

PSE - Abschlusspräsentation

Automatisches Prüfen der Korrektheit von Programmen

Simon Bischof, Jan Haag, Adrian Hermann, Lin Jin, Tobias Schlumberger, Matthias Schnetz

Institut für Theoretische Informatik – Anwendungsorientierte formale Verifikation



Motivation

- Zunehmende Bedeutung der Qualitätssicherung in der Softwareentwicklung
- Formale Verifikation als Teil der Qualitätssicherung

Ziel: Formaler Beweis, dass Programm seine Spezifikation erfüllt

- Anwendungsgebiete
 - Hardwaredesign
 - Systeme die sehr zuverlässig arbeiten müssen

Aufgabe

■ Definition einer Sprache

➡ While – Sprache:

- Nur einfache Sprachkonstrukte
- Ergänzung durch Annotationssprache
- Notation in Form einer Grammatik

■ Übersetzen von Programmen

➡ Parser:

- Erzeugung einer Baumrepräsentation (AST) mithilfe der Grammatik
- Im Fehlerfall: Detaillierte Rückmeldung

Aufgabe

■ Ausführen von Programmen



Interpreter:

- Ausführen der Anweisungen in Programmen
- Rückgabe des Zustands an den Benutzer



Run-time-checker:

- Prüfung des Typs bei Variablen (Type-checker)
- Überwachung von Unterbrechungsbedingungen

Aufgabe

■ Beweisen der Korrektheit von Programmen

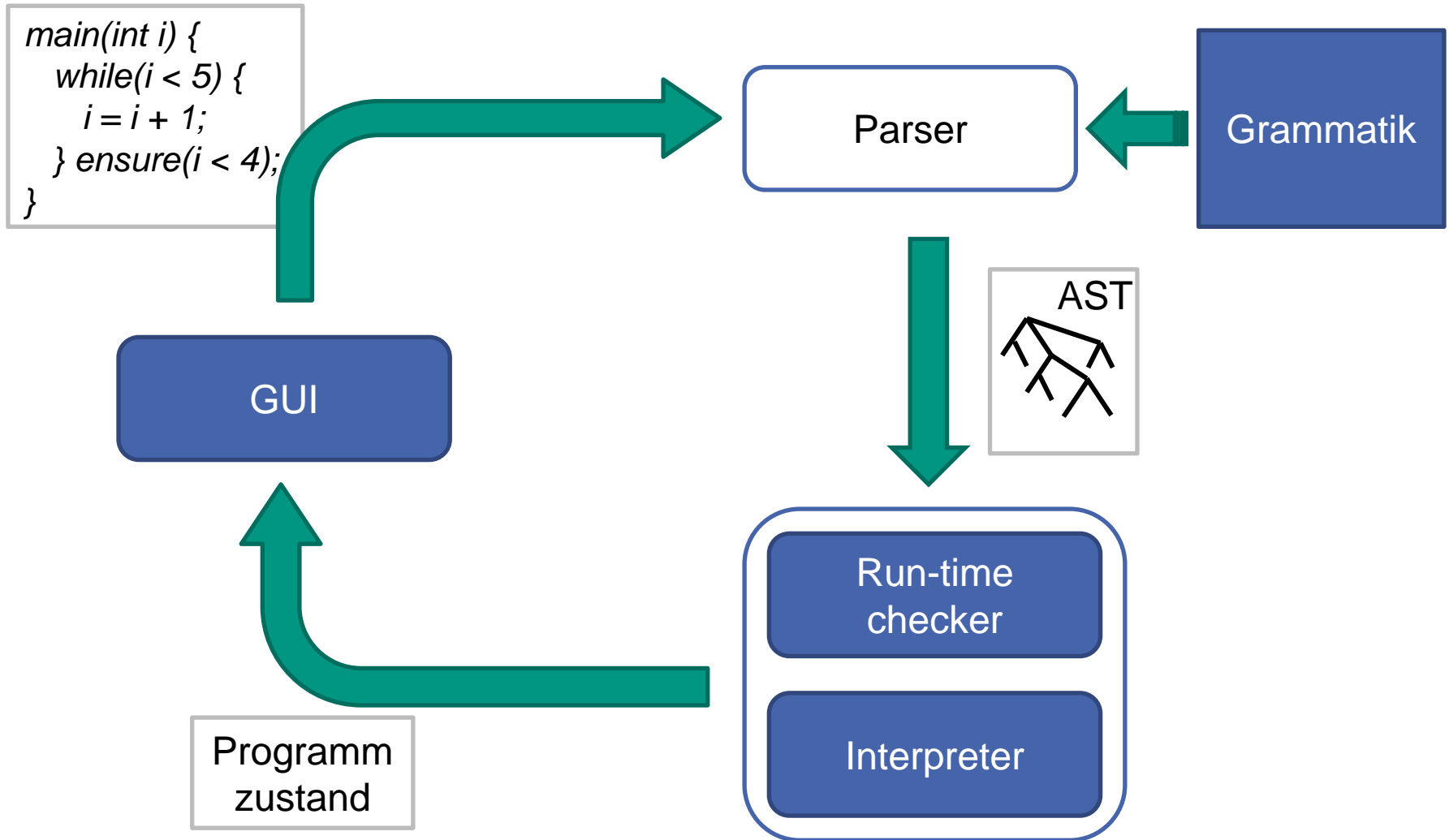


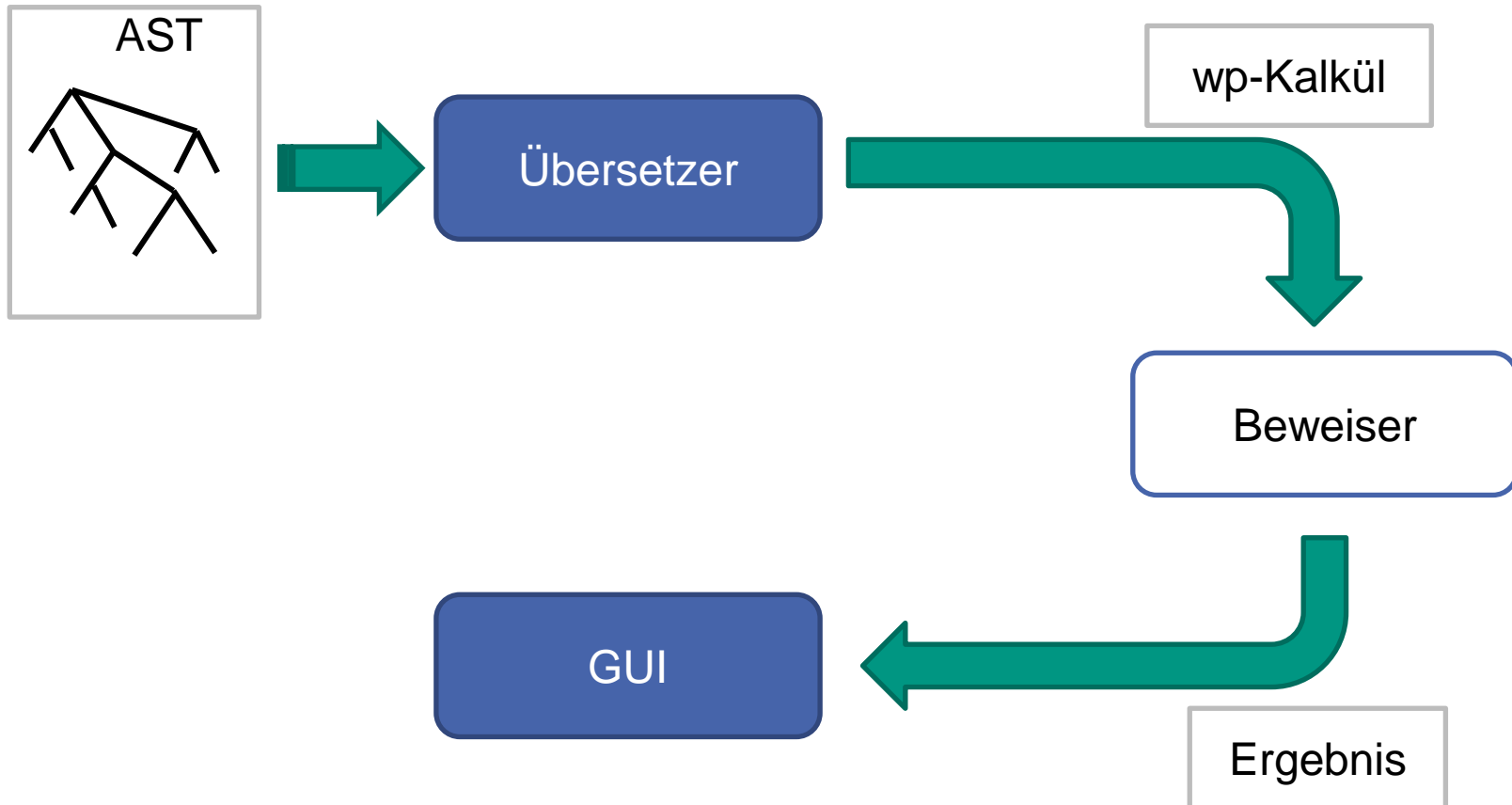
Beweiser:

- Übersetzung des AST in beweisbare Formel
- Beweis durch externen Beweiser
- Detaillierte Rückmeldung an den Benutzer

■ Entwicklung einer grafischen Oberfläche (GUI)

Parser & Interpreter





Formale Verifikation

Zur Verifikation eines Programmes sind zwei Schritte nötig:

- ① Ziel des Programms klarmachen
 - Was ist der Sinn dieses Programms?
 - Welches Endergebnis erwarte ich?
 - Wie soll dieses Ergebnis erreicht werden?

Formale Verifikation

Zur Verifikation eines Programmes sind zwei Schritte nötig:

- ① Ziel des Programms klarmachen
 - Was ist der Sinn dieses Programms?
 - Welches Endergebnis erwarte ich?
 - Wie soll dieses Ergebnis erreicht werden?

- ② Das Programm um beweisbare Annotationen erweitern
 - Was gilt für die Variablen?
 - Wie lauten geeignete Invarianten für Schleifen?

Formale Verifikation

- 1 Ziel des Programms klarmachen
 - Was ist der Sinn dieses Programms?
Berechnung der Summe von 1 bis n
 - Welches Endergebnis erwarte ich?
 - Wie soll dieses Ergebnis erreicht werden?

```
main(int n) {  
    int sum = 0;  
    int i = 0;  
    while (i < n) {  
        i = i + 1;  
        sum = sum + i;  
    }  
}
```

Formale Verifikation

1 Ziel des Programms klarmachen

- Was ist der Sinn dieses Programms?
- Welches Endergebnis erwarte ich?

Die Variable `sum` enthält die Summe von 1 bis `n`

- Wie soll dieses Ergebnis erreicht werden?

```
main(int n) {  
    int sum = 0;  
    int i = 0;  
    while (i < n) {  
        i = i + 1;  
        sum = sum + i;  
    }  
}
```

Formale Verifikation

1 Ziel des Programms klarmachen

- Was ist der Sinn dieses Programms?
- Welches Endergebnis erwarte ich?
- Wie soll dieses Ergebnis erreicht werden?

n Schleifendurchläufe zur schrittweisen Berechnung

```
main(int n) {  
    int sum = 0;  
    int i = 0;  
    while (i < n) {  
        i = i + 1;  
        sum = sum + i;  
    }  
}
```

Formale Verifikation

- 2 Das Programm um beweisbare Annotationen erweitern
- Was gilt für die Variablen?
 - Wie lauten geeignete Invarianten für Schleifen?

```
...  
int sum = 0;  
int i = 0;  
while (i < n)  
  invariant {  
    i <= n;  
    sum == i*(i+1)/2;  
  }  
  {  
    i = i + 1;  
    sum = sum + i;  
  } ensure sum == n*(n+1)/2;  
...
```

Zahlen, Daten, Fakten

- 6 Entwickler
- 17.000 LOC
 - 100 Klassen
 - 15 Pakete
- Lauffähig unter:
 - Windows XP & Windows 7
 - Linux
 - Mac OS X
- Beweisbare Programme:
 - Berechnung der Summe von 1 bis n
 - Russische Multiplikation
 - ...

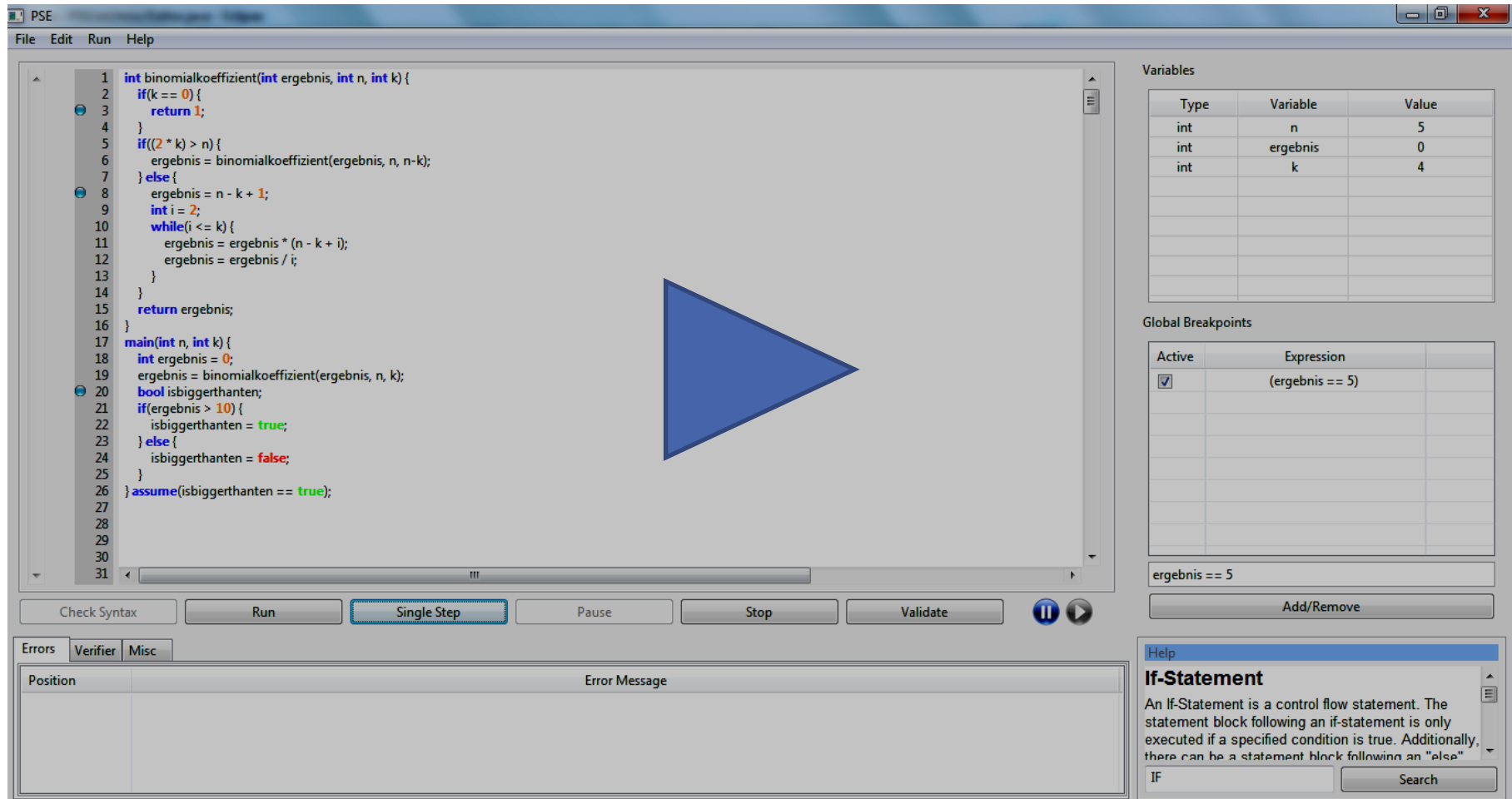
9.500 Zeilen
Code

3.500 Zeilen
generierter Code

3.500 Zeilen
Code aus der QS

800 Zeilen
Dokumentation

Tool Demonstration



Code:

```
1 int binomialkoeffizient(int ergebnis, int n, int k) {
2   if(k == 0) {
3     return 1;
4   }
5   if((2 * k) > n) {
6     ergebnis = binomialkoeffizient(ergebnis, n, n-k);
7   } else {
8     ergebnis = n - k + 1;
9     int i = 2;
10    while(i <= k) {
11      ergebnis = ergebnis * (n - k + i);
12      ergebnis = ergebnis / i;
13    }
14    return ergebnis;
15  }
16 }
17 main(int n, int k) {
18   int ergebnis = 0;
19   ergebnis = binomialkoeffizient(ergebnis, n, k);
20   bool isbiggerthanten;
21   if(ergebnis > 10) {
22     isbiggerthanten = true;
23   } else {
24     isbiggerthanten = false;
25   }
26   assume(isbiggerthanten == true);
27 }
28
29
30
31
```

Variables:

Type	Variable	Value
int	n	5
int	ergebnis	0
int	k	4

Global Breakpoints:

Active	Expression
<input checked="" type="checkbox"/>	(ergebnis == 5)

ergebnis == 5

Help: If-Statement

An If-Statement is a control flow statement. The statement block following an if-statement is only executed if a specified condition is true. Additionally, there can be a statement block following an "else".

IF Search

We
Prove

Anforderungen

- While-Sprache
 - Zuweisungen, Bedingte Anweisungen, Schleifen ✓
 - Arrays ✓
 - Turing-Vollständigkeit ✓
 - Methodenaufrufe ✓
- Annotationssprache
- Run-time-checker
- Interpreter
- Beweiser
- GUI

Anforderungen

- While-Sprache
- Annotationssprache
 - **Syntax und Semantik aus Programmiersprache übernommen** ✓
 - **Zusicherungen erlauben Aussagen der Prädikatenlogik** ✓
 - **Grundlegende Annotationen: assert, assume** ✓
 - Angabe verschiedener Vor-/Nachbedingungen ✓
 - *Weitere Annotationen: invariant, ensure* ✓
- Interpreter
- Run-time-checker
- Beweiser
- GUI

Anforderungen

- While-Sprache
- Annotationssprache
- Interpreter
 - **Schrittweise Ausführung eines Programms** ✓
 - **Inspektion des aktuellen Programmzustands** ✓
 - Benutzerdefinierte Ausdrücke über Programmzustände ✗
 - Änderung des Programmzustandes während der Ausführung ✗
 - *Tests mit zufälligen Eingabeparametern* ✓
- Run-time-checker
- Beweiser
- GUI

Anforderungen

- While-Sprache
- Annotationssprache
- Interpreter
- Run-time-checker
 - **Auswertung der im Programm eingebetteten Annotationen** ✓
 - **Rückmeldung im Fehlerfall an GUI** ✓
 - Auswertung von Formeln mit Quantoren über eingeschränkten Bereich ✓
 - Auswertung von Formeln mit Quantoren mit einem Beweiser ✗
 - *Breakpoints auf Zeilen und Bedingungen* ✓
- Beweiser
- GUI

Anforderungen

- While-Sprache
- Annotationssprache
- Interpreter
- Run-time-checker
- Beweiser
 - **Formale Verifikation mit Hilfe eines Beweisers** ✓
- GUI

Anforderungen

- While-Sprache
- Annotationssprache
- Interpreter
- Run-time-checker
- Beweiser
- GUI
 - **Steuerung der Komponenten, Anzeige von Rückmeldungen** ✓
 - **Sprache: Englisch** ✓
 - Unicode Symbole für logische Operatoren ✗
 - Verwaltung von Beweisverpflichtungen ✗
 - *Ausführliche Dokumentation/Hilfe zur While-Sprache* ✓
 - *Editor Funktionen (Undo, Redo, Cut, Copy, ...)* ✓
 - *Syntax highlighting* ✓

Qualitätssicherung

■ Werkzeuge

- Bug - Tracker
- JUnit, EMMA
- GUI - Testplan

➡ 108 gefundene Bugs

➡ Code Coverage: ca. 90 %

Qualitätssicherung

■ Werkzeuge

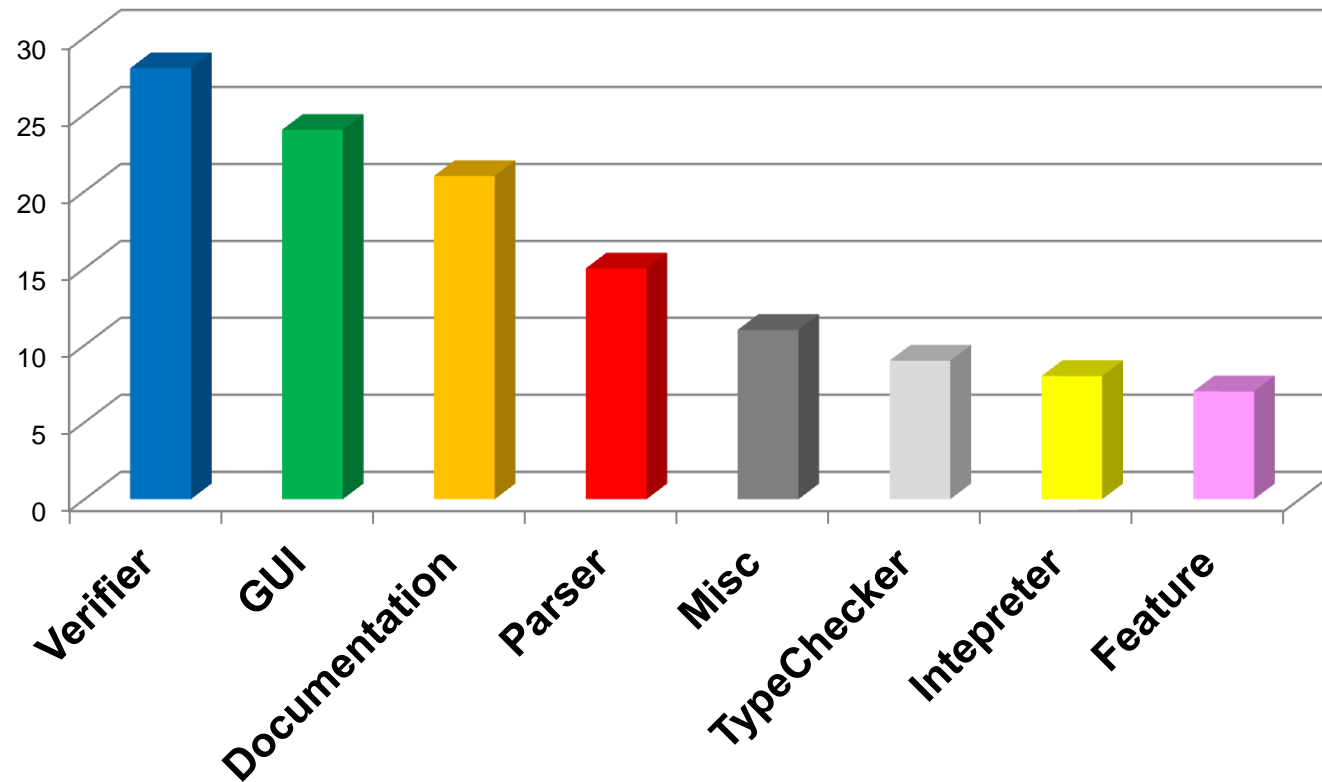
- Bug - Tracker
- JUnit
- GUI - Testplan

➡ 108 gefundene Bugs

➡ Code Coverage: ca. 90 %

Probleme:

- Klare Abgrenzung der einzelnen Module zum Teil schwierig
- Testen von generiertem Code nur teilweise möglich
- Einige Klassen haben internen Zustand



Zukunft von WeProve

- Es existieren andere Tools mit größerem Funktionsumfang und Entwicklerteams mit mehr Erfahrung/Mitteln

- KeY
- Isabelle/HOL



- ➡ Tool ist ein guter Einstieg in den Themenbereich der formalen Verifikation (v.a. auch für Studenten)
- ➡ Lizenzierung unter BSD Lizenz

Herausforderungen & Erfahrungen

■ Herausforderungen

- Themengebiet der formalen Verifikation sehr abstrakt / komplex
- Grammatikerstellung für Parser erfordert Behandlung vieler Spezialfälle
- In manchen Phasen hoher Zeitdruck

■ Erfahrungen

- Fehler / ungelöste Probleme des Entwurfs wirken sich sehr stark auf die Implementierung aus
- Ablauf der Implementierung weicht von Zeitplanung ab
- Qualitätssicherung bringt mehr Fehler zum Vorschein als erwartet
- Beteiligung/Engagement schwankt zwischen wenig und viel