Strukturorientierter Softwaretest
Erstellung eines Tools zur Überdeckungsmessung

Ronny Treyße & Hendrik Seffler

HU Berlin

Aufgabenstellung

’Überdeckungstesttool für C++ & Java’
Das umfasst:

- Überdeckung des Programmcodes durch Testfälle nach verschiedenen Kriterien übersprüfen
- Ableitung von Aussagen über Programmkomplexität
- Architekturinformation (Zsh. Testfall/Programmcode)
- Anwendung als Fallstudie auf das Subsystem ’manuelle Justage’ von XCTL

Gliederung

Der Vortrag ist folgendermaßen geliedert

- Was ATOS leistet, was es nicht leistet
- Ansätze des strukturorientierten Tests
- Formales Modell: Kontrollflussgraph
- Überdeckungsmaße
- Unsere Aufgabe
- Ein Ablaufbeispiel
- Offene Fragen

Schön ... aber wozu?

Testsystem ATOS ist funktionsorientiert

- Ableitung von Testfällen aus Spezifikation und Beschreibung des Programmes oder von Programmeinheiten
- *nicht* Herleitung von Testfällen aus Programmcode, Kenntnis der Programmcodes wird nicht genutzt
- Zweck des Tests: Überprüfen der Übereinstimmung mit der Spezifikation, nicht auf korrekte Programmfunktion ;-)

Schön ... aber wozu?

Einschränkungen dieses Ansatzes

- Wie genau ist die Spezifikation?
- Wurde Code aufgrund späterer Designentscheidungen geschrieben, die die Autoren der Spezifikation nicht berücksichtigen konnten?
- Funktionsorientierter Test testet auf Vorhandensein von Funktionen. Woher weiß man, ob es nicht unerwünschte Funktionen gibt?

Ziele von strukturorientiertem Test

Fragestellungen, die der strukturorientierte Test beantworten will

- Wird der gesamte Quellcode wirklich durchlaufen?
- Werden alle vorhandenen Kontrollstrukturen ausgenutzt?
- Welche Kriterien bewirken die Ausführung welches Codesegments?

Mögliche Herangehensweise: Formales Modell der Kontrollstruktur entwickeln....

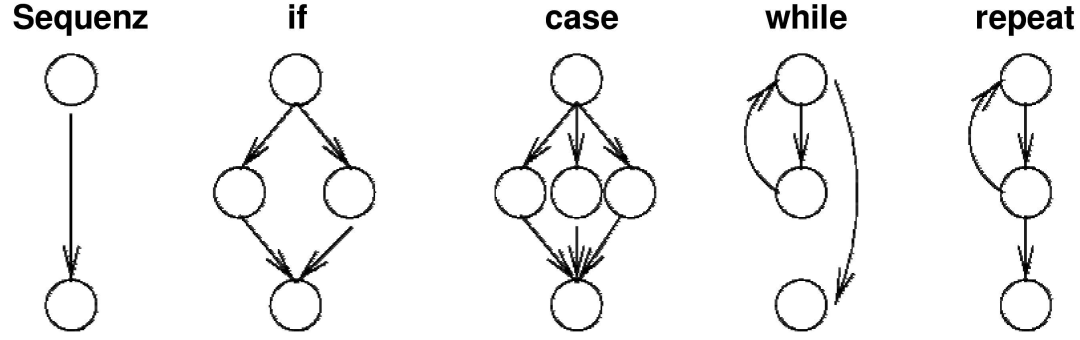
Kontrollflussgraphen

Gesucht ist eine Darstellung, die alle Anweisungen und alle Kontrollstrukturen abbildet:

Kontrollflussgraphen

Merkmale des Graphen:

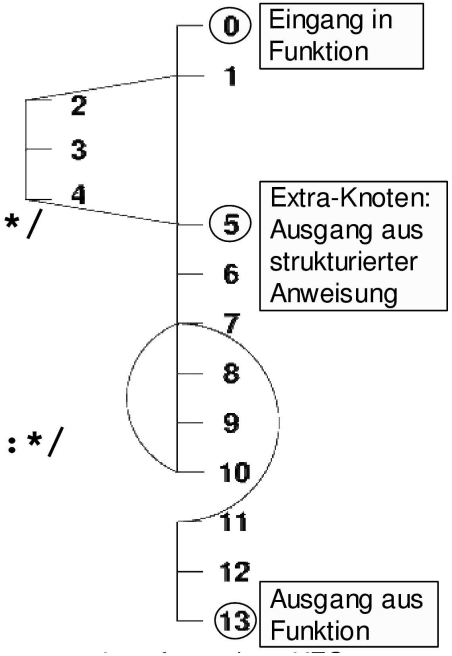
- Anweisungen dargestellt als Knoten
- Kontrollstrukturen als Kanten des Graphen



Kontrollflussgraphen - Ein Beispiel

Ein Beispiel eines Kontrollflussgraphen für den ’Euklidischen Algorithmus’ zur Ermittlung des ggT:

```
int euklid(int m, int n)
{ /* m,n > 0 */          /*0*/
  int r;
  if(n > m)              /*1*/
  {                      /*2,3,4:*/
     r = m; n = r; n = r;
  }                      /*5*/
  r = m % n;             /*6*/
  while(r != 0)          /*7*/
  {                      /*8,9,10:*/
     m = n; n = r; r = m % n;
  }                      /*11*/
  return r;              /*12*/
}                        /*13*/
```



Komplexitätsbestimmung

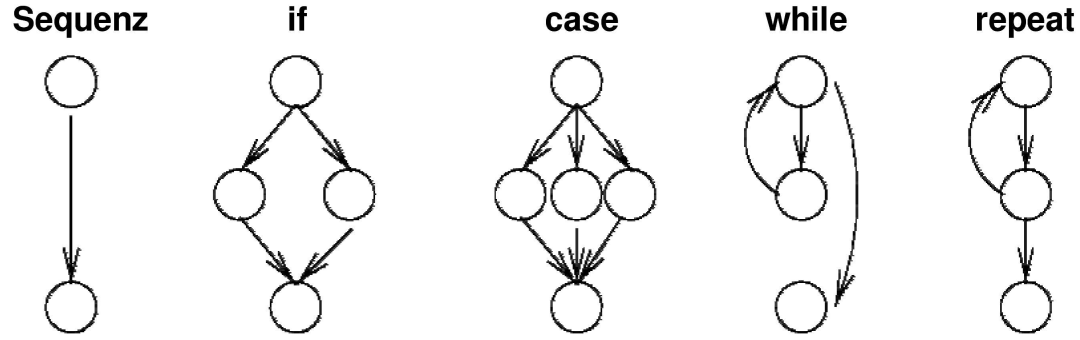
Aus dem Kontrollflussgraphen lassen sich einfach Maßzahlen zur Bestimmung der Programmkomplexität ableiten:

- zyklomatische Komplexität
  Zahl der möglichen Pfade durch den KFG
- essentielle Komplexität
  misst Unstrukturiertheit eines Programmes

Komplexitätbestimmung

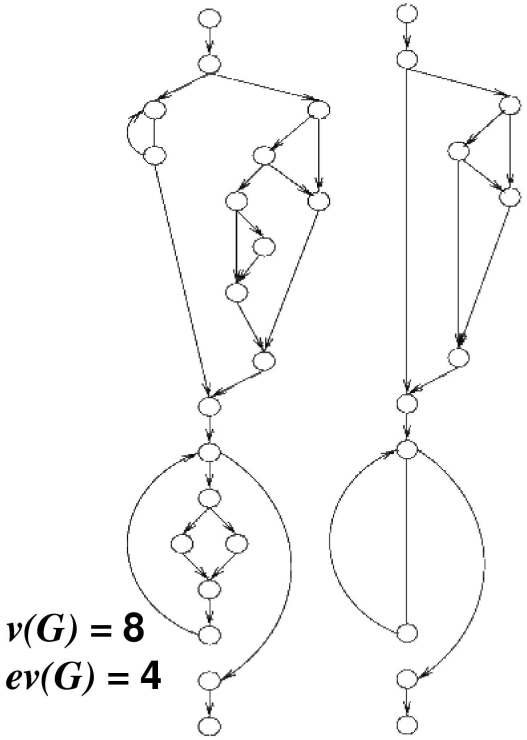
Bestimmen der essentiellen Komplexität:

entferne interativ Strukturen der strukturierten Programmierung aus dem ursprünglichen Kontrollflussgraphen und messe zyklomatische Komplexität des neuen Graphen

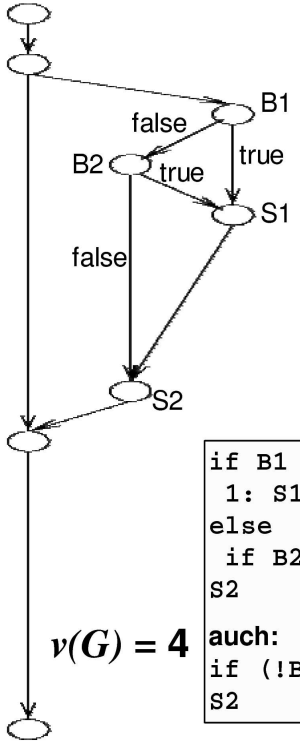


Komplexitätbestimmung

Schau ’mer mal:



$v(G) = 8$
$ev(G) = 4$



$v(G) = 4$

```
if B1 then
 1: S1
else
 if B2 then goto 1;
S2
```

auch:

```
if (!B1 || !B2) S1;
S2
```

Überdeckungsmaße

Stellt eine Menge von Testfällen sicher, daß ... ?

- Anweisungsüberdeckung (C0)
  ...alle Anweisungen durchlaufen werden, also jeder Knoten des KFG wenigstens einmal durchlaufen wird
- Zweigüberdeckung (C1)
  ...alle möglichen Verzweigungen genutzt werden, also alle Kanten des KFG durchlaufen werden
- Boundary-Interior
  .. jeder Zykluskörper wenigstens einmal wiederholt wird

Überdeckungsmaße

Stellt eine Menge von Testfällen sicher, daß ... ?

- einfache Bedingungsüberdeckung (C2)
  ...alle atomaren Bedingungen einmal mit *true* und mit *false* belegt werden
- mehrfache Bedingungsüberdeckung (C3)
  ...alle möglichen Kombination von atomaren Bedingungen auftreten

Überdeckungsmaße

Problem: mehrfache Bedingungsüberdeckung erfordert 2^n Testfälle um eine Bedingung aus n atomaren Bedingungen zu überprüfen

- mehrfache, minimale Bedingungsüberdeckung
  alle Bedingungen (atomare und zusammengesetzte) werten einmal zu *true* und einmal zu *false* aus
- Modified Condition/Decision Coverage (MC/DC)
  Auswahl der Testfälle so, daß für jede atomare Bedingung gilt, daß sie einmal unabhängig das Ergebnis bestimmt hat
  (Zu *(A or B)* liefern etwa die Wahrheitwerte *(FT)*, *(TF)* und *(FF)* eine MC/DC-Überdeckung)

Wie ermitteln: Instrumentierung

Wie können diese Kriterien überwacht und geprüft werden?

$\rightsquigarrow$ Einfügen von Code in den Quelltext, der Programmablauf

und Bedingungsbelegung überwacht und aufzeichnet.

Instrumentierung am Beispiel

Beispiel für Zweigüberdeckung:

```
coverage.whileEntry++;
while(a) {
    coverage.interior++;
    if (a) {
        coverage.ifMain++;
        do.something();
    }
    else {
        coverage.ifElse++;
        do.somethingElse();
    }
}
coverage.whileExit++;;
```

Rückblick

Wir haben wesentliche Hilfsmittel zum strukturorientierten Test kennengelernt:

- Kontrollflussgraphen
- Komplexitätmessung
- Überdeckungsmaße
- Quellcodeinstrumentierung

Wie kann man dies praktisch umsetzen, welche Fragen entstehen?

Unsere Arbeit

Unsere weitere Arbeit zerfällt in zwei Teile

- Pflichtenheft
  Studienarbeit
- Dokumentation, Implementierung etc.
  Diplomarbeit

Für den Entwurf den Pflichtenheftes müssen wir grundlegende Designentscheidungen treffen

Zweiteilige Programmstruktur

Die Aufgabenstellung legt folgende Struktur nahe:

- Prä-Compiler
  Ein Prä-Compiler übernimmt die Instrumentierung des Quellcodes und fügt Code ein, der die Ergebnisse in eine Log-Datei ausschreibt
- Auswertung (GUI)
  Ein Programmmodul wertet diese Log-Datei aus und stellt die Ergebnisse anschaulich dar

Wie kann das in der Benutzung aussehen?

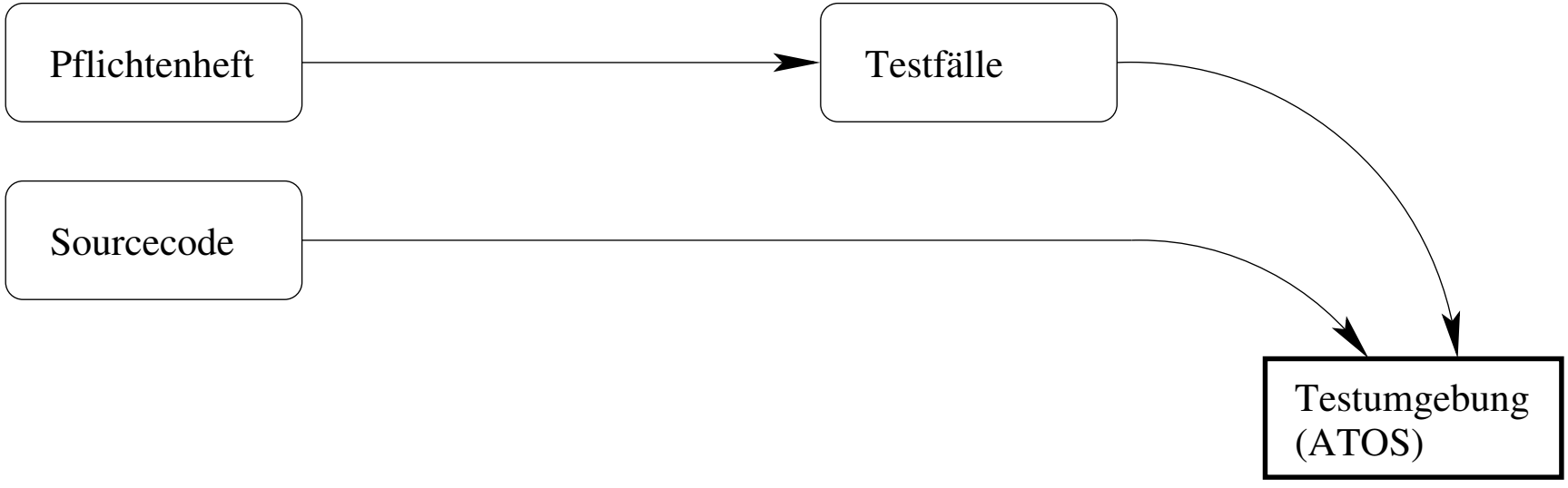
Allgemeiner Anwendungsfall

- Ziel:
  - Informationen/Maße zum Sourcecodes
  - Bewertung und Verbesserung der Testfallmenge
- Phasen:
  - Instrumentieren
  - Testen
  - Auswerten

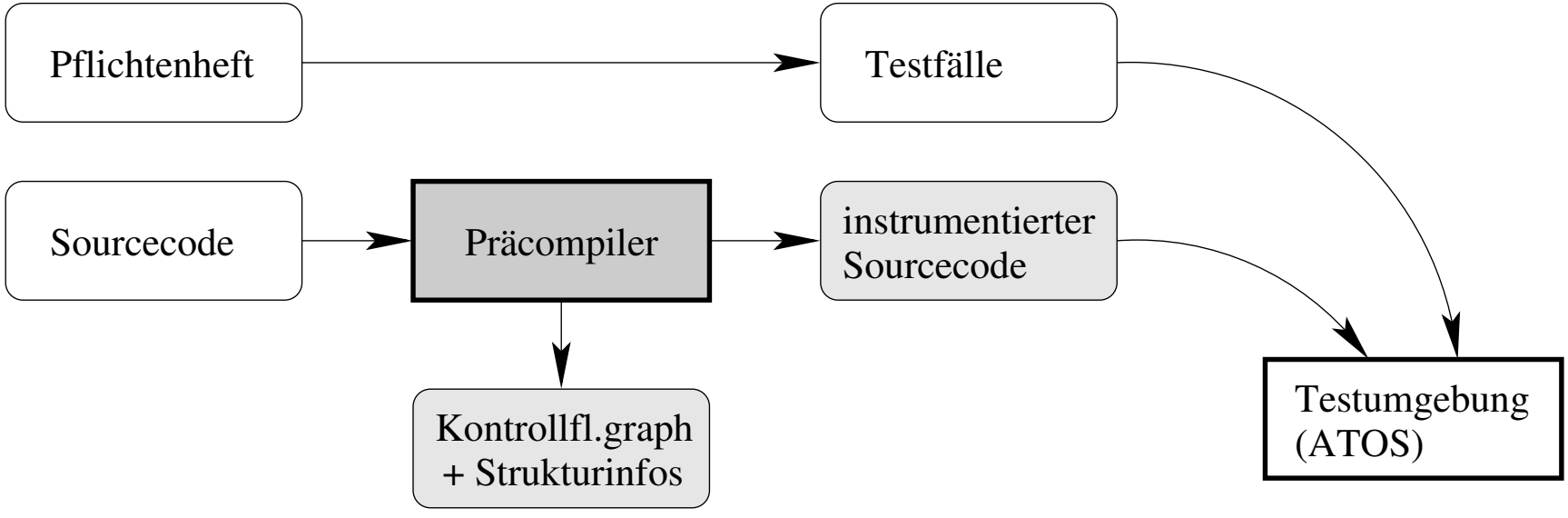
Phase I – Instrumentierung

- Ausgangsbasis:
  - kompilierbarer Sourcecode
  - Präcompiler
- vor Erstellen des Objektcodes wird der Sourcecode durch den Präcompiler geparst und instrumentiert
- Ergebnis:
  - instrumentierter Sourcecode
  - Kontrollflussgraph
  - Strukturinformationen (statische Maße)

Überblick – Testsystem (1)



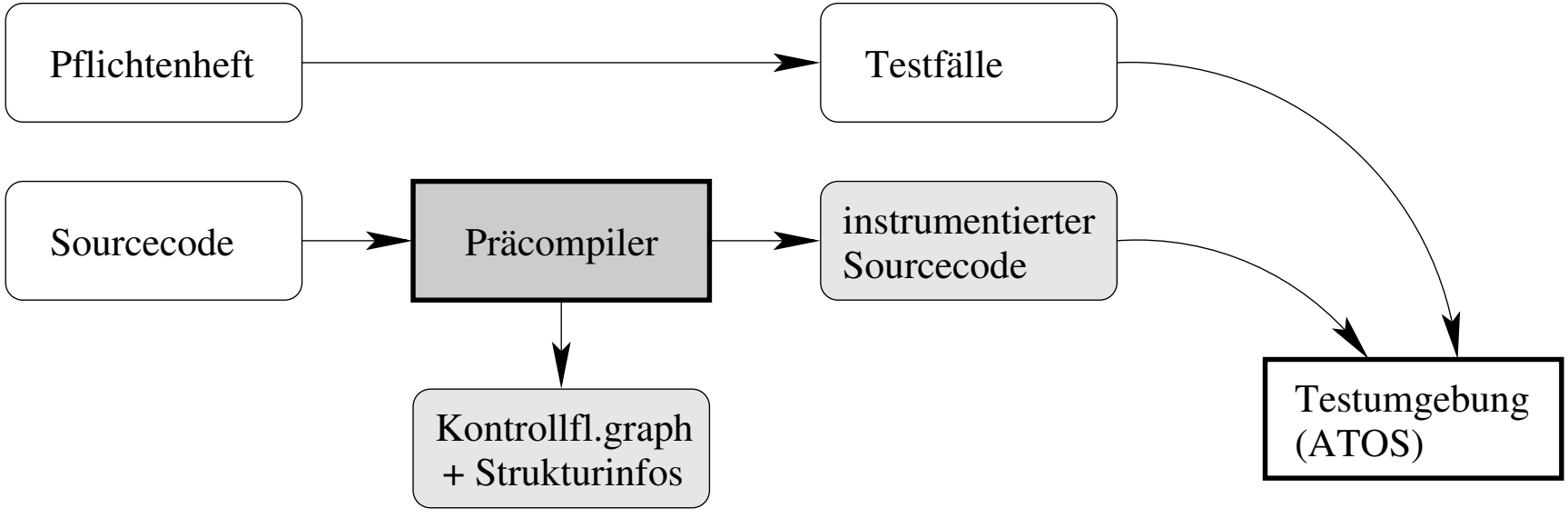
Überblick – Testsystem (2)



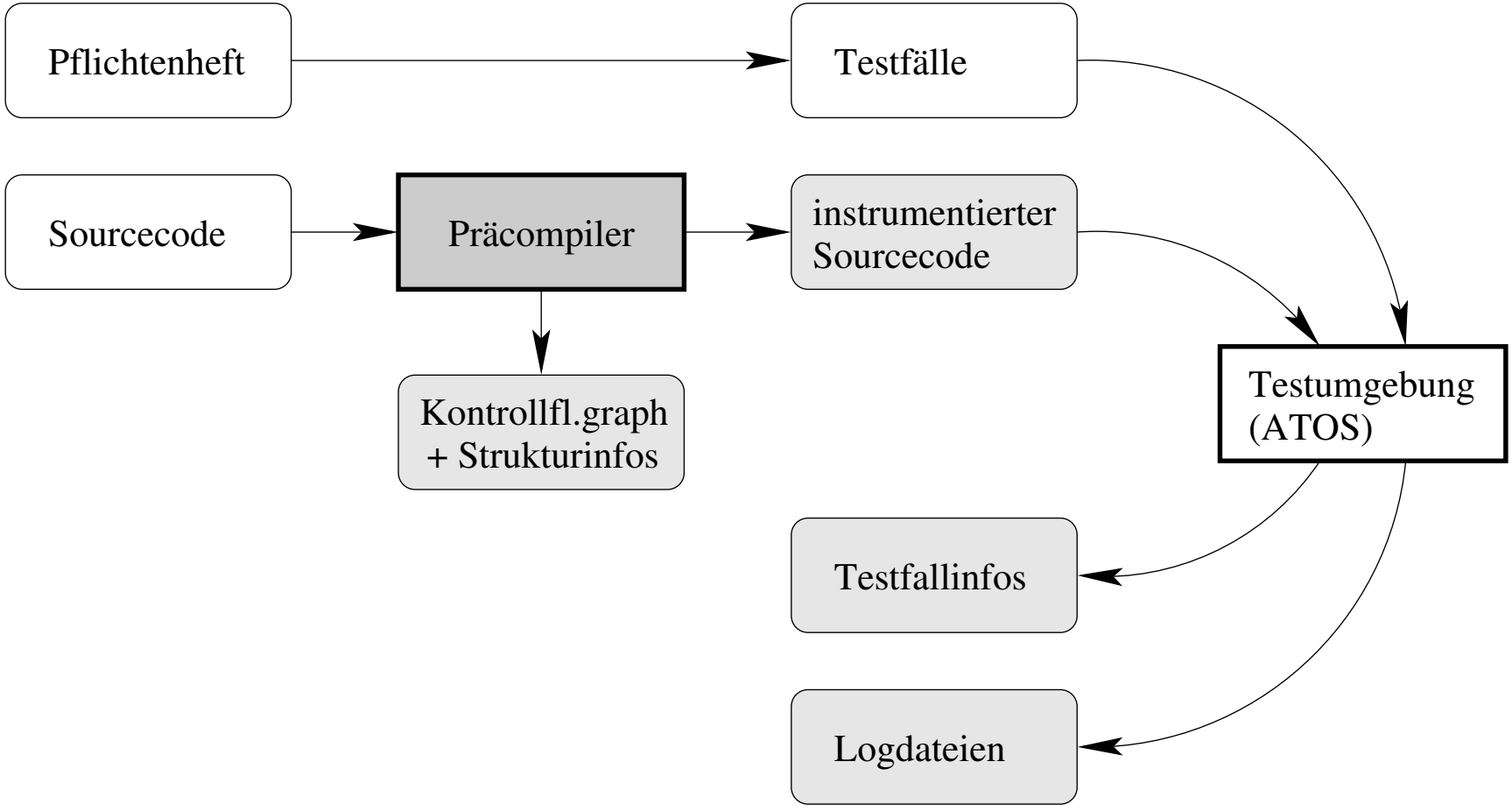
Phase II – Testen

- kompilierte Programm wird ausgeführt bzw. in Testumgebung automatisch getestet (ATOS)
- Forderung: keine Beeinflussung der Programmabarbeitung/Testumgebung
- aber notwendig:
  - Sammeln von Informationen
  - Ausschrift in Logdateien
- Ergebnis:
  - Logdateien mit Informationen zum Programmfluss
  - Informationen zu den Testfällen (woher?)

Überblick – Testsystem (2)



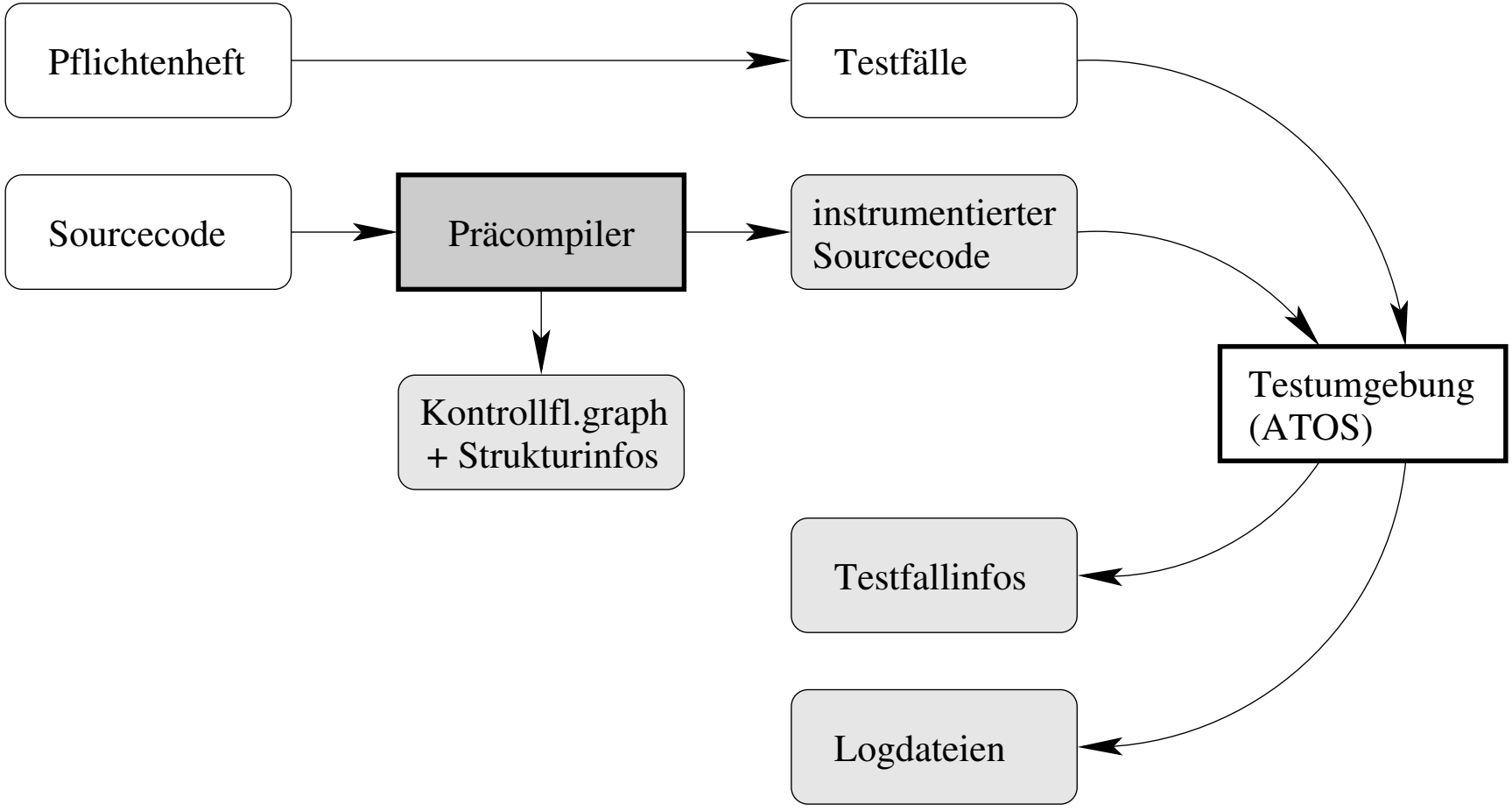
Überblick – Testsystem (3)



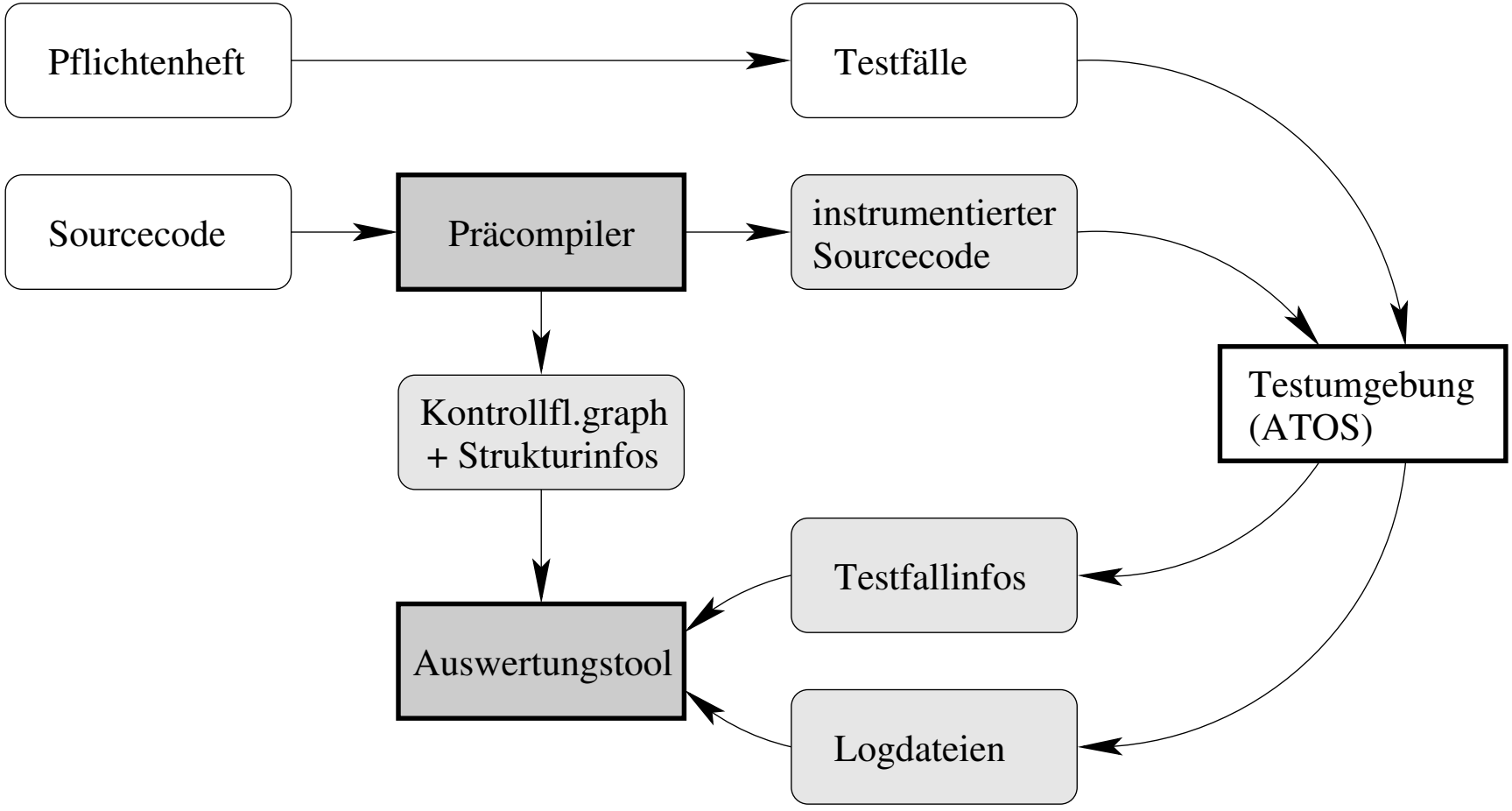
Phase III – Auswertung

- Ausgangsbasis:
  - Präcompiler: instrumentierter Sourcecode, KFG
  - Testsystem: Logdateien, Testinformationen
- graphisches Werkzeug wertet diese Daten aus und bietet übersichtlichen Zugang zu der Auswertung
- Zusammenfassung von Testfällen und Codesegmenten/Modulen zu Testeinheiten
- Bereistellen von Informationen zu Sourcecode und Testfallmengen/-einheiten

Überblick – Testsystem (3)



Überblick – Testsystem (4)

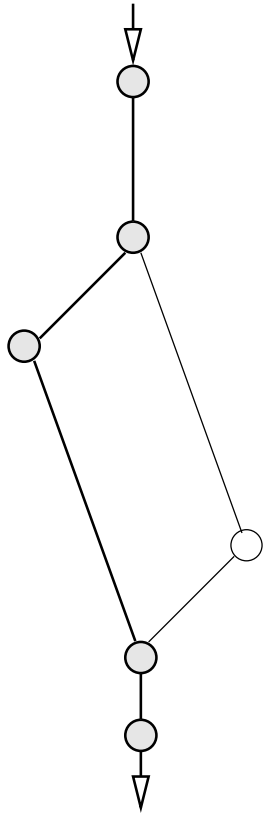


Informationen des Auswertungstool

- statische Maße:
  - zyklomatische + essentielle Komplexität
  - weitere Maße (LOC, ...) ?
- Überdeckungsmaße:
  - C0, C1, boundary-interior-Pfadtest
  - C2, min. mehrfache Bedingungsüberdeckung, MC/DC, C3
- Beziehungen:
  - was berührt ein(e) Testfall(-menge)?
  - welcher Test berührt ein Codesegment/Funktion?

GUI-Elemente des Auswertungstools

Testfälle	Klassenstruktur	Sourcecode	Kontrollflussgraph

Test 1
Test 2
Test 3
Test 4
	glob. Funktionen
	main()
	foo()
	Klassen
	class A
	foo_1()
	foo_2()
	class B
	foo_1()
	...
		`int main() {`
		`...`
		`if(a==1) {`
		`a.foo_2();`
		`}`
		`else {`
		`a.foo_1();`
		`}`
		`b.foo_2();`
		`}`
			

Fragen (1)

- aktuelle Fragen:
  - wie mächtig muss unser Parser sein?
  - wie genau muss der Sourcecode geparst werden?
  - welche möglichen Tools und Bibliotheken gibt es, die Aufgabenbereiche abdecken können?
- zur Instrumentierung:
  - welche Forderungen an Ressourcennutzungen der Instrumentierung? (Speicher+Zeit)
  - welche Beschränkung der Namensräume?

Fragen (2)

- zur Testumgebung/ATOS:
  - welche Informationen zu den Testfällen kann ATOS wie liefern?
  - wie ist Zuordnung einzelner Testfälle zu Programmdurchläufen möglich?
  - wie lässt sich Logmanagment unter ATOS realisieren?
  - allgemein: welche Schnittstelle zu ATOS?
- zur Auswertung:
  - zusätzliche Soll/Wunschkriterien?
  - überflüssige Informationen?
  - Besonderheiten für Einsatz in der Lehre?