SE 2014 - Doktorandensymposium

# Kombinierte statische Programmanalysen zur Erkennung semantischer Codeklone

**Torsten Görg**

**Institut für Software-Technology (ISTE)**
**Abteilung Programmiersprachen und Übersetzerbau**
**Universität Stuttgart**

# Codeklon-Beispiel

```
class Auto {
   private int gefahreneKilometer;
   private double tankFuellstand;
   private Motor motor;

   public Auto (Motor motor) {
      this.gefahreneKilometer = 0;
      this.tankFuellstand = 60.0;
      this.motor = motor;
   }
}
```

# Codeklon-Beispiel

```
class Auto {
   private int gefahreneKilometer;
   private double tankFuellstand;
   private Motor motor;

   public Auto (Motor motor) {
      this.gefahreneKilometer = 0;
      this.tankFuellstand = 60.0;
      this.motor = motor;
   }

   public Auto () {
      this.gefahreneKilometer = 0;
      this.tankFuellstand = 60.0;
      this.motor = new Motor ();
   }
}
```

# Codeklon-Beispiel

```
class Auto {
   private int gefahreneKilometer;
   private double tankFuellstand;
   private Motor motor;

   public Auto (Motor motor) {
      this.gefahreneKilometer = 0;
      this.tankFuellstand = 60.0;
      this.motor = motor;
   }

   public Auto () {
      this.gefahreneKilometer = 0;
      this.tankFuellstand = 60.0;
      this.motor = new Motor ();
   }
}
```

**Klon!**

# Codeklone

- Codeklone sind Redundanzen im Quelltext
- Kloninformationen wichtig bei der Wartung
  - Konsistentes Beheben von Fehlern
  - Konsistentes Modifizieren und Erweitern von Programmteilen
  - Grundlage für Refaktorisierungen, bei denen Coderedundanzen durch Abstraktionen eliminiert werden
- Auch zur Prüfung auf Plagiate einsetzbar

# Codeklon-Erkennung

- Formales Kriterium für semantische Äquivalenz

  $f1 : I \rightarrow O,\ f2 : I \rightarrow O,\ \forall i \in I : f1(i) = f2(i)$

- Problem: Semantische Äquivalenz im Allgemeinen nicht entscheidbar

- Daher Klonerkennung häufig approximiert durch strukturvergleichende Verfahren
  - Text-, Token- oder AST-basiert
  - Ggf. Abbildung von Struktureigenschaften auf Metriken

# Codeklon-Beispiel 2

```
public Element find (ElemList list) {
   ElemList current = list;
   while (current != null) {
      Element elem = current.data;
      if (elem.value > 153) {return elem;}
      current = current.next;
   }
   return null;
}
```

# Codeklon-Beispiel 2

```
public Element find (ElemList list) {
   ElemList current = list;
   while (current != null) {
      Element elem = current.data;
      if (elem.value > 153) {return elem;}
      current = current.next;
   }
   return null;
}

public Element search (Vector<Element> elements) {
   for (int i=0; i<elements.size (); i++) {
      if (elements.elementAt (i).value > 153) {
         return elements.elementAt (i);
      }
   }
   return null;
}
```

# Codeklon-Beispiel 2

```
public Element find (ElemList list) {
    ElemList current = list;
    while (current != null) {
        Element elem = current.data;
        if (elem.value > 153) {return elem;}
        current = current.next;
    }
    return null;
}
```

```
public Element search (Vector<Element> elements) {
    for (int i=0; i<elements.size (); i++) {
        if (elements.elementAt (i).value > 153) {
            return elements.elementAt (i);
        }
    }
    return null;
}
```

**semantischer Klon!**

# Semantische Codeklone

- Aufspüren verschiedener Implementierungen semantisch äquivalenter oder semantisch ähnlicher Funktionalitäten
- Ermöglicht Anpassung nichtfunktionktionaler Eigenschaften
  - Ersetzen von Kloninstanzen durch bereits existierende Implementierungsalternativen
  - Ggf. auch zur Parallelisierung sequentiellen Codes geeignet
- Weitreichendere Kloneliminierung möglich
  - Ggf. Etablierung von DSLs auf Basis der in den Klonen repräsentierten Konzepte

# Erkennen semantischer Klone (1)

- Testbasierte, dynamische Verfahren
  - Gehen unmittelbar von der formalen Definition bzgl. des IO-Verhaltens aus
  - Erkennen Äquivalenzen auch bei stark unterschiedlichen Implementierungen
  - Haben die üblichen Probleme von Tests (kombinatorische Vielfalt, Testüberdeckung, oft bestimmte Umgebung notwendig)
  - Für semantische Ähnlichkeiten kaum geeignet

# Erkennen semantischer Klone (2)

- Mittels lokaler Baum- bzw. Graphtransformationen
  - Termersetzungssystem
  - Notwendige Eigenschaften
    - Terminierung
    - Konfluenz
  - Vollständige Konfluenz für beliebigen Quellcode nicht möglich
  - Ersetzungsregeln müssen vorgegeben werden

# Erkennen semantischer Klone (3)

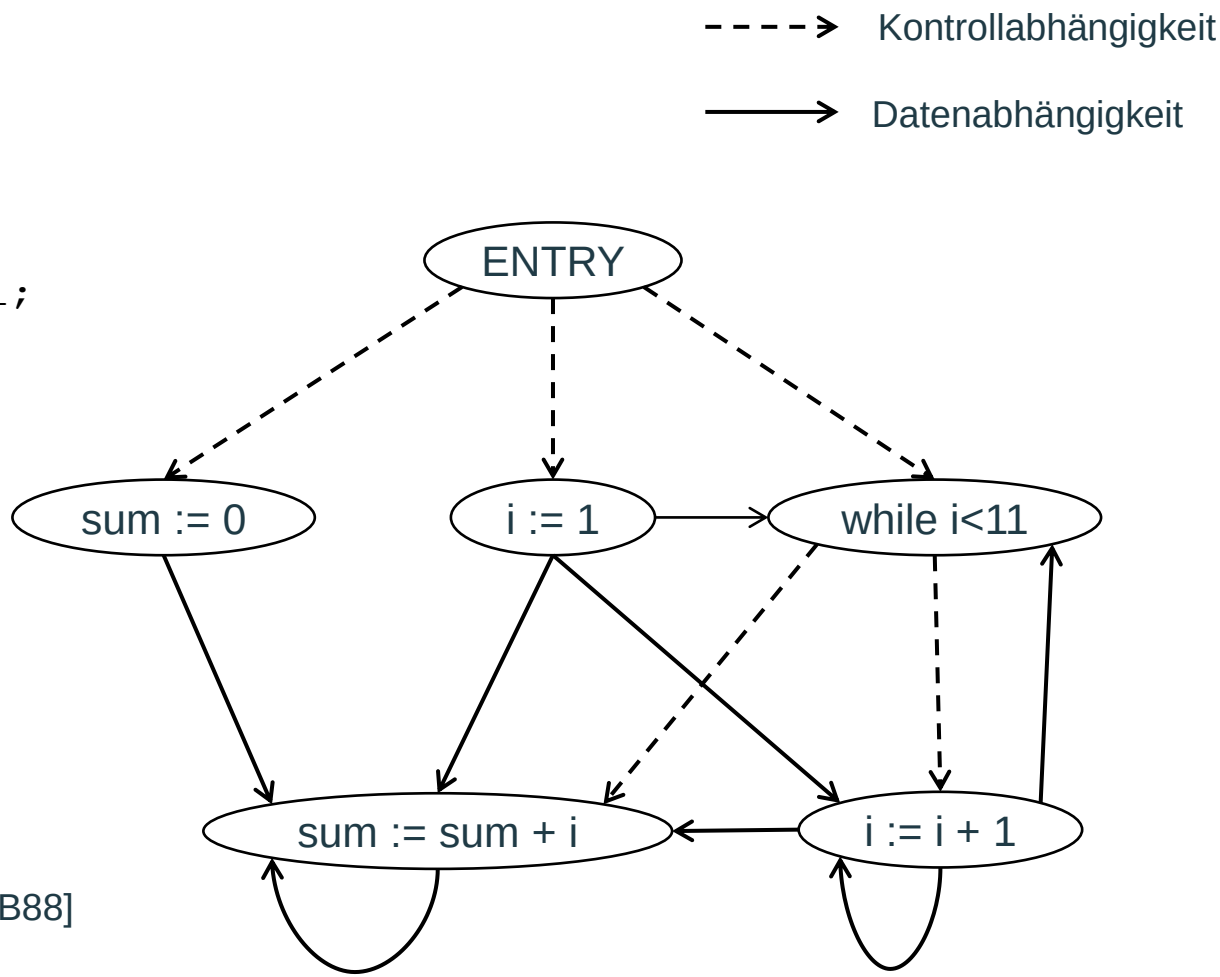
- Abstrakte Interpretation der Speicherzustände
  - Berechnet statisch für jeden Programmpunkt aktuelle Werte für alle Programmvariablen
  - Alle Werte mit Pfadbedingungen annotiert
  - Dafür pfadsensitive Analyse notwendig
  - Zusammenfassungen für das Verhalten der Unterprogramme
  - Gut zum Erkennen von Klonen über Routinengrenzen hinweg
  - Pfadweises Vereinfachen von Ausdrücken
    - Vereinfachungen möglich, die bei anderen Verfahren nicht möglich sind
    - Umformungen deutlich rechenaufwendiger

# Erkennen semantischer Klone (4)

- Basierend auf Programmabhängigkeitsgraphen (PDG)
  - Explizite Darstellung der Daten- und Kontrollabhängigkeiten
  - Toleriert Vertauschungen von Anweisungen
  - Toleriert Verzahnungen des Codes unterschiedlicher Features
  - Toleriert zusätzliche Zuweisungen für Zwischenergebnisse
  - Unterschiedliche Aufteilungen in Unterprogramme problematisch
  - Normalisierung der Datenflüsse aber nicht der Kontrollflusses

# PDG-Beispiel

```
program Main
    sum := 0;
    i := 1;
    while i<11 do
        sum := sum + i;
        i := i + 1;
    od
end
```



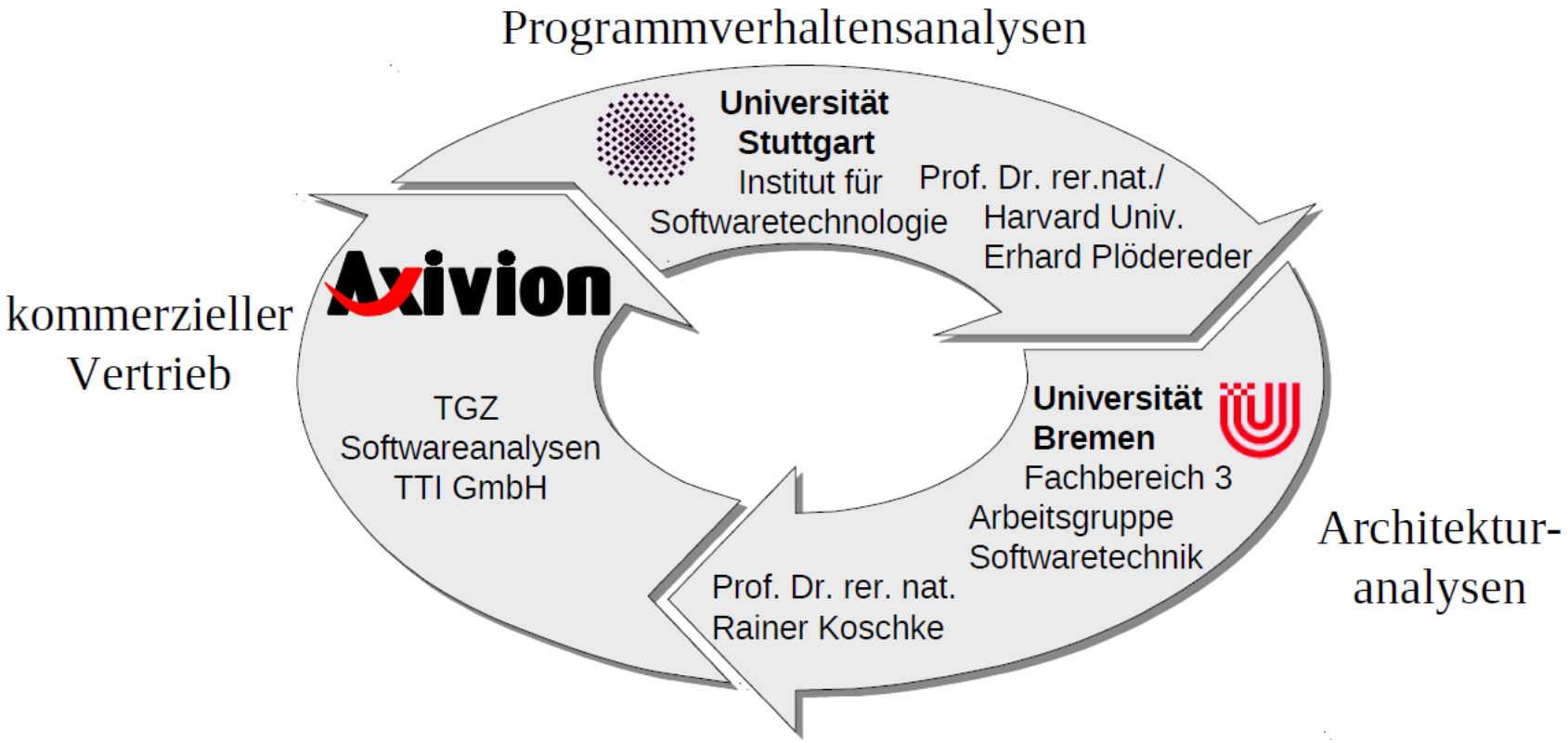

Beispiel entnommen aus [HRB88]

# Dissertationsthese

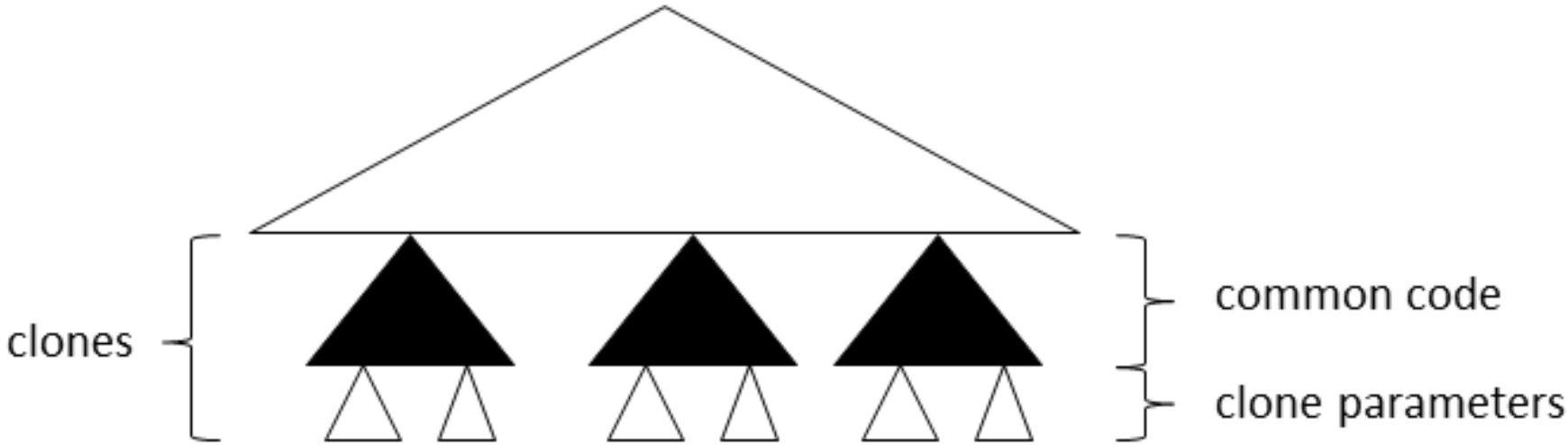
1. In realer Software gibt es einen signifikanten Anteil semantisch redundanten Codes, der sich mit reinen PDG-basierten Klonerkennungs-verfahren nicht erfassen lässt.

2. Ein Teil dieser semantischen Redundanzen lässt sich mittels kombinierter statischer Programmanalysen erkennen.

# Programmanalysen-Framework Bauhaus



# Äquivalenz vs. Ähnlichkeit




- 2 horizontale Schnitte im AST
- Unterschiede bei Ähnlichkeit durch Klonparameter ausgedrückt

# Variationsmöglichkeiten (1)

- Umbenennungen
- Unterschiedliche aber äquivalente arithmetische Ausdrücke
- Unterschiede im Kontrollfluss
  - Vertauschen von Anweisungen
  - Verzahnung mehrerer Features
  - Unterschiedliche Sprachkonstrukte (z.B. Iteration vs. Rekursion)
  - Loop-Unrolling
  - Schleifeninvarianter Code
  - Aufteilung von Schleifen und bedingten Ausführungen

# Variationsmöglichkeiten (2)

- Ausprägung der Datenflüsse
  - Zusätzliche Variablen für Zwischenergebnisse
  - Verschiedene Aufteilungen von Prozeduren
  - Globale vs. lokale Variablen
- Ausprägung der Algorithmik
  - Verallgemeinerungen von Funktionalitäten
  - Unterschiedliche Algorithmen
  - Unterschiedliche Datenstrukturen
- Parametrisierungen (Konzept-Klone)
- Parallelisierte Ausführung

# Lösungsansatz

- Kombiniertes Verfahren
  - Ausnutzen der Stärken mehrerer Ansätze
- PDG als Grundlage
- Weitere (partielle) Normalisierungen
  - Lokale Graphtransformationen
- Strukturvergleich auf dem normalisierten Graphen
  - Teilgraphisomorphieproblem

# Kanonische Form für Ausdrücke

- Vollständige Normalform für einfache Ausdrücke möglich
  - Partielle Ordnung der Operatoren definieren
  - Ausdrücke entsprechend der Operator-Ordnung umformen
  - Partielle Ordnung auf Ausdrücken definieren
  - Operanden kommutativer Operatoren entsprechend dieser Ordnung sortieren
  - Algebraische Regeln zur Termvereinfachung anwenden
- Kanonisierte Ausdrücke zu Stellvertreterknoten kontrahieren

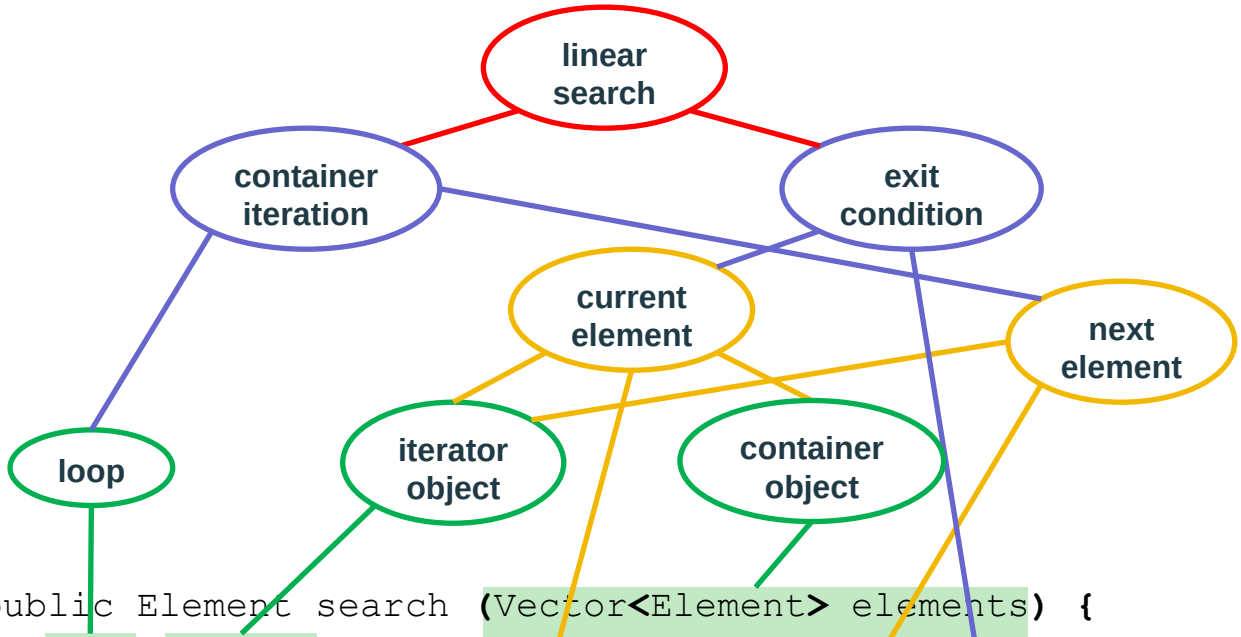
# Code-Optimierungs-Techniken

- Compiler nutzen Variationsmöglichkeiten zur Code-Optimierung
  - Bildet unterschiedlichen Code auf den gleichen optimierten Code ab
  - Optimierungen sind konservativ:

$$optim: C \rightarrow C',\ C' \subset C,$$
$$\forall c_1, c_2 \in C: optim(c_1) = optim(c_2) \Rightarrow (\forall i \in I: f_{c_1}(i) = f_{c_2}(i))$$

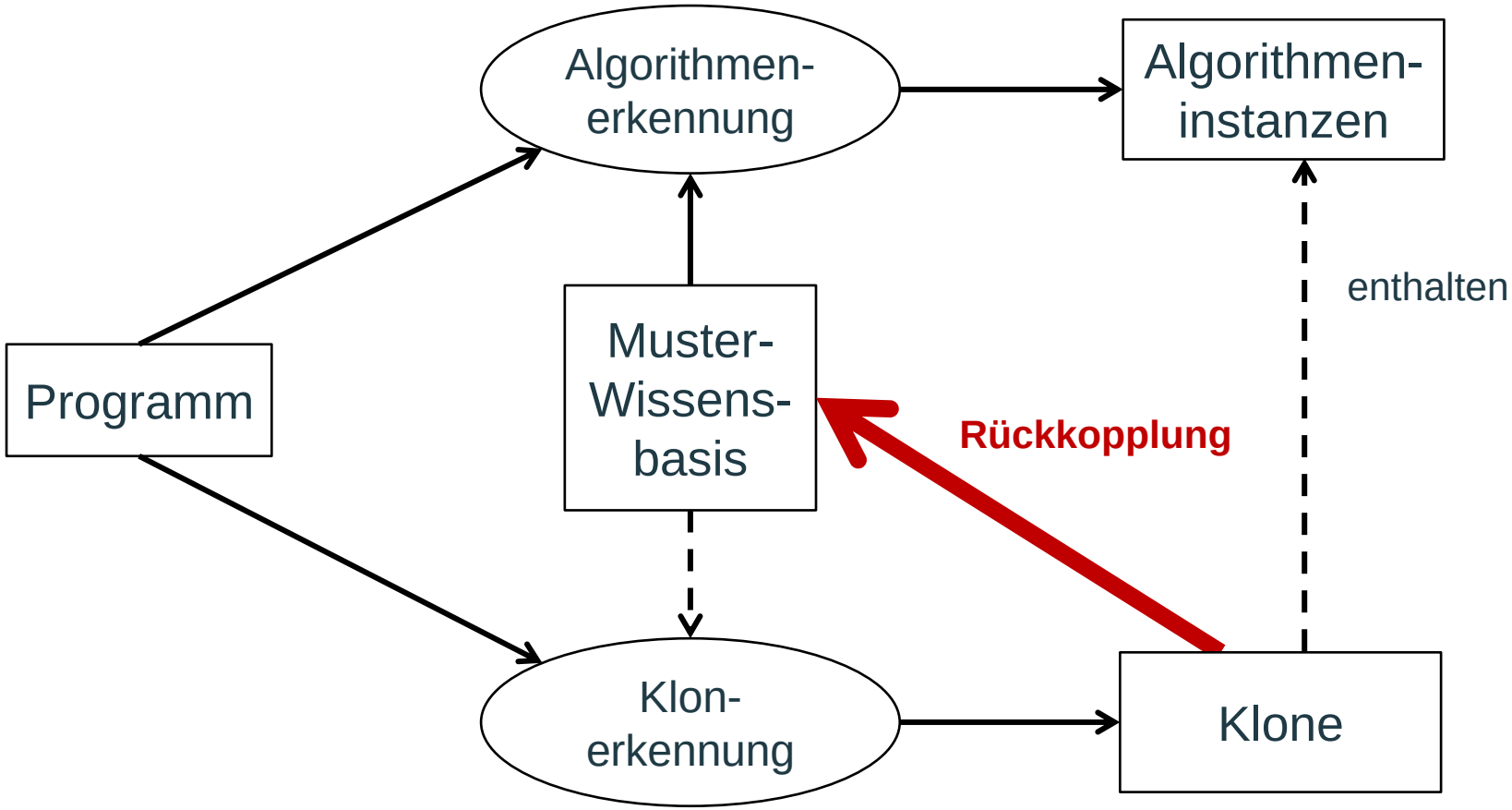
- Beispiele:
  Konstantenpropagierungen, Konstantenfaltung, Strength Reduction, Extraktion schleifeninvarianten Code, usw.

# Wissensbasiertes Programmverstehen



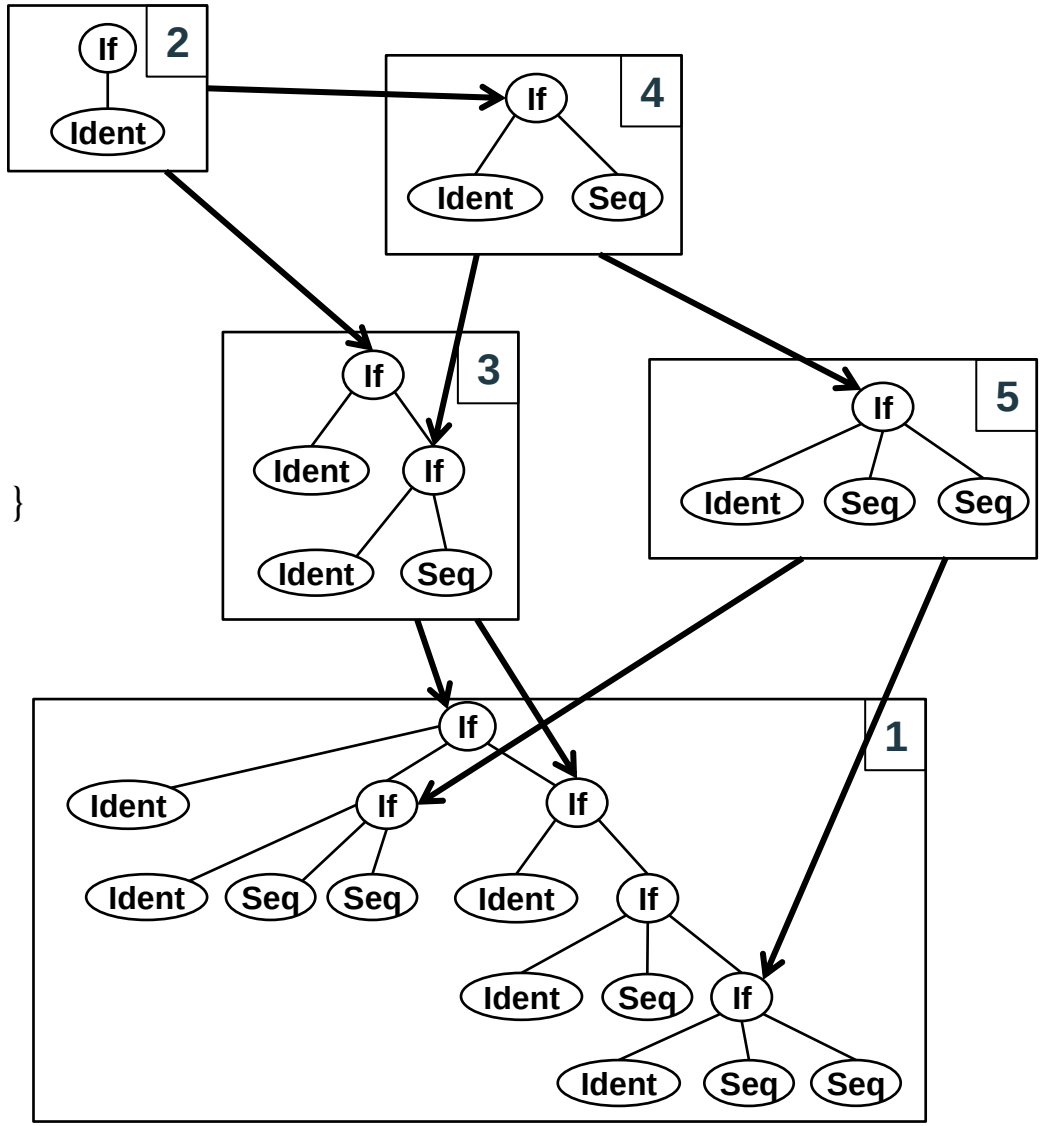

```
public Element search (Vector<Element> elements) {
    for (int i=0; i<elements.size (); i++) {
        if (elements.elementAt (i).value > 153) {
            return elements.elementAt (i);
        }
    }
    return null;
}
```

# Bezug zur Algorithmenerkennung



# Klongruppenhierarchie

```
if (cond1)
   if (cond2) {}
   else {}
else
   if (cond3)
      if (cond4) {}
      else
         if (cond5) {}
         else {}
```



# Normalisierung der Typen

- Strukturen vs. Einzelvariablen
  - Strukturen auf ihre einzelnen Komponenten herunterbrechen?
- Strukturäquivalente Typen
- Rekursive Typdefinitionen deuten auf Container hin
- Oft Referenzübergaben statt Wertübergaben

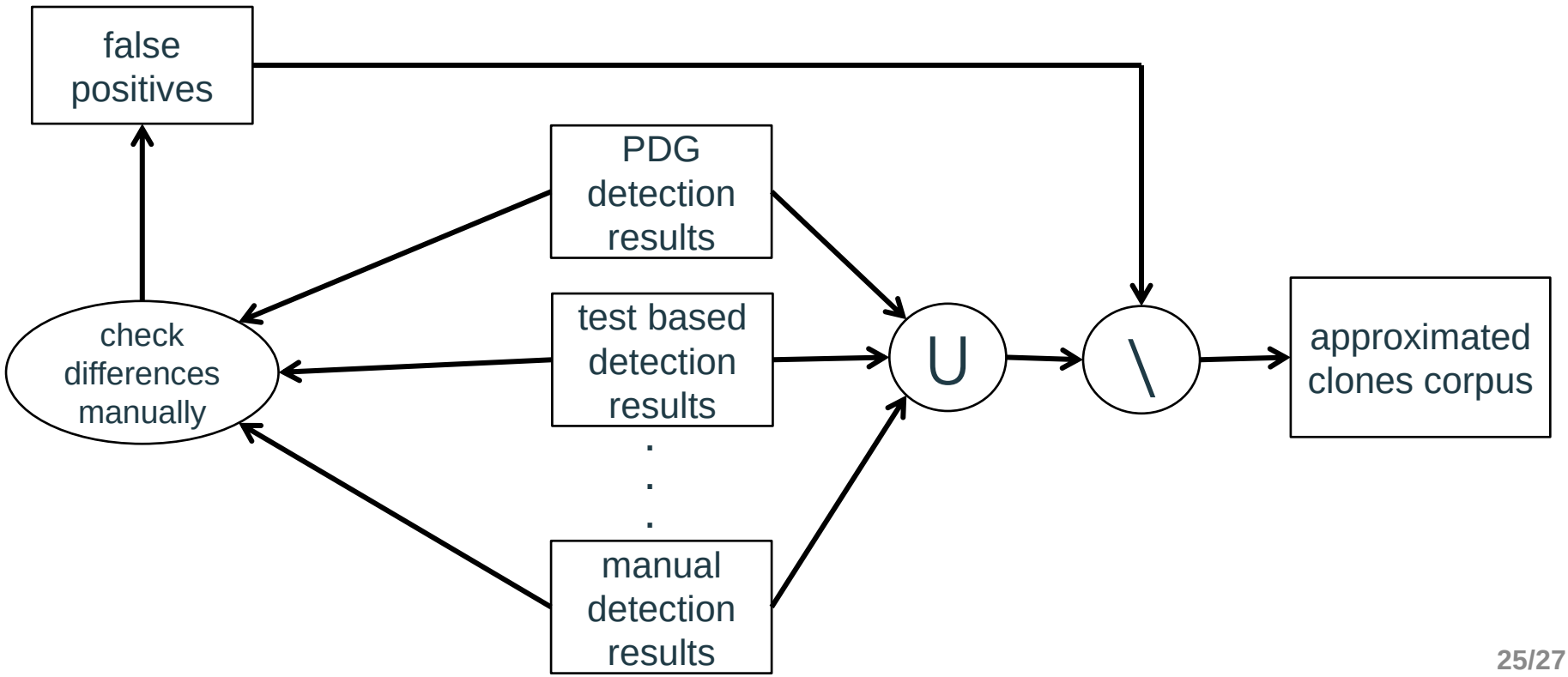
# Validierung (1)

- In Anlehnung an eine Untersuchung von Stefan Bellon et al. [BKAKM07]

- Bewertungsmetriken
  - $Recall(P,T) = \frac{|DetectedRefs(P,T)|}{|Refs(P)|}$ , $Recall(P,T) \in [0;1]$
  - $Precision(P,T) = \frac{|DetectedRefs(P,T)|}{|Cands(P,T)|}$ , $Precision(P,T) \in [0;1]$
  - $Quality(P,T) = Recall(P,T) * Precision(P,T)$ , $Quality(P,T) \in [0;1]$
  - $OnlyRefs(P,T) = DetectedRefs(P,T) \setminus (\cup\, DetectedRefs(P,Ti), Ti \neq T)$

# Validierung (2)

- Approximation eines Referenz-Korpus



# Quellen

[HRB88] Susan Horwitz, Thomas Reps, and DavidBinkley: Interprocedural Slicing Using Dependece Graphs, Proceedings of the SIGPLAN ’88 Conference on Programming Language Design and Implementation, ACM, 1988

[BKAKM07] Stefan Bellon, Rainer Koschke, Giuliano Antoniol, Jens Krinke, Ettore Merlo: Comparison and Evaluation of Clone Detection Tools, IEEE Transaction on Software Engineering, Vol. 33, No. 9, September 2007

# Offene Fragen

- Weitreichendere Formulierung der Dissertationsthese?
- Transformationsrichtung hin zum Abstrakteren oder zum Konkreteren?
- Ausreichende Performanz erreichbar?
- SSA-Form statt PDGs, um die Anzahl der Kanten zu reduzieren?