Aktueller Stand: Tool zur Überdeckungsmessung und zum strukturorientierten Softwaretest

Ronny Treyße & Hendrik Seffler

HU Berlin

Entwicklungsziele

Erstellung eines Programms, das den strukturorientierten Softwaretest unterstützt

- Analyse (d.h. Parsen) von Quellcode in C++ und Java
- Ableitung von Komplexitätsmaßzahlen
- Ermittlung von Überdeckungsmaßen (im Zusammenspiel mit ATOS)
- Instrumentierung von Programmcode in beiden Sprachen
- Auswertung und Darstellung der Resultate

Entwicklungsziele

- Darstellung des Kontrollflusses
- Nachvollziehen des Kontrollflusses für einzelne Testfälle bzw. Programmdurchläufe
- Architekturinformationen gewinnen

Struktur

Wie haben die Aufgabenstellung in zwei Teile zerlegt

- Parser/Instrumentierer
  Parsen des Quelltextes
  Instrumentierung des Quelltextes
- Parser sowohl stand-alone als auch aus der grafischen Oberfläche einsetzbar
- Auswertung/Steuerung
  Grafische Oberfläche zur Konfiguration/Projekterstellung
  Auswertung der Ergebnisse und deren Darstellung
- Kapselung der Sprachabhängigkeiten im Parser

Grundsätzliche Implementierungentscheidungen

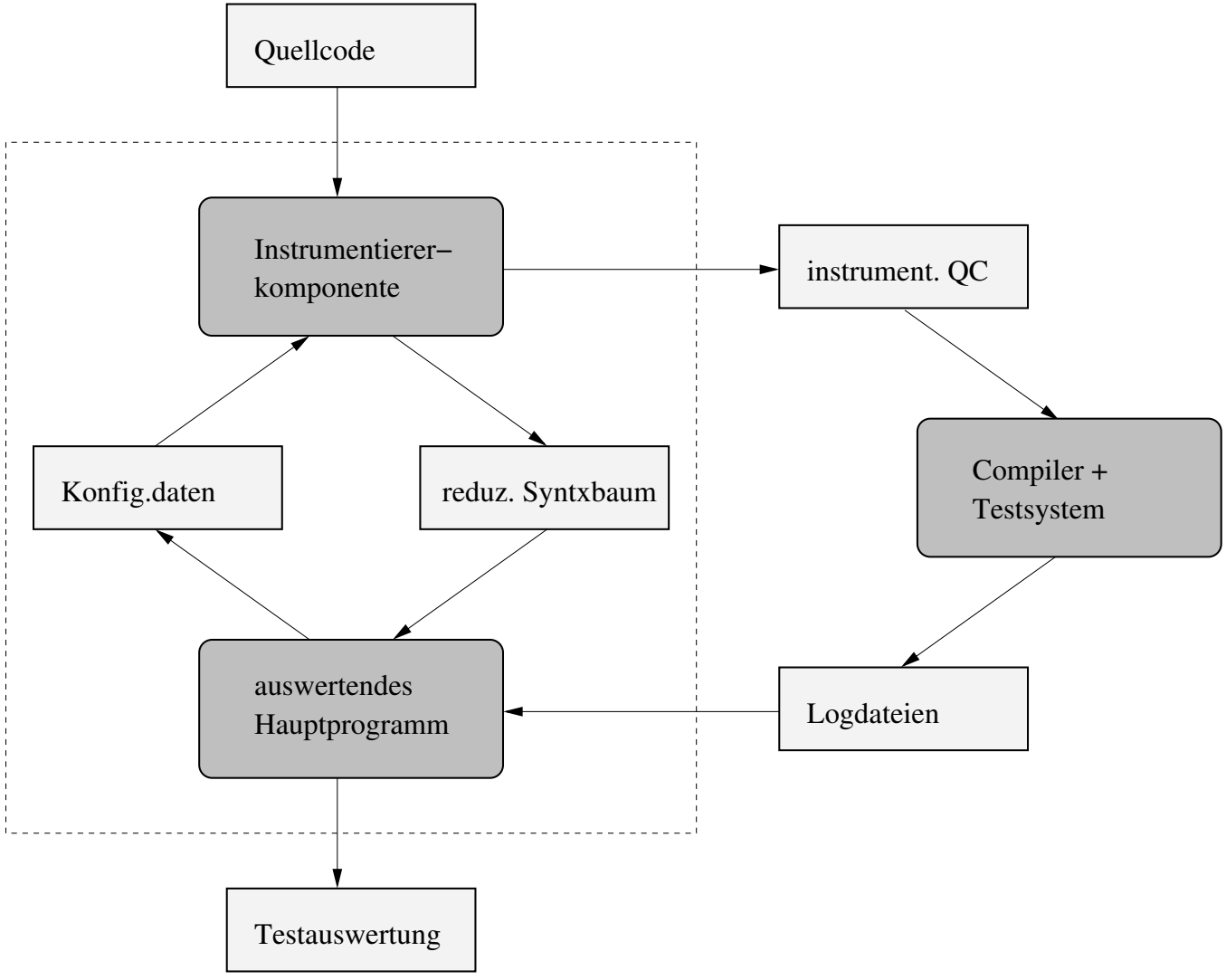
- Implementierung in Java (Instrumentierer und Auswerter)
- Nutzung eines Parser-Generators (JavaCC)
- Kontrollflussgraphen mit JGraph-Bibliothek
- keine weiteren Abhängigkeiten

Überblick Parserkomponente

Die Parserkomponente muss Folgendes leisten:

- Parsen von C++ und Java-Programme
- Filtern von Strukturinformationen des Programmes
- Konstruktion eines Abstrakten Syntaxbaums (AST)
- Erstellen einer instrumentierten Version des Programms anhand des AST

Subsystem Parser im Gesamtsystem



wesentliche Komponenten des Subsystems

- Parser-Klassen für C++ und Java
- ASTManager-Klasse und entsprechende Knotenstrukturen
- Klassen/Funktionen, die Logausschriften in C++ bzw. Java übernehmen

JavaCC

- ist populärster CompilerCompiler für Javaprojekte
- entwickelt bei Sun Labs
- seit letztem Sommer OpenSource und unter BSD-Lizenz
- geschrieben in Java und produziert Java-Code
- generiert Lexer und TopDown-Parser (LL(1)) aus einem (EBNF-)Grammatikfile
- bietet syntaktischen und semantischen Lookahead für Konfliktauflösung

vorgefundene Grammatiken für JavaCC

- Java-Grammatik:
  - aktuell und wird gewartet
  - bisherige Testläufe ohne Probleme
- C++-Grammatik:
  - Date 97 ?!
  - arbeitet nur auf präprozessierten Quellcode
  - kein namespace, typename, type_id, keine differenzierten Casts (dynamic_cast, ...)
  - bei Testläufen kleinere Unverträglichkeiten bei speziellen Konstrukten (z.B.: const double Pi(3);)

aktueller Stand - Parser

- Java-Parser:
  - funktionstüchtig, aber noch nicht umfangreich getestet
- C++-Parser:
  - parst mit gefixter Grammatik neuere Schlüsselworte
  - Unverträglichkeiten sind noch zu beseitigen
  - offene Fragen:
    - lässt sich das Präprozessieren umgehen?
    - wie mit Makros umgehen?

Abstrakte Syntaxbaum

- AST-Manager
  - baut während des Parsens den AST auf
  - führt Aktionen über den AST aus
- AST-Struktur
  - entspricht strukturell dem Kontrollflussgraphen
  - enthält spezielle Knoten für kontrollflussrelevante Strukturen (if, while, ..)
  - lässt komplette Rekonstruktion des Quellcodes inkl. Instrumentiereungsansweisungen zu
  - speichert Informationen über die Erfüllung von Überdeckungsmaßen nach Logauswertung

aktueller Stand - AST

- Konstruktion des AST für Javaprogramme und rudimentäre Instrumentierung funktioniert
- noch zu erledigen:
  - Bedingungsanalyse vollenden
  - Instrumentierung nach verschiedenen Level implementieren

Logger-Klasse

- stellt Klassen bzw. Funktionen zur Verfügung, die die Logging-Funktion im instrumentierten Programm übernehmen
- globale Struktur, die überall im Quellcode zugänglich sein muss
- offene Fragen:
  - Synchronisation des Logfilezugriffs und der Datenaktualisation bei geforkten Prozessen
  - eleganteste Logging-Methode

Logging-Testversuch

- Testsystem: Simulationsprogramm(24kB) auf Basis der Odem-Bibliothek (J.Fischer)
- Laufzeit: <2 Sek, startet ca. 1000 Prozesse
- Ergebnis: naives, episodisches Logging erstellt Logfile >1MB
- Ergo: Logging-Struktur muss während Laufzeit Datenstruktur halten, warten und sichern

Fazit - Parsersubsystem

- Parser für Java einsatzbereit
- Parser für C++ noch mit grundlegenden Problemen
- AST wird optimiert und bzgl. der Bedürfnisse bei der Auswertung modifiziert
- Logger-Funktionalität muss geeignet für komplexere Szenarien entworfen werden

Aufgaben des Hauptprogramms

Aufgaben des Hauptprogramms

- Projekterstellung und Speicherung
  Pfade, Parameter und Einstellungen
- statische Quelltextanalyse
- Ergebnisse von Testdurchläufen auswerten

Struktur

Grundsätzliche Struktur der Oberfläche

- Projekt-Management
- Komplexitätsauswertung
- Überdeckungsmaße bestimmen
- Architektur

Projekt-Management

Daten über Projekte werden in Projekt-Dateien abgelegt

- beim Laden: Prüfung auf Aktualität
- Ablage in XML
- enthalten: Schlüssel-Wert Zuordnungen in Kategorien
- Erstellung von Projekt manuell oder mit Assistent

Format festgelegt, wird gelesen und geschrieben. Abzulegende Daten festgelegt.

Sichten auf ein Projekt

Drei grundsätzliche Sichten auf ein geladenes Projekt

- Komplexität
- Architektur
- Überdeckung

Diese Sichten werden jeweils zusammengesetzt aus Grundtypen, die auf den gleichen Daten operieren und diese jeweils passend darstellen

Sichten auf ein Projekt

Sichten auf ein Projekt

- Kontrollflüsse
- Quelltextansicht
- Klassen-/Struktursicht
- Testfälle

Kontrollfluss

Aufgabe der Komponente: Darstellung des Kontrollflusses einer Methode

- Darstellung von Graphen komplexe Materie
- Verwendung einer Grafik-Bibliothek: JGraph
- Open-Source unter LGPL
- Alternativen? Grappa (Java Graph Package von AT&T) vergleichbar, Doku aber leider grausig

Kontrollfluss

Stand der Entwicklung
JGraph bietet keinen Layout-Algorithmus für Graphen, das Layout ist selbst auszuführen

- Nutzung eines generischen Algorithmus´ vs. Nutzung eines eigenen?
- Einfachere Prüfung der Korrektheit vs. Kontrolle über das Layout
- Noch nicht entschieden. Im Moment eigener, nicht kompletter Algorithmus, der nicht alle Sprachkonstrukte kennt
- Algorithmus wählen, Unterstützung aller Sprachkonstrukte

Quellcodeansicht

Sicht auf einzelne Methoden des Programms

- Quellcodeanzeige
- Formatierung wie im Original-Programm
- Farbige Hervorhebung von Kontrollflüssen
- Anzeige von (atomaren) Bedingungen und deren Belegungen

Quellcodeansicht

Stand der Dinge

- Einfache Darstellung von Quellcode ohne weitere Formatierung realisiert
- Darstellung von strukturiertem, formatiertem Text mit Java-Mechanismen möglich, aber recht komplexer Mechanismus

Testfalldarstellung

Darstellung von Testfällen

- Primär Anbindung an ATOS
- Auslesen und Verwalten von Testfällen aus ATOS-Projektdateien

Zusammen mit den Informationen, die der instrumentierte Quellcode liefert, ist so eine Zuordnung zwischen Programmdurchläufen und Testfällen, die sie ausgelöst haben, möglich

Klassen-/Strukturansicht

Darstellung der logischen Struktur des zugrundeliegenden Quellcodes: Klassen und Methoden

- Darstellung in einem Baum
- Übersicht über Programm
- Navigationshilfe
- Auswahl eines Elementes fokussiert andere Ansichten mit ausgewähltes Objekt

Fertig, funktioniert. Allerdings reagieren nicht alle Sichten auf Fokussierung

Logik

Von den GUI-Komponenten getrennt existiert eine Logikschicht, die die Auswertung der Ergebnisse der statischen und dynamischen Analyse übernimmt Insbesondere

- Komplexitätsbestimmung (essentiell, zyklomatisch)
- Überdeckungsbewertung

Logik-Komplexität

Auswertung der Komplexität von Quelltext

- Da Graph intern schon konstruiert relativ einfach
- Durchlauf durch Datenstruktur einer Methode und zählen von Knoten und Kanten
- essentieller Komplexität: nicht alle Knoten zählen, je nach Typ

Noch nicht realisiert, aber Infrastruktur dafür schon da

Logik-Überdeckung

Komplex. Erfordert intensive Auswertung der Daten der Log-Dateien des instrumentierten Codes inkl. überwachen von Konditionalen

- abhängig von instrumentiertem Programm
- Instrumentierung noch nicht fertig
- daher noch nicht in Angriff genommen

Ende