

Advanced Slicing of Sequential and Concurrent Programs

Jens Krinke
Universität Passau*

Abstract: Program Slicing ist eine Technik zur Identifikation von Anweisungen, die andere Anweisungen beeinflussen können. Trotz seit nunmehr 25 Jahren anhaltender Forschung hat Program Slicing immer noch Probleme, die eine verbreitete Benutzung verhindern: Slices können zu groß oder zu unverständlich werden, oder ihre Berechnung kann zu teuer oder zu kompliziert für echte Programme sein. Diese Dissertation präsentiert Lösungen und Auswege aus diesen Problemen. Sie enthält einerseits eine Reihe von Slicing-Verfahren unterschiedlicher Präzision und Geschwindigkeit. Andererseits enthält sie verschiedenste Verfahren, die dem Benutzer helfen, Slices leichter zu verstehen indem die berechneten Slices mehr auf seine Bedürfnisse fokussiert werden. Alle vorgestellten Verfahren wurden im VALSOFT-System implementiert und gründlich evaluiert.

Die dem Slicing zugrunde liegende Datenstruktur sind Programmabhängigkeitsgraphen. Diese können auch für andere Anwendungen benutzt werden: Ein neues Verfahren zur Entdeckung von dupliziertem Code identifiziert ähnliche Teilgraphen in Programmabhängigkeitsgraphen. Dieses Verfahren kann modifizierte Duplikate besser erkennen als andere Verfahren.

Auf der theoretischen Seite präsentiert diese Dissertation ein hochpräzises Verfahren zum Slicing nebenläufiger prozeduraler Programme, wobei optimales Slicing bekannterweise nicht entscheidbar ist. Dieses Verfahren ist das erste zum Slicen nebenläufiger Programme, das nicht auf Inlining aufgerufener Prozeduren zurückgreift.

1 Einführung

Program-Slicing beantwortet die Frage „Welche Anweisungen können die Berechnungen in einer anderen, gegebenen Anweisung beeinflussen?“. Diese Frage stellt sich in den verschiedensten Anwendungen, doch besonders im Bereich der Software-Wartung und des -Reengineering. Daher spielt Program-Slicing dort eine besondere Rolle, daneben gibt es spezifische Anwendungen beim Debuggen, beim Testen, beim Programm-Vergleich und der -Integration, bei der Impact-Analyse, der Funktions-Extraktion und -Restrukturierung, oder auch der Kohäsions-Messung. Sogar zum Debuggen und Testen von Spreadsheets oder zur Typ-Prüfung von Programmen wurde Slicing inzwischen benutzt.

Nach Weisers erster Veröffentlichung zu Program-Slicing [We79] sind fast 25 Jahre vergangen und es wurden verschiedenste Ansätze zur Berechnung von Slices entwickelt. Normalerweise werden Innovationen in der Informatik nach etwa 10 Jahren weitverbreitet eingesetzt. Warum sind aber Slicing-Techniken heute noch nicht einfach benutzbar? Die

*Jens Krinke, FernUniversität in Hagen, krinke@acm.org, <http://www.fernuni-hagen.de/ST/>

beiden Hauptprobleme sind:

1. Vorhandene Slicer sind langsam und unpräzise.
2. Slicing in der jetzigen Form ist nicht adäquat für typische Software-Entwicklungs-Probleme.

Der erste Punkt entstammt der Beobachtung, dass es zwar inzwischen schnelle und präzise Slicing-Verfahren gibt, aber es immer noch ein Problem ist, diese Verfahren auf Millionen-Zeilen-Programme zu skalieren. Für sequentielle Programme ist die Präzision der Slicer schon sehr hoch, aber Präzision ist immer noch eine Herausforderung bei der Analyse nebenläufiger Programme. Erst seit kurzem werden Sprachen mit expliziter Nebenläufigkeit wie Ada oder Java ‘gesliced’. Der zweite Punkt trifft leider immer noch zu: Normalerweise sind Slices schwer zu verstehen. Dies liegt zum Teil an schlechten Benutzeroberflächen, aber hauptsächlich daran, dass Slicing den Benutzer einfach mit den Resultaten ‘überschüttet’ – ohne jegliche Erklärung.

Diese Arbeit zeigt, wie diese Probleme und Herausforderungen in Angriff genommen und gelöst werden können. Die drei Hauptziele sind:

1. Ein hochpräzises Verfahren zum Slicing nebenläufiger Programme.
2. Den Benutzer beim Verstehen eines Slices zu helfen, indem Slicing an die Probleme und Bedürfnisse der Nutzer angepasst wird.
3. Die Probleme und Konsequenzen der Slicing-Algorithmen für zukünftige Entwicklungen aufzuzeigen.

Dabei gibt diese Arbeit eine in sich geschlossene Einführung in Program Slicing. Sie kann keine komplette Übersicht geben, denn seit Tips exzellenter Übersicht [Ti95]¹ hat sich die verfügbare Literatur vervielfältigt: CiteSeer hat vor kurzem noch 257 Zitierungen von Weisers Slicing-Artikel [We84] aufgeführt (und 95 für [We82]).

2 Program-Slicing

Ein Slice enthält diejenigen Anweisungen eines Programms, die möglicherweise einen Einfluss auf eine bestimmte, interessierende Anweisung haben, dem sog. Slicing-Kriterium. Abbildung 1 (a) zeigt ein Beispielprogramm, bei dem ermittelt werden soll, welche Anweisungen die Ausgabe der Variable p in Zeile 10 beeinflussen können. Der entsprechende Slice zu diesem Kriterium $(10, p)$ ist in Abbildung 1 (b) gezeigt. Dieser Slice enthält nur noch diejenigen Anweisungen, die das Produkt der Zahlen $1-n$ berechnen, wohingegen die Anweisungen zur Berechnung und Ausgabe der Summe in den Zeilen 3, 6 und 9 keinen Einfluss auf das Kriterium haben.

¹Auf Tips Übersicht [Ti95] folgten einige andere [BG96, HG98, DL01, HH01].

1	read(n)	1	read(n)
2	i := 1	2	i := 1
3	s := 0	3	
4	p := 1	4	p := 1
5	while (i <= n)	5	while (i <= n)
6	s := s + i	6	
7	p := p * i	7	p := p * i
8	i := i + 1	8	i := i + 1
9	write(s)	9	
10	write(p)	10	write(p)

(a) Beispiel-Programm (b) Slice zum Kriterium (10, p)

Abbildung 1: Ein Beispiel-Programm mit einem Slice

Ursprünglich wurde Slicing 1979 von Weiser als Hilfsmittel beim Debugging definiert, denn Programmierer machen nämlich bei der Fehlersuche auch nichts anderes als von fehlerhaften Zwischenergebnissen rückwärts nach verursachenden Anweisungen zu suchen [We79, We84]. Er präsentierte auch den ersten Ansatz, Slices durch iterative Datenflussanalyse zu berechnen. Der andere, oft benutzte Ansatz zur Berechnung von Slices benutzt Erreichbarkeits-Analyse in Programmabhängigkeitsgraphen (Program Dependence Graph, PDG) [FOW87]. Programmabhängigkeitsgraphen bestehen hauptsächlich aus Knoten, die die Anweisungen des Programms repräsentieren, und Kontroll- und Datenabhängigkeitskanten:

- Zwischen zwei Anweisungen besteht eine Kontrollabhängigkeit, wenn eine Anweisung die Ausführung der anderen kontrolliert. Dies ist z. B. bei if- oder while-Anweisungen der Fall.
- Eine Datenabhängigkeit besteht, wenn die Definition einer Variablen bei der einen Anweisung eine Benutzung bei einer anderen Anweisung erreichen kann. Beispielsweise wenn eine Anweisung einer Variablen einen Wert zuweist und diese Variable bei einer anderen Anweisung ausgewertet wird, ohne dass im Programmablauf zwischen diesen beiden Punkten eine weitere Zuweisung an diese Variable stattfindet.

Ein Slice lässt sich nun einfach berechnen, in dem das Slicing-Kriterium auf einen Knoten abgebildet wird und alle rückwärts erreichbaren Knoten dem Slice hinzugefügt werden. Dabei werden die erreichten Knoten wiederum auf die entsprechenden Anweisungen abgebildet.

Dies ist nur eine informale Beschreibung, wie auch die gesamte Darstellung in dieser Kurzfassung. In der vollständigen Dissertation finden sich die formalen Definitionen, Beschreibungen und Beweise.

Abbildung 2 zeigt den Programmabhängigkeitsgraphen des Programms aus Abbildung 1. Die Kontrollabhängigkeitskanten sind gestrichelt, die Datenabhängigkeitskanten durchgängig. Der Slice zum Kriterium (10, p) aus Abbildung 1 (b) ist hervorgehoben.

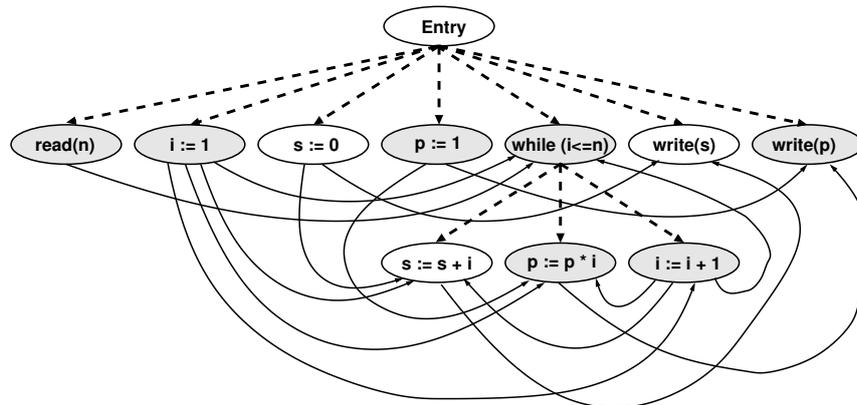


Abbildung 2: Der Programmabhängigkeitsgraph zu Abbildung 1.

2.1 Slicing sequentieller Programme

Beispiel 1 (Slicing ohne Prozeduren) Abbildung 3 zeigt ein Beispiel, bei dem in einem Programm ein Slice berechnet werden soll. Um für die Anweisung 'print a' den Slice zu berechnen, müssen wir einfach nur die dargestellten Abhängigkeiten zurückverfolgen. Dieses Beispiel enthält zwei Datenabhängigkeiten und der Slice enthält die Zuweisung an a und die read-Anweisung für b.

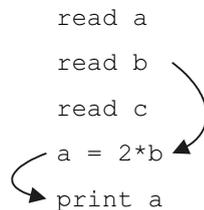


Abbildung 3: Ein Programm ohne Prozeduren

In diesem und allen folgenden Beispielen werden alle Kontrollabhängigkeiten ignoriert; wir konzentrieren uns nur auf die Datenabhängigkeiten um die Verständlichkeit zu wahren. Außerdem werden wir immer einen Slice von der Anweisung 'print a' aus (rückwärts) berechnen.

Slicing ohne Prozeduren ist trivial: Man muss nur die (rückwärts) erreichbaren Knoten im PDG finden [FOW87]. Die zugrunde liegende Annahme ist, dass alle Pfade *realisierbar* sind. Das heißt, dass eine mögliche Ausführung des Programms existiert, bei der für jeden Pfad die Anweisungen in der korrespondierenden Reihenfolge ausgeführt werden.

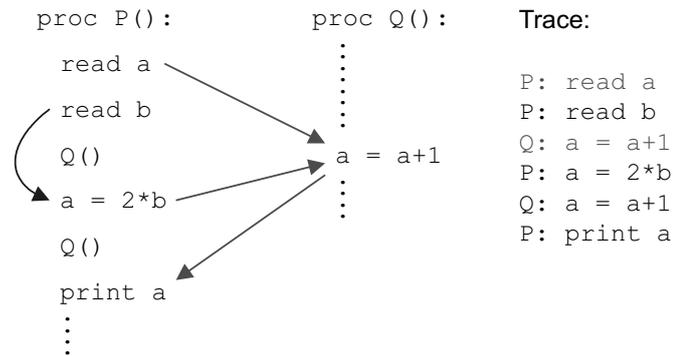


Abbildung 4: Ein Programm mit zwei Prozeduren

Beispiel 2 (Slicing mit Prozeduren) Jetzt erweitern wir das Beispiel durch zwei Prozeduren in Abbildung 4. Wenn wir den Aufruf-Kontext ignorieren und einfach nur die Abhängigkeiten traversieren, würden wir die Anweisung ‘read a’ zum Slice für ‘print a’ hinzufügen. Dies sollte aber nicht sein, denn diese Anweisung hat offensichtlich keinen Einfluss auf die Anweisung ‘print a’. Die Anweisung ‘read a’ hat nur einen Einfluss auf den ersten Aufruf der Prozedur Q, aber die Variable a wird vor dem zweiten Aufruf von Q durch die Zuweisung ‘a=2*b’ in Prozedur P überschrieben.

Eine solche Analyse wird als *kontext-insensitiv* bezeichnet, da der Aufruf-Kontext ignoriert wird. Pfade sind jedoch nur realisierbar, wenn sie den Aufruf-Kontext beachten. Daher ist Slicing *kontext-sensitiv*, wenn die traversierten Pfade immer den Aufruf-Kontext beachten. Kontext-sensitive Slicing lässt sich effizient berechnen – an den Aufrufstellen werden sogenannte *Summary-Kanten* eingefügt [HRB90]: Summary-Kanten repräsentieren die transitiven Abhängigkeiten der aufgerufenen Prozeduren an den Aufrufstellen.

Innerhalb der implementierten Infrastruktur zur Berechnung und Analyse von PDGs für ANSI-C-Programme wurden verschiedene Slicing-Algorithmen implementiert und evaluiert. Eine der Evaluierungen dieser Arbeit (vorveröffentlicht in [Kr02]) zeigt, dass kontext-insensitives Slicing sehr viel weniger präzise ist als kontext-sensitives Slicing. Im Durchschnitt sind Slices, die durch kontext-insensitive Algorithmen berechnet worden sind, um 67% größer als die Slices der kontext-sensitiven Algorithmen. Dies zeigt dass kontext-sensitives Slicing, wenn möglich, vorzuziehen ist, da der kontext-insensitive Verlust von Präzision nicht akzeptabel ist. Ein überraschendes Ergebnis ist dabei, dass kontext-insensitives Slicing sogar *langsamer* als das komplexere kontext-sensitive Slicing ist (im Durchschnitt 23%). Der Grund dafür ist, dass der kontext-sensitive Algorithmus wegen der höheren Präzision sehr viel weniger Knoten während der Graph-Traversierung besuchen muss.

Die Komplexität von Slicing in Abhängigkeitsgraphen ist linear über die Anzahl der Kanten im Graphen. Normalerweise wird jede Kante und jeder Knoten maximal einmal besucht, nur die kontext-sensitive Variante besucht manche Knoten doppelt. Hingegen haben die zum Aufbau der Programmabhängigkeitsgraphen notwendigen Datenflussanalysen eine deutlich höhere Komplexität.

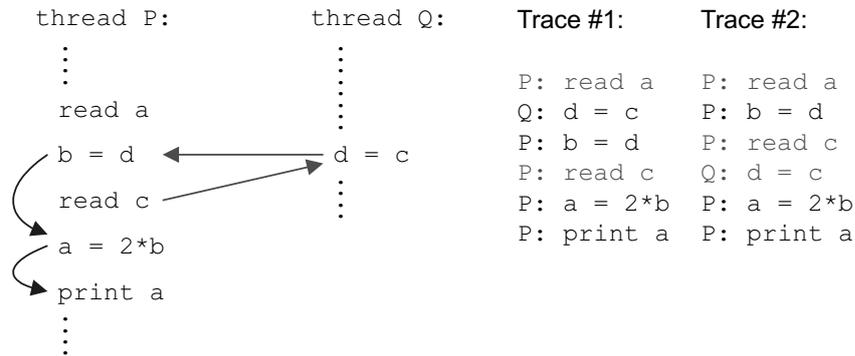


Abbildung 5: Ein Programm mit zwei Threads

2.2 Slicing nebenläufiger Programme

Betrachten wir nun nebenläufige Programme. In nebenläufigen Programmen mit gemeinsamen Variablen entsteht ein neuer Typ von Abhängigkeit: *Interferenz*. Interferenz entsteht, wenn eine Variable in einem Thread zugewiesen (definiert) wird und in einem parallel ausgeführten Thread ausgewertet (benutzt) wird.

Beispiel 3 (Slicing nebenläufiger Programme) *Im Beispiel der Abbildung 5 gibt es zwei Threads P und Q , die parallel ausgeführt werden. In diesem Beispiel gibt es zwei Interferenz-Abhängigkeiten: Die eine entsteht durch eine Zuweisung und Benutzung der Variable d , die andere durch die Variable c .*

Ein einfaches Traversieren der Interferenz beim Slicing führt wieder zu unpräzisen Slices, da Interferenz zu nicht realisierbaren Pfaden führt. Im Beispiel der Abbildung 5 würde ein einfaches Traversieren die Anweisung ‘read c’ dem Slice hinzufügen. Allerdings gibt es keine mögliche Ausführung bei der die ‘read c’ Anweisung Einfluss auf die Zuweisung ‘b = d’ hat. Eine solche Ausführung würde eine *Zeitreise* erfordern, denn die Zuweisung ‘b = d’ wird immer vor der Anweisung ‘read c’ ausgeführt. Ein Pfad durch mehrere Threads ist jedoch nur dann realisierbar, wenn er nur aus chronologisch legalen Ausführungsreihenfolgen besteht.

Sogar wenn nur realisierbaren Pfade betrachtet werden, wird der berechnete Slice nicht so präzise sein wie möglich. Diese Unpräzision liegt daran, dass parallel ausgeführte Threads die Definitionen (Zuweisungen) von anderen Threads wieder löschen können:

Beispiel 4 *Im Beispiel der Abbildung 6 ist die Anweisung ‘read a’ über einen realisierbaren Pfad (rückwärts) erreichbar von der ‘print a’ Anweisung. Aber es gibt keine mögliche Ausführung, bei der die ‘read’ Anweisung einen Einfluss auf die ‘print’ Anweisung hat (wenn wir annehmen, dass Anweisungen atomar sind). Entweder erreicht die ‘read’ Anweisung die Benutzung im Thread Q , wird aber danach durch die Zuweisung*

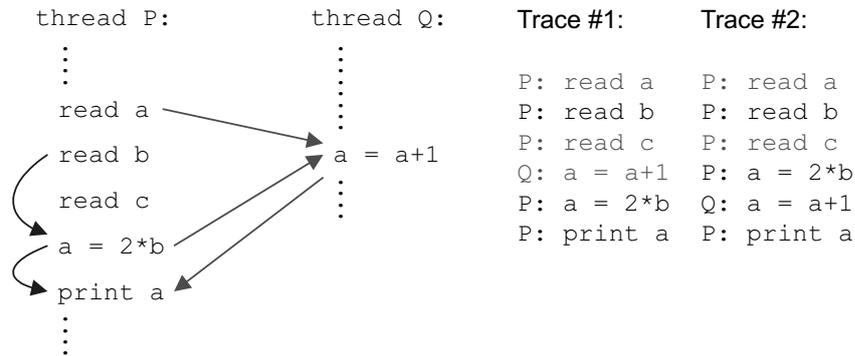


Abbildung 6: Ein weiteres Programm mit zwei Threads

*‘a=2*b’ im Thread P überschrieben, oder die ‘read’ Anweisung wird sofort durch die Zuweisung ‘a=2*b’ überschrieben, bevor sie die Benutzung in Thread Q erreichen kann.*

Müller-Olm hat gezeigt, dass präzises kontext-sensitives Slicing von nebenläufigen Programmen im allgemeinen nicht entscheidbar ist [MOS01]. Daher müssen wir konservative Approximationen zur Analyse nebenläufiger Programme benutzen. Eine naive Approximation wäre es, Zeitreisen zuzulassen – dies würde aber zu einem nicht akzeptablen Verlust an Präzision führen. Außerdem können Summary-Kanten nicht mehr benutzt werden, denn Summary-Kanten würden die Effekte parallel ausgeführter Threads ignorieren. Summary-Kanten stellen die transitiven Abhängigkeiten der aufgerufenen Prozedur ohne Interferenz dar. Sie lassen sich nicht um Interferenz erweitern, denn Interferenz ist im Gegensatz zu Daten- und Kontrollabhängigkeit nicht transitiv. Wenn man daher auf kontext-insensitives Slicing ausweicht, erzeugt man wiederum einen nicht akzeptablen Verlust an Präzision.

Um nun ohne Summary-Kanten präzise Slices berechnen zu können, wurde ein neues Verfahren entwickelt, das auf der expliziten Darstellung des Aufruf-Kontexts durch *Call-Strings* [SP81] beruht. Call-Strings können als abstrakte Repräsentation des Aufruf-Stacks gesehen werden. Sie werden in der kontext-sensitiven Programmanalyse häufig benutzt, z. B. in der Pointer-Analyse. Die Call-Strings werden entlang der Kanten des PDGs propagiert: Bei Kanten, die zwei Prozeduren mit einander verbinden, wird der Call-String benutzt um sicherzustellen, dass ein Aufruf immer an die richtige Aufrufsstelle zurückkehrt. Dadurch werden Call-Strings niemals entlang nicht realisierbarer Pfade propagiert.

Die grundsätzliche Idee für das hoch-präzise Verfahren zum Slicen nebenläufiger Programme ist die Adaption des Call-String-Verfahrens auf nebenläufige Programme. Dazu wird der Aufruf-Kontext für jeden Thread durch einen eigenen Call-String dargestellt. Damit ist der Aufruf-Kontext ein Tupel von Call-Strings, der entlang der Kanten des PDGs propagiert wird. So kann einerseits bei aufgerufenen Prozeduren sichergestellt werden, dass an die richtige Aufrufsstelle zurückgekehrt wird, andererseits kann anhand des Aufrufkontexts beim Wechsel zwischen den Threads sichergestellt werden, dass keine Zeitreise auftritt.

Ein kombiniertes Verfahren vermeidet dabei eine kombinatorische Explosion der Call-Strings: Dabei werden Summary-Kanten benutzt um Slices innerhalb eines Threads zu berechnen. Zusätzlich werden Call-Strings nur dann generiert und entlang Interferenz-Abhängigkeiten propagiert, wenn der Slice mehrere Threads umfasst. Mit diesem Verfahren werden deutlich weniger Kontexte propagiert.

Dies skizziert nur die grundsätzliche Idee des Verfahrens. Diese Dissertation präsentiert die Grundlagen und Algorithmen zum Slicing sequentieller und nebenläufiger Programme im Detail (vorveröffentlicht in [Kr03b]). Zusätzlich präsentiert ein Großteil dieser Dissertation Optimierungen und hochentwickelte Anwendungen des Slicings, wie sie im folgenden Abschnitt kurz besprochen werden.

3 Beiträge

Diese Dissertation ist soweit wie möglich in sich abgeschlossen. Neben einer gründlichen Darstellung des Program Slicings enthält diese Dissertation folgende Errungenschaften:

- Feingranulare Programmabhängigkeitsgraphen zur Darstellung von ANSI-C, die auch nicht-deterministische Ausführungsreihenfolge darstellen können. Diese Graphen sind eine abgeschlossene Zwischendarstellung und die Basis für die Entdeckung von dupliziertem Code und für die Berechnung von Pfadbedingungen.
- Ein neues hochpräzises Verfahren zum Slicing nebenläufiger Programme mit Prozeduren. Dieses kontext-sensitive Verfahren erreicht eine hohe Präzision, obwohl präzises (bzw. optimales) Slicing nicht entscheidbar ist. Dieses Verfahren ist das erste, das nicht auf Inlining basiert und dadurch auch rekursive Programme analysieren kann (vorveröffentlicht in [Kr03b]).
- Einige Varianten der Slicing-Algorithmen für (interprozedurale) PDGs und eine gründliche Evaluierung derselben. Ein Großteil wurde schon in [Kr02] präsentiert. Diese Verfahren werden ergänzt durch Techniken zum Verkleinern der Programmabhängigkeitsgraphen, ohne dabei die Präzision des Slicings zu verschlechtern.
- Fundamentale Ideen und Verfahren um Abhängigkeitsgraphen und Slices zu visualisieren – durch Graph-basierte, textuelle und abstrakte Repräsentationen. Dazu gehören einige Methoden um die Resultate vom Slicing stärker zu fokussieren oder zu abstrahieren. Teile davon wurden in [Kr03a] veröffentlicht.
- Ein PDG-basiertes Verfahren um duplizierten Code zu entdecken². Dieses Verfahren identifiziert ähnliche Teilgraphen in den PDGs und hat höhere Erkennungsraten für modifizierte Duplikate als andere Verfahren, denn es erkennt ähnliche Semantik statt nur ähnlichem Text. Nach der Veröffentlichung in [Kr01] wurden die Vor- und Nachteile dieses Verfahren in einem Clone-Detection-Wettbewerb evaluiert.

²Die Erkennung von dupliziertem Code wird auch „Clone Detection“ genannt

- Methoden, um *Pfadbedingungen* für komplexe Datenstrukturen, für Prozeduren und für nebenläufige Programme zu berechnen. Slicing kann nur zeigen, dass eine Anweisung von einer anderen beeinflusst werden kann. Pfadbedingungen geben notwendige Bedingungen an, die erfüllbar sein müssen, damit ein Einfluss tatsächlich stattfinden kann. Der grundsätzliche Ansatz für Pfadbedingungen wurde von Snelting vorgestellt [Sn96] und von Robschink weiterentwickelt [RS02, SRK03].
- Die Architektur des VALSOFT-Systems und die Entwicklung der Datenflussanalyse, der Konstruktion der Abhängigkeitsgraphen und der verschiedenen Slicing-Algorithmen.

4 Ausblick

Mit der vorliegenden Dissertation wurden dringende Probleme des Program-Slicings in Angriff genommen und einige Lösungen präsentiert. Mit der Neuentwicklung und der Evaluation von Slicing-Verfahren sowohl für traditionell sequentielle als auch für nebenläufige Programme konnte gezeigt werden, dass hochpräzises und effizientes Slicing möglich ist. Bei der Implementierung des VALSOFT-Systems hat sich gezeigt, dass nicht das sequentielle Slicing an sich für Skalierungsprobleme verantwortlich ist, sondern die zum Aufbau der Abhängigkeitsgraphen notwendigen Datenflussanalysen, insbesondere die Pointer-Analyse. Für das Slicing wirklich großer Programme brauchen wir daher deutliche Fortschritte im Bereich dieser Datenflussanalysen.

In der ursprünglichen Form ist Program-Slicing zum Programm-Verstehen weniger geeignet. Diese Dissertation hat Mittel und Wege gezeigt, wie neue Slicing-Verfahren zu verständlicheren Ergebnissen führt. Insbesondere Verfahren, die am Ende nicht nur ein Ergebnis präsentieren, sondern dieses Ergebnis auch noch erklären, haben sich als vielversprechend herausgestellt und sollten Gegenstand zukünftiger Arbeiten sein.

Literatur

- [BG96] Binkley, D. und Gallagher, K. B.: Program slicing. *Advances in Computers*. 43:1–50. 1996.
- [DL01] De Lucia, A.: Program slicing: Methods and applications. In: *IEEE workshop on Source Code Analysis and Manipulation (SCAM 2001)*. 2001. Invited paper.
- [FOW87] Ferrante, J., Ottenstein, K. J., und Warren, J. D.: The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*. 9(3):319–349. July 1987.
- [HG98] Harman, M. und Gallagher, K. B.: Program slicing. *Information and Software Technology*. 40(11–12):577–581. 1998.
- [HH01] Harman, M. und Hierons., R.: An overview of program slicing. *Software Focus*. 2(3):85–92. 2001.

- [HRB90] Horwitz, S. B., Reps, T. W., und Binkley, D.: Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*. 12(1):26–60. January 1990.
- [Kr01] Krinke, J.: Identifying similar code with program dependence graphs. In: *Proc. Eighth Working Conference on Reverse Engineering*. S. 301–309. 2001.
- [Kr02] Krinke, J.: Evaluating context-sensitive slicing and chopping. In: *International Conference on Software Maintenance*. S. 22–31. 2002.
- [Kr03a] Krinke, J.: Barrier slicing and chopping. In: *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation*. S. 81–87. 2003.
- [Kr03b] Krinke, J.: Context-sensitive slicing of concurrent programs. In: *Proceedings ESEC/FSE*. S. 178–187. 2003.
- [MOS01] Müller-Olm, M. und Seidl, H.: On optimal slicing of parallel programs. In: *STOC 2001 (33th ACM Symposium on Theory of Computing)*. S. 647–656. 2001.
- [RS02] Robschink, T. und Snelting, G.: Efficient path conditions in dependence graphs. In: *Proceedings of the 24th International Conference of Software Engineering (ICSE)*. S. 478–488. 2002.
- [Sn96] Snelting, G.: Combining slicing and constraint solving for validation of measurement software. In: *Static Analysis Symposium*. volume 1145 of *LNCS*. S. 332–348. Springer. 1996.
- [SP81] Sharir, M. und Pnueli, A.: Two approaches to interprocedural data flow analysis. In: *Program Flow Analysis: Theory and Applications*. S. 189–233. Prentice-Hall. 1981.
- [SRK03] Snelting, G., Robschink, T., und Krinke, J.: Efficient path conditions in dependence graphs for software safety analysis. Submitted for publication. 2003.
- [Ti95] Tip, F.: A survey of program slicing techniques. *Journal of programming languages*. 3(3). September 1995.
- [We79] Weiser, M.: *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis. University of Michigan. Ann Arbor. 1979.
- [We82] Weiser, M.: Programmers use slices when debugging. *Communications of the ACM*. 25(7):446–452. 1982.
- [We84] Weiser, M.: Program slicing. *IEEE Transactions on Software Engineering*. 10(4):352–357. July 1984.

Jens Krinke war nach Abschluss seines Diploms in Informatik 1995 als wissenschaftlicher Mitarbeiter in der Abteilung Softwaretechnik an der Technischen Universität Braunschweig beschäftigt. Nach dem Wechsel als wissenschaftlicher Assistent an die Universität Passau, Lehrstuhl Softwaresysteme (Prof. Gregor Snelting), setzte er seine Forschungstätigkeit im Rahmen des Projekts „VALSOFT – Validierung softwaregesteuerter Systeme“ fort. Unterbrochen durch zwei Aufenthalte am IBM T.J. Watson Research Center, schloss er 2003 seine Promotion mit der hier besprochenen Dissertation ab. Nach dem Wechsel an die FernUniversität in Hagen liegt sein Forschungsschwerpunkt als Juniorprofessor für Softwaretechnik im Bereich dynamische Programmanalyse, Aspect Mining und Distant Learning.