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.

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.

Somit werden zuerst die Proxies auf ihr semantisches Matching überprüft, in denen lediglich ein Target-Typ verwendet wird. Die Menge der Proxies aus einer Menge von Proxies P mit einer Anzahl a von Target-Typen wird durch folgende Funktion beschrieben:

$$proxiesMitTargets(P, a) := \{P | P.targetCount = a\}$$

Die maximale Anzahl der Target-Typen in einerm Proxy zu einem required Typ R ist gleich der Anzahl der Methoden in P.

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

So kann der Algorithmus für die semantische Evaluation der Menge P von Proxies, die für einen required Typ R erzeugt wurden, mit der Menge von Testfällen T wie folgt im Pseudo-Code beschrieben werden. Dabei sei davon auszugehen, dass ein Test aus T mit einem Proxy p über eine Methode `eval(p)` ausgewertet werden kann. Diese Methode gibt bei erfolgreicher Durchführung den Rückgabewert `true` und anderenfalls `false` zurück.

```
function semanticEval(R, P, T){
  for( i = 1; i <= maxTargets(R); i++ ){
    proxy = evalProxiesMitTarget(P,i,T)
    if( proxy != null ){
      // passenden Proxy gefunden
      return proxy
    }
  }
  // kein passenden Proxy gefunden
  return null;
}

function evalProxiesMitTarget(P,anzahl,T){
  for( proxy : relevantProxies(P,anzahl) ){
    if( evalProxy(proxy, T) ){
      // passenden Proxy gefunden
      return proxy
    }
  }
  // kein passenden Proxy gefunden
  return null
}

function relevantProxies(P,anzahl){
  return proxiesMitTargets(P,anzahl);
}

function evalProxy(proxy, T){
  for( test : T ){
    if( !test.eval(proxy) ){
      \\ wenn ein Test fehlschlaegt, dann entspricht der
      \\ Proxy nicht den semantischen Anforderungen
      return false
    }
  }
  return true
}
```

Die Dauer der Laufzeit der oben genannten Funktionen hängt maßgeblich von der Anzahl der Proxies *PA* ab. 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.

1.1 Heuristiken

Die Heuristiken werden immer innerhalb der Methode `relevantProxies` angewendet. So kann diese Methode wie in folgendem Listing erweitert werden. Die jeweilige Heuristik wird dann über die Methode `applyHeuristic` beschrieben.

```
function relevantProxies(P,anzahl){
  proxies = proxiesMitTargets(P,anzahl)
  optmizedProxies = applyHeuristic(proxies)
  return optmizedProxies
}
```

1.2 Heuristiken für die Optimierung der Reihenfolge

Die folgende Heuristik hat zum Ziel, die Reihenfolge, in der die Proxies hinsichtlich der vordefinierten Tests evaluiert werden, so anzupassen, dass ein passender Proxy möglichst früh überprüft wird.

1.2.1 Heuristik: Low Matcherrating First (LMF)

Die folgende Heuristik hat zum Ziel, die Reihenfolge, in der die Proxies hinsichtlich der vordefinierten Tests evaluiert werden, so anzupassen, dass ein passender Proxy möglichst früh überprüft wird. Dabei dient jeweils ein so genanntes *Matcherrating* der Proxies als Kriterium für die Festlegung der Reihenfolge.

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:

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

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]

[1] Der *GenTypeMatcher* und der *SpecTypeMatcher* wurden gemeinsam in der Klasse `GenSpecTypeMatcher` umgesetzt. Der *ContentTypeMatcher* und der *ContainerTypeMat-*

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 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:

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

cher wurden gemeinsam in der Klasse `WrappedTypeMatcher` umgesetzt. (siehe angehängter Quellcode)

Weiterhin seien die folgenden Funktionen gegeben:

$$basesMD(MD) = \begin{array}{l} base(dRT, cRT) \cup base(cPT[0], dPT[pos[0]]) \\ \cup ... \cup base(cPT[pc], dPT[pos[pc]]) \end{array}$$

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

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

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

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 übergebenen Funktion `rateFunc` 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:

```
function applyHeuristic(proxies){
  return sort(proxies, rating)
}
```

1.2.2 Heuristik: Passed Tests Targets First (PTTF)

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. Zum einen müssen die Informationen bzgl. der bestandenen Tests ausgewertet werden können. Dazu wird von der Methode `evalProxy(proxy)` ein Objekt vom Typ `TestResult` zurückgegeben (siehe Abbildung 1.2.2). Die Attribute der Klasse `TestResult`, die für diese Heuristik relevant sind, werden in Tabelle 1.2.2 kurz beschrieben. Um das Objekt vom Typ `TestResult` mit korrekten Daten befüllen zu können, müssen die bestandenen Tests innerhalb der Methode `evalProxy(proxy)` mitgezählt werden.

Mit Hinblick auf die Laufzeit der Suche, muss in Frage gestellt werden, ob in diesem Fall alle Testfälle durchgeführt werden sollen, oder ob die Evaluation des Proxies, wie gehabt abbrechen soll, sofern ein Testfall nicht erfolgreich durch geführt wurde. Beide Ansätze erfordern zwei unterschiedliche Implementierungen (siehe Listing 1.2.2 und 1.2.2).

```
function evalProxy(proxy, T){
  passedTests = 0
  for( test : T ){
    if( !test.eval(proxy) ){
    \\ wenn ein Test fehlschlaegt, dann entspricht der
    \\ Proxy nicht den semantischen Anforderungen
      return new TestResult(passedTests, false)
    }
    passedTests++
  }
  return new TestResult(passedTests, true)
}
```

Listing 1: Variante 1: Abbruch bei fehlschlagendem Test

```
function evalProxy(proxy, T){
  passedTests = 0
  result = true
  for( test : T ){
    if( !test.eval(proxy) ){
    //alle Tests werden durchgefuehrt
      result = false
    }
    else {
      passedTests++
    }
  }
  return new TestResult(passedTests, result)
}
```

Listing 2: Variante 2: Alle Tests durchführen

Die Auswirkung auf die Laufzeit dieser beiden Varianten ist von der Komplexität der einzelnen Testfälle abhängig. Auf eine Reduktion der zu prüfenden Proxies hat die Wahl zwischen Variante 1 und Variante 2 keine Auswirkung. Da Variante 1 aufgrund dessen, dass nicht alle Testfälle durchgeführt werden, eine kürzere Laufzeit verspricht, wird die Evaluation der Heuristiken mit dieser Variante durchgeführt.

Um die erzielten Optimierungen vornehmen zu können, wird zu Beginn der semantischen Evaluation ein Objekt der Klasse `OptimizeInfo` (siehe Abbildung **??**) erzeugt. Dieses Objekt existiert während der gesamten Laufzeit der semantischen Evaluation und wird um die gewonnenen Informationen, die für dieses und weitere Heuristiken relevant sind, währenddessen angereichert.

Im Vergleich zu der Heuristik LMF aus dem vorherigen Abschnitt biete die Heuristik PTTF die Möglichkeit den auch die Reihenfolge der Proxies aus der aktuellen Iteration zu optimieren. Dazu muss die Methode `evalProxiesMitTarget` wie in Listing 1.2.2 erweitert werden.

```
function evalProxiesMitTarget(P, anzahl, T){
  proxies = relevantProxies(P,anzahl)
  testedProxies = []
  for( proxy : proxies ){
    testResult =  evalProxy(proxy, T)
    if ( testResult.passed() ){
      // passenden Proxy gefunden
      return proxy
    }
    else {
      testedProxies.add(proxy)
      if ( testResult.getPassedTests() > 0){
        leftProxies = proxies.removeAll(testedProxies)
        proxies = applyHeuristic(leftProxies)
    }
  }
```

```
  // kein passenden Proxy gefunden
  return null
}
```

Listing 3: Auswertung des Testergebnisses

Das Testergebnis wird bei der Applikation der Testfälle auf den Proxy erzeugt. Aufgrund dessen muss innerhalb des oben beschriebenen Pseudo-Codes für die semantische Evaluation die Methode `evalProxy(proxy)` angepasst werden. Zusätzlich muss die Methoden `eval(proxy)` die auf dem Test-Objekt selbst aufgerufen wird, soangepasst werden, dass sie ein Objekt mit

1.3 Heuristiken für den Ausschluss aus der weiteren semantischen Evaluation

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.

1.3.1 Heuristik: Pivot-Method Call Elimination

Bei dieser Heuristik handelt es sich um ein Ausschlussverfahren, welches auf früher gewonnene Erkenntnisse basiert.

1.3.2 Heuristik: Single-Method-Test Elimination