



# *Programmverstehen*

Andreas Zeller

Lehrstuhl Softwaretechnik  
Universität des Saarlandes, Saarbrücken



# Software Reengineering

---

An der Universität lernen Sie in der Regel das Entwickeln eines Programms ohne Vorgaben – „from scratch“ oder „auf der grünen Wiese“.

Viel realistischer ist es jedoch, daß Sie *alte Software* vorfinden und warten – ergänzen, portieren oder erweitern.

*Der Umgang mit Software-Altlasten ist das Problem der Softwaretechnik!*



# *Software Reengineering (2)*

---

- Wartungskosten: nach wie vor 50–75% der Gesamtkosten



# *Software Reengineering (2)*

---

- Wartungskosten: nach wie vor 50–75% der Gesamtkosten
- 80% der kommerziellen Anwendungssoftware ist in COBOL geschrieben



# *Software Reengineering (2)*

---

- Wartungskosten: nach wie vor 50–75% der Gesamtkosten
- 80% der kommerziellen Anwendungssoftware ist in COBOL geschrieben – 3/4 davon monolithisch und > 20 Jahre



# Software Reengineering (2)

---

- Wartungskosten: nach wie vor 50–75% der Gesamtkosten
- 80% der kommerziellen Anwendungssoftware ist in COBOL geschrieben – 3/4 davon monolithisch und > 20 Jahre
- Ein typischer Anwender in den USA hat 2200 Programme mit insgesamt 1,15 Mio LOC



# Software Reengineering (2)

---

- Wartungskosten: nach wie vor 50–75% der Gesamtkosten
- 80% der kommerziellen Anwendungssoftware ist in COBOL geschrieben – 3/4 davon monolithisch und > 20 Jahre
- Ein typischer Anwender in den USA hat 2200 Programme mit insgesamt 1,15 Mio LOC
- Deutsche COBOL-Programme
  - sind zu 80% monolithisch
  - sind zu 77% unstrukturiert
  - enthalten zu 93% redundant gehaltene Daten, die aus Unwissenheit oder Unsicherheit nicht gelöscht werden





# Software Reengineering (2)

---

- Wartungskosten: nach wie vor 50–75% der Gesamtkosten
- 80% der kommerziellen Anwendungssoftware ist in COBOL geschrieben – 3/4 davon monolithisch und > 20 Jahre
- Ein typischer Anwender in den USA hat 2200 Programme mit insgesamt 1,15 Mio LOC
- Deutsche COBOL-Programme
  - sind zu 80% monolithisch
  - sind zu 77% unstrukturiert
  - enthalten zu 93% redundant gehaltene Daten, die aus Unwissenheit oder Unsicherheit nicht gelöscht werden
- Jede dritte Anweisung verwendet eine numerische Konstante oder ein Literal (d.h. hartcodierte Datenwerte)



# *Software Reengineering (3)*

---

- Ein Wartungsprogrammierer ist für 32.000 Codezeilen verantwortlich





# Software Reengineering (3)

---

- Ein Wartungsprogrammierer ist für 32.000 Codezeilen verantwortlich
- Ein Wartungsprogrammierer benötigt
  - 47% seiner Zeit für Programmanalyse
  - 15% für Programmierung
  - 28% für den Test
  - 9% für die Dokumentation





# Software Reengineering (3)

---

- Ein Wartungsprogrammierer ist für 32.000 Codezeilen verantwortlich
- Ein Wartungsprogrammierer benötigt
  - 47% seiner Zeit für Programmanalyse
  - 15% für Programmierung
  - 28% für den Test
  - 9% für die Dokumentation
- *Entropiezuwachs*: Die Komplexität einer Prozedur steigt
  - nach einer Fehlerkorrektur um durchschnittlich 4%
  - nach einer Änderung um 17% und
  - nach einer Erweiterung um 26%





# Software Reengineering (3)

---

- Ein Wartungsprogrammierer ist für 32.000 Codezeilen verantwortlich
- Ein Wartungsprogrammierer benötigt
  - 47% seiner Zeit für Programmanalyse
  - 15% für Programmierung
  - 28% für den Test
  - 9% für die Dokumentation
- *Entropiezuwachs*: Die Komplexität einer Prozedur steigt
  - nach einer Fehlerkorrektur um durchschnittlich 4%
  - nach einer Änderung um 17% und
  - nach einer Erweiterung um 26%
- Bei der *US Air Force* kostet die Änderung einer einzelnen Zeile 2500–3000\$ (1990)



# *Einige Begriffe*

---

**Reverse Engineering** Extraktion und Repräsentation von Informationen aus einem Software-System

- in einer anderen Form oder
- auf höherem Abstraktionsniveau



4/30



# *Einige Begriffe*

---

**Reverse Engineering** Extraktion und Repräsentation von Informationen aus einem Software-System

- in einer anderen Form oder
- auf höherem Abstraktionsniveau

**Restrukturierung** Transformation zwischen Repräsentationsformalismen ohne Änderung der Funktionalität





# *Einige Begriffe*

---

**Reverse Engineering** Extraktion und Repräsentation von Informationen aus einem Software-System

- in einer anderen Form oder
- auf höherem Abstraktionsniveau

**Restrukturierung** Transformation zwischen Repräsentationsformalismen ohne Änderung der Funktionalität

**Reengineering** Alle Aktivitäten, die nach Inbetriebnahme eines Programmsystems

- das Verständnis von Software erhöhen oder
- Wartbarkeit, Wiederverwendbarkeit oder Weiterentwickelbarkeit verbessern oder erst ermöglichen





# *Einige Begriffe*

---

**Reverse Engineering** Extraktion und Repräsentation von Informationen aus einem Software-System

- in einer anderen Form oder
- auf höherem Abstraktionsniveau

**Restrukturierung** Transformation zwischen Repräsentationsformalismen ohne Änderung der Funktionalität

**Reengineering** Alle Aktivitäten, die nach Inbetriebnahme eines Programmsystems

- das Verständnis von Software erhöhen oder
- Wartbarkeit, Wiederverwendbarkeit oder Weiterentwickelbarkeit verbessern oder erst ermöglichen

*Reengineering = Reverse Engineering + Restrukturierung*





# Programmverstehen

---

**Programmverstehen** ist der wichtigste (und älteste) Bestandteil des *Reverse Engineering*

Typischerweise Konstruktion *alternativer Sichten*, die helfen, das System besser zu verstehen:

- Erzeugen eines Ablaufdiagramms / Struktogramms
- Querverweis-Tabellen (*cross references*)
- Anreichern von Quellcode um Metrikwerte
- UML-Diagramme aus OO-Quellcode
- Software-Visualisierung

Wir betrachten nun einige typische Werkzeuge.





demo Project Documentation - Mozilla (Build ID: 2002052309)

File Edit View Go Bookmarks Tools Window Help

http://www.imagix.com/doc\_samp/index.htm Search

Project Files Classes Functions LibFuncs Macros Variables Types PreTypes

## Variables

A	B	C	D	E	F	G
H	I	J	K	L	M	N
O	P	Q	R	S	T	U
V	W	X	Y	Z	other	

codefile\_name (main.c)  
column  
comment\_level (bsl.lex.c)  
cond (IfStmt)  
cond (WhileStmt)  
condval (YYSTYPE)  
condval (YYSTYPE)  
currId (List)  
current (List)  
current\_scope (SymbolTable)

### codefile\_name

category: variable  
type: char\*  
scope: static

#### Location

file: /files4/test/rels/3.3.4/imagix/data/demo/main.c  
line: 11  
owner: imagix  
mod date: 03 Jul 1995 (18:08:03)

#### Declaration

```
static char *codefile_name = NULL;
```

#### Variable Usage

Variable: codefile\_name

main codefile\_name  
process\_args

http://www.imagix.com/doc\_samp/htm/302.htm#302



# Imagix: Funktionen und Variablen



The screenshot shows the Imagix 4D / Imagix 2000 interface in 'Project: tk1 -- Mode: Browse'. The main window displays a dependency graph with nodes for functions and variables, connected by lines. The right-hand pane shows a project browser with a tree structure. The 'OptionInit' node is highlighted in yellow. The bottom of the window has a toolbar with various icons and checkboxes for 'Files', 'Functions', 'Variables', 'Classes', 'Macros', 'Types', '3D', 'Vertical', 'XRef', and 'Compact'. A 'Control' button is visible at the bottom right.

Imagix 4D / Imagix 2000 -- Project: tk1 -- Mode: Browse

File Edit Mode Select Options Help

tk3.s

- tkInth
  - TkMainInfo
  - optionRootPtr
  - winPtr
- tkOption.c
  - GetDefaultOptions
  - NewArray
  - OptionInit**
  - SetupStacks
  - Tk\_AddOption
  - defaultMatch
  - levels
  - numLevels
  - stacks
- Element
  - child
  - flags
  - nameUid
  - priority
  - child
  - valueUid
- StackLevel

Files Functions Variables 3D Vertical  
Classes Macros Types XRef Compact

Contains Grep Attr.

Control



# Imagix: Benutzung



The screenshot shows the Imagix 4D / Imagix 2000 interface. The main window displays a call graph for the 'optionInt' function. The graph shows 'optionInt' as a central node with arrows pointing to various other functions and variables. A pop-up window is open over the 'NewArray' node, providing detailed information about it.

**optionInt** call graph nodes:

- numLevels
- levels
- malloc
- stacks
- NewArray
- bases
- nameUi
- default
- valueUi
- child
- priority
- flags
- optionR
- Tcl\_Cre
- winPtr
- GetDefa
- Tcl\_De

**SetupStacks** and **Tk\_AddOption** also have arrows pointing to **optionInt**.

**Pop-up window for NewArray:**

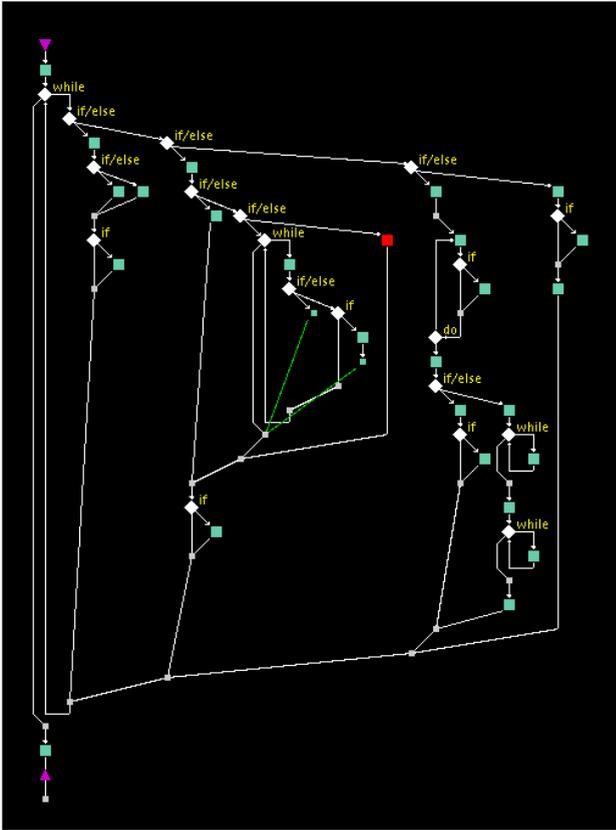
- category:** function
- scope:** static
- Location**
  - file:** /files4/src/tcl/3.6/tk3.6/tkoption.c
  - lines:** 913 - 923 (11 lines)
  - mod date:** 17 Jan 1996 (12:29:55)
- Metrics**
  - lines:** 11 code, 11 total
  - complexity:** 1
  - callers:** 2
- Declaration**

```
static ElArray *
NewArray(numEls)
    int numEls;                /* How many elements of space to allocate. */
```

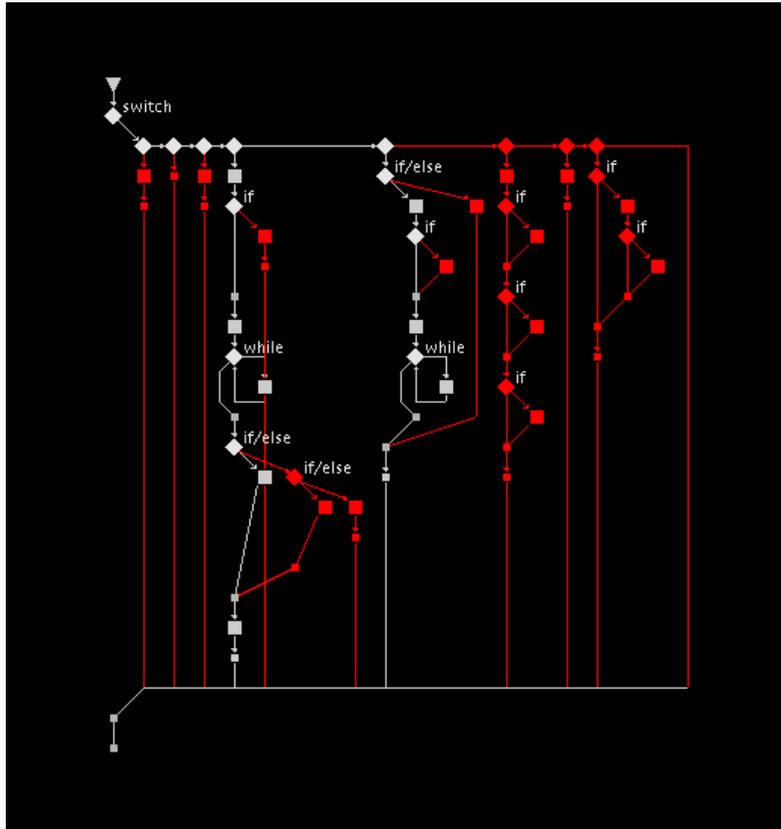




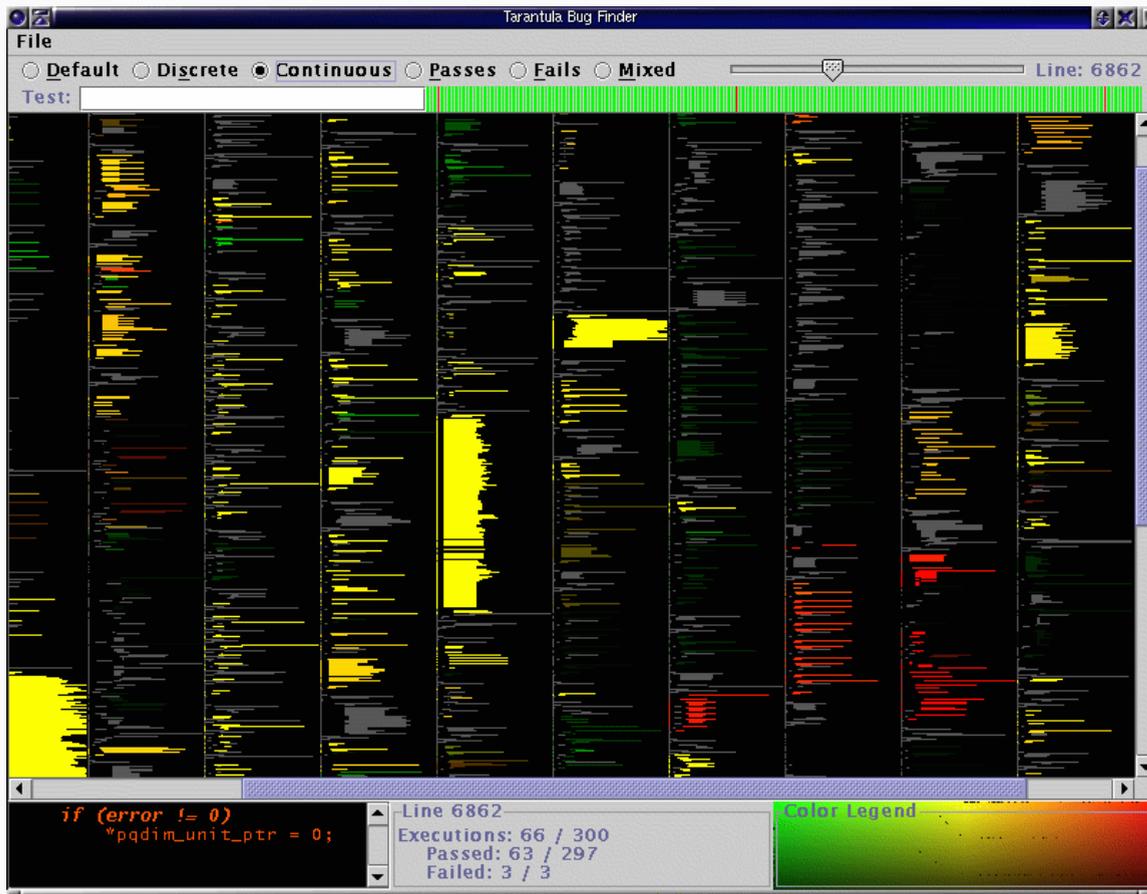
# Imagix: Flussdiagramm



# Imagix: Abdeckung



# Tarantula: Abdeckung vergleichen



# Jinsight: Programmläufe analysieren



The screenshot displays several windows from the Jinsight application:

- Call Tree:** Shows a tree view of method calls starting from `java.io.PrintStream.println(String)`. The tree includes `print`, `write`, and `write` sub-nodes.
- Execution [Workspace 1]:** A thread execution view showing multiple threads (Thread-2 to Thread-7) and the `main` thread. It includes a timeline and a table of execution times.
- Invocations [Workspace 1]:** A table listing method names and their invocation counts. The table has columns for 'method name' and 'count'.
- Execution Pattern:** A detailed view of a specific execution pattern, showing a sequence of method calls like `getObject`, `handleGetObject`, `loadLookup`, and `getContents`.
- Reference Pattern:** A diagram showing the relationships between objects in memory. It includes nodes for `StringSortedV`, `MethodDrawStr`, `ObjectDrawStr`, `AWTEvent`, and `MenuItem`.
- Histogram of methods [Workspace 1]:** A bar chart showing the frequency of method calls across different classes. The x-axis lists classes like `java.lang.String`, `jinsight.Main/Controller`, and `java.lang.Runtime`. The y-axis represents the number of calls, with a scale up to 33000+.





## ***Fazit: Visualisierung***

---

Die Darstellung *alternativer Sichten* kann sehr hilfreich beim Programmverstehen sein – insbesondere,

- wenn mit Navigation gekoppelt
- wenn dynamische Information dargestellt wird, die sich sonst der Untersuchung entzieht

Weitere Ansätze:

Vorlesung *Software-Visualisierung* (WS 2002/03)



# Programmanalyse

---

Jede Visualisierung hat ihre Grenzen in der Aufnahmefähigkeit des Menschen

Deshalb wichtig: *Fokussierung* auf Teilaspekte des Programms

Das ist die Aufgabe der *Programmanalyse*





## ***Programmanalyse (2)***

---

Man unterscheidet

**Statische Programmanalyse** bestimmt Eigenschaften eines Programms (= über alle möglichen Läufe)

**Dynamische Programmanalyse** bestimmt Eigenschaften eines Programmlaufs (oder mehrerer Programmläufe)

Wichtiges Hilfsmittel im *Reverse Engineering* (aber auch für Prüfung der Korrektheits und der Sicherheit)





# Program Slicing

---

Program Slicing hat das Ziel, *Anweisungen zu isolieren*, die für bestimmte Programmzustände relevant sind:

- *Abhängigkeiten* zwischen Ein- und Ausgaben erkennen
- Anweisungen bestimmen, die andere *beeinflussen* können
- Auswirkungen von *Code-Änderungen* eingrenzen

Zentrale Technik der *Fokussierung* auf relevante Teilaspekte des Programms



# Beispiel: Summe und Produkt berechnen

```
int main() {
    int a, b, sum, mul;
    sum = 0;
    mul = 1;
    a = read();
    b = read();
    while (a <= b) {
        sum = sum + a;
        mul = mul * a;
        a = a + 1;
    }
    write(sum);
    write(mul);
}
```



# Beispiel: Summe und Produkt berechnen



```
int main() {
    int a, b, sum, mul;
    sum = 0;
    mul = 1;
    a = read();
    b = read();
    while (a <= b) {
        sum = sum + a;
        mul = mul * a;
        a = a + 1;
    }
    write(sum);
    write(mul); ← Was kann mul hier beeinflussen?
}
```



# *Slicing*

---

*Program Slicing* versucht, für eine Berechnung irrelevante Anweisungen eines Programms zu entfernen oder relevante Anweisungen zu markieren.

Ergebnis: Eine Teilmenge des Programms – ein *Slice*.



# Slicing

---

*Program Slicing* versucht, für eine Berechnung irrelevante Anweisungen eines Programms zu entfernen oder relevante Anweisungen zu markieren.

Ergebnis: Eine Teilmenge des Programms – ein *Slice*.

Grundlage: Das *Slicing-Kriterium*  $(v, n)$  spezifiziert den Slice für eine Variable  $v$  bei einer Anweisung  $n$ .

Ein Slice wird berechnet, indem möglichst viele Anweisungen des Programms *gelöscht* werden, ohne daß sich die Berechnung des Wertes für die Variable  $v$  bei der Anweisung  $n$  ändert.





# Backward Slicing

---

*Backward-Slice*: Anweisungen, die den Wert einer Variablen an einer Stelle im Programm beeinflussen

```
int main() {  
    int a, b, sum, mul;  
    sum = 0;  
    mul = 1;  
    a = read();  
    b = read();  
    while (a <= b) {  
        sum = sum + a;  
        mul = mul * a;  
        a = a + 1;  
    }  
    write(sum);  
    write(mul);  
}
```

Programm

```
int main() {  
    int a, b, sum, mul;  
  
    mul = 1;  
    a = read();  
    b = read();  
    while (a <= b) {  
        mul = mul * a;  
        a = a + 1;  
    }  
  
    write(mul);  
}
```

Backward-Slice für (mul, 13)

Man betrachtet *rückwärts* Wirkungen früherer Anweisungen





# Forward Slicing

*Forward-Slice*: alle Anweisungen, die durch eine Änderung der Variablen  $v$  bei der Anweisung  $n$  betroffen sind

```
int main() {
    int a, b, sum, mul;
    sum = 0;
    mul = 1;
    a = read();
    b = read();
    while (a <= b) {
        sum = sum + a;
        mul = mul * a;
        a = a + 1;
    }
    write(sum);
    write(mul);
}
```

Programm

```
int main() {
    int a, b, sum, mul;

    b = read();
    while (a <= b) {
        sum = sum + a;
        mul = mul * a;
        a = a + 1;
    }
    write(sum);
    write(mul);
}
```

Forward-Slice für (b, 6)

Man betrachtet *vorwärts* die Auswirkungen im Programm





# Programm-Abhängigkeits-Graph

---

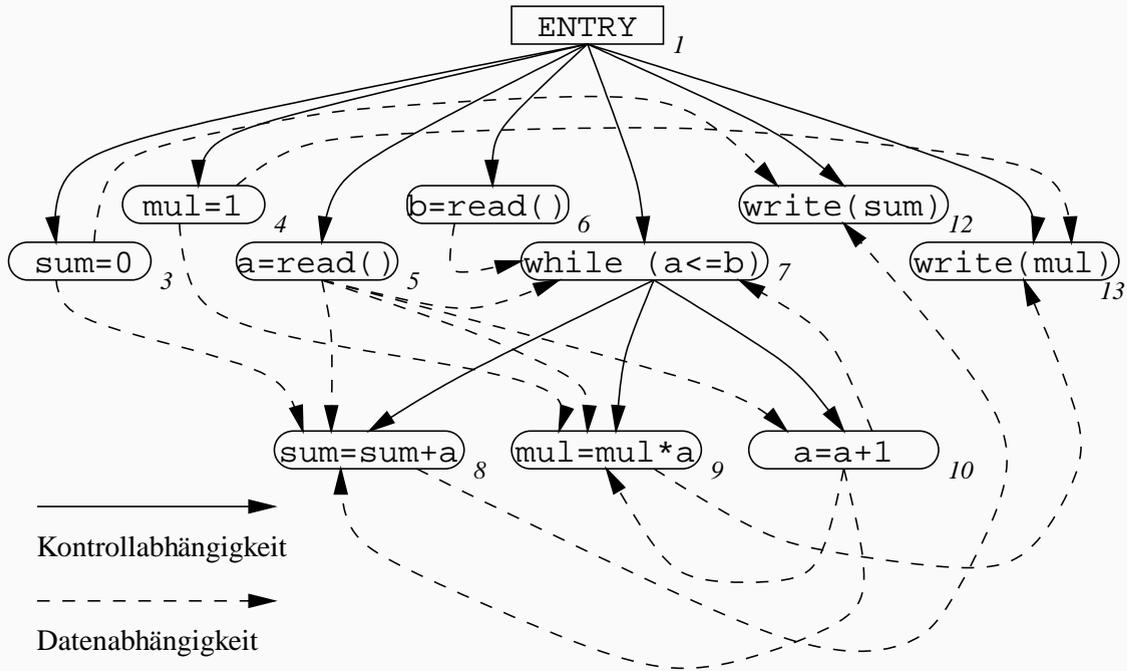
Grundlage des Program Slicing sind *Programm-Abhängigkeits-Graphen* (engl. *program dependency graph*, PDG)

PDGs sind ähnlich aufgebaut wie Kontrollflußgraphen; ihre Kanten geben jedoch *Kontroll-* und *Datenabhängigkeiten* zwischen zwei Anweisungen *A* und *B* an:

- Zwischen *A* und *B* besteht eine *Kontrollabhängigkeit*, wenn *A* beeinflusst, ob oder wie oft *B* ausgeführt wird.
- Zwischen *A* und *B* besteht eine *Datenabhängigkeit*, wenn in *A* einer Variablen ein Wert zugewiesen und dieser Wert in *B* durch Auswertung dieser Variablen benutzt wird.



# Programm-Abhängigkeits-Graph (2)





# *Slicing anhand des PDG*

---

Liegt der PDG vor, ist das Slicing selbst nur noch ein Erreichbarkeitsproblem.

Zuerst besteht ein Slice-Kriterium im Graphen nur noch aus einem Knoten  $n$ , der auch für die in diesem Knoten zugewiesene Variable steht.

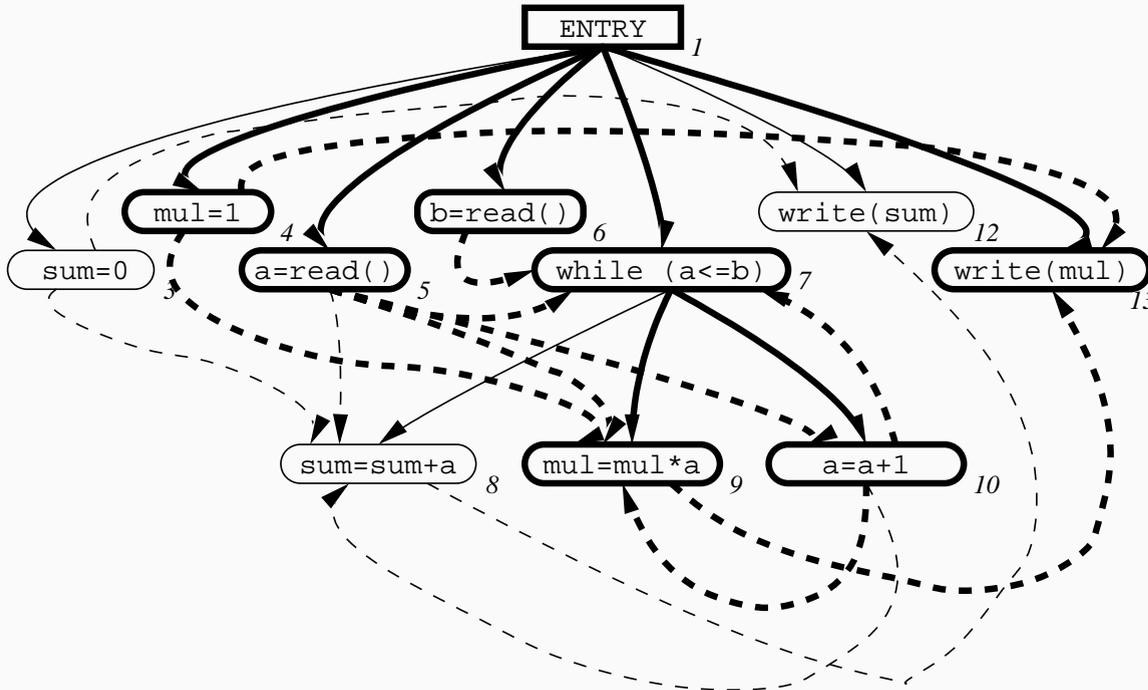
- Ein *Forward-Slice* besteht aus dem Teilgraphen, der alle von  $n$  aus erreichbaren Knoten enthält.
- Ein *Backward-Slice* besteht aus dem Teilgraphen, der alle Knoten enthält, von denen aus  $n$  zu erreichen ist.



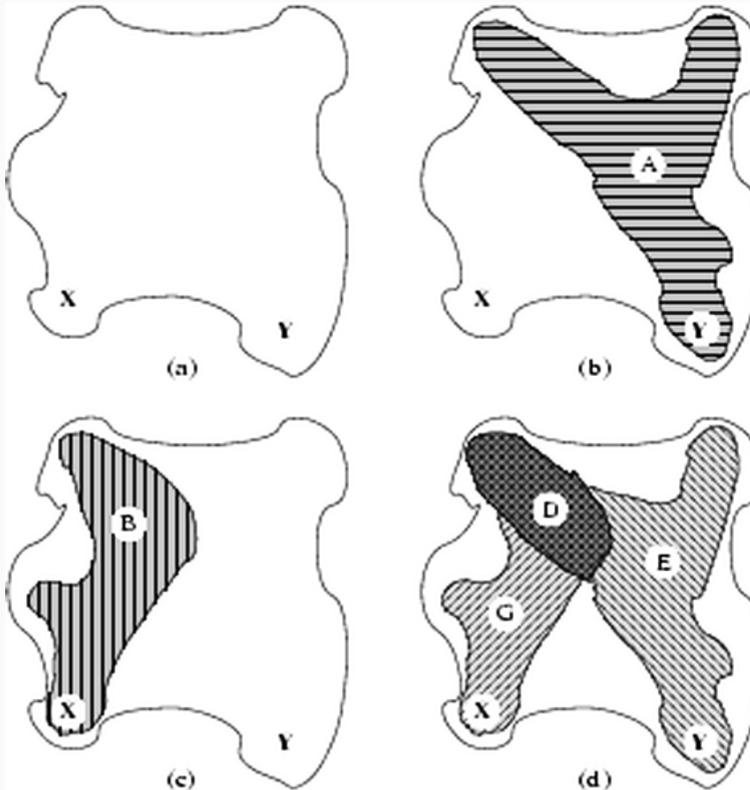
# Slice für (mul, 13) im PDG



Backward-Slice für Zeile 13:



# Mengenoperationen mit Slices



- A: Slice für Y
- B: Slice für X
- D: Schnitt  $A \cup B$  (*Backbone*)
- E: Subtraktion  $A - B$  (*Dice*)
- G: Subtraktion  $B - A$  (*Dice*)



## Dice: Subtraktion zweier Slices

```
int main() {
    int a, b, sum, mul;
    sum = 0;
    mul = 1;
    a = read();
    b = read();
    while (a <= b) {
        sum = sum + a;
        mul = mul * a;
        a = a + 1;
    }
    write(sum);
    write(mul);
}
```

Programm

```
int main() {
    int a, b, sum, mul;

    mul = 1;

    mul = mul * a;

    write(mul);
}
```

Dice für (sum, 12) und (mul, 13)

Grundidee: *Fehlereingrenzung durch Dices*

(Subtraktion „fehlerhafter“ – „korrekter“ Slice)





# *Erweiterungen und Varianten*

---

**Interprozedurales Slicing** bestimmt Slices über  
Prozedurgrenzen hinweg

Herausforderung: Trade-off zwischen

- Entfalten der Prozeduren an Aufrufstelle ( $\Rightarrow$  viel Platz)
- Zusammenfassen der Aufrufe ( $\Rightarrow$  Ungenauigkeit)





# *Erweiterungen und Varianten*

---

**Interprozedurales Slicing** bestimmt Slices über Prozedurgrenzen hinweg

Herausforderung: Trade-off zwischen

- Entfalten der Prozeduren an Aufrufstelle ( $\Rightarrow$  viel Platz)
- Zusammenfassen der Aufrufe ( $\Rightarrow$  Ungenauigkeit)

**Dynamisches Slicing** bestimmt Slices für einen bestimmten Programmablauf

Vorteil: Kontrollfluß ist bekannt

$\Rightarrow$  exaktere Datenabhängigkeiten

$\Rightarrow$  höhere Präzision als statisches Slicing

Herausforderungen: Effiziente Code-Instrumentierung, präzise Kontrollabhängigkeiten





# *Weitere Techniken der Programmanalyse*

**Laufzeitanalyse** Bestimmen einer Obergrenze für die maximale Laufzeit eines Programms oder Programmteils (vgl. Lehrstuhl Wilhelm)





# *Weitere Techniken der Programmanalyse*

**Laufzeitanalyse** Bestimmen einer Obergrenze für die maximale Laufzeit eines Programms oder Programmteils (vgl. Lehrstuhl Wilhelm)

**Ursache-Wirkungs-Ketten** Bestimmen von Ursachen und Wirkungen im Programm durch systematische Experimente (vgl. Lehrstuhl Zeller)

All diese Ergebnisse müssen geeignet dargestellt werden!





# ***Weitere Techniken der Programmanalyse***

**Laufzeitanalyse** Bestimmen einer Obergrenze für die maximale Laufzeit eines Programms oder Programmteils (vgl. Lehrstuhl Wilhelm)

**Ursache-Wirkungs-Ketten** Bestimmen von Ursachen und Wirkungen im Programm durch systematische Experimente (vgl. Lehrstuhl Zeller)

All diese Ergebnisse müssen geeignet dargestellt werden!

Mehr zum Thema „Programmanalyse und Programmverstehen“:  
Vorlesung *Automated Debugging* (WS 2002/03)





# Zusammenfassung

---

- *Software Reengineering* = Reverse Engineering + Restrukturierung
- *Reverse Engineering* ist die Extraktion und Repräsentation von Informationen aus einem Software-System
- *Programmverstehen* ist der wichtigste Bestandteil des Reverse Engineering
- Sichten wie Dokumentation, Benutzung, Abdeckung helfen, das Programm zu verstehen
- *Program Slicing* kann helfen, die für ein Verhalten *wesentliche Aspekte* eines Programms zu isolieren

