Kombination von Slicing mit Constraint-Solving für Software-Reengineering

Jens Krinke* Torsten Robschink*

Zusammenfassung

Um sicherheitsrelevante Software zu überprüfen, reichen herkömmliche Softwareanalyse-Verfahren nicht aus. Unser Softwareanalyse-System ValSoft setzt Datenflußanalysen und deduktive Verfahren ein, um Beeinflussungen von relevanten Informationspfaden innerhalb von Programmen zu erkennen und zu analysieren. Die Grundlage der Analyse bilden sog. Programmabhängigkeitsgraphen. Mit deren Hilfe berechnen wir konservativ approximierte Pfadbedingungen: diese geben die Umstände an, unter denen es eine Abhängigkeit zwischen interessierenden Punkten im Programm geben kann.

1 Einleitung

In Software-Reengineering-Projekten kann man sich kaum auf etwas verlassen: Dokumentation ist unzureichend, fehlt oder ist nicht (mehr) vorhanden. Im Grunde kann man sich nur auf den Programmtext selbst verlassen und ist deshalb beim Reengineering auf Hilfsmittel angewiesen, die Unterstützung bei der Analyse der Software bieten.

Program-Slicing ist eine Technik, die dafür gut geeignet ist: Ein Slice besteht aus den Teilen eines Programms, die einen Punkt von Interesse (dem Slicing-Kriterium) in einem Programm direkt oder indirekt beeinflussen können. Beispielsweise kann damit bestimmt werden, welche Teile eines Programms Einfluß auf den Wert einer Variablen an einer bestimmten Stelle des Programms haben können.

2 Program-Slicing

Slicing läßt sich natürlich und effizient mit Hilfe von Programmabhängigkeitsgraphen (engl. Program Dependence Graph, PDG) implementieren. Mittels Datenflußanalyse wird festgestellt, welche Abhängigkeiten zwischen den Anweisungen des Programms auftreten. Diese werden dann im Programmabhängigkeitsgraphen als Kanten dargestellt, wobei die Anweisungen des Programms durch Knoten dargestellt werden. Dabei wird zwischen *Kontrollabhängigkeitskanten* und *Datenabhängigkeitskanten* unterschieden: Kontrollabhängigkeiten entstehen, wenn Anweisungen die Ausführung von anderen Anweisungen kontrollieren (If-, While- u. a. Anweisungen). Datenabhängigkeiten entstehen zwischen Anweisungen, die Variablen definieren und Anweisungen, die diese Variablen verwenden.

Den Slice zu einer Anweisung *s* erhält man nun einfach dadurch, daß man alle Knoten aufsammelt, von denen aus ein Pfad zur Zielanweisung *s* führt. Dies kann man

*Universität Passau, Lehrstuhl für Softwaresysteme, D-94030 Passau, krinke@fmi.uni-passau.de

```
① if (i > 0)
②   a = x;
③   j = b;
④ c = z;
⑤ if (j < 5)
⑥   if (i < 8)
⑦     y = a;
⑧ print(y);
```

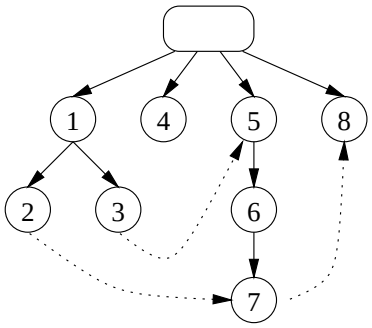

Abbildung 1: Beispielprogramm mit Abhängigkeitsgraph

durch einfaches Rückwärtsverfolgen der PDG-Kanten – ausgehend von *s* – mittels Tiefensuche realisieren.

Im Beispiel von Abb. 1 sei das Slicing-Kriterium die Zeile 8. Der zugehörige Slice enthält den gesamten Ausschnitt bis auf Zeile 4 (siehe Abb. 2

```
① if (i > 0)
②   a = x;
③   j = b;
④ c = z;
⑤ if (j < 5)
⑥   if (i < 8)
⑦     y = a;
⑧ print(y);
```

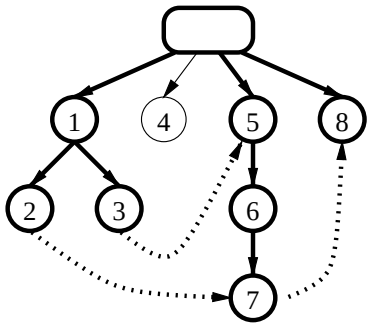

Abbildung 2: Slice des Beispielprogramms

Als direkt ablesbares Ergebnis stellen wir fest, daß der Wert der Variablen `i` im Prädikat der If-Anweisung in ① den Wert der Variablen `y` in ⑧ beeinflußt. Mit Programmabhängigkeitsgraphen kann man sogar die Frage „Warum ist dies so?“ beantworten, indem man alle Pfade im Graphen zwischen ① und ⑧ verfolgt: Dies sind ① → ② → ⑦ → ⑧ und ① → ③ → ⑤ → ⑥ → ⑦ → ⑧. Wenn wir beispielsweise den Pfad ① → ② → ⑦ → ⑧ interpretieren, ergibt sich folgende Begründung: *`y` in ⑧ ist von `i` in ① abhängig, weil `y` in ⑧ direkt von `a` in ⑦ abhängt und `a` in ⑦ wiederum direkt von `x` in ② abhängig ist. Das Prädikat in ① kontrolliert ob ② ausgeführt wird, daher die Abhängigkeit zwischen `i` in ① und `a` in ②.*

Mit normalen Mitteln können wir aber die Frage „Unter welchen Bedingungen ist dies möglich?“ nicht mehr beantworten. Mit Slices kann nur festgestellt werden, ob eine Beeinflussung möglich ist, aber nicht, unter welchen Bedingungen. Unser Analysesystem ValSoft stetzt daher deduktive Verfahren ein, um genau solche Bedingungen zu ermitteln.

Deshalb berechnen wir sog. *Pfadbedingungen*, das sind Bedingungen, unter denen ein Pfad über die Abhängigkeitskanten zwischen zwei Knoten existiert und damit der eine Knoten (transitiv) von dem anderen Knoten abhängig ist. Pfadbedingun-

gen sind allgemein gesprochen aus jenen Kontrollprädikaten zusammengesetzt, die die Ausführung der Knoten des Pfades kontrollieren; hinzu kommen spezielle Bedingungen für Arrays und andere Datenstrukturen. Es handelt sich also um Formeln der Aussagen- oder Prädikatenlogik (inklusive Arithmetik usw.) über den Programmvariablen.

3 Pfadbedingungen

Ein einfacher Ansatz zur Berechnung der Pfadbedingungen zwischen zwei Knoten x und y besteht aus den folgenden Schritten:

1. Bestimme alle Pfade P_i zwischen x und y im Abhängigkeitsgraphen.
2. Bestimme für alle Knoten n auf den Pfaden die *Ausführungsbedingung* $E(n)$
3. Kombiniere die Ausführungsbedingungen zur Pfadbedingung $PC(x,y)$.

$$
\begin{array}{lcl}
E(①) & = & \texttt{true} \\
E(②) & = & \texttt{i} > 0 \\
E(③) & = & \texttt{i} > 0 \\
E(⑤) & = & \texttt{true} \\
E(⑥) & = & \texttt{j} < 5 \\
E(⑦) & = & (\texttt{j} < 5) \wedge (\texttt{i} < 8) \\
E(⑧) & = & \texttt{true}
\end{array}
$$

Abbildung 3: Ausführungsbedingungen

$$
\begin{array}{lcl}
PC(P_1) & = & \texttt{true} \wedge (\texttt{i} > 0) \wedge (\texttt{j} < 5) \wedge (\texttt{i} < 8) \wedge \texttt{true} \\
 & = & (\texttt{i} > 0) \wedge (\texttt{j} < 5) \wedge (\texttt{i} < 8) \\
PC(P_2) & = & \texttt{true} \wedge (\texttt{i} > 0) \wedge \texttt{true} \wedge (\texttt{j} < 5) \wedge (\texttt{j} < 5) \wedge (\texttt{i} < 8) \wedge \texttt{true} \\
 & = & (\texttt{i} > 0) \wedge (\texttt{j} < 5) \wedge (\texttt{i} < 8)
\end{array}
$$

Abbildung 4: Pfadbedingungen

Die Ausführungsbedingung bestimmt die Umstände, unter denen eine Anweisung ausgeführt werden kann. Dazu werden die Kontrollabhängigkeitskanten rückwärts verfolgt und die Prädikate an den Knoten aufgesammelt. Im Beispiel ist die Ausführungsbedingung für Anweisung 7: $E(⑦) = (\texttt{j} < 5) \wedge (\texttt{i} < 8)$. Die Ausführungsbedingungen eines Pfades P werden zur Pfadbedingung verknüpft: $PC(P) = \bigwedge_\nu E(k_\nu)$ wobei ν über die Knoten des Pfades läuft. Gibt es mehrere Pfade $P_1, P_2, \ldots, P_m$ von x nach y, werden deren Pfadbedingungen disjunktiv verknüpft. Mithin ist $PC(x,y) = \bigvee_\mu PC(P_\mu)$ wobei μ über die Pfade läuft. Im Beispiel ergibt sich die Pfadbedingung $PC(2,8) = (\texttt{i} > 0) \wedge (\texttt{j} < 5) \wedge (\texttt{i} < 8)$.

Werden die Programme größer, lassen sich die Pfadbedingungen nicht nach diesem einfachen Verfahren berechnen, da es potentiell unendlich viele Pfade geben kann. Obwohl Zyklen in Pfaden ignoriert werden können [Sne96] und daher die Zahl der Pfade

endlich bleibt, explodiert die Größe der Pfadbedingungen bei großen Programmen. Wir haben daher Techniken entwickelt, die solche Pfadbedingungen effizient berechnen. Diese Techniken beruhen einerseits auf intelligenten Verfahren zum Aufstellen von redundanzarmen Bedingungen und andererseits dem Einsatz von externen Constraint-Solvern, die die generierten Pfadbedingungen vereinfachen.

4 Anpassungen

Die Erzeugung von Pfadbedingungen führte zu notwendigen Änderungen an den bekannten Slicing- und Abhängigkeitsgraph-Techniken:

- Unser Abhängigkeitsgraph ist feingranular, d. h. Knoten stellen nicht nur ganze Anweisungen sondern Variablen und Operationen dar.
- Slicing wurde an den feingranularen Graphen angepaßt.
- Zusätzliche Abhängigkeiten wurden eingeführt um die Pfadbedingungen noch präziser zu machen.

4.1 Feingranulare PDGs

Abweichend vom traditionellen Ansatz enthält unsere Variante des PDGs nicht nur Knoten für Anweisungen, sondern auch für (Teil-)Ausdrücke. Die Anweisungen sind also Subgraphen, zusammengesetzt aus Variablen und Operationen. Diese Erweiterung ist notwendig, da die gesamte Information aus dem ursprünglichen Programm später bei der Erzeugung der Pfadbedingungen aus dem PDG ohne Verlust rekonstruiert werden muß. Außerdem können Ausdrücke mehrere Seiteneffekte haben, aber die PDG-Darstellung darf nur einen Seiteneffekt pro Knoten enthalten. Würde ein Knoten mehr als einen Seiteneffekt enthalten, könnte nicht entschieden werden, durch welchen Seiteneffekt eine ausgehende Datenabhängigkeit erzeugt wird und die Analyse würde ungenauer werden. Wenn der PDG hingegen auf Ausdrücken basiert, sind keine Programmtransformation zum Entfernen von Seiteneffekten notwendig und der erzeugte PDG ist strukturell sehr ähnlich zum Syntaxbaum und damit zum Quelltext.

Im traditionellen PDG werden Knoten, die keine Prädikate sind, durch unmarkierte Kontrollabhängigkeitskanten mit ihren Söhnen verbunden. Solche Kanten werden im feingranularen PDG durch *Unbedingte Kontrollabhängigkeitskanten* ersetzt. Um die Abhängigkeiten innerhalb von Ausdrücken darzustellen, sind neben unbedingten Kontrollabhängigkeitskanten zusätzlich *Wertabhängigkeitskanten* notwendig. Wertabhängigkeitskanten sind Datenabhängigkeiten die darstellen, daß der Wert eines Gesamtausdrucks abhängig von den Ergebnissen seiner Teilausdrücke ist. In Abb. 5 ist ein Auszug eines feingranularen PDG für den folgenden Programmtext dargestellt.

Die Bedingung `a == b` erzeugt die Knoten 11 bis 13. Der Vergleich in Knoten 11 benötigt die Werte der Variablen `a` und `b` und ist deshalb wertabhängig von Knoten 12 und 13. Die Zuweisung `b = c` erzeugt Knoten 14 bis 16 und ist kontrollabhängig von Knoten 11. Das Ziel dieser Zuweisung ist die Variable `b` in Knoten 16, daher die Wertabhängigkeiten zwischen 14, 15 und 16. Die Werte der Variablen `a`, `b` und `c` in den Knoten 12, 13 und 15 sind datenabhängig von nicht abgebildeten Knoten. Der Wert der Variablen `b` wird später benutzt, daher ist Knoten 16 Ausgang einer Datenabhängigkeitskante.

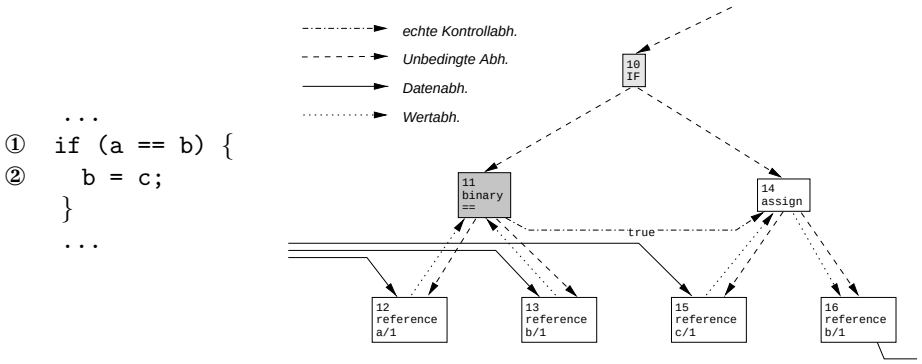

Abbildung 5: Teil eines feingranularen PDGs

Abbildung 6: Beispiel mit Schleife

Aufgrund der feineren Struktur unseres PDGs konnten die bekannten Slicing-Algorithmen nicht ohne Änderungen angewandt werden, da innerhalb von Ausdrücken die Daten- und Kontrollabhängigkeiten Zyklen erzeugen. Die klassischen Algorithmen würden deshalb immer komplette Anweisungen in den Slice einfügen, was zwar zu gleichen Ergebnissen wie traditionelle PDGs führt, aber unsere feingranularen PDGs konterkariert.

4.2 Schleifenprädikate

Ein Beispiel für zusätzlich benötigte Abhängigkeiten ist die Behandlung von Schleifen wie in Abbildung 6. Der traditionelle PDG enthält keine Kontrollabhängigkeit zwischen der Schleife und den nachfolgenden Anweisungen, da davon ausgegangen wird, daß jede Schleife terminiert und so die auf Schleifen folgenden Anweisungen auf jeden Fall ausgeführt werden. Dies hat den Nachteil, daß in den Pfadbedingungen für Zeile 5 nicht die Nachbedingung $(\texttt{i} \leq \texttt{n})$ aufgenommen wird. Wir modifizieren deshalb die Abhängigkeitsgraphen, damit die auf Schleifen nachfolgenden Anweisungen kontrollabhängig von der Schleife sind. (Die nachfolgenden Anweisungen werden nur dann ausgeführt, wenn die Schleife terminiert.) In Abbildung 7 ist der so modifizierte PDG dargestellt.

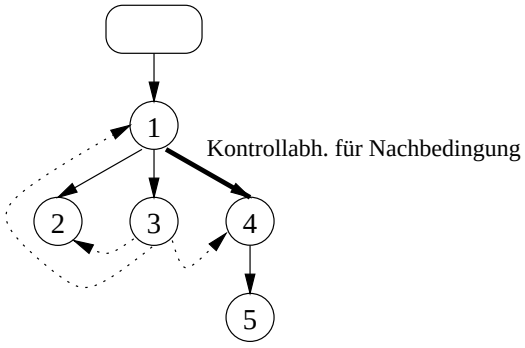

Abbildung 7: Angepaßter PDG zu Abb. 6

5 Erfahrungen

Unsere Techniken wurden im VALSOFT-Projekt [GGS96] in einem Werkzeug für mittelgroße C-Programme implementiert. Gegenüber vollständigem C existieren derzeit noch folgende Einschränkungen:

- striktes ANSI-C
- kein `setjmp`/`longjmp` etc.
- Behandlung von unions und gotos noch nicht implementiert
- bisher nur *flußunabhängige* Points-to-Analyse,
  keine Pointer-Arithmetik

Teil des System ist ein Visualisierungswerkzeug mit den folgenden Aufgaben (siehe Abb. 9):

```
PATH CONDITION

        (i_0 > 0)
    AND (j_5 < 5)
    AND (i_0 < 8)

SUBSTITUTION

j_5 = j_0 OR
    = b_0

a_7 = a_0 OR
    = x_0

y_8 = y_0 OR
    = a_7
```

Abbildung 8: Solver-Ausgabe

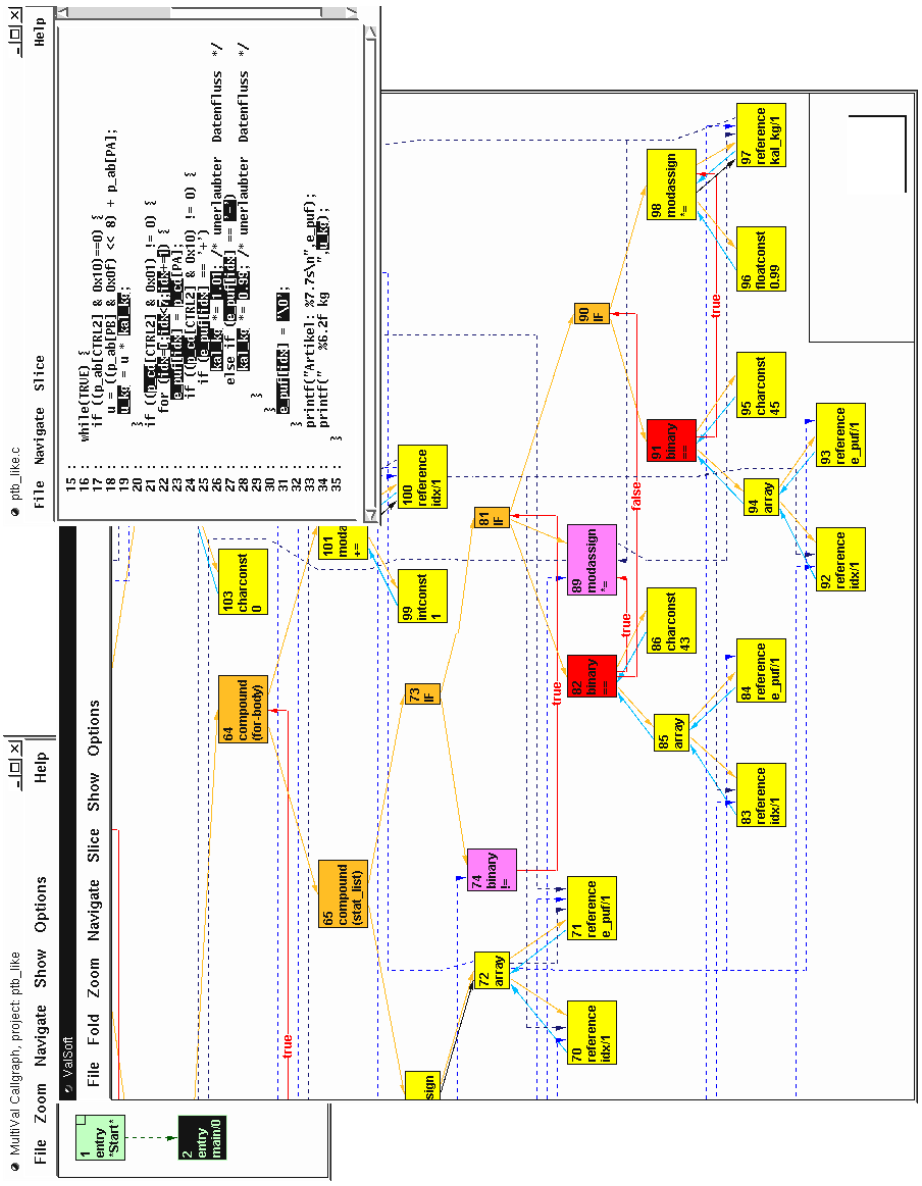

Abbildung 9: GUI

- Visualisierung der PDGs und Aufrufgraphen
- Interaktives Slicing und Chopping
- Darstellung der Slices im PDG und im Quelltext
- Berechnung von Pfadbedingungen „on demand“

Der integrierte Solver kann zwischen interessierenden Knoten im Abhängigkeitsgraphen die Pfadbedingungen berechnen. Diese werden im Visualisierungswerkzeug wie in Abb. 8 präsentiert. Die dort berechneten Bedingungen entstammen dem Beispiel 1 und geben die Pfadbedingungen zwischen Zeile 1 und 8 an. Die Indices an den Variablennamen geben zusätzlich die Zeilennummer des Auftretens der Variablen an. Die zusätzlich angegebenen Substitutionen sind möglich, wurden aber nicht angewendet, da sie die Pfadbedingungen nicht vereinfachen würden.

Wir haben versucht, durch die graphische Präsentation der Programmabhängigkeitsgraphen dem Benutzer ein tieferes Verständnis der analysierten Software zu geben. Leider mussten wir feststellen, daß diese Darstellung trotz aller Hilfsmittel zu komplex ist. Die Visualisierung im Quellext hat deutlich bessere Ergebnisse. Wir glauben aber weiterhin, daß durch geeignete Abstraktionen die Darstellung der Graphen soweit vereinfacht werden kann, daß die Abhängigkeitsbeziehungen deutlich werden.

Unsere Erfahrungen mit diesem Werzeug zeigen, daß für das Programmverstehen die verschiedenen Slicingtechniken nicht ausreichen, da häufig unklar ist, unter welchen Bedingungen Anweisungen Teil von Slices sind. Mit Pfadbedingungen können genau diese Umstände berechnet werden, was zu wirklichem Programm*verstehen* führt.

Literatur

[GGS96] M. Goldapp, U. Grottker und G. Snelting. *Validierung softwaregesteuerter Meßysteme durch Program Slicing und Constraint Solving*. In *Statusseminar des BMBF Softwaretechnologie*, S. 405–425. Berlin, 1996. Auch als Informatik-Bericht 96–02 (März 1996), TU Braunschweig, FB Informatik.

[Kri98] J. Krinke. *Static slicing of threaded programs*. In *Proc. ACM SIGPLAN/SIGFSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98)*, S. 35–42. Montreal, Canada, 1998. ACM SIGPLAN Notices 33(7).

[KRS99] J. Krinke, T. Robschink und G. Snelting. *Software-Sicherheitsprüfung mit VALSOFT*. *Informatik - Forschung und Entwicklung*, 14(2):62–73, 1999.

[KS98] J. Krinke und G. Snelting. *Validation of measurement software as an application of slicing and constraint solving*. *Information and Software Technology*, 40(11-12):661–675, 1998. (special issue: program slicing).

[Sne96] G. Snelting. *Combining slicing and constraint solving for validation of measurement software*. In *Static Analysis; Third International Symposium, SAS'96*, Bd. 1145 von *LNCS*, S. 332–348. Springer Verlag, Aachen, 1996.