Eine Sprache für die Spezifikation disziplinübergreifender Änderungsausbreitungsregeln

Bachelorarbeit von

Martin Löper

an der Fakultät für Informatik
Institut für Programmstrukturen und Datenorganisation (IPD)

Erstgutachter:	Prof. Dr. Ralf H. Reussner
Zweitgutachter:	Jun.-Prof. Dr.-Ing. Anne Koziolek
Betreuender Mitarbeiter:	Dipl.-Inform. Kiana Busch
Zweiter betreuender Mitarbeiter:	M.Sc. Dominik Werle

1. Juni 2018 – 30. November 2018

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Änderungen entnommen wurde.

Karlsruhe, 1. Juni 2018

. . . . . .

(Martin Löper)

Zusammenfassung

In der Änderungsausbreitungsanalyse wird untersucht, wie sich Änderungen in Systemen ausbreiten. Dazu werden unter anderem Algorithmen entwickelt, die identifizieren, welche Elemente in einem System von einer Änderung betroffen sind. Für die Anpassung bestehender Algorithmen existiert keine spezielle Sprache, weshalb Domänenexperten universelle Programmiersprachen, wie Java, verwenden müssen, um Änderungsausbreitungen zu formulieren. Durch den imperativen Charakter von Java, benötigen Domänenexperten mehr Code und mehr Wissen über Implementierungsdetails, als sie mit einer, auf die Änderungsausbreitungsanalyse zugeschnittenen, Sprache bräuchten. Eine Sprache sollte stets an den Algorithmus der jeweiligen Änderungsausbreitungsanalyse angepasst sein. Für den in dieser Arbeit betrachteten Ansatz zur Änderungsausbreitungsanalyse mit der Bezeichnung Karlsruhe Architectural Maintainability Prediction (KAMP), besteht noch keine spezielle Sprache. KAMP ist ein Ansatz zur Bewertung architekturbasierter Änderungsanfragen, der in einem gleichnamigen Softwarewerkzeug implementiert ist. Diese Arbeit präsentiert mit der Change Propagation Rule Language (CPRL) eine spezielle Sprache für den, in KAMP verwendeten, Algorithmus der Änderungsausbreitungsanalyse. Zum Abschluss wird der Vorteil der entwickelten Sprache, gegenüber drei konkurrierenden Sprachen, ermittelt. Die Arbeit kommt zum Schluss, dass CPRL kompakter als konkurrierende Sprachen ist und es gleichzeitig erlaubt die Mehrheit an denkbaren Änderungsausbreitungen zu beschreiben.

Inhaltsverzeichnis

Zusammenfassung i

1 Einleitung 1
- 1.1 Motivation . . . 1
- 1.2 Zielsetzung der Arbeit . . . 2
- 1.3 Struktur der Arbeit . . . 2

2 Grundlagen 5
- 2.1 Modellgetriebene Softwareentwicklung . . . 5
  - 2.1.1 Allgemein . . . 5
  - 2.1.2 Konzepte von Modellierungssprachen . . . 6
  - 2.1.3 Domänenspezifische Sprachen . . . 8
- 2.2 Entwicklungswerkzeuge . . . 9
  - 2.2.1 Java und Eclipse . . . 10
  - 2.2.2 Eclipse Modeling Framework (EMF) und Ecore . . . 11
  - 2.2.3 Xtext . . . 14
  - 2.2.4 Xtend . . . 15
  - 2.2.5 Vitruvius . . . 15
  - 2.2.6 Palladio Komponentenmodell . . . 15
  - 2.2.7 Weitere Werkzeuge . . . 15
- 2.3 Änderungsausbreitung und Bewertung von Änderungsanfragen . . . 16
  - 2.3.1 Auswirkungsanalyse und Änderungsausbreitungsanalyse . . . 16
  - 2.3.2 Karlsruhe Architectural Maintainability Prediction (KAMP) . . . 18
  - 2.3.3 Algorithmus zur Änderungsausbreitungsanalyse in KAMP . . . 19
  - 2.3.4 Abweichende Terminologie . . . 20
- 2.4 Change Propagation Rule Language (CPRL) . . . 21
- 2.5 Weitere deklarative Sprachen . . . 21
  - 2.5.1 VIATRA Query Language (VQL) . . . 22
  - 2.5.2 Object Constraint Language (OCL) . . . 22
- 2.6 Xtext-Grammatiksprache . . . 23

3 Verwandte Arbeiten 25

4 Erweiterbarkeit von KAMP 27
- 4.1 Erstellung neuer Module . . . 27
- 4.2 Probleme für den Domänenexperten bei der Modulerstellung . . . 29
- 4.3 Problemlösung mithilfe einer DSL . . . 30

5 Change Propagation Rule Language **31**
5.1 Einführung in die CPRL . . . 31
5.1.1 Entstehung . . . 31
5.1.2 Struktur der Sprache . . . 32
5.1.3 Basis-Sprachkonstrukte . . . 35
5.2 Regeldatei . . . 38
5.2.1 Aufbau einer Regeldatei . . . 38
5.2.2 Importieren externer Elemente . . . 38
5.2.3 Zugriff auf die Metamodellebene . . . 40
5.2.4 Zugriff auf die Modellebene . . . 42
5.2.5 Gruppierung von Regeln . . . 44
5.3 Regel . . . 46
5.3.1 Aufbau einer Regel . . . 47
5.3.2 Selektion der Eingabeelemente . . . 47
5.3.3 Definition der Änderungsausbreitung . . . 48
5.4 Regelquelle . . . 48
5.4.1 Aufbau einer Regelquelle . . . 49
5.4.2 Selektion basierend auf Metaklasse . . . 49
5.4.3 Selektion basierend auf Instanz . . . 50
5.4.4 Selektion der Ausgabe einer anderen Regel . . . 51
5.5 Abfrage . . . 52
5.5.1 Aufbau einer Abfrage . . . 52
5.5.2 Ermittlung anderer Elemente durch Navigation . . . 53
5.5.3 Filterung der Elemente durch Selektion . . . 54
5.5.4 Wiederverwendung bestehender Logik durch Regelaufruf . . . 58
5.6 Vorwärtsnavigation . . . 59
5.6.1 Veranschaulichung . . . 59
5.6.2 Aufbau der Vorwärtsnavigation . . . 60
5.6.3 Navigation basierend auf Metaklasse . . . 61
5.6.4 Navigation basierend auf Strukturmerkmal . . . 63
5.6.5 Navigation basierend auf Instanz . . . 64
5.7 Rückwärtsnavigation . . . 66
5.7.1 Veranschaulichung . . . 67
5.7.2 Aufbau der Rückwärtsnavigation . . . 67
5.7.3 Navigation basierend auf Metaklasse . . . 68
5.7.4 Navigation basierend auf Metaklasse und Strukturmerkmal . . . 69
5.7.5 Navigation basierend auf Instanz . . . 71
5.8 Verursachungselementmarkierung . . . 72

6 Implementierung **77**
6.1 Struktur . . . 77
6.2 Einbindung in KAMP . . . 78
6.3 Komplexitätsübergang . . . 79
6.4 Installation . . . 84

7 Evaluation **87**
7.1 Vollständigkeit . . . 87
7.2 Korrektheit . . . 89
7.3 Anwendbarkeit . . . 89
7.3.1 Abdeckung von KAMP . . . 89
7.3.2 Fehlende Funktionalität . . . 90
7.4 Vorteil . . . 92
7.4.1 Goal Question Metric (GQM) . . . 92
7.4.2 Präzisierung der LLOC-Metrik . . . 93
7.4.3 Modell für die Evaluation . . . 94
7.4.4 Quantifizierung der LLOC-Metrik . . . 95
7.4.5 Ergebnis . . . 107

8 Zusammenfassung und Ausblick **109**

9 Anhang **111**
9.1 Einbindung von OCL-Regeln in KAMP . . . 111
9.2 Einbindung von Xtend-Regeln in KAMP . . . 111
9.3 Einbindung von VIATRA-Regeln in KAMP . . . 111
9.4 Regulärer Ausdruck für CPRL . . . 115

Literatur **117**

Listings

2.1 Beispiel eines Querverweises 24

5.1 Basis-Sprachkonstrukte in CPRL 37
5.2 Beispiel der Farbgebung für CPRL-Listings in dieser Arbeit 37
5.3 Grammatik einer ⟨*Regeldatei*⟩ 38
5.4 Grammatik eines ⟨*RegeldateiKopfs*⟩ 40
5.5 Beispiel eines ⟨*RegeldateiKopfs*⟩ 40
5.6 Sprachelemente zur Referenzierung der Metamodellebene 41
5.7 Beispiel einer ⟨*MetaklasseReferenz*⟩ 42
5.8 Grammatik einer ⟨*InstanzDeklaration*⟩ 43
5.9 Beispiel einer ⟨*InstanzPrädikatDeklaration*⟩ 44
5.10 Beispiel einer ⟨*InstanzIdDeklaration*⟩ 44
5.11 Grammatik für das Sprachelement ⟨*Block*⟩ 45
5.12 Beispiel für ⟨*StandardBlock*⟩ und ⟨*RekursivenBlock*⟩ 46
5.13 Grammatik einer ⟨*Regel*⟩ 47
5.14 Beispiel einer ⟨*Regel*⟩ 47
5.15 Beispiel einer ⟨*RegelQuelle*⟩, die alle Elemente der temporären Ergebnismenge selektiert 49
5.16 Grammatik einer ⟨*RegelQuelle*⟩ 49
5.17 Grammatik einer ⟨*RegelQuelle*⟩, die nach Metaklasse filtert 50
5.18 Beispiel einer ⟨*RegelQuelle*⟩, die nach Metaklasse filtert 50
5.19 Grammatik einer ⟨*RegelQuelle*⟩, die nach Instanz filtert 50
5.20 Beispiel einer ⟨*RegelQuelle*⟩, die nach Instanz filtert 51
5.21 Grammatik einer ⟨*RegelQuelle*⟩ für das Aufrufen einer externen Regel . . 51
5.22 Beispiel für mehrere ⟨*RegelQuellen*⟩, die eine andere Regel aufrufen . . . 52
5.23 Grammatik einer ⟨*Abfrage*⟩ 53
5.24 Grammatik einer ⟨*Navigation*⟩ 54
5.25 Grammatik einer ⟨*Selektion*⟩ 56
5.26 Beispiel von ⟨*TypSelektionen*⟩ 57
5.27 Beispiel von ⟨*Selektionen*⟩ auf Modellebene 58
5.28 Grammatik für einen ⟨*RegelAufruf*⟩ 58
5.29 Beispiel für einen ⟨*RegelAufruf*⟩ 59
5.30 Grammatik einer ⟨*Vorwärtsnavigation*⟩ 61
5.31 Grammatik einer ⟨*Vorwärtsnavigation*⟩ auf Basis der Metaklasse 61
5.32 Beispiel von ⟨*Vorwärtsnavigationen*⟩ basierend auf Metaklasse 62
5.33 Grammatik einer ⟨*Vorwärtsnavigation*⟩ auf Basis des Strukturmerkmals . 63
5.34 Beispiel von ⟨*Vorwärtsnavigationen*⟩ basierend auf Strukturmerkmal . . . 64

5.35 Grammatik einer ⟨*Vorwärtsnavigation*⟩ auf Basis einer Instanz 65
5.36 Beispiel von ⟨*Vorwärtsnavigationen*⟩ basierend auf Instanz 66
5.37 Grammatik einer ⟨*Rückwärtsnavigation*⟩ 68
5.38 Grammatik einer ⟨*Rückwärtsnavigation*⟩ auf Basis der Metaklasse 68
5.39 Beispiel von ⟨*Rückwärtsnavigationen*⟩ basierend auf Metaklasse 69
5.40 Grammatik einer ⟨*Rückwärtsnavigation*⟩ auf Basis von Metaklasse und Strukturmerkmal 70
5.41 Beispiel von ⟨*Rückwärtsnavigationen*⟩ basierend auf Metaklasse und Strukturmerkmal 71
5.42 Grammatik einer ⟨*Rückwärtsnavigation*⟩ auf Basis einer Instanz 72
5.43 Beispiel von ⟨*Rückwärtsnavigationen*⟩ basierend auf Instanz 72
5.44 Grammatik einer ⟨*Verursachungselementmarkierung*⟩ 75
5.45 Beispiel von ⟨*Verursachungselementmarkierungen*⟩ 76

6.1 Beispiel einer Methode, die den Abhängigkeitsgraph der Regeln anzeigen lässt 82
6.2 Beispiel einer benutzerdefinierten Java-Regel 83

7.1 Beispiel für die Einschränkung in Abfragen 88

9.1 Einbindung von OCL-Regeln mittels Java 112
9.2 Einbindung von Xtend-Regeln - Version A 113
9.3 Einbindung von Xtend-Regeln - Version B 113
9.4 Einbindung von Xtend-Regeln - Version C 113
9.5 Einbindung von Xtend-Regeln - Version D 114
9.6 VIATRA-Regeldatei 114
9.7 Regulärer Ausdruck für CPRL 116

Abbildungsverzeichnis

2.1 Begriffe und Zusammenhänge in MDSD [SVE+07] 6
2.2 Hierarchie der Ecore Komponenten [Ecl15] 12
2.3 Beziehungen, Attribute und Operationen der Ecore-Komponenten [Ecl15] 13
2.4 Das Interface EObject von Ecore [Ecl15] 14

5.1 Syntax der CPRL - Übersicht über zentrale Bestandteile 33
5.2 Syntax der CPRL - Detailansicht der Navigation 34
5.3 Beispiel zur Visualisierung der Vorwärtsnavigation 60
5.4 Beispiel zur Visualisierung der Rückwärtsnavigation 67
5.5 Markierung einer beliebigen ⟨*Navigation*⟩ einer Regel 74
5.6 Standardmarkierung der ersten ⟨*Navigation*⟩ einer Regel 74

6.1 Projekt-Explorer eines CPRL-Beispielprojekts mit der Bezeichnung Example 79
6.2 Wizard für die Erstellung einer Regelvorlage für eine benutzerdefinierte Regel 81
6.3 Abhängigkeitsgraph für die Dependency Injection der Regeln 82
6.4 Änderungsausbreitungsansicht 82
6.5 Startansicht des KAMP-Installers 85
6.6 Startansicht des KAMP-Installers im Expertenmodus 85

7.1 UML-Klassendiagramm für einen Metamodell-Ausschnitt aus dem PCM . 94
7.2 UML-Objektdiagramm eines Beispielmodells für die Evaluation des Vorteils von CPRL gegenüber konkurrierenden Sprachen 95

Tabellenverzeichnis

2.1 Beispiele weit verbreiteter DSLs und ihr Anwendungsgebiet [MHS05] . . 9

5.1 DSLs, die teilweise in CPRL eingebunden sind und welche Funktion sie erfüllen 32

7.1 Anzahl an LLOC für eine Regelquelle 96
7.2 Anzahl an LLOC für eine Selektion auf Modellebene 98
7.3 Anzahl an LLOC für eine Selektion auf Metamodellebene 99
7.4 Anzahl an LLOC für die verschiedenen Arten an Vorwärtsnavigation . . 101
7.5 Anzahl an LLOC für die verschiedenen Arten an Rückwärtsnavigation . 103
7.6 Anzahl an LLOC für eine Verursachungselementmarkierung 104
7.7 Anzahl an LLOC für eine Rekursion 107
7.8 Übersicht über die Anzahl LLOC pro Regelbestandteil und DSL 108

9.1 Bedeutung der Platzhalter aus dem regulären Ausdruck 115

1 Einleitung

1.1 Motivation

In der Änderungsausbreitungsanalyse werden unter anderem Änderungen in Systemen und deren Auswirkungen untersucht. Um eine Änderungsausbreitung computergestützt zu untersuchen, muss zuerst ein Domänenmodell eines Systems angefertigt werden. Dazu bedienen sich viele Ansätze der modellgetriebenen Softwareentwicklung. Kern dessen ist eine spezielle Metadaten-Architektur. Diese spezifiziert ein Meta-Metamodell, welches als Ausgangspunkt für die Modellierung von Systemen, in Form eines Architekturmodells, verwendet werden kann.

Am KIT wurde ein Ansatz entwickelt, der die Bewertung architekturbasierter Änderungsanfragen ermöglicht. Der Ansatz mit der Bezeichnung Karlsruhe Architectural Maintainability Prediction (KAMP), wurde durch ein gleichnamiges, modularisiertes Softwarewerkzeug umgesetzt. Er ermöglicht es, zwei unterschiedliche Änderungsanfragen in komponentenbasierten Systemen bezüglich dem Aufwand, eine bestimmte Änderungsanfrage umzusetzen, abzuwägen. Außerdem kann KAMP dazu verwendet werden, mehrere Änderungsanfragen zu betrachten und deren Auswirkungen zu vergleichen.

Die Entwicklung des KAMP-Frameworks wird von Programmierern in Java vorangetrieben. Für die Erstellung neuer Module und die Modellierung neuer Domänen, bedarf es allerdings eine Person, die Wissen über die zu modellierende Domäne besitzt. Diese Person wird als Domänenexperte bezeichnet. Es liegt auf der Hand, dass Entwickler und Domänenexperte nur selten in einer Person vereint sind. Wenn festgestellt werden soll, wie gut die Implementierung von KAMP erweiterbar ist, muss zwischen dem Aufwand für Entwickler und Domänenexperten unterschieden werden. Es wäre beispielsweise wünschenswert für Domänenexperten, dass sie Module erstellen können, ohne tiefe Kenntnisse in Java zu besitzen. Anders als für Entwickler, gehört das Verständnis von Java nicht zu den Voraussetzungen für die Tätigkeit von Domänenexperten. Ein Beispiel hierfür ist die Domäne der automatisierten Produktionsanlagen. Die dortigen Domänenexperten sind typischerweise Maschinenbauer, welche sich nur selten in Java auskennen.

Für einige Schritte der Modulerstellung existieren bereits Softwarelösungen, die Domänenexperten unterstützen. So ist es beispielsweise möglich, Modelle mithilfe von grafischen Komponenten des Eclipse Modeling Framework (EMF) zu erstellen. Ein Bereich, in dem es allerdings an Eingabehilfen mangelt, ist die Formulierung von Änderungsausbreitungen. Diese werden in KAMP durch Java-Code ausgedrückt. Das stellt eine hohe Hürde für Domänenexperten dar. Zum einen müssen sie viel imperativen Code in einer universellen Programmiersprache schreiben und sich zudem mit der KAMP-Implementierung auseinandersetzen. Erstrebenswert wäre somit eine plattform- und programmiersprachenunabhängige Formulierungsart.

Eine Lösung dieses Problems stellt die Verwendung domänenspezifischer Sprachen dar. Durch die Entwicklung einer domänenspezifischen Sprache ließe sich eine kompakte Möglichkeit für Domänenexperten erstellen, ihr Wissen mit Sprachmitteln zu formulieren, die ihrem Vokabular ähneln oder zumindest einfach erlernbar sind. Die einfache Erlernbarkeit resultiert daraus, dass eine Sprache, die lediglich auf die Änderungsausbreitungsanalyse zugeschnitten ist, nur die nötigsten Sprachelemente und -semantiken enthält. Sie ist viel kompakter als eine universelle Programmiersprache wie Java.

1.2 Zielsetzung der Arbeit

Der Beitrag dieser Arbeit ist das Konzept einer domänenspezifischen Sprache mit der Bezeichnung Change Propagation Rule Language (CPRL). Das Konzept wird zugleich in Form eines Machbarkeitsnachweises umgesetzt. Das Ziel dieser Arbeit ist es, einem Domänenexperten die Formulierung von Regeln zur Änderungsausbreitungsanalyse zu erleichtern. Um das Ziel zu erreichen, soll CPRL so wenige Sprachelemente wie nötig umfassen und gleichzeitig einen möglichst großen Anteil der bestehenden Java-Regeln abdecken. Es wird in dieser Arbeit somit eine Sprache entwickelt, für welche die Abwägung zwischen Mächtigkeit und Kompaktheit entscheidend ist. Als Abschluss wird eine Evaluation durchgeführt, welche aufzeigt, welches Verhältnis zwischen Mächtigkeit und Kompaktheit gewählt wurde. Dabei wird ermittelt, wie groß die Abdeckung bereits bestehender Java-Regeln durch CPRL ist und wie kompakt CPRL im Vergleich zu anderen Sprachen ist. Zusammengefasst, deckt CPRL 90% der bestehenden Java-Regeln ab und bietet für acht von elf Regelbestandteile kompaktere Sprachmittel als konkurrierende Sprachen. Die dabei betrachteten Regelbestandteile, welche ein Domänenexperte typischerweise verwendet, werden im Zuge dieser Arbeit bei der schrittweisen Konzeption von CPRL erarbeitet.

1.3 Struktur der Arbeit

Im Folgenden wird kurz die Gliederung der Arbeit und der Inhalt der einzelnen Kapitel beschrieben.
Kapitel 2 thematisiert die Grundlagen. Dies sind alle Softwarewerkzeuge und Standards, welche allgemein für Modellierung und darüber hinaus für die Entwicklung domänenspezifischer Sprachen, verwendet werden. Außerdem werden bestehende Ansätze zur Änderungsausbreitungsanalyse vorgestellt und ein Klassifikationsverfahren für diese beschrieben. In Kapitel 3 werden Sprachen vorgestellt, welche für die Änderungsausbreitungsanalyse entwickelt wurden oder sich anbieten, dafür eingesetzt zu werden. Es wird dabei auf die Unterschiede und Gemeinsamkeiten zu CPRL eingegangen. Kapitel 4 beschreibt die Art und Weise, in der KAMP erweitert werden kann. Dabei wird erläutert, welche Probleme des Domänenexperten bei der Modellierung neuer Domänen auftreten und durch domänenspezifische Sprachen gelöst werden könnten. Kapitel 5 erläutert die entwickelte Sprache CPRL und jedes ihrer Sprachbestandteile auf konzeptioneller Ebene. Kapitel 6 beschreibt kurz, wie der konzeptionelle Entwurf der Sprache in einem Machbar-

keitsnachweis umgesetzt wird. In Kapitel 7 wird die entwickelte Sprache evaluiert, wobei sie vordergründig mit konkurrierenden Sprachen verglichen wird, um einen messbaren Vorteil nachzuweisen. Den Abschluss bildet Kapitel 8 mit einer Zusammenfassung der Arbeit und einem Ausblick auf weitere Verbesserungsmöglichkeiten der entwickelten Sprache.

2 Grundlagen

Das folgende Kapitel enthält die Themen, welche für das Verständnis der Arbeit erforderlich sind. Dies beginnt in Abschnitt 2.1 mit der modellgetriebenen Softwareentwicklung und Modellierung im Allgemeinen. Das Thema ist besonders wichtig, da diese Arbeit sich mit Änderungsausbreitungen beschäftigt und für die Analyse einer Änderungsausbreitung zuerst ein Modell eines zu untersuchenden Systems angefertigt werden muss.

Abschnitt 2.2 befasst sich mit den Entwicklungswerkzeugen, die für Implementierungsarbeiten zum Einsatz kommen. Er enthält außerdem Informationen zu Abhängigkeiten der entwickelten Sprache, wie beispielsweise Modellen, die in Beispielen Verwendung finden.

In Abschnitt 2.3 wird beschrieben, welche Arten der Änderungsausbreitungsanalyse existieren. Außerdem werden die Begriffe in diesem Forschungsgebiet erläutert. Anschließend wird mit KAMP ein konkreter Ansatz zur Änderungsausbreitungsanalyse eingeführt, der für die entwickelte Sprache eine besondere Rolle spielt. Die entwickelte Sprache wird in die Änderungsausbreitungsanalyse von KAMP integriert, um die Anwendbarkeit der Sprache zu zeigen und letztendlich eine Evaluation durchzuführen.

Abschnitt 2.4 thematisiert die entwickelte Sprache und ihren Ursprung. Es werden zudem Eigenschaften der Sprache erläutert.

Im darauf folgenden Abschnitt 2.5 werden deklarative Sprachen beschrieben, die für die Evaluation der entwickelten Sprache herangezogen werden.

Der letzte Abschnitt 2.6 erläutert die Xtext-Grammatiksprache. Diese Sprache wird verwendet, um CPRL zu implementieren. Sie erleichtert die Erstellung von domänenspezifischen Sprachen.

2.1 Modellgetriebene Softwareentwicklung

Die modellgetriebene Softwareentwicklung (engl. model-driven software development, MDSD) ist ein moderner Ansatz zur Softwareentwicklung. Er basiert auf dem Einsatz von Modellen. Der erste Abschnitt 2.1.1 definiert den Modellbegriff und leitet anschließend über zu Softwareentwicklungsansätzen, die Modelle verwenden.

In Abschnitt 2.1.2 werden typische Konzepte der Modellierung erläutert und Zusammenhänge mit domänenspezifischen Sprachen erklärt.

Schließlich werden in Abschnitt 2.1.3 domänenspezifische Sprachen im Detail thematisiert. Es werden deren Eigenschaften und Vorteile erklärt. Zudem wird ein Beispiel mit einer Übersicht bekannter domänenspezifischer Sprachen gegeben.

2.1.1 Allgemein

Modelle besitzen in der allgemeinen Modelltheorie nach Stachowiak drei Eigenschaften:

Abbildungsmerkmal Sie sind „stets Modelle von etwas, nämlich Abbildungen, Repräsentationen natürlicher oder künstlicher Originale, die selbst wieder Modelle sein können“ [Sta73].

Verkürzungsmerkmal Sie „erfassen im allgemeinen nicht alle Attribute des durch sie repräsentierten Originals, sondern nur solche, die den jeweiligen Modellerschaffern und/oder Modellbenutzern relevant scheinen“ [Sta73].

Pragmatisches Merkmal Sie sind „ihren Originalen nicht per se eindeutig zugeordnet. Sie erfüllen ihre Ersetzungsfunktion a) für bestimmte [...] modellbenutzende Subjekte b) innerhalb bestimmter Zeitintervalle und c) unter Einschränkung auf bestimmte gedankliche oder tatsächliche Operationen“ [Sta73].

Zusammengefasst, ist ein Modell ein vereinfachtes Abbild der Wirklichkeit. In der Softwareentwicklung werden Modelle als Dokumentations- und Planungswerkzeug eingesetzt. Zu diesem Zweck wird beispielsweise die Unified Modeling Language (UML) verwendet. Wenn zwischen einer Softwareimplementierung und dem zugehörigen Modell lediglich eine gedankliche Verbindung besteht, wird dies präziser als „*modellbasierte* Softwareentwicklung“ bezeichnet [SVE+07].

Einen deutlich weitläufigeren Ansatz bietet in Abgrenzung dazu, die „*modellgetriebene* Softwareentwicklung“ [SVE+07]. Diese sieht Modelle als Bestandteil der Software vor. Diese Modelle sind abstrakt, formal und erhalten die Bedeutung von Programmcode. Dadurch ist es möglich, durch den Einsatz von Codegeneratoren, einen Großteil der letztendlichen Implementierung zu generieren und somit lauffähige Software zu erzeugen. Außerdem orientieren sich die Modelle bezüglich der verwendeten Ausdrucksmittel am Problemraum der jeweiligen Domäne. Dies ermöglicht eine hohe Kompaktheit. Zur Formalisierung der Modelle wird eine höhere, domänenspezifische Modellierungssprache benötigt.

2.1.2 Konzepte von Modellierungssprachen

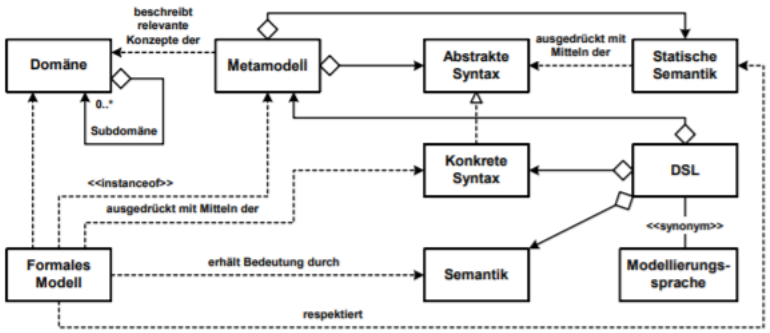

Abbildung 2.1: Begriffe und Zusammenhänge in MDSD [SVE+07]

Stahl u. a. betonen in ihrem Buch [SVE+07], dass viele der folgenden Begriffe zur Modellierung bereits lange existiert hätten, aber ein einheitlicher Gesamtkontext inklusive

Terminologie fehlte. Ein Standardisierungsprozess wurde durch die Object Management Group (OMG) gestartet. Die OMG ist ein 1989 gegründetes Konsortium, welches sich mit der Entwicklung von Standards in der objektorientierten Programmierung beschäftigt [Obj18]. Die OMG veröffentlichte mit der Meta Object Facility (MOF) eine Spezifikation für die Definitionen von Metamodellen und die Handhabung von Modellen im Allgemeinen [Obj16].

Stahl u. a. untersuchen in ihrem Buch [SVE+07] unterschiedliche Ansätze zur MDSD und identifizieren die folgenden Konzepte, sowie deren Beziehungen untereinander. Sie sind zudem in Abbildung 2.1 dargestellt.

Domäne Eine Domäne ist ein begrenztes Interessen- oder Wissensgebiet. Sie ist Ausgangspunkt der Modellierung. Eine Person, die sich mit der Struktur eines bestimmten Wissensgebiets - auch Ontologie genannt - auskennt, wird Domänenexperte genannt. *Bsp. einer Domäne: Geschäftsprozesse*

Subdomäne Eine Domäne kann sich aus Subdomänen zusammensetzen. *Bsp.: IEC-Standard 61131-3 als Subdomäne für Software von automatisierten Produktionssystemen*

Metamodell Ein Metamodell ist die Formalisierung der Domänenstruktur. Es umfasst die abstrakte Syntax und die statische Semantik einer Sprache (s.u.). *Bsp.: Ecore-Metamodell und MOF (siehe Abschnitt 2.2.2)*

Abstrakte Syntax Eine abstrakte Syntax spezifiziert die Struktur einer Sprache. Es ist damit die interne Darstellung von Strukturen gemeint, wie sie von einem Compiler oder Interpreter im Speicher gehalten werden. *Bsp.: Abstrakter Syntaxbaum (AST) der Sprache Java*

Konkrete Syntax Die konkrete Syntax ist die Realisierung einer abstrakten Syntax. Es wird dadurch spezifiziert, welche Eingaben ein Parser für eine bestimmte Sprache akzeptiert. Verschiedene konkrete Syntaxformen können eine gemeinsame abstrakte Syntax besitzen. Dadurch lässt sich ein und dasselbe Metamodell in verschiedenen Darstellungen abbilden (z.B. grafisch, UML-basiert oder textuell). *Bsp.: Syntax der Programmiersprache Java als eine mögliche Instanz des AST von Java*

Statische Semantik Die statische Semantik eines Metamodells ist eine Menge an Kriterien, die mit Mitteln der abstrakten Syntax formuliert wird. Diese Kriterien formulieren Anforderungen an die Wohlgeformtheit einer Eingabe. Fehler in der statischen Semantik können typischerweise durch den Compiler erkannt werden. Die Erkennung kann jedoch nicht durch den Parser durchgeführt werden, sondern durch die statische Analyse des Compilers. *Bsp.: Deklarationszwang von Variablen in der Programmiersprache Java*

Dynamische Semantik Die (dynamische) Semantik gibt den Konstrukten eines Metamodells eine Bedeutung. Ein Modellierer muss die Bedeutung der verfügbaren Konstrukte kennen, um sinnvolle Modelle erstellen zu können. *Bsp.: Bedeutung des Java-Konstruktes* `x=foo`*: Variablenzuweisung*

Domänenspezifische Sprache Eine domänenspezifische Sprache (engl. domain-specific language, DSL) hat zum Ziel, die wichtigsten Aspekte einer Domäne formal ausdrückbar zu machen. Eine DSL besitzt ein Metamodell (inklusive statischer Semantik), eine konkrete Syntax und eine dynamische Semantik. Der Begriff Modellierungssprache wird oft synonym zu DSL verwendet. *Bsp.: Apache Camel DSL [Apa18] - Zweck: Konfiguration von Pattern in der Unternehmensanwendungsintegration*

Formales Modell Ein formales Modell ist eine Instanz eines bestimmten Metamodells. Es steht dadurch im Zusammenhang zu einer bestimmten Domäne. Das formale Modell wird durch eine konkrete Syntax ausgedrückt und erhält die Bedeutung durch eine Semantik. Somit kann das formale Modell als ein "Satz" einer bestimmten DSL interpretiert werden. Das Modell sollte die statische Semantik des Metamodells respektieren, um als valide bezeichnet werden zu können. *Bsp.: Eine bestimmte Konfiguration, die anhand Apache Camel DSL formuliert ist*

2.1.3 Domänenspezifische Sprachen

Bereits im vorangehenden Abschnitt wurde die Rolle von domänenspezifischen Sprachen bei der Modellierung einer Domäne thematisiert. Mit der Fragestellung *wann* man sich für die Entwicklung einer neuen DSL entschließen und *wie* die Herangehensweise aussehen sollte, beschäftigen sich Mernik u. a. in einem Artikel [MHS05]. Der folgende Abschnitt bedient sich dieser Quelle und fasst die dort geschilderten Eigenschaften von DSLs zusammen:

- intern vs. extern:
  Eine interne DSL ist in eine Wirtssprache eingebettet. Das senkt den Implementierungsaufwand. Eine externe DSL ist hingegen eine von Grund auf neu definierte Sprache. Die konkrete Syntax und Semantik können dabei maximal an die Domäne angepasst werden, sodass externe DSLs meist kompakter sind als interne.

- imperativ vs. deklarativ:
  In einer imperativen Sprache definiert der Entwickler, welche Schritte der Computer zur Erfüllung einer Aufgabe ausführen muss. Es spezifiziert also *WIE* eine bestimmte Aufgabe gelöst werden soll. In Abgrenzung dazu, spezifiziert der Entwickler in einer deklarativen Sprache lediglich *WAS* für eine Aufgabe gelöst werden soll.

- visuell vs. textuell:
  Wie in Abschnitt 2.1.2 beschrieben, ist die Repräsentation der Sprache mit der konkreten Syntax gleichzusetzen. Meistens wird eine DSL in Textform definiert. Es sind aber auch visuelle Repräsentationen möglich.

- ausführbar vs. nicht ausführbar:
  Mernik u. a. beschreiben in ihrem Artikel [MHS05] die Grade, in der eine DSL ausführbar sein kann. Sie erwähnen ausdrücklich, dass eine DSL nicht ausführbar sein muss. Stattdessen kann eine DSL beispielsweise Eingabe eines Source-to-source Compilers - eines sogenannten Transpilers - sein, der sie in eine ausführbare Sprache übersetzt.

Weitere Konzepte zur Erstellung und Abgrenzung verschiedener DSLs liefert zudem Fowler in seinem Buch [Fow10]. Tabelle 2.1.3 listet Beispiele für DSLs auf, die weit verbreitet sind. Sie verdeutlicht nochmals, dass eine DSL sich stets auf eine bestimmte Domäne bezieht.

DSL	Domäne
Backus-Naur-Form (BNF)	Kontextfreie Grammatiken
Hypertext Markup Language (HTML)	Hypertext Webseiten
LaTeX	Typesetting
Make	Build Management
SQL	Datenbankabfragen
VHSIC Hardware Description Language (VHDL)	Hardwaredesign

Tabelle 2.1: Beispiele weit verbreiteter DSLs und ihr Anwendungsgebiet [MHS05]

Nun werden einige zentrale Vorteile der Verwendung einer DSL (im Kontext einer bestimmten Domäne) gegenüber einer universellen Programmiersprache betrachtet, die im Artikel [MHS05] von Mernik u. a. beschrieben sind:

Ausdruckskraft DSLs wägen Mächtigkeit (engl. power), bzw. Allgemeinheit (engl. generality) gegenüber Ausdruckskraft (engl. expressiveness), bzw. Kompaktheit (engl. conciseness) ab. Durch den Fokus auf eine bestimmte Domäne bieten sie meist verständlichere Konstrukte als universelle Programmiersprachen. Dies liegt auch daran, dass nicht alle Konzepte einer Domäne sich unmittelbar auf Funktionen und Objekte einer Bibliothek für universelle Programmiersprachen abbilden lassen.

Einfachheit DSLs müssen lediglich Bestandteile einer bestimmten Domäne abbilden. Somit können sie deutlich schlanker als universelle Programmiersprachen gehalten werden. Das ermöglicht es einer DSL, von Menschen ohne Programmiererfahrung eingesetzt zu werden, solange sie Wissen über die entsprechende Domäne besitzen und die Sprachkonstrukte ihnen intuitiv geläufig sind. Dafür müssen diese Personen lediglich die Semantik der DSL verstehen, da Hilfe bei der Syntax durch den Einsatz von Autovervollständigung bereitgestellt werden kann. Man spricht auch von erleichterter **Erlernbarkeit** von DSLs.

Wiederverwendbarkeit Die Verwendung von DSLs mindert die Redundanz in Softwaresystemen. Mernik u. a. identifizieren die folgenden Typen an Artefakten, die durch eine DSL wiederverwendet werden können: Syntax, Source Code, Software Designs und Domänenabstraktionen.

2.2 Entwicklungswerkzeuge

Für die Implementierung einer domänenspezifischen Sprache werden Programmiersprachen, Entwicklungswerkzeuge und externe Abhängigkeiten benötigt. Einige der folgenden

Technologien sind durch die Implementierung von KAMP faktisch gewissermaßen vorgegeben. So beispielsweise die Wahl der Programmiersprache Java und der Eclipse-Plattform, die in Abschnitt 2.2.1 beschrieben werden.

Ebenfalls von KAMP vorgegeben, ist das in Abschnitt 2.2.2 beschriebene, Framework zur Modellierung in Eclipse. Es beinhaltet mit Ecore ein Meta-Meta-Modell, das für die Implementierung von CPRL eine Rolle spielt. In CPRL werden Modellelemente und Metaklassen aus Ecore referenziert, um Änderungsausbreitungen auszudrücken.

In Abschnitt 2.2.3 wird mit Xtext ein Framework zur Erstellung von DSLs beschrieben. Es erleichtert die Implementierung von CPRL, indem es viele Bestandteile, wie einen Parser oder Elemente zur Interaktion mit Eclipse, generiert und vorkonfiguriert.

Anschließend wird die Sprache Xtend thematisiert. Das Framework Xtext macht von Xtend Gebrauch, indem es Code in Xtend generiert. Xtext lässt dem Entwickler einer DSL die Möglichkeit offen, Java zu verwenden. Es bietet allerdings Vorteile, Xtend anstelle von Java für die Implementierung zu verwenden. Diese sind in Abschnitt 2.2.4 beschrieben.

Abschnitt 2.2.5 beschreibt das Framework Vitruvius, aus welchem Quellcode für die Referenzierung von Metamodellen verwendet wird. Es handelt sich hierbei lediglich um eine implementierungsspezifische Abhängigkeit, die durch Copy-and-paste entfernt werden könnte.

In Abschnitt 2.2.6 wird mit dem PCM ein Metamodell beschrieben, das im Folgenden für sämtliche Beispiele in dieser Arbeit verwendet wird. Beispielregeln in CPRL werden in den folgenden Kapiteln mit Metaklassen aus dem PCM formuliert.

Abschließend werden in Abschnitt 2.2.7 einige Softwarewerkzeuge und Dienste beschrieben, die nicht unmittelbar mit der Entwicklung von CPRL zusammenhängen. Dazu gehört die Verwaltung von Quelltexten, die Erstellung von binären Komponenten und deren Auslieferung an Endbenutzer mittels Update-Seiten.

2.2.1 Java und Eclipse

Java ist eine objektorientierte, plattformunabhängige Programmiersprache. Es handelt sich dabei um eine Universalsprache (engl. General Purpose Language, GPL) [GJS+18]. Eine Universalsprache ist Turing-mächtig. Turing-Mächtigkeit ist die Eigenschaft einer Programmiersprache, sämtliche Funktionen berechnen zu können, die eine universelle Turingmaschine berechnen kann [Tur37]. Dies gilt unter der Annahme, sie könne auf unendlichen Speicherplatz zugreifen. Eine Universalsprache steht im Gegensatz zu den in Abschnitt 2.1.3 vorgestellten domänenspezifischen Sprachen.

Eclipse ist eine plattformunabhängige, integrierte Entwicklungsumgebung (engl. integrated development environment, IDE). Eine IDE unterstützt Programmierer bei der Durchführung wiederkehrender Aufgaben im Softwareentwicklungsprozess. Eclipse bietet darüber hinaus eine Rahmenarchitektur zur Erstellung neuer Software. Die Grundlage für die Erstellung neuer Software wird Eclipse-Plattform genannt. Diese bietet eine Menge an Frameworks und Integrationsmöglichkeiten, um eine erweiterbare, komponentenbasierte Architektur umzusetzen. Dazu zählt beispielsweise Equinox. Equinox ist ein Framework, welches eine Plugin-basierte Struktur ermöglicht. Es implementiert die Kernspezifikation der Open Services Gateway initiative (OSGi), heute: OSGi Alliance. OSGi spezifiziert eine

Softwareplattform, die es ermöglicht, Anwendungen und ihre Dienste per Komponentenmodell zu modularisieren [Ecl16].

Außerdem stellt Eclipse eine Vielzahl an quelloffenen Softwarepaketen zur Verfügung. Diese sind in Form von Plugins frei verfügbar und folgen der *Eclipse Platform API Specification* [Ecl17]. Einige dieser Softwarepakete sind in Java implementiert und werden unter dem Namen *Eclipse SDK* zum Download angeboten. Zusätzlich zur Eclipse-Plattform umfasst das SDK die folgenden Komponenten [Gui18]:

- Werkzeuge zur Java-Entwicklung: Java Development Tools (JDT)
- Umgebung zur Entwicklung von Eclipse-Plugins: Plug-in Development Environment (PDE)

Beide diese Komponenten werden bei der Integration von CPRL in KAMP verwendet. CPRL wird als eigenständiges Plugin für die Eclipse-Plattform entwickelt. Es wird zusammen mit den KAMP-Plugins ausgeliefert.

2.2.2 Eclipse Modeling Framework (EMF) und Ecore

Das Eclipse Modeling Framework (EMF) ist ein quelloffenes Framework für Modellierung. Es ist als Plugin in die Eclipse IDE integrierbar. Das zugrunde liegende Metamodell wird als Ecore bezeichnet [Gro18]. Ecore kann auf der Ebene der Meta-Meta-Modelle (M3-Ebene) angesiedelt werden, da es sich selbst beschreibt und als Ausgangspunkt weiterer Metamodelle dient. Ecore ist weitestgehend mit der Untermenge *Essential MOF* des MOF 2.0 Standards kompatibel.

In Abbildung 2.2 ist die Hierarchie des Ecore Meta-Meta-Modells zu sehen. Im Folgenden werden jene Elemente daraus beschrieben, die als Grundlage dieser Arbeit von Bedeutung sind. Als *EObject* wird ein Interface von Ecore bezeichnet. Jede Komponente in Ecore ist ein EObject. Des weiteren stellt Ecore eine Menge an Komponenten bereit, die das Interface *EModelElement* implementieren bzw. erweitern. Diese werden dazu verwendet, um Metamodelle zu beschreiben. Das Interface *ENamedElement* erweitert das EModelElement-Interface und stellt Zugriff auf das Attribut *name* zur Verfügung. Dieses Attribut, sowie Beziehungen der Komponenten untereinander, sind in Abbildung 2.3 aufgezeigt.

Eine EClass ist ein Interface, dass zur Definition von Metaklassen verwendet wird. Eine EClass kann verwendet werden, um eine Hierarchie an Metaklassen zu modellieren. Sie ist stets in einem *EPackage* enthalten und innerhalb dessen eindeutig durch ihr Name-Attribut identifizierbar. Die Identität des EPackages wird durch das Attribut *nsURI* bestimmt. Eine EClass enthält beliebig viele *EStructuralFeature*s, welche im Folgenden als *Strukturmerkmale* bezeichnet werden. Ein Strukturmerkmal ist entweder ein *EAttribute* oder eine *EReference*. Eine EReference ist ein Verweis auf eine konkrete Instanz einer Metaklasse. Ein Attribut ist ein Wert mit einem bestimmten Datentyp - in Ecore als *EDataType* bezeichnet. Da ein Strukturmerkmal das Interface *ETypedElement* erweitert, ist jedem Attribut und jeder Referenz eine Kardinalität und ein Typ zugeordnet. Der Typ entspricht der referenzierten Metaklasse bzw. dem referenzierten Datentyp. Die Kardinalität wird durch die Attribute *lowerBound*, *upperBound* und *many* bestimmt. Ein Datentyp erweitert ebenso wie die Metaklasse das Interface *EClassifier*. Ein EClassifier stellt eine Typ dar.

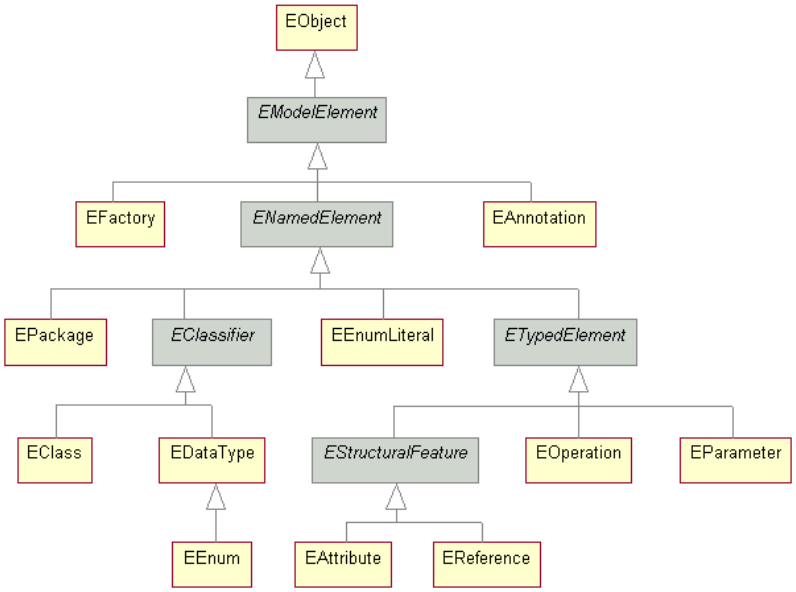

Abbildung 2.2: Hierarchie der Ecore Komponenten [Ecl15]

Allerdings sind Datentypen anders als Metaklassen nicht frei definierbar, sondern sind als eine Referenz auf Java-Datentypen und Java-Klassen zu verstehen. Eine Übersicht über die vorhandenen Datentypen findet sich in den Schaubildern der Eclipse Ecore Referenz [Ecl15]. Eine Spezialfall ist der Aufzählungstyp, der in Ecore als *EEnum* bezeichnet wird. Dieser ermöglicht es, einen eigenen Datentyp zu definieren, der aus einer endliche Wertemenge besteht. Alle zulässigen Werte des Aufzählungstyps werden durch Symbole dargestellt. Ein Symbol wird durch ein *EEnumLiteral* beschrieben.

Die Komponenten aus dem obigen Absatz werden allesamt bei der Metamodellierung verwendet. Bei der Erstellung von Modellen - dem einfachen Modellieren - werden konkrete Instanzen von Metaklassen angelegt. Diese Instanzen implementieren nicht das EModelElement-Interface, sondern lediglich das EObject-Interface. Da streng genommen die Modellierungskomponenten auch EObjects sind, wird in dieser Arbeit die Bezeichnung Instanz oder Modellobjekt für ein Objekt auf der Modellebene verwendet. Die Struktur eines EObjects ist in Abbildung 2.4 zu sehen. Ein EObject bietet über die Methode *eClass()* Zugriff auf die Metaklasse, deren Instanz es ist. Im Folgenden wird die Metaklasse, welche als eClass eines EObjects referenziert wird, als der Typ dieses EObjects bezeichnet. Weiterhin bietet ein EObject Zugriff auf ein Element oder eine Liste von Modellelementen, die den Strukturmerkmalen zugeordnet sind. In dieser Arbeit wird als **Inhalt eines Strukturmerkmals** die Menge an Modellelementen bezeichnet, die ein EObject durch ein bestimmtes Strukturmerkmal referenziert. Der Inhalt eines bestimmten Strukturmerkmals kann für jedes Modellelement anhand der Methode *eGet(feature: EStructuralFeature)* abgerufen werden. Der Inhalt aller Strukturmerkmale eines Modellelements kann durch die Methode *eAllContents()* ermittelt werden.

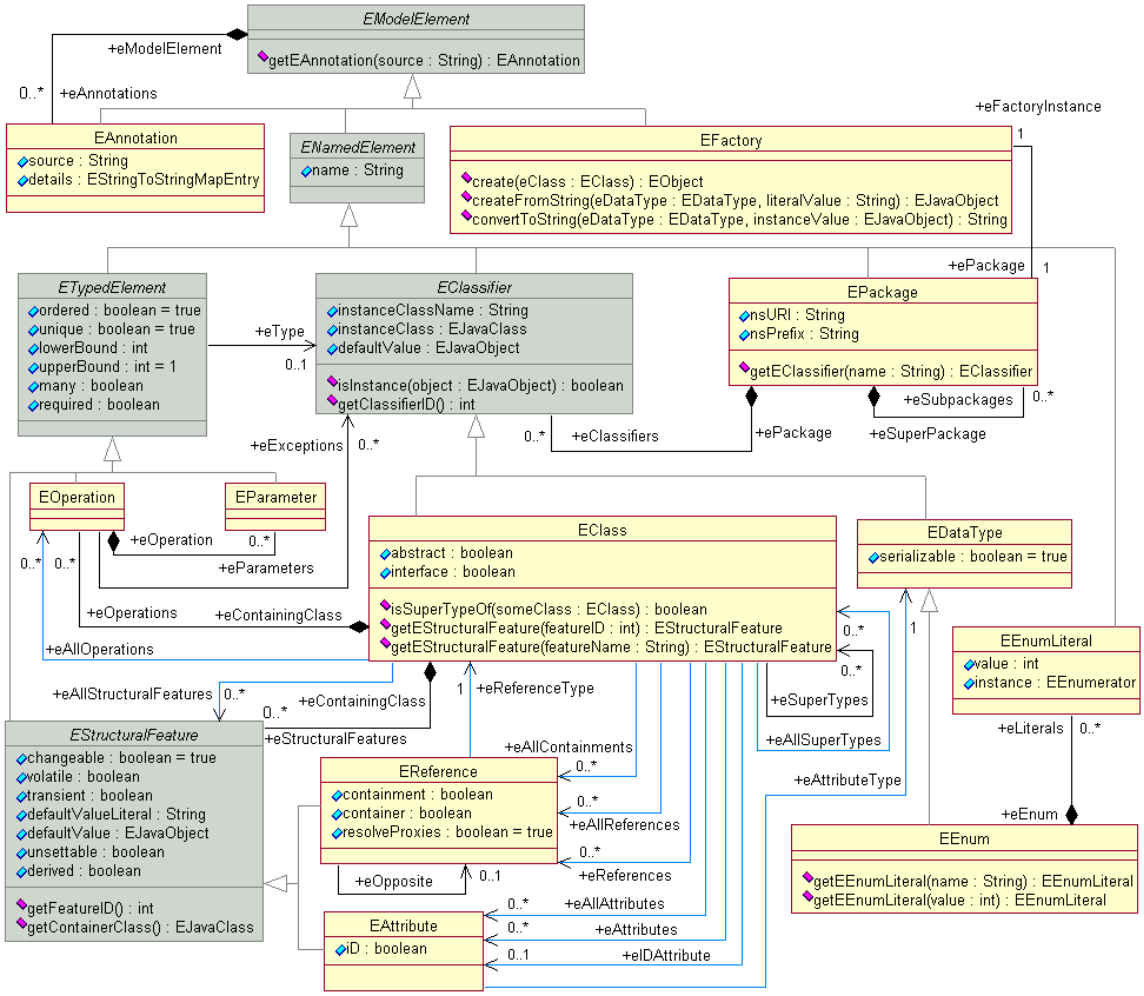

Abbildung 2.3: Beziehungen, Attribute und Operationen der Ecore-Komponenten [Ecl15]

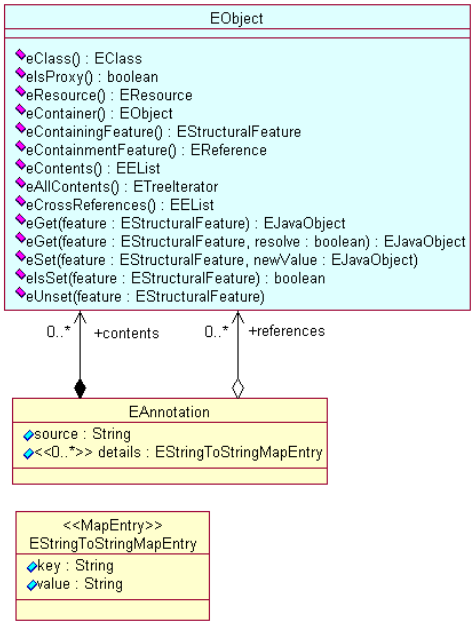

Abbildung 2.4: Das Interface EObject von Ecore [Ecl15]

2.2.3 Xtext

Xtext ist ein Open-Source-Framework zur Erstellung von Programmiersprachen und domänenspezifischen Sprachen. Es ist Teil von EMF [ES18e].

Xtext stellt eine eigene Grammatiksprache zur Verfügung, um DSLs zu beschreiben. Anhand der Grammatiksprache wird sowohl die konkrete, als auch die abstrakte Syntax einer DSL beschrieben. Dazu verwendet Xtext den Parsergenerator von ANTLR (ANother Tool for Language Recognition). Der Generator erstellt ein Ecore-Metamodell und einen ANTLR-Parser, welcher das Metamodell instanziiert. Außerdem stellt Xtext neben den oben genannten Compilerkomponeten noch Integrationen für Eclipse und andere IDEs bereit. Eine ausführliche Auflistung der Funktionalität, die Xtext den IDEs zur Verfügung stellt, befindet sich in der Xtext-Dokumentation [ES18a].

Zwei Technologien, die von Xtext intern verwendet werden und für das Verständnis dieser Arbeit von Bedeutung sein könnten, sind die **Modeling Workflow Engine 2 (MWE2)** und **Xbase**.

Bei MWE2 handelt es sich um eine Ausführungsumgebung, welche die Instanziierung, Konfiguration und Zusammenstellung von sog. *Workflow Komponenten* ermöglicht [Ecl13]. Damit kann die Generierung des Parsers und des Metamodells in Xtext konfiguriert werden. Außerdem kann festgelegt werden, welche Klassen- und Methodenrümpfe erzeugt werden sollen, um verschiedene Compiler- und IDE-Konzepte zu implementieren.

Werden bestimmte Konzepte implementiert, so werden die benutzerdefinierten Klassen anhand von Dependency Injection (DI) in Xtext geladen. Andernfalls werden Standardimplementierungen von Xtext verwendet. Xtext verwendet das DI-Framework Guice, welches 2008 erstmals von Google unter der Apache-Lizenz veröffentlicht wurde.

Xbase ist eine partielle Programmiersprache, die als Teil von Xtext implementiert ist. Sie wird auch als Ausdruckssprache (engl. expression language) bezeichnet. Konzeptionell und syntaktisch ist Xbase sehr stark an Java angelehnt. DSLs können Xbase erweitern oder verwenden, um eine Integration in das Java Typsystem zu erreichen und typische Ausdrücke in einer Java-ähnlichen Syntax zu integrieren. Dies kann zu einer erheblichen Zeitersparnis bei der Entwicklung von DSLs führen. Xbase wird zusammen mit der domänenspezifischen Sprache Xtype ausgeliefert und nutzt diese als Grundlage. Xtype stellt einige Java-ähnliche Datentypen bereit. Weitere Informationen dazu finden sich in der Xtext-Dokumentation im Kapitel über Xbase und Java Integration [ES18d].

2.2.4 Xtend

Xtend ist ein Dialekt der Programmiersprache Java. Die Sprachkonstrukte sind denen von Java deshalb sehr ähnlich. Dies ist darin begründet, dass Xtend als Grundlage Xbase verwendet. Der Xtend Sourcecode wird in Java Sourcecode transpiliert. Das Ziel von Xtend ist es, Schwächen von Java auszumerzen und die Produktivität von Entwicklern zu erhöhen. Xtend gilt als kompakter als Java, da es beispielsweise Typinferenz unterstützt. Eine detaillierte Auflistung der Unterschiede zu Java und der vollständigen Zielsetzung, finden sich in der zugehörigen Xtend-Dokumentation [ES18c].

2.2.5 Vitruvius

Vitruvius (kurz Vitruv) ist der Name eines Frameworks, das Mittel zur Konsistenzhaltung mehrerer Metamodelle bereitstellt. Vitruvius steht für *VIew-cenTRic engineering Using a VIrtual Underlying Single model*. Das Framework ist in mehrere Komponenten unterteilt, unter anderem Framework, Extensions, DSLs und Domains. Es wird in der quelloffenen Repository [vit18] auf GitHub bereitgestellt und entwickelt.

Die DSL-Komponente von Vitruvius enthält Java-Quelltext, der für die Implementierungen von CPRL wiederverwendet wird. Dazu gehört die Sprachfamilie MIR (engl. mappings, invariants and responses). Ein Bestandteil von MIR ist die MIR-Basissprache (engl. MIR-Base), welche Sprachkonstrukte zur Verfügung stellt, um Metamodelle und deren Inhalte zu referenzieren.

2.2.6 Palladio Komponentenmodell

Das Palladio Komponentenmodell (engl. Palladio component model, PCM) wird von Becker u. a. im Fachzeitschriftenartikel [BKR09] als ein Metamodell, das die Spezifikation von Performance-relevanten Informationen einer komponentenbasierten Architektur ermöglicht, bezeichnet. Besonders im Bereich der komponentenbasierten Informationssysteme bietet das PCM ein etabliertes Vokabular und eignet sich daher gut zu deren Modellierung.

2.2.7 Weitere Werkzeuge

Zur Entwicklung von CPRL wird die Eclipse IDE verwendet. Zur Versionsverwaltung kommt das Versionsverwaltungssystem **Git** zum Einsatz. Der entstehende Quelltext wird

von KAMP-Research, durch den Online-Dienst **GitHub**, in der Repository [KAM18] quelloffen verwaltet und bereitgestellt.

Es wird eine kontinuierliche Integration (engl. continuous integration, CI) aufgesetzt, welche automatisiert Plugins baut und über Eclipse-Update-Seiten für Eclipse-Nutzer zur Verfügung stellt. Um dies zu bewerkstelligen, werden folgende Technologien und Dienste verwendet:

Travis CI Ein Online-Dienst, der in GitHub integriert ist und Rechenleistung in der Cloud zur Verfügung stellt, um Softwareartefakte zu bauen. Ein neues Softwareartefakt wird gebaut, sobald eine Quelltextänderung am Hauptentwicklungsstrang, in der zugehörigen GitHub Repository, ausgelöst wird.

Maven Tycho Maven ist ein Build-Management-Werkzeug. Tycho ist eine Sammlung an Plugins für Maven, um Eclipse Plugins, Features, Update-Seiten und vieles mehr zusammenzubauen. Es berücksichtigt die Abhängigkeiten von Eclipse-Plugins und OSGi-Bundles, welche ein eigenes Metadaten-Format besitzen, das in sogenannten Manifest-Dateien ausgedrückt ist [Sie18].

Oomph Oomph ist ein Eclipse-Projekt, das Werkzeuge bereitstellt, um bestimmte Eclipse Konfigurationen zu formulieren. Das Projekt beinhaltet das sogenannten Oomph-Installationsprogramm, welches in der Lage ist, bestimmte Konfigurationen zu laden und somit reproduzierbare Eclipse-Umgebungen zu erzeugen [MS18]. Dieses Installationsprogramm wird verwendet, um die Software, die im Zuge dieser Arbeit entsteht, bei Endanwendern und Entwicklern zu installieren.

p2 Das p2-Projekt ist ein Unterprojekt von Equinox. Es bietet eine Infrastruktur zur Bereitstellung von Software. Insbesondere die Installation von Eclipse- und Equinox-basierten Applikationen wird unterstützt [RNL+]. Deshalb werden p2-Repositorys verwendet, um sämtliche Plugins im Eclipse-Umfeld zu verteilen. Die Eclipse-Update-Seiten referenzieren direkt bestehende p2-Repositorys.

CBI Aggregator Der CBI Aggregator wird für die Erstellung von aggregierten Eclipse-Update-Seiten verwendet. Plugins aus mehreren p2-Repositorys werden zusammengeführt und auf ihre Kompatibilität hin überprüft. Anschließend wird eine neue p2-Repository erstellt. Somit können alle Abhängigkeiten eines Eclipse-Plugin-Projekts aus einer einzigen Quelle installiert werden. Zudem kann das Layout verändert werden, indem beispielsweise die Kategorisierung angepasst wird [Ecl18a].

2.3 Änderungsausbreitung und Bewertung von Änderungsanfragen

2.3.1 Auswirkungsanalyse und Änderungsausbreitungsanalyse

Der Begriff Auswirkungsanalyse (engl. impact analysis, IA), wie er von Lehnert u.a. verwendet wird, bezeichnet Algorithmen zur Identifizierung, welche Elemente in einem

System von einer (potentiellen) Änderung betroffen sind [Leh11b; BA96]. Der Begriff Änderungsausbreitungsanalyse (engl. change propagation analysis) wird von Stammel u.a synonym verwendet [Sta17; CSE04]. Bei diesem Begriff wird hervorgehoben, dass eine Änderung an einem bestimmten Element, Änderungen an weiteren Elementen zur Folge hat. Die Änderung propagiert sich gewissermaßen durch das gesamte System. In dieser Arbeit wird der Begriff Änderungsausbreitungsanalyse statt Auswirkungsanalyse verwendet.

Dieser Begriff ist nicht beschränkt auf die Softwareentwicklung, sondern kann für beliebige Systeme verwendet werden. Allerdings hat sich herausgestellt, dass die Änderungsausbreitungsanalyse für Entwickler und Softwarearchitekten von überragender Bedeutung ist. Vor allem in der Phase der Wartung und Pflege bestehender Software werden Entwickler häufig mit den folgenden zwei Fragen konfrontiert, wenn sie eine Änderung vornehmen wollen:

1. Welche Entität muss verändert werden?
2. Welche weiteren Entitäten sind von der Änderung betroffen?

Es hat sich herausgestellt, dass die zweite Frage in großen Softwaresystemen manuell nicht mit ausreichender Genauigkeit beantwortet werden kann [LS98]. Es besteht somit ein Bedarf an automatisierten und halb-automatisierten Softwarelösungen.

Es existieren bereits zahlreiche Ansätze zur Änderungsausbreitungsanalyse, welche von Lehnert verglichen und kategorisiert wurden, um eine einheitliche Terminologie zu erreichen. In dieser Arbeit wird mit CPRL eine DSL für einen bestimmten Typ von Auswirkungsanalyse entwickelt. Um unseren Typ der Auswirkungsanalyse später einordnen zu können, werden im Folgenden die Klassifizierungskriterien aus Lehnerts Taxonomie [Leh11b] beschrieben:

Anwendungsbereich Auf was basiert der Ansatz? Es wird grob unterschieden in Code, formale Modelle und weitere Artefakte (wie z.B. Dokumentation). Die formalen Modelle werden feiner unterschieden in Architekturmodelle und Anforderungsmodelle.

Granularität Mit der Granularität der Entitäten sind folgenden Unterkategorien gemeint:

1. Wie präzise lässt sich das System abbilden?
2. Wie präzise lassen sich die Änderungsanfragen formulieren?
3. Welche Elemente werden als betroffen ermittelt?

Algorithmus Welcher Algorithmus wird verwendet um die betroffenen Elemente abzuleiten?

Analysetyp Wie interaktiv ist die Analyse? Wird i) das gesamte System analysiert (global) oder ii) die Möglichkeit geboten, Änderungsanfragen zu formulieren (Suche-basiert) oder iii) dem Benutzer betroffene Elemente vorgeschlagen (explorativ)

Tool-Support Besteht ein Tool oder Framework, das den Ansatz umsetzt?

Sprachen Welche Programmier- und Modellierungssprachen werden unterstützt?

Skalierbarkeit Wie wirkt sich die Größe des analysierten Systems auf die Parameter Laufzeit und Speicherbedarf aus?

Evaluierung Wurde der Ansatz durch experimentelle Resultate evaluiert?

Arnold und Bohner schreiben in ihrem Tagungsbandartikel [AB93], dass die meisten Ansätze zur Änderungsausbreitungsanalyse die folgenden Definitionen benutzen, um die Eingaben und Ausgaben ihrer Algorithmen zu benennen:

SIS Das Starting Impact Set (SIS) bezeichnet die Elemente, welche vom Entwickler als direkt modifiziert gekennzeichnet werden. Es stellt die Eingabe des Algorithmus dar.

EIS Das Estimated Impact Set (EIS) stellt die möglicherweise betroffenen Elemente dar. Es bezeichnet die Ausgabe des Algorithmus.

AIS Das Actual Impact Set (AIS) stellt die tatsächlich betroffenen Elemente dar.

Umso größer die Überschneidung der Mengen EIS und AIS, desto besser ist ein Ansatz. Es kommt aber laut Lehnert oft vor, dass die Ansätze die Auswirkungen überschätzen, sodass meist $|EIS| > |AIS|$ gilt. Es kommt außerdem vor, dass Elemente fälschlicherweise nicht berücksichtigt werden, sodass gilt: $(\exists a \in AIS : a \notin EIS)$ [Leh11b].

Eine Übersicht zu Ansätzen in der Änderungsausbreitungsanalyse ist in einem Review [Leh11a] und einer Taxonomie [Leh12] von Lehnert zu finden.

2.3.2 Karlsruhe Architectural Maintainability Prediction (KAMP)

Karlsruhe Architectural Maintainability Prediction (KAMP) ist ein Ansatz zur Bewertung architekturbasierter Änderungsanfragen. Er ermöglicht es, zwei unterschiedliche Änderungsanfragen in komponentenbasierten Systemen bezüglich dem Aufwand, eine bestimmte Änderungsanfrage umzusetzen, abzuwägen (Anwendungsfall 1). Außerdem kann KAMP dazu verwendet werden, mehrere Änderungsanfragen zu betrachten und deren Auswirkungen zu vergleichen (Anwendungsfall 2) [SR18].

Der Ansatz besteht aus einer formalen Definition, die von Stammel in Form seiner Dissertation [Sta17] verfasst wurde. Zugleich bezeichnet KAMP ein Softwarewerkzeug, das in Form von Plugins für die Eclipse IDE entwickelt wurde und den oben beschriebenen Ansatz umsetzt.

KAMP besitzt eine modulare Struktur und nutzt dazu die Modularisierungsmöglichkeiten, die Eclipse durch Equinox zur Verfügung stellt. Es werden explizite Architekturmodelle eingesetzt, um Systeme darzustellen. Als Grundlage dazu dienen Metamodelle für verschiedene Domänen. In der Domäne der Informationssysteme beispielsweise vordergründig das PCM.

Mit der Zeit wurde KAMP stetig um neue Metmodelle ergänzt, die in Form neuer Module hinzugefügt wurden. Dadurch lässt sich mit KAMP eine Vielzahl unterschiedlicher Systeme untersuchen. Zum jetzigen Zeitpunkt zählen dazu: Informationssysteme

(KAMP4IS), Geschäftsprozesse (KAMP4BP), Anforderungen (KAMP4REQ) und Automatisierte Produktionssysteme (KAMP4aPS). Besonders letztere unterstreichen die Stärke KAMPs, nicht alleine auf Softwaresysteme beschränkt zu sein, sondern stattdessen die Abbildung interdisziplinärer Sachverhalte über Domänengrenzen hinaus, zu ermöglichen.

Mit KAMP ist es möglich, die Wartbarkeit von Architekturen zu evaluieren. Dies geschieht in zwei Phasen [RSHR15; SR18]:

1. Änderungsanfragen werden formuliert. Anschließend wird eine Änderungsausbreitungsanalyse ausgeführt und ein Arbeitsplan mit einzelnen Arbeitspaketen abgeleitet (Top-down Phase).

2. Der Aufwand jedes einzelnen Arbeitspakets wird bestimmt (Bottom-up Phase).

Es ist nicht möglich, eine Architektur derart zu gestaltet, dass sich alle Änderungsanfragen mit dem gleichen Aufwand umsetzen lassen. Daher müssen zur Untersuchung diejenigen Änderungsanfragen ermittelt werden, die für die Zukunft am wahrscheinlichsten und am öftesten eintreten [SR18].

Der Vollständigkeit halber soll an dieser Stelle noch erwähnt sein, dass vor dem Beginn der obigen zwei Phasen zuerst ein Modell der zu untersuchenden Architektur angelegt werden muss. Eine Übersicht zu den Phasen ist in Abbildung 1 des Papiers [SR18] von Stammel und Reussner zu finden.

2.3.3 Algorithmus zur Änderungsausbreitungsanalyse in KAMP

Die in KAMP implementierte Änderungsausbreitungsanalyse wird wie folgt durch die in Abschnitt 2.3.1 genannten Kriterien klassifiziert:

Anwendungsbereich Formale Architekturmodelle und auch weitere Artefakte in Form von formalen Modellen (z.B. Entwicklungsteam-Struktur, Entwicklungsumgebung, Dokumentation, Laufzeitumgebung) [Sta17; SR18]

Granularität Die Granularität wird auf den folgenden drei Ebenen untersucht:

System Beliebige Elemente des zugrunde liegenden Metamodells können als Eingabe verwendet werden. Dies bietet eine sehr hohe Granularität. Diese geht weit über die von Lehnert in seiner Taxonomie [Leh11b] beschriebene Methodenebene hinaus.

Änderungsanfrage KAMP unterstützt die Modellierung expliziter Architekturveränderungen. Eine Unterscheidung nach Aktion (wie z.B. add oder remove) wird in der Ermittlung der Arbeitspläne vorgenommen.

Die Änderungsausbreitungsanalyse berücksichtigt allerdings die Aktion nicht. Sie unterstützt einen einzigen atomaren Typ von Änderungsanfrage. Elemente können lediglich als *betroffen* markiert werden.

Ergebnisse Jedes Element des zugrunde liegenden Metamodells kann durch den Algorithmus als *betroffen* markiert werden. Dies ermöglicht eine sehr hohe Granularität.

Algorithmus Die Änderungsausbreitung wird anhand von *expliziten Regeln* bestimmt. Die zugrunde liegenden Regeln basieren auf Domänenwissen. Dieser Algorithmus wird von Lehnert in seiner Taxonomie [Leh11b] als eine der zehn bekanntesten Techniken aufgeführt und unter anderem in den Ansätzen von Briand u. a. [BLO03], Feng und Maletic [FM06] und Vora [Vor10] verwendet.

Analysetyp Der Algorithmus von KAMP ist Suche-basiert. Es werden Änderungsanfragen gestellt und die Architektur wird lediglich auf diese Anfragen hin untersucht.

Tool-Support Die Ausbreitungsanalyse wurde in KAMP implementiert. Es existiert somit ein Softwarewerkzeug, das den Ansatz umsetzt.

Sprachen Die Änderungsausbreitungsanalyse, sowie alle anderen Komponenten von KAMP sind in Java implementiert. Die vordefinierten Modelle für Informationssysteme basieren auf dem PCM. Es können aber jegliche Ecore Metamodelle benutzt werden. Damit grenzt sich KAMP von Ansätzen ab, die lediglich UML-basiert sind.

Skalierbarkeit Es sind keine umfänglichen Informationen zur Skalierbarkeit vorhanden. Dies ist ein bekanntes Problem in der Domäne der Änderungsausbreitungsanalyse und bietet Möglichkeiten für zukünftige Arbeiten. Lehnert stellt in seiner Studie [Leh11a] fest, dass lediglich 10% der untersuchten Ansätze die Skalierbarkeit evaluieren. Für einzelne KAMP-Module sind Überlegungen zur Skalierbarkeit vorhanden. So zum Beispiel in dem Artikel [VHC+17] zu KAMP4aPS, der die Laufzeitkomplexität mit $O(n)$ angibt, wobei n für die Anzahl an Modellelementen steht.

Evaluierung Stammel führt in seiner Dissertation [Sta17] eine Evaluierung anhand einer empirischen Studie durch. Die Studie beschreibt die enthaltenen Aufgaben, liefert jedoch kein explizites *Größenmaß* wie von Lehnert gefordert. Die beiden Metriken *Trefferquote* (engl. recall) und *Genauigkeit* (engl. precision) wurden betrachtet. Bei der Gegenüberstellung wurde sich jedoch für das F1-Maß entschieden [Sta17]. Die Behandlungsgruppe weist mit jeweils 0,914 und 0,839 ein höheres F1-Maß auf als die Kontrollgruppe mit 0,850 und 0,416. Dadurch wurde belegt, dass mit KAMP mindestens genauso vollständige Ergebnisse erzielt werden können wie bei manueller Erstellung von Aktivitätslisten.

Stammel hebt in seiner Dissertation [Sta17] die Rolle der Änderungsausbreitung in KAMP hervor. Er fügt außerdem hinzu, dass es in komponentenbasierten Systemen wichtig ist, die Änderungsausbreitung in beiden Dimensionen zu untersuchen: zwischen Modulen (Inter-Modul-Ausbreitung) und innerhalb von Modulen (Intra-Modul-Ausbreitung).

2.3.4 Abweichende Terminologie

Die vorangehenden Abschnitte haben Begriffe aus der Domäne der Änderungsausbreitung eingeführt. Dieser Abschnitt geht ergänzend dazu auf abweichende Terminologie innerhalb von KAMP ein.

Die Elemente des SIS werden in KAMP als *seed modifications* bezeichnet. Im Folgenden wird dafür der deutsche Begriff *Anfangsmarkierung* verwendet. Die Elemente des EIS

werden in KAMP als *affected elements* bezeichnet. Dafür wird die deutsche Bezeichnung *betroffene Elemente* verwendet. Außerdem existiert in KAMP die Bezeichnung *causing entity*. Diese benennt ein Element, welches aus der Sicht eines betroffenen Elements dafür verantwortlich ist, dass es markiert wurde. Dafür wird die deutsche Bezeichnung *verursachendes Element* verwendet.

2.4 Change Propagation Rule Language (CPRL)

Die Change Propagation Rule Language (CPRL) - ehemals Kamp Rule Language (KARL) - ist eine domänenspezifische Sprache für die Änderungsausbreitungsanalyse. Sie ist speziell für den Algorithmus der Änderungsausbreitungsanalyse in KAMP ausgelegt, der in Abschnitt 2.3.2 beschrieben ist. Kern dieser Arbeit ist die Formalisierung und Weiterentwicklung von CPRL, sowie deren nahtlose Integration in KAMP.

Die CPRL wurde erstmals im Papier [BWLH] von Busch u. a. vorgestellt. Ergänzend dazu, werden im Folgenden die Eigenschaften der Sprache zusammengefasst. Es wird dabei auf die, in Abschnitt 2.1.3 genannten, Unterscheidungsmerkmale von DSLs eingegangen.

deklarativ Die Änderungsausbreitung in KAMP ist regelbasiert. Dies bedingt den deklarativen Charakter von CPRL. Der Benutzer spezifiziert lediglich zwischen welchen Elementen sich eine Änderung ausbreitet, aber nicht genau *wie* dies implementiert wird.

extern Es handelt sich bei CPRL um keine eingebettete Sprache, sondern eine von Grund auf neu definierte Sprache (externe DSL). Die Entscheidung für eine externe DSL wurde deshalb getroffen, da mit einer externen DSL eine höhere Kompaktheit als mit einer internen DSL erreicht werden kann. Eine hohe Kompaktheit ist eines der Hauptziele der zu entwickelnden Sprache.

textuell CPRL ist eine textuelle Sprache. Es werden typische Eingabehilfen zur Verfügung gestellt. Es existieren jedoch keinerlei visuelle Repräsentationen.

nicht ausführbar Die Sprache ist nicht ausführbar. Sie wird in die Programmiersprache Java transpiliert. Mittels spezieller Komponenten von Equinox kann der transpilierte Code zur Laufzeit in KAMP eingebunden werden.

Die Implementierung von CPRL erfolgt mit Xtext und der darin enthaltenen Xtext Grammatiksprache. Die Syntax und Semantik von CPRL wird in Abschnitt 5 erläutert.

2.5 Weitere deklarative Sprachen

Die folgenden Abschnitte führen zwei weitere deklarative DSLs ein, die als Vergleich zu CPRL in der Evaluation in Kapitel 7 herangezogen werden.

2.5.1 VIATRA Query Language (VQL)

Die VIATRA Query Language (VQL) ist eine deklarative DSL zur Formulierung von Abfragen an Ecore-Modelle mittels Graph-Pattern. Ein Graph-Pattern ist in VQL als n-Tupel formuliert. Jedes Element des Tupels wird als Parameter bezeichnet und stellt ein beliebiges Modellelement dar. Innerhalb des Patterns können mehrere Bedingungen (sogenannte *constraints*) definiert werden. Es gibt verschiedene Arten von Bedingungen. Diese beziehen sich entweder auf einen oder zwei Parameter. Eine Art von Bedingung ist beispielsweise die EClass-Bedingung *m1(p1)*. Diese legt fest, dass der Parameter p1 mit einem Modellelement mit der dazugehörigen Metaklasse m1 belegt sein muss. Eine weitere Art von Bedingung ist beispielsweise die Beziehungsbedingung *m1.r(p1, p2)*. Diese legt fest, dass der Parameter p1 mit einem Modellelement belegt sein muss, das über sein Strukturelement r jenes Modellelement referenziert, das Parameter p2 zugewiesen ist. Die semantische Bedeutung eines Patterns ist, dass es eine Menge an Tupeln darstellt, welche derart belegt sind, dass sie als Parameter alle Bedingungen erfüllen. Es wird somit eine Menge an Modellelementen beschrieben, die in einer bestimmten Beziehung zueinander stehen. Die Informationen in diesem Absatz stammen aus der offiziellen Dokumentation von VQL im VIATRA Eclipse Projekt [Ecl18c].

2.5.2 Object Constraint Language (OCL)

Die Object Constraint Language (OCL) ist eine deklarative DSL und dient vordergründig der textuellen Spezifikation von Ausdrücken für Bedingungen (sogenannten *constraints*) und Abfragen (sogenannten *object queries*) in UML-Modellen. Sie ist seit der UML-Version 1.1 Bestandteil der UML. Jeder Ausdruck in OCL kann folgende Bestandteile enthalten:

Kontext Der Kontext definiert, für welche Situation der Ausdruck gültig ist. Er wird nur in einer gültigen Situation evaluiert. *Bsp.: ‚context CompositeComponent' bedeutet, dass der Ausdruck sich nur auf Modellelemente vom Typ CompositeComponent bezieht.*

Eigenschaft Im Ausdruck kann eine Eigeschaft des Kontexts referenziert werden. Ist der Kontext beispielsweise eine Klasse, so kann als Eigenschaft ein Attribut dieser Klasse referenziert werden. *Bsp.: self.assemblyContexts__ComposedStructure beschreibt das Attribut assemblyContexts__ComposedStructure von self, wobei self sich auf den Kontext einer Klasse vom Typ CompositeComponent bezieht.*

Operation Es gibt unterschiedliche Operationen, die ein Modellelement oder eine Menge an Modellelementen manipuliert. *Bps.: mengenorientierte Operationen wie collect oder includes*

Schlüsselwort Ein Schlüsselwort wird verwendet, um Bedingungen zu formulieren. *Bsp.: if, then, else, and, or, not, implies*

Es existiert eine Implementierung von OCL in Eclipse mit dem Namen *Classic OCL* [Ecl07]. Diese unterstützt neben UML-Metamodellen auch Ecore-Metamodelle. Außerdem erlaubt sie die Evaluierung von OCL-Abfragen auf beliebigen Modellelementen. Sie kann somit verwendet werden, um eine Menge an Modellelementen zu ermitteln, die in einer

bestimmten Beziehung zueinander stehen. Die Informationen in diesem Absatz stammen aus der Spezifikation der Sprache [Obj14] durch die OMG.

2.6 Xtext-Grammatiksprache

Die konkrete Syntax von CPRL wird anhand der *Grammatiksprache von Xtext* definiert. Der folgende Abschnitt verwendet die offizielle Dokumentation [ES18b] von Xtext als Quelle.

Es handelt sich bei der Xtext-Grammatiksprache um eine Grammatiksprache, welche auf die Definition von textuellen domänenspezifischen Sprachen ausgelegt ist. Sie wird in Xtext verwendet, um den Quelltext zu generieren, der die Transformation einer CPRL-Eingabe in ein semantisches in-memory Modell durchführt. Dieses in-memory Modell wird auch als Abstrakter Syntaxbaum (engl. abstract syntax tree, AST) bezeichnet. Die Transformation eines Eingabestroms an Zeichen in einen AST geschieht in den folgenden vier Phasen:

Lexing Die erste Phase wird als Lexing bezeichnet. Hierbei wird der Eingabetext in eine Sequenz sogenannter *Token* zerlegt. Ein Token ist eine Zeichenkette, der von einer formalen Grammatik ein Typ zugewiesen wird. Es ist die lexikalische Grundeinheit für den Parser. Die Vorgehensweise beim Lexing wird in der Xtext-Grammatiksprache durch *Terminalregeln* und *Schlüsselwörter (engl. keywords)* definiert.

Parsing Die zweite Phase ist das Parsing. Dazu generiert Xtext mithilfe von ANTLR einen LL(*)-Parser. Dabei handelt es sich um einen Top-Down-Parser, der die Eingabe von links nach rechts abarbeitet und dabei einen endlosen Lookahead verwendet. Der Parser verarbeitet die Token aus Phase 1, indem er *Parserregeln* verwendet, die mit der Xtext-Grammatiksprache definiert sind. Aus den Parserregeln wird ein Ableitungsbaum gebildet, der aus Terminalsymbolen und Nichtterminalsymbolen besteht. Unter Verwendung eines Ableitungsbaums transformiert der Parser den Eingabestrom an Token in einen AST. Der AST besteht aus Java-Objekten, die auf Grundlage von generierten Ecore-Klassen instantiiert werden. Dabei erzeugt jede Anwendung einer Parserregel durch den Parser ein Objekt innerhalb des AST. Dem Objekt können mehrere Attribute zugeordnet werden, indem innerhalb von Parserregeln sogenannte *Zuweisungen* und *Querverweise* definiert werden. Die am häufigsten verwendete Zuweisung stellt das *name*-Attribut dar. Es wird zur eindeutigen Identifizierung eines Objekts verwendet und ermöglicht es, dass ein Objekt über einen Querverweis referenziert wird. Im Folgenden wird für ein Objekt im AST die weniger implementationsspezifische Bezeichnung *Element* verwendet. Als Typ des Elements wird in der Regel der Name der Parserregel verwendet. Die Xtext-Grammatiksprache bietet zudem die Möglichkeit, dass der Typ von Elementen, die durch eine Parserregel erstellt werden, geändert werden kann. Wenn von dieser Möglichkeit Gebrauch gemacht wurde, wird im Folgenden explizit darauf hingewiesen.

Linking Die Xtext-Grammatiksprache unterstützt die Definition von *Querverweisen*. Ein Querverweis wird durch eine Terminalregel definiert. Zusätzlich wird der Typ des

Elements angegeben, welches durch das angegebene Terminal referenziert wird. Querverweise werden in Xtext nach der Parsing-Phase durch den Linker aufgelöst. Ein Querverweis bildet das folgende Kriterium an die statische Semantik einer DSL: Das Terminal muss eine Zeichenkette darstellen, welche dem Name-Attribut eines beliebigen Objekts mit dem angegebenen Typ entspricht (siehe Beispiel in Listing 2.1). Es ist dabei unerheblich, ob das Objekt vor oder nach dem referenzierenden Objekt definiert wurde. Das liegt daran, dass der Linker Zugriff auf den bereits vollständigen AST hat. Elemente, die einen Querverweis darstellen, werden vom Parser anhand von einem sogenannten Stellvertreter-Objekt (engl. proxy) angelegt. Der Linker legt fest, wie diese Stellvertreter aufgelöst werden.

Validation Die letzte Phase stellt die Validierung des AST dar. Hierbei werden Methoden aufgerufen, welche die Einhaltung der statischen Semantik überprüfen. Die Methoden werden nicht in der Xtext-Grammatiksprache definiert, sondern durch den Entwickler in einer durch Xtext generierten Xtend-Klasse. Das Xtext-Framework identifiziert eine benutzerdefinierte Methode zur Validation anhand der Annotation *@Check*.

In der Arbeit wird die Bezeichnung *Sprachelement* für konkrete Terminalregeln und Parserregeln verwendet. Die Zeichenfolge, welche durch eine Terminalregel oder ein Schlüsselwort definiert ist, wird Terminal genannt.

```
// cross-referencing rule B
rule A: metaclass(ecore:EObject) → rule(B);

// rule named B
rule B: metaclass(ecore:EObject) → ...;
```

Listing 2.1: Beispiel eines Querverweises

Das Beispiel in Listing 2.1 beschreibt eine Regel mit dem Namen A und eine Regel mit dem Namen B. Die Regel A verweist auf eine Regel B mit dem Regelverweis *rule(B)*. Das Terminal *B* ist ein Querverweis, da es den Namen der darunterliegenden Regel B referenziert.

3 Verwandte Arbeiten

Der folgende Abschnitt geht auf Arbeiten ein, die analog zu dieser Arbeit, eine domänenspezifische Sprache für die Änderungsausbreitungsanalyse entwickeln. Ein Überblicks-Artikel zu unterschiedlichen Ansätzen der Änderungsausbreitungsanalyse ist das Review [Leh11a] von Lehnert.

Lehnert u. a. präsentieren mit dem Konferenzpapier [LFR13] einen Ansatz zur Änderungsausbreitungsanalyse, welcher dem in KAMP sehr ähnlich ist. Gemeinsamkeiten bestehen vor allem darin, dass die Autoren die Wichtigkeit erkannt haben, verschiedene Artefakte in die Analyse miteinzubeziehen. Analog zu KAMP unterstützt der Ansatz die Verwendung beliebiger EMF-basierter Modelle. Außerdem wird eine vertikale und horizontale, regelbasierte Änderungsanalyse durchgeführt. Im Konferenzpapier [LFR13] formulieren die Autoren eine zugehörige DSL. Unterschiede zu dieser Arbeit bestehen darin, dass die Sprache auf XML basiert und somit weniger ausdrucksstark wirkt. Weiterhin sind konzeptionelle Differenzen zu erkennen:

1. Die Regeln in CPRL werden iterativ ausgeführt. Jede Regel stellt dabei einen vollständigen Ausbreitungspfad mit mehreren Navigationen zwischen zwei einzelnen Modellelementen (sog. *Abfragen*) dar. Im Ansatz von Lehnert et al. besteht eine Regel aus einer einzigen Abfrage. Die Regeln werden außerdem alle rekursiv ausgeführt. Dies erfordert Maßnahmen zur Zyklen-Erkennung.

2. Regeln in KAMP werden ausgehend vom Typ der Elemente im SIS aktiviert. Die Regeln von Lehnert et al. verwenden sog. Abfragebedingungen (englisch query conditions), um aktiviert zu werden.

3. Lehnert et al. unterscheiden zwischen verschiedenen Arten von Änderungsanfragen. Wie in Abschnitt 2.3.2 beschrieben, unterscheidet KAMP nicht zwischen Arten von Änderungsanfragen in der Ausbreitungsanalyse. Elemente werden lediglich als *betroffen* markiert.

4. CPRL markiert alle Elemente, die durch die Ausführung einer Regel erfasst werden. Lehnert et al. lassen den Nutzer beliebig viele vordefinierte Aktionen auswählen, die von einer Regel ausgeführt werden.

Insgesamt kann zusammengefasst werden, dass der verglichene Ansatz viele Gemeinsamkeiten besitzt. Die Unterschiede sind auf leicht unterschiedliche Ansätze zur Änderungsausbreitungsanalyse zurückzuführen, für welche die jeweiligen Sprachen entwickelt wurden.

Ein weiterer Ansatz stammt aus dem Eclipse Projekt VIsual Automated model TRAnsformations (VIATRA), welches sich mit Modelltransformationen beschäftigt. Im Tagungsbandartikel [BV06] stellen Balogh und Varró mit der VIATRA Textual Command Language (VTCL) eine regelbasierte, textuelle DSL für Graphtransformationen vor. Diese Sprache wird bisher nicht in der Änderungsausbreitungsanalyse eingesetzt. Metamodelle und Änderungsausbreitungen lassen sich jedoch auch in Graphen ausdrücken, wie Stammel in seiner Dissertation [Sta17] festgestellt hat. Eine Weiterentwicklung der VTCL stellt die VQL dar, die in Kapitel 2.5.1 beschrieben wurde. Die VQL wurde nicht explizit für die Formulierung von Änderungsausbreitungen entworfen, sondern für Abfragen an Ecore-Modelle. Da CPRL auf Abfragen an Ecore-Modelle basiert, könnte VQL für die Implementierung dieser Unterfunktionalität verwendet werden. Da VQL allerdings sehr generell gehalten ist, besitzt sie eine deutlich größere Anzahl an Sprachelementen. Es wurde versucht, die Anzahl der Sprachelemente in CPRL möglichst klein zu halten, weswegen sich gegen eine Einbettung von VIATRA entschieden wurde.

Eine weitere DSL, welche verwendet werden kann, um Abfragen an ein Ecore-Modell zu stellen, ist die OCL. Bei dieser Sprache wurde sich aus den gleichen Gründen wie bei VIATRA gegen eine Einbettung entschieden.

Zum Abschluss wird der Ansatz von Müller und Rumpe betrachtet. In ihrem Tagungsbandartikel [MR14] stellen die Autoren eine regelbasierte DSL vor, welche auf die Ergebnisse der Änderungsausbreitungsanalyse reagiert. Mit dem Ansatz lassen sich automatisiert Checklisten erstellen. Der Unterschied zu KAMP besteht darin, dass die Entwickler im ersten Schritt manuell zwei verschiedene UML Modelle erstellen müssen. Die Änderungen und ihre Propagierung werden somit manuell modelliert. Anschließend wird mit EMF Compare die Differenz der beiden Modelle gebildet. Die Regelsprache definiert dann nur noch Regeln, die auf Differenzen der Modelle reagieren und trägt zur Generierung von Checklisten bei.

Die betrachtete Regelsprache umfasst also nicht die Änderungsausbreitung, sondern nur deren nachgelagerte Phase - die Verarbeitung der Ergebnisse. Der CPRL mangelt es bisher an Konstrukten, um die Ergebnisse weiterzuverarbeiten. Im folgenden Kapitel wird vorgeschlagen, eine separate DSL zur Verarbeitung der Ergebnisse zu entwickeln. Diese soll nicht Teil einer Sprache sein, die einzig und alleine Änderungsausbreitungen beschreibt.

4 Erweiterbarkeit von KAMP

Es existieren mehrere Werkzeuge, die es ermöglichen, Änderungsanfragen an Architekturmodelle zu stellen. Diese Arbeit betrachtet das in Abschnitt 2.3.2 vorgestellte KAMP. Es handelt sich um ein Ansatz zur Bewertung architekturbasierter Änderungsanfragen. In Abschnitt 4.1 wird erklärt, wie die Änderungsausbreitung in KAMP implementiert ist und wie sie durch Domänenexperten erweitert werden kann. Anschließend werden in Abschnitt 4.2 Probleme bei der Erweiterbarkeit thematisiert. Schließlich wird in Abschnitt 4.3 beschrieben, wie eine spezielle DSL die Erweiterbarkeit durch Domänenexperten erleichtern könnte.

4.1 Erstellung neuer Module

Der KAMP-Implementierung liegt eine modulare Software-Architektur zu Grunde. Das bedeutet, dass es möglich ist, die Software derart zu erweitern, dass sie neue Domänen unterstützt. Dazu muss ein neues Eclipse-Plugin erstellt und mit domänenspezifischen Informationen konfiguriert werden. Eine Sammlung domänenspezifischer Informationen und deren Einbindung in mehrere OSGi-Bundles, wird KAMP-Modul genannt. Ein KAMP-Modul benutzt Schnittstellen und abstrakte Klassen des sogenannten Kerns (engl. core). Der Kern stellt somit gemeinsam genutzte Funktionalität für mehrere Erweiterungen bereit. Folgende Elemente werden von Heinrich u. a. in ihrem Artikel [HBK18] erläutert und sind Teil eines domänenspezifischen KAMP-Moduls:

1. Strukturmodell für das System in der jeweiligen Domäne
2. nicht-strukturelle Modelle, die beispielsweise technische oder organisatorische Details abbilden - optional
3. Modell für Modifikationsmarkierungen (engl. modificationmarks model)
4. Änderungsausbreitungsregeln (für den Algorithmus der Änderungsausbreitungsanalyse)
5. Algorithmen zur Ableitung der Modell-Unterschiede und der daraus resultierenden Arbeitspakete (mittels Modell der Aufgabenarten) - optional

Ob die Punkte zwei und fünf berücksichtigt werden müssen, hängt von der jeweiligen Domäne ab. Sie sind daher in der Regel optional. Alle oben genannten Bestandteile müssen momentan von Entwicklern mittels Java ins KAMP-Framework eingebunden werden. Die folgenden fünf Schritte sind dafür nötig:

1. **Meta-Modellierung der Domäne anhand von Ecore**
   Die Metaklassen der Domäne müssen erstellt werden, damit Endbenutzer sie später bei der Modellierung verwenden können. Dabei sollte auf existierende Metamodelle zurückgegriffen werden. *Für Informationssysteme wird beispielsweise das etablierte PCM als Strukturmodell benutzt.*

2. **Meta-Modellierung von nicht-strukturellen Details anhand von Ecore**
   Die Elemente des Strukturmodells sollen später durch den Endbenutzer mit zusätzlichen Informationen angereichert werden können. Dazu müssen Metamodelle erstellt werden, deren Metaklassen nicht-strukturelle Sachverhalte abbilden. *Beispiel aus KAMP4IS: Das Modell fieldofactivityannotations stellt Elemente wie z.B. Rollen (organisatorisch), Dokumentationsartefakte (technisch) oder Deployment-Spezifikationen (örtlich) bereit. Diese können den einzelnen Modellelementen des Informationssystems zugeordnet werden. Diese Zuordnung spielt bei der Ableitung der Arbeitspakete für den Arbeitsplan eine Rolle.*

3. **Meta-Modellierung von Modifikationsmarkierungen anhand von Ecore**
   Das Modifikationsmarkierungen-Metamodell wird in den bestehenden KAMP-Modulen als *modificationmarks model* bezeichnet und erfüllt die folgenden drei Aufgaben:

   a) Es beinhaltet Modifikationsmarkierungen, die während der Änderungsausbreitungsanalyse verwendet werden können. Eine Markierung ermöglicht es dem Endbenutzer, ein Modellelement als geändert zu markieren, indem er es den Anfangsmarkierungen hinzufügt. Folglich können lediglich Modellelemente des Strukturmodells als Ausgangspunkt der Änderungsausbreitungsanalyse verwendet werden, für deren Metaklasse eine zugehörige Modifikationsmarkierung erstellt wurde.

   b) Lediglich Modellelemente, für deren Metaklasse eine zugehöriger Markierung besteht, können als betroffene Elemente und somit als Ergebnis der Änderungsausbreitungsanalyse verwendet werden. Die Markierung enthält neben dem betroffenen Element zusätzlich noch eine Liste an verursachenden Elementen. Diese Zuordnung ermöglicht es dem Endbenutzer nachzuvollziehen, weshalb ein bestimmtes Element markiert worden ist.

   c) Das Metamodell stellt Gruppierungen von betroffenen Elementen bereit (sog. *change propagation steps*). *Beispiel aus KAMP4IS: Die Gruppierungen ISIntercomponentPropagation und ISIntracomponentPropagation grenzen die Elemente voneinander ab, die durch Inter-Modul-Ausbreitung und Intra-Modul-Ausbreitung markiert wurden. Es handelt sich dabei folglich um eine semantische Gruppierung der betroffenen Elemente nach Art der Abhängigkeit von den Anfangselementen.*

   Das Modifikationsmarkierungen-Modell wird für die Änderungsausbreitungsanalyse benötigt und muss folglich von Domänenexperten angelegt werden.

4. **Erstellung von Algorithmen zur Änderungsausbreitung anhand von Java**

   a) Es müssen sogenannte *Lookup-Methoden* erstellt werden. Diese beschreiben, wie ausgehend von einem markierten Element, weitere Elemente markiert

werden sollen. Sie bilden die Logik der Änderungsausbreitung einer bestimmten Domäne ab. *Beispiel-Regel aus KAMP4IS: "Markiere alle Signaturen innerhalb eines markierten Interfaces."*

b) Es muss formuliert werden, welchen Gruppierungen die Markierungen betroffener Elemente zugeordnet werden sollen.

Für die Implementierung von Algorithmen zur Änderungsausbreitung in KAMP wird bis zum Zeitpunkt dieser Arbeit ausschließlich Java verwendet.

5. **Ableiten des Arbeitsplans anhand von Java**
Das Ergebnis der Änderungsausbreitungsanalyse wird benutzt, um den Arbeitsplan (engl. workplan) abzuleiten. Wie die betroffenen Elemente in Arbeitspakete umgewandelt werden sollen, wird mittels Java festgelegt. Zur Darstellung der einzelnen Arbeitspakete wird ein zusätzliches Aufgabenarten-Metamodell (engl. metamodel of task types) benötigt.

   Neben der Änderungsausbreitungsanalyse wird eine weitere Methode verwendet, um Arbeitspakete abzuleiten: **Explizite Strukturmodellveränderungen**.

   Im Stammordner des Projekts werden die originalen Modell-Dateien - die *Basisversion* des Modells - abgelegt. In dem Verzeichnis *modified* wird eine modifizierte Variante - die *Zielversion* des Modells - abgelegt. Die Zielversion entspricht standardmäßig der Basisversion. Bei der Modellierung bestimmter Änderungsanfragen fügt der Benutzer der Zielversion Modellelemente hinzu oder löscht welche heraus. Anschließend wird die Differenz aus Basis- und Zielversion der Modelle berechnet. Somit können neben der klassischen Art von Änderungsanfrage *Modifikation* auch andere Arten unterstützt werden, wie zum Beispiel das *Hinzufügen* oder *Entfernen*.

   Es muss bei der Modul-Erstellung somit a) ein Algorithmus zur Differenzberechnung für Modelle und b) ein Algorithmus zur Ableitung von Arbeitspaketen aus Differenzen angegeben werden. Dies wird mit Java umgesetzt. Zum Vergleich von Modellen wird das Eclipse Plugin EMF Compare [Ecl18b] verwendet.

4.2 Probleme für den Domänenexperten bei der Modulerstellung

Die fünf Schritte einer Modulerstellung in KAMP aus dem vorangehenden Abschnitt werden typischerweise von einem Domänenexperten durchgeführt. Bei einem Domänenexperten handelt es sich häufig nicht um einen Informatiker, sondern eine Person, die sich in der zu modellierenden Domäne auskennt. Es ist somit nicht zu erwarten, dass diese Person sich mit einer universellen Programmiersprache, wie Java, auskennt. Das kann zu den folgenden Problemen bei der Implementierung oder Anpassung von KAMP-Modulen durch Domänenexperten führen:

- Die Erstellung eines Moduls erfordert Wissen darüber, wie der Kern von KAMP implementiert ist. Die Domänenexperten müssen sich zumindest mit den abstrakten

Klassen auseinandersetzen, welche die Basis für neue Module bilden. Das stellt bei dem aktuellen Umfang an Quelltextzeilen von KAMP eine Hürde dar.

- Die Implementierung von Algorithmen zur Änderungsausbreitungsanalyse erfordert ein Verständnis der Programmiersprache Java.
- Die Erstellung von Änderungsausbreitungsregeln in Java ist zeitaufwändig, da sie imperativ formuliert wird. Eine Vielzahl von Änderungsausbreitungsregeln führen ähnliche Methodenaufrufe aus, sodass die Gefahr von Quelltextklonen entsteht.

4.3 Problemlösung mithilfe einer DSL

Die im vorherigem Absatz beschriebenen Probleme bei der Modulerstellung werden besonders im vierten Schritt deutlich. Die Erstellung von Algorithmen zur Änderungsausbreitung besteht ausschließlich aus dem Programmieren in Java. Da eine Vielzahl der Regeln einen ähnlichen Aufbau haben, bietet es sich an, dass die Domänenexperten lediglich ausdrücken, was für eine Art von Änderungsausbreitung sie festlegen wollen und nicht wie diese im Detail abläuft. Dazu muss in einem ersten Schritt identifiziert werden, was denn typische Arten einer Änderungsausbreitung sind. In einem nächsten Schritt kann dann eine deklarative DSL entwickelt werden, welche für jeden typischen Bestandteil einer Änderungsausbreitung ein Sprachelement beinhaltet. Die Domänenexperten können anschließend mit dieser DSL die Änderungsausbreitungsregeln formulieren. Die Regeln werden schließlich in Java-Quelltext transpiliert und in das KAMP-Framework eingebunden.

Die Änderungsausbreitungsanalyse in KAMP ist regelbasiert. Das erleichtert die Integration einer DSL in das bestehende Framework. Die zu entwickelnde Sprache muss lediglich ein Sprachelement zur Definition von Regeln und deren Ausführungsreihenfolge besitzen. Eine Regel in der Sprache beschreibt damit eine Änderungsausbreitung für ein bestimmtes Modellelement oder eine Menge bestimmter Modellelemente.

Durch eine domänenspezifische Sprache wird erreicht, dass Domänenexperten den Algorithmus zur Änderungsausbreitung in KAMP konfigurieren können, ohne zwingend Java schreiben zu müssen. Der Vorteil einer solchen DSL besteht somit darin, dass Domänenexperten weder Java können müssen, um typische Änderungsausbreitungen zu formulieren, noch eine Integration der Regel in das Framework durchführen müssen. Mit welchen Mitteln eine solche Integration erreicht werden kann, ist in Kapitel 6 beschrieben. Das folgende Kapitel führt schrittweise alle Sprachelemente einer Sprache ein, die für den beschriebenen Zweck geeignet ist.

Der deklarative Charakter der zu entwickelnden DSL senkt die Anzahl der benötigten Quelltextzeilen und reduziert somit die Gefahr von Quelltextklonen. Außerdem erhalten Domänenexperten durch eine angepasste Autovervollständigung zusätzliche Unterstützung.

Es wird eine weitere DSL benötigt, um die Ergebnisse der Änderungsausbreitungsanalyse, also die einzelnen betroffenen Elemente, bestimmten Gruppierungen aus dem Modifikationsmarkierungen-Metamodell zuzuordnen. Eine solche **Mapping-Sprache** stellt ein Thema für anknüpfende Arbeiten dar.

5 Change Propagation Rule Language

Im vorangehenden Kapitel wird beschrieben, dass es möglich ist, den Domänenexperten die Erweiterung von KAMP durch den Einsatz einer DSL zu erleichtern. In diesem Kapitel wird der Beitrag dieser Arbeit zu einer bestimmten domänenspezifischen Sprache vorgestellt. Es handelt sich dabei um eine Weiterentwicklung der in Abschnitt 2.4 eingeführten Change Propagation Rule Language (CPRL). Der Name lässt bereits erkennen, dass es sich hierbei um eine regelbasierte Sprache zur Definition von Änderungsausbreitungen handelt. Im Folgenden werden die Ideen und Konzepte hinter der Sprache vorgestellt. Seit dem vergangenen Konferenzpapier [BWLH] von Busch u. a. wurde die Syntax grundlegend überarbeitet. Es wird deshalb auf die vollständige Struktur der neuen Sprachversion eingegangen.

In Abschnitt 5.1 wird ein Überblick über die Struktur von CPRL und deren Basis-Sprachkonstrukte gegeben. Unter Basis-Sprachkonstrukten werden Sprachelemente aus anderen DSLs verstanden, die als Grundlage zur Beschreibung von CPRL verwendet werden. Es existieren vorgefertigte Grammatikregeln für Bezeichner, die von vielen DSLs verwendet werden. So eine Grammatikregel ist beispielsweise das Terminal Qualifizierter-Name, das für die Benennung von Java-Klassen verwendet wird.

Die darauf folgenden Abschnitte erläutern die einzelnen Sprachelemente aus CPRL. Abschnitt 5.2 erläutert das Wurzelelement Regeldatei, Abschnitt 5.3 die Regel, Abschnitt 5.4 die Regelquelle, Abschnitt 5.5 die Abfrage, Abschnitt 5.6 die Vorwärtsnavigation, Abschnitt 5.7 die Rückwärtsnavigation und Abschnitt 5.8 die Verursachungselementmarkierung.

5.1 Einführung in die CPRL

In Abschnitt 5.1.1 wird beschrieben, wie die Eigenschaften der Änderungsausbreitungsanalyse von KAMP die Entstehung von CPRL beeinflusst haben. Es werden außerdem weitere, in CPRL eingebundene, DSLs genannt und deren Funktion wird erläutert.

Abschnitt 5.1.2 beschreibt die Struktur von CPRL und gibt einen Überblick über die enthaltenen Sprachelemente und deren Beziehung zueinander.

Abschnitt 5.1.3 thematisiert allgemeine Sprachkonstrukte aus Xtext, deren Verständnis vorausgesetzt wird, da die CPRL-Sprachelemente aus ihnen zusammengesetzt sind. Des Weiteren wird die Notation der Grammatik erklärt.

5.1.1 Entstehung

Das Ziel der CPRL besteht darin, eine deklarative und somit kompakte Formulierung von Änderungsausbreitungen zu ermöglichen. In Abschnitt 2.3.3 wurden die Kriterien herausgearbeitet, nach denen sich die Änderungsausbreitungsanalyse in KAMP richtet. Die DSL

DSL	Funktion
Xbase	Definition von imperativen, Turing-mächtigen Ausdrücken in einer Java-ähnlichen Syntax
Xtype	Definition von generellen Datentypen (z.B. String, ID, usw.) und eines Java-ähnlichen Typsystems (z.B. Generics, Arrays, Funktionen, usw.)
MIR-Base	Referenzierung von bestehenden Ecore-Metamodellen und deren Eigenschaften (z.B. Strukturmerkmale, Attribute, usw.)

Tabelle 5.1: DSLs, die teilweise in CPRL eingebunden sind und welche Funktion sie erfüllen

soll Sprachkonstrukte enthalten, um typische Bestandteile der Änderungsausbreitungsanalyse in KAMP zu unterstützen. Im Folgenden wird beschrieben, welche Eigenschaften der Änderungsausbreitungsanalyse in KAMP die Gestaltung der Sprache beeinflusst haben.

Der Anwendungsbereich erfordert, dass CPRL in der Lage ist, mit formalen Architekturmodellen umzugehen. Um dies zu erreichen, werden Sprachkonstrukte aus MIR-Base [Wer16] benutzt. Diese ermöglichen es, beliebige Ecore-Metamodelle zu importieren und deren Inhalt zu referenzieren. Eine weitere Anforderung ist, dass nur ein einziger Typ von Änderungsanfrage unterstützt wird. Ein Element kann zu jedem Zeitpunkt der Analyse immer nur einen der beiden Zustände besitzen: *als betroffen markiert* oder *nicht markiert*. Ist ein Element bereits als betroffen markiert, so kann diese Markierung bis zum Ende der Analyse nicht mehr entfernt werden. Die Markierung von Elementen ist somit *final*. Eine weitere Anforderung ist, dass der Algorithmus zur Änderungsausbreitungsanalyse *regelbasiert* ist. Die Sprache bietet Sprachkonstrukte, um Regeln zu definieren. Diese beschreiben, wie ausgehend von bereits markierten Elementen, weitere Elemente markiert werden sollen. Aufgrund der Finalität der Markierungen erhöht der Algorithmus die Anzahl der markierten Elemente *sukzessiv* pro Regelausführung. Um die Definition von Prädikaten innerhalb von Regeln zu ermöglichen, ist mit Xbase eine weitere DSL Teil von CPRL. Eine Übersicht aller in CPRL verwendeten Sprachen ist in Tabelle 5.1 zu finden.

Den Ausgangspunkt der Änderungsausbreitungsanalyse stellen die Anfangsmarkierungen dar, welche vom Endbenutzer gesetzt werden. Diese beinhalten jeweils das betroffene Element, welches dem Algorithmus als Eingabe übermittelt wird. Der Algorithmus führt anschließend die durch CPRL formulierten Regeln aus und gibt als Ausgabe die Menge aller markierten Elemente zurück.

5.1.2 Struktur der Sprache

CPRL wird mithilfe der Xtext Grammatiksprache formuliert. Deshalb wird CPRL durch eine LL(k)-Grammatik beschrieben. Dies ist ein spezielle kontextfreie Grammatik, deren Ableitungsbaum eindeutig ist.

Im Folgenden werden die wichtigsten Sprachelemente und deren Funktion erklärt. In Abbildung 5.1 findet sich eine Übersicht dieser Sprachelemente. In der Abbildung sind die Elemente der Vorwärts- und Rückwärtsnavigation nicht im Detail ausgeführt. Diese befinden sich in der Folgeabbildung 5.2. Bestandteile, die in der Übersicht eingeführt werden, sind in der Folgeabbildung grau hinterlegt. Um die Syntax textuell nachvollziehen

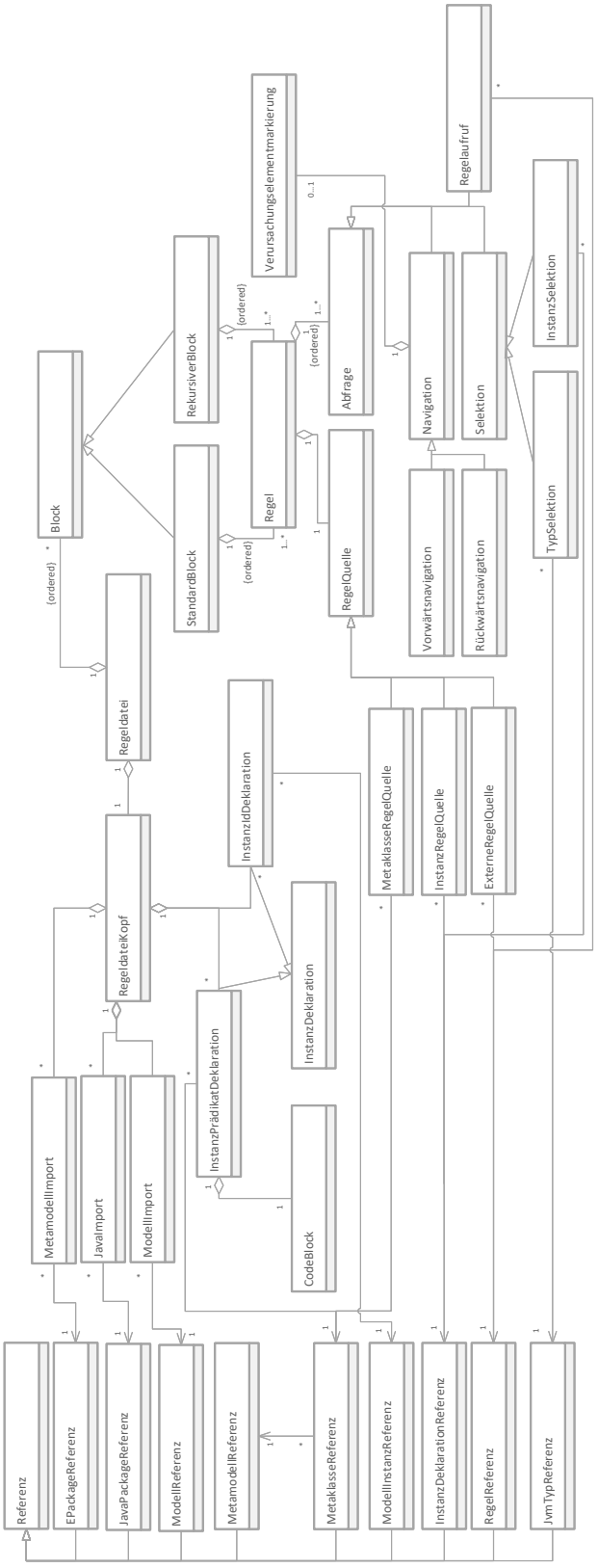

Abbildung 5.1: Syntax der CPRL - Übersicht über zentrale Bestandteile

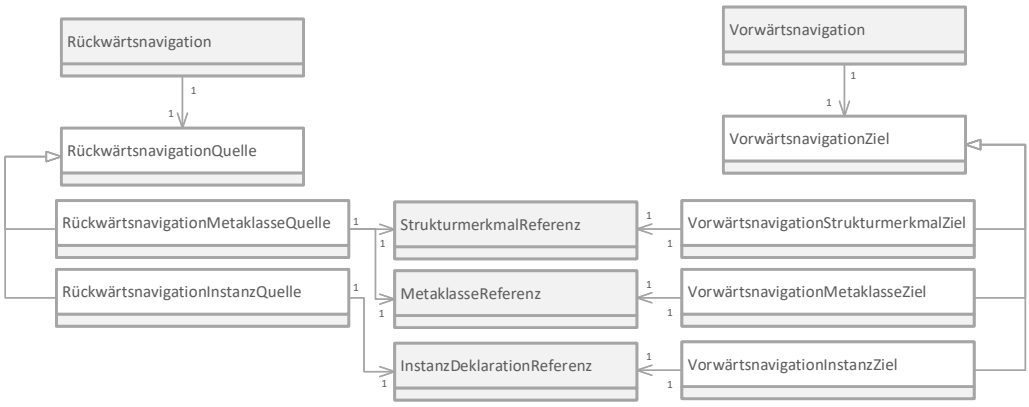

Abbildung 5.2: Syntax der CPRL - Detailansicht der Navigation

zu können, ist ab dem nächsten Abschnitt zum Abschluss jeweils ein Listing mit der Grammatik angehängt, die in dem jeweiligen Abschnitt thematisiert wurde.

Das Kernelement der Sprache ist die *Regel* (engl. rule). Eine Regel definiert, wie aus einer Menge an markierten Eingabeelementen eine Menge an markierten Ausgabeelementen ermittelt wird. Dabei durchläuft jedes Eingabeelement eine Reihe von *Abfragen* (engl. lookups). Eine Abfrage bekommt als Eingabeelemente stets die Ausgabeelemente der vorangehenden Abfrage. Das Konzept der Abfrage ermöglicht a) von einem Element zu einem anderen zu navigieren und b) die aktuell betrachteten Elemente zu filtern . Eine spezielle Art von Abfrage stellt die erste Abfrage innerhalb einer Regel dar. Sie wird als *Regelquelle* (engl. rule source) bezeichnet. Diese Abfrage hat keine vorangehende Abfrage. Sie verwendet deshalb als Eingabeelemente die Menge der bereits markierten Elemente.

Die Anordnung der Regeln ist von Bedeutung. Das oberste Element der Sprache stellt die *Regeldatei* (engl. rule file) dar. Diese enthält Konfigurationsdetails (wie z.B. Imports) auf die in späteren Abschnitten eingegangen wird. Weiterhin enthält sie eine Menge an *Blöcken* (engl. blocks). Die Blöcke stellen eine Gruppierung von Regeln dar. Ein Block beinhaltet eine oder mehrere Regeln. Die Reihenfolge der Blöcke ist bedeutsam. Der Algorithmus der Änderungsausbreitungsanalyse führt Blöcke in der gegebenen Reihenfolge - also von oben nach unten - aus. Ebenso verhält es sich mit der Reihenfolge der Regeln innerhalb von Blöcken. Die obere Regel wird stets vor der unteren ausgeführt. Die Ausführungsreihenfolge der Regeln ist aus dem folgenden Grund von Bedeutung: Die markierten Elemente, welche von einer bestimmten Regel gefunden werden, stehen den Regeln unterhalb ebendieser zur Verfügung. Deshalb operieren Regelquellen immer sowohl auf den markierten Elementen der Anfangsmarkierungen als auch auf denjenigen der vorangehenden Regelausgaben. Im Folgenden wird diese Menge als *temporäre Ergebnismenge* bezeichnet. Die temporäre Ergebnismenge nach der Ausführung der letzten Regel wird als *endgültige Ergebnismenge* bezeichnet. Sie wird als Ergebnis vom Algorithmus der Änderungsausbreitungsanalyse zurückgegeben.

Schließlich gibt es noch ein besonderes Konstrukt in CPRL, das speziell für die Änderungsausbreitungsanalyse ausgelegt und in keiner vergleichbaren Sprache vorhanden ist. Es handelt sich um die *Verursachungselementmarkierung* (engl. causing entity marker). Zweck der Verursachungselementmarkierung ist es, das Modellelement anzugeben, welches ausschlaggebend für die Markierung eines bestimmten Modellelements durch eine Regel war. Diese spezielle Markierung kann benutzt werden, um eine Abfrage zu kennzeichnen.

5.1.3 Basis-Sprachkonstrukte

Dieser Abschnitt geht auf grundlegende Sprachkonstrukte ein. Alle Konstrukte, die CPRL-spezifisch sind, werden in nachfolgenden Abschnitten im Detail thematisiert.

Abschnitt 2.1.2 beschreibt, dass eine DSL aus abstrakter Syntax, statischer und dynamischer Semantik, sowie konkreter Syntax besteht. Die abstrakte Syntax wird im Folgenden konzeptionell beschrieben. Es wird dabei auf die Idee hinter den erarbeiteten Konzepten eingegangen. Die statische und dynamische Semantik wird textuell beschrieben. Die endgültige dynamische Semantik wird letztendlich durch die Transformation der Sprache in Java Code definiert.

In Abschnitt 2.6 wird auf die Xtext-Grammatiksprache eingegangen, welche zur Implementierung der konkreten Syntax für CPRL verwendet wird. Für dieses Kapitel wird mit der Erweiterten Backus-Naur-Form (EBNF) [ISO96] eine abweichende Notation für die Grammatik der CPRL verwendet. Die EBNF wird verwendet, da sie weiter verbreitet und somit geläufiger ist als die Xtext-Grammatiksprache. Sie verwendet eckige Klammern für optionale und geschweifte Klammern für sich wiederholende Elemente.

Ein Text in CPRL besteht zunächst aus Terminalsymbolen, das heißt, aus sichtbaren Zeichen wie zum Beispiel Buchstaben, Ziffern und Satzzeichen. Terminalsymbole können nicht ersetzt werden. Um festzulegen, welche Zeichenfolgen über einem Alphabet einen gültigen Text in CPRL darstellen, wird eine Grammatik für die Sprache angegeben. Eine Grammatik besteht aus einem Startsymbol - hier auch Wurzelelement genannt. Das Wurzelelement ist ein Nichtterminalsymbol, das heißt, ein ersetzbares Symbol. Um einen gültigen Text zu ergeben, muss es so lange durch weitere Nichtterminale oder Terminale ersetzt werden, bis der Text nur noch aus Terminalen besteht. Wie ein Nichtterminal ersetzt werden kann, wird durch Produktionsregeln beschrieben. Alpha sei ein Nichtterminal und Beta mindestens ein beliebiges Symbol. Produktionsregeln bestehen aus einer Relation $\alpha \rightarrow \beta$. Die Relation beschreibt, dass α durch β ersetzt werden kann. Eine einzelne Ersetzung wird auch als Ableitung bezeichnet. Eine Zeichenkette ist aus einem Nichtterminal ableitbar, wenn mindestens eine Reihenfolge von Produktionsregeln existiert, die auf das Nichtterminal angewandt werden können, sodass daraus die Zeichenkette resultiert.

In dieser Arbeit wird eine Produktionsregel dargestellt, indem α in die linke Spalte und β in die rechte Spalte eines Listings geschrieben wird. Die beiden Spalten sind durch ein Gleichheitszeichen oder einen Doppelpunkt getrennt. Ein Doppelpunkt bedeutet, dass es sich um eine textuelle Beschreibung statt um eine Beschreibung in EBNF handelt.

Die Terminale sind durch die großgeschriebenen Symbole ⟨*ID*⟩ und ⟨*STRING*⟩ angegeben. Außerdem sind alle in Anführungszeichen eingeschlossenen Zeichenfolgen innerhalb von Produktionsregeln Schlüsselwörter und somit Terminale.

Alle Zeichenketten außer *ID* und *STRING*, die in eckigen Klammern eingeschlossen sind, werden als Nichtterminale bezeichnet. Nichtterminale werden in diesem Kapitel Sprachelementen gleichgesetzt. Dies ist darin begründet, dass die Grammatik derart gestaltet ist, dass jedes Nichtterminal eine eigenständige Semantik für die Person besitzt, welche die Sprache verwendet. Eine Zeichenkette aus Terminalen in einer CPRL-Datei, die aus einem Nichtterminal abgeleitet wurde, wird als Instanz des jeweiligen Sprachelements bezeichnet.

Ein Konzept aus Xtext ist der Querverweis. Dieser wird verwendet, um Instanzen von Sprachelementen zu referenzieren. Querverweise haben eine besondere statische Semantik. Sie müssen auf eine Instanz eines Sprachelements verweisen, das existiert und die jeweilige Bezeichnung trägt. Diese Semantik ist in der EBNF standardmäßig nicht abgebildet. Um sie dennoch erkennbar zu machen, werden alle Nichtterminale, die einen Querverweis darstellen, mit dem Suffix *Referenz* versehen. Nichtterminale, die einen Querverweis darstellen, werden in den Listings lediglich textuell beschrieben. Die Information, welche Terminale innerhalb eines Sprachelements dessen Bezeichnung bilden, kann in EBNF nicht dargestellt werden. Es wird stattdessen im zum Sprachelement gehörenden Absatz des Fließtexts beschrieben.

CPRL verwendet Sprachelemente aus anderen DSLs, wie zum Beispiel Xtype und Xbase. Sie sind in den Listings lediglich textuell beschrieben.

Die Grammatik der folgenden Basis-Sprachkonstrukte ist in Listing 5.1 notiert.

Eine ⟨*ID*⟩ stellt einen einfachen Bezeichner dar. CPRL bedient sich hierbei des ID-Terminals der Xtype Sprache. Da Xtype stark an Java angelehnt ist, gilt, dass jeder gültige Java-Bezeichner [GJS+18] eine gültige ID in CPRL ist, solange er ausschließlich Zeichen der Latin-1 Kodierung [ISO98] verwendet. Java lässt darüber hinaus Zeichen aus Unicode [ISO12] zu. Eine bestimmte ID darf lediglich einer einzigen Instanz eines Sprachelements zugewiesen werden. Sie dient als Identifikator für diese Instanz.

Ein ⟨*QualifizierterName*⟩ stellt einen erweiterten Bezeichner dar. Er besteht aus mehreren einfachen Bezeichnern, die mit einem Punkt getrennt sind. Ein Qualifizierter-Name wird als Identifikator für eine Instanz eines Sprachelements verwendet, für die ein einfacher Bezeichner nicht ausreicht, da er ambivalent ist.

Das Terminal ⟨*STRING*⟩ ist in Xtype definiert. Es bezeichnet eine beliebige Unicode-Zeichenkette, die entweder in einfachen oder doppelten Anführungszeichen eingeschlossen ist. Die Anführungszeichen, sowie das Backslash-Symbol dürfen nur in einer *Escape-Sequenz* innerhalb der Zeichenkette vorkommen. Weitere Informationen dazu sind in der Java-Referenz [GJS+18] zu finden. Der Unterschied zwischen dem String-Terminal in dieser Sprache und einem String-Literal in Java ist, dass Java lediglich doppelte Anführungszeichen als Trennzeichen erlaubt. Xtype erlaubt sowohl doppelte, als auch einfache Anführungszeichen.

Ein ⟨*CodeBlock*⟩ wird durch das Sprachelement *XBlockExpression* aus Xbase definiert und besitzt den Typ *XExpression*. In CPRL besteht die dynamische Semantik eines Code-Blocks darin, ein Turing-mächtiges Prädikat darzustellen. Es handelt sich dabei um einen

Anweisungsblock, der einzelne Anweisungen enthält, die in geschweiften Klammern eingeschlossen sind. Die Syntax der einzelnen Anweisungen ist sehr umfangreich und wird durch mehrere Sprachelemente der Sprache Xbase beschrieben. Die Anweisungen in Form von XExpressions sind syntaktisch denen in Java sehr ähnlich. Eine Anweisung in Xbase ist jedoch immer zugleich ein Ausdruck. Die statische Semantik eines Prädikats erfordert, dass der letzte Ausdruck im Code-Block einen booleschen Typ besitzt. Eine Besonderheit des Prädikats besteht darin, dass dem Sichtbarkeitsbereich des Code-Blocks eine Variable mit der Bezeichnung *it* bereitgestellt wird. Der Typ dieser Variable ist vom Kontext abhängig, in dem das Prädikat in der Sprache vorkommt.

⟨*ID*⟩	:	Zeichenkette in Form eines validen Bezeichners
⟨*QualifizierterName*⟩	=	⟨*ID*⟩ { '.' ⟨*ID*⟩ }
⟨*STRING*⟩	:	beliebige Zeichenkette in Anführungszeichen
⟨*CodeBlock*⟩	:	Xbase Ausdruck in geschweiften Klammern
⟨*Referenz*⟩	:	Querverweis auf bereits definiertes Element

Listing 5.1: Basis-Sprachkonstrukte in CPRL

Für Beispiele in diesem Kapitel gilt: Schlüsselwörter von CPRL werden rot gefärbt, während jene von Xbase blau gefärbt werden. Durch Xbase zur Verfügung gestellte Variablen werden grün gefärbt. In Listing 5.2 wird diese Farbgebung demonstriert. Das Schlüsselwort *find* gehört zur konkreten Syntax von CPRL und *return* ist ein Schlüsselwort von Xbase. Das grün markierte Wort *it* ist eine von Xbase zur Verfügung gestellte Variable. In diesem Kontext hat diese Variable den Typ der Metaklasse *BasicComponent* aus dem PCM. Auslassungspunkte - gekennzeichnet durch den Dreipunkt "..." - sind nicht Teil der Sprache. Sie werden an bestimmten Stellen in Beispielen verwendet, um anzudeuten, dass dort zur syntaktischen Korrektheit weitere Symbole stehen müssten, diese allerdings für das Beispiel unerheblich sind und deshalb ausgelassen wurden. CPRL erlaubt die Verwendung von Kommentaren in Java-Syntax. Es lässt sich entweder eine bestimmte Zeichenfolge zwischen den Symbolen "/*" und "*/" oder der Rest einer Zeile nach dem Symbol "//" auskommentieren.

```
// this is a single line comment in CPRL
find ComponentABC: pcm::BasicComponent {

  /* this is a comment, too, which might span multiple lines */
  return it.entityName.equals("ABC")
}
```

Listing 5.2: Beispiel der Farbgebung für CPRL-Listings in dieser Arbeit

5.2 Regeldatei

Die ⟨*Regeldatei*⟩ ist das Wurzelelement (engl. root element) der Sprache. Das bedeutet, dass der Inhalt jeder gültigen CPRL-Datei ausgehend vom Nichtterminal ⟨*Regeldatei*⟩ ableitbar sein muss.

Abschnitt 5.2.1 beschreibt, aus welchen Sprachelementen sich eine Regeldatei zusammensetzt. Auf die einzelnen Sprachelemente wird dann in den folgenden Abschnitten eingegangen.

Abschnitt 5.2.2 beschreibt die Struktur des Kopfbereichs.

Abschnitt 5.2.5 beschreibt den Rumpfbereich.

Die Sprache unterscheidet zwischen zwei Modellierungsebenen: **Metamodellebene** und **Modellebene**. Die unterschiedlichen Sprachelemente zum Zugriff auf Metamodelle beziehungsweise Modelle auf diesen Ebenen sind in den Abschnitten 5.2.3 und 5.2.4 beschrieben.

5.2.1 Aufbau einer Regeldatei

Die Grammatik einer ⟨*Regeldatei*⟩ ist in Listing 5.3 notiert. Die ⟨*Regeldatei*⟩ besteht aus genau einem Kopfbereich, gefolgt von einer beliebigen Anzahl Blöcke. Im Kopfbereich werden interne und externe Elemente deklariert, die später in Blöcken durch Querverweise referenziert werden können.

Ein Element wird als extern bezeichnet, wenn es außerhalb der DSL deklariert ist. Die Deklaration findet nicht in Textform statt, sondern wird durch die **Laufzeitumgebung der Sprache** aufgelöst. Die Laufzeitumgebung der Sprache ist die Umgebung, in welcher die Sprache eingebunden ist. Sie löst Referenzen auf externe Elemente, wie zum Beispiel Modelle oder Metamodelle auf, die vom Endbenutzer oder Entwickler zuvor hinterlegt wurden. Xtext unterstützt Eclipse, diverse Browser, sowie alle Editoren, die das Language Server Protocol (LSP) [Mic18] umsetzen, als Laufzeitumgebung [ES18e].

Ein internes Element wird im Text der CPRL-Datei deklariert.

⟨*Regeldatei*⟩ = ⟨*RegeldateiKopf*⟩
{⟨*Block*⟩}

Listing 5.3: Grammatik einer ⟨*Regeldatei*⟩

5.2.2 Importieren externer Elemente

Der ⟨*RegeldateiKopf*⟩ wird verwendet, um externe Elemente zu importieren und Instanzen zu deklarieren. Das ermöglicht den Regeln im Dateirumpf externe Elemente und bestimmte Modellelemente zu referenzieren. Als *Instanz* wird ein einzelnes Modellelement bezeichnet. Es handelt sich um eine Instanz einer bestimmten Metaklasse eines Metamodells.

Der ⟨*RegeldateiKopf*⟩ beinhaltet die folgenden Nichtterminale: ⟨*MetamodellImport*⟩, ⟨*JavaImport*⟩, ⟨*ModellImport*⟩, ⟨*InstanzPrädikatDeklaration*⟩ und ⟨*InstanzIdDeklaration*⟩. Diese müssen in exakt der obigen Reihenfolge deklariert werden. Jedes Nichtterminal kann dabei beliebig oft vorkommen.

Die zugehörige Grammatik ist in Listing 5.4 notiert. Die Sprachelemente ⟨*InstanzPrädikat-Deklaration*⟩ und ⟨*InstanzIdDeklaration*⟩ aus der Grammatik werden in Abschnitt 5.2.4 erläutert.

Ein ⟨*MetamodellImport*⟩ ist eine Anweisung, die ein bestimmtes Metamodell unter einem Alias bekannt macht. Die Anweisung besteht aus einer ⟨*EPackageReferenz*⟩, die das Metamodell eindeutig identifiziert und einem Alias, unter dem das Metamodell im Dateirumpf referenziert werden kann. Der Alias stellt zugleich den Namen des Imports dar und muss ein wohlgeformtes ⟨*ID*⟩-Terminal sein.

Eine ⟨*EPackageReferenz*⟩ ist ein Querverweis auf ein Ecore-Paket. Die Referenz wird durch den Namensraum-Identifikator (engl. namespace Uniform Resource Identifier, kurz: nsURI) des Pakets aufgelöst. Da es sich um eine Referenz auf ein externes Element handelt, wird der Sichtbarkeitsbereich durch die Laufzeitumgebung bestimmt, in der die Sprache ausgeführt wird. Die Eclipse IDE stellt beispielsweise alle Ecore-Modelle bereit, die in aktivierten Plugins vorhanden sind. Die Syntax einer nsURI erfordert, dass die Referenz aus einem wohlgeformten ⟨*STRING*⟩-Terminal besteht. Die statische Semantik des Querverweises erfordert, dass ein Paket mit der angegebenen nsURI der Laufzeitumgebung bekannt ist.

Das Sprachelement ⟨*JavaImport*⟩ wird für die Implementierung von CPRL in Xtext benötigt. Um die Änderungsausbreitung in KAMP zu konfigurieren, wird die DSL in Java-Quelltext übersetzt. Diese Quelltextgenerierung wird von Xtext ausgeführt. Um die Klassen zu lokalisieren, welche für die importierten Ecore-Modelle generiert wurden, verwendet Xtext den Klassenpfad (engl. classpath) von Java. Durch die Java-Import-Anweisung wird veranlasst, dass der Inhalt des angegebenen Java-Pakets auf den Klassenpfad geladen wird. Das Java-Paket wird durch eine ⟨*JavaPackageReferenz*⟩ beschrieben.

Eine ⟨*JavaPackageReferenz*⟩ stellt einen Querverweis auf ein Java-Paket dar. Diese Referenz muss dem Nichtterminal ⟨*QualifizierterName*⟩ entsprechen. Die statische Semantik erfordert, dass die Referenz auf ein Java-Paket verweist, dass der Laufzeitumgebung der Sprache bekannt ist. In der Eclipse IDE muss beispielsweise ein Plugin aktiviert sein, welches das angegebene Java-Paket mithilfe des Manifests exportiert.

Ein ⟨*ModellImport*⟩ ist eine Anweisung, die ein Modell unter einem Alias bekannt macht. Die Anweisung besteht aus einer ⟨*ModellReferenz*⟩, die das Modell anhand seines Pfades im Dateisystem eindeutig identifiziert und einem Alias, unter dem das Modell innerhalb einer ⟨*InstanzIdDeklaration*⟩ im Dateikopf referenziert werden kann.

Eine ⟨*ModellReferenz*⟩ ist ein Querverweis auf eine Modell-Datei im Dateisystem. Die Referenz muss aus einem wohlgeformten ⟨*STRING*⟩-Terminal bestehen. Die statische Semantik setzt voraus, dass die Laufzeitumgebung der Sprache auf eine valide Datei im XMI-Format [Obj15] unter dem angegebenen Pfad zugreifen kann. In der Eclipse IDE wird der Dateipfad beispielsweise relativ zum Workspace-Verzeichnis angegeben. Somit können alle Modelle im Workspace referenziert werden. Der Workspace ist die Bezeichnung für die Arbeitsfläche, die beim Start der Eclipse IDE ausgewählt wird. Auf die Arbeitsfläche kann der Benutzer bestehende Projekte importieren oder neue Projekte anlegen.

⟨*RegeldateiKopf*⟩	=	{⟨*MetamodellImport*⟩}
		{⟨*JavaImport*⟩}
		{⟨*ModellImport*⟩}
		{⟨*InstanzPrädikatDeklaration*⟩}
		{⟨*InstanzIdDeklaration*⟩}
⟨*MetamodellImport*⟩	=	'import' ⟨*EPackageReferenz*⟩ 'as' ⟨*ID*⟩
⟨*JavaImport*⟩	=	'import-package' ⟨*JavaPackageReferenz*⟩
⟨*ModellImport*⟩	=	'import-model' ⟨*ModellReferenz*⟩ 'as' ⟨*ID*⟩
⟨*ModellReferenz*⟩	:	Pfad zu einer Modell-Datei im XMI Format
⟨*EPackageReferenz*⟩	:	nsURI eines vorhandenen Ecore Packages
⟨*JavaPackageReferenz*⟩	:	Qualifizierter Name eines Java Pakets

Listing 5.4: Grammatik eines ⟨*RegeldateiKopfs*⟩

```
import "http://palladiosimulator.org/PalladioComponentModel/5.2" as pcm

import-package edu.kit.ipd.sdq.kamp4bp.core

import-model "/MyProject/example.repository" as repo
```

Listing 5.5: Beispiel eines ⟨*RegeldateiKopfs*⟩

Ein Beispiel für die in diesem Abschnitt beschriebenen Sprachelemente befindet sich in Listing 5.5. In Zeile 1 wird das Metamodell mit der nsURI *http://palladiosimulator.org/PalladioComponentModel/5.2* importiert und unter der Bezeichnung *pcm* bekannt gemacht. In Zeile 3 wird das Paket *edu.kit.ipd.sdq.kamp4bp.core* auf den Klassenpfad geladen. In Zeile 5 wird das Modell aus der Datei */MyProject/example.repository* importiert und unter der Bezeichnung *repo* bekannt gemacht.

5.2.3 Zugriff auf die Metamodellebene

In dieser Arbeit wurden zwei Ebenen identifiziert, auf denen Änderungsausbreitungen typischerweise formuliert werden. Dieser Abschnitt thematisiert die Metamodellebene. Im darauf folgenden Abschnitt wird die Modellebene thematisiert.

Auf der Metamodellebene können Aussagen über eine Menge an Instanzen gemacht werden, die denselben Typ haben. In Kapitel 2.2.2 wird definiert, dass der Typ einer bestimmten Instanz in Ecore der Metaklasse entspricht, die als eClass hinterlegt ist. Eine Instanz hat also immer jene Metaklasse als Typ, auf deren Basis sie erstellt wurde.

Die folgenden Sprachelemente sind in der Grammatik in Listing 5.6 notiert.

Eine ⟨*MetamodellReferenz*⟩ besteht aus dem ⟨*ID*⟩-Terminal. Es handelt sich um einen Querverweis auf einen ⟨*MetamodellImport*⟩. Die statische Semantik erfordert, dass es sich um den Alias eines bereits deklarierten Metamodell-Imports handelt.

Eine ⟨*MetaklasseReferenz*⟩ wird verwendet, um eine beliebige Metaklasse aus einem bestimmten Metamodell zu referenzieren. Die Referenz besteht aus einer ⟨*MetamodellReferenz*⟩, gefolgt von zwei Doppelpunkten und dem Nichtterminal ⟨*QualifizierterName*⟩. Dieses entspricht dem qualifizierten Namen einer EClass. Die statische Semantik setzt voraus, dass eine valide ⟨*MetamodellReferenz*⟩ angegeben wird und die zugehörige Metaklasse in dem referenzierten Metamodell vorhanden ist. Bei dem Namen einer Metaklasse handelt es sich um ein externes Element, das von der Laufzeitumgebung der Sprache aus dem referenzierten Metamodell ermittelt werden muss.

Eine ⟨*StrukturmerkmalReferenz*⟩ ist ein Querverweis auf ein Strukturmerkmal einer bestimmten Metaklasse und besteht aus dem ⟨*ID*⟩-Terminal. Die statische Semantik sieht vor, dass die Referenz den Namen einer bestehenden EReference enthält. Strukturmerkmale vom Typ EAttribute werden in der DSL nicht berücksichtigt, da festgestellt wurde, dass Ecore-Datentypen nicht zur Änderungsausbreitung geeignet sind. Aus welcher Metaklasse das Strukturmerkmal referenziert wird, hängt vom Kontext ab, in dem der Querverweis verwendet wird. Da es sich bei der EReference um ein externes Element handelt, muss die Laufzeitumgebung der Sprache es aus der zugehörigen Metaklasse ermitteln.

⟨*MetamodellReferenz*⟩	:	Referenz auf ein importiertes Metamodell
⟨*MetaklasseReferenz*⟩	:	Referenz auf die Metaklasse eines importierten Metamodells
⟨*StrukturmerkmalReferenz*⟩	:	Referenz auf ein Strukturmerkmal einer bestimmten Metaklasse

Listing 5.6: Sprachelemente zur Referenzierung der Metamodellebene

Ein Beispiel für die Elemente auf Metamodellebene ist in Listing 5.7 dargestellt. Zeile 1 besteht aus einem Import des Metamodells mit der nsURI *http://palladiosimulator.org /PalladioComponentModel/5.2*, welches unter der Bezeichnung *pcm* bekannt gemacht wird. In Zeile 3 ist ein Beispiel einer ⟨*MetaklasseReferenz*⟩ dargestellt. Sie referenziert die Metaklasse *BasicComponent* aus dem vorher importierten Metamodell *pcm*.

```
import "http://palladiosimulator.org/PalladioComponentModel/5.2" as pcm

// metaclass BasicComponent from metamodel pcm
pcm::BasicComponent
```

Listing 5.7: Beispiel einer ⟨*MetaklasseReferenz*⟩

5.2.4 Zugriff auf die Modellebene

Im vorherigen Abschnitt wurde die Metamodellebene betrachtet. Der folgende Abschnitt befasst sich mit der darunter liegenden Modellebene. Auf der Modellebene können Aussagen über eine oder mehrere Instanzen anhand ihrer konkreten Eigenschaften gemacht werden. Als Eigenschaften von Modellelementen werden dessen Strukturmerkmale bezeichnet. Der Zugriff auf Instanzen erfordert deren vorherige Deklaration. Dies liegt daran, dass Instanzen im Gegensatz zu Metaklassen nicht immer eindeutig identifiziert werden können. Sie enthalten keinen eindeutigen Bezeichner, beziehungsweise es wurde keine Eigenschaft als Identifikator standardisiert.

Im Folgenden werden zwei Möglichkeiten beschrieben, wie Instanzen dem Regeldatei-Rumpf bekannt gemacht werden können. Im Listing 5.8 ist eine zugehörige Grammatik notiert. Eine ⟨*InstanzDeklaration*⟩ ordnet eine beliebig große Menge an Instanzen einem eindeutigen Namen zu.

Die umfangreiche Art einer Instanz-Deklaration ist die ⟨*InstanzPrädikatDeklaration*⟩. Diese Deklaration beinhaltet ein Prädikat, welches durch einen ⟨*CodeBlock*⟩ formuliert wird. Außerdem besteht die Deklaration aus einem Instanz-Namen, der durch ein ⟨*ID*⟩ Terminal bestimmt wird und eine ⟨*MetaklasseReferenz*⟩. Die referenzierte Metaklasse stellt den Typ der Deklaration dar. Sie wird ebenfalls als Typ der *it*-Variable im Code-Block verwendet. Das Prädikat wird benutzt, um eine Menge an Instanzen auf bestimmte Bedingungen hin zu überprüfen. Evaluiert das Prädikat für eine bestimmte Instanz zu wahr, so wird diese dem Instanz-Namen zugewiesen. Einem Instanz-Namen können somit beliebig viele Instanzen zugewiesen werden. Für eine Instanz, die nicht dem Typ der Deklaration entspricht, evaluiert das Prädikat stets zu falsch.

Die spezielle Art einer Instanz-Deklaration ist die ⟨*InstanzIdDeklaration*⟩. Diese weist eine einzige Instanz einem Instanz-Namen zu. Als Name wird ein ⟨*ID*⟩ Terminal verwendet. Die Instanz wird durch eine ⟨*ModellInstanzReferenz*⟩ beschrieben. Es handelt sich hierbei um eine kürzere, weniger mächtigere Deklarationsform als die Prädikat-Deklaration. Sie wertet lediglich einige bestimmte Attribute aus, die als Identifikator einer Instanz in Frage kommen. Der Typ der Deklaration wird automatisch aus der Metaklasse der referenzierten Instanz abgeleitet.

Die ⟨*ModellInstanzReferenz*⟩ ist ein Querverweis auf eine Instanz eines bereits importierten Modells. Sie besteht aus einer Referenz auf ein ⟨*ModellImport*⟩ und dem Nichtterminal ⟨*QualifizierterName*⟩, welches die Instanz möglichst eindeutig identifiziert. Die beiden Komponenten sind durch zwei Doppelpunkte voneinander getrennt. Da es kein vordefinier-

tes Attribut gibt, welches eine Instanz eindeutig identifiziert, wird die folgende Heuristik verwendet um eine deterministische ID zu generieren:

1. Verwende als Präfix den Inhalt des Attributs mit dem Namen *name* oder *entityName*.
2. Konkateniere den Präfix mit einem Unterstrich als Trennzeichen.
3. Füge als Suffix den Inhalt des EID-Attributs hinzu. Falls das EID-Attribut einen Bindestrich enthält, ersetze diesen durch ein Dollarzeichen. Diese Konvertierung ist erforderlich, da ein qualifizierter Name keine Bindestriche erlaubt, eine EID allerdings welche enthalten kann.

Diese ID ist für eine bestimmte Instanz mit gewisser Wahrscheinlichkeit innerhalb eines Modells eindeutig. Die Wahrscheinlichkeit für Eindeutigkeit innerhalb eines Modells ist relativ hoch, wenn das EID-Attribut gesetzt ist. Andernfalls hängt sie von der Namensgebung der Instanzen ab. Falls eine Instanz keines der oben genannten Attribute beinhaltet, ist eine Referenzierung über eine ⟨*ModellInstanzReferenz*⟩ nicht möglich und es muss stattdessen ein Prädikat verwendet werden. Da es sich bei der Instanz um ein externes Element handelt, muss die Generierung der gültigen IDs für die ⟨*ModellInstanzReferenz*⟩ von der Laufzeitumgebung der Sprache durchgeführt werden. Die statische Semantik der Sprache erfordert, dass lediglich Instanzen referenziert werden, die im zugehörigen Modell existieren.

Eine ⟨*InstanzDeklarationReferenz*⟩ ist ein Querverweis, der im Regeldatei-Rumpf verwendet wird, um eine ⟨*InstanzDeklaration*⟩ anhand des Namens zu referenzieren. Der Verweis besteht aus dem ⟨*ID*⟩-Terminal. Er besitzt den Typ der zugehörigen Deklaration. Die statische Semantik setzt voraus, dass eine Deklaration mit dem angegebenen Namen im Regeldatei-Kopf existiert.

⟨*InstanzDeklaration*⟩	=	⟨*InstanzPrädikatDeklaration*⟩
	|	⟨*InstanzIdDeklaration*⟩
⟨*InstanzPrädikatDeklaration*⟩	=	'find' ⟨*ID*⟩ ':'
		⟨*MetaklasseReferenz*⟩
		⟨*CodeBlock*⟩
⟨*InstanzIdDeklaration*⟩	=	'instance' ⟨*ID*⟩ ':' ⟨*ModellInstanzReferenz*⟩
⟨*InstanzDeklarationReferenz*⟩	:	Referenz auf eine bereits definierte Instanz-Deklaration
⟨*ModellInstanzReferenz*⟩	:	Referenz auf das Objekt eines importierten Modells

Listing 5.8: Grammatik einer ⟨*InstanzDeklaration*⟩

Ein Beispiel einer ⟨*InstanzPrädikatDeklaration*⟩ ist in Listing 5.9 gegeben. In Zeile 1 wird ein Metamodell mit der Bezeichnung *pcm* importiert. In Zeile 5 wird eine Instanz-Deklaration mittels Prädikat festgelegt, welche die Bezeichnung *MyComponents* trägt. Sie deklariert

Modellelemente vom Typ *BasicComponent*. Der Typ wird durch die Metaklasse *BasicComponent* bestimmt, die aus dem importierten Metamodell *pcm* stammt. In Zeile 6 befindet sich der Xbase-Ausdruck, durch den bestimmt wird, ob ein bestimmtes Modellelement der Deklaration zugeordnet wird. Das Modellelement, welches durch das Prädikat getestet wird, ist als Variable *it* zwischen den geschweiften Klammern erreichbar. Das Prädikat ermittelt den Rückgabewert wahr, falls das Strukturmerkmal *passiveResource_BasicComponent* von *it* mindestens ein Modellelement beinhaltet.

Ein Beispiel einer ⟨*InstanzIdDeklaration*⟩ ist in Listing 5.10 zu finden. In Zeile 1 wird ein Modell mit der Bezeichnung *repo* importiert. In Zeile 6 wird eine Instanz-Deklaration festgelegt, die das Modellelement mit der generierten ID *examplaryComponent__TVJd022yEeipyr94Phon1Q*, aus dem Modell *repo* beinhaltet.

```
import "http://palladiosimulator.org/PalladioComponentModel/5.2" as pcm

// declare all components with at least one passive resource as MyComponents
// the type is explicitly declared as pcm::BasicComponent
find MyComponents: pcm::BasicComponent {
  return it.passiveResource_BasicComponent.length > 1
}
```

Listing 5.9: Beispiel einer ⟨*InstanzPrädikatDeklaration*⟩

```
import-model "/MyProject/example.repository" as repo

// declare an instance named MyComponent
// consists of id _TVJd022yEeipyr94Phon1Q and entity name examplaryComponent
// type is automatically inferred to pcm::BasicComponent from its metaclass
instance MyComponent: repo::examplaryComponent__TVJd022yEeipyr94Phon1Q
```

Listing 5.10: Beispiel einer ⟨*InstanzIdDeklaration*⟩

5.2.5 Gruppierung von Regeln

Der Rumpf der Regeldatei besteht aus einer beliebigen Anzahl an Blöcken und wird dazu verwendet, die Logik der Änderungsausbreitung zu definieren. Ein Block enthält eine oder mehrere Regeln. Er ist somit ein Container für Regeln. Ein ⟨*Block*⟩ kann entweder ein ⟨*StandardBlock*⟩ oder ein ⟨*RekursiverBlock*⟩ sein. Die Grammatik ist in Listing 5.11 zu finden.

Bei der Entwicklung der Sprache hat sich herausgestellt, dass es sich anbietet, zwei verschiedene Arten von Blöcken anzubieten. Der Unterschied der beiden Arten an Blöcken liegt besonders in ihrer dynamischen Semantik.

Ein ⟨*StandardBlock*⟩ stellt eine Aneinanderreihung an Regeln dar. Er wird genau einmal evaluiert.

Ein ⟨*RekursiverBlock*⟩ stellt ebenfalls eine Aneinanderreihung an Regeln dar. Er wird mindestens einmal evaluiert. Falls sich bei der Evaluation eines Blocks die temporäre Ergebnismenge ändert - also betroffene Elemente hinzugefügt werden, die vorher nicht enthalten waren - dann wird der Block nochmals evaluiert. Dies geschieht so lange, bis sich die temporäre Ergebnismenge durch die Evaluation des Blocks nicht mehr verändert. Die Verwendung eines rekursiven Blocks ermöglicht die Formulierung von rekursiven Änderungsausbreitungen, wie sie beispielsweise bei verschachtelten Elementen vorkommen. Ein weiterer typischer Anwendungsfall sind Kompositionen und Aggregationen. Das sind Beziehungen zwischen dem Ganzen und seinen Teilen.

Die Reihenfolge der Blöcke ist von Bedeutung, da diese von oben nach unten evaluiert werden. Mit der Evaluation eines Blocks ist die Ausführung aller in ihm enthaltenen Regeln gemeint. Die Regeln innerhalb eines Blocks werden ebenso von oben nach unten evaluiert.

⟨*Block*⟩	=	⟨*StandardBlock*⟩ | ⟨*RekursiverBlock*⟩
⟨*StandardBlock*⟩	=	⟨*Regel*⟩ {⟨*Regel*⟩}
⟨*RekursiverBlock*⟩	=	'recursive' '{' ⟨*Regel*⟩ {⟨*Regel*⟩} '}'

Listing 5.11: Grammatik für das Sprachelement ⟨*Block*⟩

Ein Beispiel für die konkrete Syntax beider Arten von Blöcken ist in Listing 5.12 zu finden. In Zeile 5 und 8 sind die zwei Regeln A und B definiert, die zusammen einen Standard-Block bilden. Die Regeln C und D in Zeile 13 und 16 sind Teil eines rekursiven Blocks.

```
import "http://palladiosimulator.org/PalladioComponentModel/5.2" as pcm

// standard block containing rules A and B
// it is evaluated exactly once
rule A: metaclass(pcm::ComposedStructure) -> feature(
    ↪ assemblyContexts__ComposedStructure) -> feature(
    ↪ encapsulatedComponent__AssemblyContext);
rule B: ...

// recursive block containing rules C and D
// it is evaluated repeatedly until no more affected elements are found
recursive {
  rule C: metaclass(pcm::ComposedStructure) -> feature(
      ↪ assemblyContexts__ComposedStructure) -> feature(
      ↪ encapsulatedComponent__AssemblyContext);
  rule D: ...
}
```

Listing 5.12: Beispiel für ⟨*StandardBlock*⟩ und ⟨*RekursivenBlock*⟩

5.3 Regel

Die Regel ist das zentrale Sprachelement. Sie definiert eine Änderungsausbreitung entlang bestimmter Modellelemente einer Domäne. Dazu selektiert eine Regel eine Teilmenge aus den zuvor markierten Modellelementen und ermittelt daraus eine Menge an markierten Ausgabeelementen. Die Ausgabeelemente werden der temporären Ergebnismenge hinzugefügt und stehen den nachfolgenden Regeln als Eingabeelemente zur Verfügung. Eine Regel ist stets typisiert. Sie besitzt einen **Eingabetyp** und einen **Ausgabetyp**. Der Eingabetyp ist der Typ der Elemente, für die die Regel eine Änderungsausbreitung definiert. Der Ausgabetyp ist der Typ der Elemente, welche von der Regel in die temporäre Ergebnismenge eingefügt werden.

Abschnitt 5.3.1 beschreibt den Aufbau einer Regel.

In Abschnitt 5.3.2 und 5.3.3 wird kurz auf die beiden Sprachelemente Regel-Quelle und Abfrage eingegangen, die Bestandteil einer Regel sind. Es wird lediglich die Rolle dieser Bestandteile für die Regel als Ganzes thematisiert.

Die Regel-Quelle ist in Abschnitt 5.4 im Detail erläutert. Die Abfrage ist in Abschnitt 5.5 im Detail erläutert. Die detaillierten Abschnitte enthalten Beispiele zur Verdeutlichung der Semantik dieser beiden Sprachelemente.

5.3.1 Aufbau einer Regel

Die Grammatik einer Regel ist in Listing 5.13 notiert. Eine ⟨*Regel*⟩ besteht aus einer Signatur und einem Rumpf. Die Signatur enthält den Regelnamen. Der Rumpf enthält die Quelle und eine oder mehrere Abfragen. Der Name besteht aus dem ⟨*ID*⟩-Terminal.

Eine ⟨*RegelReferenz*⟩ stellt einen Querverweis auf eine Regel dar. Der Verweis besteht aus einem ⟨*ID*⟩-Terminal. Die statische Semantik der Sprache sieht vor, dass der Verweis den Namen einer Regel in derselben Regeldatei referenziert.

⟨*Regel*⟩	=	'rule' ⟨*ID*⟩ ':' ⟨*RegelQuelle*⟩ ⟨*Abfrage*⟩ {⟨*Abfrage*⟩} ';'
⟨*RegelReferenz*⟩	:	Referenz auf eine bereits definierte Regel

Listing 5.13: Grammatik einer ⟨*Regel*⟩

Ein Beispiel für eine Regel ist in Listing 5.14 gegeben. In Zeile 1 wird ein Metamodell importiert, damit die folgende Regel Zugriff auf die Metaklassen darin hat. In Zeile 4 ist eine Regel mit der Bezeichnung *A* definiert. Die Regel besteht aus einer Regel-Quelle, die Modellelemente selektiert, welche die Metaklasse *ComposedStructure* aus dem importierten Metamodell besitzen. Außerdem besitzt die Regel eine Abfrage, welche aus den selektierten Modellelementen jene Modellelemente ermittelt, die über das Strukturmerkmal *assemblyContexts__ComposedStructure* erreichbar sind.

Das Beispiel soll lediglich einen Überblick geben, wie eine Regel als Ganzes aussieht. Die Bestandteile Regel-Quelle und Abfrage werden in den folgenden Abschnitten im Detail thematisiert.

```
import "http://palladiosimulator.org/PalladioComponentModel/5.2" as pcm

// rule named A with a metaclass-source and a single lookup
rule A: metaclass(pcm::ComposedStructure) -> feature(
    ↪ assemblyContexts__ComposedStructure);
```

Listing 5.14: Beispiel einer ⟨*Regel*⟩

5.3.2 Selektion der Eingabeelemente

Der erste Teil jedes Regelrumpfs ist die Regelquelle. Diese dient der Selektion der Eingabeelemente. Mit der Regelquelle wird festgelegt, für welche Modellelemente die Regel eine Änderungsausbreitung definiert. Die selektierten Modellelemente besitzen einen gemeinsamen **Quelltyp** (engl. source type). Das bedeutet, dass jedes der selektierten Modellelemente entweder genau dem Quelltyp oder einer Unterklasse des Quelltyps entspricht. Der Quelltyp ist immer zugleich der Eingabetyp der Regel.

5.3.3 Definition der Änderungsausbreitung

Der zweite Teil jedes Regelrumpfs ist eine Folge von mindestens einer Abfrage. Abfragen sind die atomaren Bausteine zur Definition von Änderungsausbreitungen innerhalb von Regeln. Sie ermöglichen die Ermittlung der markierten Elemente ausgehend von der Regelquelle. Jede Abfrage bekommt eine Eingabemenge an Elementen und transformiert diese in eine Ausgabemenge. Dabei werden, je nach Abfrageart, unterschiedliche Operationen auf der Eingabemenge ausgeführt.

Der Unterschied zwischen einer Regel und einer Abfrage ist, dass die Ausgabemenge einer Abfrage standardmäßig nicht der temporären Ergebnismenge hinzugefügt wird. Lediglich die Ausgabemenge der letzten Abfrage ist der Rückgabewert der Regel und wird somit der temporären Ergebnismenge hinzugefügt. Innerhalb einer Regel stellt die Ausgabe einer Abfrage stets die Eingabe der ihr folgenden Abfrage dar.

Als die folgende Abfrage wird die Abfrage rechts von der aktuell betrachteten Abfrage bezeichnet. Abfragen werden somit von links nach rechts ausgewertet. Die erste Abfrage einer Regel hat keine vorangehende Abfrage. Deren Eingabe wird von der Regelquelle übergeben.

Abfragen besitzen ebenso wie Regeln einen **Eingabetyp** und einen **Ausgabetyp**. Die statische Semantik der Sprache erfordert, dass der Ausgabetyp einer Abfrage dem Eingabetyp der ihr folgenden Abfrage entspricht. Der Eingabetyp der ersten Abfrage muss dem Quelltyp entsprechen und der Ausgabetyp der letzten Abfrage definiert den Ausgabetyp der zugehörigen Regel.

5.4 Regelquelle

Die Regelquelle ist eine spezielle Art von Abfrage, die als Eingabe die temporäre Ergebnismenge besitzt und den Anfang der Abfragenfolge bildet. Die Aufgabe einer Regelquelle ist es, bestimmte Elemente aus der temporären Ergebnismenge herauszufiltern und den darauf folgenden Abfragen zur Verfügung zu stellen. Als Begriff für die Menge der herausgefilterten Elemente wird die Bezeichnung **Selektionsmenge** verwendet. Alle Elemente der Selektionsmenge müssen einen gemeinsamen Typ besitzen. Die Regelquelle ist somit typisiert durch den sogenannten **Quelltyp**. Der allgemeinste gemeinsame Quelltyp zweier beliebiger Modellelemente ist *org.eclipse.emf.ecore.EObject*. Das Beispiel in Listing 5.15 zeigt, wie man damit alle Elemente in der temporären Ergebnismenge selektieren kann. Der Quelltyp hängt von der Selektionsmethode der Regelquelle ab. Wird beispielsweise eine bestimmte Metaklasse zur Selektion verwendet, so entspricht der Quelltyp dieser Metaklasse. Wird hingegen die Ausgabe einer anderen Regel als Quelle verwendet, so entspricht der Quelltyp dem Ausgabetyp dieser Regel.

Abschnitt 5.4.1 nennt die verschiedenen Arten einer Regelquelle. Die darauf folgenden Abschnitte erläutern die verschiedenen Arten einer Regelquelle:

Abschnitt 5.4.2 erläutert, wie nach Metaklasse selektiert werden kann.

Abschnitt 5.4.3 erläutert, wie auf Modellebene selektiert werden kann.

Abschnitt 5.4.4 erläutert die Verwendung der Ausgabe einer anderen Regel als Quelle.

```
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

// rule named A which selects all elements from temporary result set
rule A: metaclass(ecore::EObject) -> ...
```

Listing 5.15: Beispiel einer ⟨*RegelQuelle*⟩, die alle Elemente der temporären Ergebnismenge selektiert

Das Beispiel in Listing 5.15 veranschaulicht eine bestimmte Art von Regelquelle. In Zeile 1 wird zunächst das Ecore-Metamodell importiert. In Zeile 4 wird die Regel mit der Bezeichnung *A* definiert. Die Regel besteht aus einer Regel-Quelle. Der Rest der Regel ist durch Auslassungspunkte gekennzeichnet, da er für dieses Beispiel unerheblich ist. Die angegebene Regel-Quelle selektiert alle Modellelemente, die entweder den gleichen Typ oder einen Untertyp der referenzierten Metaklasse besitzen. Die Regelquelle selektiert alle Modellelemente in der temporären Ergebnismenge. Das ist der Fall, da für alle verfügbaren Eingabeelemente die referenzierte Klasse *EObject* aus dem Metamodell *ecore* ein Obertyp darstellt.

Dies ist eine von drei Arten einer Regel-Quelle. Alle drei Arten werden in den folgenden drei Abschnitten mit Beispielen erläutert.

5.4.1 Aufbau einer Regelquelle

Eine Regelquelle kann auf eine der drei folgenden Arten beschrieben werden.
Sie ist entweder ⟨*MetaklasseRegelQuelle*⟩, ⟨*InstanzRegelQuelle*⟩ oder ⟨*ExterneRegelQuelle*⟩.
Die zugehörige Grammatik ist in Listing 5.16 notiert.

Die verschiedenen Arten unterscheiden sich durch die jeweils verwendete Selektionsmethode und werden im Folgenden thematisiert.

```
⟨RegelQuelle⟩ = ⟨MetaklasseRegelQuelle⟩
              | ⟨InstanzRegelQuelle⟩
              | ⟨ExterneRegelQuelle⟩
```

Listing 5.16: Grammatik einer ⟨*RegelQuelle*⟩

5.4.2 Selektion basierend auf Metaklasse

Die ⟨*MetaklasseRegelQuelle*⟩ wird verwendet, um Elemente auf Basis ihres Typs aus der temporären Ergebnismenge herauszufiltern. Diese Art der Regelquelle besteht aus einer ⟨*MetaklasseReferenz*⟩. Die zugehörige Grammatik ist in Listing 5.17 notiert.

Der zu filternde Typ entspricht dem Typ der referenzierten Metaklasse. Er ist zugleich der Quelltyp. Die dynamische Semantik dieser Regelquelle definiert, dass jedes Element aus der temporären Ergebnismenge in die Selektionsmenge eingefügt wird, wenn es a) den gleichen Typ besitzt wie die referenzierte Metaklasse oder b) einen Untertyp davon.

⟨*MetaklasseRegelQuelle*⟩ = 'metaclass(' ⟨*MetaklasseReferenz*⟩ ')'

Listing 5.17: Grammatik einer ⟨*RegelQuelle*⟩, die nach Metaklasse filtert

Ein Beispiel für die konkrete Syntax einer ⟨*MetaklasseRegelQuelle*⟩ ist in Listing 5.18 gegeben. In Zeile 1 wird das PCM importiert. In Zeile 4 ist beispielhaft ein Teil einer Regel mit der Bezeichnung *B* definiert. Der Teil besteht aus einer Regel-Quelle, die nach Metaklasse selektiert. Es werden alle Modellelemente selektiert, welche eine Instanz der Metaklasse *ComposedStructure* aus dem importierten PCM sind.

```
import "http://palladiosimulator.org/PalladioComponentModel/5.2" as pcm

// rule which selects all ComposedStructures from temporary result set
rule B: metaclass(pcm::ComposedStructure) -> ...
```

Listing 5.18: Beispiel einer ⟨*RegelQuelle*⟩, die nach Metaklasse filtert

5.4.3 Selektion basierend auf Instanz

Die ⟨*InstanzRegelQuelle*⟩ wird verwendet, um bestimmte Instanzen auf der Modellebene aus der temporären Ergebnismenge herauszufiltern. Diese Art der Regelquelle besteht aus einer ⟨*InstanzDeklarationReferenz*⟩. Die zugehörige Grammatik ist in Listing 5.19 notiert.

Der Quelltyp entspricht dem Typ der referenzierten Instanz-Deklaration. Die dynamische Semantik dieser Regelquelle definiert, dass jedes Modellelement aus der temporären Ergebnismenge in die Selektionsmenge eingefügt wird, wenn es in der referenzierten Instanz-Deklaration vorkommt. Das bedeutet insbesondere, dass der Typ des Modellelements entweder dem Quelltyp oder dessen Untertyp entsprechen muss.

⟨*InstanzRegelQuelle*⟩ = 'instance(' ⟨*InstanzDeklarationReferenz*⟩ ')'

Listing 5.19: Grammatik einer ⟨*RegelQuelle*⟩, die nach Instanz filtert

Ein Beispiel für die ⟨*InstanzRegelQuelle*⟩ ist in Listing 5.20 gegeben. In Zeile 1 wird ein Modell mit der Bezeichnung *repo* importiert. Zeile 4 beinhaltet eine Instanz-Deklaration mit

der Bezeichnung *MyComponent*. Die Instanz-Deklaration beinhaltet ein Modellelement mit der generierten ID *MyComponent__TVJd022yEeipyr94Phon1Q* aus dem importierten Modell *repo*. In Zeile 7 ist eine Regel mit der Bezeichnung *B* definiert, deren Quelle alle Modellelemente aus der Instanz-Deklaration *MyComponent* selektiert. Das bedeutet, dass wenn ein Modellelement mit der generierten ID *MyComponent__TVJd022yEeipyr94Phon1Q* in der temporären Ergebnismenge vorhanden ist, dieses von der Quelle der Regel *B* selektiert wird.

```
import-model "/MyProject/example.repository" as repo

// instance with id _TVJd022yEeipyr94Phon1Q and type pcm::CompositeComponent
instance MyComponent: repo::MyComponent__TVJd022yEeipyr94Phon1Q

// rule which selects "MyComponent" from temporary result set
rule B: instance(MyComponent) -> ...
```

Listing 5.20: Beispiel einer ⟨*RegelQuelle*⟩, die nach Instanz filtert

5.4.4 Selektion der Ausgabe einer anderen Regel

Die ⟨*ExterneRegelQuelle*⟩ dient der Wiederverwendung von bestehender Logik. Sie besteht aus einer ⟨*RegelReferenz*⟩. Die zugehörige Grammatik ist in Listing 5.21 notiert.

Die ⟨*RegelReferenz*⟩ muss auf eine andere Regel verweisen, die im Folgenden *aufgerufenen Regel* genannt wird. Zusätzlich besagt die statische Semantik, dass Regeln nicht derart aufeinander verweisen dürfen, dass sich ein Zyklus bildet. Die dynamische Semantik beschreibt, dass diese Art der Regelquelle nicht direkt auf der temporären Ergebnismenge operiert, sondern stattdessen eine andere Regel aufruft. Die aufgerufene Regel operiert dabei auf der aktuellen temporären Ergebnismenge der aufrufenden Regel. Die Ausgabe der aufgerufenen Regel wird als Selektionsmenge der Regelquelle verwendet. Der Ausgabetyp der aufgerufenen Regel stellt den Quelltyp der aufrufenden Regel dar.

⟨*ExterneRegelQuelle*⟩ = 'rule(' ⟨*RegelReferenz*⟩ ')'

Listing 5.21: Grammatik einer ⟨*RegelQuelle*⟩ für das Aufrufen einer externen Regel

Ein Beispiel für die ⟨*ExterneRegelQuelle*⟩ ist in Listing 5.22 gegeben. Das Beispiel verdeutlicht anhand der fünf Regeln *C* bis *G*, dass es nicht erlaubt ist, einen Zyklus zu bilden.

Die Regel *C* besitzt eine gültige Quelle. Sie referenziert die Regel *D*. Regel *D* wiederum verwendet beliebige Sprachelemente, ohne sich selber oder die Regel *C* zu referenzieren. Wird Regel *C* während der Änderungsausbreitungsanalyse ausgeführt, so ruft sie anfangs

Regel *D* auf und wartet deren Ergebnis ab. Das Ergebnis der Evaluierung von Regel *D* wird an die Quelle der Regel *C* zurückgegeben und als deren Selektionsmenge verwendet.

Die Regel *E* enthält eine ungültige Quelle. Es ist nicht erlaubt einen Zyklus zu bilden, indem eine Regel sich selbst referenziert.

Die Regeln *F* und *G* sind ungültig, da sie sich gegenseitig durch die Regel-Quelle referenzieren und somit einen Zyklus bilden.

```
// valid: rule C calls rule D and uses its result
rule C: rule(D) -> ...
rule D: ...

// error: a rule which is calling itself is not allowed
rule E: rule(E) -> ...

// error: the rules form a cycle: F->G->F
rule F: rule(G) -> ...
rule G: rule(F) -> ...
```

Listing 5.22: Beispiel für mehrere ⟨*RegelQuellen*⟩, die eine andere Regel aufrufen

5.5 Abfrage

Eine Abfrage ist der atomare Baustein zur Definition von Änderungsausbreitungen innerhalb einer Regel. Die Bedeutung einer Abfrage für eine Regel und die Abgrenzung zu dieser wurden in Abschnitt 5.3.3 beschrieben. Dort wurde ebenfalls beschrieben, dass es verschiedene Abfragearten gibt, die sich in der Operation unterscheiden, welche sie auf den Eingabeelementen ausführen. Jede Abfrageart modifiziert die Menge der betrachteten Elemente innerhalb der Regelausführung.

Abschnitt 5.5.1 nennt die verschiedenen Arten einer Abfrage. Die darauf folgenden Abschnitte erläutern die verschiedenen Arten einer Abfrage.

Abschnitt 5.5.2 erläutert wie durch Navigation neue Modellelemente innerhalb einer Regel ermittelt werden können.

Abschnitt 5.5.3 erläutert wie die Menge der betrachteten Modellelemente innerhalb einer Regel eingeschränkt werden kann.

Abschnitt 5.5.4 erläutert die Abfrage einer anderen Regel durch einen Regel-Aufruf.

5.5.1 Aufbau einer Abfrage

Der Aufbau einer Abfrage unterscheidet sich je nach deren Art. Es gibt drei verschiedene Arten von Abfragen: ⟨*Navigation*⟩, ⟨*Selektion*⟩ und ⟨*RegelAufruf*⟩. Der Navigation kann

optional eine ⟨*Verursachungselementmarkierung*⟩ vorangestellt werden. Die zugehörige Grammatik ist in Listing 5.23 notiert.

```
⟨Abfrage⟩ = [⟨Verursachungselementmarkierung⟩] ⟨Navigation⟩
          | ⟨Selektion⟩
          | ⟨RegelAufruf⟩
```

Listing 5.23: Grammatik einer ⟨*Abfrage*⟩

5.5.2 Ermittlung anderer Elemente durch Navigation

Die ⟨*Navigation*⟩ ist eine Abfrageart. Ihre zugehörige Grammatik ist in Listing 5.24 notiert.

Eine ⟨*Navigation*⟩ wird verwendet, um andere Modellelemente aus den Eingabeelementen zu ermitteln. Dazu werden Referenzen zwischen Modellelementen betrachtet. Die ermittelten Modellelementen werden als Ausgabeelemente der Abfrage zurückgegeben. Die ursprünglichen Eingabeelemente werden nicht explizit der Ausgabe hinzugefügt. Somit wird die Menge der Eingabeelemente durch eine neue Menge an referenzierten Modellelementen ersetzt. Die Art und Weise wie die Eingabemenge in die Ausgabemenge umgewandelt wird, ist durch die Navigationsperspektive und deren Navigationsbedingung gegeben, welche im Folgenden genauer erläutert werden.

Es gibt die folgenden zwei **Navigationsperspektiven**: Vorwärtsnavigation und Rückwärtsnavigation. Diese stellen zugleich die beiden Arten der Navigation dar. Eine Navigationsperspektive legt fest, von welchem Modellelement die Navigation ausgeht, also welches das referenzierende Element ist.

Bei der ⟨*Vorwärtsnavigation*⟩ werden die Referenzen ausgehend von den Eingabeelementen verfolgt, indem der Inhalt ihrer Strukturmerkmale abgerufen wird. Es sei E die Eingabemenge einer Abfrage, A die Ausgabemenge einer Abfrage und X die Menge der Modellelemente im gesamten Architekturmodell. Die Vorwärtsnavigation ermittelt für ein bestimmtes Modellelement $e \in E$ die Menge M_e an Modellelementen, auf die e durch den Inhalt eines beliebigen Strukturmerkmals s verweist:

$$M_e = \{x \in X \mid e \underset{s}{\longrightarrow} x\}$$

Die Ausgabemenge ist die Vereinigung aller M_e für jedes Eingabeelement e:

$$A = \bigcup_{e \in E} M_e$$

Die entgegengesetzte Perspektive ist die ⟨*Rückwärtsnavigation*⟩. Es sei E die Eingabemenge einer Abfrage, A die Ausgabemenge einer Abfrage und X die Menge der Modellelemente im gesamten Architekturmodell. Die Rückwärtsnavigation ermittelt als Ausgabemenge jedes Modellelement e aus der Eingabemenge E, auf das ein Modellelement x aus der

Menge X der Modellelemente im Architekturmodell, durch den Inhalt eines beliebigen Strukturmerkmals s verweist:

$$A = \{e \in E \mid \exists x \in X : e \underset{s}{\leftarrow} x\}$$

Die verfügbaren **Navigationsbedingungen** werden durch die jeweilige Navigationsperspektive bestimmt. Eine Navigationsbedingung legt fest, welche Referenzen berücksichtigt werden sollen. Die Abschnitte 5.6 und 5.7 thematisieren die einzelnen Navigationsperspektiven und deren Navigationsbedingungen im Detail und enthalten Beispiele.

⟨*Navigation*⟩	=	⟨*Vorwärtsnavigation*⟩
	|	⟨*Rückwärtsnavigation*⟩

Listing 5.24: Grammatik einer ⟨*Navigation*⟩

5.5.3 Filterung der Elemente durch Selektion

Die ⟨*Selektion*⟩ ist eine Abfrageart. Ihre zugehörige Grammatik ist in Listing 5.25 notiert.

Eine ⟨*Selektion*⟩ wird verwendet, um die Menge an aktuell betrachteten Modellelementen innerhalb ihrer Regel einzuschränken. Eine Selektion enthält somit stets eine Menge C mit mindestens einer **Selektionsbedingung**, welche definiert, ob ein Modellelement aus der Eingabemenge in die Ausgabemenge übernommen wird.

Es sei X die Menge der Modellelemente im gesamten Architekturmodell. Eine Selektionsbedingung $c \in C$ wird durch ein Prädikat $c : X \mapsto \{wahr, falsch\}$ dargestellt.

Es sei E die Eingabemenge einer Selektion. Die Ausgabemenge A ergibt sich als

$$A = \{e \in E \mid \forall c \in C : c(e)\}.$$

Eine Selektion besitzt zudem einen **Selektionstyp** T. Der Selektionstyp stellt den Ausgabetyp der Selektion dar. Damit sichergestellt werden kann, dass jedes Ausgabelement entweder dem Selektionstyp oder einer Unterklasse davon entspricht, wird eine spezielle Selektionsbedingung c_1 zu C hinzugefügt:

$$c_1(e) = \text{"Modellelement } e \text{ besitzt als Typ } T \text{ oder eine Unterklasse davon."}$$

Im Folgenden werden die drei verschiedenen Selektionsarten ⟨*TypSelektion*⟩, ⟨*InstanzSelektion*⟩ und ⟨*InlineInstanzPrädikatSelektion*⟩ thematisiert.

Eine ⟨*TypSelektion*⟩ ist vergleichbar mit einem Typecasting. Diese Art der Selektion wird dazu verwendet, um einen bestimmten Typ an Elementen zu erlangen. Der Typ wird durch eine ⟨*JvmTypReferenz*⟩ dargestellt und ist zugleich der Selektionstyp. Die ⟨*TypSelektion*⟩ verwendet dazu nur eine Selektionsbedingung: $C = \{c_1\}$.

Es gibt zwei Arten einer ⟨*TypSelektion*⟩: ⟨*AllgemeineTypSelektion*⟩ und ⟨*SubtypSelektion*⟩. Diese Unterscheidung wurde eingeführt, um die Verwendung der Autovervollständigung

für die Sprache zu verbessern. Der Unterschied liegt nur in der statischen Semantik der beiden Sprachelemente. Das bedeutet, dass beide Arten der ⟨*TypSelektion*⟩ sich gleich verhalten und sich nur darin unterscheiden, welche Metaklasse als Selektionstyp angegeben werden darf.

Die ⟨*AllgemeineTypSelektion*⟩ erlaubt einen beliebigen Selektionstyp, während bei der ⟨*SubtypSelektion*⟩ der Selektionstyp ein Untertyp des Eingabetyps sein muss. Wenn der Benutzer der Sprache eine Unterklasse für den aktuellen Eingabetyp verwenden möchte, kann es vorkommen, dass er keinen passenden Untertyp kennt. Verwendet er die Subtyp-Selektion, so schlägt die Laufzeitumgebung der Sprache dem Benutzer die verfügbaren Unterklassen mittels Autovervollständigung vor. Ein Beispiel für die Typ-Selektion ist in Listing 5.26 zu finden.

Eine ⟨*JvmTypReferenz*⟩ ist ein Querverweis auf einen Java-Typ [GJS+18]. Es wurde sich auf Klassen- und Interface-Typen beschränkt, da hauptsächlich diese durch Ecore für Metaklassen generiert werden. Die Referenz besteht aus dem Nichtterminal ⟨*QualifizierterName*⟩. Die statische Semantik erfordert, dass es sich bei dem Verweis um den voll qualifizierten Namen eines Java-Typs handelt, der auf dem Klassenpfad verfügbar ist. Ein Java-Typ ist ein externes Element, das durch den Klassenpfad der Laufzeitumgebung der Sprache zur Verfügung gestellt werden muss. Welche Java-Typen auf dem Klassenpfad verfügbar gemacht werden, kann durch das Sprachelement ⟨*JavaImport*⟩ im Kopf der Regeldatei festgelegt werden. CPRL verwendet Metaklassen, um den Typ eines Elements auszudrücken. Im Hintergrund wird der Typ der Java-Klasse verwendet, die von EMF für die Metaklasse generiert wurde. Die Verwendung einer ⟨*JvmTypReferenz*⟩ bietet also Zugriff auf eine größere Anzahl an Typen, als Metaklassen referenzierbar sind. Es gilt folglich: $MetaklassenTypenMenge \subset JavaTypenMenge$. Dies liegt daran, dass für eine Metaklasse-Referenz ein zugehöriges Metamodell vorhanden sein muss. Eine ⟨*JvmTypReferenz*⟩ kommt allerdings ohne Metamodell aus. CPRL verwendet an anderen Stellen ausschließlich Metaklassen als Typen, da dies intuitiver ist, als die zugehörigen Java-Klassen und -Interfaces zu referenzieren.

Die ⟨*InstanzSelektion*⟩ und die ⟨*InstanzPrädikatSelektion*⟩ sind Selektionsarten. Sie können verwendet werden, um Elemente auf der Modellebene zu selektieren. Die ⟨*InstanzSelektion*⟩ verweist auf eine Instanz-Deklaration. Ihr Selektionstyp entspricht dem der referenzierten Instanz-Deklaration. Sie besteht aus den folgenden zwei Selektionsbedingungen: $C = \{c_1, c_2\}$. Dabei stellt c_2 das folgende Prädikat für die referenzierte Instanz-Deklaration dar:

$$c_2(e) = \text{"Modellelement } e \text{ kommt in der referenzierten Instanz-Deklaration vor."}$$

Bei der ⟨*InstanzPrädikatSelektion*⟩ handelt es sich um syntaktischen Zucker. Sie bringt lediglich eine Vereinfachung der Schreibweise für den Fall, dass die Bedingung c_1 vernachlässigt wird. Es gilt bei dieser Art der Selektion $Eingabetyp = Selektionstyp = Ausgabetyp$. Statt einer Referenz auf eine Instanz-Deklaration, beinhaltet sie ein Prädikat P, das anhand eines Code-Blocks ausgedrückt wird. Dadurch ist es möglich, sich das vorherige Hinzufügen einer Deklaration im Regeldatei-Kopf zu sparen. Die Selektionsbedingung für die ⟨*InstanzPrädikatSelektion*⟩ lautet $C = \{c_3\}$. Dabei stellt c_3 das folgende Prädikat dar:

$$c_3(e) = P(e)$$

Die statische Semantik erfordert, dass P einen booleschen Typ als Rückgabewert besitzt. Ein Beispiel für die beiden Selektionsarten auf Modellebene findet sich in Listing 5.27.

```
⟨Selektion⟩                    =  ⟨TypSelektion⟩
                               |  ⟨InstanzSelektion⟩
                               |  ⟨InlineInstanzPrädikatSelektion⟩
⟨TypSelektion⟩                 =  ⟨AllgemeineTypSelektion⟩
                               |  ⟨SubtypSelektion⟩
⟨AllgemeineTypSelektion⟩       =  '<'
                                     ⟨JvmTypReferenz⟩
                                     { 'AND' ⟨JvmTypReferenz⟩ }
                                  '>'
⟨SubtypSelektion⟩              =  '<!'
                                     ⟨JvmTypReferenz⟩
                                     { 'AND' ⟨JvmTypReferenz⟩ }
                                  '>'
⟨JvmTypReferenz⟩               :  Referenz auf einen Java Typ
⟨InstanzSelektion⟩             =  '[' ⟨InstanzDeklarationReferenz⟩ ']'
⟨InlineInstanzPrädikatSelektion⟩ = '[' ⟨CodeBlock⟩ ']'
```

Listing 5.25: Grammatik einer ⟨*Selektion*⟩

Ein Beispiel für die Verwendung von ⟨*TypSelektionen*⟩ ist in Listing 5.26 dargestellt. In Zeile 1 wird das PCM importiert. Die Regeln mit der Bezeichnung *A* bis *D* zeigen beispielhaft, was der Unterschied zwischen einer ⟨*AllgemeinenTypSelektion*⟩ und einer ⟨*SubtypSelektion*⟩ ist.

Die Hierarchie der Metaklassen aus dem PCM ist folgendermaßen aufgebaut: Die Metaklasse *ImplementationComponentType* ist die Oberklasse von *BasicComponent* und *CompositeComponent*. Die Metaklasse *InterfaceRequiringEntity* ist die Oberklasse von *ImplementationComponentType*.

Die Regeln *A* und *B* besitzen eine korrekte statische Semantik. Die referenzierten Metaklassen in der ⟨*SubtypSelektion*⟩ sind beides Unterklassen der Metaklasse *ImplementationComponentType*.

Die Regel *C* ist semantisch nicht korrekt. Die referenzierte Metaklasse in der ⟨*SubtypSelektion*⟩ ist eine Oberklasse statt eine Unterklasse des Eingabetyps.

Die Regel *D* unterscheidet sich von der Regel *C* lediglich durch das Fehlen eines Ausrufezeichens vor dem Selektionstyp. Dadurch ist es eine ⟨*AllgemeineTypSelektion*⟩ statt eine ⟨*SubtypSelektion*⟩. Die statische Semantik ist korrekt, da eine ⟨*AllgemeineTypSelektion*⟩ einen beliebigen Selektionstyp zulässt.

```
import "http://palladiosimulator.org/PalladioComponentModel/5.2" as pcm

// each rule uses a subtype selection to select a different subclass
rule A: metaclass(pcm::ImplementationComponentType)<!org.palladiosimulator.pcm.
    ↪ repository.BasicComponent> -> ...;
rule B: metaclass(pcm::ImplementationComponentType)<!org.palladiosimulator.pcm.
    ↪ repository.CompositeComponent> -> ...;

// error: InterfaceRequiringEntity is not a subtype of rule source type
rule C: metaclass(pcm::ImplementationComponentType)<!org.palladiosimulator.pcm.
    ↪ core.entity.InterfaceRequiringEntity> -> ...;

// okay: it is a general type selection; notice lack of exclamation mark
rule D: metaclass(pcm::ImplementationComponentType)<org.palladiosimulator.pcm.
    ↪ core.entity.InterfaceRequiringEntity> -> ...;
```

Listing 5.26: Beispiel von ⟨*TypSelektionen*⟩

Ein Beispiel für die Verwendung von ⟨*Selektionen*⟩ auf Modellebene ist in Listing 5.27 dargestellt. In Zeile 1 wird das PCM importiert. Die Regeln *A* und *B* zeigen beispielhaft wann eine ⟨*InlineInstanzPrädikatSelektion*⟩ verwendet werden kann und wann stattdessen eine ⟨*InstanzSelektion*⟩ verwendet werden muss.

Die Hierarchie der Metaklassen im PCM ist folgendermaßen aufgebaut: Die Metaklasse *ImplementationComponentType* ist eine Oberklasse von *CompositeComponent*.

Die Regel *A* verwendet eine ⟨*InstanzSelektion*⟩. Diese Art der ⟨*Selektion*⟩ ist nötig, da das Prädikat auf Modellelemente mit der Metaklasse *CompositeComponent* zugreifen muss, welche einen Untertyp des Eingabetyps *ImplementationComponentType* besitzt.

Die Regel *B* hingegen besteht aus einer ⟨*InlineInstanzPrädikatSelektion*⟩, da das Prädikat in Zeile 13 auf Modellelemente mit demselben Eingabetyp *ImplementationComponentType* wie der Eingabetyp zugreifen muss. Es ist somit nicht nötig eine Instanz-Deklaration, wie in Zeile 4, für die ⟨*Selektion*⟩ in Regel *B* zu erstellen.

```
import "http://palladiosimulator.org/PalladioComponentModel/5.2" as pcm

// note: CompositeComponent is a subtype of ImplementationComponentType
find MyComponent: pcm::CompositeComponent {
  return it.entityName.endsWith("MyComponent")
}

// use an instance selection if selection and declaration type differ
rule A: metaclass(pcm::ImplementationComponentType) [MyComponent] -> ...;

// syntactic sugar using inline predicate if no type constraint in selection
rule B: metaclass(pcm::ImplementationComponentType) [{
  return it.entityName.endsWith("MyComponent")
}] -> ...;
```

Listing 5.27: Beispiel von ⟨*Selektionen*⟩ auf Modellebene

5.5.4 Wiederverwendung bestehender Logik durch Regelaufruf

Der ⟨*RegelAufruf*⟩ ist eine Abfrageart. Die zugehörige Grammatik ist in Listing 5.28 notiert.

Der ⟨*RegelAufruf*⟩ wird verwendet, um die Abfrage an eine andere Regel zu delegieren. Welche Regel gemeint ist, wird durch eine ⟨*RegelReferenz*⟩ ausgedrückt. Die Eingabeelemente des Regelaufrufs werden der referenzierten Regel als Eingabeelemente übergeben. Die Regel erhält somit die Eingabeelemente vom Regelaufruf, statt von der temporären Ergebnismenge. Die referenzierte Regel wird evaluiert und ihre Ausgabeelemente werden dem Regelaufruf zurückgegeben. Schließlich werden die Ausgabeelemente der Regel vom Regelaufruf als Ausgabeelemente zurückgegeben.

Der Regelaufruf ist semantisch damit gleichzusetzen, als seien die Quelle und die Abfragen der referenzierten Regel an Stelle des Regelaufrufs eingefügt worden. Es wird jedoch immer, wenn möglich, ein Regelaufruf verwendet, statt Code zu duplizieren. Der Regelaufruf ermöglicht folglich die Wiederverwendung bestehender Logik und die Vermeidung von Codeduplikaten. Die statische Semantik erfordert, dass der Eingabetyp der referenzierten Regel mit dem Eingabetyp des Regelaufrufs übereinstimmt. Außerdem stellt der Ausgabetyp der referenzierten Regel den Ausgabetyp des Regelaufrufs dar.

⟨*RegelAufruf*⟩ = '→' 'rule(' ⟨*RegelReferenz*⟩ ')'

Listing 5.28: Grammatik für einen ⟨*RegelAufruf*⟩

Ein Beispiel für einen ⟨*RegelAufruf*⟩ ist in Listing 5.29 gegeben. Die Regel *C* ruft die Regel *D* in ihrer ersten Abfrage in Zeile 4 auf. Die Regel *E* hat die selbe Bedeutung wie Regel *C*. Der erste Aufruf aus Regel *D* ist in Regel *E* enthalten.

```
import "http://palladiosimulator.org/PalladioComponentModel/5.2" as pcm

// rule C uses a rule reference to invoke rule D inline
rule C: metaclass(pcm::BasicComponent) -> rule(D) -> feature(
    ↪ delay_TimeSpecification);
rule D: metaclass(pcm::BasicComponent) -> metaclass(pcm::PassiveResource) ->
    ↪ feature(capacity_PassiveResource);

// rule E has equivalent meaning to rule C but is duplicating code
rule E: metaclass(pcm::BasicComponent) -> metaclass(pcm::PassiveResource) ->
    ↪ feature(capacity_PassiveResource) -> feature(delay_TimeSpecification);
```

Listing 5.29: Beispiel für einen ⟨*RegelAufruf*⟩

5.6 Vorwärtsnavigation

Die Vorwärtsnavigation ist eine bestimmte Art von Navigation. Es ist eine der beiden Navigationsperspektiven. Für die mathematischen Formulierungen in diesem Abschnitt werden die Definitionen aus Abschnitt 5.5.2 wiederverwendet. Insbesondere sei E die Eingabemenge einer Abfrage, A die Ausgabemenge einer Abfrage und X die Menge der Modellelemente im gesamten Architekturmodell. Der Relationspfeil in folgendem Kontext bedeutet, dass ein Modellelement a ein Strukturmerkmal s besitzt, dessen Inhalt das Modellelement b beinhaltet: $a \underset{s}{\longrightarrow} b$.

In Abschnitt 5.6.1 wird die Vorwärtsnavigation durch ein Beispiel visualisiert.

Abschnitt 5.6.2 nennt die verschiedenen Arten einer Vorwärtsnavigation. Die darauf folgenden Abschnitte erläutern die verschiedenen Arten einer Vorwärtsnavigation.

Abschnitt 5.6.3 erläutert die Vorwärtsnavigation basierend auf einer Metaklasse.

Abschnitt 5.6.4 erläutert die Vorwärtsnavigation basierend auf einem Strukturmerkmal.

Abschnitt 5.6.5 erläutert die Vorwärtsnavigation basierend auf einer Instanz.

5.6.1 Veranschaulichung

Das Beispiel in Abbildung 5.3 soll veranschaulichen, welche Modellelemente bei der Vorwärtsnavigation ermittelt werden. Die Richtung der Navigation wird als *vorwärts* bezeichnet, da anschaulich gesprochen alle Referenzen verfolgt werden, die von den Eingabeelementen (nach vorne) weg zeigen.

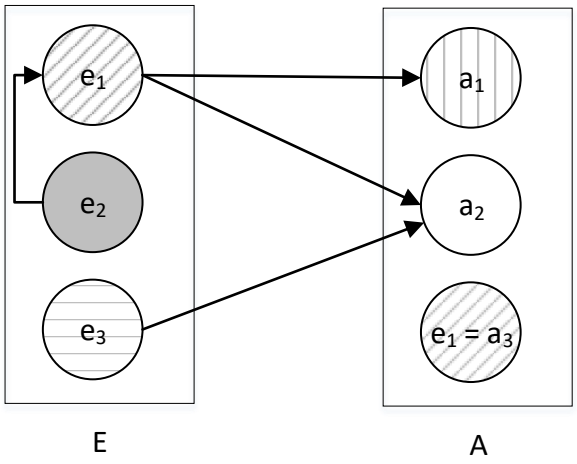

Abbildung 5.3: Beispiel zur Visualisierung der Vorwärtsnavigation

Die Identität der Modellelemente ist durch ihren Hintergrund gegeben. Kreise mit dem gleichen Muster im Hintergrund beziehen sich auf dasselbe Modellelement. Das Beispiel enthält folglich fünf unterschiedliche Modellelement und es gilt $e_1 = a_3$.

Die Pfeile stellen Referenzen zwischen den Modellelementen dar. Die Modellelemente a_1 und a_2 sind somit der Inhalt eines Strukturmerkmals von e_1. Das Modellelement a_2 ist zugleich der Inhalt des Strukturmerkmals von e_3. Das Modellelement e_1 ist der Inhalt des Strukturmerkmals von e_2.

Dieses Beispiel zeigt, dass die Vorwärtsnavigation eine Eingabemenge $E = \{e_1, e_2, e_3\}$ in die Ausgabemenge $A = \{e_1, a_1, a_2\}$ überführt. Es wird deutlich, dass die Eingabeelemente nicht explizit der Ausgabemenge hinzugefügt werden. Wenn allerdings ein Eingabeelement e_r durch ein anderes Eingabeelement e_i referenziert wird, so ist e_r dennoch Teil der Ausgabemenge. Dieser Zusammenhang wurde exemplarisch mit dem Element e_1 veranschaulicht, welches durch e_2 referenziert wird und somit Teil der Ausgabemenge ist.

Im Beispiel wurde keine Unterscheidung danach gemacht, welche Referenzen betrachtet werden. Im Folgenden wird der vollständige Aufbau einer Vorwärtsnavigation beschrieben, wobei auf Navigationsbedingungen eingegangen wird. Diese ermöglichen es, einzuschränken, welche referenzierten Elemente der Ausgabemenge hinzugefügt werden.

5.6.2 Aufbau der Vorwärtsnavigation

Eine ⟨*Vorwärtsnavigation*⟩ besteht aus einem ⟨*VorwärtsnavigationZiel*⟩. Dieses bestimmt die Navigationsbedingungen der Vorwärtsnavigation. Es gibt drei verschiedene Arten von ⟨*VorwärtsnavigationZiel*⟩, die in den nächsten Abschnitten betrachtet werden: ⟨*VorwärtsnavigationMetaklasseZiel*⟩, ⟨*VorwärtsnavigationInstanzZiel*⟩ und ⟨*VorwärtsnavigationStrukturmerkmalZiel*⟩. Eine zugehörige Grammatik ist in Listing 5.30 notiert.

⟨*Vorwärtsnavigation*⟩	=	'⟶' ⟨*VorwärtsnavigationZiel*⟩
⟨*VorwärtsnavigationZiel*⟩	=	⟨*VorwärtsnavigationMetaklasseZiel*⟩
	|	⟨*VorwärtsnavigationInstanzZiel*⟩
	|	⟨*VorwärtsnavigationStrukturmerkmalZiel*⟩

Listing 5.30: Grammatik einer ⟨*Vorwärtsnavigation*⟩

5.6.3 Navigation basierend auf Metaklasse

Die Grammatik eines ⟨*VorwärtsnavigationMetaklasseZiels*⟩ ist in Listing 5.31 notiert. Das ⟨*VorwärtsnavigationMetaklasseZiel*⟩ beinhaltet eine Referenz auf eine Metaklasse. Der Typ, den die Metaklasse darstellt, sei T.

Dieses Sprachelement stellt eine bestimmte Navigationsbedingung dar, welche den Inhalt von Strukturmerkmalen herausfiltert, der nicht einem bestimmten Typ entspricht. Formal bestimmt sich die Ausgabemenge der Navigation mit der Metaklasse-Navigationsbedingung als

$$A = \{x \in X \mid \exists e \in E : \left[(e \underset{s}{\longrightarrow} x) \wedge p(x)\right]\}$$

wobei E die Eingabemenge, A die Ausgabemenge, s ein bestimmtes Strukturmerkmal und X die Menge aller Modellelemente im Architekturmodell darstellt. Zudem ist durch $p : X \mapsto \{wahr, falsch\}$ die Navigationsbedingung als Prädikat

$$p(x) = \text{"Der Typ von } x \text{ entspricht } T \text{ oder einer Unterklasse von } T\text{."}$$

gegeben.

Als Ausgabetyp der Navigation wird T gesetzt. Die statische Semantik erfordert, dass die zum Eingabetyp zugehörige Metaklasse mindestens ein Strukturmerkmal besitzt, dessen Typ entweder genau T, eine Unterklasse von T oder eine Oberklasse von T darstellt. Andernfalls würde das Prädikat p für keines der Modellelemente jemals zu wahr evaluieren. Es wird dadurch verhindert, dass eine Vorwärtsnavigation definiert werden kann, die immer eine leere Menge an Ausgabeelementen produziert.

Die statische Semantik lässt zusätzlich eine Oberklasse von T als Typ des Strukturmerkmal zu, da ein Strukturmerkmal mit einem Obertyp O von T theoretisch auch ein Modellelement beinhalten kann, dessen Typ spezifischer als O ist.

⟨*VorwärtsnavigationMetaklasseZiel*⟩	=	'metaclass(' ⟨*MetaklasseReferenz*⟩ ')'

Listing 5.31: Grammatik einer ⟨*Vorwärtsnavigation*⟩ auf Basis der Metaklasse

Ein Beispiel für eine ⟨*Vorwärtsnavigation*⟩ auf Basis der Metaklasse ist in Listing 5.32 zu finden. In Zeile 1 wird das PCM importiert. Anschließend sind die Regeln *A* bis *D* definiert. Mit diesen Regeln wird verdeutlicht, welche Navigationen eine korrekte statische Semantik besitzen. Alle Metaklassen in diesem Beispiel stammen aus dem PCM.

Die Regel *A* verweist mittels Metaklasse-Navigation auf die Metaklasse *AssemblyContext*. Diese Metaklasse-Navigation ist erlaubt, da der Eingabetyp *CompositeComponent* ein Strukturelement besitzt, das vom Typ *AssemblyContext* ist. Alle Modellelemente, die in diesem Strukturelement beinhaltet sind, werden durch die Regel zurückgegeben.

Die Regel *B* verweist ebenfalls auf die Metaklasse *AssemblyContext*. Diese Metaklasse-Navigation ist in ihrer statischen Semantik nicht korrekt, da der Eingabetyp *BasicComponent* kein Strukturmerkmal enthält, in dessen Typhierarchie *AssemblyContext* vorkommt.

Die Regel *C* verweist auf die Metklasse *ComposedStructure*. Sie ist semantisch korrekt, da der Eingabetyp *AssemblyContext* ein Strukturmerkmal besitzt, das vom Typ *ComposedStructure* ist.

Die Regel *D* verweist auf die Metaklasse *CompositeComponent*. Sie ist semantisch korrekt, führt jedoch zu einer Warnmeldung durch den Validator der Sprache, da sie sehr unwahrscheinlich irgendwelche Modellelemente zurückgeben wird. Das Strukturmerkmal *parentStructure__AssemblyContext* der Klasse *AssemblyContext* ist von Typ *ComposedStructure*. Die Metaklasse *ComposedStructure* ist eine Oberklasse von *CompositeComponent*. Es ist also theoretisch möglich, dass ein Modellelement vom Typ *ComposedStructure* ebenfalls vom Typ *CompositeComponent* ist. Dies ist aber nicht zwingend der Fall. Deshalb wird die Warnung ausgegeben.

```
import "http://palladiosimulator.org/PalladioComponentModel/5.2" as pcm

// ok: CompositeComponent has a feature which holds AssemblyContext
rule A: metaclass(pcm::CompositeComponent) -> metaclass(pcm::AssemblyContext);

// error: no feature shares a type hierarchy with AssemblyContext
rule B: metaclass(pcm::BasicComponent) -> metaclass(pcm::AssemblyContext);

// ok: AssemblyContext has a feature which holds a ComposedStructure
rule C: metaclass(pcm::AssemblyContext) -> metaclass(pcm::ComposedStructure);

// warning: the match is unlikely, but not impossible
// a feature containing ComposedStructure might contain a CompositeComponent
rule D: metaclass(pcm::AssemblyContext) -> metaclass(pcm::CompositeComponent);
```

Listing 5.32: Beispiel von ⟨*Vorwärtsnavigationen*⟩ basierend auf Metaklasse

5.6.4 Navigation basierend auf Strukturmerkmal

Das ⟨*VorwärtsnavigationStrukturmerkmalZiel*⟩ beinhaltet eine Referenz auf ein Strukturmerkmal f. Die zugehörige Grammatik ist in Listing 5.33 notiert.

Der Typ eines Strukturmerkmals f wird als T bezeichnet. Dieses Sprachelement stellt eine bestimmte Navigationsbedingung dar, welche den Inhalt aller Strukturmerkmale außer dem von f herausfiltert. Es werden somit nur Elemente, die von Strukturmerkmal f beinhaltet sind, der Ausgabemenge hinzugefügt. Formal bestimmte sich die Ausgabemenge der Navigation mit der Strukturmerkmal-Navigationsbedingung folglich als

$$A = \{x \in X \mid \exists e \in E : \left[(e \underset{f}{\longrightarrow} x) \wedge p(x)\right]\}$$

wobei E die Eingabemenge, A die Ausgabemenge, f das referenzierte Strukturmerkmal und X die Menge aller Modellelemente im Architekturmodell darstellt. Zudem ist durch $p : X \mapsto \{wahr, falsch\}$ die Navigationsbedingung als Prädikat

$$p(x) = \text{"Der Typ von } x \text{ entspricht } T \text{ oder einer Unterklasse von } T\text{."}$$

gegeben.

Als Ausgabetyp der Navigation wird T gesetzt. Die statische Semantik erfordert, dass ein Strukturmerkmal f referenziert wird, das in der zugehörigen Metaklasse des Eingabetyps vorhanden ist. Es wird dadurch verhindert, dass eine Vorwärtsnavigation definiert werden kann, die immer eine leere Menge an Ausgabeelementen produziert.

⟨*VorwärtsnavigationStrukturmerkmalZiel*⟩	=	'feature(' ⟨*StrukturmerkmalReferenz*⟩ ')'

Listing 5.33: Grammatik einer ⟨*Vorwärtsnavigation*⟩ auf Basis des Strukturmerkmals

Ein Beispiel für eine ⟨*Vorwärtsnavigation*⟩ auf Basis des Strukturmerkmals ist in Listing 5.34 zu finden. In Zeile 1 wird das PCM importiert. Im Beispiel werden ausschließlich Metaklassen des PCMs verwendet. Das Beispiel beinhaltet die Regeln *A* und *B*. An ihnen wird veranschaulicht, welche Vorwärtsnavigationen auf Strukturmerkmal-Basis semantisch korrekt sind.

Die Regel *A* referenziert das Strukturmerkmal *connectors__ComposedStructure*. Dieses Strukturmerkmal ist in der Metaklasse *CompositeComponent*, beziehungsweise in dessen Oberklasse *ComposedStructure* vorhanden. Somit ist die Regel *A* semantisch korrekt.

Die Regel *B* referenziert das Strukturmerkmal *non_existent_feature*. Dieses Strukturmerkmal ist nicht in der Metaklasse *CompositeComponent*, beziehungsweise in einer dessen Obertypen vorhanden. Somit ist Regel *B* semantisch ungültig.

```
import "http://palladiosimulator.org/PalladioComponentModel/5.2" as pcm

// ok: connectors__ComposedStructure exists for metaclass CompositeComponent
rule A: metaclass(pcm::CompositeComponent) -> feature(
    ↪ connectors__ComposedStructure);

// error: non_existent_feature is not found for metaclass CompositeComponent
rule B: metaclass(pcm::CompositeComponent) -> feature(non_existent_feature);
```

Listing 5.34: Beispiel von ⟨*Vorwärtsnavigationen*⟩ basierend auf Strukturmerkmal

5.6.5 Navigation basierend auf Instanz

Das ⟨*VorwärtsnavigationInstanzZiel*⟩ beinhaltet eine Referenz auf eine Instanz-Deklaration d. Die zugehörige Grammatik ist in Listing 5.35 notiert.

Der Typ der Instanz-Deklaration d wird als T bezeichnet. Dieses Sprachelement stellt eine bestimmte Navigationsbedingung dar, welche den Inhalt aller Strukturmerkmale mit den Elementen der referenzierten Instanz-Deklaration abgleicht. Es werden nur jene Elemente der Ausgabemenge hinzugefügt, die durch d definiert sind. Formal bestimmt sich die Ausgabemenge der Navigation mit der Instanz-Navigationsbedingung als

$$A = \{x \in X \mid \exists e \in E : \left[(e \underset{s}{\rightarrow} x) \wedge p(x)\right]\}$$

wobei E die Eingabemenge, A die Ausgabemenge, s ein beliebiges Strukturmerkmal und X die Menge aller Modellelemente im Architekturmodell darstellt. Zudem ist durch $p : X \mapsto \{wahr, falsch\}$ die Navigationsbedingung als Prädikat

$$p(x) = \text{”Modellelement } x \text{ kommt in der referenzierten Instanz-Deklaration } d \text{ vor.”}$$

gegeben.

Als Ausgabetyp der Navigation wird T gesetzt. Die statische Semantik erfordert, dass die zum Eingabetyp zugehörige Metaklasse mindestens ein Strukturelement enthält, dessen Typ entweder T, einer Oberklasse von T oder einer Unterklasse von T entspricht. Dadurch wird sichergestellt, dass nur Instanz-Deklarationen referenziert werden können, welche potentiell ein Element der Eingabemenge enthalten könnten. Es wird dadurch verhindert, dass eine Vorwärtsnavigation definiert werden kann, die immer eine leere Menge an Ausgabeelementen produziert.

⟨*VorwärtsnavigationInstanzZiel*⟩ = 'instance('
⟨*InstanzDeklarationReferenz*⟩
')'

Listing 5.35: Grammatik einer ⟨*Vorwärtsnavigation*⟩ auf Basis einer Instanz

Ein Beispiel für eine ⟨*Vorwärtsnavigation*⟩ auf Basis der Instanz ist in Listing 5.36 zu finden. Das Beispiel verdeutlicht, welche Instanz-Deklarationen in einer ⟨*Vorwärtsnavigation*⟩ referenziert werden dürfen, um eine semantisch korrekte Regel zu erhalten.

In Zeile 1 wird das *PCM* importiert. In Zeile 2 wird ein Modell mit der Bezeichnung *repo* importiert. Die Metaklassen und Modelle in diesem Beispiel referenzieren Elemente aus diesen beiden Imports. In Zeile 4 bis 7 sind vier ⟨*InstanzPrädikatDeklarationen*⟩ definiert, die in den darauf folgenden Regeln referenziert werden.

Die Regel *A* referenziert die Instanz-Deklaration *MyComposedStructure*. Sie ist gültig, da die Metaklasse *AssemblyContext* mindestens ein Strukturmerkmal mit exakt dem Typ *ComposedStructure* der referenzierten Instanz-Deklaration besitzt. Es handelt sich um das Strukturmerkmal *parentStructure__AssemblyContext*.

Die Regel *B* referenziert die Instanz-Deklaration *MyEntity*. Sie ist gültig, da die Metaklasse *AssemblyContext* mindestens ein Strukturmerkmal mit einem Untertyp der Metaklasse *Entity* der referenzierten Instanz-Deklaration besitzt. Es handelt sich um das Strukturmerkmal *parentStructure__AssemblyContext*.

Die Regel *C* referenziert die Instanz-Deklaration *MyBasicComponent*. Sie ist gültig, da die Metaklasse *AssemblyContext* mindestens ein Strukturmerkmal mit einem Obertyp der Metaklasse *BasicComponent* der referenzierten Instanz-Deklaration besitzt. Es handelt sich um das Strukturmerkmal *parentStructure__AssemblyContext*.

Die Regel *D* referenziert die Instanz-Deklaration *MyAbstractAction*. Sie ist nicht gültig, da die Metaklasse *AssemblyContext* kein Strukturmerkmal besitzt, das in einer gemeinsamen Typhierarchie mit der Metaklasse *AbstractAction* der referenzierten Instanz-Deklaration steht.

```
import "http://palladiosimulator.org/PalladioComponentModel/5.2" as pcm
import-model "/MyProject/example.repository" as repo

find MyEntity: pcm::Entity { it.entityName.equals(...) }
find MyComposedStructure: pcm::ComposedStructure { it.entityName.equals(...) }
find MyBasicComponent: pcm::BasicComponent { it.entityName.equals(...) }
find MyAbstractAction: pcm::AbstractAction { it.entityName.equals(...) }

// ok: AssemblyContext has a feature with type ComposedStructure
rule A: metaclass(pcm::AssemblyContext) -> instance(MyComposedStructure);

// ok: Entity is a supertype of feature with type ComposedStructure
rule B: metaclass(pcm::AssemblyContext) -> instance(MyEntity);

// ok: BasicComponent is a subtype of feature with type ComposedStructure
rule C: metaclass(pcm::AssemblyContext) -> instance(MyBasicComponent);

// error: no feature shares a type hierarchy with AbstractAction
rule D: metaclass(pcm::AssemblyContext) -> instance(MyAbstractAction);
```

Listing 5.36: Beispiel von ⟨*Vorwärtsnavigationen*⟩ basierend auf Instanz

5.7 Rückwärtsnavigation

Die Rückwärtsnavigation ist eine bestimmte Art von Navigation. Es ist eine der beiden Navigationsperspektiven. Für die mathematischen Formulierungen in diesem Abschnitt werden die Definitionen aus Abschnitt 5.5.2 wiederverwendet. Insbesondere sei E die Eingabemenge einer Abfrage, A die Ausgabemenge einer Abfrage und X die Menge der Modellelemente im gesamten Architekturmodell. Der Relationspfeil in folgendem Kontext bedeutet, dass ein Modellelement a von dem Inhalt eines Strukturelements s beinhaltet wird, das zu Modellelement b gehört: $a \underset{s}{\leftarrow} b$. Das Modellelement b in diesem Kontext wird **Quellelement** genannt.

In Abschnitt 5.7.1 wird die Rückwärtsnavigation durch ein Beispiel visualisiert.

Abschnitt 5.7.2 nennt die verschiedenen Arten einer Rückwärtsnavigation. Die darauf folgenden Abschnitte erläutern die verschiedenen Arten einer Rückwärtsnavigation.

Abschnitt 5.7.3 erläutert die Rückwärtsnavigation basierend auf einer Metaklasse.

Abschnitt 5.7.4 erläutert die Rückwärtsnavigation basierend auf einem Strukturmerkmal.

Abschnitt 5.7.5 erläutert die Rückwärtsnavigation basierend auf einer Instanz.

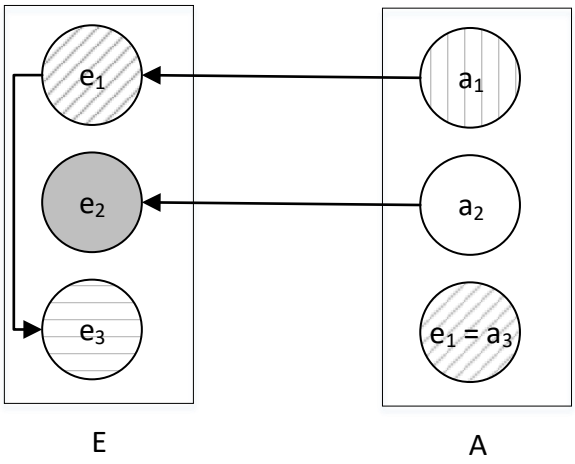

Abbildung 5.4: Beispiel zur Visualisierung der Rückwärtsnavigation

5.7.1 Veranschaulichung

Das Beispiel in Abbildung 5.4 soll veranschaulichen, welche Modellelemente bei der Rückwärtsnavigation ermittelt werden. Die Richtung wird als *rückwärts* bezeichnet, da anschaulich gesprochen alle Referenzen verfolgt werden, die zu den Eingabeelementen (zurück) zeigen.

Die Identität der Modellelemente ist durch ihren Hintergrund gegeben. Kreise mit dem gleichen Muster im Hintergrund beziehen sich auf dasselbe Modellelement. Das Beispiel enthält folglich fünf unterschiedliche Modellelemente und es gilt $e_1 = a_3$.

Die Pfeile stellen Referenzen zwischen den Modellelementen dar. Das Modellelement e_1 ist folglich Inhalt eines Strukturmerkmals von a_1. Das Modellelement e_2 ist Inhalt eines Strukturmerkmals von a_2. Das Modellelement e_3 ist Inhalt eines Strukturmerkmals von e_1.

Dieses Beispiel zeigt, dass die Rückwärtsnavigation eine Eingabemenge $E = \{e_1, e_2, e_3\}$ in die Ausgabemenge $A = \{e_1, a_1, a_2\}$ überführt.

Im Beispiel wurde keine Unterscheidung danach gemacht, welche referenzierenden Elemente a_i betrachtet werden. Im Folgenden wird der vollständige Aufbau einer Rückwärtsnavigation beschrieben. Insbesondere werden Navigationsbedingungen thematisiert. Diese ermöglichen es, einzuschränken, welche referenzierenden Elemente ausgewertet und der Ausgabemenge hinzugefügt werden. Ein referenzierendes Element wird im Folgenden Quellelement genannt.

5.7.2 Aufbau der Rückwärtsnavigation

Eine ⟨*Rückwärtsnavigation*⟩ besteht aus einer ⟨*RückwärtsnavigationQuelle*⟩. Diese bestimmt die Navigationsbedingungen der Rückwärtsnavigation. Es gibt zwei verschiedene Arten einer ⟨*RückwärtsnavigationQuelle*⟩, die in den nächsten Abschnitten betrachtet werden: ⟨*RückwärtsnavigationMetaklasseQuelle*⟩ und ⟨*RückwärtsnavigationInstanzQuelle*⟩. Eine Grammatik für die ⟨*Rückwärtsnavigation*⟩ ist in Listing 5.37 notiert.

⟨*Rückwärtsnavigation*⟩	=	'←' ⟨*RückwärtsnavigationQuelle*⟩
⟨*RückwärtsnavigationQuelle*⟩	=	⟨*RückwärtsnavigationMetaklasseQuelle*⟩
	|	⟨*RückwärtsnavigationInstanzQuelle*⟩

Listing 5.37: Grammatik einer ⟨*Rückwärtsnavigation*⟩

5.7.3 Navigation basierend auf Metaklasse

Die ⟨*RückwärtsnavigationMetaklasseQuelle*⟩ beinhaltet eine Referenz auf eine Metaklasse c. Die zugehörige Grammatik ist in Listing 5.38 notiert.

Der Typ, den die Metaklasse darstellt, sei T. Dieses Sprachelement stellt eine bestimmte Navigationsbedingung dar, welche Quellelemente ausschließt, die nicht einem bestimmten Typ entsprechen. Formal bestimmt sich die Ausgabemenge der ⟨*Rückwärtsnavigation*⟩ mit der Metaklasse-Navigationsbedingung als

$$A = \{e \in E \mid \exists x \in X : \left[(e \underset{s}{\leftarrow} x) \wedge p(x)\right]\}$$

wobei E die Eingabemenge, A die Ausgabemenge, s ein beliebiges Strukturmerkmal und X die Menge aller Modellelemente im Architekturmodell darstellt. Zudem ist durch $p : X \mapsto \{wahr, falsch\}$ die Navigationsbedingung als Prädikat

$$p(x) = \text{"Der Typ von } x \text{ entspricht } T \text{ oder einer Unterklasse von } T\text{."}$$

gegeben.

Als Ausgabetyp der Navigation wird T gesetzt. Die statische Semantik erfordert, dass die referenzierte Metaklasse c mindestens ein Strukturmerkmal besitzt, dessen Typ entweder genau den Eingabetyp, dessen Unterklasse oder dessen Oberklasse darstellt. Andernfalls würde das Prädikat p für keines der Eingabeelemente jemals zu wahr evaluieren. Es wird dadurch verhindert, dass eine Rückwärtsnavigation definiert werden kann, die immer eine leere Menge an Ausgabeelementen produziert.

⟨*RückwärtsnavigationMetaklasseQuelle*⟩	=	'metaclass(' ⟨*MetaklasseReferenz*⟩ ')'

Listing 5.38: Grammatik einer ⟨*Rückwärtsnavigation*⟩ auf Basis der Metaklasse

Ein Beispiel einer ⟨*Rückwärtsnavigation*⟩ auf Basis der Metaklasse ist in Listing 5.39 zu finden. In Zeile 1 wird das PCM importiert. Die Metaklassen in diesem Beispiel sind im PCM

enthalten. Das Beispiel veranschaulicht, welche Rückwärtsnavigationen mit Metaklassen semantisch korrekt sind.

In Regel *A* werden alle Modellelemente aus der temporären Eingabemenge markiert, die den Typ der Metaklasse *AssemblyContext* haben und von einem Modellelement vom Typ *CompositeComponent* referenziert werden. Die Regel ist semantisch korrekt, da die Metaklasse *CompositeComponent* mindestens ein Strukturmerkmal mit dem Typ *AssemblyContext* besitzt.

Die Regel *B* ist nicht korrekt, da die Metaklasse *BasicComponent* kein Strukturmerkmal besitzt, das in einer gemeinsamen Typhierarchie mit *AssemblyContext* liegt.

Die Regel *C* ist korrekt, da die Metaklasse *AssemblyContext* das Strukturmerkmal *parentStructure__AssemblyContext* mit dem Typ *ComposedStructure* beinhaltet. Die Metaklasse *CompositeComponent* stellt einen Untertyp von *ComposedStructure* dar. Ein Modellelement, das Inhalt des Strukturmerkmals ist, kann somit theoretisch auch vom Typ *CompositeComponent* sein. Da dies allerdings nicht garantiert ist, wird vom Validator eine Warnung ausgegeben.

```
import "http://palladiosimulator.org/PalladioComponentModel/5.2" as pcm

// ok: CompositeComponent has a feature which holds AssemblyContext
rule A: metaclass(pcm::AssemblyContext) <- metaclass(pcm::CompositeComponent);

// error: no feature shares a type hierarchy with AssemblyContext
rule B: metaclass(pcm::AssemblyContext) <- metaclass(pcm::BasicComponent);

// warning: the match is unlikely, but not impossible
// a feature containing ComposedStructure might contain a CompositeComponent
rule C: metaclass(pcm::CompositeComponent) <- metaclass(pcm::AssemblyContext);
```

Listing 5.39: Beispiel von ⟨*Rückwärtsnavigationen*⟩ basierend auf Metaklasse

5.7.4 Navigation basierend auf Metaklasse und Strukturmerkmal

In diesem Abschnitt wird das Sprachelement ⟨*RückwärtsnavigationMetaklasseQuelle*⟩ aus dem vorherigen Abschnitt präzisiert, indem eine optionale ⟨*StrukturmerkmalReferenz*⟩ hinzugefügt wird. Die zugehörige Grammatik ist in Listing 5.40 notiert.

Es kann vorkommen, dass zusätzlich zu der Einschränkung des Typs, auch das Strukturmerkmal des Quellelements eingeschränkt werden soll. Das ist insbesondere dann nützlich, wenn das Quellelement mehrere Strukturmerkmale mit demselben Typ oder einem Untertyp des jeweils anderen, besitzt.

Um diese Einschränkung zu ermöglichen, wurde der ⟨*RückwärtsnavigationMetaklasseQuelle*⟩ eine optionale ⟨*StrukturmerkmalReferenz*⟩ hinzugefügt. Es sei f das referenzierte Strukturmerkmal, c die referenzierte Metaklasse und T der zugehörige Typ der Metaklasse.

Formal bestimmt sich die Ausgabemenge der ⟨*Rückwärtsnavigation*⟩ mit der Metaklasse-Strukturmerkmal-Navigationsbedingung als

$$A = \{e \in E \mid \exists x \in X : \left[(e \underset{f}{\leftarrow} x) \wedge p(x)\right]\}$$

wobei E die Eingabemenge, A die Ausgabemenge, f das referenzierte Strukturmerkmal und X die Menge aller Modellelemente im Architekturmodell darstellt. Zudem ist durch $p : X \mapsto \{wahr, falsch\}$ die Navigationsbedingung als Prädikat

$$p(x) = \text{"Der Typ von } x \text{ entspricht } T \text{ oder einer Unterklasse von } T\text{."}$$

gegeben.

Als Ausgabetyp der Navigation wird T gesetzt. Die statische Semantik erfordert, dass ein Strukturmerkmal f referenziert wird, dass in c vorhanden ist. Der Typ von f muss entweder dem Eingabetyp, dessen Obertyp oder dessen Untertyp entsprechen. Es wird dadurch verhindert, dass eine Rückwärtsnavigation definiert werden kann, die immer eine leere Menge an Ausgabeelementen produziert.

⟨*RückwärtsnavigationMetaklasseQuelle*⟩ = 'metaclass('
⟨*MetaklasseReferenz*⟩
[',' ⟨*StrukturmerkmalReferenz*⟩]
')'

Listing 5.40: Grammatik einer ⟨*Rückwärtsnavigation*⟩ auf Basis von Metaklasse und Strukturmerkmal

Ein Beispiel für die ⟨*Rückwärtsnavigation*⟩ basierend auf Metaklasse und Strukturmerkmal ist in Listing 5.41 zu finden. In Zeile 1 wird das PCM importiert. Alle Metaklassen in diesem Beispiel sind im PCM enthalten. Das Beispiel verdeutlicht, welche Kombinationen aus Metaklasse und Strukturmerkmal semantisch korrekt sind.

In Regel *A* werden alle Modellelemente vom Typ *RepositoryComponent* ermittelt, die durch das Strukturmerkmal *encapsulatedComponent__AssemblyContext* von Modellelementen des Typs *AssemblyContext* beinhaltet werden. Regel *A* ist semantisch korrekt.

Die Regel *B* verweist in der ⟨*RückwärtsnavigationMetaklasseQuelle*⟩ auf ein Strukturmerkmal mit der Bezeichnung *non_existent_feature*, welches nicht in der Metaklasse *AssemblyContext* existiert.

Die Regel *C* verweist in der ⟨*RückwärtsnavigationMetaklasseQuelle*⟩ auf das Strukturmerkmal *parentStructure__AssemblyContext*, welches den Typ *ComposedStructure* besitzt, der nicht in einer gemeinsamen Typhierarchie mit *RepositoryComponent* liegt.

Die Regeln *B* und *C* sind somit semantisch nicht korrekt.

```
import "http://palladiosimulator.org/PalladioComponentModel/5.2" as pcm

// ok: encapsulatedComponent__AssemblyContext is of type RepositoryComponent
rule A: metaclass(pcm::RepositoryComponent) <- metaclass(pcm::AssemblyContext,
    ↪ encapsulatedComponent__AssemblyContext);

// error: no feature with name 'non_existent_feature' available
rule B: metaclass(pcm::RepositoryComponent) <- metaclass(pcm::AssemblyContext,
    ↪ non_existent_feature);

// error: parentStructure__AssemblyContext never holds a RepositoryComponent
rule C: metaclass(pcm::RepositoryComponent) <- metaclass(pcm::AssemblyContext,
    ↪ parentStructure__AssemblyContext);
```

Listing 5.41: Beispiel von ⟨*Rückwärtsnavigationen*⟩ basierend auf Metaklasse und Strukturmerkmal

5.7.5 Navigation basierend auf Instanz

Die ⟨*RückwärtsnavigationInstanzQuelle*⟩ besteht aus einer Referenz auf eine Instanz-Deklaration d. Die zugehörige Grammatik ist in Listing 5.42 notiert.

Der Typ der Instanz-Deklaration d wird als T bezeichnet. Dieses Sprachelement stellt eine bestimmte Navigationsbedingung dar, welche jene Instanzen der Ausgabemenge hinzufügt, die durch d deklariert sind und ein Modellelement aus der Eingabemenge durch ein beliebiges Strukturmerkmal referenzieren. Formal bestimmt sich die Ausgabemenge der ⟨*Rückwärtsnavigation*⟩ mit der Instanz-Navigationsbedingung als

$$A = \{e \in E \mid \exists x \in X : \left[(e \underset{s}{\leftarrow} x) \wedge p(x)\right]\}$$

wobei E die Eingabemenge, A die Ausgabemenge, s ein beliebiges Strukturmerkmal und X die Menge aller Modellelemente im Architekturmodell darstellt. Zudem ist durch $p : X \mapsto \{wahr, falsch\}$ die Navigationsbedingung als Prädikat

$$p(x) = \text{”Modellelement } x \text{ kommt in der referenzierten Instanz-Deklaration } d \text{ vor.”}$$

gegeben.

Als Ausgabetyp der Navigation wird T gesetzt. Die statische Semantik erfordert, dass die zugehörige Metaklasse zu T ein Strukturelement enthält, dessen Typ entweder dem Eingabetyp, dessen Oberklasse oder dessen Unterklasse entspricht. Dadurch wird sichergestellt, dass nur Instanz-Deklarationen referenziert werden können, welche potentiell ein Element der Eingabemenge enthalten könnten. Es wird dadurch verhindert, dass eine Rückwärtsnavigation definiert werden kann, die immer eine leere Menge an Ausgabeelementen produziert.

⟨*RückwärtsnavigationInstanzQuelle*⟩ = 'instance(' ⟨*InstanzDeklarationReferenz*⟩ ')'

Listing 5.42: Grammatik einer ⟨*Rückwärtsnavigation*⟩ auf Basis einer Instanz

Ein Beispiel einer ⟨*Rückwärtsnavigation*⟩ auf Basis einer Instanz ist in Listing 5.43 zu finden. Das Beispiel verdeutlicht, welche Instanz-Deklarationen in einer ⟨*Rückwärtsnavigation*⟩ referenziert werden dürfen, um eine semantisch korrekte Regel zu erhalten.

In Zeile 1 wird das *PCM* importiert. In Zeile 2 wird ein Modell mit der Bezeichnung *repo* importiert. Die Metaklassen und Modelle in diesem Beispiel referenzieren Elemente aus diesen beiden Imports. In Zeile 4 und 5 sind zwei ⟨*InstanzPrädikatDeklarationen*⟩ definiert, die in den darauf folgenden Regeln referenziert werden.

Die Regel *A* markiert alle Modellelemente aus der temporären Ergebnismenge, die von einem Modellelement aus der Instanz-Deklaration *MyComposedStructure* durch ein beliebiges Strukturmerkmal referenziert werden. Die Regel ist semantisch korrekt, da der Typ der referenzierten Instanz-Deklaration *ComposedStructure* durch das Strukturmerkmal *assemblyContexts__ComposedStructure* mindestens ein Strukturmerkmal vom Typ *AssemblyContext* besitzt.

Die Regel *B* ist semantisch nicht korrekt, da der Typ *AbstractAction* der referenzierten Instanz-Deklaration *MyAbstractAction* kein Strukturmerkmal hat, dessen Typ eine gemeinsame Hierarchie mit *AssemblyContext* besitzt.

```
import "http://palladiosimulator.org/PalladioComponentModel/5.2" as pcm
import-model "/MyProject/example.repository" as repo

find MyComposedStructure: pcm::ComposedStructure { it.entityName.equals(...) }
find MyAbstractAction: pcm::AbstractAction { it.entityName.equals(...) }

// ok: ComposedStructure has a feature with type AssemblyContext
rule A: metaclass(pcm::AssemblyContext) <- instance(MyComposedStructure);

// error: no feature of AbstractAction contains an AssemblyContext
rule B: metaclass(pcm::AssemblyContext) <- instance(MyAbstractAction);
```

Listing 5.43: Beispiel von ⟨*Rückwärtsnavigationen*⟩ basierend auf Instanz

5.8 Verursachungselementmarkierung

In den vorherigen Abschnitten wurde beschrieben, dass eine Regel eine Menge an Ausgabeelementen als das Ergebnis ihrer Ausführung zurückgibt. Dies war eine vereinfachte

Darstellung. Genau genommen, besteht die Regelausgabe aus einer Menge an Zuordnungen (engl. mappings) zwischen einem Ausgabeelement und der Menge an Verursachern. Im Folgenden wird diese Zuordnung als **Ergebnis-Verursacher-Zuordnung** bezeichnet. Ein Verursacher ist ein Modellelement, das dafür verantwortlich ist, dass das Ergebnis zu Stande gekommen ist. Es soll dem Endbenutzer einen Hinweis darauf liefern, warum ein bestimmtes Modellelement in der endgültigen Ergebnismenge gelandet ist.

Die ⟨*Verursachungselementmarkierung*⟩ wird verwendet, um eine beliebige Anzahl an Navigationen innerhalb einer Regel zu markieren. Die zugehörige Grammatik ist in Listing 5.44 notiert.

Wird eine Navigation markiert, so wird jedes Eingabeelement dieser Navigation als Verursacher an die folgende Abfrage weitergereicht. Der Verursacher wird von Abfrage zu Abfrage bis zur Regelausgabe weitergereicht und schließlich in die Menge der Verursacher der Ergebnis-Verursacher-Zuordnung hinzugefügt.

In Abbildung 5.5 ist ein Beispiel gegeben, in dem eine Navigation markiert wurde, die nicht die erste in der Regel ist. In diesem Beispiel ist eine Regel und deren Ausgabe abgebildet. Innerhalb der Regel sind die Ausgabelemente a_i ($i = 1, 2, 3$) jeder Abfrage und der Regelquelle dargestellt. Die Regelquelle selektiert als einziges Element a_1. Die Navigation 1 ermittelt aus a_1 das Ausgabeelement a_2. Die Navigation 2 ermittelt aus a_2 das Modellelement a_3. Zusätzlich ist anhand eines Sterns und der grauen Hintergrundfarbe kenntlich gemacht, dass Navigation 2 durch eine Verursachungselementmarkierung markiert wurde. Das hat den Effekt, dass die Regelausgabe aus der Zuordnung von a_3 zum Verursacher a_2 besteht. Das Eingabeelement a_2 der Navigation 2 ist folglich der Verursacher.

Wenn eine Regel ohne Verursachungselementmarkierung formuliert wird, so wird standardmäßig die erste Navigation in der Regel markiert. Sobald eine beliebige Navigation markiert wird, muss die erste Navigation explizit markiert werden, wenn sie bei der Ermittlung der Verursacher berücksichtigt werden soll. Die standardmäßige Markierung der ersten Navigation gilt nur, wenn keine Verursachungselementmarkierung innerhalb der Regel verwendet wird. Ein Quelltext-Beispiel dazu ist in Listing 5.45 zu finden.

Ein Beispiel dafür, wenn entweder keine explizite Markierung verwendet wird oder die erste Navigation markiert wird, ist in Abbildung 5.6 zu finden. Dieses Beispiel ist weitgehend identisch zu dem in Abbildung 5.5. Ausschließlich der Verursacher in der Regelausgabe unterscheidet sich. Es wird das Modellelement a_1 statt wie im anderen Beispiel a_2 als Verursacher zurückgegeben. Dies liegt daran, dass Navigation 1 markiert wurde und dessen Eingabeelement a_1 ist.

Wenn zusätzlich zu Navigation 1 auch noch Navigation 2 markiert wäre, so wären a_1 und a_2 als Verursacher zum Ergebnis a_3 zugeordnet worden. Dadurch wird deutlich, dass die Menge an Verursachern mehrere Modellelemente umfassen kann.

Zu jedem Element in der temporären Ergebnismenge wird eine Zuordnung an Verursachern abgespeichert. Wenn ein Ergebnis einer Regelausgabe der temporären Ergebnismenge hinzugefügt wird, so werden ebenfalls dessen Verursacher abgespeichert. Ist ein Ergebnis bereits in der temporären Ergebnismenge vorhanden, so wird es nicht nochmals hinzugefügt. Sind jedoch Verursacher ermittelt worden, die dem Ergebnis in der temporären Ergebnismenge noch nicht zugeordnet sind, so werden diese dem Ergebnis hinzugefügt. Die endgültige Ergebnismenge enthält somit zu jedem ihrer Elemente r die Vereinigung aller Mengen an Verursachern jeder Regelausgabe, die r als Ergebnis haben.

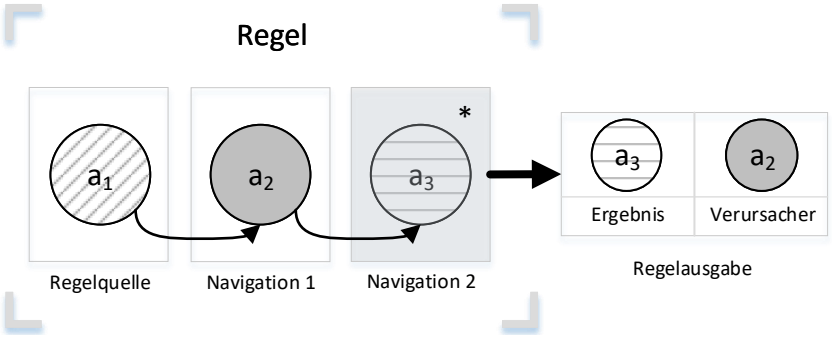

Abbildung 5.5: Markierung einer beliebigen ⟨*Navigation*⟩ einer Regel

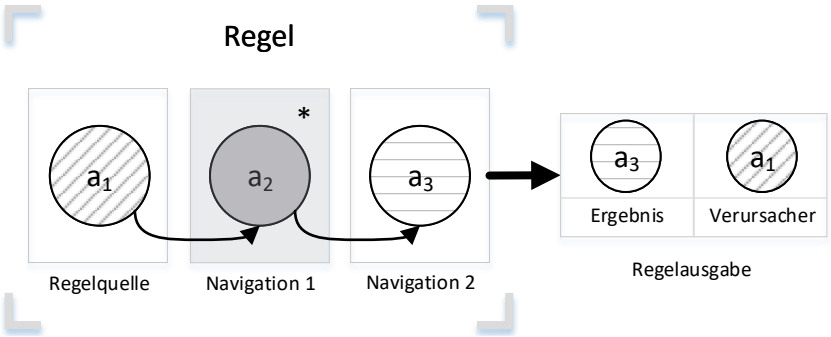

Abbildung 5.6: Standardmarkierung der ersten ⟨*Navigation*⟩ einer Regel

Ein Randfall besteht darin, dass eine Regel ohne mindestens eine Navigation keine Verursacher in der Regelausgabe enthält.

```
⟨Verursachungselementmarkierung⟩  =  '*'
```

Listing 5.44: Grammatik einer ⟨*Verursachungselementmarkierung*⟩

In Listing 5.45 ist ein Beispiel für verschiedene Kombinationen der ⟨*Verursachungselementmarkierung*⟩ gegeben.

Die Regel *A* enthält keine explizite ⟨*Verursachungselementmarkierung*⟩. Somit werden die Eingabeelemente der ersten Navigation auf Basis des Strukturmerkmals *assemblyContexts__ComposedStructure* als Verursacher ermittelt.

Die Regel *B* enthält eine ⟨*Verursachungselementmarkierung*⟩ auf der ersten Navigation. Die Regel *B* hat somit die selbe Bedeutung wie Regel *A*. Das liegt daran, dass die ⟨*Verursachungselementmarkierung*⟩ standardmäßig auf der ersten Navigation liegt, wenn sie nicht explizit angegeben wird. Die ⟨*Verursachungselementmarkierung*⟩ der Regel *B* ist quasi obsolet.

Die Regel *C* enthält keine explizite ⟨*Verursachungselementmarkierung*⟩. Somit sind standardmäßig die Eingabeelemente der ersten Navigation die Verursacher.

Die Regel *D* enthält eine explizite ⟨*Verursachungselementmarkierung*⟩ auf der zweiten Navigation. Somit sind die Eingabeelemente der zweiten Navigation die Verursacher.

Die Regel *E* enthält zwei ⟨*Verursachungselementmarkierungen*⟩. Somit sind die Eingabeelemente der beiden Navigationen dieser Regel die Verursacher.

```
import "http://palladiosimulator.org/PalladioComponentModel/5.2" as pcm

// no causing entity marker set, so first navigation is marked by default
rule A: metaclass(pcm::CompositeComponent) -> feature(
    ↪ assemblyContexts__ComposedStructure);

// the same meaning as rule A, but explicitly marking first navigation
rule B: metaclass(pcm::CompositeComponent) *-> feature(
    ↪ assemblyContexts__ComposedStructure);

// CompositeComponent as causing entity by default
rule C: metaclass(pcm::CompositeComponent) -> feature(
    ↪ assemblyContexts__ComposedStructure) -> feature(
    ↪ encapsulatedComponent__AssemblyContext);

// explicitly using AssemblyContext as causing entity instead first entity
rule D: metaclass(pcm::CompositeComponent) -> feature(
    ↪ assemblyContexts__ComposedStructure) *-> feature(
    ↪ encapsulatedComponent__AssemblyContext);

// using both AssemblyContext and CompositeComponent as causing entity
rule E: metaclass(pcm::CompositeComponent) *-> feature(
    ↪ assemblyContexts__ComposedStructure) *-> feature(
    ↪ encapsulatedComponent__AssemblyContext);
```

Listing 5.45: Beispiel von ⟨*Verursachungselementmarkierungen*⟩

6 Implementierung

Die vorherigen Kapitel erläutern die Bestandteile von CPRL auf konzeptioneller Ebene. Die konzipierte Sprache wird zudem implementiert und evaluiert. Die Struktur des Implementierungsprojekts ist in Abschnitt 6.1 beschrieben. Der darauf folgende Abschnitt 6.2 beschreibt die Integration von CPRL in KAMP. Schließlich wird in Abschnitt 6.3 beschrieben, wie ein Domänenexperte auf Java zurückgreifen kann, um eine Regel zu formulieren.

6.1 Struktur

Bei der Implementierung von CPRL handelt es sich um einen Machbarkeitsnachweis (engl. Proof of Concept, PoC). Das Ergebnis ist somit nicht eine vollständig ausgereifte Software, sondern lediglich ein Prototyp, der die Kernfunktionalität enthält und Evaluationszwecken dient.

Zur Umsetzung wird Xtext verwendet. Xtext bietet sich als Framework für die Entwicklung domänenspezifischer Sprachen an, da es einem Entwickler die Erstellung eines eigenen Parsers abnimmt. Xtext enthält dazu eine eigene Grammatiksprache, die in Kapitel 2.6 beschrieben wurde. Nachdem die Syntax von CPRL in der Xtext-Grammatiksprache definiert wurde, kann ein zugehöriger Parser für CPRL generiert werden. Xtext generiert also aus der angegebenen Grammatik den benötigten Java-Code für einen CPRL-Parser und für die Integration dieses Parsers in Eclipse. Es wird außerdem Java-Code generiert, den der Entwickler erweitern kann, um die Darstellung der Sprache in der Eclipse-Applikation anzupassen [ES18e].

Das Xtext-Projekt wird nach der Standardvorlage für kontinuierliche Integration aufgesetzt [ES18f]. Dadurch lässt sich automatisiert eine Eclipse-Update-Seite[1] für das Projekt erstellen. Mittels dieser Update-Seite lässt sich das Projekt in Eclipse als eigenständiges Feature an Endbenutzer ausliefern. Die Update-Seite wird vom Online-Dienst GitHub aus, in dessen Dienst *GitHub Pages*, betrieben.

Damit das CPRL-Projekt[2] in KAMP eingegliedert werden kann, wird das gesamte KAMP-Projekt umstrukturiert, sodass es der Tycho-Ordnerstruktur [Fau15] entspricht. Anschließend wird das CPRL-Projekt KAMP als eigenständiges Feature hinzugefügt. Die CI wird für das gesamte KAMP-Projekt eingerichtet. Ein Ergebnis dieser Arbeit ist somit auch die Umstellung von KAMP auf die Tycho-Ordnerstruktur, sowie die Konfiguration dessen CI.

[1] https://martinloeper.github.io/updatesite/nightly/

[2] Es existiert eine erste Version in der Repository KAMP-DSL und eine Folgeversion mit kontinuierlicher Integration in der Repository KAMP-2.0. Beide sind unter dem Benutzeraccount MartinLoeper in Github zu finden unter: https://github.com/MartinLoeper/

Der Quelltext des CPRL-Projekts ist unter dem folgenden Pfad in der KAMP-Repository[3] zu finden: *bundles/extensions/KAMP-DSL/*. Teile der CPRL-Implementierung, die von den einzelnen KAMP-Modulen verwendet werden, sind in den Kern[4] von KAMP ausgelagert. Innerhalb des Kerns ist die CPRL-Integration unter folgendem Pfad zu erreichen: */src/edu/kit/ipd/sdq/kamp/ruledsl/support/*.

6.2 Einbindung in KAMP

Möchten Domänenexperten CPRL-Regeln formulieren, dann müssen sie zuerst in Eclipse die Plugins aller KAMP-Features von der Update-Seite installieren und aktivieren. Anschließend können sie in ihrem Workspace ein Plug-in Projekt erstellen, welches im Folgenden als **CPRL-Projekt** bezeichnet wird. Innerhalb dieses Projekt können die Domänenexperten eine **Regeldatei** mit der Dateiendung "*.karl*" erstellen. Sobald sie diese erstellen möchten, wird eine Eclipse-Meldung angezeigt, welche sie dazu auffordert, das Projekt in ein Xtext-Projekt umzuwandeln. Diese Meldung muss in jedem Fall bestätigt werden. Anschließend können CPRL-Regeln in die Regeldatei geschrieben werden.

Sobald Regeln in die Regeldatei geschrieben und abgespeichert werden, wird eine Codegenerierung in Gang gesetzt. Es wird zuerst, falls noch nicht vorhanden, ein neues Plugin-in Projekt im aktuellen Workspace erstellt. Dieses neue Projekt wird im Folgenden als **generiertes Projekt** bezeichnet. Es trägt den gleichen Namen wie das CPRL-Projekt und zusätzlich den Suffix "*-rules*".

Jede Regel in der Regeldatei wird in eine separate Java-Klasse im Paket *gen.rules* des generierten Projekts umgewandelt. Die Umwandlung wird durch den *JvmModelInferrer* von Xtext durchgeführt. Dieser wird in der Implementierung von CPRL erweitert. Wie die Überführung von CPRL in Java im Detail funktioniert, kann in der folgenden Klasse des KAMP-Projekts auf GitHub nachgelesen werden: *KampRuleLanguageJvmModelInferrer.xtend* [KAM18]. Ein Vergleich zwischen den CPRL-Konstrukten und den zugehörigen, generierten Konstrukten für die Java-ähnliche Sprache Xtend, ist in Abschnitt 7.4.4 der Evaluation gegeben. Zusammengefasst, werden die Regeln aus der Regeldatei des CPRL-Projekts in einzelne Java-Klassen des generierten Projekts umgewandelt.

Das generierte Projekt ist ein Plug-in Projekt und stellt ein OSGi-Bundle dar. Das OSGi-Bundle wird bei jeder Änderung der generierten Klassen neu kompiliert und in der Eclipse-Umgebung aktiviert. Die generierten Klassen ändern sich, sobald die Regeldatei geändert und gespeichert wird. Das Abschließen einer Regelformulierung in CPRL führt somit immer zu einem Neuladen des zugehörigen OSGi-Bundles. Das Wechseln von Modulen im laufenden Betrieb des Systems wird als *Hot Swapping* bezeichnet.

Das Hot Swapping ermöglicht es den Domänenexperten, die Regeln in einer modifizierten Regeldatei sofort mittels KAMP auszuführen, ohne zunächst die Eclipse-Umgebung neu starten zu müssen. Das führt zu einer erheblichen Zeitersparnis und verbirgt die Überführung der CPRL-Regeln in Java-Code vor den Domänenexperten nahezu vollständig.

[3] https://github.com/KAMP-Research/KAMP

[4] Der Kern befindet sich unter folgendem Pfad in der KAMP-Repository: *bundles/framework/edu.kit.ipd.sdq.kamp*

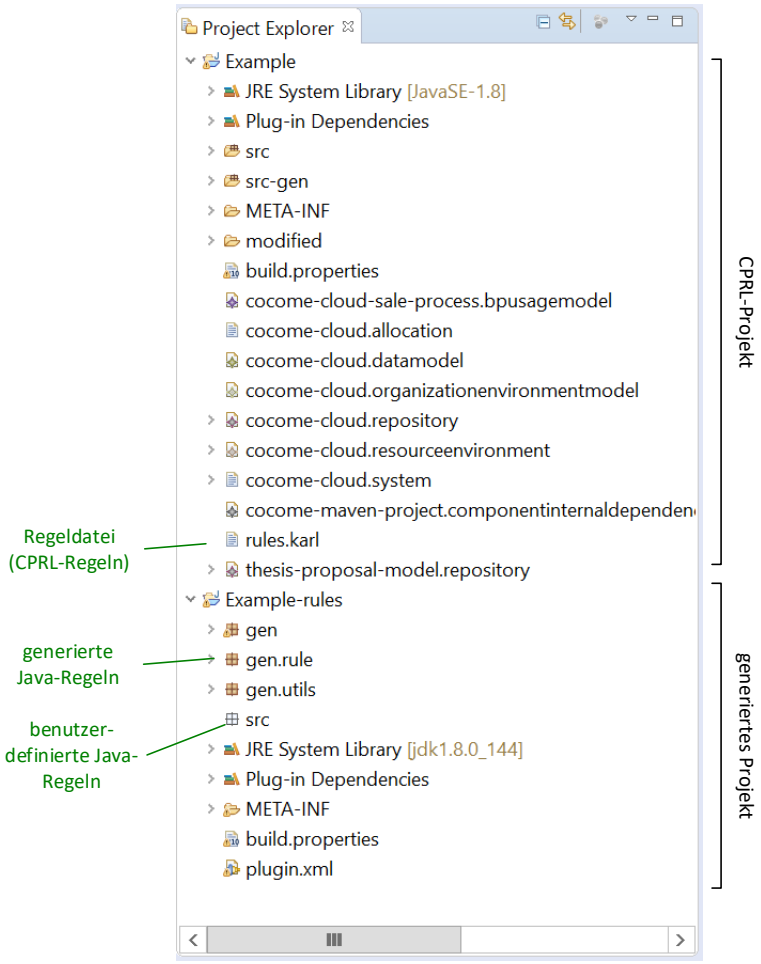

Abbildung 6.1: Projekt-Explorer eines CPRL-Beispielprojekts mit der Bezeichnung Example

Das generierte Projekt registriert bei jeder Aktivierung einen OSGi-Dienst, welcher Zugriff auf die im Projekt enthaltenen Regeln und ihre Ausführungsreihenfolge bietet. Die einzelnen KAMP-Module greifen innerhalb ihrer Änderungsausbreitungsanalyse auf diesen Dienst zu. Sie ermitteln die Regeln und führen diese aus. Für ein Verständnis von OSGi-Diensten in Eclipse, ist der Artikel [Vog18] von Vogel nützlich.

In Abbildung 6.1 ist der Projekt-Explorer eines Beispielprojekts abgebildet. Sie enthält Beschriftungen, welche auf die, in diesem Abschnitt beschriebenen, Komponenten verweisen.

6.3 Komplexitätsübergang

Domänenexperten können in KAMP, anhand eines CPRL-Projekts, Regeln in CPRL formulieren. Es gibt allerdings auch Gründe, weshalb sie lieber Java verwenden möchten. Zum

einen kann es sein, dass eine bestimmte Regel sich einfacher in Java darstellen lässt oder gar nicht in CPRL darstellbar ist. Meist ist das der Fall, wenn die Regel einen bestimmten Grad an Komplexität überschreitet.

Die Integration von CPRL in KAMP bietet den Domänenexperten die Möglichkeit, sich, je nach Komplexität der zu beschreibenden Regel, entweder für die Formulierung in CPRL oder Java zu entscheiden. Dieses Gestaltungsprinzip eines Softwaresystems wird als *Weicher Komplexitätsübergang* bezeichnet [LDPW17; MCLM90].

Um es den Domänenexperten zu ermöglichen, Java-Code zu schreiben, werden Techniken der DI und Java-Annotationen verwendet. Sie können entweder eine neue Java-Regel erstellen oder eine bestehende CPRL-Regel mittels Java-Code erweitern. Im Folgenden wird erläutert, wie dies erreicht werden kann.

Zuerst muss eine neue Java-Klasse im *src* Paket des generierten Projekts erstellt werden. Die Datei stellt eine neue Regel für die Änderungsausbreitung dar. Damit die Regel wohlgeformt ist, muss die neu erstellte Java-Klasse das Interface *IRule* implementieren. In den Methodenrümpfen des IRule-Interfaces können Domänenexperten ihre eigene Änderungsausbreitungslogik in Java-Code formulieren. Damit das Framework die Regel ausfindig machen kann, muss der neu erstellten Klasse außerdem die Annotation *@KampRule* hinzugefügt werden. Mit der Annotation kann dem Framework mitgeteilt werden, ob eine bestehende Regel erweitert oder eine neue Regel erstellt werden soll. Dazu wird die Parameterliste der Annotation verwendet. Der Parameter *enabled* gibt an, ob die benutzerdefinierte Regel vom Framework berücksichtigt werden soll. Der Parameter *parent* gibt an, welche Regel erweitert werden soll. Lassen Benutzer diesen Parameter weg, erstellen sie eine neue Regel, statt eine bestehende zu erweitern. Es kann in einer benutzerdefinierten Regel maximal eine Regel erweitert werden. Die Regel mit dem parent-Verweis wird **Kindregel**, die erweiterte Regel **Elternregel**, genannt. Entscheiden sich Benutzer, eine Regel zu erweitern, so können sie mit dem Parameter *disableAncestors* definieren, ob die Elternregeln aktiviert bleiben soll. Wird der Parameter auf *true* gesetzt, so wird die Elternregeln und rekursiv all deren Elternregeln deaktiviert.

Das Framework verwaltet die Eltern-Kind-Beziehung zwischen benutzerdefinierten Regeln und generierten Regeln in einem gerichteten, azyklischen Graphen. Wird eine Änderungsausbreitungsanalyse ausgelöst, so nutzt KAMP den erstellten Graphen, um zu ermitteln welche Regeln ausgeführt werden sollen. In einem vorangehenden Schritt werden dazu die Regeln topologisch sortiert und instanziiert. Eine Kindregel bekommt stets ihre Elternregel über Konstruktor-Injektion übergeben. Auf diese Weise können Benutzer auf die generierte Regel zugreifen und diese erweitern. Sie müssen dabei darauf achten, dass sie dem Konstruktor der Kindregel ein Parameter mit dem Typ *IRule* hinzufügen. Andernfalls schlägt die Injektion fehl und verursacht eine Ausnahme. Für eine benutzerdefinierte Regel ohne Elternregel muss der Standardkonstruktor implementiert werden.

Ein Beispiel einer benutzerdefinierten Regel, welche die Regel *ARule* erweitert und deaktiviert, ist in Listing 6.2 zu finden. Damit Benutzer sich die Eigenheiten der Regelerstellung und Regelerweiterung nicht merken müssen, wird ein Wizard angeboten. Nach Abschluss des Wizards wird eine Java-Datei mit einer Regelvorlage erstellt. Der Benutzer braucht nur noch die Methodenrümpfe der Regelvorlage auszufüllen. In Abbildung 6.2 ist ein Screenshot des Wizards dargestellt.

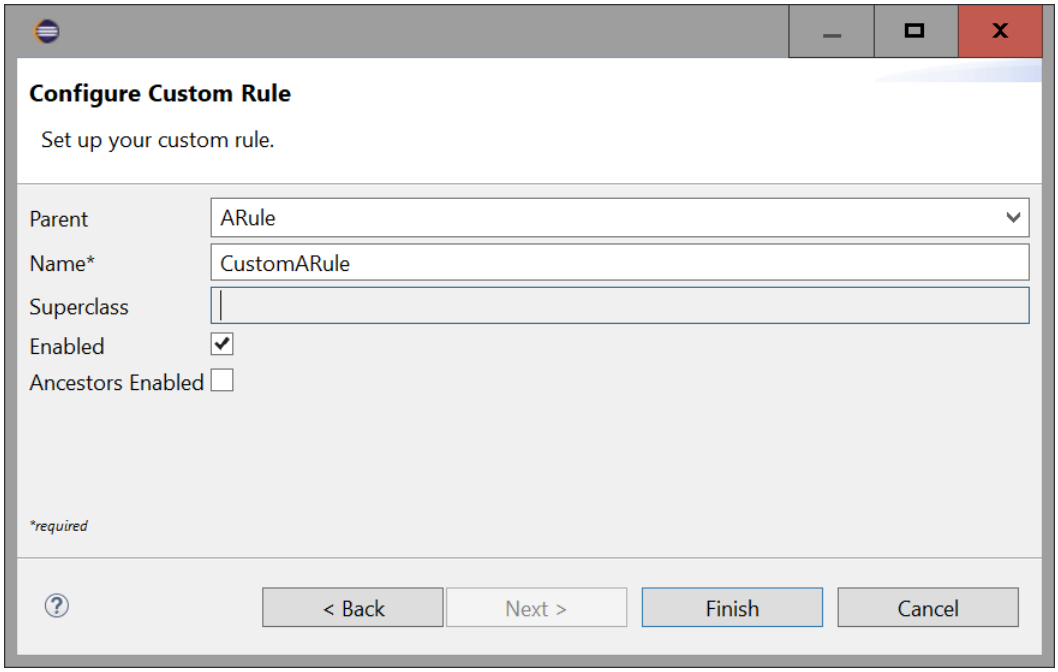

Abbildung 6.2: Wizard für die Erstellung einer Regelvorlage für eine benutzerdefinierte Regel

Ein weiteres Hilfswerkzeug für Benutzer, stellt die Anzeige des Abhängigkeitsgraphen der Regeln dar. Der Graph kann benutzt werden, um Probleme bei der Ausführung von Regeln zu untersuchen. Um Zugriff auf den Graphen zu erlangen, kann einer beliebigen Methode die Annotation *@KampGraph* hinzugefügt werden. Diese Annotation veranlasst das Framework dazu, den Graphen als Parameter zu injizieren. Ein Beispiel für eine Methode, die den injizierten Graph visualisiert, ist in Listing 6.1 gegeben. Ein Beispiel für die Darstellung des Graphen ist in Abbildung 6.3 zu finden. Jede Regel wird in einem Rechteck dargestellt und nach ihrem Klassennamen benannt. Die benutzerdefinierten Regeln werden unterstrichen dargestellt. Aktive Regeln werden grün umrandet. Die Pfeilspitzen zeigen von der Kindregel zur Elternregel.

Schließlich besteht noch eine weiteres Hilfswerkzeug, das besonders für Domänenexperten beim Debuggen ihrer Regelausführungen, nützlich sein kann. Es handelt sich um die *CPRL Query Result*-Ansicht in Eclipse - auch **Änderungsausbreitungsansicht** genannt. Jede Änderungsausbreitungsanalyse in KAMP wird in diese Ansicht eingefügt und erlaubt einen detaillierten Einblick in einzelne Regelausführungen. Es wird für jede Änderungsausbreitungsanalyse eine Auflistung der ausgeführten Regeln und deren Reihenfolge dargestellt. Innerhalb jeder Regelausführung können die Eingabe- und Ausgabeelemente betrachtet werden. Es ist sogar möglich, in die einzelnen Regeln hineinzusehen und die Ergebnisse deren Abfragen nachzuvollziehen. Ein Screenshot der Änderungsausbreitungsansicht ist in Abbildung 6.4 zu finden. Sie zeigt das Ergebnis einer einzigen Änderungsausbreitungsanalyse.

```
@KampGraph
public void showGraph(KampRuleGraph graph) throws IOException,
    ↪ InterruptedException {
    graph.show("<path to a Graphviz dot executable>");
}
```

Listing 6.1: Beispiel einer Methode, die den Abhängigkeitsgraph der Regeln anzeigen lässt

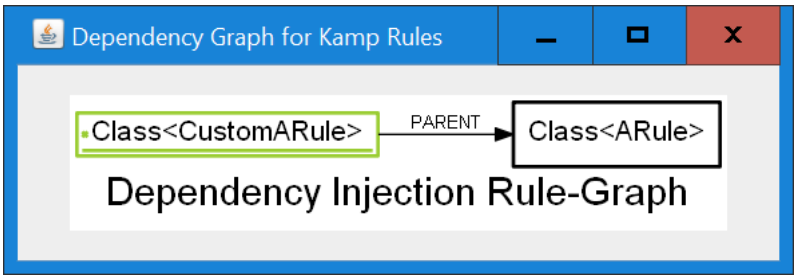

Abbildung 6.3: Abhängigkeitsgraph für die Dependency Injection der Regeln

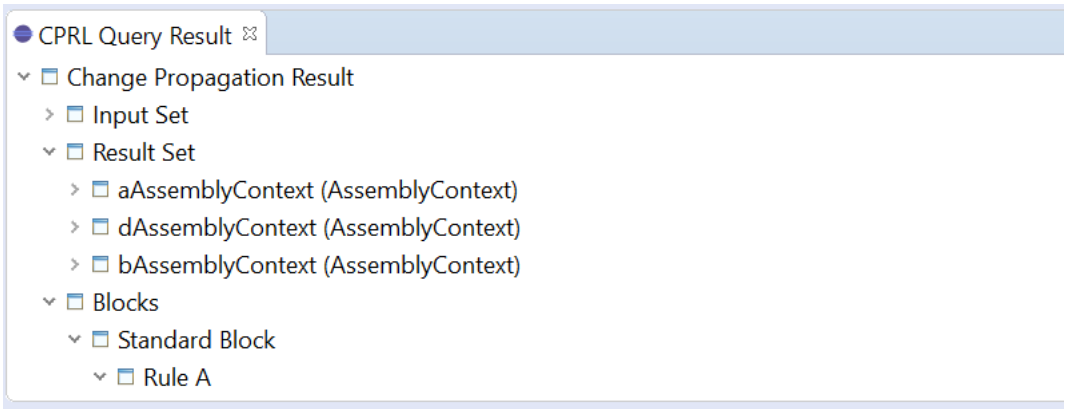

Abbildung 6.4: Änderungsausbreitungsansicht

```
package src;

import [...]

@KampRule(enabled = true, disableAncestors = true, parent = ARule.class)
public class CustomARule implements IRule<CompositeComponent, AssemblyContext,
    ↪ AbstractArchitectureVersion<AbstractModificationRepository<?, ?>>,
    ↪ AbstractModificationRepository<?, ?>> {

    private final IRule<CompositeComponent, AssemblyContext,
        ↪ AbstractArchitectureVersion<AbstractModificationRepository<?, ?>>,
        ↪ AbstractModificationRepository<?, ?>> parentRule;

    private static final String CUSTOM_RULE_NAME = "MyCustomRuleA";

    // do not forget the ARule parameter for dependency injection!
    public CustomARule(ARule parentRule) {
        super();
        this.parentRule = parentRule;
    }

    @Override
    public RuleResult<CompositeComponent, AssemblyContext> lookup(final Result<?,
        ↪  CompositeComponent> input) {
        @SuppressWarnings("unchecked")
        AbstractLookup<EObject, EObject>[] lookups = [...] // custom lookups
        RuleResult<CompositeComponent, AssemblyContext> result = LookupUtil.
            ↪ runLookups(input, lookups, CUSTOM_RULE_NAME);

        return result;
    }

    @Override
    public void setArchitectureVersion(AbstractArchitectureVersion<
        ↪ AbstractModificationRepository<?, ?>> architectureVersion) {
        this.parentRule.setArchitectureVersion(architectureVersion);
    }

    [...] // some other getter and setter
}
```

Listing 6.2: Beispiel einer benutzerdefinierten Java-Regel

6.4 Installation

Im Rahmen dieser Arbeit wurde ein Installationsprogramm entwickelt, das Eclipse auf dem Rechner installiert und KAMP darin einrichtet. Es wird im Folgenden **KAMP-Installer** genannt. Bei dem KAMP-Installer handelt es sich um eine angepasste Version des Oomph Eclipse-Installationsprogramms. Für die gängigen Betriebssysteme und Architekturen liegt jeweils eine angepasste Binärdatei zum Herunterladen auf GitHub[5] bereit. Der KAMP-Installer unterscheidet sich von der offiziellen Oomph-Version darin, dass er einen vordefinierten Index mitbringt. Ein Index beinhaltet einen Produktkatalog mit Software, die installiert werden kann. Der Index des KAMP-Installers liegt auf GitHub[6] und wird mittels GitHub Pages[7] ausgeliefert.

Eine vordefinierte Menge an Software, die zusammen in eine Eclipse-Umgebung installiert wird, trägt die Bezeichnung Produkt. In diese vordefinierte Menge fallen sowohl binäre Softwareartefakte, wie beispielsweise Plugins, aber auch Quelltexte in Form von Eclipse-Workspace-Projekten. Der KAMP-Installer bietet zwei Produkte zur Installation an. Das erste Produkt ist KAMP für Endbenutzer. Es besteht aus einer standardmäßigen Eclipse-Installation, vorinstallierten KAMP Features und deren zugehörigen Abhängigkeiten. Das zweite Produkt ist KAMP für Entwickler. Es besteht ebenfalls aus einer standardmäßigen Eclipse-Installation und zusätzlich allen Plugins, die für eine Weiterentwicklung von KAMP benötigt werden (beispielsweise die Eclipse Modeling Tools). Außerdem wird KAMP nicht in Binärform, sondern in Quelltextform heruntergeladen. Das geschieht, indem die einzelnen GitHub- und SVN-Repositorys in den Workspace heruntergeladen werden.

Der Vorteil des KAMP-Installers im Vergleich zu einer manuellen Installation liegt darin, dass Eclipse schneller eingerichtet wird. Zudem wird Eclipse stets korrekt aufgesetzt. Damit werden fachfremden Endbenutzern die Hürden genommen, sich mit Eclipse und dessen Plugin-Management auseinandersetzen zu müssen. Für Entwickler besteht der Vorteil, dass nicht alle Quelltextablagen manuell zusammengesucht und importiert werden müssen. Außerdem besteht ein Vorteil bei der Speicherplatzverwaltung, wenn ein Entwickler mehrere Eclipse-Installationen parallel verwaltet. Durch den Einsatz von p2-Bundle-Pools werden Downloadzeiten reduziert und es wird Speicherplatz gespart, da Plugins zwischen Installationen geteilt werden.

In Abbildung 6.5 ist die vereinfachte Startansicht des KAMP-Installers abgebildet. Das obere Produkt liefert KAMP für den Endbenutzer aus. Das untere Produkt, mit der Bezeichnung *Eclipse Modeling Tools (KAMP Development)*, ist für Entwickler gedacht.

In Abbildung 6.6 ist eine erweiterte Startansicht des KAMP-Installers abgebildet. Es handelt sich dabei gewissermaßen um einen Expertenmodus. Dieser wird aktiviert, indem in der vereinfachten Ansicht auf das Hamburger-Symbol in der rechten obere Ecke geklickt und der Menüpunkt *Advanced Mode* ausgewählt wird. Der Expertenmodus erlaubt es dem Benutzer, die zu installierenden Artefakte im Detail auszuwählen, sowie den Speicherort und weitere Parameter der Eclipse-Installation anzupassen.

[5] https://github.com/MartinLoeper/KAMP-Windows-Installer
[6] https://github.com/MartinLoeper/KAMP-Installer-Index
[7] https://martinloeper.github.io/KAMP-Installer-Index/setups/org.eclipse.setup

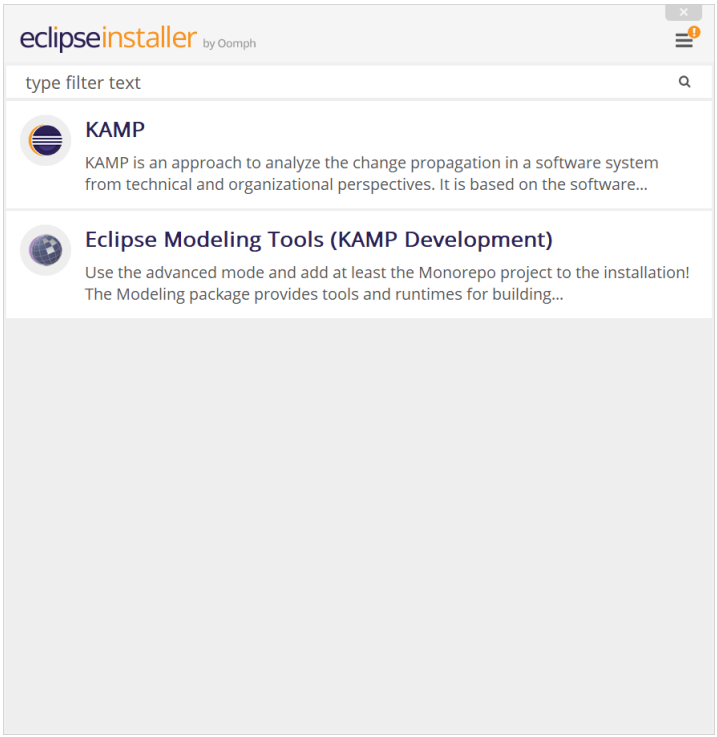

Abbildung 6.5: Startansicht des KAMP-Installers

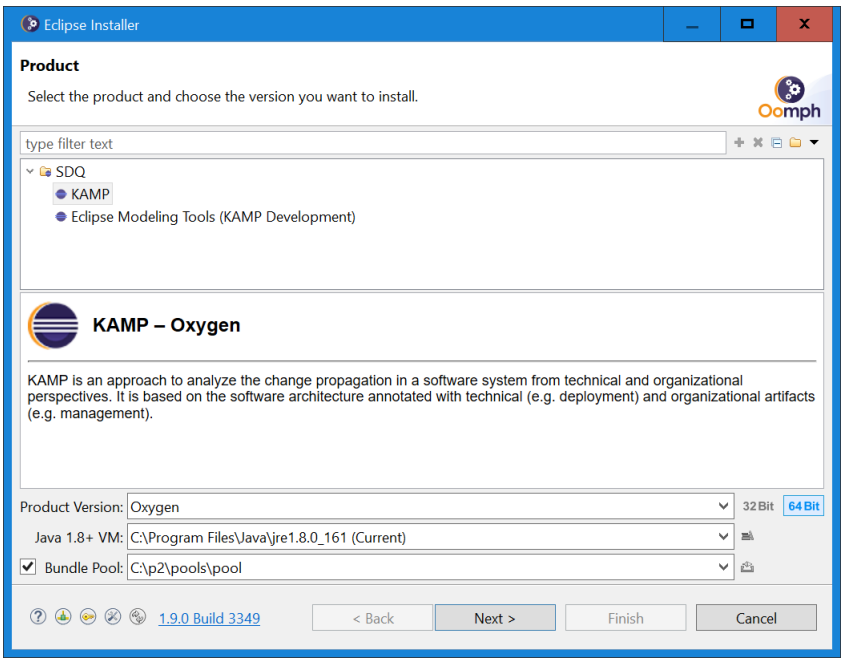

Abbildung 6.6: Startansicht des KAMP-Installers im Expertenmodus

7 Evaluation

Um eine neu erstellte DSL zu evaluieren, schlägt Kramer in seiner Dissertation [Kra17] vor, die folgenden Eigenschaften zu untersuchen. Diese werden jeweils in einem eigenen Abschnitt thematisiert.

Vollständigkeit Unterstützt die Sprache alle Anwendungsfälle die denkbar sind?

Korrektheit Produziert die Sprache für alle Anwendungsfälle korrekte Ergebnisse?

Anwendbarkeit Kann die Sprache tatsächliche in der Praxis eingesetzt werden?

Vorteil Besitzt die Sprache Vorteile gegenüber anderen Sprachen?

7.1 Vollständigkeit

CPRL ist eine DSL, die den Algorithmus zur Änderungsausbreitungsanalyse in KAMP konfiguriert. Aus der Sprache heraus werden Java-Quelltextfragmente generiert, die an bestimmten Stellen in KAMP eingebunden werden. Die bestehenden KAMP-Module sind vollständig in Java geschrieben. Da Java eine universelle Programmiersprache ist, können damit beliebige Regeln zur Änderungsausbreitung definiert werden. Dies liegt vor allem daran, dass eine Vielzahl an Kombinationen aus Listen, Schleifen und Bedingungen möglich ist. Das Ziel der DSL ist es nicht, alle möglichen Arten an Änderungsausbreitungen zu unterstützen, sondern sich auf jene zu beschränken, die am häufigsten verwendet werden und den größten Anteil in den vorhandenen KAMP-Modulen stellen. Wie in Kapitel 6 beschrieben, können komplexe Regeln, die nicht durch CPRL darstellbar sind, weiterhin mithilfe von Java implementiert werden. Dazu bietet CPRL eine Technik, die vom KAMP-Framework abstrahiert. Die Java-Regeln können mit einer Annotation versehen werden, welche KAMP dazu veranlasst, diese anhand von Dependency Injection zu laden. Dadurch muss der Domänenexperte keine Kenntnis über die interne Implementierung von KAMP haben, um Java-Regel zu schreiben. In der aktuellen Version von CPRL werden nur Änderungsausbreitungen unterstützt, welche den folgenden Kriterien entsprechen:

mengenbasiert Die Regeln und Abfragen in CPRL basieren auf Mengen an Modellelementen. Alle Prädikate und sonstige Sprachelemente operieren implizit auf einzelnen Elementen aus diesen Mengen. Diese Eigenschaft ist aus der Erkenntnis gewachsen, dass imperative Formulierungen in der Änderungsausbreitungsanalyse aus einer Vielzahl an Schleifen bestehen. Die Schleifen werden durch die DSL verborgen, kommen jedoch im generierten Quelltext immer noch zum Einsatz.

lokal Die Ermittlung neuer Mengen durch Navigation operiert auf nur einem Eingabeelement. Die Prädikate haben keinen Zugriff auf globale Mengen, wie beispielsweise die temporäre Ergebnismenge oder die Menge an Anfangsmarkierungen. Es ist nicht einmal möglich, auf die Modellelemente in der aktuell betrachteten Abfrage oder in vorangehenden Abfragen zuzugreifen. Die Sprache ist in dem Sinne lokal, als dass sie nur Zugriff auf das Element in der innersten Schleife bietet. Dies stellt eine große Einschränkung für komplexe Regeln dar. Die Navigationsbedingung für eine konkrete Instanz kann somit nur von Attributen ebendieser Instanz abhängen. Sie kann niemals von Modellelementen aus vorangehenden Abfragen oder von Referenzen durch beliebige Modellelemente im Architekturmodell abhängen. Der Abfragepfad, auf den man in imperativen Programmiersprachen normalerweise über die Elemente der äußeren Schleifen Zugriff hat, geht verloren. Ein Beispiel ist in Listing 7.1 gegeben. Eine einfache Regel mit zwei aufeinander folgenden Abfragen, stellt im generierten Quelltext zwei Schleifen dar. Die letzte der beiden Abfragen hat lediglich Zugriff auf die grün eingefärbte Variable *innerElement*. Es fehlen Sprachmittel, um die anderen beiden Variablen *outerElements* und *outerElement* zu referenzieren.

```
for (EObject outerElement : outerElements) {
    for (EObject innerElement : outerElement.getInnerElements()) {
        // do something with innerElement
    }
}
```

Listing 7.1: Beispiel für die Einschränkung in Abfragen

rekursiv markierend Die Rekursion in CPRL basiert auf Elementen, die markiert werden. Es ist nicht möglich, eigene Mengen zu definieren und diese als Entscheidungsgrundlage für die Rekursion zu verwenden. Ein rekursiver Block wird so lange ausgeführt, bis sich die Anzahl der markierten Elemente durch dessen Ausführung nicht mehr verändert. Ein Beispiel für eine Abfrage, in der diese Einschränkung deutlich wird, ist die Methode *lookUpUserActionsAfterMarkedActorSteps (BP1)* in Abschnitt 7.3.2.

ohne benutzerdefinierte Mengen Es können keine benutzerdefinierten Mengen angegeben werden. Dies ist eine erhebliche Einschränkung für die Rückwärtsnavigation. Bestimmte Referenzen können dadurch nicht abgebildet werden. Ein Beispiel ist die Methode *lookUpUserActionsAfterMarkedActorSteps (BP1)* in Abschnitt 7.3.2.

ohne benutzerdefinierte Eigenschaften Es können keine benutzerdefinierten Eigenschaften an Modellelemente angehängt werden. Es besteht außerdem keine Möglichkeit, Zuordnungen zwischen Modellelementen und bestimmten Werten (z.B. mittels einer HashMap) zu definieren. Ein Beispiel für eine Abfrage, in der diese Einschränkung deutlich wird, ist die Methode *lookUpUserActionsUpToReleaseDeviceResource (BP4)* in Abschnitt 7.3.2.

Dass die oben genannten Einschränkungen dazu führen, dass nicht alle vorhandenen und denkbaren Änderungsausbreitungsregeln in CPRL dargestellt werden können, wird im Abschnitt 7.3 im Detail thematisiert.

7.2 Korrektheit

Zur Prüfung der Korrektheit sind diversifizierende Testmethoden denkbar. Zum Beispiel könnte ein Regressionstest mit dem Common Component Modelling Example (CoCoME) durchgeführt werden. Dazu werden verschiedene Änderungsanfragen in KAMP erstellt. Anschließend wird zu jeder bestehenden Java-Regel für die Änderungsausbreitung eine CPRL-Regel formuliert. Es wird jeweils ein Durchlauf mit CPRL-Regeln und einer mit Java-Regeln durchgeführt, um die betroffenen Elemente zu ermitteln. Die Differenz der betroffenen Elemente für beide Regelformulierungen sollte dabei null ergeben. Die Durchführung solcher A/B-Tests beweist zwar niemals die Abwesenheit von Fehlern, gewährleistet allerdings ein gewisses Maß an Korrektheit, das dem Umfang der Tests entspricht. Im Zuge dieser Arbeit werden Unit-Tests und exemplarisch auch A/B-Tests, auf Grundlage eines eigenen Beispielmodells, geschrieben.

Es ist zu bestimmen, ob es grundsätzlich möglich ist, eine formale Verifikation durchzuführen, die beweist, dass alle denkbaren Java-Regeln auch durch CPRL darstellbar sind. Die Syntax der CPRL ist, mit Ausnahme der imperativen Prädikate, regulär (siehe dazu Abschnitt 9.4 im Anhang). Betrachtet man allerdings die statische Semantik, so kommen Einschränkungen hinzu, die nicht mehr durch einen regulären Ausdruck beschrieben werden können. Für die Darstellung der CPRL wird also eine mindestens kontextfreie Grammatik benötigt, die nicht regulär ist. Java besteht aus einer Grammatik, die mindestens kontextfrei ist [Jim12]. Es ist nicht möglich zu entscheiden, ob zwei kontextfreie Grammatiken dieselbe Sprache darstellen [Sip13].

7.3 Anwendbarkeit

Die Anwendbarkeit kann dadurch evaluiert werden, dass alle bestehenden Java-Regeln aus KAMP in CPRL überführt werden. Mit dieser Aufgabe hat sich eine andere Arbeit beschäftigt, deren Ergebnis im folgenden Abschnitt thematisiert wird. Der darauf folgende Abschnitt 7.3.2 geht auf die derzeit nicht unterstützten Regeln aus KAMP ein.

7.3.1 Abdeckung von KAMP

Die Masterarbeit [Bel18] von Inna Belyantseva hat die Anwendbarkeit von CPRL untersucht, bevor diese Arbeit fertig gestellt wurde. Sie ist zu dem Schluss gekommen, dass mit zwölf Ausnahmen, alle Abfrage-Methoden in den vorhandenen KAMP-Modulen in CPRL überführbar sind. Das entspricht in etwa einer Abdeckung von 76%.

7.3.2 Fehlende Funktionalität

Die folgenden zwölf Methoden beinhalten Abfragen, die von Belyantseva in ihrer Masterarbeit [Bel18] als nicht in CPRL umsetzbar identifiziert wurden. Alle Methoden mit dem Präfix *IS* sind aus dem Modul KAMP4IS, jene mit dem Präfix *REQ* aus KAMP4REQ und jene mit dem Präfix *BP* aus KAMP4BP. Zu jeder Methode wird geschrieben, ob sie mit den aktuellen Sprachkonstrukten abbildbar ist und falls nicht, wieso dies der Fall ist.

BP1 lookUpUserActionsAfterMarkedActorSteps
Status: nicht unterstützt
Grund: Es wird eine Rekursion über Modellelemente durchgeführt, die nicht markiert werden. Dies ist momentan nicht möglich. Aktuell kann eine Rekursion nur über die Ausgabe einer Regel erfolgen. Elemente der Regelausgabe werden automatisch alle markiert.
Lösung: Es könnte ein Konstrukt *K* in CPRL hinzugefügt werden, welches es erlaubt, Mengen zu definieren. *K* hätte eine ähnliche Semantik wie eine Regel. Allerdings wären die zwei semantischen Unterschiede: *K* würde die Ausgabe in eine eigene Menge emittieren, statt in die temporäre Ergebnismenge. Außerdem würde *K* als Quelle entweder die Anfangsmarkierungen oder die temporäre Ergebnismenge oder die Menge aller Elemente im Architekturmodell zur Verfügung stellen. Anschließend könnte man die Rekursion so abändern, dass sie entweder über die temporäre Ergebnismenge oder über die eigens durch *K* definierte Menge durchgeführt wird.

BP2 lookUpCompositeAndCollectionDataObjectsWithDataObjects
Status: nicht unterstützt
Grund: siehe BP1
Lösung: siehe BP1

BP3 lookUpCompositeAndCollectionDataTypesOfDataTypes
Status: nicht unterstützt
Grund: siehe BP1
Lösung: siehe BP1

BP4 lookUpUserActionsUpToReleaseDeviceResource
Status: nicht unterstützt
Grund: Es ist nicht möglich innerhalb von Abfragen eigene Attribute zu definieren oder den Modellelementen Attribute hinzuzufügen.
Lösung: Es könnte ein Konstrukt *A* in CPRL hinzugefügt werden, welches es erlaubt, Metaklassen ein neues Attribut hinzuzufügen. Dieses neue Attribut müsste anschließend im Sichtbarkeitsbereich der Prädikate vorhanden gemacht werden, damit der Benutzer es entweder modifizieren oder in Bedingungen verwenden kann.

IS1 lookUpAssemblyConnectorsAttachedToProvidedRole
Status: nicht unterstützt
Grund: Es ist nicht möglich, die Ausführungsreihenfolge von Abfragen zu verändern. Somit ist es auch nicht möglich, eine Rückwärtsnavigation über eine Kette mehrere Strukturmerkmale durchzuführen.

Lösung: Es könnte eine spezielle Art von Regel *R* eingeführt werden, welche in der Kopfzeile der Regeldatei deklariert werden kann und nicht automatisch ausgeführt wird. Eine Regel des Typs R kann nur über Regelreferenzen ausgeführt werden und emittiert seine Ausgabe somit immer an die aufrufende Abfrage. Zusätzlich müsste der Rückwärtsnavigation ein Regelaufruf als Quelle hinzugefügt werden. Dadurch ließe sich die Ausführungsreihenfolge von Abfragen verändern. Alternativ ließe sich die Methode auch mit der Lösung aus BP1 umsetzen. Dazu müsste eine benutzerdefinierte Menge innerhalb einer Rückwärtsnavigation als Quelle referenzierbar sein.

IS2 lookUpComponentsAndRolesWithInterfaces
Status: unterstützt
Lösung: Die Methode muss in zwei einzelnen Regeln formuliert werden. Eine davon untersucht Referenzen auf *RequiredRole* und die andere auf *ProvidedRole*.

REQ1 lookUpInterfacesReferencedByDecisions
Status: unterstützt
Lösung: Die Methode muss durch viele einzelne Regeln formuliert werden. Für jede Optionsart muss eine eigene Regel erstellt werden.

REQ2 lookUpComponentsReferencedByDecisions
Status: unterstützt
Lösung: siehe REQ1

REQ3 lookUpComponentsReferencedByOptions
Status: unterstützt
Lösung: siehe REQ1

REQ4 lookUpObjectsDependOnObjects
Status: unterstützt
Lösung: Dies ist ein Standardfall einer Rekursion in CPRL.

REQ5 lookUpObjectsAnotherObjectDependsOn
Status: unterstützt
Lösung: siehe REQ4

REQ6 lookUpEntitiesReferencedByDecisions
Status: unterstützt
Lösung: Die Methode muss durch drei Regeln formuliert werden, die eine Vorwärtsreferenz auf die jeweilige Metaklasse beinhalten.

Die Methoden sind in den folgenden Klassen von KAMP im GitHub-Repository [KAM18] zu finden:

KAMP4BP edu.kit.ipd.sdq.kamp4bp.core.BPArchitectureModelLookup

KAMP4IS edu.kit.ipd.sdq.kamp4is.core.ISArchitectureModelLookup

KAMP4REQ edu.kit.ipd.sdq.kamp4req.core.ReqArchitectureModelLookup

CPRL unterstützt momentan fünf Methoden aus den oben genannten Modulen nicht. Insgesamt ist mit 46 von 51 Methoden eine Abdeckung von etwa 90% durch CPRL gegeben. Die Anwendbarkeit ist weitestgehend erfüllt. Sie ist vollständig erfüllt, wenn die Lösungsvorschläge aus diesem Abschnitt implementiert werden.

7.4 Vorteil

Der Vorteil kann evaluiert werden, indem ein Vergleich von Regeln in CPRL mit Regeln in anderen Sprachen durchgeführt wird. Abschnitt 7.4.1 definiert eine Metrik, die für konkurrierende Sprachen quantifiziert wird. Dazu wird in Abschnitt 7.4.2 zunächst eine Präzisierung der verwendeten Metrik vorgenommen, indem für einzelne Sprachen beschrieben wird, wie die Metrik anzuwenden ist. Abschnitt 7.4.4 stellt Regelbestandteile aus CPRL denen in OCL, Xtend und VIATRA gegenüber. Dazu wird ein Beispielmodell verwendet, das in Abschnitt 7.4.3 beschrieben ist. Für das Beispielmodell werden bestimmte Arten von Änderungsausbreitungen formuliert, die typischerweise von einem Domänenexperten verwendet werden.

7.4.1 Goal Question Metric (GQM)

Laut Böhme und Reussner muss ein Ansatz seine Überlegenheit gegenüber konkurrierenden Ansätzen demonstrieren [BR08, S.15]. Den Ansatz zur Änderungsausbreitungsanalyse in KAMP mit anderen Ansätzen zu vergleichen, wäre sehr aufwändig und nur schwer realisierbar, da einige Ansätze (laut Lehnert etwa 25%) rein konzeptioneller Natur sind. Es wird sich daher in dieser Evaluation lediglich darauf beschränkt, unterschiedliche Sprachen miteinander zu vergleichen, mit denen es möglich ist, die KAMP-spezifischen Regeln zur Änderungsausbreitung zu formulieren. Dazu kommt Goal Question Metric (GQM) zum Einsatz. GQM ist eine systematische Vorgehensweise zur Erstellung von Qualitätsmodellen im Bereich der Softwareentwicklung [BCR94]. Aus einer oder mehreren Zielsetzungen werden zuerst Fragen und anschließend daraus wiederum Metriken abgeleitet.

Das **Ziel** der Evaluation ist es, zu zeigen, dass CPRL besser für die Darstellung der in Abschnitt 7.1 genannten Arten der Änderungsausbreitungsanalyse geeignet ist, als konkurrierende Sprachen. Die konkurrierenden Sprachen werden in zwei Kategorien eingeteilt: *imperativ* und *deklarativ*. Als imperativer Sprache wird Xtend der Vorzug gegenüber Java gegeben. Xtend hat den Ruf, ein ausdrucksstarker Dialekt von Java zu sein. Es wird daher davon ausgegangen, dass Xtend in den meisten Fällen gleich kompakte oder kompaktere Sprachkonstrukte bietet als Java. Als deklarative Sprachen werden VIATRA und OCL mit CPRL verglichen. Der Vergleich mit deklarativen Sprachen ist von besonderer Bedeutung, da CPRL selbst ebenfalls eine deklarative Sprache ist.

Es lässt sich die folgende zentrale **Frage** zu dem oben genannten Ziel ableiten:

Q Welche Sprache ist leichter zu verwenden?

Des Weiteren könnte danach gefragt werden, welche Sprache zu weniger semantischen Fehlern führt oder welche Sprache verständlicher für Personen mit wenig Erfahrung in

universellen Programmiersprachen ist. Für diese Fragen lassen sich nur schwer Metriken finden, die sich einzig auf die Syntax der Sprache beziehen. Eine Fallstudie wäre die geeignete Möglichkeit, Metriken für diese Fragen aufzustellen und diese zu evaluieren. Da dies allerdings den Rahmen dieser Arbeit überschreiten würde, wird lediglich die folgende Metrik M für die Frage Q aufgestellt und evaluiert:

M minimale Anzahl an Sprachelementen, um eine bestimmte Art von Regel zu formulieren

Die Metrik M kann alleine durch die Analyse der Syntax einer Sprache quantifiziert werden. Es wird davon ausgegangen, dass eine Sprache S_1 leichter zu verwenden ist, als eine Sprache S_2, wenn die Anzahl der Sprachkonstrukte C_1 für S_1 für die Mehrheit der betrachteten Regelbestandteile kleiner oder gleich der Anzahl an Sprachkonstrukten C_2 für S_2 ist.

Der technische Bericht [SBT14] von Siket u. a. listet verschiedene Varianten der Metrik *Anzahl Programmzeilen* (engl. lines of code, LOC) auf. Der Bericht unterscheidet zwischen physischen Programmzeilen (engl. physical lines of code, PLOC) und logischen Programmzeilen (engl. logical lines of code, LLOC). Die Definition von PLOC schließt alle Zeilen eines Programms, einschließlich Kommentaren und Leerzeilen, ein. Die Metrik M ist besser geeignet als PLOC. Jede der verglichenen Sprachen erlaubt es, mehrere Anweisungen in einer Zeile unterzubringen. Die Anzahl an Programmzeilen kann dadurch für jede Art von Regel in jeder Vergleichssprache auf 1 reduziert werden. Das macht PLOC weniger aussagekräftig als M. Die LLOC-Metrik hingegen wird verwendet, um eine bereinigte Anzahl an Programmzeilen zu quantifizieren. Es gibt dabei zwei verschiedene, nicht standardisierte, Definitionen für die LLOC-Metrik [SBT14]. Die erste Definition verwendet die Anzahl der Programmzeilen im Quelltext, wobei bestimmte Zeilen (z.B. Leerzeilen, Kommentarzeilen) ausgenommen werden. Die zweite Definition verwendet die tatsächlichen syntaktischen Sprachelemente der zu Grunde liegenden Programmiersprache. Diese Metrik ist somit abhängig von der verwendeten Sprache und der Zählweise deren Sprachelemente. Verwendet man die zweite Definition, so kann die Metrik M definiert werden als: **die Anzahl LLOC, um eine bestimmte Art von Regel zu formulieren**.

7.4.2 Präzisierung der LLOC-Metrik

Da die entscheidende Metrik M auf der LLOC-Metrik aufbaut, wird im Folgenden für jede der betrachteten Sprachen festgelegt, was ein zählbares syntaktisches Sprachelement im Sinne von LLOC ist.

CPRL Die folgenden Sprachelemente innerhalb von Regeln werden berücksichtigt: alle Arten an Regelquellen, Abfragen und Referenzen. Insbesondere Referenzen auf Metaklassen, Java-Typen, Instanz-Deklarationen und Strukturmerkmalen werden separat gezählt.

Xtend Alle Anweisungen und Ausdrücke werden berücksichtigt.

OCL Alle OCL-Anweisungen werden berücksichtigt. Dies umfasst den Kontext, Model Entity, Operationen und Schlüsselwörter.

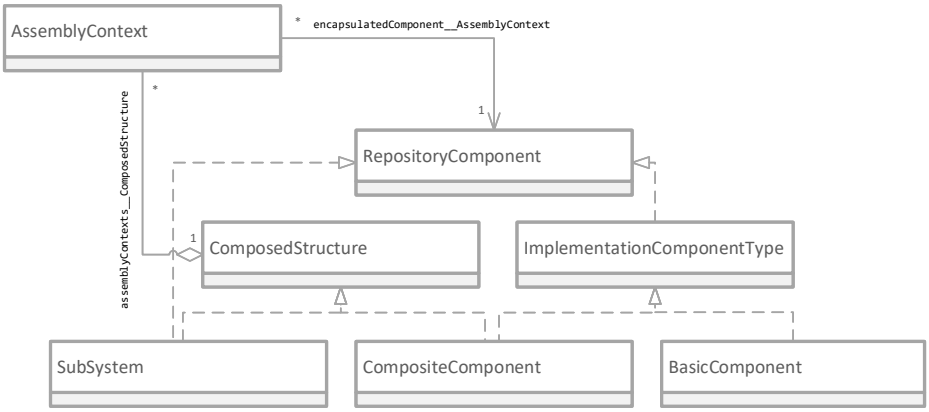

Abbildung 7.1: UML-Klassendiagramm für einen Metamodell-Ausschnitt aus dem PCM

VIATRA Alle Parameter eines Patterns und Bedingungen innerhalb eines Patterns werden berücksichtigt. Innerhalb von Bedingungen zählen alle Referenzen auf Metaklassen, Strukturmerkmale und Parameter als separate syntaktische Sprachelemente. Der Typ eines Parameters bei seiner Deklaration wird als eigenständiges syntaktisches Element gewertet, da er als Bedingung interpretiert werden kann. Die Variable selber wird nicht gewertet.

Das Sprachelement Regel in seiner Gesamtheit und der Regelname werden nicht als Sprachelemente gewertet. Es werden nur die Sprachelemente berücksichtigt, die innerhalb einer Regel enthalten sind. Variablen werden nicht bei ihrer Deklaration, sondern nur bei ihrer Verwendung in einem Ausdruck gewertet. Imports werden nicht berücksichtigt.

7.4.3 Modell für die Evaluation

Das im Folgenden beschriebene Beispielmodell wird verwendet, um die Formulierung bestimmter Änderungsausbreitungsregeln in verschiedenen Sprachen zu vergleichen. Das UML-Klassendiagramm in Abbildung 7.1 beinhaltet zunächst eine Übersicht über die betrachteten Metaklassen aus dem PCM. Es handelt sich dabei um einen Ausschnitt, der nur die wesentlichen Merkmale der jeweiligen Metaklassen beinhaltet, die für das Beispiel relevant sind.

In Abbildung 7.2 ist das Beispielmodell in Form eines UML-Objektdiagramms dargestellt. Es beinhaltet eine Struktur aus zusammengesetzten Komponenten. Durch die Verwendung von zusammengesetzten Komponenten, ist es möglich einen Zyklus an Referenzen auf Modellebene zu bilden. Das eignet sich besonders gut, um in den folgenden Abschnitten die Rekursion in Regeln zu veranschaulichen und zu evaluieren. Auf das Beschriften der Assoziationen wurde, im Hinblick auf Übersichtlichkeit, verzichtet. Es handelt sich dabei jeweils implizit um die Referenz *assemblyContexts__ComposedStructure* von *ComposedStrukture* zu *AssemblyContext* oder *encapsulatedComponent__AssemblyContext* von *AssemblyContext* zu *RepositoryComponent*.

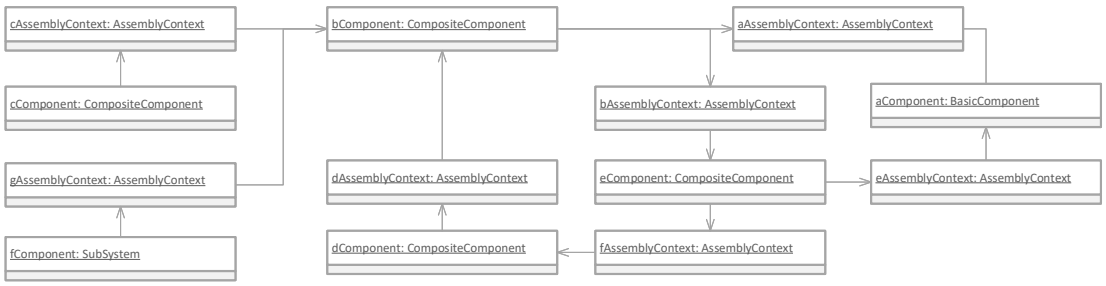

Abbildung 7.2: UML-Objektdiagramm eines Beispielmodells für die Evaluation des Vorteils von CPRL gegenüber konkurrierenden Sprachen

7.4.4 Quantifizierung der LLOC-Metrik

Um die unterschiedlichen Sprachen in KAMP einzubinden, müssen Schnittstellen geschaffen werden. Diese sind in Kapitel 9 angehängt. CPRL besitzt standardmäßig eine Anbindung an KAMP. Für OCL ist die Anbindung in Abschnitt 9.1, für Xtend in Abschnitt 9.2 und für VIATRA in Abschnitt 9.3 beschrieben.

In den folgenden Abschnitten werden typische Regelbestandteile in jeder der vier zu vergleichenden Sprachen formuliert. Sie sind kompakt formuliert und bestehen folglich aus so wenigen Sprachelementen wie möglich. Was als Sprachelement in der jeweiligen Sprache gewertet wird, ist in Abschnitt 7.4.2 aufgezählt. Die Listings in den folgenden Abschnitten beinhalten eine grüne Färbung für jedes Sprachelement, das gezählt wird.

7.4.4.1 Regelquelle

Als erstes wird nur ein Teil einer Regel betrachtet: die Selektion von Eingabeelementen auf der Metamodellebene. Alle Anfangsmarkierungen mit der zugehörigen Metaklasse *SubSystem* werden durch die folgenden Regeln ermittelt. Angenommen, *fComponent* ist eine Anfangsmarkierung im Beispielmodell, so ist es auch das Ergebnis der folgenden Regelausführungen. Der Xtend-Quelltext in den folgenden Listings muss zur Ausführung in Eclipse in eine sogenannte Wrapper-Methode eingefügt werden. Das ist nötig, da das KAMP-Framework den Xtend-Ausdrücken Eingabewerte zur Verfügung stellen muss. Es werden für die Evaluation insgesamt vier Wrapper-Methoden mit jeweils unterschiedlichen Parameterlisten verwendet. Die Methoden sind im Abschnitt 9.2 des Anhangs zu finden und mit Version A bis D benannt. Für den Xtend-Quelltext in diesem Abschnitt wird die Variante A der Wrapper-Methode aus Listing 9.2 verwendet. Es wird im Folgenden immer diese Variante verwendet, wenn nichts Abweichendes festgelegt ist.

CPRL

```
rule A: metaclass(pcm::SubSystem) // 2
```

Xtend

```
source.filter(o | o instanceof SubSystem) // 5
```

OCL

```
context SubSystem; // 2
```

VIATRA

```
pattern isSeedElementByName(seedElement) {
    SubSystem.entityName(seedElement, "fComponent"); // 4
}
```

DSL	LLOC
CPRL	2
Xtend	5
OCL	2
VIATRA	4 + 5 pro zusätzlicher Anfangsmarkierung

Tabelle 7.1: Anzahl an LLOC für eine Regelquelle

In Tabelle 7.1 sind die Anzahl LLOC pro DSL für die Regelquelle eingetragen. Es zeigt sich, dass CPRL und OCL durch spezielle Sprachelemente zum Filtern der Eingabe nach Typ der Metaklasse, nur zwei LLOC zur Darstellung brauchen. Die iterative Vorgehensweise von Xtend benötigt fünf LLOC. Da es in VIATRA keine Möglichkeit gibt, Anfangsmarkierungen von KAMP zu übergeben, muss für jede Anfangsmarkierung eine eigene Bedingung formuliert werden. Jede dieser Bedingungen besteht aus vier LLOC. Wenn mehr als eine Anfangsmarkierung verwendet wird, so muss jede weitere Bedingung mit einer Pattern-Disjunktion angefügt werden, die wiederum als eigenes Sprachelement gewertet wird. Dadurch sind es in VIATRA fünf LLOC pro zusätzlicher Anfangsmarkierung.

Es werden keine weiteren Arten von Regelquellen betrachtet. Die Selektion von Elementen auf der Modellebene wird im folgenden Abschnitt unabhängig von der Regelquelle betrachtet.

7.4.4.2 Selektion auf Modellebene

Dieser Abschnitt betrachtet eine Regel, die Modellelemente anhand bestimmter Eigenschaften selektiert und anschließend markiert. Als Anfangsmarkierung werden *cComponent* und *dComponent* gesetzt. Die Regeln sollen das Modellelement *cComponent* anhand seines Namens markieren. Es werden im Folgenden nur jene Sprachelemente gewertet, die Teil der Selektion sind. Die Regel als Ganzes, sowie die Regelquelle werden nicht gezählt.

CPRL

```
find cComponent: pcm::CompositeComponent { // 2
    it.entityName === "cComponent" // 4
}

rule A: metaclass(ecore::EObject) [cComponent] // 2
```

Xtend

```
source.filter(o | o instanceof CompositeComponent // 4
    && o.entityName === "cComponent") // 5
```

OCL

```
context EObject;
self -> select(e | oclIsKindOf(pcm::repository::CompositeComponent) // 4
    And e.entityName = "cComponent") // 5
```

VIATRA

```
pattern ruleA(element) {
    find isSeedElement(element);
    CompositeComponent.entityName(element, "cComponent"); // 4
}
```

VIATRA*

```
pattern ruleA2(element) {
    find isSeedElement(element);
    CompositeComponent.entityName(element, z); // 4
    check(z === "cComponent"); // 4

    // or as block expression
    check({ // 1
        z === "cComponent" // 3
    });
}
```

* Verwendung eines allgemeinen Prädikats

In Tabelle 7.2 sind die Anzahl LLOC pro DSL für die Selektion auf Modellebene eingetragen. In CPRL werden acht LLOC benötigt. Es existiert eine kürzere Form, wenn statt des *entityName*-Attributs das *ID*-Attribut verglichen werden soll. Für eine bessere Vergleichbarkeit werden allerdings keine Randfälle betrachtet, sondern allgemeine Prädikate. In Xtend und OCL werden jeweils 9 LLOC benötigt. In VIATRA gibt es ein spezielles Konstrukt, um Attributwerte von Modellelementen auf Gleichheit zu prüfen. Daher benötigt VIATRA im Beispielfall nur vier LLOC. Verwendet man

DSL	LLOC
CPRL	8
Xtend	9
OCL	9
VIATRA	8* (4)

Tabelle 7.2: Anzahl an LLOC für eine Selektion auf Modellebene

allerdings das allgemeine Prädikat, so benötigt VIATRA acht LLOC. Die Sprachen VIATRA, CPRL und Xtend verwenden jeweils Xbase, um Prädikate darzustellen. Es ist daher möglich, gleich mächtige Bedingungen durch sie zu formulieren.

7.4.4.3 Selektion auf Metamodellebene

Die Selektion von Elementen auf der Metamodellebene an der Regelquelle wurde bereits in Abschnitt 7.4.4.1 betrachtet. Dieser Abschnitt betrachtet die Selektion innerhalb einer Regel. Als Anfangsmarkierung sind *cComponent* und *aComponent* gegeben. Die Regeln sollen alle Elemente vom Typ *CompositeComponent* markieren. Das ist im Beispielmodell: *cComponent*. Es werden im Folgenden nur jene Sprachelemente gewertet, die Teil der Selektion sind. Die Regel als Ganzes, sowie die Regelquelle werden nicht gezählt.

CPRL

```
rule B: metaclass(ecore::EObject)
    <org.palladiosimulator.pcm.repository.CompositeComponent>; // 2
```

Xtend

```
source.filter(o | o instanceof CompositeComponent) // 4
```

OCL

```
context EObject;
oclAsType(pcm::repository::CompositeComponent) // 2
```

VIATRA

```
pattern ruleB(element: CompositeComponent) { // 1
    find isSeedElement(element);
}
```

In Tabelle 7.3 sind die Anzahl LLOC pro DSL für die Selektion auf Metamodellebene eingetragen. CPRL und OCL schneiden mit jeweils zwei LLOC gleich gut ab. Xtend benötigt, ebenso wie bei der Regelquelle, vier LLOC. VIATRA besitzt mit nur einer LLOC die kompakteste Syntax zur Selektion auf Metamodellebene. Dies liegt daran, dass VIATRA den Typ einer Variable an den deklarierten Typ des zugehörigen Parameters bindet. Bei der Zuweisung eines Modellelements wird überprüft, ob dessen Typ entweder dem Typ der Variable oder einem Untertyp davon entspricht.

DSL	LLOC
CPRL	2
Xtend	4
OCL	2
VIATRA	1

Tabelle 7.3: Anzahl an LLOC für eine Selektion auf Metamodellebene

7.4.4.4 Vorwärtsnavigation

Dieser Abschnitt thematisiert drei Arten von Vorwärtsnavigation. Das sind die Navigation basierend auf Strukturmerkmal, Metaklasse und Instanz. Als Anfangsmarkierung werden *aAssemblyContext* und *bAssemblyContext* verwendet. Die folgenden Regeln verwenden das Strukturmerkmal zur Navigation, um *aComponent* und *eComponent* zu markieren.

CPRL

```
rule C: metaclass(pcm::AssemblyContext)
    -> feature(encapsulatedComponent__AssemblyContext); // 2
```

Xtend

```
source.filter(o | o instanceof AssemblyContext)
    .map(o | o.encapsulatedComponent__AssemblyContext) // 3
```

OCL

```
context AssemblyContext;
encapsulatedComponent__AssemblyContext // 1
```

VIATRA

```
pattern ruleC(input, output) {
    find isSeedElement(input);
    AssemblyContext.encapsulatedComponent__AssemblyContext(input, output); // 4
}
```

Am kompaktesten wird die Navigationsart über das Strukturelement durch OCL dargestellt. Da der Kontext implizit ist, muss lediglich der Name des Strukturmerkmals angegeben werden. Somit ist die Anzahl LLOC für OCL eins. CPRL benötigt zwei LLOC, Xtend drei und VIATRA vier.

Als Nächstes wird die Navigation mittels Metaklasse betrachtet. Bei dieser Navigationsart werden alle Referenzen verfolgt, die auf ein Modellelement mit einer bestimmten Metaklasse zeigen. Als Anfangsmarkierung dienen wieder *aAssemblyContext* und *bAssemblyContext*. Es soll auf die Metaklasse *BasicComponent* navigiert werden. Folglich markieren alle Regeln das Modellelement *aComponent*.

CPRL

```
rule D: metaclass(pcm::AssemblyContext)
    -> metaclass(pcm::BasicComponent); // 2
```

Xtend

```
source.filter(o | o instanceof AssemblyContext)
    .flatMap([ // 1
        it.eClass.EStructuralFeatures.stream.map(f | it.eGet(f)) // 8
            .filter([o | // 1
                BasicComponent.isAssignableFrom(o.class) // 4
            ]);
    ])
```

In CPRL werden zwei LLOC für die Vorwärtsnavigation basierend auf Metaklasse benötigt. In Xtend sind es 14 LLOC. Sowohl VIATRA als auch OCL unterstützen diese Art der Navigation überhaupt nicht. Es wurde keine Möglichkeit gefunden, um in diesen Sprachen über die Strukturelemente zu iterieren.

Schließlich wird die Navigation mittels Instanz betrachtet. Bei dieser Navigationsart werden alle Referenzen verfolgt, die auf ein Modellelement zeigen, welches ein bestimmtes Prädikat erfüllt. Im folgenden Beispiel sind die Modellelemente *dAssemblyContext* und *fAssemblyContext* als Anfangsmarkierung gesetzt. Die Regeln markieren alle Modellelemente, welche auf ein *CompositeComponent* verweisen, das auf genau zwei *AssemblyContext*s verweist. Im Beispiel ist das *bComponent*.

CPRL

```
find componentWithTwoAssemblyContexts: pcm::CompositeComponent { // 2
    it.assemblyContexts__ComposedStructure.length === 2 // 5
}

rule E: metaclass(pcm::AssemblyContext)
    -> instance(componentWithTwoAssemblyContexts); // 2
```

Xtend

```
source.filter(o | o instanceof AssemblyContext)
    .flatMap([ // 1
        it.eClass.EStructuralFeatures.stream.map(f | it.eGet(f)) // 8
            .filter([o | // 1
                CompositeComponent.isAssignableFrom(o.class) // 4
            ])
            .map([o | CompositeComponent.cast(o)]) // 4
            .filter([o | // 1
                o.assemblyContexts__ComposedStructure.length === 2 // 5
            ])
    ])
```

In CPRL werden neun LLOC für die Vorwärtsnavigation basierend auf Instanz benötigt. In Xtend sind es 24 LLOC. Sowohl VIATRA als auch OCL unterstützen diese Art der Navigation überhaupt nicht. Es wurde keine Möglichkeit gefunden, um in diesen Sprachen über die Strukturelemente zu iterieren. In Tabelle 7.4 sind die Anzahl LLOC pro Navigationsart und DSL zusammengefasst.

	Navigationsart		
DSL	Strukturmerkmal	Metaklasse	Instanz
CPRL	2	2	9
Xtend	3	14	24
OCL	1	nicht vorhanden	nicht vorhanden
VIATRA	4	nicht vorhanden	nicht vorhanden

Tabelle 7.4: Anzahl an LLOC für die verschiedenen Arten an Vorwärtsnavigation

7.4.4.5 Rückwärtsnavigation

Dieser Abschnitt thematisiert drei Arten von Rückwärtsnavigation. Das sind Navigation basierend auf Strukturmerkmal, Metaklasse und Instanz. Als Anfangsmarkierung werden *dAssemblyContext* und *fAssemblyContext* gewählt. Die folgenden Regeln verwenden das Strukturmerkmal zur Navigation, um *dComponent* und *eComponent* zu markieren. Für den Xtend-Quelltext in diesem Abschnitt wird die Variante B der Wrapper-Methode aus Listing 9.3 verwendet.

CPRL

```
rule F: metaclass(pcm::AssemblyContext)
    <– metaclass(pcm::CompositeComponent, assemblyContexts__ComposedStructure); // 3
```

Xtend

```
source.filter(o | o instanceof AssemblyContext)
    .flatMap([ // 1
        adapter.getInverseReferences(it).stream.filter([s | // 5
            CompositeComponent.isAssignableFrom(s.EObject.class) // 5
            && s.EStructuralFeature.name === "assemblyContexts__ComposedStructure" // 6
        ]).map(s | s.EObject); // 3
    ]);
```

OCL

```
context AssemblyContext;
pcm::repository::CompositeComponent // 1
    .allInstances() // 1
    –> select(s | s.assemblyContexts__ComposedStructure –> includes(self)) // 5
```

VIATRA

```
pattern ruleF(input, output) {
    find isSeedElement(input);
    CompositeComponent.assemblyContexts__ComposedStructure(output, input); // 4
}
```

In CPRL werden drei LLOC für die Rückwärtsnavigation basierend auf Strukturmerkmal benötigt. In Xtend sind es 20 LLOC, in OCL sieben LLOC und in VIATRA vier LLOC.

Die Rückwärtsnavigation basierend auf Instanz und Metaklasse wird von OCL und VIATRA nicht unterstützt. Der Grund ist der gleiche wie bei der Vorwärtsnavigation: Es existiert keine Möglichkeit, um über die Strukturmerkmale eines Modellelements zu iterieren. Es werden folglich nur die Formulierungen in Xtend und CPRL verglichen.

Als Nächstes wird die Navigation basierend auf Metaklasse evaluiert. Als Anfangsmarkierungen sind *gAssemblyContext* und *cAssemblyContext* gesetzt. Die Regeln markieren jene Modellelemente, deren zugehörige Metaklasse *SubSystem* entspricht. Das ist im Beispielmodell das Modellelement *fComponent*.

CPRL

```
rule G: metaclass(pcm::AssemblyContext)
    <– metaclass(pcm::SubSystem); // 2
```

Xtend

```
source.filter(o | o instanceof AssemblyContext)
    .flatMap([ // 1
        adapter.getInverseReferences(it).stream // 4
        .map(s | s.EObject) // 3
        .filter([o | // 1
            SubSystem.isAssignableFrom(o.class) // 4
        ]);
    ]);
```

In CPRL werden zwei LLOC und in Xtend 13 LLOC für diese Navigationsart benötigt. Schließlich wird die Navigation basierend auf Instanz evaluiert. Als Anfangsmarkierungen werden *dAssemblyContext* und *fAssemblyContext* gesetzt. Die Regel soll jene *CompositeComponent*s markieren, die auf ein *AssemblyContext* aus der Menge der Anfangsmarkierungen zeigen. Zudem soll das zu markierende Modellelement auf genau zwei *AssemblyContexts* verweisen. Das Modellelement *eComponent* erfüllt diese Bedingungen und wird deshalb durch die folgenden Regeln markiert.

CPRL

```
find componentWithTwoAssemblyContexts: pcm::CompositeComponent { // 2
    it.assemblyContexts__ComposedStructure.length === 2 // 5
}

rule H: metaclass(pcm::AssemblyContext)
    <– instance(componentWithTwoAssemblyContexts); // 2
```

Xtend

```
source.filter(o | o instanceof AssemblyContext)
    .flatMap([ // 1
        adapter.getInverseReferences(it).stream // 4
        .map(s | s.EObject) // 3
        .filter([o | // 1
            CompositeComponent.isAssignableFrom(o.class) // 4
        ])
        .map([o | CompositeComponent.cast(o)]) // 4
        .filter([o | // 1
            o.assemblyContexts__ComposedStructure.length === 2 // 5
        ]);
    ]);
```

In CPRL werden neun LLOC und in Xtend 23 für diese Navigationsart benötigt. In Tabelle 7.5 sind die Anzahl LLOC pro Navigationsart und DSL zusammengefasst.

	Navigationsart		
DSL	Strukturmerkmal	Metaklasse	Instanz
CPRL	3	2	9
Xtend	20	13	23
OCL	7	nicht vorhanden	nicht vorhanden
VIATRA	4	nicht vorhanden	nicht vorhanden

Tabelle 7.5: Anzahl an LLOC für die verschiedenen Arten an Rückwärtsnavigation

7.4.4.6 Verursachungselement

Das Verursachungselement ist ein KAMP-spezifischer Bestandteil der Änderungsausbreitung. Es sei *dComponent* die einzige Anfangsmarkierung. Die Regel soll nun die verschachtelte Komponente ermitteln. Hierzu wird zunächst das Strukturmerkmal *assemblyContexts__ComposedStructure* und anschließend *encapsulatedComponent__AssemblyContext* verfolgt. Die verschachtelte Komponente in diesem Beispiel ist *bComponent*. Als Verursachungselement soll jenes Modellelement ermittelt werden, das direkt auf das zu markierende Modellelement zeigt. In diesem Beispiel also *dAssembly-Context*. Für den Xtend-Quelltext in diesem Abschnitt wird die Variante C der Wrapper-Methode aus Listing 9.4 verwendet.

CPRL

```
rule I: metaclass(pcm::CompositeComponent)
    -> feature(assemblyContexts__ComposedStructure) // 2
    *-> feature(encapsulatedComponent__AssemblyContext); // 3
```

Xtend

```
source.filter(o | o instanceof AssemblyContext)
    .flatMap([ // 1
        it.affectedElement.assemblyContexts__ComposedStructure.stream.map(e | // 5
            new CausingEntityMapping(e) // 3
        )
    ]).map([ // 1
        new CausingEntityMapping( // 2
            it.affectedElement.encapsulatedComponent__AssemblyContext, // 3
            it // 1
        )
    ]);
```

VIATRA

```
pattern ruleI(input, causingEntity, output) {
    find isSeedElement(input);
    CompositeComponent.assemblyContexts__ComposedStructure(input, causingEntity); // 4
    AssemblyContext.encapsulatedComponent__AssemblyContext(causingEntity, output); // 4
}
```

Die Regeln bestehen jeweils aus zwei Vorwärtsnavigationen, die mittels Strukturmerkmal referenzieren. Um die Anzahl an LLOC für die Verursachungselementmarkierung zu ermitteln, wird jeweils die Anzahl an LLOC für zwei Vorwärtsnavigationen basierend auf Strukturmerkmal von der Anzahl aus obigen Regeln abgezogen. Das ergibt eine LLOC in CPRL, zehn in Xtend und null in VIATRA. Der Wert liegt bei null für VIATRA, da das Verursachungselement nicht explizit selektiert werden muss, sondern das Hinzufügen eines Parameters ausreicht. Es ist streitbar, ob ein Parameter in VIATRA als eigenständiges Sprachelement zählen sollte. Ein Argument dagegen ist, dass er weder ein Ausdruck noch eine typische Anweisung ist. Ein Argument dafür ist hingegen, dass er faktisch den Rückgabewert der Regel modifiziert. Um eine konsistente Definition der LLOC aufrecht zu halten, wird der Parametername in VIATRA als Deklaration gewertet. Daraus folgt, dass in dieser Sprache null LLOC für die Verursachungselementmarkierung notwendig sind. In OCL konnte keine praktikable Lösung gefunden werden. Für die Evaluation ist es unerheblich, da ohnehin keine kompaktere Lösung als in VIATRA oder CPRL möglich ist. Eine Zusammenfassung der Ergebnisse für diesen Absatz ist in Tabelle 7.6 zu finden.

DSL	LLOC
CPRL	1
Xtend	10
OCL	nicht vorhanden
VIATRA	0 (1)

Tabelle 7.6: Anzahl an LLOC für eine Verursachungselementmarkierung

7.4.4.7 Rekursion

Dieser Abschnitt thematisiert die Rekursion. Als Rekursion wird die Vorgehensweise bezeichnet, dass eine Regel immer wieder ausgeführt wird, solange sie neue Modellelemente markiert. Jede weitere Ausführung verwendet die Anfangsmarkierungen und die markierten Elemente aus vorangehenden Regelausführungen als Eingabe. Als Anfangsmarkierung wird das Modellelement *bComponent* gesetzt. Die Rekursion wird für insgesamt zwei Vorwärtsnavigationen ausgeführt. Diese verfolgen zuerst das Strukturmerkmal *assemblyContexts__ComposedStructure* und anschließend *encapsulatedComponent__AssemblyContext*. Die Regeln markieren die Modellelemente *aComponent*, *eComponent*, *dComponent* und *bComponent*. Für den Xtend-Quelltext in diesem Abschnitt wird die Variante D der Wrapper-Methode aus Listing 9.5 verwendet.

CPRL

```
recursive { // 1
    rule J: metaclass(pcm::CompositeComponent)
        -> feature(assemblyContexts__ComposedStructure) // 2
        -> feature(encapsulatedComponent__AssemblyContext); // 2
}
```

Xtend

```
val isNewValue = new AtomicBoolean
val List<RepositoryComponent> res = newArrayList;
val sourceElements = source.filter(o | o instanceof CompositeComponent).collect(Collectors.
    ↪ toList)

do {
    isNewValue.set(false)
    val List<RepositoryComponent> temp = newArrayList;
    Stream.concat(sourceElements.stream, res.stream)
        .filter(o | CompositeComponent.isAssignableFrom(o.class))
        .map(o | CompositeComponent.cast(o))
        .flatMap(o | o.assemblyContexts__ComposedStructure.stream)
        .map(o | o.encapsulatedComponent__AssemblyContext)
        .forEach() [e |
            val contains = new AtomicBoolean;
            res.forEach[e2 |
                if(eqHelper.equals(e, e2)) {
                    contains.set(true)
                }
            ];

            if(!contains.get) {
                temp.add(e)
                isNewValue.set(true)
            }
        ];

    res.addAll(temp)
} while(isNewValue.get());

return res.stream;
```

OCL

```
context CompositeComponent
self->closure(e | if e.oclIsKindOf(CompositeComponent) then // 7
    e.oclAsType(CompositeComponent) // 3
        .assemblyContexts__ComposedStructure // 1
        .encapsulatedComponent__AssemblyContext // 1
    else Bag{e} endif) // 4
```

VIATRA

```
pattern ruleJ(element) {
    find isSeedElement(seedElement);
    find oneLookup + (seedElement, element); // 5
}

pattern oneLookup(seedElement, result) {
    CompositeComponent.assemblyContexts__ComposedStructure(seedElement,
        ↪ anotherElement); // 4
    AssemblyContext.encapsulatedComponent__AssemblyContext(anotherElement, result); // 4
}
```

Die Regeln bestehen jeweils aus zwei Vorwärtsnavigationen, die mittels Strukturmerkmal referenzieren. Um die Anzahl an LLOC für die Rekursion zu ermitteln, wird jeweils die Anzahl an LLOC für zwei Vorwärtsnavigationen basierend auf Strukturmerkmal von der Anzahl aus obigen Regeln abgezogen. Das ergibt eine LLOC in CPRL, über 60 in Xtend, 14 in OCL und fünf in VIATRA. Die Anzahl der Sprachelemente in Xtend ist sehr groß und stark von der konkreten Implementierung abhängig. Da es an einigen Stellen streitbar ist, welches Sprachelement gezählt werden sollte, wird lediglich eine Abschätzung nach unten angegeben. Die Xtend-Implementierung im obigen Listing besitzt mindestens 40 Sprachelemente, welche der Rekursion zuzuschreiben sind. Eine Zusammenfassung der Ergebnisse für diesen Absatz ist in Tabelle 7.7 zu finden.

DSL	LLOC
CPRL	1
Xtend	>40
OCL	14
VIATRA	5

Tabelle 7.7: Anzahl an LLOC für eine Rekursion

7.4.5 Ergebnis

Die Anzahl der ermittelten LLOC für die einzelnen Regelbestandteile pro DSL, ist in Tabelle 7.8 zusammengefasst. Die Summe der Regelbestandteile mit minimaler Anzahl an LLOC ist für CPRL mit acht am größten. Auf dem zweiten Platz ist VIATRA mit drei, gefolgt von OCL mit zwei und Xtend mit null. Somit gilt, dass CPRL, verglichen zu konkurrierenden Sprachen, für die meisten Regelbestandteile die kompakteste Formulierung ermöglicht. Daraus lässt sich schlussfolgern, dass CPRL auch für eine bestimmte Art von Regel die minimale Anzahl an Sprachelementen benötigt. Dadurch lässt sich nach dem GQM-Ansatz ableiten, dass CPRL leichter zu verwenden ist, als konkurrierende Sprachen. Damit konnte gezeigt werden, dass CPRL besser für die Darstellung der in Abschnitt 7.1 genannten Arten der Änderungsausbreitungsanalyse geeignet ist, als konkurrierende Sprachen. Es konnte durch die Evaluation ein Vorteil von CPRL gegenüber den verglichenen Sprachen ermittelt werden.

	DSL			
Regelbestandteil	CPRL	Xtend	OCL	VIATRA
Regelquelle	**2**	5	**2**	4 + 5x
Selektion				
- Modell	**8**	9	9	**8** (4)
- Metamodell	2	4	2	**1**
Vorwärtsnavigation				
- Strukturmerkmal	2	3	**1**	4
- Metamodell	**2**	14	n.v.	n.v.
- Modell	**9**	24	n.v.	n.v.
Rückwärtsnavigation				
- Strukturmerkmal	**3**	20	7	4
- Metamodell	**2**	13	n.v.	n.v.
- Modell	**9**	23	n.v.	n.v.
Verursachungselement	1	10	n.v.	**0** (1)
Rekursion	**1**	>40	14	5
$\sum$ Regelbestandteile mit minimaler Anzahl LLOC	**8**	0	2	3

Tabelle 7.8: Übersicht über die Anzahl LLOC pro Regelbestandteil und DSL
x: Anzahl der weiteren Anfangsmarkierungen
n.v.: nicht vorhanden

8 Zusammenfassung und Ausblick

Die Arbeit hat mit CPRL eine neue domänenspezifische Sprache für die Änderungsausbreitungsanalyse vorgestellt. Die Sprache unterstützt einen Großteil der momentan in KAMP vorhandenen Regeln. Da es sich allerdings um eine deklarative Sprache mit imperativen Prädikaten handelt, kann mit ihr nicht jede erdenkliche Java-Regel umgesetzt werden. Es wurde somit eine Abwägung zwischen der Mächtigkeit von Java und der Einfachheit von DSLs im Allgemeinen getroffen. Die dafür aufgestellten Annahmen wurden im Abschnitt 7.1 beschrieben. Die daraus entstehenden Einschränkungen wurden in Abschnitt 7.3.2 thematisiert. Es wurden insgesamt fünf Methoden in der Java-Implementierung der KAMP-Module BP, IS und REQ ermittelt, die nicht mittels CPRL formuliert werden können. Das entspricht einem Anteil von etwa zehn Prozent. Für diese nicht darstellbaren Methoden wurde in KAMP eine Möglichkeit implementiert, Regeln weiterhin in Java zu schreiben. In Abschnitt 6.3 wurde eine Lösung vorgestellt, wodurch die Java-Regeln mittels Dependency Injection in KAMP geladen werden können. Dies ermöglicht zugleich eine Verfeinerung von CPRL-Regeln. Die mittels Codegenerierung erstellten Java-Klassen für CPRL-Regeln können durch die injizierten Java-Regeln erweitert werden. Die Arbeit präsentierte somit einen hybriden Ansatz der Regelformulierung für KAMP.

Die Adressaten von CPRL sind Domänenexperten, die KAMP erweitern möchten. Sie können von den in dieser Arbeit vorgestellten Verbesserungen profitieren. Domänenexperten können unter Umständen kein Java oder möchten sich nicht in die Eclipse-Implementierung von KAMP einarbeiten, um ihre Modellierung durchzuführen. Nun müssen sie sich für die Formulierung rudimentärer Änderungsausbreitungen lediglich in die Syntax und Semantik von CPRL einarbeiten. Sollten sie dennoch umfangreichere Änderungsausbreitungen formulieren wollen, die von CPRL nicht unterstützt werden, so können sie dies mittels Java tun.

Schließlich wurde untersucht, ob sich für Domänenexperten Vorteile daraus ergeben, CPRL zu verwenden. Dazu wurde im GQM-Ansatz eine Metrik ermittelt, welche als Indikator dafür betrachtet werden kann, wie leicht eine Sprache zu verwenden ist. Diese Metrik wurde für CPRL und drei konkurrierende Sprachen evaluiert. Das Ergebnis dieser Evaluation ergab, dass die meisten Regelbestandteile in CPRL deutlich kompakter als in konkurrierenden Sprachen dargestellt werden können. Domänenexperten müssen somit für ein und dieselbe Regel in CPRL weniger Code schreiben als in anderen Sprachen. Die in dieser Arbeit vorgestellte Sprache bringt somit einen messbaren Vorteil für die Erweiterbarkeit des Softwarewerkzeugs KAMP.

In zukünftigen Arbeiten könnten der Sprache neue Sprachelemente hinzugefügt werden, um die Ausdrucksstärke zu erhöhen. Einige potentiellen Sprachelemente wurden bereits in Abschnitt 7.3.2 kurz beschrieben. Werden diese vollständig konzipiert und implementiert, müssen Domänenexperten bei der Formulierung komplexer Regeln seltener zu Java wechseln. Des Weiteren fehlt ein Konzept dafür, wie Regeln aus unterschiedlichen Regeldateien referenziert werden können. Diese Funktionalität ist wichtig, um Wiederverwendung von Regeln über Modulgrenzen hinweg zu ermöglichen. Letztlich könnte eine weiterführende Arbeit sich noch damit beschäftigen, wie gut die Performance des generierten Java-Codes im Vergleich zu manuell geschriebenen Java-Regeln ist. Es könnte zudem ein Verfahren erarbeitet werden, dass die Codegenerierung dahingehend optimiert, dass die Laufzeit verbessert wird.

9 Anhang

Der Anhang beinhaltet Codefragmente, welche für die Evaluation benötigt wurden, um Regeln in unterschiedlichen Sprachen in KAMP einzubinden.

9.1 Einbindung von OCL-Regeln in KAMP

Das Listing 9.1 wurde verwendet, um OCL-Regeln in KAMP einzubinden. Die Regeln werden als String an die Java-Implementierung von Classic OCL übergeben.

9.2 Einbindung von Xtend-Regeln in KAMP

In KAMP kommt Xtend bereits zum Einsatz. Das bedeutet, dass die Xtend-Regeln dort direkt in eine Xtend-Methode eingebettet werden können. Um die Regeln auszuführen, muss das KAMP-Framework bestimmte Parameter zur Verfügung stellen. Je nach Feature sind die Parameter unterschiedlich. Deshalb werden vier verschiedene Versionen einer Methode zur Ausführung einer Xtend-Regel bereitgestellt. Diese sind in den Listings 9.2, 9.3, 9.4 und 9.5 aufgeführt. Die Xtend-Regel muss jeweils an den Platzhalter *<Xtend Rule>* eingesetzt werden.

9.3 Einbindung von VIATRA-Regeln in KAMP

Um VIATRA-Regeln zu formulieren, muss zuerst das *VIATRA Query and Transformation SDK* installiert werden. Anschließend wird eine Instanz von Eclipse geöffnet, welche die KAMP-, VIATRA- und PCM-Plugins geladen hat. Dann wird ein neues *Query Project* erstellt. Innerhalb des Projektes wird eine neue *Query Definition* erstellt. Dies ist eine VQL-Datei, in welcher die jeweiligen Regeln als einzelne Pattern formuliert werden. Die Regeln werden schließlich mit der Ansicht *Query Results* ausgeführt. Das Ergebnis sind die betroffenen Elemente, welche als eine der Komponenten jedes Tupels für jedes Pattern abgelesen werden können.

Die VQL unterstützt das Konzept von Anfangsmarkierungen nicht, da sie nicht auf die Domäne der Änderungsausbreitung zugeschnitten ist. Es ist möglich dieses Konzept zu simulieren, indem ein Pattern mit dem Namen *isSeedElement* erstellt wird. Das Pattern selektiert ein Element basierend auf seiner ID. Für jedes Element, das anfänglich markiert ist, muss eine Pattern-Disjunktion verwendet werden. Die einzelnen Regeln müssen das isSeedElement-Pattern mittels find-Komposition einbeziehen. Ein Beispiel für das isSeedElement-Pattern ist in Listing 9.6 gegeben.

Des Weiteren müssen alle referenzierten Metaklassen importiert werden. Die Imports für alle in der Evaluation verwendeten Metaklassen sind ebenfalls in Listing 9.6 vorhanden.

```
import org.eclipse.emf.ecore.EObject;
import org.eclipse.ocl.ParserException;
import org.eclipse.ocl.ecore.OCL;
import org.eclipse.ocl.ecore.OCL.Helper;
import org.eclipse.ocl.ecore.OCLExpression;

class OclEvaluation {

    /**
    * Executes the rule which is given by the {@code queryString} for the {
       ↪ @code inputElement} and returns the result as object of type T.
    * @param queryString the ocl query
    * @param inputElement the input element the query is performed on
    * @param outputElementClass the type of the elements which are returned
       ↪ by the query
    * @return the elements which are affected by the rule
    * @throws ParserException thrown if {@code queryString} is invalid
    */
    public static <T extends EObject> T executeRule(String queryString,
            EObject inputElement,
            Class<T> outputElementClass) throws ParserException {

        OCL ocl = OCL.newInstance();
        Helper helper = ocl.createOCLHelper();
        helper.setContext(inputElement.eClass());

        OCLExpression query = helper.createQuery(queryString);

        // correct outputElementClass must be passed to this method
        @SuppressWarnings("unchecked")
        T result = (T) ocl.evaluate(inputElement, query);

        return result;
    }

}
```

Listing 9.1: Einbindung von OCL-Regeln mittels Java

```
import org.eclipse.emf.ecore.EObject

public static def Stream<EObject> executeRule(Stream<EObject> source) {

   <Xtend Rule>
}
```

Listing 9.2: Einbindung von Xtend-Regeln - Version A

```
import org.eclipse.emf.ecore.EObject
import org.eclipse.emf.ecore.util.ECrossReferenceAdapter

public static def Stream<EObject> executeRule(Stream<EObject> source,
    ↪ ECrossReferenceAdapter adapter) {

   <Xtend Rule>
}
```

Listing 9.3: Einbindung von Xtend-Regeln - Version B

```
import org.eclipse.emf.ecore.EObject
import edu.kit.ipd.sdq.kamp.ruledsl.support.CausingEntityMapping

public static def Stream<CausingEntityMapping<EObject, EObject>> executeRule(
    Stream<CausingEntityMapping<EObject, EObject>> source) {

    <Xtend Rule>
}
```

Listing 9.4: Einbindung von Xtend-Regeln - Version C

```
import org.eclipse.emf.ecore.EObject
import org.eclipse.emf.ecore.util.EcoreUtil.EqualityHelper

public static def Stream<EObject> executeRule(Stream<EObject> source,
    ↪ EqualityHelper eqHelper) {

    <Xtend Rule>
}
```

Listing 9.5: Einbindung von Xtend-Regeln - Version D

```
package thesis

// imports for all metaclasses which are used during evaluation
import "http://palladiosimulator.org/PalladioComponentModel/Repository/5.2"
import "http://palladiosimulator.org/PalladioComponentModel/SubSystem/5.2"
import "http://palladiosimulator.org/PalladioComponentModel/Core/Composition/5.2"
import "http://palladiosimulator.org/PalladioComponentModel/Core/Entity/5.2"
import "http://sdq.ipd.uka.de/Identifier/2.1"

// example which selects the two seed elements with ID
    ↪ _TVJd022yEeipyr94Phon1Q and _3tCCxH7-Eeite-xwFOHPjg
pattern isSeedElement(identifyable: Identifier) {
  Identifier.id(identifyable, "_TVJd022yEeipyr94Phon1Q");
} or {
  Identifier.id(identifyable, "_3tCCxH7-Eeite-xwFOHPjg");
}
```

Listing 9.6: VIATRA-Regeldatei

9.4 Regulärer Ausdruck für CPRL

In Listing 9.7 ist ein regulärer Ausdruck für die CPRL angegebenen. Mit diesem regulären Ausdruck ist gezeigt, dass CPRL, bis auf eine Ausnahme, eine reguläre Sprache ist. Die Platzhalter in Tabelle 9.1 wurden zur besseren Übersicht im regulären Ausdruck nicht aufgelöst. Das Nichtterminal *XBlockExpression* kann nicht mittels eines regulären Ausdrucks formuliert werden. Somit ist CPRL durch die Xbase-Prädikate keine reguläre Sprache. Es kann gezeigt werden, dass CPRL durch die Xbase-Prädikate mindestens kontextfrei ist. Da die Sprache CPRL mittels Xtext in einer LL(k)-Grammatik formuliert ist, kann sie maximal kontextfrei sein. Somit ist CPRL kontextfrei.

Platzhalter	Sprache	Symbolart	Produktionsregel
CodeBlock	Xbase	Nichtterminal	`XBlockExpression`
ID	Xtype	Terminal	`'^'? ('a'..'z'|'A'..'Z'|'$'\|'_') ('a'..'z'\|'A'..'Z"\|'$'|'_'|'0'..'9')*`
QualifizierterName	Xtype	Terminal	`ID ('.' ID)*`
STRING	Xtype	Terminal	`'"' ( '\\' . /* ('b'|'t'|'n'|'f'|'r''|'u'|'"'|"'- '\\') */ | !('\\'|'"') )* '"'? | "'"( '\\' . /* ('b'|'t'|'n'|'f'|'r'|'u'|'"'|"'- '\\') */ | !('\\'|"'") )* "'"?;`

Tabelle 9.1: Bedeutung der Platzhalter aus dem regulären Ausdruck

```
(('import' (STRING) 'as' (ID))*|('import-package' (QualifizierterName))*|('import-model' (
    ↪ STRING) 'as' (ID))*|('find' (ID) ':' (ID) '::' (QualifizierterName) (CodeBlock))*|('instance' (
    ↪ ID) ':' (ID) '::' (QualifizierterName))*)((((('rule' (ID) ':' (('metaclass(' (ID) '::' (
    ↪ QualifizierterName) ')')|('instance(' (ID) ')')|('rule(' (ID) ')'))(('*')? (('->' (('metaclass(' (
    ↪ ID) '::' (QualifizierterName) ')')|('instance(' (ID) ')')|('feature(' (ID) ')'))))|('<-' (('metaclass
    ↪ (' (ID) '::' (QualifizierterName) (',' (ID))? ')')|('instance(' (ID) ')'))))))|(((':' (
    ↪ QualifizierterName) ('AND' (QualifizierterName))* '>')|(('<!' (QualifizierterName) ('AND
    ↪ ' (QualifizierterName))* '>')))|(('[' (ID) ']'))|(('[' (CodeBlock) ']')))|(('->' 'rule(' (ID) ')')))
    ↪ ((((*')? (('->' (('metaclass(' (ID) '::' (QualifizierterName) ')')|('instance(' (ID) ')')|('feature
    ↪ (' (ID) ')')))|('<-' (('metaclass(' (ID) '::' (QualifizierterName) (',' (ID))? ')')|('instance(' (ID)
    ↪  ')'))))))|(((':' (QualifizierterName) ('AND' (QualifizierterName))* '>')|(('<!' (
    ↪ QualifizierterName) ('AND' (QualifizierterName))* '>')))|(('[' (ID) ']'))|(('[' (CodeBlock)
    ↪ ']')))|(('->' 'rule(' (ID) ')'))))* ';')((('rule' (ID) ':' (('metaclass(' (ID) '::' (QualifizierterName)
    ↪  ')')|('instance(' (ID) ')')|('rule(' (ID) ')'))(('*')? (('->' (('metaclass(' (ID) '::' (
    ↪ QualifizierterName) ')')|('instance(' (ID) ')')|('feature(' (ID) ')'))))|('<-' (('metaclass(' (ID)
    ↪ '::' (QualifizierterName) (',' (ID))? ')')|('instance(' (ID) ')'))))))|(((':' (QualifizierterName) ('
    ↪ AND' (QualifizierterName))* '>')|(('<!' (QualifizierterName) ('AND' (QualifizierterName)
    ↪ )* '>')))|(('[' (ID) ']'))|(('[' (CodeBlock) ']')))|(('->' 'rule(' (ID) ')'))))(((*')? (('->' ((
    ↪ metaclass(' (ID) '::' (QualifizierterName) ')')|('instance(' (ID) ')')|('feature(' (ID) ')')))|('<-'
    ↪  (('metaclass(' (ID) '::' (QualifizierterName) (',' (ID))? ')')|('instance(' (ID) ')'))))))|(((':' (
    ↪ QualifizierterName) ('AND' (QualifizierterName))* '>')|(('<!' (QualifizierterName) ('AND
    ↪ ' (QualifizierterName))* '>')))|(('[' (ID) ']'))|(('[' (CodeBlock) ']')))|(('->' 'rule(' (ID) ')'))))*
    ↪ ';'))*)|('recursive' '{' (('rule' (ID) ':' (('metaclass(' (ID) '::' (QualifizierterName) ')')|('
    ↪ instance(' (ID) ')')|('rule(' (ID) ')'))(('*')? (('->' (('metaclass(' (ID) '::' (QualifizierterName
    ↪ ) ')')|('instance(' (ID) ')')|('feature(' (ID) ')'))))|('<-' (('metaclass(' (ID) '::' (
    ↪ QualifizierterName) (',' (ID))? ')')|('instance(' (ID) ')'))))))|(((':' (QualifizierterName) ('
    ↪ AND' (QualifizierterName))* '>')|(('<!' (QualifizierterName) ('AND' (QualifizierterName)
    ↪ )* '>')))|(('[' (ID) ']'))|(('[' (CodeBlock) ']')))|(('->' 'rule(' (ID) ')'))))(((*')? (('->' ((('
    ↪ metaclass(' (ID) '::' (QualifizierterName) ')')|('instance(' (ID) ')')|('feature(' (ID) ')')))|('<-'
    ↪  (('metaclass(' (ID) '::' (QualifizierterName) (',' (ID))? ')')|('instance(' (ID) ')'))))))|(((':' (
    ↪ QualifizierterName) ('AND' (QualifizierterName))* '>')|(('<!' (QualifizierterName) ('AND
    ↪ ' (QualifizierterName))* '>')))|(('[' (ID) ']'))|(('[' (CodeBlock) ']')))|(('->' 'rule(' (ID) ')'))))*
    ↪ ';'))(('rule' (ID) ':' (('metaclass(' (ID) '::' (QualifizierterName) ')')|('instance(' (ID) ')')|('
    ↪ rule(' (ID) ')'))(('*')? (('->' (('metaclass(' (ID) '::' (QualifizierterName) ')')|('instance(' (ID
    ↪ ) ')')|('feature(' (ID) ')'))))|('<-' (('metaclass(' (ID) '::' (QualifizierterName) (',' (ID))? ')')|('
    ↪ instance(' (ID) ')'))))))|(((':' (QualifizierterName) ('AND' (QualifizierterName))* '>')|(('<!'
    ↪ (QualifizierterName) ('AND' (QualifizierterName))* '>')))|(('[' (ID) ']'))|(('[' (CodeBlock)
    ↪ ']')))|(('->' 'rule(' (ID) ')'))))(((*')? (('->' (('metaclass(' (ID) '::' (QualifizierterName) ')')|('
    ↪ instance(' (ID) ')')|('feature(' (ID) ')'))))|('<-' (('metaclass(' (ID) '::' (QualifizierterName)
    ↪ (',' (ID))? ')')|('instance(' (ID) ')'))))))|(((':' (QualifizierterName) ('AND' (
    ↪ QualifizierterName))* '>')|(('<!' (QualifizierterName) ('AND' (QualifizierterName))* '>')))
    ↪ |(('[' (ID) ']'))|(('[' (CodeBlock) ']')))|(('->' 'rule(' (ID) ')'))))* ';'))* '}')*)*
```

Listing 9.7: Regulärer Ausdruck für CPRL

Literatur

[AB93] R. S. Arnold und S. A. Bohner, “Impact analysis-Towards a framework for comparison”, in *Proceedings, Conference on Software Maintenance 1993*, D. N. Card, Hrsg., Los Alamitos, Calif: IEEE Computer Society Press, 1993, S. 292–301, ISBN: 0-8186-4600-4. DOI: 10.1109/ICSM.1993.366933.

[Apa18] Apache Software Foundation, *Apache Camel: DSL*, 23.04.2018. Adresse: http://camel.apache.org/dsl.html.

[BA96] S. A. Bohner und R. S. Arnold, *Software change impact analysis*. Los Alamitos Calif. u.a.: IEEE Computer Soc. Pr, 1996, ISBN: 0-8186-7384-2.

[BCR94] V. Basili, G. Caldiera und D. Rombach, “The goal question metric approach”, Jg. 1, 1994.

[Bel18] I. Belyantseva, “Eine Domänenspezifische Sprache für Änderungsausbreitungsregeln”, Diss., KIT, Karlsruhe, 15.06.2018.

[BKR09] S. Becker, H. Koziolek und R. Reussner, “The Palladio component model for model-driven performance prediction”, *Journal of Systems and Software*, Jg. 82, Nr. 1, S. 3–22, 2009, ISSN: 01641212. DOI: 10.1016/j.jss.2008.03.066.

[BLO03] L. Briand, Y. Labiche und L. O’Sullivan, *Impact analysis and change management of UML models*, 2003.

[BR08] R. Böhme und R. Reussner, “Validation of Predictions with Measurements”, in *Dependability metrics*, Ser. Lecture Notes in Computer Science, I. Eusgeld, Hrsg., Bd. 4909, Berlin: Springer, 2008, S. 14–18, ISBN: 978-3-540-68946-1. DOI: 10.1007/978-3-540-68947-8{\textunderscore}3.

[BV06] A. Balogh und D. Varró, *Advanced model transformation language constructs in the VIATRA2 framework*, 2006. DOI: 10.1145/1141277.1141575. Adresse: http://dl.acm.org/ft_gateway.cfm?id=1141575&type=pdf.

[BWLH] K. Busch, D. Werle, M. Löper und R. Heinrich, “A Cross-Disciplinary Language for Change Propagation Rules”, in.

[CSE04] P. J. Clarkson, C. Simons und C. Eckert, “Predicting change propagation in complex design”, *Journal of Mechanical Design*, Jg. 126, Nr. 5, S. 788–797, 2004.

[Ecl07] Eclipse Foundation, Hrsg., *Help - Eclipse Platform: Working with Classic OCL*, 2007. Adresse: https://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.ocl.doc%2Fhelp%2FOCLInterpreterTutorial.html.

[Ecl13] Eclipse Foundation, Hrsg., *Help - Eclipse Platform: MWE2*, 2013. Adresse: http://help.eclipse.org/kepler/index.jsp?topic=%2Forg.eclipse.xtext.doc%2Fcontents%2F118-mwe-in-depth.html.

[Ecl15] Eclipse Foundation, Hrsg., *EMF Documentation*, 2015. Adresse: http://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/package-summary.html.

[Ecl16] Eclipse Foundation, Hrsg., *Equinox | The Eclipse Foundation*, 2016. Adresse: http://www.eclipse.org/equinox/.

[Ecl17] Eclipse Foundation, Hrsg., *Help - Eclipse Platform: Eclipse Platform API Specification*, 2017. Adresse: https://help.eclipse.org/oxygen/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fapi%2Foverview-summary.html.

[Ecl18a] Eclipse Foundation, Hrsg., *CBI Aggregator Manual: Eclipsepedia*, 17.05.2018. Adresse: https://wiki.eclipse.org/CBI/aggregator/manual.

[Ecl18b] Eclipse Foundation, Hrsg., *EMF Compare: Compare and Merge Your EMF Models*, 2018. Adresse: https://www.eclipse.org/emf/compare/.

[Ecl18c] Eclipse Foundation, Hrsg., *Viatra - Scalable reactive model transformations: THE VIATRA QUERY LANGUAGE*, 2018. Adresse: https://www.eclipse.org/viatra/documentation/query-language.html.

[ES18a] S. Efftinge und M. Spoenemann, *Xtext - Eclipse Support*, 30.07.2018. Adresse: https://www.eclipse.org/Xtext/documentation/310_eclipse_support.html.

[ES18b] ——, *Xtext - The Grammar Language*, 30.07.2018. Adresse: http://www.eclipse.org/Xtext/documentation/301_grammarlanguage.html#keywords.

[ES18c] ——, *Xtend - Modernized Java*, 8.05.2018. Adresse: http://www.eclipse.org/xtend/.

[ES18d] ——, *Xtext - Integration with Java*, 8.05.2018. Adresse: https://www.eclipse.org/Xtext/documentation/305_xbase.html#xbase-language-ref-introduction.

[ES18e] ——, *Xtext - Language Engineering Made Easy!*, 8.05.2018. Adresse: https://www.eclipse.org/Xtext/.

[ES18f] ——, *Xtext - Continuous Integration (with Maven)*, 2018. Adresse: https://www.eclipse.org/Xtext/documentation/350_continuous_integration.html.

[Fau15] D. Fauth, *POM-less Tycho builds for structured environments | vogella blog*, 2015. Adresse: http://blog.vogella.com/2015/12/15/pom-less-tycho-builds-for-structured-environments/.

[FM06] T. Feng und J. I. Maletic, "Applying dynamic change impact analysis in component-based architecture design", in *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2006. SNPD 2006. Seventh ACIS International Conference on*, 2006, S. 43–48.

[Fow10] M. Fowler, *Domain-specific languages*. Pearson Education, 2010.

[GJS+18] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley und D. Smith, "The Java® Language Specification", 2018. Adresse: https://docs.oracle.com/javase/specs/jls/se10/jls10.pdf.

[Gro18] R. Gronback, *Eclipse Modeling Project | The Eclipse Foundation*, Eclipse Foundation, Hrsg., 2018. Adresse: http://www.eclipse.org/modeling/emf/.

[Gui18] C. Guindon, *Eclipse Project: About the Eclipse Project*, The Eclipse Foundation, Hrsg., 2018. Adresse: https://www.eclipse.org/eclipse/.

[HBK18] R. Heinrich, K. Busch und S. Koch, "A Methodology for Domain-spanning Change Impact Analysis", in *Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, IEEE, 2018.

[ISO12] ISO International Organization for Standardization, *Information technology – Universal Coded Character Set (UCS)*, 2012. Adresse: https://www.iso.org/standard/56921.html.

[ISO96] ——, *Information technology – Syntactic metalanguage – Extended BNF*, 1996. Adresse: https://www.iso.org/standard/26153.html.

[ISO98] ——, *Information technology – 8-bit single-byte coded graphic character sets – Part 1: Latin alphabet No. 1*, 1998. Adresse: https://www.iso.org/standard/28245.html.

[Jim12] T. Jim, *Is Java context free?*, 2012. Adresse: http://trevorjim.com/is-java-context-free/.

[KAM18] KAMP-Research, Hrsg., *KAMP*, 2018. Adresse: https://github.com/KAMP-Research/KAMP.

[Kra17] M. Kramer, "Specification Languages for Preserving Consistency between Models of Different Languages", Dissertation, Karlsruher Instituts für Technologie, Karlsruhe, 10.02.2017. Adresse: http://publikationen.bibliothek.kit.edu/1000069284/4139803.

[LDPW17] T. Ludwig, J. Dax, V. Pipek und V. Wulf, "A Practice-oriented Paradigm for End-User Development", Diss., 2017.

[Leh11a] S. Lehnert, "A review of software change impact analysis A Review of Software Change Impact Analysis", 2011.

[Leh11b] ——, *A taxonomy for software change impact analysis*, 2011. DOI: 10.1145/2024445.2024454. Adresse: http://dl.acm.org/ft_gateway.cfm?id=2024454&type=pdf.

[Leh12] Lehnert, Steffen & Farooq, Qurat-ul-ann & Riebisch, Matthias, "A Taxonomy of Change Types and Its Application in Software Evolution", *Proceedings - 2012 IEEE 19th International Conference and Workshops on Engineering of Computer-Based Systems (ECBS)*, 2012.

[LFR13] S. Lehnert, Q. Farooq und M. Riebisch, "Rule-Based Impact Analysis for Heterogeneous Software Artifacts", in *2013 17th European Conference on Software Maintenance and Reengineering (CSMR)*, A. Cleve, Hrsg., Piscataway, NJ: IEEE, 2013, S. 209–218, ISBN: 978-0-7695-4948-4. DOI: 10.1109/CSMR.2013.30.

[LS98] M. Lindvall und K. Sandahl, "Traceability aspects of impact analysis in object-oriented systems", *Journal of Software Maintenance: Research and Practice*, Jg. 10, Nr. 1, S. 37–57, 1998. DOI: 10.1002/(SICI)1096-908X(199801/02)10:1{\textless}37::AID-SMR163{>}3.0.CO;2-R.

[MCLM90] A. MacLean, K. Carter, L. Lövstrand und L. Moran, "User-Tailorable Systems: Pressing the Issues with Buttons", in *Proceedings of the Conference on Human Factors in Computer Systems (CHI)*, ACM, Hrsg., 1990.

[MHS05] M. Mernik, J. Heering und A. M. Sloane, "When and how to develop domain-specific languages", *ACM Computing Surveys*, Jg. 37, Nr. 4, S. 316–344, 2005, ISSN: 03600300. DOI: 10.1145/1118890.1118892.

[Mic18] Microsoft Corporation, Hrsg., *Specification*, 2018. Adresse: https://microsoft.github.io/language-server-protocol/specification.

[MR14] K. Müller und B. Rumpe, *A Model-Based Approach to Impact Analysis Using Model Differencing*, 2014. Adresse: http://arxiv.org/pdf/1406.6834.

[MS18] E. Merks und E. Stepper, *Eclipse Oomph*, 22.05.2018. Adresse: https://projects.eclipse.org/projects/tools.oomph.

[Obj14] Object Management Group, *Object Constraint Language: Version 2.4*, 2014. Adresse: https://www.omg.org/spec/OCL/.

[Obj15] ——, *XML Metadata Interchange (XMI) Specification: Version 2.5.1*, 2015. Adresse: https://www.omg.org/spec/XMI/2.5.1.

[Obj16] ——, *OMG Meta Object Facility (MOF) Core Specification: Version 2.5.1*, 2016. Adresse: https://www.omg.org/spec/MOF/2.5.1/.

[Obj18] ——, *About OMG | Object Management Group*, 2018. Adresse: http://www.omg.org/about/index.htm.

[RNL+] P. Rapicault, A. Niefer, D. Leberre, D. Houghton, H. Lindberg, I. Bull, J. McAffer, J. Arthorne, S. Kaegi, S. Liebig, S. McCourt und T. Hallgren, *Equinox p2 | The Eclipse Foundation*, Eclipse Foundation, Hrsg. Adresse: http://www.eclipse.org/equinox/p2/.

[RSHR15] K. Rostami, J. Stammel, R. Heinrich und R. Reussner, *Architecture-based Assessment and Planning of Change Requests*, 2015. DOI: 10.1145/2737182.2737198. Adresse: http://dl.acm.org/ft_gateway.cfm?id=2737198&type=pdf.

[SBT14] I. Siket, Á. Beszédes und J. Taylor, "Differences in the Definition and Calculation of the LOC Metric in Free Tools", 2014. Adresse: http://www.inf.u-szeged.hu/~beszedes/research/SED-TR2014-001-LOC.pdf.

[Sie18] J. Sievers, *Tycho home | The Eclipse Foundation*, Eclipse Foundation, Hrsg., 2018. Adresse: https://www.eclipse.org/tycho/.

[Sip13] M. Sipser, *Introduction to the theory of computation*, 3rd ed. 2013, ISBN: 113318779X.

[SR18] J. Stammel und R. Reussner, "Kamp: Karlsruhe architectural maintainability prediction", 2018.

[Sta17] J. J. Stammel, "Architekturbasierte Bewertung und Planung von Änderungsanfragen", *(Keine Angabe)*, Jg. 19, 2017, ISSN: 1867-0067. DOI: 10.5445/KSP/1000054452.

[Sta73] H. Stachowiak, *Allgemeine Modelltheorie*. Wien, New York: Springer Verlag, 1973.

[SVE+07] T. Stahl, M. Völter, S. Efftinge, A. Haase und J. Bettin, *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*, 2., aktual. u. erw. Aufl. 2007. Heidelberg: dpunkt.verlag, 2007, ISBN: 9783898644488. Adresse: http://gbv.eblib.com/patron/FullRecord.aspx?p=1049742.

[Tur37] A. M. Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem", *Proceedings of the London Mathematical Society*, Nr. 1, S. 230–265, 1937. DOI: 10.1112/plms/s2-42.1.230. Adresse: https://londmathsoc.onlinelibrary.wiley.com/doi/full/10.1112/plms/s2-42.1.230.

[VHC+17] B. Vogel-Heuser, R. Heinrich, S. Cha, K. Rostami, F. Ocker, S. Koch, R. Reussner und S. Ziegltrum, "Maintenance effort estimation with KAMP4aPS for cross-disciplinary automated PLC-based Production Systems - a collaborative approach", *IFAC-PapersOnLine*, Jg. 50, Nr. 1, S. 4360–4367, 2017, ISSN: 24058963. DOI: 10.1016/j.ifacol.2017.08.877.

[vit18] vitruv-tools, Hrsg., *Vitruv: View-based development and model consistency framework*, 2018. Adresse: https://github.com/vitruv-tools/Vitruv.

[Vog18] L. Vogel, *OSGi Services - Tutorial*, 2018. Adresse: http://www.vogella.com/tutorials/OSGiServices/article.html.

[Vor10] U. Vora, "Change impact analysis and software evolution specification for continually evolving systems", in *Software Engineering Advances (ICSEA), 2010 Fifth International Conference on*, 2010, S. 238–243.

[Wer16] D. Werle, "A Declarative Language for Bidirectional Model Consistency", Diss., Karlsruhe Institute of Technology (KIT), 2016.