0.1 Semantische Evaluation

Das Ziel der semantischen Evaluation ist es, einen der Proxies, die im Rahmen der 1. Stufe der Exploration erzeugt wurden, hinsichtlich der vordefinierten Testfälle zu evaluieren. Da die gesamte Exploration zur Laufzeit des Programms durchgeführt wird, stellt sie hinsichtlich der nicht-funktionalen Anforderungen eine zeitkritische Komponente dar.

Da die Anforderungen an die gesuchte Komponente mit bedacht spezifiziert werden müssen, ist es irrelevant, ob es mehrere Proxies gibt, die den vordefinierten Testfällen standhalten. Vielmehr soll bei der semantischen Evaluation lediglich ein Proxy gefunden werden, dessen Semantik zu positiven Ergebnissen hinsichtlich aller vordefinierten Testfälle führt. Somit wird die semantische Evaluation beendet, sobald ein solcher Proxy gefunden ist.

0.1.1 Besonderheiten der Testfälle

Bei den vordefinierten Tests handelt es sich auf formaler Ebene um Typen, die eine eval-Methode mit der Struktur `boolean eval( proxy )` anbieten, welche einen Proxy als Parameter erwartet und ein Objekt vom Typ `boolean` zurückgibt. Weiterhin verfügt ein Test über die in Tabelle 1 aufgeführten Felder, deren Werte bei der Abarbeitung der Methode `eval` verändert werden.

Feldname	**Erläuterung**
`calledMethods`	Eine Liste von Namen von Methoden des Proxies, die bei der Durchführung der eval-Methode aufgerufen wurden.
`failedMethod`	Der Name einer Methode des Proxies, bei der es während der Ausführung der eval-Methode zu einem Fehler kam.

Tabelle 1: Felder der Tests

Die Implementierung der eval-Methode ist an folgende Bedingungen geknüpft:

1. Nach einem erfolgreichen Aufruf einer Methode auf dem als Parameter übergebenen Proxy-Objekt, wird der Name der dieser Methode in der Liste im Feld `calledMethods` ergänzt.

2. Nach einem fehlgeschlagenen Aufruf einer Methode auf dem als Parameter übergebenen Proxy-Objekt, wird das Feld `failedMethod` mit dem Namen der fehlgeschlagenen Methode belegt. Zusätzlich wird die eval-Methode direkt danach mit dem Rückgabewert `false` beendet.

3. Wenn der Proxy den Test erfüllt, wird der Wert `true` zurückgegeben. Anderenfalls wird der Wert `false` zurückgegeben.

0.1.2 Algorithmus für die semantische Evaluation

Bei der Exploration soll letztendlich in einer Bibliothek L zu einem vorgegebenen required Type R ein Proxy gefunden werden. Die Menge dieser Proxies wurde im vorherigen über $cover(R, L)$ beschrieben. Die in dieser Menge befindlichen Proxies können eine unterschiedliche Anzahl von Target-Typen enthalten.

Das in dieser Arbeit beschriebene Konzept basiert auf der Annahme, dass bei der Entwicklung davon ausgegangen wird, dass der gesamte Anwendungsfall - oder Teile davon - , der mit der vordefinierten Struktur und den vordefinierten Tests abgebildet werden soll, schon einmal genauso oder so ähnlich in dem gesamten System implementiert wurde. Aus diesem Grund kann für die semantische Evaluation grundsätzlich davon ausgegangen werden, dass die erfolgreiche Durchführung aller relevanten Tests umso wahrscheinlicher ist je weniger Target-Typen im Proxy verwendet werden.

Daher sei folgende Funktion für eine Menge von Proxies P und eine ganze Zahl $a > 0$ definiert:

$$proxies\,With\,Targets(P, a) := \{P | P.targetCount = a\}$$

Die maximale Anzahl der Target-Typen eines Proxies für einen required Typ R ist gleich der Anzahl der Methoden in R.

$$maxTargets(R) := |methoden(R)|$$

So kann der Algorithmus für die semantische Evaluation der Menge von Proxies (Parameter `proxies`), die für einen required Typ (Parameter `reqType`) erzeugt wurden, mit der Menge von Tests (Parameter `tests`) wie folgt im Pseudo-Code beschrieben werden. Die globale Variable `passedTests` enthält dabei die Anzahl der für den aktuell zu überprüfenden Proxy erfolgreich durchgeführten Tests.

```
1  passedTests = 0
2
3  function semanticEval( reqType , proxies , tests ){
4    for( i = 1; i <= maxTargets( reqType ); i++ ){
5      relProxies = relevantProxies( proxies, i )
6      proxy = evalProxiesMitTarget(relProxies, tests)
7      if( proxy != null ){
8        // passenden Proxy gefunden
9        return proxy
10     }
11   }
12   // kein passenden Proxy gefunden
13   return null;
```

```
14 }
15
16 function relevantProxies(P,anzahl){
17   return proxiesWithTargets(P,anzahl);
18 }
19
20 function evalProxiesMitTarget(proxies, tests){
21   for( proxy : proxies ){
22     passedTests = 0
23     evalProxy(proxy, tests)
24     if( passedTests == tests.size ){
25       // passenden Proxy gefunden
26       return proxy
27     }
28   }
29   // kein passenden Proxy gefunden
30   return null
31 }
32
33 function evalProxy(proxy, tests){
34   for( test : tests ){
35     if( !test.eval( proxy ) ){
36       \\ wenn ein Test fehlschlaegt, dann entspricht der
37       \\ Proxy nicht den semantischen Anforderungen
38       return
39     }
40     passedTests = passedTests + 1
41   }
42 }
```

Listing 1: Semantische Evaluation ohne Heuristiken

Die Dauer der Laufzeit der oben genannten Funktionen hängt maßgeblich von der Anzahl der Proxies für required Typs R in einer Bibliothek L ab (siehe auch Funktion *libProxyCount*). Im schlimmsten Fall müssen alle Proxies hinsichtlich der vordefinierten Tests evaluiert werden. Um die Anzahl der zu prüfenden Proxies zu reduzieren werden, die im folgenden Abschnitt beschriebenen Heuristiken verwendet.

Um die einzelnen Stufen der semantischen Evaluation besser unterscheiden zu können, sei der Algorithmus aus Listing 1 in zwei Iterationsstufen unterteilt. Die erste Iterationsstufe beschreibt die Iteration über Proxies mit einer bestimmten Anzahlen von Target-Typen in der Funktion `semanticEval` (siehe Zeile 3-14). Die zweite Itertationsstufe beschreibt die Iteration über die Proxies mit einer gleichen Anzahl an Target-Typen in der Funktion `evalProxiesMitTarget` (siehe Zeile 20-31). Die Heuristiken sind so gestaltet, dass sie die Itertationsobjekte dieser beiden Iterationsstufen beeinflussen.

0.2 Heuristiken

Die Heuristiken werden an unterschiedlichen Stellen des Algorithmus' für die semantische Evaluation aus Listing 1 eingebaut. Teilweise ist es für die Verwendung einer Heuristik notwendig, weitere Information während der semantischen Evaluation zu ermitteln und zu speichern. In den folgenden Abschnitten werden die Heuristiken und die dafür notwendigen Anpassungen an den jeweiligen Funktionen beschrieben.

0.2.1 Heuristiken für die Optimierung der Reihenfolge

Die folgenden Heuristiken haben zum Ziel, die Reihenfolge, in der die Proxies hinsichtlich der vordefinierten Tests geprüft werden, so anzupassen, dass ein passender Proxy möglichst früh geprüft wird.

Heuristik LMF: Beachtung des Matcherratings
Bei dieser Heuristik werden die Proxies auf der Basis eines so genannten Matcherratings bewertet. Bei dem Matcherrating eines Proxies handelt es sich um einen numerischen Wert. Um diesen Wert zu ermitteln, wird für jeden Matcher ein Basisrating vergeben. Folgende Funktion beschreibt das Basisrating für das Matching zweier Typen S und T:

$$base(S, T) := \left\{ \begin{array}{l} 100|S \Rightarrow_{exact} T \\ 200|S \Rightarrow_{gen} T \\ 200|S \Rightarrow_{spec} T \\ 300|S \Rightarrow_{contained} T \\ 300|S \Rightarrow_{container} T \end{array} \right\}$$

Dabei ist zu erwähnen, dass einige der o.g. Matcher über dasselbe Basisrating erfügen. Das liegt daran, dass sie technisch jeweils gemeinsam umgesetzt wurden.[1]

Das Matcherrating eines Proxies P wird über die Funktion $rating(P)$ beschrieben. Dieses ist von dem Matcherrating der Methoden-Delegation innerhalb des Proxies P abhängig. Das Matcherrating einer Methoden-Delegation ist von den Basisratings der Matcher abhängig, über die die Parameter- und Rückgabe-Typen der aufgerufenen Methode und der Delegationsmethoden gematcht werden können. Das qualitative Rating einer Methoden-Delegation

[1] Der *GenTypeMatcher* und der *SpecTypeMatcher* wurden gemeinsam in der Klasse `GenSpecTypeMatcher` umgesetzt. Der *ContentTypeMatcher* und der *ContainerTypeMatcher* wurden gemeinsam in der Klasse `WrappedTypeMatcher` umgesetzt. (siehe angehängter Quellcode)

MD soll über die Funktion *mdRating*(*MD*) beschrieben werden.

Für die Definition der beiden Funktionen *rating*(*P*) und *mdRating*(*MD*) gibt es unterschiedliche Möglichkeiten. In dieser Arbeit werden 4 Varianten als Definitionen vorgeschlagen, die in einem späteren Abschnitt untersucht werden.

Für die Vorschläge zur Definition von *rating*(*P*) sei P ein struktureller Proxy mit n Methoden-Delegation. Darüber hinaus gelten für die Definition von *mdRating*(*MD*) für eine Methoden-Delegation *MD* folgende verkürzte Schreibweisen:

$$pc := MD.call.paramCount$$
$$cRT := MD.call.returnType$$
$$dRT := MD.del.returnType$$
$$cPT := MD.call.paramTypes$$
$$dPT := MD.del.paramTypes$$
$$pos := MD.call.posModi$$

Weiterhin seien die folgenden Funktionen gegeben:

$$basesMD(MD) := base(dRT, cRT) \cup \bigcup_{i=0}^{pc-1} base(cPT[i], dPT[pos[i]])$$

$$sum(v_1, ...v_n) = \sum_{i=1}^{n} v_i$$

$$max(v_1, ..., v_n) = v_m | 1 \leq m \leq n \wedge \forall i \in \{1, ..., n\} : v_i \leq v_m$$

$$min(v_1, ..., v_n) = v_m | 1 \leq m \leq n \wedge \forall i \in \{1, ..., n\} : v_i \geq v_m$$

Variante 1: Durchschnitt

$$mdRating(MD) = \frac{sum(basesMD(MD))}{pc + 1}$$

$$rating(P) = \frac{sum(mdRating(P.dels[0]), ..., mdRating(P.dels[n-1]))}{n}$$

Variante 2: Maximum

$$mdRating(MD) = max(basesMD(MD))$$

$$rating(P) = \frac{max(mdRating(P.dels[0]), ..., mdRating(P.dels[n-1]))}{n}$$

Variante 3: Minimum

$$mdRating(MD) = min(basesMD(MD))$$

$$rating(P) = \frac{min(mdRating(P.dels[0]), ..., mdRating(P.dels[n-1]))}{n}$$

Variante 4: Durchschnitt aus Minimum und Maximum

$$mdRating(MD) = \frac{max(basesMD(MD)) + min(basesMD(MD))}{2}$$

$$rating(P) = \frac{\begin{array}{c} max(mdRating(P.dels[0]), ..., mdRating(P.dels[n-1])) \\ + min(mdRating(P.dels[0]), ..., mdRating(P.dels[n-1])) \end{array}}{2}$$

Da die Funktion *rating* von *mdrating* abhängt und für *mdrating* 4 Variante gegeben sind, ergeben sich für jede gegebene Variante für die Definition von *rating* weitere 4 Varianten. Dadurch sind insgesamt 16 Varianten für die Definition von *rating* gegeben.

Zur Anwendung der Heuristik muss das qualitative Rating bei der Auswahl der Proxies in der semantischen Evaluation beachtet werden. Die erfolgt innerhalb der Methode `applyHeuristic(proxies)`. Für diese Heuristik sei dazu eine Methode `sort(proxies, rateFunc)` angenommen, die eine Liste zurückgibt, in der die Elemente in der übergebenen Liste `proxies` aufsteigend nach den Werten sortiert, die durch die Applikation der im Parameter `rateFunc` übergebenen Funktion auf ein einzelnes Element aus der Liste `proxies` ermittelt werden. Darauf aufbauend wird die Methode `applyHeuristic(proxies)` für diese Heuristik in Pseudo-Code wie folgt definiert:

```
1  function relevantProxies( proxies, anzahl ){
2    relProxies = proxiesMitTargets( proxies, anzahl );
3    return LMF( relProxies )
```

```
4  }
5
6  function LMF( proxies ){
7    for ( n=proxies.size(); n>1; n--){
8      for( i=0; i<n-1; i++){
9        if( rating( proxies[i] ) < rating( proxies[i+1] ) ){
10         tmp = proxies[i]
11         proxies[i] = proxies[i+1]
12         proxies[i+1] = tmp
13       }
14     }
15   }
16   return proxies
17 }
```

Heuristik PTTF: Beachtung bestandener Tests
Das Testergebnis, welches bei Applikation eines Testfalls für einen Proxy ermittelt wird, ist maßgeblich von den Methoden-Delegationen des Proxies abhängig. Jede Methoden-Delegation MD enthält ein Typ in dem die Delegationsmethode spezifiziert ist. Dieser Typ befindet sich im Attribut $MD.del.delTyp$. Im Fall der sturkturellen Proxies, handelt es sich bei diesem Typ um einen der Target-Typen des Proxies.

Für einen required Typ R aus einer Bibliothek L, kann ein Target-Typ T in den Mengen der möglichen Mengen von Target-Typen $cover(R, L)$ mehrmals auftreten. Die gilt insbesondere dann, wenn es in $cover(R, L)$ Mengen gibt, deren Mächtigkeit größer ist, als die Mächtigkeit der Menge, in der T enthalten ist. Daher gilt:

$$\frac{TG, TG' \in cover(R, L) \wedge T \in TG \wedge |TG| < |TG'|}{\exists TG'' \in cover(R, L) : |TG'| = |TG''| \wedge T \in TG''}$$

Beweis: Sei R ein required Typ aus der Bibliothek L. Sei weiterhin $T \in TG$ und $TG \in cover(R, L)$.

Wie bereits erwähnt, ist das Ergebnis der semantischen Tests ausschlaggebend für diese Heuristik. Es wird davon ausgegangen, dass wenn ein Teil der Testfälle durch einen Proxy P erfolgreich durchgeführt werden, sollte die Reihenfolge der zu prüfenden Proxies so angepasst werden, dass die Proxies, die einen Target-Typen des Proxies P verwenden, zuerst geprüft werden.

Dafür sind mehrere Anpassungen bzgl. der Implementierung von Nöten.

Für die Methoden `evalProxiesMitTarget(P,anzahl,T)` ergeben sich darüber hinaus mehrere Änderungen. Die Implementierung mit allen An-

passungen ist Listing 2 zu entnehmen. Die einzelnen Änderungen werden im Folgenden erläutert.

Merken der priorisierten Target-Typen
Um die Optimierungen auf der Basis dieser Heuristik vornehmen zu können, wird von einer globalen Variable `priorityTargets` ausgegangen. In dieser Variablen wird eine Liste von Target-Typen der Proxies gehalten, für die wenigsten ein Testfall erfolgreich durchgeführt wurde (siehe Listing 2 Zeile 14).

Aktualisierung der Proxy-Liste aus der aktuellen Iteration
Im Vergleich zu der Heuristik LMF aus dem vorherigen Abschnitt bietet die Heuristik PTTF die Möglichkeit auch die Reihenfolge der Proxies aus der aktuellen Iteration zu optimieren. Dazu muss die Heuristik PTTF auf die Proxies, die in dieser Iterationsstufe noch nicht evaluiert wurden, angewandt werden (siehe Listing 2 Zeile 17). Zu diesem Zweck werden die in dieser Iterationsstufe bereits evaluierten Proxies in einer Liste die in der Variablen `testedProxies` gespeichert (siehe Listing 2 Zeile 11). Diese Liste dient dann zur Reduktion der Proxy-Liste, über die in dieser Methode iteriert wird (siehe Listing 2 Zeile 16).

```
1  function evalProxiesMitTarget(proxies, tests){
2    testedProxies = []
3    for( proxy : proxies ){
4      passedTestcases = 0
5      evalProxy(proxy, tests)
6      if( passedTestcases == T.size ){
7        // passenden Proxy gefunden
8        return proxy
9      }
10     else{
11       testedProxies.add(proxy)
12       if( passedTests > 0 ){
13         priorityTargets.addAll( proxy.targets )
14         // noch nicht evaluierte Proxies ermitteln
15         leftProxies = proxies.removeAll( testedProxies )
16         optmizedProxies = PTTF( leftProxies )
17         return evalProxiesMitTarget( optmizedProxies, tests )
18       }
19     }
20   }
21   // kein passenden Proxy gefunden
22   return null
23 }
24
25 function relevantProxies( proxies, anzahl ){
26   relProxies = proxiesMitTargets( proxies, anzahl );
27   return PTTF( relProxies )
28 }
29
```

```
30 function PTTF(proxies){
31   for ( n=proxies.size ; n>1; n--){
32     for( i=0; i<n-1; i++){
33       targetsFirst = proxies[i].targets
34       targetsFirst = proxies[i+1].targets
35       if( !priorityTargets.contains(targetsFirst) &&
             priorityTargets.contains(targetsSecond) ){
36         tmp = proxies[i]
37         proxies[i] = proxies[i+1]
38         proxies[i+1] = tmp
39       }
40     }
41   }
42   return proxies
43 }
```

Listing 2: Auswertung des Testergebnisses mit Heuristik PTTF

0.2.2 Heuristiken für den Ausschluss von Methodendelegationen

Bei den folgenden Heuristiken handelt es sich um Ausschlussverfahren. Das bedeutet, dass bestimmte Proxies auf der Basis von Erkenntnissen, die während der laufenden semantischen Evaluation entstanden sind, für den weiteren Verlauf ausgeschlossen werden. Dadurch soll die erneute Prüfung eines Proxies, der ohnehin nicht zum gewünschten Ergebnis führt, verhindert werden.

Die Heuristiken zielen darauf ab, Methodendelegationen, die immer fehlschlagen, zu identifizieren. Wurde eine solche Methodendelegation gefunden, können alle Proxies, die diese Methodendelegation enthalten von der weiteren Exploration ausgeschlossen werden.

Um eine solche Methodendelegation identifizieren zu können, müssen die Testfälle weitere Besonderheiten erfüllen und Informationen bereitstellen. Zum Einen ist es notwendig, dass spezielle Tests verwendet werden, in denen lediglich eine Methode getestet wird. So kann auf der Basis eines Fehlgeschlagenen Tests geschlussfolgert werden, dass die Methodendelegation, die für die getestete Methode verwendet wurde, ebenfalls in anderen Proxies zu einem Fehlschlag des Testfalls führt.

Zu diesem Zweck sei angenommen, dass ein einzelner Test über ein Attribut `isSingleMethodTest` und `singleMethodName`. Die beiden Attribute sind durch den Entwickler bei der Implementierung dieser Tests zu füllen. Dabei ist vorgesehen, dass das Attribut `isSingleMethodTest` mit dem Wert `true` belegt ist, wenn es sich um einen Test handelt, in dem lediglich eine Methode des Proxies aufgerufen wird. Darüber hinaus ist die Attribut

`singleMethodName` mit dem Namen der in dem Testfall aufgerufene Methode des Proxies zu belegen.

Basierend auf diesen Werten der beiden Attribute, können die Tests in folgende Kategorien unterteilt werden:

Test-Kategorie	**Eigenschaften**
Single-Method-Test	`isSingleMethodTest` = `true` `singleMethodName` $\neq$ `null`
Multi-Method-Test	`isSingleMethodTest` = `false`

Die Methodendelegationen, die auf der Basis der beiden folgenden Heuristiken aussortiert werden sollen, werden zu diesem Zweck in einer globalen Variable gehalten. Aus einer Liste von Proxies können darauf aufbauend diejenigen Proxies entfernt werden, die eine jener Methodendelegationen enthalten. Dabei wird davon ausgegangen, dass die Methoden eines required Typen über den Namen identifiziert werden können.

Zusätzlich ist zu erwähnen, dass die folgenden Heuristiken, wie auch die Heuristik PTTF, eine Optimierung der zu testenden Proxies nach jeder fehlgeschlagenden Evaluation eines Proxies erlauben. Ob eine Optimierung an diesem Punkt möglich ist, wird durch eine neue globale Variable `blacklistChanged` gesteuert.

Listing 0.2.2 zeigt die allgemeinen Anpassungen für die folgenden Heuristiken basieren auf den Funktionen aus Listing 0.2.2.

```
1  methodDelegationBlacklist = []
2  blacklistChanged = false
3
4  function evalProxiesMitTarget(proxies, tests){
5    testedProxies = []
6    for( proxy : proxies ){
7      passedTestcases = 0
8      evalProxy(proxy, tests)
9      if( passedTestcases == tests.size ){
10       // passenden Proxy gefunden
11       return proxy
12     }
13     else{
14       testedProxies.add(proxy)
15       if( blacklistChanged ){
16         // noch nicht evaluierte Proxies ermitteln
17         leftProxies = proxies.removeAll(testedProxies)
18         optimizedProxies = BL( leftProxies )
19         return evalProxiesMitTarget( optimizedProxies, tests )
20       }
```

```
21      }
22    }
23    // kein passenden Proxy gefunden
24    return null
25  }
26
27  function relevantProxies( proxies, anzahl ){
28    relProxies = proxiesMitTargets( proxies, anzahl );
29    return BL( optimizedFSMT )
30  }
31
32  function BL( proxies ){
33    optimizedProxies = []
34    for( proxy : proxies ){
35      blacklisted = false
36      for( md : methodDelegationBlacklist ){
37        if( proxy.dels.contains( md ) ){
38          blacklisted = true
39          break
40        }
41      }
42      if( !blacklisted ){
43        optimizedProxies.add( proxy )
44      }
45    }
46    return optimizedProxies
47  }
```

Die folgenden Heuristiken erfordern jeweils eine Anpassung der Funktion `evalProxy`.

Heuristik BL_FSMT: Beachtung fehlgeschlagener Single-Method-Test

Basierend darauf, dass ein

```
1
2
3   function evalProxy(proxy, T){
4     for( test : T ){
5       if( test.eval( proxy ) ){
6         passedTestcases = passedTestcases + 1
7       }elseif( test.isSingleMethodTest ){
8         methodName = test.singleMethodName
9         mDel = getMethodDelegation(proxy, methodName)
10        blacklistChanged = true
11      }
12    }
13  }
14
15  function getMethodDelegation( proxy, methodName ){
16    for( i=0; i < proxy.dels.size; i++ ){
17      if( proxy.dels[i].call.name == methodName ){
18        return proxy.dels[i]
19      }
20    }
21    return null
```

```
22 }
```

Listing 3: Semantische Evaluation mit Heuristik SMTE

Heuristik BL_FFMD: Beachtung fehlgeschlagener Methoden-Delegationen

```
1  failedMethodDelegation = []
2
3  function evalProxy(proxy, T){
4    for( test : T ){
5      //alle Tests werden durchgefuehrt
6      try{
7        if( !test.eval( proxy ) ){
8          return
9        }
10       passedTestcases = passedTestcases + 1
11     }
12     catch (SigMaGlueException e){
13       mDel = e.failedMethodDelegation
14       if( test.isSingleMethodTest &&
15         mDel.call.name == test.singleMethodName){
16         failedMethodDelegation.add(mDel)
17         blacklistChanged = true
18       }
19       return
20     }
21   }
22 }
```

Listing 4: Abfangen der SigMaGlueException beim Testen eines Proxies

0.2.3 Kombination der Heuristiken

Die oben genannten Heuristiken können miteinander Kombiniert werden. Listing 5 zeigt die Implementierung der Funktionen, die für diese Kombination auf der Basis von Listing 1 angepasst werden müssen. Dabei ist davon auszugehen, dass die Funktionen `LMF`, `PTTF`, `FSMT` und `FFMD` definiert sind.

```
1  function evalProxiesMitTarget( proxies, tests ){
2    testedProxies = []
3    for( proxy : proxies ){
4      passedTestcases = 0
5      blacklistChanged = false
6      evalProxy(proxy, tests)
7      if( passedTests == T.size ){
8        // passenden Proxy gefunden
9        return proxy
10     }
11     else{
12       testedProxies.add(proxy)
13       if( passedTests > 0 || blacklistChanged ){
14         // noch nicht evaluierte Proxies ermitteln
15         optmizedProxies = proxies.removeAll( testedProxies )
```

```
16          // Heuristik PTTF
17          if( passedTests > 0 ){
18            priorityTargets.addAll( proxy.targets )
19            optmizedProxies = PTTF( optmizedProxies )
20          }
21          // Heuristik BL_FFMD und BL_FSMT
22          if( blacklistChanged ){
23            optmizedProxies = BL( optmizedProxies )
24          }
25          return evalProxiesMitTarget( optmizedProxies, tests )
26        }
27      }
28    }
29    // kein passenden Proxy gefunden
30    return null
31  }
32
33  function evalProxy(proxy, tests){
34    for( test : tests ){
35      //alle Tests werden durchgefuehrt
36      try{
37        if( test.eval( proxy ) ){
38          passedTestcases = passedTestcases + 1
39        }elseif( test.isSingleMethodTest ){
40          methodName = test.testedSingleMethodName
41          mDel = getMethodDelegation( proxy, methodName )
42          methodDelegationBlacklist.add( mDel )
43          blacklistChanged = true
44          return
45        }
46      }
47      catch (SigMaGlueException e){
48        mDel = e.failedMethodDelegation
49        methodDelegationBlacklist.add( mDel )
50        blacklistChanged = true
51        return
52      }
53    }
54  }
55
56  function relevantProxies( proxies, anzahl ){
57    relProxies = proxiesMitTargets( proxies, anzahl );
58    optimizedLMF = LMF( relProxies )
59    optimizedPTTF = PTTF( optimizedLMF )
60    return BL( optimizedPTTF )
61  }
```

Listing 5: Kombination aller Heuristiken