



Entwicklung eines Werkzeugs zur Überdeckungsmessung für strukturorientierte Programmtests

Ronny Treyße

HU Berlin

Gliederung

- ⑥ Überblick über das Projekt
- ⑥ Die Komponenten
- ⑥ Die Schnittstellen
- ⑥ Die Umsetzung für Java und C++
- ⑥ Die Benutzerinteraktion per GUI
- ⑥ Verbindung zu ATOSj
- ⑥ Ausblick

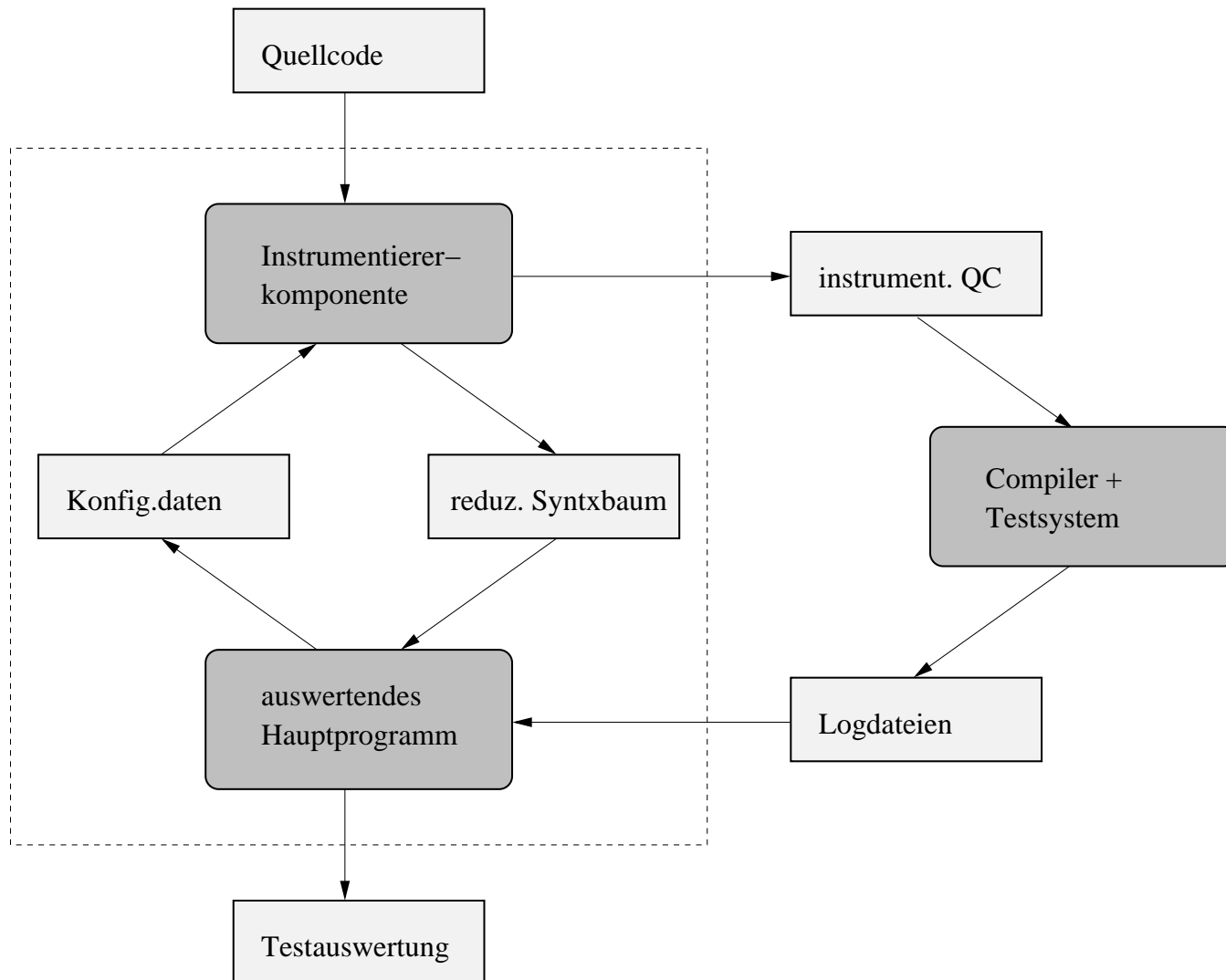
Überblick über das Projekt

- ⑥ Ziel: Bewertung strukturorientierter Programmtests
- ⑥ Programmiersprachen: Java und C++
- ⑥ 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

- ⑥ Zerteilung der Aufgabenstellung:
 - △ Parser-Subsystem
 - Parsen und Instrumentieren von Quellcode (Java, C++)
 - Erstellen eines Abstrakten Syntaxbaumes (AST)
 - Einlesen der Logdatei
 - △ GUI und Auswertung (Hendrik Seffler)
 - Projektverwaltung und Nutzerinteraktion
 - grafische Darstellung der Überdeckungsmaße
 - Testfallverwaltung
 - Report-Erstellung

Überblick über das Projekt



Die sprachunabh. Komponenten

- ⑥ ASTManager:
 - △ verwaltet AST-Baum
 - △ vermittelt Zugriff auf Parser, Logreader
 - △ veranlasst Parsen, Instrumentieren und Loglesen
 - △ verarbeitet Konfigurationen für die Instrumentierung

- ⑥ grundlegender AST-Baum:
 - △ bietet sprachunabhängigen Zugriff auf die geparste Quelldatei
 - △ fast komplett abstrakt

- ⑥ LogReader:
 - △ liest standardisiertes Logfile in AST-Baum ein

Die sprachabh. Komponenten

- ⑥ Parser:
 - △ parst Quelldatei (gegebenenfalls zusätzliche Dateien)
 - △ erstellt kompletten AST-Baum
- ⑥ abgeleitete AST-Baum:
 - △ sorgt für sprachspezifische Verknüpfungen im AST-Baum
 - △ ermöglicht Instrumentierung in der Zielsprache
- ⑥ LoggingKomponente:
 - △ stellt Logging-Funktionalität in der Zielsprache zur Verfügung

Schnittstellen zum GUI-/Auswertungssystem

- ⑥ Parserkomponente kann sowohl alleinstehend als auch von der Auswertungskomponente angesprochen werden
- ⑥ 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

- ⑥ 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
- ⑥ die Spezifikation kann über die GUI oder eine Konfigurationsdatei erfolgen

Instrumentierungslevel

- ⑥ die Instrumentierungslevel haben dabei folgende Bedeutung:
- ⑥ Level 0 - keine Instrumentierung
- ⑥ Level 1 - einfach Instrumentierung, betrifft nur CFG-relevante Statements (C0 und C1)
- ⑥ Level 2 - zusätzlich Instrumentierung für Bedingungsüberdeckungen (alle Überdeckungsmaße)
- ⑥ Level 3 - vollständige Instrumentierung (jedes einzelne Statement)

Der Parsergenerator JavaCC

- ⑥ JavaCC ist populärster Parsergenerator für Javaprojekte, wird aktiv supported
- ⑥ ist OpenSource und unter BSD-Lizenz
- ⑥ ist Top-Down-Parser, d.h. generiert LL(1)-Parser, aber ermöglicht einfach lokalen, syntaktischen und semantischen Lookahead
- ⑥ bietet Unterstützung durch andere Java-Tools, z.B. JJTree zur Syntaxbaum-Konstruktion
- ⑥ ToDo: aktuellere Grammatik für Java im Projekt umsetzen

Die Umsetzung für Java

⑥ Grammatik

- △ Java ist prinzipiell einfach zu parsen, da klar und einfach definiert
- △ vorgefundene Java-Grammatik für JavaCC ist aktuell und wird weiterhin gewartet

⑥ Problem bei der AST-Erstellung

- △ schwierigstes Problem liegt in der Unterscheidung von booleschen und binären Operatoren
- △ betrifft alle Bedingungsauwertungen
- △ vollständige Lösung scheint relativ schwierig zu sein
- △ (trifft auch auf C++ zu)

Problem der Operatorunterscheidung

⑥ Beispiel:

△ `if(a == b) ...`

⑥ Frage: ist hier `==` boolescher oder binärer Operator?

⑥ Antwort muss statisch erfolgen, damit der AST korrekt konstruiert werden kann

⑥ Lösungsansatz: in Symboltabelle wird der Variablentyp vermerkt und beim Parsen verglichen

⑥ Problem: Operand kann nichttrivial sein:
`ClassX.getVariable() ...`

Die Umsetzung für C++

⑥ 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++

- ⑥ 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++ - Lösungsansätze

- ⑥ Lösungsansatz: Symboltabelle für ALLE Typen
- ⑥ empfohlene Lösung:
 - △ Parsen präprozessierter Quelldateien
 - △ unpraktisch, da der Präprozessor den Quellcode stark verändern kann
- ⑥ alternativer Ansatz:
 - △ Parsen der Header nach Typen zur Erstellung der Symboltabelle
 - △ unangenehm, da Headerdateien oft nichtstandardisierte Makrodefinitionen enthalten

Probleme bei C++ - Lösung

- ⑥ praktikable Lösung - zwei Parserdurchläufe
 - △ 1. Parsen der präprozessierten Quelldateien nur nach Typen
 - △ Ergebnis: vollständige Symboltabelle für den Quellcode
 - △ 2. Parsen der originalen Quelldateien mit Hilfe der Symboltabelle
 - △ Ergebnis: AST des originalen Quellcodes

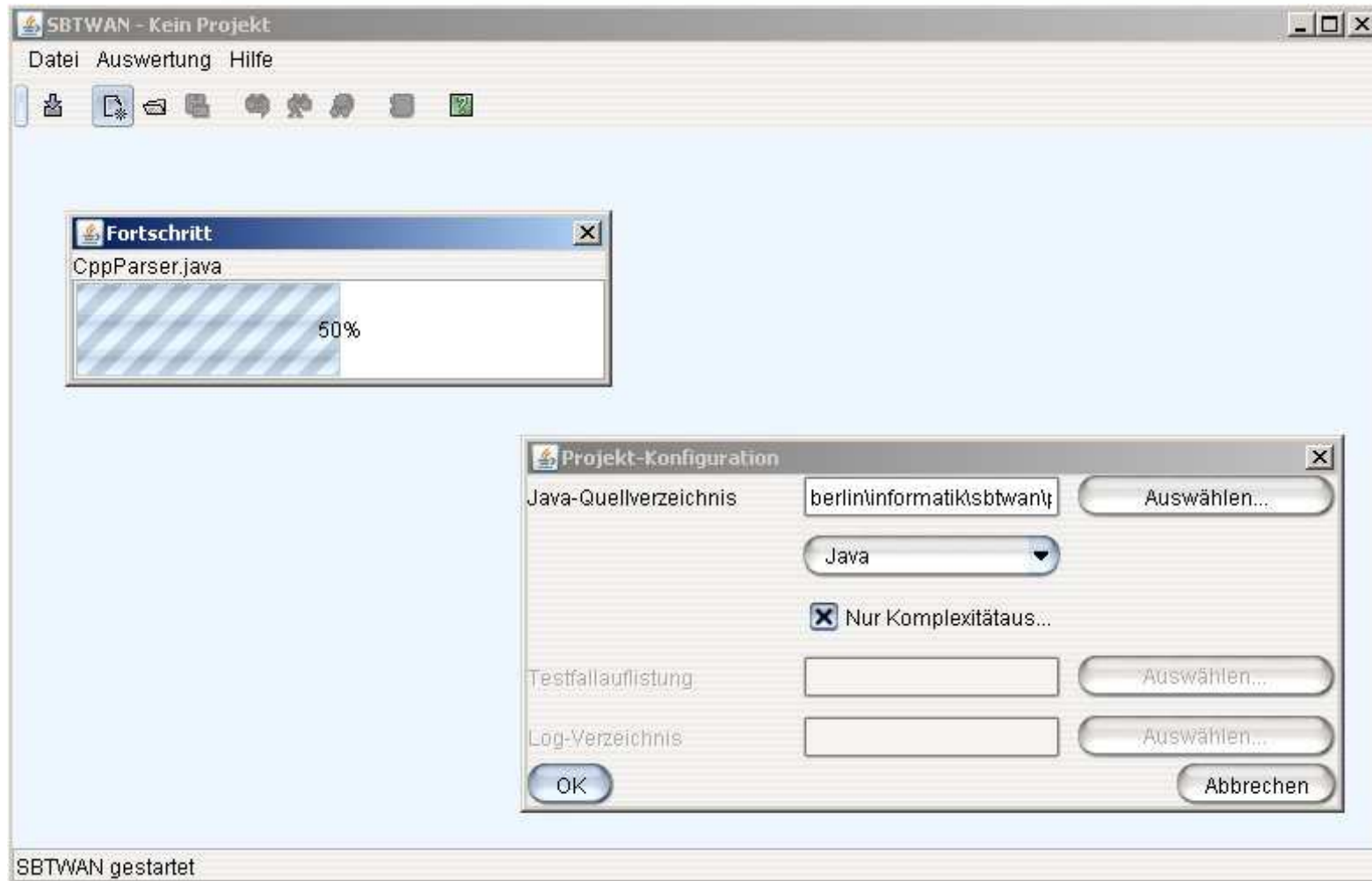
Benutzerinteraktion per GUI

- ⑥ vor dem Programmtest:
 - △ Projekterstellung
 - △ Einlesen der Quelldateien und unmittelbare Darstellung von Architekturinformationen, statischen Komplexitätsmaße und das KFG
 - △ Konfiguration und Instrumentierung
- ⑥ nach dem Programmtest:
 - △ Einlesen der Logfiles und Testfalldefinitionen
 - △ Berechnung der Überdeckungsmaße und ihre Darstellung am KFG und Quellcode in Abh. einer Testfallmenge
 - △ Erzeugung eines Reports

Schema der Überdeckungsdarstellung

Testfälle	Klassenstruktur	Sourcecode	Kontrollflussgraph
Test 1	glob. Funktionen		
Test 2	main()	int main() {	
Test 3	foo()	... if(a==1) {	
Test 4	Klassen class A foo_1() foo_2() class B foo_1() ...	a.foo_2(); } else { a.foo_1(); } b.foo_2(); }	<pre>graph TD; Start(()) --> Node1(()); Node1 --> Node2(()); Node2 --> Node3(()); Node3 --> Node4(()); Node4 --> Node5(()); Node5 --> Node6(()); Node6 --> End(());</pre>

Projekterstellung

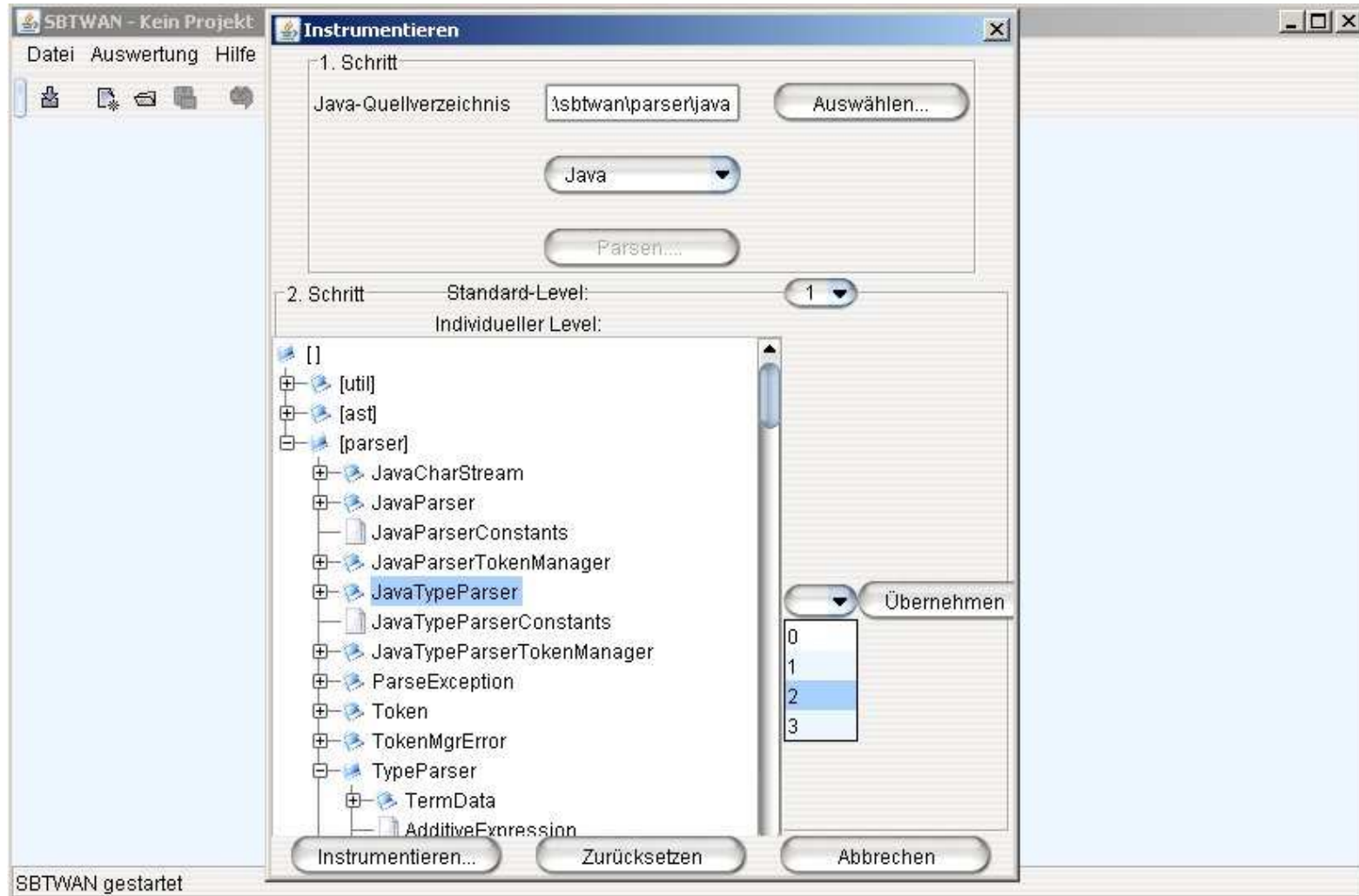


Statische Komplexitätsmaße

The screenshot displays the SBTWAN - Neues Projekt application window. The main area is titled 'Komplexität' and shows a complexity analysis of a code snippet. On the left, a project tree lists various AST nodes and methods, with 'linkNodes(22,1)' highlighted. The central part of the window shows a control flow graph (CFG) with nodes for 'while', 'switch', 'case', and 'jump'. The 'while' node is connected to a 'switch' node, which branches into five 'case' nodes. Each 'case' node leads to a 'jump' node, which then loops back to the 'while' node. On the right, the corresponding Java code is displayed, showing a loop with a switch statement and several break statements.

```
    }  
    // Breaks  
  
    it = tableOfBreaks.keySet().iterator();  
    while (it.hasNext()){  
        jn = (ASTJumpNode) it.next();  
        dn = (ASTNode) tableOfBreaks.get(jn);  
        switch (dn.type())  
        {  
            case ASTNode.SWITCH:  
                jn.setTarget(((ASTSwitchNode) dn).getEndN  
                break;  
            case ASTNode.DO:  
                jn.setTarget(((ASTDoNode) dn).getEndNode(  
                break;  
            case ASTNode.ITERATION:  
                jn.setTarget(((ASTIterationNode) dn).getE  
                break;  
            case ASTNode.TRY:  
                jn.setTarget(((ASTTryNode) dn).getFinally  
                break;  
            default:  
                System.out  
  
                .println("Error: JumpManager - couldn'  
        }  
    }  
}
```

Konfiguration der Instrumentierung



Darstellung der Quellcode-Überdeckung

The screenshot displays the SBTWAN - Neues Projekt interface. The main window is titled "Überdeckung" and is divided into four panes:

- Project Tree:** Shows a folder named "Fibonacci" containing files "fib(2,1)" and "main(2,1)".
- Control Flow Graph (CFG):** A flowchart with red nodes and arrows. It starts with a "Start" node, followed by a node, then an "if" node. The "if" node branches into two "Jump" nodes, which then merge into a final node.
- Source Code:** Displays the following Java code:

```
public static long fib(int n) {  
    if (n <= 1) return n;  
    else return fib(n-1) + fib(n-2);  
}
```
- Testfälle:** An empty pane for test cases.

The status bar at the bottom shows the following coverage metrics:

C0 :Nein (0,0%)	C1 :Nein (0,0%)	BI :Nein (0,0%)	C2 :Nein (□%)	C3 :Nein (□%)	MMDC :Nein (0,0%)	MCDC :Nein (□%)
-----------------	-----------------	-----------------	---------------	---------------	-------------------	-----------------

Below the status bar, the text "Überdeckungsansicht geöffnet" is displayed.

Verbesserungsmöglichkeiten

- ⑥ Sicherheit: nicht alle Exceptions werden behandelt
- ⑥ Information: Definition der Komplexitätsmaße für Klassen
- ⑥ Nutzerfreundlichkeit: Projektmanagement intuitiver gestalten
- ⑥ bisher von Hand: Testfalldefinition (XML)

sbtwan und ATOSj

- ⑥ inwieweit ist Kooperation sinnvoll?
- ⑥ Kontra: Überdeckungsmaße jenseits von Methodenüberdeckung sind für den oberflächenbas. Regressionstest irrelevant, da zu genau (Tegos)
- ⑥ Pro: Instrumentierung lässt sich sehr variabel einstellen, nicht vollständige Überdeckung ist das Ziel sonder Information über Aussagequalität des Tests
- ⑥ Pro: Einsatz in der Lehre wäre mit aufeinander abgestimmten Programmen in jedem Fall praktischer
- ⑥ aber beide Werkzeuge sollten in einer autonom funktionstüchtigen Version vorhanden sein!

Verknüpfung mit ATOSj

- ⑥ mögliche Verknüpfungsszenarien:
 - △ Integration der vollständigen Funktionalität in ATOSj
 - △ Darstellung einer abgespeckten Überdeckungsmessung in ATOSj
 - △ lediglich Austausch über Testfälle

- ⑥ Vorschläge für weiterführende Arbeiten durch Herrn Seffler:
 - △ flexiblere und informationsträchtigere Darstellung der Überdeckungsmaße, des KFG und des Quellcodes
 - △ variablerer Umgang mit Projektdateien und Testfallmengen
 - △ Auswertung der Informationen aus den Testfällen zu Profiling-Zwecken
 - △ Konstruktion von Aufrufgraphen
 - △ Unterstützung bei der Optimierung von Testfallmengen (z.B. für MC/DC)