Aktueller Stand Entwicklung eines Parser- und Instrumentierungssystems für den strukturorientierten Programmtest

Ronny Treyße

HU Berlin

Gliederung

- Überblick über das Projekt
- Die Komponenten
- Die Schnittstellen
- Der Parsergenerator
- Die Umsetzung für Java und C++
- Aktueller Stand

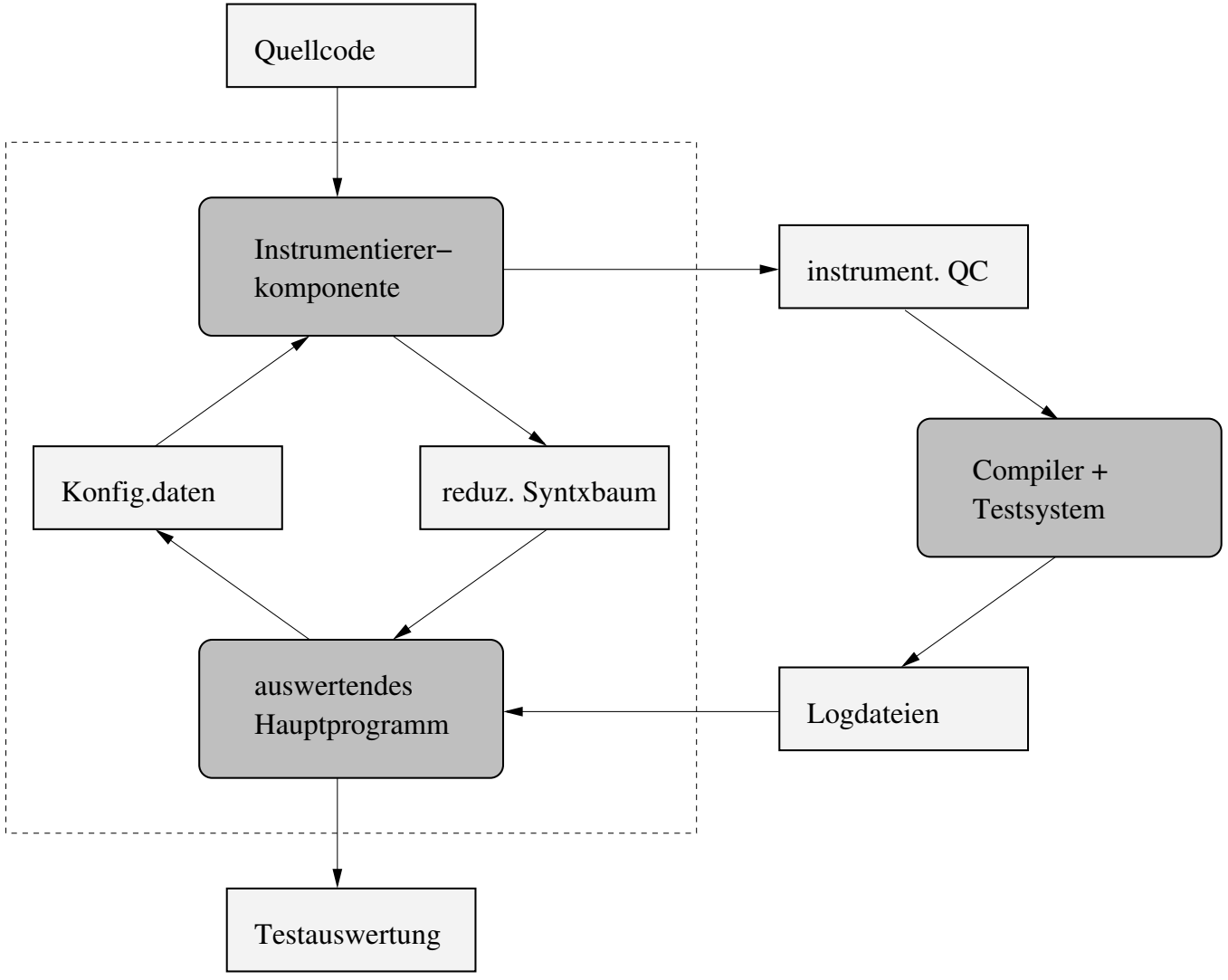
Überblick über das Projekt I

- Testsystem soll Überdeckungsmessung für strukturorientierten Programmtest für die Sprachen Java und C++ vornehmen
- folgende Überdeckungsmaße sollen unterstützt werden:
  - Anweisungsüberdeckung (C0)
  - Zweigüberdeckung (C1)
  - einfache (C2) und mehrfache (C3) Bedingungsüberdeckung
  - minimal mehrfache Bedingungsüberdeckung
  - MC/DC-Überdeckung
  - Boundary-Interior-Pfadtest

Überblick über das Projekt II

- Zweitteilung der Aufgabenstellung:
  - Parser- und Instrumentierungsfunktionalität
  - Auswertender Teil (Hendrik Seffler)
- Parsersubsystem soll folgende Funktionalität gewährleisten:
  - Parsen von Quellcode in Java und C++
  - Erstellen eines Abstrakten Syntaxbaumes (AST)
  - variable Instrumentierung des Quellcodes zur Ermittlung der Überdeckung
  - Erstellen und Einlesen der Logdatei

Überblick über das Projekt III



Die Komponenten

- Sprachunabhängige Komponenten:
  - ASTManager
  - grundlegender AST-Baum
  - LogReader
- Sprachabhängige Komponenten (je Sprache):
  - abgeleiteter AST-Baum
  - Parser inkl. Symboltabelle
  - Logging-Komponente für jede Sprache

Die sprachunabhängigen Komponenten

- ASTManager:
  - verwaltet AST-Baum
  - vermittelt Zugriff auf Parser, Logreader
  - veranlasst Parsen, Instrumentieren und Loglesen
  - verarbeitet Konfigurationen für die Instrumentierung
- grundlegender AST-Baum:
  - bietet sprachunabhängigen Zugriff auf die geparste Quelldatei
  - fast komplett abstrakt
- LogReader:
  - liest standardisiertes Logfile in AST-Baum ein

Die sprachabhängigen Komponenten

- abgeleitete AST-Baum:
  - sorgt für sprachspezifische Verknüpfungen im AST-Baum
  - ermöglicht Instrumentierung in der Zielsprache
- Parser:
  - parst Quelldatei (gegebenenfalls zusätzliche Dateien)
  - erstellt kompletten AST-Baum
- LoggingKomponente:
  - stellt Logging-Funktionalität in der Zielsprache zur Verfügung

Schnittstellen zur Auswertungskomponente

- Parserkomponente kann sowohl alleinstehend als auch von der Auswertungskomponente angesprochen werden
- Funktionalität gegenüber Auswertungskomponente:
  - Initialisierung
  - Auslösen des Parsevorganges
  - Konfiguration der Instrumentierung
  - Auslösen der Instrumentierung
  - Auslösen des Loglesens
  - Übergabe projektbezogener Daten und des AST-Baumes

Konfiguration des Parsersystems I

- die Arbeit des Instrumentierers lässt sich wie folgt spezifizieren:
  - Sprache des Quellcodes
  - Globales Instrumentierungslevel
  - Zuordnung einzelner Klassen und Funktionen zu Instrumentierungsleveln
  - Liste der zu verarbeitenden Dateien
  - Angabe eines Präfixes für generierte Variablennamen zur Vermeidung von Konflikten
  - Wahl des Namensformat für erstellte Dateien
- die Spezifikation kann über die GUI oder eine Konfigurationsdatei erfolgen

Konfiguration des Parsersystems II

- die Instrumentierungslevel haben dabei folgende Bedeutung:
- Level 0 - keine Instrumentierung
- Level 1 - einfach Instrumentierung, betrifft nur CFG-relevante Statements (C0 und C1)
- Level 2 - zusätzlich Instrumentierung für Bedingungsüberdeckungen (alle Überdeckungsmaße)
- Level 3 - vollständige Instrumentierung (jedes einzelne Statement)

Verknüpfung mit ATOS

- Problem: jeder einzelne Testfall in ATOS muss mit einer Logdatei verknüpft werden
- vorgeschlagene Lösung: Umbenennung der Logdatei in einen entsprechenden Dateinamen nach Programmdurchlauf durch ATOS
- Verwaltung und Auswertung der Logdateien erfolgt durch die Auswertungskomponente

Der Parsergenerator JavaCC

- JavaCC ist populärster Parsergenerator für Javaprojekte, wird aktiv supported
- ist OpenSource und unter BSD-Lizenz
- ist Top-Down-Parser, d.h. generiert LL(1)-Parser, aber ermöglicht einfach lokalen, syntaktischen und semantischen Lookahead
- bietet Unterstützung durch andere Java-Tools, z.B. JJTree zur Syntaxbaum-Konstruktion
- ist benutzerfreundlich, modern und mit viele Grammatiken verfügbar

Die Umsetzung für Java

- Grammatik
  - Java ist prinzipiell einfach zu parsen, da klare und einfach definiert
  - vorgefundene Java-Grammatik für JavaCC ist aktuell und wird weiterhin gewartet
- Problem bei der AST-Erstellung
  - schwierigstes Problem liegt in der Unterscheidung von boolschen und binären Operatoren
  - betrifft alle Bedingungsauswertungen
  - vollständige Lösung scheint relativ schwierig zu sein
  - (trifft auch auf C++ zu)

Problem der Operatorunterscheidung

- Beispiel:
  - `if( a == b ) ...`
- Frage: ist hier == boolscher oder binärer Operator?
- Antwort muss statisch erfolgen, damit der AST korrekt konstruiert werden kann
- Lösungsansatz: in Symboltabelle wird der Variablentyp vermerkt und beim Parsen verglichen
- Problem: Operand kann nichttrivial sein: `ClassX.getVariable()` ...

Die Umsetzung für C++

- Grammatik
  - C++ generell extrem schwer zu parsen, eigentlich unparsebar
  - vorgefundene Grammatik ist Date 97 ?!
  - ist für die Arbeit auf präprozessierten Quellcode gedacht
  - verarbeitet kein `namespace`, `typename`, `type_id`, keine differenzierten Casts (`dynamic_cast`, ...), ...
  - bei Testläufen kleinere Unverträglichkeiten bei speziellen Konstrukten

Probleme bei C++ - I

- Typ-Erkennung für JavaCC-Grammatik
  - Unterscheidung zwischen Typ-Identifiers und Variablen-Identifiers notwendig
  - Beispiel:
    - `<ID> <ID> ( <ID> ) ;`
  - I Konstruktorinitialisierung, z.B.:
    - `ClassX instance_A(initValue);`
    - d.h.: `<Typ> <ID> ( <ID> ) ;`
  - II Funktionsdefinition, z.B.:
    - `ClassX function(ClassY);`
    - d.h.: `<Typ> <ID> ( <TYP> ) ;`

Probleme bei C++ - II

- d.h.: für diese Grammatik ist es notwendig, jeden eingeführte Typ in einer Symboltabelle zu speichern
- unangenehm: dafür ist die Verarbeitung der Headerdateien ist zwingend notwendig
- empfohlene Lösung ist die Anwendung des Parsers auf präprozessierten Quelldateien
- aus verschiedenen Gründen ist dies unbequem:
  - verschiedene Präprozessoren liefern verschiedenen Output (Erkennen des relevanten Teils ...)
  - Output entspricht nicht mehr dem vom Nutzer erstellten Quellcode
  - Informationen gehen verloren

Probleme bei C++ - III

- Alternativer Ansatz: Parsen der Headerdateien nach Typen
- Problem: rekursives Parsen der Headerdateien endet meist in C++System-headern
- diese enthalten meist nicht-standardisierte Makrodefinitionen und sind entsprechend sehr schwer parsbar
- weiteres Problem: Umgang mit Makros überhaupt

Aktueller Stand I

- Sprachunabhängige Komponenten:
  - fertig und sehr ausführlich getestet
- Umsetzung Java:
  - Unterscheidung binärer/boolscher Operator nur für nicht-komplexe Fälle korrekt
  - ansonsten komplett und durch die Selbsttests an unserem Projekt im Rahmen der DA Hendrik Sefflers in großen Umfang positiv getestet

Aktueller Stand II

- Umsetzung C++:
  - der AST und seine Funktionalität ist fertig implementiert
  - aus der Grammatik ergeben sich kleinere Unverträglichkeiten mit speziellen Konstrukten
  - analog zu Java ist Unterscheidung binärer/boolscher Operator nur für nicht-komplexe Fälle korrekt
  - Problem des Umgangs mit Typerkennung noch offen - Präprozessieren vs. Headerparsen
  - Makroverarbeitung noch offen

Fazit

- Sprachenunabhängige Komponenten komplett, fertige Schnittstellen zum Auswertungssystem
- System für Java fast uneingeschränkt einsatzbereit
- C++-Funktionalität bis auf Makros komplett, allerdings ist die Verarbeitung der Headerdateien offen