde. Bezogen auf eingeschachtelte Verzweigungen und Schleifen gilt dies aber nur für deren Kopf und nicht für die Anweisungen in deren Rumpf. Ob die Anweisungen des Rumpfes einer eingeschachtelten Struktur ausgeführt wurden, läßt sich aber wiederum daraus ersehen, ob der Breakpoint dieses Rumpfes erreicht wurde oder nicht. Eingeschachtelten Strukturen, für die kein Breakpoint gesetzt werden konnte, müssen dabei gesondert behandelt werden.

Die Aufgabe besteht nun darin festzustellen, welche Knoten und Kanten des SDG für die Anweisungen und Abhänigigkeiten des Program-Execution-Trace stehen. Bestimmt man für sämtliche Anweisungen des Program-Execution-Trace deren zugehörige Knoten des SDG, so erhält man die Menge, der für den Program-Execution-Trace benötigten Knoten. Da jedem Breakpoint eindeutig bestimmte Anweisungen zuordenbar sind und jeder Anweisung eindeutig bestimmte Knoten, können dadurch die für je eine Anweisung benötigten Knoten, eindeutig einem Breakpoint zugeordnet werden, wodurch man die für den Program-Execution-Trace benötigten Knoten bestimmt. Die Zuordnung der Knoten zu den Breakpoints kann erfolgen:

- Vor der Ausführung des Programms beim Setzen der Breakpoints, was den Nachteil hat, daß die Zuordnung auch für solche Breakpoints erfolgt, die später eventuell nicht erreicht werden. Der Vorteil liegt aber darin, daß bereits beim Erreichen eines Breakpoints die entsprechenden Knoten ausgegeben werden können und somit alle wesentlichen Funktionen in einem Programm vereinigt sind.

- Nach der Ausführung des Programms, was den Vorteil hat, daß die Zuordnung nur für solche Breakpoints erfolgt, die auch tatsächlich erreicht wurden. Der Nachteil liegt aber darin, daß die beiden wesentlichen Funktionen, Setzen der Breakpoints und Zuordnung der Breakpoints zu den

Anweisungen, auf zwei Programme verteilt werden müssen.

Die folgende Vorgehensweise bezieht sich auf eine Zuordnung der Breakpoints zu den Knoten vor der Ausführung des Programms. Sie müßte für eine Zuordnung nach der Ausführung des Programms nur geringfügig ergänzt werden.

Um die Vorgehensweise besser verstehen zu können, soll zunächst auf die zu setzenden Breakpoints eingegangen werden. Ziel ist es, daß nach Erreichen eines Breakpoints

1. die dem Breakpoint zugeordneten Knoten ausgegeben werden und
2. die Ausführung des Programms fortgesetzt wird.

Einem Breakpoint werden folgende Knoten zugeordnet:

- Knoten, die dem Kopf angehören (nur bei Prozeduren).
- Knoten der Anweisungen des Rumpfes, die keiner eingeschachtelten Struktur angehören.
- Knoten, die dem Kopf einer eingeschachtelten Struktur angehören.
- Knoten des Rumpfes einer eingeschachtelten Struktur, für die kein Breakpoint gesetzt werden konnte.

Zum Setzen der Breakpoints, bei gleichzeitiger Zuordnung der Knoten, kann nun wie folgt vorgegangen werden (s. a. Abb. 15, 16):

1. Die Analyse beginnt am Anfangsknoten einer Prozedur, einer Verzweigung oder einer Schleife, welcher auf den Kopf und den Rumpf verweist.
2. Ausgehend vom ersten Knoten des Kopfes, wird die letzte Zeile des Kopfes bestimmt.
3. Ausgehend vom ersten Knoten des Rumpfes, wird die erste Zeile des Rumpfes gesucht, die größer ist als die letzte Zeile des Kopfes und nicht zum Rumpf einer eingeschachtelten Struktur gehört.
4. Es muß unterschieden werden, ob eine solche Zeile gefunden werden konnte oder nicht:
   - Wenn eine solche Zeile gefunden werden konnte, muß
     (a) der Breakpoint gesetzt werden, müssen
     (b) die Knoten des Rumpfes, die nicht dem Rumpf einer eingeschachtelten Struktur angehören, dem Breakpoint zugeordnet werden und müssen
     (c) die Knoten der Rümpfe eingeschachtelter Strukturen, für welche keine eigenen Breakpoints gesetzt werden können, ebenfalls dem Breakpoint zugeordnet werden.
   - Wenn eine solche Zeile nicht gefunden werden konnte, müssen

   (a) die Knoten des Rumpfes, die nicht dem Rumpf einer eingeschachtelten Struktur angehören, der übergeordneten Struktur zugeordnet werden und müssen
   (b) die Knoten der Rümpfe eingeschachtelter Strukturen, für welche keine Breakpoints gesetzt werden können, ebenfalls der übergeordneten Struktur zugeordnet werden.

5. Um diese Vorgehensweise rekursiv mittels einer Routine realisieren zu können, muß unterschieden werden, nach welchen eingeschachtelten Strukturen im gegenwärtigen Durchlauf gesucht wird:
   (a) Für die ein eigener Breakpoint gesetzt werden kann oder
   (b) für die kein eigener Breakpoint gesetzt werden kann und die dem übergeordneten Breakpoint zugeordnet werden müssen.
6. Ausgehend von den Knoten des Rumpfes muß nach eingeschachtelten Strukturen gesucht werden, für die ein eigener Breakpoint gesetzt werden kann.

Das Heraussuchen der Anfangsknoten einer eingeschachtelten Struktur und die Zuordnung der Breakpoints zu den Knoten wird auf die Routinen `ProcessNode` (Abb. 15) und `Zuordnung` (Abb. 16) verteilt. Dabei wird mit einer indirekten Rekursion gearbeitet, indem von `ProcessNode` aus die Routine `Zuordnung` aufgerufen wird, welche ihrerseits wieder `ProcessNode` aufruft. Eine genauere Beschreibung der Implementierung dieser beiden Routinen ist unter Abschnitt 5.3 zu finden.

```
1  PROCEDURE ProcessNode( Knoten,
2                         Kein_eigener_Breakpoint_Durchlauf )
3  {
4    IF ( Knoten == Anfangsknoten einer Struktur ) {
5      Zuordnung( Knoten, Kein_eigener_Breakpoint_Durchlauf )
6    }
7    ELSE {
8      FOR ( alle Knoten abgehender Kanten ) {
9        ProcessNode( Knoten abgehender Kante,
10                    Kein_eigener_Breakpoint_Durchlauf )
11     }
12   }
13 }
```

Abbildung 15: Anfangsknoten einer eingeschachtelten Struktur heraussuchen

Bei dieser Vorgehensweise wurden drei Fälle noch nicht berücksichtigt, für die kein Pseudocode angegeben werden soll:

- Prozeduren:
  Bei Prozeduren müssen neben den Knoten des Rumpfes auch die Knoten

```
1  PROCEDURE Zuordnung( Knoten,
2                       Kein_eigener_Breakpoint_Durchlauf )
3  {
4    Bestimme letzte_Zeile_des_Kopfes
5
6    firstGreaterRow=FirstGreaterRow( Anfangsknoten des Rumpfes,
7                                     Letzte_Zeile_des_Kopfes )
8
9    IF ( firstGreaterRow > Letzte_Zeile_des_Kopfes ) {
10     IF ( !Kein_eigener_Breakpoint_Durchlauf ) {
11       Setze Breakpoint
12
13       Ordne Knoten des Rumpfes zu
14
15       //--- Knoten der Rümpfe eingeschachtelter Strukturen
16       //    ohne eigenem Breakpoint.
17       ProcessNode( Anfangsknoten des Rumpfes, TRUE)
18
19       Schließe Breakpoint ab
20
21       //--- Eingeschachtelte Strukturen mit eigenem Breakpoint.
22       ProcessNode( Anfangsknoten des Rumpfes, FALSE)
23     }
24   }
25   ELSE {
26     IF ( Kein_eigener_Breakpoint_Durchlauf ) {
27       Ordne Knoten des Rumpfes zu
28
29       //--- Knoten der Rümpfe eingeschachtelter Strukturen
30       //    ohne eigenem Breakpoint.
31       ProcessNode( Anfangsknoten des Rumpfes, TRUE)
32     }
33     ELSE {
34       //--- Eingeschachtelte Strukturen mit eigenem Breakpoint.
35       ProcessNode( Anfangsknoten des Rumpfes, FALSE)
36     }
37   }
38 }
```

Abbildung 16: Zuordnung der Breakpoints zu den Knoten

des Kopfes zugeordnet werden, da Prozeduren bei dieser Vorgehensweise keine übergeordnete Struktur haben, denen die Knoten des Kopfes zugeordnet werden könnten.

- REPEAT-UNTIL-Schleife:

Bei einer REPEAT-UNTIL-Schleife liegt das Prädikat hinter dem Rumpf. Es muß daher eine Zeile des Rumpfes gefunden werden, die kleiner als die erste Zeile des Prädikats ist und nicht dem Rumpf einer eingeschachtelten Struktur angehört oder gleich der letzten Zeile des Kopfes einer übergeordneten Struktur ist.

- IF-THEN-ELSE- und CASE-Verzweigungen:
  Hier müssen für einen Kopf mehrere Rümpfe berücksichtigt werden, die sich unter Umständen in einer Zeile befinden können. Pro Zeile darf aber nur ein Breakpoint gesetzt werden, der nur für genau einen Rumpf gilt und nicht für zwei.

In der praktischen Umsetzung ist darauf zu achten, daß jeder erreichte Breakpoint entfernt wird. Ansonsten würden bei mehrmaligem Aufruf einer Prozedur oder bei mehreren Schleifendurchläufen Redundanzen entstehen, die mit vermehrtem Aufwand wieder herausgefiltert werden müßten.

Wenn die Möglichkeit besteht, daß in einem Programm sowohl Strukturierungsmechanismen als auch GOTOs vorkommen können, sind Redundanzen nur schwer vermeidbar. Bei der vorgestellten Vorgehensweise, die sich nur auf Strukturierungsmechanismen bezieht, wird jeder Knoten maximal einem Breakpoint zugeordnet. Kommen GOTOs hinzu, kann ein Knoten aber zwei Breakpoints zugeordnet werden, nämlich neben dem Breakpoint für den Strukturierungsmechanismus, auch dem Breakpoint für das Label des GOTOs. In diesem Fall ist es sicherlich einfacher, mehrfach vorkommende Knoten herauszufiltern.

4.2 Approximate Dynamic Slice

Von ausgeführten Prozeduren, Verzweigungen und Schleifen sollte der Program-Execution-Trace nun mindestens folgende Knoten enthalten, wenn im jeweiligen Rumpf ein Breakpoint gesetzt werden konnte:

- Prozeduren:
  Alle Knoten des Kopfes und alle Knoten des Rumpfes, die nicht dem Rumpf einer eingeschachtelten Struktur angehören. Wurde der Breakpoint im Rumpf einer Prozedur nicht erreicht, fielen alle Knoten des Kopfes und des Rumpfes weg.

- Verzweigungen:
  Alle Knoten des Rumpfes, die nicht dem Rumpf einer eingeschachtelten Struktur angehören. Die Knoten des Kopfes wurden schon von der übergeordneten Struktur berücksichtigt. Wurde ein Breakpoint in den Rümpfen einer Verzweigung nicht erreicht, fielen alle Knoten des jeweiligen Rumpfes weg.

- Schleifen:
  Alle Knoten des Rumpfes, die nicht dem Rumpf einer eingeschachtelten Struktur angehören. Die Knoten des Kopfes wurden schon von der übergeordneten Struktur berücksichtigt. Wurde der Breakpoint im Rumpf einer Schleife nicht erreicht, fielen alle Knoten des Rumpfes weg.

Von Prozeduren, Verzweigungen und Schleifen sollte der Program-Execution-Trace folgende Knoten enthalten, wenn im jeweiligen Rumpf kein Breakpoint gesetzt werden konnte und die Knoten dem Breakpoint der übergeordneten Struktur zugeordnet werden mußten:

- Prozeduren:
  Alle Knoten des Kopfes und alle Knoten des Rumpfes, die nicht dem Rumpf einer eingeschachtelten Struktur angehören.

- Verzweigungen und Schleifen:
  Alle Knoten des Rumpfes, die nicht dem Rumpf einer eingeschachtelten Struktur angehören. Die Knoten des Kopfes wurden schon von der übergeordneten Struktur berücksichtigt.

Den Approximate Dynamic Slice eines Slicing-Kriteriums erhält man, indem man für das Slicing-Kriterium, auf der Grundlage des Program-Execution-Trace, den Static Slice berechnet. Zur Berechnung des Static Slice dürfen also nur solche Knoten und Kanten des SDG herangezogen werden, die zu den Knoten und Kanten des Program-Execution-Trace gehören. Verfahren zur Berechnung eines Static Slice werden beispielsweise in [BG96] und [Tip94] vorgestellt und werden daher hier nicht weiter behandelt.

5 Implementierung

In diesem Abschnitt wird auf die Implementierung des im vorigen Abschnitt beschriebenen Algorithmus zur Ermittlung eines Program-Execution-Trace eingegangen. Das Testen der Kanten und die letztendliche Berechnung des Approximate Dynamic Slice wird hier nicht implementiert, da dies mit wenig Aufwand anhand der bereits bestehenden Routinen erfolgen kann. Neben den Strukturen und Abhängigkeiten der benötigten Datenstrukturen und Routinen, sollten auch deren Voraussetzungen und Einschränkungen beachtet werden.

5.1 System Dependence Graph

Als Grundlage für die Berechnung eines Approximate Dynamic Slice wird ein System Dependence Graph herangezogen. Daher soll in diesem Abschnitt eine überblicksartige Beschreibung der verwendeten Knoten und Kanten, des im Verbundprojekt VALSOFT implementierten SDG, gegeben werden. Dieser Überblick basiert auf den Artikeln [Bru97, Kön98], auf der Dokumentation zum VALSOFT-Projekt [Kri97] und auf den Beschreibungen im Sourcecode.

Anfangsknoten von Prozeduren, Verzweigungen und Schleifen werden an ihrer Operation (oper-Attribut) erkannt:

Struktur	Operation
Prozedur	`entry`
Verzweigung	`IF`
Schleife	`loop`

Tabelle 1: Anfangsknoten

Beim Durchwandern des PDG werden nur Kanten für unbedingte Kontrollabhängigkeiten (unconstrained control dependence — UN; vormals: IN) weiterverfolgt. Andere Kanten sollen bei dieser Beschreibung als nicht existent gelten.

Vom Anfangsknoten einer Prozedur zeigen maximal zwei Kanten auf den Kopf und eine Kante auf den Rumpf. In der Liste der Kanten muß die Kante auf den Rumpf als letztes stehen. Die jeweiligen Teile einer Prozedur können an der Art des ersten Knotens, auf den die Kanten zeigen, festgestellt werden:

Teil	Knotenart
Kopf	`FormIn`, `FormOut`
Rumpf	sonst

Tabelle 2: Teile einer Prozedur

Verzweigungen müssen zunächst anhand des Wertes (value-Attribut) des Anfangsknotens in IF-THEN(-ELSE)-Verzweigungen und CASE-Verzweigungen unterschieden werden:

Konstrukt	Wert
IF-THEN(-ELSE)	""
CASE	`switch`

Tabelle 3: Verzweigungen

Vom Anfangsknoten einer IF-THEN(-ELSE)-Verzweigung zeigen maximal eine Kante auf den Kopf und zwei Kanten auf die Rümpfe, für den THEN- und den ELSE-FALL. Die Kanten auf die Rümpfe müssen als letztes stehen.

Vom Anfangsknoten einer CASE-Verzweigung zeigt maximal eine Kante auf den Kopf und eine zweite Kante auf einen Knoten, dessen Kanten auf die einzelnen CASE-Fälle zeigen. Dabei wurden solche CASE-Fälle nicht aufgenommen, deren vorhergehender CASE-Fall nicht mit BREAK beendet wurde. Wurde ein CASE-Fall mit einem BREAK beendet, dann kann der Knoten für das BREAK anhand des Wertes (value-Attribut) "break" erkannt werden.

Die jeweiligen Teile einer Verzweigung können an der Art des ersten Knotens, auf den die Kanten zeigen, festgestellt werden:

Teil	Knotenart
Kopf	`Pred`
Rumpf	sonst

Tabelle 4: Teile einer Verzweigung

Schleifen müssen zunächst anhand des Wertes (value-Attribut) des Anfangsknotens in WHILE-, REPEAT-UNTIL- und FOR-Schleifen unterschieden werden:

Schleife	Wert
WHILE	`while`
REPEAT-UNTIL	`do`
FOR	`for`

Tabelle 5: Schleifen

Vom Anfangsknoten einer WHILE- und einer REPEAT-UNTIL-Schleife zeigt maximal eine Kante auf den Kopf und eine Kante auf den Rumpf. In der Liste der Kanten muß die Kante auf den Rumpf zuletzt stehen. Die jeweiligen Teile dieser Schleifen können an der Art des ersten Knotens, auf den die Kanten zeigen, festgestellt werden:

Teil	Knotenart
Kopf	`Pred`
Rumpf	sonst

Tabelle 6: Teile einer WHILE- und einer REPEAT-UNTIL-Schleife

Vom Anfangsknoten einer FOR-Schleife zeigen maximal zwei Kanten auf den Kopf, das Prädikat und den Zählvorgang und eine Kante auf den Rumpf. In der Liste der Kanten muß die Kante auf den Rumpf zuletzt stehen. Die jeweiligen Teile dieser Schleife können anhand des Wertes des ersten Knotens, auf den die Kanten zeigen, festgestellt werden:

Teil	Wert
Kopf	$\neq$ "`(for-body)`"
Rumpf	`(for-body)`

Tabelle 7: Teile einer FOR-Schleife

5.2 Breakpoints

Der `gdb` erlaubt es, bei Erreichen eines Breakpoints bestimmte Befehle auszuführen. Nach dem Setzen des Breakpoints durch ein

```
break <Dateiname>:<Zeilennummer>
```

können die Befehle mittels

```
commands
<Erster Befehl>
...
<Letzter Befehl>
end
```

an den Breakpoint angehängt werden. Für die hier vorgestellte Implementierung sind das die Befehle:

1. `silent`

   um unnötige Ausgaben zu unterdrücken.

2. ```
   printf "Node: <Nummer des ersten Knotens>\n"
   ...
   printf "Node: <Nummer des letzten Knotens>\n"
   ```

   für die Ausgabe der Nummern der dem Breakpoint zugeordneten Knoten.

3. `disable breakpoints <Nummer des Breakpoints>`

   damit jeder Breakpoint nur einmal ausgeführt wird. Ansonsten würden die Knoten bei mehrmaligem Aufruf einer Prozedur oder bei mehreren Schleifendurchläufen mehrfach ausgegeben. Diese Informationen wären allerdings redundant und würden zu vermehrtem Aufwand führen.

4. `continue`

   da die Analyse des Programms automatisch erfolgt, muß es nach Erreichen des Breakpoints und Ausgabe der zugeordneten Knoten, automatisch fortgesetzt werden.

Zusammengesetzt ergibt sich damit folgendes Bild:

```
break <Dateiname>:<Zeilennummer>
commands
silent
printf "Node: <Nummer des ersten Knotens>\n"
...
printf "Node: <Nummer des letzten Knotens>\n"
disable breakpoints <Nummer des Breakpoints>
continue
end
```

Anfangs- und Endknoten des SDG, die dem Einstieg in das Programm dienen, sowie Knoten von globalen Variablen werden unabhängig von einem Breakpoint ausgegeben. Ebenso Knoten solcher Prozeduren, für die kein Breakpoint gesetzt werden konnte. Den Abschluß bildet der Befehl

```
run
```
```

wodurch der `gdb` das Programm startet und die Nummern der Anfangs- und Endknoten des SDG, der Knoten von Prozeduren ohne Breakpoint, sowie der Knoten erreichter Breakpoints ausgegeben werden.

5.3 Routinen

Die verwendete Programmiersprache ist C++, wobei wegen der Kürze des Programms auf eine explizite Einteilung in verschiedene Klassen verzichtet wurde.

Die Ein- und Ausgaben erfolgen über StdIn und StdOut, wobei der SDG für das zu untersuchende Programm übergeben werden muß und die Befehle für den `gdb` zurückgegeben werden.

In der Routine `main` wird der SDG in seine einzelnen PDGs zerlegt. Mit dem Anfangsknoten des PDG wird `ProcessNode` aufgerufen (s. a. Abb. 17).

In `ProcessNode` wird unterschieden, ob es sich bei dem übergebenen Knoten um einen Anfangsknoten handelt oder nicht:

- Je nachdem, um welche Art von Anfangsknoten es sich handelt, werden für die Zuordnung der Breakpoints zu den Knoten die Routinen `ProcessEntryNode`, `ProcessIfNode` oder `ProcessLoopNode` für Prozeduren, Verzweigungen oder Schleifen aufgerufen.

- Handelt es sich nicht um einen Anfangsknoten, werden die Knoten, auf die der übergebene Knoten zeigt, rekursiv mittels `ProcessNode` ebenfalls überprüft.

In den einzelnen Routinen zum Zuordnen der Breakpoints zu den Knoten wird die Liste der vom Anfangsknoten abgehenden Kanten durchgegangen (s. a. Abb. 16):

1. Zeigt eine Kante auf einen Anfangsknoten des Kopfes, wird dieser Knoten der Routine `SubTreeEndRow` übergeben, welche die letzte Zeile des von diesem Knoten abgehenden Unterbaums ermittelt. Ist sie größer als die bisherige, als letzte Zeile des Kopfes ermittelte Zeile `headerEndRow`, so wird sie als neue letzte Zeile des Kopfes gesetzt.

2. Zeigt eine (die letzte) Kante auf den Anfangsknoten des Rumpfes, wird zunächst mit `FirstGreaterRow` versucht eine Zeile für einen Breakpoint herauszufinden.

   - Wenn dies gelingt und in diesem Durchlauf ein Breakpoint gesetzt werden soll, werden
     (a) in `OutputBreakpointHead` der Breakpoint gesetzt und mit `OutputSubTreeNodeNumbers` die Knoten des Rumpfes zugeordnet,
     (b) mit `ProcessNode` die Knoten eingeschachtelter Strukturen für die kein eigener Breakpoint gesetzt werden kann, ebenfalls diesem Breakpoint zugeordnet und
     (c) wird mit `OutputBreakpointTail` der Breakpoint abgeschlossen.
     (d) Wird mit `ProcessNode` nach eingeschachtelten Strukturen gesucht, für die ein eigener Breakpoint gesetzt werden kann.

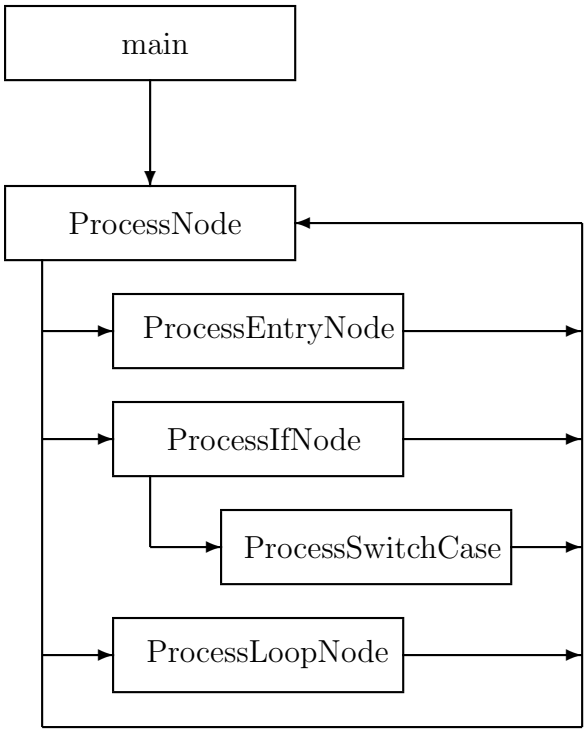

Abbildung 17: Aufrufabhängigkeiten

- Wenn dies nicht gelingt und in diesem Durchlauf kein Breakpoint gesetzt werden soll, werden
  (a) mit `OutputSubTreeNodeNumbers` die Knoten des Rumpfes der übergeordneten Struktur zugeordnet und
  (b) mit `ProcessNode` die Knoten eingeschachtelter Strukturen, für die kein eigener Breakpoint gesetzt werden kann, ebenfalls der übergeordneten Struktur zugeordnet.

  Wenn dies nicht gelingt und in diesem Durchlauf ein Breakpoint gesetzt werden soll,
  (a) wird mit `ProcessNode` nach eingeschachtelten Strukturen gesucht, für die ein eigener Breakpoint gesetzt werden kann.

Da Prozeduren die oberste Struktur darstellen, entfällt in `ProcessEntryNode` die Unterscheidung, welcher Durchlauf gerade stattfindet. Vielmehr erfolgt hier die eigentliche Aufspaltung in zwei Durchläufe.

Bei Prozeduren muß deshalb die Zuordnung der Knoten ab deren Anfangsknoten beginnen und nicht, wie bei eingeschachtelten Strukturen, ab dem ersten Knoten des Rumpfes.

In der Variablen `breakpoints` wird die aktuelle Anzahl der Breakpoints durchgereicht. Sie wird zum Entfernen eines Breakpoints, nachdem dieser erreicht wurde, benötigt.

5.4 Ausgaben

Dem `gdb` müssen neben dem zu untersuchenden Programm die erstellten Befehle der Breakpoints übergeben werden. Während des Einlesens der Befehle und beim nachfolgenden Programmlauf werden zahlreiche Ausgaben getätigt. Sie sind Zeilenweise aufgebaut und können

- Statusmeldungen des `gdb` sein,
- von den den Breakpoints angehängten Befehlen herrühren oder
- vom Programm selbst stammen.

Zur Ermittlung des Program-Execution-Trace sind letztendlich nur die Nummern der den Breakpoints zugeordneten Knoten interessant. Wie aus Abschnitt 5.2 über die Breakpoints hervorgeht, lautet der, einem Breakpoint angehängte Befehl zur Ausgabe der Nummer eines dem Breakpoint zugeordneten Knotens,

```
printf "Node: <Nummer des Knotens>\n"
```

so daß

```
Node: <Nummer des Knotens>
```

ausgegeben wird. „Node:“ stellt dabei eine Kennung dar, um die Zeile für die Nummer eines Knotens identifizieren zu können. Alle Zeilen, die nicht mit dieser Kennung beginnen, können beim Einlesen der Ausgaben zur Ermittlung des Program-Execution-Trace als irrelevant aussortiert werden. Selbstverständlich kann diese Kennung den eigenen Erfordernissen nach modifiziert werden, um beispielsweise Kollisionen mit den Ausgaben des zu untersuchenden Programms zu vermeiden.

5.5 Korrektheit

Für jeden Algorithmus muß die Frage gestellt werden, ob er prinzipiell seine Aufgabe erfüllt und demzufolge ebenso die Frage, ob er versagen kann und unter welchen Umständen dies geschehen könnte. Diese Frage ist umso bedeutender, als daß diese Software als ein Hilfsmittel zur Software-Validierung dienen soll.

Für die hier vorgestellte Implementierung ist von Bedeutung, daß durch die möglicherweise auftretenden Fehler entweder

- unnötig Anweisungen in den Slice aufgenommen werden oder
- benötigte Anweisungen im Slice fehlen.

Während zu viele Anweisungen im Slice wohl nur die Genauigkeit beeinträchtigen, könnten bei zu wenigen ausgerechnet diejenigen Anweisungen fehlen, die bei einem Backward Slice einen ungewünschten Einfluß auf das Slicing-Kriterium haben.

Es sollte daher versucht werden, mögliche Fehler anhand der Nichterfüllung bestimmter Bedingungen zu erkennen, um entsprechend reagieren zu können:

1. Nach dem Erstellen der Befehle für den `gdb` kann z. B. überprüft werden, ob
   - jeder Knoten des SDG genau einmal zugeordnet wurde und
   - die Anzahl der jeweiligen Anfangsknoten für Prozeduren, Verzweigungen und Schleifen, gleich der Summe der jeweils gesetzten und nicht gesetzten Breakpoints ist.
2. Nach der Bestimmung der Knoten und Kanten des Program-Execution-Trace bzw. nach der Berechnung des Approximate Dynamic Slice kann überprüft werden, ob die Knoten zusammenhängend sind.

Eventuell auftretende Unregelmäßigkeiten sollten auf jeden Fall eine genauere Untersuchung wert sein. Sollten keine Unregelmäßigkeiten auftreten, ist dies nur ein Indiz für Fehlerfreiheit, aber keine Garantie.

Es muß daher gewarnt werden, daß die hier gewählte Implementierung der vorgestellten Vorgehensweise zur Berechnung eines Approximate Dynamic Slice, keine 100%-ige Korrektheit der Ergebnisse garantieren kann!

6 Anwendung

6.1 Vorgehensweise

Der hier verwendete SDG kann bisher nur für C-Quellen erstellt werden. Da zur Implementierung der vorgestellten Vorgehensweise Konstrukte aus C++ verwendet wurden, ist eine Selbstanwendung noch nicht möglich.

Zur Ermittlung des Approximate Dynamic Slice müssen sechs Schritte durchgeführt werden. Dafür werden die Tools

- `g++` (GNU-Projekt, Version 2.8.1),
- `gdb` (GNU-Projekt, Version 4.16.patched),
- `mess` (VALSOFT-Projekt) und
- `ads` (Implementierung der vorgestellten Vorgehensweise)

benötigt:

1. Übersetzen des zu analysierenden Programms mit der Debug-Option:

   ```
   g++ -Wall -g program.c -o program
   ```

2. Den SDG erzeugen:

   ```
   g++ -E -nostdinc program.c >program.i
   mess -p program program.i
   ```

3. Die Befehle für den Debugger erstellen:

   ```
   ads <program.pdg >program.ads
   ```

4. Den Debugger starten:

```
gdb program --command=program.ads --batch
```

5. Die Ausgaben einlesen und die Knoten des Program-Execution-Trace im SDG markieren (hier nicht implementiert).

6. Auf der Grundlage der markierten Knoten des SDG den Static Slice für das Slicing-Kriterium berechnen, um den Approximate Dynamic Slice zu erhalten (hier nicht implementiert).

6.2 Geschwindigkeit

In diesem Abschnitt soll die Frage erläutert werden, in welchem Maße die Ausführungsgeschwindigkeit eines Programms durch das Setzen von Breakpoints und deren Verarbeitung verringert wird. Dazu wurden auf der Basis eines Intel Pentium Prozessors mit 75 MHz Taktfrequenz einige Geschwindigkeitstests durchgeführt. Für diesen Zweck wurden drei Beispielprogramme erstellt:

- Die ersten beiden Beispielprogramme (Abb. 18, 19) dienten der Untersuchung, welche Zeit das Setzen von Breakpoints beansprucht.
- Anhand des dritten Beispielprogramms (Abb. 20) wurde untersucht, inwieweit sich die Ausführungsgeschwindigkeit eines Programms durch die Ausführung im `gdb` ändert und wovon die Verarbeitungsgeschwindigkeit eines erreichten Breakpoints abhängt.

Um die Geschwindigkeiten besser miteinander vergleichen zu können wurde jeweils eine ganze Versuchsreihe erzeugt.

Die zum Setzen von Breakpoints benötigte Zeit wurde anhand von zwei Beispielprogrammen (Abb. 18, 19) jeweils mittels mehrerer Versuchsreihen untersucht. In diesen Versuchsreihen wurde jeweils für beide Beispielprogramme gemessen,

- wieviel Zeit der `gdb` zum Laden eines Beispielprogramms benötigt, ohne daß Breakpoints gesetzt werden (Tab. 8, 9: 2. Spalte),
- wieviel Zeit der `gdb` zum Laden eines Beispielprogramms und anschließendem Setzen von Breakpoints benötigt, ohne daß an die Breakpoints `printf` Befehle angehängt wurden (Tab. 8, 9: 4. Spalte) und
- wieviel Zeit der `gdb` zum Laden eines Beispielprogramms und anschließendem Setzen von Breakpoints benötigt, wenn an jedem Breakpoint 11 bzw. 5 `printf` Befehle angehängt wurden (Tab. 8, 9: 6. Spalte).

Anzumerken sei, daß der zweiten und dritten Versuchsreihe der Umstand zugrunde liegt, daß sich die Breakpoints, bei der hier vorgestellten Vorgehensweise des Approximate Dynamic Slicing, nur in der Anzahl ihrer `printf` Befehle unterscheiden. Weiterhin wurde analog zu jeder der drei vorgenannten Versuchsreihen gemessen, wieviel Zeit insgesamt benötigt wurde, wenn die Beispielprogramme anschließend auch noch ausgeführt wurden (Tab. 8, 9: 3., 5.,

7. Spalte). Die Beispielprogramme wurden für diesen Zweck so konzipiert, daß jeder Breakpoint erreicht wird (worst-case). Um die beanspruchten Zeiten besser einordnen zu können wurde zusätzlich die vom Programm `mess` benötigte Zeit zum Erzeugen der entsprechenden SDGs gemessen.

Das erste Beispielprogramm (Abb. 18) besteht aus einer Hauptroutine, von der aus (Zeile 13) unterschiedlich viele Unterfunktionen (Zeile 1–5) aufgerufen werden. Dadurch soll untersucht werden, welchen Einfluß die Anzahl der Unterfunktionen und der dafür zu setzenden Breakpoints auf die benötigte Zeit hat.

```
1   int function<NUMMER>(int a)
2   {
3     a=a+1;
4     return a;
5   }
6
7   <WIEDERHOLUNGEN DER ZEILEN 1-5>
8
9   int main()
10  {
11    int c=0;
12
13    c=c+function<NUMMER>(c);
14
15    <WIEDERHOLUNGEN DER ZEILE 13>
16
17    return c;
18  }
```

Abbildung 18: 1. Beispielprogramm für Geschwindigkeitstests

Das zweite Beispielprogramm (Abb. 19) besteht im wesentlichen aus einer Verzweigung (Zeile 7–10), die unterschiedlich oft wiederholt wird. In Anlehnung an das erste Beispielprogramm soll dadurch untersucht werden, welchen Einfluß die Anzahl der Breakpoints innerhalb einer Routine auf die benötigte Zeit hat und ob es eventuell signifikante Unterschiede im Vergleich zum ersten Beispielprogramm gibt.

Aus dem Vergleich der Versuchsreihen für das erste Beispielprogramm (Tab. 8) und der Versuchsreihen für das zweite Beispielprogramm (Tab. 9) gehen im wesentlichen die folgenden Ergebnisse hervor:

- Das bloße Laden eines Programms in den Debugger `gdb` und das anschließende Setzen der Breakpoints nimmt erheblich weniger Zeit in Anspruch (s. `quit`-Spalten), als die Verarbeitung der Breakpoints, wenn sie erreicht wurden (s. `run`-Spalten).

- Das Laden eines Programms und anschließende Setzen der Breakpoints ist weitaus weniger zeitintensiv (s. `quit`-Spalten), als das Erzeugen des

```
1   int main()
2   {
3     int f=0;
4     int t=1;
5     int c=0;
6
7     if ( f != t )
8     {
9       c = c + 1;
10    }
11
12    <WIEDERHOLUNGEN DER ZEILEN 7-10>
13
14    return c;
15  }
```

Abbildung 19: 2. Beispielprogramm für Geschwindigkeitstests

zugehörigen SDGs (s. `mess`-Spalten).

Der Versuch von 1024 Verzweigungen für das zweite Beispielprogramm (Tab. 9, letzte Zeile) wurde nach einer halben Stunde abgebrochen, da kein Ende der Berechnungen durch das Programm `mess` abzusehen war.

Unterfunktionen	Zeit [sec]						
	ohne Breakpoints		mit Breakpoints				`mess`
			0 × `printf`		11 × `printf`		
	`quit`	`run`	`quit`	`run`	`quit`	`run`	
32	1	3	1	4	1	4	1
64	1	3	1	4	1	5	4
128	1	3	1	5	1	6	7
256	1	3	1	10	1	13	15
512	1	4	2	27	2	34	33
1024	1	4	4	98	6	116	75

Tabelle 8: Zeiten zum Laden und Setzen von Breakpoints bei Funktionen

Das dritte Beispielprogramm (Abb. 20) besteht aus einer einfachen Schleife, in deren Rumpf ein Zähler inkrementiert wird. Die Versuchsreihen wurden bei diesem Beispielprogramm durch die Erhöhung der Schleifendurchläufe erzeugt.

Zunächst wurden anhand des dritten Beispielprogramms die beiden folgenden Versuchsreihen durchgeführt:

- Für die erste Versuchsreihe wurde die reine Ausführungszeit des Beispielprogramms ohne eine Beeinträchtigung durch den `gdb` gemessen (Tab. 10, mittlere Spalte).

Verzweigungen	Zeit [sec]						
	ohne Breakpoints		mit Breakpoints				`mess`
			0 × `printf`		5 × `printf`		
	`quit`	`run`	`quit`	`run`	`quit`	`run`	
32	1	3	1	4	1	4	1
64	1	3	1	4	1	4	3
128	1	3	1	5	1	6	7
256	1	3	1	9	1	11	28
512	1	4	2	27	2	28	151
1024							> 1800

Tabelle 9: Zeiten zum Laden und Setzen von Breakpoints bei Verzweigungen

```
1   int main()
2   {
3     long i;
4     long t = 0;
5
6     for ( i = 0; i < <SCHLEIFENDURCHLÄUFE>; i++ )
7     {
8       t = t + 1;
9     }
10
11    return t;
12  }
```

Abbildung 20: 3. Beispielprogramm für Geschwindigkeitstests

- Für die zweite Versuchsreihe wurde innerhalb der Schleife ein Breakpoint gesetzt (Tab. 10, rechte Spalte). Wie in dieser Studienarbeit beschrieben wurde er deaktiviert, nachdem er das erste Mal erreicht wurde.

Die Zeit für einen Schleifendurchlauf unter Beeinflussung des `gdb` kann als die rein zum Aufruf des `gdb` benötigte Zeit betrachtet werden. Sie beträgt daher ca. 3 Sekunden. Betrachtet man die Ausführungszeiten ohne und mit Beeinflussung durch den `gdb` für 10.000.000, 100.000.000 und 1.000.000.000 Schleifendurchläufe, so ist der Betrag der Differenz jeweils in etwa so groß, wie die reine Aufrufzeit des `gdb`. Demnach werden die Ausführungszeiten von deaktivierten Breakpoints nicht beeinträchtigt.

Im Vergleich zu den vorigen Versuchsreihen ist die Ausführungszeit von maximal 10.000 Schleifendurchläufen gegenüber der von 10.000.000 Schleifendurchläufen bzw. der Aufrufzeit des `gdb` verschwindend gering. Demnach kann die reine Ausführungszeit des Beispielprogramms für die folgenden Versuchsreihen vernachlässigt werden. Dafür soll die Verarbeitungsgeschwindigkeit eines Breakpoints untersucht werden. Folglich mußte der Befehl zur Deaktivierung eines Breakpoints entfernt werden, so daß der Breakpoint bei jedem Schleifen-

Schleifendurchläufe	Zeit [sec]	
	ohne `gdb`	mit `gdb`
1	<1	3
10.000.000	1	5
100.000.000	12	16
1.000.000.000	121	125

Tabelle 10: Ausführungsgeschwindigkeiten eines Programms

durchlauf erreicht wurde. Die Breakpoints selbst unterscheiden sich nur in der Anzahl ihrer `printf` Befehle, weshalb

- eine Versuchsreihe ohne `printf` Befehle (Tab. 11, 2. Spalte),
- eine mit 10 `printf` Befehlen und 1-facher Stringlänge (Tab. 11, 3. Spalte) und
- eine mit einem `printf` Befehl und 10-facher Stringlänge (Tab. 11, 4. Spalte)

durchgeführt wurde.

Es zeigt sich, daß die Verarbeitungsgeschwindigkeit eines Breakpoints wesentlich von der Anzahl der `printf` Befehle abhängig ist. Für 10 Stück ist das ungefähr der Faktor 2,5. Weiterhin kann sich die Ausführungszeit des Beispielprogramms allein anhand gesetzter Breakpoints um den Faktor 10.000 verringern. Dies ist sicherlich ein Extrembeispiel, da durch den Breakpoint das Beispielprogramm so komplex wird, daß es wahrscheinlich nicht mehr in den internen Cache des Prozessors paßt. Bei einem längeren Programm, welches von vornherein nicht in den Cache paßt, müßte der Faktor wesentlich geringer ausfallen. Anhand der dritten Versuchsreihe ist zu erkennen, daß sich die Verarbeitungsgeschwindigkeit eines Breakpoints erhöhen ließe, wenn sämtliche `printf` Befehle zu einem zusammengefaßt würden.

Schleifendurchläufe	Zeit [sec]		
	0 × `printf`	10 × `printf` 1 × Stringlänge	1 × `printf` 10 × Stringlänge
1	3	3	3
10	3	3	3
100	4	4	4
1.000	5	8	5
10.000	17	42	23

Tabelle 11: Verarbeitungsgeschwindigkeiten der Breakpoints

7 Zusammenfassung und Ausblick

Nach einer Darstellung der Grundlagen des Slicings wurde eine Vorgehensweise zur Berechnung eines Approximate Dynamic Slice vorgestellt, wobei der Schwerpunkt auf der Ermittlung des Program-Execution-Trace lag. Da die Vorgehensweise auf einem SDG basiert, hängt sie nicht von einer bestimmten Programmiersprache ab.

Drei wesentliche Vorteile würden sich ergeben, wenn ein Debugger die Möglichkeit bieten würde, einen Breakpoint auch innerhalb einer Zeile zu setzen, mehrere Breakpoints in einer Zeile eingeschlossen:

- Die Genauigkeit des Program-Execution-Trace würde erhöht, da für alle Strukturierungsmechanismen ein Breakpoint gesetzt werden könnte. Es müßten keine Rümpfe von Strukturierungsmechanismen mehr aufgenommen werden, nur weil sich für sie, wegen des ungünstigen Aufbaus des Sourcecodes, kein eigener Breakpoint setzen läßt.

- Zum Erzeugen der Breakpoints muß der SDG bisher nach solchen Strukturierungsmechanismen durchsucht werden, für die sich ein Breakpoint setzen läßt und nach solchen, für die das nicht möglich ist. Würde diese Unterscheidung entfallen, könnte sich die zum Erzeugen der Breakpoints benötigte Zeit in etwa halbieren.

- Die Implementierung könnte vereinfacht werden.

Würden erreichte Breakpoints nicht deaktiviert werden, enthielte der daraus resultierende Program-Execution-Trace zusätzlich die genaue Reihenfolge der Anweisungen und die Anzahl ihrer Ausführung. Dadurch könnte dieser als Grundlage für die Berechnung eines Exact Dynamic Slice dienen.

Zur Bestimmung eines Program-Execution-Trace können ebenfalls die Programme `gcov` und `gprof` betrachtet werden. Mit ihrer Hilfe kann zum einen ermittelt werden, wie oft jede Zeile des Sourcecodes ausgeführt wurde und zum anderen, wieviel Zeit die einzelnen Routinen in Anspruch genommen haben. Aus diesen Informationen ließe sich ein Program-Execution-Trace in der Art bestimmen, daß alle Anweisungen eines Programms nicht zum Program-Execution-Trace gehören würden, die in Zeilen stehen, welche 0-mal ausgeführt wurden. Entsprechend gehörten Routinen nicht zum Program-Execution-Trace, die keine (0) Zeit beanspruchten.

Abschließend bleibt zu sagen, daß Approximate Dynamic Slicing genau auf die Mitte des Spannungsbogens zwischen benötigter Genauigkeit und vertretbarem Aufwand zielt. Somit erhält der Software-Entwickler ein weiteres Werkzeug für den täglichen Einsatz an die Hand, was die Anzahl seiner Möglichkeiten vergrößert.

Literatur

[ADS90] Hiralal Agrawal, Richard A. DeMillo, Eugene H. Spafford. *Efficient Debugging with Slicing and Backtracking.* Technical Report SERC-TR-80-P, Software Engineering Research Center, Purdue University, October 1990.

[AH90] Hiralal Agrawal, Joseph R. Horgan. Dynamic Program Slicing. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, ACM Press, New York, pp. 246–256, June 1990.

[ASU88] Alfred V. Aho, Ravi Sethi, Jeffry D. Ullman. *Compilerbau.* Addison-Wesley, 1988.

[BG96] David W. Binkley, Keith Brian Gallagher. *Program Slicing.* To appear in Advances in Computers, 1996.

[Bru97] Christian Bruns. *Constant-Propagation im Program-Dependence-Graph.* Studienarbeit, TU Braunschweig, Dezember 1997.

[HRB90] Susan B. Horwitz, Thomas W. Reps and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[Kön98] Torsten Königshagen. *Konstruktion von Call-Multigraphen mit Hilfe von Abhängigkeitsgraphen.* Studienarbeit, TU Braunschweig, Januar 1998.

[Kri97] Jens Krinke. *System-Dependence-Graphs.* Programmdokumentation, 10. April 1997.

[KS98] Jens Krinke und Gregor Snelting. *Validation of measurement software as an application of slicing and constraint solving.* Informatikbericht 98-01, TU Braunschweig, Januar 1998.

[KSR] Jens Krinke, Gregor Snelting, Torsten Robschink. *Software-Sicherheitsprüfung mit VALSOFT.* TU Braunschweig, 1998.

[Tip94] Frank Tip. *A survey of program slicing techniques.* Technical Report CS-R9438, Centrum voor Wiskunde en Informatica, July 1994.

[Wei84] Mark Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[ZK97] Andreas Zeller, Jens Krinke. *Software-Werkzeuge.* TU Braunschweig, 1997.