1 Explorationsalgorithmus

Mit diesen Voraussetzungen kann eine Komponente entwickelt werden, welche die Erwartungen der nachfragenden Komponente mit den bestehenden Funktionalitäten der angebotenen Komponenten zusammenbringt. In Abbildung 1 ist dies als Explorationskomponente dargestellt. Die Abhängigkeiten zu der nachfragenden und den angebotenen Komponenten ist nicht direkt vorhanden, da sie lediglich durch reflexive Aufrufe zur Laufzeit zustande kommen.

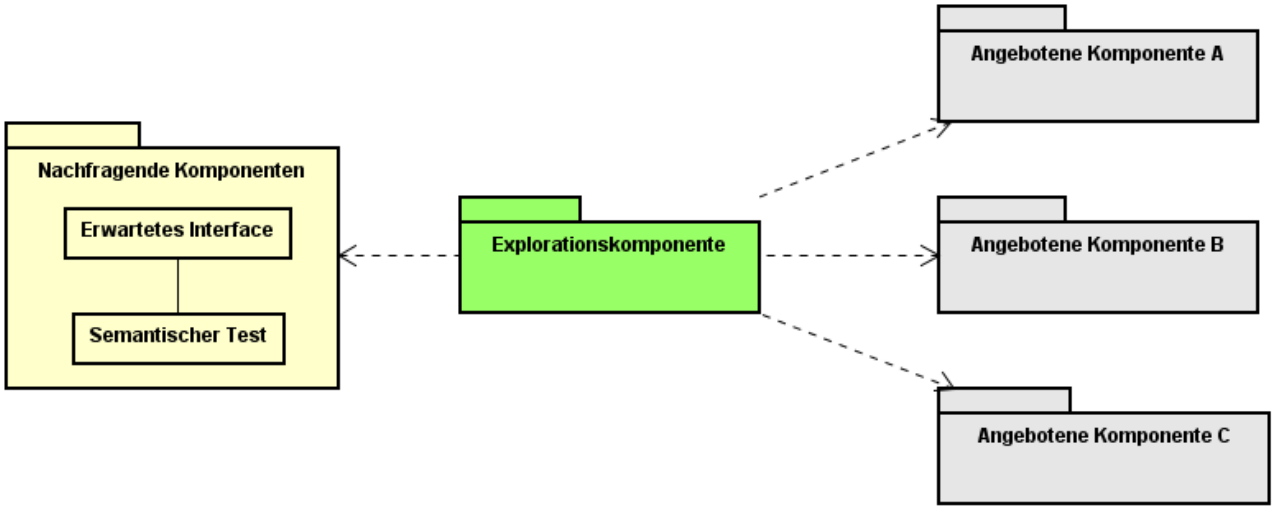

Abbildung 1: Allgemeiner Aufbau des System mit der Explorationskomponente

Um die Explorationskomponente anzusprechen, muss der Entwickler eine Instanz der Klasse DesiredComponentFinder, die von der Explorationskomponente bereitgestellt wird, erzeugen. Dabei müssen dem Konstruktor dieser Klasse zwei Parameter übergeben werden. Der erste Parameter ist eine Liste aller angebotenen Interfaces. Der zweite Parameter ist eine java.util.Function, über die die konkreten Implementierungen der angebotenen Interfaces ermittelt werden können. Die Suche wird mit dem Aufruf der Methode getDesiredComponent gestartet, welcher das erwartete Interface als Parameter übergeben werden muss. Somit kann ein Objekt der Klasse DesiredComponentFinder für mehrere Suchen mit unterschiedlichen erwarteten Interfaces verwendet werden.

Zu erwähnen ist noch, dass die in der nachfragenden Komponente spezifizierten Erwartungen

mitunter nur durch eine Kombination von angebotenen Komponenten erfüllt werden können. Aus diesem Grund wird innerhalb der Explorationskomponente eine so genannte benötigte Komponente erzeugt, in der das Zusammenspiel einer solchen Kombination von angebotenen Komponenten verwaltet wird. Ein solches Szenario ist Abbildung 2 zu entnehmen.

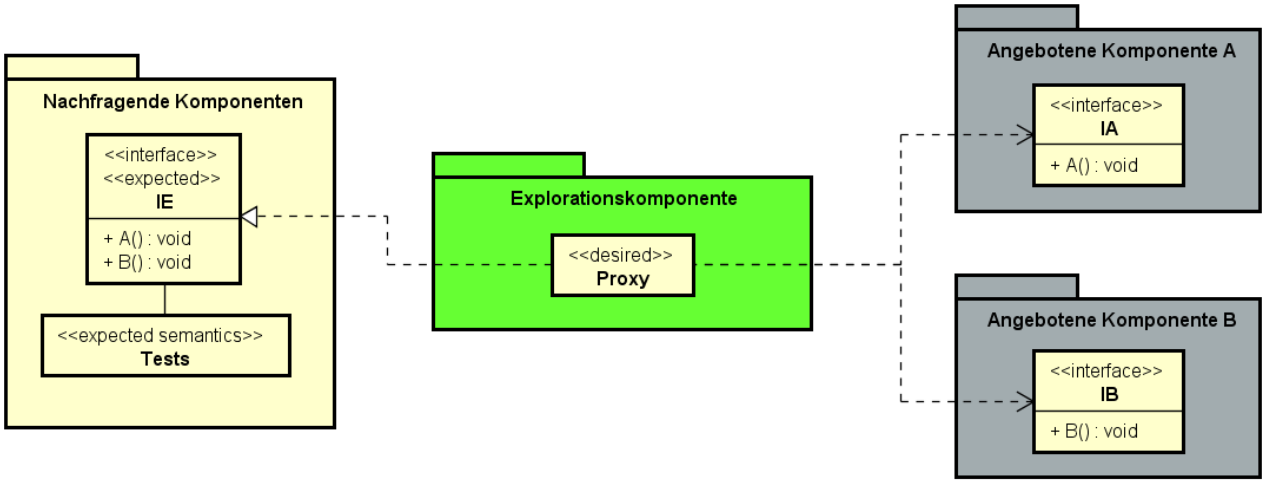

Abbildung 2: Kombination von angebotenen Komponenten

Die Suche nach einer benötigten Komponente innerhalb der Explorationskomponente erfolgt in zwei Schritten. Im ersten Schritt werden die angebotenen Interfaces hinsichtlich ihrer Struktur mit dem erwarteten Interface abgeglichen. Im zweiten Schritt werden die Ergebnisse aus dem ersten Schritt hinsichtlich der semantischen Tests überprüft. Dieser mehrstufige Ansatz baut auf der Arbeit von Hummel [Hum08] auf.

1.1 1. Stufe - Strukturelle Übereinstimmung

Wie in [Hum08] wird in der ersten Stufe der Suche versucht die angebotenen Interfaces herauszusuchen, die strukturell mit dem erwarteten Interface übereinstimmen. Zu diesem Zweck wird ein Structural-Type-Matcher verwendet, der in Abschnitt 1.1.1 beschrieben wird. Darüber hinaus werden weitere Type-Matcher verwendet (siehe Abschnitte **??**-**??**), die das Matching zweier Typen auf der Basis der Beziehung, in der diese beiden Typen zueinander stehen, feststellen. Allgemein beschrieben, kann durch jeden dieser Type-Matcher festgestellt werden, ob sich ein

Typ in einen anderen Typ konvertieren lässt.

Die Konvertierung erfolgt zur Laufzeit über die Erzeugung von Proxies, die ihre Methodenaufrufe delegieren. So wird bspw. bei der Konvertierung eines Objektes von TypA in ein Objekt von TypB ein Proxy-Objekt für TypB erzeugt, welches die Methodenaufrufe auf dem Objekt von TypA delegiert (vgl. Abbildung 2).

Hummel hatte hierzu bereits auf einige Matcher von Zaremski und Wing [ZW95] zurückgegriffen, die in dieser Arbeit ebenfalls zum Einsatz kommen (siehe Abschnitte **??**-**??**). Weiterhin wurde in [Hum08] ein Anwendungsfall für einen Matcher skizziert, der in der Lage ist Container-Typen zu ihren enthaltenen Typen zu matchen. Auf diese Idee wird in den Abschnitten **??** und **??** weiter eingegangen. Die Definitionen der Matcher beziehen sich vorrangig auf die Programmiersprache Java, weshalb grundlegend von einer nominalen Typkonformität auszugehen ist.

Die Typen seien in einer Bibliothek L in folgender Form zusammengefasst:

Regel	**Erläuterung**
$L ::= TD^*$	Eine Bibliothek L besteht aus einer Menge von Typdefinitionen.
$TD ::= PD \vert RD$	Eine Typdefinition kann entweder die Definition eines provided Typen (PD) oder eines required Typen (RD) sein.
PD ::= provided T extends T' $\{FD^*MD^*\}$	Die Definition eines provided Typen besteht aus dem Namen des Typen T, dem Namen des Super-Typs T' von T sowie mehreren Feld- und Methodendeklarationen.
RD ::= required T $\{MD^*\}$	Die Definition eines required Typen besteht aus dem Namen des Typen T sowie mehreren Methodendeklarationen.
FD ::= f : T	Eine Felddeklaration besteht aus dem Namen des Feldes f und dem Namen seines Typs T.
MD ::= m(T):T'	Eine Methodendeklaration besteht aus dem Namen der Methode m, dem Namen des Parameter-Typs T und dem Namen des Rückgabe-Typs T'.

Tabelle 1: Struktur für die Definition einer Bibliothek von Typen

Weiterhin sei die Relation $<$ auf Typen durch folgenden Regel definiert:

$$T < T' := \text{provided } T \text{ extends } T\text{'} \in L \vee (\text{provided } T \text{ extends } T\text{''} \in L \wedge T'' < T')$$

Darüber hinaus seien folgende Funktionen definiert:

$$felder(T) := \left\{ f : T' \;\middle|\; f : T\text{'} \text{ ist Felddeklaration von } T \right\}$$
$$methoden(T) := \left\{ m(T') : T'' \;\middle|\; m(T\text{'}):T\text{''} \text{ ist Methodendeklaration von } T \right\}\}$$

Das Matching eines Typs A zu einem Typ B wird durch die asymmetrische Relation $A \Rrightarrow B$ beschrieben. Dabei wird A auch als *Source-Typ* und B als *Target-Typ* bezeichnet.

Ein Proxy wird über die folgende Struktur beschrieben:

Regel	Erläuterung
$STRUCTPROXY$ ::= `structproxy for` R $\{TARGET*\}$	Ein struktureller Proxy wird für ein *required Interface* R mit einer Mengen von Targets erzeugt.
$TARGET$ ::= P $\{MDEL*\}$	Ein Target besteht aus dem Typ P des Targets (ein *provided Interface*) und einer Mengen von Methodendelegationen.
$MDEL ::= CALLM \rightarrow DELM$	Eine Methodendelegation besteht aus einer aufgerufenen Methode und aus einem Delegationsziel.
$CALLM$ ::= $m(SP) : STPROXY$	Eine aufgerufene Methode besteht aus dem Namen der Methode m, dem Parametertyp SP und einem Single-Target-Proxy zur Konvertierung des Rückgabetyps des Delegationsziels.
$DELM$ ::= $n(STPROXY) : R$	Ein Delegationsziel besteht aus demdem Namen der Methode n, dem Rückgabetyp TR und einem Single-Target-Proxy zur Konvertierung des Parametertyps der aufgerufenen Methode.
$STPROXY ::= NPX$	Ein Nominal-Proxy ist ein Single-Target-Proxy.
$STPROXY$ ::= `contentproxy for` P `with` P' $\{CEMDEL*\}$	Ein Content-Proxy ist ein Single-Target-Proxy, der für ein *provided Interface* P mit einem *provided Interface P'* als Target-Typ sowie einer Mengen von Content-Proxy-Methodendelegationen erzeugt wird.
$STPROXY$ ::= `containerproxy for` P `with` P' $\{f = NPX\}$	Ein Container-Proxy ist ein Single-Target-Proxy, der für ein *provided Interface* P mit einem *provided Interface P'* als Target-Typ sowie der Zuweisung eines Nominal-Proxies für den Target-Typ zu einem Feld f erzeugt wird.

Tabelle 2: Struktur für die Definition eines Proxies

Regel	Erläuterung
NPX ::= `subproxy for` *P* `with` *P′* {*NOMMDEL**}	Ein Sub-Proxy ist ein Nominal-Proxy, derfür ein *provided Interface P* mit einem *provided Interface P'* als Target-Typ sowie einer Mengen von Nominal-Proxy-Methodendelegationen erzeugt wird. Dabei gilt $P < P'$.
NPX ::= `simpleproxy for` *P*	Ein Simple-Proxy ist ein Nominal-Proxy, der aus einem Typen *P*, für den der Proxy erzeugt wird, besteht. Der Target-Typ ist in diesem Fall ebenfalls *P*. Alle Methoden werden in diesem Fall an den Target-Typ delegiert.
NOMMDEL ::= $m(SP) : SR \rightarrow m(TP) : TR$	Eine Nominal-Proxy-Methodendelegation besteht aus zwei Methoden mit demselben Namen m und den jeweiligen Parameter- und Rückgabetypen *SP* und *SR* bzw. *TP* und *TR*.
CEMDEL ::= $m(SP) : NPX \rightarrow f.m(NPX) : TR$	Eine Content-Proxy-Methodendelegation besteht aus zwei Methoden mit demselben Namen m, wobei die delegierte Methode (rechte Seite) auf einem Feld f des Target-Typs aufgerufen wird. Dabei besteht die aufgerufene Methode aus dem Parametertyp *SP* und einem Nominal-Proxy für den Rückgabetyp. Ferner besteht die delegierte Methode aus dem jeweiligen Rückgabetyp *TR* und einem Nominal-Proxy für den Parametertyp.

Tabelle 3: Struktur für die Definition eines Proxies (Fortsetzung)

Ein Ziel dieser Arbeit ist es Typen, die keinerlei Assoziationen zueinander haben, miteinander zu matchen und so zu konvertieren, dass darauf aufbauend die erwartete Semantik überprüfen werden kann. Hierfür soll wie auch in [Hum08] die strukturelle Übereinstimmung der beiden Typen genutzt werden. Diesem Zweck dient der StructuralTypeMatcher.

Um ein Beispiel für ein solches Matching und die daran anschließende Konvertierung zu geben, sei von folgender Bibliothek von Typen auszugehen:

```
provided Fire extends Object{}

provided FireState extends Object{
        isActive : boolean
}

provided Medicine extends Object{
        String getDescription()
}

provided Injured extends Object{
        void heal(Medicine med)
}

provided Patient extends Injured{}

provided FireFighter extends Object{
        FireState extinguishFire(Fire fire)
}

provided Doctor extends Object{
        void heal( Patient pat, Medicine med )
}

provided MedCabinet extends Object{
        med : Medicine
}

required MedicalFireFighter {
        void heal( Injured injured, MedCabinet med )
        boolean extinguishFire( Fire fire )
}
```

Listing 1: Bibliothek von Typen

Ein Proxy für das *required Interface* $MedicalFireFighter$ könnte in diesem Szenario folgende Struktur aufweisen:

```
structproxy for MedicalFireFither{
        FireFighter {
         extinguishFire(Fire):
                containerproxy for FireState with boolean {
                 isActive = simpleproxy for boolean
                }
                → extinguishFire(simpleproxy for Fire):boolean
        }
```

```
        Doctor {
         heal(Injured, MedCabinet):simpleproxy for void
                → heal(subproxy for Patient with Injured{
                         heal(Medicine): void
                                 → heal(Medicine):void

                    }, contentproxy for Medicine with MedCabinet{
                         getDescription(): simpleproxy for String
                                 → med.getDescription():String
                       }):void
        }
}
```

Listing 2: Proxy für MedicalFireFighter

Um die Regeln für das Matching und der darauf aufbauenden Konvertierung zu beschreiben, seien unterschiedliche Matcher definiert.

1.1.1 StructuralTypeMatcher

StructuralTypeMatcher

Das strukturelle Matching zwischen einem *required Interface* R und einem *provided Interface* P ist gegeben, sofern eine Methode aus R zu einer Methode aus P gematcht werden kann. Die Menge der aus R in P gematchten Methoden wird wie folgt beschrieben:

$$structM(R,P) := \left\{ m(T) : T' \in methoden(R) \;\middle|\; \begin{array}{l} \exists n(S) : S' \in methoden(P). \\ S \Rightarrow_{egsc} \wedge T' \Rightarrow_{egsc} S' \end{array} \right\}$$

Da die Notation es nicht hergibt, ist zusätzlich zu erwähnen, dass die Reihenfolge der Parameter in m und n irrelevant ist.

Die Relation $\Rightarrow_{egsc}$ wird durch die übrigen Matcher in folgender Form beschrieben:

$$\frac{\begin{array}{r} A \Rightarrow_{exact} B \vee A \Rightarrow_{spec} B \vee A \Rightarrow_{gen} B \\ \vee A \Rightarrow_{container} B \vee A \Rightarrow_{content} B \end{array}}{A \Rightarrow_{egsc} B}$$

Das strukturelle Matching von R und P wird dann durch folgende Regel beschrieben.

$$\frac{structM(R,P) \neq \emptyset}{R \Rightarrow_{struct} P}$$

Ein struktureller Proxy für ein *required Interface* R aus einer Menge von *provided Interfaces* P wird durch folgende Regeln und Nebenbedingungen beschrieben:

Regel	**Nebenbedingungen**
$STRUCTPROXY ::=$ `structproxy for` R $\{TARGET_1 \ \ldots \ TARGET_n\}$	$typ(STRUCTPROXY) = R$ $methoden(STRUCTPROXY) =$ $cmethoden(TARGET_1) \cup \ldots \cup cmethoden(TARGET_n)$ $methoden(R) = methoden(STRUCTPROXY)$
$TARGET ::=$ $P \ \{MDEL_1 \ \ldots \ MDEL_n\}$	$typ(TARGET) = P$ $cmethoden(TARGET) =$ $cmethode(MDEL_1) \cup \ldots \cup cmethode(MDEL_n)$ $dmethoden(TARGET) =$ $dmethode(MDEL_1) \cup \ldots \cup dmethode(MDEL_n)$ $dmethoden(TARGET) \subseteq methoden(P)$
$MDEL ::=$ $CALLM \rightarrow DELM$	$cmethode(MDEL) = methode(CALLM)$ $dmethode(MDEL) = methode(DELM)$ $paramTargetTyp(DELM) = paramTyp(CALLM)$ $returnTargetTyp(CALLM) = returnTyp(DELM)$
$CALLM ::=$ $m(SP) : STPROXY$	$SR = typ(STPROXY)$ $methode(CALLM) = m(SP) : SR$ $paramTyp(CALLM) = SP$ $targetTyp(STPROXY) = returnTargetTyp(CALLM)$
$DELM ::=$ $n(STPROXY) : R$	$DP = typ(STPROXY)$ $methode(DELM) = n(DP) : R$ $returnTyp(DELM) = R$ $targetTyp(STPROXY) = paramTargetTyp(DELM)$

Tabelle 4: Grammatik für die Definition eines Proxies

Regeln für das Nonterminal $STPROXY$ unterliegen Nebenbedingungen, die teilweise erst unter Zuhilfenahme der folgenden Matcher erfüllt werden können.

ExactTypeMatcher

Die Matchingrelation für diesen Matcher wird durch folgende Regel beschrieben:

$$\frac{}{T \Rightarrow_{exact} T}$$

Ein Proxy für einen Typ T, der mit demselben Typ als Target-Typ erzeugt werden soll, ist ein Simple-Proxy. Die Regeln für den Simple-Proxy, sind im folgenden Abschnitt zum *GenType-Matcher* beschrieben.

GenTypeMatcher

Die Matchingrelation für diesen Matcher wird durch folgende Regel beschrieben:

$$\frac{T > T'}{T \Rightarrow_{gen} T'}$$

Ein Proxy für einen Typ T, der mit einem Typen-Typ T' mit $T \Rightarrow_{gen} T'$ erzeugt werden soll, ist ein Simple-Proxy und wird über die folgenden Regeln und Nebenbedingungen beschrieben:

Regel	**Nebenbedingungen**
$STPROXY ::= NPX$	$typ(STPROXY) = typ(NPX)$ $targetTyp(STPROXY) = targetTyp(NPX)$
$NPX ::=$ `simpleproxy for` P	$targetTyp(NPX) \Rightarrow_{gen} P$ $typ(NPX) = P$ $methoden(NPX) = methoden(P)$

Tabelle 5: Regeln und Nebenbedingungen für Simple-Proxies

SpecTypeMatcher

Die Matchingrelation für diesen Matcher wird durch folgende Regel beschrieben:

$$\frac{T < T'}{T \Rightarrow_{spec} T'}$$

Ein Proxy für einen Typ T, der mit einem Target-Typ T' mit $T \Rightarrow_{spec} T'$ erzeugt werden soll, ist ein Sub-Proxy und wird durch die folgenden Regeln und Nebenbedingungen beschrieben:

Regel	Nebenbedingungen
$NPX ::=$ `subproxy for` P `with` P' $\{NOMMDEL_1$ `...` $NOMMDEL_n\}$	$targetTyp(NPX) = P'$ $typ(NPX) = P$ $P \Rightarrow_{spec} P'$ $methoden(NPX) = cmethode(NOMMDEL_1) \cup$ $\ldots \cup cmethode(NOMMDEL_n)$ $methoden(NPX) \subseteq methoden(P)$ $methoden(P') \supseteq dmethode(NOMMDEL_1) \cup$ $\ldots \cup dmethode(NOMMDEL_n)$
$NOMMDEL ::=$ $m(SP) : SR \rightarrow$ $m(TP) : TR$	$SP >= TP$ $SR <= TR$ $cmethode(MOMMDEL) = m(SP) : SR$ $dmethode(MOMMDEL) = m(TP) : TR$

Tabelle 6: Regeln und Nebenbedingungen für Sub-Proxies

ContentTypeMatcher

Die Matchingrelation für diesen Matcher wird durch folgende Regel beschrieben:

$$\frac{\exists f : T'' \in felder(T').T \Rightarrow_{esg} T''}{T \Rightarrow_{content} T'}$$

Für die Relation $\Rightarrow_{esg}$ gilt dabei:

$$\frac{T \Rightarrow_{exact} T' \vee T \Rightarrow_{gen} T' \vee T \Rightarrow_{spec} T'}{T \Rightarrow_{esg} T'}$$

Ein Proxy für einen Typ P, der mit einem Target-Typ P' mit $P \Rightarrow_{content} P'$ erzeugt werden soll, ist ein Content-Proxy und wird durch die folgenden Regeln und Nebenbedingungen beschrieben:

Regel	Nebenbedingungen
$STPROXY ::=$ `contentproxy for` P `with` P' $\{CEMDEL_1$ ... $CEMDEL_n\}$	$typ(STPROXY) = P$ $targetTyp(STPROXY) = P'$ $P \Rightarrow_{content} P'$ $methoden(STPROXY) = cmethode(CEMDEL_1) \cup$ $\ldots \cup cmethode(CEMDEL_n)$ $methoden(STPROXY) \subseteq methoden(P)$ $containerType(CEMDEL_1) = P'$ $\ldots$ $containerType(CEMDEL_n) = P'$
$CEMDEL ::=$ $CECALLM \rightarrow$ $f.CEDELM$	$f : FT \in felder(containerType(CEMDEL))$ $methode(CEDELM) \in methoden(FT)$ $paramTargetTyp(CEDELM) = paramTyp(CECALLM)$ $returnTargetTyp(CECALLM) = returnTyp(CEDELM)$
$CECALLM ::=$ $m(SP) : NPX$	$paramTyp(CECALLM) = SP$ $SR = typ(NPX)$ $targetTyp(NPX) = returnTargetTyp(CECALLM)$ $methode(CECALLM) = m(SP) : SR$
$CEDELM ::=$ $m(NPX) : TR$	$returnTyp(CEDELM) = TR$ $TP = typ(NPX)$ $targetTyp(NPX) = paramTargetTyp(CEDELM)$ $methode(CEDELM) = m(TP) : TR$

Tabelle 7: Regeln und Nebenbedingungen für Contentproxies

ContainerTypeMatcher

Die Matchingrelation für diesen Matcher wird durch folgende Regel beschrieben:

$$\frac{\exists f : T'' \in felder(T).T'' \Rightarrow_{esg} T'}{T \Rightarrow_{container} T'}$$

Ein Proxy für einen Typ P, der mit einem Target-Typ P' mit $P \Rightarrow_{container} P'$ erzeugt werden soll, ist ein Container-Proxy und wird durch die folgenden Regeln und Nebenbedingungen beschrieben:

Regel	Nebenbedingungen
$STPROXY$ ::= `containerproxy for` P `with` P' $\{f = NPX\}$	$targetTyp(STPROXY) = P'$ $typ(STPROXY) = P$ $P \Rightarrow_{container} P'$ $f : FT \in felder(P)$ $targetTyp(NPX) = P'$ $typ(NPX) = FT$

Tabelle 8: Regeln und Nebenbedingungen für Container-Proxies

Literatur

[Hum08] Hummel, Oliver: *Semantic Component Retrieval in Software Engineering.* , April 2008.

[ZW95] Zaremski, Amy Moormann Jeannette M. Wing: *Signature Matching: A Tool for Using Software Libraries.* ACM Trans. Softw. Eng. Methodol., 4(2):146?170, 1995.