

Invasive Komponentenkonfiguration

Volker Kuttruff

Forschungszentrum Informatik FZI
Haid-und-Neu-Str. 10-14
76131 Karlsruhe, Germany
kuttruff@fzi.de

Zusammenfassung Um einzelne Komponenten bis hin zu ganzen Softwaresystemen in verschiedenen Einsatzkontexten verwenden zu können, müssen diese zuvor gemäß der Anwenderanforderungen konfiguriert werden. Die Praxis zeigt, dass hierzu invasive Anpassungen notwendig sind, eine rein modulare Komposition also nicht ausreicht. In diesem Papier stellen wir daher ein Verfahren zur invasiven Konfiguration generischer Komponenten vor. Dieses Verfahren erweitert Typgenerizität, indem nicht nur Typparameter, sondern allgemeine Programmfragmente und Metaoperatoren zur Parametrisierung zugelassen werden. Darüberhinaus wird ein Verfahren gezeigt, wie Konflikte bei der Anwendung von Metaoperatoren erkannt und aufgelöst werden können.

1 Einleitung

Heutzutage ist man daran interessiert, die Wiederverwendbarkeit möglichst grobgranularer Softwareartefakte bis hin zu vollständigen Softwaresystemen zu erhöhen. Damit dies gelingt, müssen die beteiligten Softwareartefakte bezüglich bestimmter Freiheitsgrade (= optionale oder alternative Merkmale (engl. features) eines Systems) *generisch* sein, so dass sie in gewissem Umfang an die unterschiedlichen Einsatzkontexte anpassbar sind. Die Bestimmung und Organisation dieser Freiheitsgrade sowie der zugehörigen Softwareartefakte ist hierbei Teil der sog. Produktlinienentwicklung.

In diesem Papier wird der technische Teil unserer laufenden Arbeiten an einer Methodik zur merkmalsgetriebenen, invasiven Konfiguration von auf generischen Komponenten basierenden Softwaresystemen vorgestellt. Die Konfiguration erfolgt hierbei statisch, d.h. Zieldarstellung ist Quelltext. Die Methodik kann somit als eine konkrete Ausprägung der Applikationsentwicklung innerhalb des Produktlinienansatzes bzw. der generativen Vorgehensweise [2] angesehen werden.

Nicht-invasive, d.h. auf die Komposition von Schnittstellenelementen eingeschränkte Techniken zum Binden der Freiheitsgrade versagen im allgemeinen Fall, da die notwendigen Anpassungen in vielen Fällen nicht lokal bzgl. der genutzten Dekompositionstechnik sind. Dies liegt daran, dass die Implementierung eines solchen Merkmals nicht modular vorliegt, sondern in Form von verteilten Programmfragmenten.

Die Beschreibung einer konkreten Konfiguration eines Systems bzw. einer Komponente sollte deklarativ erfolgen, d.h. es soll durch den Nutzer nur spezifiziert werden, *was* erreicht werden soll, aber nicht *wie* dies im einzelnen technisch durchgeführt werden muss. Darüberhinaus sollte die Effizienz des resultierenden Codes mit manueller Implementierung vergleichbar sein. Insbesondere sollen keine unnötigen Indirektionen hinzugefügt werden. Durch eine nicht als fehlerhaft erkannte Konfigurationsspezifikation darf der Vertrag einer Komponente nicht gebrochen werden. Darüberhinaus muss der Bau einer konkreten Systemkonfiguration automatisierbar sein.

Der Rest dieses Papiers gestaltet sich wie folgt. In Abschnitt 2 wird der verfolgte Ansatz zur invasiven Konfiguration generischer Komponenten vorgestellt, indem das zugrunde liegende Modell und die notwendigen Konzepte zur Durchführung der Konfiguration beschrieben werden. Abschnitt 3 beschäftigt sich detaillierter mit Problemen und Lösungsmöglichkeiten, die während der Konfiguration infolge von Abhängigkeiten zwischen Merkmalen auftreten. Einen kurzen Überblick über verwandte Arbeiten bietet Abschnitt 4. Im letzten Abschnitt 5 wird schließlich noch eine kurze Zusammenfassung gegeben.

2 Ansatz

Ausgangspunkt des Ansatzes ist die in vielen Programmiersprachen zu findende Typgenerizität. Es handelt sich hierbei um eine einfache invasive, d.h. nicht auf Schnittstellenelemente beschränkte Technik. Typgenerizität erfüllt alle in Abschnitt 1 geforderten Eigenschaften. Für einen Nutzer einer generischen Klasse ist die Spezifikation deklarativ. Darüber hinaus hat sie in Verbindung mit Typschränken eine wohldefinierte Semantik. Leider ist Typgenerizität auf die Parametrisierung von Klassen mit Typen, oder genauer mit Typpräferenzfragmenten, eingeschränkt.

Der im Folgenden vorgestellte Ansatz erweitert Typgenerizität, indem nicht nur Typpräferenzfragmente zur Parametrisierung von Klassen zugelassen werden, sondern beliebige Programmfragmente zur Parametrisierung von gröbergranularen Programmfragmenten genutzt werden können.

2.1 Systemmodell

Wir betrachten ein Programm als eine *hierarchische Komposition von Programmfragmenten*. Der Bau einer konkreten Programmkonfiguration ist somit die Selektion einer Menge von geeigneten Programmfragmenten \mathcal{F}_S aus einer zur Verfügung stehenden Programmfragment-Grundmenge \mathcal{F} , und der anschließenden hierarchischen Komposition dieser Fragmente zum Gesamtprogramm. Die Komposition der einzelnen Programmfragmente geschieht hierbei an sogenannten *Webepunkten*¹.

¹ Webepunkte sind vergleichbar zu den aus AspectJ bekannten *join points* bzw. den in [1] eingeführten *hooks*.

Damit die einzelnen Programmfragmente überhaupt komponierbar sind, müssen diese eine Kompositionsschnittstelle besitzen, die festlegt, welche Dinge benötigt werden (z.B. Kontrollfluss), aber auch welche Dinge zur Verfügung gestellt werden (z.B. Variablendeklarationen). Darüber hinaus muss ein Vertrag die Kompositionsbedingungen festlegen. So ist es zum Beispiel nicht zulässig, ein Ausdrucksfragment direkt mit einem Paketfragment zu komponieren. Ein Programmfragment kann somit als eine abgeschlossene Kompositionseinheit angesehen werden, welche die Implementierungsinterna verbirgt.

Die Beschreibung einer konkreten Programmkonfiguration basierend auf Programmfragmenten ist ein Kompositionsplan, der beschreibt, welche Programmfragmente wie komponiert werden müssen. Die Komposition der Fragmente erfolgt dabei bottom-up, d.h. gröbergranulare Einheiten werden aus feingranularen Einheiten erstellt. Die einzelnen Programmfragmente werden dabei im sog. *Konfigurationsmodell* bezüglich des zu konfigurierenden Programms organisiert.

Definition 1. *Das Konfigurationsmodell \mathcal{K}_P^x eines Programms P ist die Menge \mathcal{F}_S aller aus \mathcal{F} selektierten, (teil-)verbundenen Programmfragmente. Das Konfigurationsmodell \mathcal{K}_P durchläuft während des Konfigurationsvorgangs die Zustände $\mathcal{K}_P^0 \dots \mathcal{K}_P^n$. Jedes Konfigurationsmodell \mathcal{K}_P^x beschreibt die konkrete Belegung der momentanen Programminterpretation \mathcal{I}_P .*

Das Konfigurationsmodell \mathcal{K}_P ist im Wesentlichen eine Ausprägung des in [6] und [7] vorgestellten Adaptionmodells \mathcal{A} . Dieses modelliert Softwaresysteme in Form eines Baums von Programmfragmenten, d.h. es ist im Wesentlichen eine abstraktere Sicht auf den AST. Die möglichen Knotentypen besitzen dabei Attribute, welche die Ergebnisse verschiedenster statischer Analysen aufnehmen, angefangen bei einfachen Namens- und Typanalysen bis hin zu Querverweisinformationen und Metrikberechnungen. Weiterhin definiert das Adaptionmodell sog. semantische Transformationen, mit deren Hilfe der Baum an explizit deklarierten sowie implizit gegebenen Webepunkten verändert werden kann. Die Transformationen decken dabei sowohl strukturelle Transformationen wie z.B. die aus [4] bekannten Restrukturierungen ab, aber auch verhaltensändernde Transformationen wie z.B. die aus ASPECTJ bekannten `advice`-Anweisungen werden unterstützt. Die Implementierung dieses Adaptionmodells \mathcal{A} ist durch das Werkzeug INJECT/J [5] gegeben, welches zusätzlich eine zur Spezifikation von Quelltexttransformationen angepasste Skriptsprache bereitstellt.

Ein wesentlicher Unterschied zwischen \mathcal{A}_P und \mathcal{K}_P^x ist allerdings, dass \mathcal{K}_P^x nicht zu jeder Zeit ein Baum ist. Der Baum wird bottom-up aufgebaut, repräsentiert durch die Zustände $\mathcal{K}_P^0 \dots \mathcal{K}_P^n$. Die in \mathcal{A} beschriebenen Attribute werden hierbei sukzessive berechnet, sobald dies möglich ist. Es gilt $\mathcal{K}_P^n = \mathcal{A}_P$.

Insbesondere für große Programme wird der Kompositionsplan sehr groß. Zudem muss er für jede Programmkonfiguration neu erstellt werden. Ziel muss es daher sein, neben den in der Programmfragment-Grundmenge \mathcal{F} bereitgestellten Programmfragmenten auch Teile des Kompositionsplans wiederzuverwenden. Dies funktioniert, da sich der Kompositionsplan in einen für alle Programmkonfigurationen invarianten Teil und einen die Unterschiede zwischen den Konfigurationen beschreibenden variablen Teil zerlegen lässt.

2.2 Generische Fragmenttypen

In oo-Sprachen beschreiben generische Klassen, wie durch geeignete Parametrisierung mit Typen (genauer: Referenzen auf Typen) konkrete Klassenfragmente erzeugt werden können. Der invariante Teil des Bauplans ist hier durch die Implementierung der generischen Klasse gegeben, der variable Teil durch entsprechende Platzhalter (Typparameter). Generische Klassen stellen somit einen neuen (Meta-)Typ dar, der beschreibt, wie konkrete Ausprägungen, d.h. konkrete Klassen, erzeugt werden können.

Die bereits zu Beginn des Abschnitts 2 vorgestellte verallgemeinerte Sichtweise auf Typgenerizität führt zusammen mit dem Systemmodell zum Begriff des *generischen Fragmenttyps*.

Definition 2. *Ein generischer Fragmenttyp ist ein Tripel $(S, \mathcal{B}, \mathcal{I})$ mit einer Schnittstelle S , einem Bauplan \mathcal{B} und einer generischen Implementierung \mathcal{I} mit explizit deklarierten Webepunkten. Die Schnittstelle $S = (n, \mathcal{ST}, \mathcal{P}, \mathcal{C})$ besteht wiederum aus einem Namen n , welcher den Fragmenttyp eindeutig identifiziert, einer Liste von Obertypen \mathcal{ST} , einer Menge von getypten formalen Konfigurationsparametern \mathcal{P} sowie einer Menge von Konfigurationsbedingungen \mathcal{C} .*

Der Name n identifiziert den generischen Fragmenttyp eindeutig und beschreibt zusammen mit den Obertypen \mathcal{ST} die Semantik des Fragments. Die möglichen Basis-Obertypen werden dabei durch die Sprachsyntax vorgegeben, wie z.B. `ClassFragment`, `MethodFragment` etc. und entsprechen den in [7] und [6] beschriebenen Knotentypen des dort spezifizierten Adaptionmodells. Im Gegensatz zu den festen Knotentypen des Adaptionmodells lässt sich durch die Untertypbeziehung zusammen mit den Namen n eine weitergehende Semantik mit den Fragmenten assoziieren. So wird mit einem generischen Fragmenttyp `HashSetRemoveElementMethodFragment` auch eine gewisse Ausführungssemantik beschrieben, im Gegensatz zum sehr allgemeinen Basistyp `MethodFragment`.

Die formalen Parameter aus \mathcal{P} beschreiben einerseits, mit welchen konkreten Fragmenten oder Metaoperatoren (siehe Abschnitt 2.3) dieser generische Fragmenttyp während der Instantiierung parametrisiert werden kann, andererseits welche Fragmente von diesem Fragment zur Verfügung gestellt werden. Letztere können notwendig sein, wenn ein zur Parametrisierung notwendiges konkretes Fragment seinerseits als Ausprägung eines generischen Fragmenttyps vorab erzeugt werden muss. Die durch die formalen Parameter beschriebene Schnittstelle lässt sich in eine *geforderte* Schnittstelle und in eine *bereitgestellte* Schnittstelle aufteilen.

Der Bauplan \mathcal{B} beschreibt die zur Instantiierung des generischen Fragmenttyps notwendigen Aktionen, wie zum Beispiel das Binden der an der Konfigurationsschnittstelle bereitgestellten Fragmente an die explizit deklarierten Webepunkte der generischen Implementierung.² Der Bauplan \mathcal{B} kapselt somit die

² Der Bauplan \mathcal{B} wird bei Typgenerizität im Allgemeinen durch den Übersetzer bereitgestellt.

Spezifikation der variablen Teile des Kompositionsplans, bezogen auf das Programmfragment, wohingegen die generische Implementierung \mathcal{I} den invarianten Teil beschreibt.

Eine *generische Komponente* ist in der vorgestellten Sichtweise ein generischer Fragmenttyp mit ausreichendem Abstraktionsniveau (z.B. Klassenfragmente, Paketfragmente). Das folgende Beispiel gibt einen Eindruck, wie generische Fragmenttypen in unserem Ansatz spezifiziert werden können. Die verwendete Syntax ist hierbei eine Erweiterung der Transformationssprache unseres Quelltext-Transformationswerkzeugs INJECT/J³. Eine ausführlicheres Beispiel in Form einer initialen Fallstudie ist in [8] zu finden.

```

fragment HashSet <
  TypeReferenceFragment Tin, required;
  HashSetPutElementMethodFragment <TypeReferenceFragment Tout,
    Attribute ds> putMethod, optional;
  HashSetRemoveElementMethodFragment <TypeReferenceFragment Tout,
    Attribute ds> removeMethod, optional;
  SynchronizeAttributeAccessOperator <Attribute ds> synchronizer, optional;
  boolean immutable = false, required
> extends ClassFragment,
conditions { immutable -> putMethod==null && removeMethod==null &&
  forall a in data.referenceAccesses.filter(
    [x|return !x.surroundingMethod.isConstructor]):
    a.isReadAccess; };
{
  construction { /* construction plan */ }
  implementation { /* generic implementation */ }
}

```

2.3 Metaoperatoren

Aufbauend auf Programmfragmenten bzw. durch Instantiierung geeigneter generischer Fragmenttypen lässt sich prinzipiell jedes Programm zusammenbauen, insbesondere also auch die variablen Teile eines Programms binden. Die Praxis zeigt jedoch, dass die Implementierung zahlreicher Merkmale nicht durch modulares Einsetzen bzw. Austausch einzelner Programmfragmente erfolgen kann. Die Implementierung dieser Merkmale schlägt sich bzgl. der gewählten Dekompositionstechnik (hier: oo-Dekomposition) an vielen Stellen nieder. Die explizite Spezifikation aller Webepunkte sowie u.U. sogar ähnlicher einzufügender Programmfragmente für diese querliegenden Merkmale ist allerdings in der Praxis unbequem und oftmals nicht oder nur schwer beherrschbar. Als Lösung für dieses Problem führen wir alternativ zu Programmfragmenten *getypte Metaoperatoren* ein.

Definition 3. Ein *getypter Metaoperator* ist ein Tripel (S, T, M) mit einer Schnittstelle S , einer Menge von höherwertigen Spezifikationen bzw. Fragment-schablonen T sowie einem Metaprogramm M . Die Schnittstelle $S = (n, ST, \mathcal{P}, pre, post)$ besteht aus einem Namen n , einer Liste von Obertypen ST , einer Menge von formalen Parametern \mathcal{P} sowie den zur Anwendung des Metaoperators notwendigen Vorbedingungen pre und die nach der Anwendung zugesicherten Nachbedingungen $post$.

³ <http://injectj.sf.net>

Das Metaprogramm \mathcal{M} beschreibt, wo und wie höherwertige Spezifikationen bzw. Fragmentschablonen aus \mathcal{T} aufbauend auf Kontextinformationen (z.B. an formale Parameter gebundene Fragmente oder analysierbare Eigenschaften des Programms) instantiiert und mit Hilfe von Generierung (bei höherwertigen Spezifikationen) oder syntaktischer Transformationen eingefügt werden müssen. Um das Metaprogramm besser strukturieren zu können, kann das Metaprogramm in Einzelfunktionen zerlegt werden und auf weitere Metaoperatoren zurückgreifen.

Wie im Fall generischer Programmfragmente beschreibt ein Metaoperator einen neuen (Meta-) Typ. Ein Metaoperator ist somit vergleichbar mit Closures bzw. Blöcken in SmallTalk auf der Metaebene. Der Typ eines Metaoperators beschreibt seine Semantik, d.h. die Effekte der Ausführung auf ein Programm.

Da jeder Metaoperator einen oder mehrere Obertypen besitzt, ist es auch möglich, eine entsprechende Typhierarchie aufzubauen. Ein Beispiel hierfür sind Metaoperatoren, welche verschiedene Synchronisationsstrategien implementieren (z.B. exklusiver Zugriff, Priorisierung von parallelen Leseoperationen, Priorisierung von Schreiboperationen). Insbesondere kann ein Metaoperator auch einen Fragmenttyp als Obertyp haben. In diesem Fall kann der Metaoperator anstelle eines konkreten Fragments zur Parametrisierung eines generischen Fragmenttyps benutzt werden. Hierbei werden mit dem Typ die folgenden Eigenschaften des Metaoperators assoziiert:

- Im Falle einer Generierung eines Fragments durch den Metaoperator aus einer höherwertigen Spezifikation ist das resultierende Fragment kompatibel zum Obertyp.
- Falls die Ausführung des Metaoperators in einer Reihe von Transformationen resultiert, so wird durch den Metaoperator garantiert, dass das Ergebnis der Transformation den gleichen Effekt hat, wie wenn man mit einem konkreten Fragment des Obertyps parametrisiert hätte. Ein einfaches Beispiel ist ein Metaoperator, welcher ein Methodenfragment an allen Aufrufstellen einfügt, anstatt die Methode tatsächlich in der Klasse einzufügen.

Mit Hilfe der Typisierung lassen sich somit die semantischen Auswirkungen des Metaoperators abschätzen. Dies ist vor allem bei sog. offenen Freiheitsgraden hilfreich, d.h. der Freiheitsgrad wurde bei der generischen Implementierung zwar erkannt und berücksichtigt, alle konkreten Belegungen in Form entsprechender Programmfragmente/Metaoperatoren konnten zu diesem Zeitpunkt aber noch nicht ermittelt werden. Die Abschätzung semantischer Auswirkungen ist bei allgemeinen Metaprogrammieren nicht in allen Fällen möglich. Mit Hilfe von Metaoperatoren ist es darüber hinaus möglich, die Implementierung von Merkmalen zu kapseln, wobei sich die Implementierung sowohl lokal als auch querliegend bzgl. der benutzten Dekompositionstechnik niederschlagen kann.

3 Ausführungszeitpunkte und Ausführungsreihenfolge von Metaoperatoren

In den folgenden Abschnitten werden zwei Probleme genauer betrachtet, die beim Einsatz von Metaoperatoren auftreten: die Bestimmung eines geeigneten

Ausführungszeitpunkts sowie die Bestimmung einer geeigneten Ausführungsreihenfolge. Letzteres ist notwendig, da die Ergebnisse einer unterschiedlichen Reihenfolge im Allgemeinen nicht semantisch äquivalent sind.

3.1 Ausführungszeitpunkte

Da der Bau eines Programms bottom-up erfolgt, kann es vorkommen, dass während der Instantiierung eines generischen Fragmenttyps ein als Konfigurationsparameter übergebener Metaoperator noch nicht ausgeführt werden kann, da noch nicht alle von ihm benötigten Informationen zur Verfügung stehen – der Metaoperator hat zu diesem Zeitpunkt nur eine lokale Sicht auf den generischen Fragmenttyp bzw. dessen Ausprägung. Ein Beispiel ist ein Metaoperator, welcher eine Methode an den Aufrufstellen einfügt. Dies kann erst erfolgen, nachdem alle tatsächlichen Aufrufstellen (auch außerhalb des Fragments) bekannt sind.

Der spätestmögliche Zeitpunkt zur Ausführung eines Metaoperators ist dann gegeben, wenn der Programmfragmentbaum vollständig erstellt wurde, d.h. alle Programminformationen zur Verfügung stehen. Aus praktischer Sicht ist man allerdings an einer möglichst frühen Ausführung interessiert, sei es um die Übergabe teil- bzw. fertig konfigurierter Fragmente an andere Nutzer zu erleichtern, bereits früh mit ersten Tests beginnen zu können, oder um den Bauprozess besser zu parallelisieren.

Der frühestmögliche Ausführungszeitpunkt eines Metaoperators *op* ist der Zeitpunkt, wenn sich die Menge der betroffenen Webepunkte nicht mehr ändert. Dies kann aber erst bei Betrachtung des Gesamtsystems mit Sicherheit gesagt werden. Die Grundidee zur Lösung dieses Problems liegt in einer Kombination aus der Interpretation der zur Bestimmung konkreter Webepunkte notwendigen Webepunktspezifikationen und der Nutzung der Sichtbarkeitsregeln der zugrunde liegenden Programmiersprache. Durch die Interpretation der Webepunktspezifikationen wie z.B. `declaredElement.referenceAccesses` kann zunächst abgeschätzt werden, ob ein Metaoperator bezüglich eines Fragments nicht-lokale Auswirkungen hat. Ist dies der Fall, so kann bestimmt werden, welche Nutzungsstellen eines innerhalb des Fragments deklarierten Elements hierfür verantwortlich sind. (z.B. `declaredElement` für die nicht-lokalen Webepunkte `declaredElement.referenceAccesses`). Unter Nutzung der Sichtbarkeitsregeln kann nun konservativ abgeschätzt werden, ab wann keine Referenzen auf das deklarierte Element mehr auftreten können.

3.2 Ausführungsreihenfolge

Während die in Abschnitt 3.1 beschriebene Bestimmung von Ausführungszeitpunkten im Wesentlichen eine Optimierung für die Bauphase des Programms ist, muss die Bestimmung der Ausführungsreihenfolge von Metaoperatoren auf jeden Fall erfolgen. Auch im AOP-Umfeld ist man sich bewusst, dass die Reihenfolge der Aspektanwendung mitunter wichtig ist und über die durch AspectJ bereitgestellten Möglichkeiten hinaus genauer untersucht werden muss [3]. Das im Folgenden vorgestellte Verfahren ist im Wesentlichen eine Erweiterung und

Anpassung des in [3] vorgestellten allgemeinen Rahmens zur Erkennung und Auflösung von Aspektinteraktionen.

Zunächst untersuchen wir die möglichen Abhängigkeiten zwischen je zwei Metaoperatoren op_i und op_j . Hierbei ist in der Praxis wichtig, dass ausgehend von einem Zustand x des Konfigurationsmodells \mathcal{K}_P^x für ein Programm P die Menge der von einem Metaoperator op_y benötigten, erzeugten und entfernten Webepunkte berechnet werden kann. Dies ist mit Hilfe der Vorbedingungen *pre* bzw. Nachbedingungen *post* möglich. Gelten die im Folgenden vorgestellten (Un-)Abhängigkeiten für alle möglichen Programme P , so spricht man von einer starken (Un-)Abhängigkeit. Die möglichen (Un-)Abhängigkeiten und die daraus resultierenden Ausführungsreihenfolgen sind:

- (*Starke*) *Lösch-Abhängigkeit*: Entfernt op_i Webepunkte, welche von op_j benötigt werden, so muss op_i nach op_j ausgeführt werden.
- (*Starke*) *Erzeugungs-Abhängigkeit*: Erzeugt op_i Webepunkte, welche von op_j benötigt werden, so muss der op_i vor op_j ausgeführt werden.
- (*Starke*) *Konkurrenz-Abhängigkeit*: Sowohl op_i als auch op_j beziehen sich auf mindestens einen gemeinsamen Webepunkt. Die Reihenfolge kann nicht ohne weitergehende Informationen aufgelöst werden.
- (*Starke*) *Unabhängigkeit zwischen op_i und op_j* (analog zu [3]): op_i und op_j teilen keine gemeinsamen Webepunkte, sie stehen also insbesondere nicht in einer Lösch-, Erzeugungs- oder Konkurrenz-Abhängigkeit. Die Ausführungsreihenfolge ist in diesem Fall beliebig.

Die Auflösung einer (starken) Konkurrenz-Abhängigkeit kann nicht ohne Berücksichtigung weiterer Bedingungen erfolgen, da von der Anwendungsreihenfolge die Semantik des Ergebnisses abhängt, wie zum Beispiel in [3] gezeigt wird. Wie in [10] gezeigt wird, ist die in AspectJ benutzte Möglichkeit zur Erzwingung einer partiellen Ordnung für die Aspektausführung (hier: Metaoperatortausführung) mit Hilfe der `dominates`-Anweisung nicht ausreichend.

Das Entdecken einer Konkurrenz-Abhängigkeit kann mit Hilfe unseres Verfahrens automatisch erfolgen. Die Auflösung einer Konkurrenz-Abhängigkeit erfolgt in unserem Ansatz durch ein Ersetzen der in Konkurrenz stehenden Metaoperatoren op_i und op_j durch einen neuen Metaoperator $op_{(i,j)}$, wobei der Typ von $op_{(i,j)}$ ein Untertyp sowohl des Typs von op_i als auch op_j ist. Die Konstruktion des neuen Metaoperators $op_{(i,j)}$ muss dabei von außerhalb spezifiziert werden. Hierzu stehen folgende Operationen zur Verfügung:

- *Sequenz*: $op_{(i,j)} = op_i; op_j$ (entspricht der AspectJ `dominates`-Beziehung, vgl. [3])
- *Beliebig*: $op_{(i,j)} = op_i; op_j$ oder $op_{(i,j)} = op_j; op_i$ (d.h. Ergebnisse sind semantisch äquivalent)
- *Bedingte Auswahl*: $op_{(i,j)} = op_i$ oder $op_{(i,j)} = op_j$, abhängig von einer Bedingung b
- *Neuer Operator*: $op_{(i,j)}$ ist nicht durch Kombination aus op_i und op_j konstruierbar. $op_{(i,j)}$ kann allerdings auf Funktionen von op_i und op_j zurückgreifen.

Ein Vorteil dieses Verfahrens ist, dass keine explizite Verbindung zwischen den einzelnen (Basis-) Metaoperatoren op_i und op_j existieren, was eine unabhängige Wiederverwendung erleichtert. Konflikte werden auf höherer Ebene durch explizites Konfigurationswissen aufgelöst.

4 Verwandte Arbeiten

Es existieren bereits einige Techniken zur invasiven Komposition bzw. Konfiguration von Komponenten. Die bekanntesten sind sicherlich die zeichenbasierten Präprozessoren. Sie werden insbesondere benutzt, um verschiedene alternative Implementierungen auszuwählen. Die Struktur des zu transformierenden Programms wird aufgrund der zeichenweisen Bearbeitung nicht betrachtet, daher können keinerlei Korrektheitsaussagen gemacht werden. Das im C++ Umfeld angesiedelte Template Metaprogrammieren (TMP) ist eine invasive Technik, welche die Struktur von C++ Programmen respektiert. Allerdings ist TMP nur durch Missbrauch des C++ Template-Mechanismus möglich. Es ist sehr schwierig, systematisch Metaprogramme zu implementieren. Darüberhinaus sind die so entstandenen Metaprogramme für einen durchschnittlichen Programmierer kaum mehr wartbar.

Eine Technik, welche die geschlossene Spezifikation von verteilt implementierten Merkmalen zum Ziel hat, ist das aspektorientierte Programmieren (AOP). Das bekannteste Werkzeug zur aspektorientierten Programmierung ist ohne Zweifel ASPECTJ. Die zentrale Kritik ist hierbei, dass ASPECTJ nur die Fragmentkomposition unterstützt, nicht aber die Fragmentdekomposition, so wie es unter Umständen im Rahmen von Optimierungsoperationen notwendig ist. Darüberhinaus sind die Möglichkeiten zur Spezifikation der Anwendungsreihenfolge von Aspektanweisungen nicht ausreichend (vgl. [10]). Eine weitere bekannte Technik ist der Hyperspace-Ansatz (MSDOC, [9]), zusammen mit dem Werkzeug HYPERJ. Eine Einschränkung ist hier allerdings, dass die Komposition der Merkmale in HYPERJ auf Klassen- und Methodenebene beschränkt ist.

Die in [1] vorgestellte invasive Softwarekomposition zusammen mit dem COMPOST-System könnte als Implementierungstechnik für den vorgestellten Ansatz ebenso wie das tatsächlich eingesetzte Werkzeug INJECT/J genutzt werden. Sowohl COMPOST als auch INJECT/J basieren auf der Metaprogrammierbibliothek RECODER.

5 Zusammenfassung

In diesem Papier wurde ein Ansatz zur invasiven Konfiguration von Programmfragmenten vorgestellt. Programmfragmente sind hierbei Gegenstand einer hierarchischen Komposition. Die Semantik eines Programmfragments wird in unserem Ansatz durch seinen Typ sowie weitere Bedingungen beschrieben. Es wurden generische Fragmenttypen vorgestellt, welche als Bauplan für konkrete Programmfragmente dienen. Diese bestehen aus einer Konfigurationsschnittstelle, einem Bauplan und einer generischen Implementierung. Der Bauplan beschreibt

hierbei, wie die Parameter der Konfigurationsschnittstelle an die Platzhalter der generischen Implementierung gebunden werden. Die Konfigurationsbeschreibung eines generischen Fragments kann somit wie im Falle der Typgenerizität deklarativ erfolgen, da die Details durch den Bauplan verborgen werden. Um die verteilte Implementierung von Merkmalen geschlossen beschreiben zu können wurden getypte Metaoperatoren eingeführt. Diese bestehen neben einer Schnittstelle aus einem Metaprogramm, welches beschreibt, wo und wie die ebenfalls gegebenen Fragmenteschablonen durch Generierung bzw. Transformation in das resultierende System eingebracht werden müssen. Ähnlich wie im Fall von Programmfragmenten beschreibt der Typ eines Metaoperators zusammen mit Vor- und Nachbedingungen seine Semantik. Durch die Typisierung der Fragmente und der Metaoperatoren können bei einer Fragmentkomposition die Korrektheitsaussagen gemacht werden, die im Rahmen eines Typsystems ausdrückbar sind. Da bei Anwendung mehrerer Metaoperatoren Konflikte auftreten können, wurde ein Verfahren vorgestellt, um diese Konflikte einerseits automatisch erkennen zu können, andererseits wurden Möglichkeiten zur Konfliktauflösung aufgezeigt. Die nächsten Schritte im Verlauf dieser Arbeit bestehen in der Implementierung des vorgestellten Ansatzes in Form einer Erweiterung des Werkzeugs INJECT/J.

Danksagung. Teile dieser Arbeit entstanden im Rahmen des Projekts CollaBaWü des Forschungsverbunds PRIMUM, welches durch das Ministerium für Wissenschaft, Forschung und Kunst des Landes Baden-Württemberg gefördert wird.

Literatur

1. ASSMANN, UWE: *Invasive Software Composition*. Springer Verlag, 2003.
2. CZARNECKI, KRZYSZTOF und ULRICH W. EISENECKER: *Generative Programming – Methods, Tools, and Applications*. Addison-Wesley, Mai 2000.
3. DOUENCE, RÉMI, PASCAL FRADET und MARIO SÜDHOLT: *A Framework for the Detection and Resolution of Aspect Interactions*. In: *GPCE '02: The ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*. Springer LNCS, 2002.
4. FOWLER, MARTIN: *Refactoring*. Addison Wesley, 1999.
5. GENSSLER, T. und V. KUTTRUFF: *Source-to-Source Transformation in the Large*. In: *Proceedings of the Joint Modular Language Conference*. LNCS 2789, 2003.
6. GENSSLER, THOMAS: *Werkzeuggestützte Adaption objektorientierter Programme*. Doktorarbeit, 2004. Universität Karlsruhe.
7. KUTTRUFF, VOLKER: *Ein Modell für invasive Softwareadaption*. Diplomarbeit, Institut für Programmstrukturen und Datenorganisation, Universität Karlsruhe, Mai 2002.
8. KUTTRUFF, VOLKER und THOMAS GENSSLER: *Invasive Configuration of Generic Components*. In: *Proceedings of the Workshop Software Composition SC/ETAPS*, LNCS 3628. Springer-Verlag, 2005.
9. OSSHER, HAROLD und PERI TARR: *Using Multidimensional Separation of Concerns to (Re)Shape Evolving Software*. *Communications of the ACM*, 44(10), 2001.
10. STÖRZER, MAXIMILIAN: *Analysis of AspectJ Programs*. In: *Proceedings of the 3rd German Workshop on Aspect-Oriented Software Development*. GI, 2003.