Programmverstehen

Andreas Zeller
Lehrstuhl für Softwaretechnik
Universität des Saarlandes, Saarbrücken

2005-12-12

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
- 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
- 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[1]

- 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

[1] Vgl. *Forward Engineering:* Transformation der *abstrakten* Spezifikation in eine *konkrete* Implementierung

Programmverstehen

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

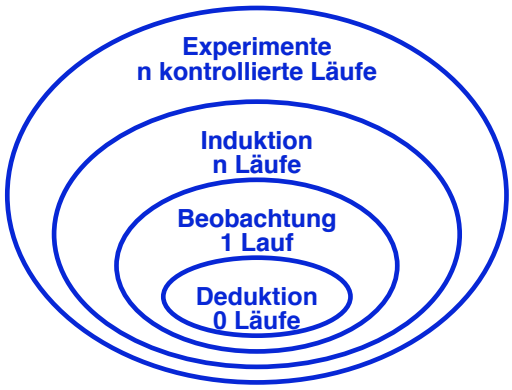
Typischerweise Konstruktion *alternativer Sichten*, die helfen, das System besser zu verstehen:

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

Schließen über Programme

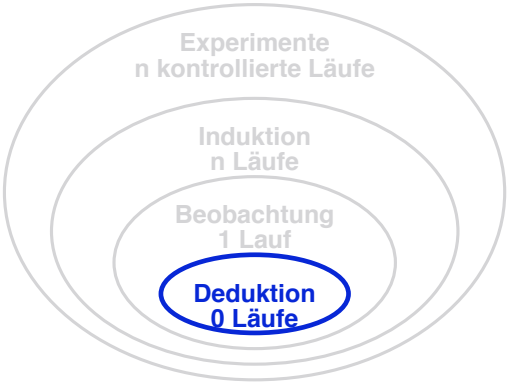
Zutaten: Programmcode, Programmläufe und Experimente

Abhängig von den Zutaten unterscheiden wir folgende Verfahren, ein Programm zu verstehen:



Deduktion

Deduktion: Schließen vom (abstrakten) Code auf die (konkreten) möglichen Läufe



Grundlage: Programmcode (genauer: *nur* Programmcode)

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); ⇐ 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*.

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

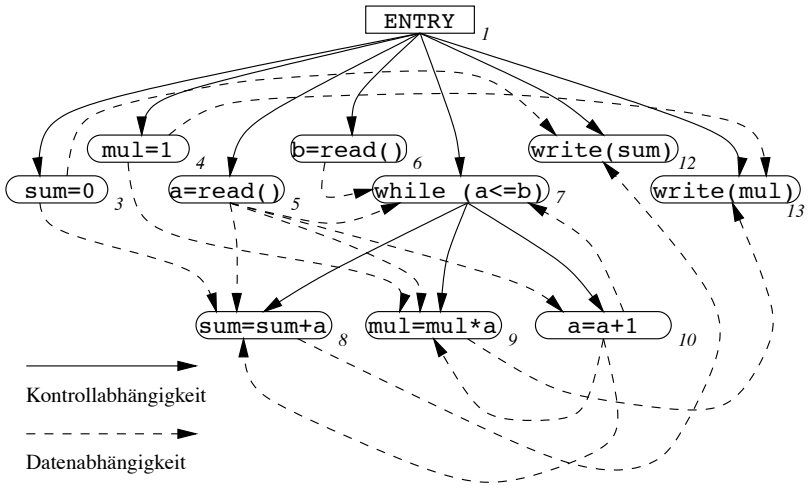
Programm-Abhängigkeits-Graph

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

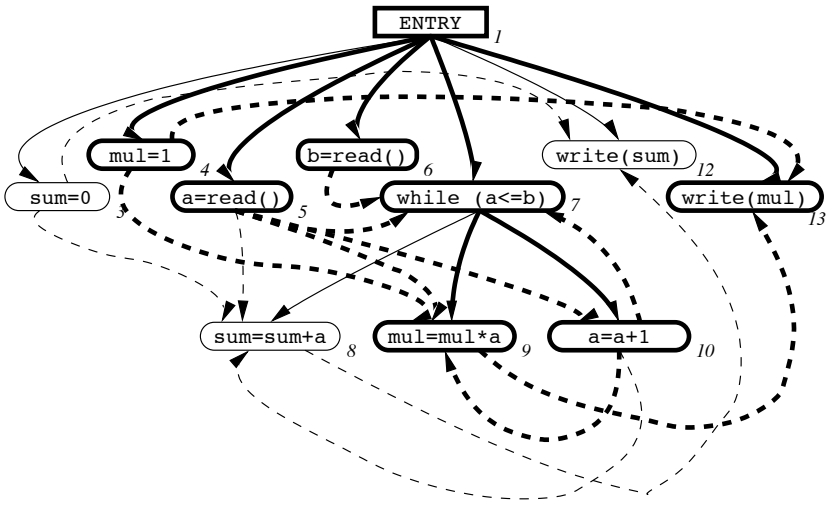
Liegt der PDG vor, ist das Slicing selbst nur noch ein Erreichbarkeitsproblem.

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

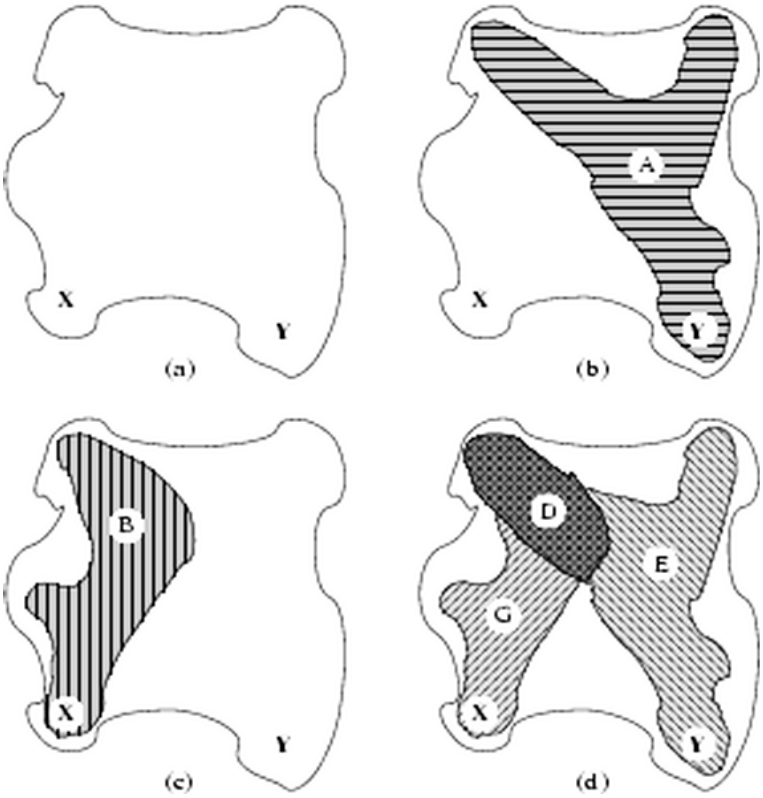
- Ein *Forward-Slice* besteht aus dem Teilgraphen, der alle von n aus erreichbaren Knoten enthält.
- Ein *Backward-Slice* besteht aus dem Teilgraphen, der alle Knoten enthält, von denen aus n zu erreichen ist.

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

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 $(\texttt{sum}, 12)$ und $(\texttt{mul}, 13)$

Grundidee: *Fehlereingrenzung durch Dices*
(Subtraktion „fehlerhafter“ − „korrekter“ Slice)

Erweiterungen und Varianten

Interprozedurales Slicing bestimmt Slices über Prozedurgrenzen hinweg

Herausforderung: Trade-off zwischen

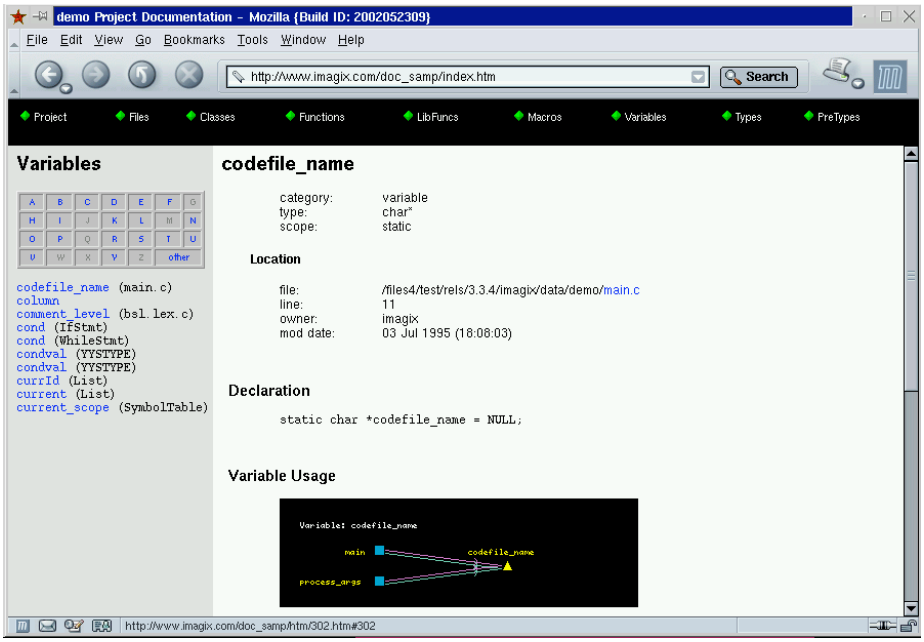
- Entfalten der Prozeduren an Aufrufstelle (⇒ viel Platz)
- Zusammenfassen der Aufrufe (⇒ Ungenauigkeit)

Dynamisches Slicing bestimmt Slices für einen bestimmten Programmlauf

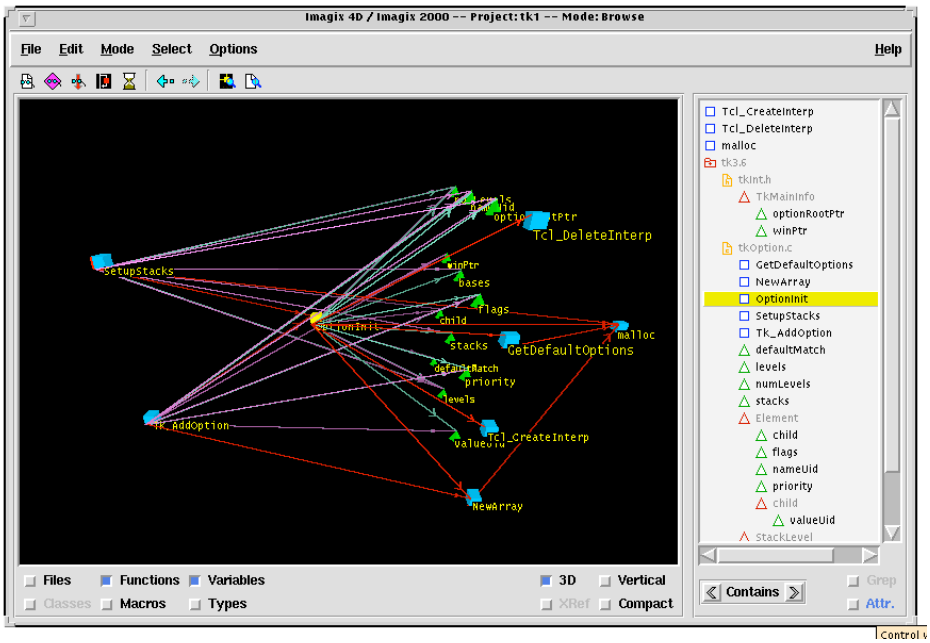
Vorteil: Kontrollfluß ist bekannt
⇒ exaktere Datenabhängigkeiten
⇒ höhere Präzision als statisches Slicing

Herausforderungen: Effiziente Code-Instrumentierung, präzise Kontrollabhängigkeiten

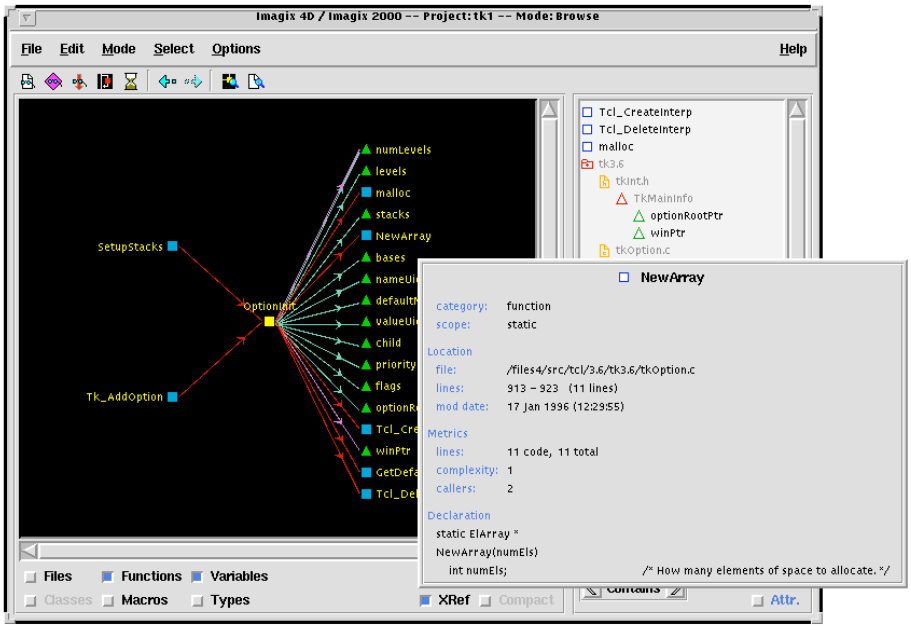
Visualisierung mit Imagix



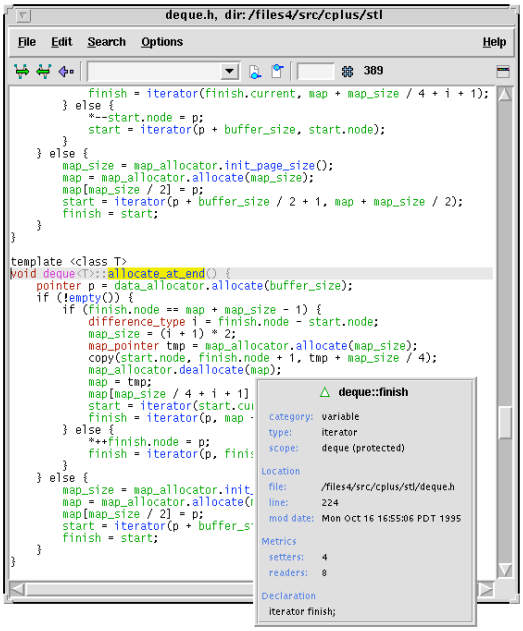
Imagix: Funktionen und Variablen



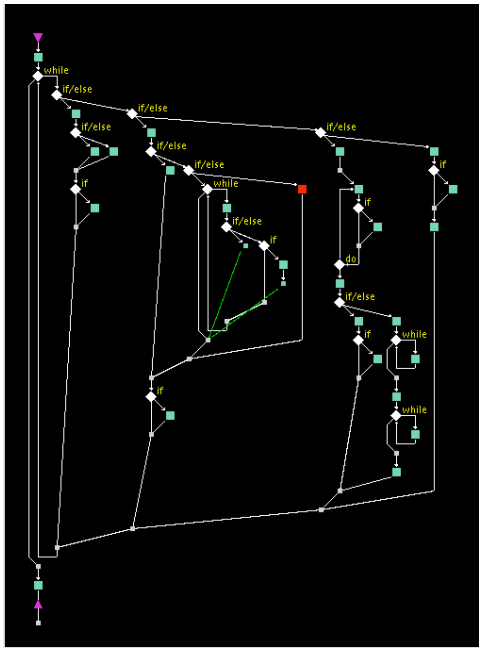
Imagix: Benutzung



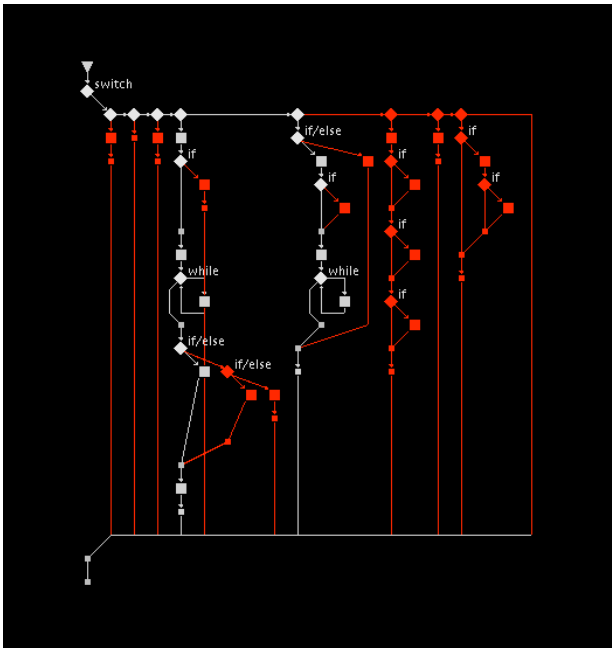
Imagix: Pretty Printing



Imagix: Flussdiagramm



Imagix: Abdeckung



Statisch vs. Dynamisch

Man unterscheidet

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

Technik: Deduktion

Haupteinsatz: *Verifikation*

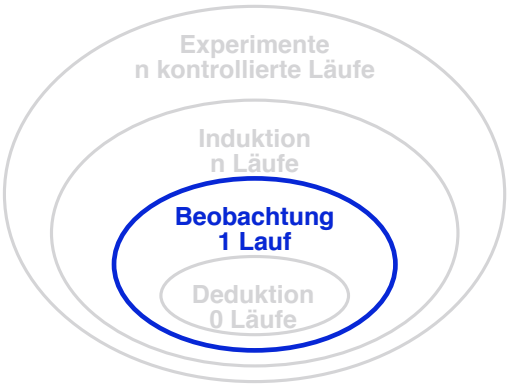
Dynamische Programmanalyse bestimmt Eigenschaften eines Programmlaufs (oder mehrerer Programmläufe)

Techniken: Beobachtung, Induktion, Experimentieren

Haupteinsatz: *Anomalien finden, Fehlersuche*

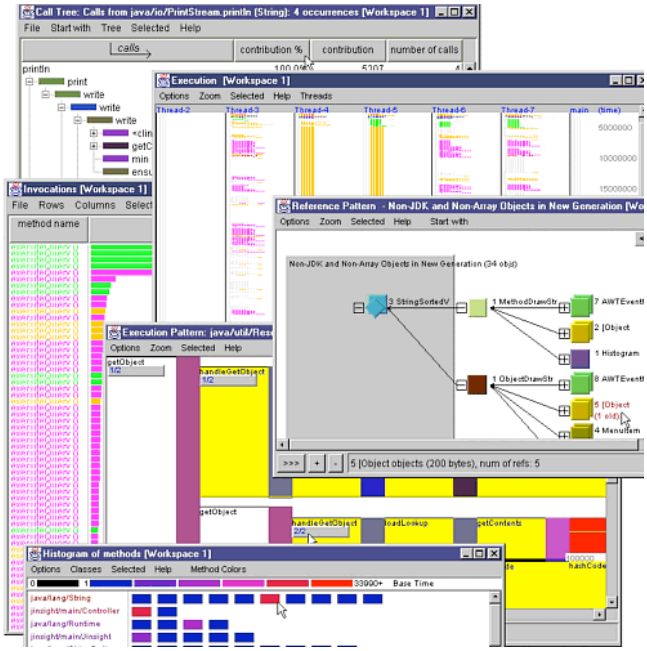
Beobachtung

Mit *Beobachtung* kommen *Fakten* über einen konkreten Lauf hinzu:



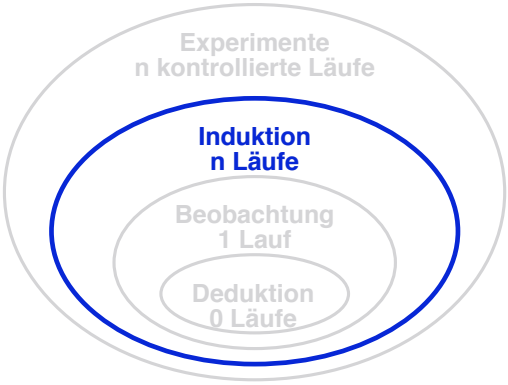
Beispiel: Debugger, Abdeckung, Dynamische Slices…
Gewöhnlich beobachten Menschen das Programm

Jinsight: Programmläufe analysieren



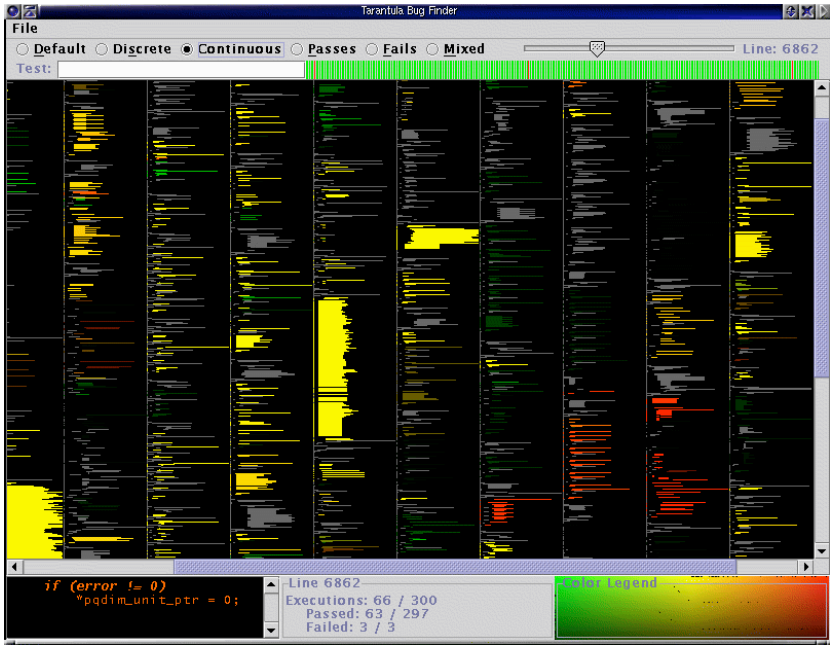
Induktion

Induktion generiert *Abstraktionen* aus mehrfachen Läufen:



Beispiel: Abdeckung vergleichen, dynamische Dices, dynamische Invarianten…
Induktion hebt Anomalien hervor

Tarantula: Abdeckung vergleichen



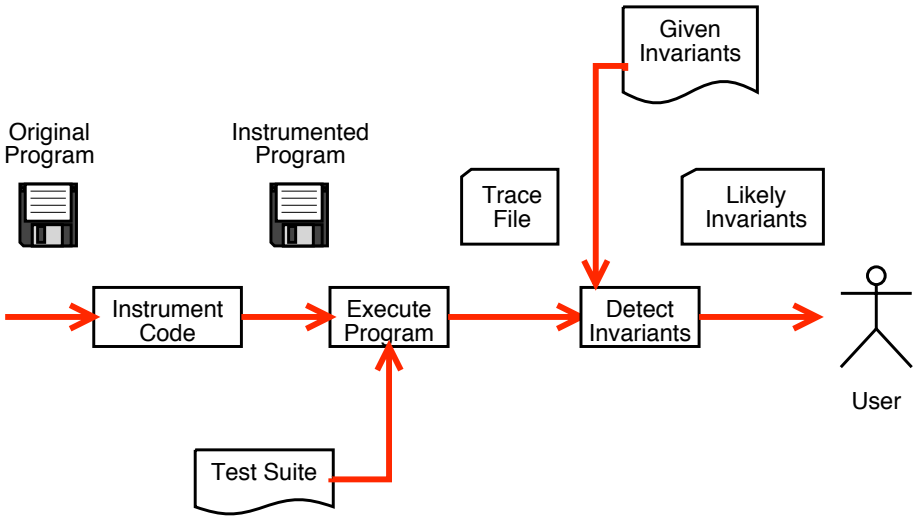
Invarianten bestimmen

Grundidee: *Aus konkreten Läufen Invarianten bestimmen*

Invarianten beschreiben, was über *alle* beobachteten Läufe gilt.

Werden nur funktionierende Läufe betrachtet, kann man die so gewonnenen Invarianten benutzen, um ggf. Verletzungen in fehlschlagenden Läufen zu bestimmen.

Invarianten bestimmen mit DAIKON[2]



Invarianten bestimmen mit DAIKON (2)

```
$ ./sort 1 2 3
1 2 3
$ ./sort 4 5 6
4 5 6
```

DAIKON erkennt folgende Invarianten:

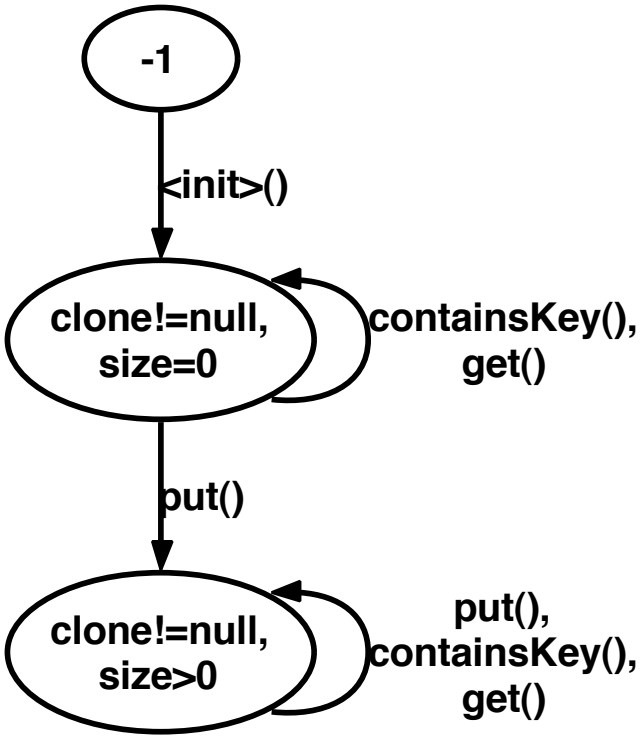
```
std.shell_sort(int *;int;):::ENTER
size == size(a[])
a[] one of  [1, 2, 3], [4, 5, 6]
size == 3

std.shell_sort(int *;int;):::EXIT1
a[] == orig(a[])
```

Model Mining

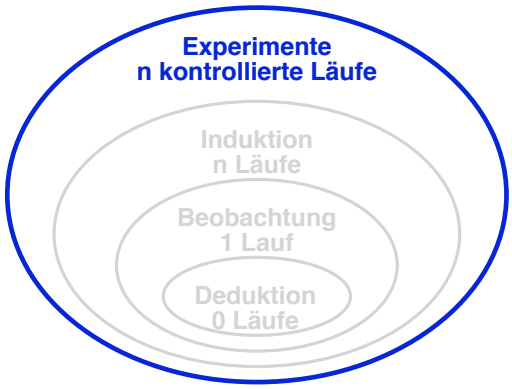
Aus Methodenfolgen endliche Automaten generieren

[2] Daikon – http://pag.lcs.mit.edu/daikon/



Experimentieren

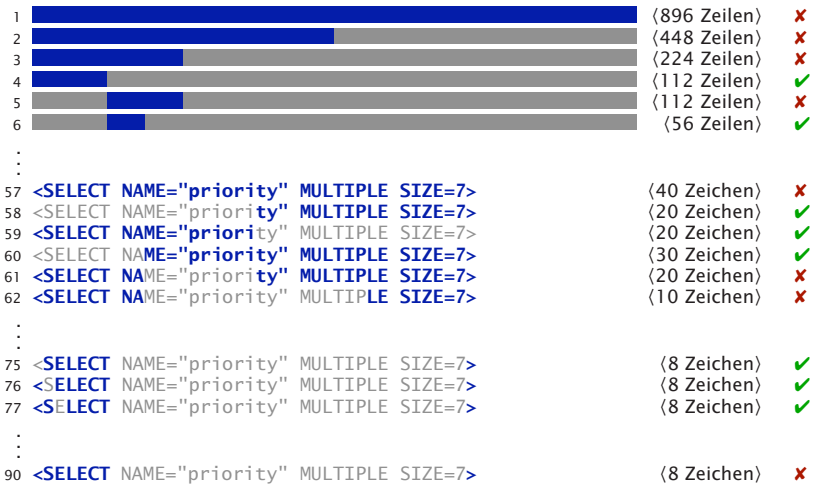
Experimentation findet *Ursachen* gegebener Wirkungen:



Beispiele: (manuelle) Fehlersuche
Kann ebenfalls automatisiert werden

Systematisches Experimentieren

Mozilla stürzt beim Drucken einer Webseite ab.



Vereinfachte Fehlerbeschreibung:
`Mozilla stürzt ab, wenn ich <SELECT> drucke.`

Weitere Anwendungen

Fehlerverursachende Eingaben. Eine Eingabe verursacht einen Fehler.

Welche (Teil-)Eingabe war fehlerverursachend?

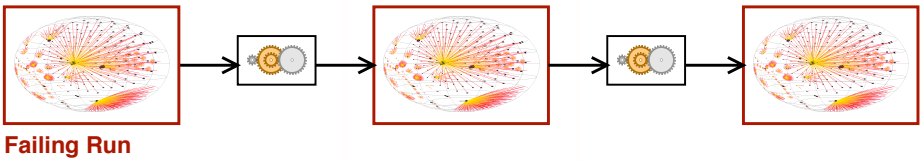
Fehlerverursachende Änderungen. Nach einer Änderung funktioniert das Programm nicht mehr.

Welche (Teil-)Änderung war fehlerverursachend?

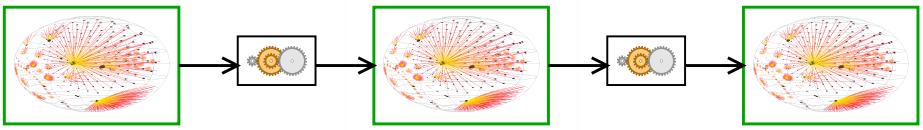
Fehlerverursachende Programmzustände. Zwei Läufe – einer funktioniert, einer schlägt fehl.

Welche Unterschiede im Programmzustand sind fehlerverursachend?

Delta Debugging

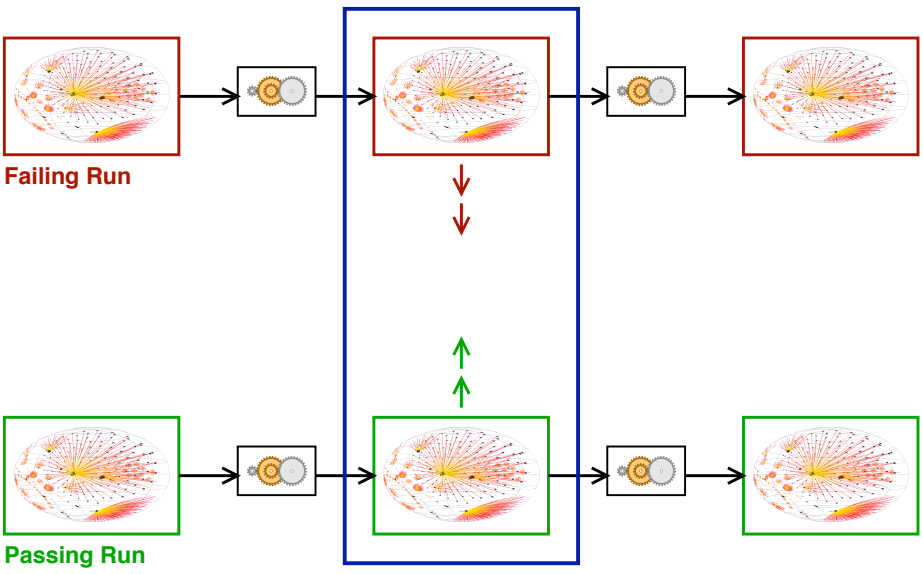


Failing Run



Passing Run

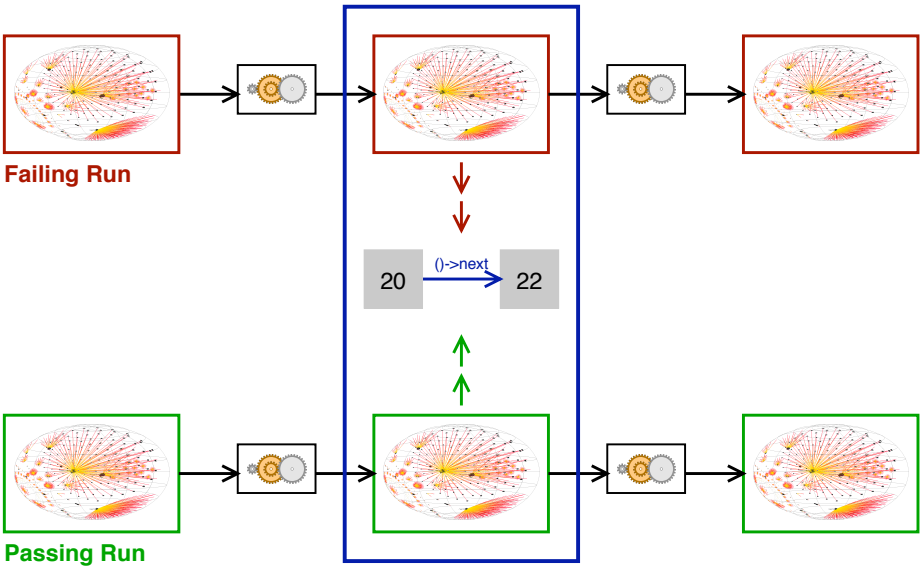
Delta Debugging



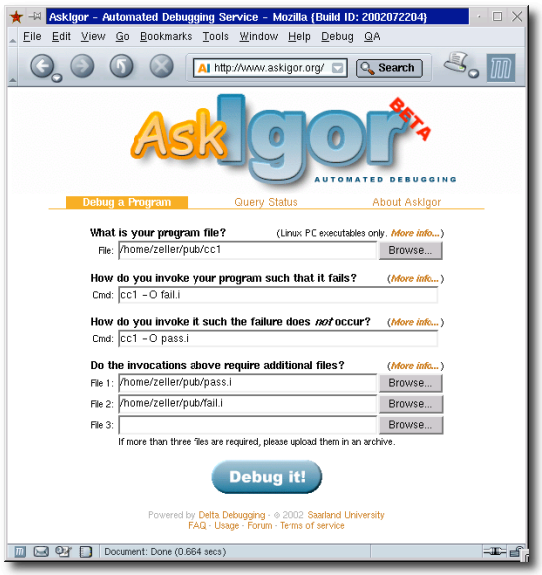
Failing Run

Passing Run

Delta Debugging



www.askigor.org

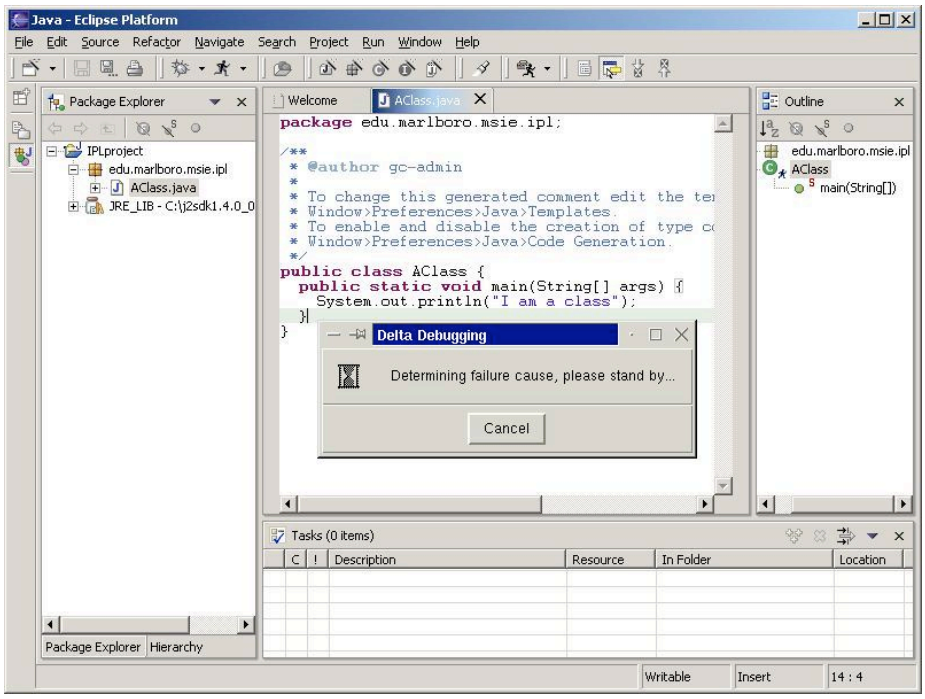


Programm einreichen
⇓
Aufrufe angeben
⇓
Auf „Debug it“ klicken
⇓
Diagnose kommt per e-mail

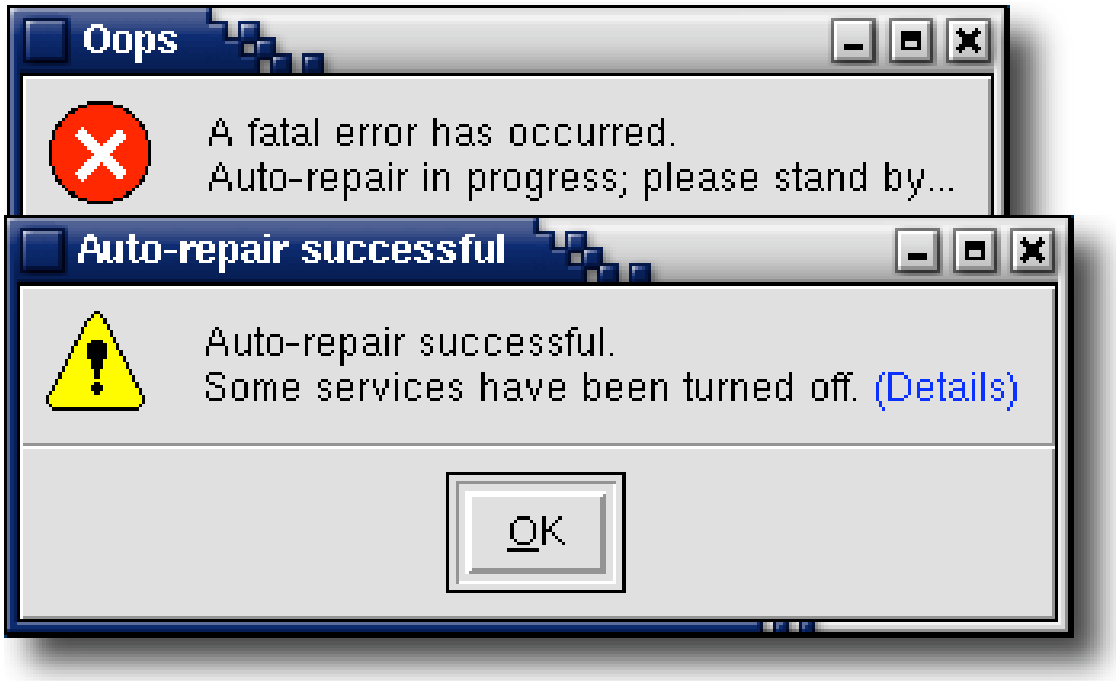
600 Einreichungen
seit Dezember 2003

56% „pinpoints the bug“
22% „helpful insights“

Delta Debugging Plug-Ins

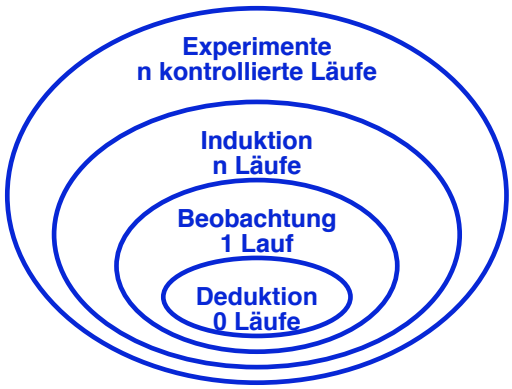


Selbstheilende Programme



Rückblick und Ausblick

Letzte 20 Jahre: *Deduktion* und *Observations-* Techniken



Nächste 20 Jahre: *Induktion* und *Experimentieren?*

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

Zusammenfassung (2)

- *Deduktion* schließt vom Code auf mögliche Läufe
- *Beobachtung* betrachtet einen konkreten Lauf
- *Induktion* schließt von konkreten Läufen auf Abstraktionen
- *Experimente* finden (Fehler-)ursachen

Mehr zum Thema „Programmanalyse und Programmverstehen“:
Vorlesung *Automated Debugging* (SS 2006)

Mehr zu aktuellen Forschungsthemen:
Vortrag “Dynamische Invarianten”, Do 13–14