Universität Zürich[UZH]

Institut für Informatik

Martin Glinz Thomas Fritz

Software Engineering

Kapitel 11

Statische Analyse

11.1 Grundlagen

11.2 Formen der statischen Analyse

11.3 Programmanalyse

11.4 Abhängigkeitsgraphen

Grundlagen

Statische Analyse ist die Untersuchung des statischen Aufbaus eines Prüflings auf die Erfüllung vorgegebener Kriterien.

- Keine Programmausführung (Abgrenzung zum Testen)
- Formal und automatisierbar (Abgrenzung zum Review)

Ziele:

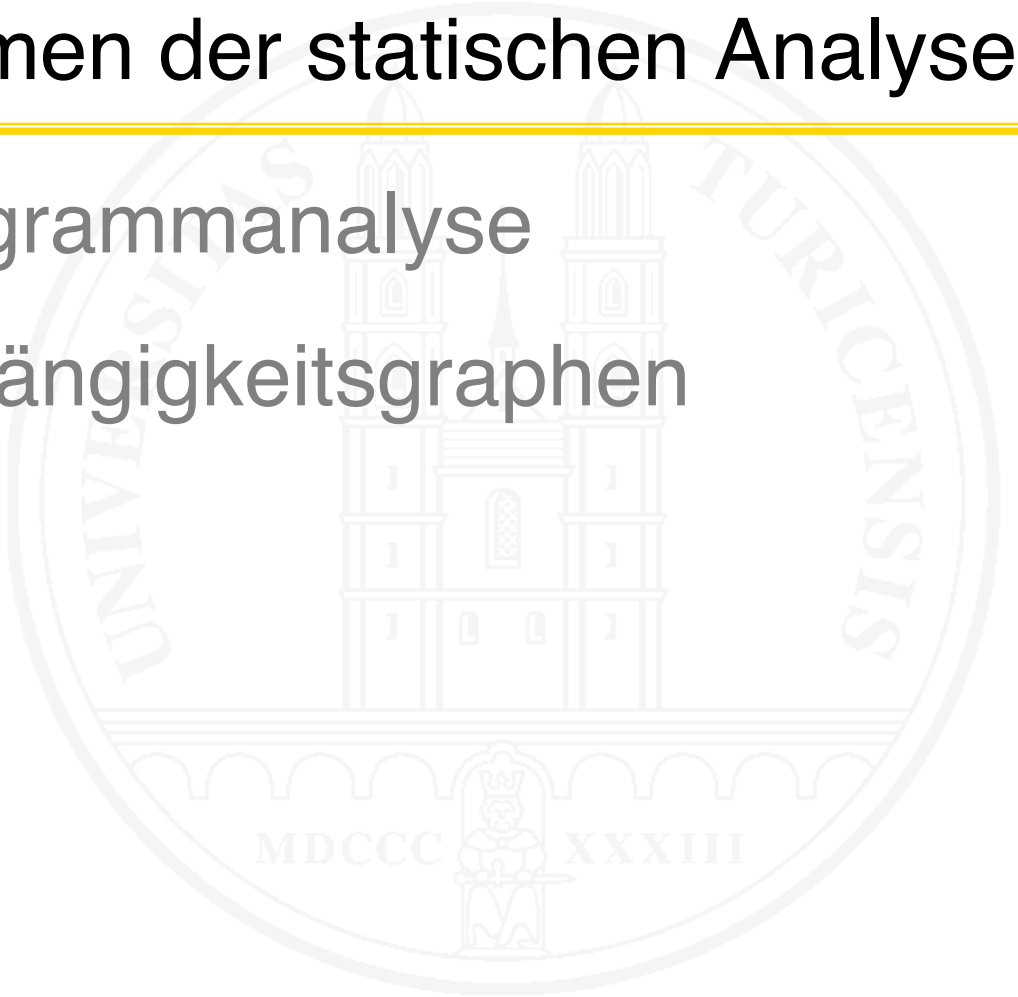
- Bewertung von Qualitätsmerkmalen, z.B. Pflegbarkeit, Testbarkeit aufgrund struktureller Eigenschaften des Prüflings
- Erkennung von Fehlern und Anomalien in neu erstellter Software
- Erkennung und Analyse von Programmstrukturen bei der Pflege und beim Reengineering von Altsystemen (*legacy systems*)
- Prüfen, ob ein Programm formale Vorgaben einhält (Codierrichtlinien, Namenskonventionen, etc.)

11.1 Grundlagen

11.2 Formen der statischen Analyse

11.3 Programmanalyse

11.4 Abhängigkeitsgraphen



Syntaktische Analyse von Programmen

Analyse in der Regel bei der Programmübersetzung durch Compiler

- Syntaktische Korrektheit des Quellcode
- Datentypen, Typverträglichkeit
  - Typ-unverträgliche Operationen
  - Typ-unverträgliche Zuweisungen
  - Typ-unverträgliche Operationsaufrufe

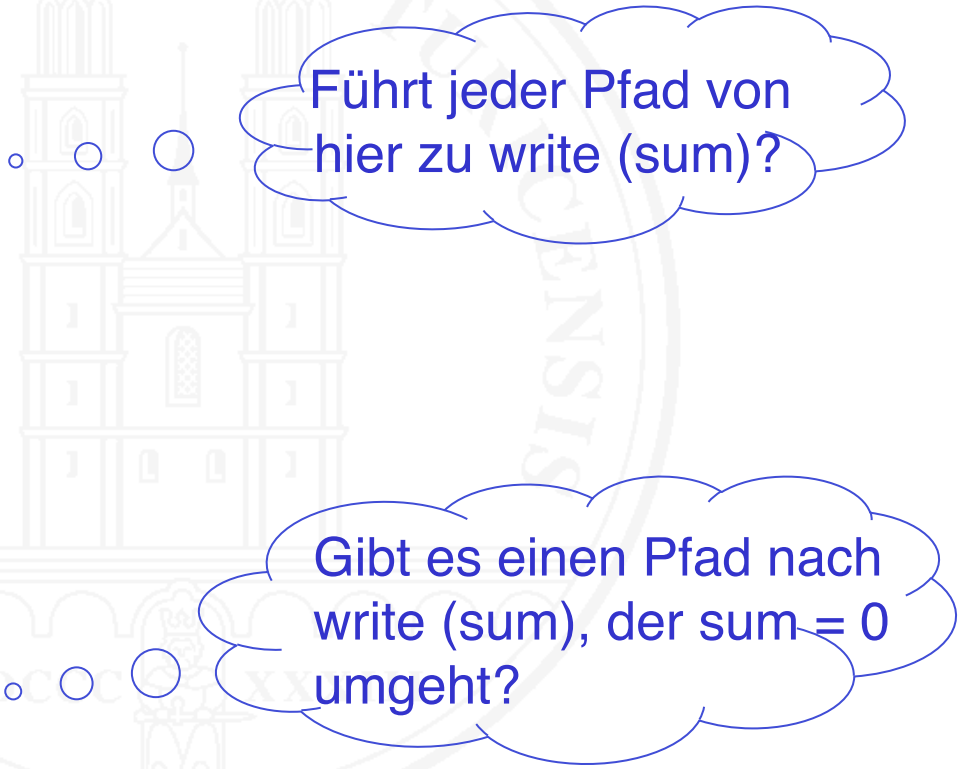
Strukturanalyse von Programmen

- Formale Eigenschaften
  - Richtlinien eingehalten
  - Art und Menge der internen Dokumentation
- Strukturkomplexität des Programms
  - Einhaltung von Strukturierungsregeln
  - Art und Tiefe von Verschachtelungen
- Strukturfehler bzw. -anomalien
  - nicht erreichbarer Code
  - vorhandene, aber nicht benutzte Operationen
  - benutzte, aber nicht vorhandene Operationen
- Fehler / Anomalien in den Daten
  - nicht deklarierte Variablen
  - nicht benutzte Variablen
  - Benutzung nicht initialisierter Variablen

Kontrollflussanalyse in Programmen

Wege im Flussgraphen

```
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);
}
```



Datenflussanalyse in Programmen

Zusammenhänge zwischen Definition und Verwendung von Daten

```
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 wird durch diese Anweisung beeinflusst?

Wie entsteht dieser Wert?

Statische Analyse anderer Software-Artefakte

- Analyse von Algorithmen
  - Laufzeiteffizienz
  - Speichereffizienz
  - Gültigkeitsbereich
  - Erfolgt weitestgehend manuell

- Analyse von Spezifikationen, Entwürfen, Prüfvorschriften
  - Syntaxanalyse der formal beschriebenen Teile
  - Teilweise Struktur- und Flussanalysen (je nach verwendeter Modellierungsmethode)
  - Analyse der Anforderungsverfolgung, wenn dokumentiert ist, welche Anforderung wo umgesetzt bzw. geprüft wird (werkzeuggestützt möglich)

Vorgehen

- Bei Programmen immer mit Werkzeugen
  - Compiler
  - syntaxgestützte Editoren
  - spezielle Analyse-Werkzeuge
- Programm wird wie bei der Übersetzung durch ein Werkzeug zerlegt (parsing) und in einer analysefreundlichen internen Struktur repräsentiert
- Dabei werden Syntaxfehler erkannt und erste Kenngrößen (z. B. Anzahl Codezeilen, Anzahl Kommentare) ermittelt
- Die resultierende interne Repräsentation des Programms wird verschiedenen Analysen unterworfen (z.B. statische Aufrufhierarchie, Strukturkomplexität, Datenflussanalyse)
- Die Befunde werden gesammelt, ggf. verdichtet und bewertet

11.1 Grundlagen

11.2 Formen der statischen Analyse

11.3 Programmanalyse

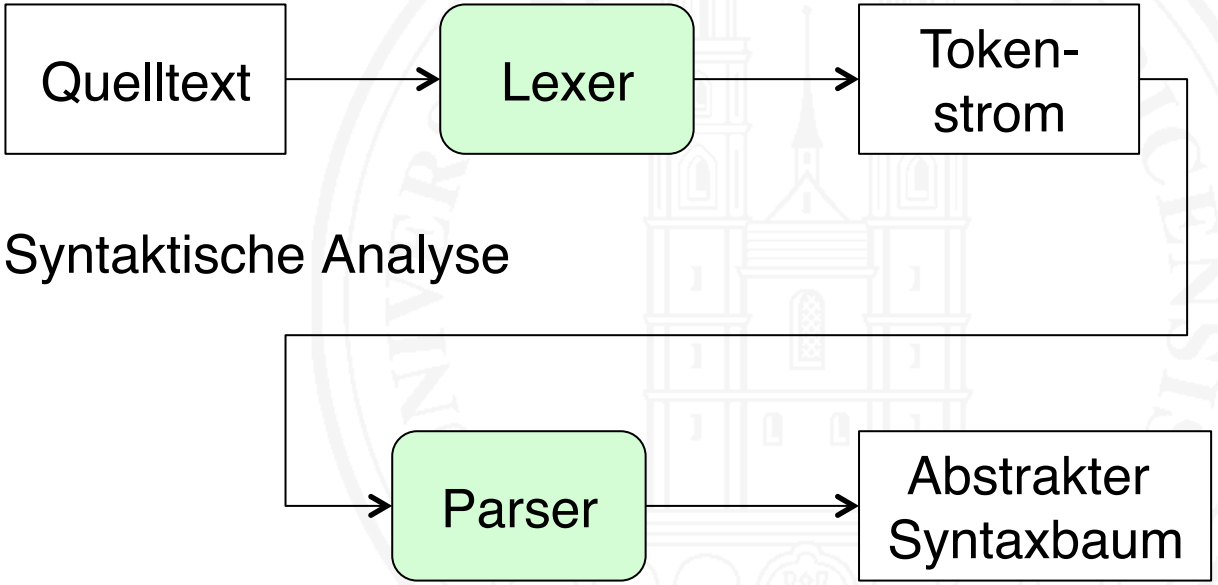
11.4 Abhängigkeitsgraphen

Grundlagen

- Grundlegende Analysen
  - Kontrollflussanalyse (control flow analysis, Steuerflussanalyse)
  - Datenflussanalyse (data flow analysis)
- Basis für alle weitergehenden Analysen
- Analyse erfolgt nicht auf dem Quelltext, sondern dem abstrakten Syntaxbaum

Vom Quelltext zum abstrakten Syntaxbaum

- Lexikalische Analyse

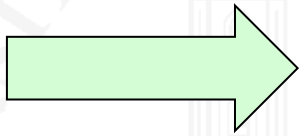


- Syntaktische Analyse

Der Lexer

Quelltext:

```
declare
  X : real;
begin
  X := A * 5;
end
```



Tokenstrom:

```
9 -- “declare”
4 -- id,   “X”
7 -- “:”
4 -- id,   “real”
5 -- “;”
6 -- “begin”
4 -- id,   “X”
3 -- “:=“
4 -- id,   “A”
8 -- “*”
10 -- int_lit,   “5”
5 -- “;”
11 -- “end”
```

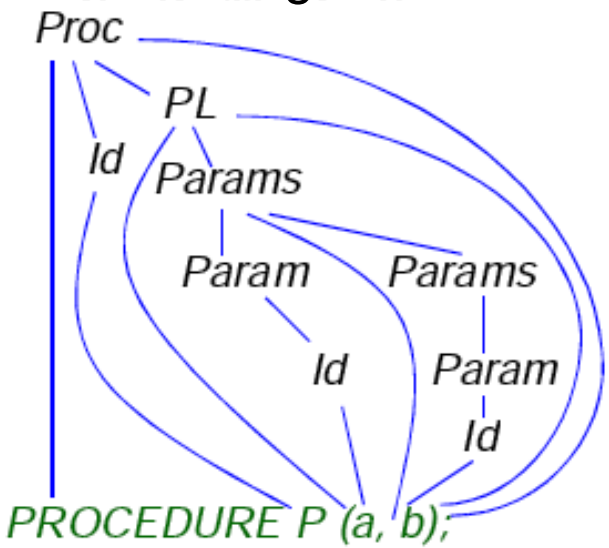
Der Parser

Analyse des Tokenstroms nach den Regeln der Grammatik der verwendeten Programmiersprache

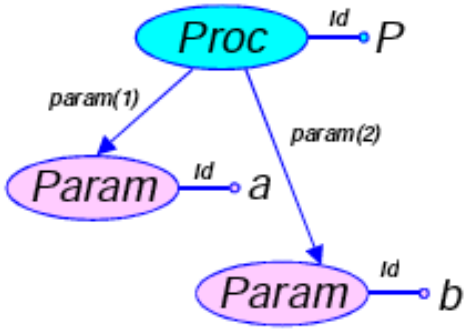
Grammatik:

```
Proc ::= PROCEDURE Id PL ";" .
PL ::= "(" Params ")" | e .
Params ::= Param "," Params | Param .
Param ::= Id .
```

Ableitungsbaum

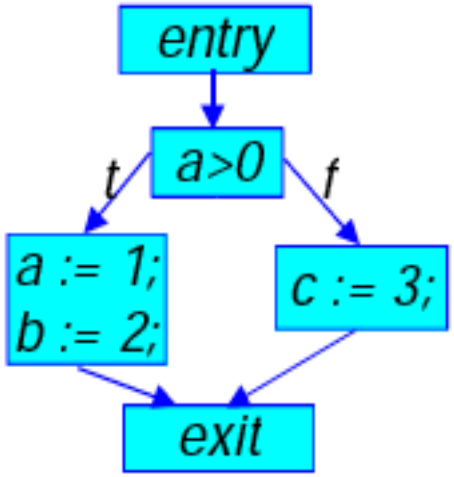


Abstrakter Syntaxbaum

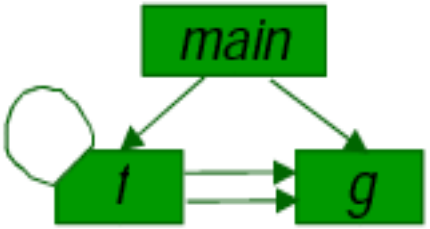


Der Kontrollfluss (control flow, Steuerfluss)

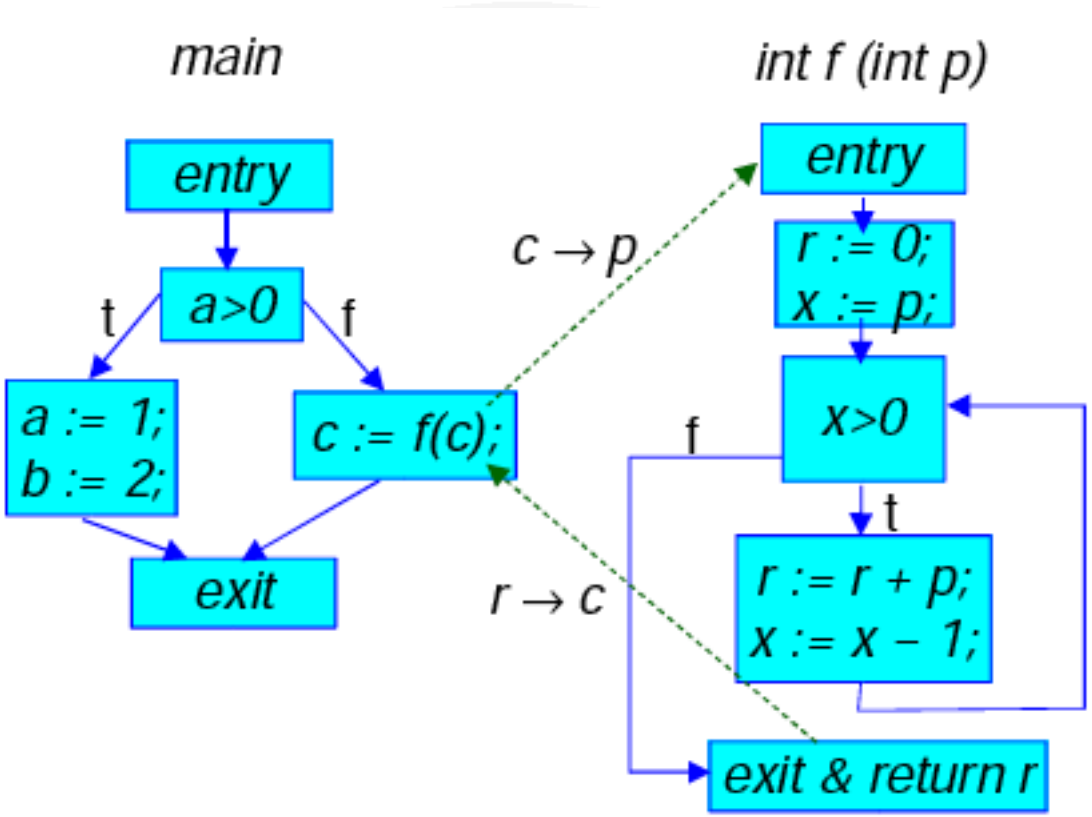
- Intra-prozedural: Flussgraph
  - Knoten: Grundblöcke
  - Kanten: (bedingter/unbedingter Kontrollfluss
  - ergibt sich aus Abläufen, Verzweigungen und Schleifen



- Inter-prozedural: Aufrufgraph
  - Multigraph
  - Knoten: Prozeduren
  - Kanten: Aufruf
  - ergibt sich aus expliziten Aufrufen im Programmcode sowie indirekten Aufrufen (beispielsweise über Funktionszeiger in C)



Intra- und interprozeduraler Kontrollfluss

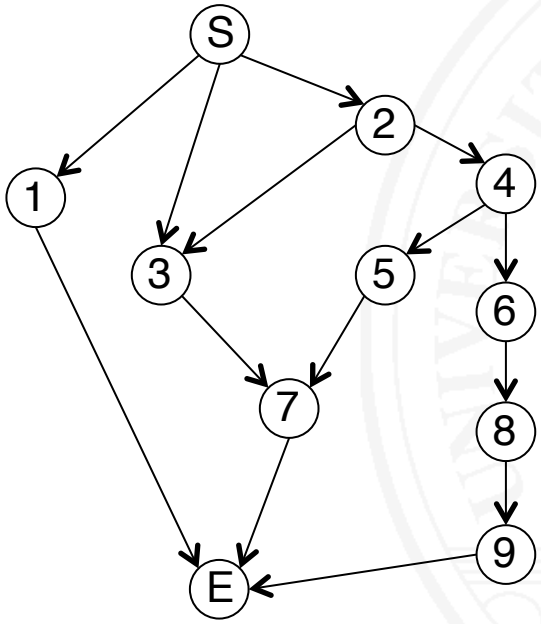


Dominanz

- Fragestellungen
  - Gibt es eine Prozedur D, über nur die ein Aufruf von N erfolgt?
  - Welcher Block D im Flussgraph muss in jedem Falle passiert werden, damit Block N ausgeführt werden kann?
- Antwort: D ist der Dominator von N
  - Ein Knoten D dominiert einen Knoten N, wenn D auf allen Pfaden vom Startknoten zu N liegt.
  - Ein Knoten D ist der direkte Dominator von N, wenn
    - D dominiert N und
    - alle weiteren Dominatoren von N dominieren D

Dominanz – 2

Flussgraph

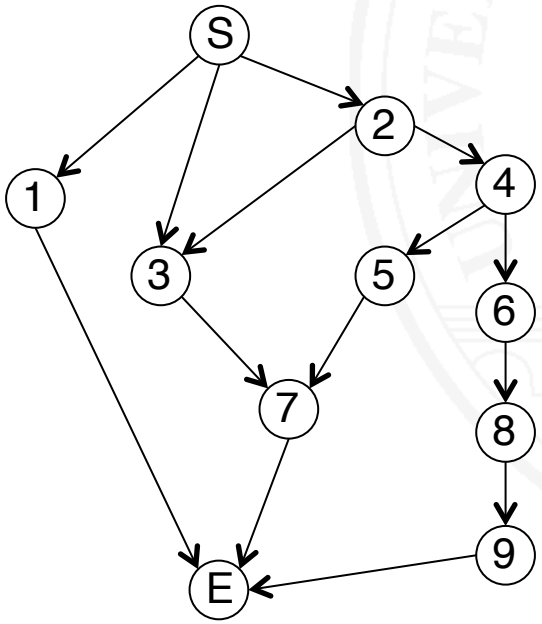


- Knoten 2 dominiert 4, 5, 6, 8, 9
- Knoten 2 dominiert nicht S, 1, 3, 7, E
- Knoten 6 ist direkter Dominator von 8
- Knoten S, 2, 4 dominieren 8, sind aber keine direkten Dominatoren von 8
- Knoten S dominiert alle Knoten des Flussgraphen

Postdominanz

Ein Knoten P postdominiert einen Knoten N, wenn jeder Pfad von N zum Endknoten den Knoten P enthält (entspricht Dominanz des umgekehrten Graphen).

Flussgraph

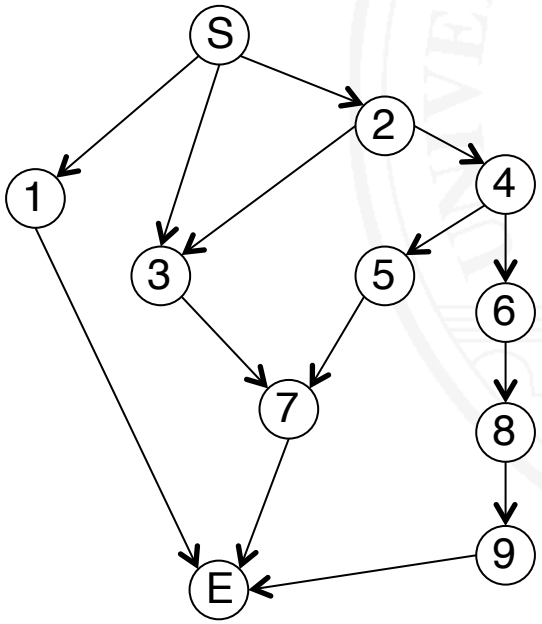


- Knoten 7 postdominiert 3, 5, aber nicht 2, 4, S
- Knoten 9 postdominiert 6, 8 aber nicht 4
- Exit postdominiert alle Knoten

Kontrollabhängigkeit

Ein Knoten X ist kontrollabhängig von einem Knoten B genau dann, wenn

- Es existiert ein nicht-leerer Pfad von B nach X, so dass X jeden Knoten auf diesem Pfad postdominiert (mit Ausnahme von B selbst)
- X postdominiert B nicht oder X=B



Die Knoten 6, 8, 9 sind kontrollabhängig von 4, aber nicht von 2 und S

Kontrollabhängigkeit – 2

- Für strukturierte Programme gilt:
  - eine Anweisung ist kontrollabhängig von der Bedingung der nächstumgebenden Schleife oder Alternative

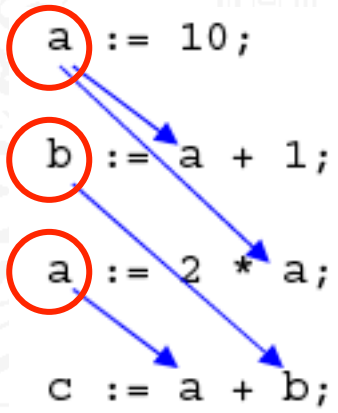
```
while a loop
   if b then
      x := y;  --- kontrollabhängig von b
   end if;
   z := 1;  --- kontrollabhängig von a
end loop;
```


Datenabhängigkeitsanalyse

- Set = Setzen eines Werts
- Use = Verwendung eines Werts
- Daraus ergeben sich folgende relevanten Beziehungen zwischen zwei Anweisungen (resp. Knoten) A und B:
  - Set-Use Beziehung (Datenabhängigkeit)
    - A setzt den Wert, der von B verwendet wird
  - Use-Set Beziehung (Anti-Dependency)
    - A liest den Wert und B überschreibt ihn danach
  - Set-Set Beziehung (Output-Dependency)
    - der von A gesetzte Wert wird von B überschrieben

Datenabhängigkeit (Set-Use)

Für die Zwecke der Analyse ist die Set-Use Beziehung die wichtigste Beziehung



Zwischen der 2. und 3. Anweisung besteht eine “Use-Set”, zwischen der 1. und 3. Anweisung eine “Set-Set”-Beziehung.

11.1 Grundlagen

11.2 Formen der statischen Analyse

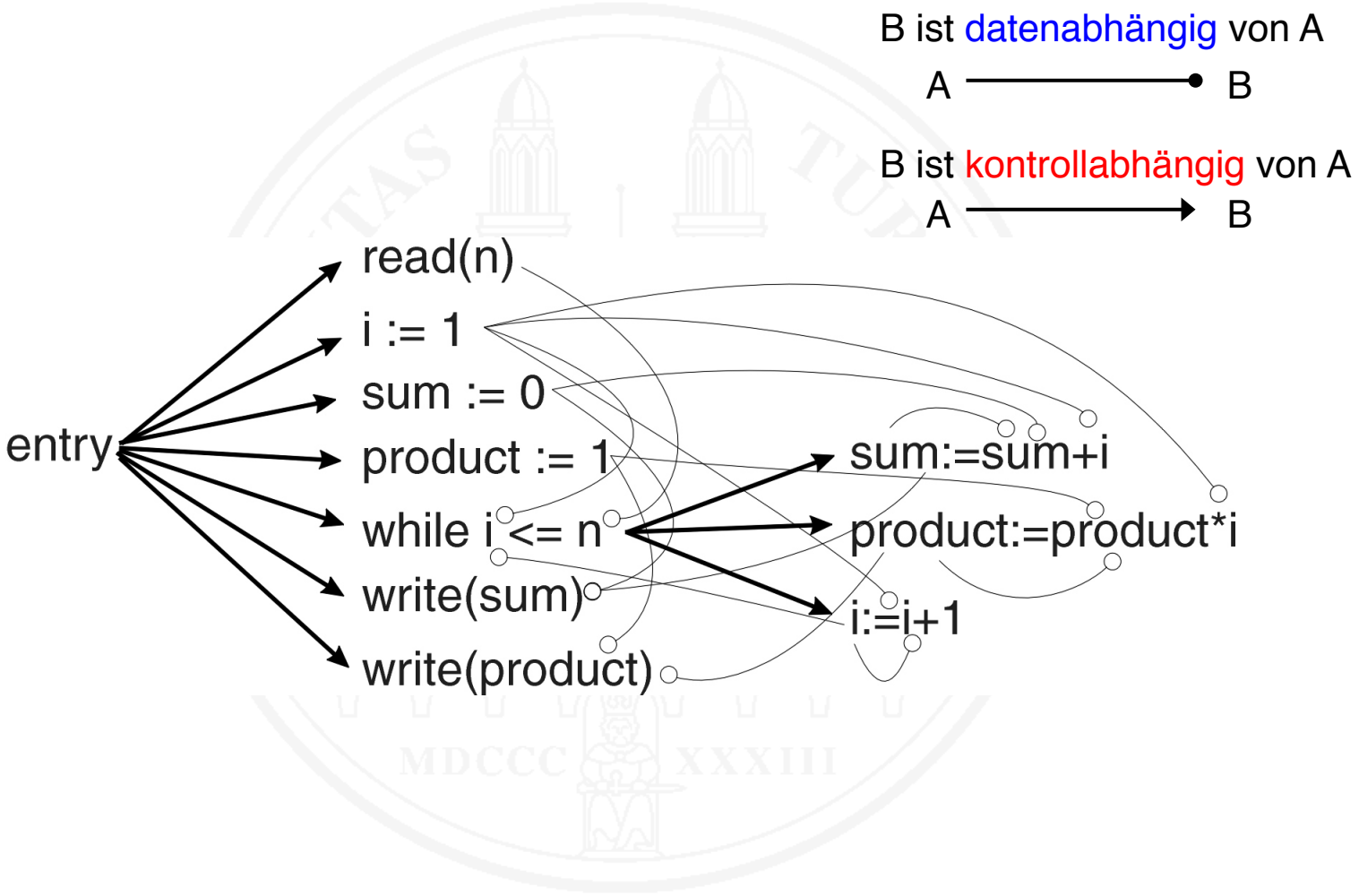
11.3 Programmanalyse

11.4 Abhängigkeitsgraphen

Program Dependency Graph (PDG)

- PDG = gerichteter Multigraph für Daten- und Kontrollflussabhängigkeiten
  - Knoten repräsentieren Zuweisungen und Prädikate
    - zusätzlich einen speziellen Entry-Knoten
    - zusätzlich Ø-Knoten zur Reduktion der Datenabhängigkeitskanten
    - (sowie einen Initial-Definition-Knoten für jede Variable, die benutzt werden kann, bevor sie gesetzt wird)
  - Kontroll-Kanten repräsentieren Kontrollabhängigkeiten
    - Startknoten jeder Kontrollabhängigkeitskante ist entweder der Entry-Knoten oder ein Prädikatsknoten
  - Fluss-Kanten repräsentieren Set-Use-Abhängigkeiten

Beispiel PDG



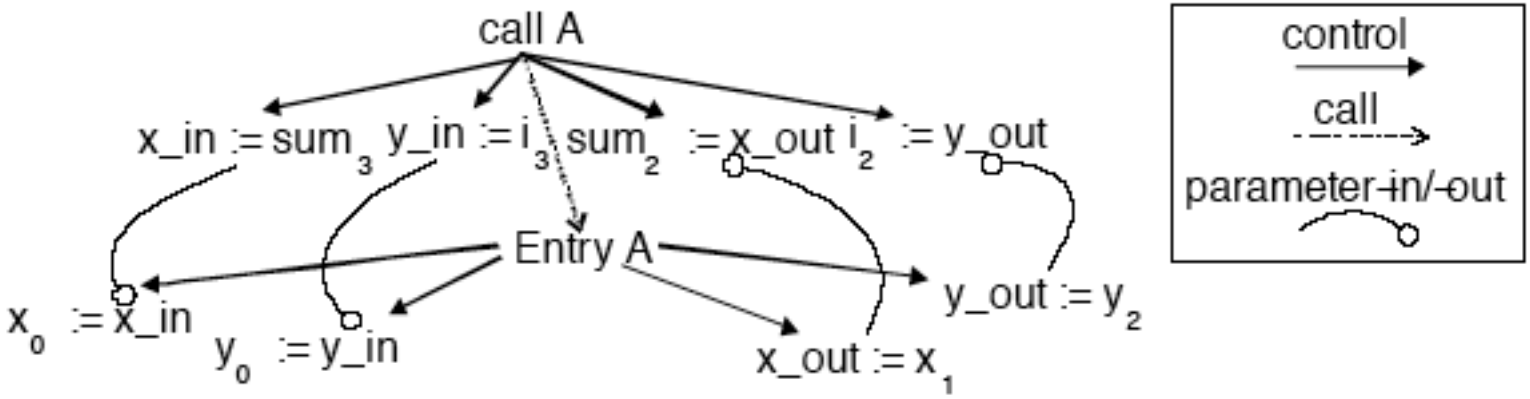
SDG und PDG

- PDG stellt Abhängigkeiten innerhalb einer Funktion dar
- System Dependency Graph (SDG) stellt globale Abhängigkeiten dar: PDGs für verschiedene Unterprogramme werden vernetzt über interprozedurale Kontroll- und Datenflusskanten
  - Aufruf: Call-Knoten
  - aktuelle Parameter: actual-in / actual-out-Knoten (copy-in/copy-out-Parameterübergabe vorausgesetzt)
  - formale Parameter: formal-in / formal-out-Knoten
  - transitive Abhängigkeiten via PDG: Summary-Edges

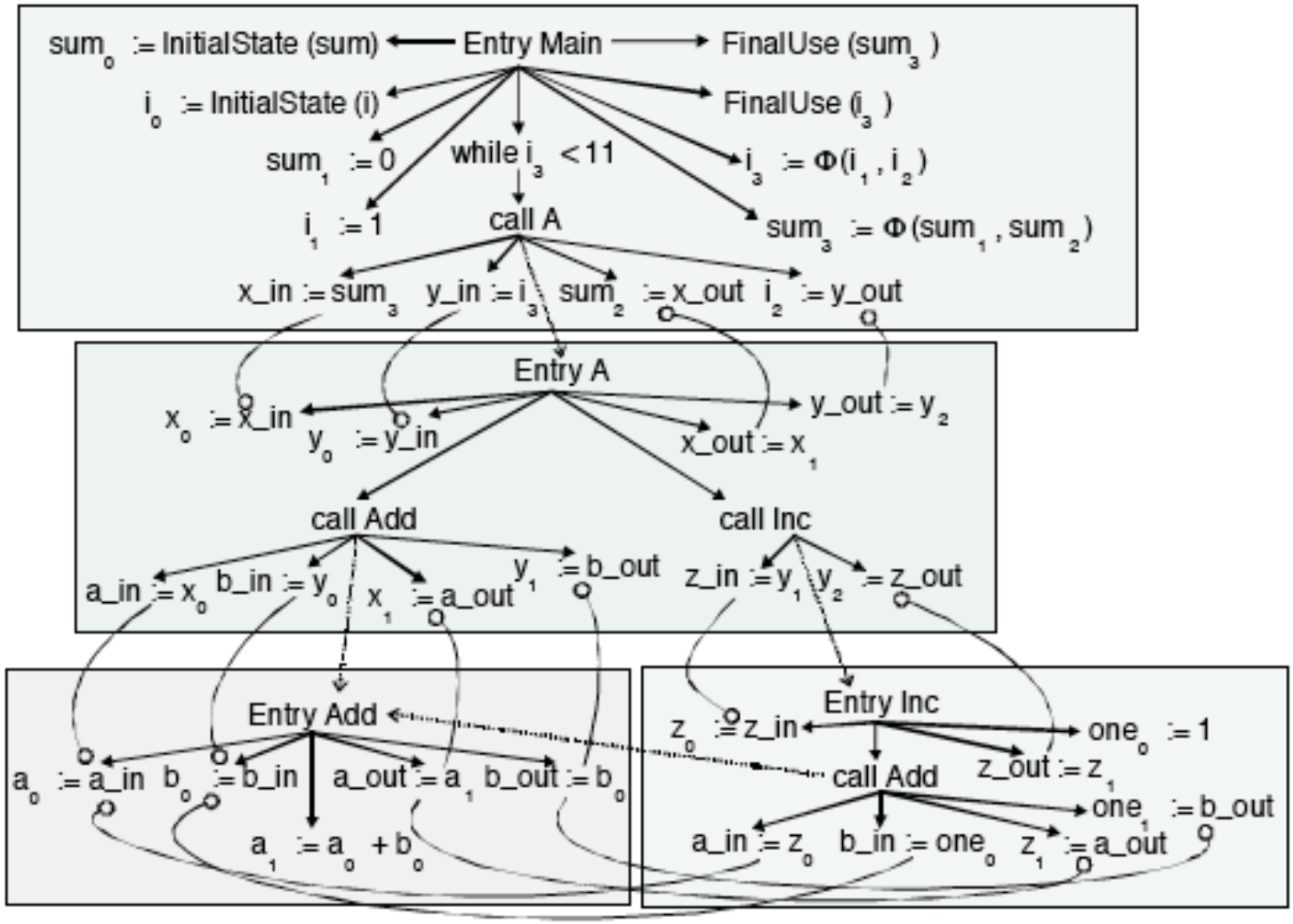
Modellierung des Prozeduraufrufs

```
x_in := sum3;
y_in := i3;
A;
sum2 := x_out;
i2 := y_out;
```

```
procedure A is
begin
  x0 := x_in; y0 := y_in;
  ...
  x_out := x1; y_out := y1;
end A;
```



System Dependence Graph



Literatur

K.B. Gallagher, J.R. Lyle (1991). Using Program Slicing in Software Maintenance. *IEEE Transactions on Software Engineering*, **17**(8):751-761.

S. Horwitz, T. Reps, D. Binkeley (1990). Interprocedural Slicing using Dependence Graphs. *ACM Transactions on Programming Languages and Systems* **12**(1):35-46.

R. Koschke, J.-F. Girard, M. Würthner (1998). An Intermediate Representation for Reverse Engineering Analyses. *Proceedings of the Working Conference on Reverse Engineering*.

R. Morgan (1998). *Building an Optimizing Compiler.* Digital Press.

M. Weiser (1984). Program Slicing. *IEEE Transactions on Software Engineering* **10**(4):352-257.