Fakultät für Elektrotechnik, Informatik und Mathematik
Institut für Informatik
Fachgebiet Softwaretechnik
Warburger Straße 100
33098 Paderborn

Musterbasiertes Re-Engineering von Softwaresystemen

Genehmigte Dissertation
zur Erlangung des Grades
„Doktor der Naturwissenschaften“
(Dr. rer. nat.)

vorgelegt von

Dipl.-Wirt.-Inf. Matthias Meyer

Paderborn, im Dezember 2009

Promotionskommission:
Vorsitzender: Prof. Dr. Wilhelm Schäfer (Universität Paderborn)
Koreferat: Prof. Dr. Gregor Engels (Universität Paderborn)
Koreferat: Prof. Dr. Ekkart Kindler (Technical University of Denmark)
Beisitzer: Prof. Dr. Heike Wehrheim (Universität Paderborn)
Beisitzer: Prof. Dr. Albert Zündorf (Universität Kassel)

Die Dissertation wurde am 16. Oktober 2009 bei der Fakultät für Elektrotechnik, Informatik und Mathematik der Universität Paderborn eingereicht und am 18. Dezember 2009 vor der Promotionskommission verteidigt und durch die Fakultät angenommen.

Zusammenfassung

Software muss durch *Re-Engineering* kontinuierlich an veränderte Rahmenbedingungen und neue Anforderungen angepasst werden. Daher ist die Wartbarkeit von Software, also wie einfach Anpassungen und Erweiterungen vorgenommen werden können, von entscheidender Bedeutung. Die vorliegende Arbeit stellt einen Ansatz vor, der das Re-Engineering von (objektorientierter) Software durch eine Beurteilung und Verbesserung ihrer Wartbarkeit auf Basis von Softwaremustern unterstützt.

Zur Beurteilung der Wartbarkeit wird ein bestehendes Verfahren zur automatisierten Erkennung und Bewertung von Entwurfsmusterinstanzen für die Erkennung und Bewertung von Schwachstellen, Instanzen von Bad Smells und Anti Patterns, leicht erweitert. Aufbauend darauf wird eine grafische Sprache zur Spezifikation von Programmtransformationen entwickelt, mit denen erkannte Schwachstellen in bessere Lösungen zum Beispiel auf Basis von Entwurfsmustern transformiert werden können.

Bei einer solchen Transformation soll in der Regel die Struktur einer Software verbessert werden, ohne ihr von außen beobachtbares Verhalten zu verändern (*Refactoring*). Dass eine Transformation das Verhalten eines Programms nicht ändert, kann häufig nur mit erheblichem manuellem Aufwand oder gar nicht bewiesen werden. Um bestimmte Verhaltensänderungen auszuschließen, entwickelt diese Arbeit ein automatisches Verfahren, das eine Verifikation von Transformationen auf Einhaltung definierbarer Kriterien erlaubt. Dies wurde insofern bereits bei der Entwicklung der Sprache berücksichtigt, als dass ihre Ausdrucksmächtigkeit sowohl auf die Durchführung von Transformationen als auch ihre Verifikation ausgelegt wurde.

Die Kriterien formulieren strukturelle Eigenschaften des abstrakten Syntaxgraphen eines Programms, die durch eine Transformation erhalten oder vermieden werden müssen. Das Verfahren versucht nachzuweisen, dass die Kriterien (induktive) strukturelle Invarianten sind, die in allen Versionen beliebiger Syntaxgraphen gelten, die durch Ausführung vollständiger Transformationen entstehen können. Kann eine Transformation ein Kriterium verletzen, werden systematisch repräsentative Beispiele für problematische (Teil-)Ausführungen ermittelt und dem Re-Engineer so Hinweise für eine Korrektur gegeben.

Danksagung

Die vorliegende Dissertation ist nicht ohne die Mithilfe zahlreicher Personen entstanden, bei denen ich mich an dieser Stelle bedanken möchte.

Mein Dank gilt zunächst meinem Doktorvater Wilhelm Schäfer. Wilhelm hat mir nicht nur die Möglichkeit gegeben, in seiner Gruppe zu forschen und mich bei meiner Promotion betreut, sondern mir auch darüber hinaus spannende Tätigkeiten ermöglicht. Ich danke den weiteren Mitgliedern meiner Prüfungskommission Gregor Engels, Ekkart Kindler, Heike Wehrheim und Albert Zündorf dafür, dass sie sich mit meiner Arbeit befasst haben. Gregor Engels und Ekkart Kindler danke ich für die Anfertigung eines Gutachtens.

Bei meinen Kollegen Robert Wagner und Lothar Wendehals, mit denen ich eng zusammen arbeiten durfte und die mir durch viele fruchtbare Diskussionen (auch gerne mal fernab der Promotion) geholfen haben, möchte ich mich besonders bedanken. Für mindestens ebenso viele solcher Diskussionen und noch vieles andere mehr danke ich ganz herzlich meinem langjährigen Bürokollegen Matthias Tichy.

Des Weiteren danke ich allen weiteren (ehemaligen) Kollegen, die während meiner Promotionszeit durch die sehr gute Arbeitsatmosphäre zum Gelingen dieser Arbeit beigetragen haben. Dies sind Björn Axenath, Christian Bimmermann, Christopher Brink, Sven Burmester, Markus von Detten, Matthias Gehrke, Holger Giese, Joel Greenyer, Stefan Henkler, Martin Hirsch, Jörg Holtmann, Ekkart Kindler, Ahmet Mehic, Jan Meyer, Jörg Niere, Claudia Priesterjahn, Jan Rieke, Vladimir Rubin, Daniela Schilling, Florian Stallmann, Oliver Sudmann, Dietrich Travkin und Jörg Wadsack sowie alle Kollegen im Software Quality Lab (s-lab).

Dass ich mit der Verwaltung der Universität so gut zurecht gekommen bin, habe ich Jutta Haupt zu verdanken und Jürgen Maniera stand mir bei allen technischen Problemen zur Seite.

Danke auch an alle Studenten, die als studentische Hilfskräfte oder im Rahmen ihrer Projektgruppe, Studien- oder Diplomarbeit zu dieser Arbeit beigetragen haben.

Schließlich bedanke ich mich bei all meinen Freunden und meiner Familie.

Marion Peter hat mich ein langes Stück des Weges begleitet und unterstützt; Danke dafür! Ganz besonders aber danke ich meinen Eltern Ingrid und Willi Meyer. Sie haben mir meine Ausbildung ermöglicht und standen und stehen mir bei allen wichtigen Entscheidungen beratend zur Seite.

Inhalt

Kapitel 1

Einleitung

Software ist grundsätzlich leicht veränderbar. Dies ermöglicht eine Anpassung an sich ändernde Rahmenbedingungen und damit einhergehend von Anforderungen, die an eine Software gestellt werden. Software wird daher insbesondere in solchen Bereichen eingesetzt, in denen sich die Rahmenbedingungen mitunter rasant verändern, so dass bei fast allen im Einsatz befindlichen Systemen eine kontinuierliche Anpassung und Erweiterung an neue Anforderungen stattfindet, die als *Wartung* bezeichnet wird.

Zahlreiche Studien belegen, dass dies mit einem enormen Aufwand verbunden ist: 60-90% der Kosten für Software entfallen auf Wartungsaktivitäten (vgl. [Huf90, Moa90, Eas93, Erl00]). Seacord, Plakosh und Lewis sprechen sogar von einer *Legacy Crisis* [SPL03], da immer mehr über Jahre gewachsene *Legacy*-Systeme einen Großteil der Ressourcen verbrauchen und kaum mehr etwas für Neuentwicklungen übrig bleibt.

Die hohen Kosten resultieren zu einem großen Teil daraus, dass eine bestehende Software zunächst verstanden werden muss, bevor sie angepasst oder erweitert werden kann. Die Dokumentation sowie etwaige Entwurfsdokumente sind häufig aufgrund von Zeit- und Kostendruck bei der Entwicklung sowie der späteren Wartung vernachlässigt worden. Durchgeführte Anpassungen wurden in den Dokumenten nicht nachgehalten, so dass diese inkonsistent zur tatsächlichen Implementierung der Software sind. Bei über Jahre hinweg gewachsenen Systemen sind auch die verantwortlichen Entwickler häufig nicht mehr verfügbar. Die einzig verlässliche Basis, anhand derer ein Verständnis des Systems erarbeitet werden muss, stellen dann die oft Millionen von Zeilen umfassenden Quelltexte dar.

Diese liefern Informationen auf einem sehr niedrigen Abstraktionsniveau. Höherwertige Dokumentation und Entwurfsdokumente müssen durch *Reverse Engineering* [CC90] aufwändig aus den Quelltexten rekonstruiert werden. So-

mit ist es nicht verwunderlich, dass nach Frazer [Fra92] 50% der Kosten des *Re-Engineering* – also der Anpassung von Software an neue Anforderungen – auf das Verstehen vorhandener Quelltexte entfallen. Sind die Quelltexte erst einmal verstanden, müssen sie im nächsten Schritt angepasst und erweitert werden, um letztendlich die neuen Anforderungen umzusetzen. Die Planung und Durchführung von Änderungen verursachen laut Frazer [Fra92] weitere 20% der Kosten. Die übrigen 30% enfallen auf das Testen und Dokumentieren von Änderungen.

Die Qualität der Quelltexte und des darin implizit enthaltenen Entwurfs ist insbesondere im Hinblick auf Verständlichkeit und Änderungsfreundlichkeit von entscheidender Bedeutung. Schlecht strukturierte Quelltexte können zum einen das Verstehen ihrer Funktion massiv erschweren und tragen so wesentlich zu den hohen Kosten des Programmverstehens bei. Auch bei noch so guter Dokumentation des Systems, die eine schnelle Bestimmung der für die geplanten Wartungsaktivitäten relevanten Quelltexte erlaubt, müssen diese letztendlich ausreichend verstanden werden. Zum anderen kann eine schlechte Struktur die Erweiterung des Systems behindern oder gar unmöglich machen.

Die Beurteilung der Verständlichkeit und Änderungsfreundlichkeit, letztendlich der Wartbarkeit, von Quelltexten und Entwurfsmodellen ist damit von ebenso großer Bedeutung. Zum einen kann sie bereits während des Entwurfs und der Entwicklung einer Software stattfinden, um frühzeitig das Entstehen schlecht wartbarer Software zu verhindern. Beim Re-Engineering von Legacy-Systemen kann sie helfen, mögliche Probleme bereits zu Beginn, in der Planungsphase, zu erkennen und geeignet darauf zu reagieren. Dass 23% der in [SPL03] betrachteten Modernisierungsprojekte vorzeitig abgebrochen und nur 28% innerhalb der geplanten Zeit mit den kalkulierten Kosten und der erwarteten Funktionalität abgeschlossen werden zeigt, dass hier dringend Unterstützung benötigt wird. Ist zur Umsetzung neuer Funktionalität die Integration bisher unbekannter Komponenten von Fremdanbietern vorgesehen, kann deren Wartbarkeit ebenfalls entscheidend für den Erfolg des aktuellen sowie zukünftiger Vorhaben sein.

Aufgrund der Größe und Komplexität heutiger Softwaresysteme werden geeignete Verfahren und Werkzeuge benötigt, die eine Beurteilung der Wartbarkeit weitgehend automatisiert durchführen beziehungsweise den Re-Engineer wesentlich dabei unterstützen. Dazu werden häufig Softwareproduktmetriken [FP96, LK94] eingesetzt. Metriken erfassen ein System quantitativ, indem sie Kennzahlen berechnen wie zum Beispiel die Anzahl Quelltextzeilen (Lines Of Code) von einzelnen Modulen oder Klassen, die Anzahl Methoden oder Attribute einer Klasse, die Komplexität des Kontrollflusses von Methoden [McC76]

oder diverse Durchschnittswerte. In Kombination mit Visualisierungstechniken wie zum Beispiel polymetrischen Sichten [Lan03] erlauben sie einen ersten Eindruck von einer Software zu gewinnen und Ausreißer zu identifizieren, die potentielle Schwachstellen darstellen und einer feingranularen Analyse bedürfen. Eine ergänzende Herangehensweise auch zur feingranularen Analyse ist die qualitative Beurteilung der Wartbarkeit auf Basis von Softwaremustern.

1.1 Softwaremuster

Eine heute allgemein akzeptierte Möglichkeit, eine Qualitätssteigerung von Software im Hinblick auf Änderungsfreundlichkeit zu erreichen, ist die Verwendung von *Entwurfsmustern* (engl. *Design Patterns*). Entwurfsmuster beschreiben häufig anzutreffende Probleme des Softwareentwurfs und bieten bewährte Lösungen, die insbesondere gut erweiterbar sind. Solche Entwurfsmuster sind in der Regel nicht auf dem Reißbrett entstanden, sondern schon immer von guten Entwicklern verwendet und mit der Zeit weiter verfeinert worden.

Mitte der 1990er Jahre wurde damit begonnen, bewährte Entwurfslösungen für charakteristische Probleme aus bestehenden Programmen zu extrahieren und in wiederverwendbarer Form als Entwurfsmuster zu dokumentieren. Mit [GHJV95] ist 1995 ein Standardwerk für den objektorientierten Entwurf entstanden. Darüber hinaus gibt es eine Vielzahl weiterer Veröffentlichungen von Mustern für unterschiedliche Anwendungsgebiete wie zum Beispiel [BMR+96, CKV96, Cop92, Fow02, Lan99, PLo, Ris00, The].

Etwa zur gleichen Zeit wurden *Anti Patterns* [BMMM98] und *Bad Smells* [Fow99] dokumentiert. Bei Anti Patterns handelt es sich um das Gegenstück zu Entwurfsmustern. Ein Anti Pattern beschreibt eine problematische Lösung eines charakteristischen Problems, die häufig schwer verständlich und insbesondere schlecht erweiterbar ist. Objektorientierte Anti Patterns verletzen häufig objektorientierte Prinzipien. Eine solche Lösung kann auf den ersten Blick zwar zweckmäßig erscheinen, eine spätere Wartung oder Erweiterung aber unnötig erschweren.

Während Anti Patterns sich eher auf die Entwurfsebene und größere Zusammenhänge beziehen, bezeichnen Bad Smells eher auffällige Stellen im Quelltext, die auf Probleme hindeuten können. Die Grenze zwischen Anti Patterns und Bad Smells ist dabei fließend. Im Unterschied zu Entwurfsmustern beinhalten Anti Patterns und Bad Smells über strukturelle Eigenschaften hinaus häufiger quantifizierbare Merkmale wie eine Methode ist „zu lang“ oder eine Klasse hat „zu viele“ Attribute und Methoden. Für solche Merkmale können nur schwer

absolute Grenzwerte definiert werden. Allerdings können sie genutzt werden um zu bewerten, wie schwerwiegend ein konkretes Vorkommen ist.

Durch die Dokumentation und Benennung der Muster ist ein gemeinsames Vokabular für Entwickler entstanden. So hilft das Wissen über konkrete Implementierungen von Entwurfsmustern in einer Software, so genannte *Entwurfsmusterinstanzen*, bei ihrem Verständnis. Zudem kennzeichnen Entwurfsmusterinstanzen wohl strukturierte und einfach erweiterbare Teile einer Software. Im Gegensatz dazu identifizieren Instanzen von Anti Patterns und Bad Smells, im Weiteren *Schwachstellen* genannt, schwer verständliche und schlecht erweiterbare Stellen, die eine Anpassung der Software an neue Anforderungen behindern können. Die Kenntnis über vorhandene Entwurfsmusterinstanzen und Schwachstellen gestattet wohl strukturierte, wartbare Software von schlecht strukturierter, schwer verständlicher Software zu unterscheiden.

Da die Dokumentation einer Software häufig unzureichend ist, sind auch Entwurfsmusterinstanzen häufig nicht dokumentiert. Schwachstellen sind es noch viel weniger. Musterinstanzen müssen durch Reverse Engineering der Quelltexte erkannt werden. In den vergangenen Jahren wurde eine Reihe von Ansätzen entwickelt, die eine formale Spezifikation und (semi-)automatische Erkennung von Entwursmusterinstanzen oder Schwachstellen ermöglichen.

Dabei gibt es mit [BC98, Ciu99, EM02, KBSD04] Ansätze insbesondere zur Erkennung von Schwachstellen, die sich rein auf strukturelle Eigenschaften beziehen und die quantitativen Aspekte nicht adäquat berücksichtigen können. Andere Ansätze wie zum Beispiel [Mar04, Mun05] basieren auf quantitativen Analysen mit Hilfe von Softwareproduktmetriken und lassen strukturelle Aspekte außer Acht oder berücksichtigen sie versteckt bei der Berechnung von Metriken. Es gibt wenige Ansätze [MGMD08, TR07], die strukturelle und quantitative Analyse explizit kombinieren. Der Ansatz [GA08] kann ebenfalls dazu gerechnet werden und erlaubt zusätzlich eine Bewertung von Funden auf Basis (nicht) erfüllter Eigenschaften der Musterspezifikationen, aber keine kontinuierliche Bewertung auf Basis quantitativer Merkmale.

1.2 Verbesserung der Wartbarkeit

Eine Schwachstelle beschreibt eine schlechte Lösung, für die in der Regel eine bessere Lösung bekannt ist. So lassen sich für einige Schwachstellen sehr konkrete Hinweise zur Verbesserung geben, während für andere lediglich generelle, wenig konkrete Vorschläge gemacht werden können. In einigen Fällen ist sogar eine automatisierte Restrukturierung einer Schwachstelle in eine bessere

Lösung zum Beispiel unter Verwendung von Entwurfsmustern möglich. Auf diese Weise können die Wartbarkeit der Software verbessert und Erweiterungen geeignet vorbereitet werden.

Unter Restrukturierung wird die Transformation einer Software auf derselben Abstraktionsebene, zum Beispiel der Quelltextebene, verstanden. Dabei wird lediglich die Struktur der Software verändert, ihre Funktionsweise und ihr von außen beobachtbares Verhalten bleiben davon unberührt [CC90].

Opdyke systematisiert von Entwicklern häufig durchgeführte verhaltenserhaltende Restrukturierungen objektorientierter Software und bezeichnet sie als *Refactorings* [Opd92]. Fowler [Fow99] und Kerievsky [Ker04] beschreiben zahlreiche weitere Refactorings, die zur Verbesserung von Schwachstellen eingesetzt werden können und sie zum Teil in Entwurfsmusterinstanzen transformieren.

Refactorings werden in der Literatur weitgehend informal beschrieben. Die einzelnen Schritte werden textuell skizziert und anhand kleinerer Beispiele illustriert. Die Durchführung liegt weitgehend in Händen des Re-Engineers. Die manuelle Durchführung von Refactorings ist allerdings mühsam und fehleranfällig insbesondere im Fall von umfangreichen und komplexen Restrukturierungen. Heutige Entwicklungsumgebungen [Ecla, Int, Emb, SA, Sun, Mic] bieten daher eine automatisierte Ausführung für eine Reihe von einfachen Refactorings an. Diese gehen aber häufig nicht weit genug, so dass umfangreiche Restrukturierungen immer noch gänzlich manuell oder durch manuelle Kombination kleinerer Refactorings durchgeführt werden müssen.

Darüber hinaus gibt es keine Integration mit Werkzeugen zur Erkennung von Schwachstellen. Hier ist der Re-Engineer gefragt, aus den Ergebnissen einer solchen Erkennung auf die durchführbaren Refactorings zu schließen und diese geeignet anzustoßen. Es fehlt ein geschlossener Ansatz, der eine Verbindung zwischen Schwachstellenerkennung und Restrukturierung herstellt, so dass Refactorings direkt auf erkannte Schwachstellen anwendbar sind. In [TR07] werden Ideen präsentiert, die in diese Richtung gehen. Allerdings wurde bisher nur die Erkennung von Schwachstellen vertiefend behandelt und umgesetzt.

Ein weiterer wesentlicher Aspekt von Refactorings ist die Verhaltenserhaltung. Fowler und Kerievsky empfehlen eine umfangreiche Testsuite und das regelmäßige Testen während der Durchführung von Refactorings, um zu überprüfen, ob das Programm noch wie gewünscht funktioniert. Opdyke formalisiert in [Opd92] Vorbedingungen für seine Refactoring-Operationen, bei deren Erfüllung das von ihm definierte von außen beobachtbare Verhalten nicht verändert wird. In [Rob99, KK04] werden Nachbedingungen hinzugefügt und Vor- und Nachbedingungen für zusammengesetzte Refactorings bestimmt. Dass die Vor- und Nachbedingungen tatsächlich eingehalten werden, kann nicht

bewiesen werden, da eine Formalisierung der Durchführung der Refactorings fehlt. Auch wenn alle Vorbedingungen erfüllt sind und dadurch tatsächlich Verhaltenserhaltung garantiert wird, bleibt es weiterhin möglich, dass eine fehlerhafte Durchführung des Refactorings die transformierte Software zerstört oder ihr Verhalten verändert. Erst eine vollständige Formalisierung von Refactorings ermöglicht es, Aussagen zur Verhaltenserhaltung zu beweisen.

In der Literatur sind viele Refactorings beschrieben, aber selbstverständlich nicht alle, die je benötigt werden. Auch ist nicht zu erwarten, dass in Zukunft alle beschriebenen Refactorings formalisiert sein werden. Ein Re-Engineer muss daher die Möglichkeit haben, selbst nach Bedarf Refactorings oder allgemeiner Programmtransformationen formal zu spezifizieren und auszuführen. In [Cas94, DD04, FTK+05, KETF07, MHGV08, Moo96, MS99, SS04, ST00] werden spezialisierte Ansätze vorgestellt, die bestimmte Restrukturierungen vornehmen und nicht erweiterbar sind. Die Arbeiten [AAG01, GAA01, FMv97, O'C01, SGMZ98, Tah03, ZKDZ07] befassen sich mit der (teilweisen) Instanzierung von Entwurfsmustern in bestehenden Systemen. Die dabei eingesetzten Restrukturierungsoperationen sind ebenfalls fest definiert.

Für beliebige Transformationen kann die Überprüfung der Verhaltenserhaltung schwierig oder gar unmöglich sein. Allerdings können eine Reihe von Kriterien formuliert werden, die durch eine Transformation auf keinen Fall verletzt werden dürfen oder sollen. Bei Einhaltung solcher Kriterien können bestimmte Verhaltensänderungen ausgeschlossen werden. Bei geeigneter Formalisierung von Kriterien sowie Transformationen, ist eine automatische Verifikation der Transformationsspezifikationen auf Einhaltung der Kriterien möglich. Wenn auf diese Weise Verhaltenserhaltung auch nicht in jedem Fall garantiert werden kann, so können zumindest einige verhaltensverändernde Transformationsspezifikationen erkannt werden. In [MDJ02, DHJ+07, RLK+08, VEM06] werden Formalismen für die Spezifikation beliebiger Refactorings vorgeschlagen. Eine Verifikation der Spezifikationen im Hinblick auf Aussagen zur Verhaltenserhaltung wird aber nicht ausreichend unterstützt oder ist nicht für die Transformation komplexer Programme einsetzbar.

1.3 Ergebnisse der Arbeit

In der vorliegenden Arbeit wird daher ein geschlossener, musterbasierter Ansatz zur Unterstützung und Vorbereitung von Re-Engineering-Aktivitäten in bestehenden Softwaresystemen entwickelt, der die zuvor präsentierten Probleme adressiert [Mey06]. Abbildung 1.1 gibt einen Überblick über den Ansatz.

Er unterteilt sich in die Bereiche der Erkennung von Schwachstellen im linken Teil der Abbildung und ihrer Verbesserung im rechten Teil. Die beiden Bereiche untergliedern sich wiederum jeweils in eine Spezifikationsebene im oberen Bereich der Abbildung und eine Ausführungsebene im unteren Bereich.

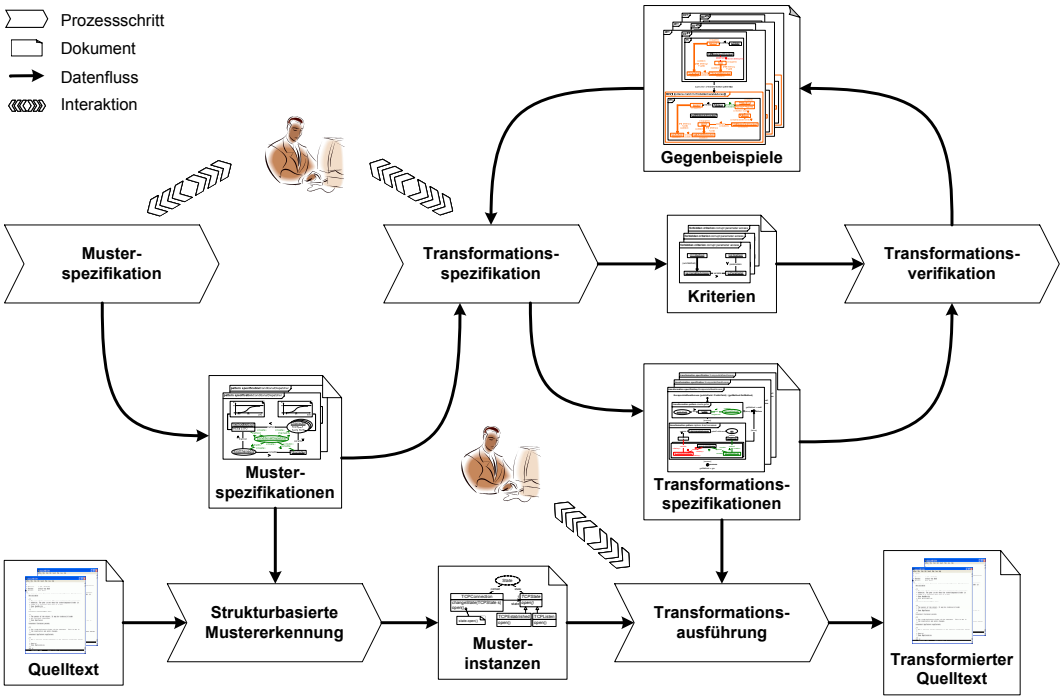

Abbildung 1.1: Musterbasiertes Re-Engineering

Mit der strukturbasierten Mustererkennung wird ein Verfahren zur Analyse des Quelltextes eines zu verändernden Systems im Hinblick auf seine Verständlichkeit und Erweiterbarkeit bereitgestellt. Durch die Analyse werden explizit schwer verständliche und schlecht erweiterbare Schwachstellen identifiziert und bewertet, die ein Re-Engineering des Systems erschweren oder sogar unmöglich beziehungsweise nicht-wirtschaftlich machen können.

Dazu wird der auf Graphen und Graphtransformationen basierende Ansatz zur Erkennung von Entwurfsmusterinstanzen im Quelltext einer Software aus [Nie04] für die Erkennung struktureller Schwachstellen um die Berücksichtigung quantitativer Merkmale (auf Basis von Softwareproduktmetriken) erweitert. Der Ansatz stellt eine werkzeuggestützte formale Spezifikation und automatische Erkennung sowie Bewertung von Entwurfsmusterinstanzen und insbesondere Schwachstellen zur Verfügung. Die Erweiterung gestattet insbe-

sondere eine kontinuierliche Bewertung von Musterinstanzen auf Basis quantitativer Merkmale, um schwerwiegende, signifikante Schwachstellen von weniger schwerwiegenden zu unterscheiden.

Ergebnis der Analyse sind bewertete Musterinstanzen, die zum einen wertvolle Informationen für die Abschätzung des Risikos von Re-Engineering-Aktivitäten liefern und zum anderen Ansatzpunkte zur Verbesserung der Wartbarkeit durch eine Restrukturierung der Software aufzeigen können.

Um den Re-Engineer bei der Restrukturierung von Software zu unterstützen, entwickelt die vorliegende Arbeit eine formale Spezifikationssprache für Programmtransformationen, ebenfalls auf Basis von Graphtransformationen. Die Transformationsspezifikation und -ausführung sind so mit der strukturbasierten Mustererkennung integriert, dass Transformationen auf spezifizierte Muster Bezug nehmen und auf erkannte Instanzen, zum Beispiel zur Verbesserung von Schwachstellen, angewendet werden können. Die Transformationsspezifikationen können bis auf notwendige Benutzereingaben automatisch ausgeführt werden. Die Entscheidung, ob und wo eine Transformation durchgeführt wird, wird dabei nicht automatisch getroffen, sondern bleibt dem Re-Engineer überlassen.

Ziel bei der Restrukturierung von Software ist im Allgemeinen die Verbesserung der Struktur ohne das Verhalten zu verändern. Die Überprüfung der semantischen Äquivalenz von beliebigen Programmen ist ein nicht entscheidbares Problem. Formal spezifizierte Restrukturierungen bieten aber eine Grundlage, um Aussagen betreffend die Erhaltung bestimmter Aspekte des Verhaltens eines restrukturierten Programms beweisen zu können.

Die vorliegende Arbeit ermöglicht dem Re-Engineer die Spezifikation von Kriterien in Form struktureller Eigenschaften des abstrakten Syntaxgraphen eines Programms, die zu definierten Zeitpunkten gelten müssen oder nicht gelten dürfen. Dies können Kriterien sein, bei deren Einhaltung durch eine Transformation bestimmte Änderungen des Verhaltens der transformierten Software ausgeschlossen werden können.

Aufbauend auf einen Ansatz zum Nachweis induktiver Strukturinvarianten in Graphtransformationssystemen [Sch06] wird ein Verfahren entwickelt, mit dem Transformationsspezifikationen automatisch auf Einhaltung solcher Kriterien verifiziert werden können. Dies wurde insofern bereits bei der Entwicklung der Spezifikationssprache für Transformationen berücksichtigt, als dass ihre Ausdrucksmächtigkeit sowohl auf die Durchführung von Programmtransformationen als auch eine Vereinfachung ihrer Verifikation ausgelegt wurde.

Das Verifikationsverfahren betrachtet die Transformationsspezifikationen im Allgemeinen, ohne von einem konkreten zu transformierenden Programm aus-

zugehen. Es versucht den Nachweis zu führen, dass die Einhaltung eines jeden Kriteriums eine (induktive) strukturelle Invariante ist, die in allen Versionen beliebiger Programme gilt, die durch Ausführung vollständiger Transformationsdiagramme erzeugt werden können.

Damit wird keine (im Allgemeinen unentscheidbare) Überprüfung der semantischen Äquivalenz zweier Programme verfolgt, sondern vielmehr wird der Nachweis von Kriterien für Programmtransformationen unterstützt, deren Verletzung zu einer Verhaltensänderung im transformierten Programm führt oder führen kann. Der Umkehrschluss, dass bei Einhaltung aller Kriterien eine Verhaltensänderung ausgeschlossen ist, muss nicht gelten.

Das Verifikationsverfahren versucht systematisch Gegenbeispiele zu konstruieren, bei denen die Ausführung einer Transformation zu einer Kriterienverletzung führt. Die Gegenbeispiele beschreiben repräsentative (Teil-)Ausführungen einer Transformation und charakterisieren ausschnittsweise wie ein Programm beschaffen sein muss, damit die problematische Ausführung auftreten kann. Diese Gegenbeispiele bieten dem Re-Engineer wertvolle Informationen für die Korrektur der Spezifikationen.

Zudem wird das Verifikationsverfahren so entwickelt, dass keine Kriterienverletzungen unerkannt bleiben - kann also kein Gegenbeispiel gefunden werden, ist eine Kriterienverletzung ausgeschlossen. Wenn dies in dieser Arbeit auch nicht mathematisch formal bewiesen, sondern bei der Beschreibung des Verfahrens informal argumentiert wird, so bietet allein die systematische Berechnung von Gegenbeispielen eine wesentliche Unterstützung für den Re-Engineer und ist damit ein wesentliches Ergebnis dieser Arbeit.

Der in dieser Arbeit entwickelte Ansatz wurde prototypisch als Erweiterung des FUJABA4ECLIPSE-Werkzeugs [Fuj] umgesetzt und in die Entwicklungsumgebung ECLIPSE [Ecla] integriert. Darüber hinaus wurde der Ansatz anhand von fünf Softwaresystemen evaluiert.

1.4 Aufbau der Arbeit

Im zweiten Kapitel der vorliegenden Arbeit werden die für das weitere Verständnis notwendigen Grundlagen vorgestellt. Dazu zählen insbesondere die strukturbasierte Entwurfsmustererkennung aus [Nie04] und ihre Erweiterung um die Berücksichtigung quantitativer Merkmale sowie der Verifikationsansatz aus [Sch06], der später erweitert wird.

In Kapitel 3 wird die formale Spezifikation von Programmtransformationen vorgestellt, die im Rahmen dieser Arbeit zur Formalisierung von ausführbaren

Refactorings eingesetzt wird. Ausgehend von Anforderungen an eine solche Sprache wird diese zunächst informal eingeführt. Danach werden ihre Syntax und Semantik formal definiert. Abschließend wird ein Verfahren vorgestellt, mit dem die Anfertigung von Spezifikationen auf Basis von Quelltextbeispielen vereinfacht werden kann.

Kapitel 4 beschreibt das Verfahren zur Verifikation von Transformationsspezifikationen auf Einhaltung definierbarer Kriterien. Zunächst wird die Definition von Verifikationskriterien beschrieben. Danach wird ein Überblick über das Verfahren gegeben und seine einzelnen Schritte werden genauer vorgestellt. Abschließend wird resümiert, welche Aussagen das Verfahren treffen kann.

Das fünfte Kapitel beschreibt die prototypisch umgesetzte Werkzeugunterstützung des Ansatzes in Form von Plug-Ins für die weit verbreitete Entwicklungsumgebung Eclipse. Es werden sowohl die Benutzungsschnittstelle des Prototyps als auch seine Architektur beschrieben.

Das Thema des sechsten Kapitels ist die Evaluierung des vorgestellten Ansatzes. Dazu werden zunächst einige Musterspezifikationen vorgestellt und danach die Ergebnisse der Analyse von fünf Softwaresystemen präsentiert. Für ausgewählte Schwachstellen werden Transformationsspezifikationen erstellt. Diese werden im Anschluss mit Hilfe des Verifikationsverfahrens auf Einhaltung einiger Kriterien überprüft.

In Kapitel 7 werden verwandte Arbeiten anderer Wissenschaftler vorgestellt und zur vorliegenden Arbeit in Bezug gesetzt.

Das achte Kapitel gibt eine Zusammenfassung der erzielten Ergebnisse und einen Ausblick auf mögliche zukünftige Arbeiten zur weiteren Verbesserung des Ansatzes.

Kapitel 2

Grundlagen

Dieses Kapitel beschreibt die zum Verständnis der vorliegenden Arbeit notwendigen Grundlagen. Zu Beginn werden einige Anti Patterns vorgestellt, gefolgt von Refactorings die zur Verbesserung von Schwachstellen eingesetzt werden können. Im Anschluss wird die strukturbasierte Entwurfsmustererkennung vorgestellt und wie sie in dieser Arbeit zur Erkennung von Schwachstellen eingesetzt und erweitert wird. Den Abschluss bildet der Ansatz des *Story Driven Modeling*, der im Fachgebiet Softwaretechnik der Universität Paderborn entwickelt wurde, und die Grundlagen sowohl für die Spezifikation von Programmtransformationen als auch deren Verifikation darstellt.

2.1 Anti Patterns

Anti Patterns beschreiben Probleme und im Gegensatz zu Entwurfsmustern ihre typischen aber schlechten Lösungen. Eine solche Lösung kann auf den ersten Blick zwar zweckmäßig und ausreichend erscheinen, eine spätere Wartung oder Erweiterung aber unnötig erschweren, insbesondere wenn die Software durch die Umsetzung zusätzlicher Anforderungen komplexer wird. Objektorientierte Anti Pattern zeichnen sich häufig durch die Verletzung objektorientierter Prinzipien aus.

Der Begriff Anti Pattern wird in der Literatur im Gegensatz zum Begriff Entwurfsmuster nicht so einheitlich oder zum Teil überhaupt nicht verwendet. In [BMMM98] werden unter der Bezeichnung Anti Pattern problematische Lösungen in Bezug auf sowohl das Management von Softwareprojekten als auch die Architektur beziehungsweise den Entwurf von Softwareprodukten beschrieben. Fowler et al. beschreiben in [Fow99] problematische Implementierungen beziehungsweise Indikatoren dafür unter der Bezeichnung *Bad Smells*. Während sich ein Großteil der Bad Smells auf die Implementierungsebene be-

ziehen, sind auch einige darunter, die Entwurfsprobleme beschreiben. Während Kerievsky in [Ker04] den Begriff der Bad Smells aufgreift und sich im Weiteren eher mit Entwurfsproblemen befasst, beschreiben Demeyer et al. in [DDN03] Entwurfsprobleme, ohne diese explizit als Anti Pattern oder Bad Smell zu bezeichnen. Riel stellt in [Rie96] Heuristiken für einen guten Entwurf auf, aus deren Verletzung sich wiederum problematische Lösungen ableiten lassen.

Im Rahmen dieser Arbeit wird unter einem Anti Pattern eine problematische Lösung in Bezug auf den Entwurf einer Software verstanden während problematische Implementierungen beziehungsweise Hinweise darauf als Bad Smells bezeichnet werden. Wenn nicht explizit unterschieden werden soll, wird auch allgemein von *Schwachstelle* gesprochen.

Die Dokumentation von Schwachstellen ist nicht so einheitlich strukturiert, wie im Falle von Entwurfsmustern. Dennoch werden sie häufig benannt, es werden die Lösung und die damit verbundenen Nachteile sowie typischerweise eine bessere Lösung beschrieben. Die Beschreibungen sind ebenso wenig formal wie bei Entwurfsmustern. Vielmehr sind sie häufig noch allgemeiner und vager gehalten. Damit geht einher, dass sich Schwachstellen häufig auf quantifizierbare Merkmale beziehen, dabei aber nur davon sprechen, dass es von etwas *zu viel* oder *zu wenig* gibt (zum Beispiel Methoden in einer Klasse).

Da Schwachstellen im Allgemeinen weniger bekannt sind als Entwurfsmuster und zudem eine wesentliche Motivation für die vorliegende Arbeit darstellen, werden im Folgenden einige Beispiele und ihre besseren Lösungen vorgestellt.

In [Fow99] wird der Bad Smell *Large Class* beschrieben. Damit wird eine Klasse bezeichnet, die zu viele Attribute und Methoden hat und/oder zu viel Quelltext enthält. Eine scharfe Grenze, ab wann dies der Fall ist, kann nicht allgemein angegeben werden. Eine solche Klasse sollte aufgeteilt werden auf mehrere Klassen. Das Gegenstück dazu ist eine *Lazy Class*, die zu wenig tut, um ihre Existenz zu rechtfertigen, und deren Funktionalität besser in andere Klassen integriert werden sollte.

Ein weiteres Beispiel für einen einfachen Bad Smell der sich sowohl in [Fow99] als auch in [Rie96] findet, ist ein öffentlich sichtbares, nicht konstantes Attribut einer Klasse. Durch solche Attribute wird das Prinzip des *Information Hiding* verletzt und Klienten der Klasse werden abhängig von der internen Repräsentation ihrer Daten. Auch erbende Klassen sollten nicht von der Datenrepräsentation ihrer Basisklassen abhängig sein. So konkret wie das Problem beschrieben ist, ist auch seine Lösung. Zur Verbesserung sind Zugriffsmethoden mit entsprechender Sichtbarkeit für das betreffende Attribut zu erzeugen und in einem nächsten Schritt müssen alle direkten Zugriffe durch einen Aufruf der entsprechenden Zugriffsmethode ersetzt werden. Abschließend kann die

Sichtbarkeit des Attributs auf privat gesetzt werden.

Das wohl bekannteste Anti Pattern wird in [BMMM98] als *The Blob* und in [DDN03, Rie96] als *God Class* bezeichnet. Damit wird eine Klasse bezeichnet, die zuviel Verantwortung in ihrem System oder Subsystem übernimmt. Riel unterscheidet in [Rie96] noch zwei Extremformen, die *Behavioral God Class* und die *Data God Class*. Bei der *Behavioral God Class* handelt es sich um eine Klasse, die (beinahe) das gesamte Verhalten ihres (Sub-)Systems kapselt. Um die Klasse herum existieren viele kleinere Datenklassen (Anti Pattern *Data Class*, siehe zum Beispiel [Fow99]), die fast nur aus Attributen bestehen und selbst kein beziehungsweise kaum Verhalten haben. Bei der *Data God Class* handelt es sich dagegen um eine Klasse, die (beinahe) die gesamten Daten ihres (Sub-)Systems verwaltet und von vielen kleineren Klassen als zentraler Datenspeicher genutzt wird.

In beiden Fällen wurden Daten und die darauf arbeitenden Funktionen getrennt, was eher prozeduralem als objektorientiertem Entwurf entspricht. Für die Auflösung des Problems gilt die Maxime Daten und Operationen darauf zusammenzuführen. Dazu sind Attribute und/oder Methoden geeignet zu verschieben. Möglicherweise kann eine God Class auch in mehrere Klassen zerlegt werden.

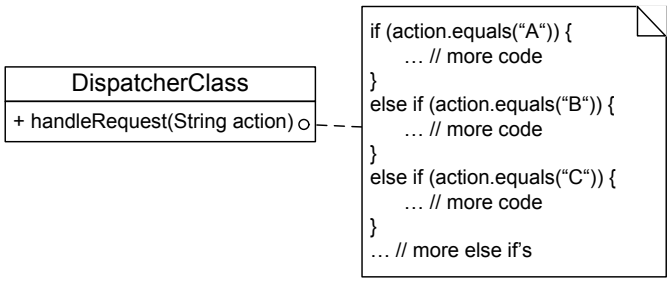

Abbildung 2.1: Beispiel für das Anti Pattern *Conditional Dispatcher*

Abschließend wird das Anti Pattern *Conditional Dispatcher* [Ker04] etwas ausführlicher vorgestellt, da es später häufig zur Veranschaulichung eingesetzt wird. Bei einem Conditional Dispatcher handelt es sich um eine Methode, die Anfragen entgegen nimmt und dazu passende Aktionen ausführt, die zum Beispiel mit Hilfe von verketteten If/Elseif-Anweisungen bestimmt werden.

Abbildung 2.1 zeigt ein verkürztes Beispiel zur Verdeutlichung des Prinzips. Die Methode handleRequest bekommt die Anfrage durch den Parameter action in Form einer Zeichenkette übergeben. Die folgenden If/Elseif-Anweisungen

überprüfen den Wert des Parameters auf bestimmte Literale, um die Behandlung der Anfrage zu bestimmen. In der Regel ist die Anfragebehandlung direkt in dem then-Block der jeweiligen If-Anweisung implementiert.

Das Anti Pattern weist im Wesentlichen zwei Nachteile auf. Zum einen ist die Behandlung von Anfragen hart codiert, so dass ein Austausch der Behandlung bestimmter Anfragen zur Laufzeit nicht möglich ist. Soll sich die Behandlung einer Anfrage irgendwann zur Laufzeit ändern, so muss dies ebenfalls über If-Anweisungen und entsprechende Bedingungen codiert werden. Zum anderen werden solche Methoden oft sehr groß und unübersichtlich. Verschiedene Anfragebehandlungen können auf unterschiedliche Daten angewiesen sein, die über Anfragen hinweg vorgehalten werden müssen. In diesem Fall werden die Daten oft direkt in Attributen der umgebenden Klasse gehalten, wodurch diese zusätzlich größer und unübersichtlicher wird. Zudem sind die Daten auf diese Weise schlecht verkapselt.

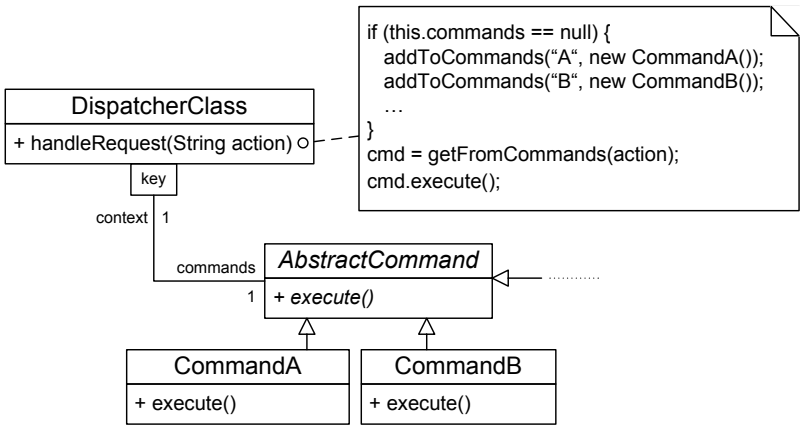

Abbildung 2.2: Verbesserte Struktur mit Hilfe des *Command*-Entwurfsmusters

Eine Conditional-Dispatcher-Implementierung kann groß und unübersichtlich werden und ist wenig flexibel. Eine bessere Lösung verwendet wie in Abbildung 2.2 dargestellt das *Command*-Entwurfsmuster [GHJV95, Ker04]. In dieser Struktur wird für jeden Anfragetyp eine Kommandoklasse erstellt (CommandA, CommandB, ...), in deren execute-Methode jeweils die Anfragebehandlung implementiert wird. Alle Kommandoklassen verfügen über eine gemeinsame abstrakte Basisklasse (AbstractCommand), in der ihre Schnittstelle definiert ist. Die DispatcherClass verfügt über eine qualifizierte Assoziation zur abstrakten Kommandoklasse. Auf diese Weise kann bei ihr unter Verwen-

dung eines Schlüssels eine Instanz einer Kommandoklasse hinterlegt werden. Die handleRequest-Methode initialisiert die Assoziation sofern dies noch nicht geschehen ist. Dazu erzeugt sie von jeder Kommandoklasse eine Instanz und legt sie unter Verwendung des Literals, das die jeweilige Anfrage eindeutig identifiziert, in der Assoziation ab. Anstatt eine lange Kette von If/Elseif-Anweisungen zu verwenden, um die Anfragebehandlung anhand der übergebenen Zeichenkette action zu bestimmen, wird action nun einfach als Schlüssel benutzt, um das zuständige Kommandoobjekt zu ermitteln. An dieses wird die Anfrage dann durch einen Aufruf der execute-Methode delegiert.

Neue Anfragen und deren Behandlung können einfach durch Implementierung einer weiteren Kommandoklasse und Hinzufügen eines addToCommands-Aufrufs zur Initialisierung hinzugefügt werden. Alle Daten, die zur Behandlung einer Anfrage auch über verschiedene Anfragen hinweg benötigt werden, können in Attributen der speziellen Kommandoklassen gehalten werden anstatt in der DispatcherClass selbst. Soll zu irgendeinem Zeitpunkt zur Laufzeit des Programms eine Anfrage anders behandelt werden, so kann einfach ein anderes Kommandoobjekt in der Assoziation hinterlegt werden. In Verbindung mit dynamischem Nachladen von Klassen kann so sogar eine Erweiterung zur Laufzeit vorgenommen werden.

Anstatt einer Kommandostruktur kann auch eine Struktur auf Basis des Entwurfsmusters *Chain of Responsibility* [GHJV95] realisiert werden. Dabei werden auch Klassen für die Behandlung der Anfragen implementiert, deren Objekte zur Laufzeit in einer Kette organisiert werden. Eine Anfrage wird an das erste Objekt der Kette übergeben. Wenn es zuständig ist, behandelt es die Anfrage und sonst gibt es sie an das nächste Objekt der Kette weiter.

Diese Struktur erlaubt komplexere Bedingungen zur Überprüfung der Zuständigkeit als die einfache Prüfung auf ein Literal. Des Weiteren lassen sich so auch Strukturen von aufeinander folgenden If-Anweisungen gemischt mit Elseif-Anweisungen abbilden, bei denen mehrere Anfragebehandlungen für eine Anfrage ausgeführt werden.

2.2 Refactoring

Im vorangehenden Abschnitt 2.1 werden einige Schwachstellen und ihre besseren Lösungen vorgestellt. Bei solchen Schwachstellen handelt es sich nicht um Fehler in Bezug auf die funktionale Korrektheit einer Software, sondern um unvorteilhafte Strukturen. Die besseren Lösungen zeichnen sich daher in der Regel durch eine verbesserte Struktur der Software aus, die weiterhin dasselbe

Verhalten zeigt.

Die Transformation einer Software auf demselben relativen Abstraktionsniveau (zum Beispiel Entwurf oder Implementierung) ohne ihr von außen beobachtbares Verhalten zu verändern, wird nach [CC90] als *Restrukturierung* (engl. *restructuring*) bezeichnet. Die Restrukturierung objektorientierter Systeme wird *Refactoring* genannt.

Der Begriff Refactoring wurde durch Opdyke geprägt, der in [Opd92] die Restrukturierung von objektorientierten Frameworks untersucht und daraus typische Restrukturierungsschritte ableitet, die das von außen beobachtbare Verhalten der Software nicht verändern.

Opdyke definiert das von außen beobachtbare Verhalten informal über den Zusammenhang zwischen Eingaben und Ausgaben eines Programms: ein Programm soll nach einer Restrukturierung für alle Eingaben dieselben Ausgaben liefern wie vor der Restrukturierung. Darüber hinaus gibt es eine Reihe weiterer Aspekte des Verhaltens von Programmen, wie zum Beispiel Ausführungszeiten von bestimmten Vorgängen oder der dabei benötigte Arbeitsspeicher, die je nach Einsatzkontext eines Programms ebenfalls wichtig sein können, von Opdyke aber nicht betrachtet werden.

Opdyke definiert 26 *low-level*-Refactorings, wie zum Beispiel das Erzeugen, Umbenennen, Verschieben von Klassen, Attributen oder Methoden, die zu umfangreicheren Restrukturierungen zusammengesetzt werden können. Für low-level-Refactorings definiert er formale Vorbedingungen, bei deren Erfüllung die Ausführung eines Refactorings das von außen beobachtbare Verhalten nicht verändert. Die Durchführung selbst wird nicht formal definiert.

Fowler et al. beschreiben später in [Fow99] über Bad Smells hinaus 72 umfangreichere Refactorings. Analog zu Entwurfsmustern sind die Beschreibungen einheitlich strukturiert. Jedes Refactoring verfügt über einen *Namen*, so dass sich auch hier bei Entwicklern ein gemeinsames Vokabular entwickelt hat. Des Weiteren werden die *Motivation* für das Refactoring sowie die zu seiner Durchführung notwendigen Schritte (engl. *Mechanics*) beschrieben. Schließlich wird mindestens ein *Beispiel* gegeben, bei dem Quelltexte unter Anwendung der Schritte sukzessive restrukturiert werden.

Ebenfalls analog zur Dokumentation von Entwurfsmustern und Schwachstellen sind die Beschreibungen hochgradig informal. Es werden das allgemeine Vorgehen sowie zu berücksichtigende Besonderheiten angegeben, die ein Entwickler manuell umsetzen muss. Um sicherzustellen, dass die Refactorings keine Verhaltensänderungen vornehmen, werden eine Zerlegung in kleine überschaubare Schritte und die regelmäßige Überprüfung der Software durch eine umfangreiche Testsuite empfohlen.

Ein einfaches Beispiel für ein Refactoring ist *Encapsulate Field*. Das Refactoring beschreibt die Verkapselung eines öffentlichen Attributs durch lesende und schreibende Zugriffsmethoden. Die Zugriffsmethoden werden in einem ersten Schritt erzeugt. Danach werden alle direkten Lese- und Schreibzugriffe auf das Attribut durch einen Aufruf der jeweiligen Zugriffsmethode ersetzt. Schließlich wird die Sichtbarkeit des Attributs auf privat gesetzt.

Ein weiteres sehr bekanntes und häufig eingesetztes Refactoring ist *Extract Method*. Dabei wird eine Menge von Anweisungen in eine eigene Methode ausgelagert und in der ursprünglichen Methode durch einen Aufruf der neuen Methode ersetzt. Ein einfaches Beispiel dafür zeigt Abbildung 2.3.

Beim Extrahieren von Anweisungen in neue Methoden müssen Zugriffe auf nicht mit extrahierte lokale Variablen sowie Parameter der ursprünglichen Methode berücksichtigt werden. Lesezugriffe sind dabei unproblematisch. Gelesene Variablen der Ursprungsmethode müssen in der neuen Methode als Parameter deklariert und beim Aufruf übergeben werden. Greifen die zu extrahierenden Anweisungen schreibend auf eine nicht mit zu extrahierende Variable zu, so muss die extrahierte Methode den neuen Wert für diese Variable als Ergebnis zurückliefern und das Ergebnis muss der Variable beim Aufruf der Methode zugewiesen werden. Wird mehr als eine Variable auf diese Weise geschrieben, ist das Extrahieren nicht ohne Weiteres möglich. Ebenso muss berücksichtigt werden, ob Variablen mit extrahiert würden, auf die später in der Ursprungsmethode noch zugegriffen wird. In diesem Fall ist das Extrahieren der Anweisungen ebenfalls zunächst nicht möglich.

Ein weiterer Aspekt, auf den bei [Fow99] nicht hingewiesen wird, sind Return-Anweisungen. Wird eine Return-Anweisung extrahiert, kann sich durch das Refactoring ein verändertes Verhalten ergeben, denn durch sie wird nun die extrahierte Methode verlassen und nicht mehr die Ursprungsmethode.

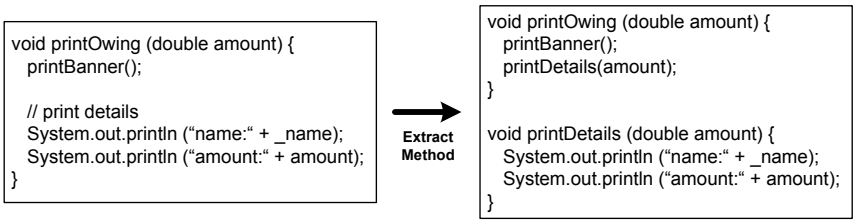

Abbildung 2.3: Beispiel für das *Extract Method*-Refactoring aus [Fow99]

Analog zu [Fow99] beschreibt Kerievsky in [Ker04] weitere Bad Smells sowie 27 Refactorings, die eine Software so restrukturieren, dass dadurch explizit

Entwurfsmuster instanziert werden. Dazu zählt auch *Replace Conditional Dispatcher with Command*, das bereits im vorangehenden Kapitel 2.1 skizziert wurde. Das Refactoring verwendet dabei eine Reihe weiterer Refactorings, wie zum Beispiel *Extract Method*.

2.3 Strukturbasierte Mustererkennung

Die Dokumentation von Entwurfsmustern und Schwachstellen war ein notwendiger Schritt zur Entwicklung eines gemeinsamen Verständnisses von guten und schlechten Entwurfs- und Implementierungslösungen. Da die Muster selbst lange Zeit nicht dokumentiert waren, sind es ihre konkreten Implementierungen noch viel weniger. Auch heute werden Entwurfsmusterimplementierungen häufig nicht dokumentiert, sei es aus Zeit-/Kostengründen, weil es eine unliebsame Aufgabe ist oder weil Muster unbewusst implementiert werden. Implementierungen von Schwachstellen werden noch viel seltener dokumentiert.

Aufgrund des gemeinsamen Verständnisses würde die Kenntnis über in einem System vorhandene Implementierungen von Entwurfsmustern oder auch von Schwachstellen das Verstehen des Systems erleichtern. Darüber hinaus hilft die Kenntnis über Musterimplementierungen auch, gut verständliche und einfacher wartbare Teile einer Software von schwer verständlichen und schlecht wartbaren Teilen zu unterscheiden. Auf solch einer Basis kann der Umfang von anstehenden Wartungsaktivitäten besser eingeschätzt werden.

Daher widmet sich ein Zweig des Reverse Engineerings, insbesondere im Rahmen der Wiedergewinnung von Entwurfsinformationen (engl. *Design Recovery*) [CC90], der Identifikation von Entwurfsmusterimplementierungen in bestehenden Softwaresystemen. Im Rahmen der Forschung sind eine Reihe von Ansätzen zur weitgehend automatischen Erkennung von Entwurfsmusterimplementierungen im Quelltext einer Software entstanden. Einer dieser Ansätze [NSW+02, Nie04] wurde im Fachgebiet Softwaretechnik an der Universität Paderborn entwickelt. Der Ansatz erlaubt die formale Spezifikation von *Strukturmustern*, für die mit Hilfe einer automatisierten statischen Analyse *Instanzen* im Quelltext einer Software gesucht werden.

Da dieser Ansatz im Rahmen der vorliegenden Arbeit für die Erkennung von Schwachstellen eingesetzt und erweitert wird, wird er im Folgenden genauer vorgestellt. Dabei wird zunächst gezeigt, wie der Quelltext einer Software für die Analyse repräsentiert wird. Aufbauend darauf wird die Spezifikation von Strukturmustern erklärt und anschließend wie die Suche danach funktioniert. Abschließend wird gezeigt, wie erkannte Musterinstanzen präsentiert und be-

wertet werden.

2.3.1 Repräsentation eines Softwaresystems

Die strukturbasierte Mustererkennung durchsucht den Quelltext einer Software nach Instanzen von zuvor spezifizierten Mustern. Die Analyse erfolgt nicht direkt im Text, sondern in einer Repräsentation des Quelltextes als zum Teil vereinfachter abstrakter Syntaxgraph. Bei dieser Repräsentation handelt es sich um eine Objektstruktur, die eine Instanz eines objektorientierten Strukturmodells ist, das als UML-Klassendiagramm [Obj] angegeben wird. Sie besteht aus Objekten, die Instanzen von Klassen des Strukturmodells sind, und Links zwischen den Objekten, die Instanzen von Assoziationen des Strukturmodells sind.

Im Rahmen dieser Arbeit werden objektorientierte Systeme analysiert, die in der Programmiersprache JAVA implementiert sind. Daher wird ein Strukturmodell verwendet, das an die abstrakte Syntax von JAVA angelehnt ist. Abbildung 2.4 zeigt einen Ausschnitt dieses Strukturmodells als UML-Klassendiagramm.

Ein JAVA-Programm besteht aus einer Menge von Klassen (Class), die in Paketen (Package) organisiert sind. Die Klassen verfügen jeweils über beliebig viele Methoden (Method) und Attribute (Field). Vererbungen zwischen Klassen werden über die Selbstassoziation subClasses der Klasse Class repräsentiert.

Für die Suche nach Schwachstellen sind insbesondere die Rümpfe von Methoden von Interesse. Jede nicht-abstrakte Methode verfügt über eine Implementierung und damit über einen Anweisungsblock (Block). Ein Block kann via StatementContext beliebig viele Anweisungen (Statement) enthalten. Da ein Block selbst eine Anweisung ist, können Blöcke so auch Blöcke enthalten. Es gibt eine Reihe von weiteren Anweisungen, wie zum Beispiel Return-Anweisungen (Return) oder If-Anweisungen (If), die alle von der abstrakten Klasse Statement erben. If-Anweisungen können via StatementContext wiederum Anweisungen im then- beziehungsweise else-Zweig enthalten.

Anweisungen können sich unterhalb verschiedener Typen von Anweisungen befinden. Um Anweisungen später einfach durch andere Anweisungen austauschen zu können, wurde die Klasse StatementContext eingeführt, die den Kontext, in dem sich eine Anweisung befindet, generisch repräsentiert, so dass er bei einem Austausch nicht exakt spezifiziert werden muss.

Des Weiteren besteht ein Methodenrumpf hauptsächlich aus Ausdrücken (Expression), wie zum Beispiel ein Lesezugriff (VariableAccess) auf eine Variable (Variable) oder ein Methodenaufruf (MethodCall). Ausdrücke können sich analog zu Anweisungen in verschiedenen Kontexten befinden, die durch die

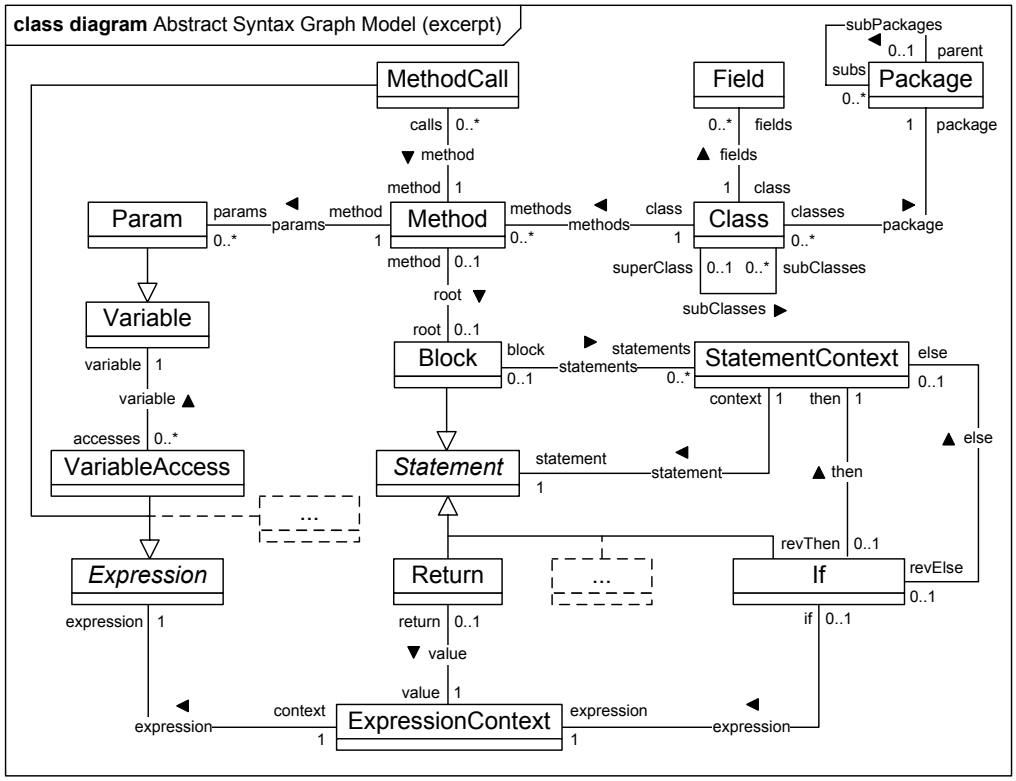

Abbildung 2.4: Ausschnitt aus dem Strukturmodell des abstrakten Syntaxgraphen

Klasse ExpressionContext generisch repräsentiert werden. Das Strukturmodell besteht noch aus einer Reihe weiterer Klassen und Assoziationen, die hier aus Gründen der Übersichtlichkeit nicht aufgeführt werden.

Die strukturbasierte Mustererkennung ist grundsätzlich nicht auf das hier gezeigte Strukturmodell festgelegt. Vielmehr ist das Strukturmodell austauschbar. So wird zum Beispiel in [GMW06] ein Strukturmodell für MATLAB/-SIMULINK [Mat] verwendet. Die strukturbasierte Mustererkennung wird dort eingesetzt, um MATLAB/SIMULINK-Modelle auf die Verletzung von Modellierungsrichtlinien zu untersuchen.

2.3.2 Spezifikation von Strukturmustern

Ein Strukturmuster wird mit Hilfe eines speziellen UML-Objektdiagramms [Obj] spezifiziert, das eine Objektstruktur definiert, die eine Instanz des im

vorherigen Abschnitt eingeführten Strukturmodells repräsentiert und die in einem abstrakten Syntaxgraph existieren muss, damit eine Instanz des Musters vorliegt. Diese Struktur wird später von der Mustererkennung im abstrakten Syntaxgraphen gesucht, indem sie darin auf eine *isomorphe* Struktur abgebildet wird. Dabei werden alle Objekte und Links des Strukturmusters an typkonforme Objekte und Links im abstrakten Syntaxgraphen *gebunden*. Daher enthält ein Strukturmuster nicht tatsächlich Objekte und Links, sondern vielmehr Platzhalter dafür, so genannte *Objekt-* und *Linkvariablen*.

Darüber hinaus spezifiziert ein Strukturmuster, wie eine Fundstelle im abstrakten Syntaxgraphen markiert werden soll. Jedes Strukturmuster spezifiziert dazu zusätzlich eine so genannte *Annotation* beziehungsweise *Annotationsvariable*, die über Linkvariablen mit bestimmten Elementen der Objektstruktur des Musters verbunden ist. Findet die Mustererkennung die gesuchte Struktur in einem abstrakten Syntaxgraphen, werden eine Annotation zusammen mit den Links erzeugt und mit den gebundenen Objekten im Syntaxgraphen verbunden.

Formal wird das Strukturmodell des abstrakten Syntaxgraphen als *Typgraph* und eine Instanz davon als durch den Typgraph *typisierter Graph* definiert. Strukturmuster werden darauf aufbauend mit Hilfe von typisierten *Graphtransformationsregeln* [Roz97] formalisiert. Eine Graphtransformationsregel beschreibt die Modifikation eines Graphen durch Hinzufügen und Entfernen von Knoten und Kanten. Sie besteht aus zwei Graphen, die linke und rechte Regelseite genannt werden. Die linke Regelseite beschreibt einen Teilgraphen, der in einem Wirtsgraphen gesucht wird. Die rechte Regelseite beschreibt wie dieser Teilgraph nach der Transformation beschaffen sein soll.

Knoten und Kanten, die sich in beiden Regelseiten befinden, bleiben unverändert erhalten. Elemente, die in der linken Regelseite enthalten sind, nicht aber in der rechten, werden im Wirtsgraphen gelöscht. Analog werden Elemente der rechten Regelseite, die nicht in der linken Regelseite enthalten sind, erzeugt. Damit eine Graphtransformationsregel auf einen Wirtsgraphen angewendet werden kann, muss ein zur linken Regelseite isomorpher Teilgraph, eine so genannte *Anwendungsstelle*, im Wirtsgraphen gefunden werden.

Bei einem Strukturmuster entspricht die zu suchende Struktur der linken Seite einer Graphtransformationsregel. Die Annotationsvariable und ihre Linkvariablen bilden zusammen mit der zu suchenden Struktur die rechte Seite der Graphtransformationsregel.

Abbildung 2.5 zeigt vier Strukturmuster, die zur Erkennung von Conditional-Dispatcher-Instanzen genutzt werden, die eine If/Elseif-Struktur aufweisen. Das erste Strukturmuster DispatchingElseIfs erkennt Paare von If-

Anweisungen (if1:If und if2:If), die eine If/Elseif-Struktur bilden, indem sie über einen elseStatement-Link miteinander verbunden sind, und deren Bedingungen, repräsentiert durch die beiden :ExpressionContext-Objektvariablen, auf dieselbe Variable variable:Variable zugreifen. Als Bedingungen sind dabei unterschiedliche Ausdrücke denkbar. Das Strukturmuster fordert nur unscharf mit Hilfe von Pfaden, dass es innerhalb der beiden Ausdrücke jeweils einen Variablenzugriff (:VariableAccess) auf variable:Variable geben muss.

Pfade abstrahieren von beliebigen Strukturen des abstrakten Syntaxgraphen, deren konkrete Ausprägung für das Vorliegen einer Musterinstanz nicht von Bedeutung ist. In diesen Fall ist nur wichtig, dass beide Ausdrücke einen Zugriff auf dieselbe Variable enthalten. Beim Binden einer Anwendungsstelle für ein Muster werden Pfade an Ketten von Objekten und Links gebunden, über die das Zielobjekt ausgehend vom Startobjekt erreicht werden kann. Damit die Pfade in diesem Muster dabei den Ausdruck nicht verlassen können, sind sie durch Angabe von (w/o Method, Type) so eingeschränkt, dass sie kein Methoden- oder Typobjekt traversieren dürfen.

Die bisher beschriebenen Elemente des Strukturmusters bilden gemeinsam die zu suchende Struktur. Zur Markierung einer Fundstelle für diese Struktur dienen die ovale Annotationsvariable :DispatchingElseIfs sowie die Linkvariablen if, elseif und variable zwischen der Annotationsvariable und den beiden If-Anweisungen sowie der Variable. Die Annotationsvariable und ihre Linkvariablen werden im abstrakten Syntaxgraphen erzeugt, wenn in ihm die zu suchende Struktur und damit eine Instanz des Musters gefunden werden konnte. Sie sind in der Abbildung grün gefärbt und mit dem Stereotyp ≪create≫ versehen. Die erzeugten Linkvariablen werden auch als die *Schnittstelle* eines Strukturmusters bezeichnet.

Das Strukturmuster DispatchingElseIfs beschreibt mit zwei verketteten If-Anweisungen nur einen Teil einer Conditional-Dispatcher-Instanz, der mehrfach vorliegen muss. Darüber hinaus müssen diese Teile entsprechend miteinander verbunden sein. Dies wird durch weitere Strukturmuster beschrieben, die aufeinander aufbauen.

So baut das Muster FirstDispatcherIf auf eine DispatchingElseIfs-Annotation auf und erkennt die erste If-Anweisung if1:If einer Dispatcher-Instanz. Damit eine Instanz für dieses Strukturmuster gefunden werden kann, muss zuvor eine Instanz des DispatchingElseIfs-Muster gefunden und annotiert worden sein. Bei der gebundenen If-Anweisung muss es sich um die erste Anweisung handeln, da explizit verboten wird, dass es eine weitere DispatchingElseIfs-Annotation gibt, die die Anweisung in der Rolle eines elseif annotiert. Zusätzlich werden die Methode und die Variable gebunden und zusammen mit der If-Anweisung

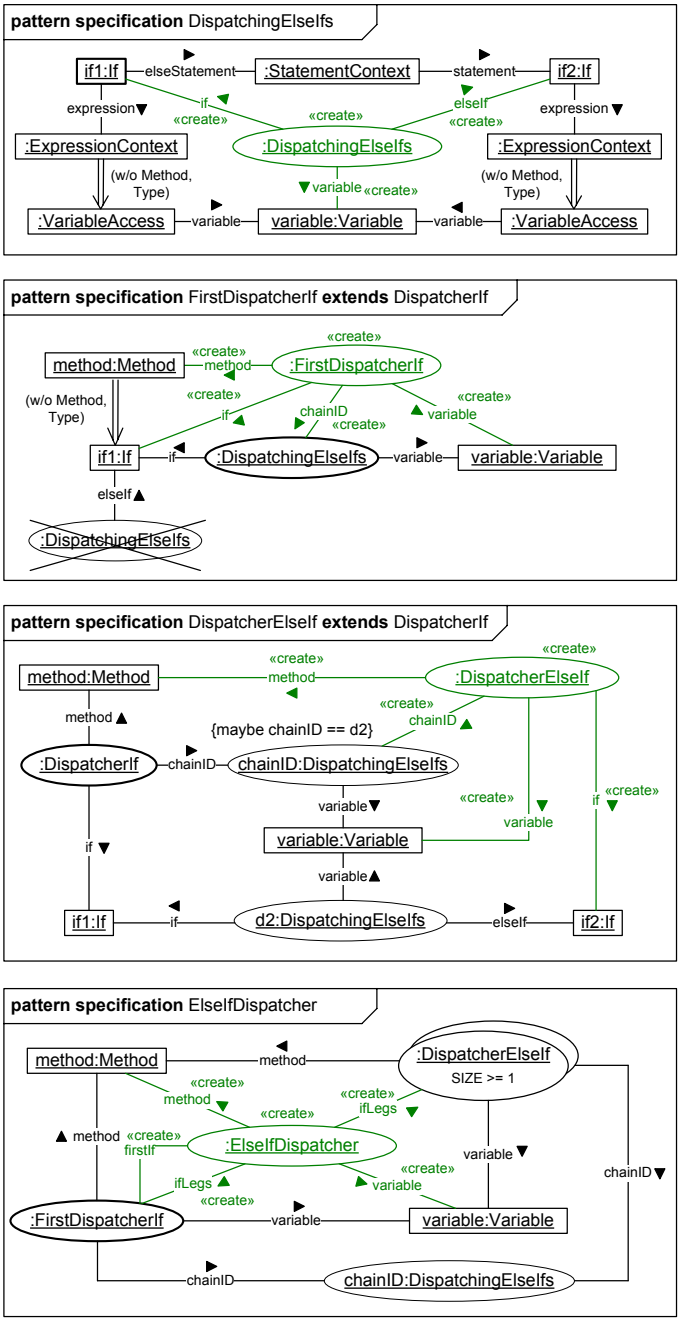

Abbildung 2.5: Musterspezifikationen zur Erkennung eines *Conditional Dispatcher* mit If/Elseif-Struktur

als method und variable annotiert. Darüber hinaus wird die DispatchingElseIfs-Annotation als chainID annotiert. Dies wird von den weiteren Strukturmustern dazu genutzt, um nur zusammenhängende Dispatcher-Instanzen zu erkennen, deren If-Anweisungen eine ununterbrochene Kette bilden.

Die Elseif-Anweisungen eines Dispatchers werden durch das Strukturmuster DispatcherElseIf erkannt. Dabei kann es sich sowohl um die Elseif-Anweisung der ersten If-Anweisung des Dispatchers als auch bereits einer Elseif-Anweisung handeln. Dies wird dadurch möglich, dass die Strukturmuster FirstDispatcherIf und DispatcherElseIf von dem abstrakten Strukturmuster DispatcherIf (nicht in der Abbildung dargestellt) erben. Vererbung wird bei Strukturmustern eingesetzt um auszudrücken, dass ein Muster eine speziellere Variante eines anderen Musters ist. Dabei erben Strukturmuster die Schnittstelle des anderen Musters, dass heißt, sie verbinden dieselben Typen von Objekten mit der Annotation unter Verwendung derselben Rollenbezeichner. Abstrakte Strukturmuster definieren nur eine Schnittstelle und keine vollständige Struktur.

Das Muster DispatcherElseIf baut auf eine DispatcherIf-Annotation auf und bindet von ihr aus die annotierte If-Anweisung if1:If und über d2:DispatchingElseIfs die Elseif-Anweisung if2:If, sowie die behandelte Variable variable:Variable. Zusätzlich wird die chainID gebunden. Bei der Erkennung der ersten *Elseif*-Anweisung eines Dispatchers muss es sich bei chainID und d2 um dieselbe Annotation handeln. Dies wird durch {maybe chainID == d2} erlaubt. Normalerweise wird die zu suchende Struktur eines Strukturmusters an eine isomorphe Struktur im abstrakten Syntaxgraphen gebunden, dass heißt, dass keine zwei Elemente des Musters an dasselbe Element im Syntaxgraphen gebunden werden. Durch eine maybe-Anweisung kann für zwei Objektvariablen des Musters erlaubt werden, dass sie an dasselbe Objekt im Syntaxgraphen gebunden werden.

Wird die Struktur gefunden, annotiert das Muster die Methode mit method, die Elseif-Anweisung mit if, die Variable mit variable sowie chainID als chainID.

Schließlich erkennt und annotiert das Muster ElseIfDispatcher ausgehend von einer FirstDispatcherIf-Annotation Dispatcher-Instanzen. Dazu werden zunächst die Methode, die Variable und die chainID gebunden und davon ausgehend alle DispatcherElseIf-Annotationen, die dieselbe Methode, Variable und chainID annotieren. Enthält eine Methode mehrere Ketten von If/Elseif-Anweisungen, die zwar alle dieselbe Variable behandeln aber zwischen denen sich weitere Anweisungen befinden, so werden diese Ketten dadurch als eigenständige Dispatcher-Instanzen annotiert.

Mehrere Strukturmuster werden zu *Strukturmusterkatalogen* zusammengefasst. Abbildung 2.6 zeigt einen Musterkatalog bestehend aus den zuvor gezeig-

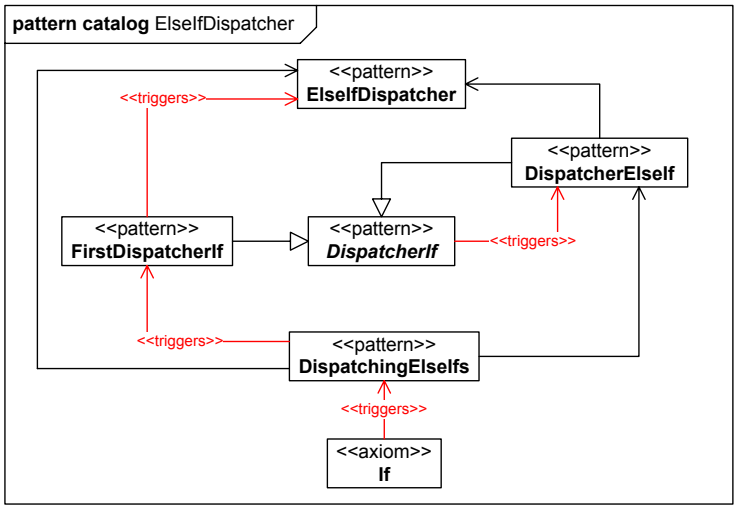

Abbildung 2.6: Strukturmusterkatalog mit Musterabhängigkeiten

ten Strukturmustern. Die Strukturmuster werden durch die mit ≪pattern≫ markierten Knoten repräsentiert. Die Abhängigkeiten zwischen den Mustern werden durch Kanten dargestellt. Ein Strukturmuster ist von einem anderen Strukturmuster strukturell abhängig, wenn es durch Annotationsvariablen in seiner linken Regelseite die Existenz von Instanzen des anderen Musters fordert. Die Pfeilspitze zeigt dabei auf das abhängige Muster. Darüber hinaus werden Vererbungsbeziehungen zwischen Strukturmustern in derselben Notation wie Vererbungen in UML-Klassendiagrammen dargestellt.

Bei den strukturellen Abhängigkeiten wird zwischen so genannten *Trigger-Abhängigkeiten* und reinen strukturellen Abhängigkeiten unterschieden. Die Trigger-Abhängigkeiten sind rot gefärbt und mit ≪triggers≫ markiert. Sie werden in den Strukturmustern definiert, indem ausgewählte Objektvariablen als so genannte *Trigger* ausgezeichnet werden. Im Strukturmuster DispatchingElseIfs wird zum Beispiel die Objektvariable if1:If als Trigger ausgezeichnet (dargestellt durch eine fette Umrandung). Dies bedeutet, dass wann immer im abstrakten Syntaxgraphen ein Objekt vom Typ If gefunden wird, von ihm ausgehend versucht werden soll, das gesamte Strukturmuster zu binden.

Ein solches Objekt löst die Anwendung des Strukturmusters in seinem Kontext aus. Bei einem If-Objekt handelt es sich um ein Objekt, das bereits im Syntaxgraphen vorhanden sein muss und nicht durch Anwendung irgendeines Strukturmusters erzeugt werden kann. Solche Elemente werden als *Axiome* bezeichnet. Da ein If-Objekt ein Trigger für das DispatchingElseIfs-Muster ist,

wird es mit ≪axiom≫ gekennzeichnet im Musterkatalog aufgenommen.

Über Objektvariablen hinaus können auch Annotationsvariablen als Trigger ausgezeichnet werden. Dies bedeutet, wann immer eine Annotation des entsprechenden Typs erzeugt werden konnte, soll von ihr ausgehend versucht werden, eine andere Musterinstanz zu binden. Zum Beispiel löst eine DispatcherIf-Annotation die Anwendung des DispatcherElseIf-Strukturmusters aus. Aufgrund der Vererbungsbeziehungen zwischen den Mustern kann dies sowohl durch eine FirstDispatcherIf- als auch eine DispatcherElseIf-Annotation geschehen.

2.3.3 Erkennungsprozess

Die Strukturmuster beschreiben deklarativ, wie Instanzen eines Musters im Syntaxgraphen aussehen. Die Spezifikationen werden initial auf Basis der informalen Beschreibung der Mustern erstellt, wobei versucht wird, möglichst viele Implementierungsvarianten mit möglichst wenigen Spezifikationen zu erkennen.

Allerdings ist es nicht möglich, alle Implementierungsvarianten vorherzusehen. Zudem weisen konkrete Softwaresysteme häufig Besonderheiten in ihrer Implementierung auf, sei es aufgrund besonderer Vorlieben der Entwickler oder aufgrund von Implementierungsrichtlinien, zum Beispiel für die Implementierung von Assoziationen. Die initialen Strukturmusterspezifikationen müssen in einer Kalibrierungsphase an diese Besonderheiten angepasst werden. In [NSW+02, Nie04] wird daher ein iterativer Erkennungsprozess definiert.

Damit Instanzen der Muster im Syntaxgraphen gefunden werden, müssen sie durch einen Algorithmus in geeigneter Reihenfolge an geeigneter Stelle im Syntaxgraphen gesucht beziehungsweise *angewendet* werden. Strukturmuster können aufeinander aufbauen, indem sie die Existenz von Instanzen anderer Muster fordern. Daher müssen abhängige Strukturmuster prinzipiell nach den Strukturmustern angewendet werden, von denen sie abhängig sind, damit die geforderten Musterinstanzen bereits gefunden sein können. Des Weiteren müssen Trigger-Abhängigkeiten berücksichtigt werden.

Die strukturbasierte Mustererkennung erhält den Quelltext des zu analysierenden Systems sowie einen Musterkatalog als Eingabe. Der Quelltext wird in die Repräsentation als abstrakter Syntaxgraph überführt. Der Musterkatalog wird auf Basis der Abhängigkeiten topologisch in Ebenen sortiert, so dass sich in der untersten Ebene die Axiome befinden. Der Katalog in Abbildung 2.6 ist topologisch sortiert angeordnet.

Typische Algorithmen würden so vorgehen, dass sie Strukturmuster Ebene für Ebene von unten nach oben (*bottom-up*) anwenden. Auf diese Weise müssen für jedes Strukturmuster zum Zeitpunkt seiner Anwendung alle Vorbedingungen bereits erfüllt sein sofern dies im Syntaxgraph möglich ist. Der Nachteil solcher Ansätze ist, dass Strukturmuster der höchsten Ebenen erst ganz am Ende angewendet werden. Gerade diese Strukturmuster beschreiben aber typischerweise besonders interessante Ergebnisse, wie zum Beispiel Entwurfsmuster oder wie in diesem Fall einen Elseif-Dispatcher. Wünschenswert ist, solche hochwertigen Ergebnisse möglichst frühzeitig zu erzielen. Dies gilt insbesondere für die ersten Iterationen des Erkennungsprozesses zur Kalibrierung der Spezifikationen. Daher wurde in [NSW+02, Nie04] ein Algorithmus entwickelt, der durch ein kombiniertes *bottom-up/top-down*-Vorgehen versucht, hochwertige Ergebnisse frühzeitig zu erzielen.

Der Algorithmus beginnt damit, alle Objekte im Syntaxgraphen zu ermitteln, die typkonform zu einem Axiom sind. Für jedes gefundene Objekt, werden die Strukturmuster zur Anwendung vorgesehen, die über eine Trigger-Abhängigkeit vom Axiom verfügen. Dazu werden *Muster-Kontext-Paare* aus anzuwendendem Strukturmuster und jeweils einem Axiom-Objekt gebildet und in einer *Prioritätswarteschlange* einsortiert. Im Beispiel wird für jede If-Anweisung ein Paar mit dem DispatchingElseIfs-Strukturmuster gebildet.

Die Warteschlange sortiert die enthaltenen Paare so gemäß der Ebene, auf der sich das Strukturmuster befindet, dass Muster höherer Ebenen zuerst behandelt werden. Danach werden die Paare der Reihe nach abgearbeitet, indem für jedes Paar das Strukturmuster unter Verwendung des Kontextobjektes angewendet wird. Ist die Anwendung erfolgreich und entsteht eine neue Annotation, werden unter Verfolgung der Trigger-Abhängigkeiten im Musterkatalog (bottom-up) neue Muster-Kontext-Paare aus auszulösenden Strukturmustern und der neuen Annotation als Kontextobjekt gebildet und in die Prioritätswarteschlange einsortiert.

Im Beispiel wird durch eine DispatchingElseIfs-Annotation das FirstDispatcherIf-Muster ausgelöst, indem ein Muster-Kontext-Paar bestehend aus der Annotation und dem Muster in die Schlange einsortiert wird. Da sich das FirstDispatcherIf-Muster auf einer höheren Ebene befindet, wird das Paar vorne einsortiert und als nächstes verarbeitet. Wird eine FirstDispatcherIf-Annotation gefunden, so wird zum einen aufgrund der Vererbungsbeziehung zu DispatcherIf das DispatcherElseIf-Muster ausgelöst. Zum anderen wird bereits das ElseIfDispatcher-Muster ausgelöst.

Bei diesem Vorgehen kann es passieren, dass Strukturmuster angewendet werden sollen, für die noch nicht alle Vorbedingungen in Form von geforder-

ten Annotationen vorliegen müssen, da die zu deren Etablierung notwendigen Strukturmuster zu dem Zeitpunkt noch nicht angewendet wurden. In diesem Fall könnte die Anwendung allein aufgrund ihres Zeitpunkts fehlschlagen.

Der Algorithmus wechselt hier in den *top-down*-Modus. Ausgehend vom aktuellen Kontextobjekt werden im Syntaxgraphen alle Kontextobjekte ermittelt, die für die Anwendung der fehlenden Strukturmuster benötigt werden. Daraus werden Muster-Kontext-Paare gebildet, die in einem Stack verwaltet werden. Ist das Strukturmuster eines Paares ebenfalls von Strukturmustern abhängig, so werden für diese analog Muster-Kontext-Paare gebildet. Dies wird rekursiv fortgesetzt. Die Paare im Stack werden schließlich der Reihe nach abgearbeitet. Danach müssen alle möglichen Vorbedingungen im Kontext erfüllt worden sein, so dass die Anwendung des hochwertigen Musters vorgenommen werden kann. Schlägt es dennoch fehl, so liegt in dem Kontext keine Musterinstanz vor. Nach Beendigung des top-down-Modus wird im bottom-up-Modus mit der Abarbeitung der Schlange fortgefahren. Der Algorithmus endet, wenn die Schlange leer ist.

Der Algorithmus verhindert, dass ein Strukturmuster mehrfach auf dasselbe Kontextobjekt angewendet wird. Dies steigert zum einen die Effizienz und zum anderen wird vermieden, dass identische Instanzen mehrfach annotiert werden. Des Weiteren werden Ergebnisse, die im top-down-Modus erzielt werden, auch in die bottom-up-Warteschlange einsortiert, da sie bei reinem bottom-up-Vorgehen ohnehin irgendwann gefunden worden wären.

Der Re-Engineer hat bereits während der Analyse die Möglichkeit, gefundene Musterinstanzen zu inspizieren. Darüber hinaus kann er die Analyse anhalten und ihren weiteren Verlauf beeinflussen, indem er zum Beispiel die weitere Verarbeitung von gefundenen Annotationen unterbindet oder selbst Annotationen einfügt. Auf diese Weise kann auch Vorwissen des Re-Engineers über das System in die Analyse einfließen.

Die Ergebnisse der strukturbasierten Mustererkennung werden dem Re-Engineer grafisch in wiedergewonnenen UML-Klassendiagrammen des Systems präsentiert. Dies wird in Abbildung 2.7 im oberen Teil gezeigt. Annotationen werden als Ovale dargestellt und mit annotierten Elementen, die im Klassendiagramm sichtbar sind, über beschriftete Kanten verbunden. Die Abbildung zeigt darüber hinaus, wie prinzipiell ElseIfDispatcher-Instanzen insgesamt im abstrakten Syntaxgraphen durch eine Menge von Annotationen markiert werden.[1] Wie der untere Teil der Abbildung zeigt, sind längst nicht alle annotier-

[1] Die Abbildung vermischt zur Illustration abstrakte und konkrete Syntax und zeigt nicht alle Annotationen und Verbindungen.

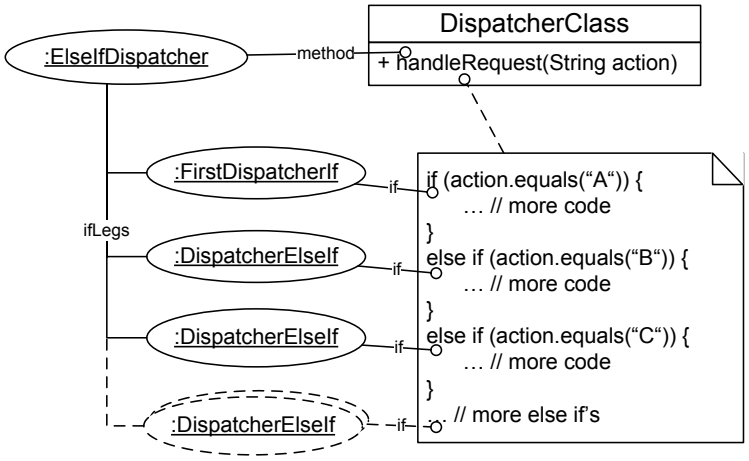

Abbildung 2.7: Prinzipielle Darstellung einer annotierten Conditional-Dispatcher-Instanz

ten Elemente im Klassendiagramm sichtbar. Daher werden alle Annotationen zusätzlich tabellarisch präsentiert, mit Navigationsunterstützung zu den entsprechenden Stellen entweder im Klassendiagramm oder direkt im Quelltext.

2.3.4 Bewertung der Ergebnisse

Bei der Implementierung von Entwurfsmustern oder Schwachstellen kann es eine Reihe verschiedener Varianten geben. Die Strukturmuster werden so spezifiziert, dass sie auf die wesentlichen Aspekte eines Muster, sozusagen den kleinsten gemeinsamen Nenner der Varianten, fokussieren und dadurch möglichst mehrere Varianten erkennen. Die Spezifikationen werden dadurch zu einem gewissen Grad unscharf, so dass bei der Erkennung Strukturen als Musterinstanzen erkannt werden können, die zwar der Spezifikation entsprechen aber tatsächlich keine Instanzen des Musters sind. Solche falschen Ergebnisse werden *false-positives* genannt.

Der Ansatz aus [Nie04] versucht diese Unschärfe eines Strukturmusters durch eine Kennzahl auszudrücken, die durch ein Muster korrekt erkannte Funde zu allen durch das Muster erkannten Funden einschließlich false-positives ins Verhältnis setzt. Diese Kennzahlen müssen initial durch den Re-Engineer geschätzt und dann auf Basis von Erfahrungswerten angepasst werden. Diese Art der Bewertung versucht auszudrücken, wie gut ein Strukturmuster im Allgemeinen das spezifizierte Entwurfsmuster beziehungsweise die spezifizierte

Schwachstelle erkennt. Konkret durch ein Strukturmuster erkannte Instanzen werden nicht bewertet und dem Re-Engineer gleichwertig präsentiert.

In [Wen07, Tra06] wird daher ein anderer Ansatz der Bewertung entwickelt, der in der vorliegenden Arbeit für die Erkennung von Schwachstellen erweitert wird. Bei diesem Ansatz wird bei der Definition eines Strukturmusters wiederum ein kleinster gemeinsamer Nenner einer Reihe von Varianten spezifiziert, der zwingend gefunden werden muss, damit eine Musterinstanz vorliegt. Darüber hinaus werden zusätzliche Eigenschaften angegeben, die nicht zwingend erfüllt sein müssen, die aber wenn sie erfüllt sind, eine höhere Übereinstimmung mit dem spezifizierten Entwurfsmuster beziehungsweise der spezifizierten Schwachstelle bedeuten. Auf Basis von Zusatzeigenschaften wird dann eine Bewertung gefundener Instanzen berechnet: je mehr Zusatzeigenschaften erfüllt sind, desto höher wird die Instanz bewertet.

Dies wird im Folgenden anhand des *Singleton*-Entwurfsmuster erklärt. Ein Singleton ist eine Klasse, von der es zur Laufzeit nur eine Instanz im System gibt. Diese eine Instanz wird von der Klasse selbst verwaltet und auf Anfrage zunächst erzeugt, sofern dies noch nicht geschehen ist, und dem anfragenden Klienten zur Verfügung gestellt.

Die Implementierung eines Singleton weist einige charakteristische Merkmale auf. Zunächst muss die Klasse über ein statisches Attribut verfügen, mit dem es seine einzige Instanz verwaltet. Im Strukturmuster in Abbildung 2.8 wird dies durch die Objektvariable instance:Field spezifiziert, die zu den Attributen von c:Class gehört und dieselbe Klasse als Typ hat. Durch die Bedingung static == true wird gefordert, dass das Attribut statisch ist.

Ein Singleton muss seine einzige Instanz auf Anfrage zur Verfügung stellen. Dies kann zum einen dadurch erfolgen, dass das Instanzattribut öffentlich zugänglich ist. Zum anderen kann es eine statische Zugriffsmethode für das Instanzattribut geben. Eine Implementierung mit Hilfe eines öffentlichen Attributs ist problematisch, da es von außen nicht nur gelesen, sondern auch geschrieben werden kann. Das Singleton hat also nicht die volle Kontrolle über die Lebenszeit seiner Instanz. Grundsätzlich bleibt zu diskutieren, ob es sich bei einer solchen Implementierung noch um ein Singleton handelt oder ob der Entwickler seinerzeit überhaupt ein Singleton implementieren wollte und es fehlerhaft umgesetzt hat, so dass es nun eine Schwachstelle darstellt.

In dem Strukturmuster wird die Bedingung, dass das Attribut private Sichtbarkeit hat, als Zusatzeigenschaft (gekennzeichnet durch den Zusatz {additional}) spezifiziert. Ist die Bedingung erfüllt, wird eine Instanz höher bewertet als eine, bei der sie nicht erfüllt ist. Darüber hinaus wird eine statische Zugriffsmethode durch weitere Zusatzeigenschaften spezifiziert, die durch

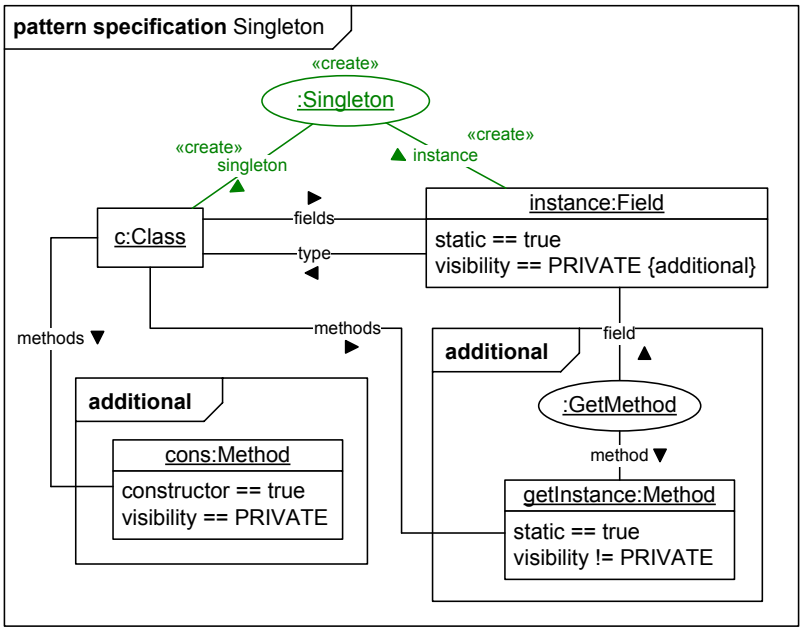

Abbildung 2.8: Strukturmuster mit Zusatzbedingungen

ein additional-Fragment gekennzeichnet werden. Ist eine Zugriffsmethode vorhanden, wird eine Instanz noch höher bewertet.

Ein weiterer Aspekt einer Singleton-Implementierung ist der Konstruktor der Klasse. Bei einer hundertprozentigen Implementierung muss der Konstruktor privat sein, damit nur die Klasse selbst Instanzen von sich erzeugen kann. Auch hier stellt sich die Frage, ob eine Implementierung mit öffentlichem Konstruktor noch eine Singleton-Implementierung ist. Ebenso kann es sich um eine fehlerhafte Implementierung handeln. Daher wird ein privater Konstruktor ebenfalls als Zusatzeigenschaft definiert.

Zur Berechnung einer Bewertung für eine Instanz werden die durch die Instanz erfüllten Eigenschaften zu allen Eigenschaften eines Strukturmusters einschließlich der Zusatzeigenschaften ins Verhältnis gesetzt. Als Eigenschaften werden Objektvariablen, Linkvariablen und Bedingungen berücksichtigt, die zusätzlich gewichtet werden können. So kann zum Beispiel das Vorhandensein eines privaten Konstruktors hoch gewichtet werden, so dass sich dadurch große Unterschiede in der Bewertung von Singleton-Instanzen ergeben. Der Re-Engineer kann dann die gefundenen Instanzen nach ihrer Bewertung sortieren und gezielter inspizieren.

2.3.5 Erweiterung um quantitative Merkmale

Mit Hilfe der Musterspezifikationen aus Abschnitt 2.3.2 werden Conditional-Dispatcher-Instanzen zuverlässig erkannt. Eine Instanz liegt vor, wenn es in einer Methode mindestens eine If-Anweisung mit einer Elseif-Anweisung gibt, deren Bedingungen dieselbe Variable prüfen. Solche minimalen Instanzen werden genauso annotiert und dem Re-Engineer präsentiert, wie Instanzen mit zahlreichen Anweisungen. Letztere stellen aber deutlich signifikantere Funde dar. Ein weiteres Kriterium, das die Signifikanz von Conditional-Dispatcher-Instanzen bestimmt, sind die Anzahl der Anweisungen, die sich in den then-Zweigen der verketteten If-Anweisungen befinden. Je mehr dies sind, desto größer ist der Gesamtumfang des Dispatchers und desto signifikanter ist die Schwachstelle.

Schwachstellen beziehen sich häufig auf quantifizierbare Merkmale, die dazu genutzt werden können, signifikante, schwerwiegende Schwachstellen von weniger schwerwiegenden zu unterscheiden. Im Falle einer Conditional-Dispatcher-Instanz kann dies auf Basis zum einen der Anzahl der zugehörigen If-Anweisungen sowie deren Umfangs, bestimmt durch die Anzahl der Anweisungen im then-Zweig, geschehen. Die Anzahl der Attribute und Methoden einer Klasse in Verbindung mit der Anzahl der Quelltextzeilen kann zum Beispiel Hinweise auf *Lazy*, *Data* oder *God Classes* liefern beziehungsweise zur Bewertung ihrer Signifikanz genutzt werden. Dies lässt sich mit der bisherigen Musterspezifikation nicht ausdrücken.

Quantitative Merkmale werden mit Hilfe von Softwareproduktmetriken berechnet. Bekannte Metriken berechnen zum Beispiel die Anzahl Quelltextzeilen in einer Methode oder Klasse (engl. Lines Of Code, LOC), Anzahl Anweisungen einer Methode (Number Of Statements, NOS) Anzahl Methoden (Number Of Methods, NOM) oder Attribute (Number Of Attributes, NOA) einer Klasse [CK94, FP96, HS96, LK94].

Um quantitative Merkmale einbeziehen zu können, wurde die Mustererkennung gemeinsam mit [Wen07, Tra06] um die Spezifikation und Überprüfung von Aussagen über Metrikwerte von Elementen des abstrakten Syntaxgraphen erweitert. Dies geschieht indem einer Objektvariable eines Strukturmusters, analog zu Attributbedingungen, so genannte *Metrikbedingungen* hinzugefügt werden können.

Eine solche Metrikbedingung erlaubt die Definition eines Schwellwerts, den eine bestimmte Metrik für das Element des abstrakten Syntaxgraphen (mindestens/höchstens) erreichen, übertreffen oder unterschreiten muss/darf, damit eine Instanz des Musters vorliegt. Solche Metrikbedingungen können zu

Ja/Nein-Entscheidungen über das Vorliegen einer Musterinstanz führen und damit die untersuchten Elemente des abstrakten Syntaxgraphen einschränken. Ebenso können sie als nicht notwendig ({additional}) gekennzeichnet werden und auf diese Weise bereits eine Bewertung einer Musterinstanz bewirken: ist eine Metrikbedingung erfüllt, zum Beispiel weil eine If-Anweisung eines Dispatchers mehr als 5 Anweisungen in ihrem then-Zweig enthält, so wird die *DispatcherIf*-Instanz gemäß der Gewichtung der Metrikbedingung höher bewertet.

Damit auch eine kontinuierliche Bewertung in Abhängigkeit eines Metrikwertes möglich ist, werden *Fuzzy-Metrikbedingungen* eingeführt. Für solche Bedingungen kann eine mathematische Funktion angegeben werden, die einen gegebenen Metrikwert auf einen Wert zwischen 0 und 1 abbildet. Dieser Wert wird bei der Bewertung der Musterinstanz mit dem Gewicht der Metrikbedingung multipliziert. Auf diese Weise kann eine If-Anweisung eines Dispatchers umso höher bewertet werden desto mehr Anweisungen ihr then-Zweig enthält.

Abbildung 2.9 zeigt ein erweitertes FirstDispatcherIf-Strukturmuster. Darin wird ausgehend von if1:If zusätzlich die den then-Zweig repräsentierende Anweisung then:Statement gebunden. Für diese Anweisung wurde eine Fuzzy-Metrikbedingung spezifiziert. Die Bedingung verwendet die skizzierte Funktion, um die Anzahl der in then enthaltenen Anweisungen (Metrik Number Of Statements, NOS) auf einen Wert zwischen 0 und 1 abzubilden. Dieser Wert geht gemäß der Gewichtung der Metrikbedingung in die Bewertung der Musterinstanz ein.

Die Gewichtung der Bedingung sollte so gewählt werden, dass sie einen Großteil der Bewertung ausmacht. Zudem sollte eine Bewertungsfunktion zum einen so gewählt werden, dass sie Metrikwerte im interessanten Bereich gut differenziert bewertet. Zum anderen sollte sie sich asymptotisch einem Wert annähern, damit beliebig hohe oder niedrige Werte immer noch unterschiedlich bewertet werden. Auf diese Weise bleibt eine Sortierung möglich. Exakte Werte für die Bewertungsfunktion wurden im Rahmen der Evaluierung in Abhängigkeit des analysierten Systems bestimmt (siehe Kapitel 6). Das Strukturmuster DispatcherElseIf aus Abbildung 2.5 wird analog erweitert.

Das Strukturmuster ElseIfDispatcher bindet eine Menge von DispatcherElseIf-Annotationen. Die Mengenbedingung SIZE ≥ 1 schreibt vor, dass die Menge mindestens ein Element enthalten muss. Die Größe einer Menge entspricht einer speziellen Metrik. Daher kann auch für sie wie in Abbildung 2.9 gezeigt eine Fuzzy-Metrikbedingung angegeben werden. In diese Funktion geht allerdings nicht nur die Anzahl der Elemente der Menge ein, sondern auch deren Bewertung. Dazu werden alle Bewertungen der enthaltenen Elemente aufsum-

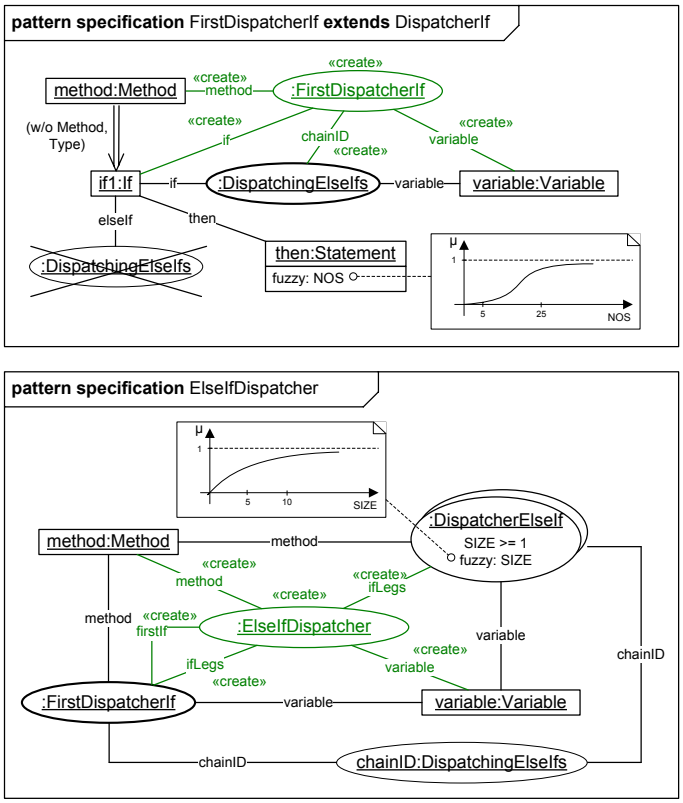

Abbildung 2.9: Um Fuzzy-Metrikbedingungen erweiterte Conditional-Dispatcher-Musterspezifikationen

miert und mit Hilfe der Funktion auf einen Wert zwischen 0 und 1 skaliert. Dieser wird dann mit dem Gewicht multipliziert, den ein Element der Menge hat. [2]

Auf diese Weise werden insgesamt sowohl die Anzahl der If-Anweisungen einer Dispatcher-Instanz als auch deren Umfang zur Bewertung genutzt. Gefundene Schwachstellen können dem Re-Engineer auf Basis ihrer Bewertung sortiert präsentiert werden, so dass er gezielt Funde mit hoher Signifikanz inspizieren kann, um Schwachstellen zur Verbesserung zu bestimmen.

[2] Weitere Details zur Bewertung von Musterinstanzen auf Basis von Fuzzy-Metrikbedingungen sowie zur Umsetzung des Verfahrens können [Tra06] entnommen werden.

2.4 Story Driven Modeling

Bei der strukturbasierten Mustererkennung werden Strukturmuster mit Hilfe von Graphtransformationsregeln [Roz97] formalisiert. Die Graphtransformationsregeln erweitern den abstrakten Syntaxgraphen, allgemein gesprochen den Wirtsgraphen, dabei lediglich um einen Knoten und einige Kanten zur Markierung von Fundstellen. Die Theorie der Graphtransformationsregeln erlaubt weitaus umfangreichere Modifikationen und ist damit grundsätzlich auch zur formalen Beschreibung von Refactorings oder allgemein Programmtransformationen geeignet, die durch die Mustererkennung gefundene Schwachstellen verbessern.

In den letzten Jahren wurde im Fachgebiet Softwaretechnik der Universität Paderborn mit dem so genannten *Story Driven Modeling* [FNTZ98, KNNZ00, Zün02] ein Ansatz zur modellbasierten Struktur- und Verhaltensspezifikation basierend auf der Theorie der Graphtransformationen entwickelt. Der Ansatz verwendet UML-Klassendiagramme [Obj] zur Modellierung der Struktur einer Software. Zur Modellierung des Verhaltens werden *Story Diagramme* eingesetzt, die die Rümpfe von in Klassendiagrammen deklarierten Methoden beschreiben.

Story Diagramme sind spezielle UML-Aktivitätsdiagramme, deren Aktivitäten aus *Story Pattern* bestehen, die mit Hilfe spezieller UML-Objektdiagramme die Manipulation von Objektstrukturen beschreiben und die mit Graphtransformationsregeln formal definiert werden.

In klassischen Graphtransformationssystemen [Roz97] werden anwendbare Graphtransformationsregeln auf allen Anwendungsstellen nichtdeterministisch sequentiell oder parallel angewendet. In vielen praktischen Anwendungen, wie zum Beispiel auch bei der Spezifikation von Programmtransformationen, wird eine stärkere Kontrolle darüber benötigt, wann und wie oft einzelne Graphtransformationsregeln angewendet werden. Bei klassischen Graphtransformationssystemen führt dies dazu, dass Hilfselemente eingeführt werden, die zum Beispiel durch eine Regel erzeugt und deren Existenz durch eine andere Regel gefordert wird, so dass implizit eine sequentielle Anwendung der beiden Regeln erzwungen wird (siehe auch [Sch91]). Story Diagramme gehen hier einen anderen Weg, indem sie explizit Kontrollfluss (Sequenzen, Alternativen, Schleifen) modellieren, um die Anwendung einzelner Graphtransformationsregeln zu steuern.

Das ebenfalls im Fachgebiet Softwaretechnik entwickelte Werkzeug Fujaba [Fuj] verfügt über Editoren zur grafischen Spezifikation der Diagramme. Darüber hinaus bietet es auf Basis der Formalisierung eine automatische Ge-

nerierung von ausführbarem JAVA-Code für die Spezifikationen.

Story Driven Modeling bildet die Grundlage für die Spezifikation von Programmtransformationen im Rahmen der vorliegenden Arbeit und wird daher im Folgenden genauer vorgestellt. Dabei wird mit den Story Pattern begonnen, die einen wesentlichen Bestandteil der darauf folgenden Story Diagramme bilden. Abschließend wird ein Ansatz zur automatischen Verifikation von Story Pattern auf Einhaltung definierbarer Kriterien vorgestellt. Dieser Ansatz wird in der vorliegenden Arbeit aufgegriffen und erweitert, um automatisiert Aussagen zur Verhaltenserhaltung durch spezifizierte Programmtransformationen zu beweisen.

2.4.1 Story Pattern

Ein Story Pattern beschreibt die Modifikation einer Objektstruktur, die eine Instanz eines mit UML-Klassendiagrammen definierten Strukturmodells ist. Es wird durch ein spezielles UML-Objektdiagramm spezifiziert, das zum einen (analog zu Strukturmustern, siehe Abschnitt 2.3.2) eine Struktur definiert, die in einer Objektstruktur gefunden werden muss. Darüber hinaus wird definiert, welche Elemente der Objektstruktur durch die Anwendung des Story Pattern gelöscht werden sollen. Solche Elemente werden in rot dargestellt und mit dem Stereotyp ≪destroy≫ markiert. Ebenso werden Elemente spezifiziert, die neu erzeugt werden sollen. Sie werden in grün dargestellt und mit dem Stereotyp ≪create≫ markiert.

Ein Story Pattern wird durch eine typisierte *Graphtransformationsregel* [Roz97] formalisiert, wobei das Strukturmodell als *Typgraph* und Instanzen davon durch *typisierte Graphen* formalisiert werden. Eine Graphtransformationsregel besteht aus zwei Graphen, die linke und rechte Regelseite genannt werden. Die linke Regelseite beschreibt einen Teilgraphen, der in einem *Wirtsgraphen* gesucht wird. Die rechte Regelseite beschreibt wie dieser Teilgraph nach der Transformation beschaffen sein soll. Knoten und Kanten, die sich in beiden Regelseiten befinden, bleiben unverändert erhalten. Elemente, die in der linken Regelseite enthalten sind, nicht aber in der rechten, werden im Wirtsgraphen gelöscht. Analog werden Elemente der rechten Regelseite, die nicht in der linken Regelseite enthalten sind, erzeugt.

Die nicht weiter gekennzeichneten Elemente eines Story Pattern entsprechen zusammen mit den zu löschenden Elementen der linken Regelseite einer Graphtransformationsregel. Sie werden auch die linke Seite eines Story Pattern genannt. Die nicht gekennzeichneten Elemente zusammen mit den zu erzeugenden Elementen bilden die rechte Seite der Graphtransformationsregel sowie

des Story Pattern. Die linke Seite wird auch *Anwendungs-* oder *Vorbedingung* genannt. Die rechte Seite wird auch *Nachbedingung* genannt.

Ein Story Pattern wird auf eine Objektstruktur beziehungsweise einen Wirtsgraphen angewendet, indem seine linke Seite auf eine isomorphe Struktur im Wirtsgraphen, eine so genannte *Anwendungsstelle*, abgebildet wird. Ein Story Pattern enthält nicht tatsächliche Objekte und Links sondern vielmehr *Objekt-* und *Linkvariablen*. Bei der Anwendung werden alle Objekt- und Linkvariablen der linken Seite des Story Pattern an typkonforme Objekte und Links im Wirtsgraphen *gebunden*. Dabei dürfen keine zwei Elemente der linken Seite auf dasselbe Element im Wirtsgraphen abgebildet werden.

Das *Ausführen* eines Story Pattern bedeutet, dass eine Anwendungsstelle für seine linke Seite im Wirtsgraphen gesucht wird. Gibt es eine Anwendungsstelle, werden die spezifizierten Modifikationen vorgenommen. Gibt es mehrere Anwendungsstellen, wird nur eine davon nicht-deterministisch ausgewählt und bearbeitet. Diese *erfolgreiche* Ausführung wird *Anwendung* des Story Pattern genannt. Die Ausführung ist *nicht* erfolgreich, wenn es keine Anwendungsstelle gibt.

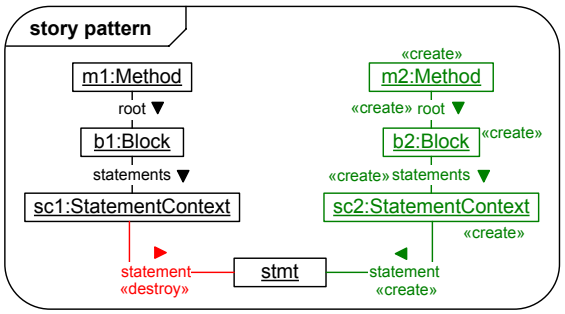

Abbildung 2.10: Beispiel für ein Story Pattern

Abbildung 2.10 zeigt ein Beispiel für ein Story Pattern, welches das Strukturmodell des abstrakten Syntaxgraphen aus Abbildung 2.4 als Strukturmodell verwendet. Das Story Pattern beschreibt, wie eine Anweisung (stmt) aus ihrem aktuellen Kontext in der Methode m1:Method in eine neu erzeugte Methode m2:Method mit neu erzeugtem Rumpf b2:Block in den ebenfalls erzeugten Kontext sc2:StatementContext verschoben wird. Dies geschieht indem der Link statement zum alten Kontext sc1:StatementContext gelöscht wird und ein neuer Link statement zwischen stmt und dem neuen Kontext sc2:StatementContext erzeugt wird.

Um die Suche nach einer Anwendungsstelle für ein Story Pattern in einem Wirtsgraphen zu vereinfachen, muss mindestens eine Objektvariable der linken Seite bereits an ein Objekt im Wirtsgraphen gebunden sein. Die sich dadurch ergebende Einschränkung der Ausdrucksmächtigkeit von Story Pattern hat sich in der Praxis als unproblematisch gezeigt. Im Beispiel ist die Objektvariable stmt gebunden. Gebundene Objektvariablen sind in der Darstellung daran zu erkennen, dass ihre Typbezeichnung ausgeblendet ist.

Story Pattern verfügen über eine Reihe weiterer Ausdrucksmöglichkeiten, die im Beispiel nicht gezeigt werden. So können Objekt- und Linkvariablen als optional gekennzeichnet werden. Optionale Elemente werden bei der Bestimmung einer Anwendungsstelle gebunden, wenn sie vorhanden sind, sie müssen aber nicht vorhanden sein, um eine Anwendungsstelle zu bestimmen. Sind für optionale Elemente Modifikationen spezifiziert, so werden diese vorgenommen, falls die Elemente gebunden wurden. Des Weiteren erlauben sie die Spezifikation von negativen Elementen: solche Elemente dürfen nicht gebunden werden können, damit eine gültige Anwendungsstelle vorliegt.

Klassen verfügen über Attribute, die in ihren Instanzen mit Werten belegt sind. In Objektvariablen in Story Pattern können Bedingungen für die Belegung von Attributen spezifiziert werden. Des Weiteren können auch Zuweisungen angegeben werden, die die Belegung eines Attributs verändern. Darüber hinaus können Einschränkungen (engl. *Constraints*) spezifiziert werden, die nach der Bestimmung einer Anwendungsstelle ausgewertet werden und über die Durchführung der Modifikationen entscheiden. Schließlich ermöglichen einzelne Anweisungen (engl. *Collaboration Messages*) zum Beispiel den Aufruf von Methoden auf gebundenen Objekten. Details zu den aufgeführten Sprachelementen sind [Zün02] zu entnehmen.

2.4.2 Story Diagramme

Zur Beschreibung kompletter Methoden eines Softwaresystems werden häufig mehrere Story Pattern benötigt, deren Anwendung mit Hilfe von Kontrollfluss gesteuert wird. Sie werden daher in Story Diagrammen zu speziellen UML-Aktivitätsdiagrammen kombiniert, deren Aktivitäten durch Story Pattern beschrieben sind. Abbildung 2.11 zeigt ein Beispiel für ein Story Diagramm, das die Verkapselung des Lesezugriffs auf ein Attribut einer Klasse und damit einen Teil des *Encapsulate-Field*-Refactorings beschreibt.

Da ein Story Diagramm eine in einem Klassendiagramm deklarierte Methode beschreibt, verfügt es über eine Signatur bestehend aus beliebig vielen Parametern und einem Rückgabetyp. Im Beispiel wird das zu verkapselnde

Attribut field:Field als Parameter deklariert. Der Rückgabetyp ist void.

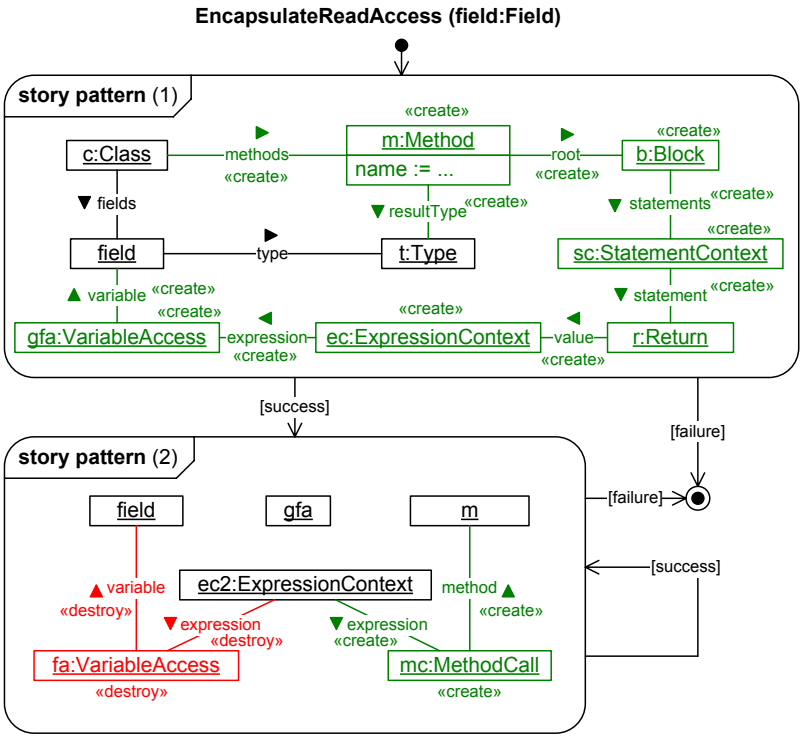

Abbildung 2.11: Beispiel für ein Story Diagramm

Die erste Aktivität wird beschrieben durch ein Story Pattern, das eine lesende Zugriffsmethode für das übergebene Attribut erzeugt. Die Objektvariable field ist an den Parameter field:Field gebunden. Dieses Objekt ist bereits im Wirtsgraphen bestimmt. Von ihm ausgehend werden die umgebende Klasse c:Class und der Typ des Attributs t:Type gebunden. Die erzeugten Elemente definieren eine Methode in deren Rumpf eine Return-Anweisung den Wert des Attributs zurückgibt.

Die erste Aktivität verfügt über zwei ausgehende Transitionen. Schlägt die Ausführung ihres Story Pattern fehl, wird die mit [failure] beschriftete Transition zur Stopaktivität traversiert und die Ausführung des Diagramms wird beendet. Kann ihr Story Pattern dagegen angewendet werden, wird die mit [success] beschriftete Transition zur zweiten Aktivität traversiert.

Das Story Pattern der zweiten Aktivität beschreibt die Ersetzung eines direkten Lesezugriffs auf das Attribut durch den Aufruf der zuvor erzeugten Zugriffsmethode. Die Objektvariable m ist gebunden und bezieht sich auf das

durch das erste Story Pattern erzeugte Methodenobjekt m:Method. Auf diese Weise können sich die Story Pattern desselben Story Diagramms auf durch vorangehende Story Pattern gebundene oder erzeugte Elemente beziehen. Die Objektvariable gfa ist ebenfalls an das gleichnamige Objekt aus dem ersten Story Pattern gebunden. Damit wird verhindert, dass auch der Lesezugriff in der Zugriffsmethode ersetzt wird.

Die zweite Aktivität verfügt über eine mit [success] beschriftete Selbsttransition. Sie modelliert damit eine *while*-Schleife: solange es noch eine Anwendungsstelle für die linke Seite ihres Story Pattern gibt, wird sie ausgeführt. Mit jeder Ausführung wird ein direkter Lesezugriff gelöscht, so dass sie terminiert, wenn alle Zugriffe ersetzt wurden. Die letzte Ausführung schlägt fehl und die [failure]-Transition zur Stopaktivität wird traversiert.

Die Transitionen eines Story Diagramms erlauben die Spezifikation komplexer Kontrollflüsse. Über die gezeigten [success]- und [failure]-Transitionen hinaus können auch Transitionen mit beliebigen Bedingungen spezifiziert werden. Ebenso können über Schleifen mit *while*-Semantik hinaus Schleifen modelliert werden, die auf allen im Wirtsgraphen vorhandenen Anwendungsstellen für ein Story Pattern genau einmal angewendet werden (*For-Each*-Aktivitäten). Details zur dabei verwendeten Semantik sowie weitere Ausdrucksmöglichkeiten sind [Zün02] zu entnehmen.

2.4.3 Verifikation von Story Pattern

Story Diagramme und ihre Story Pattern werden in zahlreichen Anwendungsbereichen eingesetzt. Im Rahmen des Sonderforschungsbereichs (SFB) 614 - „Selbstoptimierende Systeme des Maschinenbaus“[3] werden unter Anderem Techniken zur Modellierung komplexer selbstoptimierender mechatronischer Systeme erforscht. Dabei ist die Modellierungssprache Mechatronic UML [GHH+08] entstanden. Mechatronische Systeme sind in der Regel sicherheitskritisch, da sie häufig massive Auswirkungen auf Leib und Leben ihrer Nutzer haben können. Der Nachweis, dass die Modelle eines solchen Systems definierbaren Sicherheitseigenschaften genügen, ist daher von besonderer Bedeutung.

Die Mechatronic UML fasst Systemzustände als Graphen auf und beschreibt Übergänge von einem Systemzustand in einen anderen mit Hilfe von Graphtransformationsregeln in Form von Story Pattern. Sicherheitseigenschaften, wie zum Beispiel gefährliche Systemzustände, die nie erreicht werden dürfen, lassen sich ebenfalls in Form von Graphen beziehungsweise *verbote-*

[3] http://www.sfb614.de

nen Graphmustern unter Verwendung von Story Pattern beschreiben.

Ein Graphmuster beschreibt einen charakteristischen Ausschnitt eines unter Umständen umfangreicheren Graphen. Ein Graph erfüllt ein Graphmuster, wenn er eine zum Graphmuster isomorphe Struktur enthält. Dabei stellt sich die Frage, ob ein definierter gefährlicher Systemzustand – beschrieben durch ein verbotenes Graphmuster – durch eines der Story Pattern aus einem korrekten Systemzustand heraus erzeugt werden kann.

Der Zustandsraum eines mechatronischen Systems kann unendlich sein und zudem sind nicht alle möglichen Startzustände bekannt. Eine Erreichbarkeitsanalyse ist damit nicht anwendbar. Daher wurde in [Sch06] ein automatisches Verfahren entwickelt, das nachweisen kann, ob die Eigenschaft, dass ein verbotenes Graphmuster nicht erfüllt ist, eine induktive Invariante eines Systems ist. Das Verfahren kann nachweisen, dass die Anwendung eines einzelnen Story Pattern auf einen korrekten Graphen wieder zu einem korrekten Graphen führt, ohne dass dazu alle Systemzustandsgraphen, sondern lediglich eine endliche Teilmenge repräsentativer Teilgraphen davon betrachtet werden müssen.

Übertragen auf die Transformation von Programmen entspricht der abstrakte Syntaxgraph dem Zustand. Der Zustandsraum ist damit ebenfalls unendlich und der exakte Startzustand für eine Transformation ist zum Zeitpunkt ihrer Spezifikation ebenfalls nicht bekannt, da Transformationen typischerweise für die Restrukturierung beliebiger Programme spezifiziert werden. Zudem lassen sich einige Eigenschaften, die die Verhaltenserhaltung betreffen, ebenfalls durch Graphmuster beschreiben, wie später noch beschrieben wird. Daher wird das Verfahren in der vorliegenden Arbeit aufgegriffen und erweitert. Im Folgenden wird der Ansatz deshalb genauer vorgestellt.

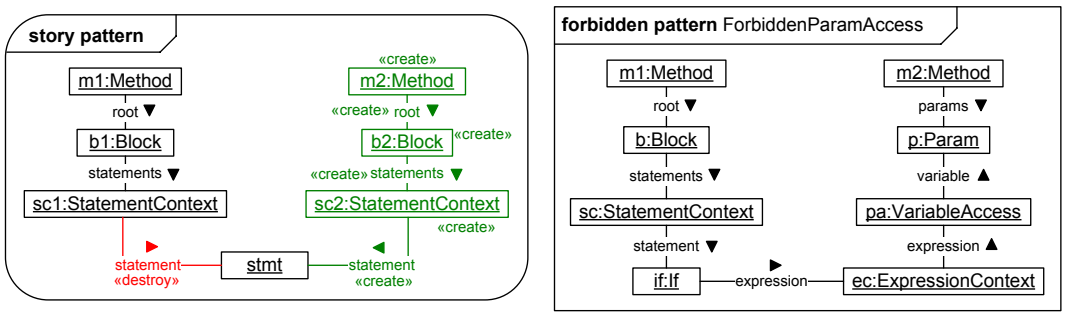

Abbildung 2.12: Ein Story Pattern und ein verbotenes Graphmuster

Dazu werden das in Abbildung 2.12 gezeigte Beispiel für ein Story Pattern, das bereits in Kapitel 2.4.1 vorgestellt wurde, und das verbotene Graphmuster

ForbiddenParamAccess verwendet. Das verbotene Graphmuster beschreibt eine Situation, die es in einem abstrakten Syntaxgraphen nicht geben darf: die Bedingung einer If-Anweisung greift auf einen Parameter einer Methode zu, in der sie sich selbst nicht befindet. Solche Situationen können entstehen, wenn Anweisungen in einem abstrakten Syntaxgraphen verschoben werden, wie es das gezeigte Story Pattern tut.

Das Ziel des Verifikationsverfahrens ist es zu überprüfen, ob eine verbotene Situation durch ein Story Pattern prinzipiell erzeugt werden kann, wenn es auf einen korrekten Graphen angewendet wird. Um dies herauszufinden, wird unterstellt, dass die Anwendung eines Story Pattern eine Instanz eines verbotenen Musters erzeugt hat. Danach wird überprüft, ob der Graph vor dieser Anwendung korrekt gewesen sein kann.

Dazu werden Beispielgraphen konstruiert, in denen eine Instanz des verbotenen Musters so vorliegt, dass mindestens ein Element der verbotenen Situation durch Anwendung des Story Pattern entstanden sein kann. Eine solche Beispielsituation wird *Ergebnisgraphmuster* genannt. Auf ein Ergebnisgraphmuster wird das Story Pattern rückwärts angewendet, um ein *Startgraphmuster* zu erhalten. Bei Vorwärtsanwendung des Story Pattern auf dieses Startgraphmuster würde das Ergebnisgraphmuster mit der verbotenen Situation entstehen. Enthält das Startgraphmuster keine Anwendungsstelle mehr für ein verbotenes Muster, so könnte das Story Pattern einen korrekten Graphen, der das Startgraphmuster erfüllt, in einen inkorrekten Graphen transformieren, der das Ergebnisgraphmuster erfüllt.

Damit die Rückwärtsanwendung von Story Pattern eindeutig ist, werden sie in Bezug auf das Löschen von Objekten eingeschränkt. Durch das Löschen von Objekten können lose Kanten (engl. *dangling edges*) entstehen, wenn die Objekte im Wirtsgraphen mit weiteren Objekten verbunden sind und diese Verbindungen nicht auch explizit durch das Story Pattern gelöscht werden. Story Pattern verfolgen in ihrer ursprünglichen Definition aus [Zün02] den *Single-Pushout-Approach* (SPO) der Graphtransformation [Roz97], der der Löschung von Elementen den Vorrang vor ihrer Erhaltung gibt. Entstehen durch das Löschen von Knoten lose Kanten, so werden sie ebenfalls gelöscht. Bei dieser Semantik ist die Vorwärtsanwendung eines Story Pattern, das Objekte löscht, nicht zwingend vollständig spezifiziert, da dabei mehr Kanten als spezifiziert gelöscht werden können. Daraus folgt, das auch die Rückwärtsanwendung nicht eindeutig ist. In [Sch06] wird daher ein eingeschränkter SPO-Ansatz definiert, in dem Story Pattern vor der Löschung mit Hilfe negativer Elemente sicherstellen, dass keine losen Kanten entstehen können. Diese zusätzlich benötigten negativen Elemente können auf Basis des zugrunde liegenden Strukturmodells

automatisch ermittelt werden.

Zur Bildung von Ergebnisgraphmustern wird die rechte Seite eines Story Pattern mit dem verbotenen Graphmuster zu einem neuen Graphen verklebt. Beim Verkleben zweier Graphen werden diese zu einem neuen Graphen vereinigt, wobei ein Teilgraph des einen Graphen auf einen isomorphen Teilgraph des anderen Graphen abgebildet wird, so dass dieser im neuen Graph nur einmal enthalten ist und beide Graphen anhand dieses Teilgraphen miteinander verbunden sind. Die Spezialfälle, dass ein Graph vollständig oder gar nicht auf den anderen Graphen abgebildet werden kann, können ebenfalls auftreten.

Abbildung 2.13 zeigt im oberen Teil ein Beispiel für ein Ergebnisgraphmuster. Die Anwendungsstelle für das verbotene Graphmuster im Ergebnisgraphmuster ist orange gefärbt. Bei den Elementen der Anwendungsstelle, die mit ≪mapped≫ markiert sind, handelt es sich um Elemente der rechten Seite des Story Pattern, auf die beim Verkleben Elemente des Graphmusters abgebildet wurden. Elemente der Anwendungsstelle, die mit ≪added≫ markiert sind, repräsentieren Elemente des Graphmusters, die nicht abgebildet wurden. Solche Elemente müssen bereits vor Anwendung des Story Pattern im Wirtsgraphen existiert haben.

Bei Rückwärtsanwendung des Story Pattern auf das Ergebnisgraphmuster entsteht das im unteren Teil der Abbildung gezeigte Startgraphmuster. Das Startgraphmuster enthält keine Anwendungsstelle mehr für das verbotene Graphmuster. Somit wurde eine Beispielsituation konstruiert, in der das Story Pattern eine verbotene Struktur erzeugen kann. Präzise formuliert kann das Story Pattern bei Anwendung auf einen Wirtsgraphen, der das Startgraphmuster erfüllt, eine Anwendungsstelle für das verbotene Graphmuster im Wirtsgraphen herstellen.

Story Pattern können negative Elemente enthalten. Gibt es eine Anwendungsstelle für die linke Seite eines Story Pattern ohne negative Elemente, die so erweitert werden kann, dass eines der negativen Elemente gebunden wird, ist das Story Pattern nicht anwendbar. Übertragen auf ein Startgraphmuster bedeutet dies, dass in einem Wirtsgraph, der über die Anwendungsstelle für das Startgraphmuster hinaus eines der negativen Elemente enthält, durch das Story Pattern keine verbotene Situation entstehen kann.

Um dies auszudrücken, werden Startgraphmuster um negative Anwendungsbedingungen erweitert. Ein Wirtsgraph erfüllt ein solches Graphmuster dann, wenn er eine Anwendungsstelle für die Anwendungsbedingung des Musters enthält, aber keine Anwendungsstelle für eine seiner negativen Anwendungsbedingungen. Darüber hinaus werden negative Anwendungsbedingungen für verbotene Graphmuster ermöglicht. Damit kann formuliert werden, dass eine

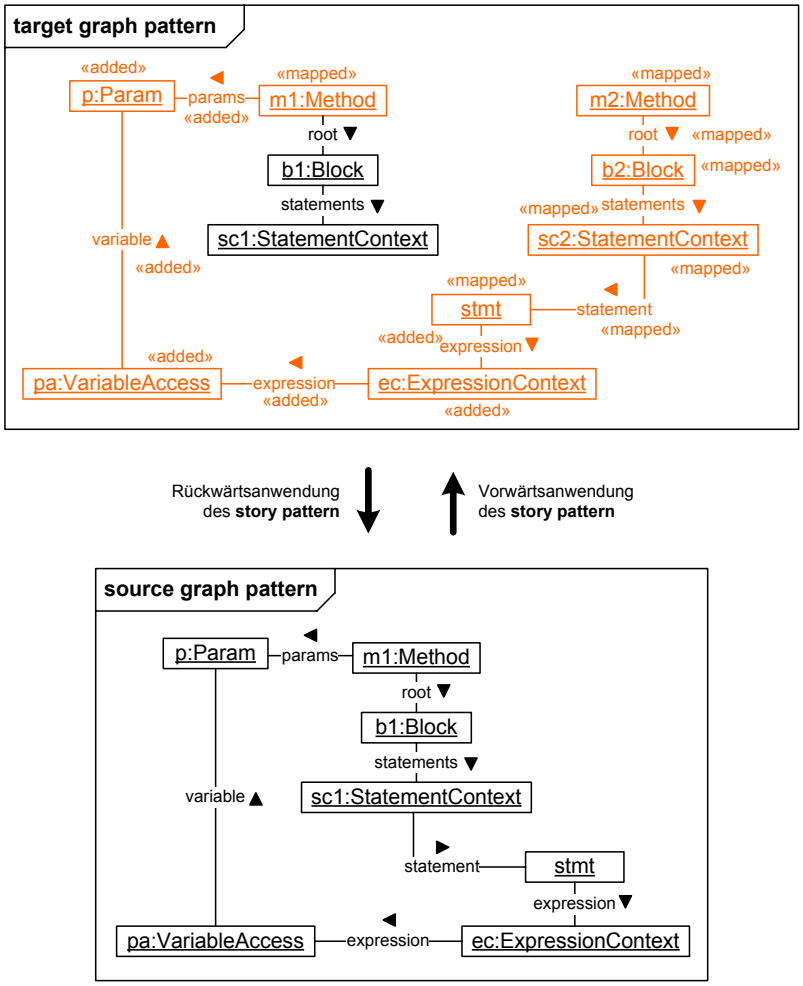

Abbildung 2.13: Ein Ergebnisgraphmuster mit zugehörigem Startgraphmuster

Struktur nur dann verboten ist, wenn der Wirtsgraph eine Anwendungsstelle für das verbotene Graphmuster enthält, aber keine Anwendungsstelle für eine der negativen Anwendungsbedingungen des Graphmusters. Sämtliche negativen Anwendungsbedingungen werden durch das Verifikationsverfahren korrekt berücksichtigt.

Kapitel 3

Formale Spezifikation von Programmtransformationen

In diesem Kapitel wird mit *Transformationsdiagrammen* eine Sprache für die formale Spezifikation von Programmtransformationen entwickelt. Dazu werden zunächst Anforderungen an eine solche Sprache formuliert und danach Transformationsdiagramme informell eingeführt. Im Anschluss daran werden die Syntax und Semantik von Transformationsdiagrammen formal definiert. Den Abschluss des Kapitels bildet ein Ansatz, der die Spezifikation von Transformationsdiagrammen anhand konkreter Quelltextbeispiele ermöglicht.

3.1 Anforderungen

Zur Verbesserung von Schwachstellen, wie zum Beispiel der Überführung einer Implementierung eines Conditional Dispatchers (siehe Abbildung 2.1) in eine Struktur auf Basis des Command-Entwurfsmusters (Abbildung 2.2), werden ausführbare Programmtransformationen benötigt.

Zur Erkennung von Schwachstellen wird ein graphbasierter Ansatz verfolgt. Das zu analysierende Programm wird als abstrakter Syntaxgraph repräsentiert. Musterspezifikationen werden auf Basis von Graphtransformationsregeln formalisiert. Die dabei entstehenden Regeln transformieren den Graphen nur insoweit, als dass sie Annotationsknoten erzeugen und mit Elementen des abstrakten Syntaxgraphen verbinden, um Musterinstanzen zu markieren – mehr ist für die Analyse nicht notwendig.

Graphtransformationsregeln erlauben durch das Löschen und Hinzufügen von Knoten und Kanten weitaus umfangreichere Modifikationen, die auch dazu genutzt werden können, den abstrakten Syntaxgraphen eines Programms zu transformieren, um dem Programm eine neue Struktur zu geben. Darüber

hinaus verfügen Graphtransformationen über eine formal definierte und leicht zu erlernende Semantik mit ebenso leicht verständlicher Syntax. Daher ist es naheliegend, die Verbesserung von Schwachstellen durch Transformation des abstrakten Syntaxgraphen vorzunehmen und diese mit Hilfe von Graphtransformationsregeln zu spezifizieren.

Die Spezifikation von Programmtransformationen erfordert umfangreiche Ausdrucksmöglichkeiten: es müssen umfangreiche, auch negative Vorbedingungen spezifiziert werden können, auf unterschiedliche Ausprägungen des abstrakten Syntaxgraphen muss unterschiedlich reagiert werden und verschiedene Anteile einer Transformation müssen mehrfach ausgeführt werden. Dazu werden Kontrollflusskonstrukte (Sequenz, Alternative, Schleife) benötigt.

Bei der Transformation eines Conditional Dispatcher muss zum Beispiel sichergestellt werden, dass alle beteiligten If-Anweisungen die übergebene Anfrage auf ein Literal prüfen (siehe Abschnitt 2.1). Ist dem nicht so, ist eine Transformation in eine Command-Struktur nicht möglich. Stattdessen kann eine Transformation in eine Chain of Responsibility vorgenommen werden. In beiden Fällen müssen Transformationsschritte einmalig ausgeführt werden, wie zum Beispiel das Erzeugen einer abstrakten Basisklasse für die Kommandos. Die Schritte, um eine If-Anweisung in eine Kommandoklasse zu transformieren, müssen dagegen mehrfach, nämlich für alle beteiligten If-Anweisungen durchgeführt werden.

In Strukturmustern wird mit Hilfe von Pfaden von konkreten Strukturen des abstrakten Syntaxgraphen abstrahiert. Dadurch werden beliebige Implementierungsvarianten eines Musters erkannt, solange sie den für das Muster relevanten Eigenschaften genügen. Diese Ausdrucksmöglichkeit wird auch in Transformationen benötigt, um bestimmte Elemente in unterschiedlich strukturierten abstrakten Syntaxgraphen binden zu können, ohne die Strukturen explizit angeben zu müssen.

Umfangreiche Transformationen wie die Restrukturierung eines Conditional Dispatcher verwenden häufig kleinere Transformationen wie das Verkapseln von Attributen oder Extrahieren von Methoden, die auch für sich alleine sinnvoll eingesetzt werden können. Daher ist es sinnvoll, sie als eigenständige Transformationen zu spezifizieren und in umfangreicheren Transformationen durch Aufruf wiederzuverwenden.

Als Ausgangsbasis für Transformationen sollen erkannte Schwachstellen dienen. Schwachstellen werden wie in Abbildung 2.7 skizziert im abstrakten Syntaxgraphen durch Strukturmusterannotationen markiert. Damit eine Transformation direkt auf eine durch eine Annotation markierte Schwachstelle angewendet werden kann, muss sich ihre Spezifikation auf Strukturmuster bezie-

hungsweise auf Annotationen im abstrakten Syntaxgraphen beziehen können.

Mit dem in Kapitel 2.4 vorgestellten Ansatz des *Story Driven Modeling* steht eine ausdrucksmächtige Graphtransformationssprache zur Verfügung. Story Diagramme verfügen grundsätzlich über die zuvor geforderten Ausdrucksmöglichkeiten. Daher dienen sie im Rahmen dieser Arbeit als Basis für die Spezifikation von Programmtransformationen.

Darüber hinaus gibt es weitere Anforderungen an die Spezifikation von Programmtransformationen. So soll bei der Transformation eines Programms zur Verbesserung einer Schwachstelle in der Regel die Struktur verbessert werden, ohne das von außen beobachtbare Verhalten zu verändern. Eine formale Spezifikation von Programmtransformationen auf Basis von Graphtransformationen erlaubt den Nachweis, dass Transformationen bestimmte Kriterien erfüllen. Dies können Kriterien sein, die mindestens erfüllt werden müssen, damit das von außen beobachtbare Verhalten der transformierten Software unverändert bleibt.

Mens et al. schlagen in [MDJ02, MEJD03] zum Beispiel die folgenden drei Kriterien vor:

- *access preservation:* Es werden nach einer Transformation mindestens dieselben Variablen gelesen wie vorher. Dies kann auch indirekt, zum Beispiel durch neu hinzugefügte Zugriffsmethoden und deren Aufruf, erfolgen.

- *update preservation:* Es werden nach einer Transformation mindestens dieselben Variablen geschrieben wie vorher. Dies kann ebenfalls indirekt durch neu hinzugefügte Zugriffsmethoden und deren Aufruf erfolgen.

- *call preservation:* Es werden nach einer Transformation mindestens dieselben Methoden aufgerufen wie vorher.

Ebenso gibt es Kriterien, die Situationen beschreiben, die nie eintreten dürfen (oder sollen). Dies umfasst zum Beispiel unerlaubte Zugriffe auf nicht sichtbare Variablen oder Aufrufe nicht sichtbarer Methoden. So darf es zum Beispiel innerhalb einer Methodenimplementierung keinen Bezeichner geben, der sich auf einen Parameter einer anderen Methode oder ein privates Attribut einer anderen Klasse bezieht. Eine solche Situation kann durch eine Transformation erzeugt werden, wenn Anweisungen von einer Methode in eine andere Methode verschoben werden, zum Beispiel im Rahmen der Restrukturierung eines Conditional Dispatchers.

Solche Kriterien können als strukturelle Eigenschaften eines abstrakten Syntaxgraphen mit Hilfe von Graphmustern beschrieben werden. In Abschnitt 2.4.3 wird ein Ansatz vorgestellt, der Kriterien in Form verbotener Graphmuster spezifiziert und automatisch nachweisen kann, dass ein Story Pattern bei Anwendung auf einen beliebigen, korrekten Graphen keinen inkorrekten Graphen erzeugen kann, in dem eine Instanz eines verbotenen Graphmusters existiert.

Story Diagramme bestehen aus mehreren Story Pattern, deren Ausführung durch Kontrollfluss gesteuert wird. Das bestehende Verifikationsverfahren kann aber nur einzelne Story Pattern auf Einhaltung von Kriterien verifizieren. Das würde bedeuten, dass Kriterien zu jeder Zeit beziehungsweise nach Ausführung jedes einzelnen Story Pattern eines Story Diagramms gelten müssen. Dies ist im Kontext von Programmtransformationen zu restriktiv. So kann es zum Beispiel beim Verschieben von Anweisungen in neue Methoden temporär zu verbotenen Variablenzugriffen kommen, die dann durch weitere Story Pattern der Transformation korrigiert werden. Kriterien müssen während einer Transformation verletzt werden können – sie müssen lediglich vor und dann wieder nach einer vollständigen Transformation – der Ausführung eines gesamten Story Diagramms beziehungsweise einer Sequenz von Story-Pattern-Anwendungen – gelten.

Zur Formulierung von Programmtransformationen wird nicht zwingend die volle Ausdrucksmächtigkeit von Story Diagrammen benötigt. Bei Verwendung von Story Diagrammen würde eine automatische Verifikation der Spezifikationen unnötig erschwert beziehungsweise wäre sie gar nicht entscheidbar. Daher wird im Rahmen dieser Arbeit eine auf Story Diagrammen aufbauende Sprache für Transformationsspezifikationen entworfen, die ausdrucksstark genug ist für Programmtransformationen und deren Worte (einfacher) auf Einhaltung von Kriterien in Form verbotener (und zu erhaltender) Graphmuster verifiziert werden können.

Die Sprache wird im Folgenden vorgestellt. Die Verifikation von Transformationsdiagrammen wird in Kapitel 4 beschrieben.

3.2 Transformationsdiagramme

Transformationsdiagramme sind analog zu den in Abschnitt 2.4.2 vorgestellten Story Diagrammen spezielle UML-Aktivitätsdiagramme. Jedes Transformationsdiagramm verfügt über genau eine Startaktivität und beliebig viele Stopaktivitäten sowie spezielle Story Pattern, die so genannten *Transformation*

Pattern, die über gerichtete Transitionen miteinander verbunden sind.

Die Transformation Pattern werden in diesem Ansatz durch das in Abschnitt 2.3.1 eingeführte Strukturmodell des abstrakten Syntaxgraphen typisiert. Damit können Transformationsdiagramme auf einen abstrakten Syntaxgraphen, der ein Programm repräsentiert, angewendet werden und diesen dabei durch Hinzufügen und Löschen von Objekten und Links transformieren. Dabei ist es für die im Weiteren präsentierten Konzepte grundsätzlich unerheblich, dass abstrakte Syntaxgraphen transformiert werden. Das zur Typisierung der Transformationen verwendete Strukturmodell ist beliebig austauschbar.

Transformationsdiagramme sind eine eingeschränkte Variante von Story Diagrammen, die leichter zu verifizieren sind:

1. Transformation Pattern enthalten keine optionalen Elemente, keine negativen Elemente[1], keine Constraints und keine Collaboration Messages. Außerdem verwenden die Transitionen zwischen den Transformation Pattern keine booleschen Bedingungen.

2. Die Anwendbarkeit von Transformation Pattern wird weiter eingeschränkt, als dies bereits durch den Einsatz von Kontrollfluss wie in Story Diagrammen oder auch in PROGRESS [Sch91, Zün96] geschieht. Transformation Pattern dürfen Elemente, die bei Ausführung von Transformation Pattern desselben Diagramms zuvor erzeugt wurden, nur binden, wenn dies explizit spezifiziert ist. Auf diese Weise müssen Schleifen, die keine anderen Transformationen aufrufen, terminieren. Unter anderem dadurch kann auch die Terminierung des Verifikationsverfahrens garantiert werden. Zudem müssen weniger Kombinationsmöglichkeiten der Hintereinanderausführung von Transformation Pattern untersucht werden, so dass sich der Verifikationsaufwand verringert.

3. Pfade in Transformation Pattern, die mittelbare Verbindungen zwischen Objekten eines Graphen über beliebig viele Objekte und Links hinweg beschreiben, werden in Ihrer Ausdrucksmächtigkeit reduziert (siehe Abschnitt 3.2.1). Zum einen ist es dadurch einfacher, bei der Verifikation ihre möglichen Ausprägungen in einem Wirtsgraphen zu analysieren und zum anderen sind sie effizient in beide Richtungen traversierbar.

4. Die Spezifikation von Schleifen wird so eingeschränkt, dass ihre Rümpfe nur weitere Schleifen aber keine Sequenzen oder Alternativen enthalten

[1] Transformation Pattern können nur als Ganzes negiert werden, wie im Weiteren noch beschrieben wird.

können (siehe Abschnitt 3.2.2). Dadurch müssen durch das Verifikationsverfahren weniger Kombinationen der Hintereinanderausführung verschiedener Iterationen einer Schleife untersucht werden, so dass sie einfacher zu handhaben sind und der Verifikationsaufwand reduziert wird.

Abbildung 3.1 zeigt ein Beispiel für ein Transformationsdiagramm, das eine sehr einfache Variante eines *Extract-Method*-Refactorings [Fow99] (siehe Abschnitt 2.2) modelliert. Das Refactoring wird mehrfach im Rahmen der Verbesserung von Conditional-Dispatcher-Instanzen eingesetzt.

Die gezeigte Variante ist zu Illustrationszwecken stark vereinfacht und nur sehr eingeschränkt nutzbar, da sie viele Aspekte unberücksichtigt lässt. Dies kann sich je nach Beschaffenheit des Programms, auf das die Transformation angewendet wird, in Änderungen des Verhaltens äußern. Einige dieser Aspekte werden in den folgenden Abschnitten aufgegriffen und ergänzt, um daran weitere Sprachkonstrukte zu motivieren und zu erläutern.

Ein Transformationsdiagramm verfügt über eine Signatur bestehend aus einem eindeutigen Namen (im Beispiel SimpleExtract), beliebig vielen Parametern und beliebig vielen Rückgabewerten. Das Transformationsdiagramm in Abbildung 3.1 verfügt über zwei Parameter: die zu extrahierende Anweisung stmt:Statement und einen Namen name:String für die neue Methode. Da es sich bei stmt:Statement zum Beispiel um einen Anweisungsblock oder eine If-Anweisung handeln kann, die wiederum weitere Anweisungen enthalten, können auf diese Weise mehrere Anweisungen extrahiert werden. Des Weiteren wird mit target:Method ein Rückgabewert deklariert, der später die neue Methode zurückliefern soll. Damit ein Transformationsdiagramm ausgeführt werden kann, müssen alle Parameter an konkrete Argumente gebunden werden.

Die Startaktivität eines Transformationsdiagramms ist immer mit genau einem Transformation Pattern verbunden, mit dem die Ausführung eines Transformationsdiagramms begonnen wird.

Bei der Ausführung eines Transformation Pattern wird versucht im Wirtsgraphen – hier dem abstrakten Syntaxgraphen – eine Anwendungsstelle für seine linke Seite zu finden. Gelingt dies, werden die spezifizierten Modifikationen im Wirtsgraphen vorgenommen.

Jedes Transformation Pattern verfügt über zwei ausgehende Transitionen. Eine der Transitionen ist mit [success], die andere mit [failure] beschriftet. Analog zu Story Diagrammen wird die mit [success] beschriftete Transition traversiert, wenn das Transformation Pattern erfolgreich ausgeführt, also angewendet, wurde. War die Ausführung nicht erfolgreich, konnte also keine An-

wendungsstelle gefunden werden, wird die mit [failure] beschriftete Transition traversiert.

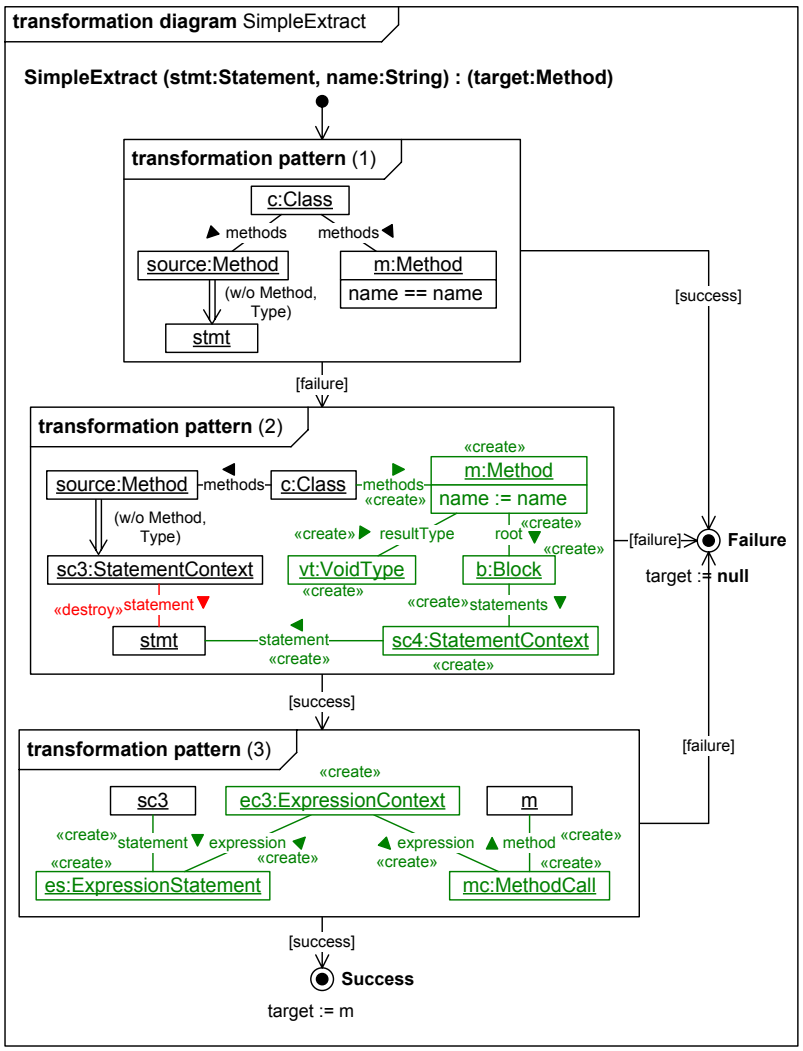

Abbildung 3.1: Einfache Variante eines *Extract-Method*-Refactorings

Das Ziel einer Transition kann ein weiteres Transformation Pattern oder eine Stopaktivität sein. Mit einem Transformation Pattern wird wie zuvor beschrieben verfahren. Auf diese Weise werden eine Reihe von Transformation Pattern hintereinander ausgeführt. In Kombination mit den Transitionen können so Sequenzen, alternative Ausführungen sowie beliebige (auch negative) Vorbedingungen für die Ausführung eines Transformationsdiagramms modelliert

werden.

Im Beispiel überprüft das erste Transformation Pattern (1), ob die Klasse, in der die neue Methode zu erstellen ist, bereits eine Methode mit dem Namen name enthält, der für die neue Methode verwendet werden soll. Ist dies der Fall, gibt es also eine Anwendungsstelle für das Pattern, wird die [success]-Transition zu einer Stopaktivität traversiert. Gibt es keine solche Methode, schlägt die Ausführung fehl und die [failure]-Transition wird zum nächsten Transformation Pattern traversiert. Das Pattern formuliert so positiv eine negative Vorbedingung für die Anwendung des Transformationsdiagramms, indem bei Traversieren der [failure]-Transition mit der Transformation fortgefahren wird.

Analog zu Story Pattern muss zur Ausführung eines Transformation Pattern immer mindestens eine definierte Objektvariable der linken Seite bereits an ein Objekt des Wirtsgraphen gebunden sein. Bei dem ersten Transformation Pattern eines Transformationsdiagramms muss es sich dabei um ein Parameter-Objekt handeln. Allgemein können sich Transformation Pattern durch Objektvariablen explizit auf Objekte beziehen, die durch vorhergehende Transformation Pattern gebunden oder erzeugt wurden. Solche Objektvariablen werden *gebundene Objektvariablen* genannt. Sie tragen denselben Bezeichner wie die Objektvariable, auf die sie sich beziehen. Bei der Bestimmung einer Anwendungsstelle wird eine gebundene Objektvariable an dasselbe Objekt im Wirtsgraphen gebunden, an das die Objektvariable gebunden wurde, auf die sich die gebundene Objektvariable bezieht.

Das erste Transformation Pattern im Beispiel beinhaltet mit stmt eine gebundene Objektvariable, die sich auf den gleichnamigen Parameter bezieht. Gebundene Objektvariablen sind daran erkennbar, dass ihre Typbezeichnung ausgeblendet ist. Das zweite Transformation Pattern bezieht sich ebenfalls mit stmt auf die als Parameter übergebene zu extrahierende Anweisung. Das Pattern erzeugt in der Klasse, in der sich die Methode source:Method, welche die Anweisung derzeit enthält, befindet, eine neue Methode mit Rumpf und fügt die zu extrahierende Anweisung dort ein.

Das dritte Transformation Pattern, das nur erreicht wird, wenn das zweite Pattern angewendet werden konnte, fügt an der Stelle, an der sich zuvor die Anweisung befunden hat, einen Aufruf der neuen Methode ein. Das Pattern bezieht sich mit sc3 auf die gleichnamige Objektvariable des zweiten Pattern und damit auf das daran gebundene Objekt. Darüber hinaus bezieht es sich mit m auf die durch das zweite Pattern erzeugte Methode.

Objektvariablen verschiedener Transformation Pattern können, zwar nicht in dem hier gezeigten Beispiel aber im Allgemeinen, an dieselben Objekte

im Wirtsgraphen gebunden werden wie Objektvariablen zuvor angewendeter Transformation Pattern, auch ohne dass dies explizit durch gebundene Objektvariablen spezifiziert wird. Bei Transformationsdiagrammen darf dies im Unterschied zu Story Diagrammen nicht bei durch die Anwendung von Transformation Pattern desselben Diagramms erzeugten Objekten und Links geschehen. Das heißt, später angewendete Transformation Pattern dürfen keine durch zuvor angewendete Transformation Pattern desselben Diagramms erzeugte Elemente binden, sofern dies nicht explizit durch gebundene Objektvariablen vorgegeben ist.

Transformation Pattern können damit nur eingeschränkt Anwendungsstellen für spätere Transformation Pattern desselben Diagramms erzeugen, so dass sich wie eingangs beschrieben deutlich weniger verschiedene Wirtsgraphen durch die Hintereinanderausführung von Transformation Pattern desselben Transformationsdiagramms ergeben können als dies bei Story Pattern in Story Diagrammen der Fall ist. Dadurch wird die Verifikation der Transformationsdiagramme erheblich vereinfacht. Gleichzeitig wird die Ausdrucksmächtigkeit nicht zu sehr eingeschränkt, da das explizit spezifizierte erneute Binden von erzeugten Elementen, wie in Transformation Pattern (3) des Beispiels, weiterhin möglich ist. Zudem kann die Einschränkung, falls unbedingt notwendig, auch durch eine Aufteilung auf verschiedene, sich aufrufende Transformationsdiagramme überwunden werden.

Die Ausführung eines Transformationsdiagramms endet bei einer Stopaktivität. Eine Stopaktivität ist analog zu einer Transition entweder mit Success oder mit Failure beschriftet. Wird eine mit Success beschriftete Stopaktivität erreicht, gilt die Ausführung des Transformationsdiagramms als erfolgreich. Die Stopaktivität definiert über *Rückgabebindungen*, welche Objekte an welchen Rückgabewert des Transformationsdiagramms gebunden werden.

Das in Abbildung 3.1 gezeigte Diagramm verfügt über eine Success-Stopaktivität am unteren Bildrand, die bei einer erfolgreichen Anwendung von Transformation Pattern (3) erreicht wird. Der Rückgabewert target wird an die neu erzeugte Methode m gebunden.

Wird eine mit Failure beschriftete Stopaktivität erreicht, so gilt die Ausführung des Transformationsdiagramms als fehlgeschlagen. In einem solchen Fall werden alle bisher am Wirtsgraphen durchgeführten Modifikationen rückgängig gemacht. Auf diese Weise nehmen nur erfolgreich beendete Ausführungen eines Transformationsdiagramms Veränderungen am abstrakten Syntaxgraphen eines Programms vor.

Das gezeigte Beispieldiagramm enthält eine Failure-Stopaktivität zu der alle Transformation Pattern des Diagramms mit einer Transition verzweigen.

Das erste Transformation Pattern ist durch eine [success]-Transition mit der Stopaktivität verbunden. Damit wird die Ausführung beendet, wenn das erste Transformation Pattern angewendet werden konnte, was der Semantik einer negativen Vorbedingung entspricht. Das zweite und dritte Transformation Pattern verzweigen zu der Stopaktivität, wenn ihre Ausführung fehlschlägt. In diesem Fall müssen alle bis zum Fehlschlag vorgenommenen Veränderungen am abstrakten Syntaxgraphen rückgängig gemacht werden.

3.2.1 Pfade

Bei der Transformation von Programmen müssen häufig Anweisungen oder Ausdrücke innerhalb einer Methodenimplementierung gebunden werden, ohne dabei den genauen Aufbau des Methodenrumpfes zu kennen. So muss in Transformation Pattern (1) in Abbildung 3.1 ausgehend von der zu extrahierenden Anweisung stmt die Methode source:Method gebunden werden, in der die Anweisung aktuell enthalten ist. Gleiches gilt für Transformation Pattern (2).

Die Methode ist nicht direkt mit der zu extrahierenden Anweisung verbunden. Dazwischen kann sich eine beliebige Kette von Objekten und Links befinden. Um von solchen Ketten abstrahieren zu können, werden Pfade eingesetzt.

Ein Pfad stellt eine mittelbare Verbindung zwischen einem Startobjekt und einem Zielobjekt dar, so dass das Zielobjekt ausgehend vom Startobjekt durch Traversieren weiterer Objekte und Links erreichbar ist. Ein Pfad repräsentiert damit prinzipiell beliebig lange, aber endliche Ketten von Objekten und Links zwischen den verbundenen Objekten. Eine solche Kette wird *Ausprägung* eines Pfades genannt. Auf diese Weise kann allgemein von konkreten Strukturen in einem abstrakten Syntaxgraphen abstrahiert werden.

Story Pattern erlauben die Spezifikation von Pfaden unter Angabe von Navigationsausdrücken in der *Object Constraint Language* (OCL), einem Teil der UML [Obj], erweitert um Konstrukte zur Berechnung transitiver Hüllen. So können von einem Objekt ausgehend durch beliebig häufiges Traversieren beliebig vieler Assoziationen in definierte Richtungen Mengen weiterer Objekte gebunden werden. Für solche Mengen können die Vereinigung, der Durchschnitt oder die Differenz gebildet und zusätzliche Eigenschaften (Attributbedingungen, Typbedingungen) definiert werden, die Objekte zur Aufnahme in eine Menge erfüllen müssen. Dies entspricht im Wesentlichen den Ausdrucksmöglichkeiten, die bereits in [ELS86] und [Zün96] definiert werden.

Auf diese Weise können komplexe Traversierungen des Wirtsgraphen zur Ermittlung von Objekten definiert werden, was eine Verifikation solcher Spe-

zifikationen sehr erschweren kann. Je nach Beschaffenheit des Strukturmodells können solche Navigationsausdrücke zudem sehr umfangreich werden. Ein weiterer Aspekt ist, dass sie unter Umständen nicht effizient in Rückwärtsrichtung traversiert werden können [Zün96].

Die im Kontext von Programmtransformationen verwendeten Pfade benötigen nicht die volle zuvor beschriebene Ausdrucksmächtigkeit. Zudem sind sie in Bezug auf die Objekte und Links, die sie in bestimmte Richtungen traversieren dürfen, im Wesentlichen identisch. In Transformation Pattern werden daher spezielle Pfade eingesetzt, die im Prinzip alle denselben Navigationsausdruck haben. Welche Links in welche Richtung durch einen Pfad traversiert werden dürfen, wird im Strukturmodell definiert.

Im Strukturmodell wird für Assoziationen angegeben, ob und wenn ja in welche Richtung sie durch einen Pfad traversiert werden dürfen. Dies erfolgt mit Hilfe von Stereotypen, die anzeigen, ob eine Assoziation durch einen Pfad in Leserichtung oder entgegen der Leserichtung traversiert werden darf. Assoziationen ohne Kennzeichnung dürfen gar nicht traversiert werden.

Pfade sind damit zum einen einfacher zu spezifizieren. Zum anderen nutzen sie von den zuvor beschriebenen Ausdrucksmöglichkeiten der Pfade in Story Pattern nur das beliebig häufige Traversieren beliebig vieler Assoziationen in bestimmte Richtungen und die Vereinigung von so bestimmten Mengen von Objekten. Im Folgenden werden sie noch um Typeinschränkungen für Objekte ergänzt, so dass sie insgesamt so vereinfacht sind, dass sie effizient durch das in Kapitel 4 entwickelte Verifikationsverfahren analysiert werden können. Auch ist eine Traversierung in Rückwärtsrichtung effizient möglich.

Abbildung 3.2 zeigt einen Ausschnitt des Strukturmodells des abstrakten Syntaxgraphen. Das Diagramm wurde um Pfadnavigationsinformationen ergänzt. Zur Darstellung werden anstatt von Stereotypen Doppelpfeile verwendet, die parallel zu pfadnavigierbaren Assoziationen verlaufen. Die Spitze des Doppelpfeils zeigt an, in welche Richtung sie traversiert werden dürfen.

Ausgehend von der Klasse Statement selbst ist keine Assoziation durch einen Pfad traversierbar. Allerdings kann es sich aufgrund der Vererbungsbeziehung zwischen Statement und Block bei einem Statementobjekt wie stmt auch um einen Anweisungsblock handeln. Von diesem ausgehend kann die Assoziation statements zu StatementContext durch einen Pfad traversiert werden. Von StatementContext aus ist via statement wiederum eine Anweisung erreichbar, bei der es sich aufgrund von Vererbung auch um eine Return-Anweisung handeln kann. Alternativ kann es ebenso eine If-Anweisung sein, von der ausgehend ein Pfad sowohl via then als auch via else zu StatementContext und damit letztendlich zu weiteren Anweisungen gelangen kann.

Der Teil eines abstrakten Syntaxgraphen, der den kontextfreien Anteil der Syntax eines Methodenrumpfs repräsentiert, ist ein Baum. Durch Typ- oder Methodenbindungen (kontextsensitive Syntax) wird er mit Teilen des übrigen abstrakten Syntaxgraphen verbunden. So ist Abbildung 3.2 zu entnehmen, dass ein Pfad zum Beispiel von einer If-Anweisung über expression, ExpressionContext, expression einen Ausdruck erreichen kann, der mit MethodCall ein Methodenaufruf sein kann. Von dort aus kann ein Pfad via method die gerufene Methode erreichen und über root weiter in deren Rumpf absteigen.

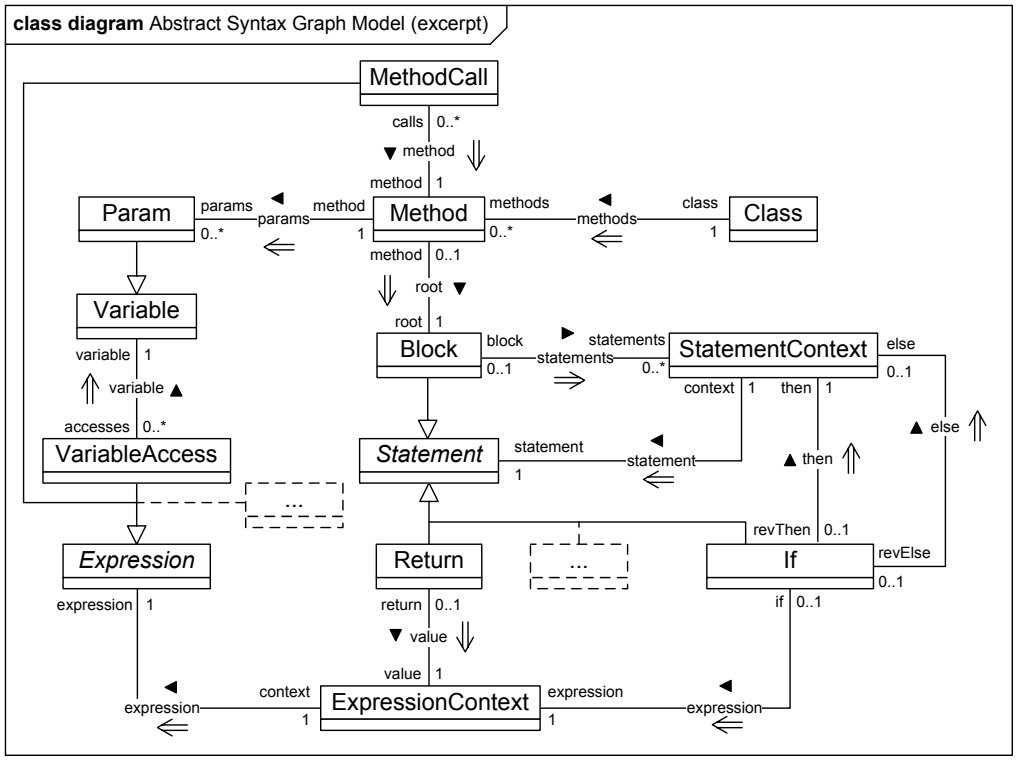

Abbildung 3.2: Ausschnitt aus dem Strukturmodell des abstrakten Syntaxgraphen mit Pfadnavigationsinformationen

In manchen Fällen ist gewünscht, dass ein Pfad innerhalb einer Methode bleibt und sie nicht verlässt. In anderen Fällen ist wiederum explizit erwünscht, dass der Pfad die Methode verlässt und zum Beispiel in einer von der ersten Methode aufgerufenen zweiten Methode weitere Objekte bindet.

Damit ein konkreter Pfad innerhalb einer Methode bleibt, kann für ihn verboten werden, dass er ein Objekt vom Typ Methode oder ein Typobjekt (zum

Beispiel eine Klasse) traversiert. Allgemein kann für einen konkreten Pfad eine Menge von Typen definiert werden, deren Instanzen nicht Bestandteil einer Ausprägung des Pfades sein dürfen.

Transformation Pattern (1) in Abbildung 3.1 setzt einen Pfad ein, dargestellt durch einen Doppelpfeil, der die Methode source:Method mit der in ihr enthaltenen zu extrahierenden Anweisung stmt verbindet. Damit es eine Anwendungsstelle für das Pattern gibt, muss im Wirtsgraphen von source:Method ausgehend durch Traversieren einer beliebig langen Kette von Links und Objekten stmt gebunden werden können.[2] Für den Pfad wird durch Angabe von (w/o Method, Type)[3] verboten, dass er ein Methoden- oder Typobjekt traversiert.

Pfade sind Bestandteil der linken Seite eines Transformation Pattern. Durch die Anwendung des Transformation Pattern können unter Umständen Elemente gelöscht werden, die Bestandteil einer Pfadausprägung sind. Pfade müssen also nach der Anwendung eines Transformation Pattern nicht mehr vorhanden sein, können es aber. Daher sind sie nicht Bestandteil der rechten Seite eines Transformation Pattern. Dies bedeutet aber nicht, dass sie durch die Anwendung explizit gelöscht werden.

Des Weiteren dürfen Pfadausprägungen keine durch Transformation Pattern desselben Transformationsdiagramms erzeugten Elemente beinhalten (siehe Einschränkungen in Abschnitt 3.2).

3.2.2 Iteration

Im Rahmen von Programmtransformationen müssen regelmäßig Aktionen mehrfach durchgeführt werden. Bei der Verbesserung eines Conditional Dispatcher werden zum Beispiel alle zugehörigen If-Anweisungen in einem ersten Schritt jeweils in eigene Methoden extrahiert. Auch beim Extrahieren von Anweisungen gibt es von dem bisher modellierten Transformationsdiagramm aus Abbildung 3.1 noch nicht berücksichtigte Schritte, die mehrfach durchgeführt werden müssen.

So können sich unterhalb der zu extrahierenden Anweisungen Zugriffe auf lokale Variablen oder Parameter der umgebenden Methode befinden, die außerhalb des zu extrahierenden Bereichs deklariert werden. Das Extrahieren der Anweisungen würde in einem solchen Fall zu verbotenen Variablenzugriffen führen. Die Zugriffe können sowohl lesend als auch schreibend sein.

[2] Zur Bestimmung einer Anwendungsstelle muss der Pfad tatsächlich in Rückwärtsrichtung von stmt ausgehend gebunden werden.

[3] Die Kurzform w/o steht dabei für engl. *without*.

Bei lesenden Zugriffen muss sichergestellt werden, dass die Variablen beziehungsweise ihre Werte auch in der neuen Methode gelesen werden können. Dies kann dadurch erreicht werden, dass für jeden extrahierten lesenden Variablenzugriff in der neuen Methode ein typkonformer Parameter deklariert wird, der Variablenzugriff auf diesen Parameter umgeleitet wird und die ursprüngliche Variable beim Aufruf der extrahierten Methode übergeben wird.

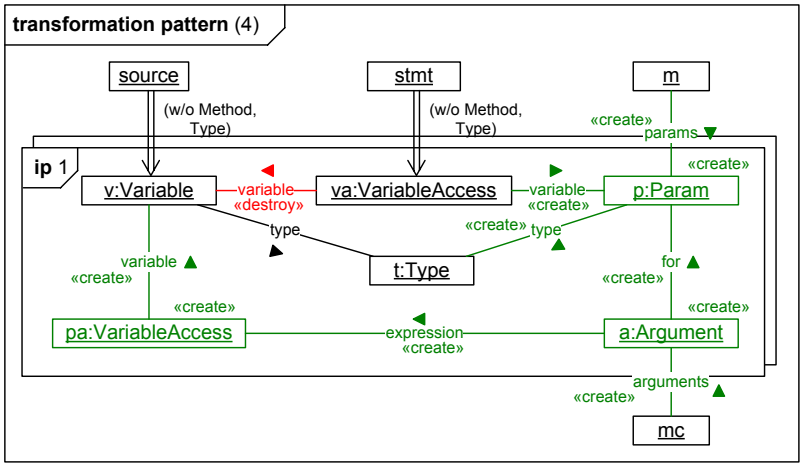

Abbildung 3.3: Transformation Pattern mit einem iterierten Anteil

Story Diagramme erlauben die Spezifikation komplexer Schleifen, deren Rümpfe Sequenzen von Story Pattern mit Alternativen und weiteren Schleifen enthalten können, so dass eine Vielzahl verschiedener Ausprägungen einer Iteration auftreten können. Zudem kann eine Schleife mit einer *while*-Semantik, die solange betreten wird, wie es eine Anwendungsstelle für ein bestimmtes Story Pattern gibt, oder mit einer *foreach*-Semantik, so dass sie für jede Anwendungsstelle eines bestimmten Story Pattern höchstens einmal betreten wird, spezifiziert werden. In beiden Fällen werden durch Ausführung einer Schleife entstehende neue Anwendungsstellen für die Schleife direkt berücksichtigt.

Im Weiteren werden Schleifen so eingeschränkt, dass sie immer eine *foreach*-Semantik haben und ihre Bedingung sowie ihr Rumpf mit Hilfe eines einzigen speziellen Transformation Pattern beschrieben werden. Zudem dürfen Schleifen ineinander geschachtelt werden. Die eingangs in Abschnitt 3.2 formulierte Einschränkung in Bezug auf das (zufällige) Binden von zuvor durch dasselbe Transformationsdiagramm erzeugten Elementen gilt auch für Schleifen. Schafft eine Schleife neue Anwendungsstellen für sich selbst, werden diese so nicht direkt berücksichtigt, es sein denn, sie entstehen durch ein von der Schleife

aufgerufenes Transformationsdiagramm (vgl. Abschnitt 3.2.3). Damit können durch die Hintereinanderausführung mehrerer Iterationen einer Schleife weniger unterschiedliche Wirtsgraphen entstehen, so dass durch das Verifikationsverfahren weniger Kombinationsmöglichkeiten untersucht werden müssen und Schleifen insgesamt einfacher zu handhaben sind. Zudem ist die Terminierung von Schleifen (ohne Transformationsaufrufe) garantiert.

Zur Spezifikation wird mit dem *iterierten Anteil* ein neues syntaktisches Konstrukt eingeführt. Abbildung 3.3 zeigt ein Transformation Pattern, das die zuvor beschriebenen Modifikationen spezifiziert. Es kann am Ende des bisher modellierten Transformationsdiagramms zwischen dem dritten Transformation Pattern und der Success-Stopaktivität eingefügt werden. Das Transformation Pattern enthält mit source, stmt, m und mc gebundene Objektvariablen für die ursprüngliche Methode, die zu extrahierende Anweisung, die neu erzeugte Methode und den Aufruf der neuen Methode.

Darüber hinaus enthält es einen iterierten Anteil, der durch zwei überlagerte Rechtecke und die Beschriftung ip (engl. *iterated part*) gekennzeichnet ist. Der iterierte Anteil bindet bei einer Anwendung einen Lesezugriff va:VariableAccess unterhalb der Anweisung stmt auf eine Variable v:Variable, die in der ursprünglichen Methode source deklariert ist. Dies geschieht über Pfade, die innerhalb desselben Methodenrumpfes bleiben müssen, da sie keine Methoden- und Typobjekte traversieren dürfen. Zusätzlich wird der Typ der Variablen gebunden.

Ist dies erfolgreich, wird ein Parameter p:Param der neuen Methode m erzeugt, der typgleich zur gelesenen Variable ist. Der Lesezugriff wird so angepasst, dass er auf den neuen Parameter zugreift. Zusätzlich wird ein neuer Lesezugriff pa:VariableAccess auf die Variable erzeugt, der als Argument a:Argument für den neuen Parameter beim Aufruf mc von m übergeben wird.

Einige der Objektvariablen des iterierten Anteils sind mit Objektvariablen des umgebenden *nicht-iterierten Anteils* des Transformation Pattern verbunden. Dies bedeutet, dass eine Anwendungsstelle für den iterierten Anteil wie spezifiziert mit diesen Elementen der Anwendungsstelle für den nicht-iterierten Anteil verbunden sein muss. Solche Objektvariablen werden im Folgenden als die *Schnittstellenobjektvariablen* eines iterierten Anteils bezeichnet. Jeder iterierte Anteil muss analog zum nicht-iterierten Anteil eines Transformation Pattern über mindestens eine gebundene Objektvariable verfügen. Schnittstellenobjektvariablen werden aus Sicht des iterierten Anteils ebenfalls wie zu ihm gehörende gebundene Objektvariablen betrachtet.

Bei der Ausführung des Transformation Pattern wird zunächst eine Anwendungsstelle für den nicht-iterierten Anteil bestimmt und er wird darauf angewendet. In diesem Fall besteht der nicht-iterierte Anteil nur aus gebundenen

Objektvariablen, so dass eine Anwendungsstelle garantiert vorhanden ist und bei der Anwendung keine Modifikationen durchgeführt werden. Danach wird eine Anwendungsstelle für den iterierten Anteil gesucht, die in den *Schnittstellenobjekten* mit der nicht-iterierten Anwendungsstelle übereinstimmt. Existiert eine Anwendungsstelle wird der iterierte Anteil angewendet. Danach wird erneut eine Anwendungsstelle für den iterierten Anteil gesucht, die noch nicht gebunden wurde, und sofern vorhanden bearbeitet. Dies geschieht solange, bis keine neue Anwendungsstelle mehr gefunden werden kann.

Das in Abbildung 3.3 gezeigte Transformation Pattern erzeugt für jeden lesenden Zugriff auf eine Variable der Ursprungsmethode einen Parameter der neuen Methode, an den der Wert der Variable beim Aufruf übergeben wird. Gibt es mehrere Lesezugriffe auf dieselbe Variable, so wird dabei für jeden ein eigener Parameter angelegt und dieselbe Variable wird mehrfach beim Aufruf übergeben. Das funktioniert zwar, aber schöner wäre es, wenn für eine gelesene Variable nur ein Parameter angelegt würde und alle weiteren Zugriffe auf dieselbe Variable auf diesen einen Parameter umgelenkt würden.

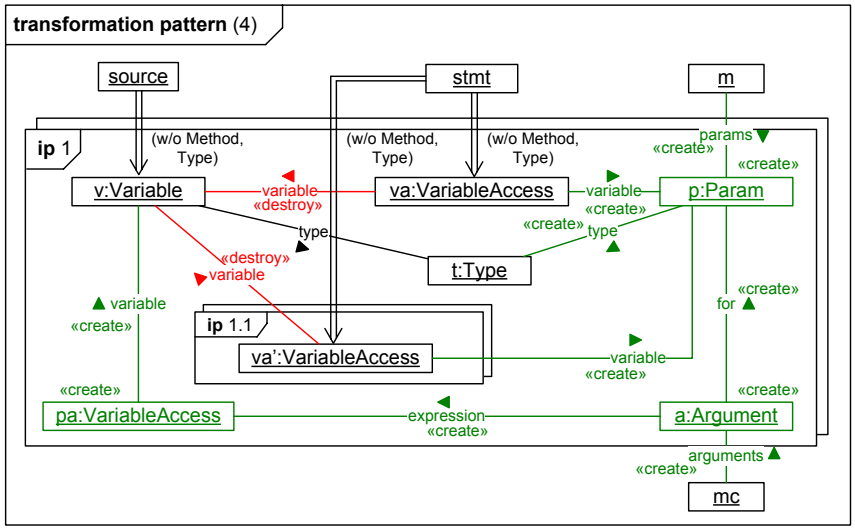

Abbildung 3.4: Transformation Pattern mit zwei geschachtelten iterierten Anteilen

Abbildung 3.4 zeigt ein erweitertes Transformation Pattern, das genau dies tut. Es entspricht im Wesentlichen dem zuvor gezeigten Pattern, verfügt aber über einen weiteren geschachtelten iterierten Anteil innerhalb des ersten iterierten Anteils. Bei der Ausführung des Transformation Pattern wird zuerst der

nicht-iterierte Anteil angewendet und danach eine Anwendungsstelle für den ersten iterierten Anteil gesucht. Dabei wird der darin enthaltene iterierte Anteil zunächst ignoriert. Ist die Anwendung erfolgreich, wird als nächstes nach einer Anwendungsstelle für den geschachtelten Anteil gesucht. Der geschachtelte Anteil versucht einen weiteren Lesezugriff unterhalb von stmt zu binden, der ebenfalls auf die durch den umgebenden Anteil gebundene Variable v zugreift. Gelingt dies, wird der Zugriff auf den durch den umgebenden Anteil erzeugten Parameter p umgelenkt. Danach wird eine weitere Anwendungsstelle für den geschachtelten Anteil gesucht und bearbeitet, solange bis es keine weitere gibt. Danach wird versucht, eine neue Anwendungsstelle für den umgebenden ersten iterierten Anteil zu finden und so fort. Der geschachtelte Anteil hat Priorität gegenüber der nächsten Anwendung des ihn enthaltenden Anteils.

Im Allgemeinen kann ein Transformation Pattern beliebig viele ineinander geschachtelte iterierte Anteile enthalten. Ein Anteil, der einen iterierten Anteil enthält, auch über mehrere Hierarchiestufen hinweg, wird *Vateranteil* genannt. Als *direkter Vateranteil* wird der Vateranteil bezeichnet, der einen iterierten Anteil unmittelbar enthält. Der nicht-iterierte Anteil ist damit Vateranteil für alle iterierten Anteile.

Innerhalb des nicht-iterierten oder eines iterierten Anteils kann es mehrere iterierte Anteile auf derselben Hierarchiestufe geben. Diese werden nummeriert und in aufsteigender Reihenfolge ausgeführt. Es darf keine Linkvariablen zwischen Objektvariablen geben, die sich in unterschiedlichen Hierarchien von Anteilen befinden.

Ein Transformation Pattern mit iterierten Anteilen gilt als erfolgreich angewendet, sobald der nicht-iterierte Anteil angewendet werden konnte.

3.2.3 Transformationsaufrufe

Die Verbesserung von Schwachstellen wie zum Beispiel von Conditional-Dispatcher-Instanzen erfordert umfangreiche Modifikationen des abstrakten Syntaxgraphen. Dabei werden häufig Teile einer Transformation an verschiedenen Stellen wiederverwendet. Bei der Verbesserung eines Conditional Dispatcher wird zum Beispiel mehrfach das Extract-Method-Refactoring verwendet. Daher wird ermöglicht, umfangreiche Transformationen in mehrere Transformationsdiagramme zu zerlegen, die einander aufrufen.

Auch Story Diagramme bieten bereits die Möglichkeit andere Story Diagramme aufzurufen. Story Driven Modeling dient der Spezifikation objektorientierter Systeme. Jedes Story Diagramm beschreibt eine Methode einer Klasse des Strukturmodells. Der Aufruf eines Story Diagramms kann durch den Auf-

ruf dieser Methode auf einem Objekt der entsprechenden Klasse erfolgen, das durch ein Story Pattern gebunden wurde. Abweichend von dieser objektorientierten Sicht werden Transformationsdiagramme explizit zur Spezifikation von Programmtransformationen verwendet und nicht zur Beschreibung von Methoden eines Strukturmodells. Daher wird hier ein eigenes Sprachelement für den Aufruf von Transformationsdiagrammen eingeführt, das zudem komfortabler zu benutzen ist, da es das direkte Binden mehrerer Ergebnisobjekte erlaubt.

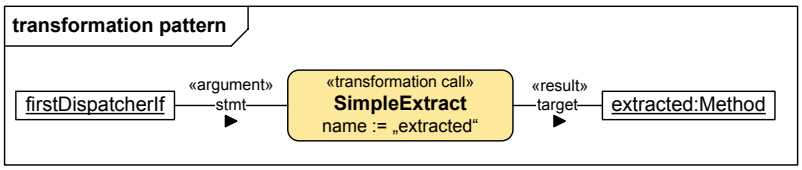

Abbildung 3.5: Transformation Pattern mit einem Transformationsaufruf

Das Transformation Pattern in Abbildung 3.5 zeigt einen der ersten Schritte bei der Verbesserung einer ElseIfDispatcher-Instanz (siehe Abschnitt 2.1), in dem die erste If-Anweisung und damit der gesamte Dispatcher in eine neue Methode extrahiert wird. Die erste If-Anweisung ist bereits an die Objektvariable firstDispatcherIf gebunden. Darüber hinaus enthält das Transformation Pattern einen Transformationsaufruf.

Der Aufruf wird als orangefarbenes Rechteck mit abgerundeten Ecken und ≪transformation call≫ beschriftet dargestellt. Er trägt mit SimpleExtract den Namen des Transformationsdiagramms, das aufgerufen werden soll. Damit das Transformationsdiagramm aufgerufen werden kann, müssen all seine Parameter an entsprechende Argumente gebunden werden.

Im Beispiel verfügt das aufgerufene Transformationsdiagramm über die zwei Parameter stmt:Statement für die zu extrahierende Anweisung und name:String für den Namen der neuen Methode. Der Parameter stmt wird an firstDispatcherIf als Argument gebunden. Dies wird spezifiziert durch den Link stmt ≪argument≫ zum Transformationsaufruf. Bei dem Parameter name handelt es sich mit einer Zeichenkette um einen primitiven Typ. Dieser wird durch die im Aufrufrechteck angegebene Zuweisung auf das Literal „extracted“ gesetzt.

Die aufgerufene Transformation liefert bei erfolgreicher Ausführung die neu erzeugte Methode als Ergebnis zurück. Der Rückgabewert ist in der Signatur des Transformationsdiagramms als target:Method deklariert. Im Transformation Pattern wird der Rückgabewert an die Objektvariable extracted:Method gebunden. Dies wird analog zur Übergabe von Argumenten durch den Link target ≪result≫ spezifiziert.

Transformationsaufrufe und Ergebnisobjekte, die Rückgabewerte eines aufgerufenen Transformationsdiagramms binden, gehören zur rechten Seite eines Transformation Pattern. Bei der Ausführung eines Transformation Pattern wird zunächst wie gehabt eine Anwendungsstelle für die linke Regelseite gesucht. Wird eine solche gefunden, werden Elemente gemäß Spezifikation zunächst gelöscht und dann erzeugt. Danach werden Transformationsaufrufe ausgeführt. Nach erfolgreicher Ausführung der Aufrufe werden die Ergebnisobjekte gebunden. Enthält ein Transformation Pattern mehrere Transformationsaufrufe, werden sie nummeriert und in aufsteigender Reihenfolge ausgeführt.

Bei iterierten Anteilen, die Transformationsaufrufe enthalten, wird analog vorgegangen. Transformationsaufrufe werden bei jeder Anwendung eines iterierten Anteils nach den Löschungen und Erzeugungen ausgeführt und bevor geschachtelte iterierte Anteile ausgeführt werden.

Transformationsaufrufe können fehlschlagen und zwar wenn die Ausführung des aufgerufenen Transformationsdiagramms in einer Failure-Stopaktivität endet. In einem solchen Fall gilt die Ausführung des aufrufenden Transformation Pattern ebenso als fehlgeschlagen und das Transformationsdiagramm wird über die [failure]-Transition fortgesetzt. Dies gilt auch, wenn der Transformationsaufruf eines iterierten Anteils fehlschlägt. In diesem Fall gilt die Anwendung des gesamten Transformation Pattern als fehlgeschlagen.

Es kann anhand der Spezifikation nicht ohne Weiteres erkannt werden, ob die Ausführung des Transformation Pattern aufgrund eines fehlgeschlagenen Transformationsaufrufs misslingt oder weil es keine Anwendungsstelle für seine linke Regelseite gab. Darüber hinaus kann ein Transformation Pattern zum Zeitpunkt eines fehlschlagenden Transformationsaufrufs bereits Änderungen am Wirtsgraphen vorgenommen haben. Dies steht im Widerspruch dazu, dass nur erfolgreich angewendete Transformation Pattern Änderungen vornehmen. Um diese Probleme aufzulösen, werden Transformationsdiagramme so eingeschränkt, dass Transformation Pattern, die Transformationsaufrufe enthalten, im Fehlerfall immer zu einer Failure-Stopaktivität verzweigen müssen, so dass alle bis dahin vorgenommenen Änderungen rückgängig gemacht werden. Transformation Pattern mit Transformationsaufrufen müssen also immer erfolgreich angewendet werden.

3.2.4 Strukturmuster

Transformationsdiagramme werden zur Verbesserung von durch die strukturbasierte Mustererkennung erkannten Schwachstellen eingesetzt. Die Mustererkennung markiert ihre Funde im abstrakten Syntaxgraphen mit Hilfe von

Strukturmusterannotationen. Für jedes Strukturmuster wird das Strukturmodell um einen konkreten Annotationstyp erweitert. Damit können Transformation Pattern Strukturmusterannotationen wie alle anderen Objekte eines abstrakten Syntaxgraphen an entsprechende Objektvariablen binden und sich explizit auf die Ergebnisse der Mustererkennung beziehen. Objektvariablen für Strukturmusterannotationen werden dabei wie in Strukturmustern oval dargestellt und im Weiteren *Annotationsvariablen* genannt.

Annotationsvariablen in Transformation Pattern können zwei verschiedene Bedeutungen haben. Zu dem Zeitpunkt, zu dem die Annotation erzeugt wurde, gab es im Wirtsgraphen die durch das zugehörige Strukturmuster definierte Struktur. Zum Zeitpunkt des Bindens einer Annotation durch ein Transformation Pattern muss dies nicht mehr der Fall sein. Dann dient sie lediglich als Markierungsknoten, von dem ausgehend bestimmte Elemente gebunden werden können, die einmal Bestandteil eines Musters waren. Es gibt Fälle, in denen diese Art der Nutzung ausreichend ist.

Dementgegen gibt es Situationen, in denen mit der Bindung einer Annotation auch ihre Gültigkeit, dass heißt das Vorhandensein der Struktur gemäß Strukturmuster, verbunden sein soll. In solchen Fällen sind die Annotationsvariablen im Transformation Pattern zusätzlich durch den Stereotyp ≪validate≫ zu kennzeichnen. Die Kennzeichnung löst beim Binden einer Annotation eine zusätzliche Überprüfung des Wirtsgraphen auf Vorhandensein der Struktur aus, die erfolgreich sein muss.

Darüber hinaus wird das Erzeugen von Annotationen ermöglicht, wobei Instanzen des durch die Annotation repräsentierten Musters erzeugt werden. Ein Muster wird durch ein Strukturmuster beschrieben, das nur zur Erkennung von Instanzen des Musters eingesetzt werden kann. Um Instanzen auch erzeugen zu können, muss zusätzlich spezifiziert werden, wie die Erzeugung durchzuführen ist. Dies erfolgt mit Hilfe von Transformationsdiagrammen.

Abbildung 3.6 zeigt dies anhand des GetMethod-Musters. Das GetMethod-Muster repräsentiert eine lesende Zugriffsmethode für ein Attribut. Die Abbildung zeigt rechts das zugehörige Strukturmuster, das eine Methode innerhalb einer Klasse als Zugriffsmethode annotiert, in der sich eine Return-Anweisung befindet, die auf ein Attribut derselben Klasse zugreift. Links daneben wird ein Transformationsdiagramm gezeigt, dass Instanzen dieses Musters erzeugt.

Ein Strukturmuster definiert Elemente des Musters, die durch eine Annotation markiert werden. Im Fall der GetMethod werden das gelesene Attribut mit field, die Zugriffsmethode mit method und der Lesezugriff mit access annotiert. Aus dieser Schnittstelle leitet sich die Signatur einer Erzeugungstransformation ab. Alle annotierten Elemente müssen mit der entsprechenden Bezeichnung

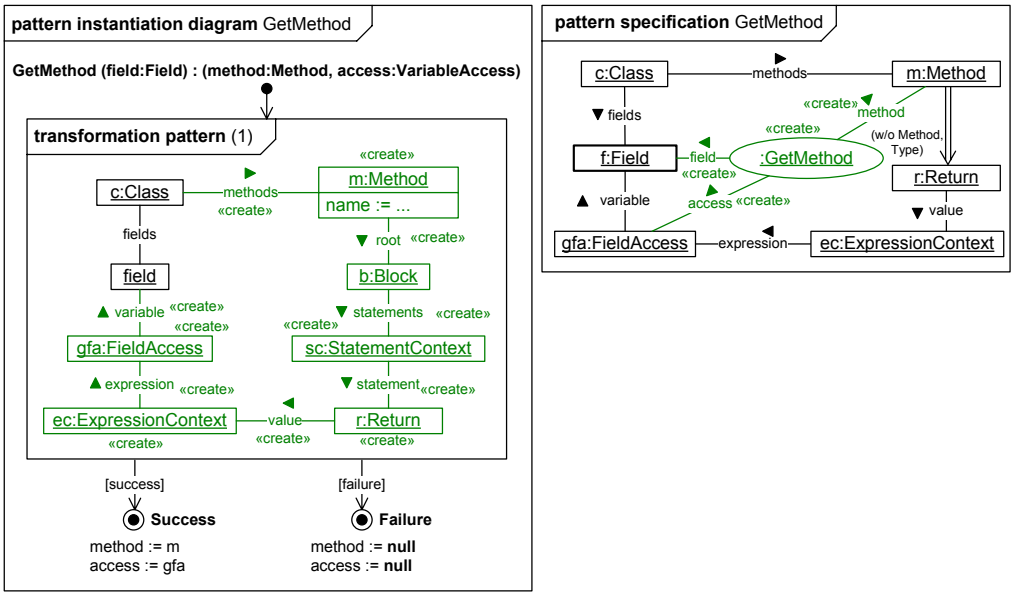

Abbildung 3.6: Transformationsdiagramm zur Instanzierung und Strukturmuster zur Erkennung einer lesenden Zugriffsmethode

entweder als Parameter oder als Rückgabewert deklariert werden. Das Transformationsdiagramm im Beispiel erwartet das Attribut field:Field als Parameter. Als Rückgabewerte werden das Methodenobjekt method:Method sowie der Variablenzugriff access:VariableAccess definiert.

Das einzige Transformation Pattern der Erzeugungstransformation bindet das übergebene Attribut field und davon ausgehend die umgebende Klasse c:Class. Danach wird eine neue Methode mit einer Return-Anweisung erzeugt, die den Wert des Attributs zurückgibt. Ist dies erfolgreich, werden die neu erzeugte Methode sowie der Lesezugriff an die entsprechenden Rückgabewerte gebunden und das Transformationsdiagramm wird verlassen.

Bei erfolgreicher Beendigung einer Erzeugungstransformation, wird automatisch eine Annotation zur Markierung der neu geschaffenen Musterinstanz erzeugt. Die Annotation wird mit allen Elementen der Signatur unter Verwendung ihrer Bezeichner verbunden, so dass eine zum Strukturmuster konforme Markierung entsteht. Eine Erzeugungstransformation muss eine zum zugehörigen Strukturmuster konforme Struktur erzeugen.

Das in Abbildung 3.7 gezeigte Transformationsdiagramm zur Verkapselung von Lesezugriffen macht Gebrauch von der Mustererzeugung. Es beschreibt einen Teil des *EncapsulateField*-Refactorings [Fow99], das unter Anderem im

Rahmen der Restrukturierung einer Conditional-Dispatcher-Instanz in eine Command-Lösung eingesetzt wird, um den einzelnen Kommandoklassen den Zugriff auf Daten der ursprünglichen Dispatcher-Klasse zu ermöglichen.

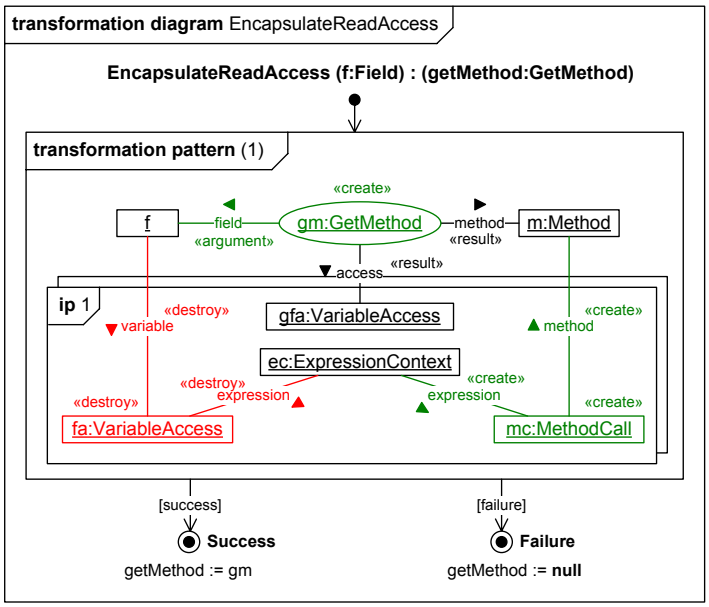

Abbildung 3.7: Transformationsdiagramm zur Verkapselung von Lesezugriffen

Das Transformationsdiagramm bekommt das zu verkapselnde Attribut f:Field übergeben. Das einzige Transformation Pattern bindet das Attribut und erzeugt mit gm:GetMethod eine GetMethod-Instanz.

Das Erzeugen von Musterinstanzen wird abgebildet auf Transformationsaufrufe. So wird in diesem Fall die in Abbildung 3.6 gezeigte Erzeugungstransformation aufgerufen. Dabei wird f als Argument für den Parameter field:Field übergeben (spezifiziert durch den Link field ≪argument≫). Musterinstanzierungen werden analog zu Transformationsaufrufen durchgeführt, nachdem alle übrigen Modifikationen erfolgt sind, im Beispiel als letzte Aktion des nicht-iterierten Anteils. Ebenso sind Transformation Pattern mit Musterinstanzierungen nur dann erfolgreich, wenn alle Musterinstanzierungen erfolgreich sind.

Bei erfolgreicher Musterinstanzierung wird automatisch eine Annotation zur Markierung erzeugt und anschließend gebunden. Darüber hinaus können nun noch Rückgabewerte von der Annotation ausgehend gebunden werden, im Beispiel m:Method via method ≪result≫. Der iterierte Anteil des Transformation Pattern ersetzt anschließend alle direkten Lesezugriffe auf das Attribut au-

ßerhalb der neuen Zugriffsmethode durch Aufrufe dieser Methode. Schließlich wird der Rückgabewert an die neu erzeugte Annotation gebunden.

Durch die explizite Instanzierung von Mustern können Transformationen Musterinstanzen explizit in andere Musterinstanzen überführen. So gibt es bei einem Conditional Dispatcher verschiedene Varianten, in denen der Inhalt einiger oder aller verketteter If-Anweisungen bereits in eigene Methoden ausgelagert wurde. In diesem Fall wird von einer extrahierten Anfragebehandlung beziehungsweise von einem extrahierten Dispatcher gesprochen, wenn alle verketteten If-Anweisungen extrahiert sind. Zur Erkennung dieser extrahierten Varianten gibt es eigene Strukturmuster. Eine erste Transformation kann nun zunächst alle nicht-extrahierten Varianten in extrahierte Varianten überführen und explizit als solche markieren. Eine weitere Transformation kann dann darauf aufsetzen, unabhängig davon, ob die extrahierten Varianten bereits vorgelegen haben und durch die Mustererkennung identifiziert wurden oder ob sie durch eine Transformation erzeugt wurden.

Schließlich ist auch das Löschen einer Annotation erlaubt, wobei nur die Annotation sowie ihre Links gelöscht werden. Die annotierte Struktur bleibt unverändert. Es ist denkbar analog zu Erzeugungstransformationen, die mit Konstruktoren vergleichbar sind, auch das Löschen einer Musterinstanz, also eine Art Destruktor, zu beschreiben. Dieser Bedarf hat sich jedoch nicht ergeben, so dass dies im Rahmen dieser Arbeit nicht weiter verfolgt wird.

3.3 Formalisierung von Transformationsdiagrammen

Die zuvor informal eingeführten Transformationsdiagramme sind zugunsten der Verifikation stärker eingeschränkt als andere Graphtransformationssprachen wie Story Diagramme [Zün02] oder PROGRESS [Sch91] sowie als klassische Graphtransformationssysteme ohne Kontrollstrukturen [Roz97] und können nicht ohne Weiteres auf diese abgebildet werden, um ihre Semantik eindeutig zu definieren. Daher werden sie in diesem Abschnitt aufbauend auf [Roz97, Sch06] auf Basis von Typgraphen, typisierten Graphen und speziellen Graphtransformationen formalisiert.

Auf einige Vorbemerkungen betreffend die im Weiteren verwendete Notation folgt eine Reihe von Definitionen zur Formalisierung der Syntax von Transformationsdiagrammen. Anschließend wird darauf basierend die Semantik von Transformationsdiagrammen mit Hilfe von Auswertungsrelationen operational

formalisiert.

3.3.1 Vorbemerkungen

In den folgenden Definitionen werden zahlreiche Strukturen definiert, die aus Mengen und Abbildungen zwischen diesen Mengen bestehen. Seien X und Y zwei Mengen, dann wird eine totale Abbildung f von X nach Y durch $f\colon X \to Y$ notiert. Dass es sich um eine totale Abbildung handelt, wird dabei nicht explizit angegeben.

Ist f eine partielle Abbildung, so wird sie durch $f\colon X \rightharpoonup Y$ notiert. Zusätzlich wird explizit ausgesagt, dass sie partiell ist. Eine partielle Abbildung darf auch total sein. Die Menge aller partiellen Abbildungen zwischen X und Y wird durch $[X \rightharpoonup Y]$ notiert.

Für einige Abbildungen wird gefordert, dass sie *zyklenfrei* sind:

Definition 1 *Eine Abbildung $f\colon X \rightharpoonup X$ heißt* zyklenfrei*, wenn ihre beliebig häufige Hintereinanderausführung ($f^i(x)$) für ein beliebiges $x \in X$ nicht wieder x als Ergebnis liefert: $\nexists n \in \mathbb{N}, x \in X\colon (\forall i \in \{1,\ldots,n-1\}\colon f^i(x) \in X) \wedge f^n(x) = x$.*

Zudem werden Abbildungen als Relationen aufgefasst, so dass die transitive Hülle einer Abbildung f, notiert durch f^+, berechnet werden kann.

Die Menge aller Teilmengen einer Menge X wird durch $\wp(X)$ bezeichnet.

Eine Struktur S, die aus den Mengen X und Y besteht, wird durch $S := (X, Y)$ definiert. Wird eine Ausprägung einer solchen Struktur in weiteren Definitionen benutzt und soll dabei zum Beispiel auf die Menge X als Bestandteil von S Bezug genommen werden, so wird dies durch X_S notiert. Allgemein wird an den Bezeichner eines Bestandteils einer Struktur der Bezeichner jener Struktur als Index geschrieben, um den Zusammenhang auszudrücken. Darüber hinaus wird die Indexschreibweise auch anderweitig genutzt, zum Beispiel um andere Zusammenhänge zwischen den bezeichneten Elementen herzustellen oder gleichartige Elemente voneinander zu unterscheiden.

3.3.2 Typisierte Graphen

Wie in Abschnitt 2.3.1 beschrieben wird der Quelltext eines Softwaresystems in dieser Arbeit als abstrakter Syntaxgraph repräsentiert. Daher werden hier zunächst Graphen formal definiert.

Definition 2 *Ein* Graph *ist ein Tupel* $G := (N, E, src, tgt)$ *wobei* N *die Menge der Knoten und* E *die Menge der Kanten von* G *sind. Die Abbildungen* $src : E \to N$ *und* $tgt : E \to N$ *ordnen jeder Kante ihren Start- respektive Zielknoten zu. Der leere Graph wird mit* $G_\emptyset$ *bezeichnet.*

Der Syntaxgraph ist wie in Abschnitt 3.2.1 gezeigt durch ein um Pfadnavigationsinformationen ergänztes Strukturmodell typisiert. Daher wird hier ein Strukturmodell als Typgraph formal definiert. Die Knoten eines solchen Typgraphen entsprechen den Klassen eines Strukturmodells und die Kanten den Assoziationen. Für jede Seite einer Assoziation wird eine der Kardinalitäten 0..1 (maximal ein Objekt darf zugeordnet werden) oder 0..∗ (beliebig viele Objekte dürfen zugeordnet werden) angegeben. Zusätzlich werden Vererbungsbeziehungen zwischen Klassen sowie die Traversierbarkeit von Assoziationen durch Pfade formalisiert. Attribute werden zur Vereinfachung nicht berücksichtigt.

Definition 3 *Ein* Typgraph mit Pfadinformationen *ist ein Tupel* $G_\Omega := (N_\Omega, E_\Omega, src_\Omega, srcCard_\Omega, tgt_\Omega, tgtCard_\Omega, inh_\Omega, PTE_\Omega)$. *Dabei ist* $(N_\Omega, E_\Omega, src_\Omega, tgt_\Omega)$ *ein Graph gemäß Definition 2 und die Abbildungen* $srcCard_\Omega, tgtCard_\Omega \colon E \to \{0..1, 0..*\}$ *ordnen jeder Seite einer Kante ihre Kardinalität zu.*

$inh_\Omega \colon N_\Omega \rightharpoonup N_\Omega$ *ist eine partielle, zyklenfreie Abbildung, die definiert, welche Klassen voneinander erben.*

$PTE_\Omega \subseteq \{(n_{src}, e, n_{tgt}) \in N_\Omega \times E_\Omega \times N_\Omega | ((n_{src} = src_\Omega(e) \wedge n_{tgt} = tgt_\Omega(e)) \vee (n_{tgt} = src_\Omega(e) \wedge n_{src} = tgt_\Omega(e)))\}$ *legt fest, welche Assoziationen in welche Richtung durch einen Pfad traversiert werden dürfen (engl. Path Traversible Edges).*

Wenn $n_{super} \in N_\Omega$ die Oberklasse von $n_{sub} \in N_\Omega$ ist, gilt $n_{super} = inh_\Omega(n_{sub})$. Zusätzlich wird die Menge aller (transitiven) Oberklassen für eine Klasse $n \in N_\Omega$ definiert als $super_\Omega(n) := \{n' | (n, n') \in inh_\Omega^+\}$. Die Menge $types_\Omega(n) := \{n\} \cup super_\Omega(n)$ enthält zusätzlich zu den Oberklassen von n auch noch die Klasse n selbst. Darauf aufbauend wird schließlich noch die Menge $PTE_\Omega^{inh} := \{(n', e, n'') | (n_{src}, e, n_{tgt}) \in PTE_\Omega \wedge n_{src} \in types_\Omega(n') \wedge n_{tgt} \in types_\Omega(n'')\}$ definiert, die alle pfadnavigierbaren Assoziationen auch in Verbindung mit allen Subklassen der durch die jeweilige Assoziation verbundenen Klassen enthält.

Ein typisierter Graph wird dann wie folgt definiert.

Definition 4 *Ein durch einen Typgraph* G_Ω *gemäß Definition 3* typisierter Graph *ist ein Tupel* $G := (N, E, src, tgt, l^N, l^E)$ *wobei* (N, E, src, tgt) *ein*

Graph gemäß Definition 2 ist. Die Abbildung $l^N : N \to N_\Omega$ *ordnet jedem Knoten des Graphen seinen Typ (seine Klasse) in* G_Ω *zu.* $l^E : E \to E_\Omega$ *ordnet analog jeder Kante ihren Typ (ihre Assoziation) in* G_Ω *zu.*

Damit die Typkonformität sichergestellt ist, muss zum einen gelten:

$$\begin{aligned}&\forall e \in E\colon\\ &\quad src_\Omega(l^E(e)) \in types_\Omega(l^N(src(e))) \wedge\\ &\quad tgt_\Omega(l^E(e)) \in types_\Omega(l^N(tgt(e))).\end{aligned}$$

Zum anderen müssen die definierten Kardinalitäten eingehalten werden:

$$\begin{aligned}&\forall e \in E\colon\\ &\quad (tgtCard_\Omega(l^E(e)) = 0..1 \Rightarrow\\ &\qquad \nexists e' \in E, e' \neq e\colon l^E(e') = l^E(e) \wedge src(e') = src(e)) \wedge\\ &\quad (srcCard_\Omega(l^E(e)) = 0..1 \Rightarrow\\ &\qquad \nexists e' \in E, e' \neq e\colon l^E(e') = l^E(e) \wedge tgt(e') = tgt(e)).\end{aligned}$$

Ein typisierter Graph formalisiert so eine Objektstruktur, die Instanz eines als Typgraph formalisierten Strukturmodells ist: seine Knoten sind Objekte und seine Kanten sind Links, die jeweils Instanzen einer Klasse beziehungsweise einer Assoziation des Strukturmodells sind. Die Menge aller durch einen Typgraphen G_Ω typisierten Graphen wird mit $\mathcal{G}[G_\Omega]$ bezeichnet. Im Weiteren wird vorausgesetzt, dass jeder Graph typisiert ist, so dass dies im Text nicht mehr gesondert betont wird.

Zur weiteren Formalisierung von Transformationsdiagrammen wird der Begriff des Teilgraphen sowie die Vereinigung zweier Graphen $G_1, G_2 \in \mathcal{G}[G_\Omega]$ benötigt.

Ein Graph G_1 ist Teilgraph eines Graphen G_2, wenn die Knoten- und Kantenmengen von G_1 Teilmengen der Knoten- und Kantenmengen von G_2 sind und die Funktionen für die Start- und Zielknoten sowie für die Typzuordnung eingeschränkt auf G_1 gleich sind.

Definition 5 *Ein Graph* $G_1 \in \mathcal{G}[G_\Omega]$ *ist ein* Teilgraph *eines Graphen* $G_2 \in \mathcal{G}[G_\Omega]$*, geschrieben* $G_1 \leq G_2$*, falls* $N_{G_1} \subseteq N_{G_2}$*,* $E_{G_1} \subseteq E_{G_2}$*,* $src_{G_1} = src_{G_2}|_{E_{G_1}}$*,* $tgt_{G_1} = tgt_{G_2}|_{E_{G_1}}$*,* $l^N_{G_1} = l^N_{G_2}|_{N_{G_1}}$*,* $l^E_{G_1} = l^E_{G_2}|_{E_{G_1}}$*.*

Die Vereinigung zweier Graphen G_1 und G_2 resultiert in einem Graphen, der alle Knoten und Kanten beider Graphen enthält. Er wird gebildet, indem die

Knoten- und Kantenmengen vereinigt und die Funktionen für die Start- und Zielknoten von Kanten sowie für die Typzuordnungen entsprechend kombiniert werden.

Definition 6 *Die* Vereinigung zweier Graphen $G_1, G_2 \in \mathcal{G}[G_\Omega]$ *ist definiert als* $G_1 \cup G_2 := (N', E', src', tgt', l^{N'}, l^{E'})$ *mit* $N' := N_{G_1} \cup N_{G_2}$, $E' := E_{G_1} \cup E_{G_2}$, $src' := src_{G_1} \oplus src_{G_2}|_{(E_{G_2} \setminus E_{G_1})}$, $tgt' := tgt_{G_1} \oplus tgt_{G_2}|_{(E_{G_2} \setminus E_{G_1})}$, $l^{N'} := l^N_{G_1} \oplus l^N_{G_2}|_{(N_{G_2} \setminus N_{G_1})}$ *und* $l^{E'} := l^E_{G_1} \oplus l^E_{G_2}|_{(E_{G_2} \setminus E_{G_1})}$.

Dabei ist $\oplus$ für zwei Mengen X_1 und X_2 und zwei Funktionen f_1 und f_2 definiert als:

$$f_1 \oplus f_2|_{(X_2 \setminus X_1)}(x) := \begin{cases} f_1(x) & \text{falls } x \in X_1, \\ f_2(x) & \text{falls } x \in X_2 \setminus X_1. \end{cases}$$

Die Vereinigung typkonformer Graphen ist bei dieser Definition nicht zwingend typkonform. Dies ist aufgrund der späteren Verwendung der Vereinigung beabsichtigt und stellt kein Problem dar.

3.3.3 Transformationsdiagramme

Transformationsdiagramme verfügen über eine Signatur und bestehen aus einer Menge von Transformation Pattern sowie einer Menge von Stopaktivitäten, die über Transitionen miteinander verbunden sind. Transformation Pattern wiederum bestehen aus einem nicht-iterierten Anteil sowie beliebig vielen (geschachtelten) iterierten Anteilen, die letztlich durch spezielle Graphtransformationsregeln beschrieben werden. Transformation Pattern (Definition 14) und Graphtransformationsregeln (Definition 13) werden im folgenden Abschnitt definiert.

Zur Formalisierung gebundener Objektvariablen definiert ein Transformationsdiagramm eine Menge von Knotenbezeichnern, mit denen die Knoten in den Graphtransformationsregeln seiner Transformation Pattern beschriftet werden. Während der Ausführung eines Transformationsdiagramms werden die Knotenbezeichner durch Anwendung der Graphtransformationsregeln an Objekte des Wirtsgraphen gebunden. Zudem müssen die Graphtransformationsregeln andere Transformationsdiagramme aufrufen können. Daher werden Transformationsdiagramme hier als erstes definiert. Dies ist unproblematisch, da dabei keine Eigenschaften der noch zu definierenden Transformation Pattern oder Graphtransformationsregeln ausgenutzt werden.

Um die Formalisierung zu vereinfachen, wird davon ausgegangen, dass ein Transformationsdiagramm genau zwei Stopaktivitäten enthält, eine um eine erfolgreiche Ausführung (Success) zu beenden und eine für einen Fehlschlag (Failure); tatsächlich können es beliebig viele Stopaktivitäten sein.

Definition 7 *Ein* Transformationsdiagramm *ist ein Tupel* $TD := (TP, \lambda, s^{\oplus}, s^{\ominus}, succ, fail, V^N, V^P, V^R)$ *wobei* $TP \subseteq \mathcal{TP}$ *eine Menge von Transformation Pattern gemäß Definition 14 ist.* $\lambda \in TP$ *bezeichnet das Transformation Pattern, mit dem die Ausführung des Transformationsdiagramms startet.* $s^{\oplus}$ *bezeichnet die Stopaktivität zur Beendigung einer erfolgreichen Ausführung und* $s^{\ominus}$ *ist die Stopaktivität für den Fehlerfall.*

Die Abbildungen $succ, fail \colon TP \to TP \cup \{s^{\oplus}, s^{\ominus}\}$ *bilden die* success*- beziehungsweise* failure*-Transitionen ab, indem sie jedem Transformation Pattern dasjenige Transformation Pattern, das bei erfolgreicher beziehungsweise fehlgeschlagener Ausführung des ersten Pattern als nächstes ausgeführt werden soll, oder eine Stopaktivität zuweisen.*

Die Menge V^N *stellt diagrammweit gültige Knotenbezeichner zur Verfügung.* $V^P \subseteq V^N$ *definiert die Parameter eines Diagramms und* $V^R \subseteq V^N$ *seine Rückgabewerte.*

Die Menge aller Transformationsdiagramme wird mit $\mathcal{TD}$ bezeichnet.

3.3.4 Transformation Pattern

Die Syntax und Semantik eines Transformation Pattern soll mit Hilfe spezieller Graphtransformationsregeln definiert werden, welche die speziellen syntaktischen Konstrukte (gebundene Objektvariablen, Pfade, Transformationsaufrufe) berücksichtigen.

Der nicht-iterierte Anteil sowie die iterierten Anteile eines Transformation Pattern werden jeweils durch eine solche Graphtransformationsregel beschrieben. Die Hierarchie der Anteile findet sich in der Definition der Transformation Pattern wieder. Die durch Schnittstellenobjektvariablen eines iterierten Anteils definierten Abhängigkeiten von den Anwendungsstellen seiner Vateranteile werden durch gebundene Objektvariablen abgebildet: eine Graphtransformationsregel, die einen iterierten Anteil beschreibt, enthält seine Schnittstellenobjektvariablen als gebundene Objektvariablen.

Eine solche Graphtransformationsregel besteht zunächst aus zwei typisierten *Regelgraphen*, einem für ihre linke Seite und einem für ihre rechte Seite. Ein Regelgraph kann über einfache Graphen hinaus Pfade sowie gebundene Knoten

enthalten und seine Knoten sind mit den Knotenbezeichnern des Transformationsdiagramms beschriftet, zu dem er gehört. Er wird wie folgt definiert:

Definition 8 *Ein* Regelgraph*, der die linke oder rechte Seite einer Graphtransformationsregel eines Transformation Pattern eines Transformationsdiagramms TD beschreibt, ist ein Tupel $RG := (N, E, src^E, tgt^E, P, src^P, tgt^P, N^B, v^N)$ wobei (N, E, src^E, tgt^E) ein Graph gemäß Definition 2 ist, dessen Knoten Objekt- und dessen Kanten Linkvariablen repräsentieren.*

P ist eine Menge von Pfaden und die Abbildungen $src^P, tgt^P \colon P \to N$ ordnen jedem Pfad seinen Start- beziehungsweise Zielknoten zu. Die nicht-leere Menge $N^B \subseteq N$ definiert gebundene Knoten (Objektvariablen). Die injektive Abbildung $v^N : N \to V^N_{TD}$ schließlich ordnet jedem Knoten einen der Knotenbezeichner von TD zu.

Analog zu Graphen werden auch Regelgraphen über einen Typgraphen G_Ω typisiert:

Definition 9 *Ein durch einen Typgraphen G_Ω* typisierter Regelgraph *ist ein Tupel $RG := (N, E, src^E, tgt^E, l^N, l^E, P, src^P, tgt^P, l^P, N^B, v^N)$ wobei $(N, E, src^E, tgt^E, P, src^P, tgt^P, N^B, v^N)$ ein Regelgraph gemäß Definition 8 und $(N, E, src^E, tgt^E, l^N, l^E)$ ein durch G_Ω typisierter Graph gemäß Definition 4 ist. Die Abbildung $l^P : P \to \wp(N_\Omega)$ weist jedem Pfad eine Teilmenge von N_Ω als verbotene Typen zu.*

Zudem muss gelten, dass alle Pfade aufgrund des Typgraphen ausprägbar sein müssen, dass heißt $\forall p \in P \colon \exists \langle n_1, \ldots, n_k \rangle \in N^*_\Omega, \langle e_1, \ldots, e_{k-1} \rangle \in E^*_\Omega$ so dass gilt

$$n_1 = l^N(src^P(p)) \wedge n_k = l^N(tgt^P(p)), \tag{3.1}$$

$$\forall i \in \{1, \ldots, k-1\} \colon src_\Omega(e_i) = n_i \wedge tgt_\Omega(e_i) = n_{i+1} \wedge (n_i, e_i, n_{i+1}) \in PTE^{inh}_\Omega \text{ und} \tag{3.2}$$

$$\forall i \in \{1, \ldots, k\} \colon types_\Omega(n_i) \cap l^P(p) = \emptyset. \tag{3.3}$$

Mit Hilfe der Abbildung l^P werden einem Pfad die Typen zugeordnet, deren Instanzen eine Ausprägung des Pfades nicht enthalten darf. Um sicherzustellen, dass ein Pfad ausprägbar ist, wird gefordert, dass es einen Weg im Typgraphen geben muss, bei dem Start- und Zielknoten den Typen des Start- und

Zielknotens des Pfades entsprechen (3.1), aufeinanderfolgende Knoten durch eine Assoziation verbunden sind, die durch einen Pfad in die entsprechende Richtung traversiert werden darf (3.2), und keiner der Knoten typkonform zu einem verbotenen Typ ist (3.3). Dabei wird Vererbung berücksichtigt.

Die Menge aller durch einen Typgraphen G_Ω typisierten Regelgraphen wird mit $\mathcal{RG}[G_\Omega]$ bezeichnet.

Im Weiteren werden die Teilgraphenbeziehung zwischen Regelgraphen sowie später auch die Vereinigung von Regelgraphen benötigt. Diese werden aufbauend auf die in Abschnitt 3.3.2 gegebenen Definitionen für Graphen hier nun für Regelgraphen definiert.

Definition 10 *Ein Regelgraph* $SG \in \mathcal{RG}[G_\Omega]$ *ist ein* Teilgraph *eines Regelgraphen* $RG \in \mathcal{RG}[G_\Omega]$*, geschrieben* $SG \leq RG$*, falls* $(N_{SG}, E_{SG}, src^E_{SG}, tgt^E_{SG}, l^N_{SG}, l^E_{SG}) \leq (N_{RG}, E_{RG}, src^E_{RG}, tgt^E_{RG}, l^N_{RG}, l^E_{RG})$ *und* $P_{SG} \subseteq P_{RG}$*,* $src^P_{SG} = src^P_{RG}|_{P_{SG}}$*,* $tgt^P_{SG} = tgt^P_{RG}|_{P_{SG}}$*,* $l^P_{SG} = l^P_{RG}|_{P_{SG}}$*,* $N^B_{SG} = N^B_{RG}|_{N_{SG}}$ *und* $v^N_{SG} = v^N_{RG}|_{N_{SG}}$*.*

Definition 11 *Die* Vereinigung zweier Regelgraphen[4] $RG_1, RG_2 \in \mathcal{RG}[G_\Omega]$ *ist* $RG_1 \cup RG_2 := (N', E', src^{E'}, tgt^{E'}, l^{N'}, l^{E'}, P', src^{P'}, tgt^{P'}, l^{P'}, N^{B'}, v^{N'})$ *mit* $(N', E', src^{E'}, tgt^{E'}, l^{N'}, l^{E'}) := (N_{RG_1}, E_{RG_1}, src^E_{RG_1}, tgt^E_{RG_1}, l^N_{RG_1}, l^E_{RG_1}) \cup (N_{RG_2}, E_{RG_2}, src^E_{RG_2}, tgt^E_{RG_2}, l^N_{RG_2}, l^E_{RG_2})$ *und* $P' := P_{RG_1} \cup P_{RG_2}$*,* $src^{P'} := src^P_{RG_1} \oplus src^P_{RG_2}|_{(P_{RG_2} \setminus P_{RG_1})}$*,* $tgt^{P'} := tgt^P_{RG_1} \oplus tgt^P_{RG_2}|_{(P_{RG_2} \setminus P_{RG_1})}$*,* $l^{P'} := l^P_{RG_1} \oplus l^P_{RG_2}|_{(P_{RG_2} \setminus P_{RG_1})}$*,* $N^{B'} := N^B_{RG_1} \cup N^B_{RG_2}$ *und* $v^{N'} := v^N_{RG_1} \oplus v^N_{RG_2}|_{(N_{RG_2} \setminus N_{RG_1})}$*.*

Über die Regelgraphen hinaus besteht eine Graphtransformationsregel aus einer Sequenz von Transformationsaufrufen. Ein Transformationsaufruf wird wie folgt definiert:

Definition 12 *Ein* Transformationsaufruf*, der zu einer Graphtransformationsregel eines Transformation Pattern des Transformationsdiagramms* TD_{caller} *gehört, ist ein Tupel* $tc := (TD, args, res)$ *wobei* $TD \in \mathcal{TD}$ *das aufgerufene Transformationsdiagramm ist und* $args : V^P_{TD} \to V^N_{TD_{caller}}$ *jeden Parameter des aufgerufenen Diagramms auf einen Knotenbezeichner des aufrufenden Diagramms abbildet.* $res : V^R_{TD} \to V^N_{TD_{caller}}$ *bildet jeden Rückgabewert auf einen Knotenbezeichner des Aufrufers ab.*

Die Menge aller Transformationsaufrufe wird mit $\mathcal{TC}$ bezeichnet.

[4] Die Vereinigung ist auch hier nicht zwingend typkonform, vgl. Definition 6.

Durch die Abbildung $args$ wird für jeden Parameter des aufgerufenen Diagramms ein Knotenbezeichner des Aufrufers bestimmt. Beim Aufruf wird das Objekt, das zu dem Zeitpunkt an den Knotenbezeichner gebunden ist, an den entsprechenden Parameter gebunden. Analog bestimmt res für jeden Rückgabewert des aufgerufenen Diagramms einen Knotenbezeichner des Aufrufers, an den das entsprechende Ergebnisobjekt gebunden werden soll.

In der in Abschnitt 3.2.3 gezeigten konkreten Syntax für Transformationsaufrufe werden diese als Rechtecke dargestellt, die mit den Objektvariablen, welche als Argument an bestimmte Parameter gebunden werden, über entsprechend beschriftete Links verbunden sind. Analog sind sie mit Objektvariablen über Links verbunden, die an Ergebnisse des Aufrufs gebunden werden.

Hier werden Transformationsaufrufe nicht als Bestandteil der Regelgraphen sondern abstrakter als Elemente der Graphtransformationsregel repräsentiert. Die Links zu Argumenten und Ergebnissen werden durch die Funktionen $args$ und res des Transformationsaufrufs repräsentiert.

Auf Basis der zuvor definierten typisierten Regelgraphen und der Transformationsaufrufe wird dann eine Graphtransformationsregel wie folgt definiert:

Definition 13 *Eine* Graphtransformationsregel $r := (L, R, TC)$ *auch geschrieben* $L \rightarrow_r R$ *besteht aus zwei Regelgraphen* $L, R \in \mathcal{RG}[G_\Omega]$ *für die linke sowie die rechte Seite, die einen gemeinsamen, nicht-leeren Teilgraphen besitzen:*

$$\exists SL \leq L \colon SL \leq R \wedge SL \neq G_\emptyset.$$

Darüber hinaus sind Pfade nur Bestandteil der linken Seite einer Graphtransformationsregel, so dass $P_R = \emptyset$.

$TC := \langle tc_1, \ldots, tc_n \rangle \in \mathcal{TC}^*$ *ist eine Sequenz von Transformationsaufrufen, die auch leer sein kann.*

Die Menge aller Graphtransformationsregeln wird mit $\mathcal{R}$ bezeichnet.

Damit kann schließlich ein Transformation Pattern wie folgt definiert werden:

Definition 14 *Ein* Transformation Pattern *ist ein Tupel* $tp := (nip, IP, parent, first, next)$ *wobei* $nip \in \mathcal{R}$ *eine Graphtransformationsregel ist, die den nicht-iterierten Anteil beschreibt.* $IP \subseteq \mathcal{R}$ *ist eine Menge von Graphtransformationsregeln, die iterierte Anteile beschreiben.*

Die zyklenfreie Abbildung $parent : IP \rightarrow IP \cup \{nip\}$ *ordnet jedem iterierten Anteil den Anteil zu, in dem er enthalten ist.*

Die zyklenfreie Abbildung $first : IP \cup \{nip\} \rightarrow IP \cup \{\bot\}$ *ordnet jedem Anteil den ersten iterierten Anteil zu, der in ihm enthalten ist, oder* $\bot$ *falls es keinen solchen gibt, so dass gilt* $\forall rp \in IP \cup \{nip\}, first(rp) \neq \bot : ip = first(rp) \Rightarrow rp = parent(ip)$.

Die zyklenfreie Abbildung $next : IP \rightarrow IP \cup \{\bot\}$ *ordnet jedem iterierten Anteil den nächsten iterierten Anteil auf derselben Hierarchiestufe oder* $\bot$ *zu, falls es keinen solchen gibt, so dass gilt* $\forall ip \in IP : next(ip) \neq \bot \Rightarrow parent(ip) = parent(next(ip))$.

Die Menge aller Graphtransformationsregeln eines Transformation Pattern tp wird mit $rules(tp) := \{nip_{tp}\} \cup IP_{tp}$ bezeichnet.

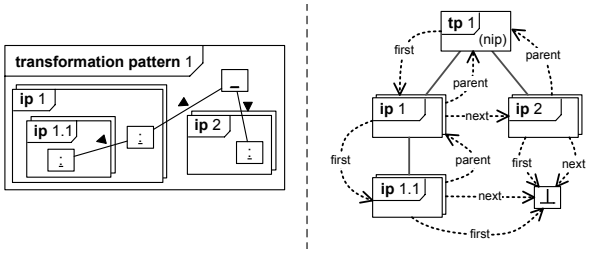

Abbildung 3.8: Schematische Darstellung eines Transformation Pattern

Abbildung 3.8 zeigt schematisch ein Transformation Pattern mit insgesamt drei iterierten Anteilen: links in konkreter Syntax, rechts als eine Baumstruktur. Die Wurzel der Baumstruktur repräsentiert seinen nicht-iterierten Anteil *nip*, seine beiden Kinder ip 1 und ip 2 sind iterierte Anteile auf derselben Hierarchiestufe. Der erste iterierte Anteil ip 1 enthält einen weiteren iterierten Anteil ip 1.1. Die gestrichelten Pfeile markieren die in einer formalen Darstellung des Transformation Pattern gemäß Definition 13 in den Abbildungen $first$, *next* und *parent* enthaltenen Tupel. Ein Pfeil zeigt dabei jeweils von einem Element der Definitionsmenge zu dem Element der Zielmenge, das ihm die als Pfeilbeschriftung angegebene Abbildung zuordnet.

3.3.5 Zustand

Die bisherigen Definitionen formalisieren die Syntax von Transformationsdiagrammen. Ihre Semantik wird im Folgenden operational definiert. Dazu wird zunächst ein Zustand während der Ausführung eines Transformationsdiagramms definiert.

Ein solcher Zustand besteht zum einen aus dem Wirtsgraphen zum jeweiligen Zeitpunkt der Ausführung. Ebenso werden die durch die bisherige Ausführung des Diagramms erzeugten Knoten und Kanten festgehalten. Schließlich wird festgehalten, welche diagrammweiten Knotenbezeichner an welche Knoten des Wirtsgraphen gebunden sind.

Definition 15 *Ein* Zustand $\sigma = (G, C^N, C^E, bound)$ *während der Ausführung eines Transformationsdiagramms TD besteht aus dem Wirtsgraphen $G \in \mathcal{G}[G_\Omega]$, der Menge der Knoten $C^N \subseteq N_G$, die durch das Diagramm bisher erzeugt wurden, der Menge der Kanten $C^E \subseteq E_G$, die durch das Diagramm bisher erzeugt wurden und $bound : V^N_{TD} \to N_G \cup \{\perp\}$, den Bindungen der Knotenbezeichner an Knoten des Wirtsgraphen oder $\perp$, falls es noch keine Bindung für einen Bezeichner gibt.*

Die Menge aller prinzipiell möglichen (nicht unbedingt erreichbaren) Zustände während der Ausführung eines Transformationsdiagramms TD wird mit Σ_{TD} bezeichnet.

3.3.6 Anwendungsstellen

Graphtransformationsregeln werden auf einen Wirtsgraphen angewendet. Dazu wird im Wirtsgraphen eine Anwendungsstelle (engl. *match*), ein zur linken Seite der Regel strukturgleicher (*isomorpher*) Teilgraph, bestimmt. Bei Anwendung der Regel wird dieser Teilgraph im Prinzip durch die rechte Regelseite ersetzt, indem Knoten und Kanten gelöscht und erzeugt werden.

Die Graphtransformationsregeln in dieser Arbeit enthalten zum einen Pfade. Zum anderen sind ihre Anwendungsstellen abhängig von den Anwendungsstellen zuvor ausgeführter Regeln desselben Transformationsdiagramms beziehungsweise vom im vorherigen Abschnitt definierten Zustand während der Ausführung. Daher müssen die zum Beispiel aus [Roz97] oder auch [Sch06] bekannten Definitionen von Anwendungsstellen hier angepasst werden.

Zunächst wird ein Graphhomomorphismus, der einen Regelgraphen mit Pfaden auf einen Graphen abbildet, aufbauend auf [Sch06] und erweitert um die Abbildung von Pfaden wie folgt definiert:

Definition 16 *Für einen Regelgraphen $RG \in \mathcal{RG}[G_\Omega]$ und einen Graphen $G \in \mathcal{G}[G_\Omega]$ wird ein* Graphhomomorphismus $m : RG \to G$, *der den Regelgraphen RG auf G abbildet, durch die vier Abbildungen $m := \langle m^N : N_{RG} \to N_G, m^E : E_{RG} \to E_G, m^{PN} : P_{RG} \to N^*_G, m^{PE} : P_{RG} \to E^*_G \rangle$ beschrieben.*

Dabei bilden m^N und m^E jeden Knoten und jede Kante von RG auf einen Knoten beziehungsweise eine Kante von G ab, wobei Start- und Zielknoten von Kanten sowie Typzuordnungen von Kanten erhalten bleiben:

$$\begin{aligned}\forall e \in E_{RG}\colon\ & m^N(src^E_{RG}(e)) = src^E_G(m^E(e)) \wedge \\ & m^N(tgt^E_{RG}(e)) = tgt^E_G(m^E(e)) \wedge \\ & l^E_G(m^E(e)) = l^E_{RG}(e).\end{aligned}$$

Knoten des Regelgraphen dürfen auch auf Knoten abgebildet werden, deren Typ ein Subtyp des jeweils abgebildeten Knotens ist:

$$\forall n \in N_{RG}\colon l^N_{RG}(n) \in types_\Omega(l^N_G(m^N(n))).$$

Die Funktionen m^{PN} und m^{PE} bilden jeden Pfad $p \in P_{RG}$ auf eine Sequenz von Knoten $\langle n_1, \ldots, n_k\rangle = m^{PN}(p)$ und Kanten $\langle e_1, \ldots, e_{k-1}\rangle = m^{PE}(p)$ in G ab, so dass gilt:

$$n_1 = m^N(src^P_{RG}(p)) \wedge n_k = m^N(tgt^P_{RG}(p)),$$

$$\forall i \in \{1, \ldots, k-1\}\colon src^E_G(e_i) = n_i \wedge tgt^E_G(e_i) = n_{i+1},$$

$$\forall i \in \{1, \ldots, k-1\}\colon (l^N_G(src^E_G(e_i)), l^E_G(e_i), l^N_G(tgt^E_G(e_i))) \in PTE^{inh}_\Omega \text{ und}$$

$$\forall i \in \{1, \ldots, k\}\colon types_\Omega(l^N_G(n_i)) \cap l^P_{RG}(p) = \emptyset.$$

Bei einem Graphhomomorphismus können mehrere Knoten oder Kanten des Regelgraphen auf denselben Knoten oder dieselbe Kante in einem Graphen abgebildet werden. Dies ist bei einem Graphisomorphismus nicht erlaubt:

Definition 17 *Für einen Regelgraphen RG und einen Graphen G ist ein* Graphisomorphismus *ein Graphhomomorphismus $m : RG \to G$, bei dem die Funktionen m^N und m^E bijektiv sind.*

Darauf aufbauend wird eine Anwendungsstelle für eine Graphtransformationsregel eines Transformationsdiagramms in einem Zustand während seiner Ausführung definiert als:

Definition 18 *Eine* Anwendungsstelle *(engl.* match*) für eine Graphtransformationsregel $L \rightarrow_r R$ eines Transformationsdiagramms TD in einem Zustand $\sigma \in \Sigma_{TD}$, geschrieben $m_{r,\sigma}$, ist ein Graphisomorphismus $m_{r,\sigma} : L \rightarrow SG$, der L auf einen Teilgraphen SG von G_σ ($SG \leq G_\sigma$) abbildet, so dass gilt:*

$$\forall n \in N_L \colon m^N_{r,\sigma}(n) \in C^N_\sigma \Rightarrow n \in N^B_L, \tag{3.4}$$

$$\forall e \in E_L \colon m^E_{r,\sigma}(e) \notin C^E_\sigma, \tag{3.5}$$

$$\begin{aligned}\forall p \in P_L, \langle n_1, \ldots, n_k\rangle = m^{PN}_{r,\sigma}(p), \langle e_1, \ldots, e_{k-1}\rangle = m^{PE}_{r,\sigma}(p) \colon \\ \forall i \in \{1, \ldots, k\} \colon n_i \notin C^N_\sigma \wedge \forall i \in \{1, \ldots, k-1\} \colon e_i \notin C^E_\sigma \text{ und}\end{aligned} \tag{3.6}$$

$$\forall n \in N^B_L \colon m^N_{r,\sigma}(n) = bound_\sigma(v^N_L(n)). \tag{3.7}$$

Dabei wird zum einen verboten, dass ein Knoten an einen durch die bisherige Ausführung des Diagramms erzeugten Knoten gebunden wird, es sei denn dies geschieht explizit durch einen gebundenen Knoten (3.4). Ebenso wird verhindert, dass Kanten an erzeugte Kanten gebunden werden (3.5) oder das ein Pfad erzeugte Knoten oder Kanten bindet (3.6). Schließlich wird sichergestellt, dass jeder als gebunden markierte Knoten erneut an den Knoten im Wirtsgraphen gebunden wird, an den er bereits zuvor gebunden wurde (3.7).

Somit werden alle in Abschnitt 3.2 beschriebenen Einschränkungen für Anwendungsstellen berücksichtigt. Die Menge aller Anwendungsstellen wird mit $\mathcal{M}$ bezeichnet.

3.3.7 Ausführung von Graphtransformationsregeln

Nach der Formalisierung einer Anwendungsstelle für eine Graphtransformationsregel wird nun definiert, wird sich die Anwendung einer Regel auf eine solche Anwendungsstelle auf den Wirtsgraphen und auf den Ausführungszustand eines Transformationsdiagramms auswirkt.

Gibt es eine Anwendungsstelle für eine Regel in einem Wirtsgraphen so werden durch die Anwendung alle Elemente der linken Regelseite, die nicht auch in der rechten Regelseite enthalten sind, einschließlich dabei entstehender loser Kanten (engl. *dangling edges*) gelöscht. Umgekehrt werden alle Elemente, die in der rechten Seite enthalten sind, nicht aber in der linken, erzeugt. Sollen dabei Kanten zwischen zwei Knoten erzeugt werden, so dass aufgrund bereits bestehender Kanten dadurch eine Kardinalitätsverletzung auftreten würde, werden

die bestehenden Kanten vorher ebenfalls gelöscht. Diese als direkte Transformation bezeichnete Anwendung entspricht prinzipiell dem *Single-Pushout-Approach* der Graphtransformation [Roz97] und wird hier definiert als:

Definition 19 *Die* direkte Transformation eines Graphen $G_\sigma \in \mathcal{G}[G_\Omega]$ *im Zustand* σ *in einen Graphen* $G' \in \mathcal{G}[G_\Omega]$ *durch eine Graphtransformationsregel* $L \rightarrow_r R$ *an einer Anwendungsstelle* $m_{r,\sigma}$ *ist definiert als ein Graphhomomorphismus* $o : L \cup R \rightarrow G_\sigma \cup G'$ *mit* $o|_L = m_{r,\sigma}$ *so dass gilt:*

- $o(L) \leq G_\sigma$ *und* $o(R) \leq G'$*, dass heißt, die linke Seite kann auf einen Teilgraphen von* G_σ *abgebildet werden und die rechte Seite auf einen Teilgraphen von* G' *und*

- $o(N_L \setminus N_R) = N_{G_\sigma} \setminus N_{G'}$*, dass heißt, die Anwendung der Regel löscht genau die Knoten aus* G_σ*, auf die Knoten von* L *aber nicht von* R *abgebildet werden, und*

- $o(E_L \setminus E_R)$

$$
\begin{aligned}
\cup \quad & \{e \in E_{G_\sigma} | src^E_{G_\sigma}(e) \in (N_{G_\sigma} \setminus N_{G'}) \vee tgt^E_{G_\sigma}(e) \in (N_{G_\sigma} \setminus N_{G'})\} \\
\cup \quad & \{e \in E_{G_\sigma} | (tgtCard_\Omega(l^E_{G_\sigma}(e)) = 0..1 \wedge \\
& \qquad \exists e' \in (E_R \setminus E_L), l^E_{G_\sigma}(e) = l^E_R(e') : \\
& \qquad\quad o(src^E_R(e')) = src^E_{G_\sigma}(e)) \quad \vee \\
& \qquad (srcCard_\Omega(l^E_{G_\sigma}(e)) = 0..1 \wedge \\
& \qquad \exists e' \in (E_R \setminus E_L), l^E_{G_\sigma}(e) = l^E_R(e') : \\
& \qquad\quad o(tgt^E_R(e')) = tgt^E_{G_\sigma}(e))\} \\
= \quad & E_{G_\sigma} \setminus E_{G'},
\end{aligned}
$$

  dass heißt, es werden die Kanten aus G_σ *gelöscht, auf die Kanten von* L *aber nicht von* R *abgebildet werden können, ebenso die Kanten, die mit einem gelöschten Knoten verbunden sind und ebenso die Kanten, die wie oben beschrieben zu einer Kardinalitätsverletzung führen würden, und*

- $o(N_R \setminus N_L) = N_{G'} \setminus N_{G_\sigma}$ *und* $o(E_R \setminus E_L) = E_{G'} \setminus E_{G_\sigma}$*, dass heißt, es werden genau die Elemente in* G' *erzeugt, auf die Elemente aus* R *aber nicht aus* L *abgebildet werden.*

Darüber hinaus wird G' *gegenüber* G_σ *nicht verändert.* o *wird* Auftreten *(engl.* occurrence*) genannt.*

Wird ein Graph G_σ durch eine Regel r an einer Anwendungsstelle $m_{r,\sigma}$ in einen Graphen G' transformiert, wird dies auch $G_\sigma \rightarrow_{m_{r,\sigma}} G'$ geschrieben.

Ein Zustand während der Ausführung eines Transformationsdiagramms wird dann durch eine direkte Transformation wie folgt in einen Folgezustand transformiert:

Definition 20 *Ein Zustand σ während der Ausführung eines Transformationsdiagramms TD wird durch eine direkte Transformation durch eine Graphtransformationsregel $L \rightarrow_r R$ an einer Anwendungsstelle $m_{r,\sigma}$ in einen Zustand $\sigma' := (G_{\sigma'}, C^N_{\sigma'}, C^E_{\sigma'}, bound_{\sigma'})$ transformiert, so dass gilt:*

$$\begin{aligned} &G_{\sigma'} \textit{ ist das Ergebnis der direkten Transformation} \\ &G_\sigma \rightarrow_{m_{r,\sigma}} G_{\sigma'} \textit{mit Auftreten } o, \end{aligned} \tag{3.8}$$

$$C^N_{\sigma'} := C^N_\sigma \cup o(N_R \setminus N_L), \tag{3.9}$$

$$C^E_{\sigma'} := C^E_\sigma \cup o(E_R \setminus E_L) \textit{ und} \tag{3.10}$$

$$bound_{\sigma'} := \lambda v \in V^N_{TD}. \begin{cases} o^N(n) & \textit{falls } \exists n \in N_L \cup N_R: \\ & v = v^N_L(n) \vee v = v^N_R(n), \\ bound_\sigma(v) & \textit{sonst.} \end{cases} \tag{3.11}$$

Der Wirtsgraph des Folgezustands wird durch direkte Transformation gemäß Definition 19 des Wirtsgraphen des Ausgangszustands ermittelt (3.8) und die dabei erzeugten Knoten (3.9) und Kanten (3.10) werden zusätzlich festgehalten. Zudem werden die gebundenen Knoten angepasst (3.11): falls ein Knoten bei der direkten Transformation (erneut oder erstmalig) gebunden wurde, wird der Knotenbezeichner nun auf diesen Knoten abgebildet, ansonsten wird die bisherige Abbildung beibehalten.

Die Transformation eines Zustands σ in einen Zustand σ' durch die direkte Transformation durch eine Graphtransformationsregel r an der Anwendungsstelle $m_{r,\sigma}$ wird auch geschrieben als $\sigma \rightarrow_{m_{r,\sigma}} \sigma'$.

Eine Graphtransformationsregel kann darüber hinaus eine Sequenz von Transformationsaufrufen enthalten, die es nach einer direkten Transformation auszuführen gilt. Durch die aufgerufenen Diagramme wird auch der Zustand während der Ausführung des aufrufenden Diagramms verändert. Um dies zu

formalisieren, wird nun eine Auswertungsrelation für Graphtransformationsregeln angegeben, die beschreibt, wie der Zustand schrittweise verändert wird. Dabei werden jeweils alle beobachtbaren Zwischenzustände zusammen mit den als nächstes auszuführenden syntaktischen Konstrukten oder *success* bei erfolgreicher Terminierung oder $failure$ bei einem Fehlschlag angegeben.

Bei der folgenden Definition der Auswertungsrelation für Graphtransformationsregeln wird die binäre Auswertungsrelation für ein Transformationsdiagramm $\triangleright_{TD}$ (siehe Definition 24) über $\left(\Sigma_{TD} \times (TP_{TD} \cup \{s^{\oplus}_{TD}, s^{\ominus}_{TD}, success, failure\})\right)$ benutzt, um den Aufruf eines Transformationsdiagramms auszuwerten. Der Aufruf eines Transformationsdiagramms TD ist erfolgreich, wenn die transitive Hülle seiner Auswertungsrelation $\triangleright^{+}_{TD}$ ein Tupel enthält, dessen rechte Seite $(\sigma, success)$ entspricht, wobei $\sigma \in \Sigma_{TD}$ dann der Zustand direkt nach der Ausführung des Diagramms ist. Enthält sie ein Tupel mit der rechten Seite $(\sigma, failure)$, ist die Ausführung fehlgeschlagen und $\sigma \in \Sigma_{TD}$ ist der Zustand direkt vor der Ausführung des Diagramms.

Definition 21 *Die* Auswertungsrelation für eine Graphtransformationsregel r *eines Transformation Pattern eines Transformationsdiagramms* TD *ist eine binäre Relation* $\triangleright_r$ *über* $(\Sigma_{TD} \times (\mathcal{M} \cup TC_r \cup \{success, failure\}))$. *Sie ist durch die folgenden Regeln definiert, wobei* $m_{r,\sigma}$ *eine Anwendungsstelle für* r *in Zustand* σ *und* $tc_i \in TC_r$ *ein Transformationsaufruf von* r *sind:*

$$\frac{\sigma \rightarrow_{m_{r,\sigma}} \sigma' \wedge TC_r = \langle \emptyset \rangle}{(\sigma, m_{r,\sigma}) \triangleright_r (\sigma', success)}, \tag{3.12}$$

$$\frac{\sigma \rightarrow_{m_{r,\sigma}} \sigma' \wedge TC_r = \langle tc_1, \ldots, tc_n \rangle}{(\sigma, m_{r,\sigma}) \triangleright_r (\sigma', tc_1)}, \tag{3.13}$$

$$\frac{\begin{array}{c}((G_\sigma, \emptyset, \emptyset, before_{\sigma,tc_i}), \lambda_{TD_{tc_i}}) \triangleright^{+}_{TD_{tc_i}} (\sigma', success) \wedge \\ TC_r = \langle tc_1, \ldots, tc_n \rangle \wedge tc_i = tc_n\end{array}}{(\sigma, tc_i) \triangleright_r ((G_{\sigma'}, C^N_\sigma, C^E_\sigma, after_{\sigma,\sigma',tc_i}), success)}, \tag{3.14}$$

$$\frac{\begin{array}{c}((G_\sigma, \emptyset, \emptyset, before_{\sigma,tc_i}), \lambda_{TD_{tc_i}}) \triangleright^{+}_{TD_{tc_i}} (\sigma', success) \wedge \\ TC_r = \langle tc_1, \ldots, tc_i, tc_{i+1}, \ldots, tc_n \rangle\end{array}}{(\sigma, tc_i) \triangleright_r ((G_{\sigma'}, C^N_\sigma, C^E_\sigma, after_{\sigma,\sigma',tc_i}), tc_{i+1})}, \tag{3.15}$$

$$\frac{((G_\sigma, \emptyset, \emptyset, before_{\sigma,tc_i}), \lambda_{TD_{tc_i}}) \triangleright^+_{TD_{tc_i}} (\sigma', failure)}{(\sigma, tc_i) \triangleright_r (\sigma, failure)}, \tag{3.16}$$

wobei jeweils

$$before_{\sigma,tc_i}(v) := \begin{cases} bound_\sigma(args_{tc_i}(v)) & falls\ v \in V^P_{TD_{tc_i}}, \\ \bot & sonst \end{cases} \quad und$$

$$after_{\sigma,\sigma',tc_i}(v) := \begin{cases} bound_{\sigma'}(w) & falls\ \exists w \in V^R_{TD_{tc_i}} : v = res_{tc_i}(w), \\ bound_\sigma(v) & sonst \end{cases}.$$

Die Auswertung eine Graphtransformationsregel r beginnt mit der Transformation des Zustands σ in σ' durch direkte Transformation an der gegebenen Anwendungsstelle $m_{r,\sigma} \in \mathcal{M}$. Enthält r keine Transformationsaufrufe, ist die Auswertung erfolgreich beendet (Regel 3.12). Wenn r dagegen einen oder mehrere Transformationsaufrufe enthält, wird stattdessen der erste Aufruf als nächstes ausgeführt (Regel 3.13).

Die erfolgreiche Ausführung eines Transformationsaufrufs tc_i wird durch die Regeln 3.14 und 3.15 beschrieben. Die erste Regel beschreibt den Fall, dass es nach $tc_i \in TC_r = \langle tc_1, \ldots, tc_n \rangle$ keinen weiteren Transformationsaufruf gibt $(tc_i = tc_n)$ und die Auswertung von r deshalb erfolgreich endet. Die zweite Regel beschreibt den Fall, dass es einen weiteren Transformationsaufruf $tc_{i+1} \in TC_r$ gibt, der als nächstes auszuführen ist.

In beiden Regeln wird die Auswertung des Transformationsaufrufs tc_i mit Hilfe der Auswertungsrelation für das jeweils aufgerufene Transformationsdiagramm TD_{tc_i} vorgenommen. Dabei beginnt die Ausführung des aufgerufenen Diagramms mit seinem ersten Transformation Pattern $\lambda_{TD_{tc_i}}$. Der Zustand σ_{tc_i}, in dem die Ausführung des Diagramms beginnt, wird aus dem Zustand des Aufrufers σ ermittelt: die Ausführung von TD_{tc_i} beginnt mit demselben Wirtsgraphen G_σ, die beiden Mengen zur Speicherung der erzeugten Knoten und Kanten sind zu Beginn leer und die Parameter von TD_{tc_i} müssen an Argumente – an Knoten des Wirtsgraphen – gebunden sein. Daher wird $before_{\sigma,tc_i}$ nur für die Parameter von TD_{tc_i} definiert und diese werden an die Argumente des Transformationsaufrufs aus σ gebunden.

Aus dem Zustand σ' nach der erfolgreichen Ausführung von TD_{tc_i} wird der nächste Zustand des Aufrufers abgeleitet, indem der Wirtsgraph $G_{\sigma'}$ nach Ausführung von TD_{tc_i} übernommen wird, die Mengen der erzeugten Knoten und Kanten werden unverändert aus dem Zustand σ des Aufrufers vor dem

Aufruf übernommen und die Bindungen der Knotenbezeichner werden durch $after_{\sigma,\sigma',tc_i}$ um die Bindungen für Rückgabewerte erweitert.

Schlägt die Ausführung eines Transformationsaufrufs beziehungsweise des aufgerufenen Transformationsdiagramms fehl, schlägt die Ausführung der Graphtransformationsregel insgesamt fehl und der Zustand wird nicht weiter verändert (Regel 3.16).

3.3.8 Ausführung von Transformation Pattern

Aufbauend auf die Auswertung von Graphtransformationsregeln kann nun die Auswertung von Transformation Pattern definiert werden. Bei einem Transformation Pattern wird als erstes die Graphtransformationsregel für seinen nicht-iterierten Anteil einmalig ausgeführt. Ist dies erfolgreich, werden nacheinander die Graphtransformationsregeln für seine iterierten Anteile auf allen existierenden Anwendungsstellen angewendet. Dabei muss berücksichtigt werden, dass jede Anwendungsstelle eines iterierten Anteils relativ zur Anwendungsstelle seines Vateranteils nur einmal bearbeitet wird. Daher werden während der Auswertung bearbeitete Anwendungsstellen zusätzlich zum Zustand und zur als nächstes auszuführenden Graphtransformationsregel festgehalten.

Zwei Anwendungsstellen für eine Graphtransformationsregel sind verschieden, wenn sie mindestens einen Knoten oder eine Kante der linken Regelseite auf unterschiedliche Elemente im Wirtsgraphen abbilden:

Definition 22 *Zwei Anwendungsstellen m_1 und m_2 für dieselbe Graphtransformationsregel $L \to_r R$ in beliebigen Zuständen sind voneinander verschieden, geschrieben $m_1 \neq m_2$, wenn:*

$$\exists n \in N_L \colon m_1^N(n) \neq m_2^N(n) \vee \exists e \in E_L \colon m_1^E(e) \neq m_2^E(e).$$

Definition 23 *Die* Auswertungsrelation für ein Transformation Pattern *tp eines Transformationsdiagramms TD ist eine binäre Relation $\triangleright_{tp}$ über*

$$\begin{aligned}(\Sigma_{TD} \times (rules(tp) \cup (rules(tp) \times \mathcal{M}) \cup \{success, failure\}) \\ \times [rules(tp) \rightharpoonup \wp(\mathcal{M})]).\end{aligned}$$

Sie ist durch die folgenden Regeln definiert:

$$\frac{r = nip_{tp} \wedge \nexists m_{r,\sigma}}{(\sigma, r, ms) \triangleright_{tp} (\sigma, failure, ms)}, \tag{3.17}$$

$$\frac{r = nip_{tp} \wedge \exists m_{r,\sigma}}{(\sigma, r, ms) \triangleright_{tp} (\sigma, (r, m_{r,\sigma}), ms)}, \tag{3.18}$$

$$\frac{r = nip_{tp} \wedge (\sigma, m_{r,\sigma}) \triangleright_r^+ (\sigma', failure)}{(\sigma, (r, m_{r,\sigma}), ms) \triangleright_{tp} (\sigma, failure, ms)}, \tag{3.19}$$

$$\frac{r = nip_{tp} \wedge (\sigma, m_{r,\sigma}) \triangleright_r^+ (\sigma', success) \wedge first_{tp}(r) = \bot}{(\sigma, (r, m_{r,\sigma}), ms) \triangleright_{tp} (\sigma', success, ms)}, \tag{3.20}$$

$$\frac{r = nip_{tp} \wedge (\sigma, m_{r,\sigma}) \triangleright_r^+ (\sigma', success) \wedge first_{tp}(r) \neq \bot}{(\sigma, (r, m_{r,\sigma}), ms) \triangleright_{tp} (\sigma', first_{tp}(r), ms)}, \tag{3.21}$$

$$\frac{r \in IP_{tp} \wedge (\exists m_{r,\sigma} \colon \forall m' \in ms(r) \colon m_{r,\sigma} \neq m')}{(\sigma, r, ms) \triangleright_{tp} (\sigma, (r, m_{r,\sigma}), ms)}, \tag{3.22}$$

$$\frac{r \in IP_{tp} \wedge (\nexists m_{r,\sigma} \colon \forall m' \in ms(r) \colon m_{r,\sigma} \neq m') \wedge next_{tp}(r) \neq \bot}{(\sigma, r, ms) \triangleright_{tp} (\sigma, next_{tp}(r), ms[\emptyset/r])}, \tag{3.23}$$

$$\frac{\begin{array}{c} r \in IP_{tp} \wedge (\nexists m_{r,\sigma} \colon \forall m' \in ms(r) \colon m_{r,\sigma} \neq m') \wedge \\ next_{tp}(r) = \bot \wedge parent_{tp}(r) \in IP_{tp} \end{array}}{(\sigma, r, ms) \triangleright_{tp} (\sigma, parent_{tp}(r), ms[\emptyset/r])}, \tag{3.24}$$

$$\frac{\begin{array}{c} r \in IP_{tp} \wedge (\nexists m_{r,\sigma} \colon \forall m' \in ms(r) \colon m_{r,\sigma} \neq m') \wedge \\ next_{tp}(r) = \bot \wedge parent_{tp}(r) = nip_{tp} \end{array}}{(\sigma, r, ms) \triangleright_{tp} (\sigma, success, ms)}, \tag{3.25}$$

$$\frac{r \in IP_{tp} \wedge (\sigma, m_{r,\sigma}) \triangleright_r^+ (\sigma', failure)}{(\sigma, (r, m_{r,\sigma}), ms) \triangleright_{tp} (\sigma, failure, ms)}, \tag{3.26}$$

$$\frac{r \in IP_{tp} \wedge (\sigma, m_{r,\sigma}) \triangleright_r^+ (\sigma', success) \wedge first_{tp}(r) = \bot}{(\sigma, (r, m_{r,\sigma}), ms) \triangleright_{tp} (\sigma', r, ms[(ms(r) \cup \{m_{r,\sigma}\})/r])}, \tag{3.27}$$

$$\frac{r \in IP_{tp} \wedge (\sigma, m_{r,\sigma}) \triangleright_r^+ (\sigma', success) \wedge first_{tp}(r) \neq \bot}{(\sigma, (r, m_{r,\sigma}), ms) \triangleright_{tp} (\sigma', first_{tp}(r), ms[(ms(r) \cup \{m_{r,\sigma}\})/r])}. \tag{3.28}$$

Dabei ist für eine Funktion $f\colon X \to V$ und $y \in X, v \in V$

$$f[v/y](x) := \begin{cases} v & \text{falls } x = y, \\ f(x) & \textit{sonst}. \end{cases}$$

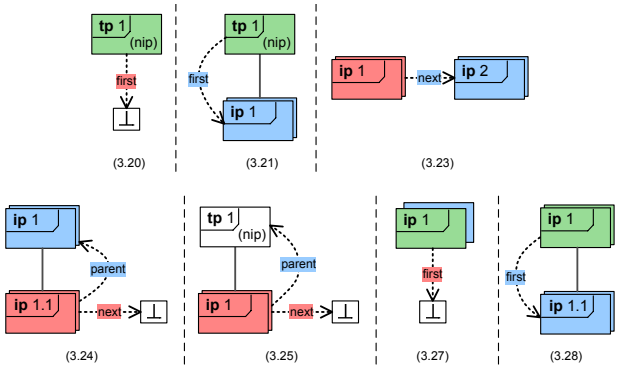

Abbildung 3.9: Veranschaulichung einiger Regeln der Auswertungsrelation für Transformation Pattern

Einige der Regeln, die im Folgenden noch erklärt werden, werden in Abbildung 3.9 anhand schematisch dargestellter Beispiele von Transformation Pattern veranschaulicht. Die Transformation Pattern werden analog zu Abbildung 3.8 dargestellt, hier werden aber ausschließlich die Elemente, die für die jeweilige Regel relevant sind, gezeigt. Ein nicht-iterierter Anteil ist mit tp und (nip) beschriftet, ein iterierter Anteil ist durch zwei überlagerte Rechtecke dargestellt und mit ip beschriftet. Der zuletzt erfolgreich ausgeführte Anteil ist jeweils grün markiert. Gab es für den zuletzt auszuführenden Anteil keine Anwendungsstelle ist er rot markiert. Der durch die jeweilige Regel bestimmte als nächstes auszuführende Anteil ist falls vorhanden blau gefärbt. Die Verbindungen, die zum als nächstes auszuführenden Anteil führen, sind ebenfalls blau gefärbt. Verbindungen, die zur Bestimmung des nächsten Anteils beitragen, da sie zu $\bot$ führen, sind rot markiert.

Die Ausführung eines Transformation Pattern tp beginnt damit, dass die Graphtransformationsregel für seinen nicht-iterierten Anteil nip_{tp} ausgeführt wird. Dies wird durch die ersten fünf Regeln beschrieben, in denen jeweils $r = nip_{tp}$ gilt.

Gibt es für r $(= nip_{tp})$ keine Anwendungsstelle, schlägt die Ausführung von tp unmittelbar fehl und der Zustand wird nicht verändert (Regel 3.17).

Gibt es eine Anwendungsstelle $m_{r,\sigma}$ so ist nip_{tp} als nächstes auf dieser Anwendungsstelle auszuführen (Regel 3.18). Durch $\exists m_{r,\sigma}$ wird dabei ausgedrückt, dass es mindestens eine Anwendungsstelle für die Graphtransformationsregel r in Zustand σ gibt und dass eine solche bestimmt wird. Wenn es mehrere Anwendungsstellen gibt, wird eine davon (erratisch) nicht-deterministisch ausgewählt.

Die nächsten drei Regeln beschreiben die eigentliche Ausführung von nip_{tp} auf einer Anwendungsstelle $m_{r,\sigma}$. Diese kann fehlschlagen zum Beispiel aufgrund eines fehlschlagenden Transformationsaufrufs. In diesem Fall schlägt die Ausführung von tp insgesamt fehl und wird beendet (Regel 3.19).

Ist die Anwendung von nip_{tp} erfolgreich, wird dadurch der Ausgangszustand σ in σ' transformiert. Enthält r $(= nip_{tp})$ keinen iterierten Anteil $(first_{tp}(r) = \bot)$ ist die Auswertung damit erfolgreich beendet (Regel 3.20). Enthält r $(= nip_{tp})$ einen oder mehrere iterierte Anteile, ist die Regel für den ersten $first_{tp}(r)$ (mit dem niedrigsten Ausführungsrang) als nächstes in Zustand σ' auszuführen (Regel 3.21).

Die weiteren sieben Regeln behandeln die Ausführung von iterierten Anteilen. Ist eine Regel für einen iterierten Anteil r als nächstes auszuführen, muss überprüft werden, ob es eine noch nicht bearbeitete Anwendungsstelle für sie gibt. Dass eine Anwendungsstelle noch nicht bearbeitet wurde, wird mit Hilfe der Funktion $ms : rules(tp) \rightharpoonup \wp(\mathcal{M})$ ermittelt, die genutzt wird, um für eine Graphtransformationsregel eine Menge von bearbeiteten Anwendungsstellen festzuhalten. Eine neue Anwendungsstelle $m_{r,\sigma}$ für eine Regel r muss von allen bereits in $ms(r)$ hinterlegten Anwendungsstellen verschieden sein. Gibt es eine solche Anwendungsstelle, so ist r als nächstes auf ihr auszuführen (Regel 3.22). Mit $\exists m_{r,\sigma}$ wird hier erneut die nicht-deterministische Bestimmung einer von möglicherweise mehreren Anwendungsstellen notiert.

Gibt es keine noch unbearbeitete Anwendungsstelle (mehr) für r, wird als nächstes der iterierte Anteil auf derselben Hierarchiestufe mit dem nächst höheren Ausführungsrang $next_{tp}(r)$ in demselben Zustand ausgeführt (Regel 3.23). Dabei werden alle für r hinterlegten bearbeiteten Anwendungsstellen aus ms entfernt. Auf diese Weise wird sichergestellt, dass wenn r erneut ausgeführt werden sollte (was nur geschieht, wenn ihr Vateranteil erneut ausgeführt wird) wieder alle existierenden Anwendungsstellen bearbeitet werden.

Gibt es keinen nächsten iterierten Anteil auf derselben Hierarchiestufe mehr, so wird als nächstes erneut die Regel für den Vateranteil $parent_{tp}(r)$ von r ausgeführt, falls es sich dabei ebenfalls um einen iterierten Anteil handelt (Regel 3.24). Dabei werden ebenfalls die durch r bearbeiteten Anwendungsstellen gelöscht. Ist der Vateranteil der nicht-iterierte Anteil nip_{tp}, ist die Ausführung

des Transformation Pattern erfolgreich beendet (Regel 3.25).

Die letzten drei Regeln beschreiben die eigentliche Ausführung der Regel r eines iterierten Anteils auf einer Anwendungsstelle $m_{r,\sigma}$. Schlägt diese (aufgrund eines fehlschlagenden Transformationsaufrufs) fehl, schlägt die Anwendung des Transformation Pattern insgesamt fehl ohne den Zustand weiter zu verändern (Regel 3.26). Ist ihre Anwendung erfolgreich, wird der Ausgangszustand σ in σ' transformiert. Enthält der iterierte Anteil r keinen geschachtelten iterierten Anteil ($first_{tp}(r) = \bot$), so ist r erneut als nächstes in Zustand σ' auszuführen, wobei die aktuell bearbeitete Anwendungsstelle in $ms(r)$ hinterlegt wird (Regel 3.27). Enthält r einen oder mehrere iterierte Anteile, so ist die Regel für den ersten $first_{tp}(r)$ davon als nächstes in Zustand σ' auszuführen. Auch in diesem Fall wird die zuletzt bearbeitete Anwendungsstelle in $ms(r)$ hinterlegt (Regel 3.28).

3.3.9 Ausführung von Transformationsdiagrammen

Mit Hilfe der Auswertungsrelation für Transformation Pattern wird schließlich die Auswertungsrelation für Transformationsdiagramme definiert.

Definition 24 *Die* Auswertungsrelation $\triangleright_{TD}$ für ein Transformationsdiagramm TD *ist eine binäre Relation über* $\big(\Sigma_{TD} \times (TP_{TD} \cup \{s^{\oplus}_{TD}, s^{\ominus}_{TD}, success, failure\})\big)$. *Sie ist durch die folgenden Regeln definiert, wobei* $ms_{\emptyset}(r) := \emptyset$ *und* σ_0 *den Zustand vor Ausführung des Diagramms bezeichnet, in dem alle Parameter von* TD *gebunden sind* ($\forall v \in V^P_{TD} \colon bound_{\sigma_0}(v) \neq \bot$):

$$\frac{(\sigma, nip_{tp}, ms_{\emptyset}) \triangleright^{+}_{tp} (\sigma', success, ms')}{(\sigma, tp) \triangleright_{TD} (\sigma', succ_{TD}(tp))}, \tag{3.29}$$

$$\frac{(\sigma, nip_{tp}, ms_{\emptyset}) \triangleright^{+}_{tp} (\sigma', failure, ms')}{(\sigma, tp) \triangleright_{TD} (\sigma, fail_{TD}(tp))}, \tag{3.30}$$

$$\frac{}{(\sigma, s^{\oplus}_{TD}) \triangleright_{TD} (\sigma, success)}, \tag{3.31}$$

$$\frac{}{(\sigma, s^{\ominus}_{TD}) \triangleright_{TD} (\sigma_0, failure)}. \tag{3.32}$$

Bei der Ausführung eines Transformationsdiagramms TD werden hintereinander Transformation Pattern ausgeführt bis eine Stopaktivität erreicht wird.

Ist die Ausführung eines Transformation Pattern tp erfolgreich, wird dadurch der Ausgangszustand σ in σ' transformiert und als nächstes ist das Transformation Pattern oder die Stopaktivität in σ' auszuführen, die mit tp über seine ausgehende success-Transition verbunden ist (Regel 3.29). Schlägt die Ausführung von tp dagegen fehl, bleibt der Zustand unverändert und als nächstes ist das Transformation Pattern oder die Stopaktivität auszuführen, die mit tp über seine ausgehende failure-Transition verbunden ist (Regel 3.30).

Wird die erfolgreiche Stopaktivität $s^{\oplus}_{TD}$ erreicht, ist die Ausführung von TD erfolgreich beendet (Regel 3.31). Wird dagegen die Stopaktivität für einen Fehlschlag $s^{\ominus}_{TD}$ erreicht, ist die Ausführung von TD fehlgeschlagen und der Zustand σ_0 vor Ausführung von TD wird wiederhergestellt (Regel 3.32).

3.3.10 Strukturmusterannotationen

Jede durch die strukturbasierte Mustererkennung (siehe Abschnitt 2.3) erkannte Instanz eines Strukturmusters wird durch eine typisierte Annotation im abstrakten Syntaxgraphen markiert.

Auf Basis der Spezifikation von Strukturmustern sind die Annotationstypen bekannt und werden einem Typgraphen G_Ω hinzugefügt und die typisierten Links, mit denen bestimmte Objekte des abstrakten Syntaxgraphen gekennzeichnet werden, werden zu Assoziationen zwischen dem Annotationstyp und den verbundenen Typen in G_Ω. Damit können Strukturmusterannotationen prinzipiell ebenso behandelt werden wie herkömmliche Knoten von typisierten Graphen und Regelgraphen.

In Abschnitt 3.2.4 werden zwei Fälle beim Binden einer Annotation durch ein Transformation Pattern unterschieden: im ersten Fall wird beim Binden auch gefordert, dass die durch das zugehörige Strukturmuster spezifizierte Struktur im Wirtsgraphen vorhanden ist, im zweiten Fall reicht es aus, wenn nur die Annotation mit den ebenfalls spezifizierten Links vorhanden ist. Die bisherige Formalisierung unterstützt nur den zweiten Fall. Das in Kapitel 4 beschriebene Verifikationsverfahren setzt ebenfalls nur diesen Fall voraus, weshalb hier von einer Erweiterung der Formalisierung abgesehen wird. Von der in Kapitel 5 beschriebenen Werkzeugunterstützung werden dagegen beide Fälle unterstützt.

Die ebenfalls in Abschnitt 3.2.4 beschriebene Erzeugung von Strukturmustern wird auf den Aufruf eines Transformationsdiagramms abgebildet, dass zum einen die Struktur des Musters und zum anderen eine Annotation erzeugt und mit den entsprechenden Objekten verbindet. Das Transformations-

diagramm liefert diese Annotation als einen zusätzlichen Rückgabewert zurück. Damit deckt die bisherige Formalisierung auch Mustererzeugungen ab.

3.4 Synthese von Transformationsspezifikationen anhand von Quelltextbeispielen

Die Spezifikation von Programmtransformationen geschieht in diesem Ansatz auf Basis der abstrakten Syntax eines Programms beziehungsweise einer Programmiersprache, repräsentiert als Graph. In der konkreten Syntax eines Programms sind die Informationen über die Beziehungen der einzelnen Elemente zueinander in ihrer Anordnung enthalten oder ausgeblendet. Im abstrakten Syntaxgraph werden diese Beziehungen explizit repräsentiert, wodurch sie gezielt modifizierbar werden. Auf diese Weise stehen die für Transformationen notwendigen Ausdrucksmöglichkeiten zur Verfügung.

Dies bedingt aber auch, dass die Darstellung eines Programms in konkreter Syntax (in Textform) in der Regel kompakter ist als die Darstellung als abstrakter Syntaxgraph. Wenige Zeilen Quelltext können einem umfangreichen Graphen entsprechen. Zudem ist ein Re-Engineer typischerweise weitaus vertrauter mit der konkreten Syntax einer Programmiersprache als mit ihrer abstrakten Syntax. Aus diesen Gründen wird hier ein Ansatz vorgestellt, mit dem die Spezifikation von Transformationen vereinfacht werden soll.

Der Ansatz basiert darauf, dass der Re-Engineer bei der Spezifikation von Transformationen häufig von Quelltextausschnitten in konkreter Syntax ausgeht, sei es in seiner Vorstellung oder weil die Quelltextausschnitte tatsächlich vorliegen. Um eine Transformation zu spezifizieren, muss der Re-Engineer den relevanten Quelltextausschnitt bisher manuell in die Spezifikation als abstrakter Syntaxgraph in einem Transformation Pattern überführen. Danach kann er die gewünschten Modifikationen im Transformation Pattern definieren. Abbildung 3.10 zeigt dies anhand der Erzeugung einer lesenden Zugriffsmethode.

Im oberen Teil der Abbildung werden zwei Quelltextausschnitte gezeigt. Der linke Ausschnitt enthält eine Klassendefinition mit einem privaten Attribut. Der rechte Ausschnitt entspricht dem linken und enthält zusätzlich eine lesende Zugriffsmethode für das Attribut. Der Teil des Quelltextes, der gegenüber dem linken Ausschnitt hinzugefügt wurde, ist grau hinterlegt. Das Transformation Pattern im unteren Teil der Abbildung beschreibt die Transformation des linken Quelltextausschnitts in den rechten – das Hinzufügen einer lesenden Zugriffsmethode – in verallgemeinerter Form. Die linke Seite des Transformation

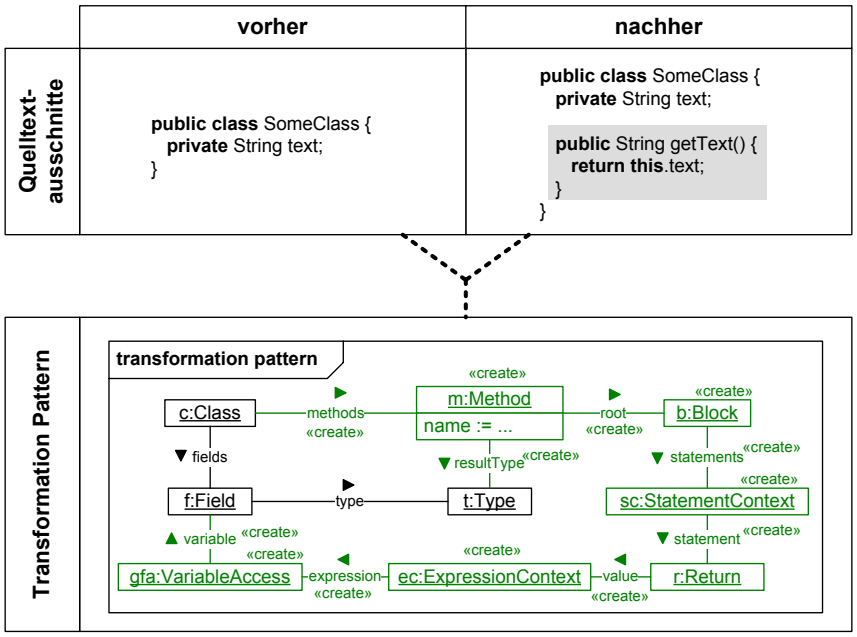

Abbildung 3.10: Erzeugen einer lesenden Zugriffsmethode – im Quelltext und als Transformation Pattern

Pattern ist isomorph zu einem Teilgraphen des abstrakten Syntaxgraphen des linken Quelltextausschnitts. Die linke Seite ist lediglich weniger spezifisch in Bezug auf Namen und Sichtbarkeiten. Die rechte Regelseite enthält zusätzlich einen erzeugten Teilgraphen, der isomorph zum abstrakten Syntaxgraphen für die im rechten Quelltextausschnitt hinzugefügten Zeilen ist.

Die Idee des hier vorgestellten Ansatzes ist, das Transformation Pattern im unteren Teil der Abbildung weitgehend automatisiert aus den Quelltextausschnitten im oberen Teil zu synthetisieren. Allgemein gibt der Re-Engineer zunächst ein Quelltextbeispiel an, das die Ausgangssituation darstellt. In diesem Beispiel nimmt er dann die gewünschten Modifikationen vor, so dass ein Quelltextbeispiel für die Endsituation entsteht. Aus den Beispielen wird automatisch ein Transformation Pattern generiert, das exakt die Ausgangssituation in die Endsituation transformiert. Dieses Transformation Pattern muss dann zum Teil manuell nachbearbeitet und verallgemeinert werden. Danach kann es in umfangreichere Transformationsdiagramme eingebettet werden.

In dem gezeigten Beispiel ist die Angabe der Transformation in Form von Quelltextbeispielen kompakter und einfacher möglich als in Form eines Transformation Pattern. Insbesondere wenn ein Quelltextbeispiel für die Ausgangs-

situation bereits vorliegt, kann es deutlich bequemer sein, eine benötigte Transformation durch direkte Modifikation des Quelltextes anzugeben. Wenn bei der Analyse eines Programms Instanzen eines Muster erkannt werden, die alle auf dieselbe Weise verändert werden sollen, es die dazu benötigte Transformation aber noch nicht gibt, kann diese durch Modifikation einer Instanz im Quelltext ermittelt und dann für alle anderen Instanzen wiederverwendet werden.

In [Wag09] wird ein ähnlicher Ansatz im Zusammenhang mit der Spezifikation von Modelltransformationen zur Übersetzung und Konsistenzerhaltung von Modellen in unterschiedlichen Sprachen, dass heißt mit unterschiedlichen Metamodellen, verfolgt. Dabei werden eine Menge paarweise zueinander konsistenter Beispielsituationen jeweils in der konkreten Syntax der ineinander zu übersetzenden Sprachen angegeben, um daraus automatisiert Triple-Graph-Grammatik-Regeln zu synthetisieren. Im Unterschied dazu werden hier Transformation Pattern synthetisiert, die nur ein Modell transformieren, wozu Triple-Graph-Grammatik-Regeln nicht so gut geeignet sind, da sie dabei eine Kopie des Modells anfertigen würden.

Im Folgenden wird zunächst das Verfahren zur Synthese von Transformation Pattern genauer beschrieben. Danach wird gezeigt, wie dieses Verfahren eingesetzt werden kann, um umfangreiche Transformationen anhand von Beispielen zu spezifizieren.

3.4.1 Syntheseverfahren

Abbildung 3.11 gibt einen Überblick über das Syntheseverfahren. Die Ausgangsbasis bilden zwei Quelltextbeispiele – eines für die Ausgangssituation (Vorherbeispiel) und eines für die Endsituation (Nachherbeispiel), die sich durch Änderung der Ausgangssituation ergibt. Da die Änderungen direkt im Quelltext vorgenommen werden, handelt es sich bei den Beispielen um zwei Versionen desselben Quelltextausschnitts, die in zwei getrennte abstrakte Syntaxgraphen überführt werden. Abbildung 3.12 zeigt dies anhand des Beispiels der lesenden Zugriffsmethode. Der Quelltext der Beispiele (erste Zeile der Abbildung) wird jeweils in die Repräsentation als abstrakter Syntaxgraph (zweite Zeile) überführt.

In einem ersten Schritt gilt es nun, die vorgenommenen Änderungen nicht textuell, sondern auf Ebene der beiden abstrakten Syntaxgraphen zu identifizieren. Die Endsituation entsteht durch Veränderungen aus der Ausgangssituation heraus. Dabei werden in der Regel nicht alle Elemente der Ausgangssituation gelöscht, so dass beide Situationen beziehungsweise ihre abstrakten Syntaxgraphen einander entsprechende Elemente enthalten. Im Beispiel in Ab-

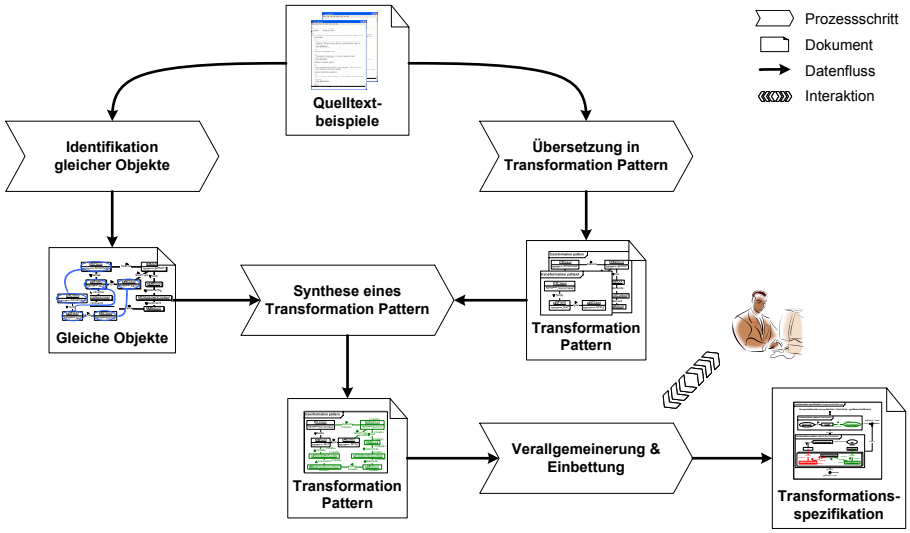

Abbildung 3.11: Syntheseverfahren

bildung 3.12 bleiben die Klasse und das Attribut erhalten und werden um eine Zugriffsmethode ergänzt. Daher repräsentieren die Objekte o1:Class im Vorherbeispiel und o1:Class im Nachherbeispiel dieselbe Klasse. Ebenso repräsentieren die Objekte o2:Field und o3:Field sowie o3:Class und o4:Class dieselben Elemente. Entsprechendes gilt für die jeweiligen Links zwischen diesen Objekten. Ziel ist es, die einander entsprechenden – gleichen – Objekte zu identifizieren, um von diesen dann auf die vorgenommenen Modifikationen schließen zu können.

Im Beispiel werden nur Elemente hinzugefügt. Genauso können Elemente gelöscht werden, was zum Beispiel beim Verschieben von Anweisungen geschieht. Die beiden abstrakten Syntaxgraphen entstehen unabhängig voneinander durch das Parsen von Quelltext, so dass es keine eindeutigen gleichbleibenden Objektbezeichner gibt. Gleiche Objekte können nur anhand ihrer Eigenschaften und der Verbindungen zu anderen Objekten identifiziert werden. Bei den meisten Objekten, die Bestandteil eines Methodenrumpfes sind, gibt es nur sehr wenige Eigenschaften, so dass den Verbindungen zu anderen Objekten eine große Bedeutung zukommt.

Dem Problem des Vergleichs von Bäumen, Graphen sowie Programmen zur Ermittlung von Unterschieden beziehungsweise gleichen Elementen widmen sich eine Reihe von Ansätzen mit unterschiedlichen Annahmen und Rahmenbedingungen wie zum Beispiel [AOH04, CRGMW96, Hor90, JL94, LS92, MGMR02, XS05]. Der in [WK06] beschriebene Ansatz erscheint am besten für

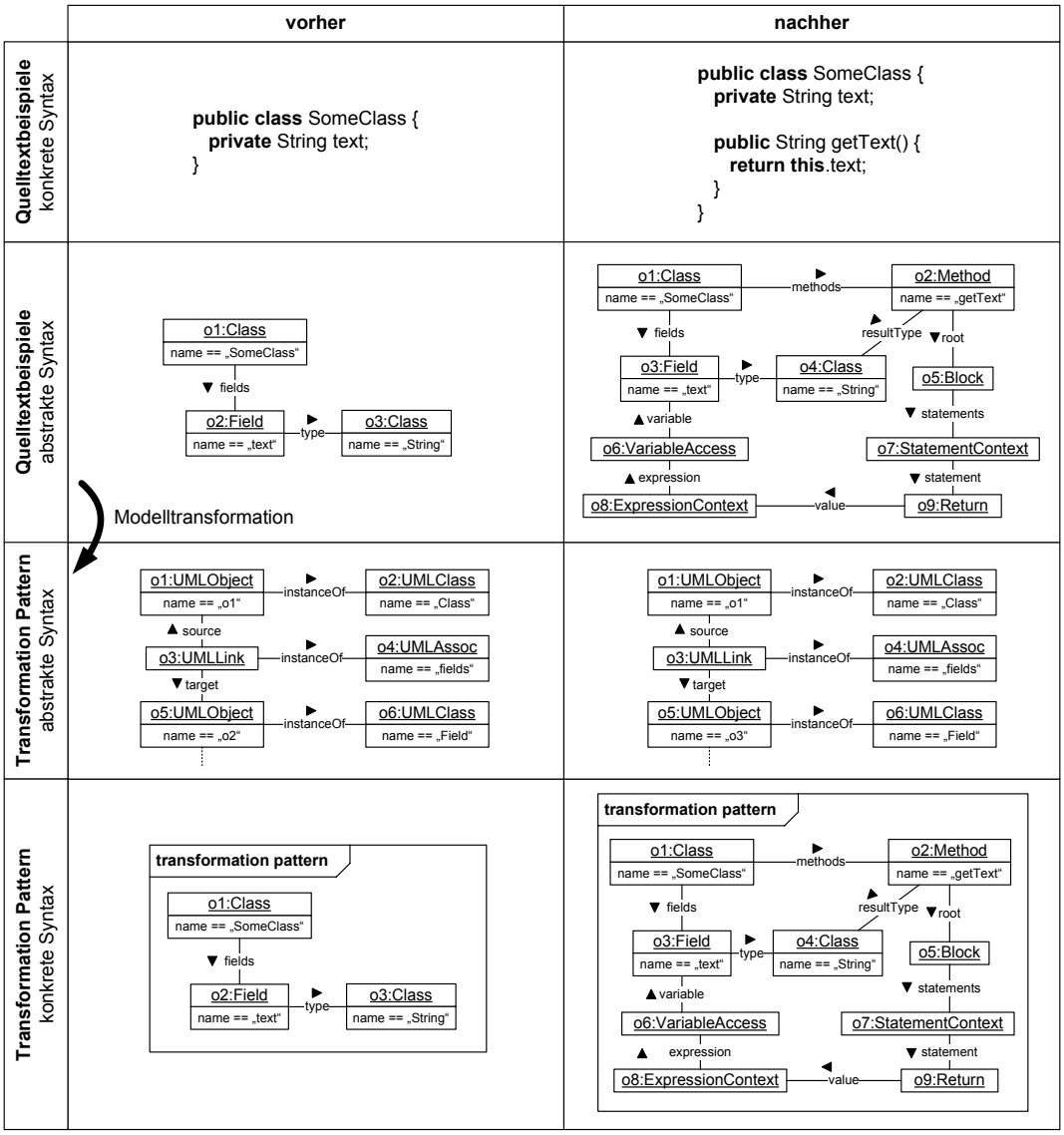

Abbildung 3.12: Ausgangs- und Endsituation in unterschiedlichen Repräsentationen

den Einsatz im Rahmen dieser Arbeit geeignet. Der Ansatz erlaubt die Spezifikation mehrerer Kriterien pro Objekttyp anhand derer Ähnlichkeiten berechnet werden. Dabei werden auch die Verbindungen zwischen den Objekten berücksichtigt. Daher wird hier kein eigener Ansatz zur Identifikation gleicher Objekte entwickelt, sondern es kann der in [WK06] beschriebene eingesetzt werden.[5]

Parallel zur Identifikation gleicher Objekte werden die beiden Syntaxgraphen in jeweils ein Transformation Pattern übersetzt (vgl. Abbildung 3.11). Diese Transformation Pattern nehmen keine Modifikationen vor und ihre linke und rechte Seite beschreiben den abstrakten Syntaxgraphen des jeweiligen Beispiels. Die Übersetzung erfolgt mit Hilfe einer Modelltransformation in abstrakte Syntaxgraphen für Transformation Pattern (siehe dritte Zeile in Abbildung 3.12), die in der konkreten Syntax der Transformation Pattern wie in der vierten Zeile von Abbildung 3.12 gezeigt aussehen. Die Modelltransformation kann zu einem großen Teil deklarativ mit Hilfe einer Triple-Graph-Grammatik [Sch94, Wag06] beschrieben und automatisch ausgeführt werden. Details dazu sind in [Ceb07] nachzulesen.

Die gleichen Objekte sowie die beiden Transformation Pattern für die Beispiele werden im Syntheseschritt verwendet, um ein Transformation Pattern zu erzeugen, das exakt die Ausgangssituation in die Endsituation überführt. Dabei werden die beiden Transformation Pattern für die Beispiele zu einem Transformation Pattern anhand der gleichen Objekte beziehungsweise den sie repräsentierenden Objektvariablen verklebt. In dem entstehenden Transformation Pattern werden dann die Elemente als zu löschend markiert, die im Vorherbeispiel enthalten sind aber nicht im Nachherbeispiel. Analog werden Elemente des Nachherbeispiels, die nicht im Vorherbeispiel enthalten sind, als zu erzeugend markiert. Das Verfahren wird detailliert in [Ceb07] beschrieben.

Abbildung 3.13 zeigt das Ergebnis des Syntheseschritts für das Beispiel der Zugriffsmethode. Das Transformation Pattern ist aus den konkreten Beispielen entstanden und daher noch speziell auf diese zugeschnitten. So enthält zum Beispiel die Objektvariable o1:Class noch die Attributbedingung name == „SomeClass“, weil die Klasse in den Beispielen diesen Namen trägt. Darüber hinaus gibt es noch einige weitere solcher Bedingungen.[6] Über zu spezielle Attributbe-

[5] Leider stand die Implementierung des Ansatzes nicht zur Einbindung in den im Rahmen dieser Arbeit entstandenen Prototyp zur Verfügung, so dass die automatische Ermittlung gleicher Objekte nicht realisiert wurde.

[6] Die Objekte eines abstrakten Syntaxgraphen verfügen in der Regel über eine Reihe weiterer Eigenschaften, die ebenfalls in Form von Attributbedingungen in den Transformation Pattern aufgenommen werden. In den gezeigten Beispielen sind sie im Interesse der Über-

dingungen hinaus können auch Objekt- und Linkvariablen insbesondere in der linken Seite des Transformation Pattern enthalten sein, die die Anwendungsstelle für die Transformation unnötig einschränken.

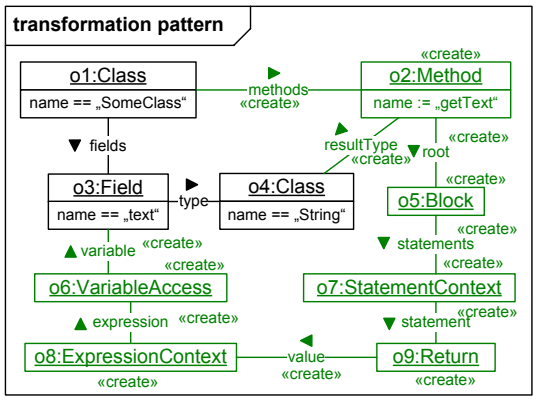

Abbildung 3.13: Synthetisiertes Transformation Pattern

Damit das Transformation Pattern auch in anderen Situationen als der Ausgangssituation angewendet werden kann, muss es verallgemeinert werden. Dies erfolgt im letzten Schritt des Verfahrens. Die Verallgemeinerung kann nicht automatisch vorgenommen werden, da nicht allgemein entscheidbar ist, welche Objekt- oder Linkvariablen oder Attributbedingungen zu speziell sind. Daher wird der Re-Engineer einbezogen. Er kann dabei durch verschiedene automatisierte Verallgemeinerungen unterstützt werden.

So gibt es einen Automatismus, der alle Objekt- und Linkvariablen aus der linken Seite des Transformation Pattern entfernt, die nicht zwingend für die Durchführung der Modifikationen benötigt werden. Dies trifft auf alle Elemente zu, die weder gelöscht werden noch mit gelöschten oder erzeugten Elementen verbunden sind. Des Weiteren können alle Attributbedingungen entfernt werden, die keine Zuweisungen darstellen. Ebenso sind Mischformen möglich, bei denen der Re-Engineer bestimmte Elemente markiert, die erhalten bleiben sollen und alle übrigen nicht zwingend für die Durchführung der Modifikationen benötigten Elemente automatisch entfernt werden.

Transformation Pattern stehen nicht für sich allein sondern beschreiben einzelne Schritte eines Transformationsdiagramms. Ein synthetisiertes und verallgemeinertes Transformation Pattern wird daher schlussendlich manuell in ein Transformationsdiagramm eingebettet. Dabei muss insbesondere minde-

sichtlichkeit ausgeblendet.

stens eine Objektvariable des Transformation Pattern als gebunden definiert werden. Darüber hinaus können sich noch weitere Objektvariablen auf solche beziehen, die bereits in vorangegangenen Schritten des Transformationsdiagramms gebunden oder erzeugt wurden. An manchen Stellen kann es sinnvoll sein, von konkreten Strukturen im abstrakten Syntaxgraphen zu abstrahieren, indem sie durch Pfade ersetzt werden. Solche Anpassungen können nicht automatisiert vorgenommen werden. Auch können Modifikationen, die iteriert ausgeführt werden sollen, nicht ohne Weiteres automatisch erkannt werden. Der Re-Engineer muss solche Anteile manuell als iterierte Anteile kennzeichnen.

3.4.2 Verwendung des Verfahrens

Das beschriebene Verfahren kann verwendet werden, um einzelne Transformation Pattern automatisch aus konkreten Quelltextbeispielen zu synthetisieren, die im Anschluss noch manuell nachbearbeitet werden müssen. Transformation Pattern beschreiben nur einzelne Schritte umfangreicherer Transformationsdiagramme. Dementsprechend sollte auch die Durchführung einer umfangreichen Transformation am Beispiel in sinnvolle Schritte unterteilt werden. Für jeden dieser Schritte wird dann ein Transformation Pattern synthetisiert. Alle dabei entstehenden Transformation Pattern werden abschließend manuell in ein Transformationsdiagramm eingebettet. Wird die Spezifikation nicht in einzelnen Schritten vorgenommen, entstehen große und unübersichtliche Transformation Pattern, deren Nachbearbeitung schwieriger ist und die dann unter Umständen nachträglich in mehrere Transformation Pattern zerlegt werden müssen.

So unterteilt sich die Verkapselung von Lesezugriffen im Rahmen des Encapsulate-Field-Refactorings [Fow99] sinnvollerweise in zwei Schritte: erstens die Erzeugung einer lesenden Zugriffsmethode und zweitens die Ersetzung aller direkten Lesezugriffe durch einen Aufruf der Zugriffsmethode. Die gesamte Transformation kann anhand von Beispielen spezifiziert werden, indem der erste Schritt wie in Abbildung 3.10 auf Seite 91 gezeigt angegeben und synthetisiert wird.

Der zweite Schritt kann wie in Abbildung 3.14 gezeigt angegeben werden. Im Quelltext muss dazu ein direkter Lesezugriff im Vorherbeispiel existieren, der im Nachherbeispiel durch einen Aufruf der Zugriffsmethode ersetzt wurde. Daraus wird automatisch mit Hilfe des Syntheseverfahrens und anschließender vollständiger Minimierung der linken Seite das Transformation Pattern im unteren Teil der Abbildung erzeugt.

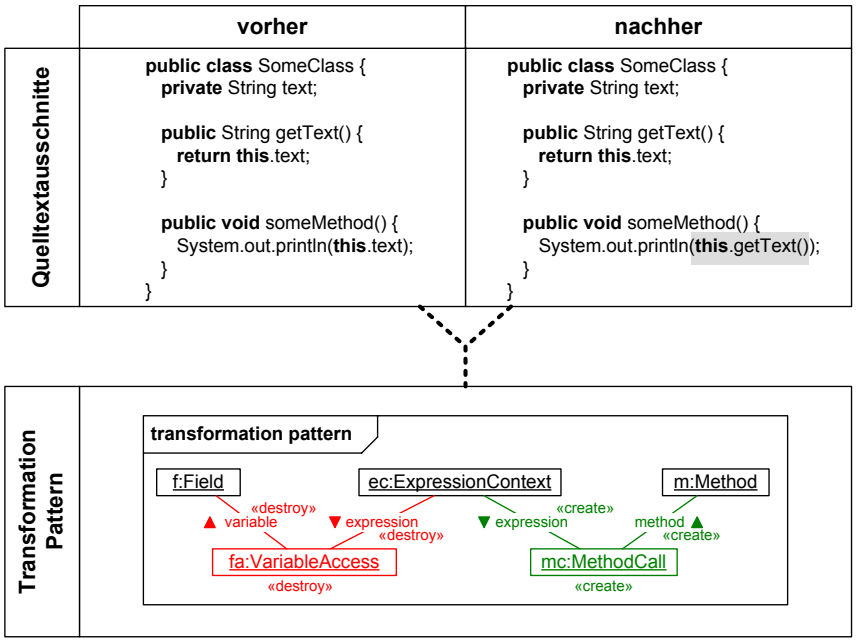

Abbildung 3.14: Ersetzen von direkten Lesezugriffen durch Aufruf einer Zugriffsmethode – im Quelltext und als Transformation Pattern

Die beiden Transformation Pattern müssen dann manuell in ein Transformationsdiagramm eingebettet werden. Abbildung 3.15 zeigt das Ergebnis der Einbettung. In der Signatur des Transformationsdiagramms wird mit field:Field ein Parameter für das zu verkapselnde Attribut eingeführt. Die Objektvariable für das Attribut im ersten Transformation Pattern wird an diesen Parameter gebunden. Im zweiten Transformation Pattern muss sich die Objektvariable für das Attribut ebenfalls auf den Parameter beziehen und die die Zugriffsmethode repräsentierende Objektvariable muss an das in Transformation Pattern (1) erzeugte Methodenobjekt gebunden werden.

Des Weiteren müssen die Modifikationen des zweiten Transformation Pattern iteriert, also für alle direkten Lesezugriffe, ausgeführt werden, außer für den Lesezugriff, der sich innerhalb der Zugriffsmethode befindet. Dies wird durch Einfügen eines iterierten Anteils erreicht. Der im ersten Transformation Pattern erzeugte Lesezugriff der Zugriffsmethode gfa:VariableAccess kann durch den iterierten Anteil nicht gebunden werden, da er durch ein zuvor ausgeführtes Transformation Pattern desselben Transformationsdiagramms erzeugt wird – dies wäre nur explizit durch eine gebundene Objektvariable möglich.

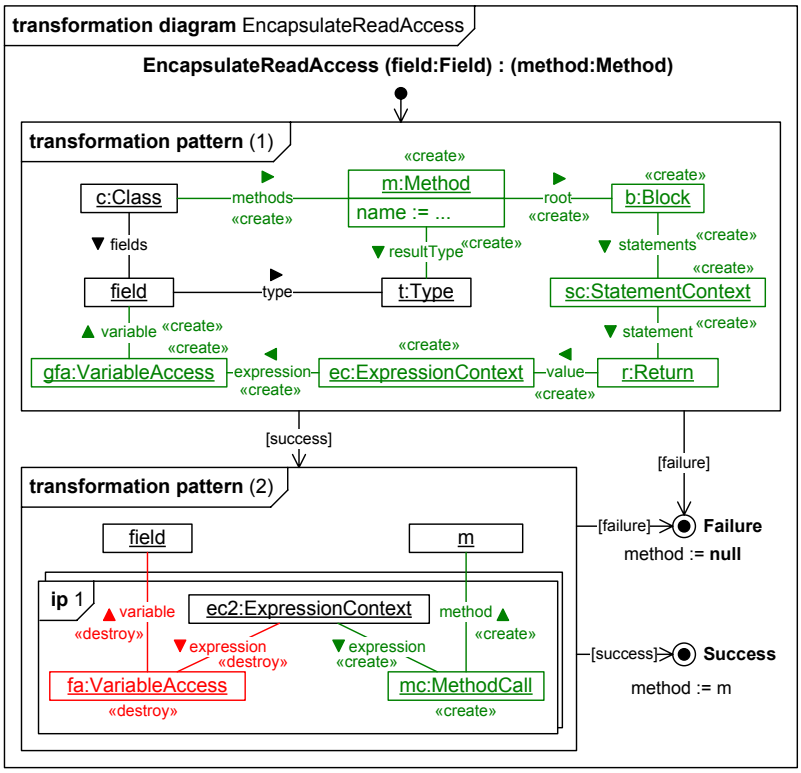

Abbildung 3.15: Transformationsdiagramm zur Verkapselung von Lesezugriffen

Kapitel 4

Verifikation von Transformationsspezifikationen

Im Rahmen dieser Arbeit werden schlecht wartbare Implementierungen im Quelltext bestehender Software identifiziert und mit Hilfe von ausführbaren Transformationsspezifikationen verbessert. Dabei soll in der Regel die Struktur von Software transformiert und verbessert werden, ohne das von außen beobachtbare Verhalten der Software zu verändern.

Programmtransformationen werden hier durch Transformationsdiagramme formalisiert (Kapitel 3). Transformationsdiagramme spezifizieren sowohl die Vorbedingungen für ihre Anwendbarkeit als auch ihre Durchführung. Ihre formal definierte Semantik macht sie zum einen automatisch ausführbar (via Codegenerierung). Zum anderen erlaubt die formale Semantikdefinition den Nachweis, dass Transformationen bestimmte Kriterien erfüllen. Dies können Kriterien sein, deren Erfüllung notwendig sind, damit das von außen beobachtbare Verhalten der transformierten Software unverändert bleibt.

In diesem Kapitel wird ein Verfahren vorgestellt, das aufbauend auf dem in Abschnitt 2.4.3 vorgestellten Verfahren aus [Sch06, BBG+06] die formale Spezifikation von Kriterien und die automatische Verifikation von Transformationsdiagrammen auf Einhaltung der Kriterien erlaubt. Im Folgenden wird die Definition von Verifikationskriterien beschrieben. Danach wird ein Überblick über das Verfahren gegeben und seine einzelnen Schritte genauer vorgestellt. Abschließend wird resümiert, welche Aussagen das Verfahren treffen kann.

4.1 Verifikationskriterien

Transformationsdiagramme erlauben aufgrund ihrer formalen Semantik prinzipiell den Nachweis, dass sie das von außen beobachtbare Verhalten der trans-

formierten Software nicht verändern. Für beliebige Transformationsdiagramme ist dies jedoch im Allgemeinen nicht automatisch möglich, sondern ein aufwändiger und komplizierter manueller Vorgang.

Allerdings ist es möglich Kriterien aufzustellen, die durch ein Transformationsdiagramm eingehalten werden müssen, damit dieses das Verhalten nicht verändern kann, und deren Einhaltung automatisch mit dem in dieser Arbeit entwickelten Verfahren überprüft werden kann. Genau genommen handelt es sich bei den Kriterien um Eigenschaften des abstrakten Syntaxgraphen – also der Software –, die vor und nach einer Transformation erfüllt sein müssen. Im Rahmen dieser Arbeit werden Kriterien unterstützt, die Strukturen beschreiben, die entweder erhalten bleiben müssen oder nicht erzeugt werden dürfen.

In Abschnitt 3.1 werden beispielhaft einige Kriterien genannt, darunter das Kriterium *access preservation*. Dieses besagt, dass wann immer es vor der Ausführung einer Transformation einen lesenden Zugriff auf eine Variable gibt, es nach einer Transformation weiterhin einen Lesezugriff auf dieselbe Variable geben muss. Dieser Zugriff muss allerdings nicht direkt an derselben Stelle stattfinden, sondern er muss von derselben Stelle ausgehen, darf aber zum Beispiel indirekt über eine Zugriffsmethode erfolgen.

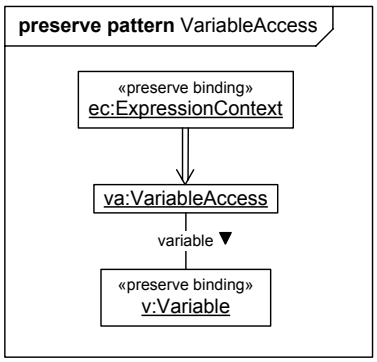

Abbildung 4.1: Erhaltung von Lesezugriffen auf Variablen als zu erhaltendes Graphmuster

Verifikationskriterien werden aufbauend auf [Sch06] in Form von Graphmustern beschrieben. Abbildung 4.1 zeigt das Access-Preservation-Kriterium als Graphmuster. Der Lesezugriff wird durch die Objektvariable va:VariableAccess repräsentiert, die über die Linkvariable variable mit der gelesenen Variable v:Variable verbunden ist. Bei einem Lesezugriff auf eine Variable handelt es sich um einen Ausdruck (engl. *expression*), der in einem bestimmten Kontext, etwa als Bedingung einer If-Anweisung, stattfindet. Dieser Kontext wird durch

die Objektvariable ec:ExpressionContext repräsentiert. Der Kontext und der Lesezugriff sind über einen Pfad miteinander verbunden. Pfade in Graphmustern haben dieselbe Semantik wie die in Abschnitt 3.2.1 beschriebenen Pfade in Transformationsdiagrammen. Der Kontext und der Lesezugriff müssen also nicht unmittelbar verbunden sein. Auf diese Weise erfüllt zum Beispiel auch ein abstrakter Syntaxgraph, in dem ein Lesezugriff indirekt über den Aufruf einer Zugriffsmethode erfolgt, das Graphmuster.

Ein abstrakter Syntaxgraph erfüllt ein Graphmuster, wenn er mindestens einen zum Muster isomorphen Teilgraphen unter entsprechender Berücksichtigung der Pfade enthält, es also eine Anwendungsstelle für das Graphmuster gibt. Beim zuvor beschriebenen Kriterium handelt es sich um eine Situation, die wenn sie vor der Ausführung einer Transformation im abstrakten Syntaxgraphen vorhanden ist, erhalten bleiben muss. Daher wird es als zu erhaltendes Graphmuster, in der Abbildung erkennbar an der Beschriftung *preserve pattern*, formuliert.

Damit eine Transformation das Kriterium erfüllt, muss es für jede Anwendungsstelle für das Graphmuster, die vor Ausführung der Transformation im abstrakten Syntaxgraphen existiert auch danach eine Anwendungsstelle geben. Eine solche Anwendungsstelle wird im Weiteren auch *zu erhaltende Struktur* genannt.

Dabei müssen bestimmte Elemente des Graphmusters dieselben bleiben, in diesem Fall der Kontext und die gelesene Variable. Dies wird an den Objektvariablen ec und v durch den Stereotyp ≪preserve binding≫ definiert. Alle übrigen Elemente, in diesem Fall der Pfad, der Lesezugriff selbst sowie die Linkvariable zur Variable, dürfen verändert und neu gebunden werden, solange dabei wieder eine vollständige Anwendungsstelle entsteht. Auf diese Weise können direkte Lesezugriffe durch den Aufruf einer Zugriffsmethode ersetzt werden. Solange in demselben Kontext ein Lesezugriff auf dieselbe Variable erreichbar ist, bleibt das Graphmuster erfüllt.

Weitere Beispiele für zu erhaltende Graphmuster werden in Abschnitt 6.4 gegeben.

Im Gegensatz zu Graphmustern, die erhalten bleiben müssen, gibt es auch Graphmuster, die Strukturen beschreiben, die durch eine Transformation nicht erzeugt werden dürfen. Abbildung 4.2 zeigt ein verbotenes Graphmuster (erkennbar an der Beschriftung mit *forbidden pattern*), das einen unerlaubten Variablenzugriff beschreibt.

Innerhalb einer Methode m1:Method findet ein Lesezugriff va:VariableAccess auf eine Variable v:Variable statt, die in einer anderen Methode m2:Method deklariert wird. Dies ist offensichtlich nicht erlaubt, kann aber geschehen, wenn

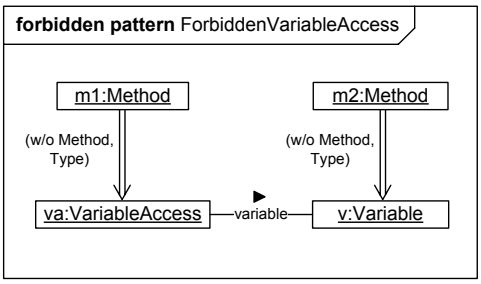

Abbildung 4.2: Unerlaubter Variablenzugriff als verbotenes Graphmuster

eine Transformation Anweisungen verschiebt, die Variablenzugriffe beinhalten. Sowohl der Zugriff als auch die Variablendeklaration kann irgendwo *innerhalb* der jeweiligen Methode m1 beziehungsweise m2 geschehen. Dies wird durch die beiden Pfade jeweils von m1 zu va und von m2 zu v ausgedrückt. Damit diese Pfade die jeweilige Ausgangsmethode nicht verlassen, wird durch Angabe von (w/o Method, Type) die Traversierung von Objekten des Typs Method sowie Type verboten.

Ein abstrakter Syntaxgraph darf ein verbotenes Graphmuster nicht erfüllen, dass heißt er darf keine Anwendungsstelle für das Muster enthalten. Ein Transformationsdiagramm erfüllt dann ein solches Kriterium, wenn der abstrakte Syntaxgraph nach seiner Ausführung keine Anwendungsstelle für das verbotene Muster enthalten kann, die nicht bereits vor der Ausführung existiert haben muss. Eine solche Anwendungsstelle für ein verbotenes Muster wird im Weiteren auch *verbotene Struktur* genannt.

Abschnitt 6.4 gibt weitere Beispiele für verbotene Graphmuster.

4.2 Überblick über das Verifikationsverfahren

Das Ziel des Verifikationsverfahrens ist nachzuweisen, dass ein Transformationsdiagramm Kriterien einhält, die wie zuvor beschrieben spezifiziert wurden. Eine Kriterienverletzung liegt vor, wenn nach der Ausführung eines Diagramms im abstrakten Syntaxgraphen eines zu transformierenden Programms eine Anwendungsstelle eines verbotenen Graphmusters existiert, die vor seiner Ausführung nicht existierte, oder eine zu Beginn seiner Ausführung vorhandene Anwendungsstelle eines zu erhaltenden Graphmusters am Ende zerstört ist. Während der Ausführung dürfen temporär verbotene Muster erzeugt oder zu erhaltende Muster zerstört werden, sofern sie am Ende der Ausführung nicht

mehr beziehungsweise wieder existieren.

In Kenntnis des konkreten zu transformierenden Programms könnte die Einhaltung von Kriterien prinzipiell mit Hilfe einer Erreichbarkeitsanalyse nachgewiesen werden, die ausgehend vom aktuellen abstrakten Syntaxgraphen als Startgraph alle durch Ausführung eines Diagramms erreichbaren Graphen berechnet. Die Graphen könnten dann auf Kriterienverletzungen analysiert werden. Allerdings gilt der Nachweis dann nur für das konkrete Programm und muss für jedes weitere Programm, auf das die Transformation angewendet werden soll, wiederholt werden. Zudem ist fraglich, ob die Menge der zu berechnenden Graphen nicht zu groß wird.

Das hier entwickelte Verfahren arbeitet daher ohne Kenntnis des konkreten Programms und erbringt Nachweise, die allgemein für die Transformationen, unabhängig vom konkret transformierten Programm gelten. Ziel ist nachzuweisen, dass die Einhaltung eines jeden Kriteriums eine induktive Invariante ist, die in allen Versionen beliebiger transformierter Programme gilt, die bei Ausführung kompletter Transformationen entstehen können.

Dazu wird für jedes Transformationsdiagramm und jedes Graphmuster versucht, ein Beispiel für eine Ausführung des Diagramms zu berechnen, die zu einer Kriterienverletzung führt, indem sie einen korrekten Graphen in einen inkorrekten Graphen transformiert. Ein solches Beispiel wird *Gegenbeispiel* genannt. Gegenbeispiele können vom Re-Engineer dazu genutzt werden, die Spezifikationen geeignet anzupassen. Kann kein Gegenbeispiel berechnet werden, so kann das Diagramm das entsprechende Kriterium nicht verletzen, egal wie das Programm beschaffen ist, auf das es angewendet wird.

Ein Transformationsdiagramm besteht aus einer Menge von Transformation Pattern deren Ausführung durch Kontrollfluss gesteuert wird. Jedes Transformation Pattern besteht aus genau einem nicht-iterierten Anteil und beliebig vielen geschachtelten iterierten Anteilen. Jeder Anteil ist durch eine Graphtransformationsregel beschrieben. Im Weiteren werden daher die Begriffe Anteil und Graphtransformationsregel (kurz Regel) synonym verwendet.

Bei der Ausführung eines Transformationsdiagramms ergibt sich damit letztlich eine Sequenz von angewendeten Graphtransformationsregeln, im Weiteren auch *Regelsequenz* genannt. Gegenbeispiele sind spezielle Regelsequenzen, bei denen eine Kriterienverletzung auftritt.

Da es unendlich viele verschiedene Programme geben kann, können sich in Verbindung mit den durch iterierte Anteile spezifizierten Schleifen auch unendlich viele verschiedene Regelsequenzen bei der Ausführung eines Transformationsdiagramms ergeben, die natürlich nicht alle analysiert werden können.

Das hier entwickelte Verfahren nutzt daher aufbauend auf Ideen aus [Sch06]

zum einen die Struktur der Graphmuster und zum anderen die Spezifikation der Regeln sowie des Kontrollfluss, um nur repräsentative (Teil-)Ausführungen von Transformationen auf repräsentativen Teilgraphen eines abstrakten Syntaxgraphen zu betrachten. Es ist ausreichend Teilgraphen zu betrachten, da die Regeln eines Transformationsdiagramms nur lokale Änderungen vornehmen.

Die Struktur der Graphmuster wird genutzt, um die Regeln eines Transformationsdiagramms zu identifizieren, die durch Löschen oder Erzeugen bestimmter Typen von Elementen zu einer Verletzung eines Kriteriums beitragen können. Da nur diese Regeln eine Kriterienverletzung auslösen können, reicht es aus, Teilausführungen eines Diagramms zu betrachten. Daher werden gezielt solche Regelsequenzen berechnet, in denen diese Regeln so zur Anwendung kommen, dass eine Kriterienverletzung auftreten kann.

Die Spezifikation der Regeln sowie des Kontrollfluss werden ausgenutzt, um darauf zu schließen, wie der abstrakte Syntaxgraph für die Ausführung solcher Regelsequenzen mindestens beschaffen sein muss und welche (weiteren) Anwendungen von Regeln Bestandteil einer Regelsequenz sein müssen. Der abstrakte Syntaxgraph wird dabei nicht vollständig sondern lediglich ausschnittsweise charakterisiert.

Der Kontrollfluss wird zum einen durch die Transitionen zwischen den Transformation Pattern eines Diagramms und zum anderen durch die Schachtelung iterierter Anteile in den Transformation Pattern modelliert. Durch die Transitionen ergibt sich eine endliche Menge von *Ausführungspfaden* von der Startaktivität eines Transformationsdiagramms zu einer seiner Stopaktivitäten.

Für einen Ausführungspfad steht fest, welche Transformation Pattern erfolgreich angewendet werden und welche nicht. Daraus lassen sich bereits Rückschlüsse darauf ziehen, wie der abstrakte Syntaxgraph beschaffen sein muss, damit eine entsprechende Ausführung des Diagramms möglich ist: er muss mindestens Anwendungsstellen für die nicht-iterierten Anteile der erfolgreichen Transformation Pattern enthalten und darf keine Anwendungsstellen für die nicht-iterierten Anteile fehlgeschlagener Transformation Pattern enthalten. Da die Ausführungspfade unabhängig von den zu verifizierenden Kriterien sind, werden sie im ersten Schritt des Verifikationsverfahrens bestimmt (siehe Abbildung 4.3[1]). Wie das geschieht, wird in Abschnitt 4.3 beschrieben.

Wie häufig welche iterierten Anteile der erfolgreich angewendeten Transformation Pattern eines Ausführungspfades angewendet werden, hängt von der Beschaffenheit des abstrakten Syntaxgraphen ab und lässt sich allein aus der

[1]Die Abbildung zeigt den Ablauf innerhalb des Prozessschritts der Transformationsverifikation aus Abbildung 1.1 als UML-Aktivitätsdiagramm [Obj].

Spezifikation eines Transformationsdiagramms nicht erkennen.

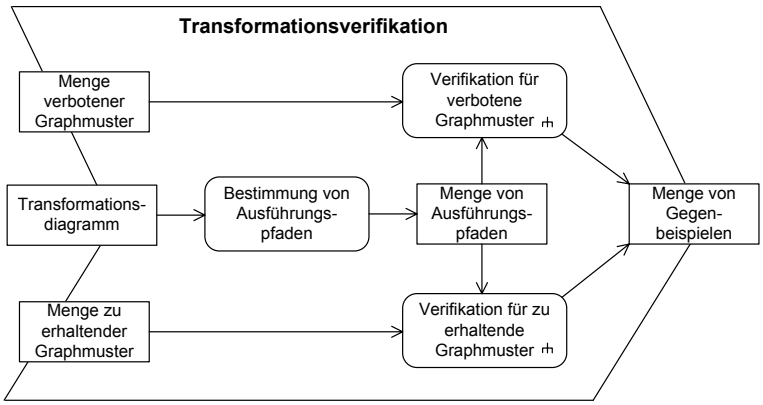

Abbildung 4.3: Überblick über das Verifikationsverfahren

Für die Berechnung von Gegenbeispielen wird anhand der Struktur des jeweils zu verifizierenden Graphmusters bestimmt, wie häufig welche iterierten Anteile durch Erzeugen oder Löschen von Elementen zum Entstehen oder Zerstören einer Anwendungsstelle eines verbotenen oder zu erhaltenden Graphmusters beitragen können. Da die Graphmuster endlich sind, können (iterierte) Anteile nur endlich häufig einen Teil einer solchen Anwendungsstelle erzeugen oder löschen. In Verbindung mit den in Abschnitt 3.2 beschriebenen Einschränkungen und der in Abschnitt 4.4.3 beschriebenen Betrachtung von Transformationsaufrufen wird erreicht, dass garantiert nur endliche Regelsequenzen berechnet werden.

Die *problematischen* Anteile werden dann zusätzlich zu den nicht-iterierten Anteilen, die aufgrund des Ausführungspfades angewendet werden müssen, bei der Berechnung von Regelsequenzen unter Umständen mehrfach eingeplant. Dies geschieht so, dass alle möglichen Kombinationen der Anwendung (und Nicht-Anwendung) von kontrollflussbedingt notwendigen und problematischen Anteilen gebildet werden, die zum Entstehen oder zur Zerstörung einer Anwendungsstelle eines verbotenen oder zu erhaltenden Musters führen.

Da es sich bei den Regelsequenzen nur um Teilausführungen handelt, können weitere Regeln prinzipiell anwendbar sein und dabei einen Einfluss auf die Regelsequenzen haben, zum Beispiel indem sie Kriterienverletzungen wieder korrigieren oder dafür benötigte Elemente löschen. Solche Einflüsse werden ebenfalls berücksichtigt und gegebenenfalls pessimistisch abgeschätzt. Auch ist es sehr aufwändig bis hin zu unmöglich die Auswirkungen von Transformations-

aufrufen automatisch exakt zu analysieren, so dass diese ebenfalls pessimistisch abgeschätzt werden.

Dadurch können Gegenbeispiele berechnet werden, die nicht eintreten können. Solche Gegenbeispiele werden in der vorliegenden Arbeit in Anlehnung an die Terminologie der Strukturmustererkennung *false-positives* genannt, da das Verfahren fälschlicherweise eine Kriterienverletzung anzeigt. Dementgegen werden nicht erkannte Kriterienverletzungen als *false-negatives* bezeichnet, da das Verfahren fälschlicherweise nichts anzeigen würde. Dies soll durch die vorgenommenen Abschätzungen ausgeschlossen werden.

Für verbotene und zu erhaltende Graphmuster wird dabei unterschiedlich vorgegangen (siehe Abbildung 4.3). Im Fall von verbotenen Graphmustern wird unterstellt, dass es am Ende einer jeden Regelsequenz eine Anwendungsstelle des verbotenen Musters gibt. Ausgehend davon wird versucht, Regelsequenzen so rückwärts zu planen, dass problematische Anteile zur Entstehung dieser verbotenen Struktur beitragen und die Sequenzen konform zu einem Ausführungspfad sind. Dies geschieht für jeden zuvor bestimmten Ausführungspfad und jedes verbotene Graphmuster und wird in Abschnitt 4.4 detailliert beschrieben.

Für zu erhaltende Graphmuster wird unterstellt, dass es eine Anwendungsstelle des Musters zu Beginn einer jeden Regelsequenz gibt. Ausgehend davon wird versucht, Regelsequenzen so vorwärts zu planen, dass problematische Anteile zur Zerstörung dieser Anwendungsstelle beitragen und die Sequenzen konform zu einem Ausführungspfad sind. Dies geschieht ebenfalls für jeden Ausführungspfad und für jedes zu erhaltende Graphmuster. Das Vorgehen wird in Abschnitt 4.5 beschrieben.

Da die im Folgenden beschriebenen Verfahrensschritte versuchen Gegenbeispiele zu konstruieren, werden zu ihrer Erklärung zum Teil (unfertige) Gegenbeispiele beziehungsweise Regelsequenzen gezeigt. Daher werden im nächsten Abschnitt zunächst Gegenbeispiele und ihre Notation anhand eines Beispiels vorgestellt.

4.2.1 Gegenbeispiele

Ein Gegenbeispiel beschreibt als eine spezielle Regelsequenz den Teil einer Ausführung eines Transformationsdiagramms, der zu einer Kriterienverletzung führen kann.

Eine solche Regelsequenz besteht aus Graphmustern, die einen Ausschnitt eines abstrakten Syntaxgraphen zu verschiedenen Zeitpunkten während der Ausführung der Regelsequenz beschreiben. Die Graphmuster sind über gerichtete Transitionen miteinander verbunden, welche die Anwendung eines Anteils

eines Transformation Pattern, also einer Graphtransformationsregel, repräsentieren. Die so verbundenen Graphmuster charakterisieren den abstrakten Syntaxgraphen jeweils vor und nach der Regelanwendung.

Da ein Gegenbeispiel eine Kriterienverletzung zeigt, existiert entweder im ersten Graphmuster eine zu erhaltende Struktur, die spätestens im letzten Graphmuster nicht mehr besteht oder es gibt im letzten Graphmuster eine verbotene Struktur, die zu Beginn noch nicht existierte.

Ein Gegenbeispiel enthält nur solche Regelanwendungen, die entweder aufgrund des Kontrollfluss des zugehörigen Transformationsdiagramms und/oder des durch die Graphmuster charakterisierten Syntaxgraphen geschehen müssen oder die ein oder mehrere Elemente einer verbotenen Struktur erzeugen beziehungsweise ein oder mehrere Elemente einer zu erhaltenden Struktur löschen.

Der abstrakte Syntaxgraph – oder Wirtsgraph – wird durch die Graphmuster nicht vollständig beschrieben. Er kann weitere Elemente und damit Anwendungsstellen für weitere Anteile des zugehörigen Transformationsdiagramms enthalten. Daher können bei einer tatsächlichen Ausführung des Transformationsdiagramms weitere iterierte Anteile von erfolgreichen Transformation Pattern des jeweiligen Ausführungspfades angewendet werden.

Wenn weitere potentielle Anwendungen, die nicht Bestandteil der Regelsequenz sind, Auswirkungen auf die Ausführbarkeit der Regelsequenz haben können, werden diese durch das Verifikationsverfahren berücksichtigt und zum Teil pessimistisch abgeschätzt; dies wird später noch beschrieben.

Abbildung 4.4 zeigt ein durch das Verifikationsverfahren berechnetes Gegenbeispiel für die erste Version des Transformationsdiagramms *SimpleExtract* aus Abbildung 3.1 auf Seite 51 in Abschnitt 3.2 und das in Abbildung 4.2 gezeigte verbotene Graphmuster *ForbiddenVariableAccess*. Das Gegenbeispiel zeigt, wie eine Ausführung des Transformationsdiagramms eine Instanz des verbotenen Graphmusters herstellen kann.

Es besteht aus drei Graphmustern, die über Transitionen miteinander verbunden sind. Im Beispiel steht die erste Transition für die Anwendung des nicht-iterierten Anteils von Transformation Pattern (2) des Transformationsdiagramms und die zweite für die Anwendung des nicht-iterierten Anteils von Transformation Pattern (3).

Ein Graphmuster besteht aus genau einer *Anwendungsbedingung*, engl. *application condition (ac)*, sowie beliebig vielen *negativen Anwendungsbedingungen*, engl. *negative application condition (nac)*. Die Anwendungsbedingungen sind vergleichbar mit Regelgraphen (Abschnitt 3.3.4). Ein Wirtsgraph erfüllt ein Graphmuster, wenn er eine Anwendungsstelle für seine Anwendungsbedingung enthält und keine Anwendungsstelle für eine der negativen Anwendungsbedin-

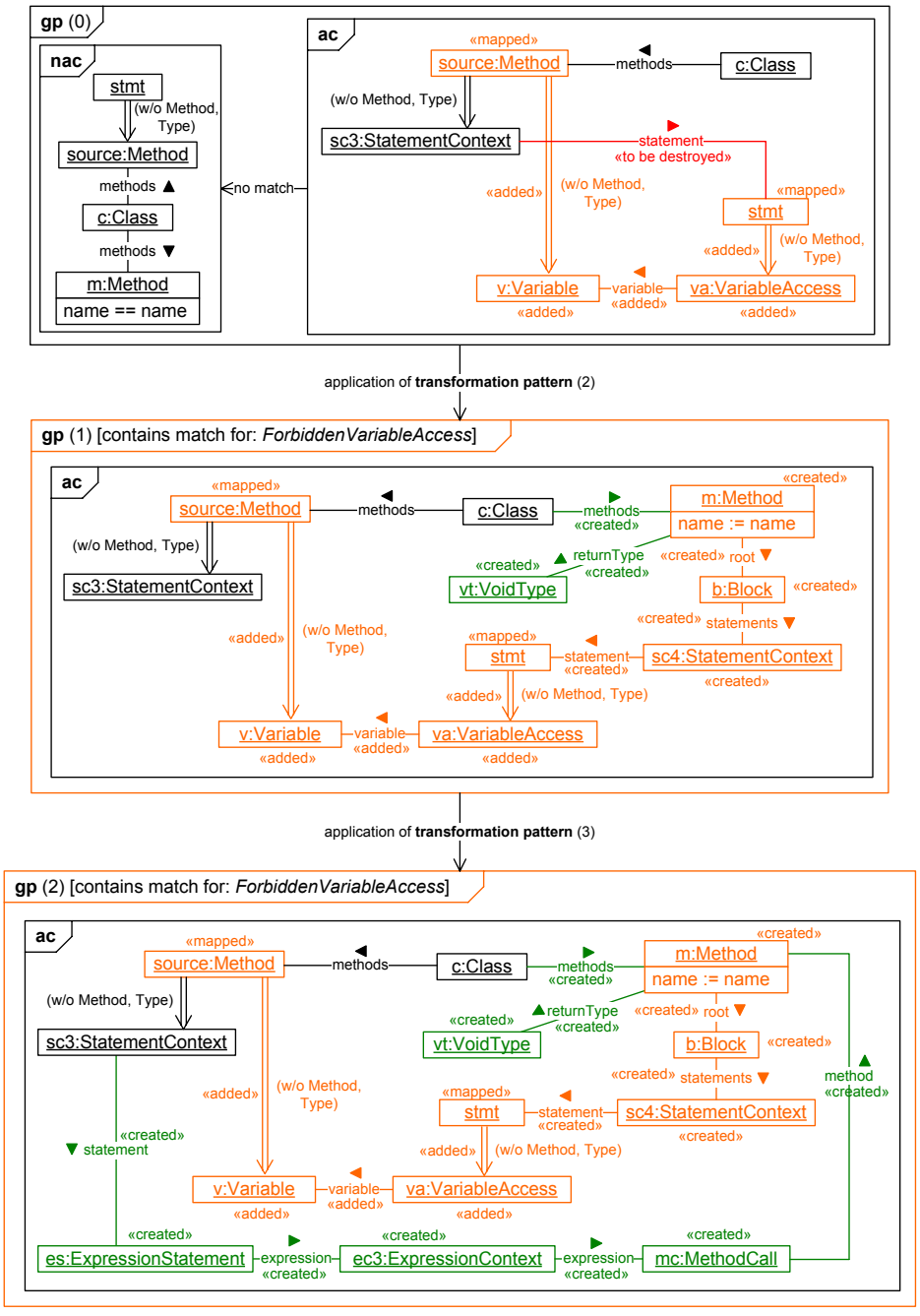

Abbildung 4.4: Ein Gegenbeispiel

gungen.

Dabei ist zu beachten, dass kein Element zu einer Anwendungsstelle gehören darf, das durch die Ausführung des zum Gegenbeispiel gehörenden Transformationsdiagramms erzeugt wird, es sei denn, es wird explizit gebunden (siehe Abschnitt 3.2). Durch das zugehörige Transformationsdiagramm erzeugte Elemente sind in den Graphmustern mit ≪created≫ markiert. Alle anderen Elemente können nicht durch dieselbe Ausführung des Transformationsdiagramms erzeugt worden sein, sondern müssen bereits im Wirtsgraphen existiert haben oder durch andere aufgerufene Transformationsdiagramme erzeugt worden sein. Dies gilt sowohl für die Anwendungsbedingung eines Graphmusters als auch für seine negativen Anwendungsbedingungen.

Das erste Graphmuster gp (0) des Gegenbeispiels zeigt, wie das Programm zu Beginn der Ausführung des Transformationsdiagramms beschaffen sein muss, damit die Regelsequenz möglich ist. Seine Anwendungsbedingung enthält eine Anwendungsstelle[2] für die linke Seite von Transformation Pattern (2) des Diagramms, bestehend aus c:Class, source:Method, stmt, sc3:StatementContext sowie den Linkvariablen zwischen diesen Objektvariablen. Die Objektvariable stmt ist als gebunden dargestellt, da sie sich explizit auf den gleichnamigen Parameter des Transformationsdiagramms bezieht.

Darüber hinaus muss bereits ein Teil der verbotenen Struktur vorhanden sein, die durch die weiteren Regelanwendungen vervollständigt wird. Dabei handelt es sich um v:Variable, va:VariableAccess sowie die mit ihnen verbundenen Linkvariablen und Pfade. Alle Elemente des Graphmusters, die Bestandteil der verbotenen Struktur sind beziehungsweise sein werden sind orange gefärbt. Die Linkvariable statement zwischen sc3:StatementContext und stmt ist rot gefärbt und mit ≪to be destroyed≫ markiert, da sie durch die folgende Regelanwendung gelöscht wird.

Zusätzlich verfügt das erste Graphmuster über eine negative Anwendungsbedingung. Damit bei Ausführung des Transformationsdiagramms eine Anwendung von Transformation Pattern (2) möglich ist, muss direkt vorher Transformation Pattern (1) fehlgeschlagen sein, so dass der Wirtsgraph keine Anwendungsstelle für dessen nicht-iterierten Anteil beinhalten kann. Genau dies drückt die negative Anwendungsbedingung des Graphmusters aus.

Das zweite Graphmuster gp (1) zeigt, wie die Anwendungsbedingung des vorhergehenden Graphmusters nach der Anwendung des nicht-iterierten An-

[2] In Abschnitt 3.3.6 werden Anwendungsstellen für Anteile von Transformation Pattern in einfachen Graphen definiert. Anwendungsstellen in Anwendungsbedingungen von Graphmustern sind dazu analog definiert, nur können Pfadausprägungen konforme Pfade enthalten.

teils von Transformation Pattern (2) beschaffen ist. Die markierte Linkvariable statement wurde gelöscht. Die Objektvariablen m:Method, b:Block, sc4:StatementContext, vt:VoidType sowie die mit ihnen verbundenen Linkvariablen werden erzeugt und sind mit ≪created≫ markiert sowie grün gefärbt, sofern sie nicht Bestandteil der verbotenen Struktur sind. Alle übrigen Elemente entsprechen den gleichnamigen Elementen des vorangehenden Graphmusters.

Die verbotene Struktur ist in diesem Moment bereits vollständig (die Umrandung des Graphmusters ist daher orange gefärbt und es ist entsprechend beschriftet). Die nächste Regelanwendung ist damit nicht mehr relevant für das Entstehen des verbotenen Musters. Allerdings muss sie gemäß Ausführungspfad erfolgen, damit das Transformationsdiagramm erfolgreich terminiert. Außerdem hätte sie möglicherweise noch eine Korrektur herbeiführen können.

Über die in diesem Beispiel gezeigten Konstrukte einer Regelsequenz hinaus können Graphmuster noch optionale Elemente enthalten, die im Wirtsgraphen zum jeweiligen Zeitpunkt existieren können aber nicht zwingend müssen, und es kann Transitionen zwischen Graphmustern geben, welche die Ausführung aufgerufener Transformationsdiagramme repräsentieren. Diese Konstrukte werden später bei der Beschreibung der für ihre Einführung verantwortlichen Verfahrensschritte erklärt.

4.3 Bestimmung von Ausführungspfaden

Transformationsdiagramme können komplexe Vorbedingungen für ihre Anwendbarkeit spezifizieren, zum Beispiel in Form negativer Anwendungsbedingungen, oder Alternativen modellieren, die je nach Beschaffenheit des abstrakten Syntaxgraphen unterschiedliche Modifikationen vornehmen.

Bei der Ausführung eines Transformationsdiagramms wird ein Ausführungspfad von der Startaktivität zu einer Stopaktivität durchlaufen. Alternativen manifestieren sich in unterschiedlichen Ausführungspfaden. Um ein Transformationsdiagramm dahingehend zu überprüfen, ob es ein Kriterium verletzen kann, muss untersucht werden, welche Modifikationen es bei seiner Ausführung prinzipiell vornehmen kann. Dazu müssen all die Ausführungspfade betrachtet werden, die in einer Erfolg anzeigenden Stopaktivität enden. Fehlgeschlagene Transformationen können ignoriert werden, da durch sie keine Änderungen vorgenommen werden (siehe Abschnitt 3.2).

Abbildung 4.5 zeigt verschiedene Ausführungspfade eines erweiterten Transformationsdiagramms zum Extrahieren von Methoden.

Auf der linken Seite der Abbildung wird ein Ausführungspfad in rot her-

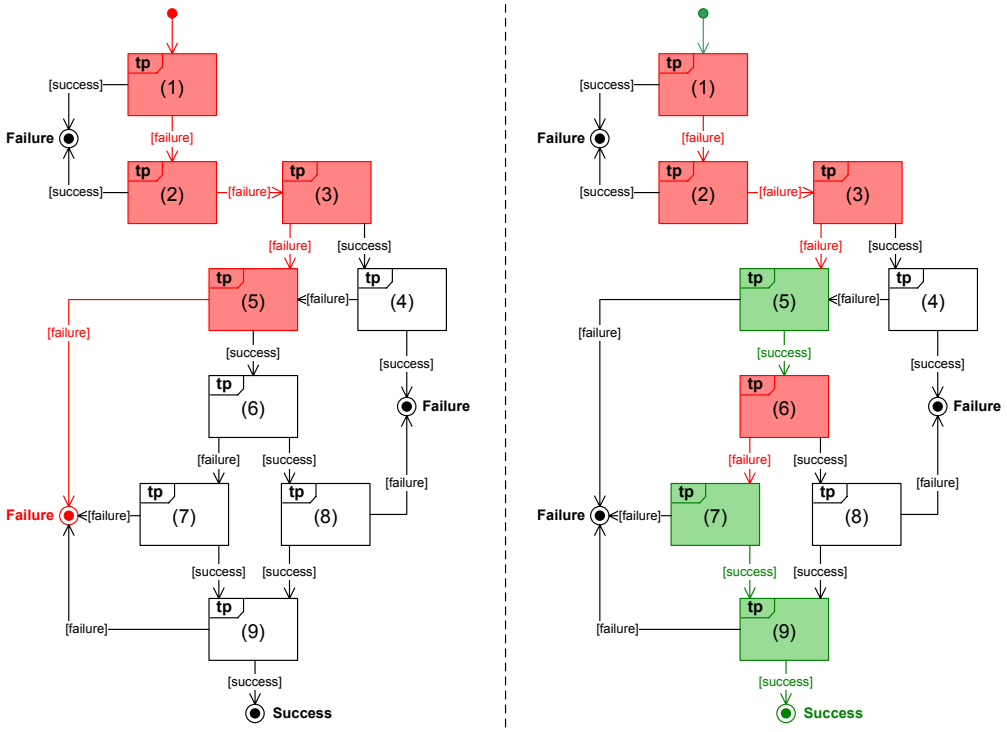

Abbildung 4.5: Unterschiedliche Ausführungspfade desselben Transformationsdiagramms

vorgehoben, der einen Fehlschlag des Transformationsdiagramms darstellt. Für diesen Ausführungspfad steht fest, dass die Transformation Pattern (1), (2), (3) und (5) fehlschlagen. Des Weiteren endet der Pfad in einer Failure-Stopaktivität. In einem solchen Fall führt ein Transformationsdiagramm keinerlei Änderungen am abstrakten Syntaxgraphen durch, egal ob auf dem Pfad Transformation Pattern erfolgreich angewendet wurden oder nicht. Damit muss ein solcher Pfad auch nicht verifiziert werden. Endete derselbe Pfad in einer Success-Stopaktivität, müsste er ebenso wenig verifiziert werden, da kein Transformation Pattern angewendet wurde und daher ebenfalls keine Modifikationen erfolgt sind.

Die rechte Seite der Abbildung markiert dagegen einen Pfad zu einer Success-Stopaktivität, auf dem die Transformation Pattern (1), (2), (3) und (6) fehlschlagen und die Transformation Pattern (5), (7) und (9) erfolgreich sind.

Jeder Ausführungspfad besteht aus einer Sequenz von Transformation Pattern. Für die einzelnen Transformation Pattern steht dabei fest, ob sie ange-

wendet wurden oder nicht. Bei allen angewendeten Transformation Pattern kommt mindestens der nicht-iterierte Anteil zur Anwendung.

Die Aktivitäten, Transformation Pattern und Transitionen eines Transformationsdiagramms bilden zusammen einen gerichteten Graphen ohne Zyklen. Der Graph verfügt mit der Startaktivität über genau eine Quelle und mit den Stopaktivitäten über beliebig viele Senken. Damit können alle Ausführungspfade des Diagramms durch einen rekursiven Algorithmus ermittelt werden, der ausgehend von der Startaktivität eine Breitensuche anwendet.

4.4 Verifikation für verbotene Graphmuster

Das Ziel der Verifikation für verbotene Graphmuster ist es festzustellen, ob einer der Ausführungspfade eines Transformationsdiagramms angewendet auf einen korrekten Graphen das Potential hat, eine Anwendungsstelle eines verbotenen Graphmusters zu erzeugen, die am Ende der Ausführung noch existiert und zu Beginn noch nicht vorhanden gewesen sein muss.

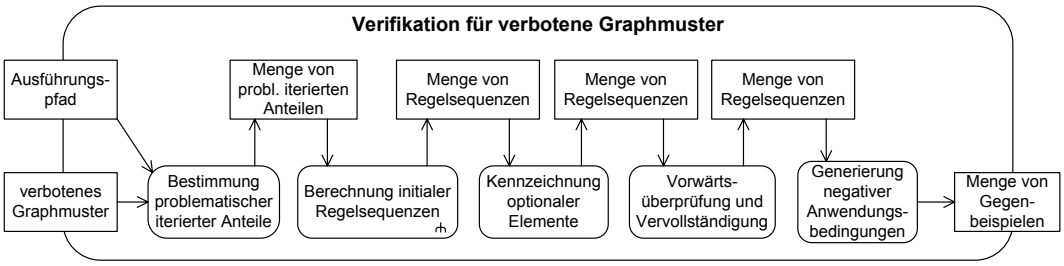

Abbildung 4.6: Verifikation für verbotene Graphmuster

Das Verfahren zeigt die Korrektheit eines Transformationsdiagramms in Bezug auf ein verbotenes Graphmuster, indem es für jeden Ausführungspfad und jedes verbotene Graphmuster versucht, Gegenbeispiele zu berechnen. Dazu werden für jeden Ausführungspfad und jedes verbotene Graphmuster nacheinander die in Abbildung 4.6 gezeigten Schritte durchlaufen:

1. werden iterierte Anteile bestimmt, die auf dem Ausführungspfad prinzipiell angewendet werden könnten und dabei einen Teil des verbotenen Graphmusters erzeugen könnten (Abschnitt 4.4.1).

2. werden Regelsequenzen berechnet, indem ausgehend von einer Anwendungsstelle des verbotenen Graphmusters am Ende einer jeden Sequenz

Regeln des Ausführungspfades rückwärts eingeplant werden. Dabei werden nur Regeln eingeplant, die entweder aufgrund des Kontrollfluss angewendet werden müssen oder die angewendet werden könnten und dabei einen Teil der verbotenen Struktur erzeugen würden (problematisch sind). Die dazu notwendige Beschaffenheit des Wirtsgraphen wird unterstellt und durch die Sequenzen dokumentiert. Es werden alle Kombinationen von notwendigen und problematischen Regelanwendungen gebildet.

3. können die Graphmuster der Regelsequenzen Elemente enthalten, die zum jeweiligen Zeitpunkt nicht mehr für die Anwendung späterer Regeln der Sequenz benötigt werden. Diese könnten daher durch weitere, nicht eingeplante Regelanwendungen oder durch aufgerufene Transformationsdiagramme gelöscht werden, ohne dass die Sequenz unmöglich würde. Solche Elemente werden als optional gekennzeichnet (Abschnitt 4.4.3).

4. enthalten die Regelsequenzen bisher nur zwingend notwendige oder problematische Regelanwendungen. Aufgrund der dazu notwendigen Beschaffenheit des Wirtsgraphen (ohne optionale Elemente) können zu verschiedenen Zeitpunkten der Sequenz weitere Regeln garantiert anwendbar sein, die bei der Rückwärtsplanung nicht berücksichtigt wurden. Diese könnten zu einer Korrektur der verbotenen Struktur oder zu einem Widerspruch zur geplanten Sequenz von Regelanwendungen führen. Regelsequenzen werden dahingehend geprüft und entweder ergänzt oder verworfen (Abschnitt 4.4.4).

5. werden in den Regelsequenzen die fehlschlagenden Transformation Pattern des Ausführungspfades noch nicht berücksichtigt. Da sie fehlschlagen, darf es für ihre nicht-iterierten Anteile zum jeweiligen Zeitpunkt keine Anwendungsstelle im Wirtsgraphen geben. Darüber hinaus muss die Anwendbarkeit weiterer Regeln ausgeschlossen werden, die ein Element löschen können, das für problematische Regelanwendungen benötigt wird. Um die Existenz von bestimmten Anwendungsstellen auszuschließen, werden die Graphmuster der Regelsequenzen geeignet um negative Anwendungsbedingungen ergänzt (Abschnitt 4.4.5).

Das Ergebnis ist eine (eventuell leere) Menge von Gegenbeispielen (Abschnitt 4.4.6).

4.4.1 Bestimmung problematischer iterierter Anteile

Für jeden nicht-iterierten sowie iterierten Anteil eines Transformation Pattern sind drei mögliche Auswirkungen in Bezug auf ein gegebenes verbotenes Muster zu unterscheiden. Ein Anteil kann:

1. das Potential haben, einen Teil des verbotenen Musters zu erzeugen. Ist dieser Teil mehrfach in dem verbotenen Muster enthalten, kann das verbotene Muster unter Umständen erst durch die mehrfache Anwendung eines iterierten Anteils erzeugt werden. Zudem kann ein verbotenes Muster auch erst durch die kombinierte einfache oder mehrfache Anwendung verschiedener Anteile entstehen. Anteile dieser Kategorie werden im Weiteren *problematisch* genannt.

2. das Potential haben, einen Teil des verbotenen Musters zu zerstören und damit eine während der Anwendung des Transformation Pattern entstandene verbotene Struktur zu korrigieren. Dies kann auch mehrfach geschehen. Ein Transformation Pattern kann zum Beispiel durch eine Modifikation des Wirtsgraphen mehrfach eine verbotene Struktur hervorrufen, wobei all diese Strukturen durch einen iterierten Anteil garantiert korrigiert werden. Anteile dieser Kategorie werden im Weiteren *korrigierend* genannt.

3. keinen Einfluss auf die Entstehung oder Korrektur des verbotenen Musters haben. Anteile dieser Kategorie werden im Weiteren *neutral* genannt.

Dementsprechend können problematische Anteile in beliebiger Kombination mit anderen zur Erzeugung einer verbotenen Struktur beitragen, unter Umständen sogar mehrfach, oder auch gar nicht.

Ein Anteil kann einen Teil eines verbotenen Musters erzeugen, wenn er mindestens ein Objekt oder einen Link erzeugt, das/der isomorph beziehungsweise typkonform zu einer Objekt- oder Linkvariable des verbotenen Musters ist. Theoretisch könnten alle übrigen Elemente des verbotenen Muster bereits im Wirtsgraphen existiert haben und die Anwendung des Anteils erzeugt das oder die noch fehlenden Elemente. Natürlich kann ein Muster auch vollständig erzeugt werden.

Die in Abschnitt 4.1 gezeigten Beispiele für Graphmuster verwenden Pfade. Pfade werden dazu genutzt, um von Strukturen im abstrakten Syntaxgraphen zu abstrahieren. Für die Struktur muss lediglich gelten, dass es einen Weg

vom Start- zum Zielobjekt des Pfades über beliebig viele andere Objekte und Links hinweg gibt. Dabei sind nur bestimmte Abfolgen von Objekten und Links erlaubt, die durch das Strukturmodell des abstrakten Syntaxgraphen definiert werden (siehe Abschnitt 3.2.1). Zusätzlich kann ein Pfad die Typen der Objekte einschränken, die durch ihn traversiert werden dürfen.

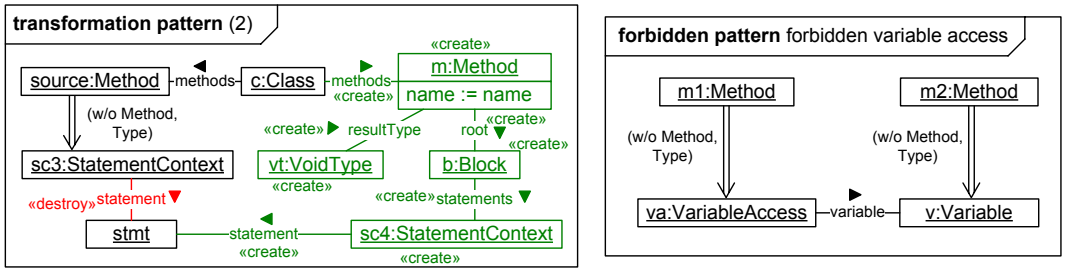

Abbildung 4.7: Ein Transformation Pattern und ein verbotenes Graphmuster mit Pfaden

Eine Instanz eines verbotenen Graphmusters mit Pfaden kann daher auch entstehen, indem mindestens ein Link erzeugt wird, der Bestandteil einer Ausprägung für einen der Pfade des Musters sein kann. Dies veranschaulicht das folgende Beispiel.

Abbildung 4.7 zeigt das Transformation Pattern (2) aus den vorherigen Beispielen und das bereits bekannte verbotene Graphmuster *ForbiddenVariableAccess*. Das Transformation Pattern extrahiert eine gegebene Anweisung stmt in eine neu erzeugte Methode. Abbildung 4.8 zeigt ein Beispiel für die Anwendung des Transformation Pattern in Form einer Regelsequenz. Das erste Graphmuster gp (0) zeigt einen Ausschnitt eines Wirtsgraphen, auf den das Transformation Pattern angewendet wird. Das zweite Graphmuster zeigt denselben Ausschnitt nach der Anwendung.

Während gp (0) keine vollständige Anwendungsstelle[3] für das verbotene Muster enthält, kann es in gp (1) auf die orange gefärbten Elemente abgebildet werden: m2 auf source, v auf v, va auf va und m1 auf m. Der Pfad zwischen m2 und v kann auf den Pfad zwischen source und v und die Linkvariable zwischen va und v auf die entsprechende Linkvariable zwischen den Objektvariablen va und v in gp (1) abgebildet werden.

[3] Anwendungsstellen für Graphmuster in Graphmustern sind analog zu Anwendungsstellen für Regelgraphen in Graphen (vgl. Abschnitt 3.3.6) definiert, nur können Pfadausprägungen konforme Pfade enthalten und sie sind unabhängig von einem Transformationsdiagramm und seinem Ausführungszustand.

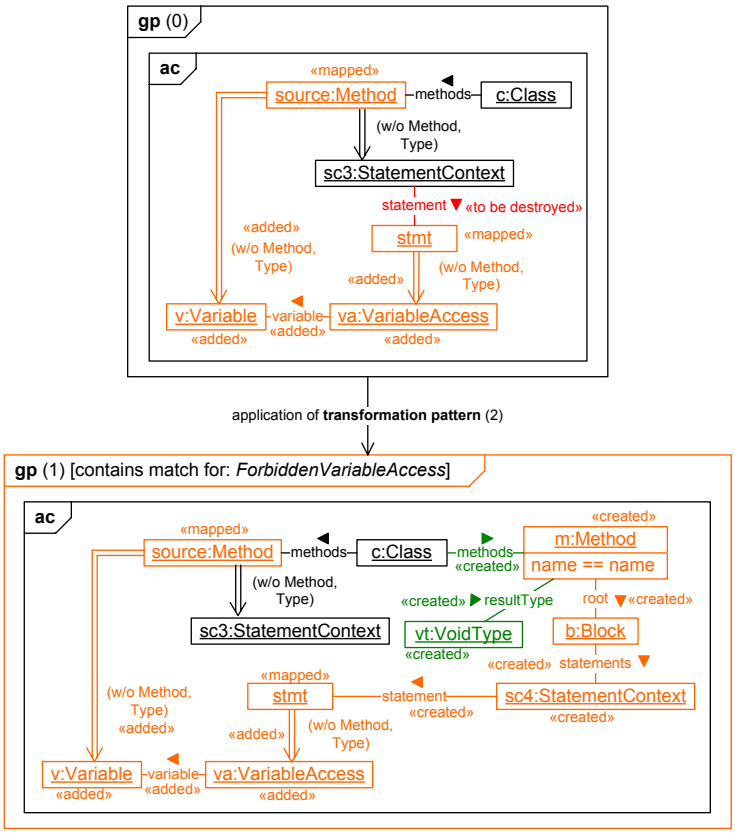

Abbildung 4.8: Beispiel für die Erzeugung einer Pfadausprägung

Der Pfad zwischen m1 und va im verbotenen Muster kann in gp (1) zum einen auf die Kette m:Method – root – b:Block – statements – sc4:StatementContext – statement – stmt abgebildet werden (vgl. Anhang A), die Bestandteil eines Pfades von einer Methode zu einem Lesezugriff sein kann, der kein weiteres Methodenobjekt und kein Typobjekt beinhaltet. Dass dies möglich ist, geht aus dem um Pfadinformationen erweiterten Strukturmodell des abstrakten Syntaxgraphen hervor. Zum anderen kann der Pfad von stmt zu va einbezogen werden, um die Pfadausprägung zu vervollständigen, so dass va von m aus erreicht werden kann.

Einige der Elemente, auf die der Pfad abgebildet werden kann, wie zum Beispiel die Linkvariable statement zwischen sc4:StatementContext und stmt, werden erst durch die Anwendung des nicht-iterierten Anteils des Transformation Pattern erzeugt, wodurch die Pfadausprägung insgesamt erzeugt beziehungs-

weise vervollständigt wird.

Damit können Transformation Pattern beziehungsweise deren nicht-iterierte oder iterierte Anteile eine verbotene Struktur prinzipiell dann erzeugen, wenn sie ein Objekt oder einen Link erzeugen können, das/der Bestandteil einer Ausprägung eines der Pfade des verbotenen Musters sein kann.

Welche Typen von Objekten und Links Bestandteil einer Pfadausprägung sein können, kann mit Hilfe einer Wegesuche im um Pfadinformationen erweiterten Strukturmodell des abstrakten Syntaxgraphen ermittelt werden. Dazu müssen alle Wege (ohne Zyklen) vom Typ des Startknotens eines Pfades zum Typ des Zielknotens berechnet werden, auf denen nur pfadnavigierbare Assoziationen in die angegebene Richtung und keine verbotenen Typen traversiert werden. Dabei müssen auch Assoziationen von und zu allen Subtypen erreichbarer Typen einbezogen werden. Alle Typen inklusive all ihrer Subtypen sowie Assoziationen, die Bestandteil eines der berechneten Wege sind, können Bestandteil einer Pfadausprägung sein.

Anteile eines Transformation Pattern können analog verbotene Strukturen zerstören beziehungsweise korrigieren, wenn sie mindestens eine Objekt- oder Linkvariable löschen, die isomorph beziehungsweise typkonform zu einer Objekt- oder Linkvariable des verbotenen Musters ist. Zusätzlich können sie auch korrigieren, wenn sie eine Ausprägung für einen Pfad zerstören, indem sie mindestens ein Element löschen, das Bestandteil einer solchen Ausprägung sein könnte.

Alle iterierten Anteile der erfolgreichen Transformation Pattern eines Ausführungspfades werden auf Basis der Typen der Elemente, die sie erzeugen oder löschen können, sowie der Typen der Elemente des jeweiligen Graphmusters unter Berücksichtigung von Pfaden in die Kategorien problematisch, korrigierend und neutral eingestuft. Die problematischen iterierten Anteile werden an den nächsten Schritt (siehe Abbildung 4.6) übergeben, der im folgenden Abschnitt beschrieben wird.

4.4.2 Berechnung initialer Regelsequenzen

In diesem Schritt werden für einen Ausführungspfad alle repräsentativen Regelsequenzen berechnet, die eine Anwendungsstelle eines verbotenen Graphmusters erzeugen. Dabei wird aufbauend auf die Ideen aus [Sch06] unterstellt, dass es am Ende der Ausführung eine Anwendungsstelle des verbotenen Musters gibt. Davon ausgehend werden Regelsequenzen unter Berücksichtigung des Kontrollfluss sowie problematischer iterierter Anteile sukzessive rückwärts – von hinten nach vorne – berechnet, bis eine Anwendung des nicht-iterierten

Anteils des ersten erfolgreichen Transformation Pattern des Ausführungspfades Bestandteil der Sequenz ist.

In einer Regelsequenz enthaltene Anwendungen problematischer iterierter Anteile müssen einen Teil der Anwendungsstelle für das verbotene Muster erzeugen. Gleichzeitig darf die Anwendungsstelle des verbotenen Musters maximal vollständig erzeugt werden, so dass in Verbindung mit der zweiten in Abschnitt 3.2 beschriebenen Einschränkung garantiert nur endliche Sequenzen berechnet werden. Das Ergebnis sind, sofern sie konstruiert werden können, Regelsequenzen, in denen eine verbotene Struktur entsteht, die zu Beginn nicht enthalten ist.

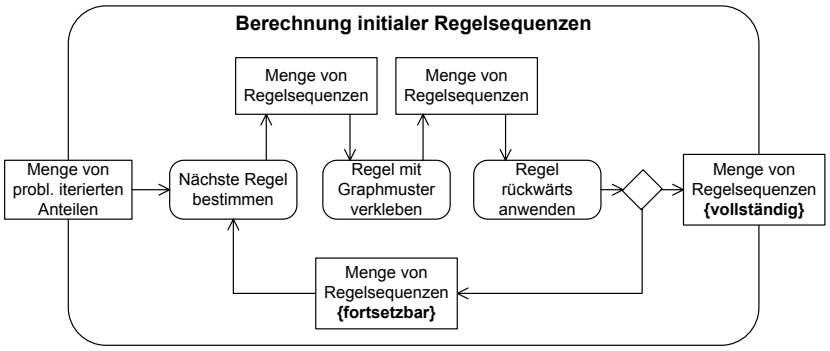

Abbildung 4.9: Berechnung initialer Regelsequenzen

Als Eingabe für die Berechnung (Abbildung 4.9) dient eine Menge problematischer iterierter Anteile, die für einen Ausführungspfad und ein verbotenes Graphmuster im vorangehenden Schritt (Abschnitt 4.4.1) bestimmt wurde. Die Berechnung der Regelsequenzen findet in drei aufeinander folgenden Teilschritten statt, die sich jeweils eine Menge von Regelsequenzen übergeben:

1. Für jede Regelsequenz werden die darin als nächstes anwendbaren Regeln (problematische oder aufgrund des Kontrollfluss notwendige) bestimmt (Abschnitt 4.4.2.1).

2. Damit eine Regel in einer Regelsequenz angewendet werden kann, muss es eine Anwendungsstelle für sie geben; aufgrund der Rückwärtsplanung eine Anwendungsstelle für ihre rechte Seite im aktuell ersten Graphmuster der Sequenz. In jeder Regelsequenz werden daher Anwendungsstellen für die als nächstes anzuwendende Regel durch Verkleben mit dem aktuell ersten Graphmuster bestimmt beziehungsweise falls notwendig durch Hinzufügen von Elementen geschaffen (Abschnitt 4.4.2.2),

3. In jeder Regelsequenz wird die als nächstes anzuwendende Regel rückwärts auf die zuvor geschaffene Anwendungsstelle angewendet und das nächste (neue erste) Graphmuster der Sequenz ermittelt (Abschnitt 4.4.2.3).

Diese drei Teilschritte werden für Regelsequenzen, die noch nach vorne fortgesetzt werden können, wiederholt, bis dies nicht mehr möglich ist. Begonnen wird mit einer initialen Regelsequenz, die nur eine Kopie des verbotenen Musters als (letztes) Graphmuster und damit eine verbotene Struktur enthält.

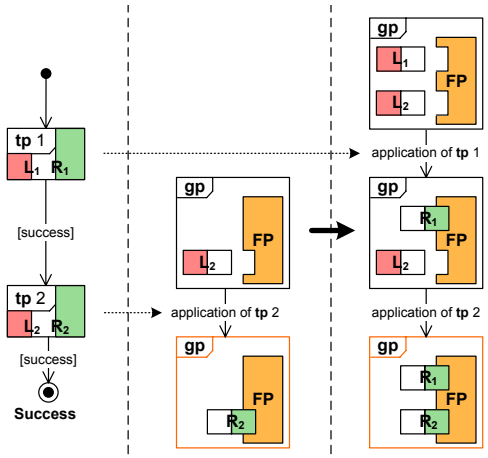

Abbildung 4.10: Schematische Darstellung der Berechnung initialer Regelsequenzen

Abbildung 4.10 zeigt dies zunächst schematisch anhand eines Beispiels. Auf der linken Seite wird ein Ausführungspfad mit zwei erfolgreichen Transformation Pattern ohne iterierte Anteile dargestellt. In den Transformation Pattern sind jeweils die linke und rechte Seite angedeutet: die linke Seite besteht aus dem rot und weiß gefärbten Teil und die rechte Seite ist weiß und grün. Der rote Teil wird bei der Anwendung gelöscht, der grüne erzeugt und der weiße bleibt erhalten.

In der Mitte der Abbildung wird eine initiale Regelsequenz dargestellt, die im letzten Graphmuster eine vollständige Anwendungsstelle des verbotenen Musters (FP) enthält. Das verbotene Muster und die rechte Seite von tp 2 wurden so verklebt, dass die Anwendung von tp 2 einen Teil dieser Anwendungsstelle erzeugt. Im durch Rückwärtsanwendung von tp 2 berechneten Graphmuster vor der Anwendung ist dieser noch nicht enthalten und FP ist (noch) nicht vollständig. Die durch tp 2 gelöschten Elemente sind dagegen noch vorhanden.

Auf der rechten Seite wird gezeigt, wie eine Anwendung von tp 1 hinzukommt, die vor tp 2 erfolgen muss. Diese Anwendung erzeugt ebenfalls einen Teil von FP. Dies wird dann analog im letzten Graphmuster ergänzt.

Anders als im Beispiel gezeigt, gibt es typischerweise mehr als eine mögliche Regelsequenz. Das Verfahren berechnet alle möglichen Sequenzen.

4.4.2.1 Nächste Regel bestimmen

Jede Regelsequenz muss solange rückwärts fortgesetzt werden, bis sie eine Anwendung des nicht-iterierten Anteils des ersten erfolgreichen Transformation Pattern des Ausführungspfades enthält.

Die initiale Regelsequenz enthält noch keine Regelanwendung. In diesem Fall kommen als nächste Regel zum einen der nicht-iterierte Anteil des letzten erfolgreichen Transformation Pattern des Ausführungpfades oder einer seiner problematischen iterierten Anteile infrage. Tatsächlich könnte auch ein unproblematischer iterierter Anteil zuletzt angewendet werden. Da wir aber an der Erzeugung einer verbotenen Struktur interessiert sind, werden zu diesem Zeitpunkt nur problematische iterierte Anteile betrachtet.

In Regelsequenzen, die bereits Regelanwendungen enthalten, muss die darin aktuell erste angewendete Regel betrachtet werden, um die möglichen nächsten Regeln zu bestimmen.

Handelt es sich bei der aktuell ersten angewendeten Regel um einen nicht-iterierten Anteil, so muss ausgehend von der in den Abschnitten 3.3.8 und 3.3.9 formalisierten Semantik davor entweder

- der nicht-iterierte Anteil des davor auf dem Ausführungspfad erfolgreichen Transformation Pattern oder
- einer dessen iterierter Anteile

angewendet werden.

Der linke Teil der Abbildung 4.11 zeigt dies schematisch; ein Transformation Pattern mit iterierten Anteilen wird als Baumstruktur dargestellt, die die Hierarchie der Anteile repräsentiert (vgl. Abbildung 3.8). Der zuletzt angewendete nicht-iterierte Anteil ist fett umrandet. Die davor anwendbaren Anteile sind blau gefärbt.

Ist die aktuell erste angewendete Regel ein iterierter Anteil, so muss davor entweder

- dessen Vateranteil oder

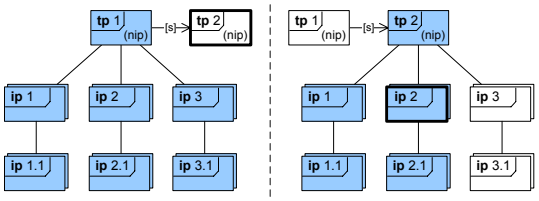

Abbildung 4.11: Direkt vor einem Anteil anwendbare Anteile (schematische Darstellung)

- er selbst ein weiteres Mal oder
- einer der in ihm transitiv enthaltenen iterierten Anteile oder
- einer der iterierten Anteile seines Vateranteils mit kleinerem Ausführungsrang oder einer der darin transitiv enthaltenen iterierten Anteile

angewendet werden. Dies wird im rechten Teil von Abbildung 4.11 dargestellt.

Zur Fortsetzung einer Regelsequenz müssen nicht alle der genannten Möglichkeiten unterstellt werden. Vielmehr werden die Regelsequenzen so kurz wie möglich gehalten, indem nur solche Anteile berücksichtigt werden, die aufgrund des Kontrollfluss angewendet werden müssen oder die tatsächlich etwas zum Entstehen der verbotenen Struktur beitragen können.

Aufgrund des Kontrollfluss müssen nicht-iterierte Anteile erfolgreicher Transformation Pattern des Ausführungspfades sowie Vateranteile angewendeter iterierter Anteile angewendet werden, da iterierte Anteile immer relativ zu einer Anwendung ihres Vateranteils angewendet werden. Ist die Fortsetzung einer Regelsequenz mit einem solchen *notwendigen* Anteil möglich, so wird sie wie im Folgenden beschrieben unterstellt.

Können zwischen einem solchen notwendigen Anteil und dem aktuell ersten angewendeten Anteil der Sequenz weitere problematische iterierte Anteile angewendet werden, so müssen für jeden dieser Anteile Fortsetzungen der Regelsequenz berechnet werden, in denen seine Anwendung mindestens ein Element der verbotenen Struktur erzeugt.

Für jede Regelsequenz werden so alle als nächstes anwendbaren Regeln bestimmt. Für jede dieser Regeln wird die Regelsequenz kopiert und um eine Transition für die Anwendung der Regel ergänzt. Die so entstehende Menge von Regelsequenzen wird an den folgenden Teilschritt weitergegeben.

4.4.2.2 Regel mit Graphmuster verkleben

Damit die zuvor ermittelte nächste Regel einer Regelsequenz angewendet werden kann, muss es im aktuell ersten Graphmuster der Sequenz eine Anwendungsstelle für die rechte Seite der Regel geben. Eine solche Anwendungsstelle muss nicht vollständig vorliegen, sondern notfalls durch Hinzufügen von Elementen vervollständigt werden. Dies geschieht, indem die rechte Regelseite mit dem Graphmuster *verklebt* wird (siehe auch Abschnitt 2.4.3 und Anhang A).

Beim Verkleben der rechten Seite einer Graphtransformationsregel mit einem Graphmuster entsteht eine Menge neuer Graphmuster, die sowohl eine Anwendungsstelle für die rechte Seite als auch für das ursprüngliche Graphmuster enthalten. Für jedes dieser Graphmuster wird eine Kopie der Regelsequenz angelegt.

Alle zur rechten Seite gehörenden Elemente, die durch die Regel erzeugt werden, werden in einem neuen Graphmuster mit dem Stereotyp ≪created≫ markiert. Elemente, auf die beim Verkleben Elemente der verbotenen Struktur im ursprünglichen Graphmuster abgebildet werden, werden mit ≪mapped≫ markiert, sofern sie nicht bereits ≪created≫ sind. Elemente, die zur verbotenen Struktur gehören und in einem neuen Graphmuster hinzugefügt werden, werden mit ≪added≫ markiert.

Zusätzlich muss beim Verkleben folgendes beachtet werden:

Gebundene Objektvariablen Regeln können sich über gebundene Objektvariablen explizit auf von zuvor angewendeten Regeln gebundene oder erzeugte Objekte beziehen. Enthält die neu einzuplanende Regel eine Objektvariable, auf die sich eine bereits in der Regelsequenz eingeplante Regel durch eine gebundene Objektvariable bezieht, so muss die Objektvariable der rechten Regelseite auf eben diese Objektvariable im Graphmuster abgebildet werden. Dies gilt für alle solchen Objektvariablen der rechten Regelseite.

Binden erzeugter Elemente Regeln können keine Elemente binden, die von zuvor angewendeten Regeln erzeugt wurden, außer dies geschieht explizit über gebundene Objektvariablen. Daher dürfen beim Verkleben durch die Regel erzeugte Elemente der rechten Regelseite nur auf Elemente im Graphmuster abgebildet werden, die zur verbotenen Struktur gehören und mit ≪added≫ markiert sind. Alle anderen Elemente des Graphmusters sind entweder durch Regelanwendungen gebunden oder erzeugt. Eine Ausnahme bilden durch die Regel erzeugte Links, die bereits beste-

hende Links überschreiben können (siehe Abschnitt 3.3.7). Diese können auf alle isomorphen Links im Graphmuster abgebildet werden.

An dieser Stelle zeigt sich, wie die Einschränkung in Bezug auf das Binden zuvor erzeugter Elemente aus Abschnitt 3.2 zum einen dazu führt, dass bei der Verifikation weniger Kombinationen von Regelanwendungen untersucht werden müssen. Zum anderen können (auch iterierte) Regeln so nur endlich häufig angewendet werden, da sie sich allein (ohne Transformationsaufrufe, siehe Abschnitt 4.4.3) nicht beliebig neue Anwendungsstellen selbst schaffen können. Dies trägt auch dazu bei, die Terminierung des Verfahrens zu garantieren.

Pfade der Regel Die rechte Seite einer Regel enthält eigentlich keine Pfade. Da Ausprägungen für Pfade der linken Seite aber auch nach der Anwendung der Regel noch existieren können, werden die Pfade der linken Seite auch in den neu gebildeten Graphmustern hinzugefügt, sofern ihre Start- und Zielobjektvariablen darin enthalten sind.

Es ist möglich, dass Teile des jeweiligen Graphmusters Bestandteil eines dieser Pfade beziehungsweise seiner Ausprägung sind. Für die Regelanwendung ist dies zu diesem Zeitpunkt ohne Bedeutung, sofern Ausprägungen für die Pfade existieren können, ohne gegen Kardinalitätseinschränkungen zu verstoßen. Allerdings kann es passieren, das solche Pfadausprägungen im weiteren Verlauf zerstört werden.

Dies kann dadurch ermittelt werden, dass die Pfade auf Teilausprägungen abgebildet werden. Praktische Erfahrungen haben aber gezeigt, dass dabei sehr viele weitere neue Graphmuster entstehen können. Daher werden die Pfade ohne sie auf Teilausprägungen abzubilden ergänzt, sofern dadurch keine Kardinalitätseigenschaften verletzt werden; sonst ist das Graphmuster nicht möglich. Zu einem späteren Zeitpunkt (Abschnitt 4.4.3) wird überprüft, ob Pfadausprägungen zerstört worden sein können und wenn ja, werden solche Pfade als *optional* gekennzeichnet. Dies bedeutet, dass sie vorhanden sein *können* aber nicht *müssen*.

Pfade der verbotenen Struktur Durch die neue Regelanwendung kann eine Ausprägung für einen Pfad der verbotenen Struktur erzeugt werden. Um dies zu überprüfen werden Pfade, die zur verbotenen Struktur gehören, in den neuen Graphmustern entfernt und auf darin enthaltene Teilausprägungen abgebildet, sofern dies möglich ist.

Die Start- und Zielobjektvariablen der Pfade sind in jedem neuen Graphmuster bekannt. In jedem Graphmuster werden ausgehend davon für jeden Pfad alle Teilausprägungen, also Ketten von Objekt-, Linkvariablen und Pfaden bestimmt, die Bestandteil einer Ausprägung des Pfades zwischen den bekannten Objektvariablen sein können. Dies geschieht auf Basis des um Pfadinformationen erweiterten Strukturmodells des abstrakten Syntaxgraphen (vgl. Anhang A).

Die Teilausprägungen werden verwendet, um einen Pfad beziehungsweise eine Ausprägung dafür auf alle möglichen Weisen in das Graphmuster *erneut einzukleben*. Teilausprägungen werden dabei falls erforderlich am Anfang und am Ende um Pfade ergänzt, damit Start- und Zielobjektvariable tatsächlich verbunden sind. Die Pfade werden nur ergänzt, wenn dies unter Berücksichtigung der im Graphmuster vorhandenen Elemente und der im Strukturmodell definierten Kardinalitäten möglich ist; sonst wird die Teilausprägung verworfen. Elemente einer Teilausprägung werden mit ≪mapped≫ markiert, sofern sie nicht bereits anderweitig markiert sind. Ergänzte Pfade werden mit ≪added≫ markiert.

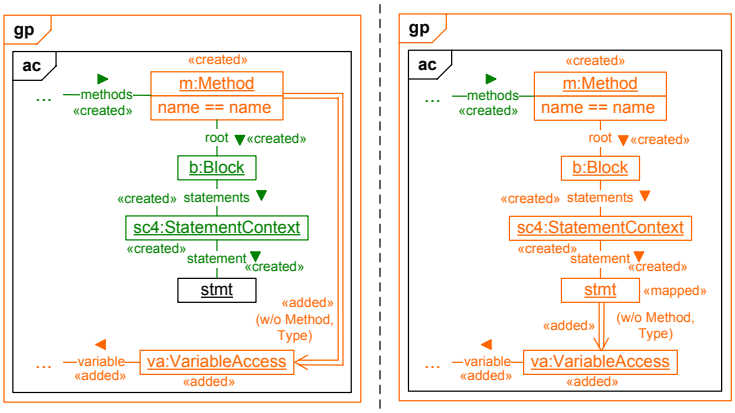

Abbildung 4.12: Erneutes Einkleben eines Pfades

Abbildung 4.12 zeigt auf der linken Seite einen Ausschnitt eines Graphmusters, in dem der Pfad von m:Method zu va:VariableAccess Teil der verbotenen Struktur ist. Durch Verkleben mit einer Regel sind die Objektvariablen b:Block, sc4:StatementContext sowie die mit ihnen verbundenen Linkvariablen als erzeugte Elemente hinzugekommen, so dass es sich bei der Kette m:Method – root – b:Block – statements – sc4:StatementContext – statement – stmt um eine Teilausprägung des

Pfades handeln kann. Die rechte Seite der Abbildung zeigt den Ausschnitt des Graphmusters, nachdem der Pfad entfernt und auf diese Teilausprägung abgebildet wurde. Damit er wieder komplett ist, wurde zwischen stmt und va:VariableAccess ein konformer Pfad hinzugefügt. Er ist konform, da er zwischen den verbundenen Typen ausgeprägt werden kann und dabei kein Objekt eines Typs traversiert werden muss, dass der ursprüngliche Pfad nicht auch traversieren durfte.

Beim erneuten Einkleben eines Pfades entsteht für jedes neue Graphmuster eine Menge weiterer Graphmuster. In jedes dieser Graphmuster wird jeder weitere Pfad wie beschrieben auf alle möglichen Weisen eingeklebt, so dass letztendlich eine Menge vollständiger Graphmuster entsteht, in denen alle Kombinationsmöglichkeiten des erneuten Einklebens von Pfaden gebildet wurden.

Erzeugung von Pfadteilausprägungen durch iterierte Anteile Beim zuvor beschriebenen erneuten Einkleben von Pfaden werden gegebenenfalls neue Pfade ergänzt. Dadurch kann prinzipiell der Fall eintreten, dass ein iterierter Anteil immer wieder eine Teilausprägung für einen solchen hinzugefügten Pfad erzeugen kann, die wieder durch einen konformen Pfad ergänzt wird. Würde dies nicht erkannt, terminierte das Verifikationsverfahren nicht.

Da der abstrakte Syntaxgraph eines Programms endlich ist, kann unter der Voraussetzung, dass die Ausführung eines Transformationsdiagramms terminiert, ein iterierter Anteil nur endlich oft angewendet werden. Des Weiteren ist unerheblich, wie häufig derselbe iterierte Anteil zum Entstehen einer Ausprägung im Prinzip desselben Pfades einer verbotenen Struktur beiträgt, solange er bereits durch eine Anwendung entstehen kann.

Eine Regelsequenz muss und darf damit nicht so durch Anwendung eines iterierten Anteils fortgesetzt werden, dass diese wie oben beschrieben eine Teilausprägung für einen Pfad erzeugt, der aufgrund einer vormaligen Anwendung desselben iterierten Anteils hinzugefügt wurde.

Erzeugung eines Elements der verbotenen Struktur Problematische iterierte Anteile müssen bei ihrer Anwendung mindestens ein Element der verbotenen Struktur erzeugen. Sonst würden unnötig lange Regelsequenzen berechnet und die Terminierung des Verfahrens wäre nicht sichergestellt. Kontrollflussbedingt notwendige Anwendungen müssen dagegen kein Element erzeugen.

Eindeutigkeit der Anwendungsstelle Nach dem Verkleben ist in jedem dabei entstandenen Graphmuster die Anwendungsstelle für die Rückwärtsanwendung der Regel vollständig bestimmt. Handelt es sich um einen iterierten Anteil, muss die durch Rückwärtsanwendung aus der Anwendungsstelle für seine rechte Seite resultierende Anwendungsstelle für die linke Seite eindeutig sein (vgl. Definition 22), da jeder iterierte Anteil auf einer Anwendungsstelle relativ zur Anwendungsstelle seines Vateranteils nur genau einmal angewendet wird.

Löscht der iterierte Anteil mindestens eine Objektvariable, führt seine Rückwärtsanwendung in jedem Fall zu einer eindeutigen Anwendungsstelle für die Vorwärtsanwendung.

Löscht der iterierte Anteil keine Objektvariable, muss der Teilgraph seiner Anwendungsstelle, der aus den bei seiner Anwendung unveränderten Elementen besteht, sich von allen solchen Teilgraphen der übrigen Anwendungsstellen desselben iterierten Anteils in mindestens einem Element unterscheiden, deren Anwendungen relativ zu derselben Anwendung des Vateranteils stattfinden. Sonst würde unterstellt, dass der iterierte Anteil vorwärts an derselben Stelle im Wirtsgraphen mehrfach ausgeführt wird.

Ist die Eindeutigkeit der neuen Anwendungsstelle nicht gegeben, ist eine Anwendung so nicht möglich und die Regelsequenz kann so nicht fortgesetzt werden, so dass das betreffende neue Graphmuster verworfen wird.

Durch das Verkleben entsteht für jede Regel, die für eine Fortsetzung einer Regelsequenz infrage kommt, eine Menge von Graphmustern, die alle zur Bildung einer neuen Regelsequenz verwendet werden. Für jedes neue Graphmuster wird die Regelsequenz kopiert und in der Kopie wird das aktuell erste Graphmuster durch das neue ersetzt. Im neuen Graphmuster durch das Verkleben hinzugefügte Elemente oder Neubindungen von Pfaden werden in alle späteren Graphmuster der Sequenz übertragen. Auf das neue Graphmuster am Anfang wird die Regel wie im nächsten Abschnitt 4.4.2.3 beschrieben rückwärts angewendet. Das dabei neu entstehende Graphmuster wird zusammen mit einer Transition für die Regelanwendung am Anfang der Sequenz eingefügt.

Abbildung 4.13 zeigt ein konkretes Beispiel anhand des SimpleExtract-Transformationsdiagramms aus Abbildung 3.1 auf Seite 51 und des *Forbidden-VariableAccess*-Graphmusters (siehe Abbildung 4.7). Bei dem SimpleExtract-Transformationsdiagramm gibt es nur einen erfolgreichen Ausführungspfad: Transformation Pattern (1) schlägt fehl, danach ist Transformation Pattern

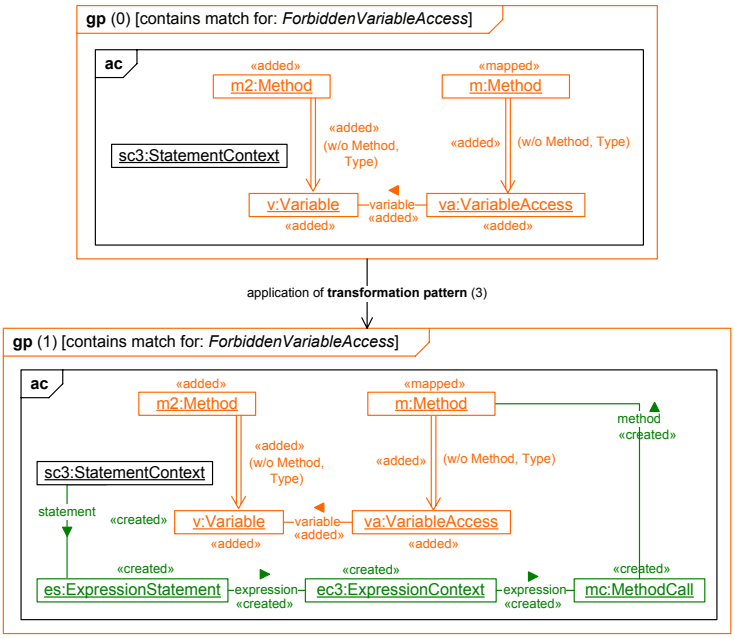

Abbildung 4.13: Eine unvollständige Regelsequenz für das *SimpleExtract*-Transformationsdiagramm und das *ForbiddenVariableAccess*-Graphmuster

(2) erfolgreich, danach ist Transformation Pattern (3) erfolgreich. Transformation Pattern (3) enthält keine iterierten Anteile, so dass sein nicht-iterierter Anteil als letztes angewendet werden muss.

Abbildung 4.13 zeigt mit gp (1) eines der Graphmuster, das beim Verkleben der rechten Seite des nicht-iterierten Anteils von Transformation Pattern (3) mit dem verbotenen Muster entsteht. Die orange gefärbten Elemente bilden die verbotene Struktur. Bis auf m1:Method des Graphmusters, das auf m:Method in der rechten Regelseite abgebildet ist, wurden seine übrigen Elemente hinzugefügt. Graphmuster gp (0) ist durch Rückwärtsanwendung der Regel entstanden.

Abbildung 4.14 zeigt eine Erweiterung der Regelsequenz um die Anwendung des nicht-iterierten Anteils von Transformation Pattern (2) des *SimpleExtract*-Transformationsdiagramms.

Die rechte Seite des nicht-iterierten Anteils wurde mit dem Graphmusters gp (0) verklebt, wobei die gebundenen Objektvariablen m:Method und sc3:StatementContext konsistent aufeinander abgebildet wurden. Des Weiteren wurde source:Method aus der rechten Regelseite auf m2:Method im Gra-

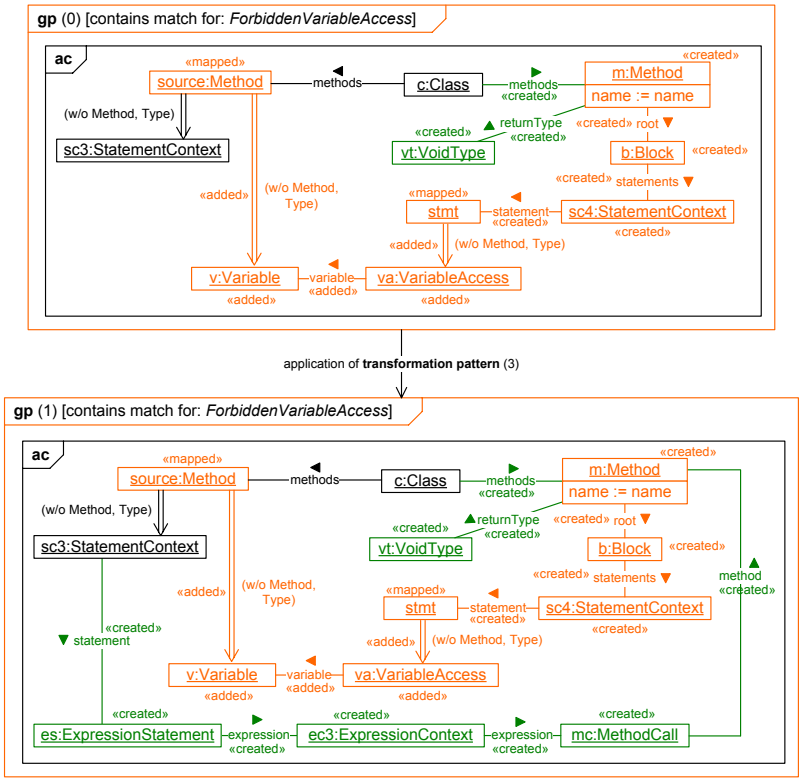

Abbildung 4.14: Eine erweiterte Regelsequenz für das *SimpleExtract*-Transformationsdiagramm und das *ForbiddenVariableAccess*-Graphmuster

phmuster abgebildet und dort entsprechend umbenannt. Die Elemente c:Class, vt:VoidType, b:Block, sc4:StatementContext, stmt sowie die mit ihnen verbundenen Linkvariablen wurden hinzugefügt.

Der Pfad der verbotenen Struktur zwischen m:Method und va:VariableAccess wurde aufgrund der neu hinzugekommenen Elemente neu gebunden an m:Method – root – b:Block – statements – sc4:StatementContext – statement – stmt = (w/o Method, Type) => va:VariableAccess. Der Pfad am Ende der Kette wurde zur Vervollständigung hinzugefügt und mit ≪added≫ markiert.

Die beim Verkleben hinzugefügten Elemente existieren nach dem jetzigen Kenntnisstand auch noch in den späteren Graphmustern der Regelsequenz. Daher werden sie dort analog hinzugefügt. Auch die Neubindung des Pfades der verbotenen Struktur wird in alle späteren Graphmuster übertragen, ebenso wie die Umbenennung der Objektvariable m2:Method in source:Method, die

aufgrund der Abbildung von m2 auf source beim Verkleben in gp (0) erfolgt. In diesem Fall gibt es nur ein späteres Graphmuster gp (1), in dem die Änderungen vorgenommen werden.

4.4.2.3 Regel rückwärts anwenden

Nach dem Verkleben wird in jeder Regelsequenz die Graphtransformationsregel rückwärts auf das Graphmuster angewendet, so dass ein neues Graphmuster entsteht, welches den Wirtsgraphen vor Anwendung der Regel charakterisiert.

In Abschnitt 3.3 wurde die Anwendung von Graphtransformationsregeln nur für einfache Graphen definiert. Bei den Anwendungsbedingungen von Graphmustern handelt es sich aber im Prinzip um spezialisierte Regelgraphen. Graphtransformationsregeln können auch auf solche Graphen angewendet werden, indem zum einen in Anwendungsstellen Pfade an Ausprägungen gebunden werden können, die auch konforme Pfade enthalten, und zum anderen ihre Ausführung ohne Berücksichtigung von Transformationsaufrufen und losgelöst von einem Ausführungszustand definiert wird.

Bei der Rückwärtsanwendung werden die rechte und linke Seite der Regel vertauscht, so dass bei Vorwärtsanwendung erzeugte Elemente gelöscht und gelöschte Elemente erzeugt werden. Dabei wird das Graphmuster zunächst kopiert und dann die Anwendung vorgenommen. Das neue Graphmuster wird mit einer Transition für die Regelanwendung in die Regelsequenz eingefügt.

Dies wird bereits in Abbildung 4.13 gezeigt. Durch Rückwärtsanwendung des nicht-iterierten Anteils von Transformation Pattern (3) auf die in gp (1) enthaltene Anwendungsstelle für seine rechte Regelseite entsteht das darüber abgebildete Graphmuster gp (0). In der erweiterten Regelsequenz aus Abbildung 4.14 enthält gp (0) eine Anwendungsstelle für die rechte Seite des nicht-iterierten Anteils von Transformation Pattern (2), der anhand derer rückwärts angewendet wird. Dabei entsteht ein neues Graphmuster, das zusammen mit einer Transition für die Regelanwendung am Anfang der Regelsequenz eingefügt wird. Damit entspricht die Regelsequenz dem bereits in Abbildung 4.4 gezeigten Gegenbeispiel, noch ohne die negative Anwendungsbedingung.

Bei der Vorwärtsanwendung einer Graphtransformationsregel können hier, anders als in [Sch06] (siehe auch Abschnitt 2.4.3), durch das Löschen von Objekten lose Kanten entstehen, die ebenfalls gelöscht werden, ohne dass dies explizit in der Regel spezifiziert ist (siehe Abschnitt 3.3.7). Solche Löschungen von Kanten können bei der hier beschriebenen Rückwärtsanwendung einer Regel nicht berücksichtigt werden. Somit beschreibt ein Graphmuster, das durch Rückwärtsanwendung einer Regel ermittelt wird, den Wirtsgraphen vor einer

Anwendung der Regel nicht unbedingt vollständig.

Graphmuster tun dies allerdings ohnehin nicht: sie beschreiben lediglich Strukturen, die mindestens im Wirtsgraphen enthalten sein müssen. Zudem können durch das Löschen von Kanten offensichtlich keine verbotenen Strukturen entstehen, so dass dies unproblematisch ist. Auch eine zu erhaltende Struktur kann nur dann durch das Löschen einer losen Kante zerstört werden, wenn das Objekt, dessen Löschung zum Entstehen der losen Kante führt, ebenfalls Bestandteil der Struktur ist, so dass das Zerstörungspotential in jedem Fall anhand der Regelspezifikation erkennbar ist. Für das hier entwickelte Verfahren ist daher die in [Sch06] vorgenommene Einschränkung der Anwendbarkeit von Regeln nicht notwendig.

Darüber hinaus können Regeln in diesem Ansatz bestehende Links, deren zugrunde liegende Assoziation an einer Seite eine 0..1-Kardinalität hat, überschreiben, das heißt implizit löschen und neu erzeugen (siehe Abschnitt 3.3.7). Nur anhand einer solchen Regel kann damit nicht entschieden werden, ob ein solcher Link vor ihrer Anwendung bereits existiert hat oder nicht. In Bezug auf die Rückwärtsanwendung hat dies dieselben Konsequenzen wie lose Kanten.

4.4.2.4 Ergebnis

Das Ergebnis der Berechnungen ist eine Menge von Regelsequenzen für jeden Ausführungspfad eines Transformationsdiagramms und jedes verbotene Graphmuster. Die Regelsequenzen enthalten mindestens in ihrem letzten Graphmuster eine verbotene Struktur von der mindestens ein Element durch eine Regel der Sequenz erzeugt wird.

Regelsequenzen, in denen bereits zu Beginn, im ersten Graphmuster, eine verbotene Struktur existiert, werden verworfen. Die übrigen Regelsequenzen sind Kandidaten für Gegenbeispiele und werden an den folgenden Schritt des Verfahrens weitergereicht.

4.4.3 Kennzeichnung optionaler Elemente

Die Graphmuster einer Regelsequenz können Elemente enthalten, die zu dem Zeitpunkt der Transformationsausführung, den das Graphmuster repräsentiert, noch nicht oder nicht mehr zwingend existieren müssen. Dies kann zum einen aufgrund von Transformationsaufrufen und zum anderen aufgrund von nicht eingeplanten Anwendungen iterierter Anteile der Fall sein, die ein Element löschen können, das für später geplante Anwendungen nicht mehr benötigt

wird. Ebenso können Ausprägungen von Pfaden, die zur linken Seite einer Regel gehören, durch ihre Anwendung zerstört werden.

Die Aufgabe dieses Schrittes ist es, solche Elemente in den jeweiligen Graphmustern als optional zu kennzeichnen.

4.4.3.1 Auswirkungen von Transformationsaufrufen

Transformationsdiagramme können während ihrer Ausführung andere Transformationsdiagramme (synchron) aufrufen. Die Ausführung des aufrufenden Transformationsdiagramms wird dazu unterbrochen, das aufgerufene Diagramm wird vollständig ausgeführt und danach wird die Ausführung des Aufrufers fortgesetzt.

Ein aufgerufenes Transformationsdiagramm kann aus Sicht des Aufrufers beliebige Auswirkungen auf den Wirtsgraphen haben. Es kann

- Elemente löschen, die der Aufrufer vor dem Aufruf zur Anwendung von Regeln benötigt hat oder die durch diese Anwendungen erzeugt wurden,
- eine zuvor durch den Aufrufer erzeugte verbotene Struktur wieder korrigieren beziehungsweise ihr Entstehen vereiteln, indem eines ihrer Elemente gelöscht wird,
- Elemente löschen, die spätere Regelanwendungen des Aufrufers zwingend benötigt hätten und dadurch das Entstehen oder die Korrektur einer verbotenen Struktur vereiteln,
- Elemente erzeugen, die spätere Anwendungen von Regeln des Aufrufers ermöglichen,
- so einen Teil einer verbotenen Struktur erzeugen, dass diese erst durch die Ausführung beider Diagramme entsteht, während jedes der Diagramme für sich genommen dies nicht vermag.

Ein aus einem iterierten Anteil heraus aufgerufenes Transformationsdiagramm kann insbesondere auch Elemente erzeugen, die eine erneute Anwendung dieses Anteils ermöglichen, so dass hier die Terminierung nicht garantiert ist. Dies hat allerdings keine Auswirkungen auf das Verfahren, da zunächst wie in Abschnitt 4.4.2 beschrieben alle Kombinationsmöglichkeiten problematischer Regelanwendungen gebildet wurden und die möglichen Auswirkungen von Transformationsaufrufen erst jetzt, nachträglich berücksichtigt werden.

Um diese Auswirkungen präzise zu berücksichtigen, sind weitere Analysen aufgerufener Diagramme notwendig, die im Rahmen dieser Arbeit nicht mehr untersucht wurden. Stattdessen betrachtet das hier entwickelte Verfahren Transformationsdiagramme einzeln und schätzt die Auswirkungen von aufgerufenen Diagrammen pessimistisch ab. Dies geschieht so, dass eventuell Gegenbeispiele ermittelt werden, die so nicht auftreten können (*false-positives*) – nicht erkannte Gegenbeispiele (*false-negatives*) sind ausgeschlossen.

Abbildung 4.15 zeigt ein weiteres Gegenbeispiel für ein leicht angepasstes *SimpleExtract*-Transformationsdiagramm und das verbotene *ForbiddenVariableAccess*-Graphmuster. Das Transformationsdiagramm wurde so angepasst, dass Transformation Pattern (2) einen Aufruf eines anderen Transformationsdiagramms (*SomeTransformation*) enthält.

Dieser Aufruf findet statt, nachdem alle Modifikationen des nicht-iterierten Anteils des Transformation Pattern durchgeführt wurden. Das in Abbildung 4.15 gezeigte Graphmuster gp (1) repräsentiert den relevanten Ausschnitt des Wirtsgraphen zu diesem Zeitpunkt. Das nächste Graphmuster gp (2) zeigt den Ausschnitt nach dem Aufruf. Die Transition zwischen den Graphmustern repräsentiert den Aufruf und die Ausführung von *SomeTransformation*.

In gp (1) sind mit v:Variable, va:VariableAccess und den damit verbundenen Linkvariablen Elemente enthalten, die Bestandteil der verbotenen Struktur sind, aber nicht durch das Transformationsdiagramm selbst erzeugt werden. Sie müssen entweder bereits im Wirtsgraphen vorhanden gewesen sein oder durch ein aufgerufenes Diagramm erzeugt werden. Der Aufruf von *SomeTransformation* ist der einzige mögliche Aufruf auf dem Ausführungspfad. Die besagten Elemente können durch ihn erzeugt werden oder bereits vorher existiert haben.

Letzteres wird dadurch angezeigt, dass die besagten Elemente in den Graphmustern vor dem Aufruf (gp (0) und gp (1)) als *optional* gekennzeichnet sind (gestrichelte Darstellung). Nach dem Aufruf müssen sie existieren, damit das verbotene Muster am Ende der Regelsequenz definitiv vollständig ist. Daher sind sie in den späteren Graphmustern nicht mehr optional. Da sie durch den Aufruf erzeugt worden sein können, sind sie dort mit ≪opt. created≫ (Kurzform für *optionally created*) markiert. Mit Elementen, die nicht zur verbotenen Struktur gehören, sondern Bestandteil späterer Anwendungsstellen sind und vor einem Aufruf noch nicht benötigt werden, wird analog verfahren.

Des Weiteren können durch das aufgerufene Diagramm Elemente gelöscht worden sein, die für die weiteren Regelanwendungen des Gegenbeispiels nicht mehr benötigt werden. Dies trifft auf die Objektvariablen c:Class, vt:VoidType und die mit ihnen verbundenen Linkvariablen in gp (2) zu. Sie sind nach dem Aufruf optional, während sie davor existiert haben müssen. Das bedeutet auch,

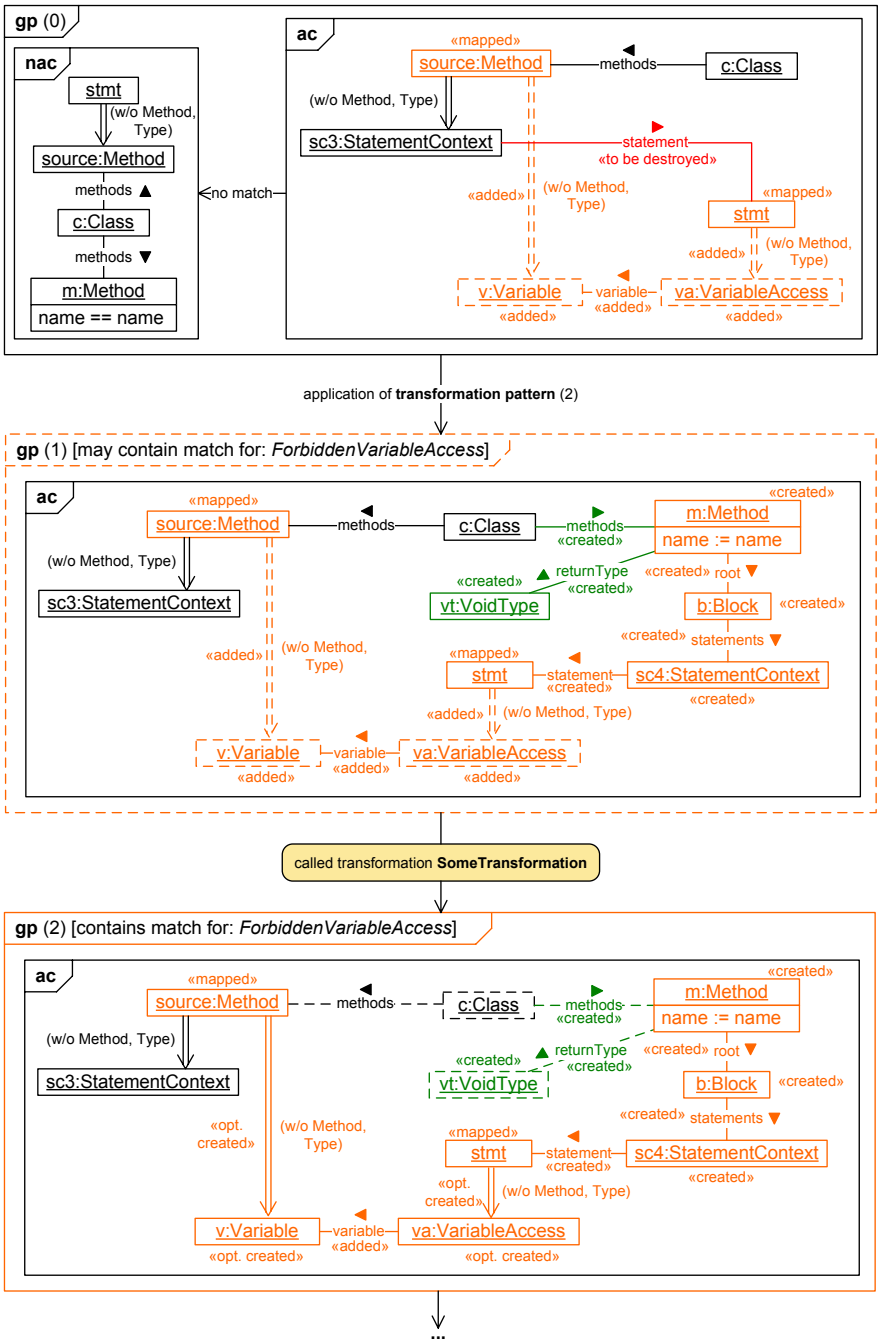

Abbildung 4.15: Ein Gegenbeispiel mit Transformationsaufruf

dass sie nach einem Aufruf nicht mehr zwingend für weitere Anwendungen iterierter Anteile zur Verfügung stehen müssen, die zum Beispiel eine bis dahin aufgetretene Verletzung eines Kriteriums wieder korrigieren oder weitere Anwendungen, die zu einer Kriterienverletzung beitragen, verhindern würden.

Das verbotene Muster ist in gp (2) vollständig. Wenn alle in gp (1) optionalen Elemente bereits existierten, so wäre es bereits hier vollständig. Dies wird durch die gestrichelte orange Umrandung von gp (1) ausgedrückt. Dass das verbotene Muster in gp (2) vollständig ist, zeigt, dass die weiteren Regelanwendungen der Sequenz nichts mehr dazu beitragen. Es kann daher sein, dass es durch das aufgerufene Transformationsdiagramm vervollständigt wird. Dass dies möglich ist, müsste sich bei der Verifikation dieses aufgerufenen Diagramms ergeben. Kann für dieses Diagramm jedoch kein Gegenbeispiel gefunden werden, so wäre dies nicht möglich. Dann müsste mindestens eines der vor dem Aufruf optionalen Elemente bereits existiert haben.

4.4.3.2 Transformationsaufrufe ermitteln

Zwischen zwei in einer Regelsequenz geplanten aufeinanderfolgenden Regelanwendungen können andere Transformationsdiagramme aufgerufen werden. Dies ist definitiv der Fall, wenn eine eingeplante Regel Transformationsaufrufe enthält. Da Regelsequenzen bisher nur solche Regelanwendungen enthalten, die zum Entstehen der verbotenen Struktur beitragen oder die aufgrund des Kontrollfluss erfolgen müssen, können darüber hinaus bei entsprechender Beschaffenheit des Wirtsgraphen zwischen zwei geplanten Anwendungen weitere iterierte Anteile angewendet werden, die nicht eingeplant sind. Enthalten diese Transformationsaufrufe, so können diese ebenfalls zwischen zwei geplanten Regelanwendungen ausgeführt werden.

Daher werden hier alle zwischen zwei geplanten Regelanwendungen potentiell möglichen Transformationsaufrufe bestimmt und durch Aufruftransitionen in der Regelsequenz repräsentiert ohne dass weitere Regelanwendungen eingeplant oder die bestehenden Graphmuster verändert werden: die Ausführung von Transformationsaufrufen wird unterstellt ohne dass die dafür nötige Beschaffenheit des Wirtsgraphen exakt bestimmt wird. Dies führt zu einer pessimistischen Abschätzung des Verhaltens von Transformationsdiagrammen.

Abbildung 4.16 zeigt schematisch, welche iterierten Anteile zwischen geplanten Regelanwendungen potentiell angewendet werden können. Ausgehend von der in den Abschnitten 3.3.8 und 3.3.9 formalisierten Semantik gibt es drei Möglichkeiten bei zwei hintereinander geplanten Regelanwendungen:

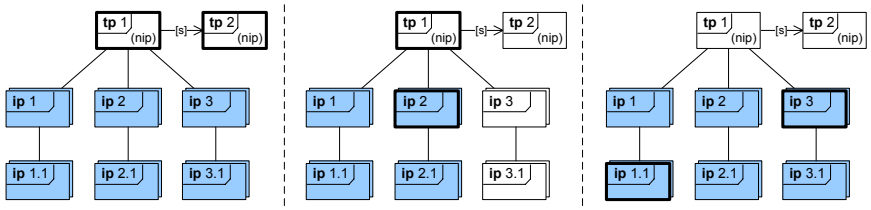

Abbildung 4.16: Zwischen zwei Anteilen anwendbare iterierte Anteile (schematische Darstellung)

1. es sind nicht-iterierte Anteile aufeinanderfolgender erfolgreicher Transformation Pattern eines Ausführungspfades,

2. der erste ist ein nicht-iterierter Anteil und der zweite ein iterierter Anteil auf oberster Hierarchieebene desselben Transformation Pattern oder

3. beide sind iterierte Anteile desselben Transformation Pattern, wobei der zweite auf oberster Hierarchieebene sein muss.

Das linke Beispiel der Abbildung zeigt die erste Möglichkeit. Zwischen den fett umrandeten nicht-iterierten Anteilen können alle iterierten Anteile des ersten Transformation Pattern, dargestellt als Baumstruktur, ausgeführt werden (blau hinterlegt).

Im mittleren Beispiel sind der nicht-iterierte Anteil des ersten Transformation Pattern und sein iterierter Anteil ip 2 geplant. Zwischen zwei Anwendungen dieser Anteile können die blau gefärbten iterierten Anteile ausgeführt werden. Dazu gehört auch der iterierte Anteil ip 2 selbst und alle in ihm transitiv enthaltenen Anteile; auch wenn ip 2 eingeplant ist, kann es aufgrund der Iteration weitere Anwendungen für ihn auch vor der geplanten Anwendung geben, die keinen Einfluss auf die verbotene Struktur haben müssen. Der Anteil ip 3 und alle darin enthaltenen Anteile können erst wieder nach ip 2 angewendet werden.

Im dritten Beispiel sind die beiden fett umrandeten iterierten Anteile ip1.1 und ip 3 eingeplant und die dazwischen zusätzlich anwendbaren iterierten Anteile sind wiederum blau gefärbt. Analog zum mittleren Beispiel kann es auch hier weitere Anwendungen für ip 1 und ip 1.1 sowie für ip 3 und darin enthaltene Anteile geben.

Gibt es mindestens einen möglichen Transformationsaufruf, wird eine Aufruftransition eingeführt. Abbildung 4.17 stellt dies schematisch dar. Die Anwendung von tp (1) wurde neu eingeplant. Zwischen tp (1) und tp (2) kann die Transformation *SomeTransformation* aufgerufen worden sein. Dies geschieht

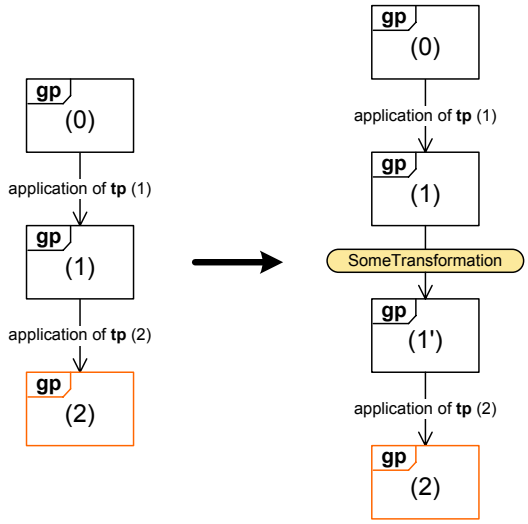

Abbildung 4.17: Einfügen einer Aufruftransition (schematische Darstellung)

am Ende der Anwendung von tp (1). Daher wird das Graphmuster nach dessen Anwendung, gp (1), kopiert und wie in der Abbildung gezeigt als gp (1') in die Sequenz eingefügt. Die möglichen Auswirkungen der Aufrufe auf den Status von Elementen in den Graphmustern, werden wie folgt berücksichtigt.

4.4.3.3 Auswirkungen von Transformationsaufrufen berücksichtigen

Abbildung 4.18 zeigt links schematisch eine Regelsequenz, bei dem eine Objektvariable der verbotenen Struktur durch keine der Regelanwendungen der Sequenz gebunden oder erzeugt wird. Dieses Element kann von Anfang an im Wirtsgraphen existiert haben. Genauso kann es durch ein aufgerufenes Transformationsdiagramm erzeugt worden sein. Nach dem letzten Transformationsaufruf muss es existieren, damit die verbotene Situation am Ende der Regelsequenz vollständig ist.

Kann während der Regelsequenz kein Transformationsaufruf stattfinden, so muss das Objekt von Anfang an existieren. Ansonsten ist die Objektvariable wie in der Abbildung gezeigt vor dem ersten Transformationsaufruf optional. Das Objekt kann durch eine der ersten aufgerufenen Transformationen erzeugt werden. Daher wird es in allen Graphmustern nach der ersten Aufruftransition als optional erzeugt (≪opt. created≫) aber weiterhin optional markiert. Die bei der letzten Aufruftransition aufgerufenen Transformationen können das Objekt ebenso erzeugt haben. Danach muss es im Unterschied zu vorher

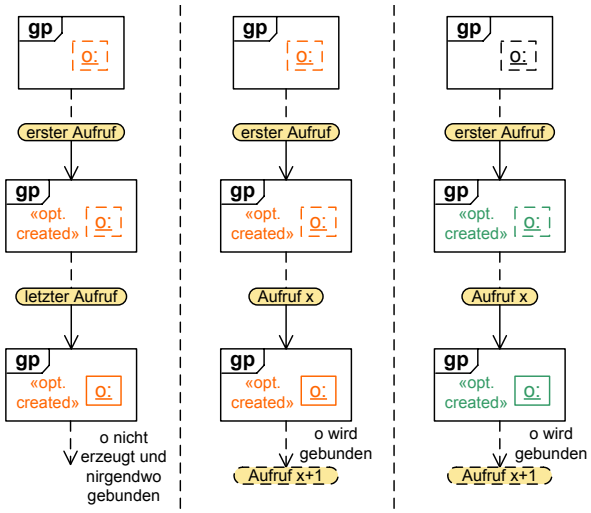

Abbildung 4.18: Status von Objektvariablen in Graphmustern (I)

existieren, damit die verbotene Struktur vollständig werden kann.

Das mittlere Beispiel der Abbildung zeigt eine Regelsequenz, bei dem ein Objekt der verbotenen Struktur nicht erzeugt, aber irgendwann gebunden wird. Gibt es in der Sequenz bevor dies geschieht keine Transformationsaufrufe, so muss das Objekt von Beginn an existieren. Ansonsten ist es analog vor der ersten Aufruftransition optional, danach optional erzeugt und weiterhin optional bis zur letzten Aufruftransition (Aufruf x) bevor das Objekt von einer Regelanwendung gebunden wird. Ab dann ist es nicht mehr optional, sondern muss vorhanden sein.

Das rechte Beispiel der Abbildung zeigt denselben Fall wie das mittlere Beispiel, nur für ein Objekt, das nicht zur verbotenen Struktur, sondern nur zu einer Anwendungsstelle einer in der Sequenz eingeplanten Regel gehört. Es kann analog vor dem ersten Transformationsaufruf existiert haben, nach dem ersten Aufruf erzeugt worden sein und nach dem letzten Aufruf bevor das Objekt für eine Regelanwendung benötigt wird, muss es existieren. Solche optional erzeugten Objekte (und Links) werden in einem etwas dunkleren grün gezeichnet, als definitiv durch Regelanwendungen erzeugte Elemente.

Aufgerufene Transformationen können Objekte nicht nur erzeugen, sondern auch löschen. Dies kann beliebige Objektvariablen eines Graphmusters betreffen, auch solche, die Bestandteil der verbotenen Struktur sind oder für spätere Regelanwendungen notwendig sind, die zum Entstehen der verbotenen Struktur beitragen. Das Verfahren geht pessimistisch davon aus, das solche Objekt-

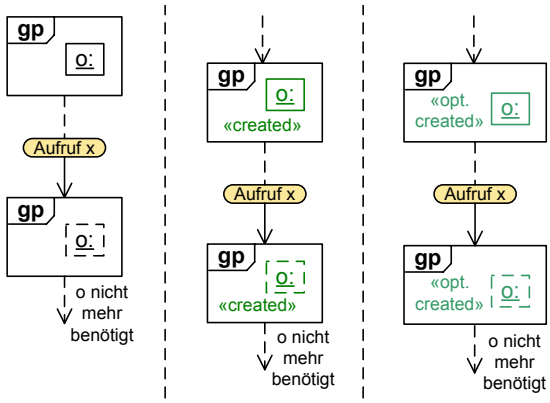

Abbildung 4.19: Status von Objekten in Graphmustern (II)

variablen nicht gelöscht werden, so dass die verbotene Struktur wie geplant entstehen kann. Dies kann zu Gegenbeispielen führen, die tatsächlich nicht auftreten können (false-positives).

Dazu konsistent ist, dass unterstellt wird, dass Objektvariablen, die ab einem bestimmten Zeitpunkt nicht mehr für das Erzeugen einer verbotenen Struktur benötigt werden, durch aufgerufene Transformationen gelöscht worden sein können. Solche Objektvariablen stehen dann auch nicht mehr zwingend für weitere, zum Beispiel korrigierende Anwendungen iterierter Anteile zur Verfügung.

Abbildung 4.19 zeigt links schematisch eine Regelsequenz, bei der eine Objektvariable vor der Aufruftransition Aufruf x für eine Regelanwendung benötigt wird und nach dieser Aufruftransition nicht mehr. Damit könnte es durch eine aufgerufene Transformation gelöscht worden sein, ohne dass die Regelsequenz unmöglich wird. In einem solchen Fall wird die Objektvariable in allen Graphmustern nach der Aufruftransition als optional gekennzeichnet.

Das mittlere Beispiel der Abbildung zeigt den Fall, dass eine Objektvariable durch irgendeine Regelanwendung der Sequenz erzeugt wird. Die Objektvariable wird nach der Aufruftransition nicht mehr benötigt, so dass es durch eine dabei aufgerufene Transformation gelöscht worden sein kann. Daher wird sie in allen Graphmustern nach der Transition als optional gekennzeichnet.

Das rechte Beispiel zeigt denselben Fall wie in der Mitte, nur dass es sich um eine optional erzeugte Objektvariable handelt, die zuvor durch eine aufgerufene Transformation erzeugt und nun durch eine aufgerufene Transformation wieder gelöscht worden sein kann.

Alle Ausführungen dieses Abschnitts gelten nicht nur für Objektvariablen in Graphmustern, sondern analog für Linkvariablen und Pfade.

4.4.3.4 Löschungen durch ungeplante Anwendungen

Regelsequenzen beschreiben nur Ausschnitte eines Wirtsgraphen. Tatsächlich kann er weitere Elemente enthalten, so dass nicht eingeplante iterierte Anteile so anwendbar sind, dass sie Elemente löschen, die nicht Bestandteil der verbotenen Struktur sind und nach dem Aufruf nicht mehr für Regelanwendungen benötigt werden.

Um dies zu überprüfen, werden die Graphmuster einer Regelsequenz der Reihe nach durchlaufen. In jedem Graphmuster werden alle nicht-optionalen, nicht-erzeugten Elemente bestimmt, die nicht Bestandteil der verbotenen Struktur sind und von keiner späteren Regelanwendung benötigt werden.

Gibt es solche Elemente, werden die iterierten Anteile bestimmt, die zwischen dem Anteil, dessen Anwendung zum aktuellen Graphmuster führt, und dem Anteil, der auf das Graphmuster als nächstes angewendet wird, angewendet werden könnten (vgl. Abschnitt 4.4.3.2). Von den so bestimmten iterierten Anteilen werden die ermittelt, die mindestens eines der infrage kommenden Elemente löschen könnten. Dies ist möglich, wenn sie ein typkonformes Element oder eines, das Bestandteil einer Ausprägung für einen infrage kommenden Pfad sein kann, löschen können (vgl. Abschnitt 4.4.1).

Für alle solchen Anteile muss auf alle möglichen Weisen unter Vervollständigung von Anwendungsstellen unterstellt werden, dass sie auf das Graphmuster angewendet werden. Löschen sie dabei eine oder mehrere infrage kommende Elemente ohne gleichzeitig nicht infrage kommende Elemente zu löschen, so müssen die gelöschten Elemente in allen späteren Graphmustern als optional gekennzeichnet werden.

Solche Regelanwendungen werden nicht in die Regelsequenzen aufgenommen. Lediglich ihre möglichen Auswirkungen auf den Status von Elementen werden berücksichtigt. Durch diese erneute pessimistische Abschätzung des Verhaltens eines Transformationsdiagramms können false-positives entstehen.

Abbildung 4.20 zeigt dies schematisch anhand eines Beispiels. Auf der rechten Seite wird eine Regelsequenz dargestellt, deren Graphmuster gp (2) überprüft wird. Links daneben ist ein Ausschnitt des zugehörigen Ausführungspfades dargestellt. Der darin fett umrandete iterierte Anteil ip 1 wird in der Regelsequenz angewendet, um gp (2) zu erhalten. Auf gp (2) wird der nicht-iterierte Anteil von tp 2 angewendet.

Zwischen ip 1 und tp 2 können zusätzlich die blau gefärbten iterierten Anteile angewendet werden. Angenommen, diese können jeweils mindestens ein dafür infrage kommendes Element von gp (2) löschen, so muss für alle blau gefärbten Anteile wie im rechten Teil der Abbildung angedeutet unterstellt werden, dass

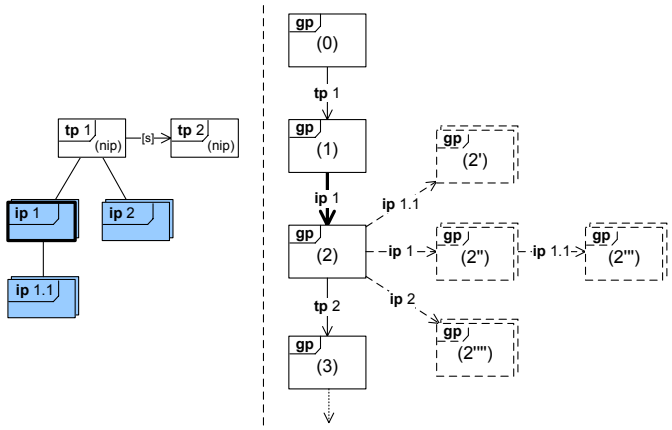

Abbildung 4.20: Überprüfung auf Löschungen durch iterierte Anteile

sie auf gp (2) als nächstes angewendet werden.

Ein iterierter Anteil wird immer relativ zu einer Anwendungsstelle für seinen Vateranteil angewendet und verwendet die darin bestimmten Bindungen für seine Schnittstellenobjektvariablen. Da im Beispiel ip 1 zuletzt angewendet wurde, kann relativ zu dieser Anwendungsstelle ip 1.1 angewendet werden. Die linke Regelseite von ip 1.1 wird daher mit gp (2) verklebt, ohne dass Elemente auf im Graphmuster erzeugte oder optionale Elemente abgebildet werden. Dabei entsteht die Menge von Graphmustern gp (2'). Um zu überprüfen, ob durch die Anwendung auch ein Pfad gelöscht werden kann, werden alle Pfade in den Graphmustern wie in Abschnitt 4.4.2.2 beschrieben auf Teilausprägungen abgebildet und eingeklebt. Wird durch die Anwendung ein infrage kommendes Element oder ein Element einer der neu berechneten Pfadausprägungen gelöscht, so wird das gelöschte Element in gp (2) als optional markiert.

Auf dieselbe Weise wird eine weitere Anwendung von ip 1, relativ zur Anwendungsstelle für seinen Vateranteil, den nicht-iterierten Anteil von tp 1, unterstellt. Relativ zu allen dabei ermittelten Anwendungsstellen werden weitere Anwendungen von ip 1.1 unterstellt und überprüft. Schließlich wird auch ip 2 auf diese Weise überprüft.

4.4.3.5 Löschung von Pfadausprägungen durch geplante Anwendungen

Für jede in einer Regelsequenz angewendete Regel muss überprüft werden, ob sie eine Ausprägung eines Pfades ihrer linken Seite zerstören kann. Dazu werden im Graphmuster vor der Anwendung alle Elemente bestimmt, die

Bestandteil einer Ausprägung eines solchen Pfades sein können, ohne dass dabei Kardinalitäten verletzt werden. Handelt es sich dabei um Elemente, die durch die Anwendung gelöscht werden, so kann auf diese Weise auch die Pfadausprägung gelöscht werden und der Pfad wird als optional gekennzeichnet. Muss ein gelöschtes Element sogar Bestandteil einer Ausprägung eines Pfades sein, so wird der Pfad entfernt. Wird er zur Anwendung später geplanter Anwendungen der Regelsequenz benötigt, so wird die Regelsequenz verworfen.

4.4.4 Vorwärtsüberprüfung und Vervollständigung

Das Ziel des Verfahrens ist, Beispiele für die Verletzung von Kriterien zu konstruieren, in denen eine Transformation eine verbotene Struktur erzeugt. Daher wird bei der Konstruktion von Regelsequenzen die Anwendbarkeit von problematischen Anteilen wohlwollend unterstellt.

Bisher wird nicht geprüft, ob in den Situationen, in denen die Anwendung problematischer Anteile unterstellt wird, andere Anteile garantiert zur Anwendung kämen, die die unterstellte Anwendung unmöglich machten. Wäre dem so, hätte die Regelsequenz so auf keinen Fall stattfinden können und muss nachträglich verworfen werden.

Dabei ist entscheidend, dass ein solcher *Widerspruch* ohne Vervollständigung von Anwendungsstellen und ohne Verwendung optionaler Elemente zwingend auftreten *muss*. Wenn er nicht zwingend auftreten *muss*, sondern lediglich auftreten *kann*, ist die unterstellte problematische Ausführung des Transformationsdiagramms dennoch möglich.

Darüber hinaus kann eine Regelsequenz noch unberücksichtigte Anwendungsstellen für korrigierende und neutrale iterierte Anteile enthalten, die nicht zu einem Widerspruch in der Ausführungsreihenfolge führen. Eine etwaige korrigierende Wirkung muss dennoch berücksichtigt werden, da sie zu Regelsequenzen führen kann, in denen die verbotene Struktur nicht mehr vollständig ist. In einem solchen Fall hätte die verbotene Struktur aufgrund (der Iteration) von korrigierenden Anteilen gar nicht entstehen können und muss ebenfalls nachträglich verworfen werden. Auch wenn ein neutraler iterierter Anteil noch anwendbar ist, wird seine Anwendung zusätzlich durchgeführt.

Die Graphmuster einer Regelsequenz werden vom Anfang zum Ende durchlaufen. In jedem Graphmuster wird geprüft, ob es eine vollständige Anwendungsstelle für einen iterierten Anteil gibt, der als nächstes, definitiv vor dem als nächstes geplanten Anteil angewendet werden muss. Diese Anwendungsstelle darf keine optionalen und keine durch Regelanwendungen erzeugten Elemente (≪created≫) enthalten.

Sollte es eine weitere Anwendungsstelle für einen als nächstes bereits geplanten iterierten Anteil geben, so wird diese nicht vor der geplanten Anwendung behandelt. Gibt es für einen iterierten Anteil mehrere Anwendungsstellen im Wirtsgraphen, so wird er hintereinander auf alle Anwendungsstellen angewendet, allerdings ist die Reihenfolge nicht-deterministisch. Daher muss nicht zwingend eine bestimmte Anwendung vor einer anderen erfolgen. In solchen Fällen wird der bereits geplanten Anwendung der Vorrang gegeben.

Gibt es eine nicht behandelte vollständige Anwendungsstelle für einen iterierten Anteil sind drei Fälle zu unterscheiden:

1. Löscht die Anwendung ein Element, das für eine spätere Regelanwendung der Sequenz noch benötigt wird, so ist diese Regelsequenz nicht wie geplant möglich und sie muss verworfen werden.

2. Löscht eine nachzuholende Anwendung ein Element der verbotenen Struktur, das nicht durch einen späteren Transformationsaufruf wieder erzeugt werden kann, so ist die Regelsequenz ebenfalls zu verwerfen, da die verbotene Struktur am Ende nicht vollständig vorliegen kann.

3. Sonst wird die Anwendung des Anteils nachgeholt. Dabei entsteht ein neues Graphmuster, das in die Sequenz zusammen mit einer Transition für die Regelanwendung direkt nach dem aktuellen Graphmuster eingefügt wird. Analog wie in Abschnitt 4.4.2.2 beschrieben, werden bei der Anwendung erzeugte Elemente in die späteren Graphmuster aufgenommen. Gelöschte Elemente werden entfernt. Ebenso wird der Status (optional oder nicht optional) der in späteren Graphmustern hinzugekommenen Elemente wie in Abschnitt 4.4.3 beschrieben überprüft.

Danach wird mit dem neu erzeugten Graphmuster wie gerade beschrieben weiter verfahren, bis das letzte Graphmuster der Sequenz erreicht ist und keine weiteren Anwendungen möglich sind.

4.4.5 Generierung negativer Anwendungsbedingungen

Bei der Konstruktion der Regelsequenzen wurden bisher nur die erfolgreichen Transformation Pattern eines Ausführungspfades und deren Anteile berücksichtigt. Darüber hinaus können Transformation Pattern auf einem Ausführungspfad fehlschlagen. Das bedeutet, dass es zum Zeitpunkt des Fehlschlagens keine Anwendungsstelle für den nicht-iterierten Anteil eines solchen Transformation Pattern im Wirtsgraphen gibt. Dies wird durch negative Anwendungsbedingungen von Graphmustern ausgedrückt.

Die Graphmuster einer Regelsequenz definieren, welche Strukturen zu verschiedenen Zeitpunkten mindestens im Wirtsgraphen enthalten sein müssen, damit die problematische Regelsequenz möglich ist. Darüber hinaus können weitere Elemente im Wirtsgraphen existieren, die zum Beispiel die Anwendung weiterer Regeln ermöglichen, die wiederum die Ausführung der geplanten Regelsequenz vereiteln würden. Eine Regelsequenz ist nur möglich, wenn diese Anwendungsstellen nicht existieren. Daher muss ihre Existenz durch negative Anwendungsbedingungen verboten werden, um eine Regelsequenz vollständig zu charakterisieren.

4.4.5.1 Fehlgeschlagene Transformation Pattern

Schlägt ein Transformation Pattern auf einem Ausführungspfad fehl, so gibt es zu dem Zeitpunkt keine Anwendungsstelle für seinen nicht-iterierten Anteil.

Das Graphmuster, auf das der nicht-iterierte Anteil des nächsten erfolgreichen Transformation Pattern des Ausführungspfades angewendet wird, repräsentiert den Zeitpunkt in der Regelsequenz, zu dem es keine Anwendungsstelle für das fehlgeschlagene Transformation Pattern geben darf. Für dieses Graphmuster wird daher eine negative Anwendungsbedingung generiert.

Die negative Anwendungsbedingung besteht aus einer Kopie der linken Regelseite des nicht-iterierten Anteils des fehlgeschlagenen Transformation Pattern. Das Graphmuster gp (0) in Abbildung 4.4 zeigt ein Beispiel dafür.

Nach dem Generieren einer negativen Anwendungsbedingung muss überprüft werden, ob durch sie ein Widerspruch zur geplanten Regelsequenz entstanden ist. Ein solcher Fall kann zum Beispiel eintreten, wenn ein Ausführungspfad durch ein Transformationsdiagramm ermittelt wurde, der aufgrund der Transitionen möglich ist, aber aufgrund der Transformation Pattern des Pfades nicht zustande kommen kann.

Ein Widerspruch liegt vor, wenn das Graphmuster, zu dem die negative Anwendungsbedingung generiert wurde, ohne optionale und ohne erzeugte Elemente eine Anwendungsstelle für diese negative Anwendungsbedingung enthält. Ist dies der Fall, muss die Regelsequenz verworfen werden.

4.4.5.2 Ausschluss von Korrektur und Vereitelung

Zwischen den geplanten Regelanwendungen einer Regelsequenz *können* weitere iterierte Anteile so anwendbar sein, dass sie die geplante Regelsequenz vereiteln oder eine Korrektur herbeiführen. Sie *müssen* aber nicht so anwendbar sein.

Zur Überprüfung wird im Wesentlichen genauso wie in Abschnitt 4.4.3.4 beschrieben vorgegangen, mit dem Unterschied, dass in allen Graphmustern überprüft wird, ob nicht erzeugte und nicht optionale Elemente gelöscht werden können, die Bestandteil der verbotenen Struktur sind oder für weitere Regelanwendungen benötigt werden.

Werden bei der Überprüfung weitere Regelanwendungen ermittelt, bei denen eines dieser infrage kommenden Elemente gelöscht wird, werden aus diesen Regelanwendungen negative Anwendungsbedingungen für das jeweilige Graphmuster generiert, so dass deren Anwendbarkeit ausgeschlossen wird.

Ist zum Beispiel die weitere Anwendung von ip 2 in Abbildung 4.20 in der Lage die geplante Regelsequenz zu vereiteln, so werden alle Elemente des Graphmusters, die zur Anwendungsstelle für die linke Regelseite von ip 2 gehören, in einen neuen Graphen als negative Anwendungsbedingung kopiert.

Kann eine weitere Anwendung von ip 1.1 auf Basis einer weiteren Anwendung von ip 1 zu einer Vereitelung führen, so muss die Anwendbarkeit der Hintereinanderanwendung der beiden Anteile ausgeschlossen werden. Dies geschieht, indem die Vereinigung der beiden Anwendungsstellen in einen neuen Graphen als negative Anwendungsbedingung kopiert wird.

4.4.6 Ergebnis

Die fünf zuvor beschriebenen Schritte werden für jeden Ausführungspfad eines Transformationsdiagramms und jedes verbotene Graphmuster durchlaufen. Dabei wird versucht, Regelsequenzen zu berechnen, die eine problematische Ausführung eines Diagramms zeigen, bei der eine verbotene Struktur entsteht. Es werden zunächst initiale Regelsequenzen berechnet, die dann weiter verfeinert und plausibilisiert werden, wobei unmögliche oder doch unproblematische Regelsequenzen verworfen werden. Bei den Regelsequenzen, die nach Schritt fünf noch existieren, handelt es sich um Gegenbeispiele, die dem Re-Engineer präsentiert werden.

4.5 Verifikation für zu erhaltende Graphmuster

Die Verifikation für zu erhaltende Graphmuster erfolgt in großen Teilen analog zur Verifikation verbotener Graphmuster (Abschnitt 4.4) nur unter umgekehrten Vorzeichen. Gibt es eine Anwendungsstelle für ein zu erhaltendes Graphmuster im Wirtsgraphen bevor ein Transformationsdiagramm ausgeführt wird, so muss es auch danach noch eine Anwendungsstelle geben. Bestimmte Elemente,

die im zu erhaltenden Graphmuster mit dem Stereotyp ≪preserve binding≫ gekennzeichnet sind (siehe Abschnitt 4.1), müssen dabei dieselben bleiben.

Das Ziel der Verifikation für zu erhaltende Graphmuster ist es festzustellen, ob ein Ausführungspfad eines Transformationsdiagramms das Potential hat, bei Anwendung auf einen Wirtsgraphen, in dem eine Anwendungsstelle für ein zu erhaltendes Graphmuster enthalten ist, diese Anwendungsstelle zu zerstören. Dazu wird unterstellt, dass es zu Beginn der Ausführung eine zu erhaltende Struktur gibt. Ausgehend davon wird vorwärts eine zum Ausführungspfad konforme Regelsequenz geplant, bei der mindestens ein Element der zu erhaltenden Struktur gelöscht wird.

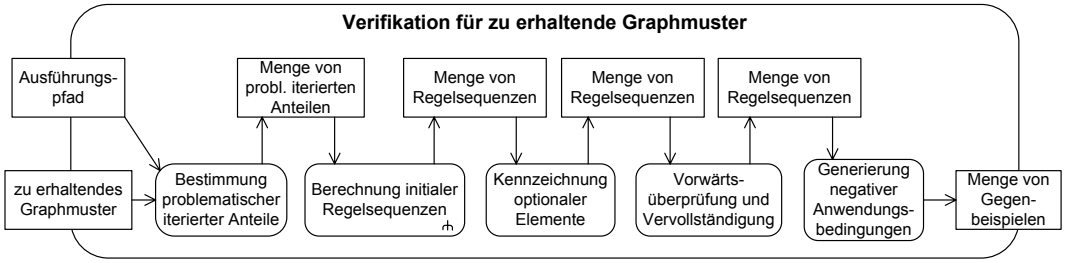

Abbildung 4.21: Verifikation für zu erhaltende Graphmuster

Bei der Konstruktion der Regelsequenzen werden für jeden Ausführungspfad eines Transformationsdiagramms und jedes zu erhaltende Graphmuster die in Abbildung 4.21 gezeigten Schritte durchlaufen. Diese stimmen im Wesentlichen mit den (gleichnamigen) Schritten der Verifikation für verbotene Graphmuster (Abbildung 4.6 auf Seite 114) überein, mit einigen Unterschieden. Diese Unterschiede werden im Folgenden beschrieben. Der Schritt der Vorwärtsüberprüfung und Vervollständigung ist identisch zu dem in Abschnitt 4.4.4 beschriebenen Schritt, so dass hier nicht darauf eingegangen wird.

4.5.1 Bestimmung problematischer iterierter Anteile

Ein Anteil ist in Bezug auf ein zu erhaltendes Graphmuster problematisch, wenn er mindestens ein Element des Musters löschen kann. Dies ist der Fall, wenn er mindestens eine Objekt- oder Linkvariable löscht, die typkonform zu einer Objekt- oder Linkvariable des verbotenen Musters ist.

Anteile sind ebenfalls problematisch, wenn sie eine Ausprägung für einen Pfad zerstören können, indem sie mindestens ein Element löschen, das Bestandteil einer solchen Ausprägung sein könnte (siehe Abschnitt 4.4.1).

4.5.2 Berechnung initialer Regelsequenzen

Regelsequenzen werden hier vorwärts geplant. Als erstes muss der nicht-iterierte Anteil des ersten erfolgreichen Transformation Pattern des Ausführungspfades angewendet werden.

Das erste Graphmuster wird durch Verkleben des zu erhaltenden Musters mit der linken Seite dieses nicht-iterierten Anteils gebildet. Das Verkleben erfolgt analog zum Verkleben in Abschnitt 4.4.2.2. Anschließend wird durch Vorwärtsanwendung des Anteils auf seine Anwendungsstelle im ersten Graphmuster das zweite Graphmuster ermittelt.

Zur Fortsetzung einer Regelsequenz wird bestimmt, welche Anteile als nächstes, nach der aktuell zuletzt angewendeten Regel anwendbar sind.

Handelt es sich bei der zuletzt angewendeten Regel um einen nicht-iterierten Anteil, so muss danach entweder

- der nicht-iterierte Anteil des danach auf dem Ausführungspfad erfolgreichen Transformation Pattern oder
- einer seiner iterierten Anteile auf oberster Hierarchiestufe

angewendet werden. Der linke Teil der Abbildung 4.22 zeigt dies schematisch.

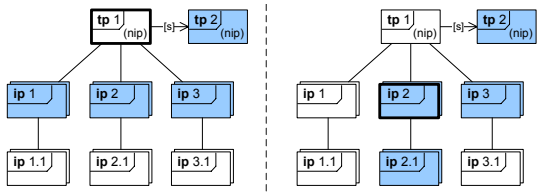

Abbildung 4.22: Direkt nach einem Anteil anwendbare Anteile (schematische Darstellung)

Handelt es sich bei der zuletzt angewendeten Regel um einen iterierten Anteil, so muss danach entweder

- er selbst ein weiteres Mal oder
- einer der in ihm direkt enthaltenen iterierten Anteile oder
- einer der iterierten Anteile seines Vateranteils mit höherem Ausführungsrang oder

- einer der iterierten Anteile auf der obersten Hierarchiestufe mit höherem Ausführungsrang als der iterierte Anteil auf oberster Hierarchiestufe hat, der den zuletzt angewendeten iterierten Anteil transitiv enthält oder

- der nicht-iterierte Anteil des nächsten erfolgreichen Transformation Pattern des Ausführungspfades

angewendet werden. Dies wird im rechten Teil von Abbildung 4.22 dargestellt.

Analog zur Verifikation für verbotene Graphmuster werden zunächst nur aufgrund des Kontrollfluss notwendige Anteile oder problematische iterierte Anteile eingeplant, die einen Teil der zu erhaltenden Struktur löschen können. Allerdings kann es problematische iterierte Anteile auf tieferer Hierarchiestufe geben, etwa ip 1.1 links in der Abbildung, die nicht direkt als nächstes anwendbar sind. So kann ip 1.1 nur direkt nach ip 1 angewendet werden. Daher müssen Regelsequenzen auch mit unproblematischen iterierten Anteilen fortgesetzt werden, wenn diese einen problematischen Anteil transitiv enthalten.

Die Fortsetzung geschieht dann durch Verkleben der linken Seite der nächsten anzuwendenden Regel mit dem aktuell letzten Graphmuster der Sequenz. Beim Verkleben werden gebundene Objektvariablen, das Binden erzeugter Elemente, Pfade der Regeln und die Eindeutigkeit der Anwendungsstelle bei iterierten Anteilen genauso wie in Abschnitt 4.4.2.2 beschrieben berücksichtigt. Auch Pfade der zu erhaltenden Struktur werden analog behandelt, nur dass hier die Zerstörung statt der Erzeugung von Ausprägungen von Interesse ist. Analog wird die wiederholte Zerstörung von Ausprägungen desselben Pfades durch denselben iterierten Anteil berücksichtigt.

Werden beim Verkleben Elemente im letzten Graphmuster hinzugefügt, so werden diese analog in allen früheren Graphmustern der Sequenz hinzugefügt. Werden Pfade der zu erhaltenden Struktur neu abgebildet, so erfolgt dies in den früheren Graphmustern ebenso. Durch Vorwärtsanwendung der Regel auf seine Anwendungsstelle im letzten Graphmuster wird das nächste Graphmuster berechnet und der Sequenz angehängt.

Dies wird für jede Regelsequenz solange wiederholt, bis gemäß Ausführungspfad und zu erhaltender Struktur keine weiteren Anwendungen mehr möglich sind.

4.5.3 Kennzeichnung optionaler Elemente

Die Kennzeichnung optionaler Elemente erfolgt wie in Abschnitt 4.4.3 beschrieben, nur mit einem Unterschied.

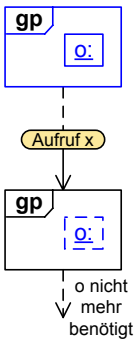

Abbildung 4.23: Status von Objekten in Graphmustern (III)

Elemente der zu erhaltenden Struktur müssen wie in Abbildung 4.23 für Objektvariablen dargestellt von Anfang an existieren. Danach können sie durch eine aufgerufene Transformation gelöscht werden, wenn sie von keiner der danach folgenden Regelanwendungen der Sequenz mehr benötigt werden. Solche Elemente werden als optional gekennzeichnet.

4.5.4 Generierung negativer Anwendungsbedingungen

Die Ergänzung negativer Anwendungsbedingungen für fehlgeschlagene Transformation Pattern eines Ausführungspfades erfolgt genauso wie in Abschnitt 4.4.5.1 beschrieben.

Der Ausschluss einer Korrektur durch weitere Anwendungen iterierter Anteile gestaltet sich bei zu erhaltenden Graphmustern allerdings etwas schwieriger. Bei verbotenen Graphmustern ist eine Korrektur bereits durch das Löschen eines einzigen Elements möglich. Zu erhaltende Graphmuster können dagegen nur korrigiert werden, wenn sie vollständig wiederhergestellt werden.

Dies kann durch die beliebig kombinierte Anwendung weiterer iterierter Anteile geschehen. Es wäre falsch, nur die Anwendbarkeit einzelner Anteile mit Korrekturpotential auszuschließen, die alleine keine vollständige Korrektur bewirken, da dadurch letztendlich Situationen als ungefährlich eingestuft würden, die es nicht unbedingt sind. Stattdessen müssen die Kombinationen der Anwendung iterierter Anteile ausgeschlossen werden, die zu einer vollständigen Korrektur führen würden.

Dies lässt sich mit den negativen Anwendungsbedingungen in den Regelsequenzen nicht ausdrücken, wenn die betreffenden Anwendungen so zu unterschiedlichen Zeitpunkten der Regelsequenz ausgeschlossen werden müssen,

dass negative Anwendungsbedingungen bei verschiedenen Graphmustern hinzugefügt werden müssen. Dadurch würde nur die Anwendbarkeit einzelner iterierter Anteile untersagt und nicht der gesamten Kombination. Sofern nicht alle Anteile der Kombination anwendbar sind, können aber beliebig viele darin enthaltene Anteile anwendbar sein.

In einem solchen Fall werden keine negativen Anwendungsbedingungen generiert. Dadurch ist eine Regelsequenz weniger präzise beziehungsweise wird die Beschaffenheit des Wirtsgraphen weniger präzise charakterisiert.

4.5.5 Ergebnis

Analog zur Verifikation für verbotene Graphmuster werden die zuvor beschriebenen Schritte für jeden Ausführungspfad eines Transformationsdiagramms und jedes zu erhaltende Graphmuster durchlaufen. Dabei wird versucht, Regelsequenzen zu berechnen, die eine problematische Ausführung eines Diagramms zeigen, bei der eine zu erhaltende Struktur zerstört wird. Auch hier werden zunächst initiale Regelsequenzen berechnet, die dann weiter verfeinert und plausibilisiert werden, wobei unmögliche Regelsequenzen verworfen werden. Bei den Regelsequenzen, die nach Schritt fünf noch existieren, handelt es sich um Gegenbeispiele, die dem Re-Engineer präsentiert werden.

Abbildung 4.25 zeigt ein solches Gegenbeispiel für das zu erhaltende Graphmuster aus Abbildung 4.24 und das *SimpleExtract*-Transformationsdiagramm aus Abbildung 3.1.

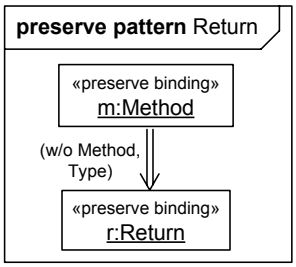

Abbildung 4.24: Zu erhaltendes Graphmuster *Return*

Das Graphmuster beschreibt, dass wenn eine Methode vor einer Transformation eine Return-Anweisung enthält, dies auch nach einer Transformation noch der Fall sein muss. Falls dem nicht so ist, kann sich das Verhalten der Methode verändert haben.

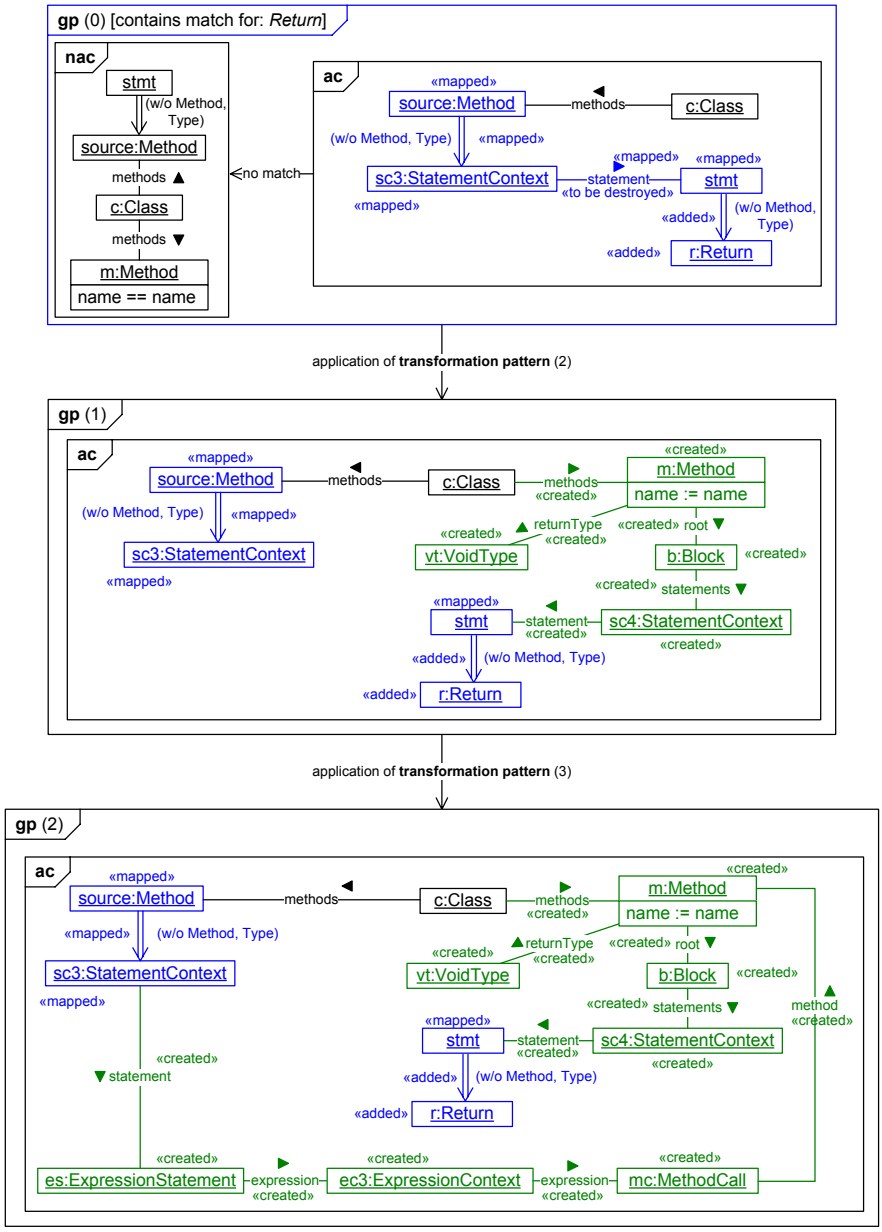

Abbildung 4.25: Ein Gegenbeispiel für ein zu erhaltendes Graphmuster

Das erste Graphmuster des Gegenbeispiels enthält eine Instanz des zu erhaltenden Musters. Dies ist der Fall, da sich unterhalb des zu extrahierenden Statements stmt eine Return-Anweisung befindet, die damit von der Methode source:Method über einen Pfad erreichbar ist. Da die zu erhaltende Struktur vollständig ist, wird das Graphmuster blau umrandet. Die Elemente der zu erhaltenden Struktur sind blau gefärbt.

Die Anwendung von Transformation Pattern (2) des Diagramms führt zum zweiten Graphmuster, in dem die Instanz zerstört ist, da die Anweisung mit der Return-Anweisung in die neu erzeugte Methode verschoben wurde. Da die Instanz nicht mehr vollständig ist, ist das Graphmuster nicht mehr blau, sondern schwarz umrandet.

Die Anwendung von Transformation Pattern (3) fügt wie im dritten Graphmuster gezeigt den Aufruf der neuen Methode ein und stellt damit auch einen Pfad von source:Method zur Return-Anweisung wieder her. Allerdings liegt auf diesem Pfad mit m:Method ein Methodenobjekt, dessen Traversierung durch den Pfad nicht erlaubt ist.

4.6 Aussage des Verfahrens

Die durch das Verfahren berechneten Gegenbeispiele beschreiben nicht zwingend vollständige Ausführungen von Transformationsdiagrammen, sondern Ausschnitte von Ausführungen, die zur Verletzung von Kriterien führen können. Das Verifikationsverfahren benutzt die Verifikationskriterien, um alle repräsentativen Kombinationen von Regelanwendungen eines Transformationsdiagramms zu berechnen, die zu einer Kriterienverletzung führen können.

Regelsequenzen können unterschiedlich viele Elemente einer verbotenen Struktur erzeugen oder einer zu erhaltenden Struktur löschen. In Bezug darauf werden alle möglichen Sequenzen durch das Verfahren gebildet. Die Anwendung von in Bezug auf ein Kriterium problematischen Regeln sowie von Regeln, die aufgrund des Kontrollfluss angewendet werden müssen, wird auf alle möglichen Weisen wohlwollend unterstellt. Daraus wird abgeleitet, wie der Wirtsgraph mindestens beschaffen sein muss, damit die jeweilige Regelsequenz möglich ist.

Sind aufgrund des abgeleiteten Wirtsgraphen weitere Regeln zwingend anwendbar, so werden sie in die Regelsequenzen aufgenommen. Werden dadurch später geplante Anwendungen oder die geplante Kriterienverletzung vereitelt, werden die betreffenden Regelsequenzen verworfen. Sofern weitere, nicht in einer Regelsequenz eingeplante Anwendungen einen Einfluss auf die geplante

Regelsequenz haben können, werden die möglichen Auswirkungen berücksichtigt und zum Teil pessimistisch abgeschätzt. Dabei wird im Zweifel immer so entschieden, dass eine Kriterienverletzung eintreten kann.

Wie in Abschnitt 4.4.3.1 beschrieben, können aufgerufene Transformationsdiagramme Auswirkungen auf die Existenz von Elementen im Wirtsgraphen haben. Die möglichen Auswirkungen werden pessimistisch abgeschätzt und finden Eingang in ein Gegenbeispiel, indem Elemente von Graphmustern als optional oder optional erzeugt gekennzeichnet werden.

Aufgerufene Transformationsdiagramme können darüber hinaus Elemente löschen oder weitere Elemente erzeugen, so dass es zu einer Vereitelung der Kriterienverletzung kommen würde. Solche möglichen Auswirkungen werden nicht betrachtet – Auswirkungen, die eine Kriterienverletzung fördern, werden dagegen immer unterstellt.

Über aufgerufene Transformationsdiagramme hinaus können auch weitere, nicht eingeplante Anwendungen iterierter Anteile Auswirkungen auf die Existenz von Elementen im Wirtsgraphen haben. Solche Anteile *können* zusätzlich anwendbar sein, wenn der Wirtsgraph weitere Elemente enthält, als durch die Graphmuster des Gegenbeispiels gefordert werden. Dabei können sie unter anderem Elemente löschen, die dann für weitere Anwendungen nicht mehr zur Verfügung stehen müssen.

Diese Auswirkungen werden ebenfalls pessimistisch abgeschätzt: wenn die Möglichkeit besteht, dass der Wirtsgraph so beschaffen ist, dass eine zusätzliche Regelanwendung ein Element löscht, das für keine nachfolgende geplante Anwendung benötigt wird, wird dieses als optional gekennzeichnet, ohne dass die betreffende Anwendung in der Regelsequenz eingeplant wird.

Dadurch wird einerseits die Anzahl der berechneten Gegenbeispiele reduziert. Alle Elemente die gelöscht worden sein können, werden als optional gekennzeichnet, wodurch mehrere verschiedene Varianten der Ausführung durch ein Gegenbeispiel repräsentiert werden. Andererseits ergeben sich Ungenauigkeiten, da durch unterstellte zusätzliche Anwendungen weitere Elemente im Wirtsgraphen vorhanden wären, die zu einer Vereitelung der Kriterienverletzung führen könnten, die aber unberücksichtigt bleibt.

Insgesamt können auf diese Weise Gegenbeispiele berechnet werden, die tatsächlich so nicht auftreten können (false-positives). Dabei ist für ein Gegenbeispiel bekannt, an welchen Stellen pessimistische Abschätzungen vorgenommen wurden und ob es sich daher um ein false-positive handeln kann. Hier ist aktuell der Re-Engineer gefragt, ein solches Gegenbeispiel durch weitere, manuelle Analysen zu plausibilisieren.

Gleichzeitig schließt das Verfahren auf diese Weise aus, dass Kriterienver-

letzungen unerkannt bleiben. Das Verfahren wurde gezielt so entworfen, dass wenn eine Kriterienverletzung irgendwie möglich ist beziehungsweise sein könnte, auch ein entsprechendes Gegenbeispiel berechnet wird, was allerdings im Rahmen dieser Arbeit nicht formal bewiesen wurde.

Kapitel 5

Werkzeugunterstützung

Dieses Kapitel beschreibt die prototypische Umsetzung der in den vorangehenden Kapiteln vorgestellten Konzepte in dem Werkzeug PaReclipse (Pattern-based Re-Engineering for Eclipse).

Die Umsetzung basiert zum einen auf der Entwicklungsumgebung Fujaba [Fuj], die seit 1998 im Fachgebiet Softwaretechnik der Universität Paderborn entwickelt wird. Fujaba ist eine modellbasierte Entwicklungsumgebung auf Basis von UML und Story Driven Modelling (SDM, siehe Abschnitt 2.4), die durch Plug-Ins erweitert werden kann [BGN+04]. Zu den Grundfunktionen gehört eine automatische Generierung von ausführbarem Java-Code aus den mit Fujaba erstellten Spezifikationen.

Zum anderen basiert die Umsetzung auf der mittlerweile weit verbreiteten Entwicklungsumgebung Eclipse [Ecla]. Eclipse ist ein Framework, das durch Plug-Ins erweitert und damit als Basis für die Entwicklung beliebiger Anwendungen genutzt werden kann. Mit dem *Java Develoment Tooling* (JDT) steht eine Sammlung von Plug-Ins zur Verfügung, die Eclipse zu einer Java-Entwicklungsumgebung macht.

Fujaba wurde ebenfalls als Plug-In in Eclipse und mit dem JDT integriert. Auf diese Weise ist die Kombination einer modellbasierten Entwicklungsumgebung mit einer Java-Entwicklungsumgebung entstanden, die Fujaba4Eclipse genannt wird.

In der vorliegenden Arbeit wird Fujaba4Eclipse durch die strukturbasierte Mustererkennung, die Spezifikation und Ausführung von Programmtransformationen sowie deren Verifikation zu dem Re-Engineering-Werkzeug PaReclipse erweitert. Dabei wurde die bereits für Fujaba implementierte strukturbasierte Mustererkennung auf Fujaba4Eclipse portiert und wie in Abschnitt 2.3.5 beschrieben erweitert. Die Transformationsspezifikation und -ausführung sowie das Verifikationsverfahren wurde durch weitere Eclipse-

Plug-Ins auf Basis von Fujaba4Eclipse realisiert.

Im Folgenden wird die Benutzungsschnittstelle des Werkzeugs beschrieben. Im Anschluss wird ein Überblick über seine Architektur gegeben.

5.1 Benutzungsschnittstelle

Die Benutzungsschnittstelle von PaReclipse integriert sich in die Benutzeroberfläche von Eclipse und Fujaba4Eclipse. Die Benutzeroberfläche von Eclipse organisiert verschiedene *Sichten* und *Editoren* zur Visualisierung und Modifikation von Informationen in *Perspektiven*.

Eine Perspektive ist eine Konfiguration ausgewählter Sichten und Editoren, die typischerweise an einer bestimmten Aufgabe orientiert zusammengestellt werden. Fujaba4Eclipse definiert eine eigene Perspektive, die verschiedene Sichten und Editoren zur Spezifikation von Modellen anbietet. Die Perspektive ist in Abbildung 5.1 geöffnet.

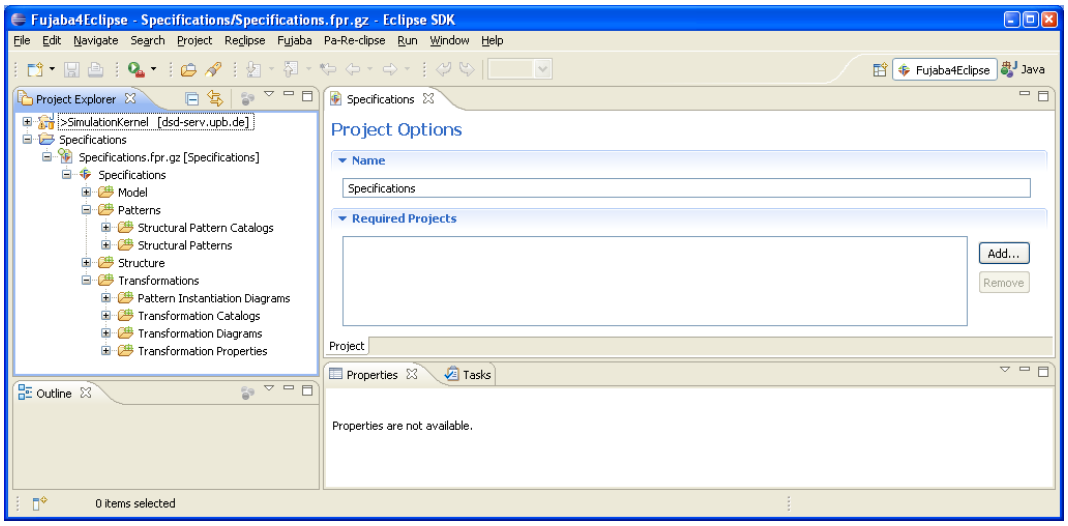

Abbildung 5.1: Benutzeroberfläche von PaReclipse

Fujaba4Eclipse speichert Spezifikationen beziehungsweise Modelle in speziellen Modelldateien im Dateisystem ab. Der *Project Explorer* (links oben in der Abbildung) bietet eine Sicht auf solche Modelldateien und auf ihren Inhalt. In der Abbildung ist eine Modelldatei geöffnet, die eine Reihe von Strukturmuster- und Transformationsspezifikationen beinhaltet, die innerhalb des Modells unter verschiedenen Kategorien abgelegt sind.

Der rechte, obere Teil der Benutzeroberfläche ist für Editoren reserviert. In der Abbildung ist der Editor einer Modelldatei geöffnet. Im Bild ist die Seite mit allgemeinen Informationen zum Modell zu sehen. Werden über den Project Explorer im Modell enthaltene Spezifikationen geöffnet, werden weitere Seiten mit zum Beispiel grafischen Editoren hinzugefügt.

Unterhalb des Editor-Bereichs ist der *Properties*-Editor angeordnet, der Eigenschaften des aktuell bearbeiteten Elements anzeigt und ihre Modifikation ermöglicht. Links unten befindet sich eine *Outline*-Sicht, die einen Überblick über das aktuell bearbeitete Element bietet.

5.1.1 Spezifikation von Strukturmustern und Transformationsdiagrammen

Zu Beginn des Re-Engineering-Prozesses werden Strukturmuster zur Analyse einer Software spezifiziert, wobei typischerweise auf bereits bestehende Spezifikationen aufgebaut wird. Aufbauend auf die Strukturmuster werden Transformationsdiagramme zur Restrukturierung spezifiziert.

Die Strukturmuster und Transformationsdiagramme werden in demselben Modell definiert.

5.1.1.1 Strukturmuster

In Abbildung 5.2 wurde im Project Explorer die Structural Patterns-Kategorie aufgeklappt, so dass die im Modell spezifizierten Strukturmuster aufgelistet werden. Das Strukturmuster *ElseIfDispatcher* aus Abschnitt 2.3.2 ist zur Bearbeitung rechts oben in einer Editor-Seite geöffnet.

Auf der linken Seite der Editor-Seite befindet sich eine *Palette*, die Werkzeuge anbietet, um Objekt-, Annotations- oder Linkvariablen im Strukturmuster zu erzeugen. Die Eigenschaften von Elementen des Strukturmusters können über den Properties-Editor geändert werden. Die Outline-Sicht unten links zeigt eine verkleinerte Ansicht des Strukturmusters.

Strukturmuster werden zu Katalogen zusammengestellt. Damit ein Katalog zur Analyse einer Software eingesetzt werden kann, muss er zunächst exportiert werden. Dabei werden aus den Strukturmustern unter Verwendung der Java-Codegenerierung so genannte *Erkennungsmaschinen* generiert, übersetzt und in ein Jar-Archiv verpackt, das bei Ausführung der strukturbasierten Mustererkennung geladen werden kann.

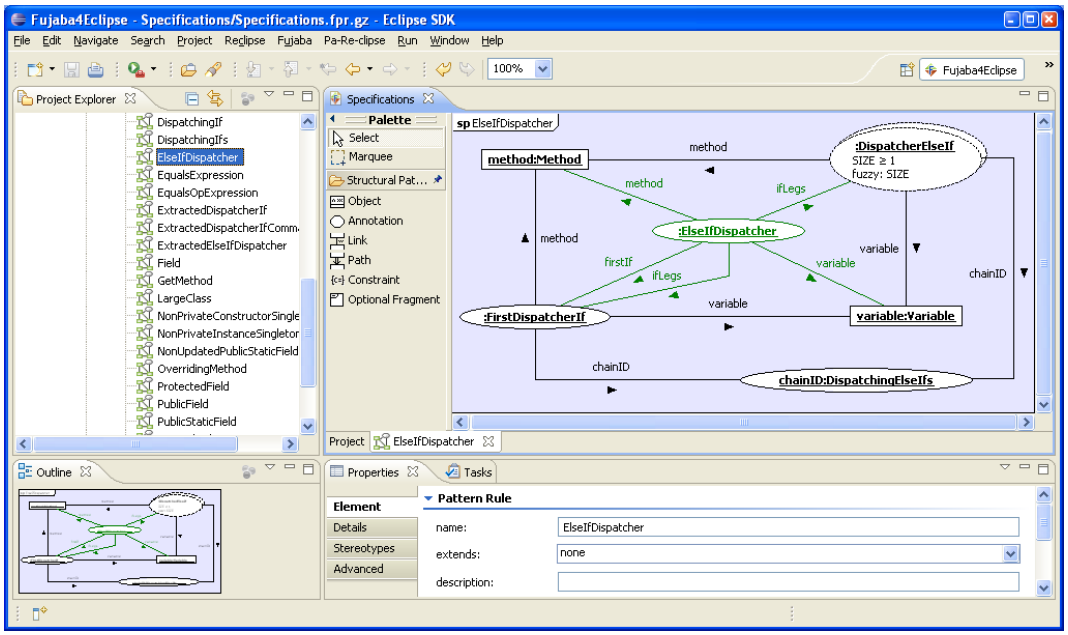

Abbildung 5.2: Spezifikation des *ElseIfDispatcher*-Strukturmusters

5.1.1.2 Transformationsdiagramme

Transformationsdiagramme werden in demselben Modell spezifiziert, in dem sich auch die Strukturmuster befinden, auf die sie sich beziehen.

Abbildung 5.3 zeigt die Spezifikation des *EncapsulateReadAccess*-Transformationsdiagramms aus Abbildung 3.7 in einer Editor-Seite.

Über die Palette auf der linken Seite des Editors können dem Transformationsdiagramm neue Spezifikationselemente hinzugefügt werden. Die Eigenschaften der Elemente werden über den Properties-Editor unterhalb der grafischen Ansicht geändert. Die Outline-Sicht unten links zeigt eine verkleinerte Ansicht des Transformationsdiagramms.

Das Transformation Pattern (1) des Diagramms erzeugt eine *GetMethod*-Musterinstanz. Damit dies möglich ist, müssen für das GetMethod-Muster in demselben Modell sowohl ein Strukturmuster als auch eine Mustererzeugungstransformation spezifiziert sein.

Zur Spezifikation von Transformation Pattern anhand von Quelltextbeispielen müssen die Beispiele in JAVA-Projekten abgelegt sein. Über einen *Wizard* wird zunächst das Vorherbeispiel in ein Transformation Pattern überführt und danach das Nachherbeispiel. In dem Wizard wird sowohl das JAVA-Projekt

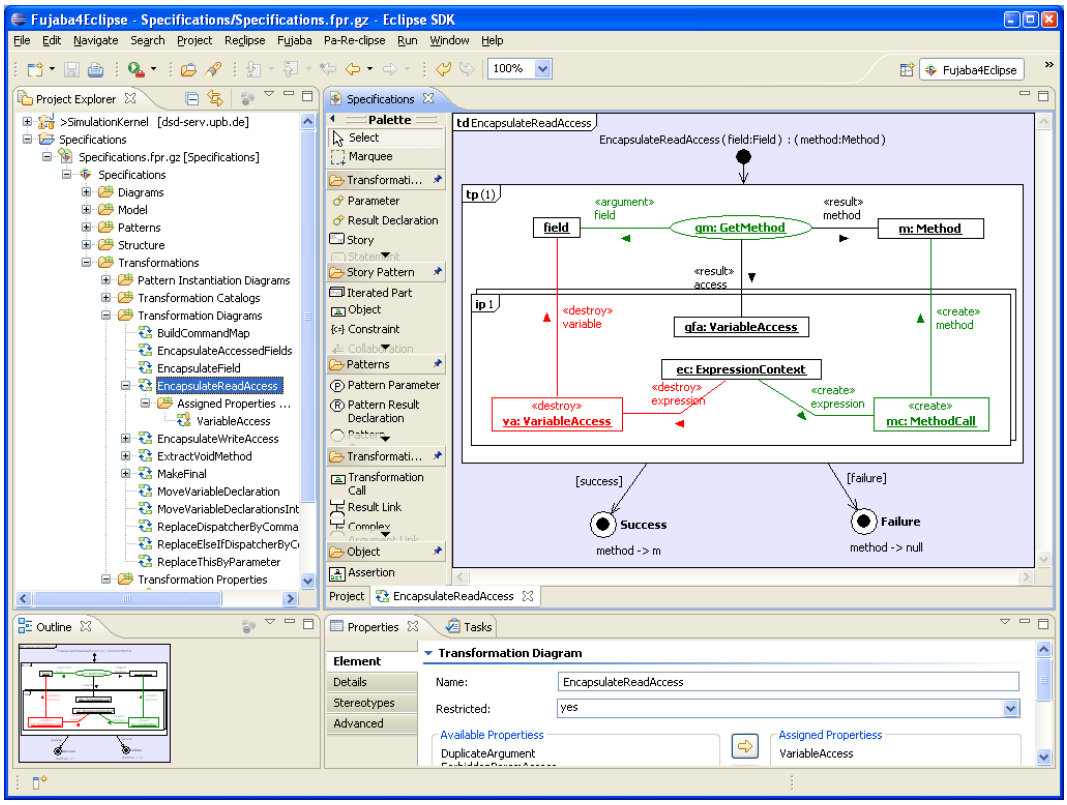

Abbildung 5.3: Spezifikation des *EncapsulateReadAccess*-Transformationsdiagramms

ausgewählt, in dem sich das Beispiel befindet, als auch das Modell, in dem die Transformation Pattern abgelegt werden.

Über einen zweiten Wizard wird die Regelsynthese angestoßen. Dabei werden die beiden Transformation Pattern für das Vorher- und Nachherbeispiel sowie das Transformationsdiagramm ausgewählt, in welches das synthetisierte Transformation Pattern eingefügt werden soll. Zusätzlich muss eine XML-Datei angegeben werden, in der definiert wird, welche Objektvariablen in den beiden Ausgangspattern einander entsprechen. Diese Datei muss manuell erstellt werden, da in der aktuellen Version des Prototyps kein automatisches Vergleichsverfahren implementiert wurde.

Analog zu Strukturmustern werden Transformationsdiagramme in Katalogen zusammengestellt, die exportiert werden müssen, damit sie ausgeführt werden können. Dabei werden sie zum Teil auf Story Diagramme abgebildet,

aus denen mit Hilfe eines an die in Kapitel 3 beschriebene Semantik angepassten Verfahrens JAVA-Code generiert und übersetzt wird. Dabei entstehen (Transformations-)Klassen, die in ein JAR-Archiv verpackt werden.

5.1.2 Verifikation von Transformationsdiagrammen

Die Verifikation sollte durchgeführt werden, bevor Transformationsdiagramme zur Restrukturierung einer Software ausgeführt werden.

Der Re-Engineer spezifiziert dafür zunächst die Kriterien, auf deren Einhaltung die Transformationsdiagramme verifiziert werden sollen und weist sie den entsprechenden Transformationsdiagrammen zu.

Danach führt er die Verifikation durch und inspiziert die dabei berechneten Gegenbeispiele.

5.1.2.1 Verifikationskriterien

Verifikationskriterien werden in demselben Modell spezifiziert, in dem sich die zu verifizierenden Transformationsdiagramme befinden.

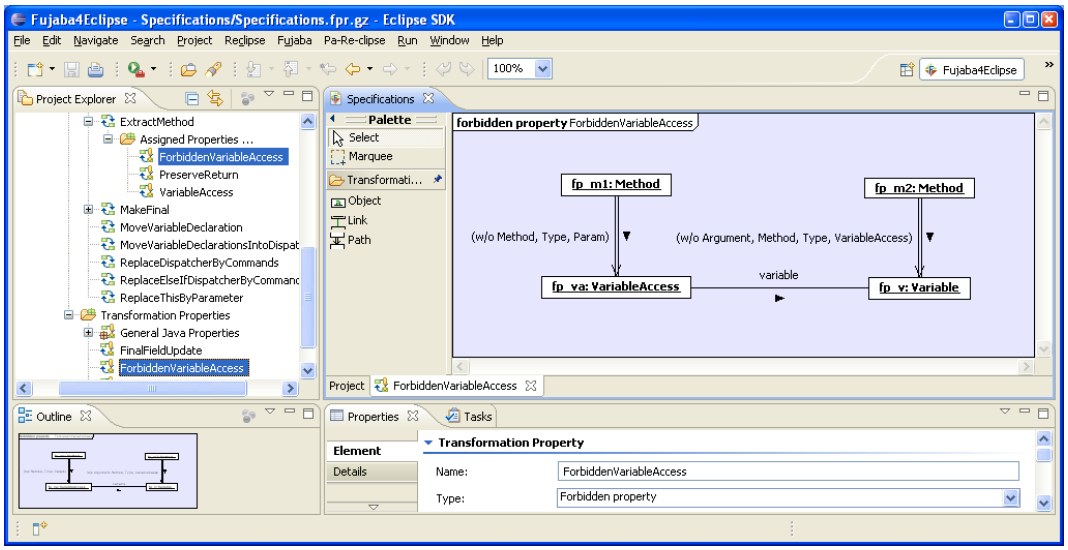

Abbildung 5.4: Spezifikation des *ForbiddenVariableAccess*-Kriteriums

Abbildung 5.4 zeigt die im Editor geöffnete Spezifikation des *ForbiddenVariableAccess*-Kriteriums, das in der Outline-Sicht verkleinert angezeigt wird.

Auch hier werden Elemente über die Palette hinzugefügt und über den Properties-Editor verändert.

Die Kriterien werden den Transformationsdiagrammen zugewiesen, für die sie gelten müssen. Dies geschieht über den Properties-Editor beim jeweiligen Transformationsdiagramm, wie am unteren Rand der Abbildung 5.3 zu sehen ist. Einem Transformationsdiagramm zugewiesene Kriterien werden zusätzlich im Project Explorer unterhalb des Diagramms angezeigt.

Kriterien können darüber hinaus zu Mengen von Kriterien zusammengefasst und ebenfalls Transformationsdiagrammen zugewiesen werden.

5.1.2.2 Verifikationsergebnisse

Die Verifikation wird über das Kontextmenü eines Transformationsdiagramms gestartet. Das Diagramm wird dann auf Einhaltung der ihm zugewiesenen Kriterien überprüft.

Werden dabei Gegenbeispiele gefunden, werden diese im Modell in der Kategorie Transformation Verification Results aufgeführt. Dies ist in Abbildung 5.5 im Project Explorer zu sehen.

Im Beispiel wurde das Transformation Pattern (4) des *SimpleExtract*-Transformationsdiagramms ohne geschachtelten iterierten Anteil aus Abbildung 3.3 einzeln auf Einhaltung des *DuplicateArgument*-Kriteriums verifiziert. Dabei wurden vier Gegenbeispiele berechnet.

Ein Gegenbeispiel wird im Project Explorer als Knoten dargestellt, der mit dem Namen des verifizierten Transformationsdiagramms beschriftet ist. Unterhalb des Knotens werden über zwei weitere Knoten das verifizierte Diagramm und das Kriterium, für welches das Gegenbeispiel berechnet wurde, referenziert. Der dritte Knoten repräsentiert die Regel- beziehungsweise Graphmustersequenz des Gegenbeispiels.

In Abbildung 5.5 ist eine der Regelsequenzen auf einer Editor-Seite geöffnet.

5.1.3 Strukturbasierte Mustererkennung

Die strukturbasierte Mustererkennung kann über das PaReclipse-Menü direkt auf einem Java-Projekt ausgeführt werden. Dabei wird ein zuvor exportierter Katalog mit Erkennungsmaschinen ausgewählt und geladen. Die Erkennungsmaschinen traversieren bei der Analyse ein Modell, das den abstrakten Syntaxgraphen adaptiert, den das JDT für das analysierte Projekt erstellt. Das Adaptermodell wird dabei *on demand* aufgebaut.

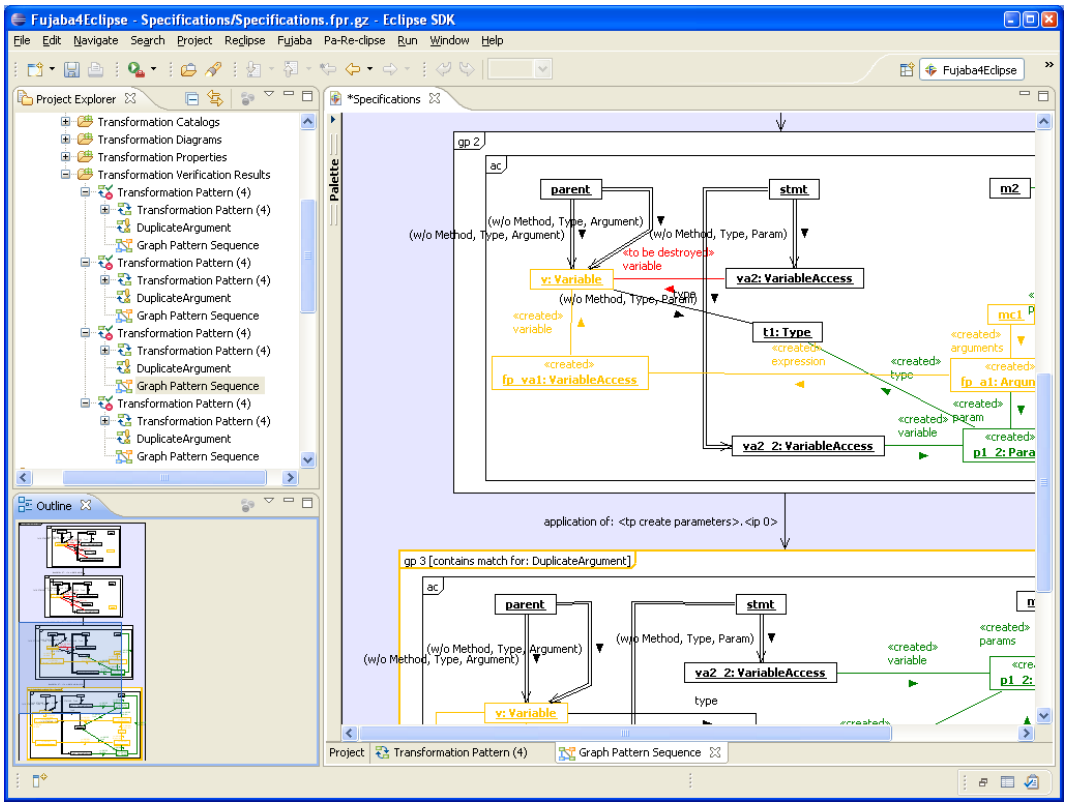

Abbildung 5.5: Darstellung eines Gegenbeispiels

Die Ergebnisse der Analyse werden wie in Abbildung 5.6 gezeigt in der *Annotations*-Sicht im rechten, unteren Bereich der Bildschirmoberfläche angezeigt. Die Annotationen werden nach ihrem Typ kategorisiert und gemäß ihrer Bewertung sortiert. Annotationen können in der Ansicht ausgeklappt werden, so dass die von ihnen annotierten Elemente angezeigt werden.

Durch einen Doppelklick auf eines der Elemente wird die zugehörige Java-Klasse im Quelltexteditor des JDT geöffnet. Der zum Element gehörende Quelltext wird dabei selektiert und der Editor scrollt den selektierten Text in den sichtbaren Bereich.

In der Abbildung ist eine *ConditionalDispatching*-Annotation ausgeklappt, die mehrere *ConditionalDispatcher-Instanzen* über dieselbe Variable innerhalb derselben Methode annotiert. Die annotierte Variable `cur` ist in der Ansicht ausgewählt. Über einen Doppelklick auf die zugehörige Tabellenzeile wurde die Java-Klasse im Quelltexteditor des JDT geöffnet. Dabei wurde automatisch

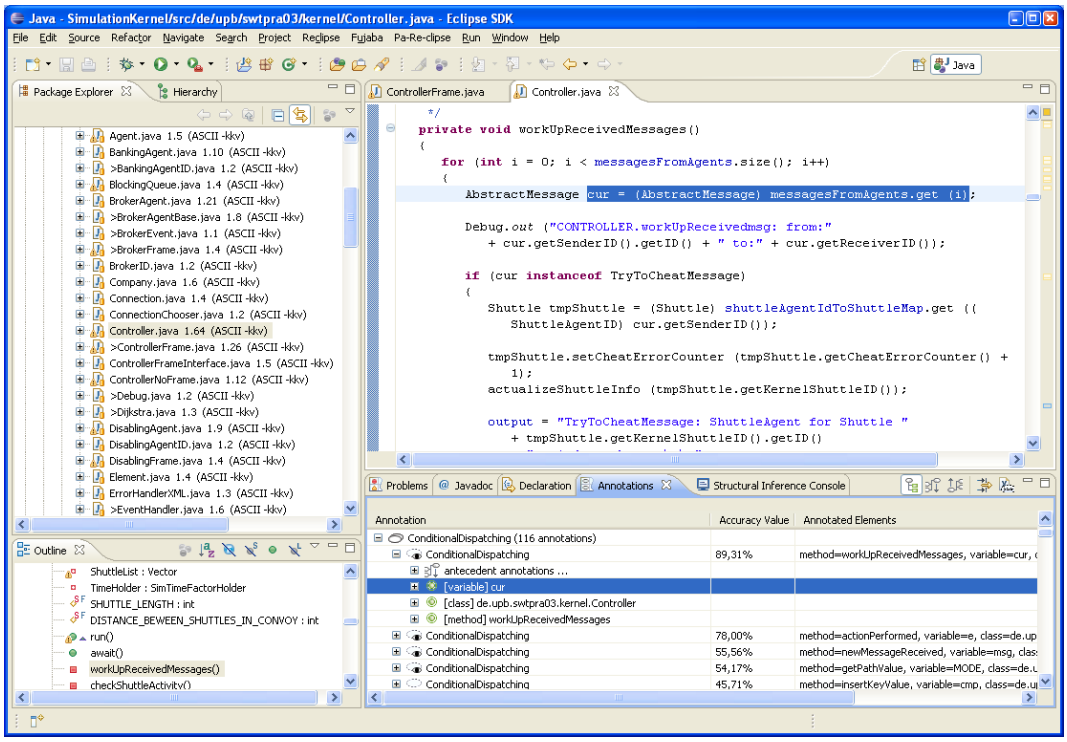

Abbildung 5.6: Ergebnisse der strukturbasierten Mustererkennung

ihre Deklaration selektiert und sichtbar gemacht.

5.1.4 Ausführung von Transformationsdiagrammen

Die Ausführung von Transformationsdiagrammen wird ebenfalls über das Pa-Reclipse-Menü angestoßen.

Dabei wird ein *Wizard* gestartet, in dem zunächst ein zuvor exportierter Transformationskatalog geladen wird.

Im zweiten Schritt werden die im Katalog enthaltenen Transformationsdiagramme aufgelistet und der Re-Engineer wählt die anzuwendende Transformation aus.

Im dritten Schritt werden die Argumente für die Parameter des Transformationsdiagramms bestimmt, unter anderem die Schwachstelle, auf die eine Transformation angewendet werden soll. Die Parameter werden dazu in einer Tabelle aufgelistet. Der Re-Engineer wählt einen Parameter in der Tabelle aus. Danach selektiert er das Element, das dem Parameter als Argument zugewie-

sen werden soll, in der Entwicklungsumgebung. Dies kann zum Beispiel eine Annotation in der Annotations-Sicht sein. Genauso kann er zum Beispiel eine Attribut-Deklaration im geöffneten Quelltexteditor selektieren. Der Wizarddialog ist nicht-modal implementiert und enthält eine Schaltfläche, bei deren Betätigung die aktuelle Selektion in der Entwicklungsumgebung ausgewertet und wenn möglich dem Parameter zugewiesen wird.

Sind alle Parameter belegt, kann das Transformationsdiagramm ausgeführt werden und der Wizard wird beendet. Die Transformationsdiagramme modifizieren das Adaptermodell für den abstrakten Syntaxgraphen, den das JDT für das Java-Projekt erstellt hat. Das Adaptermodell gibt die Veränderungen an den JDT-AST weiter, woraufhin auch der Quelltext aktualisiert wird.

5.2 Architektur

Das Werkzeug besteht aus mehreren miteinander interagierenden Komponenten, die als Eclipse-Plug-Ins auf Basis von Fujaba4Eclipse realisiert sind. Abbildung 5.7 zeigt die wesentlichen Komponenten und ihre Abhängigkeiten. Fast alle der abgebildeten Komponenten verfügen über eine Benutzungsschnittstelle, die jeweils in einem weiteren Plug-In implementiert ist.

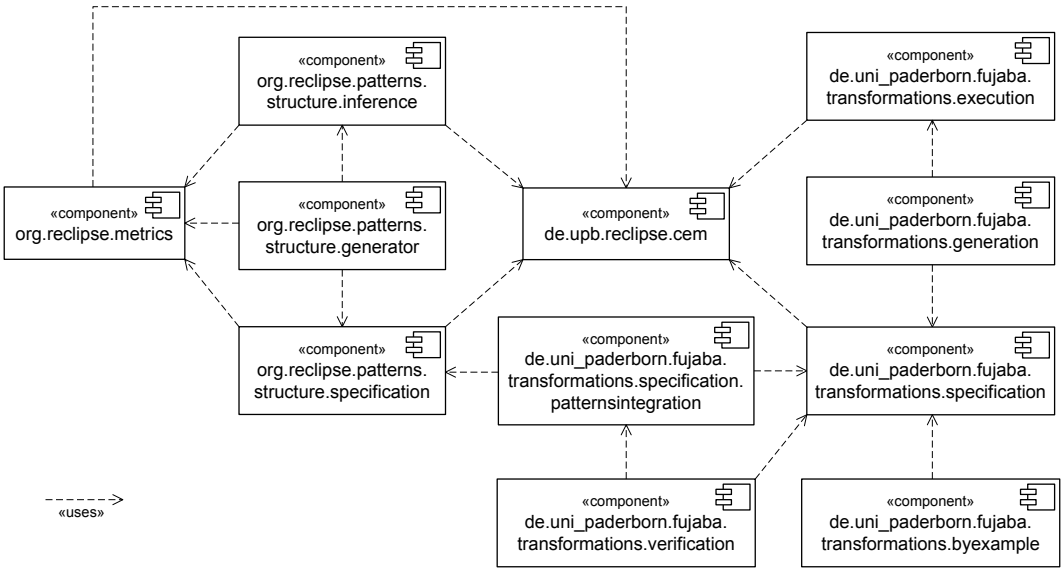

Abbildung 5.7: Komponenten von PaReclipse

Die Komponente de.upb.reclispe.cem (8.663 LOC[1]) implementiert das Strukturmodell des abstrakten Syntaxgraphen und wird daher von den meisten anderen Komponenten verwendet. Das Strukturmodell ist als ein Adaptermodell für den abstrakten Syntaxgraphen des JDT implementiert, so dass dieser mit Hilfe von Story Diagrammen und Transformationsdiagrammen analysiert und modifiziert werden kann.

Die erweiterte strukturbasierte Mustererkennung wird durch die org.reclipse.*-Komponenten realisiert:

org.reclipse.metrics implementiert die Berechnung von Softwareproduktmetriken (10.870 LOC),

org.reclipse.patterns.structure.specification stellt das Metamodell und die Logik für die Spezifikation von Strukturmustern zur Verfügung,

org.reclipse.patterns.structure.generator übersetzt die Strukturmuster in Erkennungsmaschinen, die von dem Erkennungsprozess zur Analyse eines Systems ausgeführt werden,

org.reclipse.patterns.structure.inference implementiert den in Abschnitt 2.3.3 beschriebenen Erkennungsprozess sowie Schnittstellen für Erkennungsmaschinen und Annotationen.

Die de.uni_paderborn.fujaba.transformations.*-Komponenten setzen die Transformationen um:

de.uni_paderborn.fujaba.transformations.specification stellt das Metamodell und die Logik für die Spezifikation von Transformationsdiagrammen bereit (7.549 LOC),

de.uni_paderborn.fujaba.transformations.specification.patternsintegration integriert die Transformationsspezifikation mit der Spezifikation der Strukturmuster, so dass Transformationsdiagramme Strukturmusterannotationen binden und Musterinstanzen erzeugen können (3.774 LOC),

de.uni_paderborn.fujaba.transformations.byexample setzt die Spezifikation von Transformation Pattern anhand konkreter Quelltextbeispiele um (4.085 LOC),

[1] Anzahl Quelltextzeilen ohne Leerzeilen, Kommentarzeilen und Nur-{-Zeilen

de.uni_paderborn.fujaba.transformations.generation übersetzt Transformationsdiagramme in ausführbare JAVA-Klassen (3.525 LOC),

de.uni_paderborn.fujaba.transformations.execution implementiert die Ausführung von Transformationen; dabei werden die generierten Transformationsklassen geladen und ausgeführt (2.081 LOC).

Die Komponente de.uni_paderborn.fujaba.transformations. verification (17.005 LOC) implementiert das Verifikationsverfahren. Bereits existierende Werkzeuge zur Verifikation von Graphtransformationen wie zum Beispiel GROOVE [Ren08] können dafür nicht eingesetzt werden, da sie lediglich eine Erreichbarkeitsanalyse bei bekanntem Startgraphen erlauben (vgl. Abschnitt 7.3.2). Das in der vorliegenden Arbeit entwickelte Verfahren ermittelt potentiell problematische Sequenzen von Regelanwendungen ohne einen konkreten Startgraphen zu kennen. Vielmehr wird während der Verifikation ermittelt beziehungsweise charakterisiert, wie der Startgraph beschaffen sein muss, damit die problematische Sequenz stattfinden kann.

Insgesamt wurden im Rahmen dieser Arbeit neue Softwarekomponenten im Umfang von 57.552 LOC implementiert. In welchem Umfang bestehende Komponenten erweitert wurden, wurde nicht ermittelt.

Kapitel 6

Evaluierung

Zur Erprobung des in dieser Arbeit entwickelten Ansatzes wurde ein Katalog von Strukturmustern zur Erkennung von Schwachstellen angelegt, der im Folgenden vorgestellt wird.

Mit Hilfe dieses Katalogs wurden fünf Softwaresysteme analysiert: ein kommerzielles Webinformationssystem zur Verwaltung von Lehrveranstaltungen (WebLV), eine im Fachgebiet Softwaretechnik im Rahmen von Lehrveranstaltungen von zahlreichen Studierenden erstellte Software zur Simulation von autonomen Shuttle-Systemen (ShuttleSim), JHotDraw [JHD] in der Version 6.0 beta 1 (JHD), das Standard Widget Toolkit (SWT) [Eclb] in der Version 3.4 und die Hauptanwendung von ArgoUML [Arg] in der Version 0.28. Tabelle 6.1 zeigt einige Eckdaten der Systeme. Aufgrund der großen Menge der dabei erzielten Ergebnisse werden im zweiten Abschnitt nur die Ergebnisse der Schwachstellenanalyse von SWT vorgestellt.

Anschließend wurden Transformationsdiagramme zur Verbesserung von Conditional-Dispatcher-Instanzen, genauer von Elseif-Dispatcher-Instanzen,

System	LOC[a]	Klassen[b]	Pakete	max. DIT[c]
ArgoUML	106.417	1.742	79	8
JHD	27.567	544	30	5
ShuttleSim	52.426	659	41	6
SWT	117.360	702	22	7
WebLV	83.182	394	73	4

[a] Anzahl Quelltextzeilen ohne Leerzeilen, Kommentarzeilen und Nur-{-Zeilen
[b] ohne anonyme innere Klassen
[c] Vererbungstiefe (Depth of Inheritance Tree)

Tabelle 6.1: Eckdaten der analysierten Systeme

definiert und auf ausgewählte Fundstellen angewendet (Abschnitt 6.3).

Die Transformationsdiagramme wurden schließlich auf Einhaltung einiger Kriterien verifiziert (Abschnitt 6.4).

6.1 Strukturmuster zur Erkennung von Schwachstellen

Die wie in Kapitel 2.3.5 beschrieben erweiterte strukturbasierte Mustererkennung wird zur Erkennung von Schwachstellen eingesetzt. Dazu wurden 21 Schwachstellen mit Hilfe von insgesamt 84 zum Teil aufeinander aufbauenden Strukturmustern spezifiziert.

Die Strukturmuster verwenden Metriken, um Schwachstellen zu bewerten. Diese Metriken werden im Folgenden kurz beschrieben. Danach werden die spezifizierten Schwachstellen, eingeteilt in verschiedene Kategorien, kurz beschrieben. Dabei wird angegeben, welche Schwachstellen auf Basis welcher Metriken mit Fuzzy-Metrikbedingungen (siehe Abschnitt 2.3.5) bewertet werden. Die dabei verwendeten Bewertungsfunktionen werden auf Basis der analysierten Systeme bestimmt und in Abschnitt 6.1.7 beschrieben.

6.1.1 Eingesetzte Metriken

Die folgenden Metriken werden in den Strukturmustern verwendet, um Schwachstellen zu bewerten.

Cyclomatic Complexity (CC) Diese ursprünglich von McCabe [McC76] definierte Metrik misst die Komplexität einer Methodenimplementierung auf Basis ihres Kontrollflusses. Die Berechnung erfolgt im Prinzip mit Hilfe eines Kontrollflussgraphen, dessen Knoten den Anweisungen der Methode entsprechen. Können Anweisungen unter Berücksichtigung von Schleifen oder bedingten Anweisungen prinzipiell direkt hintereinander ausgeführt werden, so sind ihre Knoten über gerichtete Kanten miteinander verbunden. Ist n die Anzahl der Knoten und e die Anzahl der Kanten des Kontrollflussgraphen, so wird die Metrik berechnet zu $CC := e-n+1$.

Lines Of Code (LOC) ist die Anzahl Quelltextzeilen, die in einem Element des Syntaxgraphen, wie zum Beispiel einer Methode oder einer Klasse, enthalten sind. Es gibt eine Reihe verschiedener Varianten dieser Metrik in Bezug auf das Zählen von Leerzeilen oder Kommentarzeilen. Um

möglichst unabhängig von der Formatierung des Quelltextes zu sein, werden im Rahmen dieser Arbeit Leerzeilen und Kommentarzeilen nicht gezählt. Ebenso werden Zeilen nicht gezählt, die nur aus einer öffnenden geschweiften Klammer ({) bestehen.

Number Of Attributes (NOA) berechnet die Anzahl aller in einer Klasse definierten Attribute.

Number Of Methods (NOM) berechnet die Anzahl aller in einer Klasse definierten Methoden.

Number Of Parameters (NOP) liefert die Anzahl der Parameter einer Methode.

Number Of Statements (NOS) misst die Anzahl der Anweisungen, die in einem Block, einer Methode oder einer Klasse enthalten sind.

6.1.2 Verkapselung und Verteilung von Verantwortung

In diese Kategorie fallen zum einen Schwachstellen, die eine Verletzung des Information-Hiding-Prinzips beschreiben. Dazu zählen:

PublicField das ein öffentliches, nicht konstantes Attribut einer Klasse repräsentiert, und

UnencapsulatedCollection bei dem direkter Zugriff auf ein Containerobjekt gewährt wird, anstatt dieses durch Zugriffsmethoden zum Hinzufügen, Entfernen, Iterieren über enthaltene Elemente oder zur Abfrage der Anzahl enthaltener Elemente zu verkapseln.

Die folgenden Schwachstellen stehen für eine ungünstige Verteilung von Verantwortung.

DataClass Eine DataClass ist eine Klasse, die im Wesentlichen aus Attributen und Zugriffsmethoden darauf besteht (siehe Abschnitt 2.1). Durch die spezifizierten Strukturmuster wird eine Klasse als DataClass erkannt, bei der 80% der Methoden (ohne Konstruktoren) Zugriffsmethoden sind. Die Bewertung eines Fundes ergibt sich zum einen aus der Anzahl ihrer Attribute: je mehr sie enthält, desto höher ihre Bewertung. Zum anderen wird die Anzahl der vorhandenen Zugriffsmethoden und deren Bewertung

so berücksichtigt, dass eine DataClass je höher bewertet wird, desto mehr hoch bewertete Zugriffsmethoden sie enthält.

Als lesende Zugriffsmethode wird eine Methode erkannt, die den Wert eines Attributs zurückgibt. Als schreibende Zugriffsmethode wird eine Methode erkannt, die den Wert eines Attributs auf den Wert eines Parameters setzt. In beiden Fällen kann durch eine solche Methode noch viel mehr geschehen als nur der Lese- oder Schreibzugriff, so dass sie höher bewertet wird, je weniger Anweisungen sie enthält. Dies geschieht durch eine Fuzzy-Metrikbedingung über die Metrik Number of Statements (NOS).

LargeClass Bei einer LargeClass handelt es sich um eine große Klasse, die viele Zeilen Quelltext, viele Attribute und/oder viele Methoden enthält. Das Strukturmuster verwendet jeweils Fuzzy-Metrikbedingungen über die Metriken Lines Of Code (LOC), Number Of Attributes (NOA) und Number Of Methods (NOM), um die Größe einer Klasse zu bewerten. Damit eine Klasse überhaupt als LargeClass erkannt wird, muss mindestens einer der drei Metrikwerte einen Schwellwert überschreiten. Die Schwellwerte werden in Abschnitt 6.1.7 bestimmt.

LazyClass Eine LazyClass ist das Gegenstück zur LargeClass. Sie enthält wenige Zeilen Quelltext, Attribute und/oder Methoden und kann zu klein sein, um ihre Existenz zu rechtfertigen. Das Strukturmuster verwendet ebenfalls Fuzzy-Metrikbedingungen über die Metriken Lines Of Code (LOC), Number Of Attributes (NOA) und Number Of Methods (NOM), wobei die Bewertung um so höher ausfällt, je geringer der jeweilige Metrikwert ist.

LongMethod beschreibt große, komplexe Methoden. Das Strukturmuster verwendet Fuzzy-Metrikbedingungen über die Metriken Lines Of Code (LOC) und Cyclomatic Complexity (CC). Je höher die Metrikwerte ausfallen, desto höher wird eine Methode bewertet. Damit eine Methode überhaupt annotiert wird, muss mindestens einer der Metrikwerte einen Schwellwert überschreiten. Die Schwellwerte werden in Abschnitt 6.1.7 bestimmt.

LongParameterList beschreibt eine Methode mit vielen Parametern. Das Strukturmuster verwendet eine Fuzzy-Metrikbedingung über die Metrik Number Of Parameters (NOP) wodurch eine Methode umso höher bewertet wird, über je mehr Parameter sie verfügt.

GodClass Eine GodClass (siehe Abschnitt 2.1) beschreibt eine große, komplexe Klasse, die zuviel Verantwortung übernimmt. Dabei werden die Extremformen der **BehavioralGodClass**, die Verhalten ohne Daten zentralisiert, und der **DataGodClass**, die Daten ohne Verhalten zentralisiert, unterschieden.

Das Strukturmuster für eine **BehavioralGodClass** setzt zum einen eine Klasse voraus, die bereits als LargeClass erkannt wurde. Darüber hinaus manipulieren solche Klassen typischerweise die Daten anderer Klassen, die sie referenzieren.[1] Solche Manipulationen (Lese- und Schreibzugriffe) auf referenzierten Klassen werden durch weitere Strukturmuster erkannt. Eine BehavioralGodClass wird umso höher bewertet, je mehr Referenzen zu anderen Klassen sie hat und je mehr Manipulationen sie durchführt.

Das Strukturmuster der **DataGodClass** setzt eine Klasse voraus, die bereits als DataClass und als LargeClass erkannt wurde. Eine zentrale DataGodClass muss von vielen anderen Klassen referenziert werden und diese lesen und schreiben typischerweise ihre Daten. Daher wird eine DataGodClass je höher bewertet, desto häufiger sie referenziert wird und desto häufiger ihre Daten gelesen und geschrieben werden.

6.1.3 Vererbungshierarchien

Schwachstellen dieser Kategorie stehen im Zusammenhang mit der Verwendung von Vererbung.

EmptyOverridingMethod zeigt zum Beispiel eine Methode an, die in einer Subklasse leer überschrieben wird. Dies zeigt, dass diese Klasse das entsprechende Verhalten ablehnt oder nicht zur Verfügung stellen kann und somit die Abstraktion falsch gewählt ist.

MissingSubclass beschreibt eine abstrakte Klasse, von der keine konkrete Klasse erbt und damit keine Objekte erzeugt werden können. Sollte eine solche Klasse nicht über statische Methoden verfügen, die auch benutzt werden, oder ein Punkt zur Erweiterung in einem Framework sein, ist sie überflüssig.

OverriddenField Java erlaubt, dass Subklassen Attribute enthalten, die den gleichen Namen haben wie Attribute ihrer Oberklassen. Dies kann zu

[1] Dabei referenziert eine Klasse eine andere Klasse, wenn sie eine (unidirektionale) Assoziation zu ihr besitzt, realisiert durch ein Attribut ihres Typs.

Verwechslungen und damit ungewollten Effekten führen und sollte daher vermieden werden.

6.1.4 Bedingte Anweisungen

Die Schwachstellen dieser Kategorie basieren auf bedingten Anweisungen wie verkettete If/Elseif-Anweisungen, die mindestens ein gemeinsames Attribut oder eine gemeinsame Variable auf bestimmte Werte prüfen und anhand dessen unterschiedliches Verhalten ausführen.

ConditionalDispatcher (siehe Abschnitte 2.1) bezeichnet eine Methode, die verkettete If/Elseif-Anweisungen verwendet, um eine Anfrage zu behandeln. Die Strukturmuster für die Erkennung von ElseIfDispatchern werden in den Abschnitten 2.3.2 und 2.3.5 ausführlich beschrieben. Ebenso gibt es ConsecutiveIfDispatcher, bei denen aufeinanderfolgende If-Anweisungen einen Dispatcher bilden. Die Strukturmuster für deren Erkennung sind analog definiert.

Darüber hinaus können Methoden kombinierte und/oder ineinander geschachtelte ElseIf- und ConsecutiveIfDispatcher über dieselbe Variable enthalten, die durch weitere Strukturmuster als ein Dispatcher erkannt und ebenfalls auf Basis der Gesamtanzahl der If-Anweisungen sowie der darin enthaltenen Anweisungen bewertet werden.

ConditionalsForCreation beschreibt den Einsatz eines Dispatchers, um den konkreten Typ eines zu erzeugenden Objektes zu bestimmen, anstatt zum Beispiel eine AbstractFactory [GHJV95] einzusetzen. Aufbauend auf die Strukturmuster zur Erkennung von Dispatcher-Instanzen werden If-Anweisungen eines Dispatchers erkannt, die ein Objekt erzeugen, dessen Typ eine Subklasse des Rückgabetyps der Dispatcher-Methode ist. Darauf aufbauend wird ein Dispatcher als ConditionalsForCreation annotiert und auf Basis der Anzahl erzeugender If-Aweisungen bewertet.

ConditionalsInsteadOfPolymorphism repräsentiert den Einsatz eines Dispatchers, um auf Basis des Typs einer Variablen das Verhalten zur Behandlung einer Anfrage zu bestimmen, wobei sich die Typen auf die geprüft wird, in derselben Vererbungshierarchie befinden. Stattdessen könnte das entsprechende Verhalten besser in die jeweiligen Klassen verschoben und Polymorphie ausgenutzt werden. Aufbauend auf die Strukturmuster zur Erkennung von Dispatcher-Instanzen werden verbunde-

ne If-Anweisungen erkannt, die jeweils eine Typprüfung vornehmen, wobei die geprüften Typen eine gemeinsame Oberklasse im System haben. Darauf aufbauend wird eine Dispatcher-Instanz als ConditionalsInsteadOfPolymorphism annotiert und auf Basis der Anzahl typprüfender If-Anweisungspaare bewertet.

StateField Bei einem StateField wird ein Attribut einer Klasse dazu benutzt, ihren Zustand zu kodieren. Typischerweise gibt es eine Reihe von konstanten Attributen, welche die konkreten Zustände repräsentieren. In den Methoden der Klasse wird immer auf den aktuellen Zustand geprüft, um das auszuführende Verhalten zu bestimmen und gegebenenfalls ein neuer Zustand gesetzt. Hier könnte besser das State-Entwurfsmuster [GHJV95] eingesetzt werden. Durch Strukturmuster werden konstante Attribute und ein typgleiches nicht-konstantes Attribut erkannt, dessen Wert mit konstanten Attributen verglichen und/oder darauf gesetzt wird. Ein solches konstantes Attribut wird als StateCode annotiert und auf Basis der Anzahl der Vergleiche und Zuweisungen bewertet. Das nicht-konstante Attribut wird als StateField annotiert und auf Basis der Anzahl und der Bewertung der zugehörigen StateCode-Instanzen bewertet.

TypeField Bei einem TypeField wird analog zum StateField ein Attribut einer Klassen genutzt, um ihren (Sub-)Typ zu kodieren. Auch hier gibt es konstante Attribute, welche die verschiedenen Typen repräsentieren. Im Unterschied zum StateField ändert sich der Typ nicht. Eine bessere Lösung ist das Verteilen des Verhaltens auf verschiedene Klassen, die von der ursprünglichen Klasse erben. Zur Erkennung eines TypeField werden die Strukturmuster zur Erkennung von StateField-Instanzen wiederverwendet. Als TypeField wird ein StateField erkannt, das nicht geschrieben wird.

6.1.5 Objekterzeugung

Zu dieser Kategorie gehören Schwachstellen in Bezug auf die Erzeugung von Objekten. Über ConditionalsForCreation (siehe Abschnitt 6.1.4) befinden sich die folgenden Schwachstellen in dieser Kategorie.

ManyConstructors bezeichnet eine Klasse, in der es viele verschiedene Konstruktoren gibt. Insbesondere Entwickler, die neu in einem System sind, wissen nicht, welchen Konstruktor sie benutzen sollen. Das zugehörige

Strukturmuster erkennt Klassen, die mehr als drei Konstruktoren besitzen und bewertet sie darüber hinaus umso höher, desto mehr Konstruktoren es sind.

ManyUnchainedConstructors liegt vor, wenn eine Klasse viele Konstruktoren besitzt und diese sich nicht gegenseitig benutzen. Hier ist die Gefahr von dupliziertem Code zur Initialisierung von Attributen groß. Kommen Attribute hinzu, müssen unter Umständen alle Konstruktoren entsprechend erweitert werden. Wird einer vergessen, kann es zu Fehlern kommen. Durch Strukturmuster werden Konstruktoren erkannt, die keinen anderen Konstruktor benutzen. Ein weiteres Muster fordert, dass eine Klasse bereits mit ManyConstructors annotiert wurde und annotiert sie mit ManyUnchainedConstructors, wenn mindestens 50% der Konstruktoren der Klasse keinen anderen Konstruktor benutzen. Gleichzeitig wird eine solche Klasse auf Basis der Anzahl solcher Konstruktoren bewertet.

6.1.6 Falsche Verwendung von Entwurfsmustern

Die Kategorie beinhaltet Schwachstellen, die eine falsche Implementierung oder Verwendung von Entwurfsmustern beschreiben. Dazu zählt auch, das vorhandene Entwurfsmusterinstanzen ignoriert werden.

DuplicateComposite Ein DuplicateComposite liegt vor, wenn mehrere Composite-Entwurfsmuster [GHJV95] mit derselben Component-Klasse implementiert wurden. Zur Erkennung werden die Strukturmuster zur Erkennung eines Composite aus [Nie04] wiederverwendet und um ein Strukturmuster erweitert.

IgnoredFactoryMethod liegt vor, wenn ein Objekt direkt erzeugt wird, für dessen Typ es eine konkrete FactoryMethod [GHJV95] gibt. Das zugehörige Strukturmuster baut auf Strukturmuster zur Erkennung von AbstractFactory-Entwurfsmusterinstanzen aus [Nie04] auf.

Singleton Singleton-Instanzen werden mit dem in Abschnitt 2.3.4 ausführlich beschriebenen Strukturmuster erkannt. Dabei werden Klassen unterschiedlich bewertet, die über ein öffentliches Instanzattribut oder keine Zugriffsmethode oder keinen privaten Konstruktor verfügen. Auf diese Weise werden gleichzeitig fehlerhafte Singleton-Implementierungen erkannt.

6.1.7 Festlegung der Bewertungsfunktionen

Für jede Fuzzy-Metrikbedingung zur Bewertung einer Schwachstelle muss eine *Bewertungsfunktion* angegeben werden, die den Metrikwert auf einen Wert zwischen 0 und 1 abbildet (siehe Abschnitt 2.3.5). Dieser Wert geht dann multiplizert mit einem Gewicht in die Gesamtbewertung einer Schwachstelle ein.

In einigen Fällen soll eine Bewertung umso höher ausfallen, desto größer der Wert einer Metrik ist. Dies ist zum Beispiel bei der Schwachstelle LargeClass und der Metrik LOC der Fall. Dabei ist zum einen wichtig, dass egal wie hoch der eingehende Metrikwert ist, ein höherer Wert immer auch höher bewertet wird, als ein niedrigerer Wert. Auf diese Weise bleibt unabhängig von der konkreten, absoluten Bewertung einer Schwachstelle immer eine Sortierung von Schwachstellen möglich, so dass die am höchsten bewerteten Schwachstellen vom Re-Engineer als erstes inspiziert werden können. Dies wird erreicht, indem eine monoton steigende Bewertungsfunktion verwendet wird, die sich asymptotisch dem Wert 1 annähert. Eine solche Funktion ist

$$f(x) = \frac{1 + dE}{1 + e(\frac{-x+dX}{dC})} + dY, \tag{6.1}$$

wobei x dem Metrikwert entspricht.

Zum anderen sollte eine Bewertungsfunktion im *interessanten Bereich* gut differenzieren, so dass unterschiedliche Metrikwerte in diesem Bereich deutlich unterschiedlich bewertet werden. Hier müssen dazu die Parameter dX, dY, dE und dC der Funktion 6.1 für jede Metrik geeignet bestimmt werden.

Welches der interessante Bereich ist, ist von Metrik zu Metrik verschieden und kann nur schwer allgemeingültig angegeben werden. Allerdings besagt ein von dem russichen Mathematiker Tschebyschow bewiesenes Theorem [Tsc67], dass unabhängig von der Verteilung der Werte einer Grundgesamtheit X, 75% der darin enthaltenen Werte in einem Bereich von zwei Standardabweichungen (2σ) um das arithmetische Mittel ($\bar{x}$) herum liegen ($\bar{x} \pm 2\sigma$).

Da hier große Werte von Interesse sind, werden die Parameter so bestimmt, dass die Bewertungsfunktion besonders gut im Bereich $\bar{x} + 2\sigma$ differenziert. Dazu werden $dX := \bar{x} + \sigma$ und $dC := \frac{1}{2}\sigma$ gesetzt, dY wird so bestimmt, dass $f(0) \approx 0$ gilt, und $dE := -dY$. Diese Festlegung der Parameter führt zu dem in Abbildung 6.1 skizzierten Verlauf der Bewertungsfunktion: ein $\bar{x}$ entsprechender Metrikwert wird mit ca. 7,5% bewertet, $\bar{x} + \sigma$ liegt bei ca. 50% und $\bar{x} + 2\sigma$ bei ca. 87,5%.

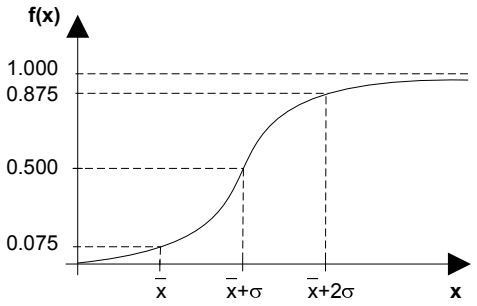

Abbildung 6.1: Skizze des Funktionsgraphen der Bewertungsfunktion 6.1

Zur Bestimmung von $\bar{x}$ und σ für die zur Bewertung von Schwachstellen verwendeten Metriken wurden die Metriken für die fünf eingangs genannten Systeme berechnet. Dabei wurden für jedes System und jede Metrik jeweils $\bar{x}$ und σ bestimmt. Aus diesen Werten wurde wiederum jeweils das arithmetische Mittel gebildet und zur Bestimmung der Bewertungsfunktionen verwendet. Die Tabellen 6.2, 6.3 und 6.4 zeigen die berechneten arithmetischen Mittel und Standardabweichungen. Tabelle 6.5 zeigt die daraus bestimmten Parameter der Bewertungsfunktionen für die jeweiligen Schwachstellen. Bei LargeClass und LongMethod wird zudem verlangt, das mindestens einer der zur Bewertung verwendeten Metrikwerte das jeweilige arithmetische Mittel überschreitet. Für eine LongParameterList müssen mindestens 3 Parameter vorhanden sein.

		ArgoUML	JHD	ShuttleSim	SWT	WebLV	$\bar{x}$
LOC	$\bar{x}$	58.61	50.95	82.11	172.36	203.92	113.59
	σ	94.48	70.93	146.18	520.95	412.82	249.07
NOA	$\bar{x}$	2.20	1.64	5.01	12.08	5.61	5.31
	σ	4.68	2.82	9.47	77.33	8.38	20.54
NOM	$\bar{x}$	6.20	10.00	7.31	13.25	8.61	9.07
	σ	8.47	10.73	9.52	50.48	10.45	17.93

Tabelle 6.2: Arithmetische Mittel und Standardabweichungen von Klassenmetriken

Anders als bei den bisher gezeigten Bewertungsfunktionen muss eine Bewertung bei Zugriffsmethoden oder einer LazyClass umso höher ausfallen, desto geringer der eingehende Metrikwert ist. Dazu wird die folgende Funktion be-

		ArgoUML	JHD	ShuttleSim	SWT	WebLV	$\bar{x}$
CC	$\bar{x}$	1.99	1.29	2.00	3.16	4.23	2.53
	σ	2.96	1.01	2.84	6.31	7.52	4.13
LOC	$\bar{x}$	8.28	4.49	9.63	11.74	22.81	11.39
	σ	12.71	5.70	19.93	26.71	41.67	21.34
NOP	$\bar{x}$	0.83	0.61	0.89	1.59	1.80	1.14
	σ	0.98	0.95	1.22	1.93	1.96	1.41

Tabelle 6.3: Arithmetische Mittel und Standardabweichungen von Methodenmetriken

		ArgoUML	JHD	ShuttleSim	SWT	WebLV	$\bar{x}$
NOS	$\bar{x}$	3.82	2.26	4.76	7.98	7.91	5.35
	σ	7.17	4.16	10.92	18.03	19.49	11.95

Tabelle 6.4: Arithmetisches Mittel und Standardabweichung der Blockmetrik NOS

Schwachstelle	Metrik	dX	dY	dE	dC
LargeClass	LOC	362.67	-0.0542	0.0542	124.54
	NOA	25.84	-0.08	0.08	10.27
	NOM	27.00	-0.049	0.049	8.96
LongMethod	CC	6.66	-0.039	0.039	2.06
	LOC	32.73	-0.046	0.046	10.67
LongParameterList	NOP	2.55	-0.026	0.026	0.7
DispatcherIf	NOS	17.30	-0.055	0.055	5.98

Tabelle 6.5: Parameter der Bewertungsfunktionen auf Basis von Funktion 6.1

nutzt:

$$f(x) = \frac{1 + dE}{1 + e(\frac{x+dX}{dC})} + dY, \tag{6.2}$$

wobei x dem Metrikwert entspricht. Diese Funktion unterscheidet sich von der Funktion 6.1 durch das positive Vorzeichen von x. Dies führt in etwa zu einem in Abbildung 6.1 an der 50%-Bewertungslinie gespiegelten Verlauf des Funktionsgraphen.

Tabelle 6.6 zeigt die Parameter der Bewertungsfunktionen auf Basis von Funktion 6.2. Für LazyClass wurden die Parameter so bestimmt, dass $\bar{x}$ jeweils mit ca. 5% bewertet wird. Bei den Zugriffsmethoden wurde unterstellt, dass eine lesende Methode kaum mehr als eine Anweisung benötigt, während bei einer schreibenden Methode häufig noch Anweisungen enthalten sind, die den neuen Wert validieren.

Muster	Metrik	dX	dY	dE	dC
LazyClass	LOC	-56.8	0.017	0.0	14.2
	NOA	-2.62	0.017	0.0	0.65
	NOM	-4.53	0.017	0.0	1.13
GetMethod	NOS	-3.00	0.0	0.0	0.25
SetMethod	NOS	-10.00	0.0	0.0	2.00

Tabelle 6.6: Parameter der Bewertungsfunktionen auf Basis von Funktion 6.2

Schließlich wird noch eine Funktion zur Bewertung von Mengen von Objekten oder Strukturmusterannotationen benötigt. Diese Funktion wird durch

$$f(x) = \frac{2.0}{1.0 + e(\frac{-x}{5.0})} - 1.0 \tag{6.3}$$

definiert, wobei x der Anzahl der Elemente der zu bewertenden Menge entspricht. Die Funktion bewertet eine Menge mit annähernd 100%, die 35 (mit 100% bewertete) Elemente enthält. Sie ist monoton steigend und nähert sich asymptotisch der 1.

Die hier bestimmten Funktionen erheben keinen Anspruch auf Allgemeingültigkeit, sondern dienen lediglich dazu, bei den hier analysierten Systemen zu einer vernünftigen Bewertung und insbesondere Sortierung von Schwachstellen zu kommen. Auch ob die hier zur Bestimmung eingesetzte Systematik allgemein sinnvoll ist, müsste noch weiter untersucht werden.

6.2 Ergebnisse der Schwachstellenanalyse

Im Folgenden wird über die Ergebnisse der mit Hilfe des zuvor beschriebenen Katalogs durchgeführten Schwachstellenanalyse von SWT berichtet. Aufgrund der großen Menge der Ergebnisse werden bei bewerteten Schwachstellen nur jeweils die fünf bis zehn am höchsten bewerteten benannt. Anschließend wird ein Fazit gezogen.

6.2.1 Schwachstellen in SWT

In der Kategorie Verkapselung und Verteilung von Verantwortung (siehe Abschnitt 6.1.2) wurden in SWT 1.717 öffentliche Attribute (PublicField) gefunden, von denen 53 statisch sind und zumindest innerhalb der SWT-Implementierung nicht geschrieben werden. Bei diesen Attributen könnte es sich um eigentliche Konstanten handeln, bei denen der final-Modifier vergessen wurde. Eine UnencapsulatedCollection wurde in SWT nicht erkannt.

Als LongMethod wurden 2.347 Methoden erkannt. Bei diesen Methoden ist mindestens einer der berechneten Werte für CC und LOC höher als der in Abschnitt 6.1.7 bestimmte Durchschnitt. Die Bewertungen reichen von 9.9% bis 100%. Die 100% wird in vier Fällen erreicht, weil die eingehenden Metrikwerte so hoch sind, dass die durch die jeweiligen Bewertungsfunktionen berechneten Werte zu nahe an 1 liegen, um den Abstand noch darstellen zu können. Da dies aber nur in vier extremen Ausnahmefällen so ist, wurde von einer Anpassung der Bewertungsfunktion, um auch in diesen Fällen noch eine sichtbare Differenzierung zu erreichen, abgesehen. Die fünf am höchsten bewerteten Funde werden in Tabelle 6.7 mit ihrer Bewertung und den zugehörigen Metrikwerten genannt. Eine Zerlegung dieser Methoden in mehrere kleinere Methoden (zum Beispiel mit Hilfe von ExtractMethod) würde ihr Verstehen erleichtern sowie ihre Testbarkeit verbessern.

Darüber hinaus wurden 1.794 Methoden mit LongParameterList annotiert; diese haben mindestens 3 Parameter. Die Bewertungen reichen von 66% bis 100% in vier Fällen (siehe Tabelle 6.8). Diese vier mit 100% bewerteten Methoden verfügen über 30 Parameter, danach folgen einige Methoden mit 17 Parametern.

Als LargeClass wurden 361 Klassen erkannt, bei denen mindestens einer der Metrikwerte überdurchschnittlich ist. Die Bewertungen reichen von 5% bis zu knapp unter 100%. Die fünf am höchsten bewerteten Funde werden in Tabelle 6.9 aufgelistet.

Demgegenüber wurden 529 Klassen als LazyClass annotiert und von 5% bis

Bew. (%)	Klasse[a]	Methode	CC	LOC
100,0000000000000	... browser.Mozilla	create	135	873
100,0000000000000	... widgets.Tree	CDDS...[b]	155	541
100,0000000000000	... layout.GridLayout	layout	152	514
100,0000000000000	... graphics.ImageData	blit	131	449
99,9999999999997	... graphics.TextLayout	draw	82	383

[a]Die Paketnamen beginnen alle mit org.eclipse.swt.
[b]vollständiger Name CDDS_ITEMPOSTPAINT

Tabelle 6.7: Top 5 LongMethod-Schwachstellen in SWT

Bew. (%)	Klasse[a]	Methode	NOP
100,0000000	... graphics.ImageData	blit[b]	30
100,0000000	... graphics.ImageData	blit	30
100,0000000	... graphics.ImageData	blit	30
100,0000000	... graphics.ImageData	blit	30
99,9999998	... graphics.ImageData	setAllFields	17
99,9999998	... internal.mozilla.XPCOM	VtblCall	17
99,9999998	... internal.mozilla.XPCOM	VtblCall	17
99,9999995	... graphics.ImageData	internal_new	16

[a]Die Paketnamen beginnen alle mit org.eclipse.swt.
[b]Es existieren vier Methoden mit diesem Namen in derselben Klasse mit derselben Anzahl Parameter

Tabelle 6.8: Top 8 LongParameterList-Schwachstellen in SWT

Bew. (%)	Klasse[a]	LOC	NOA	NOM
99,9999	... internal.win32.OS	3.817	1.909	1.108
99,9997	... widgets.Display	2.781	147	151
99,9920	... internal.image.JPEGDecoder	3.727	158	102
99,9807	... custom.CTabFolder	2.793	105	109
99,8223	... custom.StyledText	5.124	80	271

[a]Die Paketnamen beginnen alle mit org.eclipse.swt.

Tabelle 6.9: Top 5 LargeClass-Schwachstellen in SWT

99% bewertet, bei denen mindestens einer der Metrikwerte unterhalb der in Abschnitt 6.1.7 bestimmten arithmetischen Mittelwerte liegt. Tabelle 6.10 nennt die 6 am höchsten bewerteten Funde. Die Klasse CTabFolderAdapter ist eine leere Implementierung einer Listener-Schnittstelle. Da diese aber nur über eine Methode verfügt, ist die Existenz der Klasse eher fragwürdig. Die vier nächsten Klassen sind Subklassen, die lediglich einen Konstruktor implementieren und innerhalb von SWT auch nicht weiter spezialisiert werden. Die Klasse Platform definiert lediglich eine Konstante. Sie ist darüber hinaus die Wurzel einer Vererbungshierarchie bestehend aus sieben weiteren Klassen.

Bew. (%)	Klasse[a]	LOC	NOA	NOM
98,95	... custom.CTabFolderAdapter	4	0	1
98,90	... internal.ole.win32.IEnumSTATSTG	5	0	1
98,90	... internal.ole.win32.IEnumVARIANT	5	0	1
98,90	... internal.ole.win32.IMoniker	5	0	1
98,90	... internal.ole.win32.IEnumFORMATETC	5	0	1
97,89	... internal.Platform	3	1	0

[a]Die Paketnamen beginnen alle mit org.eclipse.swt.

Tabelle 6.10: Top 6 LazyClass-Schwachstellen in SWT

Die vier in Tabelle 6.11 genannten Klassen wurden als DataClass erkannt. Die vier Klassen dienen dazu, Layoutinformationen für Elemente einer Benutzeroberfläche zu verkapseln. Bei der Klasse GridData handelt es sich sicherlich um eine Datenklasse. Sie verfügt über wenig Verhalten und viele Attribute (40, von denen 17 Konstanten sind). Die Klasse FormData ist eher ein Grenzfall: sie verfügt zwar über relativ viele Attribute aber die erkannten Zugriffsmethoden beinhalten viele Berechnungen und damit viel Verhalten. Daher sind sie entsprechend niedrig bewertet worden, was zu einer insgesamt eher niedrigen Bewertung führt. Die beiden weiteren Funde enthalten zum einen weniger Attribute und die Zugriffsmethoden führen ebenfalls zusätzliche Berechnungen durch, so dass die niedrige Bewertung gerechtfertigt ist. Insgesamt besteht hier kein Verbesserungsbedarf.

In SWT wurden 14 BehavioralGodClass-Schwachstellen mit einer Bewertung von 19% bis 87% erkannt. Tabelle 6.12 zeigt die fünf am höchsten bewerteten Funde. Die Klasse StyledText implementiert ein Element einer Benutzeroberfläche zur Anzeige und Bearbeitung von Text. Die Klassen CTabFolder und Table realisieren ebenfalls Elemente von Benutzeroberflächen und StyledTextRenderer implementiert einen Algorithmus zur Ausgabe von Text auf einem

Bew. (%)	Klasse	NOA
41,18	org.eclipse.swt.layout.GridData	40
27,76	org.eclipse.swt.layout.FormData	22
4,13	org.eclipse.swt.layout.FillData	6
4,13	org.eclipse.swt.custom.CLayoutData	6

Tabelle 6.11: DataClass-Schwachstellen in SWT

Bildschirm oder einem Drucker. Die Klasse Display realisiert die Verbindung zwischen SWT und dem Betriebssystem. Die Klassen sind alle sehr groß und lesen und manipulieren Daten von benachbarten Klassen. Es sollte daher weiter untersucht werden, inwieweit Verhalten in diese benachbarten Klassen verschoben werden kann.

Bew. (%)	Klasse
86,89	org.eclipse.swt.custom.StyledText
84,71	org.eclipse.swt.widgets.Display
82,88	org.eclipse.swt.custom.CTabFolder
78,43	org.eclipse.swt.custom.StyledTextRenderer
65,87	org.eclipse.swt.widgets.Table

Tabelle 6.12: Top 5 BehavioralGodClass-Schwachstellen in SWT

In der Kategorie Vererbungshierarchien (siehe Abschnitt 6.1.3) wurden die in Tabelle 6.13 aufgeführten EmptyOverridingMethod-Schwachstellen erkannt.

Klasse	Methode
org.eclipse.swt.internal.image.JPEGFixedSizeSegment	setSegmentLength
org.eclipse.swt.internal.theme.ToolBarDrawData	draw
org.eclipse.swt.widgets.Composite	checkSubclass
org.eclipse.swt.widgets.Decorations	checkBorder
org.eclipse.swt.widgets.Item	checkSubclass
org.eclipse.swt.widgets.Shell	forceResize
org.eclipse.swt.widgets.Shell	releaseParent
org.eclipse.swt.widgets.Shell	setParent

Tabelle 6.13: EmptyOverridingMethod-Schwachstellen in SWT

Darüber hinaus wurden 16 MissingSubclass-Schwachstellen gefunden. Bei den annotierten Klassen handelt es sich ausnahmslos um leere Implementierungen

von Listener-Schnittstellen, mit mehr als einer Methode. Diese Klassen werden in SWT-Anwendungen typischerweise zahlreich spezialisiert.

Des Weiteren wurden 258 OverriddenField-Schwachstellen gefunden. Dazu zählt zum Beispiel das Attribut LAST_METHOD_ID der Klasse org.eclipse.swt.internal.mozilla.nsISupports, das in der erbenden Klasse org.eclipse.swt.internal.mozilla.nsIPromptService erneut deklariert wird.

In der Kategorie Bedingte Anweisungen (siehe Abschnitt 6.1.4) wurden 973 ConditionalDispatcher-Schwachstellen erkannt, die von 16% bis 74% bewertet wurden. Die fünf am höchsten bewerteten Fundstellen sind in Tabelle 6.14 aufgeführt. Die Methode checkGC bestimmt mit Hilfe von 20 If-Anweisungen auf Basis der state-Variable Linienart und -breite für das Zeichnen von Grafiken. Die Methode handleDOMEvent bestimmt in 12 If-Anweisungen die Reaktion auf ein als Parameter übergebenes Event; ebenso die Methode HandleEvent. Diese drei Methoden sind als korrekte Funde anzusehen, deren Verbesserung sinnvoll wäre.

Die beiden nächsten Funde sind sowohl aufgrund ihres Umfangs als auch ihres Inhalts als weniger signifikant beziehungsweise als false-positives[2] einzustufen. resizeRectangles bestimmt in 16 If-Anweisungen mit Hilfe der Variablen cursorOrientation die Neuberechnung der Größe von Rechtecken. Dieser vermeintliche Dispatcher enthält zwar mehr If-Anweisungen als die beiden zuvor beschriebenen, wird aber dennoch zurecht niedriger bewertet, da diese weniger Anweisungen enthalten. Die Methode shape dient der Generierung von Schriftzeichen. Sie überprüft in 6 If-Anweisungen, ob die Boolesche Variable shapeSucceed den Wert false hat. Wenn ja wird der Werte der Variablen durch zum Beispiel Methodenaufrufe neu bestimmt und dann in der nächsten If-Anweisung überprüft. Es handelt sich also nicht um einen Dispatcher.

Auch die übrigen niedriger bewerteten ConditionalDispatcher-Schwachstellen wurden stichprobenartig überprüft und als nicht signifikant eingestuft.

Die Methode getItem(int, int) in der Klasse org.eclipse.swt.dnd.DropTargetEffect wurde als ConditionalsInsteadOfPolymorphism-Schwachstelle erkannt. Sie besteht aus zwei If-Anweisungen, die ein Attribut vom Typ org.eclipse.swt.widgets.Control auf dessen konkrete Subtypen org.eclipse.swt.widgets.Tree und org.eclipse.swt.widgets.Table prüfen und demgemäß anderes Verhalten auslösen. Es handelt sich um einen korrekt erkannten Fund, der aber aufgrund seiner geringen Größe nicht verbesserungswürdig ist und auch nur mit ca. 6,74% bewertet worden ist.

Darüber hinaus wurden 17 StateField-Schwachstellen erkannt, deren Bewer-

[2] Als *false-positive* wird eine fälschlicherweise erkannte Schwachstelle bezeichnet.

Bew. (%)	Klasse[a]	Methode	Variable	Ifs
74,29	...graphics.GC	checkGC	state	20
51,95	...browser.IE	handleDOMEvent	eventType	12
50,96	...browser.Mozilla	HandleEvent	typeString	12
44,65	...widgets.Tracker	resizeRectangles	cursorOrientation	16
37,28	...graphics.TextLayout	shape	shapeSucceed	6

[a]Die Paketnamen beginnen alle mit org.eclipse.swt.

Tabelle 6.14: Top 5 ConditionalDispatcher-Schwachstellen in SWT

tungen von 8% bis 30% reichen. Tabelle 6.15 nennt die fünf am höchsten bewerteten zusammen mit der Anzahl der jeweils identifizierten Zustandskonstanten. In der Klasse StyledText wird das Attribut caretAlignment verwendet, um die Ausrichtung des Cursors in einem editierbaren Textfeld zu bestimmen. Das Attribut wird selten gelesen aber sehr häufig geschrieben. In OleClientSite kodiert state den Zustand eines OLE-Dokuments. Das Attribut signature in Klasse Callback ist ein false-positive. Die Attribute chevronImageState und minImageState sind dagegen korrekte Funde, die den Zustand von Schaltflächen eines CTabFolder repräsentieren.

Die manuelle Inspektion dieser und weiterer Funde hat zum einen ergeben, dass die mit mehr als 10% bewerteten Funde im Wesentlichen als korrekt angesehen werden können. Die Funde darunter sind false-positives. Zum anderen wurde festgestellt, dass häufig switch/case-Anweisungen verwendet werden, die von den bisherigen Strukturmustern noch nicht berücksichtigt werden. Dies sollte geändert werden, um Zustandsattribute vollständiger zu erkennen. Eine StateField-Schwachstelle wird zudem durch die Anzahl und Bewertung der zugehörigen Zustände bewertet, die wiederum durch die Anzahl der Zugriffe (Überprüfen des Zustands, Setzen des Zustands) bewertet werden. Somit würde durch die Berücksichtigung von switch/case-Anweisungen auch eine adäquatere Bewertung erreicht.

In der Kategorie Objekterzeugung (siehe Abschnitt 6.1.5) wurden 17 Klassen sowohl als ManyConstructors als auch als ManyUnchainedConstructors-Schwachstellen erkannt. Tabelle 6.16 zeigt die fünf am höchsten bewerteten ManyUnchainedConstructors-Funde jeweils mit ihrer Bewertung und der Anzahl der Konstruktoren, die keinen anderen Konstruktor benutzen. Dies trifft im übrigen bei allen 17 annotierten Klassen auf alle Konstruktoren zu.

In der Kategorie Falsche Verwendung von Entwurfsmustern (siehe Abschnitt 6.1.6) wurde eine DuplicateComposite-Schwachstelle erkannt, bei der

Bew. (%)	Klasse[a]	Attribut	Zustände
29,88	...custom.StyledText	caretAlignment	2
22,13	...ole.win32.OleClientSite	state	4
18,89	...internal.Callback	signature	6
15,71	...custom.CTabFolder	chevronImageState	3
15,71	...custom.CTabFolder	minImageState	3

[a]Die Paketnamen beginnen alle mit org.eclipse.swt.

Tabelle 6.15: Top 5 StateField-Schwachstellen in SWT

Bew. (%)	Klasse	Konstr.
87,52	org.eclipse.swt.ole.win32.Variant	12
74,80	org.eclipse.swt.graphics.Image	8
70,32	org.eclipse.swt.widgets.Shell	7
70,32	org.eclipse.swt.layout.FormAttachment	7
65,27	org.eclipse.swt.graphics.ImageData	6

Tabelle 6.16: Top 5 ManyUnchainedConstructors-Schwachstellen in SWT

es sich allerdings um ein false-positive handelt. Als Composite-Klassen wurden org.eclipse.swt.widgets.Composite und org.eclipse.swt.custom.SashForm mit derselben Komponenten-Klasse org.eclipse.swt.widgets.Control erkannt. Dabei wurde aber nicht berücksichtigt, dass SashForm von Composite erbt und die Composite-Funktionalität wiederverwendet und erweitert.

Darüber hinaus wurden die 9 in Tabelle 6.17 aufgeführten Singleton-Instanzen erkannt. Die vier am höchsten bewerteten Klassen entsprechen vollkommen korrekten Singleton-Implementierungen. Bei den beiden geringer bewerteten Funden HTMLTransfer und URLTransfer ist das Instanzattribut im Paket sichtbar. Damit haben die beiden Klassen nicht mehr die volle Kontrolle über ihre einzige Instanz. Die Klasse Display ist kein richtiges Singleton. Es gibt eine „Default“-Instanz, die in einem Attribut mit Paketsichtbarkeit gespeichert wird und die über eine Zugriffsmethode abgerufen werden kann. Darüber hinaus kann es beliebig viele Instanzen der Klasse geben, da es öffentliche Konstruktoren gibt. Die Klasse LRESULT hält zwei Instanzen von sich selbst als öffentliche konstante Attribute und einen öffentlichen Konstruktor; sie ist sicher kein Singleton. Die Klasse Device schließlich hält analog zu Display eine „Default“-Instanz von sich selbst. Sie besitzt eine Zugriffsmethode, die zusätzliche Berechnungen durchführt und damit zu so einer niedrigen Be-

wertung des Fundes führt. Des Weiteren gibt es öffentliche Konstruktoren. Hier handelt es sich ebenfalls nicht um ein Singleton. Aus Kommentaren im Quelltext geht zudem hervor, dass das Instanzattribut und die Zugriffsmethode zukünftig entfernt werden sollen.

Bew. (%)	Klasse
99,97	org.eclipse.swt.dnd.FileTransfer
99,97	org.eclipse.swt.dnd.TextTransfer
99,97	org.eclipse.swt.dnd.ImageTransfer
99,97	org.eclipse.swt.dnd.RTFTransfer
98,25	org.eclipse.swt.dnd.HTMLTransfer
98,25	org.eclipse.swt.dnd.URLTransfer
93,08	org.eclipse.swt.widgets.Display
10,34	org.eclipse.swt.internal.win32.LRESULT
10,34	org.eclipse.swt.graphics.Device

Tabelle 6.17: Singleton-Instanzen in SWT

6.2.2 Fazit

Ein Fazit der durchgeführten Analysen ist zunächst, dass mit Hilfe der strukturbasierten Mustererkennung und der spezifizierten Muster zahlreiche signifikante Schwachstellen in verschiedenen Systemen gefunden werden konnten. Signifikant bedeutet dabei zum einen, dass es sich um korrekt erkannte Schwachstellen handelt (keine false-positives). Zum anderen erscheint bei einigen dieser Funde auch eine Verbesserung sinnvoll. Über die zuvor aufgeführten, in SWT erkannten Schwachstellen hinaus wurden insbesondere weitere ConditionalDispatcher- und StateField-Schwachstellen in den übrigen analysierten Systemen gefunden. In den Systemen WebLV und ShuttleSim wurden dabei vergleichsweise viele signifikante ConditionalDispatcher-Schwachstellen gefunden, die dem Anti Pattern besonders gut entsprechen.

Des Weiteren wurde auf Basis der zuvor beschriebenen Bewertung eine sinnvolle Sortierung der gefundenen Schwachstellen vorgenommen. Signifikante Funde wurden durchweg hoch bewertet, so dass die Sortierung genutzt werden konnte, um voraussichtlich signifikante Schwachstellen zuerst zu inspizieren. Durch stichprobenartige manuelle Inspektion auch der niedriger bewerteten Fundstellen hat sich weiterhin gezeigt, dass ab einer bestimmten Bewertung alle niedriger bewerteten Schwachstellen vernachlässigt werden konnten.

Die absolute Höhe der Bewertung verschiedener Schwachstellen sowie ihre Differenzierung kann in einigen Fällen wie zum Beispiel LongParameterList oder auch StateField noch verbessert werden. Die StateField-Funde etwa waren überwiegend korrekt (wenige false-positives), allerdings war keiner ohne weiteres zur Transformation geeignet, da das zustandsabhängige Verhalten sehr stark mit dem übrigen Verhalten vermischt war. Außerdem wurde der Zustand bei den erkannten Funden überwiegend geschrieben und kaum gelesen. Zur Verbesserung der Bewertung sollte das Verhältnis der Zustandsänderungen zu den Zustandsüberprüfungen in die Bewertung eingehen. Darüber hinaus wurden in einigen Fällen Switch-Anweisungen vermischt mit If-Anweisungen eingesetzt, was durch die spezifizierten Strukturmuster nicht erkannt wurde.

Um die hier getroffenen Aussagen zur Präzision der Erkennung und der Güte der Bewertung zu präzisieren und quantitativ zu untermauern sind weitere, aufgrund der notwendigen manuellen Inspektion von Ergebnissen aufwändige Analysen erforderlich, die im Rahmen dieser Arbeit nicht mehr vorgenommen werden konnten. Dabei sollte auch untersucht werden inwieweit Bewertungsfunktionen durch den Einsatz geeigneter Verfahren erlernt werden können.

6.3 Transformationsspezifikationen

Von den gefundenen Schwachstellen sind einige der ConditionalDispatcher-Funde, insbesondere in den Systemen WebLV und ShuttleSim, gut für die Verbesserung durch eine automatisierte Transformation geeignet. Bei den Dispatcher-Schwachstellen gibt es einige, die aufeinanderfolgende If-Anweisungen und Elseif-Anweisungen gemischt einsetzen und/oder umfangreichere Ausdrücke zur Überprüfung der Dispatcher-Variable verwenden als Gleicheitsprüfungen. Für diese Dispatcher ist eine Transformation in eine Chain-Of-Responsibility-Struktur zu empfehlen. Einige der Dispatcher liefern zudem ein Ergebnis zurück. Bevor diese extrahiert werden können, müssen weitere Analysen durchgeführt werden. Bei einem Fund wird aufgrund der Belegung einer Variablen entschieden, welcher von verschiedenen zur Verfügung stehenden Algorithmen eingesetzt wird. Hier wäre der Einsatz des Strategy-Entwurfsmusters sinnvoll.

Bei Dispatcher-Schwachstellen, die entweder nur aufeinanderfolgende If-Anweisungen, deren Bedingungen Gleichheitsprüfungen vornehmen und von denen garantiert nur eine erfüllt sein kann, oder die nur Elseif-Anweisungen mit Gleichheitsprüfungen einsetzen, ist eine Transformation in eine Lösung auf Basis des Command-Entwurfsmusters möglich (siehe Abschnitt 2.1). Zur

Erprobung der Verbesserung erkannter Schwachstellen wurden Transformationsdiagramme erstellt, die solche Dispatcher-Instanzen in eine kommandobasierte Struktur transformieren. Abbildung 6.2 zeigt einen Überblick über die erstellten Transformationen in Form eines *Transformationskatalogs*.

Ein Transformationskatalog fasst analog zu einem Strukturmusterkatalog eine Menge von Transformationen zusammen. Dabei werden Transformationen analog zu Transformationsaufrufen durch orangefarbene Rechtecke mit abgerundeten Ecken und ≪transformation≫ beschriftet dargestellt. Strukturmuster, auf die Transformationen aufbauen, werden genauso wie in Strukturmusterkatalogen (siehe Abschnitt 2.3.2) als weiße Rechtecke mit ≪pattern≫ beschriftet dargestellt. Zudem gibt es Strukturmuster, für die eine Mustererzeugungstransformation (siehe Abschnitt 3.2.4) definiert wurde. Diese werden als Strukturmuster mit einem orangenen Rechteck mit abgerundeten Ecken hinterlegt dargestellt. Darüber hinaus enthält ein Transformationskatalog Beziehungen zwischen seinen Elementen: eine calls-Beziehung zeigt von einer Transformation auf eine von ihr aufgerufene Transformation, eine creates-Beziehung zeigt von einer Transformation auf ein Muster, das von ihr erzeugt wird und eine parameter-Beziehung zeigt von einer Transformation auf ein Muster, von dem diese eine Instanz als Parameter erwartet.

Die Transformation ReplaceDispatcherByCommands überführt mit Hilfe der übrigen Transformationen des Katalogs einen Dispatcher in eine kommandobasierte Struktur. Sie erwartet die zu transformierende Dispatcher-Instanz als Parameter.

In einem ersten Schritt verschiebt sie mit Hilfe der Transformation MoveDeclarationsIntoDispatcherIfs, die wiederum MoveDeclaration benutzt, die Deklarationen von in den Blöcken innerhalb der If-Anweisungen des Dispatcher benutzten lokalen Variablen dorthin, sofern dies möglich ist. Da die Blöcke später in eigene Methoden extrahiert werden, müssten für diese Variablen sonst Parameter angelegt werden, was so – sofern möglich – vermieden wird.

Anschließend wird der Dispatcher durch Erzeugen einer ExtractedDispatcher-Instanz in eine separate Methode in derselben Klasse extrahiert. Dazu werden die If-Anweisungen des Dispatchers mit Hilfe von ExtractDispatcher in eine neue Methode extrahiert. Bei einem ConsecutiveIfDispatcher, der aus aufeinanderfolgenden If-Anweisungen besteht, wird zunächst ein Block um diese Anweisungen gelegt (SurroundByBlock). Der Block wird dann mit ExtractVoidMethod extrahiert. Im Falle eines Elseif-Dispatchers wird die erste If-Anweisung (und damit auch alle weiteren) extrahiert. Um ein späteres Extrahieren und Verschieben der DispatcherIfs vorzubereiten, werden danach zum einen alle direkten Zugriffe auf Attribute innerhalb der neuen Methode durch Aufruf von

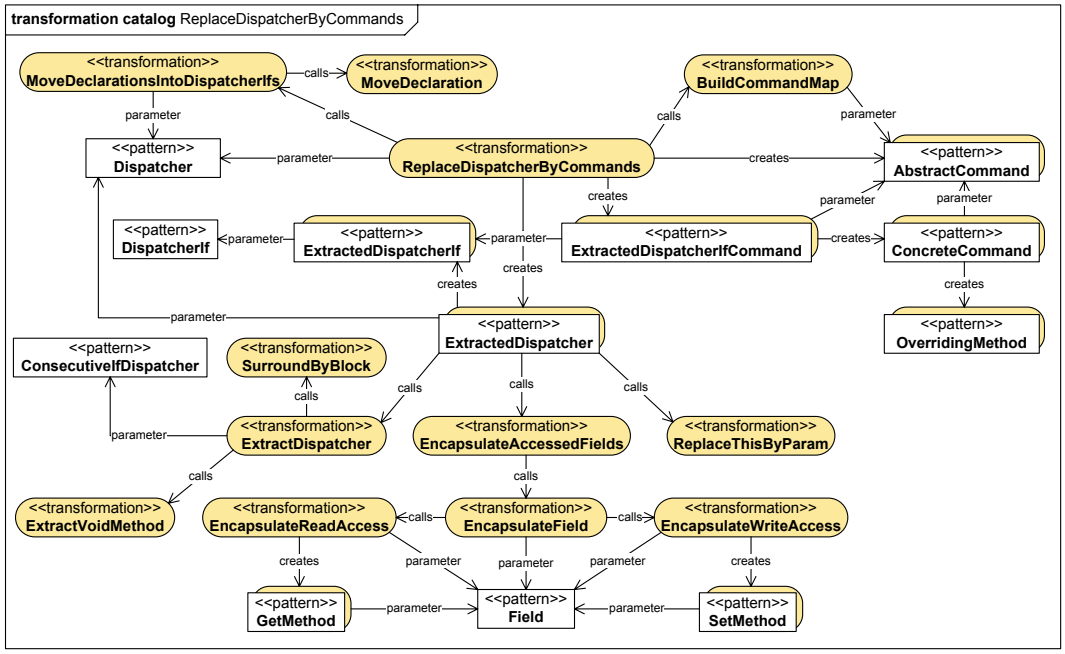

Abbildung 6.2: Transformationskatalog zur Transformation von Dispatchern

EncapsulateAccessedFields verkapselt. Zum anderen werden durch ReplaceThisByParam alle Zugriffe auf this durch Zugriffe auf einen neuen Parameter der extrahierten Methode vom Typ der umgebenden Klasse ersetzt.

Bei der Verkapselung wird für jedes zugegriffene Attribut die Transformation EncapsulateField aufgerufen, die erst alle Lesezugriffe mit EncapsulateReadAccess und danach alle Schreibzugriffe mit EncapsulateWriteAccess verkapselt. Von den beiden Transformationen werden jeweils eine GetMethod sowie eine SetMethod für das Attribut erzeugt. Die EncapsulateField-Transformation kann so auch für die Verkapsekung von PublicField-Schwachstellen genutzt werden.

Schließlich werden auch alle DispatcherIfs in eigene Methoden extrahiert, indem ExtractedDispatcherIf-Instanzen erzeugt werden.

Danach fährt die ReplaceDispatcherByCommands-Transformation mit der Erzeugung eines AbstractCommand als Basisklasse für die Kommandoklassen fort, um danach für jede der bereits extrahierten If-Anweisungen mit ExtractedDispatcherIfCommand eine konkrete Kommandoklasse zu erzeugen. Dabei wird zunächst mit ConcreteCommand die Klasse erzeugt und danach wird der Inhalt des then-Blocks der If-Anweisung in diese Klasse verschoben und durch einen Aufruf ersetzt.

Abschließend wird mit Hilfe von BuildCommandMap eine qualifizierte Assoziation zur abstrakten Kommandoklasse erzeugt und eine Initialisierung dieser Assoziation mit den entsprechenden Kommandoobjekten generiert. Die so restrukturierte Methode delegiert nun die Anfrage an ein in der Assoziation hinterlegtes Kommandoobjekt.

Die Transformationen wurden manuell erstellt. Die Transformationsspezifikation anhand von Beispielen konnte nicht eingesetzt und evaluiert werden, da keine vollständige Implementierung zur Verfügung stand (siehe Abschnitt 3.4.1). Die Transformationen wurden erfolgreich eingesetzt, um einige der erkannten ConditionalDispatcher- und PublicField-Schwachstellen zu verbessern.

6.3.1 Fazit

Die Ausdrucksmächtigkeit der in Kapitel 3 beschriebenen Sprache ist grundsätzlich ausreichend um komplexe Transformationen wie die Restrukturierung eines ConditionalDispatcher in eine kommandobasierte Lösung zu spezifizeren.

Allerdings hat sich gezeigt, dass es ermöglicht werden sollte, negative Vorbedingungen für iterierte Anteile zu spezifizieren. Prinzipiell kann dies durch Erzeugen einer weiteren Transformation, die aus einem iterierten Anteil heraus aufgerufen wird und die dann die Überprüfung der negativen Vorbedingung durchführt, realisiert werden. Dies hat jedoch zum einen Konsequenzen für die Verifikation von Kriterien und ist zum anderen sehr unhandlich.

Eine entsprechende Erweiterung der Transformationsspezifikation ist sehr einfach möglich und wurde im Rahmen der Evaluierung auf Ebene der Werkzeugunterstützung vorgenommen, indem negative Objekt- und Linkvariablen eingeführt wurden, die in Story Diagrammen bereits zur Verfügung stehen. Eine Erweiterung des Verifikationsverfahrens sollte ebenfalls möglich sein, da negative Vorbedingungen prinzipiell bereits unterstützt werden. Dies muss allerdings zunächst noch genauer untersucht werden, um unerwünschte Auswirkungen etwa auf die Korrektheit des Verfahrens auszuschließen.

Darüber hinaus wurde zur Spezifikation der MoveDeclaration-Transformation ein negativer Pfad in einer negativen Vorbedingung benötigt, um auszudrücken, dass es keinen Zugriff auf eine bestimmte Variable gibt, der nicht von einem bestimmten Anweisungsblock erreichbar ist – anders formuliert müssen alle Zugriffe auf die Variable von dem Anweisungsblock erreicht werden können.

Eine entsprechende Erweiterung der Werkzeugunterstützung war erneut sehr einfach mit Hilfe negativer Linkvariablen möglich. Das Verifikationsverfahren muss hierfür um die Unterstützung negativer Elemente in negativen Anwen-

dungsbedingungen von Graphmustern erweitert werden. Dabei ist insbesondere die in Abschnitt 4.4.5.1 beschriebene Überprüfung auf Widersprüche der negativen Anwendungsbedingungen von Graphmustern zur geplanten Regelsequenz zu erweitern. Die Erweiterung sollte prinzipiell möglich sein, muss aber ebenfalls zunächst genauer untersucht werden.

Eine dritte Erweiterung ist die explizite Unterstützung geordneter Assoziationen, so dass Objekte in einer bestimmten Reihenfolge verbunden werden können. Dies wird zum Beispiel beim Erzeugen von Anweisungen im Rahmen der BuildCommandMap-Transformation benötigt. Hier muss sichergestellt werden, dass zuerst ein Objekt einer Kommandoklasse instanziert und danach in die qualifizierte Assoziation aufgenommen wird.

Eine Erweiterung der Werkzeugunterstützung der Transformationsspezifikation war erneut sehr einfach mit Hilfe der in Story Diagrammen zur Verfügung stehenden *Multi Links* [Zün02] möglich. Eine Erweiterung des Verifikationsverfahrens wäre dann notwendig, wenn auch Aussagen über die Reihenfolge gezeigt werden sollen.

6.4 Verifikation

Zur Erprobung des Verifikationsverfahrens wurden die in Abbildung 6.3 zusamengefasst gezeigten Kriterien formuliert.

Die drei zu erhaltenden Graphmuster VariableAccess, VariableUpdate und MethodCall in der ersten Zeile der Abbildung setzen die bereits in den Abschnitten 3.1 und 4.1 beschriebenen Kriterien *access preservation*, *update preservation* und *call preservation* um. Sie besagen, dass wenn es ausgehend von einem ExpressionContext einen Lese- oder Schreibzugriff auf eine Variable oder einen Aufruf einer Methode gibt, dies auch nach einer Transformation so sein muss.

Das zu erhaltende Graphmuster Return verlangt, dass eine Methode, die eine Return-Anweisung enthält, dies auch noch nach einer Transformation tut, da sich ihr Verhalten ansonsten geändert haben kann.

Das verbotene Graphmuster FinalFieldUpdate daneben beschreibt eine Struktur, in der es einen Schreibzugriff auf ein Attribut gibt, das als konstant definiert wurde. Eine solche Struktur kann zum Beispiel eintreten, wenn ein Attribut in eine Konstante umgewandelt wird.

Die beiden verbotenen Graphmuster ForbiddenPrivateFieldAccess und ForbiddenPrivateMethodCall beschreiben den direkten Lesezugriff auf ein Attribut beziehungsweise einen Aufruf einer Methode jeweils mit privater Sichtbarkeit, aus einer anderen als der definierenden Klasse heraus. Solche Zugriffe oder

Aufrufe sind nicht gestattet und können prinzipiell beim Verschieben von Anweisungen in andere Klassen auftreten.

Das verbotene Graphmuster ForbiddenVariableAccess ist ebenfalls bereits aus Kapitel 4 bekannt und beschreibt einen verbotenen Zugriff auf Variablen, die in einer anderen Methode deklariert sind.

Das verbotene Graphmuster ForbiddenReturnExpression beschreibt eine Methode, die aufgrund ihres Rückgabetyps kein Ergebnis liefern dürfte, aber dennoch eine Return-Anweisung mit einem Ausdruck enthält. Dazu kann es ebenfalls beim Verschieben von Anweisungen in neue Methoden kommen.

Die in Abschnitt 6.3 beschriebenen Transformationsspezifikationen wurden mit Hilfe des Verifikationsverfahrens aus Kapitel 4 auf Einhaltung der zuvor beschriebenen Kriterien überprüft. Die in Tabelle 6.18 genannten Transformationsdiagramme wurden hintereinander überprüft, wobei die jeweils genannte Rechenzeit benötigt wurde und die genannte Anzahl an Gegenbeispielen berechnet wurde. Die Tabelle gibt weiterhin die Anzahl der Regeln (nicht-iterierte und iterierte Anteile) sowie der insgesamt darin enthaltenen Objekt-, Linkvariablen und Pfade an. Als Rechner wurde ein Dell Precision 390 Core Duo E6600 2.4 GHz mit 4 GB Arbeitsspeicher verwendet.

Durch Einsatz eines Profiling-Werkzeuges wurde ermittelt, dass ein großer Teil der Rechenzeit für die Berechnung von potentiellen Pfadausprägungen benötigt wird. Dies geschieht auf Basis des um Pfadnavigationsinformationen erweiterten Strukturmodells des abstrakten Syntaxgraphen, das aus 199 Klassen und 114 Assoziationen besteht, von denen 79 durch Pfade traversiert werden dürfen. Die Größe eines Diagramms ist daher nur bedingt ausschlaggebend für die zur Verifikation auf Einhaltung eines der Kriterien benötigte Rechenzeit. Diese hängt vielmehr von der Anzahl und Art der in einem Kriterium enthaltenen Pfade in Verbindung mit der Anzahl der Elemente der Regeln ab, die Bestandteil einer Ausprägung eines der Pfade sein können.

Das Transformationsdiagramm BuildCommandMap (10/95/90/0) konnte nicht auf Einhaltung aller Kriterien überprüft werden. Bei der Verifikation für einige Kriterien kam es zu einem Speicherüberlauf, obwohl der Arbeitsspeicher der *Java Virtual Machine* auf eine Größe von 2.8 GB konfiguriert war. Dies ist auf die große Menge von möglichen Pfadausprägungen zurückzuführen, die Objekte vom Typ StatementContext oder ExpressionContext enthalten können. Diese beiden Klassen des Strukturmodells verfügen über zahlreiche Assoziationen zu anderen Klassen, die durch Pfade traversiert werden können. Zum Beispiel verfügen die Klasse Block über die Assoziation statements und die Klasse If über die Assoziationen then und else zu StatementContext, welche die Verbindung zu enthaltenen Anweisungen repräsentieren. Ein StatementCon-

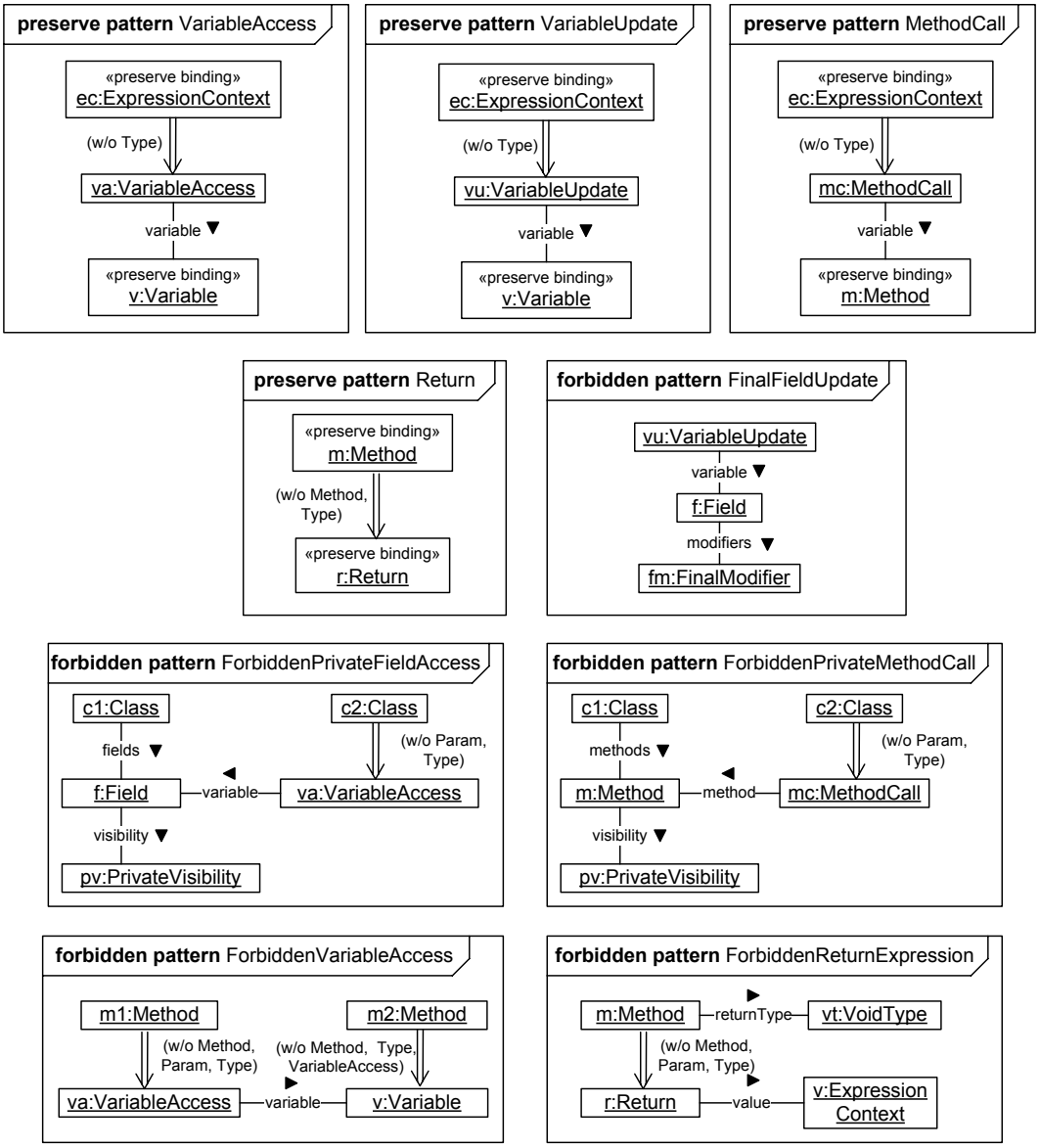

Abbildung 6.3: Verifikationskriterien

Transformationsdiagramm	R/O/L/P[a]	Zeit[b]	Gegenbsp.[c]
AbstractCommand	3/11/11/0	2,419	0
ConcreteCommand	2/13/13/0	0,825	0
EncapsulateAccessedFields	2/5/4/1	0,517	0
EncapsulateField	2/6/4/0	0,946	3
EncapsulateReadAccess	4/17/14/0	48,282	244
EncapsulateWriteAccess	4/19/17/0	253,981	134
ExtractDispatcher	5/13/3/0	0,696	0
ExtractedDispatcher	5/9/3/0	0,884	0
ExtractedDispatcherIf	4/24/24/2	24,855	52
ExtractedDispatcherIfCommand	4/31/29/0	76,122	55
ExtractVoidMethod	6/33/29/7	26,505	46
GetMethod	1/15/17/0	12,381	14
MoveDeclarationsIntoDispatcherIfs	2/9/6/3	0,319	0
MoveVariableDeclaration	2/8/5/1	1,486	86
OverriddenMethod	2/11/12/0	1,862	0
ReplaceDispatcherByCommands	6/16/11/0	0,291	0
ReplaceThisByParameter	4/15/15/2	5,669	4
SetMethod	1/20/23/0	137,077	14
SurroundByBlock	3/13/11/0	1,140	4

[a]Anzahl der Regeln/Objektvariablen/Linkvariablen/Pfade
[b]Rechenzeit in Sekunden
[c]Anzahl Gegenbeispiele

Tabelle 6.18: Transformationsdiagramme, auf die das Verifikationsverfahren erfolgreich angewendet wurde

text-Objekt kann damit aufgrund des Strukturmodells gleichzeitig mit einem Block sowie zwei If-Anweisungen verbunden sein. Dadurch kann es prinzipiell auch in mehr als einer Methode enthalten sein. Tatsächlich darf dies nicht der Fall sein, aber es kann nicht ohne Weiteres ausgeschlossen werden, da zum Beispiel Transformationsdiagramme solche Situationen prinzipiell herstellen könnten. Dies gilt analog für ExpressionContext-Objekte.

Ebenso basiert der überwiegende Teil der für die in Tabelle 6.18 genannten Diagramme berechneten Gegenbeispiele auf der eigentlich unzulässigen mehrfachen Verbindung von StatementContext- und ExpressionContext-Objekten mit weiteren Objekten. Es handelt sich dabei nicht um false-positives, vielmehr ist das Strukturmodell zu unpräzise definiert.

Dem kann zum Beispiel begegnet werden, indem mit Hilfe von Verifika-

tionskriterien strukturelle Invarianten formuliert werden, die eine mehrfache Verbindung von StatementContext- und ExpressionContext-Objekten mit weiteren Objekten verbieten und nach jedem angewendeten nicht-iterierten oder iterierten Anteil eines Transformationsdiagramms gelten müssen. Eine Verifikation der Anteile auf Einhaltung solcher Kriterien ist bereits mit dem Ansatz aus [Sch06] möglich. Die Berechnung von Pfadausprägungen müsste aber noch so erweitert werden, dass solche Invarianten korrekt berücksichtigt werden.

Eine weitere Situation, die mit Hilfe solcher Invarianten ausgeschlossen werden könnte, sind Variablendeklarationen ohne Typzuordnung. Durch die Ausführung von Transformationsdiagrammen können solche Situationen prinzipiell entstehen. Dies führt dazu, dass nicht garantiert werden kann, dass ExtractVoidMethod für alle von den extrahierten Anweisungen zugegriffenen Variablen Parameter anlegt, da diese dafür einen Typ haben müssen. Dies wird durch entsprechende Gegenbeispiele angezeigt.

Einige der Gegenbeispiele haben (weitere) Fehler in den Spezifikationen aufgedeckt. So wird zum Beispiel beim Generieren einer schreibenden Zugriffsmethode (SetMethod) derzeit nicht überprüft, ob es sich bei dem verkapselten Attribut um eine Konstante handelt. Daher wurde ein Gegenbeispiel berechnet, das eine Verletzung des FinalFieldUpdate-Kriteriums aufzeigt.

Weitere Gegenbeispiele werden (korrekt) berechnet, da aufgerufene Transformationsdiagramme nicht berücksichtigt werden. So wird bei EncapsulateField ein Gegenbeispiel für eine Verletzung des ForbiddenPrivateFieldAccess-Kriteriums berechnet, da die Sichtbarkeit des verkapselten Attributs auf privat gesetzt wird. Dass alle Zugriffe zuvor mit Hilfe der EncapsulateReadAccess- und EncapsulateWriteAccess-Transformationen durch Aufrufe von Zugriffsmethoden ersetzt wurden, kann bisher nicht berücksichtigt werden. Dies kann zum einen dadurch geschehen, dass im EncapsulateField-Transformationsdiagramm ausgeschlossen wird, dass es noch direkte Zugriffe auf das Attribut außerhalb der Zugriffsmethoden gibt. Eine andere Möglichkeit, die eine Erweiterung des Ansatzes erfordern würde, wäre die Einführung von Nachbedingungen für Transformationsdiagramme, die direkt nach ihrem Aufruf gelten. Abhilfe kann hier ebenfalls die Berücksichtigung der durch Strukturmusterannotationen repräsentierten Strukturen schaffen, anstatt Annotationen wie herkömmliche Objekte zu behandeln.

6.4.1 Fazit

Das Verifikationsverfahren konnte erfolgreich auf einige praxisrelevante Spezifikationen angewendet werden. Allerdings müssen seine Präzision sowie die

Effizienz seiner Implementierung noch deutlich verbessert werden, wie im vorangehenden Abschnitt bereits ausgeführt wurde.

Kapitel 7

Verwandte Arbeiten

Im Folgenden wird der Stand der Forschung in den Bereichen der Erkennung von Schwachstellen und deren Verbesserung durch die Restrukturierung von Software sowie der Verifikation von Graphtransformationen dargestellt und zur vorliegenden Arbeit in Bezug gesetzt.

Es wird mit Ansätzen begonnen, die sich allein mit der Erkennung von Schwachstellen befassen, gefolgt von Ansätzen, die sich der Restrukturierung von Software im Allgemeinen sowie der Verbesserung von Schwachstellen im Besonderen widmen. Darunter fallen auch Ansätze, die wie die vorliegende Arbeit eine Erkennung und Verbesserung kombinieren.

Ein wesentlicher Beitrag der vorliegenden Arbeit ist die Entwicklung eines Verifikationsverfahrens für die in Kapitel 3 entworfene, auf Graphtransformationen basierende Spezifikationssprache für Programmtransformationen. Daher werden abschließend verwandte Arbeiten aus dem Bereich der Verifikation von Graphtransformationssystemen vorgestellt.

7.1 Erkennung von Schwachstellen

Schwachstellen, seien es Entwurfsschwächen oder Implementierungsmängel, erschweren in der Regel die Wartung und insbesondere die Erweiterung von Software. Ihre Vermeidung bereits bei der Konstruktion von Software kann spätere Wartungskosten reduzieren. Ebenso wichtig ist ihre Erkennung in bereits bestehender Software, um Wartungsaktivitäten besser einschätzen oder erleichtern zu können. Da die manuelle Suche nach Schwachstellen zeitaufwändig und nicht praktikabel ist, widmen sich zahlreiche Arbeiten der automatisierten oder halbautomatisierten Erkennung von Schwachstellen in bestehenden Softwaresystemen.

Bär und Ciupke [BC98, Ciu99] teilen eine Reihe von Entwurfsheuristiken

gemäß der Überprüfbarkeit ihrer Einhaltung in Kategorien von exakt überprüfbar bis zu nicht überprüfbar ein. Zur automatischen Erkennung von verletzten Heuristiken werden (sofern möglich) Prolog-Regeln definiert und auf eine entsprechende Quelltextrepräsentation angewendet.

Van Emden und Moonen [EM02] verwenden das ASF + SDF Meta-Environment [BDH+01] um eine Grammatik für Java-Programme zu definieren auf deren Basis mit Hilfe einer Termersetzungssprache Bad Smells spezifiziert werden. Aus den Spezifikationen wird das Werkzeug jCosmo größtenteils generiert, das eine automatische Analyse von Java-Programmen ermöglicht. Die Ergebnisse der Bad-Smell-Erkennung werden mit Hilfe von Rigi [MTW93, Rig] visualisiert.

Kothari et al. [KBSD04] entwickeln einen allgemeinen Ansatz zur Analyse von Software auf das Vorhandensein von Schwachstellen. Sie entwickeln eine allgemeine Repräsentation für Quelltext in Form von XML sowie eine Sprache zur Spezifikation von Schwachstellen, die komplexe Anfragen an die XML-Repräsentation stellen und auf den Ergebnissen weitere strukturelle Eigenschaften überprüfen kann. Die Spezifikationen können automatisch zur Erkennung ausgeführt werden. Auch die Unterstützung von Transformationen ist offenbar vorgesehen. Wie weit die Unterstützung geht, ist jedoch nicht klar ersichtlich.

Zahlreiche Ansätze verwenden Softwareproduktmetriken [FP96, LK94] um verschiedene Eigenschaften einer Software zu messen und daraus Rückschlüsse auf das Vorhandensein von Schwachstellen zu ziehen. Einige Ansätze aus dem Bereich der Softwarevisualisierung stellen die Ergebnisse von Programmanalysen zum Beispiel auf Basis von Metriken grafisch dar und geben einem Re-Engineer auf diese Weise einen Überblick über das Gesamtsystem oder versuchen bestimmte Schwachstellen aufzudecken. Simon, Steinbrückner und Lewerentz [SSL01] zum Beispiel verwenden eine Distanzmetrik, um unter Anderem Methoden und Attribute zu identifizieren, die in einer anderen Klasse besser platziert wären. Die berechneten Distanzen werden dreidimensional visualisiert, so dass ein Re-Engineer sie sehen und entscheiden kann, ob bestimmte Methoden oder Attribute durch Refactoring verschoben werden sollen.

Ein weiterer Visualisierungsansatz stammt von Lanza [Lan03]. Lanza schlägt ebenfalls die Analyse eines Systems auf Basis einer Reihe von Metriken vor, deren Ergebnisse er in speziellen so genannten *polymetrischen Sichten* darstellt. Polymetrische Sichten sind zweidimensionale Graphen bestehend aus Knoten und Kanten. Jeder Knoten repräsentiert ein Element des analysierten Systems, wie zum Beispiel eine Klasse, und kann bis zu 5 verschiedene Metrikwerte darstellen: jeweils eine Metrik bestimmt die Breite, die Höhe, die horizonta-

le Position, die vertikale Position und die Farbschattierung. Lanza definiert eine Reihe von Sichten, die einen Überblick über verschiedene Aspekte eines Systems geben. Der Re-Engineer kann in den Sichten visuell Ausreißer identifizieren, die auf Schwachsellen hindeuten und einer genaueren Analyse bedürfen. Der Ansatz wurde in dem Analysewerkzeug CodeCrawler [LDGP05] implementiert.

Ansätze aus dem Bereich der Softwarevisualisierung bereiten im Allgemeinen Analysergebnisse grafisch auf und überlassen die Interpretation und damit die Identifikation von Schwachstellen mehr oder weniger vollständig dem Re-Engineer. Die vorliegende Arbeit identifiziert Schwachstellen dagegen automatisiert und so konkret, dass anschließend eine (in Teilen) automatisierte Restrukturierung erfolgen kann.

Munro [Mun05] verwendet ebenfalls Metriken jedoch nicht nur zur Visualisierung sondern zur konkreten Identifikation von Bad Smells und demonstriert dies anhand zweier Bad Smells aus [Fow99]. Für die Bad Smells werden Metriken ausgewählt und deren Werte werden miteinander beziehungsweise mit absoluten Schwellwerten verglichen, um eine Entscheidung über das Vorhandensein einer Instanz des Bad Smells zu treffen. Strukturelle Eigenschaften eines Programms werden nicht berücksichtigt, so dass viele Schwachstellen gar nicht oder nur sehr unpräzise formuliert werden könnten. Es wird keine Sprache für die Erkennung beliebiger Bad Smells auf Basis von Metriken entworfen, sondern lediglich die Erkennung der beiden ausgewählten Bad Smells in einem Prototyp auf Basis der Entwicklungsumgebung Eclipse konkret implementiert und durch Analyse zweier kleiner Systeme erprobt.

Trifu, Marinescu und Reupke [TR07, TM05, Mar04] beschreiben einen Ansatz, der die Erkennung von Entwurfsproblemen mit deren Verbesserung durch Refactoring verbinden soll. Entwurfsprobleme werden analog zu Krankheiten in der Medizin als tieferliegende Probleme aufgefasst, für die zum Beispiel Bad Smells Indikatoren im Sinne von Symptomen sein können. Entwurfsprobleme werden zunächst informal beschrieben durch einen *Context*, der die Entwurfsintention wiedergibt, eine *Pathologic Structure*, die eine problematische Umsetzung der Intention beschreibt, eine *Rationale for Refactoring*, die angibt warum die problematische Umsetzung wie verbessert werden sollte und schließlich *Indicators*, die auf eine Instanz des Problems in einer Software hindeuten. Auf diese Weise sollen Entwurfsprobleme eindeutig mit Refactorings zu deren Verbesserung verbunden werden.

Die Erkennung von Problemen auf Basis der Indikatoren wird durch so genannte *Diagnosestrategien* operationalisiert. Dabei werden automatisiert auswertbare Ausdrücke über Softwareproduktmetriken mit Aussagen über struk-

turelle Eigenschaften kombiniert. Zusätzlich werden Fragen formuliert, die dem Re-Engineer gestellt werden müssen, um für eine konkrete Fundstelle zu ermitteln, ob auf sie auch der Kontext des Entwurfsproblems zutrifft. Die Metrikausdrücke sind auf Basis der *Detection Strageies* aus [Mar04] formal beschreibbar. Inwieweit strukturelle Eigenschaften formal beschrieben werden können oder ob deren Überprüfung lediglich fest in einem Werkzeug implementiert wird, ist nicht erkennbar. Auch wird auf die Formalisierung von Refactorings bisher nicht eingegangen. Für den Teil der Erkennung gibt es eine prototypische Umsetzung als Erweiterung der Eclipse-Entwicklungsumgebung. Eclipse bietet eine Programmierschnittstelle für die Implementierung von Refactorings an, die für eine zukünftige Umsetzung von Refactorings genutzt werden soll.

Der Ansatz verfolgt im Prinzip dieselbe Zielsetzung wie die vorliegende Arbeit. Auch ist die Herangehensweise zur Erkennung von Entwurfsproblemen auf Basis von Metrikwerten in Verbindung mit strukturellen Eigenschaften ähnlich. Im Unterschied erweitert diese Arbeit eine formale, grafische Spezifikationssprache für Struktureigenschaften um eine Verbindung mit Aussagen über Metrikwerte sowie insbesondere die Verwendung von Metrikwerten zur Bewertung von Fundstellen. Der Re-Engineer wird so in die Lage versetzt, beliebige eigene Spezifikation zu erstellen, die automatisch von einem Werkzeug erkannt werden können. Eine manuelle Inspektion von Fundstellen durch den Re-Engineer ist auch in dieser Arbeit vorgesehen, allerdings wird sie nicht durch vordefinierte Fragen geleitet, so dass die Entscheidung, welches Refactoring anzuwenden ist, mehr dem Re-Engineer überlassen ist. Eine Verbindung der beiden Arbeiten könnte hier sinnvoll sein. Zudem geht die vorliegende Arbeit einen Schritt weiter, indem sie eine formale, grafische Sprache zur Spezifikation ausführbarer Refactorings bereitstellt, die Spezifikationen zur Erkennung von Entwurfsproblemen aufgreifen und wiederverwenden kann und deren Worte auf Einhaltung definierbarer Kriterien verifiziert werden können.

Kreimer [Kre05] greift die metrikbasierten Detection Strategies von Marinescu [Mar04] auf und kombiniert sie mit einem maschinellen Lernverfahren. Für einen Bad Smell wird eine Menge von Metriken ausgewählt, die Hinweise auf sein Vorhandensein in einem Programm liefern können. Aus diesen Metriken und einer Trainingsmenge von Programmstellen, für die die Metrikwerte berechnet wurden und bekannt ist, ob es sich tatsächlich um eine Instanz des Bad Smells handelt oder nicht, wird ein Entscheidungsbaum gelernt, der in der Lage ist, unbekannte Programmstellen zu klassifizieren. Auf diese Weise werden im Prinzip Schwellwerte für die Metriken sowie bestimmende Merkmale auf einer statistischen Basis ermittelt. Der Ansatz basiert ausschließlich auf Metriken und berücksichtigt darüber hinaus keine weiteren strukturellen

Eigenschaften.

Moha et al. [MGMD08, MGL06] entwickeln eine textuelle domänenspezifische Sprache für die Spezifikation von Schwachstellen. Sie stellen zunächst eine Taxonomie aus Anti Pattern, Bad Smells, messbaren und strukturellen Eigenschaften auf. Die Elemente der Taxonomie treten in der Spezifikationssprache als Literale auf und werden zur Spezifikation von Schwachstellen formal zueinander in Beziehung gesetzt. Aus den Spezifikationen werden automatisiert Erkennungsalgorithmen generiert, die ein Framework nutzen, in dem zum Beispiel die Berechnung von Metriken oder die Überprüfung von in der Taxonomie definierten strukturellen Eigenschaften implementiert ist.

Der Ansatz kombiniert ebenso wie die vorliegende Arbeit Metriken mit strukturellen Eigenschaften und bietet eine formale Sprache zur Spezifikation von Schwachstellen an, die automatisch in Erkennungsalgorithmen überführt werden. Allerdings ist die Sprache der vorliegenden Arbeit nicht domänenspezifisch und damit allgemein einsetzbar. Des Weiteren muss die Überprüfung spezieller struktureller Eigenschaften nicht manuell implementiert werden, sondern kann ebenfalls mit Hilfe der Sprache spezifiziert werden. Die Berechnung von Metriken wird in der vorliegenden Arbeit derzeit analog auf Basis des Metamodells manuell implementiert. Dies könnte allerdings auch auf Basis von Story Driven Modeling ermöglicht werden.

Alle bisher vorgestellten Ansätze haben gemeinsam, dass sie scharfe Ja/Nein-Entscheidungen über das Vorhandensein von Schwachstellen liefern. Die vorliegende Arbeit verwendet stattdessen Metrikwerte in Kombination mit strukturellen Eigenschaften zur Erkennung und Bewertung konkreter Schwachstellen. Auf diese Weise bleiben keine Schwachstellen aufgrund falsch gewählter oder gelernter Schwellwerte unerkannt. Ein Re-Engineer kann Schwachstellen auf Basis ihrer Bewertung sortieren und selbst entscheiden, ab welcher Bewertung er sie für nicht relevant hält. Des Weiteren unterstützt bisher keiner der Ansätze die Verbesserung erkannter Schwachstellen durch Spezifikation ausführbarer Refactorings.

Gueheneuc und Antoniol [GA08] stellen einen Ansatz vor, der sich explizit der Erkennung von Entwurfsmusterinstanzen widmet, aber sicher auch zur Erkennung von Schwachstellen eingesetzt werden kann. Zu erkennende Muster werden mit Hilfe einer Notation ähnlich zu UML-Klassendiagrammen spezifiziert, die automatisiert in Constraints übersetzt werden. Ein zu analysierendes System wird so repräsentiert, dass ein Constraint-Solver eingesetzt werden kann, um Musterinstanzen zu erkennen. Der Constraint-Solver kann interaktiv oder automatisch Constraints lockern, so dass auch ähnliche Instanzen erkannt werden können. Die Constraints können gewichtet werden, so dass sich auf

diese Weise eine Bewertung von Funden ergibt. Der Einsatz von Metriken in Constraints wird nicht erwähnt, scheint aber denkbar. Allerdings wäre damit nicht ohne Weiteres eine kontinuierliche Bewertung möglich.

7.2 Restrukturierung von Software

Chikofsky und Cross definieren *Restrukturierung* in [CC90] als:

> „*Restructuring is the transformation from one representation form to another at the same relative abstraction level, while preserving the subject's external behavior (functionality and semantics).*“

Opdyke bezeichnet später die Restrukturierung objektorientierter Systeme als *Refactoring* [Opd92] und definiert 23 typische, feingranulare Restrukturierungsschritte, genannt *Refactorings*, wie zum Beispiel das Hinzufügen, Löschen oder Umbenennen von Klassen, Attributen oder Methoden, das Extrahieren von Anweisungen in Methoden oder das Verschieben von Attributen. Für die einzelnen Refactorings werden Vorbedingungen mit Hilfe von Prädikatenlogik formal definiert. Die Vorbedingungen sollen sicherstellen, dass ein Refactoring das von außen beobachtbare Verhalten des transformierten Programms nicht verändert. Das von außen beobachtbare Verhalten wird über den Zusammenhang zwischen Eingaben und Ausgaben eines Programms definiert: ein Programm soll nach der Anwendung eines Refactorings für dieselben Eingaben dieselben Ausgaben liefern wie vor der Anwendung. Zusätzlich werden einige syntaktische Kriterien wie Typkonformität oder Ausschluss von Namenskollisionen definiert, die ebenfalls eingehalten werden müssen, um eine Verhaltensänderung auszuschließen. Weitere Aspekte, wie zum Beispiel die Größe von Datenstrukturen im Speicher oder Laufzeiten von Methoden werden nicht berücksichtigt. Bei Programmen, die Annahmen darüber treffen, kann sich das Verhalten demnach ändern. Dass die Vorbedingungen Verhaltenserhaltung sicherstellen, wird informal argumentiert. Die Durchführung der Refactorings wird ebenfalls nur informal beschrieben. Schließlich beschreibt Opdyke informal umfangreichere, zusammengesetzte Refactorings zum Beispiel zum Extrahieren von abstrakten Basisklassen oder zur Vereinfachung von bedingten Anweisungen.

Roberts [Rob99] erweitert die Refactorings aus [Opd92] um ebenfalls mit Hilfe von Prädikatenlogik formal definierte Nachbedingungen. Auf dieser Basis können einzelne Refactorings zu umfangreicheren Restrukturierungen kombiniert werden, wobei auf Basis der Nachbedingungen ermittelt werden kann,

dass vorangehende Refactorings die Vorbedingungen nachfolgender Refactorings erfüllen können. Die Transformationen selbst werden nicht abstrakt formalisiert sondern in einem Werkzeug, dem *Refactoring Browser* [RBJ97] implementiert, der die Transformation von Smalltalk-Programmen ermöglicht. Dabei wählt der Benutzer aus, wo welche Refactorings angewendet werden sollen. Roberts untersucht darüber hinaus, inwieweit dynamische Analysen der Programmausführung zur präziseren Überprüfung von Vorbedingungen eingesetzt werden können. Kniesel und Koch [KK04] erweitern den Ansatz von Roberts so, dass auf Basis formal definierter Vor- und Nachbedingungen einzelner Refactorings automatisch Vor- und Nachbedingungen für zusammengesetzte Refactorings berechnet werden können. Kataoka et al. [KEGN01] setzen dynamische Programmanalyse ein, um operationale Invarianten zu ermitteln, die Möglichkeiten für die Anwendung von Refactorings anzeigen, wie zum Beispiel funktionale Abhängigkeiten von Parameterwerten, so dass Parameter aus anderen Variablen berechnet und entfernt werden können.

Fowler et al. gehen einen anderen Weg, indem sie in [Fow99] 72 umfangreiche Refactorings vollkommen informal beschreiben, vergleichbar mit der Beschreibung von Entwurfsmustern. Es werden das allgemeine Vorgehen sowie zu berücksichtigende Besonderheiten angegeben, die ein Entwickler manuell umsetzen muss. Um sicherzustellen, dass die Refactorings keine Verhaltensänderungen vornehmen, werden eine Zerlegung in kleine überschaubare Schritte und die regelmäßige Überprüfung der Software durch eine umfangreiche Testsuite empfohlen.

Kerievsky greift diese Art der Darstellung auf und beschreibt in [Ker04] informal eine Reihe weiterer umfangreicher Refactorings, die bestehende Teile einer Software in Entwufsmusterinstanzen umstrukturieren. Eine Reihe von Ansätzen befasst sich mit der (Teil-)Automatisierung solcher Restrukturierungen zur Einführung von Entwurfsmusterinstanzen. Diese Ansätze werden im noch folgenden Abschnitt 7.2.1 separat vorgestellt.

Über die bisher aufgeführten, eher allgemeinen und grundlegenden Arbeiten zum Refactoring hinaus, gibt es eine Reihe von Ansätzen, die sich sehr speziellen Restrukturierungen widmen. Dies betrifft zum einen die Restrukturierung von Klassenhierarchien. Casai [Cas94] analysiert in bestehenden Klassenhierarchien, inwieweit Unterklassen Eigenschaften ihrer Oberklassen verwenden, überschreiben oder inwieweit sie Erweiterungen vornehmen und ermittelt auf Basis der Ergebnisse mit Hilfe unterschiedlicher Strategien restrukturierte Hierachien. Der Ansatz wird zur Restrukturierung einer Version der *Eiffel*-Klassenbibliothek eingesetzt und produziert überwiegend sinnvolle Ergebnisse. Moore [Moo96] beschreibt einen Ansatz, der Klassenhierarchien unter

Einbeziehung von Methoden in der Programmiersprache *Self* automatisch so restrukturiert, dass möglichst wenig Redundanz besteht. Snelting, Tip und Streckenbach [ST00, SS04] analysieren die Verwendung von Klassenhierarchien durch Client-Programme mit Hilfe von *Concept Analysis* und ermitteln auf der Basis automatisch Restrukturierungen wie das Splitten von Klassen oder das Verschieben von Attributen. Moha et al. [MHGV08] verwenden *Relational Concept Analyis* um konkrete Vorschläge für das Refactoring von Instanzen des Blob-Anti-Patterns [BMMM98] zu ermitteln. Maruyama und Schima [MS99] entwickeln einen Ansatz, der ausgehend von der Änderungshistorie einer Methodenimplementierung, diese in *Template*- und *Hook*-Methoden zerlegt: wenig geänderte, stabile Folgen von Anweisungen werden als allgemeingültige Verfahrensanteile identifiziert, die in Template-Methoden implementiert werden, während sich häufig ändernde Anweisungsfolgen variable Anteile eines Verfahrens darstellen, die in Hook-Methoden zu implementieren sind.

Weitere spezielle Ansätze befassen sich mit der automatischen Restrukturierung von Java-Programmen zur Verwendung der in Version 1.5 neu eingeführten *Generics*. Seit Java 1.5 sind zum Beispiel die im *Java Development Kit* enthaltenen *Collection*-Klassen generisch implementiert. Bestehende Programme verwenden Instanzen dieser Klassen noch ohne die neuen Typparameter zu binden. Fuhrer et al. [FTK+05] stellen einen Ansatz vor, der automatisch konkrete Argumente für die Typparameter generischer Klassen ermittelt und Programme entsprechend restrukturiert. Die Arbeiten von Kiezun et al. [KETF07] oder von von Dincklage und Diwan [DD04] gehen darüber hinaus, indem sie auch Vorschläge für die Einführung von Typparametern ermitteln.

Die Restrukturierung und insbesondere das Refactoring objektorientierter Software wird heute bereits von zahlreichen Entwicklungsumgebungen wie zum Beispiel Eclipse [Ecla], IntelliJ IDEA [Int] oder Visual Studio [Mic] unterstützt. Mit JRefactory [SA] gibt es zum Beispiel ein weiteres Werkzeug, das standalone oder als Plugin für die Entwicklungsumgebungen JBuilder [Emb] oder NetBeans [Sun] für das Refactoring von Java-Programmen eingesetzt werden kann. Die Werkzeuge haben gemeinsam, dass sie einen festgelegten Satz von Refactorings anbieten. Die Refactorings sind so programmiert, dass sie eine Reihe von Vorbedingungen prüfen, einige Benutzereingaben abfragen und dann die Transformation vornehmen. Durch das Prüfen der Vorbedingungen wird die Erhaltung des von außen beobachbaren Verhaltens des transformierten Programms sichergestellt, zumindest sofern diese korrekt implementiert sind, was nicht immer der Fall ist, wie in [VEM06, EV] gezeigt wird. Anpassungen eines Refactorings oder eine Erweiterung um zusätzliche Refactorings sind typischerweise nicht ohne Weiteres möglich. Eclipse bietet

zum Beispiel ein API an, über das eigene Refactorings in Java implementiert und in ECLIPSE verfügbar gemacht werden können. Eine abstraktere Sprache, wie sie in der vorliegenden Arbeit entwickelt wird, gibt es nicht. Auch werden eher einfache Refactorings unterstützt. Umfangreichere Refactorings wie zum Beispiel *Replace Conditional Dispatcher With Command* [Ker04] gehören nicht dazu und müssen manuell durch geeignete Kombination kleinerer Refactorings durchgeführt werden.

7.2.1 Restrukturierung auf Basis von Entwurfsmustern

Software wird häufig restrukturiert, um bevorstehende Erweiterungen zu ermöglichen oder zu vereinfachen. Entwurfsmuster haben sich insbesondere als leicht erweiterbare Strukturen etabliert. Daher befassen sich eine Reihe von Arbeiten mit der Instanzierung von Entwurfsmustern in bestehenden Systemen, um deren Erweiterung zu erleichtern.

Florijn, Meijers und van Winsen beschreiben in [FMv97] eine Ansammlung von Werkzeugen, die umfangreiche Unterstützung für die Erkennung, Instanzierung beziehungsweise Integration sowie die Erhaltung von Entwurfsmustern für bestehende Smalltalk-Programme anbieten. Die Werkzeuge arbeiten auf einer einheitlichen Repräsentation von Smalltalk-Programmen und bieten verschiedene Sichten: eine Mustersicht, eine Entwurfssicht (Klassendiagramm) sowie eine Quelltextsicht, in denen unterschiedliche zur jeweiligen Sicht passende Bearbeitungsmöglichkeiten angeboten werden. In der Mustersicht können zum Beispiel Entwurfsmuster instanziert und in das Programm integriert werden, indem bestimmte Elemente des Musters auf bereits bestehende Elemente des Programms abgebildet werden, während fehlende Elemente neu erstellt werden. In der Entwurfssicht werden Refactorings aus [Opd92] angeboten. In der Quelltextsicht kann programmiert werden. Im Programm bestehende Entwurfsmusterinstanzen werden bei Änderungen am Programm überprüft. Wird dabei festgestellt, dass bestimmte Eigenschaften nicht mehr gelten, werden diese zum Teil automatisch mit Hilfe von Refactorings wiederhergestellt. Verhaltenserhaltung wird nicht thematisiert, da lediglich verhaltenserhaltende Refactorings aus [Opd92] angewendet werden.

Schulz et al. präsentieren in [SGMZ98] die Idee, *Entwurfsmusteroperatoren* mit Hilfe von Refactorings aus [Opd92] zu definieren, die eine Instanzierung und Integration von Entwurfsmustern in bestehenden Systemen ermöglichen sollen. Die Systeme werden dabei zum Teil restrukturiert. Da dazu nur Refactorings aus [Opd92] angewendet werden, sind die Restrukturierungen insgesamt garantiert verhaltenserhaltend. Wo welche Entwurfsmuster instanziert

oder integriert werden, wird durch den Re-Engineer bestimmt. Zur Evaluierung wird ein Werkzeugprototyp implementiert, der einige Refactorings für C++-Programme untersützt. Einige Entwurfsmusteroperatoren werden mit Hilfe von Shellscripten unter Verwendung des Prototyps implementiert.

Tokuda und Batory [TB01] beschreiben ebenfalls wie Entwurfsmuster mit Hilfe von Refactorings aus [Opd92] sowie einigen Ergänzungen instanziert werden können, wobei sie allerdings von einer reinen Instanzierung „auf der grünen Wiese“ ausgehen. Die Refactorings wurden für C++-Programme implementiert. Darüber hinaus analysieren sie die Weiterentwicklung von zwei verschiedenen Programmen dahingehend, ob die dabei vorgenommenen Restrukturierungen automatisiert mit Refactorings hätten durchgeführt werden können. Sie kommen zu dem Ergebnis, dass ein signifikanter Anteil der Weiterentwicklung der untersuchten Systeme mit Hilfe von Refactorings hätte automatisiert werden können, wodurch sich eine deutliche Zeitersparnis ergeben hätte.

O'Cinnéide und Nixon [ON99, ON01, O'C01] haben einen Ansatz entwickelt, mit dem bestehende Software so transformiert wird, dass Entwurfsmuster instanziert werden. Dies geschieht mit Hilfe von automatisiert ausführbaren Transformationen. Für jede Transformation wird eine Anwendungsbedingung definiert. Zum Beispiel gibt es eine Transformation, die direkte Instanzierungen einer konkreten Produktklasse innerhalb einer anderen Klasse gemäß dem Muster *Factory Method* [GHJV95] umstrukturiert. Es werden 6 Minitransformationen, die wiederum von primitiven Refactoring-Operationen aus [Opd92] und weiteren Hilfsfunktionen Gebrauch machen, definiert. Entwufsmustertransformationen werden als Algorithmen formuliert, die Minitransformationen oder Hilfsfunktionen hintereinander sowie iteriert ausführen. Für sämtliche Transformationen werden Vor- und Nachbedingungen mit Hilfe von Prädikatenlogik formal definiert. Die Vorbedingungen sollen sicherstellen, dass eine Transformationen das Verhalten eines Programms nicht verändert. Die Nachbedingungen werden benutzt, um für hintereinander oder iteriert ausgeführte Transformationen manuell zu beweisen, dass die Vorbedingungen aller Transformationen erfüllt sind. Das zu erhaltende Verhalten sowie die Ausführung der Transformationen selbst werden nicht formal definiert. Die Auswahl, welche Transformation wo durchgeführt werden soll, wird dem Re-Engineer überlassen. Die Anwendungsbedingungen der Transformationen beschreiben keine schlechten Lösungen im Sinne von Schwachstellen, sondern Stellen im Programm, an denen ein bestimmtes Entwurfsmuster instanziert werden kann. Dass es dort auch instanziert werden sollte, kann den Anwendungsbedingungen nicht entnommen werden. Ungefähr die Hälfte der in [GHJV95] beschriebenen Entwurfsmuster können durch Transformationen vollständig umgesetzt werden und für

ein weiteres Viertel sind zumindest teilweise, noch unvollständige Umsetzungen möglich.

Tahvildari und Kontogiannis [Tah03, Tah04, TK02b, TK02a] haben einen Ansatz entwickelt, bei dem nicht-funktionale Qualitätsanforderungen an eine Software, wie zum Beispiel hohe Wartbarkeit, hohe Qualität der Kontrollstruktur, hohe Kohäsion, aufgestellt werden, deren Erfüllungsgrad im System durch Re-Engineering verbessert werden soll. Inwieweit die Anforderungen erfüllt sind, wird mit Hilfe von Metriken ermittelt. Es werden 6 primitive Entwurfsmustertransformationen definiert (nahezu identisch zu [O'C01]), die kombiniert werden können, um Entwurfsmuster wie zum Beispiel *Factory Method*, *Composite* oder *Iterator* [GHJV95] in ein System einzuführen beziehungsweise vorzubereiten, da sie nicht unbedingt vollständig umgesetzt werden. Die Zusammenhänge zwischen den Qualitätsanforderungen und welchen Einfluss von (sehr) positiv bis (sehr) negativ die primitiven Transformationen auf die nicht-funktionalen Anforderungen haben, werden in einem Graphen modelliert. Ein Algorithmus bestimmt auf Basis der Metrikergebnisse und des Graphen welche Transformationen durchgeführt werden, um die Erfüllung einer Anforderung zu verbessern. Die Transformationen sind in einer Programmiersprache implementiert. Vor- und Nachbedingungen einer Transformation werden mit Hilfe von Prädikatenlogik formal definiert. Es wird argumentiert, dass wenn die Vorbedingung einer Transformation erfüllt ist, das von außen beobachtbare Verhalten des Programms bei ihrer Durchführung nicht verändert wird. Die eigentliche Durchführung der Transformation wird nicht betrachtet.

Albin-Amiot und Guéhéneuc definieren in [GAA01, AAG01] Entwurfsmängel beziehungsweise Schwachstellen als Muster im Quelltext eines Programms, die ähnlich zu Entwurfsmustern sind, diesen aber nicht ganz entsprechen (deformiert sind). Entwurfsmuster werden auf Basis eines Metamodells spezifiziert. Aus den Spezifikationen werden manuell *Constraints* für die Elemente eines Musters und deren Zusammenhänge abgeleitet, die zusammen mit entsprechenden Quelltextelementen an einen *Constraint Solver* übergeben werden, der Constraints automatisch lockert und auch zurückliefert, welche Elemente welche Constraints nicht erfüllt haben. Dadurch wird eine Menge von Gruppen von Elementen gefunden, die einem Entwurfsmuster ähnlich sind. Aus den Abweichungen werden automatisch Quelltexttransformationen bestimmt, deren Durchführung die Abweichungen beseitigen. Die Transformationen werden von JavaXL [AA01] bereitgestellt. Es werden keinerlei Aussagen zur Erhaltung oder Veränderung des Programmverhaltens gemacht.

Zhao et al. [ZKDZ07] formalisieren Entwurfsmuster mit Hilfe von Graphtransformationsregeln, die sowohl zur Erkennung von Entwurfsmusterinstan-

zen im Rahmen des Reverse Engineerings als auch zur Validierung von vom Benutzer identifizierten Instanzen eingesetzt werden können. Für jedes Entwurfsmuster wird eine abgeschlossene Menge an spezifischen Transformationen als Graphtransformationsregeln definiert, die eine Instanz des Musters erweitern, zum Beispiel um eine neue Klasse für eine konkrete Strategie im Falle des *Strategy*-Entwurfsmusters [GHJV95]. Es wird argumentiert, dass Entwurfsmusterinstanzen auch bei Erweiterung garantiert valide bleiben, da die Transformationen für ihre Erweiterung formal definiert sind und damit automatisiert vorgenommen werden können. Ein Beweis oder Beweisverfahren, dass die Transformationen dies tatsächlich garantieren, wird nicht erbracht. Die vorliegende Arbeit ermöglicht ebenfalls die Spezifikation musterspezifischer Erweiterungstransformationen und kann diese zusätzlich auf Einhaltung definierbarer Kriterien verifizieren.

7.2.2 Vollständige Formalisierung von Refactorings

In den bisher vorgestellten Arbeiten werden Refactorings oder allgemeiner Programmtransformationen nur durch formale Vor- und Nachbedingungen beschrieben. Ihre eigentliche Durchführung, also wie ausgehend von der Vorbedingung die Nachbedingung hergestellt wird, wird nicht abstrakt spezifiziert, sondern in diversen Programmiersprachen implementiert. Die folgenden Arbeiten befassen sich mit der vollständigen, mehr oder wengier abstrakten, Formalisierung von Transformationen.

Mens et al. schlagen in [MDJ02, MEJD03] die Formalisierung von Refactorings mit Hilfe von parametrisierten Graphtransformationensregeln mit negativen Anwendungsbedingungen vor. Dazu wird der Quelltext durch einen typisierten Graphen repräsentiert. Als Beispiele werden einige Refactorings aus [Opd92, Fow99], insbesondere *Extract Method* und *Pull Up Method*, in Teilen formalisiert. Dabei wird festgestellt, dass einzelne Regeln nicht ausdrucksstark genug sind, um vollständige Refactoring-Operationen zu spezifizieren. Vielmehr werden mehrere Regeln benötigt, deren Anwendung kontrolliert wird. Die Regeln erlauben ebenfalls die Spezifikation von Vorbedingungen. Zusätzlich werden Wohlgeformtheits- und Verhaltenseigenschaften mit Hilfe von Graphausdrücken (Graphen mit Pfadausdrücken) formalisiert, die mit den zu erhaltenden und verbotenen Graphmustern der vorliegenden Arbeit vergleichbar sind. Einige der Eigenschaften werden in Abschnitt 3.1 genannt und in Kapitel 4 formalisiert und als Beispiel verwendet.

Der Ansatz wird von Van Eetvelde et al. in [EJ04, EJ05, HJE06] und [DHJ+06, DHJ+07] im Hinblick auf die Ausdrucksmächtigkeit der Graphtrans-

formationsregeln erweitert. Das Ziel ist, möglichst ausdrucksstarke Regeln formulieren zu können, um möglichst große Anteile eines Refactorings deklarativ beschreiben zu können und wenig durch Kontrollfluss ausdrücken zu müssen. Dazu werden zum einen Mengenknoten eingeführt, die im Prinzip bei Anwendung einer Regel alle konformen Knoten im Wirtsgraphen binden. Mehrere Mengenknoten können mit derselben Kardinalitätsvariable versehen werden. Auf diese Weise können iterierte Subgraphen spezifiziert werden, die mit den iterierten Anteilen der vorliegenden Arbeit vergleichbar sind. Zum anderen können Knoten spezifiziert werden, die einem Nichtterminal einer kontextfreien Grammatik entsprechen. Die Grammatik wird mit Hilfe spezieller Graphtransformationsregeln angegeben und kann sämtliche gültigen Programmgraphen generieren. Nichtterminalknoten können auch als Mengenknoten spezifiziert werden. Darüber hinaus kann derselbe Nichtterminalknoten mehrfach in der rechten Seite einer Regel spezifiziert werden. Dadurch wird die durch den Knoten repräsentierte Struktur entsprechend häufig kopiert. Auf diese Weise können auch Refactorings wie *Inline Method*, das Gegenstück zu *Extract Method*, spezifiziert werden, bei dem eine Methode aufgelöst und ihre Aufrufe jeweils durch eine Kopie des Methodeninhalts ersetzt werden. Dies ist mit den Transformationsdiagrammen der vorliegenden Arbeit nicht allgemein möglich.

Bei den so spezifizierten Regeln handelt es sich um generische Regeln, aus denen bei Anwendung auf einen Wirtsgraphen in dessen Abhängigkeit konkrete Regeln abgeleitet werden. Dabei werden Mengenknoten durch eine Menge einzelner Knoten ersetzt. Nichtterminalknoten werden durch die Strukturen im Wirtsgraphen ersetzt, von denen sie abstrahieren. Die Strukturen werden durch Anwendung von Regeln der Grammatik bestimmt. Diese Bestimmung ist nicht zwingend eindeutig, so dass der Re-Engineer in den Prozess der Regelableitung einbezogen werden muss.

Die Arbeiten sind der vorliegenden Arbeit ähnlich. Sie haben gemeinsam, dass Programmtransformationen durch Graphtransformationen und Eigenschaften in Form zu erhaltender sowie verbotener Strukturen beschrieben werden. Die Graphtransformationsregeln sind ähnlich ausdrucksstark wie die Transformationsdiagramme der vorliegenden Arbeit, ohne den Einsatz von Kontrollfluss. Dadurch können sie allerdings keine Alternativen spezifizieren: jede Alternative muss in einer eigenen Regel angegeben werden. Auf der anderen Seite ermöglichen sie das Kopieren beliebiger Strukturen, was Transformationsdiagramme nicht können. Des Weiteren kann mit Hilfe der Grammatik für Programmgraphen und den Nichtterminalknoten erreicht werden, dass die Regeln gültige Programmgraphen in wiederum gültige Programmgraphen transformieren. Einige Wohlgeformtheitseigenschaften sind dann per Konstruktion

erfüllt. Allerdings gilt dies nicht für alle Eigenschaften, da weiterhin beliebige Modifikationen spezifiziert oder notwendige Modifikationen vergessen werden können. Ein Verfahren zur Verifikation der Spezifikationen auf Einhaltung von Wohlgeformtheits- oder Verhaltenseigenschaften gibt es nicht, ebenso wenig wie eine Werkzeugunterstützung für den Ansatz.

Verbaere, Ettinger und de Moor definieren in [VEM06] eine Skriptsprache zur Spezifikation von Refactorings namens *JunGL (Jungle Graph Language)*, die eine funktionale Sprache (ML) mit Prädikatenlogik (Datalog) kombiniert. Die zu transformierende Software wird als abstrakter Syntaxgraph repräsentiert. Die Repräsentation kann um Funktionen zur Berechnung von Kanten bei Bedarf, so genannte *Lazy Edges*, erweitert werden. Solche Kanten werden erst dann mit Hilfe der hinterlegten Funktion ermittelt, wenn sie zur Ausführungszeit einer Spezifikation traversiert werden sollen. Darüber hinaus erlaubt die Sprache die Spezifikation von Funktionen und die Auswertung von Prädikaten und Pfadausdrücken auf der Graphrepräsentation sowie letztendlich die Modifikation des Graphen. Der Ansatz wurde in *C#* in Verbindung mit *F#* (einer funktionalen Sprache) auf Basis der Microsoft *.NET*-Plattform für das Refactoring von C#-Programmen implementiert. Der Nachweis der Einhaltung von Progammeigenschaften durch spezifizierte Refactorings wird nicht thematisiert.

7.2.3 Refactoring formaler Modelle

Eine Reihe von Ansätzen befasst sich mit dem Refactoring formaler Modelle. Sunye et al. [SPTJ01] sowie Boger, Sturm und Fragemann [BSF02] beschreiben Ansätze, die Refactorings auf UML-Modellen (Klassendiagrammen, Statecharts, Aktivitätsdiagrammen) durchführen anstatt auf der Quelltextebene. Zum Modell gehöriger Quelltext wird dabei entweder nicht berücksichtigt oder es bleibt unklar, ob dies geschieht. Die Durchführung der Refactorings wird in Werkzeugen implementiert und auch hier nicht abstrakter formalisiert.

Van Kempen et al. [KCKB05] formalisieren die Semantik von UML Klassen- und Zustandsdiagrammen mit *CSP*. Semantikerhaltung von Modell-Refactorings, die selbst nicht formalisiert werden, wird dann über einen Verfeinerungsbegriff auf Basis der CSP-Formalisierung der Modelle vor und nach einem Refactoring gezeigt.

Engels et al. [EHKG02] formalisieren Teile von *UML-RT*-Modellen mit Hilfe von *CSP* und formulieren Konsistenzbedingungen (zum Beispiel Deadlock-Freiheit) in CSP. Transformationsregeln werden mit Hilfe von Graphtransformationsregeln spezifiziert, für die auf Basis von CSP bewiesen werden kann,

dass sie die Konsistenzbedingungen garantiert einhalten.

Wehrheim et al. [ERW07, RW07, EW08] befassen sich ebenfalls mit dem Refactoring formaler Spezifikationen in *Z* und *CSP*. Dabei werden formale Methoden eingesetzt beziehungsweise untersucht, um auf Basis der formalen Semantik der Spezifikationen die Äquivalenz transformierter Spezifikationen beziehungsweise allgemein die Semantikerhaltung von Refactorings zu beweisen.

Die Ansätze betrachten ausschließlich die Modellebene. Damit sie für das Re-Engineering bestehender Anwendungen eingesetzt werden könnten, müssten diese vollständig auf der Modellebene repräsentiert werden.

Rangel et al. stellen in [RLK+08] aufbauend auf [RKE07] einen Ansatz für das Refactoring von Modellen mit Hilfe von Graphtransformationsregeln am Beispiel von endlichen Automaten vor. Das Metamodell für endliche Automaten wird als Typgraph definiert. Die operationale Semantik des Metamodells wird durch Graphtransformationsregeln im Double-Pushout-Ansatz mit *Borrowed Contexts* [EK04, EK06] definiert. Auf dieser Basis kann eine *Bisimulationsrelation* für zwei Modelle berechnet werden, um zu überprüfen, ob sie semantisch äquivalent sind – dies ist der Fall, wenn in beiden Modellen rekursiv dieselben Regeln, unter bestimmten Bedingungen durch Vervollständigung von Anwendungsstellen, anwendbar sind. Refactorings werden durch eine Menge von Graphtransformationsregeln ebenfalls auf Basis des Metamodells definiert, die in Schichten (engl. *layers*) angeordnet sind. Bei der Ausführung werden solange wie möglich Regeln der ersten Schicht angewendet. Ist dies nicht mehr möglich, werden solange es geht Regeln der zweiten Schicht angewendet und so fort.

Die Idee ist, die einzelnen Graphtransformationsregeln eines Refactorings auf Verhaltenserhaltung zu überprüfen, indem jeweils ihre linke und ihre rechte Seite als unvollständige Modelle aufgefasst werden und mit Hilfe der Borrowed-Context-Technik eine Bisimulationsrelation für die beiden Regelseiten berechnet wird. Gelingt dies, handelt es sich um eine verhaltenserhaltende Regel, da die durch die linke und rechte Seite beschriebenen Modelle definitv bisimilar sind. Bei dieser Art der Überprüfung muss allerdings jede einzelne Regel eines Refactorings verhaltenserhaltend sein, was in der Regel nicht der Fall ist. Dann muss überprüft werden, ob es ausgehend von der Anwendung einer nicht verhaltenserhaltenden Regel eine Sequenz weiterer Anwendungen von Regeln desselben Refactorings gibt, so dass wieder ein semantisch äquivalentes Modell hergestellt wird. Wie dies funktioniert, ist noch nicht untersucht.

Um den Ansatz auf das Refactoring von Java-Programmen wie in der vorliegenden Arbeit anzuwenden, müsste die operationale Semantik von Java

vollständig mit Hilfe von Graphtransformationsregeln beschrieben sein. Der dabei verwendete Typgraph müsste sowohl den Zustand des Programms als auch seine Struktur repräsentieren, damit auf seiner Basis auch eine Restrukturierung spezifiziert werden könnte. Würde dies gelingen, bliebe zu untersuchen, ob eine Bisimulationsrelation für zwei Programme noch effizient genug berechnet werden könnte. Außerdem würde weiterhin das Problem mit nicht verhaltenserhaltenden Regeln eines Refactorings bestehen.

7.2.4 Exogene Modelltransformationen

Die bisher vorgestellten Arbeiten befassen sich ausschließlich mit der Transformation von Instanzen desselben Modells, also von Worten derselben Sprache. Nach Mens und Van Gorp [MG06] fallen solche Transformationen – Refactorings im Allgemeinen – in die Kategorie der *endogenen* Modelltransformationen. Demgegenüber stehen *exogene* Modelltransformationen, die Instanzen verschiedener Modelle, also Worte verschiedener Sprachen, ineinander übersetzen.

Küster et al. [KHE03, Kü06] zum Beispiel verwenden Graphtransformationsregeln in Verbindung mit Kontrollflusskonstrukten, um *UML-Statecharts* in *CSP* zu übersetzen. Auf Basis der Formalisierung werden Kriterien formal formuliert, anhand derer die Terminierung sowie die Eindeutigkeit einer Übersetzung nachgewiesen werden kann. Wagner [Wag09] etwa verwendet mit *Triple-Graph-Grammatiken* (TGG) [Sch94] spezielle Graphtransformationen, mit denen die Übersetzung zweier Sprachen ineinander auf Basis ihrer Grammatiken formal definiert werden kann, und erarbeitet darauf aufbauend ein Verfahren, um Modelle in verschiedenen Sprachen zueinander konsistent zu halten.

Bei der Übersetzung stellt sich ebenso die Frage, ob ineinander übersetzte Modelle semantisch äquivalent sind. Es gibt zahlreiche Arbeiten, die sich mit exogenen Modelltransformationen befassen und auf Basis einer geeigneten Formalisierung der Semantik der jeweiligen Sprachen versuchen, Aussagen zur semantischen Äquivalenz zu treffen.

Dazu gehören zum Beispiel Engels et al. [EKR+08], die *UML-Aktivitätsdiagramme* in eine Java-ähnliche textuelle Sprache namens *TAAL* übersetzen. Die Übersetzung wird mit Hilfe TGG-ähnlicher Graphtransformationsregeln definiert. Für beide Sprachen wird die Semantik mit Graphtransformationsregeln formalisiert, so dass für Worte beider Sprachen Transitionssysteme aufgebaut werden können, deren Zustände durch Graphen beschrieben sind. Semantische Äquivalenz soll auf Basis der Transitionssysteme gezeigt werden.

Diese und weitere Arbeiten aus dem Bereich der exogenen Modelltransformationen sind für das Refactoring aber nicht einsetzbar oder nur wenig praktikabel. Daher werden sie hier nicht weiter betrachtet.

7.2.5 Transformationsspezifikation anhand von Beispielen

Für exogene Modelltransformationen stellen Varro [Var06] und Wimmer et al. [WSKK07] halbautomatische Verfahren zur Vereinfachung der Transformationsspezifikation anhand von Übersetzungsbeispielen vor.

Varro geht von einer grundlegenden Abbildung von Elementen zweier Modelle aufeinander aus und ermittelt aus zwei Instanzen – eine für jedes Modell – automatisch initiale Graphtransformationsregeln, welche die Instanzen ineinander übersetzen. Die Regeln müssen dann manuell nachbearbeitet werden.

Wimmer et al. gehen ebenfalls von zwei Instanzen in konkreter Syntax aus und lassen den Benutzer korrespondierende Elemente bestimmen. Auf Basis einer Abbildung der konkreten Syntax auf die jeweilige abstrakte Syntax werden dann aus den Instanzen unter Einsatz von Heuristiken automatisch Transformationsregeln in der Sprache ATL (ATLAS Transformation Language) [JK05] ermittelt. ATL ist ausschließlich zur Spezifikation exogener Modelltransformationen geeignet.

Beide Ansätze sind nicht ohne Weiteres auf die Synthese endogener Transformationen auf Basis von Quelltextbeispielen anwendbar, wie sie in Kapitel 3.4 vorgestellt wird. Vergleichbare Ansätze dazu sind nicht bekannt.

Baar und Whittle [BW06] stellen dagegen am Beispiel von Klassendiagrammen einen Ansatz vor, der die konkrete Syntax einer Modellierungssprache so erweitert, dass mit ihr auch endogene Transformationen ihrer Instanzen spezifiziert werden können. Das Problem ist, die Syntaxerweiterung möglichst gering und nahe an der ursprünglichen Syntax zu halten. Dieses Problem tritt insbesondere bei Quelltext auf, so dass dieser Ansatz im Rahmen der vorliegenden Arbeit nicht weiter verfolgt wurde.

7.3 Verifikation von Graphtransformationssystemen

Ein Transformationsdiagramm besteht aus einer Reihe von speziellen Graphtransformationsregeln, deren Ausführung durch den Kontrollfluss des Diagramms kontrolliert wird. Ein Transformationsdiagramm kann damit als ein

in Bezug auf die Anwendbarkeit einzelner Regeln eingeschränktes Graphtransformationssystem [Roz97] aufgefasst werden, bei dem der Startgraph zum Zeitpunkt der Spezifikation nicht feststeht. Dieser ist erst im Moment der Ausführung des Transformationsdiagramms in einem konkreten Programm mit dessen abstrakter Syntaxgraphrepräsentation bekannt. Im Rahmen der Forschung sind einige Ansätze zur Verifikation von Graphtransformationssystemen entstanden, die im Folgenden vorgestellt werden.

7.3.1 Korrektheit per Konstruktion

Heckel und Wagner [HW95] erweitern bestehende Graphtransformationsregeln so um negative Anwendungsbedingungen, dass sie definierte Konsistenzeigenschaften nicht verletzen können. Die Konsistenzeigenschaften werden in vergleichbarer Form zu den verbotenen Graphmustern der vorliegenden Arbeit ohne Pfade angegeben. Die rechten Seiten der Graphtransformationsregeln werden auf alle möglichen Weisen mit den Konsistenzeigenschaften verklebt und durch Rückwärtsanwendung werden Situationen ermittelt, in denen die Anwendung der Regel zur Verletzung der Eigenschaft führen würde. Diese Situationen werden zu negativen Anwendungsbedingungen der jeweiligen Graphtransformationsregeln, so dass diese garantiert keine Konsistenzeigenschaften mehr verletzen können.

Der Ansatz erweitert Graphtransformationsregeln automatisch um negative Anwendungsbedingungen, um die Verletzung von Konsistenzeigenschaften in Form der Erzeugung verbotener Strukturen garantiert zu vermeiden. Ob die Regeln danach überhaupt noch anwendbar sind, wird nicht betrachtet. Auch werden verbotene Strukturen getrennt voneinander betrachtet, so dass möglicherweise unnötige negative Anwendungsbedingungen erzeugt werden und zwar in Fällen, in denen bereits eine andere als die betrachtete verbotene Struktur vorliegt.

Des Weiteren können weder die Regeln noch die verbotenen Strukturen Pfade enthalten, so dass sie nicht ausdrucksstark genug für die Anwendung im Rahmen der vorliegenden Arbeit sind. Zudem werden nur einzelne Regeln ohne Iteration betrachtet. Bei einzelner Betrachtung auch von iterierten Anteilen, würde die Anwendung eines Anteils verhindert, sobald er eine verbotene Struktur herstellen kann, ohne weitere Anteile zu berücksichtigen, die garantiert eine Korrektur herbeiführen würden. Dies ist für die Anwendung im Rahmen der vorliegenden Arbeit zu restriktiv. In Fällen, in denen erst die mehrfache Anwendung eines iterierten Anteils zu einer verbotenen Situation führt, würde der Ansatz bei einzelner Betrachtung von Anteilen genau die Anwendung, die

eine verbotene Struktur vervollständigt, verhindern. Bis dahin würden Anteile angewendet.

Allgemein ist fraglich, inwieweit die automatische Anpassung von Regeln mit der Intention des Re-Engineers übereinstimmt.

Ein ähnlicher Ansatz, der ausdrucksstärkere Konsistenzeigenschaften ebenfalls ohne Pfade zulässt, wird von Ehrig et al. in [EEHP04] vorgestellt und von Taentzer und Rensink in [TR05] aufgegriffen und so erweitert, dass er für Graphtransformationsregeln und Konsistenzeigenschaften anwendbar ist, deren Typgraphen Vererbung erlauben. Die Ansätze unterliegen denselben, bereits zuvor genannten Einschränkungen in Bezug auf die Problemstellung der vorliegenden Arbeit.

Koch et al. [KMPP02, KPP06] greifen den Ansatz von Heckel und Wagner [HW95] auf und erweitern zum einen die Ausdrucksmächtigkeit der Konsistenzeigenschaften, wiederum ohne Pfade. Zum anderen werden Regeln nur dann um negative Anwendungsbedingungen erweitert, wenn sie eine Konsistenzeigenschaft tatsächlich verletzen können. Dazu wird ausgehend von einem Startgraphen des Systems ein beschränktes (engl. *bounded*) *Model Checking* (siehe nächster Abschnitt) durchgeführt. Dabei wird durch Anwendung von Regeln bis zu einer zuvor bestimmten maximalen Anzahl der Zustandsraum aufgebaut. Jeder Zustand wird auf die Verletzung von Konsistenzeigenschaften untersucht. Wird eine Verletzung festgestellt, wird die zuletzt angewendete Regel entsprechend um eine negative Anwendungsbedingung erweitert. Der Ansatz unterliegt damit denselben Einschränkungen wie die zuvor beschriebenen Ansätze. Außerdem müssen die Regeln zu stark eingeschränkt werden, um die maximale Anzahl für das beschränke Model Checking bestimmen zu können – sie dürfen nur entweder Elemente löschen oder erzeugen und nur Regeln, die Elemente löschen, dürfen negative Anwendungsbedingungen besitzen.

7.3.2 Model Checking

Eine Reihe weiterer Ansätze beweisen Eigenschaften von Graphtransformationssystemen mit Hilfe von *Model Checking*. Die Grundlagen für das Model Checking von Graphtransformationssystemen wurden von Heckel et al. [HEWC97] gelegt. Darauf aufbauend wurden die Ansätze CheckVML (Check Visual Modeling Languages) von Varro et al. [Var04, SV03], OBGG (Object-Based Graph Grammars) von Dotti et al. [DFRS03] und GROOVE (GRaph-based Object-Oriented VErification) von Rensink [Ren08] entwickelt.

Die Ansätze bilden ein Graphtransformationssystem auf ein gerichtetes Transitionssystem ab, bei dem die Zustände einem konkreten Graphen und

die Transitionen Graphtransformationsregeln entsprechen. Jede Transition verbindet dabei einen Graphen, auf den die zugehörige Regel angewendet werden kann, mit dem durch die Regelanwendung resultierenden Graphen. CheckVML und OBGG bilden die Transitionssysteme wiederum auf eine Eingabesprache für einen Model Checker ab, der dann für die Verifikation eingesetzt wird. GROOVE führt das Model Checking dagegen direkt auf den Graphen durch.

Alle Ansätze haben gemeinsam, dass sie ausgehend von einem Startgraphen durch Anwendung von Regeln den gesamten Zustandsraum aufbauen und die Erreichbarkeit bestimmter Zustände prüfen. Transformationsdiagramme entsprechen im Prinzip speziellen Graphtransformationssystemen und könnten mit Hilfe der hier vorgestellten Ansätze auf Einhaltung von Kriterien verifiziert werden, sofern ein Startgraph bekannt ist. Dies ist bei Transformationsdiagrammen jedoch zum Zeitpunkt ihrer Spezifikation nicht der Fall. Der Startgraph entspricht der Repräsentation eines Programms als abstrakter Syntaxgraph und ist damit erst bei Anwendung einer Transformation bekannt.

7.3.3 Analyse mit Petrinetztechniken

Die beiden folgenden Ansätze verwenden Petrinetztechniken zur Analyse von Graphtransformationssystemen.

Baldan, Corradini und König [BCK01, BCK04, BCK08] stellen einen Ansatz vor, der ein Graphtransformationssystem auf einen *Petrigraphen* abbildet. Ein Petrigraph besteht aus einem Graphen und einem Petrinetz, so dass jeder durch das Graphtransformationssystem erreichbare Graph einer Markierung des Petrinetzes entspricht. Zudem kann jeder erreichbare Graph durch einen Homomorphismus auf den Graphen des Petrigraphs abgebildet werden. Auf diese Weise können zum einen existierende Petrinetztechniken eingesetzt werden, um Lebendigkeitseigenschaften und Tranisitionsinvarianten auf dem Petrinetz nachzuweisen, die sich auf das Graphtransformationssysteme übertragen lassen. Zum anderen sind auf dem Graphen des Petrigraphen Erreichbarkeitsanalysen und die Erkennung von Verklemmungen möglich. Allerdings sind die Graphtransformationssysteme starken Einschränkungen unterworfen. Sie verfügen nicht über negative Anwendungsbedindungen und dürfen zudem nur Kanten aber keine Knoten löschen. Des Weiteren benötigt das Verfahren ebenso wie die zuvor vorgestellten Model-Checking-Ansätze einen Startgraph, so dass es nicht zur Verifikation von Transformationsdiagrammen geeignet ist.

Padberg und Enders [PE02] übertragen Techniken der Petrinetzanalyse auf Graphtransformationssysteme, um zum Beispiel die Erreichbarkeit von Graphen oder die Lebendigkeit von Graphtransformationssystemen zu überprüfen.

Auch das Prinzip der Transitionsinvarianten wird auf *Regelinvarianten* übertragen, die besagen, dass eine Sequenz von Graphtransformationsregeln wieder den Ausgangsgraphen herstellt. Die Regeln verfügen nicht über negative Anwendungsbedingungen, wie sie zur Spezifikation von Programmtransformationen notwendig sind. Darüber hinaus wird ein Startgraph zur Analyse benötigt.

7.4 Zusammenfassung

Dieses Kapitel beschreibt verwandte Arbeiten anderer Wissenschaftler und setzt sie zur vorliegenden Arbeit in Beziehung. Es gibt eine Reihe von Arbeiten, die sich der reinen Erkennung von Schwachstellen in bestehender Software widmen. Keiner der Ansätze kombiniert dabei strukturelle Eigenschaften so mit quantitativen Eigenschaften, dass eine kontinuierliche Bewertung erkannter Schwachstellen vorgenommen werden kann, wie es in der vorliegenden Arbeit geschieht.

Im Bereich der Restrukturierung von Software gibt es viele Arbeiten, die spezielle Transformationen auf Basis spezieller Programmanalysen vornehmen. Es gibt häufig eine abgeschlossene Menge von Transformationen deren Durchführung selbst nicht formal spezifiziert wird. Ansätzen, die dem Re-Engineer die Möglichkeit zur Spezifikation eigener Transformationen geben, fehlt eine Unterstützung für die automatische Verifikation der Transformationen auf Einhaltung definierbarer Kriterien, auf deren Basis Aussagen zur Verhaltenserhaltung von Transformationen getroffen werden können. Ansätze, die Verhaltenserhaltung der transformierten Modelle automatisch nachweisen, sind auf eine geeignete formale Semantikdefinition für die transformierten Modelle angewiesen und nicht ohne Weiteres auf die Transformation von zum Beispiel Java-Programmen anwendbar.

Die vorliegende Arbeit stellt dem Re-Engineer eine auf Graphtransformationen basierende formale Spezifikationssprache zur Verfügung, mit der auf ebenfalls graphbasiert spezifizierte Schwachstellen direkt Bezug genommen und ihre Verbesserung spezifiziert werden kann. Darüber hinaus wird ein Verfahren entwickelt, das Transformationsspezifikationen automatisch auf Einhaltung ebenfalls graphbasiert definierbarer Kriterien verifizieren kann. Ein solches Verfahren wird hier neu entwickelt, da die dargestellten verwandten Ansätze aus dem Bereich der Verifikation von Graphtransformationen beziehungsweise von Graphtransformationssystemen nicht anwendbar sind, weil entweder die Voraussetzungen nicht gegeben sind und/oder die verifizierbaren Regeln und/oder Kriterien nicht ausdrucksstark genug sind.

Einen geschlossenen, durchgängigen Ansatz zur Erkennung und Verbesserung von Schwachstellen sowie der Verifikation der Verbesserungstransformationen, vergleichbar zur vorliegenden Arbeit, gibt es nicht.

Kapitel 8

Zusammenfassung und Ausblick

In diesem letzten Kapitel der vorliegenden Arbeit werden zunächst die erzielten Ergebnisse zusammengefasst. Danach wird ein Ausblick auf mögliche Erweiterungen des vorgestellten Ansatzes gegeben.

8.1 Zusammenfassung

Die Wartung und Erweiterung bestehender Softwaresysteme durch *Re-Engineering* überwiegt heute bei weitem die reine Neuentwicklung. Daher ist die Wartbarkeit von Software, die letztlich bestimmt wie einfach Anpassungen vorgenommen werden können, von entscheidender Bedeutung.

Die vorliegende Arbeit stellt einen Ansatz vor, der das Re-Engineering von (objektorientierter) Software durch eine Beurteilung und Verbesserung ihrer Wartbarkeit auf Basis von Softwaremustern unterstützt.

Die Beurteilung der Wartbarkeit einer Software wird durch die automatisierte Erkennung und Bewertung von Strukturmustern im Quelltext einer Software unterstützt. Dazu wird mit der strukturbasierten Mustererkennung ein bestehendes Verfahren zur Erkennung von Entwurfsmusterinstanzen aufgegriffen und für die Erkennung von Schwachstellen, Instanzen von Bad Smells und Anti Patterns, eingesetzt und erweitert. Das Verfahren erlaubt eine grafische Spezifikation von Strukturmustern auf Basis einer abstrakten Syntaxgraphrepräsentation des Quelltextes. Die Strukturmuster werden mit Hilfe von Graphtransformationen formalisiert, die von einem Erkennungsalgorithmus ausgeführt werden.

Die Kenntnis über vorhandene Entwurfsmusterinstanzen und Schwachstellen gestattet wohl strukturierte, wartbare Software von schlecht strukturierter, schwer verständlicher Software zu unterscheiden.

Im Unterschied zu Entwurfsmustern beziehen sich Schwachstellen häufig auf quantifizierbare Merkmale wie zum Beispiel die Anzahl von Quelltextzeilen, Attributen und Methoden einer Klasse oder die Anzahl und den Umfang von If-Anweisungen. Anhand ihrer Ausprägung können signifikante, schwerwiegende Funde von weniger signifikanten Funden unterschieden werden. Daher wurde die strukturbasierte Mustererkennung um eine Bewertung ihrer Funde auf Basis von Softwareproduktmetriken erweitert. Zum einen wurde dazu die Spezifikation von (Fuzzy-)Metrikbedingungen zusammen mit Bewertungsfunktionen für Elemente von Strukturmustern ermöglicht und zum anderen wurde der Erkennungsalgorithmus um deren Berücksichtigung erweitert.

Über die Erkennung von Schwachstellen zur Beurteilung der Wartbarkeit hinaus, stellt die vorliegende Arbeit einen Ansatz zur Verbesserung erkannter Schwachstellen durch Transformation in bessere Lösungen vor. Zur Spezifikation von verbessernden Programmtransformationen wird mit *Transformationsdiagrammen* eine grafische Sprache entwickelt, mit der die Modifikation des abstrakten Syntaxgraphen einer Software beschrieben werden kann. Sie ist so mit der strukturbasierten Mustererkennung integriert, dass sich Transformationen auf erkannte Schwachstellen beziehungsweise allgemein Strukturmusterinstanzen beziehen und direkt auf sie angewendet werden können. Die Semantik der Sprache wird auf Basis der Theorie der Graphtransformationen formal definiert. Um die Erstellung von Transformationsdiagrammen zu vereinfachen, wird zudem ein Ansatz zur Spezifikation anhand von Quelltextbeispielen vorgestellt.

Bei der Transformation von Software zur Verbesserung ihrer Wartbarkeit soll in der Regel die Struktur der Software verbessert werden, ohne ihr Verhalten zu verändern (*Refactoring*). Dass eine Transformation das Verhalten eines Programms nicht verändert, kann häufig nur mit erheblichem manuellem Aufwand oder gar nicht bewiesen werden. Die vorliegende Arbeit stellt ein automatisches Verfahren vor, dass eine Verifikation von Transformationsdiagrammen auf Einhaltung definierbarer struktureller Kriterien erlaubt. Das Verfahren versucht nachzuweisen, dass die Kriterien (induktive) Strukturinvarianten sind, die in allen Versionen von abstrakten Syntaxgraphen beliebiger Programme gelten, die durch Ausführung kompletter Transformationsdiagramme entstehen können. Dies wurde insofern bereits bei der Entwicklung der Transformationsdiagramme berücksichtigt, als dass ihre Ausdrucksmächtigkeit sowohl auf die Durchführung von Programmtransformationen als auch ihre Verifikation ausgelegt wurde.

Mit Hilfe der Kriterien können strukturelle Eigenschaften des abstrakten Syntaxgraphen formuliert werden, die durch eine Transformation erhalten oder

vermieden werden müssen. Kann eine Transformation ein Kriterium verletzen, also eine zu erhaltende Struktur zerstören oder eine verbotene Struktur erzeugen, so liefert das Verfahren ein Beispiel für eine entsprechende problematische Ausführung der Transformation, dass heißt ein Gegenbeispiel für die Einhaltung des Kriteriums. Dies geschieht ohne Kenntnis eines konkreten zu transformierenden Programms. Vielmehr wird die für die problematische Ausführung notwendige Beschaffenheit eines Programms ausschnittsweise charakterisiert.

Dabei kann es sein, dass eine Transformation nie auf ein so beschaffenes Programm angewendet werden wird – genauso wenig kann dies jedoch ausgeschlossen werden. Im Umkehrschluss kann eine Transformation, für die für ein Kriterium kein Gegenbeispiel gefunden wurde, das Kriterium keinesfalls verletzen, egal auf was für Programme sie angewendet wird. Dies wird in der vorliegenden Arbeit zwar nicht mathematisch formal bewiesen, sondern informal argumentiert, allerdings ergibt sich allein durch die systematische Berechnung von Gegenbeispielen bereits ein wesentlicher Mehrwert für den Re-Engineer.

Wenn ein vollständiger Beweis der Verhaltenserhaltung damit auch nicht notwendigerweise möglich ist, da die Kriterien dafür nicht zwingend hinreichend sein müssen, können auf diese Weise bestimmte Verhaltensänderungen ausgeschlossen oder erkannt werden.

Für den vorgestellten Ansatz wurde auf Basis von Fujaba4Eclipse eine Werkzeugunterstützung prototypisch implementiert und in die Entwicklungsumgebung Eclipse integriert. Mit ihrer Hilfe wurden Strukturmuster für 21 Schwachstellen spezifiziert und fünf Softwaresysteme analysiert. Dabei wurden zahlreiche signifikante Schwachstellen identifiziert, die aufgrund ihrer Bewertung gut von weniger signifikanten Funden unterschieden werden konnten. Für einige der Schwachstellen wurden komplexe Verbesserungstransformationen spezifiziert und ausgeführt sowie auf Einhaltung einiger Kriterien verifiziert.

8.2 Ausblick

Die Arbeiten an dem hier vorgestellten Ansatz sind sicher nicht abgeschlossen. Vielmehr gibt es einige vielversprechende Aspekte, die weiter untersucht werden sollten.

So sollte zum Beispiel eine intensivere Evaluierung der Erkennung von Schwachstellen vorgenommen werden, insbesondere im Hinblick auf die Bewertung der Funde. Dabei sollte die im Rahmen der Evaluierung vorgestellte Systematik zur Bestimmung von Bewertungsfunktionen weiter überprüft wer-

den. Auch sollte der Einsatz von Lernverfahren untersucht werden, um die Bewertungsfunktionen möglicherweise auf Basis einer Rückmeldung durch den Re-Engineer automatisiert zu erlernen.

Die Ausdrucksmächtigkeit der hier vorgestellten Transformationsdiagramme erlaubt die Spezifikation komplexer Programmtransformationen. Allerdings gibt es auch solche Transformationen wie zum Beispiel Inline Method (das Gegenteil von Extract Method), bei der eine bestehende Methode aufgelöst und ihr Inhalt direkt an alle Aufrufstellen kopiert wird, oder Push Down Method, bei der eine Methodenimplementierung von einer Klasse in all ihre Subklassen verschoben beziehungsweise kopiert wird. Diese Kopieroperationen können derzeit nicht allgemeingültig, sondern nur für Methoden mit zur Spezifikationszeit bekanntem Inhalt formuliert werden. Hier müsste ein Kopieroperator für Teilbäume eines abstrakten Syntaxgraphen eingeführt werden. Zudem müssten dessen Auswirkungen auf die Verifikation untersucht werden.

Im Rahmen der Evaluierung wurden bereits Erweiterungen der Transformationsspezifikation auf Ebene der Werkzeugunterstützung vorgenommen. Zum einen wurden negative Elemente sowohl in iterierten Anteilen als auch in nicht-iterierten Anteilen von Transformation Pattern aufgenommen, wodurch auch negative Elemente in negativen Vorbedingungen einer Transformation spezifiziert werden können. Hierfür ist zum einen eine Erweiterung der formalen Semantikdefinition erforderlich. Zum anderen muss das Verifikationsverfahren so angepasst werden, dass für negative Elemente in iterierten Anteilen oder erfolgreichen nicht-iterierten Anteilen einer Regelsequenz weitere negative Anwendungsbedingungen für die entsprechenden Graphmuster der Sequenz erzeugt werden.

Für auf einem Ausführungspfad fehlschlagende Transformation Pattern werden bereits negative Anwendungsbedingungen in den Graphmustern generiert. Enthält eine solche negative Anwendungsbedingung wiederum negative Elemente, so muss dies bei der Überprüfung, ob die negative Anwendungsbedingung im Widerspruch zu den Anwendungsbedingungen der Graphmuster des Gegenbeispiels steht, korrekt berücksichtigt werden.

Ebenfalls im Rahmen der Evaluierung entstand der Bedarf, bei geordneten Assoziationen, in denen wie zum Beispiel bei Anweisungen in einem Block oder bei Parametern einer Methode die Reihenfolge der Elemente eine Rolle spielt, die Reihenfolge der Erzeugung von Links durch ein Transformation Pattern explizit zu bestimmen. Hier bieten sich verschiedene Möglichkeiten an. Zum einen könnten die durch ein Transformation Pattern vorzunehmenden Löschungen und Erzeugungen bereits auf Ebene der Spezifikation explizit sequentialisiert werden. Eine andere Möglichkeit, die bereits auf Ebene von Story Diagrammen

beziehungsweise Story Pattern zur Verfügung steht, sind so genannte *Multi Links*. Dabei handelt es sich um gerichtete Verbindungen zwischen Linkvariablen einer geordneten Assoziation, die eine relative Reihenfolge der durch sie repräsentierten Links festlegen. Eine Erweiterung des Verifikationsverfahrens ist nur dann erforderlich, wenn auch Kriterien Aussagen über die Reihenfolge von Elementen in einer Assoziation treffen sollen. Dies ist aber durchaus sinnvoll.

Bei der Spezifikation von Transformationen müssen häufig komplexe Ausschnitte eines abstrakten Syntaxgraphen in Transformation Pattern manuell modelliert werden. Mit der Transformationsspezifikation anhand konkreter Quelltextbeispiele hat die vorliegende Arbeit einen Ansatz vorgestellt, der dies vereinfachen kann. Aufgrund der fehlenden prototypischen Werkzeugunterstützung konnte dieser Ansatz im Rahmen der Evaluierung nicht erprobt werden. Dies sollte nachgeholt werden.

Ein letzter Aspekt in Bezug auf die Spezifikation und insbesondere die Ausführung von Transformationsspezifikationen ist die Interaktion mit dem Re-Engineer. Im Rahmen dieser Arbeit werden die Transformationen zuerst vollständig spezifiziert und danach so ausgeführt. Wenn es bei der Extraktion einer Methode zum Beispiel zu einer Namenskollision kommt, schlägt die ExtractMethod-Transformation fehl. Stattdessen könnte die Transformation in einem solchen Fall unterbrochen und der Re-Engineer nach einem anderen Namen befragt werden, so dass eine Kollision vermieden wird. Ebenso kann es bei der Bindung von Strukturmusterannotationen sinnvoll sein, den Re-Engineer zu befragen, wenn mehrere Annotationen, etwa für eine lesende Zugriffsmethode eines Attributs (GetMethod), zur Verfügung stehen. Anstatt wie bisher zufällig eine der Annotationen zu binden könnte der Re-Engineer gefragt werden, welche Annotation verwendet werden soll. Dies sollte weiter untersucht werden.

Für das Verifikationsverfahren wurden bereits einige Erweiterungsmöglichkeiten im Zusammenhang mit Erweiterungen der Transformationsspezifikation diskutiert. Darüber hinaus gibt es sinnvolle Erweiterungsmöglichkeiten für die Verifikation allein.

Dazu zählt die bereits in Abschnitt 6.4 angeführte Einführung von Strukturinvarianten sowie deren Verifikation, um bestimmte Situationen wie zum Beispiel die Zuordnung von Anweisungen zu mehr als einem Vaterknoten jederzeit ausschließen zu können. Solche Invarianten können dann genutzt werden, um insbesondere die Berechnung von Pfadausprägungen zu präzisieren.

Eine weitere Verbesserung wäre eine präzisere Analyse der Auswirkungen aufgerufener Transformationen. In der vorliegenden Arbeit werden diese als

black box betrachtet. Dadurch wird unterstellt, dass sie alle Arten von Elementen erzeugen können, obwohl ihnen dies unter Umständen gar nicht möglich ist. Darüber hinaus können im Aufrufer von Transformationen bisher Bedingungen nicht berücksichtigt werden, die durch aufgerufene Transformationen geschaffen werden. Die somit entstehende pessimistische Abschätzung bietet viel Raum für false-positives, also zu Unrecht ermittelte Gegenbeispiele, so dass hier ebenfalls eine signifikante Verbesserung der Präzision des Verfahrens erzielt werden kann.

Damit hängt auch zusammen, dass Strukturmusterannotationen derzeit wie herkömmliche Objekte behandelt werden, obwohl sie für das Vorhandensein einer bestimmten Struktur im abstrakten Syntaxgraphen stehen können. Würde dies ausgenutzt, könnte ebenfalls eine höhere Präzision erzielt werden.

Dann sollte das Verifikationsverfahren geeignet formalisiert werden, so dass auf dieser Basis mathematische Beweise betreffend seine Koffektheit geführt werden können, um die in der vorliegenden Arbeit gegebene Argumentation mathematisch zu fundieren.

Schließlich sollte untersucht werden, inwieweit eine Systematik entwickelt werden kann, die den Re-Engineer dabei unterstützt zum einen Verifikationskriterien herzuleiten und zum anderen zu entscheiden, welche Transformationsdiagramme welche der Kriterien einhalten und daher dahingehend verifiziert werden sollten.

Literatur

[AA01] Albin-Amiot, H. : JavaXL, a Java source code transformation engine / Ecole des Mines de Nantes. 2001. – Forschungsbericht

[AAG01] Albin-Amiot, H. ; Guéhéneuc, Y.-G. : Design Patterns: A Round-Trip. In: Ardourel, G. (Hrsg.); Haupt, M. (Hrsg.); Agustin, J. L. H. (Hrsg.); Ruggaber, R. (Hrsg.); Suscheck, C. (Hrsg.): *proceedings of the 11th ECOOP workshop for Ph.D. Students in Object-Oriented Systems*, 2001

[AOH04] Apiwattanapong, T. ; Orso, A. ; Harrold, M. J.: A Differencing Algorithm for Object-Oriented Programs. In: *ASE*, IEEE Computer Society, 2004. – ISBN 0–7695–2131–2, S. 2–13

[Arg] *ArgoUML*. http://argouml.tigris.org/. – Stand: Juni 2009

[BBG+06] Becker, B. ; Beyer, D. ; Giese, H. ; Klein, F. ; Schilling, D. : Symbolic Invariant Verification for Systems with Dynamic Structural Adaptation. In: *Proc. of the 28th International Conference on Software Engineering (ICSE), Shanghai, China*, ACM Press, 2006

[BC98] Bär, H. ; Ciupke, O. : Exploiting Design Heuristics for Automatic Problem Detection. In: Demeyer, S. (Hrsg.); Bosch, J. (Hrsg.): *Object-Oriented Technology, ECOOP'98 Workshop Reader, ECOOP'98 Workshops, Demos, and Posters, Brussels, Belgium, Proceedings* Bd. 1543, Springer, July 1998, S. 73–74

[BCK01] Baldan, P. ; Corradini, A. ; König, B. : A Static Analysis Technique for Graph Transformation Systems. In: *CONCUR '01: Proceedings of the 12th International Conference on Concurrency Theory*. London, UK: Springer-Verlag, 2001. – ISBN 3–540–42497–0, S. 381–395

[BCK04] Baldan, P. ; Corradini, A. ; König, B. : Verifying finite-state graph grammars: an unfolding-based approach. In: *In Proc. of CONCUR '04*, Springer-Verlag, 2004, S. 83–98

[BCK08] Baldan, P. ; Corradini, A. ; König, B. : A framework for the verification of infinite-state graph transformation systems. In: *Inf. Comput.* 206 (2008), Nr. 7, S. 869–907. – ISSN 0890–5401

[BDH+01] van den Brand, M. ; van Deursen, A. ; Heering, J. ; de Jong, H. ; de Jonge, M. T. K.; Klint, P. ; Moonen, L. ; Olivier, P. ; Scheerder, J. ; Vinju, J. ; Visser, E. ; Visser, J. : The ASF+SDF Meta-Environment: A Component-Based Language Development Environment, 2001, S. 365–370

[BGN+04] Burmester, S. ; Giese, H. ; Niere, J. ; Tichy, M. ; Wadsack, J. P.; Wagner, R. ; Wendehals, L. ; Zündorf, A. : Tool Integration at the Meta-Model Level within the FUJABA Tool Suite. In: *International Journal on Software Tools for Technology Transfer (STTT)* 6 (2004), August, Nr. 3, S. 203–218

[BMMM98] Brown, W. ; Malveau, R. ; McCormick, H. ; Mombray, T. : *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis.* New York, NY, USA: John Wiley and Sons, Inc., 1998

[BMR+96] Buschmann, F. ; Meunier, R. ; Rohnert, H. ; Somerlad, P. ; Stal, M. : *Pattern-Oriented Software Architecture - A System of Patterns.* 1st edition. John Wiley and Sons, Inc., 1996

[BSF02] Boger, M. ; Sturm, T. ; Fragemann, P. : Refactoring Browser for UML. In: *Proc. Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering (XP), Alghero, Sardinia, Italy*, 2002, S. 77–81

[BW06] Baar, T. ; Whittle, J. : On the Usage of Concrete Syntax in Model Transformation Rules. In: Virbitskaite, I. (Hrsg.); Voronkov, A. (Hrsg.): *Ershov Memorial Conference* Bd. 4378, Springer, 2006. – ISBN 978–3–540–70880–3, S. 84–97

[Cas94] Casais, E. : Automatic reorganization of object-oriented hierarchies: a case study. In: *Object Oriented Systems* 1 (1994), S. 95–115

[CC90] Chikofsky, E. J.; Cross II, J. H.: Reverse Engineering and Design Recovery: A Taxonomy. In: *IEEE Software* 7 (1990), Januar, Nr. 1, S. 13–17

[Ceb07] Cebeci, Y. : *Spezifikation von Codetransformationen anhand konkreter Beispiele*, Universität Paderborn, Paderborn, Deutschland, Studienarbeit, August 2007

[Ciu99] Ciupke, O. : Automatic Detection of Design Problems in Object-Oriented Reengineering. In: Firesmith, D. (Hrsg.); Meyer, B. (Hrsg.); Pour, G. (Hrsg.); Riehle, R. (Hrsg.): *Proc. of Technology of Object-Oriented Languages and Systems - TOOLS 30, Santa Barbara, Kalifornien, USA*, 1999, S. 18–32

[CK94] Chidamber, S. R.; Kemerer, C. F.: A Metrics Suite for Object Oriented Design. In: *IEEE Trans. Softw. Eng.* 20 (1994), Nr. 6, S. 476–493. – ISSN 0098–5589

[CKV96] Coplien, J. ; Kerth, N. ; Vlissides, J. : *Pattern Languages of Program Design*. Volume 3. Addison-Wesley, 1996

[Cop92] Coplien, J. O.: *Advanced C++, Programming Styles and Idioms*. Addison-Wesley, 1992

[CRGMW96] Chawathe, S. S.; Rajaraman, A. ; Garcia-Molina, H. ; Widom, J. : Change Detection in Hierarchically Structured Information. In: Jagadish, H. V. (Hrsg.); Mumick, I. S. (Hrsg.): *SIGMOD Conference*, ACM Press, 1996, S. 493–504

[DD04] von Dincklage, D. ; Diwan, A. : Converting Java classes to use generics. In: *SIGPLAN Not.* 39 (2004), Nr. 10, S. 1–14. – ISSN 0362–1340

[DDN03] Demeyer, S. ; Ducasse, S. ; Nierstrasz, O. : *Object-Oriented Reengineering Patterns*. Morgan Kaufmann Publishers, 2003

[DFRS03] Dotti, F. L.; Foss, L. ; Ribeiro, L. ; dos Santos, O. M.: Verification of Distributed Object-Based Systems. In: Najm, E. (Hrsg.); Nestmann, U. (Hrsg.); Stevens, P. (Hrsg.): *FMOODS* Bd. 2884, Springer, 2003. – ISBN 3–540–20491–1, S. 261–275

[DHJ+06] Drewes, F. ; Hoffmann, B. ; Janssens, D. ; Minas, M. ; Eetvelde, N. V.: Adaptive Star Grammars. In: et al., H. E. (Hrsg.): *International Conference on Graph Transformation (ICGT), Natal, Rio Grande do Norte, Brazil* Bd. 4178, Springer-Verlag, September 2006, S. 77–91

[DHJ+07] Drewes, F. ; Hoffmann, B. ; Janssens, D. ; Minas, M. ; Eetvelde, N. V.: Shaped Generic Graph Transformation. In: Schürr, A. (Hrsg.); Nagl, M. (Hrsg.); Zündorf, A. (Hrsg.): *Proceedings of Applications of Graph Transformation with Industrial Relevance (AGTIVE'07), Kassel, Germany*, Universität Kassel, Oktober 10-12 2007, S. pp. 197–212

[Eas93] Eastwood, A. : Firm fires shots at legacy systems. In: *Computing Canada* 19 (1993), Nr. 2, S. 17

[Ecla] Eclipse Foundation Inc.: *Eclipse*. http://www.eclipse.org/. – Stand: Oktober 2009

[Eclb] Eclipse Foundation Inc.: *Standard Widget Toolkit*. http://www.eclipse.org/swt/. – Stand: Oktober 2009

[EEHP04] Ehrig, H. ; Ehrig, K. ; Habel, A. ; Pennemann, K.-H. : Constraints and Application Conditions: From Graphs to High-Level Structures. In: Ehrig, H. (Hrsg.); Engels, G. (Hrsg.); Parisi-Presicce, F. (Hrsg.); Rozenberg, G. (Hrsg.): *ICGT* Bd. 3256, Springer, 2004. – ISBN 3–540–23207–9, S. 287–303

[EHKG02] Engels, G. ; Heckel, R. ; Küster, J. M.; Groenewegen, L. : Consistency-Preserving Model Evolution through Transformations. 2002, S. 212–227

[EJ04] van Eetvelde, N. ; Janssens, D. : Extending Graph Rewriting for Refactoring. In: et al., H. E. (Hrsg.): *International Conference on Graph Transformation (ICGT), Rome, Italy* Bd. 3256, Springer-Verlag, November 2004, S. 399–415

[EJ05] Eetvelde, N. V.; Janssens, D. : Refactorings as graph transformations / University of Antwerp. 2005 (UA WIS/INF 2005/04). – Forschungsbericht

[EK04] Ehrig, H. ; König, B. : Deriving bisimulation congruences in the DPO approach to graph rewriting. In: Walukiewicz (Hrsg.): *Proc. of FoSSaCS'04* Bd. LNCS 2987, Springer, 2004, S. 151–166

[EK06] Ehrig, H. ; König, B. : Deriving bisimulation congruences in the DPO approach to graph rewriting with borrowed contexts. New York, NY, USA: Cambridge University Press, 2006. – ISSN 0960–1295, S. 1133–1163

[EKR+08] Engels, G. ; Kleppe, A. ; Rensink, A. ; Semenyak, M. ; Soltenborn, C. ; Wehrheim, H. : From UML Activities to TAAL - Towards Behaviour-Preserving Model Transformations. 2008, S. 94–109

[ELS86] Engels, G. ; Lewerentz, C. ; Schäfer, W. : Graph Grammar Engineering: A Software Specification Method. In: Ehrig, H. (Hrsg.); Nagl, M. (Hrsg.); Rozenberg, G. (Hrsg.); Rosenfeld, A. (Hrsg.): *Graph-Grammars and Their Application to Computer Science* Bd. 291, Springer, 1986. – ISBN 3–540–18771–5, S. 186–201

[EM02] van Emden, E. ; Moonen, L. : Java Quality Assurance by Detecting Code Smells. In: *Proceedings of the 9th Working Conference on Reverse Engineering*, 2002

[Emb] Embarcadero Technologies: *JBuilder 2008*. http://www.codegear.com/products/jbuilder. – Stand: Oktober 2009

[Erl00] Erlikh, L. : Leveraging Legacy System Dollars for E-Business. In: *IT Professional* 02 (2000), Nr. 3, S. 17–23. – ISSN 1520–9202

[ERW07] Estler, C. H.; Ruhroth, T. ; Wehrheim, H. : Modelchecking Correctness of Refactorings - Some Experiments. In: *Electronic Notes in Theoretical Computer Science* 187 (2007), July, S. 3–17

[EV] Ettinger, R. ; Verbaere, M. : *Refactoring bugs in Eclipse, IntelliJ IDEA and Visual Studio*. http://progtools.comlab.ox.ac.uk/projects/refactoring/bugreports. – Stand: Oktober 2009

[EW08] Estler, C. H.; Wehrheim, H. : Alloy as a Refactoring Checker? In: *Electronic Notes in Theoretical Computer Science* 214 (2008), June, S. 331–357

[FMv97] Florijn, G. ; Meijers, M. ; van Winsen, P. : Tool Support for Object-Oriented Patterns. Jyväskylä, Finland, 1997. – ISBN 3–540–63089–9, S. 472–495

[FNTZ98] Fischer, T. ; Niere, J. ; Torunski, L. ; Zündorf, A. : Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In: Engels, G. (Hrsg.); Rozenberg, G. (Hrsg.): *Proc. of the* 6^{th} *International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany*, Springer Verlag, November 1998 (LNCS 1764), S. 296–309

[Fow99] Fowler, M. : *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999

[Fow02] Fowler, M. : *Patterns of Enterprise Application Architecture.* Reading, Massachusetts: Addison Wesley, November 2002. – ISBN 0321127420

[FP96] Fenton, N. E.; Pfleeger, S. L.: *Software Metrics - A Rigorous & Practical Approach.* Second Edition. International Thompson Computer Press, 1996

[Fra92] Frazer, A. : Reverse Engineering - hype, hope or here? In: Hall, P. (Hrsg.): *Software Reuse and Reverse Engineering in Practice* Bd. Jgg. 12 der UNICOM Applied Information Technology. London, Großbritannien: Chapman & Hall, 1992, S. 209–243

[FTK+05] Fuhrer, R. ; Tip, F. ; Kieżun, A. ; Dolby, J. ; Keller, M. : Efficiently refactoring Java applications to use generic libraries. In: *ECOOP 2005 — Object-Oriented Programming, 19th European Conference.* Glasgow, Scotland, Juli 27–29, 2005, S. 71–96

[Fuj] University of Paderborn, Germany: *Fujaba Tool Suite.* http://www.fujaba.de/. – Stand: Oktober 2009

[GA08] Gueheneuc, Y.-G. ; Antoniol, G. : DeMIMA: A Multilayered Approach for Design Pattern Identification. In: *Software Engineering, IEEE Transactions on* 34 (2008), Sept.-Oct., Nr. 5, S. 667–684. – ISSN 0098–5589

[GAA01] Guéhéneuc, Y.-G. ; Albin-Amiot, H. : Using Design Patterns and Constraints to Automate the Detection and Correction of Inter-Class Design Defects. In: Li, Q. (Hrsg.); Riehle, R. (Hrsg.); Pour, G. (Hrsg.); Meyer, B. (Hrsg.): *proceedings of the 39th conference on the Technology of Object-Oriented Languages and Systems*, IEEE Computer Society Press, Juli 2001, S. 296–305

[GHH+08] Giese, H. ; Henkler, S. ; Hirsch, M. ; Roubin, V. ; Tichy, M. : Modeling Techniques for Software-Intensive Systems. In: Tiako, D. P. F. (Hrsg.): *Designing Software-Intensive Systems: Methods and Principles*. Langston University, OK, 2008, S. 21–58

[GHJV95] Gamma, E. ; Helm, R. ; Johnson, R. ; Vlissides, J. : *Design Patterns: Elements of Reusable Object Oriented Software*. Reading, MA, USA: Addison-Wesley, 1995

[GMW06] Giese, H. ; Meyer, M. ; Wagner, R. : A Prototype for Guideline Checking and Model Transformation in Matlab/Simulink. In: Giese, H. (Hrsg.); Westfechtel, B. (Hrsg.): *Proc. of the 4rd International Fujaba Days 2006, Bayreuth, Deutschland* Bd. tr-ri-06-275, Universität Paderborn, September 2006

[HEWC97] Heckel, R. ; Ehrig, H. ; Wolter, U. ; Corradini, A. : Integrating the Specification Techniques of Graph Transformation and Temporal Logic. In: Prívara, I. (Hrsg.); Ruzicka, P. (Hrsg.): *MFCS* Bd. 1295, Springer, 1997. – ISBN 3–540–63437–1, S. 219–228

[HJE06] Hoffmann, B. ; Janssens, D. ; Eetvelde, N. V.: Cloning and Expanding Graph Transformation Rules for Refactoring. In: *Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005)* Bd. 152, 2006, S. 53–67

[Hor90] Horwitz, S. : Identifying the Semantic and Textual Differences Between Two Versions of a Program. In: *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, ACM Press, 1990, S. 234–244

[HS96] Henderson-Sellers, B. : *Object-Oriented Metrics: Measures of Complexity.* Prentice-Hall, 1996

[Huf90] Huff, S. : Information systems maintenance. In: *The Business Quarterly* 55:30–32 (1990)

[HW95] Heckel, R. ; Wagner, A. : Ensuring consistency of conditional graph grammars – a constructive approach. In: *Proc. of SEGRAGRA'95 "Graph Rewriting and Computation", Electronic Notes of TCS*, 1995, S. 2

[Int] IntelliJ: *IntelliJIDEA 8.1.* http://www.jetbrains.com/idea/. – Stand: Oktober 2009

[JHD] *JHotDraw.* http://www.jhotdraw.org. – Stand: Oktober 2009

[JK05] Jouault, F. ; Kurtev, I. : Transforming Models with ATL. In: *Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005.* Montego Bay, Jamaica, 2005

[JL94] Jackson, D. ; Ladd, D. : Semantic Diff: A Tool for Summarizing the Effects of Modifications. In: *Proceedings of the International Conference on Software Maintenance*, IEEE Computer Society, 1994, S. 243–252

[KBSD04] Kothari, S. C.; Bishop, L. ; Sauceda, J. ; Daugherty, G. : A Pattern-Based Framework for Software Anomaly Detection. In: *Software Quality Journal* 12 (2004), Nr. 2, S. 99–120

[KCKB05] van Kempen, M. ; Chaudron, M. ; Kourie, D. ; Boake, A. : Towards proving preservation of behaviour of refactoring of UML models. In: *SAICSIT '05: Proceedings of the 2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries.* , Republic of South Africa: South African Institute for Computer Scientists and Information Technologists, 2005. – ISBN 1–59593–258–5, S. 252–259

[KEGN01] Kataoka, Y. ; Ernst, M. D.; Griswold, W. G.; Notkin, D. : Automated Support for Program Refactoring Using Invariants. In: *Proceedings of the International Conference on Software Maintenance*, IEEE Press, 2001, S. 736–743

[Ker04] Kerievsky, J. : *Refactoring to Patterns.* Addison-Wesley, 2004

[KETF07] Kiezun, A. ; Ernst, M. D.; Tip, F. ; Fuhrer, R. M.: Refactoring for Parameterizing Java Classes. In: *ICSE '07: Proceedings of the 29th international conference on Software Engineering.* Washington, DC, USA: IEEE Computer Society, 2007. – ISBN 0–7695–2828–7, S. 437–446

[KHE03] Küster, J. M.; Heckel, R. ; Engels, G. : Defining and validating transformations of UML models, 2003, S. 145–152

[KK04] Kniesel, G. ; Koch, H. : Static composition of refactorings. In: *Sci. Comput. Program.* 52 (2004), Nr. 1-3, S. 9–51. – ISSN 0167–6423

[KMPP02] Koch, M. ; Mancini, L. V.; Parisi-Presicce, F. : Decidability of Safety in Graph-Based Models for Access Control. In: Gollmann, D. (Hrsg.); Karjoth, G. (Hrsg.); Waidner, M. (Hrsg.): *ESORICS* Bd. 2502, Springer, 2002. – ISBN 3–540–44345–2, S. 229–243

[KNNZ00] Köhler, H. J.; Nickel, U. A.; Niere, J. ; Zündorf, A. : Integrating UML Diagrams for Production Control Systems. In: *Proc. of the* 22^{nd} *International Conference on Software Engineering (ICSE), Limerick, Ireland*, ACM Press, 2000, S. 241–251

[KPP06] Koch, M. ; Parisi-Presicce, F. : UML specification of access control policies and their formal verification. In: *Software and System Modeling* 5 (2006), Nr. 4, S. 429–447

[Kre05] Kreimer, J. : Adaptive Detection of Design Flaws. In: *Electr. Notes Theor. Comput. Sci.* 141 (2005), Nr. 4, S. 117–136

[Kü06] Küster, J. M.: Definition and validation of model transformations. In: *Software and Systems Modeling* V5 (2006), Nr. 3, S. 233–259

[Lan99] Langr, J. : *Essential Java Style: Patterns for Implementation.* 1st edition. Prentice Hall, 1999

[Lan03] Lanza, M. : *Object-Oriented Reverse Engineering*, University of Berne, Switzerland, Dissertation, 2003

[LDGP05] Lanza, M. ; Ducasse, S. ; Gall, H. ; Pinzger, M. : CodeCrawler – An Information Visualization Tool for Program Comprehension, ACM Press, 2005, S. 672–673

[LK94] Lorenz, M. ; Kidd, J. : *Object-Oriented Software Metrics: A Practical Guide.* Prentice-Hall, 1994

[LS92] Laski, J. ; Szermer, W. : Identification of Program Modifications and its Applications in Software Maintenance, 1992, S. 282–290

[Mar04] Marinescu, R. : Detection strategies: metrics-based rules for detecting design flaws. In: *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on* (2004), Sept., S. 350–359. – ISSN 1063–6773

[Mat] The MathWorks, Inc.: *MATLAB.* http://www.mathworks.com/products/matlab/. – Stand: Oktober 2009

[McC76] McCabe: A Complexity Measure. In: *IEEE Transactions on Software Engineering* 2 (1976), S. 308–320

[MDJ02] Mens, T. ; Demeyer, S. ; Janssens, D. : Formalising Behaviour Preserving Program Transformations. In: *ICGT '02: Proceedings of the First International Conference on Graph Transformation.* London, UK: Springer-Verlag, 2002. – ISBN 3–540–44310–X, S. 286–301

[MEJD03] Mens, T. ; van Eetvelde, N. ; Janssens, D. ; Demeyer, S. : Formalising Refactorings with Graph Transformations. In: *Fundamenta Informaticae* (2003)

[Mey06] Meyer, M. : Pattern-based Reengineering of Software Systems. In: *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE 2006).* Benevento, Italien: IEEE Computer Society, Oktober 2006, S. 305–306

[MG06] Mens, T. ; Gorp, P. V.: A Taxonomy of Model Transformation. In: *Electr. Notes Theor. Comput. Sci.* 152 (2006), S. 125–142

[MGL06] Moha, N. ; Guéhéneuc, Y.-G. ; Leduc, P. : Automatic Generation of Detection Algorithms for Design Defects. In: *ASE*, IEEE Computer Society, 2006. – ISBN 0–7695–2579–2, S. 297–300

[MGMD08] Moha, N. ; Guéhéneuc, Y.-G. ; Meur, A.-F. L.; Duchien, L. : A Domain Analysis to Specify Design Defects and Generate Detection Algorithms. In: Fiadeiro, J. L. (Hrsg.); Inverardi, P. (Hrsg.): *FASE* Bd. 4961, Springer, 2008. – ISBN 978–3–540–78742–6, S. 276–291

[MGMR02] Melnik, S. ; Garcia-Molina, H. ; Rahm, E. : Similarity flooding: a versatile graph matching algorithm and its application to schema matching. In: *Proc. of the 18th Int'l. Conf. on Data Engineering*, 2002, S. 117–128

[MHGV08] Moha, N. ; Hacene, A. M. R.; Guéhéneuc, Y.-G. ; Valtchev, P. : Refactorings of Design Defects Using Relational Concept Analysis. In: Medina, R. (Hrsg.); Obiedkov, S. A. (Hrsg.): *ICFCA* Bd. 4933, Springer, 2008. – ISBN 978–3–540–78136–3, S. 289–304

[Mic] Microsoft: *Visual Studio 2008*. http://www.microsoft.com/products/info/. – Stand: Oktober 2009

[Moa90] Moad, J. : Maintaining the Competitive Edge. In: *Datamation* 36 (1990), Februar, S. 61–66

[Moo96] Moore, I. : Automatic inheritance hierarchy restructuring and method refactoring. In: *OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA: ACM, 1996. – ISBN 0–89791–788–X, S. 235–250

[MS99] Maruyama, K. ; Shima, K. : Automatic Method Refactoring Using Weighted Dependence Graphs. In: *ICSE*, 1999, S. 236–245

[MTW93] Müller, H. A.; Tilley, S. R.; Wong, K. : Understanding software systems using reverse engineering technology perspectives from the Rigi project. In: *CASCON '93: Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research*, IBM Press, 1993, S. 217–226

[Mun05] Munro, M. J.: Product Metrics for Automatic Identification of Bad Smell Design Problems in Java Source-Code. In: *METRICS '05: Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS'05)*. Washington, DC, USA: IEEE Computer Society, 2005. – ISBN 0–7695–2371–4, S. 15

[Nie04] Niere, J. : *Inkrementelle Entwurfsmustererkennung*, University of Paderborn, Paderborn, Germany, Dissertation, Juni 2004

[NSW+02] Niere, J. ; Schäfer, W. ; Wadsack, J. P.; Wendehals, L. ; Welsh, J. : Towards Pattern-Based Design Recovery. In: *Proc. of the 24^{th} International Conference on Software Engineering (ICSE), Orlando, Florida, USA*, ACM Press, Mai 2002, S. 338–348

[Obj] Object Management Group (OMG): *Unified Modeling Language (UML)*. http://www.uml.org/. – Stand: April 2007

[O'C01] O'Cinnéide, M. : *Automated Application of Design Patterns: a Refactoring Approach*, University of Dublin, Trinity College, Dissertation, 2001

[ON99] O'Cinnéide, M. ; Nixon, P. : A Methodology for the Automated Introduction of Design Patterns. In: *ICSM '99: Proceedings of the IEEE International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 1999. – ISBN 0–7695–0016–1, S. 463

[ON01] O'Cinnéide, M. ; Nixon, P. : Automated software evolution towards design patterns. In: *IWPSE '01: Proceedings of the 4th International Workshop on Principles of Software Evolution*. New York, NY, USA: ACM Press, 2001. – ISBN 1–58113–508–4, S. 162–165

[Opd92] Opdyke, W. F.: *Refactoring Object-Oriented Frameworks*, University of Illinois at Urbana-Champaign, Dissertation, 1992

[PE02] Padberg, J. ; Enders, B. : Rule Invariants in Graph Transformation Systems for Analyzing Safety-Critical Systems. In: Corradini, A. (Hrsg.); Ehrig, H. (Hrsg.); Kreowski, H.-J. (Hrsg.); Rozenberg, G. (Hrsg.): *ICGT* Bd. 2505, Springer, 2002. – ISBN 3–540–44310–X, S. 334–350

[PLo] *Pattern Languages of Programs Conferences and Publications.* http://hillside.net/conferences/plop.htm. – Stand: Oktober 2009

[RBJ97] Roberts, D. ; Brant, J. ; Johnson, R. E.: A Refactoring Tool for Smalltalk. In: *Theory and Practice of Object Systems* 3 (1997), Nr. 4, S. 253–263

[Ren08] Rensink, A. : Explicit State Model Checking for Graph Grammars. In: Degano, P. (Hrsg.); Nicola, R. D. (Hrsg.); Meseguer, J. (Hrsg.): *Concurrency, Graphs and Models* Bd. 5065, Springer, 2008. – ISBN 978–3–540–68676–7, S. 114–132

[Rie96] Riel, A. J.: *Object-Oriented Design Heuristics.* Addison-Wesley, 1996

[Rig] *Rigi: A Visual Tool for Understanding Legacy Systems.* http://www.rigi.csc.uvic.ca/. – Stand: Oktober 2009

[Ris00] Rising, L. ; Vlissides, J. M. (Hrsg.): *The Pattern Almanac 2000.* Boston, MA, USA: Addison-Wesley, 2000 (Software Patterns Series)

[RKE07] Rangel, G. ; König, B. ; Ehrig, H. : Bisimulation Verification for the DPO Approach with Borrowed Contexts. In: *Proc. of GT-VMT '07 (Workshop on Graph Transformation and Visual Modeling Techniques)* Bd. 6, 2007

[RLK+08] Rangel, G. ; Lambers, L. ; König, B. ; Ehrig, H. ; Baldan, P. : Behavior Preservation in Model Refactoring using DPO Transformations with Borrowed Contexts. In: *Proc. of the 4th International Conference on Graph Transformation (ICGT).* Leicester, United Kingdom, September 2008

[Rob99] Roberts, D. B.: *Practical Analysis for Refactoring*, University of Illinois at Urbana-Champaign, Dissertation, 1999

[Roz97] Rozenberg, G. (Hrsg.): *Handbook of Graph Grammars and Computing by Graph Transformation*. Bd. 1. Singapur: World Scientific Publishing, 1997

[RW07] Ruhroth, T. ; Wehrheim, H. : Refactoring Object-Oriented Specifications with Data and Processes. 2007, S. 236–251

[SA] Seguin, C. ; Atkinson, M. : *JRefactory*. http://jrefactory.sourceforge.net/. – Stand: Oktober 2009

[Sch91] Schürr, A. ; Nagl, M. (Hrsg.): *Operationales Spezifizieren mit programmierten Graphersetzungssystemen*. Deutscher Universitäts-Verlag (DUV), 1991

[Sch94] Schürr, A. : Specification of Graph Translators with Triple Graph Grammars. In: *Proc. of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science*. Herrschin, Germany: Spinger Verlag, Juni 1994

[Sch06] Schilling, D. : *Kompositionale Softwareverifikation mechatronischer Systeme*, University of Paderborn, Paderborn, Germany, Dissertation, Februar 2006

[SGMZ98] Schulz, B. ; Genssler, T. ; Mohr, B. ; Zimmer, W. : On the computer aided introduction of design patterns into object-oriented systems. In: *Proc. Int'l Conf. TOOLS*, IEEE Press, 1998

[SPL03] Seacord, R. C.; Plakosh, D. ; Lewis, G. A.: *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. – ISBN 0321118847

[SPTJ01] Sunyé, G. ; Pollet, D. ; Traon, Y. L.; Jézéquel, J.-M. : Refactoring UML Models. In: *UML'01: Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*. London, UK: Springer-Verlag, 2001. – ISBN 3–540–42667–1, S. 134–148

[SS04] Streckenbach, M. ; Snelting, G. : Refactoring class hierarchies with KABA. In: *OOPSLA '04: Proceedings of the 19th*

annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. New York, NY, USA: ACM, 2004. – ISBN 1–58113–831–9, S. 315–330

[SSL01] Simon, F. ; Steinbrückner, F. ; Lewerentz, C. : Metrics Based Refactoring. In: *In Proc. 5th European Conference on Software Maintenance and Reengineering*, IEEE Computer Society Press, 2001, S. 30–38

[ST00] Snelting, G. ; Tip, F. : Understanding class hierarchies using concept analysis. In: *ACM Trans. Program. Lang. Syst.* 22 (2000), Nr. 3, S. 540–582. – ISSN 0164–0925

[Sun] Sun Microsystems: *NetBeans IDE 6.7*. http://www.netbeans.org/. – Stand: Oktober 2009

[SV03] Schmidt, Á. ; Varró, D. : CheckVML: A Tool for Model Checking Visual Modeling Languages. In: Stevens, P. (Hrsg.); Whittle, J. (Hrsg.); Booch, G. (Hrsg.): *Proc. UML 2003: 6th International Conference on the Unified Modeling Language* Bd. 2863. San Francisco, CA, USA: Springer, October 20-24 2003, S. 92–95

[Tah03] Tahvildari, L. : *Quality-Driven Object-Oriented Re-engineering Framework*, Department of Electrical and Computer Engineering, University of Waterloo, Ontario, Canada, Dissertation, August 2003

[Tah04] Tahvildari, L. : Quality-Driven Object-Oriented Re-Engineering Framework. In: *Proc. of the 20th International Conference on Software Maintenance (ICSM 2004), Chicago, IL, USA*, IEEE Computer Society, September 2004, S. 479–483

[TB01] Tokuda, L. ; Batory, D. : Evolving Object-Oriented Designs with Refactorings. In: *Automated Software Engg.* 8 (2001), Nr. 1, S. 89–120. – ISSN 0928–8910

[The] The Hillside Group: *Patterns Library*. http://hillside.net/. – Stand: Oktober 2009

[TK02a] Tahvildari, L. ; Kontogiannis, K. : A Methodology for Developing Transformations Using the Maintainability Soft-Goal

Graph. In: VAN DEURSEN, A. (Hrsg.); BURD, E. (Hrsg.): *Proc. of the 9th Working Conference on Reverse Engineering (WCRE 2002), Richmond, VA, USA*, IEEE Computer Society, November 2002, S. 77–86

[TK02b] TAHVILDARI, L. ; KONTOGIANNIS, K. : On the Role of Design Patterns in Quality-Driven Re-engineering. In: *Proc. of the 6th European Conference on Software Maintenance and Reengineering (CSMR 2002), Budapest, Hungary*, IEEE Computer Society, March 2002, S. 230–240

[TM05] TRIFU, A. ; MARINESCU, R. : Diagnosing Design Problems in Object Oriented Systems. In: *WCRE '05: Proceedings of the 12th Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 2005. – ISBN 0–7695–2474–5, S. 155–164

[TR05] TAENTZER, G. ; RENSINK, A. : Ensuring Structural Constraints in Graph-Based Models with Type Inheritance. In: CERIOLI, M. (Hrsg.): *Fundamental Approaches to Software Engineering (FASE), Edinburgh, UK* Bd. 3442, Springer-Verlag, April 2005. – ISBN 3–540–25420–X, S. 64–79

[TR07] TRIFU, A. ; REUPKE, U. : Towards Automated Restructuring of Object Oriented Systems. In: *Software Maintenance and Reengineering, 2007. CSMR '07. 11th European Conference on* (2007), März, S. 39–48. – ISSN 1534–5351

[Tra06] TRAVKIN, D. : *Bewertung automatisch erkannter Instanzen von Software-Mustern*, Universität Paderborn, Paderborn, Deutschland, Diplomarbeit, August 2006

[Tsc67] TSCHEBYSCHOW, P. L.: On Mean Values. In: *J. Math. Pures. Appl.* 2 (1867), Nr. 12, S. 177–184

[Var04] VARRÓ, D. : Automated formal verification of visual modeling languages by model checking. In: *Software and System Modeling* 3 (2004), Nr. 2, S. 85–113

[Var06] VARRO, D. : Model Transformation by Example. In: NIERSTRASZ, O. (Hrsg.); WHITTLE, J. (Hrsg.); HAREL, D. (Hrsg.);

Reggio, G. (Hrsg.): *Proc. of 9th International Conference Model Driven Engineering Languages and Systems (MoDELS)* Bd. 4199. Genova, Italy: Springer, Oktober 2006, S. 410–424

[VEM06] Verbaere, M. ; Ettinger, R. ; de Moor, O. : JunGL: a scripting language for refactoring. In: *ICSE '06: Proceeding of the 28th international conference on Software engineering*. New York, NY, USA: ACM, 2006. – ISBN 1–59593–375–1, S. 172–181

[Wag06] Wagner, R. : Developing Model Transformations with Fujaba. In: Giese, H. (Hrsg.); Westfechtel, B. (Hrsg.): *Proc. of the 4th International Fujaba Days 2006, Bayreuth, Germany* Bd. tr-ri-06-275, University of Paderborn, September 2006, S. 79–82

[Wag09] Wagner, R. : *Inkrementelle Modellsynchronisation*, University of Paderborn, Paderborn, Germany, Dissertation, July 2009. – In German

[Wen07] Wendehals, L. : *Struktur- und verhaltensbasierte Entwurfsmustererkennung*, University of Paderborn, Paderborn, Germany, Dissertation, September 2007

[WK06] Wenzel, S. ; Kelter, U. : Model-Driven Design Pattern Detection Using Difference Calculation. In: *Proc. of the 1st International Workshop on Pattern Detection For Reverse Engineering (DPD4RE), co-located with 13th Working Conference on Reverse Engineering (WCRE'06)*. Benevento, Italy: IEEE Computer Society, Oktober 2006

[WSKK07] Wimmer, M. ; Strommer, M. ; Kargl, H. ; Kramler, G. : Towards Model Transformation Generation By-Example. In: *HICSS*, IEEE Computer Society, 2007, S. 285

[XS05] Xing, Z. ; Stroulia, E. : UMLDiff: an algorithm for object-oriented design differencing. In: *International Conference on Automated Software Engineering*. Long Beach, CA, USA: ACM Press, 2005. – ISBN 1–59593–993–4, S. 54–65

[ZKDZ07] Zhao, C. ; Kong, J. ; Dong, J. ; Zhang, K. : Pattern-based design evolution using graph transformation. In: *J. Vis. Lang. Comput.* 18 (2007), Nr. 4, S. 378–398. – ISSN 1045–926X

[Zün96] Zündorf, A. ; Nagl, M. (Hrsg.): *Programmierte Graphersetzungssysteme: Spezifikation, Implementierung und Anwendung einer integrierten Entwicklungsumgebung.* Deutscher Universitäts-Verlag (DUV), 1996

[Zün02] Zündorf, A. : *Rigorous Object Oriented Software Development with Fujaba.* http://www.se.eecs.uni-kassel.de/se/fileadmin/se/publications/Zuen02.pdf. 2002. – Stand: Oktober 2009

Anhang A

Verkleben von Graphmustern mit Pfaden

In diesem Anhang wird das Verkleben von Graphmustern, die Pfade enthalten, mit Graphen einer Regel prinzipiell veranschaulicht. Den folgenden Beispielen liegt dabei das in Abbildung A.1 gezeigte um Pfadnavigationsinformationen erweiterte Strukturmodell (vgl. Abschnitt 3.2.1) zugrunde.

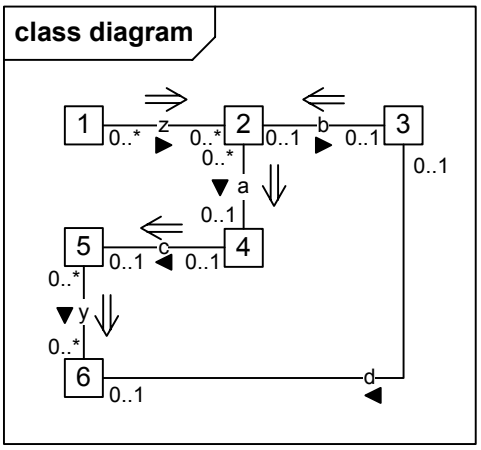

Abbildung A.1: Beispielstrukturmodell mit Pfadnavigationsinformationen

Das Verkleben von Graphmustern mit Pfaden findet in zwei Stufen statt. In der ersten Stufe wird das Graphmuster zunächst ohne Pfade mit einem Regelgraphen verklebt, wobei eine Menge neuer Graphmuster entsteht. In der zweiten Stufe werden in jedem dieser neuen Graphmuster die Pfade des ursprünglichen Graphmusters auf Teilausprägungen abgebildet.

A.1 Verkleben ohne Pfade

Ein Graphmuster mit Pfaden wird zunächst ohne Berücksichtigung der Pfade mit einem Regelgraphen zu neuen Graphmustern verklebt, wie es bereits in [Sch06] beschrieben wird:

1. werden alle Teilgraphen des Graphmusters ohne Pfade bestimmt,
2. werden für jeden Teilgraphen des Graphmusters alle dazu isomorphen Teilgraphen des Regelgraphen bestimmt,
3. wird für jedes Paar isomorpher Teilgraphen ein neues Graphmuster angelegt, in das:
   a) der Regelgraph vollständig kopiert wird,
   b) alle Elemente des Graphmusters kopiert werden, die nicht im Teilgraph enthalten sind,
   c) alle Kanten des Graphmusters zwischen Knoten des Teilgraphen und Knoten, die nicht Bestandteil des Teilgraphen sind, kopiert und mit den Knoten im neuen Graphmuster verbunden werden, auf die die ursprünglichen Knoten des Graphmusters durch den Isomorphismus zwischen den Teilgraphen abgebildet werden.

Abbildung A.2 zeigt dies anhand eines Beispiels. Auf der linken Seite sind oben ein Graphmuster mit einem Pfad und unten ein Regelgraph abgebildet. Die Knoten- und Kantenbeschriftungen zeigen ihren Typ aus dem Strukturmodell aus Abbildung A.1 an.

Die blauen Markierungen zeigen an, welche Teilgraphen durch einen Isomorphismus aufeinander abgebildet wurden. Auf der rechten Seite wird das neue Graphmuster gezeigt, das beim Verkleben der Graphen anhand der isomorphen Teilgraphen entsteht.

A.2 Abbilden von Pfaden

Pfade eines Graphmusters werden beim Verkleben in den neuen Graphmustern zunächst zwischen den entsprechenden Knoten hinzugefügt. Tatsächlich können Elemente des neuen Graphmusters Bestandteil einer Ausprägung eines solchen Pfades sein. Dies kann die in Abschnitt 4.4.1 beschriebenen Konsequenzen haben, weshalb für jeden Pfad alle möglichen Teilausprägungen in einem

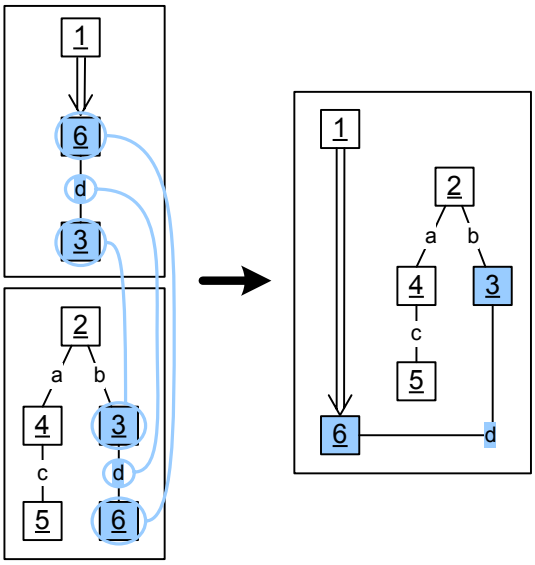

Abbildung A.2: Beispiel für das Verkleben von Graphmustern ohne Abbildung von Pfaden

neuen Graphmuster bestimmt werden müssen. Dabei wird berücksichtigt, dass Graphmuster nur Ausschnitte und keine vollständigen Graphen beschreiben.

Für jeden Pfad werden:

1. alle Knoten im neuen Graphmuster bestimmt, die Bestandteil einer Ausprägung des Pfades sein könnten. Dies kann für einen Knoten bestimmt werden, indem im Strukturmodell überprüft wird, ob es einen Weg vom Typ des Startknotens zum Typ des Knotens und von diesem zum Typ des Zielknotens des Pfades gibt, wobei nur pfadnavigierbare Assoziationen traversiert werden. Für jeden so bestimmten Knoten wird eine Teilausprägung angelegt, die zunächst nur aus dem Knoten besteht. Alle Teilausprägungen werden in einer Menge festgehalten.

2. werden alle Teilausprägungen in Vorwärtsrichtung erweitert. Für jede Teilausprägung werden alle mit dem aktuell letzten Knoten direkt verbundenen Kanten und Knoten im neuen Graphmuster bestimmt, die ebenfalls auf dem Pfad liegen können. Trifft das für eine Kante und einen Knoten zu, werden sie an eine Kopie der aktuellen Teilausprägung angehängt, die der Menge aller Teilausprägungen hinzugefügt wird. Müssen die Kante und der Knoten aufgrund von Kardinalitätseigenschaften zwin-

gend Bestandteil der Teilausprägung sein, wird die ursprüngliche Teilausprägung verworfen. Dieser Schritt wird solange wiederholt, bis keine Teilausprägung mehr vorwärts ergänzt werden kann und die Menge der Teilausprägungen stabil ist.

3. werden alle Teilausprägungen analog in Rückwärtsrichtung erweitert, bis die Menge stabil ist. Dabei werden in jeder Teilausprägung immer der aktuell erste Knoten und die mit ihm direkt verbundenen Kanten und Knoten betrachtet.

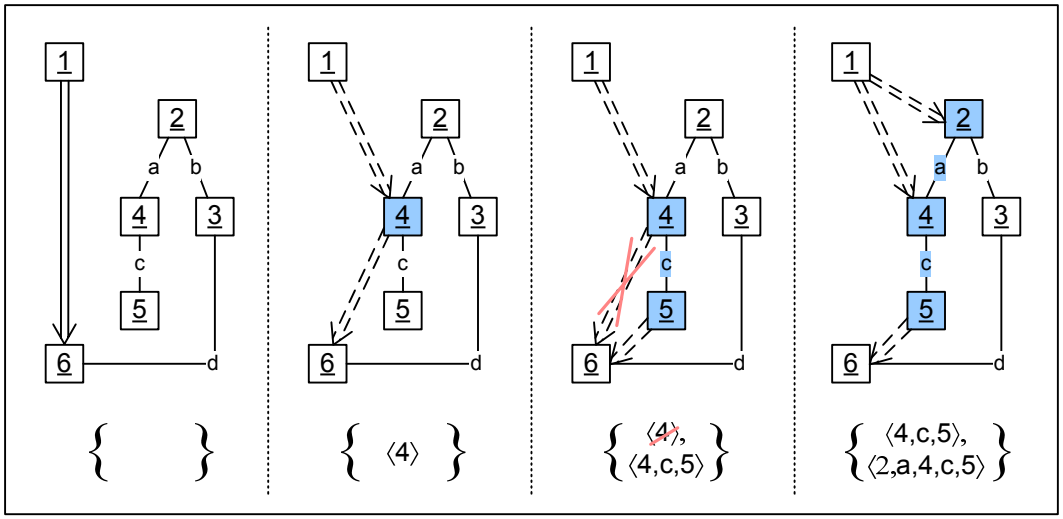

Abbildung A.3: Veranschaulichung der Bestimmung von Teilausprägungen

Abbildung A.3 veranschaulicht dieses Vorgehen anhand eines Beispiels für die Bestimmung von Teilausprägungen für den Pfad zwischen den Knoten 1 und 6. Das Beispiel zeigt die Ausgangssituation und drei Schritte des zuvor beschriebenen Verfahrens, wobei jeweils das Graphmuster und die aktuelle Menge an Teilausprägungen gezeigt werden.

Im ersten Schritt wird Knoten 4 als Knoten erkannt, der auf einem Pfad von 1 nach 6 liegen kann, da es wie gestrichelt angedeutet einen Pfad von 1 nach 4 und von 4 nach 6 geben kann. Daher wird für den Knoten eine initiale Teilausprägung festgehalten. Tatsächlich können auch die Knoten 2 und 5 auf dem Pfad liegen. Dies wird im Beispiel aber nicht betrachtet.

Im zweiten Schritt wird die Teilausprägung in Vorwärtsrichtung erweitert, da c und 5 auf dem Pfad 4 ⇒ 6 liegen können. Es wird eine neue Teilausprägung

$\langle 4, c, 5 \rangle$ festgehalten. Tatsächlich müssen die Elemente sogar auf dem Pfad liegen, da gemäß Strukturmodell 4 nur mit einem Knoten vom Typ 5 verbunden sein kann und es keinen Weg von 4 nach 6 gibt, der nicht über 5 führt. Daher wird die ursprüngliche Teilausprägung $\langle 4 \rangle$ verworfen.

Da die neue Teilausprägung vorwärts nicht mehr erweitert werden kann, wird sie im dritten Schritt rückwärts erweitert. Die Elemente 2 und a können ebenfalls auf dem Pfad liegen, so dass eine neue Teilausprägung $\langle 2, a, 4, c, 5 \rangle$ erstellt wird.

Gemäß Strukturmodell kann ein Knoten vom Typ 1 mit beliebig vielen Knoten vom Typ 2 verbunden sein. Ebenso kann ein Knoten vom Typ 4 mit beliebig vielen Knoten vom Typ 2 verbunden sein. Daher kann es auch eine Pfadausprägung zwischen 1 und 4 im Beispiel geben, zu der 2 und a nicht gehören. Daher wird die erste Teilausprägung beibehalten.

Ein Knoten vom Typ 2 kann nur mit einem Knoten vom Typ 4 verbunden sein, wie es im Beispiel der Fall ist. Damit kann es keine Pfadausprägung von 2 zu 6 geben, die a und 4 nicht beinhaltet.

Im Beispiel können keine anderen Teilausprägungen mehr ermittelt werden (auch nicht wenn von den Knoten 2 und 5 gestartet wird). Mit Hilfe der zwei Teilausprägungen werden die zwei in Abbildung A.4 gezeigten Graphmuster hergestellt.

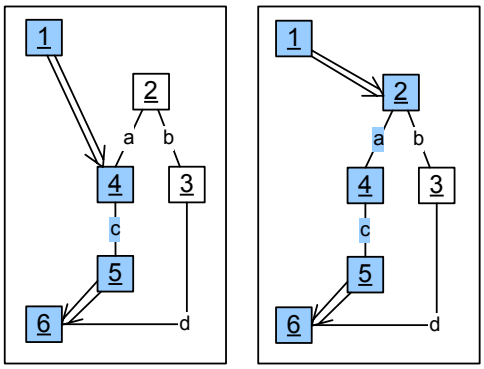

Abbildung A.4: Graphmuster mit abgebildeten Pfaden

Im Allgemeinen muss jeder weitere Pfad des ursprünglichen Graphmusters ebenso auf Teilausprägungen abgebildet werden. Danach müssen alle Kombinationen von Teilausprägungen unterschiedlicher Pfade in eigenen Graphmustern gebildet werden.

Abbildungen

Tabellen