ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming

Seminar zu “Machine Learning in Software Engineering”

Felix Groß, Paul Groß, David Riemer

Gliederung

1. Einleitung
2. Machine Learning
3. Nutzung des Softwaresystems
4. Genetische Algorithmen
5. Funktionsweise von ARJA
6. Beispiel einer Reparatur
7. Forschungsfragen
8. Fazit

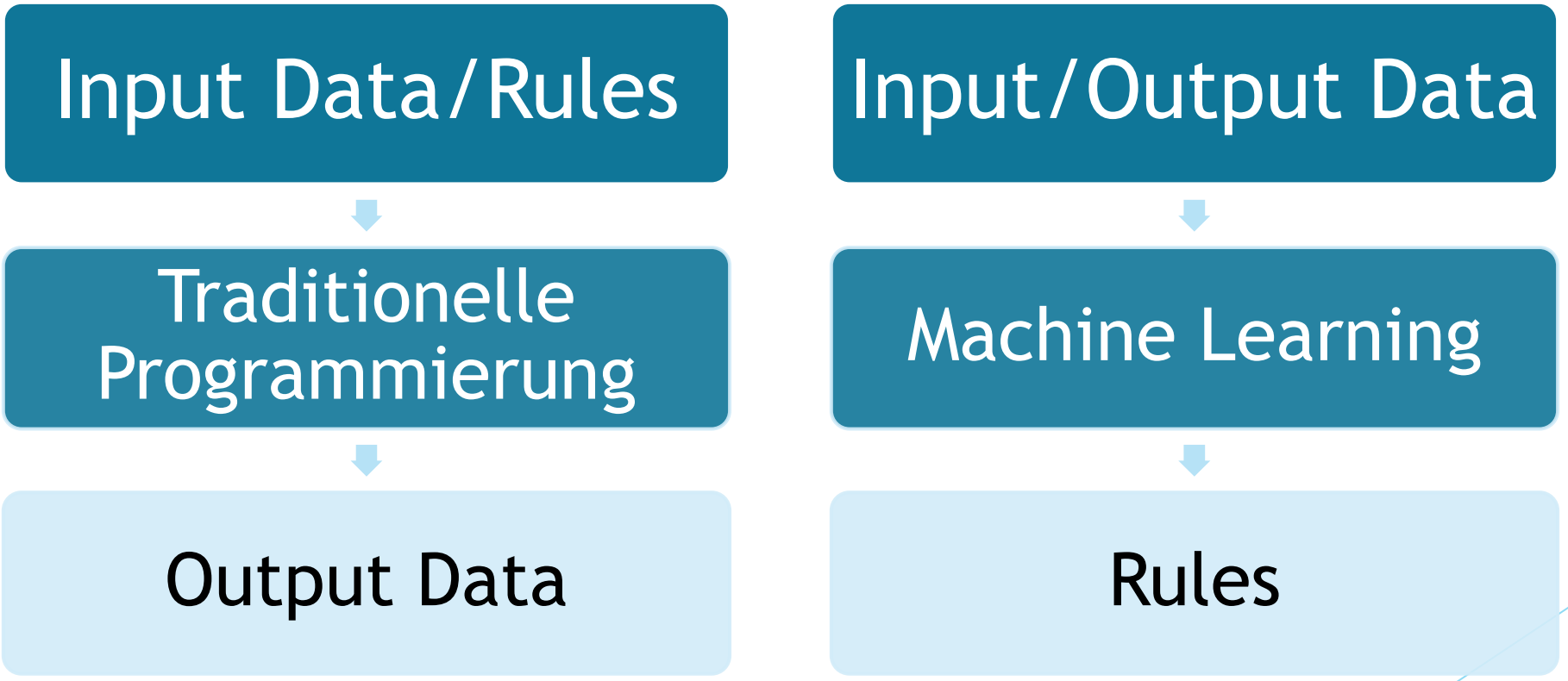
Einleitung

- Arja ist ein Tool zur automatischen Reparatur von Programmen
- Benötigt Source-Code und Testfälle des Programms
- Verwendet genetische Programmierung und den vorhanden Code zur Reparatur
- Erzeugt Patches zur Fehlerbehebung

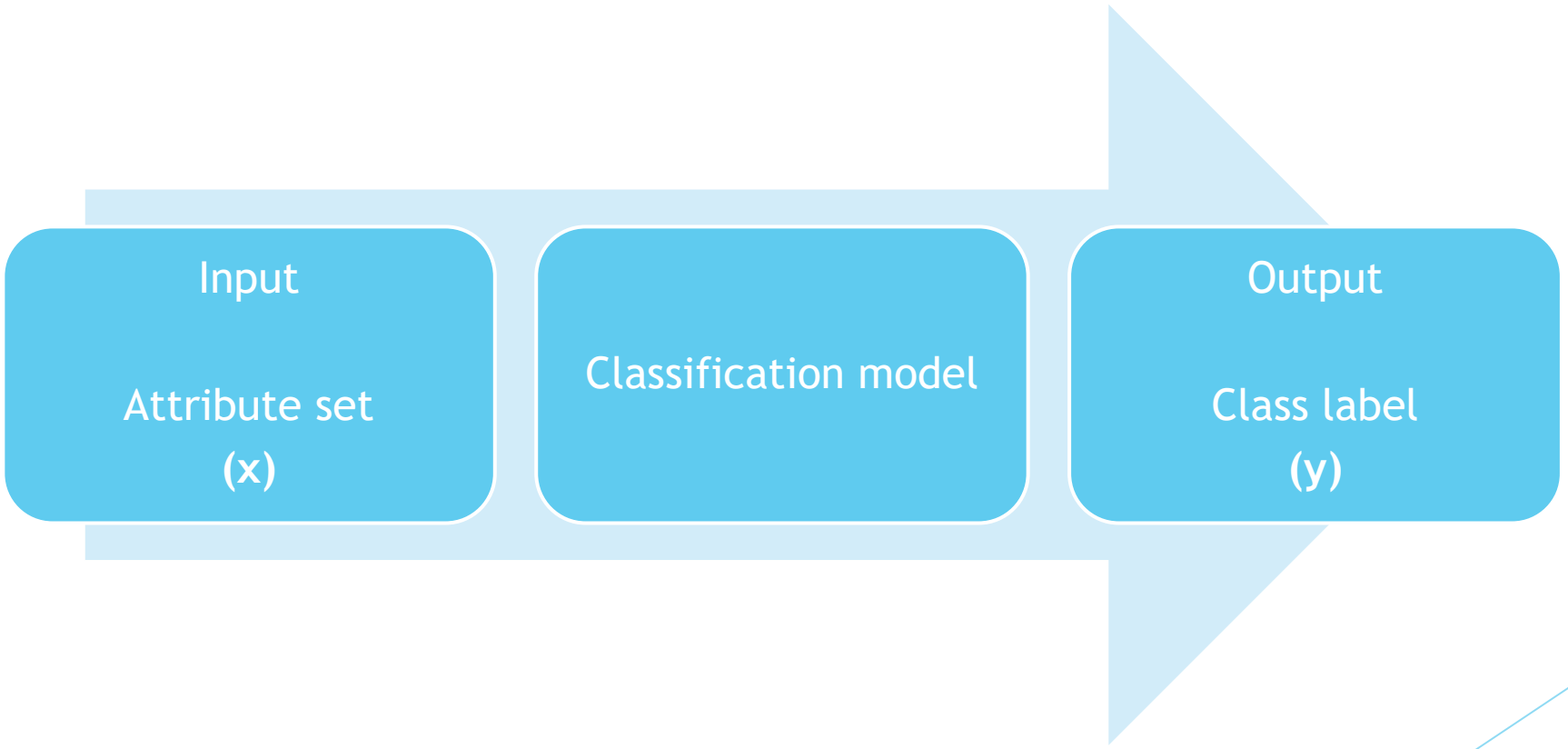
Gliederung

1. Einleitung
2. Machine Learning
3. Nutzung des Softwaresystems
4. Genetische Algorithmen
5. Funktionsweise von ARJA
6. Beispiel einer Reparatur
7. Forschungsfragen
8. Fazit

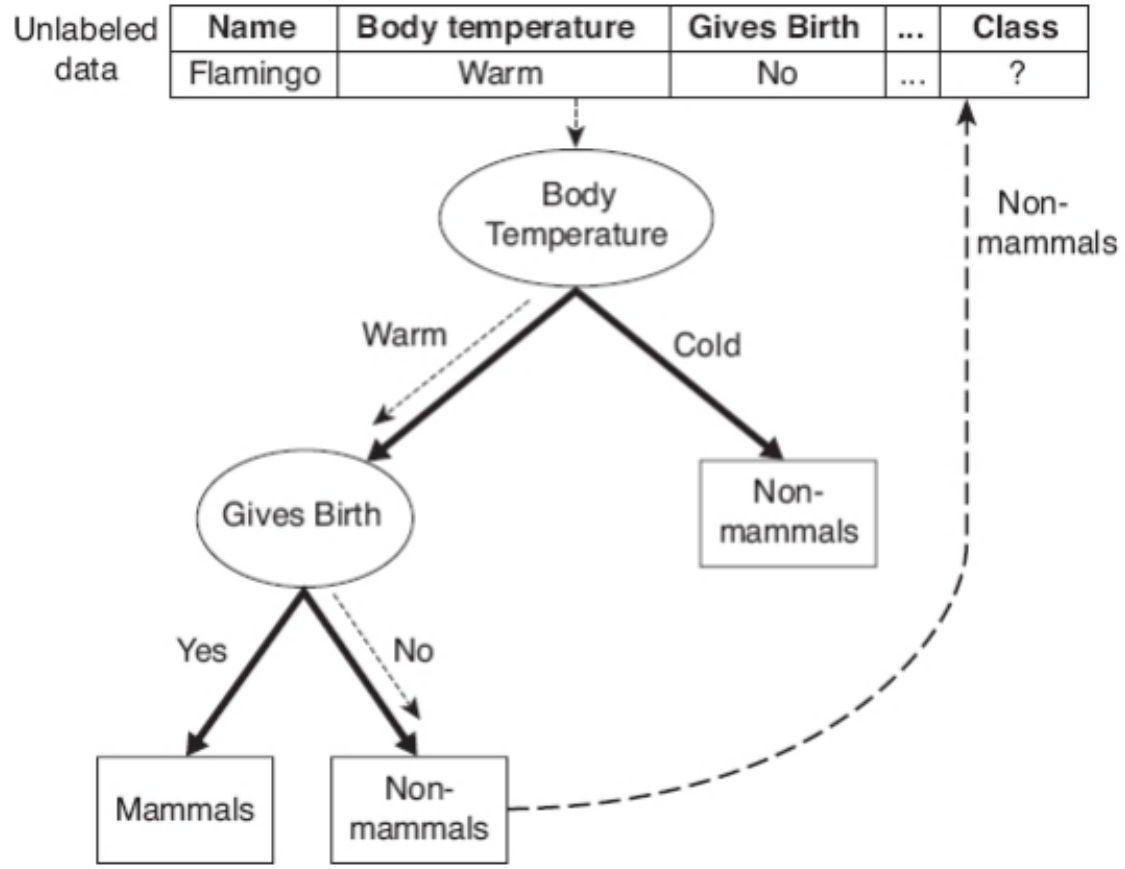
Traditionelle Programmierung vs. Machine Learning



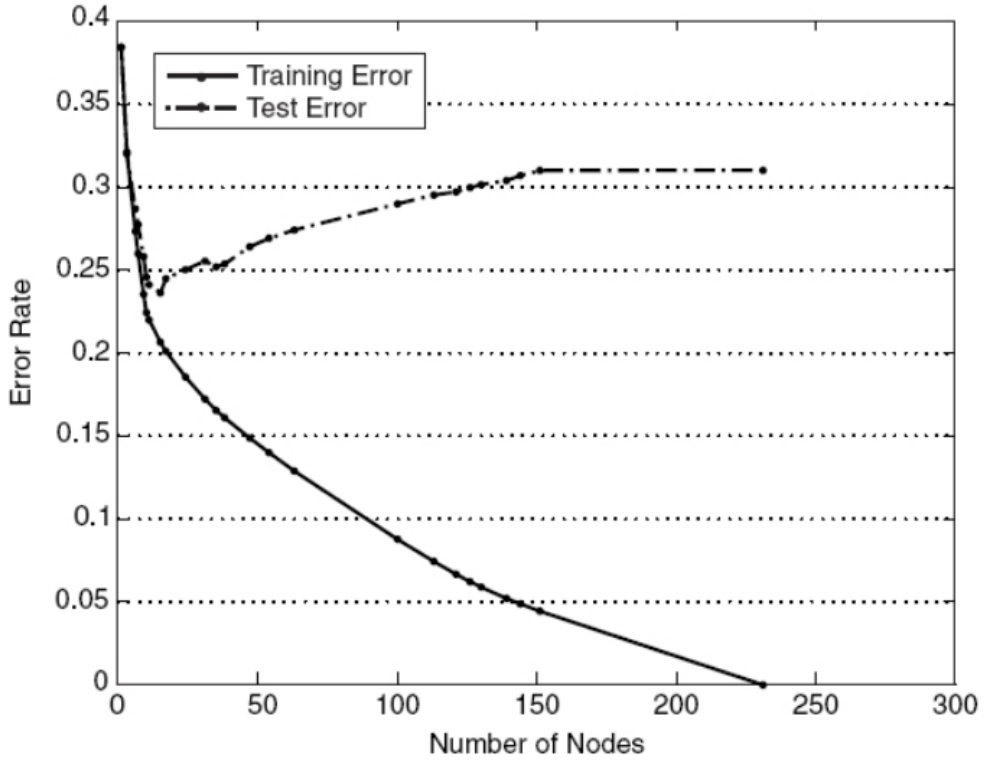
Supervised Learning



Supervised Learning



Overfitting



Gliederung

1. Einleitung
2. Machine Learning
3. Nutzung des Softwaresystems
4. Genetische Algorithmen
5. Funktionsweise von ARJA
6. Beispiel einer Reparatur
7. Forschungsfragen
8. Fazit

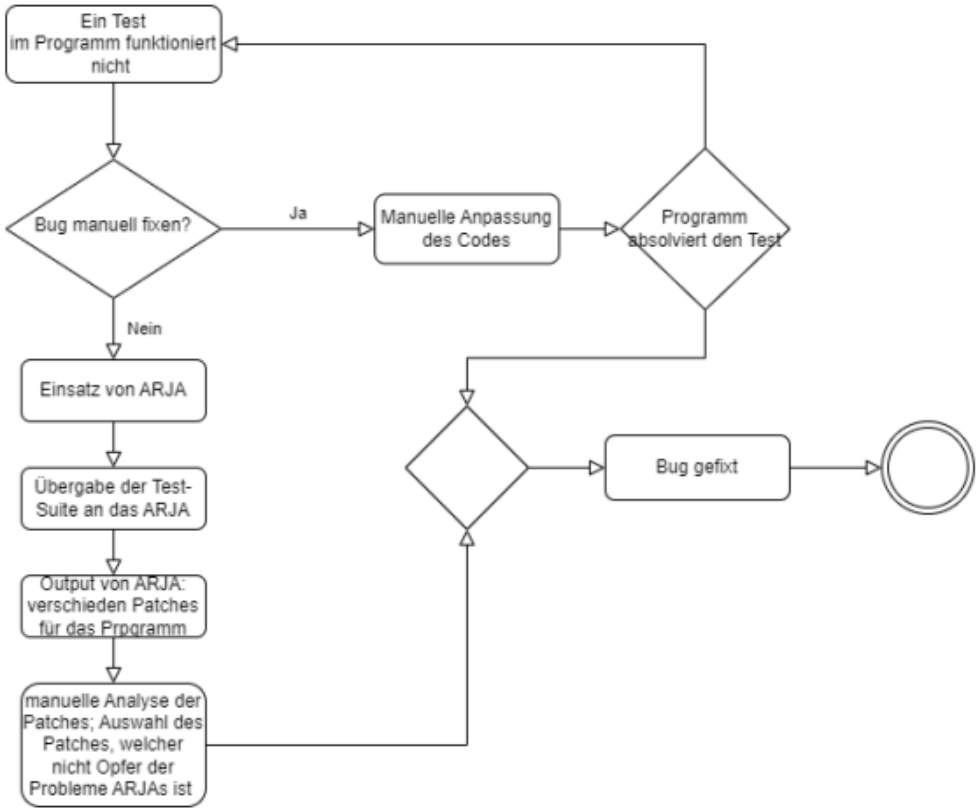
Nutzung des Softwaresystems

- Wann ein sollte man ARJA in Softwareentwicklungsprozessen einsetzen?
- 1. Entscheidung anhand äußerlicher Sachverhalte
  - Zeitmangel
  - Ressourcenbeschränkung

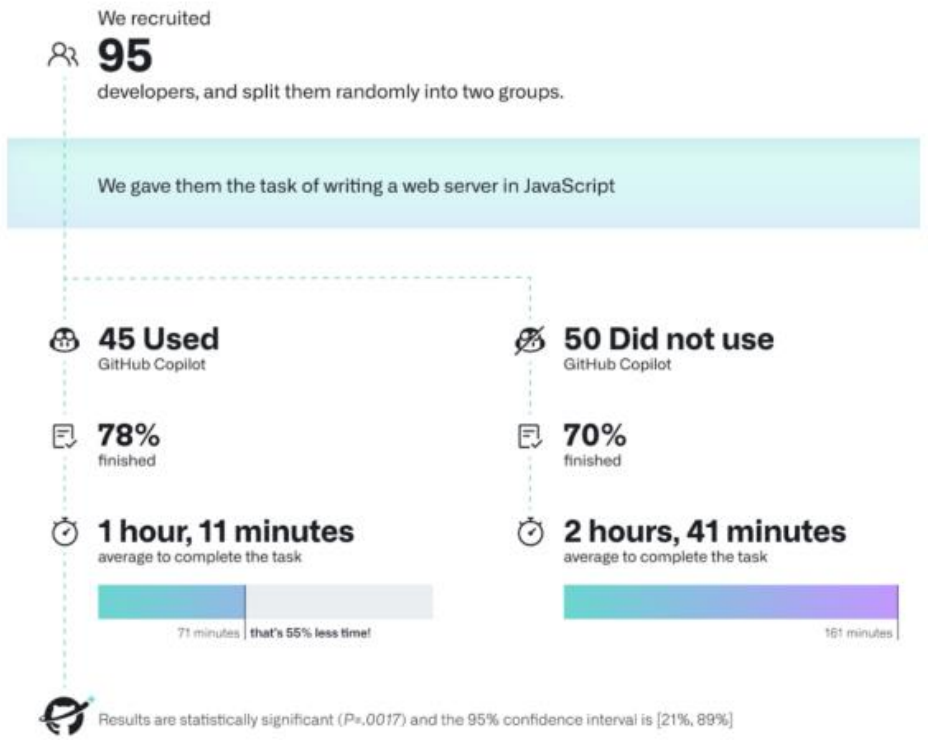
Nutzung des Softwaresystems

- Wann ein sollte man ARJA in Softwareentwicklungsprozessen einsetzen?
- 2. Entscheidung anhand des Codes
  - Automated Repair Program (ARP) kann in großen Codebasen meist schneller Fehler erkennen
  - Wartung von Legacy Code
  - ARPs immer noch fehleranfällig -> Entscheidung liegt bei dem Entwickler

Nutzung des Softwaresystems



Ausblick


We recruited
95
developers, and split them randomly into two groups.
We gave them the task of writing a web server in JavaScript
45 Used
GitHub Copilot
78%
finished
1 hour, 11 minutes
average to complete the task
71 minutes
that's 55% less time!
50 Did not use
GitHub Copilot
70%
finished
2 hours, 41 minutes
average to complete the task
161 minutes
Results are statistically significant (P=.0017) and the 95% confidence interval is [21%, 89%]

Gliederung

1. Einleitung
2. Machine Learning
3. Nutzung des Softwaresystems
4. Genetische Algorithmen
5. Funktionsweise von ARJA
6. Beispiel einer Reparatur
7. Forschungsfragen
8. Fazit

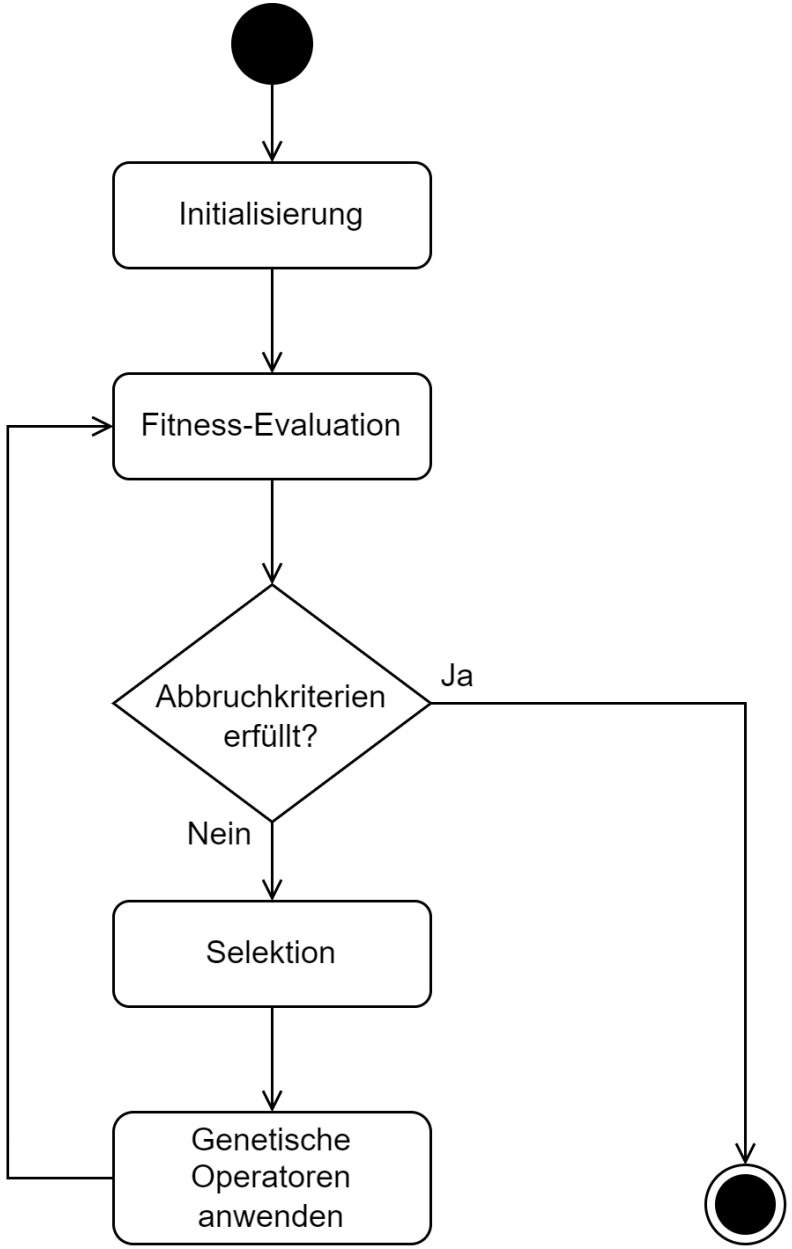
Genetische Algorithmen

- Einführung in genetische Algorithmen
- Allgemeine Implementierungsdetails
- Genetische Programmierung

Einführung

- Einsatz in komplexen Optimierungsproblemen
- Keine anderen Algorithmen vorhanden, um Problem in akzeptabler Laufzeit zu lösen
- Näherungslösung finden

Ablauf des Algorithmus



Initialisierung

- Menge an Lösungen erzeugen
- Diese Menge wird Population genannt
- Die Elemente der Menge werden Individuen genannt

Fitness-Evaluation

- Bewertung der Individuen anhand einer Fitnessfunktion
- Die Fitness beschreibt die Qualität einer Lösung
- Numerischer Rückgabewert ermöglicht leichten Vergleich

Fitnessfunktion im Detail

- Bewertung eines Individuums ist problemabhängig
- Gewichtung der Kriterien oder gleichmäßige Verteilung
- Empfindliche Änderungen einer Lösung sollen zu einer signifikant geänderten Fitness führen

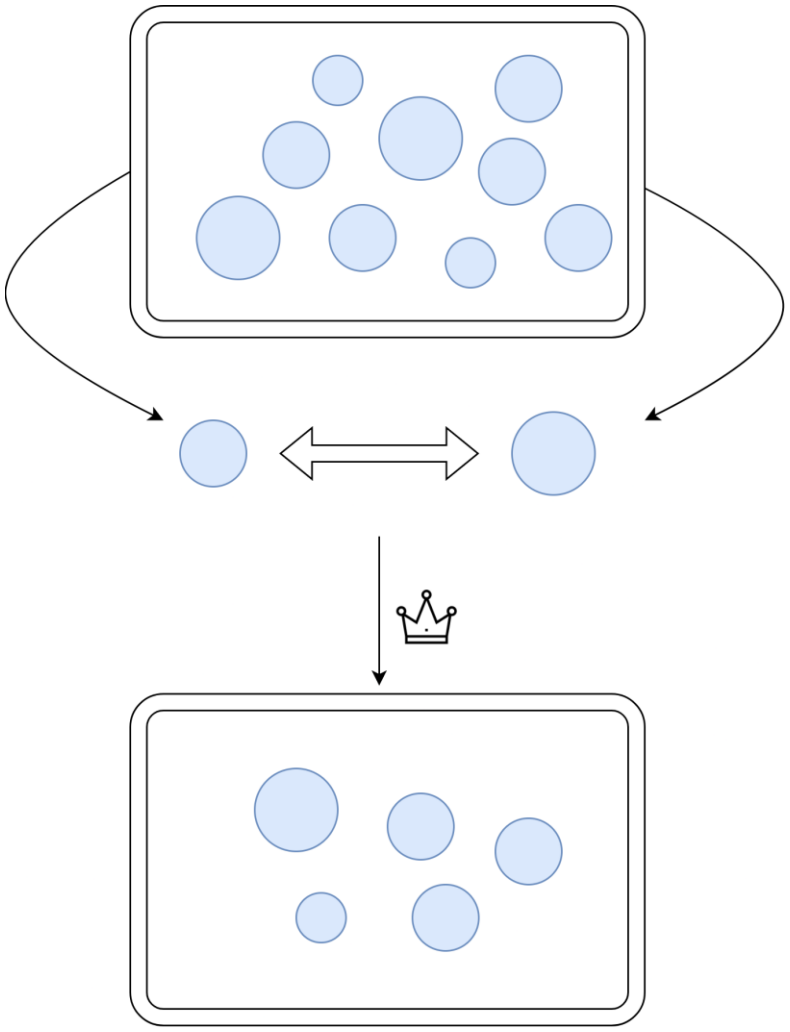
Erstellung von neuen Individuen

- Selektion
- Genetische Operatoren
  - Mutation
  - Crossover

Selektion

- Nur Individuen mit einer akzeptablen Fitness sollen beibehalten werden
- Turnierselektion
- Vergleiche jeweils die Fitness zweier Individuen
  - Das „Gewinner“-Individuum wird beibehalten
  - Der „Verlierer“-Individuum wird verworfen

Turnierselektion



Turnierselektion

- Halbierung der Population
- Auch vergleichsweise schwächere Individuen können vorrücken
- Problem: Individuen mit hohem Fitnesswert können verloren gehen
- Lösung: Bestimme in jeder Population die Elite

Genetische Operatoren

- Selektierte Individuen sind die Grundlage für die nächste Population und werden nun Eltern genannt
- Ziel: Neue Individuen mit erhöhtem Fitnesswert erschaffen
- Kombiniere dafür die Eigenschaften der Eltern

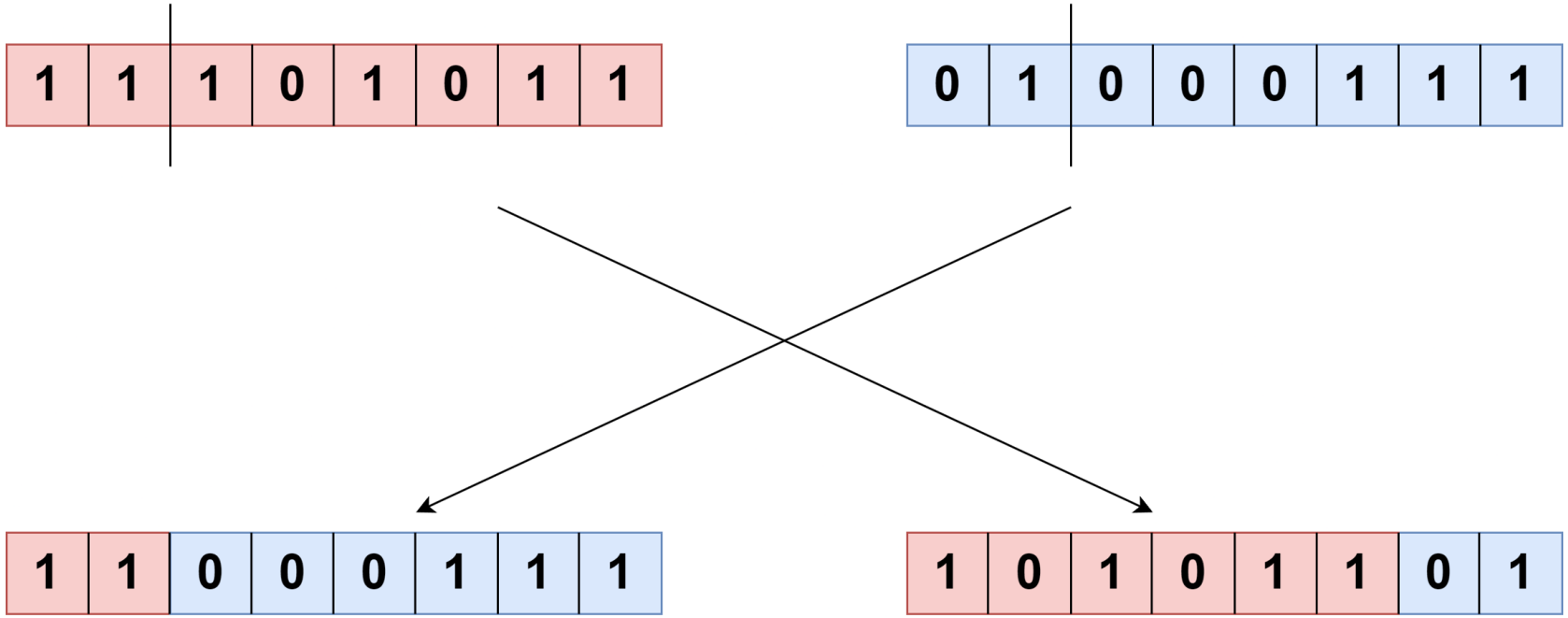
Crossover-Operator

- Rekombination von Eigenschaften der Eltern
- Bilde Eltern-Paare und kombiniere deren Eigenschaften
- Erzeuge neue Lösung mit potentiell gesteigertem Fitnesswert

Single-Point Crossover

- Angenommen Individuum ist als Bitfolge codiert
- Bestimme zufälligen Punkt in der Folge und teile die Folge in zwei Hälften
- Erzeuge zwei neue Individuen durch Rekombination der Hälften

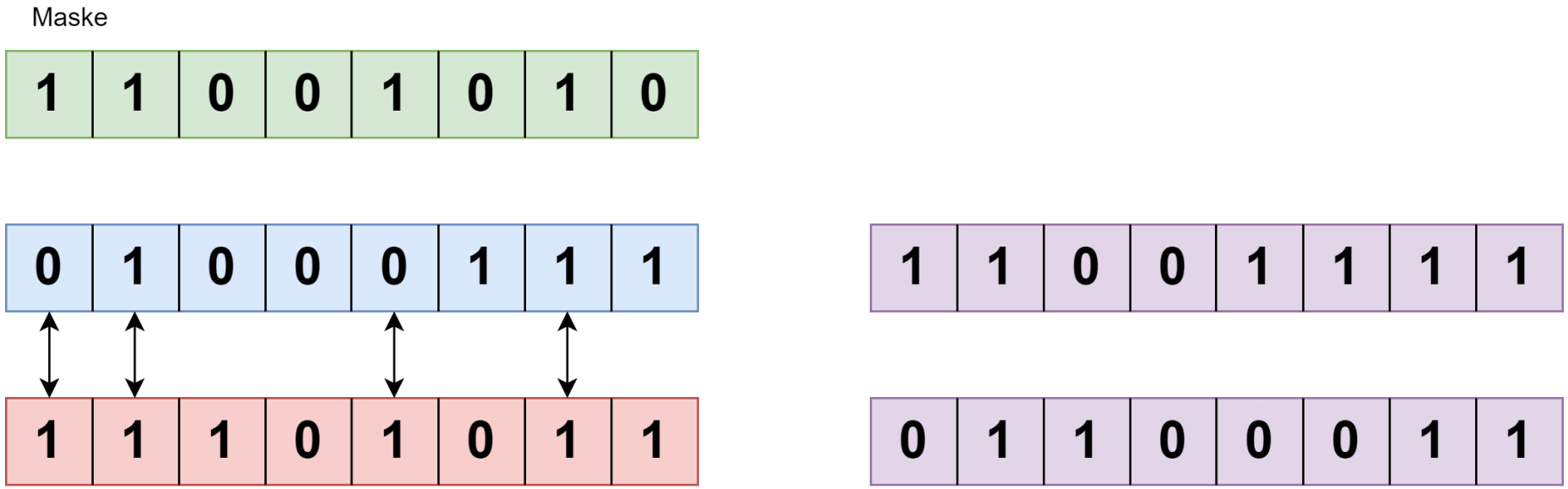
Single Point Crossover



Uniformes Crossover

- Iteriere über eine Bitfolge
- Führe an zufälligen Stellen einen Tausch durch
- Erhalte zwei neue Individuen

Uniformes Crossover



Crossover

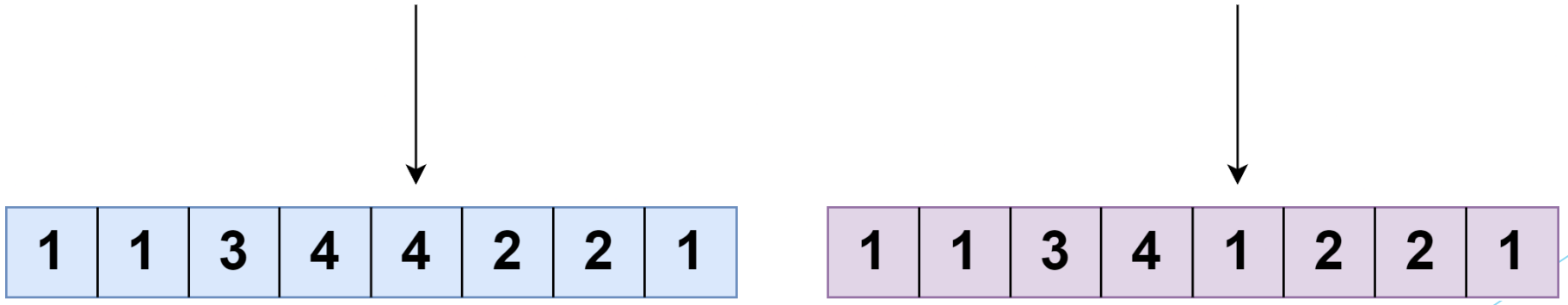
- Crossover verfahren rekombinieren nur die Eigenschaften der vorhandenen Lösungen
- Man möchte jedoch auch Individuen mit neuen Eigenschaften in die Population einführen
- Verwende daher Mutationsoperatoren

Mutationsoperatoren

- Rekombinierte Individuen nochmals bearbeiten
- Verändere zufällig die Eigenschaften einer Lösung
- Arbeiten unabhängig von den Eltern
- Erschaffung neuer Eigenschaften möglich
- Uniforme Mutation und Bit-Flip Mutation

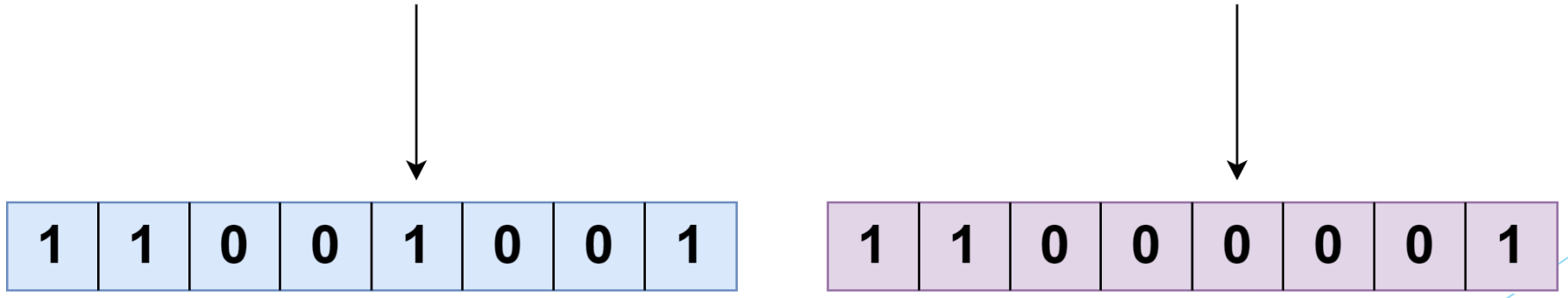
Uniforme Mutation

- Lösung ist als Folge natürlicher Zahlen codiert
- Wähle eine zufällige Zahl aus, welche mutieren soll
- Ersetze diese Zahl durch eine neue, zulässige Zahl



Bit-Flip Mutation

- Im Gegensatz zur uniformen Mutation müssen Individuen hier als Bitfolge codiert sein
- Wähle zufälliges Element der Folge
- Führe einen Bit-Flip durch



Terminierung

- Es kann nicht sichergestellt werden, dass eine optimale Lösung gefunden wird
- Abbruchkriterien:
  - Naiv: Begrenze die Anzahl der Iterationen
  - Untersuche Population auf eine Kandidatenlösung, welche minimale Eigenschaften für eine zulässige Lösung erfüllt
  - Wenn ein Optimum gefunden wird, kann ebenfalls abgebrochen werden

Genetische Programmierung

- Anwendungsform von genetischer Algorithmen
- Jedes Individuum ist ein ausführbares Computerprogramm
- Codierung als hierarchische Datenstruktur, z.B. als Abstract Syntax Tree
- Anwendung genetischer Operatoren

Genetische Programmierung

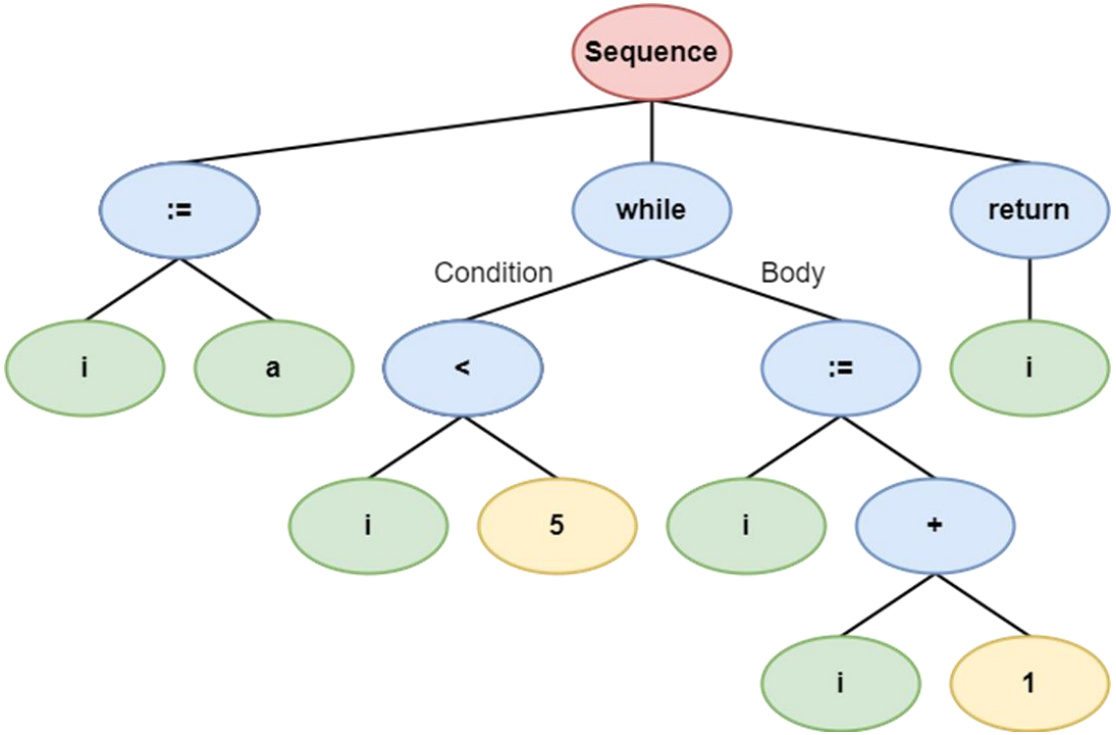
- Optimierungsprobleme mit mehreren Nebenbedingungen
- Mehrere Zielfunktionen müssen minimiert werden
- ARJA bewertet Programmcode nach "Patch size" und "weighted failure rate"
- Finde Lösungen, bei denen ein Wert nicht verbessert werden kann, ohne einen anderen zu verschlechtern

Gliederung

1. Einleitung
2. Machine Learning
3. Nutzung des Softwaresystems
4. Genetische Algorithmen
5. Funktionsweise von ARJA
6. Beispiel einer Reparatur
7. Forschungsfragen
8. Fazit

Abstract Syntax Tree (AST)

- Datenstruktur zur Darstellung des syntaktischen Aufbaus von Quellcode
- Knoten: Operationen, Schleifen, Deklarationen, …
- Blätter: Variablen, Konstanten, …
- Kanten: Beziehungen zwischen den Knoten und Blättern




```
1 i := a
2 while i < 5 do
3 |   i := i + 1
4 end while
5 return i
```

Verwendung von AST in ARJA

- Quellcode wird in AST umgewandelt
- Teile der AST werden dabei in fehlerhafte Statements, Seed-Statements und Ingredient-Statements unterschieden
- Operationen für die Reparatur:
  - Lösche einen Knoten des AST
  - Ersetze einen Knoten des AST mit einem anderen Knoten
  - Füge einen Knoten in den AST ein
- Statements und Reparatur-Operationen werden für die Erstellung von Patches verwendet

Statements

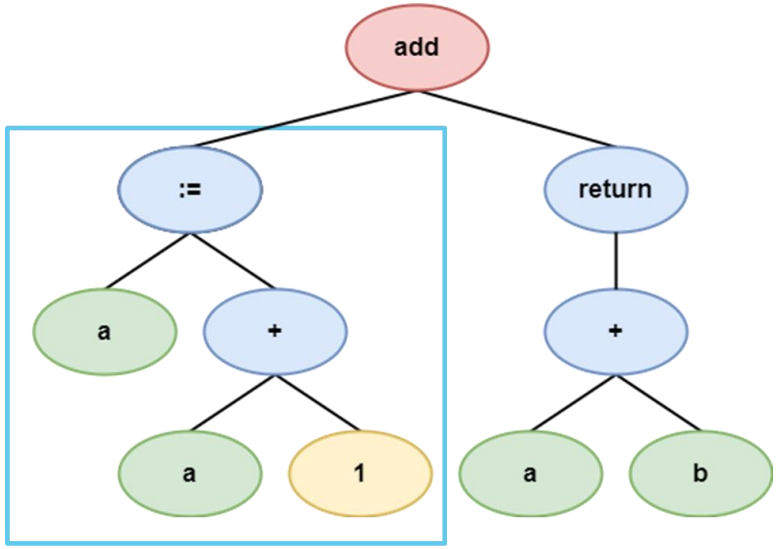
Fehlerhaftes Statement:
Codestelle, die wahrscheinlich für den Fehler im Programm verantwortlich ist

Seed-Statement:
Beliebige Codestelle, die für die Reparatur eines fehlerhaften Statements verwendet werden kann

Ingredient-Statement:
Seed-Statement, das zur Reparatur eines fehlerhaften Statements ausgewählt wird

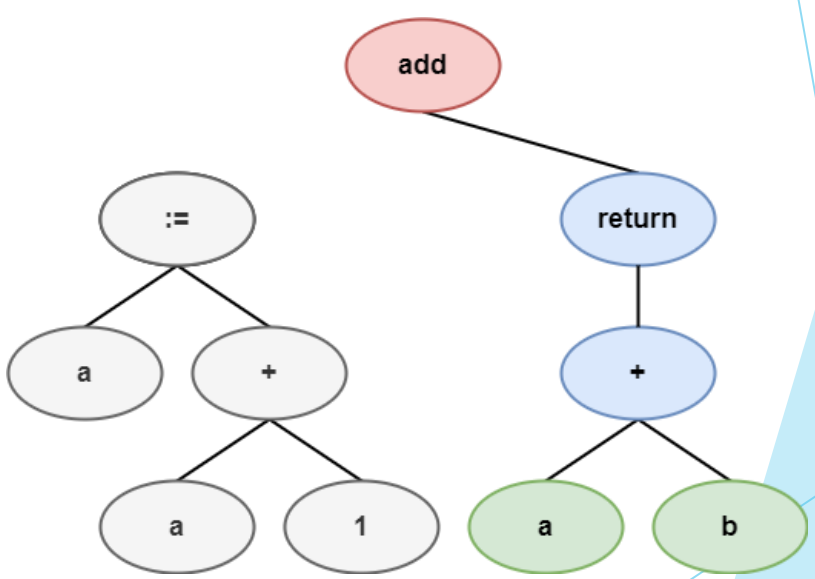
Löschen-Operation

function ADD(a, b)
$a := a + 1$
return $a + b$

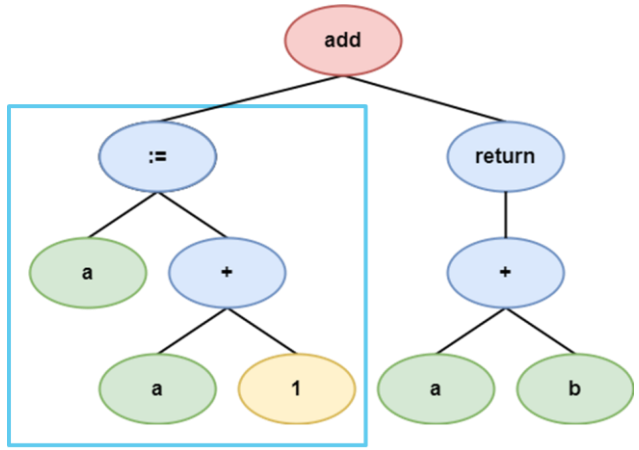


Fehlerhaftes Statement

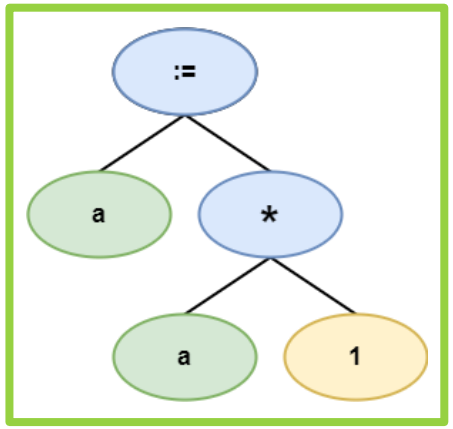
function ADD(a, b)
return $a + b$



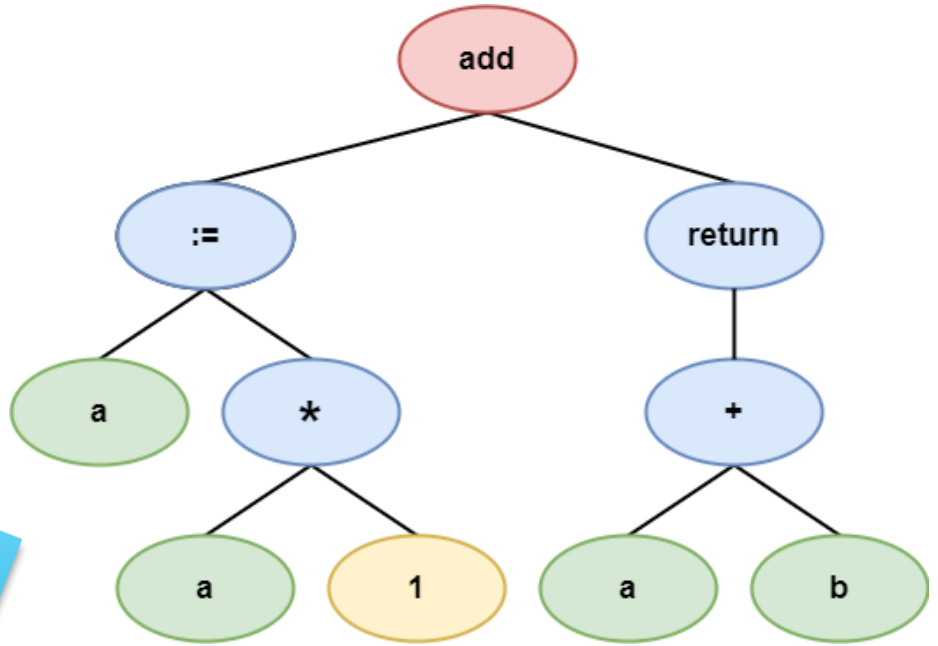
Ersetzen-Operation



Fehlerhaftes Statement

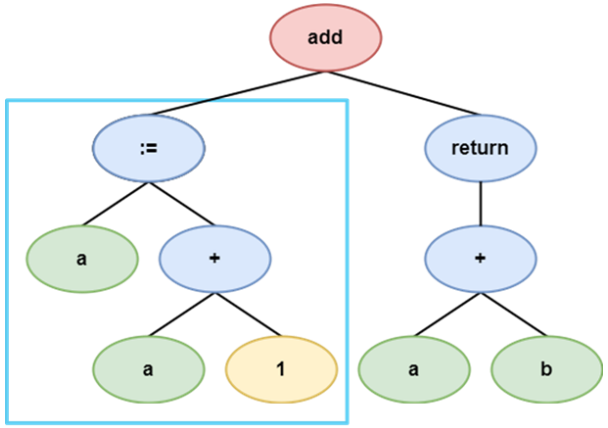


Ingredient Statement

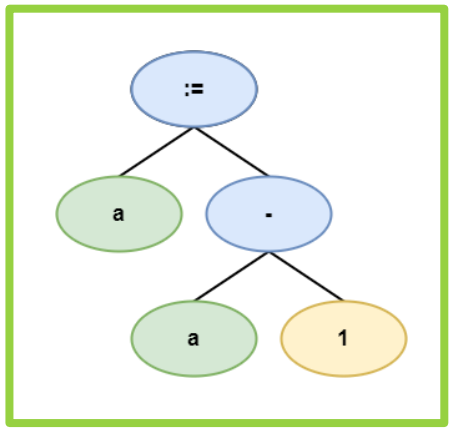


$$\begin{array}{l}\textbf{function } \text{ADD}(a, b) \\ \quad a := a * 1 \\ \quad \textbf{return } a + b\end{array}$$

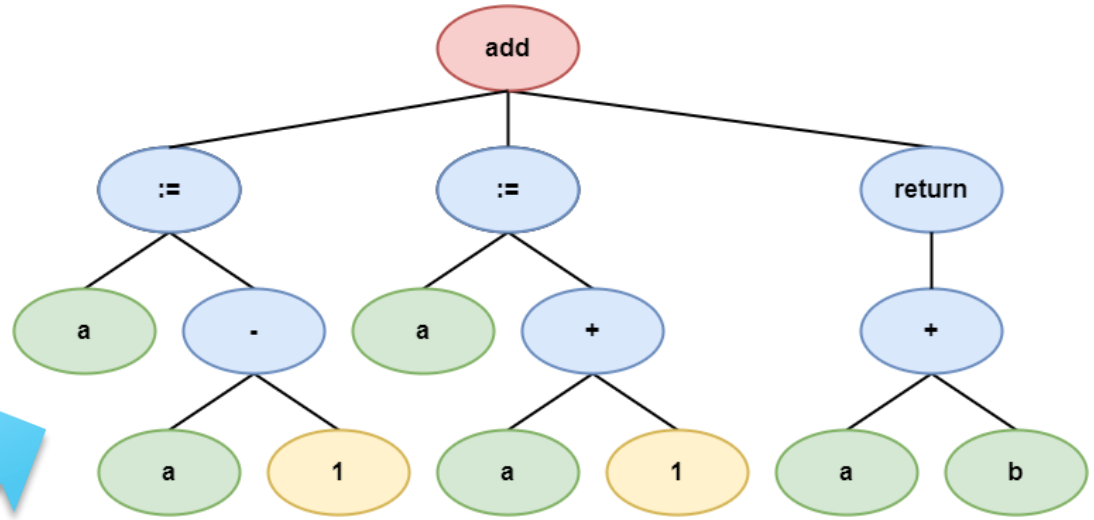
Einfügen-Operation



Fehlerhaftes Statement



Ingredient Statement



function ADD(a, b)

$a := a - 1$

$a := a + 1$

return $a + b$

Regeln für Operationen

Allgemein:
- continue/break/case-Anweisungen nur in passenden Codestellen verwenden
- return/throw-Anweisungen müssen einen passenden Typ haben
- Variablendeklarationen müssen einen passenden Typ haben

Einfügen:
- Füge keine Variablendeklaration vor einer anderen ein
- Füge return/throw Anweisung nur am Ende einer Funktion ein

Löschen:
- Lösche keine Variablendeklarationen
- Lösche keine return/throw Anweisungen

Ersetzen:
- Ersetze kein Statement mit einem Statement das den gleichen AST hat
- Ersetze Variablendeklarationen nur mit anderen Variablendeklarationen
- Ersetze return-Anweisungen nur mit return-Anweisungen

Aufbau eines Patch

Index des fehlerhaften Statements

0 = fehlerhaftes Statement wird nicht bearbeitet
1 = fehlerhaftes Statement wird bearbeitet

Verwendetet Operation:
1 = Löschen, 2 = Ersetzen, 3 = Einfügen

Index des verwendeten Ingredient-Statements in der Menge der möglichen Ingredient-Statements für das fehlerhafte Statement

Ablauf einer Reparatur

```
Input  : Sourc-Code des Programms,
         JUnit Tests
Output: Patches zur Fehlerbehebung
1  Fault Localization durchführen
2  Abstract Syntax Trees erzeugen
3  JUnit Tests filtern
4  Scope für Seed-Statements bestimmen
5  Ingredient-Statements auswählen
6  Initiale Population erzeugen
7  for n Populationen do
8  |   Patches erzeugen
9  |   foreach Patch do
10 |   |   if Patch erfüllt alle Tests then
11 |   |   |   Als gültigen Patch speichern
12 |   |   end if
13 |   end foreach
14 |   Nächste Population erzeugen
15 end for
```

- ARJA benötigt den Source-Code des Programms das repariert werden soll
- Außerdem müssen Testfälle bereitgestellt werden, welche auf den Fehler hinweisen
- Als Ergebnis liefert ARJA dann eine Menge an Patches

Ablauf einer Reparatur

```
Input  : Sourc-Code des Programms,
         JUnit Tests
Output: Patches zur Fehlerbehebung
1  Fault Localization durchführen
2  Abstract Syntax Trees erzeugen
3  JUnit Tests filtern
4  Scope für Seed-Statements bestimmen
5  Ingredient-Statements auswählen
6  Initiale Population erzeugen
7  for n Populationen do
8  |   Patches erzeugen
9  |   foreach Patch do
10 |   |   if Patch erfüllt alle Tests then
11 |   |   |   Als gültigen Patch speichern
12 |   |   end if
13 |   end foreach
14 |   Nächste Population erzeugen
15 end for
```

- Negative Tests verweisen auf Fehler
- Berechne für jede Codezeile, wie wahrscheinlich diese für Fehler verantwortlich ist
- Zeilen mit hoher Wahrscheinlichkeit sind Kandidaten um zu fehlerhaften Statements zu werden

Ablauf einer Reparatur

```
   Input  : Sourc-Code des Programms,
            JUnit Tests
   Output: Patches zur Fehlerbehebung
 1 Fault Localization durchführen
 2 Abstract Syntax Trees erzeugen
 3 JUnit Tests filtern
 4 Scope für Seed-Statements bestimmen
 5 Ingredient-Statements auswählen
 6 Initiale Population erzeugen
 7 for n Populationen do
 8 |   Patches erzeugen
 9 |   foreach Patch do
10 |   |   if Patch erfüllt alle Tests then
11 |   |   |   Als gültigen Patch speichern
12 |   |   end if
13 |   end foreach
14 |   Nächste Population erzeugen
15 end for
```

- Nutze einen Parser um den Source-Code in AST umzuwandeln
- Die Codestellen die den Fehler verursachen werden zu fehlerhaften Statements
- Der Rest wird zu Seed-Statements

Ablauf einer Reparatur

```
   Input  : Sourc-Code des Programms,
            JUnit Tests
   Output: Patches zur Fehlerbehebung
 1 Fault Localization durchführen
 2 Abstract Syntax Trees erzeugen
 3 JUnit Tests filtern
 4 Scope für Seed-Statements bestimmen
 5 Ingredient-Statements auswählen
 6 Initiale Population erzeugen
 7 for n Populationen do
 8 |   Patches erzeugen
 9 |   foreach Patch do
10 |   |   if Patch erfüllt alle Tests then
11 |   |   |   Als gültigen Patch speichern
12 |   |   end if
13 |   end foreach
14 |   Nächste Population erzeugen
15 end for
```

- Prüfe für jeden positiven Test, welche Codezeilen für diesen benötigt werden
- Positive Tests, welche kein fehlerhaftes Statement benötigen werden ausgefiltert
- Die Reparatur verändert nur fehlerhafte Statements, deshalb bleiben diese Tests immer positiv

Ablauf einer Reparatur

```
   Input  : Sourc-Code des Programms,
            JUnit Tests
   Output: Patches zur Fehlerbehebung
 1 Fault Localization durchführen
 2 Abstract Syntax Trees erzeugen
 3 JUnit Tests filtern
 4 Scope für Seed-Statements bestimmen
 5 Ingredient-Statements auswählen
 6 Initiale Population erzeugen
 7 for n Populationen do
 8 |   Patches erzeugen
 9 |   foreach Patch do
10 |   |   if Patch erfüllt alle Tests then
11 |   |   |   Als gültigen Patch speichern
12 |   |   end if
13 |   end foreach
14 |   Nächste Population erzeugen
15 end for
```

- Bestimme für jedes fehlerhafte Statement welche Seed-Statements sichtbar sind und verwendet werden können
- Variablen-Scope: alle sichtbaren Feldvariablen, Parametervariablen und lokale Variablen
- Methoden-Scope: Variablen-Scope und alle sichtbaren Methoden

Ablauf einer Reparatur

Input : Sourc-Code des Programms, JUnit Tests
Output: Patches zur Fehlerbehebung

1 Fault Localization durchführen
2 Abstract Syntax Trees erzeugen
3 JUnit Tests filtern
4 Scope für Seed-Statements bestimmen
5 Ingredient-Statements auswählen
6 Initiale Population erzeugen
7 **for** *n Populationen* **do**
8 Patches erzeugen
9 **foreach** *Patch* **do**
10 **if** *Patch erfüllt alle Tests* **then**
11 Als gültigen Patch speichern
12 **end if**
13 **end foreach**
14 Nächste Population erzeugen
15 **end for**

- Wähle aus der gleichen Klasse, dem gleichen Paket oder dem gesamten Programm als Bereich aus
- Direkter Ansatz: Wählt aus Seed-Statements im Scope aus
- Typ basierter Ansatz: Versucht Seed-Statements außerhalb des Scopes auf Anweisungen innerhalb des Scopes abzubilden

```
function DEC(x)
    x := x − 1
    return x
```

a := a - 1

```
function ADD(a, b)
    a := a − 1
    a := a + 1
    return a + b
```

Ablauf einer Reparatur

```
   Input  : Sourc-Code des Programms,
            JUnit Tests
   Output: Patches zur Fehlerbehebung
 1 Fault Localization durchführen
 2 Abstract Syntax Trees erzeugen
 3 JUnit Tests filtern
 4 Scope für Seed-Statements bestimmen
 5 Ingredient-Statements auswählen
 6 Initiale Population erzeugen
 7 for n Populationen do
 8 |   Patches erzeugen
 9 |   foreach Patch do
10 |   |   if Patch erfüllt alle Tests then
11 |   |   |   Als gültigen Patch speichern
12 |   |   end if
13 |   end foreach
14 |   Nächste Population erzeugen
15 end for
```

- Erzeuge eine initiale Population aus möglichen Patches
- Wähle für jedes fehlerhafte Statement zufällig aus, ob dieses bearbeitet werden soll, welche Operation verwendet wird und welches Ingredient-Statement verwendet wird

1	2	3	...	n
1	0	1	...	1
2	1	3	...	1
5	3	8	...	6

Ablauf einer Reparatur

```
   Input  : Sourc-Code des Programms,
            JUnit Tests
   Output: Patches zur Fehlerbehebung
 1 Fault Localization durchführen
 2 Abstract Syntax Trees erzeugen
 3 JUnit Tests filtern
 4 Scope für Seed-Statements bestimmen
 5 Ingredient-Statements auswählen
 6 Initiale Population erzeugen
 7 for n Populationen do
 8 |   Patches erzeugen
 9 |   foreach Patch do
10 |   |   if Patch erfüllt alle Tests then
11 |   |   |   Als gültigen Patch speichern
12 |   |   end if
13 |   end foreach
14 |   Nächste Population erzeugen
15 end for
```

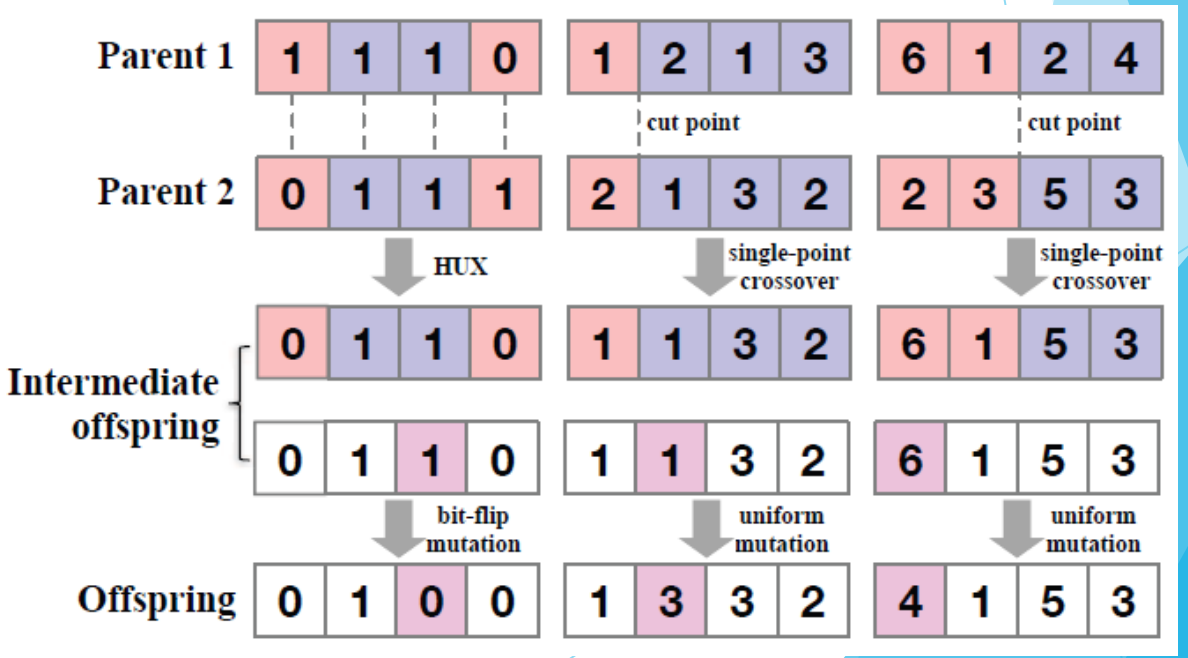
- Wende die Reparatur des Patches auf das Programm an
- Je kleiner der Umfang des Patches und je weniger negative Tests durch diesen entstehen, desto besser wird der Patch durch die Fitness Evaluation bewertet
- Patches die alle Tests erfüllen werden sind gültig und werden gespeichert

Ablauf einer Reparatur

Input : Sourc-Code des Programms, JUnit Tests
Output: Patches zur Fehlerbehebung

1 Fault Localization durchführen
2 Abstract Syntax Trees erzeugen
3 JUnit Tests filtern
4 Scope für Seed-Statements bestimmen
5 Ingredient-Statements auswählen
6 Initiale Population erzeugen
7 **for** *n Populationen* **do**
8 Patches erzeugen
9 **foreach** *Patch* **do**
10 **if** *Patch erfüllt alle Tests* **then**
11 Als gültigen Patch speichern
12 **end if**
13 **end foreach**
14 Nächste Population erzeugen
15 **end for**

- Erzeuge durch crossover- und mutation-Operationen die Nachfahren der Population



Gliederung

1. Einleitung
2. Machine Learning
3. Nutzung des Softwaresystems
4. Genetische Algorithmen
5. Funktionsweise von ARJA
6. Beispiel einer Reparatur
7. Forschungsfragen
8. Fazit

Defects4J

- Datenbank- und Framework-System zur Bereitstellung von reproduzierbaren Bugs
- Umfasst 17 Open-Source-Java-Projekte mit insgesamt 835 Bugs
- Für jeden Bug existieren Tests, wobei mindestens ein Test fehlschlägt
- Datenbank verwaltet die Versionen der Projekte, sodass Bugs erhalten bleiben, wenn diese im tatsächlichen Projekt behoben werden

Identifier	Project name	Number of active bugs	Active bug ids	Deprecated bug ids (*)
Chart	jfreechart	26	1-26	None
Cli	commons-cli	39	1-5,7-40	6
Closure	closure-compiler	174	1-62,64-92,94-176	63,93
Codec	commons-codec	18	1-18	None
Collections	commons-collections	4	25-28	1-24
Compress	commons-compress	47	1-47	None
Csv	commons-csv	16	1-16	None
Gson	gson	18	1-18	None
JacksonCore	jackson-core	26	1-26	None
JacksonDatabind	jackson-databind	112	1-112	None
JacksonXml	jackson-dataformat-xml	6	1-6	None
Jsoup	jsoup	93	1-93	None
JxPath	commons-jxpath	22	1-22	None
Lang	commons-lang	64	1,3-65	2
Math	commons-math	106	1-106	None
Mockito	mockito	38	1-38	None
Time	joda-time	26	1-20,22-27	21

Beispiel einer Reparatur

```
felix@Ubuntu-14-04-6:~$ defects4j checkout -p Lang -v 20b -w /tmp/lang_20_buggy
Checking out f08213cc to /tmp/lang_20_buggy..... OK
Init local repository..... OK
Tag post-fix revision..... OK
Excluding broken/flaky tests..... OK
Excluding broken/flaky tests..... OK
Excluding broken/flaky tests..... OK
Initialize fixed program version..... OK
Apply patch..... OK
Initialize buggy program version..... OK
Diff f08213cc:0c01b4c4..... OK
Apply patch..... OK
Tag pre-fix revision..... OK
Check out program version: Lang-20b..... OK
```

Beispiel einer Reparatur

```
felix@Ubuntu-14-04-6:/tmp/lang_20_buggy$ defects4j compile
Running ant (compile)..................................................... OK
Running ant (compile.tests)............................................... OK
felix@Ubuntu-14-04-6:/tmp/lang_20_buggy$ defects4j test
Running ant (compile.tests)............................................... OK
Running ant (run.dev.tests)............................................... OK
Failing tests: 2
  - org.apache.commons.lang3.StringUtilsTest::testJoin_ArrayChar
  - org.apache.commons.lang3.StringUtilsTest::testJoin_Objectarray
```

Beispiel einer Reparatur

```
felix@Ubuntu-14-04-6:~/src/arja$ java -cp lib/*:bin us.msu.cse.repair.Main Arja -DsrcJavaDir /tmp/lang_20_buggy/
src -DbinJavaDir /tmp/lang_20_buggy/target/classes -DbinTestDir /tmp/lang_20_buggy/target/tests -Ddependences /h
ome/felix/jars/lang_20_buggy/easymock-2.5.2.jar:/home/felix/jars/lang_20_buggy/junit-4.7.jar
```

Beispiel einer Reparatur

```
Fault localization starts...
Number of positive tests: 1874
Number of negative tests: 2
Fault localization is finished!
AST parsing starts...
AST parsing is finished!
Detection of local variables starts...
Detection of local variables is finished!
Detection of fields starts...
Detection of fields is finished!
Detection of methods starts...
Detection of methods is finished!
Ingredient screener starts...
Ingredient screener is finished!
Initialization of manipulations starts...
Initialization of manipulations is finished!
Filtering of the tests starts...
Number of positive tests considered: 4
Filtering of the tests is finished!
One fitness evaluation starts...
One fitness evaluation starts...
Number of failed tests: 3
Weighted failure rate: 1.125
One fitness evaluation is finished...
```

```
One fitness evaluation starts...
Number of failed tests: 2
Weighted failure rate: 0.625
One fitness evaluation is finished...
One fitness evaluation starts...
Number of failed tests: 1
Weighted failure rate: 0.5
One fitness evaluation is finished...
One fitness evaluation starts...
Number of failed tests: 0
Weighted failure rate: 0.0
One fitness evaluation is finished...
One fitness evaluation starts...
Number of failed tests: 4
Weighted failure rate: 0.875
One fitness evaluation is finished...
One fitness evaluation starts...
One fitness evaluation starts...
Number of failed tests: 2
Weighted failure rate: 0.625
One fitness evaluation is finished...
felix@Ubuntu-14-04-6:~/src/arja$
```

Beispiel einer Reparatur

```
1 InsertBefore /tmp/lang_20_buggy/src/main/java/org/apache/commons/lang3/StringUtils.java 3336
Faulty:
return null;
Seed:
java6Available=true;
************************************************
2 Replace /tmp/lang_20_buggy/src/main/java/org/apache/commons/lang3/StringUtils.java 3298
Faulty:
StringBuilder buf=new StringBuilder((array[startIndex] == null ? 16 : array[startIndex].toString().length()) + 1);
Seed:
StringBuilder buf=new StringBuilder(32);
************************************************
3 Replace /tmp/lang_20_buggy/src/main/java/org/apache/commons/lang3/StringUtils.java 3383
Faulty:
StringBuilder buf=new StringBuilder((array[startIndex] == null ? 16 : array[startIndex].toString().length()) + separator.length());
Seed:
StringBuilder buf=new StringBuilder(256);
************************************************
Evaluations: 1899
EstimatedTime: 510922
```

Beispiel einer Reparatur

```
felix@Ubuntu-14-04-6:/tmp/lang_20_buggy$ defects4j compile
Running ant (compile)..................................................... OK
Running ant (compile.tests)............................................... OK
felix@Ubuntu-14-04-6:/tmp/lang_20_buggy$ defects4j test
Running ant (compile.tests)............................................... OK
Running ant (run.dev.tests)............................................... OK
Failing tests: 0
```

Gliederung

1. Einleitung
2. Machine Learning
3. Nutzung des Softwaresystems
4. Genetische Algorithmen
5. Funktionsweise von ARJA
6. Beispiel einer Reparatur
7. Forschungsfragen
8. Fazit

Forschungsfragen

- Vergleich von ARJA mit vier anderen Tools zur automatischen Programmreparatur: GenProg, RSRepair, Kali, Nopol
- Verwendung von Programmcode aus der defects4j Datenbank
- Metriken:
  - Success: Anzahl der Versuche, in denen ein Patch gefunden wird
  - Evaluations: Anzahl der Iterationen, um einen Patch zu finden
  - Patch Size: Anzahl der Programmanweisungen, um den Patch anzuwenden
  - Patches: Anzahl der Patches, die gefunden wurden

Ist ARJA besser als andere Tools beim Finden von Bugs an unterschiedlichen Stellen im Code?

- Bug zieht sich über mehrere Stellen im Code
- Nicht alle Tools sind in der Lage diese Bugs zu identifizieren
- ARJA findet häufiger Patches und benötigt weniger Iterationen
- Gefundene Patches sind meistens kleiner

TABLE 8
Comparison of ARJA, GenProg, and RSRepair on multi-location bugs. (Average of 30 runs)

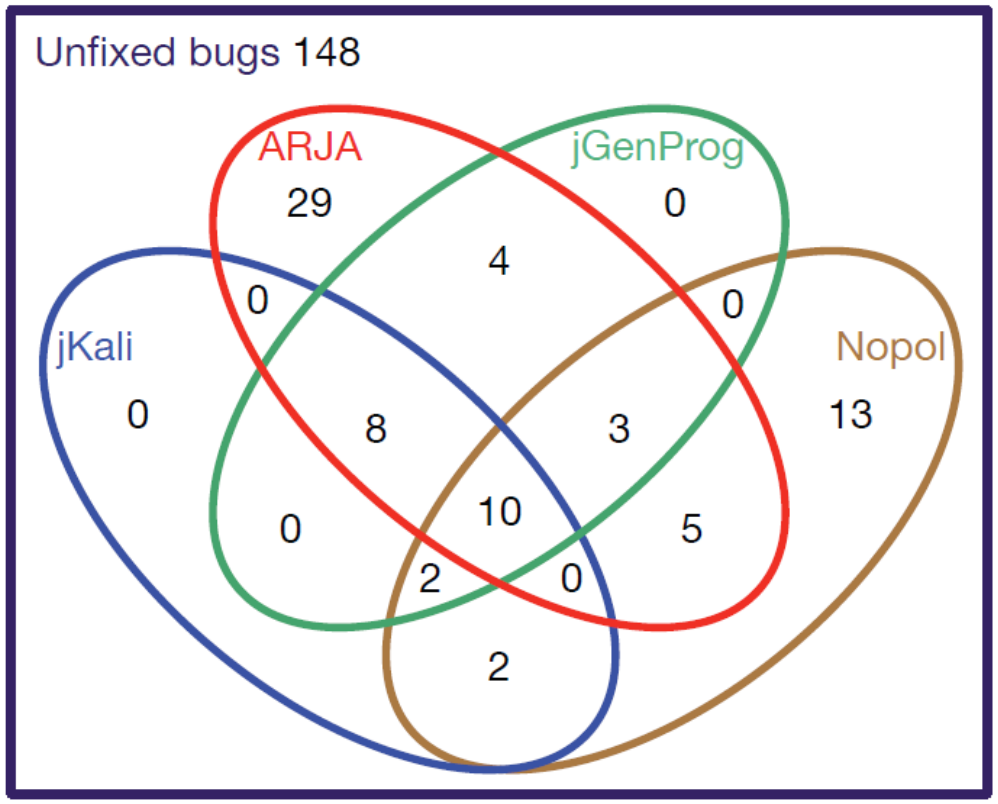
Bug Index	Success			#Evaluations			CPU (s)			Patch Size		
	ARJA	GenProg	RSRepair	ARJA	GenProg	RSRepair	ARJA	GenProg	RSRepair	ARJA	GenProg	RSRepair
F3	26	23	0	494.24	628.64	–	634.91	193.69	–	2.12	3.09	–
F4	13	2	0	746.54	1240.50	–	980.77	1129.29	–	2.23	6.50	–
F5	30	11	4	384.63	1235.30	1000.50	98.33	248.73	138.80	2.67	4.10	2.00
F6	30	11	0	624.80	894.91	–	393.25	127.18	–	2.80	7.09	–
F7	25	4	0	698.52	820.00	–	461.89	186.60	–	2.24	7.00	–
F8	29	24	3	376.21	915.21	1028.33	551.50	678.05	507.51	2.14	6.79	2.33
F9	6	1	0	1028.00	225.00	–	962.62	52.15	–	2.33	2.00	–
F10	18	2	0	936.11	1896.00	–	190.01	280.70	–	3.00	4.50	–
F11	20	9	0	777.70	1325.00	–	532.54	378.61	–	3.00	11.33	–
F12	28	15	0	742.04	973.73	–	566.57	273.59	–	3.07	9.67	–
F13	20	8	0	762.30	1383.00	–	485.07	357.65	–	3.15	14.88	–

"–" means the data is not available.

Wie performant ist ARJA im Vergleich zu anderen Tools?

- Es wird geprüft wie viele Bugs erkannt und wie viele Reparaturen gefunden werden können
- ARJA erkennt mehr Bugs als die anderen Tools
- Verschiedene Tools entdecken und reparieren auch verschiedene Bugs

Tools erkennen unterschiedliche Bugs:



Sind die generierten Patches semantisch korrekt?

- Ein Patch ist semantisch korrekt, wenn er äquivalent zu einem Patch ist, der von einem Menschen geschrieben wurde
- ARJA findet Patches, welche zu einem Bestehen der Testfälle führen
- Overfitting kann auftreten
- Der Patch sollte dann nicht als Reparatur anerkannt werden

Warum kann ARJA für manche Fehler keine korrekten Patches generieren?

- ARJA findet nicht immer eine Reparatur
- Eventuell enthält der gegebene Code nicht die benötigten Anweisungen um das Programm zu reparieren
- Genetischer Algorithmus hat nicht genug Leistung
- Codezeilen, die zum Fehlschlag eines Tests führen, werden nicht erkannt

Gliederung

1. Einleitung
2. Machine Learning
3. Nutzung des Softwaresystems
4. Genetische Algorithmen
5. Funktionsweise von ARJA
6. Beispiel einer Reparatur
7. Forschungsfragen
8. Fazit

Fazit

- ARJA ist nur für umfangreiche Softwareprojekte geeignet
- Die Qualität der Reparatur steigt mit der Menge an vorhandenem Code
- Projekte benötigen umfassende Tests
- Patches müssen auf semantische Korrektheit überprüft werden
- Ermöglicht eine Beschleunigung des Softwareentwicklungsprozesses

Vielen Dank für eure Aufmerksamkeit!